Proxies for Shortest Path and Distance Queries

Shuai Ma, Kaiyu Feng, Jianxin Li, Haixun Wang, Gao Cong, and Jinpeng Huai

Abstract—Computing shortest paths and distances is one of the fundamental problems on graphs, and it remains a *challenging* task today. This article investigates a light-weight data reduction technique for speeding-up shortest path and distance queries on large graphs. To do this, we propose a notion of *routing proxies* (or simply proxies), each of which represents a small subgraph, referred to as deterministic routing areas (DRAs). We first show that routing proxies hold good properties for speeding-up shortest path and distance queries. Then we design a *linear-time* algorithm to compute routing proxies and their corresponding DRAs. Finally, we experimentally verify that our solution is a general technique for reducing graph sizes and speeding-up shortest path and distance queries, using real-life large graphs.

Index Terms—Shortest paths, shortest distances, data reduction, speeding-up techniques

1 Introduction

We study the *node-to-node shortest path* (*distance*) problem on large graphs: given a weighted undirected graph $G(V, E)$ with non-negative edge weights, and two nodes of G, the source s and the target t, find the shortest path (distance) from s to t in G. We allow the usage of auxiliary structures generated by preprocessing, but restrict them to have a moderate size (compared with the input graph). In this work, we are only interested in shortest paths and *exact* shortest distances on *large* graphs.

Finding shortest paths and distances is one of the fundamental problems on graphs, and has found its usage as a building block in various applications, *e.g.,* measuring the closeness of nodes in social networks and Web graphs [21], [29], [33], finding the distances between physical locations in road networks [38], and drivers' routing services [23].

Algorithms for computing shortest paths and distances have been studied since 1950's and still remain an *active* area of research. The classical one is Dijkstra's algorithm [6] due to Edsger Dijkstra. Dijkstra's original algorithm runs in $O(n^2)$ [10], and the enhanced implementation with Fibonacci heaps runs in $O(n \log n + m)$ due to Fredman & Tarjan [13], where n and m denote the numbers of nodes and edges in a graph, respectively. The latter remains asymptotically the fastest known solution on arbitrary undirected graphs with non-negative edge weights [35].

However, computing shortest paths and distances remains a challenging problem, in terms of both time and space cost, for large-scale graphs such as Web graphs, social networks and road networks. The Dijkstra's algorithm [13] is not acceptable on large graphs (*e.g.,* with tens of millions of nodes and edges) for online applications [29]. Therefore, a lot of optimization techniques have been recently developed to speed up the computation [5], [14], [24], [29], [30], [32], [33], [37], [38].

To speed-up shortest path and distance queries, our approach is to use *representatives*, each of which captures a set of nodes in a graph. The task of finding a proper form of representatives is, however, *nontrivial*. Intuitively, we expect representatives to have the following properties. (1) A small number of representatives can represent a large number of nodes in a graph; (2) Shortest paths and distances involved within the set of nodes being represented by the same representative can be answered efficiently; And, (3) the representatives and the set of nodes being represented can be computed efficiently.

Contributions & Roadmap. We develop a light-weight data reduction technique for speeding-up shortest path and distance queries on large weighted undirected graphs.

(1) We first propose a notion of *routing proxies* (or simply proxies), each of which represents a small subgraph, referred to as *deterministic routing areas* (DRAs) (Section 3). We also give an analysis of routing proxies and DRAs, which shows that they hold good properties for speeding-up shortest path and distance queries.

(2) We then develop a *linear-time* algorithm for computing deterministic routing areas along with their maximal routing proxies (Section 4). This makes our solution a light-weight technique that is scalable to large graphs.

(3) Using real-life large road and co-authorship (sparse) graphs, we finally conduct an extensive experimental study (Section 6). We find that (a) on average 1/3 nodes in these graphs are captured by routing proxies, leaving the reduced graph about 2/3 of the original input graph for both road and co-authorship graphs, (b) proxies can be found efficiently, *e.g.,* they can be found in at most 6 seconds for co-authorship graphs and less than half an hour for the largest road graph, and (c) the reduced graph incurs only about 70% space overhead of the original input graph on average. Moreover, (d) using proxies benefits existing shortest path and distance algorithms in terms of time cost, *e.g.,* it reduces

- *S. Ma, J. Li and J. Huai are with the SKLSDE lab, School of Computer Science and Engineering, Beihang University, China. E-mail: {mashuai, lijx, huaijp}@buaa.edu.cn.*
- *K. Feng and G. Cong are with the School of Computer Engineering, Nanyang Technological University, Singapore. E-mail: kfeng002@e.ntu.edu.sg, gaocong@ntu.edu.sg.*
- *H. Wang is with Facebook Inc., USA. E-mail: haixun@gmail.com.*

Manuscript received XXX, 2015; revised XXX, 2015.

around 30%, 20% and 1% time for bidirectional Dijkstra [24], ARCFLAG [27] and AH [41] on road graphs, respectively, and 49%, 4% and 49% time for bidirectional Dijkstra, ARCFLAG and TNR [2] on co-authorship graphs, respectively, and (e) existing shortest path and distance algorithms also benefit from using proxies in terms of space overhead, *e.g.*, it reduces about 62%, 28% and 18% space overhead for ARCFLAG, TNR and AH on road graphs, respectively, and about 86% and 2% space overhead for ARCFLAG and TNR on co-authorship graphs, respectively. (f) Using synthetic data, we further find that proxies and DRAs are sensitive to the density and degree distribution of graphs and perform well on graphs that follow the power law distribution, and, moreover, for a given degree distribution, less nodes are captured when the average degree is higher.

2 GRAPH NOTIONS

In this section, we introduce some basic graph concepts.

Graphs. A *weighted undirected graph* (or simply a *graph*) is defined as $G(V, E, w)$, where (1) V is a finite set of nodes; (2) $E \subseteq V \times V$ is a finite set of edges, in which (u, v) or $(v, u) \in E$ denotes an undirected edge between nodes u and v; and (3) w is a total weight function that maps each edge in E to a positive rational number.

We simply denote $G(V, E, w)$ as $G(V, E)$ when it is clear from the context.

Subgraphs. Graph $H(V_s, E_s, w_s)$ is a *subgraph* of graph $G(V, E, w)$ if (1) $V_s \subseteq V$, (2) $E_s \subseteq E$, and (3) $w_s(e) = w(e)$ for each edge $e \in E_s$. That is, subgraph H simply contains a subset of nodes and a subset of edges of graph G.

We also denote a subgraph H as $G[V_s]$ if E_s is exactly the set of edges appearing in G over the set of nodes V_s.

Neighbors. We say that node v is a *neighbor* of node u if there exists an edge (v, u) or (u, v) in graph G.

Paths and cycles. A *simple path* (or simply a *path*) ρ is a sequence of nodes $v_1/\ldots/v_n$ with no repeated nodes, and, moreover, for each $i \in [1, n-1]$, (v_i, v_{i+1}) is an edge in G.

A *simple cycle* (or simply a *cycle*) ρ is a sequence of nodes $v_1/\ldots/v_n$ with $v_1 = v_n$ and no other repeated nodes, and, moreover, for each $i \in [1, n-1]$, (v_i, v_{i+1}) is an edge in G.

The *length* of a path or cycle ρ is the sum of the weights of its constituent edges, *i.e.*, $\sum_{i=1}^{n-1} w(v_i, v_{i+1})$.

We also say that a node is *reachable* to another one if there exists a path between these two nodes.

Shortest paths and distances. A *shortest path* from one node u to another node v, denoted as $path(u, v)$, is a path whose length is minimum among all the paths from u to v.

The *shortest distance* between nodes u and v, denoted as $dist(u, v)$, is the shortest length of all paths from u to v, *i.e.*, the length of a shortest path from u to v.

Connected components. A *connected component* (or simply a CC) of a graph is a subgraph in which any two nodes are connected by a path, and is connected to no additional nodes. A graph is connected if it has exactly one connected component, consisting of the entire graph.

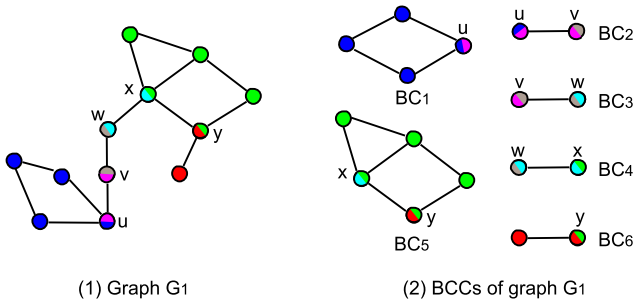


Figure 1. Cut-nodes and bi-connected components

Cut-nodes and bi-connected components. A *cut-node* of a graph is a node whose removal increases the number of connected components in the graph.

A *bi-connected component* (or simply a BCC) of a graph is a subgraph consisting of a maximal set of edges such that any two edges in the set must lie on a common simple cycle.

We next illustrate cut-nodes and bi-connected components with an example.

Example 1: Consider graph G_1 in Fig. 1(1), in which labeled nodes u, v, w, x, y are the cut-nodes of G_1, and the corresponding BCCs of G_1 are $BC_1, BC_2, BC_3, BC_4, BC_5$, and BC_6, and are shown in Fig. 1(2). □

3 ROUTING PROXIES

In this section, we first propose *routing proxies* and *deterministic routing areas* (DRAs) to capture the idea of representatives and the set of nodes being represented, respectively. We then give an analysis of the properties of DRAs and their routing proxies, and show that they indeed hold the desired properties of representatives discussed in Section 1. For clarity, the detailed proofs in this article are in the appendix.

3.1 Routing Proxies and Deterministic Routing Areas

We first present the notions of routing proxies and their deterministic routing areas.

Proxies. Given a node u in graph $G(V, E)$, we say that u is a *routing proxy* (or simply *proxy*) of a set of nodes, denoted by A_u, if and only if:

(1) node $u \in A_u$ is reachable to any node of A_u in G,

(2) all neighbors of any node $v \in A_u \setminus \{u\}$ are in A_u, and

(3) the size $|A_u|$ of A_u is equal to or less than $c \cdot \lfloor\sqrt{|V|}\rfloor$, where c is a small constant number, such as 2 or 3.

Here condition (1) guarantees the connectivity of subgraph $G[A_u]$, condition (2) implies that not all neighbors of proxy u are necessarily in A_u; and condition (3), referred to as *size restriction*, limits the size of A_u of proxy u. Intuitively, one simply checks the graph by removing u from G and its newly created CCs, and a proxy of u is a union of such CCs whose total number of nodes is at most $c \cdot \lfloor\sqrt{|V|}\rfloor - 1$.

Deterministic routing areas. A node u may be a proxy of multiple sets of nodes $A_u^1, \ldots, A_u^k$. We denote as A_u^+ the union of all the sets of nodes whose proxy is u , *i.e.*, $A_u^+ = A_u^1 \cup \ldots \cup A_u^k$, and A_u^i ($i \in [1, k]$) is said a component of A_u^+.

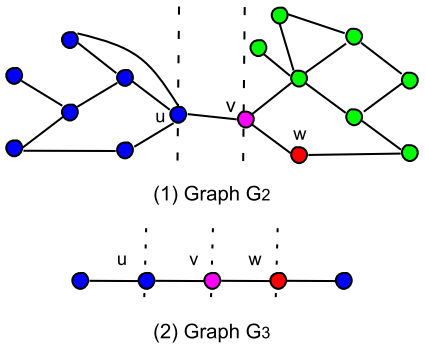


Figure 2. Example proxies and DRAs

We refer to the *subgraph* $G[A_u^+]$ as a deterministic routing area (DRA) of proxy u.

Intuitively, DRA $G[A_u^+]$ is a *maximal* connected subgraph, union of a set of CCs, that connects to the rest of graph G through proxy u only. That is, for any node v in $G[A_u^+]$ and any node v in the rest of graph G, u must appear on the shortest path $path(v, v')$.

Maximal proxies. We say that a proxy u is *maximal* if there exist no other proxies u' such that $u' \neq u$ and $A_u^+ \subset A_{u'}^+$.

Trivial proxies. We say that a maximal proxy u is *trivial* if A_u^+ contains itself only, *i.e.*, $A_u^+ = \{u\}$.

Equivalent proxies. We say that two proxies u and u' are *equivalent*, denoted by $u \equiv u'$, if $A_u^+ = A_{u'}^+$.

We next illustrate these notions with an example below.

Example 2: First consider graph $G_2(V_2, E_2)$ in Fig. 2, and let $c \cdot \lfloor\sqrt{|V_2|}\rfloor = 2 \cdot \lfloor\sqrt{16}\rfloor = 8$, where $c = 2$ and $|V_2| = 16$. (1) Node u is a proxy, and its DRA is the subgraph in the left hand side of the vertical line across u; (2) Node v is a proxy, and its DRA is the subgraph in the left hand side of the vertical line across v; (3) Node w is not a proxy since it can not find a DRA with size less or equal than 8; (4) Node v is a maximal proxy, while node u is not a maximal proxy since $A_u^+ \subset A_v^+$.

We then consider graph $G_3(V_3, E_3)$ in Fig. 2, and let $c \cdot \lfloor\sqrt{|V_3|}\rfloor = 2 \cdot \lfloor\sqrt{5}\rfloor = 4$, where $c = 2$ and $|V_3| = 5$. (1) Nodes u, v and w are three maximal proxies, whose DRAs are all the entire graph G_3, and, hence, (2) u, v and w are three equivalent proxies. □

Remark. As illustrated by the above examples, a DRA of graph $G(V, E)$ may have a size larger than $c \cdot \lfloor\sqrt{|V|}\rfloor$, and multiple equivalent proxies.

We next first justify the necessity of introducing the size restriction for proxies. Otherwise, DRAs are simply CCs, and are mostly useless.

Proposition 1: *Without the size restriction, any node u in a graph G is a maximal proxy, and its* DRA *$G[A_u^+]$ is exactly the connected component (*CC*) to which u belongs.* □

We then show that proxies and their represented subgraphs (DRAs) have a strong relationship, which justifies that proxies and DRAs are *well defined*.

Proposition 2: *Any proxy in a graph has a unique* DRA. □

Proposition 3: *Given any two distinct proxies u and u',*
(1) if $u \in A_{u'}^+$, then $A_u^+ \subseteq A_{u'}^+$,
(2) if $u' \in A_u^+$, then $A_{u'}^+ \subseteq A_u^+$, and
(3) $A_u^+ \cap A_{u'}^+ = \emptyset$, otherwise. □

By Proposition 3, it is easy to have the following, which says when maximal proxies are concerned, there exists a unique set of non-overlapping DRAs.

Corollary 4: *Given any two maximal proxies u and u', then either $A_u^+ = A_{u'}^+$ or $A_u^+ \cap A_{u'}^+ = \emptyset$ holds.* □

Remark. Trivial proxies only represent themselves, and, hence, we are only interested in non-trivial maximal proxies (or simply called proxies) in the sequel.

3.2 Properties of Proxies and DRAs

We next give an analysis of proxies and DRAs, and show that they indeed hold good properties for answering shortest path and distance queries.

Indeed, the size restriction guarantees that the shortest distance computation within a DRA can be evaluated efficiently, as shown below.

Proposition 5: *Given any two nodes v, v' in the* DRA *$G[A_u^+]$ of proxy u in graph G,*
(1) the shortest path in $G[A_u^+]$ is exactly the one in the entire graph G, and
(2) it can be computed in linear time in the size of G. □

By Proposition 5, it is easy to derive the following.

Corollary 6: *Given any two nodes v, v' in the* DRA *$G[A_u^+]$ of proxy u in graph G,*
(1) the shortest distance $dist(v, v')$ in $G[A_u^+]$ is exactly the one in the entire graph G, and
(2) it can be computed in linear time in the size of G. □

Proposition 7: *Given two nodes v and u with two distinct proxies x and y, respectively, in graph G, the shortest path from v to u is $path(v, x)/path(x, y)/path(y, u)$.* □

Here $path(v, x)/path(x, y)/path(y, u)$ is a path by concatenating paths $path(v, x)$, $path(x, y)$ and $path(y, u)$. By Proposition 7, it is easy to derive the following result.

Corollary 8: *Given two nodes v and u with two distinct proxies x and y, respectively, in graph G, the shortest distance $dist(v, u)$ = $dist(v, x) + dist(x, y) + dist(y, u)$.* □

Propositions 5, 7 and Corollaries 6, 8 guarantee that the shortest paths and distances between the nodes in the DRAs of two distinct proxies can be answered in a correct and efficient way.

4 Computing Routing Proxies and DRAs

From Section 3.2, it is easy to see that the remaining challenge is to discover DRAs and their maximal proxies on large graphs. In this section, we first present a notion of BC-Sketch graphs, based on which we then propose a linear-time algorithm for computing DRAs and their maximal proxies. This makes our solution a light weight approach to reducing graph sizes and to speeding-up shortest path and

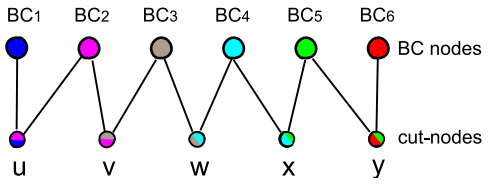


Figure 3. BC-SKETCH graph $\mathbb{G}_1$ of graph G_1

distance queries on large graphs. Without loss of generality, we consider connected graphs only.

The main result here is stated as follows.

Theorem 9: *Finding all* DRAs, *each associated with one maximal proxy, in a graph can be done in linear time.* □

We shall prove this by providing a linear time algorithm that computes DRAs and maximal proxies.

4.1 Connections with cut-nodes and BCCs

We first show that there are close connections between proxies and cut-nodes and BCCs.

Proposition 10: *Any proxy in a* CC $H(V_s, E_s)$ *of graph* $G(V, E)$ *with* $|V_s| > c \cdot \lfloor\sqrt{|V|}\rfloor$ *must be a cut-node of graph* G. □

Proposition 11: *Any node in a* BCC *with size larger than* $c \cdot \lfloor\sqrt{|V|}\rfloor$ *of graph* $G(V, E)$ *is a trivial proxy.* □

This motivates us to identify (non-trivial maximal) proxies by utilizing the cut-nodes and BCCs, which will be seen immediately. Further, as we are interested in non-trivial proxies only, those large BCCs could be simply ignored without any side effects.

We then present the notion of BC-SKETCH graphs, which are defined in terms of cut-nodes and BCCs, and are a key data structure employed by our algorithm.

A BC-SKETCH graph $\mathbb{G}(\mathbb{V}, \mathbb{E}, \omega)$ of a graph $G(V, E)$ is a bipartite graph, in which (1) $\mathbb{V} = \mathbb{V}_c \cup \mathbb{V}_{bc}$ such that $\mathbb{V}_c$ is the set of cut-nodes in G, and $\mathbb{V}_{bc}$ is the set of BCCs in G; (2) for each cut-node $v \in \mathbb{V}_c$ and each BCC $y_b \in \mathbb{V}_{bc}$, there exists an edge $(v, y_b) \in \mathbb{E}$ iff v is a cut-node of BCC y_b; and (3) ω is a weight function such that for each node $y_b \in \mathbb{V}_{bc}$, $\omega(y_b)$ is the number of nodes of G in BCC y_b.

Example 3: Consider graph G_1 in Fig. 1(1), and the corresponding BCCs of G_1 in Fig. 1(2).

The BC-SKETCH graph $\mathbb{G}_1(\mathbb{V}, \mathbb{E}, \omega)$ of graph G_1 is shown in Fig. 3, in which $\omega(BC_1) = 4$, $\omega(BC_2) = \omega(BC_3) = \omega(BC_4) = \omega(BC_6) = 2$, and $\omega(BC_5) = 5$. □

Note that BC-SKETCH graphs are an extension of block graphs [9], by further providing the weight function ω for the BCCs nodes. One may already notice that there are no cycles in the BC-SKETCH graph $\mathbb{G}_3$ in Fig. 3. This is not a coincidence, as shown below.

Proposition 12: BC-SKETCH *graphs have no cycles, which implies that they are simply trees.* □

Proposition 12 indicates that the good properties of trees can be employed for computing DRAs and maximal proxies.

Algorithm computeDRAs
Input: Graph $G(V, E)$ and constant c.
Output: The DRAs with their maximal proxies.
1. Find all cut-nodes $\mathbb{V}_c$ and BCC nodes $\mathbb{V}_{bc}$ of G;
2. Build the BC-SKETCH graph $\mathbb{G}(\mathbb{V}, \mathbb{E}, \omega)$ with $\mathbb{V} = \mathbb{V}_c \cup \mathbb{V}_{bc}$;
3. Identify and return DRAs with their maximal proxies of G.

Procedure extractDRAs
Input: BC-SKETCH graph $\mathbb{G}(\mathbb{V}, \mathbb{E}, \omega)$ of graph G and constant c.
Output: The DRAs and their maximal proxies of G.
1. **let** F be the set of cut-nodes with single non-leaf neighbors in $\mathbb{G}$; /* a leaf node must be a BCC node */
2. **while** F is **not** empty **do**
3. pick a cut-node v in F;
   let X be the neighbors of v;
   /* note that there is one non-leaf node in X*/
4. **let** $\alpha := \sum_{y' \in X} \omega(y')$ - $|X|$ + 1;
5. **if** $\alpha \leq c \cdot \lfloor\sqrt{|V|}\rfloor$ **then**
6. merge all BCC nodes in X and v into one BCC node y_n;
7. **let** $\omega(y_n) := \alpha$;
8. Replace the non-leaf node in X with y_n;
9. Add to F the cut node neighbors of y_n with single non-leaf neighbors;
10. $F := F \setminus \{v\}$;
11. **let** F' be the set of cut-nodes in the updated $\mathbb{G}$ with leaf neighbors;
12. **for** each cut-node v in F' **do**
13. **let** X' be a set of leaf neighbors of v' such that
14. for each $y' \in X'$, $\omega(y') \leq c \cdot \lfloor\sqrt{|V|}\rfloor$;
15. **let** the $A^+_{v'}$ of proxy $v' := X'$;
    /* note that BCCs are separately maintained in $A^+_{v'}$*/
16. **return** all DRAs with their maximal proxies.

Figure 4. Computing DRAs and maximal proxies

4.2 Algorithms

We now present algorithm computeDRAs shown in Fig. 4, which takes as input graph G and constant c, and outputs the DRAs of G and their maximal proxies.

(1) Finding cut-nodes and BCCs. The algorithm starts with computing all cut-nodes and bi-connected components (line 1), by using the linear-time algorithm developed by John Hopcroft and Robert Tarjan [6], [17].

(2) Constructing BC-SKETCH graphs. After all the cut-nodes and BCCs are identified, the BC-SKETCH graph $\mathbb{G}(\mathbb{V}, \mathbb{E}, \omega)$ can be easily built (line 2). To see this can be done in linear time, the key observation is that the number $|\mathbb{E}|$ of edges in $\mathbb{G}$ is exactly $|\mathbb{V}| - 1$ since $\mathbb{G}$ is a tree by Proposition 12.

(3) Identifying DRAs and their maximal proxies. Finally, the algorithm identifies and returns the DRAs and their maximal proxies (line 3), using procedure extractDRAs in Fig. 4.

Procedure extractDRAs takes as input the BC-SKETCH graph $\mathbb{G}$ of graph G and constant c, and outputs the DRAs and their maximal proxies, by repeatedly merging BCCs with size less than $c \cdot \lfloor\sqrt{|V|}\rfloor$. More specifically, the procedure starts with the set F of cut-nodes with single non-leaf neighbors (line 1). It then recursively merges the neighboring BCC nodes of cut-nodes to generate new BCC nodes (lines 2-9). For a node $v \in F$ with neighbors X, if $\sum_{y' \in X} \omega(y')$ - $|X|$ + 1 $\leq c \cdot \lfloor\sqrt{|V|}\rfloor$, they can be merged into a new BCC node (lines 3-8). Intuitively, this says cut-node v is not a maximal proxy, and it is combined into

the DRAs of maximal proxies. Then the non-leaf neighbor is replaced by the new BCC node y_n (line 8), by which the merging processing is made possible. The cut nodes connected to y_n are further considered, and those with single non-leaf neighbors are added to F (line 9). Once a cut-node is considered, it is never considered again (line 10). After no merging can be made, we have found all maximal proxies, *i.e.,* all the cut-nodes in the updated BC-SKETCH graph. We then identify DRAs for these maximal proxies (lines 11-15). For any leaf neighbor y' of a cut-node v', if $\omega(y') \leq c \cdot \lfloor\sqrt{|V|}\rfloor$, then y' is an $A_{v'}$ of proxy v'. All these together are the $A^+_{v'}$ of proxy v' (lines 13-15). Finally, all DRAs with their maximal proxies are returned (line 16).

For checking whether a cut node has a single non-leaf neighbor, we maintain an array list D for each cut node such $D[v]$ indicates the number of non-leaf neighbors of node v. At line 10 of procedure extractDRAs, if y_u is a leaf node, for each cut node v connected to y_u, we decrease $D[v]$ by 1, and if the updated $D[v]$ is 1, we simply add v to F.

We now explain the algorithm with an example.

Example 4: Consider graph G_1 in Fig. 1(1) again. Here we let $c = 2$, and then $c \cdot \lfloor\sqrt{|V|}\rfloor = 6$.

Firstly, cut-nodes and BCCs are computed as shown in Fig. 1(2). Secondly, the BC-SKETCH graph $\mathbb{G}_1$ of G_1 is constructed as shown in Fig. 3. Then in the merging step, we merge BCC nodes BC_1, BC_2, BC_3 and cut-nodes u and v together; Similarly, we merge BCC nodes BC_5 and BC_6 and cut-node y together because they also satisfy the size constraint $\alpha \leq c \cdot \lfloor\sqrt{|V|}\rfloor$. After the merging step stops, the updated BC-SKETCH graph consists of three BCC nodes: BC'_1 = $\{u, v, BC_1, BC_2, BC_3\}$, BC_4, BC'_2 = $\{y, BC_5, BC_6\}$ and two cut-nodes: w and x. Finally, the DRAs and their maximal proxies are identified: proxy w with dra_1 = $BC'_1 \cup \{w, v, u\}$ and proxy x with dra_2 = $BC'_2 \cup \{x, y\}$, shown in Fig. 5. □

Correctness & Complexity. The correctness of algorithm computeDRAs can be readily verified as follows: (1) We only consider non-trivial maximal proxies; (2) Proposition 10 tells us that in order to find all proxies, we only need to consider cut-nodes. Due to the size constraint and Proposition 11, we can ignore the BCCs with size larger than $c \cdot \lfloor\sqrt{|V|}\rfloor$ and only consider the other BCCs (Line 5). Note that it is trivial for the case when a CC has a size equal or less than $c \cdot \lfloor\sqrt{|V|}\rfloor$, and we simply omit it; (3) In the BC-SKETCH graph, we can merge the neighbors of a cut-node together if their combination does not exceed the size restriction, and any shortest paths from a node inside the combination must pass through the cut-node connected to the new combination. And (4) observe that the set of nodes X' at line 15 is indeed maximal. By the analyses in Sections 3.1 and 4.1, we know that the X' at line 15 is the unique set of nodes (so is the unique DRA) with its non-trivial maximal v'.

The linear-time complexity of algorithm computeDRAs roots from Proposition 12. To show this, it suffices to show that procedure extractDRAs can be done in linear time. It is easy to see that each node in BC-SKETCH graph $\mathbb{G}(\mathbb{V}, \mathbb{E}, \omega)$ is visited at most twice in procedure extractDRAs without the checking of cut nodes with single non-leaf neighbors at line 10, whose running time is bounded by the total number of edges in $\mathbb{G}$. From these, we know that procedure extractDRAs runs in linear time $O(|\mathbb{V}| + |\mathbb{E}|)$. This also completes the proof of Theorem 9.

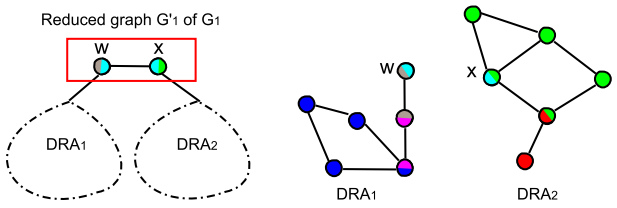


Figure 5. Example query answering

5 Query Answering with Routing Proxies

In this section, we show how to answer shortest path and distance queries using routing proxies.

Based on the previous analyses, we present a framework for speeding-up shortest path and distance query answering, which consists of two modules: *preprocessing* and *query answering*. We next introduce the details of the framework.

1. Preprocessing. Given graph $G(V, E)$, the preprocessing module executes the following.

(1) It first computes all DRAs and their maximal proxies with algorithm computeDRAs (as discussed in Section 4).

(2) It then pre-computes and stores all the shortest paths and distances between any node in a DRA and its proxy.

To support shortest distance queries, for each node in a DRA, we store its proxy u, its distance to u and the component of A^+_u to which it belongs, and, moreover, to support shortest path queries, we further keep the shortest paths from proxy u to all nodes in the DRA.

(3) It finally computes the reduced subgraph G' by removing all DRAs, but keeping their proxies, from graph G.

2. Query answering. Given two nodes s and t in graph $G(V, E)$ and the pre-computed information, the query answering module executes the following.

(1) When nodes s and t belong to the same DRA $G[A^+_u]$ with proxy u such that $A^+_u = A^1_u \cup \ldots A^h_u$.

If s and t further fall into the same component A^i_u ($i \in [1, h]$), it invokes the Dijkstra's algorithm on the subgraph $G[A^i_u]$ to compute the shortest path and distance between s and t. Otherwise, it simply returns $path(s, u)/path(u, t)$ or $dist(s, u) + dist(u, t)$ in constant time.

(2) When s and t belong to two DRAs $G[A^+_{u_s}]$ and $G[A^+_{u_t}]$ with proxies u_s and u_t, respectively.

By the analyses in Section 3.2, we know that $path(s, t)$ = $path(s, u_s)/path(u_s, u_t)/ path(u_t, t)$, in which $path(s, u_s)$ and $path(u_t, t)$ are already known. Hence, it simply invokes an algorithm (*e.g.,* bidirectional Dijkstra [24], ARCFLAG [27], CH [14], TNR [3], AH [41]) on the reduced graph G' for computing $path(u_s, u_t)$.

Similarly, the shortest distance $dist(s, t)$ = $dist(s, u_s)$ + $dist(u_s, u_t)$ + $dist(u_t, t)$ can be computed.

We next illustrate how shortest path and distance queries are processed with an example below.

Example 5: Consider graph G_1 and its BCCs in Fig. 1(1), in which the DRAs and their maximal proxies are computed by algorithm computeDRAs, *i.e.,* dra_1 = $\{u, v, BC_1, BC_2, BC_3\}$

Table 1
Real-life Datasets

Graphs G	Regions	# of Nodes	# of Edges
DBLP14	Subgraph of DBLP	141,359	246,462
DBLP	DBLP	317,080	1,049,866
CO	Colorado	435,666	521,200
FL	Florida	1,070,376	1,343,951
CA	California & Nevada	1,890,815	2,315,222
E-US	Eastern US	3,598,623	4,354,029
W-US	Western US	6,262,104	7,559,642
C-US	Central US	14,081,816	16,933,413
US	Entire US	23,947,347	28,854,312

with its proxy w and $dra_2 = \{y, BC_5, BC_6\}$ with its proxy x, as shown in Fig. 5. Note that here both A_w^+ and A_x^+ have single components. Moreover, the reduced graph G_1' of G_1 is the subgraph with nodes w and x only.

(1) Consider nodes s in dra_1 and t in dra_2, we first compute $dist(w, x)$ or $path(w, x)$ on G_1', and then let $dist(s, t) = dist(s, w) + dist(w, x) + dist(x, t)$ or $path(s, t) = path(s, w)/path(w, x)/path(x, t)$. Note that here $dist(s, w)$, $dist(x, t)$, $path(s, w)$ and $path(x, t)$ have all been computed in the precessing stage.

(2) If nodes s and t are both in dra_1 or dra_2, we directly compute their shortest path or distance on dra_1 or dra_2, as they have single components. □

Remarks. (1) As shown above, we need $O(d)$ extra space to store the routing information to compute shortest paths and distances, where d is the total number of nodes in all DRAs.

(2) It is easy to see that the main computation cost is reduced from graph G to its reduced graph G'. As shown in the experimental study (Section 6), on average about 1/3 nodes of sparse graphs are captured by non-trivial proxies and their DRAs, *i.e.*, d is about $|V|/3$. That is, the reduced graph G' is about 2/3 size of the original graph G, and hence our data reduction technique could reduce graph sizes and speed up shortest path and distance computations.

6 Experimental Study

Using real-life road networks and social graphs, we next conduct an extensive experimental study to evaluate: (1) the performance of computing routing proxies, and (2) how routing proxies speed up shortest path and distance queries, by comparing the efficiency and/or space overhead of (bidirectional) Dijkstra [24], ARCFLAG [27], TNR [2] and AH [41] with their counterparts using proxies (Proxy+Dijkstra, Proxy+ARCFLAG, Proxy+TNR and Proxy+AH) with respect to graph queries and graph sizes.

6.1 Experimental Settings

We first introduce the settings of our experimental study.

Real-life graphs. We use two types of datasets, and the details of all datasets are reported in Table 1.

(1) The first type of datasets is *co-authorship networks*. We extracted a co-authorship graph from DBLP [22], where each node in the graph represents an author and two authors are connected if they have co-published papers. The edge weight is computed by a revised Adamic/Adar similarity function: $w(u, v) = \frac{1}{\sum_{z \in \Gamma(u) \cap \Gamma(v) \cup \{u, v\}} \frac{1}{\log |\Gamma(z)|}}$, where $\Gamma(u)$ and $\Gamma(v)$ are the sets of neighbors of nodes u and v, respectively. The weight of the edge (u, v) represents the closeness between u and v and a smaller weight means the two authors are closer. TNR is designed for road networks and it is very inefficient for TNR to preprocess dense graphs such as DBLP (it took more than 1 week to finish the preprocessing). To guarantee that we can evaluate the improvement of TNR with proxies on general graphs, we remove all nodes whose degrees are higher than 14, and choose the largest connected component in the remaining graph, referred to as DBLP14.

(2) The second type of datasets is *road networks*. We chose seven standard road network datasets of various sizes from the Ninth DIMACS Implementation Challenge [11] (available at http://www.dis.uniroma1.it/challenge9/download.shtml). Each road network is released as an undirected graph representing a part of the road network in the United States, where each edge weight is the distance (an integer) required to travel between the two endpoints of the edge.

Shortest path and distance queries. Our queries were generated as follows. (1) On each road or co-authorship network, we first randomly choose a node u, and run Dijkstra's algorithm from u to find the node s that is farthest from u. Then we run Dijkstra's algorithm again from s to find the node t that is farthest from s. Let ℓ be the distance $dist(s, t)$ from s to t. (2) We then randomly chose ten thousand node pairs from the road network to compose $Q_i (i \in [1, 7])$, such that the distances of all node pairs in Q_i are in $[2^{i-9} \cdot \ell, 2^{i-8} \cdot \ell)$. For each query set Q_i $(i \in [1, 7])$, we report the average running time of 10,000 queries in the set.

Algorithms. We implemented algorithms bidirectional-Dijkstra [24], ARCFLAG [27], TNR [2] and AH [41].

For ARCFLAG, it first needs to partition graphs to precompute information on whether an edge is useful for a shortest path search. Any possible partition methods [8], [18], [19], [39] can be used here. Since we have both road networks and co-authorship networks, we adopted the latest version 5.0.2 of METIS [26], implemented with ANSI C, because it is open source and performs quite well in practice.

For TNR, since we do not have coordinates information in the co-authorship network, we implemented the CH-based TNR [2] that does not require the geometry information.

We obtained the C++ implementation of AH from [41] and adopted its default settings.

Implementations. All algorithms were implemented with C++, and all experiments were run on a PC with an Intel Xeon(R) X5650 CPU@2.67GHz and 24GB of memory.

6.2 Performance of Computing Routing Proxies

Using the datasets in Table 1, we first conduct experiments to evaluate the performance of computing proxies, *i.e.*, how many nodes can be represented by proxies, how much extra space we need to store the routing information for the proxies, and how much time we need to find all proxies. More specifically, we evaluate (1) the number of non-trivial proxies, (2) the number and percentage of the nodes represented by the proxies (excluding the proxies themselves from DRAs), (3) the extra space cost of using proxies, (4) the space overhead of storing the original graph

Table 2
Effectiveness of proxies and DRAs

Graphs G	Proxies (# : %)	Nodes in DRAs (# : %)	Extra space (MB)	G Space (MB)	G' Space (MB)	Time (Sec.)
DBLP14	(14, 090 : 9.8)	(102,085 : 72.2)	1.56	4.30	1.36	1.5
DBLP	(31,475 : 9.9)	(105,671 : 33.3)	1.61	17.22	14.29	5.7
CO	(56,277 : 12.9)	(156,329 : 35.9)	2.38	9.61	6.64	3.5
FL	(140,382 : 13.1)	(378,804 : 35.4)	5.78	24.59	17.12	9.9
CA	(273,191 : 14.4)	(623,811 : 33.0)	9.52	42.55	30.66	21.1
E-US	(546,481 : 15.2)	(1,228,876 : 34.1)	18.75	80.17	56.48	52.5
W-US	(869,907 : 13.9)	(2,116,382 : 33.8)	32.29	139.23	98.68	111.9
C-US	(2,034,358 : 14.4)	(4,583,413 : 32.5)	69.94	312.09	225.28	435.8
US	(3,452,222 : 14.4)	(7,927,453 : 33.1)	120.96	531.63	380.59	1,925.4

Table 3
Effectiveness *w.r.t.* average degrees

Distribution	Properties	Average degrees				
		2	3	4	5	6
Uniform	Proxies (#: %)	(88,649: 14.7)	(97,202: 16.2)	(75,822: 12.6)	(60,937: 10.1)	(50,829: 8.5)
	Nodes in DRAs (#: %)	(267,015: 44.5)	(129,301: 21.5)	(89,253: 14.9)	(68,232: 11.4)	(55,314: 9.2)
Power Law	Proxies (#: %)	(90,245: 9.0)	(124,247: 12.4)	(128,126: 12.8)	(128,020: 12.8)	(126,836: 12.7)
	Nodes in DRAs (#: %)	(504,401: 50.4)	(575,202: 57.5)	(580,392: 58.0)	(576,610: 57.7)	(569,697: 57.0)
Distribution	**Properties**	**Average degrees**				
		20	**30**	**40**	**50**	**60**
Power Law	Proxies (#: %)	(102,019: 10.2)	(96,300: 9.6)	(95,823: 9.6)	(95,283: 9.5)	(95,735: 9.6)
	Nodes in DRAs (#: %)	(480,356: 48.0)	(436,919: 43.7)	(415,272: 41.5)	(391,605: 39.2)	(375,196: 37.5)

G, (5) the space overhead of storing the reduced graph G', and (6) the efficiency of algorithm computeDRAs for computing proxies and their DRAs. We fixed the constant $c = 2$ for computing proxies on all graphs. When computing the space overhead for the original graphs and the reduced graphs, we assume that all graphs are stored as adjacency lists. The experimental results are reported in Table 2.

1. Effectiveness evaluation. For the DBLP graph, about $1/10$ nodes are non-trivial proxies, and about $1/3$ nodes are captured by proxies in the graph, which means basically the reduced graph is only about $2/3$ of the original input graph. For the DBLP14 graph, about $1/10$ nodes are non-trivial proxies, and about $2/3$ nodes are captured by the DRAs of proxies in the graph, which means basically the reduced graph is only about $1/3$ of the original input graph.

For all the road graphs, about $1/7$ nodes are non-trivial proxies, and about $1/3$ nodes are captured by the DRAs of these proxies, which means basically the reduced graph is only about $2/3$ of the original input graph.

Moreover, although the sizes of DRAs are restricted within $2 \cdot \lfloor\sqrt{|V|}\rfloor$, DRAs are typically small such that each proxy represents 2 or 3 other nodes on average.

2. Efficiency evaluation. In the DBLP graph, proxies can be found in about 6 seconds. In the DBLP14 graph, it only takes 1.5 seconds to find all DRAs with their proxies.

In all road networks, proxies can be found efficiently. Algorithm computeDRAs also scales well, and it can be done in less than half an hour for the largest graph US with 2.4×10^7 nodes and 5.7×10^7 edges.

3. Space evaluation. To support shortest distance queries, for each node in a DRA, we store its proxy u, its distance to u and the component of A_u^+ to which it belongs, and, moreover, to support shortest path queries, we further keep the shortest paths from proxy u to all nodes in the DRA. Assume that distances are stored as a 4-byte integer. Each node is represented as a 4-byte integer. Then the extra space that we need for shortest path and distance queries is $16 \cdot |V_{dra}|$, where V_{dra} is the set of nodes in all DRAs.

In the DBLP and DBLP14 graphs, it only takes about 1.6 MB extra space to store the routing information for all proxies, since there are similar number of nodes in the DRAs of both datasets. Observe that by using proxies to represent all nodes in DRAs, the reduced graph has a smaller size than the original graph, especially for the DBLP14 graph. In the DBLP14 graph, 72.2% of its nodes are captured by proxies, which leads to a much smaller reduced graph.

In all road networks, it incurs small extra space overhead to store the routing information for proxies. Only about 100 MB extra space is taken for the largest graph US. The use of proxies also results in a smaller reduced graph. As shown in Table 2, the size of the reduced graph is about 70% of the original graph on average.

4. Effectiveness *w.r.t.* average degrees. In this set of experiments, we further study the impact of density, measured by average degrees, on the effectiveness of proxies and DRAs.

We adopted the graph-tool library [1] to produce random graphs. It is controlled by two parameters: the number n of nodes, and a sample function that decides the degree distribution of graphs. Here we fixed $n = 10^6$, and adopted two degree distributions: (1) uniform distribution and (2) power law distribution of scale-free networks (the fraction $P(k)$ of nodes having degree k in the network goes for large values of k, *i.e.*, $P(k) \sim k^{\delta}$, where δ is a parameter). When we generate a random graph with average degree d using the uniform distribution, we randomly choose an integer from range $[1, 2 \cdot d - 1]$ as the degree of a node. When we use the power law distribution, we set the maximum degree of graphs as $5,000$, which is also the limit of the number of friends for Facebook users (www.facebook.com), and we vary δ to generate a graph with the expected average degree d. The results are shown in Table 3.

We can find the following. (1) Our algorithm performs well on sparse graphs (with average degree no larger than 3)

generated with the uniform distribution. Specifically, about 30% of the nodes in these graphs are captured by DRAs. (2) Our algorithm performs much better on graphs having the power law distribution. We can see that more than 30% of the nodes are captured by DRAs even when the average degree is 60. (3) The performance of our algorithm is sensitive to the density and degree distribution of graphs. (4) Given a degree distribution, DRAs tend to capture less nodes when the average degree is higher. This is because (1) there are less cut-nodes in graphs with a higher density, and thus less proxies can be found, and (2) for a proxy u, the set of nodes A_u^i represented by u is more likely to exceed the size restriction. Therefore, there are less nodes in its DRA.

From these tests, we can find that using proxies is a light-weight optimization technique, which scales well to large networks, and incurs appropriate space overhead.

6.3 Benefits of Using Routing Proxies

Our proxies can be used as a preprocessing step before we apply other existing approaches. We next conduct experiments to show the comparison results of an existing approach (*i.e.,* Dijkstra, ARCFLAG, TNR and AH) with its counterpart using proxies, using the datasets in Table 1.

We fixed the constant $c = 2$ for computing proxies on all graphs. Note that AH requires the coordinate information to answer shortest path or distance queries, not available in both DBLP and DBLP14. Thus, we only report results on road graphs for AH and its counterpart with proxies.

1. Effectiveness *w.r.t.* graph queries. In this set of experiments, we evaluated the efficiency of shortest path and distance queries with respect to the graph queries.

For the ARCFLAG and Proxy+ARCFLAG methods, the graph and the reduced graph are partitioned into fragments such that each fragment has at most $2 \cdot \lfloor \sqrt{|V|} \rfloor$ nodes in order to add labels to edges. We used METIS to partition graphs with the balance factor fixed to 1.003.

For the TNR and Proxy+TNR methods, we always select 10,000 transit nodes, as suggested in [2]. Note that the space cost of the CH-based TNR is very high. Consequently, we could not run TNR on C-US as it ran out of memory. However, we can successfully run Proxy+TNR on C-US. We report this to show that proxies serve as a data reduction technique and benefit existing methods in terms of space cost as well. TNR invokes Contract Hierarchies (CH) [14] to preprocess the graph and CH runs very slowly on DBLP (it took more than 7 days to run CH on DBLP). Thus we report the results on DBLP14 and all the road networks.

The results of distance queries and path queries are reported in Tables 4 and 8 (comparison between Dijkstra and Proxy+Dijkstra), Tables 5 and 9 (comparison between ARCFLAG and Proxy+ARCFLAG), Tables 6 and 10 (comparison between TNR and Proxy+TNR), and Tables 7 and 11 (comparison between AH and Proxy+AH), respectively.

In the co-authorship network DBLP14, the results show that with the help of proxies, Proxy+Dijkstra, Proxy+ARCFLAG and Proxy+TNR all achieve a better performance than their counterparts without proxies, respectively. On average, the time cost of Proxy+ARCFLAG, Proxy+Dijkstra and Proxy+TNR is about 96%, 51% and 51% of their counterparts without proxies for distance queries, and 98%, 49% and 76% of their counterparts without proxies for shortest path queries, respectively. More specifically, we can see that (1) proxies have a better speed-up effect on bidirectional Dijkstra and TNR than ARCFLAG, and (2) for ARCFLAG, proxies have a better speed-up effect when the two query nodes are far from each other, which is different from the observation on road networks. To explain the second observation, we need to notice that about 2/3 nodes are captured by proxies in DBLP14. Thus two close nodes are more likely to fall into the same DRAs. Since there are no speed-up techniques used inside single DRAs, the search space saved by proxies is less than using ARCFLAG alone.

In the road networks, the results show that with the help of proxies, Proxy+Dijkstra, Proxy+ARCFLAG and Proxy+AH can achieve a better performance than their counterparts without proxies, respectively. On average, the time cost of Proxy+ARCFLAG, Proxy+Dijkstra and Proxy+AH is about 80%, 68% and 99% of their counterparts without proxies for distance queries, and 82%, 67% and 99% of their counterparts without proxies for shortest path queries, respectively. More specifically, we can see that (1) proxies have a better speed-up effect on bidirectional Dijkstra than ARCFLAG, (2) for ARCFLAG, proxies have a better speed-up effect when the two query nodes are close to each other, (3) different from ARCFLAG, Proxy+Dijkstra has a better performance when the query nodes are far from each other, and (4) though AH is one of the state-of-art methods for shortest path and distance queries, proxies still improve its efficiency about 1%. To explain these observations, we need to think how much search space is saved by proxies. Since ARCFLAG has already used flags on edges to reduce the search space, the proportion of search space saved by proxies is smaller than bidirectional Dijkstra. That explains why proxies have a better speed-up effect on bidirectional Dijkstra. For ARCFLAG, two close nodes are more likely to fall into the same partition. In this case, the effect of flags on edges is less useful and the search space saved by proxies takes a large proportion, which explains the second observation. For bidirectional Dijkstra, proxies can save more search space when the query nodes are far from each other. As a state-of-art approach for shortest path and distance queries, the search space of AH is much smaller than Dijkstra and ARCFLAG. However, the search space of AH is still reduced, and AH still benefits from proxies in terms of efficiency. This explains the fourth observation.

Proxy+TNR achieves a comparable performance to its counterpart without proxies. This is because for TNR, a heuristic method is used to select transit nodes, based on the structure of the graph. And selection of transit nodes has a significant effect on the performance of TNR. Since we reduce the input graph by using proxies, the reduced graph has a different topology structure. Thus a different set of transit nodes will be selected. So it is hard to guarantee that Proxy+TNR outperforms TNR. We should also notice that TNR cannot run on C-US while Proxy+TNR can. To explain this, we first recall that for TNR, we have to store the access nodes and distances for each node. For the original input of C-US, there are too many nodes and it runs out of memory. By using proxies, about 1/3 nodes are captured by proxies and we only need to run TNR on 2/3 of the input graph, which is more practical. We will evaluate the space overhead

Table 4
Varying graph queries for shortest distances: Dijkstra vs. Proxy+Dijkstra

Graphs	# of nodes	Methods	Query time ($\times 10^{-6}$ sec.)						
			Q1	Q2	Q3	Q4	Q5	Q6	Q7
DBLP14	141,359	Dijkstra	2	3	5	11	34	287	13,935
		Proxy+Dijkstra	2	3	4	6	12	84	3,332
CO	435,666	Dijkstra	113	395	1,427	4,669	10,963	26,289	62,194
		Proxy+Dijkstra	105	368	1,296	4,240	10,274	24,962	59,816
FL	1,070,376	Dijkstra	444	1,648	5,701	16,684	45,184	137,532	360,603
		Proxy+Dijkstra	323	1,173	3,800	10,719	28,915	86,020	226,303
CA	1,890,815	Dijkstra	873	3,066	10,422	32,365	89,983	237,625	543,870
		Proxy+Dijkstra	640	2,226	7,439	22,699	61,405	159,870	359,532
E-US	3,598,623	Dijkstra	869	3,283	12,080	44,275	155,196	519,541	1,681,470
		Proxy+Dijkstra	672	2,503	9,051	31,749	107,962	354,352	956,205
W-US	6,262,104	Dijkstra	2,219	9,255	20,829	57,122	163,115	473,679	1,514,990
		Proxy+Dijkstra	1,587	5,059	14,104	37,537	105,377	305,627	961,300
C-US	14,081,816	Dijkstra	2,419	8,119	28,468	102,945	408,656	1,639,260	5,868,220
		Proxy+Dijkstra	1,508	4,876	16,649	59,777	244,552	978,000	3,553,810

Table 5
Varying graph queries for shortest distances: ARCFLAG vs. Proxy+ARCFLAG

Graphs	# of nodes	Methods	Query time ($\times 10^{-6}$ sec.)						
			Q1	Q2	Q3	Q4	Q5	Q6	Q7
DBLP14	141,359	ARCFLAG	70	72	73	75	80	89	134
		Proxy+ARCFLAG	70	74	75	73	75	80	123
CO	317,080	ARCFLAG	120	246	437	772	1,158	1,600	2,274
		Proxy+ARCFLAG	84	168	317	582	943	1,475	2,008
FL	435,666	ARCFLAG	339	692	1,170	1,739	2,431	3,132	4,028
		Proxy+ARCFLAG	233	491	890	1,413	2,076	2,599	3,360
CA	1,070,376	ARCFLAG	540	960	1,566	2,394	3,400	4,840	6,391
		Proxy+ARCFLAG	386	737	1,268	1,955	2,855	4,332	5,566
E-US	1,890,815	ARCFLAG	613	1,258	2,212	3,485	5,383	7,970	11,007
		Proxy+ARCFLAG	429	923	1,699	2,796	4,468	6,968	10,090
W-US	6,262,104	ARCFLAG	1,036	1,906	3,089	4,435	6,478	8,722	11,602
		Proxy+ARCFLAG	737	1,380	2,373	3,612	5,539	7,663	10,651
C-US	14,081,816	ARCFLAG	1,474	2,943	5,006	7,905	12,439	19,316	28,959
		Proxy+ARCFLAG	983	2,065	3,740	6,283	10,402	16,495	26,074

Table 6
Varying graph queries for shortest distances: TNR vs. Proxy+TNR

Graphs	# of nodes	Methods	Query time ($\times 10^{-6}$ sec.)						
			Q1	Q2	Q3	Q4	Q5	Q6	Q7
DBLP14	141,359	TNR	2.1	2.4	3.5	4.0	3.6	2.2	0.7
		Proxy+TNR	1.3	1.4	1.3	1.3	1.1	0.6	0.4
CO	435,666	TNR	4.5	3.7	2.9	2.0	1.4	1.3	1.2
		Proxy+TNR	4.2	3.8	3.4	2.7	1.9	1.5	1.5
FL	1,070,376	TNR	23.8	38.7	58.2	88.9	124.8	167.0	235.0
		Proxy+TNR	22.9	36.3	57.0	91.2	150.8	195.3	251.5
CA	1,890,815	TNR	38.0	66.4	116.1	172.1	227.0	340.7	421.6
		Proxy+TNR	39.5	67.5	107.4	159.7	221.0	286.2	828.8
E-US	3,598,623	TNR	36.1	50.1	70.1	75.3	42.6	27.5	25.5
		Proxy+TNR	35.7	52.4	72.8	80.3	57.8	32.2	29.8
W-US	6,262,104	TNR	49.5	63.5	77.3	86.1	63.0	28.9	25.2
		Proxy+TNR	47.7	57.7	74.0	89.5	77.2	30.6	26.5
C-US	14,081,816	TNR	NA	NA	NA	NA	NA	NA	NA
		Proxy+TNR	67.0	137.5	300.0	600.0	667.0	280.0	135.0

Table 7
Varying graph queries for shortest distances: AH vs. Proxy+AH

Graphs	# of nodes	Methods	Query time ($\times 10^{-6}$ sec.)						
			Q1	Q2	Q3	Q4	Q5	Q6	Q7
CO	435,666	AH	7.8	9.3	17.2	26.5	37.4	51.5	67.1
		Proxy+AH	7.8	10.9	17.2	28.1	37.5	53.0	59.3
FL	1,070,376	AH	10.9	15.6	23.4	29.7	40.5	56.2	59.3
		Proxy+AH	12.4	17.2	23.5	29.7	40.5	54.6	57.7
CA	1,890,815	AH	18.7	28.0	42.1	56.2	68.6	88.9	101.4
		Proxy+AH	18.7	28.1	42.1	54.6	67.1	85.8	99.8
E-US	3,598,623	AH	20.3	34.3	57.7	96.7	159.1	235.5	257.4
		Proxy+AH	21.9	34.3	56.2	93.6	152.8	232.4	258.9
W-US	6,262,104	AH	24.9	39.0	56.2	74.9	107.6	149.8	188.7
		Proxy+AH	26.6	37.5	53.0	73.3	104.5	146.6	185.6
C-US	14,081,816	AH	39.0	70.2	135.7	244.9	402.5	603.7	762.0
		Proxy+AH	37.3	70.2	129.5	235.5	385.3	580.3	769.1

Table 8
Varying graph queries for shortest paths: Dijkstra vs. Proxy+Dijkstra

Graphs	# of nodes	Methods	Query time ($\times 10^{-6}$ sec.)						
			Q1	Q2	Q3	Q4	Q5	Q6	Q7
DBLP14	141,359	Dijkstra	2	3	5	52	34	287	13,935
		Proxy+Dijkstra	2	3	4	6	13	94	3,732
CO	435,666	Dijkstra	153	545	1,963	6,557	16,138	40,279	99,073
		Proxy+Dijkstra	122	424	1,553	5,040	11,918	28,084	65,802
FL	1,070,376	Dijkstra	473	1,748	6,094	17,436	46,977	140,428	368,370
		Proxy+Dijkstra	332	1,185	3,832	10,837	29,155	87,459	228,535
CA	1,890,815	Dijkstra	886	3,141	10,657	33,377	92,742	257,752	550,480
		Proxy+Dijkstra	654	2,244	7,478	22,764	62,670	163,236	365,321
E-US	3,598,623	Dijkstra	912	3,455	13,033	47,028	163,397	542,802	1,681,470
		Proxy+Dijkstra	706	2,647	9,507	33,229	113,114	379,599	956,205
W-US	6,262,104	Dijkstra	886	3,141	10,657	33,377	92,742	257,752	550,480
		Proxy+Dijkstra	654	2,244	7,748	22,764	62,670	163,236	365,321
C-US	14,081,816	Dijkstra	2,419	8,014	28,083	101,077	402,928	1,580,830	5,868,220
		Proxy+Dijkstra	1,508	5,004	17,039	61,512	247,490	988,994	3,553,810

Table 9
Varying graph queries for shortest paths: ARCFLAG vs. Proxy+ARCFLAG

Graphs	# of nodes	Methods	Query time ($\times 10^{-6}$ sec.)						
			Q1	Q2	Q3	Q4	Q5	Q6	Q7
DBLP14	141,359	ARCFLAG	64	64	75	75	80	87	134
		Proxy+ARCFLAG	70	72	71	74	75	80	123
CO	435,666	ARCFLAG	120	245	438	775	1,148	1,597	4,031
		Proxy+ARCFLAG	90	177	330	604	976	1,535	3,534
FL	1,070,376	ARCFLAG	336	689	1,166	1,738	2,425	3,146	4,028
		Proxy+ARCFLAG	241	502	908	1,444	2,138	2,751	3,360
CA	1,890,815	ARCFLAG	539	965	1,566	2,387	3,400	4,833	6,391
		Proxy+ARCFLAG	393	742	1,268	1,985	2,855	4,420	5,566
E-US	3,598,623	ARCFLAG	619	1,266	2,212	3,487	5,383	7,959	11,007
		Proxy+ARCFLAG	442	946	1,742	2,873	4,468	7,173	10,090
W-US	6,262,104	ARCFLAG	1,043	1,903	3,086	4,441	6,457	8,718	11,634
		Proxy+ARCFLAG	757	1,411	2,423	3,708	5,699	7,932	11,169
C-US	14,081,816	ARCFLAG	1,486	2,958	5,006	7,992	12,439	19,517	28,670
		Proxy+ARCFLAG	1,006	2,098	3,740	6,463	10,402	16,974	25,104

Table 10
Varying graph queries for shortest paths: TNR vs. Proxy+TNR

Graphs	# of nodes	Methods	Query time ($\times 10^{-6}$ sec.)						
			Q1	Q2	Q3	Q4	Q5	Q6	Q7
DBLP14	141,359	TNR	7.9	13.1	19.8	34.7	61.4	143.6	683.9
		Proxy+TNR	5.2	10.7	14.1	21.8	43.4	111.7	660.2
CO	435,666	TNR	23.3	35.3	57.4	91.8	139.7	202.6	276.0
		Proxy+TNR	21.0	33.0	52.4	84.0	124.0	184.2	379.2
FL	1,070,376	TNR	33.7	54.4	85.6	132.9	194.1	277.2	412.1
		Proxy+TNR	31.4	53.6	84.1	135.4	220.1	309.3	440.5
CA	1,890,815	TNR	50.9	86.7	147.4	227.4	315.8	495.8	667.7
		Proxy+TNR	51.5	89.5	142.3	219.3	317.4	452.8	1,091.6
E-US	3,598,623	TNR	53.4	88.5	154.0	252.5	394.7	654.1	1,024.2
		Proxy+TNR	54.5	89.4	154.8	253.6	423.7	2,569.1	4,640.1
W-US	6,262,104	TNR	68.9	106.3	168.9	267.0	470.0	766.4	1,492
		Proxy+TNR	73.5	110.7	171.3	269.9	698.7	1,110.0	1,924.8
C-US	14,081,816	TNR	NA	NA	NA	NA	NA	NA	NA
		Proxy+TNR	90.5	175.0	378.0	775.0	1,230.0	1,760.0	2,645.0

Table 11
Varying graph queries for shortest paths: AH vs. Proxy+AH

Graphs	# of nodes	Methods	Query time ($\times 10^{-6}$ sec.)						
			Q1	Q2	Q3	Q4	Q5	Q6	Q7
CO	435,666	AH	7.8	14.0	20.3	34.4	48.4	68.6	87.4
		Proxy+AH	7.8	14.1	21.8	32.7	48.4	68.6	84.3
FL	1,070,376	AH	15.6	23.4	34.3	48.4	65.5	90.5	110.8
		Proxy+AH	15.6	21.8	32.8	53.0	71.7	87.4	107.6
CA	1,890,815	AH	23.4	39.0	59.3	81.1	104.5	145.0	181.0
		Proxy+AH	23.4	37.4	57.7	78.0	99.8	140.4	176.3
E-US	3,598,623	AH	28.1	48.4	82.7	142.0	229.3	357.2	443.1
		Proxy+AH	28.1	46.8	79.6	134.2	221.5	347.8	433.6
W-US	6,262,104	AH	35.8	56.3	82.7	124.8	187.2	276.1	394.7
		Proxy+AH	35.9	54.6	81.1	121.7	182.5	269.9	386.9
C-US	14,081,816	AH	56.1	99.8	188.8	347.9	591.2	926.7	1,324.4
		Proxy+AH	53.0	95.2	179.4	333.9	567.9	892.3	1,310.4

Table 12
Comparison *w.r.t.* space overhead

Graphs	ARCFLAG (MB)		TNR (MB)		AH (MB)	
	with proxies	no proxies	with proxies	no proxies	with proxies	no proxies
DBLP14	4.9	4.3	386.5	389.2	NA	NA
DBLP	16.1	35.2	NA	NA	NA	NA
CO	9.3	13.6	417.1	430.2	58.1	72.2
FL	29.7	52.8	491.9	530.5	150.9	187.3
CA	55.1	108.8	701.9	806.4	277.9	338.1
E-US	134.9	287.8	1,842.9	2,382.7	525.5	642.0
W-US	277.3	643.9	2,996.4	4,154.1	889.6	1092.3
C-US	838.8	2,153.1	13,894.5	NA	2082.3	2512.3
US	NA	NA	NA	NA	3512.7	4266.9

saved by proxies in detail, to be seen shortly.

2. Effectiveness *w.r.t.* graph sizes. Since DBLP14 is a single dataset, we only compare the efficiency of shortest path and distance queries on road graphs using the same settings. We vary the number of nodes from 435,666 to 14,081,816. We observe the results in Tables 4 and 8 (comparison between Dijkstra and Proxy+Dijkstra), Tables 5 and 9 (comparison between ARCFLAG and Proxy+ARCFLAG), Tables 6 and 10 (comparison between TNR and Proxy+TNR), and Tables 7 and 11 (comparison between AH and Proxy+AH) for shortest distance and shortest path queries, respectively. For a given query, we can observe its corresponding row to see how the efficiency is affected by graph sizes.

The results tell us that (1) all algorithms scale well with respect to graph sizes, (2) for Proxy+Dijkstra, its time cost is 68% and 67% of its counterpart without proxies for shortest path and distance queries on average, respectively, (3) for Proxy+ARCFLAG, its time cost is 80% and 82% of its counterpart without proxies for shortest path and distance queries on average, respectively, (4) for Proxy+TNR, its time cost is comparable to TNR for shortest path and distance queries on average, respectively, (5) for Proxy+TNR, it is applicable to handle the larger dataset C-US while TNR cannot, and (6) for Proxy+AH, its time cost is 98% and 99% of AH for shortest path and distance queries on average, respectively. Further, (7) as the size of graphs increases, the running time of Dijkstra, ARCFLAG and AH increases in the same way as their counterparts with proxies. Finally, (8) for TNR, the selection of transit nodes is based on the graph structure and has a significant effect on the efficiency. By using proxies, the reduced graph has a different set of transit nodes and it is hard to guarantee that which set of transit nodes can get a better query efficiency. Thus, Proxy+TNR does not outperform TNR constantly as the other 3 approaches do.

3. Space overhead. In the last set of experiments, we evaluate the space overhead of ARCFLAG, TNR and AH compared with their counterparts with proxies, since all of them require preprocessing for query answering. We report (1) the size of the index generated by ARCFLAG, TNR and AH on the original input graphs, and (2) the sum of the size of the index generated by each of those methods on the reduced graph and the extra space it takes to store routing information for proxies. Note that for TNR and Proxy+TNR, no compression techniques are used. The results are reported in Table 12.

In the co-authorship network DBLP14, different from our expectation, ARCFLAG requires less space overhead than Proxy+ARCFLAG. This is because that the graph is relatively small and has small number of partitions. Thus, for each edge we only need to maintain an 8-byte flag in ARCFLAG. However, when using proxies, we need to maintain 12-byte routing information for each node. As we have shown before, about 70% of the nodes in DBLP14 fall into the DRAs, thus, ARCFLAG requires less space overhead than Proxy+ARCFLAG. However, in DBLP, since the graph is larger (we need to maintain a 17-byte flag for each edge), Proxy+ARCFLAG requires only about 14% of the space overhead of ARCFLAG. We can also observe that the space overhead of Proxy+TNR is about 98% of TNR.

In the road networks, we can make the following observations: (1) Proxy+ARCFLAG incurs less space overhead than ARCFLAG (from 38% to 68%), especially for large graphs (about 38% for CUS). (2) Proxy+TNR incurs less space overhead than TNR (from 72% to 92%). Recall that for TNR, we need to maintain a list of access nodes and a list of filter nodes for each node in the graph. A larger graph usually incurs a larger size of filter nodes. Thus, by using proxies to represent nodes in DRAs, we can save more space for larger graphs. (3) Proxy+AH is about 82% of its counterpart without proxies.

From the experimental results above, we can find that the existing solutions also benefit from using proxies in terms of space overhead.

Summary. From these experimental results, we find the following. (1) Proxies are a light-weight preprocessing technique, which can be computed efficiently and takes linear space to support shortest path and distance queries. (2) According to our experiments, in sparse graphs whose average degree is less than 4, about 1/3 nodes in the graph are captured by proxies, leaving the reduced graph about 2/3 of the input graph. In some special cases (like DBLP14), about 2/3 nodes in the graph are captured by proxies, leaving the reduced graph about only 1/3 of the input graph. (3) The performance of proxies and DRAs is sensitive to the density and degree distribution of graphs, and they perform well on graphs following the power law distribution. Meanwhile, for a given degree distribution, DRAs tend to capture less nodes when the average degree is higher. (4) Proxies and their DRAs benefit existing shortest path and distance algorithms in terms of time cost. They reduce 20%, 30% and 1% time for ARCFLAG, bidirectional Dijkstra and AH on road networks, respectively. They also have comparable time cost for TNR on road networks; They reduce 49%, 4% and 49% time for bidirectional Dijkstra, ARCFLAG, and TNR on the co-authorship network DBLP14, respectively. (5) Existing shortest path and distance algorithms also benefit from

using proxies in terms of space overhead. Since the original input graph is reduced by proxies, larger datasets may be handled when existing methods are combined with proxies. Proxy+TNR can handle the road network C-US while TNR cannot. Moreover, Proxy+ARCFLAG incurs less space overhead than ARCFLAG (from 38% to 68%), Proxy+TNR is about from 72% to 92% of its counterpart without proxies, and Proxy+AH is about 82% of its counterpart without proxies.

7 Related Work

Algorithms for shortest paths and distances have been extensively studied since 1950's, and fall into different categories in terms of different criteria:

- exact distances [2], [3], [4], [5], [7], [10], [12], [13], [14], [15], [16], [20], [24], [28], [28], [30], [32], [34], [36], [38], [41] and approximate distances [29], [31], [33], [35],
- memory-based [2], [3], [7], [10], [13], [14], [16], [20], [24], [28], [29], [30], [31], [32], [33], [34], [35], [36], [37], [38], [41] and disk-based algorithms [4], [5],
- for unweighted [2], [3], [7], [29], [33], [37] and weighted graphs [2], [3], [4], [5], [7], [10], [12], [13], [14], [15], [16], [20], [24], [28], [28], [30], [31], [32], [34], [35], [36], [38], [41], and
- for directed [2], [3], [7], [12], [15], [20], [28], [34], [41] and undirected graphs [2], [3], [4], [5], [7], [10], [13], [14], [16], [24], [28], [29], [30], [31], [32], [33], [35], [36], [37], [38].

In this work, we study the memory-based (exact) shortest path and shortest distance problem on weighted undirected large graphs. We next introduce those methods that fall into this category from two aspects: on general graphs and on road networks.

Approaches for general graphs. The classic solution for shortest path and distance queries is Dijkstra's algorithm [10]. It visits nodes in an ascending order of their distances from the source node, and all the nodes that are closer to the source node than the target node are visited. Thus, many techniques are proposed to reduce the search space. Among these, (1) bidirectional Dijkstra search [24] was proposed to search from both source and target nodes; (2) ALT [15] selects a set of nodes (reffered to as landmarks), and pre-computes the distances from each node to landmarks, which are used to prune unnecessary nodes during the search process; (3) An edge labeling method named ARCFLAG [27] cuts graphs into partitions to reduce the search space; (4) An indexing and query processing scheme named TEDI [37] decomposes a graph into a tree, and uses the tree index to process shortest path queries; And (5) a 2-hop labeling based exact algorithm was proposed in [7] to deal with large networks.

Approaches for road networks. A lot of work focuses on processing shortest path and distance queries on road networks. (1) Different from general graphs, the shortest paths on road networks are often spatially coherent. Path oracles have been proposed for spatial networks [32]. Transit Node Routing (TNR) [2] is a fast and exact distance oracle for road networks. Both of them utilize the property of spatial coherence, *i.e.,* spatial positions of both source and destination nodes and the shortest paths between them that facilitate the aggregation of source and destination nodes into groups sharing common nodes or edges on the shortest paths between them. (2) Road networks are also often assumed to be planar graphs with non-negetive weights, and the properties of planar graphs are further utilized to simplify the search process [12], [16], [20], [28]. Moreover, (3) a (spatial) hierarchical index structure is used by several techniques [14], [30], [41]. For example, Sanders et al. [30] proposed a route planning method named Highway Hierarchies (HH) such that only high level edges were considered to compute the path and distance from a source to a far target. Inspired by HH, Geisberger et al. [14] proposed a road network index named Contraction Hierarchies (CH), which is indeed an extreme case of HH. Further, Zhu et al. [41] proposed Arterial Hierarchy (AH) that narrowed the gap between theory and practice in answering shortest path and distance queries.

To achieve high efficiency, except for (bidirectional) Dijkstra, all aforementioned approaches require a preprocessing stage to answer a shortest path or distance query. The query answering stage is highly dependent on the preprocessing stage. Thus, the techniques for preprocessing and for query answering are tightly coupled with each other.

These techniques are different from ours in two aspects. (1) These techniques make use of different methodologies. As a result, applying one technique precludes applying another one. However, our routing proxies are a general technique that can be easily combined with other techniques. That is, routing proxies can be used as a pre-processing step before any other technique is applied. Then, any of these technique can be adopted on the reduce graphs, which are typically much smaller than the original graphs, as we have discussed in Section 5. (2) To achieve high efficiency, these techniques usually incur high preprocessing time and space overhead. In contrast, routing proxies are a light-weight technique that scales well to large networks.

Most close to our work is the study of 1-dominator sets in [34]. Different from the aforementioned techniques, it is proposed for shortest path queries on nearly acyclic directed graphs rather than undirected graphs. When an undirected graph is converted to an equivalent directed graph, each undirected edge is replaced by a pair of inverse directed edges. Hence, 1-dominator sets [34] are not applicable for undirected graphs. However, routing proxies and deterministic routing areas proposed in this study are for undirected graphs, and significantly different from 1-dominator sets (from definitions to analyses to algorithms).

8 Conclusion

We have studied how to speed-up (exact) shortest path and distance queries on large weighted undirected graphs. To do this, we propose a light-weight data reduction technique, a notion of proxies such that each proxy represents a small subgraph, referred to as DRAs. We have shown that proxies and DRAs can be computed efficiently in linear time, and incur only a very small amount of extra space. We have also verified, both analytically and experimentally, that proxies significantly reduce graph sizes and improve efficiency of existing methods, such as bidirectional Dijkstra, ARCFLAG, TNR and AH for shortest path and distance queries.

A couple of topics are targeted for future work. We are to extend our techniques for dynamic graphs, as real-life networks are often dynamic [25], [40]. We are also to explore the possibility of revising routing proxies for directed graphs and other classes of graph queries, *e.g.*, reachability.

APPENDIX: PROOFS

Proof of Proposition 1: Without loss of generality, we only consider those CCs with at least two nodes. Given any node u in a graph G, by conditions (2) and (3) in the definition of proxy, there exists at least one neighbor v of u in A_u^+, and A_u^+ is not maximal, otherwise. By condition (2), all nodes reachable to v are in A_u^+. It is known that all nodes in a CC is reachable to any node in the CC. Hence, all nodes in a CC belong to A_u^+. By these, we have the conclusion. □

Proof of Proposition 2: We show this by contradiction. Assume first that there exists a proxy u such that it has two distinct DRAs: $G[A_{u,1}^+]$ and $G[A_{u,2}^+]$. Then by the definition of DRA, it is trivial to verify the following: (1) $A_{u,1}^+ \not\subset A_{u,2}^+$, (2) $A_{u,2}^+ \not\subset A_{u,1}^+$, and (3) $A_{u,1}^+ \cap A_{u,2}^+$ has at least 2 nodes, and one must be u. By the definition of proxy, u is a proxy of all the set of nodes in $A_{u,1}^+ \cup A_{u,2}^+$, and the DRA of u should be $G[A_{u,1}^+ \cup A_{u,2}^+]$. That is, neither $A_{u,1}^+$ nor $A_{u,2}^+$ is maximal. Hence, both $G[A_{u,1}^+]$ and $G[A_{u,2}^+]$ are not DRAs of node u. This contradicts the assumption. □

Proof of Proposition 3: Consider two distinct proxies u and u' in graph G.
(1) We first show that if $u \in A_{u'}^+$, then $A_u^+ \subseteq A_{u'}^+$.

We show this by contradiction. Assume first that $A_u^+ \not\subseteq A_{u'}^+$ and $u \in A_{u'}^+$. Then there must exist a node $w \in A_u^+$, but $w \notin A_{u'}^+$. Since $w \in A_u^+$, by the definition of proxies, w is reachable to u. Moreover, since $u \in A_{u'}^+$, by the definition of proxies, all nodes reachable to u belong to $A_{u'}^+$ Hence, $w \in A_{u'}^+$, which contradicts our previous assumption.
(2) Similarly to (1), we can show that $A_{u'}^+ \subseteq A_u^+$ if $u' \in A_u^+$.
(3) For the case when $u \notin A_{u'}^+$ and $u' \notin A_u^+$, it is easy based on the analyses of (1) and (2). □

Proof of Proposition 5: Consider a proxy u in a graph, and two nodes v and v' in the DRA $G[A_u^+]$. Let G_s be the subgraph by removing u from $G[A_u^+]$, and let cc_1, ..., cc_h be the CCs of G_s. Observe that (a) $G[A_u^+]$ is simply the union of all CCs cc_1, ..., cc_h and node u, and (b) all CCs have a size equal to or less than $c \cdot \lfloor\sqrt{|V|}\rfloor$ - 1.

There are two cases to consider.
(1) Both nodes v and v' are in a single CC cc_j ($1 \leq j \leq h$). Since CC cc_j has no more than $c \cdot \lfloor\sqrt{|V|}\rfloor - 1$ nodes, it takes a standard Dijkstra algorithm at most $O(|V|)$ time to compute the shortest path between v and v'.
(2) Nodes v and v' are in two distinct CCs cc_i and cc_j ($1 \leq i \neq j \leq h$). As u is the only node that cc_i and cc_j have in common, $path(v, v')$ between v and v' is exactly $path(v, u)$ + $path(u, v')$, which can be computed in $O(|V|)$ time.

Putting these together, we have the conclusion. □

Proof of Proposition 7: We consider non-trivial maximal proxies. By its definition, (1) $v \neq x$ and $u \neq y$, (2) v and u are not neighboring nodes, (3) for any node w not in the DRA $G[A_x^+]$ of proxy x, if w is reachable to v, then x must be a node in any shortest path from w to v, and, similarly, (4) for any node z not in the DRA $G[A_y^+]$ of proxy y, if z is reachable to u, then y must be a node in any shortest path from z to y. This shows that the shortest path from v to u is exactly $path(v, x)/path(x, y)/path(y, u)$, *i.e.,* the concatenation of the three paths. □

Proof of Proposition 10: We show this by contradiction. We first assume that a proxy u in a CC $H(V_s, E_s)$ of graph $G(V, E)$ is not a cut-node. Then we show that u is not a proxy, a contradiction to the assumption.

Let $G \setminus \{u\}$ be the subgraph of G by removing node u from G. Note that $G \setminus \{u\}$ remains connected since u is not a cut node of graph G. By the definition of (non-trivial maximal) proxies, at least one neighbor v of u must belong to A_u. As all nodes in $H \setminus \{u\}$ are reachable to v, it is easy to know that A_u contains all the nodes V_s. Since $|V_s| > c \cdot \lfloor\sqrt{|V|}\rfloor$, which violates the size condition of proxies. Hence, u is not a proxy of G, which contradicts the assumption. □

Proof of Proposition 11: We show this by contradiction.

Assume first that there exists a non-trivial proxy u in a bi-connected component with size larger than $c \cdot \lfloor\sqrt{|V|}\rfloor$ of graph $G(V, E)$. Then we show that u is not a proxy. Let G_s be the subgraph of the bi-connected component with the removal of u. Since the removal of any node in a BCC doesn't increase the number of CCs, G_s remains a CC. By the definition of proxies, A_u contains all the the set of nodes in G_s together with node u. That is, A_u has more than $c \cdot \lfloor\sqrt{|V|}\rfloor$ nodes, and, therefore, u is not a proxy. This contradicts the assumption. □

Proof of Proposition 12: We show this by contradiction.

Assume first there is a cycle $B_1, v_1, B_2, v_2, \ldots, B_k, v_k, B_1$ in a sketch graph $\mathbb{G}$ of graph G, where $B_1, \ldots, B_k$ are BCCs and $v_1, \ldots, v_k$ are cut-nodes of G. Then, it follows that the removal of any cut-node v_i ($i \in [1, k]$) increases the number of CCs in G, by the definition of cut-nodes. However, the removal of any v_i ($i \in [1, k]$) from G does not increase the number of CCs in graph G because of the cycle. This contradicts the assumption.

Note that a similar proposition without a detailed proof is also provided for block graphs in [9]. □

ACKNOWLEDGMENTS

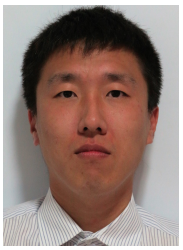
This work is supported in part by 973 Program (No. 2014CB 340300), NSFC (No. 61322207&61421003), Special Funds of Beijing Municipal Science & Technology Commission, and MSRA Collaborative Research Program. We also thank the anonymous reviewers for their valuable comments and suggestions that help improve the quality of this manuscript.

REFERENCES

[1] Graph-tool. *http://projects.skewed.de/graph-tool/*.
[2] J. Arz, D. Luxen, and P. Sanders. Transit node routing reconsidered. In *SEA*, 2013.
[3] H. Bast, D. Delling, A. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. Werneck. Route planning in transportation networks. In *Technical Report MSR-TR-2014-4*. Microsoft Research, 2014.
[4] E. P. F. Chan and H. Lim. Optimization and evaluation of shortest path queries. *VLDB J.*, 16(3):343–369, 2007.

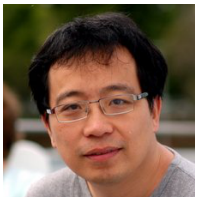
[5] J. Cheng, Y. Ke, S. Chu, and C. Cheng. Efficient processing of distance queries in large graphs: a vertex cover approach. In *SIGMOD*, 2012.
[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2001.
[7] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Robust distance queries on massive networks. In *ESA*, 2014.
[8] D. Delling, A. V. Goldberg, I. Razenshteyn, and R. F. Werneck. Graph partitioning with natural cuts. In *IPDPS*, 2011.
[9] R. Diestel. *Graph Theory*. Springer, 2005.
[10] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
[11] DIMACS. *http://www.dis.uniroma1.it/challenge9*.
[12] J. Fakcharoenphol and S. Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. *JCSS*, 72(5):868–889, 2006.
[13] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. In *FOCS*, 1984.
[14] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *WEA*, 2008.
[15] A. V. Goldberg and C. Harrelson. Computing the shortest path: *A** search meets graph theory. In *SODA*, 2005.
[16] S. Gupta, S. Kopparty, and C. Ravishankar. Roads, codes, and spatiotemporal queries. In *PODS*, 2004.
[17] J. E. Hopcroft and R. E. Tarjan. Efficient algorithms for graph manipulation [h] (algorithm 447). *Commun. ACM*, 16(6):372–378, 1973.
[18] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SISC*, 20(1):359–392, 1998.
[19] B. W. Kernighan and S. Lin. An efficientheuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(1):13–21, 1970.
[20] P. N. Klein, S. Mozes, and O. Weimann. Shortest paths in directed planar graphs with negative lengths: A linear-space o (n log 2 n)-time algorithm. *TALG*, 6(2):30, 2010.
[21] T. Lappas, K. Liu, and E. Terzi. Finding a team of experts in social networks. In *KDD*, 2009.
[22] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.
[23] S. Liu, Y. Yue, and R. Krishnan. Adaptive collective routing using gaussian process dynamic congestion models. In *KDD*, 2013.
[24] M. Luby and P. Ragde. A bidirectional shortest-path algorithm with good average-case behavior. *Algorithmica*, 4(4):551–567, 1989.
[25] S. Ma, J. Li, C. Hu, X. Lin, and J. Huai. Big graph search: challenges and techniques. *FCS*, 2016.
[26] Metis. *http://glaros.dtc.umn.edu/gkhome/views/metis*.
[27] R. H. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm. Partitioning graphs to speedup Dijkstra's algorithm. *ACM Journal of EA*, 11:1–29, 2006.
[28] S. Mozes and C. Sommer. Exact distance oracles for planar graphs. In *SODA*, 2012.
[29] M. Potamias, F. Bonchi, C. Castillo, and A. Gionis. Fast shortest path distance estimation in large networks. In *CIKM*, 2009.
[30] P. Sanders and D. Schultes. Highway hierarchies hasten exact shortest path queries. In *ESA*, 2005.
[31] J. Sankaranarayanan and H. Samet. Query processing using distance oracles for spatial networks. *TKDE*, 22(8):1158–1175, 2010.
[32] J. Sankaranarayanan, H. Samet, and H. Alborzi. Path oracles for spatial networks. *PVLDB*, 2(1):1210–1221, 2009.
[33] A. D. Sarma, S. Gollapudi, M. Najork, and R. Panigrahy. A sketch-based distance oracle for web-scale graphs. In *WSDM*, 2010.
[34] S. Saunders and T. Takaoka. Solving shortest paths efficiently on nearly acyclic directed graphs. *TCS*, 370(1-3):94–109, 2007.
[35] M. Thorup and U. Zwick. Approximate distance oracles. *J. ACM*, 52(1):1–24, 2005.
[36] D. Wagner and T. Willhalm. Speed-up techniques for shortest-path computations. In *STACS*, 2007.
[37] F. Wei. Tedi: efficient shortest path query answering on graphs. In *SIGMOD*, 2010.
[38] L. Wu, X. Xiao, D. Deng, G. Cong, A. D. Zhu, and S. Zhou. Shortest path and distance queries on road networks: An experimental evaluation. *PVLDB*, 5(5):406–417, 2012.
[39] S. Yang, X. Yan, B. Zong, and A. Khan. Towards effective partition management for large graphs. In *SIGMOD*, 2012.
[40] W. Yu, C. C. Aggarwal, S. Ma, and H. Wang. On anomalous hotspot discovery in graph streams. In *ICDM*, 2013.
[41] A. D. Zhu, H. Ma, X. Xiao, S. Luo, Y. Tang, and S. Zhou. Shortest path and distance queries on road networks: towards bridging theory and practice. In *SIGMOD*, 2013.

Shuai Ma is a professor at the School of Computer Science and Engineering, Beihang University, China. He obtained his PhD degrees from University of Edinburgh in 2010, and from Peking University in 2004, respectively. He was a post-doctoral research fellow in the database group, University of Edinburgh, a summer intern at Bell labs, Murray Hill, USA and a visiting researcher of MRSA. He is a recipient of the best paper award for VLDB 2010 and the best challenge paper award for WISE 2013. His current research interests include database theory and systems, social data and graph analysis, and data intensive computing.



Kaiyu Feng is a PhD student at the School of Computer Engineering, Nanyang Technological University, Singapore, supervised by Prof. Gao Cong. He received his BS degree in computer science and technology from Beihang University in 2012. His current research interests include databases and social data analysis.

Jianxin Li is an associate professor at the School of Computer Science and Engineering, Beihang University. He received his PhD degree from Beihang University in 2008. He was a visiting scholar in machine learning department of CMU in 2015, and a visiting researcher of MSRA in 2011. His current research interests include data analysis and processing, distributed systems and system virtualization.



Haixun Wang is a research scientist at Facebook. He received the Ph.D. degree in computer science from the University of California, Los Angeles in 2000. He had been a research scientist at Google Research from 2013 - 2015, a senior research at Microsoft Research Asia from 2009 - 2013, a research staff member at IBM T. J. Watson Research Center from 2000 - 2009. He is a recipient of the best paper award for ER 2009, the best paper award for ICDE 2015 and the ten year best paper award in ICDM 2013. His current research interests include text analytics, natural language processing and knowledge base.

Gao Cong is an associate professor at the School of Computer Engineering, Nanyang Technological University, Singapore. He received his Ph.D. degree in 2004 from the National University of Singapore. He previously worked at Aalborg University, Microsoft Research Asia, and the University of Edinburgh. His current research interests include geo-textual and mobility data management, data mining, social media mining, and POI recommendation.

Jinpeng Huai is a professor at the School of Computer Science and Engineering, Beihang University, China. He received his Ph.D. degree in computer science from Beihang University, China, in 1993. Prof. Huai is an academician of Chinese Academy of Sciences and the vice honorary chairman of China Computer Federation (CCF). His research interests include big data computing, distributed systems, virtual computing, service-oriented computing, trustworthiness and security.