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 paper 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 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 [?], [?], [?], finding the distances between physical locations in road networks [?], and drivers' routing services [?].

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 [?] due to Edsger Dijkstra. Dijkstra's original algorithm runs in $O(n^2)$ [?], and the enhanced implementation with Fibonacci heaps runs in $O(n \log n + m)$ due to Fredman & Tarjan [?], 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 [?].

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 [?] is not acceptable on large graphs (*e.g.*, with tens of millions of nodes and edges) for online applications [?]. Therefore, a lot of optimization techniques have been recently developed to speed up the computation [?], [?], [?], [?], [?], [?], [?], [?], [?].

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 to find 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. In this work, we develop a light-weight data reduction technique to speed-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 **??**). We also give an analysis of routing proxies and DRAs, and show that they hold good properties for speeding-up shortest path and distance queries.

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

(3) Using real-life large road and co-authorship graphs, we finally conduct an extensive experimental study (Section **??**). We find that (a) on average 1/3 nodes in a graph are captured by routing proxies, leaving the reduced graph about 2/3 of the input graph for both road and co-authorship graphs, and (b) routing proxies benefit existing shortest path and distance algorithms, *e.g.*, it reduces around 30% and 20% time for bidirectional Dijkstra [?] and ARCFLAG [?] on road graphs, respectively, and 49%, 4% and 49% time for birirectional Dijkstra, ARCFLAG and TNR [?] on co-authorship graphs, respectively.

Related work. (1) Algorithms for shortest paths and dis-

- *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}@act.buaa.edu.cn.
- *K. Feng and G. Cong is 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, 2014; revised XXX, 2014.

tances have been extensively studied since 1950's, and fall into different categories in terms of different criteria:

- exact distances [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?] and approximate distances [?], [?], [?], [?],
- memory-based [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?] and disk-based algorithms [?], [?],
- for unweighted [?], [?], [?], [?], [?], [?] and weighted graphs [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], and
- for directed [?], [?], [?], [?], [?], [?] and undirected graphs [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?], [?].

In this work, we study the memory-based (exact) shortest path and shortest distance problem on weighted undirected large graphs.

(2) There has recently been extensive work on speeding-up techniques for shortest path and distance queries: bidirectional search [?], hierarchical approaches [?], [?], node and edge labeling [?], [?], shortcuts [?] and graph partitioning [?], [?], [?], [?] (see [?], [?], [?], [?] for recent surveys). Routing proxies and their DRAs can serve as a data reduction technique to reduce graph sizes by removing all the nodes in DRAs, and these techniques can then be adopted to compute shortest paths and distances on the reduced graphs. Hence, these techniques and our work are complementary to each other.

(3) Path oracles have been proposed for spatial networks [?], which take advantage of the fact that shortest paths in road networks are often spatially coherent. It has recently been shown the approach incurs significant preprocessing time and space overhead, which renders them inapplicable for large road networks with millions of nodes [?].

(4) Close to our work is the study of 1-dominator sets in [?], which are proposed for shortest path queries on nearly acyclic directed 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 [?] are not applicable for undirected graphs. Indeed, 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).

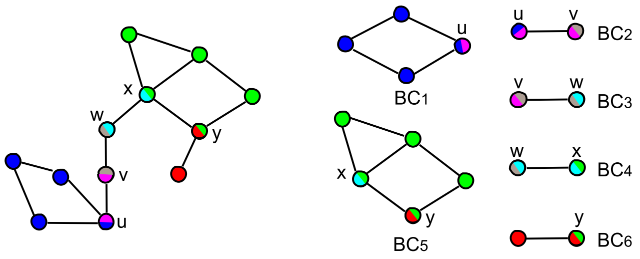
2 Graph Notions

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) for each node $u \in V_s$, $u \in V$, and, moreover, (2) for each edge $e \in E_s$, $e \in E$ and $w_s(e) = w(e)$. That is, subgraph H simply contains a subset of nodes and a subset of edges of graph G.



(1) Graph G1 (2) BCCs of graph G1

Figure 1. Cut-nodes and bi-connected components

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.

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), 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. **??**(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.

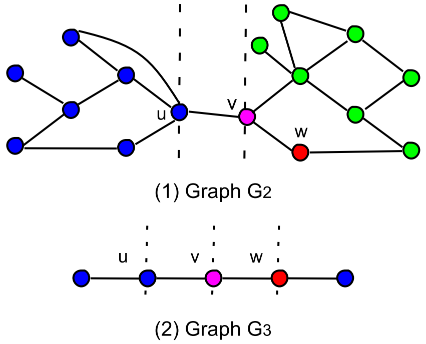


Figure 2. Example proxies and DRAs

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 **??**. We finally show how to answer shortest path and distance queries using routing proxies.

3.1 Routing Proxies and Deterministic Routing Areas

We first present 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.

Note that a node u may be a proxy of multiple sets of nodes $A_u^1, \ldots, A_u^k$ such that $A_u^i \cap A_u^j = \{u\}$ for any $i \neq j \in [1, k]$. And we denote as A_u^+ the union of all the sets of nodes whose proxy is u , *i.e.*, $A_u^+ = A_u^1 \cup \ldots \cup A_u^k$.

Deterministic routing areas . We refer to the *subgraph* $G[A_u^+]$ with nodes A_u^+ as a DRA of proxy u.

Intuitively, DRA $G[A_u^+]$ is a *maximal* connected subgraph connecting to the rest of graph G through proxy u only.

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

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. **??**, 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. **??**, 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 show that proxies and DRAs are *well defined* notions.

Proposition 1: *Any proxy in a graph has a unique* DRA. □

Proof: 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 to our previous assumption. □

Proposition 2: *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. □

Proof: 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 if $u' \in A_u^+$, then $A_{u'}^+ \subseteq A_u^+$.

(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). □

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

Corollary 3: *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 hold good properties for shortest path and distance queries. We first justify the necessity of the size restriction for proxies. Otherwise, DRAs are simply CCs, and are mostly useless.

Proposition 4: *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.* □

Proof: 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. □

The size restriction also 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. □

Proof: 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 of 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 ($1 \le j \le h$). 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 ($1 \le i \ne j \le h$). 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. □

By Proposition **??**, 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 $path(v, x) + path(x, y) + path(y, u)$.* □

Proof: We consider non-trivial maximal proxies. By its definition, (1) $v \ne x$ and $u \ne y$, (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 $path(v, x) + path(x, y) + path(y, u)$. □

By Proposition **??**, 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 **??**, **??** and Corollaries **??**, **??** 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.

Proposition 9: *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.* □

Proof: 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 our 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 to our assumption. □

This motivates us to identify (non-trivial maximal) proxies by utilizing the cut-nodes and BCCs, which will be seen immediately.

Proposition 10: *Any node in a bi-connected component (*BCC*) with size larger than $c \cdot \lfloor\sqrt{|V|}\rfloor$ of graph $G(V, E)$ is a trivial proxy.* □

Proof: 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 u is not a proxy. This contradicts to our previous assumption. □

As we are interested in non-trivial proxies only, those large BCCs could be simply ignored without any side effects.

3.3 Query Answering with Routing Proxies

Based on the above analyses, we present a framework for speeding-up shortest path and distance query answering, which consists of two modules: *preprocessing* and *query answering*. The framework for answering queries using proxies and DRAs is illustrated in Fig. **??**, in which each p_i ($i \in [1, h]$) denotes a proxy, and is associated with its DRA. We next introduce the details of the framework.

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

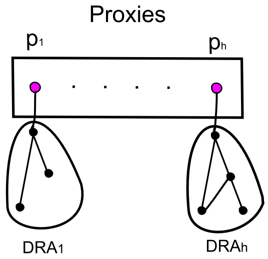


Figure 3. Framework of using proxies

(1) It first computes the DRAs and their maximal proxies, using algorithm computeDRAs (to be seen shortly in Section **??**).
(2) It then pre-computes and stores all the shortest paths and distances between any node in a DRA and its proxy.

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 A^i_u ($i \in [1,h]$), then 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. Observe that $path(s,t)$ = $path(s,u_s)$ + $path(u_s,u_t)$ + $path(u_t,t)$ and $dist(s,t)$ = $dist(s,u_s) + dist(u_s,u_t) + dist(u_t,t)$, in which $path(s,u_s)$, $path(u_t,t)$, $dist(s,u_s)$ and $dist(u_t,t)$ are already known. Hence, it simply invokes an algorithm (*e.g.*, bidirectional Dijkstra [?], ARCFLAG [?], CH [?], TNR [?]) for computing $path(u_s,u_t)$. Similarly, the shortest distance $dist(s,t)$ can be computed.

Let G' be the reduced subgraph of G by removing all the nodes in DRAs, but keeping their proxies, from graph G. It is easy to see that the main computation is reduced from G to G'. As shown in our experimental study (Section **??**), on average about 1/3 nodes of a graph are captured by non-trivial proxies and their DRAs. That is, graph G' is about 2/3 size of graph G, and hence our technique could reduce graph sizes and speed up shortest path and distance computations.

4 Discovering Routing Proxies and DRAs

From Section **??**, 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 distance queries on large graphs.

The main result here is stated as follows.

Theorem 11: *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. We first present BC-SKETCH graphs, a key notion employed by the algorithm.

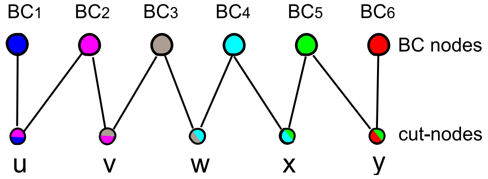


Figure 4. BC-SKETCH graph $\mathbb{G}_1$ of graph G_1

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), and the corresponding BCCs of G_1 in Fig. **??**(2).

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

One may notice that there are no cycles in the BC-SKETCH graph $\mathbb{G}_3$. This is not a coincidence, as shown below.

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

Proof: 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. That is, the removal of any v_j $j \in [1,k]$ does not increase the number of CCs in graph G. However, by the definition of cut-nodes, the removal of any nodes in $v_1, \ldots, v_k$ will increase the number of CCs in G. This is a contradiction.

Putting these together, we conclude that there exist no cycles in sketch graphs. □

Proposition **??** indicates that we can employ the good properties of trees for computing DRAs and maximal proxies.

We are now ready to present algorithm computeDRAs shown in Fig. **??**. It takes as input graph G and constant c, and outputs the DRAs of G, each associated with a maximal proxy.

(1) Finding cut-nodes and BCC*s*. 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 [?], [?].

(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.

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

```
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 the DRAs with their maximal
   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 $\mathbb{G}$;
    /* note that 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.   mark $X'$ as the DRA $A^+_{v'}$ of proxy $v'$;
16. return all DRAs with their maximal proxies.
```

Figure 5. Computing DRAs and maximal proxies

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 constitute 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. In 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.

Name	Regions	# of Nodes	# of Edges
DBLP14	A 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

Table 1
Real-world graphs

We now explain the algorithm with an example as follows.

Example 4: Consider graph G_1 in Fig. **??**(1) again. Here we let $c = 2$, and $c \cdot \lfloor\sqrt{|V|}\rfloor$ = 6. Firstly, cut-nodes and BCCs are computed as shown in Fig. **??**(2). Secondly, the BC-SKETCH graph $\mathbb{G}_1$ of G_1 is constructed as shown in Fig. **??**. After the merging step stops, the updated BC-SKETCH graph consists of three BCC nodes: BC'_1 = $\{BC_1, BC_2, BC_3\}$, BC_4, BC'_2 = $\{BC_5, BC_6\}$ and two cut-nodes: w and x. Finally, the DRAs and their maximal proxies are identified: proxy w with DRA BC'_1 and proxy x with DRA BC'_2. □

Correctness & Complexity. The correctness of algorithm computeDRAs can be readily verified based on the analyses in Section **??**.

The linear-time complexity of algorithm computeDRAs roots from Proposition **??**. 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 **??**.

5 Experimental Study

We next present an experimental study to show how proxies speed up shortest path and distance queries. Using real-life networks, we conducted three sets of experiments to evaluate: (1) the performance of proxies, (2) the efficiency of (bidirectional) Dijkstra [**?**], ARCFLAG [**?**], TNR [**?**] and their counterparts with proxies (Proxy+Dijkstra, Proxy+ARCFLAG, Proxy+TNR) with respect to graph queries, and (3) the efficiency of these algorithms with respect to graph sizes.

5.1 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.

The first type of datasets is *co-authorship networks*. We extracted co-authorship graphs from DLBP [**?**], where each node in the graph represents an author and two authors are connected if they have published papers together. The edge weight is computed by a revised Adamic/Adar similarity

Graphs	Proxies (#, %)	Nodes in DRAs (#, %)	space (MB)	time (Sec.)
DBLP14	(14, 090, 9.8)	(102,085, 72.2)	0.83	1.5
DBLP	(31,475 , 9.9)	(105,671 , 33.3)	0.93	5.7
CO	(56,277, 12.9)	(156,329, 35.9)	1.41	3.47
FL	(140,382, 13.1)	(378,804, 35.4)	3.42	9.9
CA	(273,191, 14.4)	(623,811, 33.0)	5.80	21.1
E-US	(546,481, 15.2)	(1,228,876, 34.1)	11.46	52.5
W-US	(869,907, 13.9)	(2,116,382, 33.8)	19.46	111.9
C-US	(2,034,358, 14.4)	(4,583,413, 32.5)	42.73	435.8
US	(3,452,222, 14.4)	(7,927,453, 33.1)	73.65	1,925.4

Table 2
Effectiveness of proxies and DRAs

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. Since the DBLP dataset is very dense and it takes a lot of time for the preprocessing of TNR (it took more than 1 week to finish), we removed all nodes whose degrees are higher than 14, and picked the largest connected component from the remaining graph to form a new dataset, referred to as DBLP14.

The second type of datasets is *road networks*. We chose seven road network datasets of various sizes from the Ninth DIMACS Implementation Challenge [?]. Each road network is released as an undirected graph that represents a part of the road network in the United States, where each edge weight is the distance (integers) 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 a Dijikstra algorithm from u to find the node s that is farthest from u. Then we run a Dijkstra 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 $Q_i (i \in [1,8])$, such that the grid distance of all node pairs in Q_i is in $[2^{i-9} \cdot \ell, 2^{i-8} \cdot \ell)$. For each query set Q_i $(i \in [1,8])$, we report the average running time of 10,000 queries in the set.

Algorithms. We implemented algorithms bidirectional-Dijkstra, ARCFLAG and TNR. For ARCFLAG, it needs to partition graphs first to pre-compute information on whether an arc is useful for a shortest path search. Any possible partition methods [?], [?], [?], [?] can be used here. Considering that we have both road networks and co-authorship networks, we adopted the latest version 5.0.2 of METIS [?], 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 [?] that does not require the geometry information.

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

5.2 Experimental Results

We next present our findings. In all experiments, we tested the datasets in Table 1, and fixed the constant $c = 2$ for computing proxies on all graphs. The preprocessing of ARCFLAG was slow, and it spent over 4 days on C-US. Hence, we did not test the largest road network dataset US for ARCFLAG. TNR invokes Contract Hierarchies (CH) for preprocessing and CH is very slow on dense graphs. Unfortunately DBLP is a very dense graph and it took more than 7 days for CH to finish preprocessing on DBLP. Thus we tested DBLP14, instead of DBLP, for the performance of shortest path and distance queries. The CH-based TNR whose space cost was very high, and consequently, and we could not run either TNR or Proxy+TNR on the largest road network dataset US, as both of them ran out of memory. Hence, we did not test the largest graph for TNR as well. However, on the second largest road network dataset C-US, Proxy+TNR can while TNR cannot. We report this to show that proxies serve as a data reduction technique and benefit existing methods in terms of space cost as well.

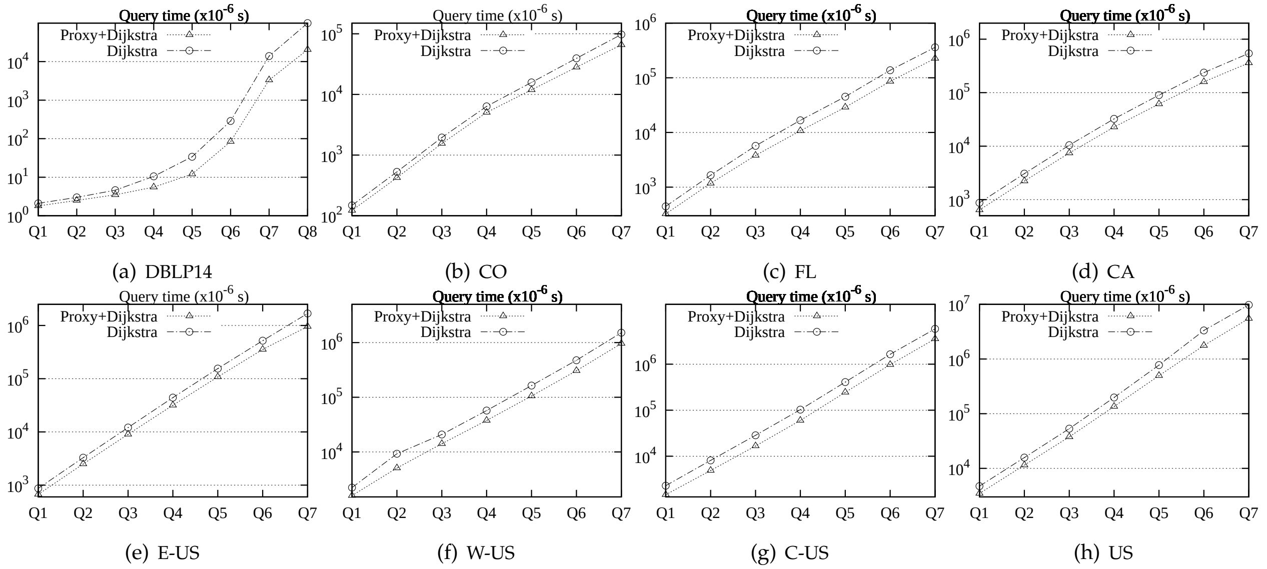
Exp-1: Performance evaluation. In the first set of experiments, we evaluated (1) the number of non-trivial proxies, (2) the number and percentage of the nodes represented by the proxies (excluding the proxies themselves from DRAs), (3) the extra space cost of using proxies, and (4) the efficiency of our algorithm computeDRAs for computing proxies and their DRAs. Note that to speed up shortest path and distance queries, for each node represented by a proxy, we need to store (1) the distance between the node and its proxy, and (2) the shortest path from the node to its proxy. Distances are stored as 4-byte integers and it takes $4 \cdot |V_{dras}|$ bytes to store all the distances, where V_{dras} is the set of nodes in all DRAs. All the shortest paths between the nodes in a DRA and its proxy form a tree whose root is the proxy. Each node is also represented as a 4-byte integer. So it takes $4 \cdot (|V_{proxy}| + |V_{dras}|)$ bytes to maintain all the trees, where V_{proxy} is the set of proxies. So the extra space cost of using proxies is $4 \cdot |V_{proxy}| + 8 \cdot |V_{dras}|$ bytes. The results are reported in Table 2.

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

In all the road graphs, about $1/7$ nodes are non-trivial proxies, and about $1/3$ nodes are captured by proxies in these graphs, which means basically the reduced graph is only about $2/3$ of the input graph. Moreover, although the size restriction is $\leq 2 \cdot \lfloor \sqrt{|V|} \rfloor$, DRAs are typically small, and each proxy represents 2 or 3 other nodes on average.

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. Furthermore, the space cost is small, and only less than 100MB extra space is taken for the largest graph US.

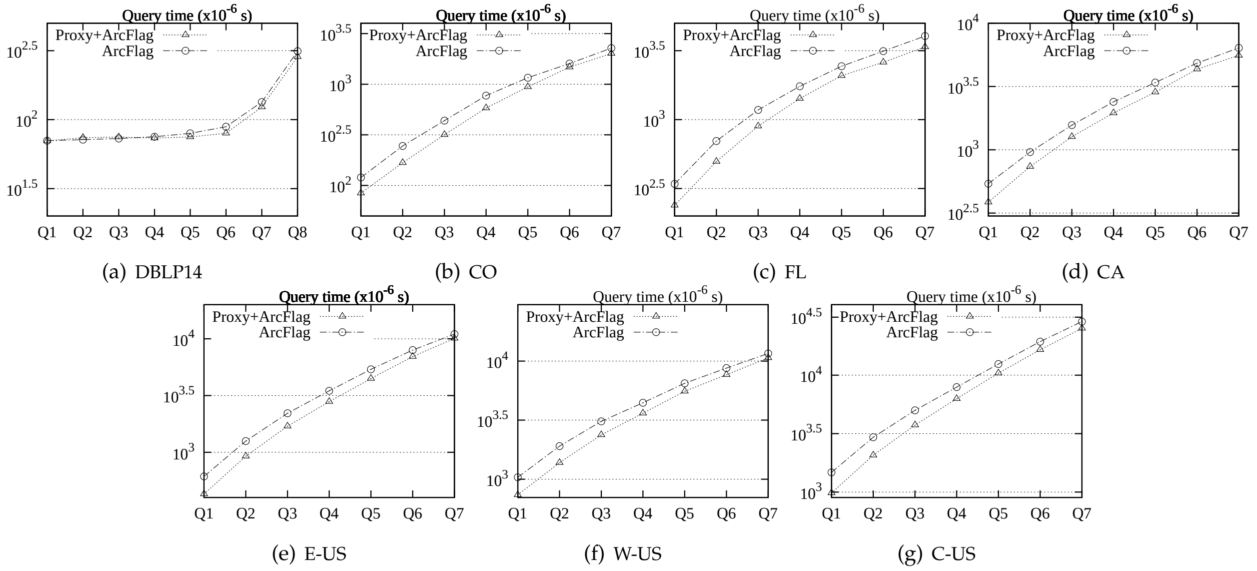
These make proxies a light-weight optimization technique, which benefits existing shortest path and distance



(a) DBLP14 (b) CO (c) FL (d) CA

(e) E-US (f) W-US (g) C-US (h) US

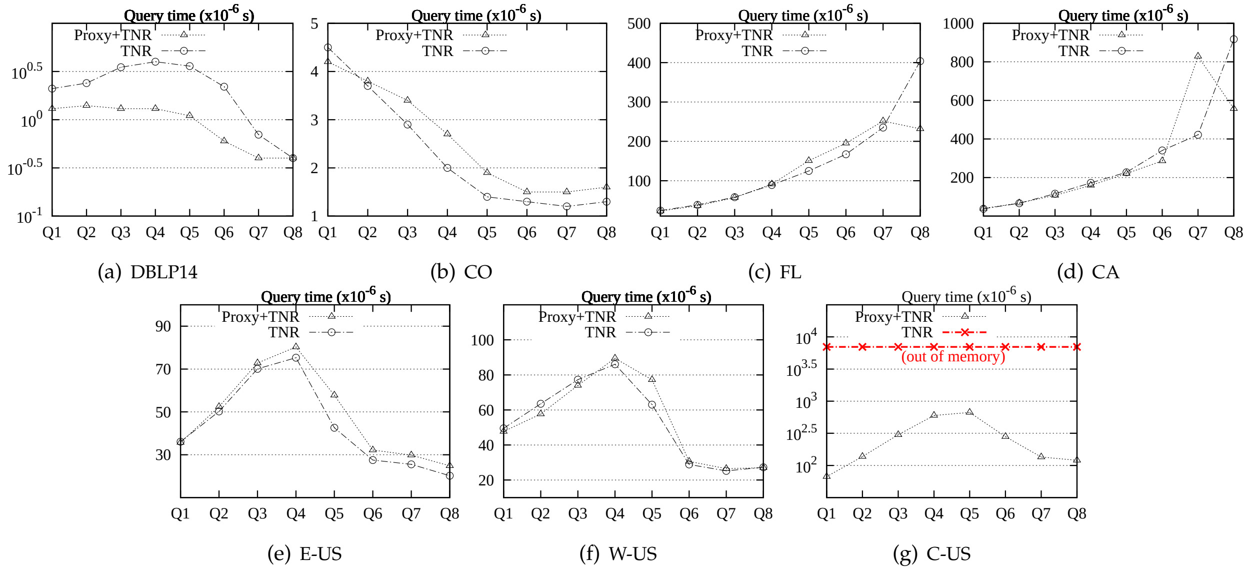
Figure 6. Varying graph queries (Dijkstra and Proxy+Dijkstra for shortest distances)



(a) DBLP14 (b) CO (c) FL (d) CA

(e) E-US (f) W-US (g) C-US

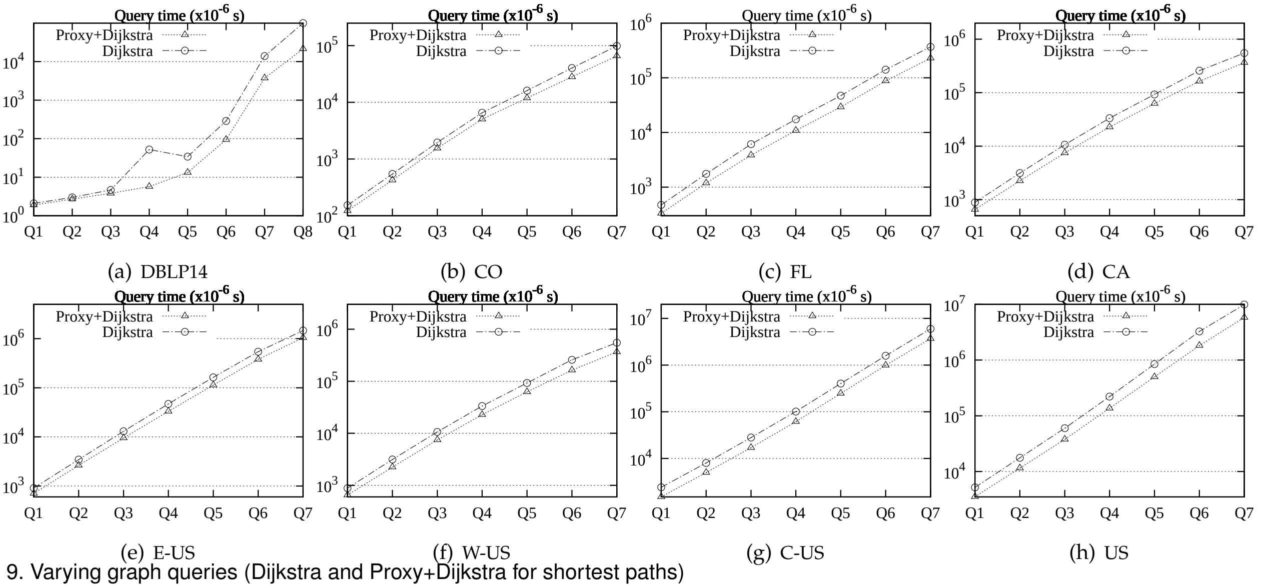
Figure 7. Varying graph queries (ARCFLAG and Proxy+ARCFLAG for shortest distances)



(a) DBLP14 (b) CO (c) FL (d) CA

(e) E-US (f) W-US (g) C-US

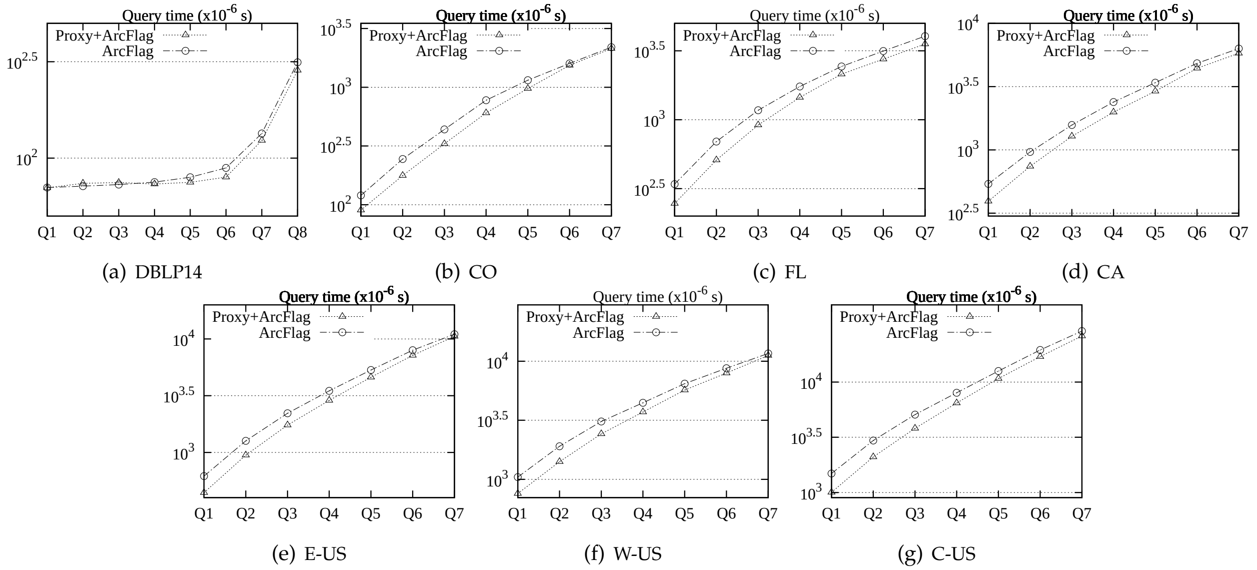
Figure 8. Varying graph queries (TNR and Proxy+TNR for shortest distances)



(a) DBLP14 (b) CO (c) FL (d) CA

(e) E-US (f) W-US (g) C-US (h) US

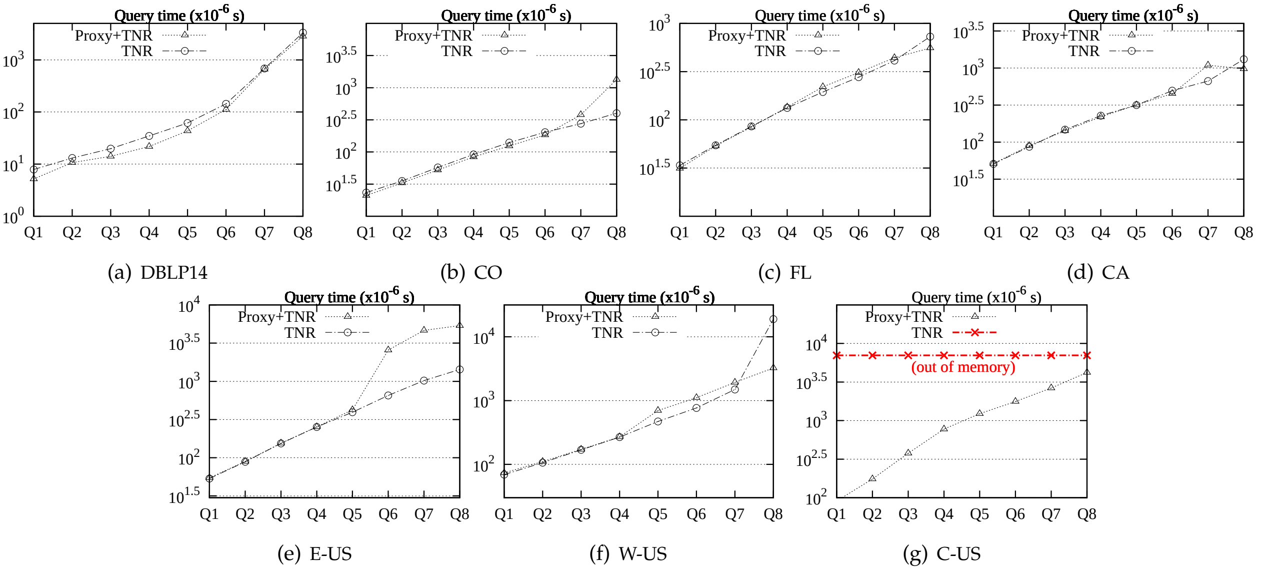
Figure 9. Varying graph queries (Dijkstra and Proxy+Dijkstra for shortest paths)



(a) DBLP14 (b) CO (c) FL (d) CA

(e) E-US (f) W-US (g) C-US

Figure 10. Varying graph queries (ARCFLAG and Proxy+ARCFLAG for shortest paths)



(a) DBLP14 (b) CO (c) FL (d) CA

(e) E-US (f) W-US (g) C-US

Figure 11. Varying graph queries (TNR and Proxy+TNR for shortest paths)

algorithms, such as (bidirectional) Dijkstra [?], ARCFLAG [?] and TNR [?] (to be seen immediately).

Exp-2: Effectiveness w.r.t. *graph queries*. In the second 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 [?]. Note that TNR invokes Contract Hierarchies (CH) to preprocess the graph and CH runs very slow on DBLP dataset (it takes 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 Figures **??**, **??**, **??** and Figures **??**,**??**, **??**, respectively.

In the co-authorship network DBLP14, the results tell us that with the help of proxies, Proxy+Dijkstra, Proxy+ARCFLAG and Proxy+TNR can all achieve a better performance than their counterparts Dijkstra, ARCFLAG and TNR without using proxies. 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, (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 in 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 in the DRAs. Since there are no speed-up techniques used inside a DRA, so the search space saved by proxies is less than using ARCFLAG alone.

In the road networks, the results tell us that with the help of proxies, Proxy+Dijkstra and Proxy+ARCFLAG can achieve a better performance than their counterparts without proxies. On average, the time cost of Proxy+ARCFLAG and Proxy+Dijkstra is about 80% and 68% of their counterparts without proxies for distance queries, and 82% and 67% 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, and (3) different from ARCFLAG, proxy+Dijkstra has a better performance when the query nodes are far from each other. 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 are 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.

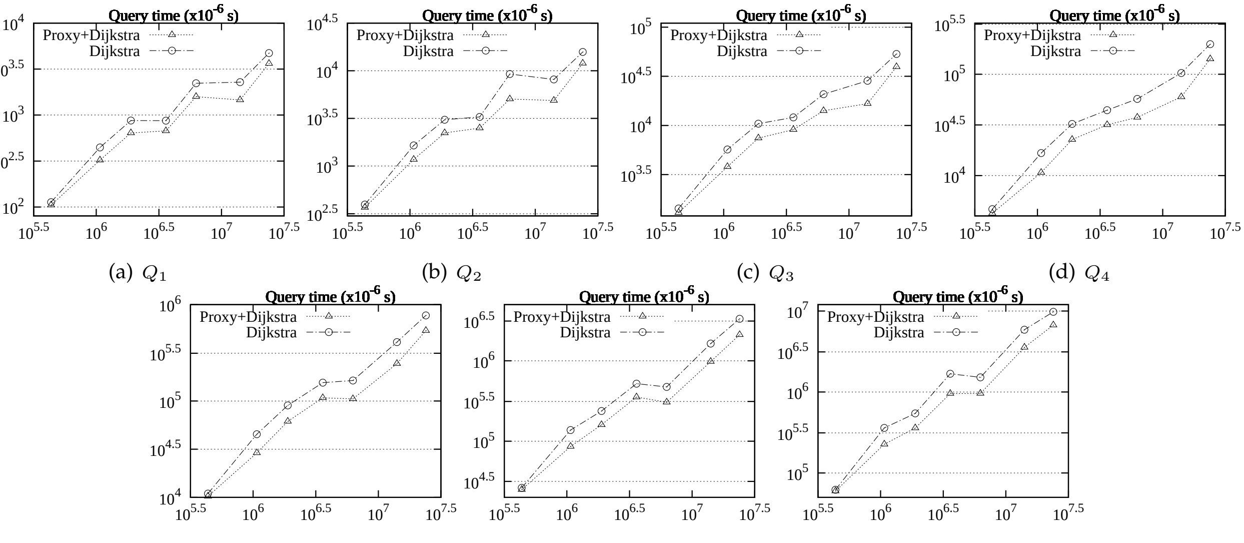
Proxy+TNR achieves a comparable performance to its counterparts without proxies. This is because in TNR, a heuristic method is used to generate the node order, based on the structure of the graph. And the node order can affect the performance of TNR. Since we reduce the input graph by using proxies, the reduced graph has a different topology structure. Thus a different node order will be generated. 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 in 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.

Exp-3: Effectiveness w.r.t. *graph sizes*.

Since DBLP14 is a single dataset, and hence we only compare the efficiency of a group of shortest path and distance queries $\{Q_1, \ldots, Q_7\}$ on road graphs using the same settings as Exp-2. The results of distance and path queries are reported in Figures **??**, **??**, **??** and Figures **??**, **??**, **??**, respectively.

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, and (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 its counterpart without proxies for shortest path and distance queries on average, respectively, (5) for Proxy+TNR, it is applicable to handle larger dataset C-US while its counterpart TNR without proxies is not.

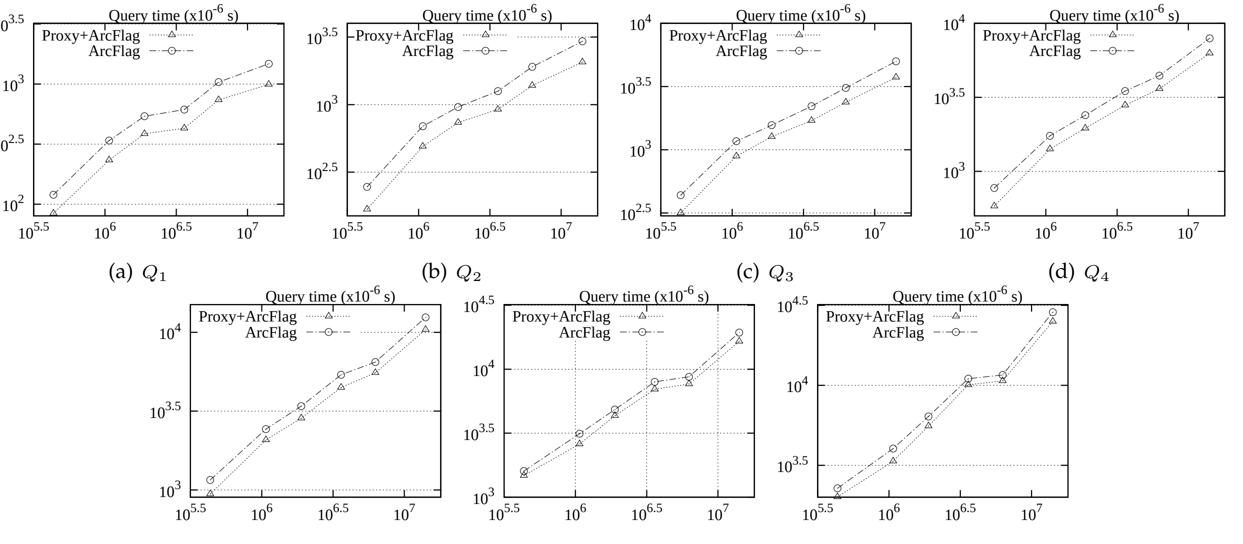
Summary. From these experimental results, we find the following. (1) Proxies are a light-weight preprocessing technique, which can be computed efficiently and take linear space to support shortest path and distance queries. (2) According to our experiments, in most cases, about 1/3 nodes in the graph 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) Proxies and their DRAs benefit existing shortest path and distance algorithms in terms of time cost and space cost. In terms of time cost, *e.g.,* they reduce 20% and 30% time for ARCFLAG and bidirectional Dijkstra in road networks, respectively. They also has comparable time cost to their counterparts without proxies for TNR in road networks; They reduce 49%, 4% and 49% time for birirectional Dijkstra, ARCFLAG, and TNR on co-authorship network DBLP14, respectively. In terms of space cost, since the input graph is reduced by proxies, larger datasets can be handled when existing methods are combined with proxies. *e.g.,*TNR with proxies can handle road network C-US while TNR cannot.



(a) Q_1 (b) Q_2 (c) Q_3 (d) Q_4

(e) Q_5 (f) Q_6 (g) Q_7

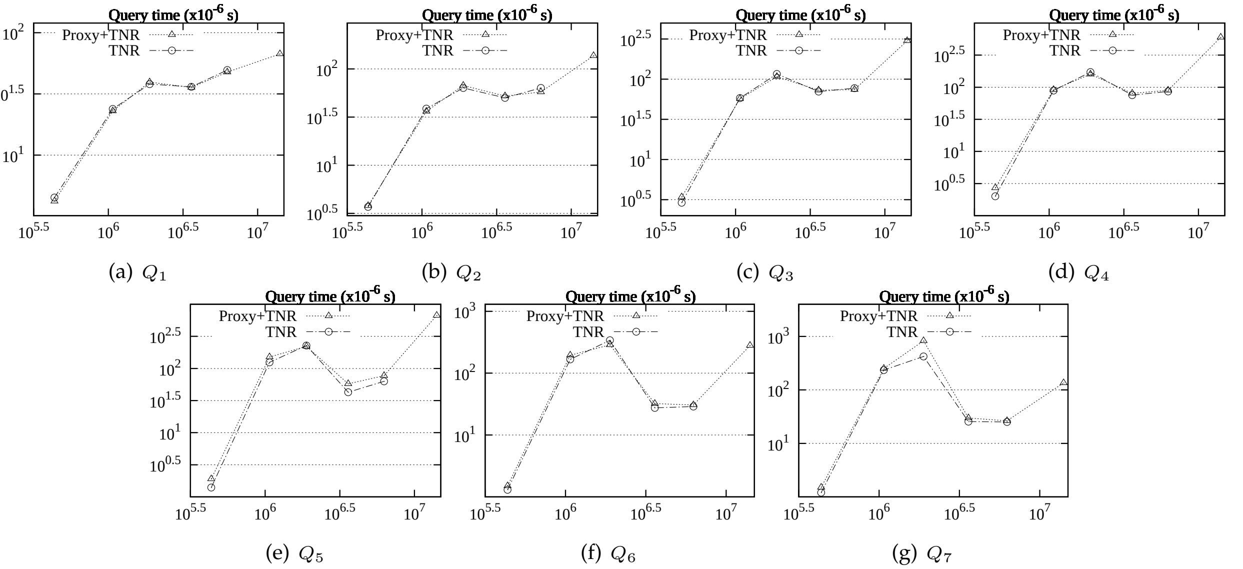
Figure 12. Varying graph sizes (Dijkstra and Proxy+Dijkstra for shortest distances)



(a) Q_1 (b) Q_2 (c) Q_3 (d) Q_4

(e) Q_5 (f) Q_6 (g) Q_7

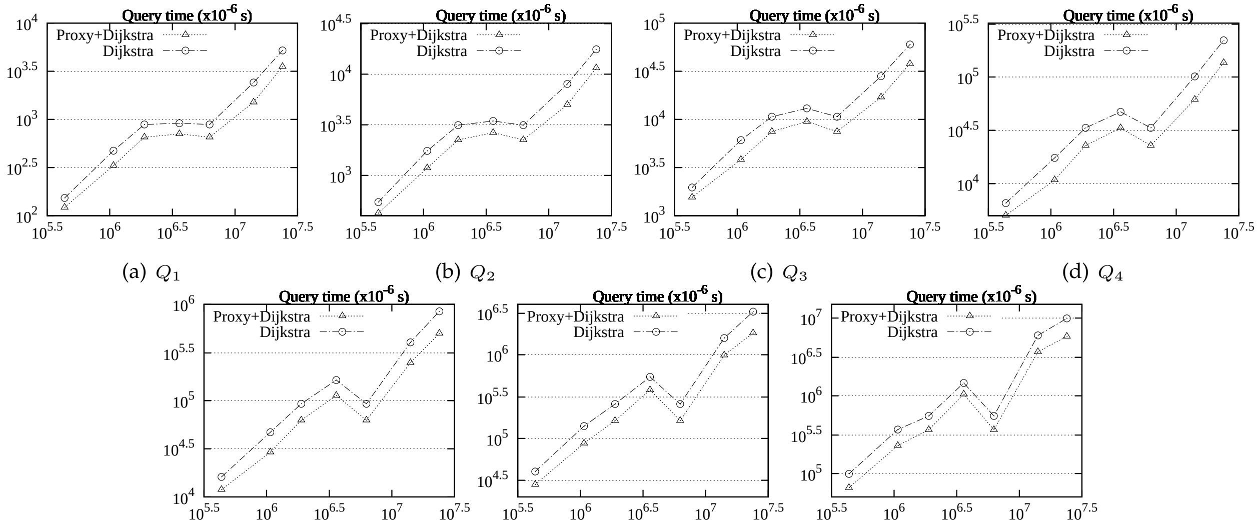
Figure 13. Varying graph sizes (ARCFLAG and Proxy+ARCFLAG for shortest distances)



(a) Q_1 (b) Q_2 (c) Q_3 (d) Q_4

(e) Q_5 (f) Q_6 (g) Q_7

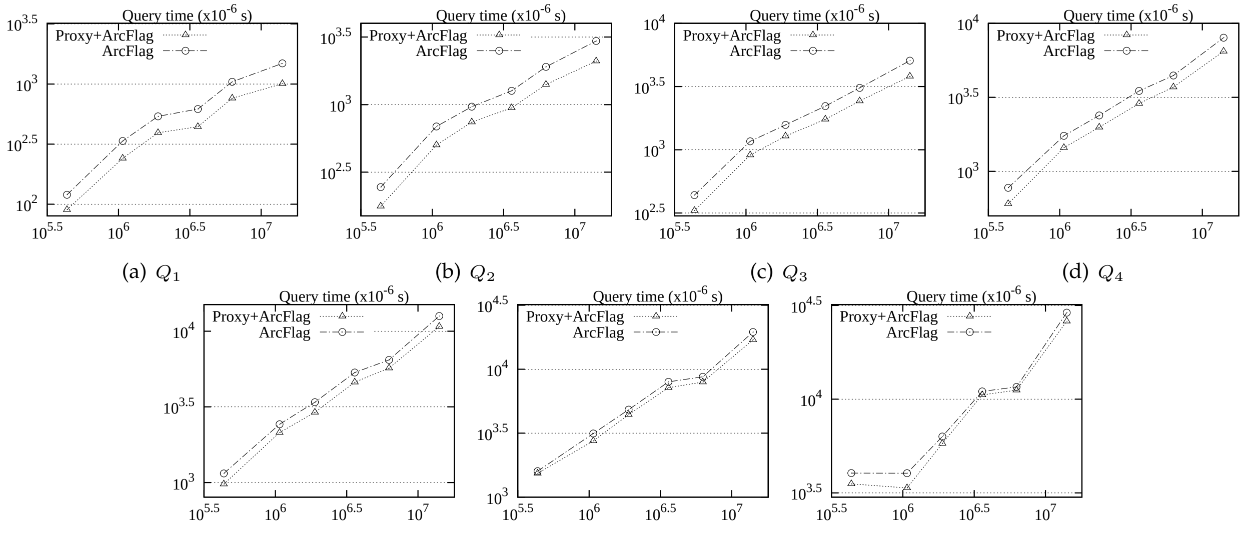
Figure 14. Varying graph sizes (TNR and Proxy+TNR for shortest distances)



(a) Q_1 (b) Q_2 (c) Q_3 (d) Q_4

(e) Q_5 (f) Q_6 (g) Q_7

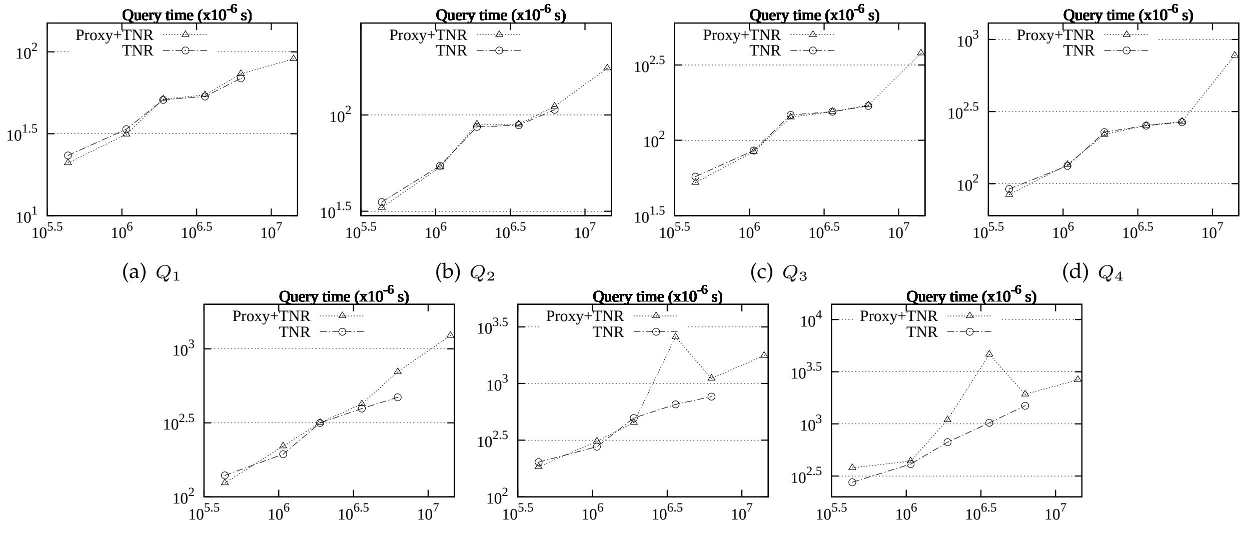
Figure 15. Varying graph sizes (Dijkstra and Proxy+Dijkstra for shortest paths)



(a) Q_1 (b) Q_2 (c) Q_3 (d) Q_4

(e) Q_5 (f) Q_6 (g) Q_7

Figure 16. Varying graph sizes (ARCFLAG and Proxy+ARCFLAG for shortest paths)



(a) Q_1 (b) Q_2 (c) Q_3 (d) Q_4

(e) Q_5 (f) Q_6 (g) Q_7

Figure 17. Varying graph sizes (TNR and Proxy+TNR for shortest paths)

6 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 and TNR for shortest path and distance queries.

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

Acknowledgments

This work is supported in part by 973 program (No. 2014CB340300) and NSFC (No. 61322207).

PLACE PHOTO HERE

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 postdoctoral research fellow in the database group, University of Edinburgh, a summer intern at Bell labs, Murray Hill, USA, in the summer of 2008, and a visiting researcher of MRSA in 2012. 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 analysis and data intensive computing.

PLACE PHOTO HERE

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 20012. His current research interests include databases and social data analysis.

PLACE PHOTO HERE

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 2007. His current research interests include distributed systems, system virtualization, large scale data management and processing systems.

PLACE PHOTO HERE

Haixun Wang is a research scientist at Facebook. He had been a research scientist at Google Research from 2015 - 2013, 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 was Technical Assistant to Stuart Feldman (Vice President of Computer Science of IBM Research) from 2006 to 2007, and Technical Assistant to Mark Wegman (Head of Computer Science of IBM Research) from 2007 to 2009. He received the Ph.D. degree in computer science from the University of California, Los Angeles in 2000. He has published more than 150 research papers in referred international journals and conference proceedings. He served PC Chair of conferences such as CIKM12 and he is on the editorial board of IEEE Transactions of Knowledge and Data Engineering (TKDE), and Journal of Computer Science and Technology (JCST). He won the best paper award in ICDE 2015, 10 year best paper award in ICDM 2013, and best paper award of ER 2009.

PLACE PHOTO HERE

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. Prior to that, he worked at Aalborg University, Microsoft Research Asia, and the University of Edinburgh. His current research interests include geo-textual data management and data mining.

PLACE PHOTO HERE

Jinpeng Huai a professor in the School of Computer Science and Engineering at 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 system, virtual computing, service-oriented computing, trustworthiness and security.