



# Efficient Distance Sensitivity Oracles for Real-World Graph Data

Jong-Ryul Lee  and Chin-Wan Chung 

**Abstract**—A distance sensitivity oracle is a data structure answering queries that ask the shortest distance from a node to another in a network expecting node/edge failures. It has been mainly studied in theory literature, but all the existing oracles for a directed graph suffer from prohibitive preprocessing time and space. Motivated by this, we develop two practical distance sensitivity oracles for directed graphs as variants of Transit Node Routing. The first oracle consists of a novel fault-tolerant index structure, which is used to construct a solution path and to detect and localize the impact of network failures, and an efficient query algorithm for it. The second oracle is made by applying the A\* heuristics to the first oracle, which exploits lower bound distances to effectively reduce search space. In addition, we propose additional speed-up techniques to make our oracles faster with a slight loss of accuracy. We conduct extensive experiments with real-life datasets, which demonstrate that our oracles greatly outperform all of competitors in most cases. To the best of our knowledge, our oracles are the first distance sensitivity oracles that handle real-world graph data with million-level nodes.

**Index Terms**—Graph algorithms, path and circuit problems, graphs and networks, distance sensitivity oracle, distance query, shortest distance, shortest path algorithms

## 1 INTRODUCTION

COMPUTING the shortest distance from a node to another is one of the fundamental issues in graphs. In this paper, we deal with an interesting variation of the shortest distance computation in a graph, called the distance sensitivity problem. Given a graph  $\mathcal{G} = (V, E)$  where  $V$  is the set of nodes and  $E$  is the set of edges, the distance sensitivity problem is to answer queries that ask to compute the distance of the shortest path from a node to another avoiding the failed part of the network. A distance sensitivity oracle is a data structure which is designed to answer such queries. The distance sensitivity oracle is important for a network where a small number of recoverable failures (node or edge) can simultaneously occur. Example scenarios for such failures are as follows:

- *Example 1: In road networks, a user may directly want to know the distance of the shortest path from a point to another avoiding user-specific roads such as frequently clogged roads. This kind of queries can be very frequently happen, because a user may want to ask multiple times for the same start and destination with different avoided roads.*
- *Example 2: In road networks, a road construction, a demonstration, or a car accident can make a road (edge) or a road junction (node) unavailable temporarily. In addition, some roads can be*

*classified to be failed depending on the level of congestion. Generally, such node or edge failures will be recovered after the construction is finished, the demonstration is over, clearing the scene of the accident, or the traffic congestion disappears.*

- *Example 3: In computer networks, network devices (nodes) and links (edges) between them can be broken down by various reasons such as a cut network cable. Such broken devices or links can be recovered by replacing a broken equipment with a normal equipment.*
- *Example 4: In online social networks such as Facebook, a user (node) blocks other users who share documents that the user dislikes. Blocking other users can be considered as recoverable edge failures.*

*Motivation.* The main motivation of a distance sensitivity oracle is that it enables us to avoid stalling queries. Consider that we have an algorithm answering a shortest distance query with an updatable data structure which is called a fully dynamic distance oracle in the theory community. Because node/edge failures change the network structure, when they occur, we have to update the distance oracle to correctly answer the query. After the failures are recovered, we need to update it again for the same reason. Note that while a distance oracle is being updated, no query can be processed. That is, we should stall queries that are issued during the update. In addition, the cost for the update can be very large, because a single edge deletion affects the shortest distances of  $O(n^2)$  node pairs, in which  $n$  is the number of nodes, and even though many failures may not be related to the actual query answer, distance oracles should be updated for all of them. Since stalling queries during such an expensive update operation can seriously be harmful to latency, a distance sensitivity oracle is essential for providing a stable service.

*Existing Sensitivity Oracles.* Most existing works for a distance sensitivity oracle deal with only a constant number of failures in a directed graph or a variable number of failures

• J.-R. Lee is with the Korea Advanced Institute of Science and Technology, Daejeon 34141, Republic of Korea. E-mail: hellcodes.kaist@gmail.com.

• C.-W. Chung is with the Chongqing Liangjiang KAIST International Program, Chongqing University of Technology (CQUT), Chongqing 400054, China, and also with the School of Computing, Korea Advanced Institute of Science and Technology (KAIST), Daejeon 34141, Republic of Korea. E-mail: chungcw@kaist.edu.

Manuscript received 29 Apr. 2018; revised 9 May 2019; accepted 6 June 2019.

Date of publication 27 June 2019; date of current version 7 Dec. 2020.

(Corresponding author: Chin-Wan Chung.)

Recommended for acceptance by X. Xiao.

Digital Object Identifier no. 10.1109/TKDE.2019.2924419

in an undirected graph [1], [2], [3], [4], [5], [6], [7]. Weimann and Yuster propose the only oracle which can answer queries containing a variable number of failures in a directed graph [8], [9]. Their oracle has  $\tilde{O}(n^{4-\alpha})^1$  preprocessing time,  $\tilde{O}(n^{3-\alpha})$  space, and  $\tilde{O}(n^{2-2(1-\alpha)/f})$  query time, where  $f$  is a parameter for the maximum number of failures,  $f = o(\log n / \log \log n)$ , and  $\alpha$  is a parameter in  $(0, 1)$  for specifying the trade-off among preprocessing time, space, and query time. However, since the cost for space and the cost for preprocessing time are too expensive, their oracle is not applicable to real-life datasets. In fact, all existing distance sensitivity oracles for a directed graph with real-valued edge weights suffer from such prohibitive costs for space and for preprocessing time.

*Our Approach.* In this work, we focus on achieving oracles that can efficiently answer distance queries without any stalling even though some network failures are happening. This feature of them is so critical, because it is necessary for stable latency and enables them to handle multiple queries in parallel, each of which is processed with a separate thread on the same index structure. Since we can easily make such threads independent of each other, they can contribute to linearly increase throughput of query processing. The key features of our approach to devise two efficient distance sensitivity oracles are as follows:

- *Fault-tolerant Index Structure:* Our oracles are based on a novel fault-tolerant index structure, which consists of two levels. The first-level is a small overlay graph called the distance graph. The second-level consists of trees of small sizes, called bounded shortest path trees, and the inverted tree index which is a map from each edge in a bounded shortest path tree to the trees containing the edge. Incorporated with the query algorithms, the second-level is used to efficiently localize the impact of network failures in the distance graph.
- *Threadable Query Algorithms:* Our oracles have efficient and threadable query algorithms. They are designed to immediately process a query without any update on the index structure against network failures. Thus, there is no stalling required by these algorithms and they can handle multiple queries in parallel to increase throughput.
- *More Sparse Distance Graph:* In order to construct a good distance graph which is at the first-level of the index structure, we borrow a decent concept called a  $k$ -path cover from [10]. Beyond the work in [10], we propose a novel way of selecting a better  $k$ -path cover so that a resulting distance graph is more sparse.
- *Exploiting the A\* Heuristics:* The query algorithm of the first oracle among our oracles is a variation of the Dijkstra's algorithm. We devise the second oracle by combining this query algorithm with the A\* heuristics. This combination is achieved by a novel method of computing a solution path on the distance graph and recomputing some part of the distance graph in a joint way. This method effectively reduces the search space on the input graph to recompute such a part.
- *Efficient Speed-up Techniques:* We propose two speed-up techniques to make our approach faster for specific classes of networks. The first speed-up technique is called partial detouring, which efficiently provides a detour of a sub-path of the original shortest path. The other

1.  $h(n) = \tilde{O}(g(n))$  if  $h(n) = O(g(n)\log^k(n))$  for some constant  $k > 0$ .

technique is distance graph sparsification, which effectively removes out unnecessary edges from the distance graph. The partial detouring is especially effective for bounded-degree networks, while the distance graph sparsification is effective for scale-free networks.

*Contributions.* This work is for bridging the gap between theory and practice for the distance sensitivity problem. The contributions of it are as follows:

- We propose two efficient distance sensitivity oracles for a variable number of edge failures in directed graphs, which can answer distance queries without any stalling.
- We present efficient maintenance strategies for our oracles in order to handle graph updates. Due to the lack of space, however, all the contents for them are only included in the supplemental material, which can be found on the Computer Society Digital Library at <http://doi.ieeecomputersociety.org/10.1109/TKDE.2019.2924419>.
- We conduct extensive experiments to evaluate our distance sensitivity oracle. In the experiments, we use three competitors: the Dijkstra's algorithm, the classic landmark-based A\* search algorithm, and a recent fully dynamic approximate distance oracle in [11]. Note that none of the existing distance sensitivity oracles for directed graphs is applicable to real-life datasets. We demonstrate that our oracles outperform the competitors in terms of query time in most cases. Along with the efficient query time, the preprocessing time and the space of our first oracle are better than those of the dynamic distance oracle and the A\* search algorithm. It is remarkable that our first oracle mostly has better query performance than the A\* search algorithm, even if it does not incorporate the A\* search heuristics, which were proved to be powerful for finding shortest paths. Our second oracle has greatly improved performance for bounded-degree networks in terms of query time with reasonable preprocessing time and space. The two speed-up techniques effectively make our oracles faster with a slight loss of accuracy. Finally, we demonstrate that the maintenance strategies for our oracles reasonably efficiently update them without losing query efficiency in the supplemental material, available online. To the best of our knowledge, this paper is the first work for the distance sensitivity problem that handles real-world graph data with million-level nodes.

*Outline.* The rest of this paper is organized as follows. In Section 2, we review existing and applicable works for the distance sensitivity problem. The distance sensitivity problem is formulated in Section 3. We propose a distance sensitivity oracle in Section 4 and an improved version of it based on the A\* heuristics in Section 5. For further speed-up, we propose two techniques which make our oracles faster with a slight loss of accuracy in Section 6. Then, we evaluate the efficiency and the accuracy of our oracles with various real-life datasets in Section 7 and conclude the paper in Section 8.

## 2 RELATED WORK

*Distance Sensitivity Oracle.* The distance sensitivity problem has been mainly studied in theory literature. Demetrescu et al. [1] proposed an oracle for handling a single network failure. It has  $O(1)$  query time,  $\tilde{O}(n^2)$  space, and  $\tilde{O}(mn^2 + n^3)$  preprocessing time. Bernstein and Karger improved it in terms of the

preprocessing time to  $\tilde{O}(\sqrt{mn}^2)$  using random sampling in [2] and improved the preprocessing time again to  $\tilde{O}(mn)$  based on a deterministic construction algorithm in [3] with the same query time and the same space. For dual failures, Duan and Pettie [5] proposed an oracle that requires  $\tilde{O}(1)$  query time,  $\tilde{O}(n^2)$  space, and  $\omega(mn)$  preprocessing time. For a variable number of failures, Weimann and Yuster proposed the oracle constructed by a randomized algorithm in [8], [9]. All the existing distance sensitivity oracles for a directed graph with real-valued edge weights suffer from  $\omega(n^2)$  space and  $\omega(mn)$  preprocessing time.

There is only one study for the distance sensitivity problem with some experiments. Qin et al. [7] proposed a distance sensitivity oracle that works only for undirected unweighted graphs with any single edge failure. This work can be applied only for a limited class of graphs and the datasets used in its experiments are rather small.

*Dynamic Distance Oracle.* One of non-trivial ways for handling the distance sensitivity problem is to use an existing fully dynamic distance oracle. Most existing works for a fully dynamic distance oracle in theory literature are not practical to be applied for recent directed graphs with million-level nodes. Instead, there are several recent studies in database literature that deal with large dynamic graphs for processing distance queries. Tretyakov et al. [11] proposed a fully dynamic approximate distance oracle, named LCA, based on landmarks without any theoretical guarantee for accuracy. The query time of this oracle is  $O(lD)$  where  $l$  is the number of landmarks and  $D$  is the diameter of a graph, while the update time for an edge deletion is  $O(l(m+n \log n))$ . We will compare our distance sensitivity oracle with this fully dynamic distance oracle in the experiments. Cheng et al. [12] proposed a fully dynamic distance oracle based on a vertex cover, but it only handles single source distance queries. Fu et al. [13] proposed a distance oracle having an index structure, called a vertex hierarchy structure, which is constructed with an independent set by a similar way as we use it for finding a  $k$ -path cover. However, in [13], the relationship between an independent set and a  $k$ -path cover is not explored. In addition, Fu et al. only focused on vertex updates. Akiba et al. [14] proposed the first practical exact distance oracle targeting on dynamic graphs, but it considers only edge/node additions, not edge/node deletions. There is a recent work of Hayashi et al. [15] for a fully dynamic distance oracle. However, since it is limited to undirected unweighted graphs, we do not compare our oracles with it.

*Dynamic Shortest Routing.* There are several existing studies [16], [17], [18], [19], [20], which are designed to compute a point-to-point shortest path on dynamic graphs with edge weight changes. By setting the edge weights of failed edges to  $\infty$ , their algorithms can solve the distance sensitivity problem.

One of famous online shortest path algorithms is the A\* search algorithm, which is a best-first search algorithm incorporating a heuristic distance estimation proposed in [21]. Delling and Wagner [16] reported that the A\* search algorithm can efficiently compute a point-to-point shortest path on a changed graph, in which some edge weights increase, without updating its preprocessed index. This observation inspires us to exploit the A\* heuristics for the distance sensitivity problem and the classical A\* search algorithm will be used as a competitor in the experiments.

Schultes and Sanders [18] proposed a dynamic shortest path algorithm based on a multi-level overlay graph. Since an edge weight of a higher level graph is the shortest

distance on its lower level graph, once the weight of an edge in  $\mathcal{G}$  is changed, many highway edges related to it may be affected through multiple levels. Instead of explicitly updating them, Schultes and Sanders proposed a modified Dijkstra's algorithm to simply avoid relaxing such affected highway edges. Bruera et al. [19] analyze this algorithm to provide a theoretical foundation about the effect of edge weight changes in the highway hierarchy, and show that it is better than the Dijkstra's algorithm. As analyzed, this approach can be efficient if the number of edges to be changed is extremely small or such edges have nothing to do with the query answer. Otherwise, however, since many highway edges may become unavailable, the algorithm would mostly use edges in  $\mathcal{G}$ , which means that it would act like the Dijkstra's algorithm. In the experiments of [18], only one thousand edges, 0.002 percent of the entire edge set, are changed when comparing with the A\* search algorithm. Even in that case, their method is actually comparable with the A\* search algorithm.

Nannicini et al. [20] proposed a heuristic algorithm based on a highway hierarchy. Given edge weight changes, the heuristic is to compute a shortest path by the multi-level bidirectional Dijkstra's algorithm after only updating corresponding edge weights of the highway hierarchy. Note that the structure of the highway hierarchy is not updated at all. Thus, the query time must be short, but there might be accuracy loss without any theoretical guarantee.

U et al. [17] proposed a stochastic way of optimizing a hierarchical index structure and an algorithm to compute a shortest path on dynamic graphs with a little recomputation on it. The recomputation includes the dynamic shortest path tree algorithm in [22] for the bounded shortest path tree as our first oracle does. However, even for changed edge weights that are not related at all to a resulting shortest path, it updates its index structure to handle all of them, while our oracles handle only a part of them if required.

There are other studies about dynamic time-dependent graphs, in which edge weights can change over time. Even if they deal with a kind of dynamic graphs, such dependency on time is out of scope of this paper. This paper focuses on accidental edge failures that are hard to predict in advance.

*Customizable Route Planing.* Customizable route planing is to compute the point-to-point shortest path defined by one of various metrics such as travel time and geographical distance. There are several existing works for customizable route planing [10], [23]. Theoretically, we can handle the distance sensitivity problem via customizable route planing by defining a cost function that returns infinity as the weights of certain edges (i.e., failed edges). However, such a cost function can handle only one set of failed edges. It means that whenever a query is given, a new cost function for the query should be defined, which would be fairly expensive. Therefore, customizable route planing is inappropriate for solving the distance sensitivity problem.

*Overlay Graphs.* For computing point-to-point shortest distances, the concept of the overlay graph has been studied in various ways [10], [24], [25], [26], [27]. The most related works with respect to a way of utilizing an overlay graph are [10] and [27]. In order to build a good distance graph, we borrow the concept of the  $k$ -path cover proposed in [10]. A  $k$ -path cover is a set of nodes such that all paths of  $k$  consecutive nodes in the entire graph include at least one of the nodes in this set. Funke et al. [10] showed the idea of constructing an

overlay graph that consists of nodes in a  $k$ -path cover. However, since they do not consider at all the density of the overlay graph, it is likely to be extremely dense. Akiba et al. [27] proposed a hierarchical  $k$ -path cover under a dynamic graph. However, this work does not consider either the density of a distance graph when it computes a  $k$ -path cover. Since our method selects a  $k$ -path cover with considering the density of a resulting distance graph, it is likely to be more sparse than distance graphs computed by their algorithms. We will compare their algorithms with ours in detail.

### 3 PRELIMINARIES

#### 3.1 Problem Definition

For convenience, we explain our distance sensitivity oracles for edge failures. It should be noticed that this work is easily extended to handle node failures. In addition, for simplicity, we regard that the shortest path is unique. All techniques in this paper are applicable for the case that there are multiple shortest paths between nodes.

To represent a general network, we consider a directed graph  $\mathcal{G} = (V, E)$ , where  $V$  is the set of nodes and  $E$  is the set of edges, as the input graph. This is because there are many cases that an edge failure can be unidirectional. For example, road networks can be represented as an undirected graph, but edge failures (road construction, demonstration, or car accident, etc.) can be unidirectional. Given a graph  $\mathcal{X}$ , for any node  $v$  in  $\mathcal{X}$ , the set of inbound neighbors of  $v$  is denoted as  $n_{\mathcal{X}}^{in}(v)$  and the set of outbound neighbors of  $v$  is denoted as  $n_{\mathcal{X}}^{out}(v)$ . For any edge  $(x, y)$  in a graph  $\mathcal{X}$ ,  $(x, y)$  is associated with a weight, denoted as  $w_{\mathcal{X}}(x, y)$ , which is a non-negative real value. For notational simplicity, if  $\mathcal{X}$  is the input graph  $\mathcal{G}$ , it is abbreviated as  $w(x, y)$ . In addition, we denote the empty set as  $\emptyset$ .

We define that a path  $P$  is a sequence of edges. Even though  $P$  is a sequence, when we focus on its elements, not the order, we handle it as a set of edges with set operations such as intersection. Given any path  $P$  in a graph  $\mathcal{X}$ , the distance of  $P$  is defined to be  $d(P) = \sum_{(u,v) \in P} w_{\mathcal{X}}(u, v)$ . In addition, the length of  $P$ , denoted as  $|P|$ , is defined to be the number of hops of  $P$ . For any two nodes  $u, v \in V$  and a set of edges  $E^* \subseteq E$ , the shortest (distance) path from  $u$  to  $v$  in the graph  $(V, E \setminus E^*)$  is denoted as  $P(u, v, E^*)$  and its distance is denoted as  $d(u, v, E^*)$ .

**Definition 3.1 (Distance Sensitivity Problem).** *Given a directed graph  $\mathcal{G} = (V, E)$ , the distance sensitivity problem is a query  $(s, t, F)$ , where  $s$  is the start node,  $t$  is the destination node, and  $F$  is a set of at most  $f$  failed edges, asking to compute  $d(s, t, F)$ .*

The failed edge set  $F$  can be filled by a user or a system depending on applications. It should be noticed that  $F$  is formulated to be different for each query. The rationale of this formulation is to achieve generality. For example, this formulation is necessary for the case that a user asks the shortest distance from a location to another with avoiding user-specific roads.

As a trivial solution for this problem, the Dijkstra's algorithm still works well, and its running time is  $O(m + n \log n)$  with the Fibonacci heap. Another trivial solution is storing the answer of every possible query. It takes  $O(n^{2+2f})$  space, since  $O(n^2)$  comes from all pairs of nodes in  $V$  and  $O(n^{2f})$  comes from all possible combinations of at most  $f$  edges among  $O(n^2)$  edges. Thus, a non-trivial distance

TABLE 1  
Frequently Used Notations

$\mathcal{G}$	the input graph
$\mathcal{D}$	a distance graph
$\mathcal{G}_u$	the bounded shortest path tree of a node $u$
$n_{\mathcal{X}}^{out}(v)$	the outbound neighbors of node $v$ in a graph $\mathcal{X}$
$n_{\mathcal{X}}^{in}(v)$	the inbound neighbors of node $v$ in a graph $\mathcal{X}$
$P(s, t, F)$	the shortest path from $s$ to $t$ in the graph $(V, E \setminus F)$
$d(s, t, F)$	the distance of $P(s, t, F)$
$d(s, t)$	the distance of $P(s, t, \emptyset)$ ( $= d(s, t, \emptyset)$ )
$d(P)$	the distance of path $P$

sensitivity oracle should be faster than the Dijkstra's algorithm with a data structure of size lower than  $O(n^{2+2f})$ .

## 4 A DISTANCE SENSITIVITY ORACLE

This section describes our first oracle, called DISTance graph-based Oracle (DISO), for the distance sensitivity problem. In the following subsections, we first briefly introduce Transit Node Routing (TNR) which our oracle is based on. Then, we explain how to adapt TNR for this problem with the fault-tolerant index structure. Finally, we show an effective strategy to choose the node set of a distance graph based on the concept of the  $k$ -path cover. Note that for all theorems and lemmas in this section and the next section, their proofs are given in the supplemental material, available online. Table 1 summarizes frequently used notations.

### 4.1 Transit Node Routing-based Query Processing

#### 4.1.1 Transit Node Routing

TNR [28] is one of famous speed-up frameworks for fast shortest path routing in road networks. The generic version of this technique consists of the following items [29]:

- *Transit Node Set:* A set of nodes  $T \subseteq V$  that are supposed to participate in many shortest paths.
- *Distance Table:* A table (or function)  $d_T : T \times T \rightarrow \mathbb{R}_0^+$ , in which  $\mathbb{R}_0^+$  is the set of non-negative real numbers, returning the shortest distance between transit nodes.
- *Out-access (in-access) Node Mapping:* A mapping  $A_{out}$  ( $A_{in}$ ) from a node  $v$  in  $V$  to the set of all transit nodes, called out-access (in-access) nodes, each of which can be the first (last) transit node on a shortest path from (to)  $v$ .
- *Locality Filter:* A boolean function that returns true for any two nodes in  $V$  if there is no transit node on the shortest path between them, and otherwise returns false.

The first three items are computed in preprocessing and the last item usually gets ready in querying. In TNR, for any two nodes  $s$  and  $t$ , if the locality filter returns false for  $s$  and  $t$ ,  $d(s, t)$  is computed as [29]

$$d(s, t) = \min_{(u,v) \in A(s,t)} \hat{d}(s, u) + d_T(u, v) + \hat{d}(v, t), \quad (1)$$

where  $A(s, t)$  denotes  $A_{out}(s) \times A_{in}(t)$  and  $\hat{d}(x, y)$  denotes the distance of the shortest path from  $x$  to  $y$  which does not pass through any other transit node except  $x$  and  $y$ . Otherwise, an alternative algorithm is used to compute  $d(s, t)$ . Note that even if we replace  $A_{out}(s)$  and  $A_{in}(t)$  with their supersets, (1) holds.

The underlying idea of TNR is to use precomputed distances stored in the distance table as highways to effectively

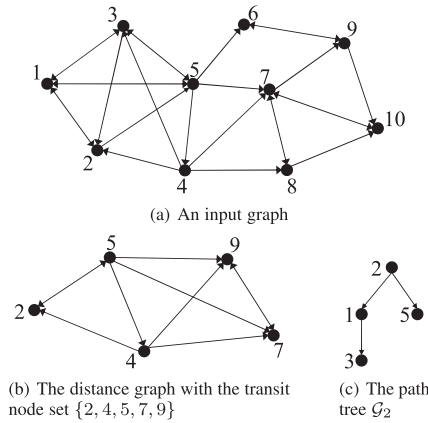


Fig. 1. Examples of an input graph and a two-level index structure.

reduce search space for computing  $d(s, t)$ . In order to enjoy such a merit, *DISO* is designed to answer a query based on (1) by adapting *TNR* to efficiently work on the graph  $(V, E \setminus F)$ .

*Adaptation.* In *DISO*, computing the access node mapping is simply done by a modified version of the Dijkstra's algorithm, called the bounded Dijkstra's algorithm. The algorithm is designed to avoid traversing beyond transit nodes except the source node. It is easy to see that the set of transit nodes visited by the bounded Dijkstra's algorithm from  $s$  is a superset of  $A_{out}(s)$ , which is denoted by  $A_{out}^*(s)$ . In addition, the reported distance from  $s$  to each node  $u$  in the superset is exactly the same as  $\hat{d}(s, u)$ . A superset of  $A_{in}(s)$ , which is denoted by  $A_{in}^*(s)$ , is similarly computed by the bounded Dijkstra's algorithm.

The locality filter is also handled by the bounded Dijkstra's algorithm. If the locality filter returns true,  $t$  must be visited by the bounded Dijkstra's algorithm from  $s$  and the reported distance from  $s$  to  $t$  must be the right query answer. Based on this fact, without computing the locality filter explicitly, if the reported distance is smaller than the distance based on (1), *DISO* returns the reported distance as the query answer.

The distance table cannot be a  $|T| \times |T|$  table for the distance sensitivity problem, because the distances stored in it may change by failures. Instead, we propose a novel fault-tolerant index structure in the next subsection. Meanwhile, since selecting a transit node set significantly depends on that index, we will discuss it later in Section 4.3.

#### 4.1.2 Fault-Tolerant Index Structure

The proposed fault-tolerant index structure consists of two levels. In the first level, we have the distance graph to summarize the distance information of the entire graph. In the second level, for any node  $v$  in the distance graph, we have a small tree to deal with local path information near  $v$  in graph  $\mathcal{G}$  and the inverted tree index over edges participating in the tree. The first level is used to construct a solution path in a global level and the second level is used to deal with the impact of network failures on the distance graph in a local level.

*First-Level Index.* Consider a set of transit nodes  $T \subseteq V$  and a query  $(s, t, F)$ . Instead of the distance table on them, we exploit a special graph, called the distance graph, which is defined as follows:

**Definition 4.1 (Distance Graph).** Given any set of transit nodes  $T \subseteq V$ , a distance graph is defined to be a graph  $\mathcal{D} = (V_{\mathcal{D}}, E_{\mathcal{D}})$  where  $V_{\mathcal{D}} = T$  and  $E_{\mathcal{D}} \subseteq T \times T$ . For any pair  $(u, v) \in T \times T$ ,  $(u, v)$  is included in  $E_{\mathcal{D}}$ , if there exists a path

from  $u$  to  $v$  in  $\mathcal{G}$  which does not pass through any other node in  $T$ . Each edge  $(u, v) \in E_{\mathcal{D}}$  is associated with a weight denoted as  $w_{\mathcal{D}}(u, v)$ . Given a failed edge set  $F \subseteq E$ , for any two nodes  $u, v \in V_{\mathcal{D}}$ , the shortest distance from  $u$  to  $v$  in  $\mathcal{D}$  is denoted as  $d_{\mathcal{D}}^F(u, v)$ .

For the rest of this section, consider that we have a set of transit nodes  $T \subseteq V$  and the distance graph  $\mathcal{D}$  having  $T$  as the node set. The topological structure of the distance graph can be computed by executing a graph traversal algorithm for each node of  $T$  in preprocessing. Fig. 1 shows an example input graph and the distance graph of it derived from a node set.

Let us describe the edge weighting scheme for the distance graph  $\mathcal{D}$ . Given any two nodes  $x \in V$  and  $y \in T$ , for any edge set  $E^* \subseteq E$ , let  $\hat{P}(x, y, E^*)$  denote the shortest path from  $x$  to  $y$  in the graph  $(V, E \setminus E^*)$  among ones that do not pass through any node in  $T \setminus \{x, y\}$  and  $\hat{d}(x, y, E^*)$  its distance. If  $\hat{P}(x, y, E^*)$  does not exist, then  $\hat{d}(x, y, E^*)$  is  $\infty$ . Note that  $\hat{P}(x, y, E^*)$  may not be  $P(x, y, E^*)$ . For each edge  $(u, v) \in E_{\mathcal{D}}$ ,  $w_{\mathcal{D}}(u, v)$  is set to  $\hat{d}(u, v, \emptyset)$  in preprocessing. Given a query  $(s, t, F)$ ,  $w_{\mathcal{D}}(u, v)$  will be recomputed to  $\hat{d}(u, v, F)$ , if  $\hat{P}(u, v, \emptyset)$  contains some failed edge and  $w_{\mathcal{D}}(u, v)$  is necessary to compute the query answer. Otherwise, we can use  $\hat{d}(u, v, \emptyset)$  as the weight of  $(u, v)$  in  $\mathcal{D}$ , because  $\hat{d}(u, v, F) = \hat{d}(u, v, \emptyset)$  or  $w_{\mathcal{D}}(u, v)$  is unnecessary. For example, in Fig. 1, consider that  $w(1, 3) > 1$  and the other edge weights in the input graph are equal to 1. Then,  $\hat{P}(1, 5, \{(1, 5)\})$  is  $\langle (1, 3), (3, 5) \rangle$  and  $P(1, 5, \{(1, 5)\})$  is  $\langle (1, 2), (2, 5) \rangle$ .

This weighting scheme gives the following lemma, which enables us to use shortest distances on the distance graph as those of the distance table.

**Lemma 1.** For any two nodes  $u, v \in T$  and any failed edge set  $F \subseteq E$ , the edge weighting scheme guarantees that  $d_{\mathcal{D}}^F(u, v) = d(u, v, F)$ .

For completing the above scheme, we need to define how to determine edge weights to be recomputed and how to recompute them. They are efficiently done with the second-level index.

*Second-Level Index.* The second-level index includes a tree structure for any node  $u \in T$ , called the bounded shortest path tree, and the inverted tree index over its edges. They are constructed in preprocessing.

**Definition 4.2 (Bounded Shortest Path Tree).** For any node  $u \in T$ , the bounded shortest path tree of  $u$  is a tree where  $u$  is the root and the path from  $u$  to any other node  $v$  in the tree is identical to  $\hat{P}(u, v, \emptyset)$ , and it is denoted as  $\mathcal{G}_u$ .  $\mathcal{G}_u$  contains every node  $v$  in  $V$  such that  $\hat{P}(u, v, \emptyset)$  exists. An example for the bounded shortest path tree is depicted in Fig. 1c.

**Definition 4.3 (Inverted Tree Index).** For any edge  $e \in E$ , the inverted tree index is an in-memory map from  $e$  to the list of all bounded shortest path trees containing  $e$ .

For computing the bounded shortest path tree of a transit node, we use the bounded Dijkstra's algorithm. Suppose that we run the bounded Dijkstra's algorithm on  $\mathcal{G}$  from a node  $u$  in the outbound direction. Then, it gives a path tree as the Dijkstra's algorithm does. It is trivial that this path tree is identical to  $\mathcal{G}_u$ .

Let us explain how to use the second-level index for the edge weight recomputation. Given a query  $(s, t, F)$ , the query algorithm finds transit nodes  $x \in T$  such that  $\mathcal{G}_x$  has

some failed edges, which are called affected nodes. Finding affected nodes is efficiently implemented with the inverted tree index. We can see that the weights of edges from such an affected node on  $\mathcal{D}$  have the possibility of change due to failures. Since the query algorithm includes a Dijkstra-like procedure on  $\mathcal{D}$ , such edge weights are determined to be recomputed right before the edges are actually relaxed. This technique for recomputing such edge weights on demand is called lazy recomputation. For the recomputation, we use an algorithm, named *DynDijkstra* [22], which updates shortest path trees on dynamic graphs. This algorithm is adapted to update a bounded shortest path tree instead of an original shortest path tree, but since the change is minor, we omit the details of the adaptation. It should be noticed that we do not explicitly update  $\mathcal{G}_x$  in the adapted algorithm, but recompute only the distances.

### 4.1.3 The Query Algorithm

Let us complete the query algorithm of *DISO* as a *TNR* variant. Given a query  $(s, t, F)$ , it first finds affected nodes with the inverted tree index. After that, as we mentioned,  $\hat{d}(s, t, F)$  and the access nodes of  $s$  and  $t$  are computed by the bounded Dijkstra's algorithm. Then, the query algorithm computes the distance of the shortest path from  $s$  to  $t$  passing through at least one transit node based on (1) with a Dijkstra-like search procedure, which is denoted by  $d_{\mathcal{D}}(s, t, F)$ . This procedure actually works like the Dijkstra's algorithm on  $\mathcal{D}$ , but it starts with a priority queue initially containing the access nodes in  $A_{out}^*(s)$ . For any node  $u$  that will be traversed in this procedure, the priority value of  $u$  is the minimum observed distance from  $s$  to  $u$ , which is denoted by  $d_o(s, u, F)$ . Note that if  $u$  is an access node in  $A_{out}^*(s)$ ,  $d_o(s, u, F) = \hat{d}(s, u, F)$  before  $d_o(s, u, F)$  is updated. Whenever a node  $v$  in  $A_{in}^*(t)$  from the queue is popped, it updates  $d_o(s, t, F)$ . In addition, if a node  $x$  popped from the queue is an affected node, the weights of out-bounding edges from  $x$  on  $\mathcal{D}$  are recomputed with  $\mathcal{G}_x$ . This recomputation is done before relaxing the edges from  $x$ . After the queue becomes empty or  $t$  is popped when it is a transit node,  $d_o(s, t, F)$  becomes  $d_{\mathcal{D}}(s, t, F)$ . Finally, the algorithm returns the minimum of  $d_{\mathcal{D}}(s, t, F)$  and  $\hat{d}(s, t, F)$ , which is  $d(s, t, F)$ . The pseudo code of this procedure is included in the supplemental material, available online, due to space limitations.

For simplicity, the query algorithm is based on the classic (not bidirectional) Dijkstra's algorithm. If we construct this query algorithm based on a more efficient online shortest path algorithm like the bidirectional Dijkstra's algorithm, the query algorithm will run faster. In Section 5, we show how the query algorithm is converted to an algorithm like the A\*-search algorithm, which is an improved version of the Dijkstra's algorithm to reduce the search space.

*Correctness.* We now prove the correctness of this query algorithm.

**Lemma 2.** *After the query algorithm is finished, for any query  $(s, t, F)$ , if  $P(s, t, F)$  contains some node in  $T$ ,*

$$d_o(s, t, F) = d_{\mathcal{D}}(s, t, F) = d(s, t, F). \quad (2)$$

By definition, for any query  $(s, t, F)$ , if  $P(s, t, F)$  does not contain any node in  $T$  as an intermediate node,  $\hat{d}(s, t, F) = d(s, t, F)$ . This proposition and the previous lemma directly imply the correctness of the query algorithm.

**Theorem 1.** *For any query  $(s, t, F)$ , the query algorithm of *DISO* correctly finds  $d(s, t, F)$ .*

*Cost Analysis.* We denote the average cost for the bounded Dijkstra's algorithm as  $c_B$ .  $c_B$  depends on the structure of  $\mathcal{G}$  and  $T$ . The cost for finding affected nodes is  $O(|F||T|)$  and the cost for computing the access nodes is  $O(c_B)$ . Building the priority queue for the Dijkstra-like procedure on  $\mathcal{D}$  requires  $O(|T|\log|T|)$  time. In the Dijkstra-like procedure, recomputing the edge weights of an affected node requires  $O(c_B)$  time, because updating a bounded shortest path tree is definitely cheaper than constructing it from scratch. Thus, the Dijkstra-like procedure uses  $O(|E_{\mathcal{D}}| + |T|\log|T| + |A_{avg}|c_B)$  time where  $|A_{avg}|$  is the average of  $|A|$ . Finally, the total query time is  $O(|E_{\mathcal{D}}| + |T|\log|T| + |F||T| + |A_{avg}|c_B)$ .

Meanwhile, it is straightforward that the preprocessing time to construct a distance graph and bounded shortest path trees is  $O((|V| + c_B)|T|)$  when  $T$  is given. The space for the inverted tree index from edges to bounded shortest path trees is  $O(|E| + |T||\mathcal{G}_{avg}|)$  where  $|\mathcal{G}_{avg}|$  is the average number of nodes in the bounded shortest path tree of a transit node. Thus, the total space complexity required by this query algorithm is  $O(|E_{\mathcal{D}}| + |T||\mathcal{G}_{avg}| + (|E| + |T||\mathcal{G}_{avg}|)) = O(|E_{\mathcal{D}}| + |T||\mathcal{G}_{avg}| + |E|)$ .

## 4.2 Stall Avoidance

One can worry that since some edge weights in the distance graph of our oracle can be updated in the query algorithm, our oracle cannot avoid such stalling. However, even if such edge weights in the distance graph are recomputed for query processing, our oracle can be implemented without explicitly updating them. In the query algorithm, the only part that changes the index structure of our oracle is to recompute edge weights on the distance graph before relaxing. There is a sweet property of the Dijkstra's algorithm that any relaxed edge will not be re-relaxed. Thus, for any edge weight  $w_{\mathcal{D}}(u, v)$  that is recomputed, it will not be used again after the edge  $u$  to  $v$  in  $\mathcal{D}$  is relaxed. This means that we do not have to store  $w_{\mathcal{D}}(u, v)$  for future reuse, so we do not have to permanently update it. Therefore, our oracle can be implemented without any explicit update on the index structure.

## 4.3 Toward a Good Distance Graph

A good distance graph for *DISO* has a small number of nodes and edges. This subsection presents an effective method for constructing such a distance graph.

### 4.3.1 A $k$ -Path Cover and its Effect

Because a set of nodes identifies a unique distance graph, we focus on choosing a transit node set for a good distance graph. For this purpose, we first define a  $k$ -path cover.

**Definition 4.4 ( $k$ -Path Cover).** *For some integer  $k > 0$ , a  $k$ -path cover  $C$  is a set of nodes such that all simple paths consisting of  $k$  nodes in  $\mathcal{G}$  pass through at least one of the nodes in  $C$ .*

For example, the node set of the distance graph in Fig. 1 is a 3-path cover in the input graph. It is easy to see that all paths consisting of 3 nodes in the input graph pass through at least one of 2, 4, 5, 7, and 9.

Let us examine how a  $k$ -path cover  $C$  affects *DISO*, when  $C$  is used as the transit node set for *DISO*. The most important property of  $k$ -path cover  $C$  is that for any node  $u \in C$ , the lengths of all paths from  $u$  in  $\mathcal{G}$ , which do not contain

any other node in  $C$  as an intermediate node, are bounded by  $k$ . It guarantees that the bounded Dijkstra's algorithm from a node  $v$  searches at most  $k$  hops from  $v$ . When  $k$  is much smaller than the diameter of  $\mathcal{G}$ ,  $c_B$ , which is the average cost for the bounded Dijkstra's algorithm, must be smaller than that for the Dijkstra's algorithm. In addition,  $k$  affects  $|C|$ ,  $|E_D|$ , and  $|\mathcal{G}_{avg}|$ . Since there is some trade-off for performance depending on  $k$ , we find an appropriate value for  $k$  by increasing  $k$  from  $k = 2$  in experiments.

### 4.3.2 Selecting a Better $k$ -Path Cover based on an Independent Set

Computing the minimum  $k$ -path cover is NP-hard [30]. Funke et al. [10] introduced a heuristic method to compute a minimal  $k$ -path cover. However, they do not consider the sparsity of the distance graph (the overlay multigraph in [10]) derived from the  $k$ -path cover. Thus, the distance graph derived from their  $k$ -path cover may be very dense, and then it harms the efficiency of *DISO*.

Motivated by this, we devise an effective method to compute a  $k$ -path cover and its derived distance graph considering density. It is based on the independent set, which is a well-known concept in graph theory. An independent set  $I$  is a set of nodes in  $V$  such that no two nodes in  $I$  are adjacent. Let us explain how a distance graph for  $\mathcal{G}$  is derived from  $I$ . Initially, we have a graph  $\mathcal{G}_I = (V_I, E_I)$  where  $V_I = V$  and  $E_I = E$ . For each node  $v \in I$ , we remove  $v$  from  $\mathcal{G}_I$  and edges incident to  $v$ . Then, we add edges  $(x, y) \in n_{\mathcal{G}}^{in}(v) \times n_{\mathcal{G}}^{out}(v)$  into  $\mathcal{G}_I$  if  $(x, y) \notin E_I$ . After this process is finished, because  $I$  is an independent set, it is trivial that  $V_I = V \setminus I$  and  $V_I$  is a 2-path cover in  $\mathcal{G}$ , and  $\mathcal{G}_I$  is a distance graph.

Given an independent set computation function  $GetIS(\cdot)$ , we construct a distance graph with a more elaborate way as follows. Consider a distance graph  $\mathcal{D}_i = (V_i, E_i)$  for  $0 \leq i \leq \tau - 1$  where  $\tau$  is some parameter larger than 0. Initially,  $\mathcal{D}_0 = \mathcal{G}$ . We iteratively compute an independent set  $IS_i$  of  $\mathcal{D}_i$  by calling  $GetIS(\mathcal{D}_i)$  and construct a new distance graph  $\mathcal{D}_{i+1}$  for  $\mathcal{G}$  with  $V_{i+1} = V_i \setminus IS_i$ . This procedure gives the following lemma.

**Lemma 3.** For all integer parameter  $\tau \geq 1$ ,  $V_\tau$  in the above procedure is a  $2^\tau$ -path cover in  $\mathcal{G}$ .

*Independent Set Selection.* The remaining part for the path cover algorithm is to complete  $GetIS(\cdot)$ . For each step  $0 \leq i \leq \tau - 1$ , if we can find an independent set from  $\mathcal{D}_i$  such that the number of nodes and the number of edges in  $\mathcal{D}_{i+1}$  are minimized, then the output distance graph eventually has a small number of nodes and edges. However, finding such an independent set is a hard problem, because finding the maximum independent set is NP-hard, which only minimizes the number of nodes in  $\mathcal{D}_{i+1}$ . Based on this background, we propose a greedy method to find an independent set  $I$  such that the number of edges in the distance graph derived from  $I$  is effectively reduced.

Our greedy method incrementally constructs an independent set and its derived distance graph as follows. Given an input distance graph  $\mathcal{D}_i = (V_i, E_i)$ , our greedy method starts with initializing  $I$  as the empty set and constructing a graph  $\mathcal{D}_I = (V_I, E_I)$  where  $V_I = V_i$  and  $E_I = E_i$ . Then, it iteratively picks a node  $v$  from  $\mathcal{D}_I$  such that  $v$  is not adjacent to any node in  $I$  on  $\mathcal{D}_i$  and  $v$  minimizes the following:

$$\sigma(v) = |\{(x, y) \in NPair(v) \setminus E_I\}| - \left( n_{\mathcal{D}_i}^{in}(v) + n_{\mathcal{D}_i}^{out}(v) \right),$$

where  $NPair(v)$  denotes  $n_{\mathcal{D}_i}^{in}(v) \times n_{\mathcal{D}_i}^{out}(v)$ . It is easy to see that  $\sigma(v)$  is the net contribution of  $v$  to the number of edges in the distance graph derived from the resulting independent set. If  $\sigma(v) \leq \theta$  where  $\theta$  is a user-specific parameter, the method removes  $v$  from  $\mathcal{D}_I$ .  $\theta$  plays an important role for controlling the number of the nodes and the sparsity of the resulting distance graph. After removing  $v$ , new edges  $(x, y) \in NPair(v) \setminus E_I$  are added into  $\mathcal{D}_I$  such that  $(x, y) \notin E_I$ . This is repeated until every node  $u$  in  $V_I$  is adjacent to some node in  $I$  on  $\mathcal{D}_i$  or  $\sigma(u) > \theta$ . This method is described in Algorithm 1. In Algorithm 1,  $V_I^*$  denotes the set of nodes in  $V_I$  that are not adjacent to any node in  $I$  on  $\mathcal{D}_i$ .

---

#### Algorithm 1. *GetIS* ( $\mathcal{D}_i$ )

---

**Input:**  $\mathcal{D}_i = (V_i, E_i)$ : the distance graph

**Output:**  $I$ : the independent set

```

1 begin
2    $I := \emptyset$ ;
3   Construct  $\mathcal{D}_I = (V_I, E_I)$  where  $V_I = V_i$  and  $E_I = E_i$ ;
4   while  $\exists v \in V_I^*$  s.t.  $\sigma(v) > \theta$  do
5      $v := \arg \min_{v \in V_I^*} \sigma(v)$ ;
6     if  $\sigma(v) > \theta$  then
7       break;
8     Remove  $v$  from  $\mathcal{D}_I$  and add it to  $I$ ;
9     Add new edges  $(x, y) \in NPair(v) \setminus E_I$  into  $\mathcal{D}_I$ ;
10  return  $I$ ;
```

---

*Comparison with [27] and [10].* Our method removes a node which minimizes the net contribution of the node to the sparsity of the distance graph derived from the resulting independent set. Recall that in [10], when a  $k$ -path cover is computed, the density of the distance graph derived from the resulting  $k$ -path cover is not considered at all. The distance graph computed by our algorithm has a relatively smaller number of edges than the distance graph derived from a  $k$ -path cover in [10].

Akiba et al. proposed a method for computing a  $k$ -path cover with the concept of the vertex cover, which is the complement of an independent set. Note that in our algorithm, since  $IS_i$  is an independent set,  $V_i \setminus IS_i$  is a vertex cover. Instead of computing  $IS_i$ , the method of Akiba et al. directly computes a vertex cover which will be the node set of the next distance graph like  $V_i \setminus IS_i$ .

The main difference between our approach and the method of Akiba et al. is that the sparsity of the resulting distance graph can be controlled via  $\theta$  in our approach, but not in their method. In fact, Akiba et al. did not much consider the sparsity of a distance graph like [10]. In addition, the idea of using an independent set itself is more appropriate for computing the net contribution of a node to the sparsity of the next distance graph than that of using a vertex cover. For example, consider a node  $v \in V_I^*$  in Algorithm 1. In order to compute  $\sigma(v)$ , we only need to pairwise check the outbound neighbors and the inbound neighbors of  $v$ , which requires  $O(d^2)$  where  $d$  is the degree of  $v$ . This is because if  $v$  will be included in the resulting independent set of Algorithm 1, the neighbors of  $v$  must be included in the next distance graph. However, there is no such a guarantee with a vertex cover. In addition to this analysis, we will conduct experiments to compare our approach and the other path cover computation methods in [10] and [27].

*Complexity.* Let us analyze the time complexity of our  $k$ -path cover algorithm. In order to implement Line 5 of

Algorithm 1, we use a priority queue, and the cost for constructing and maintaining it is  $O(n \log n)$ . Thus, the cost for  $GetIS(\cdot)$  is  $O(n(\log n + deg_{avg}^2))$  where  $deg_{avg}$  is the average degree in the observed distance graphs. In our  $k$ -path cover algorithm, we can get  $\mathcal{D}_{i+1}$  without any additional cost, because  $\mathcal{D}_I$  in Algorithm 1 becomes  $\mathcal{D}_{i+1}$  after  $I$  is computed. Therefore, the total time complexity is  $O(\tau n(\log n + deg_{avg}^2))$ .

## 5 AN A\* SEARCH-BASED DISTANCE SENSITIVITY ORACLE

This section presents our second oracle, which is an improved version of the first oracle based on the A\* heuristics. This oracle is called an A\* search-based Distance Sensitivity Oracle (ADISO). We first review the A\* heuristic search, and then explain how to apply its concept to the query algorithm.

### 5.1 Review on the A\* Heuristic Search

The A\* search algorithm is designed to compute the shortest distance from a node  $s$  to a node  $t$ . For any node  $u \in V$ , the cost function  $f$  of the A\* search algorithm is defined as,

$$f(u) = d(s, u) + h(u, t),$$

where  $h(u, t)$  is a heuristic estimate for the shortest distance from  $u$  to  $t$ . Starting from  $s$ , the A\* search algorithm greedily finds the path from  $s$  to  $t$  minimizing  $f(t)$ .

The heuristic function  $h$  is designed to return a lower bound distance from  $u$  to  $t$ . For generality, instead of non-geodesic distances like the euclidean distance, we use a lower bound scheme based on geodesic distances, which was proposed by Goldberg et al. [31]. This scheme is based on landmarks, which are specially selected nodes, and the triangle inequality. We explain it in the following subsection.

### 5.2 A Landmark-Based Lower Bound Scheme

For any failed edge set  $F$  and any two nodes  $u, v$ , we design a lower bound distance of  $d(u, v, F)$  as follows. Note that the cost for computing a lower bound distance should be cheap for the A\* search algorithm, but efficiently computing a lower bound of  $d(u, v, F)$  in query processing is not trivial, because  $F$  is given in a query. Thus, instead of  $d(u, v, F)$ , we focus on computing a lower bound of  $d(u, v)$ , because  $d(u, v)$  itself is a lower bound of  $d(u, v, F)$ . As a lower bound of  $d(u, v)$ , we use the following lower bound based on the triangle inequality, which was proposed in [31].

$$h(u, v) = \max_{x \in L} \{l_x(u, v)\} \leq d(u, v),$$

where  $l_x(u, v) = \max\{d(x, u) - d(x, v), d(u, x) - d(v, x)\}$ .

*Applying the A\* Heuristic Search.* In the query algorithm of DISO, the priority value (i.e., cost) of a node is the minimum observed distance from  $s$  to it. In order to make the query algorithm have a form of the A\* heuristic search, we design the new cost function of the query algorithm as,

$$cost(v) = d_o(s, v, F) + h(v, t).$$

This value is interpreted as the estimated distance of the shortest path from  $s$  to  $t$  passing through  $v$ .

**Theorem 2.** For a query  $(s, t, F)$ , the query algorithm with the modified cost function  $cost(\cdot)$  correctly computes  $d(s, t, F)$ .

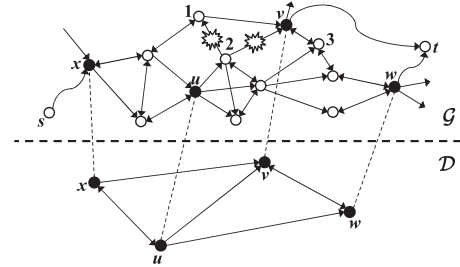


Fig. 2. An example about affected nodes in querying.

*Implementation and Complexity.* In order to compute the heuristic function  $h$  in query time, we need to store the out-bound/inbound shortest distances from each node of a landmark set  $L$  in preprocessing time. This information only requires  $O(N_L n)$  space where  $N_L$  is a user-specific parameter for the number of landmarks. Computing it requires  $O(N_L(m + n \log n))$  time, because we need to run the Dijkstra's algorithm multiple times. In addition, the asymptotical running time of the query algorithm is not changed, because computing  $h$  is a much cheaper operation than updating the priority queue of the query algorithm. Note that the number of landmarks is constant in [31], so  $N_L$  will be constant over various datasets in experiments.

### 5.3 Improved Lazy Recomputation

We improve the lazy recomputation for the edge weights of  $\mathcal{D}$  with the A\* heuristics. Recall that the weights of edges from a node  $u$  in  $\mathcal{D}$  are recomputed in the query algorithm, if  $\mathcal{G}_u$  has any failed edge. This technique efficiently works with a small number of failed edges, but its efficiency decreases as the number of failed edges increases. This is because when the tree contains many failed edges, updating it becomes similar to constructing it from scratch, which can require many unnecessary computations. Let us explain such unnecessary computations using an example in Fig. 2. The upper graph is an input graph  $\mathcal{G}$  and the lower graph is a distance graph  $\mathcal{D}$ . A black node represents a transit node, a straight arrow represents a directed edge, and a curved arrow represents a path. The dotted lines do not represent actual edges, but describe the relationships between transit nodes in  $\mathcal{G}$  and nodes in  $\mathcal{D}$ . In this example, there are failed edges, so that  $F = \{(2, 1), (2, v)\}$ . Suppose that  $\hat{P}(u, \emptyset)$  consists of  $(u, 2)$  and  $(2, v)$ . Then, since  $F$  includes  $(2, v)$ , the weight of  $(u, v)$  in  $\mathcal{D}$  should be recomputed by updating  $\mathcal{G}_u$  when a query  $(s, t, F)$  is being processed. However, the update of  $\mathcal{G}_u$  inevitably includes computing the bounded shortest path from  $u$  to  $x$  even if  $P(s, t, F)$  does not contain it. In this way, the current update method can contain many unnecessary computations having nothing to do with processing a given query.

In order to address this, we propose an improved Dijkstra-like procedure to handle affected nodes with the A\* heuristics. For such affected nodes, the improved procedure is designed to recompute their related edge weights in  $\mathcal{D}$  on the fly. For this, the improved procedure has a merged form of the procedure for computing a solution path on  $\mathcal{D}$  and that for recomputing such edge weights on  $\mathcal{G}$ . Note that it does not use bounded shortest path trees to recompute such edge weights anymore.

The improved Dijkstra-like procedure is described in Algorithm 2. In this algorithm, there are two separate priority queues  $Q_{\mathcal{D}}$  and  $Q_{\mathcal{G}}$  such that one is for nodes in  $\mathcal{D}$  and



the other is for nodes in  $\mathcal{G}$ . Let  $\text{top}(Q)$  denote the priority value of the highest priority node of priority queue  $Q$ . If  $Q$  is empty,  $\text{top}(Q)$  returns  $\infty$ . For any node  $v$ ,  $d_o(s, v, F)$  is abbreviated as  $d_o(v)$ .

This procedure starts with the queue  $Q_{\mathcal{D}}$  initially containing the nodes in  $A_{\text{out}}^*(s)$  as the query algorithm of *DISO* does. The priority value of a node  $v$  is the new cost function  $\text{cost}(v)$ . For each iteration over Lines 4-22 in Algorithm 2, it (Algorithm 2) first compares  $\text{top}(Q_{\mathcal{D}})$  and  $\text{top}(Q_{\mathcal{G}})$ . If  $\text{top}(Q_{\mathcal{D}})$  is smaller than or equal to  $\text{top}(Q_{\mathcal{G}})$ , then  $\mathcal{D}$  is assigned to  $\mathcal{X}_1$ . Otherwise,  $\mathcal{G}$  is assigned to  $\mathcal{X}_1$ . Consider that  $\mathcal{X}_1$  is  $\mathcal{D}$ . Given a popped node  $v$  in this procedure, if the minimum observed distances from the start node to neighbors on the outbound edges of  $v$  are updated, then we say that the edges are relaxed. If  $v$  is a transit node and not an affected node, then it relaxes the outbound edges of  $v$  in  $\mathcal{D}$ . Otherwise, it relaxes the outbound edges of  $v$  in  $\mathcal{G}$  with  $Q_{\mathcal{G}}$  based on the A\* heuristics instead of updating  $\mathcal{G}_v$ . In addition, consider that  $\mathcal{X}_1$  is  $\mathcal{G}$ . Then, it relaxes the outbound edges of  $v$  in  $\mathcal{G}$ . If a neighbor  $n$  of  $v$  is a transit node, then it updates  $Q_{\mathcal{D}}$  for  $n$  with  $\text{cost}(n)$ . In this way,  $d_o(n)$  is maintained on the fly instead of explicitly using the recomputed weight of the edge  $(v, n)$  in  $\mathcal{D}$  by updating  $\mathcal{G}_v$ . This is the most different thing between the improved Dijkstra-like procedure and the original procedure. Meanwhile, if  $n$  is not a transit node, it updates  $Q_{\mathcal{G}}$  for  $n$ .

---

#### Algorithm 2. Improved Dijkstra-Like Procedure

---

```

1 begin
2 Initialize a priority queue  $Q_{\mathcal{D}}$  with the nodes in  $A_{\text{out}}^*(s)$ ;
3 Initialize a priority queue  $Q_{\mathcal{G}}$  as an empty queue;
4 while  $Q_{\mathcal{D}}$  is not empty or  $Q_{\mathcal{G}}$  is not empty do
5   if  $\text{top}(Q_{\mathcal{D}}) \leq \text{top}(Q_{\mathcal{G}})$  then  $\mathcal{X}_1 := \mathcal{D}$ ;
6   else  $\mathcal{X}_1 := \mathcal{G}$ ;
7    $v := \arg \min_{u \in Q_{\mathcal{X}_1}} \text{cost}(u)$ ;
8   Pop  $v$  from  $Q_{\mathcal{X}_1}$ ;
9   if  $v = t$  then
10    break;
11   if  $\mathcal{X}_1 = \mathcal{D}$  and  $v \in A_{\text{in}}^*(t)$  then
12     Push  $t$  into  $Q_{\mathcal{D}}$  if  $\text{cost}(t) = \infty$ ;
13      $d_o(t) := \min\{d_o(t), d_o(v) + \hat{d}(v, t, F)\}$ ;
14   if  $v \in C$  and  $v \notin A$  then  $\mathcal{X}_2 := \mathcal{D}$ ;
15   else  $\mathcal{X}_2 := \mathcal{G}$ ;
16   for  $n \in n_{\mathcal{X}_2}^{\text{out}}(v)$  do
17     if  $v \notin C$  and  $n \in C$  then
18       Push  $n$  into  $Q_{\mathcal{D}}$  if  $\text{cost}(n) = \infty$ ;
19     else
20       Push  $n$  into  $Q_{\mathcal{G}}$  if  $\text{cost}(n) = \infty$ ;
21        $d_o(n) := \min\{d_o(n), d_o(v) + w_{\mathcal{X}_2}(v, n)\}$ ;
22      $\text{cost}(n) := d_o(n) + h(n, t)$ ;

```

---

Let us explain how unnecessary computations are avoided in the improved Dijkstra-like procedure. Consider the previous example described in Fig. 2. Suppose that the query  $(s, t, F)$  is given and  $u$  is popped from  $Q_{\mathcal{D}}$  in the improved Dijkstra-like procedure. Since  $u$  is an affected node, the algorithm relaxes the outbound edges of  $u$  in  $\mathcal{G}$ . Compared with updating  $\mathcal{G}_u$ , it greatly reduces many unnecessary computations like traversing from  $u$  to  $x$  because the A\* heuristics make nodes closer to  $t$  be earlier popped from the queues.

In addition to this, one decent property of the improved Dijkstra-like procedure is that no edge weight in the

distance graph is updated. Thus, the modified query algorithm still guarantees to avoid stalling as the former query algorithm does.

We prove that with such merits, this modified query algorithm still finds the correct answer of a given query as follows.

**Lemma 4.** Consider a query  $(s, t, F)$ , and any two nodes  $u, v \in C$  such that  $\hat{P}(u, v, F)$  is a sub-path of  $P(s, t, F)$  and  $u$  is an affected node. For any node  $w \in V$  on  $\hat{P}(u, v, F)$  which is popped from  $Q_{\mathcal{G}}$  with  $d_o(w)$  in Algorithm 2,

$$d_o(w) = d_o(u) + \hat{d}(u, w, F). \quad (3)$$

**Theorem 3.** The improved Dijkstra-like procedure in Algorithm 2 correctly finds  $d_{\mathcal{D}}(s, t, F)$ .

## 5.4 Landmark Selection Strategy

For the efficiency of the A\* search algorithm, we need to select a set of effective landmarks. Computing an optimal landmark set of the A\* search algorithm for the failure-free point-to-point shortest path problem is known to be NP-hard [32], so heuristic selection methods are usually used [31]. Similarly, we propose a sampling-based heuristic method for landmark selection.

This method first samples uniformly at random a certain number of nodes in  $\mathcal{G}$ , and then computes the outbound/inbound shortest distances from the sampled nodes. After that, it samples uniformly at random pairs of nodes among the sampled nodes again. For any pair of nodes  $u$  and  $v$ , we say that a node  $w$  covers the pair if  $d(u, v) - l_w(u, v) \leq \alpha d(u, v)$  where  $\alpha > 0$  is a user-specific parameter. Given the sampled pairs, we want to find a set of  $N_L$  landmarks maximizing the number of covered sampled pairs. However, computing such a landmark set is equivalent to the maximum coverage problem, which is NP-hard. Instead, we propose a greedy algorithm which iteratively picks  $k$  nodes each of which maximizes the marginal gain to the number of covered pairs. The entire procedure of this method is provided in the supplemental material, available online.

*Complexity.* Let  $N_1$  denote the number of the sampled nodes and  $N_2$  the number of the sampled pairs. We consider that  $N_1, N_2$ , and  $N_L$  are constant, so we use the same values for them over different datasets in the experiments. Thus, the time complexity of the procedure in the supplemental material, available online, is  $O(m + n \log n)$ .

Meanwhile, this method uses space mostly for storing shortest distances. In this method, shortest distances from/to sampled nodes are stored in  $O(n)$  space. Other structures are dominated by them.

## 6 BOOSTING TECHNIQUES

We have two novel boosting heuristic techniques for query efficiency: partial detouring and distance graph sparsification. They are used to make our oracles faster with a slight loss of accuracy.

### 6.1 Partial Detouring

*Idea Sketch.* Given a query  $(s, t, F)$ , suppose that an initial path  $P_{\text{init}}$  from  $s$  to  $t$ , which is like  $P(s, t)$ , is already computed by a fast algorithm. The partial detouring is a technique which computes the detours of certain edge-disjoint sub-paths of  $P_{\text{init}}$  having failures, called partial detours. Since this

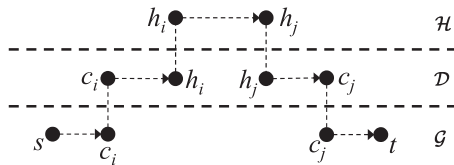


Fig. 3. Path Decomposition in the partial detouring.

technique deals with sub-paths of  $P_{init}$ , the search space of this is expected to be much smaller than that of computing  $P(s, t, F)$  from scratch, but the result may not be optimal.

Let us explain how we implement this technique. The explanation is based on *ADISO*, but the principle can be applied to *DISO*. Recall that the  $k$ -path cover is used as the transit node set. It can also be used for defining the sub-paths of a path, but we use another  $k'$ -path cover  $C_D$  in the distance graph  $D$ . From  $C_D$ , we can construct an overlay graph  $\mathcal{H} = (V_{\mathcal{H}}, E_{\mathcal{H}})$  where  $V_{\mathcal{H}} = C_D$ , which is a distance graph of  $D$ . We also have another inverted index from a node  $x$  in  $D$  to a node  $y$  in  $\mathcal{H}$  such that  $x$  participates in the bounded shortest path tree of  $y$  on  $D$ . If  $x$  is affected, then  $y$  is also defined to be affected.

Based on  $\mathcal{H}$ , the new query algorithm is as follows. Consider that given a query  $(s, t, F)$ , a modified version of *ADISO* is used to construct an initial path. The access nodes of  $s$  and  $t$  are normally computed in the graph  $(V, E \setminus F)$ . After that, Algorithm 2 completes the initial path  $P_{init}$  with the empty affected node set  $A = \emptyset$ . In this step, by considering the edges in  $\mathcal{H}$  as shortcuts,  $P_{init}$  can be efficiently computed. Then, it is easy to see that  $P_{init}$  can be decomposed as sub-paths  $\langle s \rightarrow c_i \rangle$ ,  $\langle c_i \rightarrow h_i \rangle$ ,  $\langle h_i \rightarrow h_j \rangle$ ,  $\langle h_j \rightarrow c_j \rangle$ , and  $\langle c_j \rightarrow t \rangle$ , where  $c_i \in V_D$  and  $h_i \in V_{\mathcal{H}}$ . Note that  $c_i$  and  $c_j$  are the access nodes of  $s$  and  $t$ . This decomposition is depicted in Fig. 3 where the dotted arrow represents a path. Since  $\langle s \rightarrow c_i \rangle$  and  $\langle c_j \rightarrow t \rangle$  are computed in the graph  $(V, E \setminus F)$ , we do not need to consider detouring. For the other sub-paths, each edge  $(x, y)$  of them, which is located on  $D$  or  $\mathcal{H}$ , is inspected to determine whether  $x$  is affected or not. If  $x$  is affected, then we compute the detour from  $x$  to  $y$  via *ADISO* without the shortcuts. In this way, by computing such detours, we can get the distance of the resulting path avoiding any failed edge.

For further speed-up, the lower bound function  $h(\cdot)$  is used to determine whether shortcuts are to be traversed or not. For any node  $u$  in  $D$  like  $h_i$  of Fig. 3, if  $u$  is also in  $\mathcal{H}$  and  $h(u, t) > \max_{n \in n_{\mathcal{H}}^{out}(u)} w_{\mathcal{H}}(u, n)$ , we traverse the shortcuts of  $u$ , but not the edges of it in  $D$ . Otherwise, the edges in  $D$  are only traversed. It is trivial that this usage does not cause additional accuracy loss.

*Discussion.* This approach is so efficient in terms of query time for two reasons. One reason is shortcutting with edges in  $\mathcal{H}$  when computing the initial path. The other is the fact that any edge on  $D$  or  $\mathcal{H}$  corresponds to a path of a bounded length on  $G$  because  $D$  and  $\mathcal{H}$  are constructed with path covers. Thus, the search space to compute a partial detour is effectively localized especially for bounded-degree networks.

One can worry the case that there is no detour of an edge  $(x, y)$  on the initial path even if  $x$  is affected. In that case, a simple remedy is to directly compute  $P(s, t, F)$  via *ADISO*. However, we expect that such a case happens very rarely, so the additional query time caused by it is ignorable. It should be noticed that such a case does not happen at all in the experiments.

## 6.2 Distance Graph Sparsification

Even if the distance graph is constructed with consideration of sparseness, it is still somewhat dense. Thus, there is some opportunity of improving our oracles further in terms of query time. For this, we introduce a simple sparsification method for the distance graph and experimentally show the effectiveness of it.

The key idea of the sparsification method is controlling the effect of removing each edge with a theoretical bound. Consider a simple case that we want to determine whether a single edge  $(x, y)$  on  $D$  can be removed or not. For a parameter  $\beta \geq 1$ , if a path from  $x$  to  $y$ , which does not include  $(x, y)$ , exists and its distance is lower than  $\beta w_D(x, y)$ , we can guarantee that for any shortest path  $P$  on  $D$  containing  $(x, y)$ , there is an alternative path  $P'$  such that  $d(P') \leq \beta d(P)$  without  $(x, y)$ . We can extend this idea to the case of removing multiple edges by tracking their cascaded effects on error. The detailed algorithm and the theoretical analysis are given in the supplemental material, available online, for the lack of space.

Let us denote the sparsified graph of  $D$  by  $\hat{D} = (V_{\hat{D}}, E_{\hat{D}})$ . The sparsification method guarantees that after removing edges from  $D$  by it, for any removed edge  $(u, v)$ , there is at least one path  $P$  on  $\hat{D}$  such that  $d(P) \leq \beta w_D(u, v)$ . This implies that the approximation error between shortest distances of  $D$  and  $\hat{D}$  is bounded by  $\beta$ . Note that this approximation bound is only valid if there is no failed edge. Otherwise, it is not. Nevertheless, we can still manipulate the error caused by the sparsification with  $\beta$ . Note that the errors of our oracles based on the sparsified distance graph are stable and small in practice.

One can worry that this technique removes out too many edges because we do not consider future failures when sparsifying. For example, if a node has only one remaining edge and the edge fails, then it cannot reach any other node. In order to handle this, we add a constraint that if the number of edges of a node is less than a certain number, we do not remove them. In the experiments, the number is 5 for a network whose average degree is larger than 10 and 3 for other networks. In addition to this, if the query algorithm fails to find the query answer, the Dijkstra's algorithm is used to answer the query. However, such failed cases are extremely rare.

## 7 EXPERIMENTS

We implement algorithms with C++ and run experiments on an Intel(R) i7-3970X 3.50 GHz CPU machine with 64 GB RAM.

### 7.1 Experimental Environment

*Datasets.* Real-world networks are usually scale-free or bounded-degree networks. As a representative of scale-free networks, we use three social network datasets: DBLP, Youtube, and Pokec. As a representative of bounded-degree networks, we use three road network datasets: NY, CAL, and USA. Epinions, DBLP, and Youtube were published by Jure Leskovec.<sup>2</sup> NY, CAL, and USA were published in the 9th DIMACS Implementation Challenge.<sup>3</sup> More specifically, DBLP is a co-authorship network where two authors are directly connected if they co-authored at least one paper. Youtube is a social network that is included in a video-

2. <http://snap.stanford.edu/data/>.

3. <http://www.dis.uniroma1.it/challenge9/>.

TABLE 2  
Statistics of the Real-Life Datasets

Dataset	$ V $	$ E $	Avg. deg.	Max deg.	Type
NY	264k	734k	2.8	8	Road
DBLP	317k	1M	6.6	0.7k	Social
YOU	1.1M	3.0M	5.3	57.7k	Social
POKE	1.6M	30.6M	18.8	20.5k	Social
CAL	1.9M	4.7M	2.5	7	Road
USA	24.0M	58.3M	2.4	9	Road

sharing web site where users can make friendship each other and it is abbreviated as YOU. Pokec is a popular online social network in Slovakia and it is abbreviated as POKE. NY is a road network of New York city, CAL is a road network of California and Nevada, and USA is the entire road network of USA. For simplicity, if there exist multiple edges defined over the same node pair, we only take the minimum weight edge. In addition, DBLP and YOU are undirected graphs. For the undirected graphs, we make them directed by adding an edge  $(v, u)$  for each edge  $(u, v) \in E$ . Table 2 shows the detailed statistics of all the datasets.

The core techniques in our oracles, which are the distance graph, the  $k$ -path cover and the A\* heuristics, are known to be more appropriate for bounded-degree networks than scale-free networks. Nevertheless, we include scale-free networks for experiments, because traditional techniques for computing distances on scale-free networks such as the landmark heuristics in [11] are not suitable for the distance sensitivity problem.

*Comparison Methods.* We have seven competitors as follows:

- *DISO* is our first distance sensitivity oracle.
- *ADISO* is our second distance sensitivity oracle. There are two parameters  $N_1$  and  $N_2$  for selecting landmarks.  $N_1$  and  $N_2$  are fixed to  $10N_L$  and 500 for all experiments, respectively. The other parameters will be discussed in detail.
- *DISO-S* is *DISO* with the sparsification technique. This is tested only for the scale-free networks. Since the query times of *DISO* and *ADISO* are similar for them, the sparsification technique is tested with *DISO*. The sparsification is applied to both the input graph and the distance graph with the same value of  $\beta$ . For DBLP and YOU,  $\beta = 1.5$  while for POKEC,  $\beta = 2.0$ .
- *ADISO-P* is *ADISO* with the partial detouring. This is tested only for the road networks. Note that for computing  $\mathcal{H}$ ,  $\theta$  is set to  $\infty$  and  $\tau$  is set to 4 for node reduction.
- *DI* is the classic (not bidirectional) Dijkstra's algorithm with a binary heap.
- A\* is the classical A\* search algorithm based on landmark heuristics proposed in [31]. For this algorithm, we use landmarks selected by a local search-based heuristic method, named *max-cover*, which was proposed in [33]. It is an improved version of the landmark selection method used in [31]. For reference, we implement our own version of the A\* search algorithm in [33] for the experiments, because it was designed for external memory.
- *FDDO* is a fully dynamic distance oracle without any theoretic guarantee for accuracy. This oracle is one of four algorithms proposed in [11], named *LCA* (meaning Lowest Common Ancestor), that has efficient query

time with good accuracy. For this algorithm, we use the landmark selection method of [11], named the best coverage technique. The number of landmarks for this algorithm is 50 with consideration of accuracy and efficiency. In addition, we revise their update algorithm to make it work for a weighted directed graph as described in [11]. Note that *FDDO* is an approximate distance oracle without any theoretical guarantee for accuracy.

Note that there are additional parameter sensitivity tests which are not mentioned in the following sections. They are included in the supplemental material, available online, because of the limitation of the space.

*Approximation.* *DISO-S*, *ADISO-P*, and *FDDO* are approximate algorithms. The average errors of *DISO-S* and *ADISO-P* are 0.6 and 2.9 percent, respectively, while that of *FDDO* is 1.6 percent. Thus, they are comparable in terms of accuracy.

*Edge Weights.* For road network datasets, we use the travel time as the weight of each edge. For social network datasets, since they do not provide any information about edge weights, we set the weight of each edge as a real value that is sampled uniformly at random from 0 to 1.

*Query Generation.* Because there is no well-known public dataset including a graph structure and failures, we synthetically generate queries. In order to generate a query, we randomly select two nodes  $s, t \in V$  as the source and the destination, respectively. Next, for determining a failed edge set, we introduce a syntactic failure generation method with a parameter  $f_{gen} \geq 0$  as follows.

First, a set  $F$  is initialized to  $\emptyset$ . Then, we iteratively pick uniformly at random an edge  $x$  in  $P(s, t, F)$ , add  $x$  into  $F$ , and recompute  $P(s, t, F)$  until  $|F| = f_{gen}$ . Because we pick an edge in  $P(s, t, F)$  at each iteration, every edge in  $F$  contributes to the change of the shortest path from  $s$  to  $t$ . Thus, this method makes  $P(s, t, F)$  much different from the shortest path from  $s$  to  $t$  in  $\mathcal{G}$ .

One merit of our failure generation method is that we can explicitly determine the number of essential failed edges for the result shortest path. It is sufficiently fair to the proposed oracles and *FDDO*, but it would be inappropriate to model real-world failures. This is because real-world failures are independent of the source and the destination. Thus, for each query, in addition to  $f_{gen}$  failed edges selected by our failure generation method, we use edges each of which is independently selected with probability  $p$  as failed edges.

In the experiments, if  $f_{gen}$  and  $p$  are not explicitly specified, we set that  $f_{gen} = 5$  and  $p = 0.05\%$ . The reason why the value of  $p$  is set to 0.05 percent is described in the supplemental material, available online. All results are averaged over 100 queries generated under the same environment. In addition, it should be noticed that sometimes we only present results for some of the datasets from experiments, because results for the other datasets show a tendency similar to that of the presented results.

## 7.2 Path Cover Computation

$\theta$  is the parameter for Algorithm 1 to determine a node  $u$ , which makes the minimum value of  $\sigma(u)$  for the current iteration, to be added into the output independent set. We provide experiments for analyzing  $\theta$  in the supplemental material, available online. From the experiments, we set  $\theta = 1$  for the road network datasets and  $\theta = 16$  for the social network datasets.

TABLE 3

Comparisons of Path Cover Computation Methods when  $k = 256$  ( $\tau = 8$ ) for Road Networks and  $k = 16$  ( $\tau = 4$ ) for Social Networks

	C			E <sub>D</sub>			Preprocessing time(s)			Query time(ms)			Recomputation time (ms)			Access time(ms)		
	ISC	PRU	HPC	ISC	PRU	HPC	ISC	PRU	HPC	ISC	PRU	HPC	ISC	PRU	HPC	ISC	PRU	HPC
NY	42.96k	-	22.79k	0.31M	-	0.56M	3.37	-	3.33	14.71	-	23.17	1.94	-	9.77	0.04	-	0.09
CAL	0.15M	-	95.45k	1.07M	-	1.82M	27.80	-	20.72	71.61	-	0.11k	16.75	-	55.16	0.15	-	0.24
USA	1.80M	-	1.14M	12.93M	-	22.04M	0.46k	-	0.30k	1.17k	-	1.95k	0.30k	-	1.0k	2.2	-	2.3
DBLP	56.86k	61.9K	49.54k	0.96M	2.45M	1.75M	8.37	19.98	6.55	86.63	0.14k	0.12k	4.56	28.45	16.95	0.05	0.06	0.06
YOU	0.12M	59.80k	63.25k	4.33M	39.79M	20.35M	0.27k	0.13k	0.17k	0.50k	5.20k	2.19k	90.58	3.65k	1.23k	0.18	0.26	0.20
POKE	0.88M	0.51M	0.57M	29.66M	1.08G	0.23G	0.28k	2.81k	0.52k	4.83k	92.27k	17.83k	0.13k	40.57k	4.39k	0.78	2.20	1.02

We compare the IS-based path cover algorithm, denoted as *ISC*, with the  $k$ -path cover algorithms in [10] and [27]. The pruning-based algorithm of [10] is denoted by *PRU*. In *PRU*, we set that nodes in  $V$  are visited in the increasing order of the sum of the in-degree and the out-degree of a node. This setting is easy to be implemented and presents good effectiveness in [10]. In addition, the algorithm of [27] is denoted by *HPC*. Since *HPC* hierarchically constructs a  $k$ -path cover using the concept of the vertex cover, we need a strategy to compute a vertex cover for a given graph. For this, we use the heuristic method, named *LR-deg*, which showed the best performance in [27]. We evaluated these comparisons in terms of the number of nodes, the number of edges, query time, preprocessing time, recomputation time, and access time. The recomputation time is the time for the lazy recomputation and the access time is the access node search time. We also report the average number of affected nodes in the supplemental material, available online.

The result of comparing the path cover computation methods for *DISO* is shown in Table 3 and Fig. 4. Recall that  $\tau$  is the user-parameter for the number of iterations in our path cover computation algorithm. As reported in [27], *PRU* greatly increases the number of edges of a resulting distance graph and the preprocessing time with  $\tau \geq 4$ . In the result, *ISC* shows better effectiveness than all the other methods in terms of reducing the number of edges in the resulting distance graph. Even if *ISC* has a slightly more number of nodes in the resulting distance graph than the other methods, it outperforms them in terms query time in most cases. Along with such effectiveness of *ISC*, the preprocessing time with *ISC* is comparable to that with *HPC*. This result

demonstrates that *ISC* is more appropriate for our oracles than any other methods.

In addition to the  $k$ -path cover, the set of the border nodes of graph partitioning can be the transit node set. A border node is a node having a neighbor included in a different partition. In [17] and [35], border nodes act like transit nodes for shortest path computation. Thus, we compare our  $k$ -path cover algorithm with existing graph partitioning algorithms. The partitioning algorithm used in [35] is a famous algorithm, named *METIS* [34]. Since the stochastic partitioning algorithm in [17], denoted by *SPA*, is not fully described either, our implementation for it may not be fully optimized. The number of partitions is experimentally chosen as 3,000, which makes reasonable query performance. Table 4, in which *UNIFORM* denotes uniform random partitioning, shows that *ISC* is better than the partitioning algorithms in terms of query time. For NY, *ISC* achieves a more sparse distance graph than the other algorithms. For POKE, it also achieves a much smaller distance graph than them. This is because the objectives of the graph partitioning algorithms are not related to the resulting distance graph. Instead, they are related to the number of edges or the sum of edge weights between partitions, so that the distance graphs given by such algorithms can be dense or large. Meanwhile, since *SPA* cannot handle large graphs due to multiple eigenvalue computations, it is only evaluated for NY.

As shown in Fig. 4, selecting an appropriate value for  $k$  (i.e.,  $\tau$ ) is crucial for the query performance of our oracles. For the rest of the experiments,  $\tau$  of *DISO* is 4 for the social network datasets and 8 for the road network datasets.  $\tau$  of *ADISO* is 3 for the social network datasets and 7 for the road network datasets.  $\tau$  of *ADISO-P* is 15.

### 7.3 Landmark Selection

$\alpha$  is the parameter for determining whether a node  $w$  covers a sampled node pair in the landmark selection. We conduct sensitivity test for this and the results are shown in the supplemental material, available online. From this test,  $\alpha$  is set to 0.1 for road network datasets and 0.25 for the social network datasets.

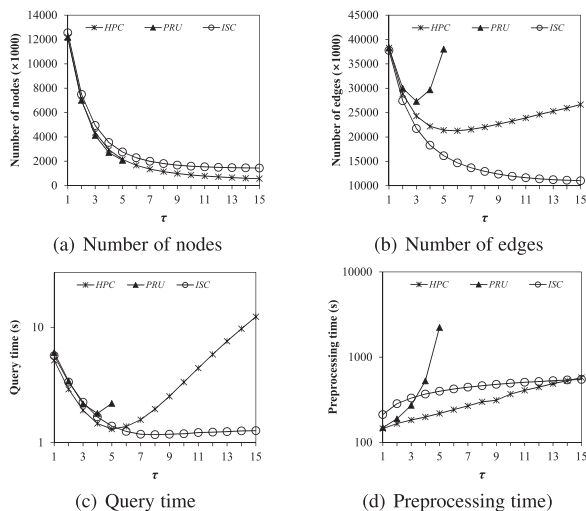
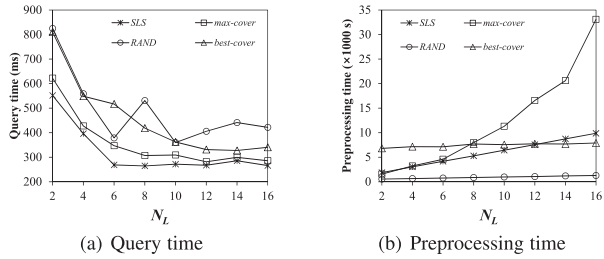


Fig. 4. Comparisons of path cover selection methods (USA).

TABLE 4  
Comparing *ISC* with Graph Partitioning Algorithms for NY and POKE (QT: Query Time, AT: Access Time)

	NY				POKE			
	C	E <sub>D</sub>	QT(ms)	AT(ms)	C	E <sub>D</sub>	QT(ms)	AT(ms)
<i>ISC</i>	42.96k	0.31M	14.71	0.04	0.88M	29.66M	5.10k	0.88
<i>UNIFORM</i>	0.26M	0.73M	64.04	0.01	1.63M	30.62M	7.06k	0.87
<i>METIS</i> [34]	47.17k	0.56M	21.09	0.07	1.43M	30.35M	6.66k	0.89
<i>SPA</i> [17]	42.52k	0.54M	41.21	0.82	-	-	-	-

Fig. 5. Comparing landmark selection methods with  $N_L$  (USA).

For ADISO, we compare the proposed Sampling-based Landmark Selection method (*SLS*) with three existing landmark selection methods. One is a random selection method, denoted as *RAND*, and another is *max-cover*, which is a local search-based heuristic selection method. They were introduced in [33]. The last method is a sampling-based method, denoted by *best-cover*, which was used to select landmarks for *FDDO* in [11]. There is another sampling-based method in [36], but it was shown to be less effective than *best-cover* in [11].

Note that since the  $A^*$  heuristics are not much helpful for the social networks, the results for them are reported in the supplemental material, available online. Fig. 5 shows that *SLS* has better performance than *max-cover* in terms of query time with much smaller preprocessing time. *SLS* is also better than *best-cover* in terms of query time and their preprocessing times are comparable. Compared to *RAND*, *SLS* stably finds appropriate landmarks to make ADISO have efficient query time with reasonable preprocessing time.

With the consideration of query time and preprocessing time, when  $N_L = 10$ , ADISO shows good performance for all the datasets. We also use 10 landmarks for  $A^*$  for fairness.

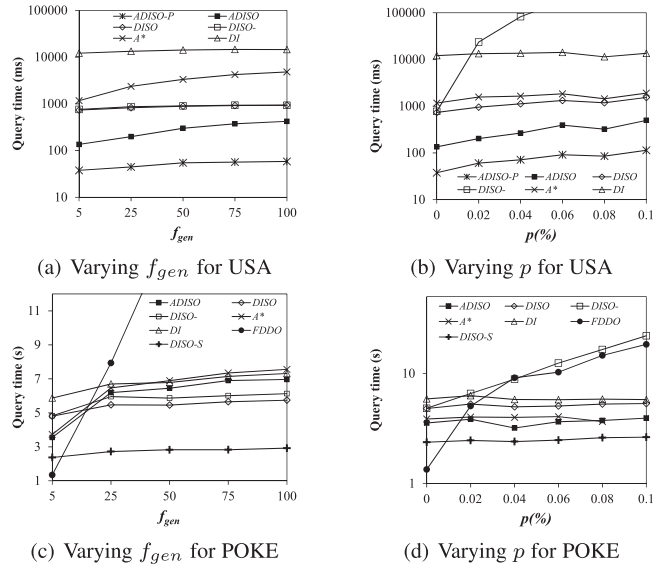
## 7.4 Overall Performance

### 7.4.1 Query Efficiency

We omit some resulting query times of *FDDO* for USA, because it requires too much time to run for that dataset.

**Robustness Test.** We conduct experiments with different sizes of  $F$ . Fig. 6 depicts the results. The size of  $F$  is determined by  $f_{gen}$  and  $p$ . With a large value of  $f_{gen}$ , we can see how fast an algorithm answers a query when the resulting shortest path of it is significantly different from the original shortest path. With a large value of  $p$ , we can evaluate an algorithm in terms of the capability for handling failed edges that are not essential for the resulting shortest path.

In Fig. 6, *DISO-* is a variation of *DISO* which does not utilize the bounded shortest path trees at all. Instead, it uses the breath-first search to find affected nodes and the bounded Dijkstra's algorithm to recompute the edge weights associated with them. As  $p$  gets bigger, the query time of

Fig. 6. Query time with different sizes of  $F$ .

*DISO-* gets even bigger, while that of *DISO* does not increase that much. This implies that the proposed technique using the bounded shortest path trees is significantly helpful for our oracles to efficiently handle failed edges.

In Fig. 6a, *ADISO*, *ADISO-P*, and  $A^*$  similarly get slower as  $f_{gen}$  increases, because they are devised via the landmark-based  $A^*$  heuristics. When  $f_{gen}$  is large, the lower bound distances derived from the  $A^*$  heuristics get incorrect, which hinders the algorithms from visiting nodes closer to the destination early. Nevertheless, for the change of  $p$ , *ADISO*, *ADISO-P*, and  $A^*$  are relatively robust in Fig. 6b. Figs. 6c and 6d describe the results for POKE. Since it is scale-free, *DISO* and *ADISO* are not that efficient, but *DISO-S* shows good query performance based on the sparsification.

**Bounded-Degree Networks.** We evaluate the comparison algorithms in terms of query time over the road network datasets. The results are shown in Table 5. As expected, *ADISO-P* is the fastest and *ADISO* is the second fastest in terms of query time. It is remarkable that *DISO* has better performance than  $A^*$ , even though it does not utilize the  $A^*$  heuristics at all. Quantitatively, *DISO* is about 9 times faster than *DI*. *ADISO* is about 5 times faster than  $A^*$ . *ADISO* is about 3 times faster than *DISO*. *ADISO-P* is about 3 times faster than *ADISO*. Compared to *FDDO*, *ADISO-P* is three orders of magnitude faster on average with better accuracy.

**Scale-Free Networks.** The results for the scale-free networks are also shown in Table 5. It is remarkable that even for the social network datasets, our oracles are more efficient than any other algorithm in most cases. *DISO* and

TABLE 5  
Query Time and Preprocessing Time over Datasets

Data	Query time(ms)								Preprocessing time(s)					
	<i>DISO-</i>	<i>DISO</i>	<i>ADISO</i>	<i>DISO-S</i>	<i>ADISO-P</i>	<i>FDDO</i>	$A^*$	<i>DI</i>	<i>DISO</i>	<i>ADISO</i>	<i>DISO-S</i>	<i>ADISO-P</i>	<i>FDDO</i>	$A^*$
NY	0.10k	14.71	4.96	-	<b>2.03</b>	6.77k	16.52	76.92	<b>3.37</b>	36.27	-	36.78	48.69	0.11k
CAL	1.18k	71.61	31.85	-	<b>7.43</b>	0.15M	0.16k	0.68k	<b>27.80</b>	0.32k	-	0.33k	0.42k	0.66k
USA	0.12M	1.17k	0.27k	-	<b>73.66</b>	-	1.52k	13.23k	<b>0.46k</b>	6.52k	-	6.92k	-	11.30k
DBLP	0.41k	86.63	0.11k	<b>55.63</b>	-	1.04k	0.30k	0.51k	<b>8.37</b>	0.18k	69.23	-	0.27k	0.14k
YOU	12.70k	0.51k	0.77k	<b>0.23k</b>	-	10.53k	1.22k	2.36k	<b>0.29k</b>	1.14k	3.04k	-	1.53k	0.76k
POKE	10.41k	5.10k	4.37k	<b>2.32k</b>	-	10.54k	3.99k	5.89k	<b>0.30k</b>	2.09k	6.26k	-	3.17k	1.56k

TABLE 6  
The index size of *DISO*, *ADISO*, *FDDO*, and *A\**

Data	Index size(MB)			
	<i>DISO</i>	<i>ADISO</i>	<i>FDDO</i>	<i>A*</i>
NY	<b>37.84</b>	122.69	422.95	84.59
CAL	<b>220.07</b>	826.88	3,025.30	605.06
USA	<b>2,799.13</b>	10,481.05	38,315.76	7,663.15
DBLP	<b>47.14</b>	150.03	507.33	101.47
YOU	<b>184.40</b>	547.34	1,815.82	363.16
POKE	1,126.14	1,646.61	2,612.48	<b>522.50</b>

*ADISO* are somewhat slower than the other competitors in terms of query time in POKE. This is because the distance graphs of our oracles are too dense in POKE whose average degree is larger than 18. Nevertheless, for the other social network datasets, our oracles are more efficient than any other algorithm. In addition, *DISO-S* is the most efficient, so we can see that the sparsification technique effectively works. It is up to 2.2 times faster than *DISO*. Even for the scale-free networks, our oracles can have reasonable query time with the sparsification.

Even if *FDDO* is an approximate algorithm and its structure is very simple, *FDDO* takes a significant time to update its structures in querying. That is why *FDDO* is extremely slower than any other competitor in all the cases. From this case, we can see that even a simple fully dynamic distance oracle like *FDDO* turns out to be inappropriate for the distance sensitivity problem.

#### 7.4.2 Preprocessing Time and Space

We evaluate the competitors in terms of preprocessing time and space (index size). The results are presented in Tables 5 and 6. *ADISO-P* and *DISO-S* are not presented either, because their sizes are comparable to those of *ADISO* and *DISO*, respectively.

The index size of *DISO* is smaller than that of any other methods containing preprocessed data in most cases. Compared to *A\**, *ADISO* is still somewhat behind in terms of index size, but the difference between *ADISO* and *A\** is not large. Even if they have comparable index sizes, *ADISO* is more efficient than *A\** in terms of query time.

In terms of preprocessing time, *DISO* is also more efficient than any other methods containing preprocessed data. Because of the efficient landmark selection method used for *ADISO*, its preprocessing time is comparable to that of *A\**.

#### 7.4.3 Summary

Overall, our distance sensitivity oracles outperform the competitors in terms of query time in most cases. It should be noted that *DISO* is also faster than *FDDO* and *A\** in terms of preprocessing time with smaller space. *ADISO* is the most efficient exact method for the road networks in terms of query time with comparable preprocessing time and space. In addition, we can see that the partial detouring and the distance graph sparsification effectively make our oracles faster.

## 8 CONCLUSIONS

The distance sensitivity problem is an important variation of the point-to-point shortest distance problem which can be used in many applications. It has been mainly studied in theory literature, but all the existing works of this

problem for directed graphs suffer from prohibitively expensive space and preprocessing time.

This paper presents two practical distance sensitivity oracles that efficiently work for directed graphs. The first oracle is based on the fault-tolerant index structure consisting of the distance graph and the bounded shortest path trees. We devise the efficient query algorithm for the oracle that answers a query without any stalling queries. In order to compute a good distance graph for this oracle, we utilize the relationship between the *k*-path cover and the independent set. Next, we propose the second oracle by applying the *A\** heuristics to the first oracle. In addition, we propose two speed-up techniques for making our oracles faster with a slight loss of accuracy. Finally, we introduce efficient maintenance strategies for our oracles to handle graph updates.

We conduct extensive experiments to evaluate our oracles. The results demonstrate that our distance sensitivity oracles outperform the competitors in terms of query time in most cases. Our first oracle is more efficient than the competitors in terms of preprocessing time and space. It is notable that our first oracle has mostly better query performance than even the *A\** search algorithm. Our second oracle has the best query performance for the road networks with reasonable preprocessing time and space. Moreover, the partial detouring and the distance graph sparsification effectively make our oracles faster. Based on these achievements, this paper is the first work for the distance sensitivity oracles that handle practical graphs with million-level nodes.

## ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers and editors. This work was supported in part by 2018 Seed Money Project of Chongqing Liangjiang KAIST International Program, Chongqing University of Technology, and in part by Chongqing Research Program of Basic Research and Frontier Technology (No. cstc2017jcyjAX0089).

## REFERENCES

- [1] C. Demetrescu, M. Thorup, R. A. Chowdhury, and V. Ramachandran, "Oracles for distances avoiding a failed node or link," *SIAM J. Comput.*, vol. 37, no. 5, pp. 1299–1318, Jan. 2008.
- [2] A. Bernstein and D. Karger, "Improved distance sensitivity oracles via random sampling," in *Proc. 19th Annu. ACM-SIAM Symp. Discr. Algorithms*, 2008, pp. 34–43.
- [3] A. Bernstein and D. Karger, "A nearly optimal oracle for avoiding failed vertices and edges," in *Proc. 41st Annu. ACM Symp. Theory Comput.*, 2009, pp. 101–110.
- [4] S. Chechik, M. Langberg, D. Peleg, and L. Roditty, "F-sensitivity distance oracles and routing schemes," in *Proc. Eur. Symp. Algorithms*, 2010, pp. 84–96.
- [5] R. Duan and S. Pettie, "Dual-failure distance and connectivity oracles," in *Proc. 20th Annu. ACM-SIAM Symp. Discr. Algorithms*, 2009, pp. 506–515.
- [6] S. Baswana, U. Lath, and A. S. Mehta, "Single source distance oracle for planar digraphs avoiding a failed node or link," in *Proc. 23rd Annu. ACM-SIAM Symp. Discr. Algorithms*, 2012, pp. 223–232.
- [7] Y. Qin, Q. Z. Sheng, and W. E. Zhang, "SIEF: Efficiently answering distance queries for failure prone graphs," in *Proc. 18th Int. Conf. Extending Database Technol.*, 2015, pp. 145–156.
- [8] O. Weimann and R. Yuster, "Replacement paths via fast matrix multiplication," in *Proc. IEEE 51st Annu. Symp. Found. Comput. Sci.*, Oct. 2010, pp. 655–662.
- [9] O. Weimann and R. Yuster, "Replacement paths and distance sensitivity oracles via fast matrix multiplication," *ACM Trans. Algorithms*, vol. 9, no. 2, pp. 14:1–14:13, Mar. 2013.

- [10] S. Funke, A. Nusser, and S. Störandt, "On k-path covers and their applications," *Proc. VLDB Endowment*, vol. 7, no. 10, pp. 893–902, 2014.
- [11] K. Tretyakov, A. Armas-Cervantes, L. García-Bañuelos, J. Vilo, and M. Dumas, "Fast fully dynamic landmark-based estimation of shortest path distances in very large graphs," in *Proc. 20th ACM Int. Conf. Inf. Knowl. Manage.*, 2011, pp. 1785–1794.
- [12] J. Cheng, Y. Ke, S. Chu, and C. Cheng, "Efficient processing of distance queries in large graphs: A vertex cover approach," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2012, pp. 457–468.
- [13] A. W.-C. Fu, H. Wu, J. Cheng, and R. C.-W. Wong, "IS-Label: An independent-set based labeling scheme for point-to-point distance querying," *Proc. VLDB Endowment*, vol. 6, no. 6, pp. 457–468, 2013.
- [14] T. Akiba, Y. Iwata, and Y. Yoshida, "Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling," in *Proc. 23rd Int. Conf. World Wide Web*, 2014, pp. 237–248.
- [15] T. Hayashi, T. Akiba, and K.-I. Kawarabayashi, "Fully dynamic shortest-path distance query acceleration on massive networks," in *Proc. 25th ACM Int. Conf. Inf. Knowl. Manage.*, 2016, pp. 1533–1542.
- [16] D. Delling and D. Wagner, "Landmark-based routing in dynamic graphs," in *Proc. 6th Int. Conf. Experimental Algorithms*, 2007, pp. 52–65.
- [17] L. H. U, H. J. Zhao, M. L. Yiu, Y. Li, and Z. Gong, "Towards online shortest path computation," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 4, pp. 1012–1025, Apr. 2014.
- [18] D. Schultes and P. Sanders, "Dynamic highway-node routing," in *Proc. 6th Int. Conf. Experimental Algorithms*, 2007, pp. 66–79.
- [19] F. Bruera, S. Cicerone, G. D'Angelo, G. Di Stefano, and D. Frigioni, "Dynamic multi-level overlay graphs for shortest paths," *Math. Comput. Sci.*, vol. 1, no. 4, pp. 709–736, Jun. 2008.
- [20] G. Nannicini, P. Baptiste, G. Barbier, D. Kroh, and L. Liberti, "Fast paths in large-scale dynamic road networks," *Comput. Optim. and Appl.*, vol. 45, no. 1, pp. 143–158, 2010.
- [21] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. Syst. Sci. Cybern.*, vol. SSC-4, no. 2, pp. 100–107, Jul. 1968.
- [22] E. P. Chan and Y. Yang, "Shortest path tree computation in dynamic graphs," *IEEE Trans. Comput.*, vol. 58, no. 4, pp. 541–557, Apr. 2009.
- [23] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck, "Customizable route planning," in *Proc. Int. Symp. Experimental Algorithms*, 2011, pp. 376–387.
- [24] P. Sanders and D. Schultes, "Highway hierarchies hasten exact shortest path queries," in *Proc. Eur. Symp. Algorithms*, 2005, pp. 568–579.
- [25] R. Geisberger, P. Sanders, D. Schultes, and D. Delling, "Contraction hierarchies: Faster and simpler hierarchical routing in road networks," in *Proc. Int. Workshop Experimental Efficient Algorithms*, 2008, pp. 319–333.
- [26] R. Geisberger, P. Sanders, D. Schultes, and C. Vetter, "Exact routing in large road networks using contraction hierarchies," *Transp. Sci.*, vol. 46, no. 3, pp. 388–404, 2012.
- [27] T. Akiba, Y. Yano, and N. Mizuno, "Hierarchical and dynamic k-path covers," in *Proc. 25th ACM Int. Conf. Inf. Knowl. Manage.*, 2016, pp. 1543–1552.
- [28] H. Bast, S. Funke, P. Sanders, and D. Schultes, "Fast routing in road networks with transit nodes," *Sci.*, vol. 316, no. 5824, pp. 566–566, 2007.
- [29] J. Arz, D. Luxen, and P. Sanders, "Transit node routing reconsidered," in *Proc. Int. Symp. Experimental Algorithms*, 2013, pp. 55–66.
- [30] B. Brešar, F. Kardoš, J. Katrenič, and G. Semanišin, "Minimum k-path vertex cover," *Discr. Appl. Math.*, vol. 159, no. 12, pp. 1189–1195, Jul. 2011.
- [31] A. V. Goldberg and C. Harrelson, "Computing the shortest path: A\* search meets graph theory," in *Proc. 16th Annu. ACM-SIAM Symp. Discr. Algorithms*, 2005, pp. 156–165.
- [32] R. Bauer, T. Columbus, B. Katz, M. Krug, and D. Wagner, "Preprocessing speed-up techniques is hard," in *Proc. Int. Conf. Algorithms Complexity*, 2010, pp. 359–370.
- [33] A. V. Goldberg and R. F. F. Werneck, "Computing point-to-point shortest paths from external memory," in *Proc. 7th Workshop Algorithm Eng. Experiments and the 2nd Workshop Analytic Algorithmics Combinatorics*, 2005, pp. 26–40.
- [34] G. Karypis and V. Kumar, "Analysis of multilevel graph partitioning," in *Proc. ACM/IEEE Conf. Supercomput.*, 1995, Art. no. 29.
- [35] R. Zhong, G. Li, K. Tan, L. Zhou, and Z. Gong, "G-tree: An efficient and scalable index for spatial search on road networks," *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 8, pp. 2175–2189, Aug. 2015.
- [36] M. Potamias, F. Bonchi, C. Castillo, and A. Gionis, "Fast shortest path distance estimation in large networks," in *Proc. 18th ACM Conf. Inf. Knowl. Manage.*, 2009, pp. 867–876.



**Jong-Ryul Lee** received the BS and PhD degrees in computer science from Korea Advanced Institute of Science and Technology (KAIST), Korea. He is a postdoctoral researcher with the Department of Computer Science, KAIST. His research interests include graph data management, deep learning, visual question answering, and network analysis.



**Chin-Wan Chung** received the BS degree in electrical engineering from Seoul National University, Korea, and the PhD degree in computer science from the University of Michigan, Ann Arbor. He is a professor with the School of Computing, Korea Advanced Institute of Science and Technology (KAIST), Korea, and the Chongqing Liangjiang KAIST International Program at Chongqing University of Technology (CQUT), China. From 1983 to 1993, he was a senior research scientist and a staff research scientist with the Computer Science Department, General Motors Research Laboratories (GMR). While at GMR, he developed Dataplex, a heterogeneous distributed database management system integrating different types of databases. At KAIST, he developed a full-scale object-oriented spatial database management system called OMEGA, and a semantic mobile social network called SIMSON. He has been on program committees of major international conferences such as ACM SIGMOD, VLDB, IEEE ICDE, WWW, and IEEE ICWS. He was an associate editor of *ACM TOIT*, and is an associate editor of the *WWW Journal*. He was the general chair of WWW 2014. His current research interests include the Web, social networks, and data/knowledge management and analysis.

▷ For more information on this or any other computing topic, please visit our Digital Library at [www.computer.org/csdl](http://www.computer.org/csdl).