Neighborhood-Privacy Protected Shortest Distance Computing in Cloud

Jun Gao† Jeffery Yu Xu‡ Ruoming Jin§ Jiashuai Zhou† Tengjiao Wang† Dongqing Yang†
† Key Laboratory of High Confidence Software Technologies, EECS, Peking University
‡ Department of systems engineering & Engineering Management,Chinese University of Hong Kong
§ Department of Computer Science,Kent State University
{gaojun,jszhou,tjwang,dqyang}@pku.edu.cn, yu@se.cuhk.edu.hk, jin@cs.kent.edu

ABSTRACT

With the advent of cloud computing, it becomes desirable to utilize cloud computing to efficiently process complex operations in large graphs without compromising their sensitive information. This paper studies shortest distance computing in the cloud, which aims at the following goals: i) preventing outsourced graphs from neighborhood attack, ii) preserving shortest distances in outsourced graphs, iii) minimizing overhead on the client side.

The basic idea of this paper is to transform an original graph G into a link graph G_l kept locally and a set of outsourced graphs $\mathcal{G}_o$. Each outsourced graph should meet the requirement of a new security model called 1-neighborhood-d-radius. In addition, the shortest distance query can be equivalently answered using G_l and $\mathcal{G}_o$. Our objective is to minimize the space cost on the client side when both security and utility requirements are satisfied. We devise a greedy method to produce G_l and $\mathcal{G}_o$, which can exactly answer the shortest distance queries. We also develop an efficient transformation method to support approximate shortest distance answering under a given additive error bound. The final experimental results illustrate the effectiveness and efficiency of our method.

Categories and Subject Descriptors

H.2.8 [**Database management**]: Database Applications

General Terms

Algorithms

Keywords

Shortest Distance, Graph Transformation, Outsource, Privacy.

1. INTRODUCTION

Graph structured data are used in numerous applications, *e.g.,* web graphs, social networks, ontology graphs, biological and chemical pathways, transportation networks. High efficiency is essential for frequent and basic graph operations. However, even basic operations on a graph can be very time-consuming due to the complexity of structural connectivities and graph size [4]. Moreover, real graph datasets are growing rapidly in size, making the attainment of high efficiency even harder.

The paradigm shift of cloud computing offers a new approach for storage- and compute- intensive tasks [5, 10, 26, 20], allowing users to migrate their burden (*e.g.,* data maintenance and computing utilities) to an outsourced server (or cloud server). The outsourced server typically has sufficient resources to maintain very large datasets and provides quick response to users' requests with its powerful distributed and parallel architecture.

Thus, it becomes desirable (and even inevitable) to employ cloud computing to manage large graphs and particularly to efficiently process complex graph operations. The biggest problem with this approach is that client companies may be unwilling to outsource their valuable datasets. Take the outsourcing of a social network as an example. Even if its owner removes the identifier of each vertex before sending the network to the cloud server, the structural relationships can still possibly be recovered [14, 19], which will surely devalue the outsourcing.

Given this, the fundamental challenge is: *How can we utilize cloud computing to efficiently process complex operations in large graphs without compromising their sensitive information?* In this paper, we focus on computing the shortest distance between two arbitrary vertices in a large edge-weighted graph. This operation is one of the most important and widely-used graph operations [22, 7, 25, 21], yet its direct online computation is expensive on large graphs [4]. The straightforward way to process shortest distance queries using cloud computing requires outsourcing the entire graph. Thus, the solution to this problem (by avoiding outsourcing sensitive information but still utilizing the high computational power of the cloud) will not only directly benefit graph distance computation and other related operations [16], but more importantly shed light on the principles and fundamental techniques for preserving privacy in cloud computing.

1.1 Related Work

In the following, we review the current state-of-the-art techniques and point out why they cannot fully address the "privacy-in-cloud" challenge.

Privacy-Preserving Graph Publishing: Privacy protection for graph publishing has been studied recently. Most of the existing work on graph publishing focuses on certain structural anonymizations, such as 1-neighborhood [2], k-degree [13], k-automorphism [17], k-isomorphism [11], cluster based vertex anonymity [9, 8], as well as many others. These techniques typically focus on using the least amount of modifications of the original graph (minimal information loss) to make it satisfy the targeted security requirement. Unfortunately, the anonymized graphs produced from these privacy

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'11, June 12–16, 2011, Athens, Greece.
Copyright 2011 ACM 978-1-4503-0661-4/11/06 ...$10.00.

protection techniques generally do not necessarily maintain the statistical and graph theoretical characteristics of the original [2, 13, 17, 11]. In particular, for any pair of vertices, there is no guarantee of the degree of similarity or preservation of shortest distances between the anonymized graph and the original graph. For example, k-isomorphism is proposed recently to partition a graph into k disjoint, isomorphic subgraphs, which unfortunately cannot be used to compute shortest distances in the original graph [11]. In addition, most of the exiting works deal with privacy on unweighted graphs, and do not consider the impact of edge weights.

A few recent works [27, 15, 24, 18] notice the importance of preserving graph theoretical characteristics during graph publishing. Ying and Wu propose to preserve the eigenvalue of a graph, which relates to average shortest distance and other topological features, during graph transformation [27]. Liu *et al.* study edge weight perturbation by Guassian random or heuristic rules [15]. Das *et al.* propose a linear programming (LP) method to change edge weights while preserving shortest paths [24]. In both [15] and [24], the topological structure of the anonymized graph remains unchanged. Thus, even with the minimal topological knowledge, such as the vertex degree [13], some sensitive information can be re-identified. Furthermore, the greedy perturbation [15] relies on an expensive matrix operation, and the LP approach can be easily overwhelmed by the number of inequality rules (shortest path preservation conditions) [24]. For example, for a connected graph with only one thousand vertices, there are one million rules for LP, which is clearly too expensive.

Recently, differential privacy [6] has emerged as a powerful model to protect against unknown adversaries with guaranteed probabilistic accuracy. Hay, Li *et al.* performed some of the first studies to support differential privacy in analyzing networks [18, 3]. Specifically, they design an efficient method for releasing a provably private estimate of the degree distribution of a network. However, it is still an open problem on how to publish a graph with respect to the differential privacy [18], and thus it is also not clear whether the techniques developed in [18] can be applied to more complicated queries, such as the shortest-path distance query.

Security Issues in Outsourced Server: Sensitive data protection and verification of query results in the outsourced server have attracted much attention recently [10, 26]. A work closely related to this paper studies the verification issue in outsourcing graphs for shortest path discovery [20]. In their solution, the original graph data are outsourced along with verification objects, and the client side will validate the correctness of the results with the verification objects. However, they do not consider how to protect the sensitive information of the original graph.

Shortest Path Discovery: Shortest path discovery is one fundamental problem in graph theory. Dijkstra's algorithm[4] is a well-known approach to find the single-source shortest path. There is also some work to exploit pre-computed indices to speed up the running time of shortest path discovery. HEPV [23] and HiTi [25] can be viewed as multiple-level indices. The combination of the A* algorithm with the landmark index is studied in [1]. 2-HOP index [7] assigns each vertex with two vertex label sets (L_{out} and L_{in}), and finds the shortest distance between two nodes with an intersection of their label sets. In addition, the indices with different error bounds and various construction methods for approximate distance answering have also been studied in [22, 12, 21]. However, we cannot simply outsource such indices as they will also disclose sensitive information of the graph. For instance, in the 2-hop index, each vertex is very likely to record its distance to its immediate neighbors. Such information often needs to be protected [2].

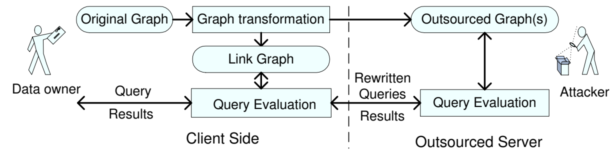


Figure 1: An Overview

1.2 Overview

Given this, our goal is to search for a computational scheme which can compute shortest distances while also protecting private information in the original graph. The key issues underlying this problem are i) what information should be protected and what can be exposed, and ii) how can we employ cloud computing while minimizing client side overhead and satisfying privacy constraints? In addition, recall that the goal in graph publishing is to generate an anonymized graph which should be similar to the original one in order to minimize information loss [2, 13, 17, 11]. Here, the information which we send to the cloud server is for computing shortest distances. Thus, it does not have to be a graph or even a subgraph; it can be any format as long as it does not disclose the private information of the original graph. This goes back to the core problem: *what information should we protect?* Indeed, most graph publishing techniques are designed to protect specific types of private information. The most common type of attack is a neighborhood attack [2]. For example, outsiders to a social network can easily collect some users' neighborhood information. However, it generally becomes increasingly difficult for attackers to learn knowledge beyond the direct neighborhood [2]. Thus, in this paper, we focus on protecting private information against neighborhood attacks.

The overall framework of our approach is illustrated in Fig. 1. Given an original edge-weighted graph G, we represent it in two parts for both privacy protection and shortest distance computation: i) an outsourced graph(s) G_o is a high level abstraction of the original graph, which records the key information for online shortest distance answering, but does not contain sensitive neighborhood information; ii) a link graph, G_l, which includes the local private information and the relationships between vertices in G and vertices in an outsourced graph(s) G_o for shortest distance computation. The graph(s) G_o is on the cloud server and the link graph G_l is on the client side. Note that this graph representation bears a certain similarity to the 2-level index for shortest paths [23, 25]. However, the existing multiple indices such as HEPV and HiTi mainly target planar graphs, since discovering good graph separators in indexing is difficult on non-planar graphs. More importantly, the present work has a different focus: to construct an outsourced graph which eliminates the private local neighborhood knowledge and makes full use of the cloud server.

The main contributions are summarized below.

- We formulate a new graph transformation problem as minimizing the size of the link graph G_l on the client side on the condition that the privacy of outsourced graphs $\mathcal{G}_o$ is protected and shortest distances can be answered using $\mathcal{G}_o$ and G_l. We propose a new security model, named 1-neighborhood-d-radius, which hides local details in direct edges or a d-radius for each vertex to counter neighborhood attacks.
- We propose a greedy approach to generate outsourced graphs and a link graph for exact shortest distance answering with protected neighborhood privacy.
- We study how to answer approximate shortest distances in the same context with an average additive distance error bound.

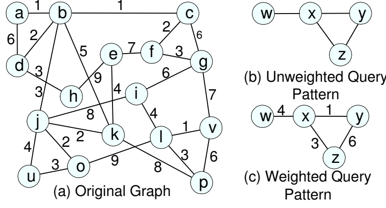


Figure 2: Sample Graph

$G(V, E)$	The original graph
$G_l(V_l, E_l)$	The link graph
$G_o(V_o, E_o)$	An outsourced graph
$\mathcal{G}_o$	Outsourced graphs
n/m	The number of vertices/edges in G
n_o	The number of vertices in G_o
n_l	The number of vertices in cluster

Table 1: Symbols

By allowing approximate shortest distance to be answered, we show that $\mathcal{G}_o$ can be constructed efficiently. We also discuss the heuristic rules used in $\mathcal{G}_o$ construction.

- We conduct extensive experiments on both real and synthetic datasets. The results show that the client side achieves significant cost saving in shortest distance computing with the aid of outsourced graphs compared to before. We confirm that our method scales well when it is used to answer approximate shortest distances within a specified error bound.

The remainder of this paper is organized as follows. In section 2, we formulate the transformation problem taking both security and utility into account. Section 3 and section 4 present methods to transform graphs with the exact and approximate distance answering respectively. Section 5 reports experimental results. Section 6 concludes the paper and discusses future works.

2. PROBLEM STATEMENT

In this section, we first define our graph-related notation and then discuss security issues for outsourced graphs and their utilization in shortest distance answering. Finally, we formulate our optimization problem.

2.1 Notation

Let $G = (V, E)$ be an edge-weighted undirected graph, where V and E are its vertex set and edge set, respectively. Each $e \in E$ takes the form of $e = (u, v)$, $u, v \in V$, and is associated with a weight denoted as $w(e)$ or $w(u, v)$ (All weights are assumed to be non-negative integers in a finite range). When the edge (u, v) does not exist, we can assume $w(u, v) = \infty$. A path is a sequence of edges $(u_0, u_1)(u_1, u_2) \dots (u_{x-1}, u_x)$, where $u_i \in V$ $(0 \leq i \leq x)$. The cost of a path p is the sum of edge weights in p, denoted as $len(p)$. A shortest distance query in graph G computes the minimal cost $\delta_G(u, v)$ of any path from u to v in G. Figure 2(a) shows a sample graph, which will be used as the running example in this paper. The character inside the circle represents the vertex's identifier, and the number annotated on the edge is its weight. The symbols used throughout the paper are listed in Table 1.

A graph pattern query can be used in re-identification over the transformed graph [14, 2, 19, 17, 11]. If an attacker has some knowledge about a fragment of the original graph, he can compose a graph pattern query to find exact matches over the transformed graph, and use the query results to infer other information. We illustrate a weighted graph pattern query in Fig. 2(c). Its result is the induced subgraph with the vertex set $\{i, l, v, p\}$ in G, since we can build a vertex mapping from the query pattern to this subgraph, under which the weight of each edge is the same as that of the mapped edge. Note, the result of the unweighted graph pattern in Fig. 2(b) contains 32 subgraphs.

2.2 Protecting Neighborhood Privacy

In this work, we target at protecting the sensitive local neighborhood information. Basically, the information of how an individual vertex links to its neighbors and what are the edge weights for these links are deemed sensitive and need to be protected. In particular, we focus on 1-neighborhood attacks since it tends to be more difficult to collect information beyond that [2]. In addition, when two vertices are very close to each other (within a threshold d), even without a direct link, their relationship can also be considered important. Formally, the outsourced graph should meet the requirement of the 1-*neighborhood-d-radius* graph.

DEFINITION 1. **1-Neighborhood-d-Radius Graph.** *Let $G = (V, E)$ be an original graph and $G_o = (V_o, E_o)$ an outsourced graph, $V_o \subseteq V$. For a vertex $u \in V$, its appearance in V_o can be denoted as $\overline{u}$. G_o is a 1-neighborhood-d-radius graph of G, if G_o meets the following conditions:*

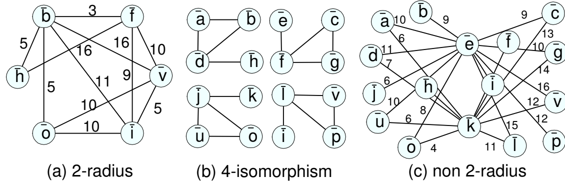
1. *for any vertex pair $\overline{u}$ and $\overline{v} \in V_o$, $(u, v) \notin E$. (1-neighborhood);*
2. *for any vertex pair $\overline{u}$ and $\overline{v} \in V_o$, $\delta_G(u, v) \geq d$. (d-radius).*

For brevity, we shorten the name to d-radius graph.

Simply speaking, we do not allow any two vertices, which are adjacent or with their distance smaller than d, to appear in the same d-radius graph. Figure 3(a) shows a 2-radius graph, $G_o(V_o, E_o)$, for $G(V, E)$ in Fig. 2(a). As shown in the figure, the vertex $\overline{u}$ in V_o is an appearance of a vertex u in V. Of the 16 vertices in G, 6 appear in G_o. For any two vertices $\overline{u}$ and $\overline{v}$ in V_o, u and v are not adjacent in the original graph, and the shortest distance $\delta_G(u, v) \geq 2$. We cannot add other vertices into G_o. For example, we cannot add $\overline{a}$ since $\delta_G(a, b) = 1$ is less than $d = 2$. Notice that the edges in G_o show the connections between vertices in a global manner, and the weight associated with an edge $(\overline{u}, \overline{v})$ is the shortest distance $\delta_G(u, v)$ in the underlying original graph G.

We show differences between our method and that used in a data publishing scenario. Figure 3(b) shows a 4-isomorphism result from a recent study [11]. It builds 4 disjoint, isomorphic subgraphs with graph splitting and structural anonymization. Given a graph pattern query as in Fig. 2(b), it can get results from the transformed graph in Fig. 3(b), but the number of matched subgraphs is 4. Our work supports edge weights. In addition, the vertices in a d-radius graph are a proper subset of those in the original graph, and no original edges are allowed in a d-radius graph. Therefore, the direct evaluation of any pattern query cannot find meaningful results, and a subsequent node re-identification is prevented. Most importantly, the operations to achieve structural anonymization in existing works [2, 13, 17, 11], including arbitrary addition/removal of edges and graph splitting, make the shortest distance computing over the transformed graph difficult or even impossible.

An interesting and important question is whether enforcing the d-radius property on each outsourced graph is too strict. For instance, can we simply remove all edges in the original graph and



(a) 2-radius (b) 4-isomorphism (c) non 2-radius

Figure 3: Transformed Graph for Outsourcing and Publishing

then only connect those pairs of vertices whose shortest distance is no smaller than d? The answer is negative. We demonstrate a successful attack on such as graph. Suppose an outsourced graph G_o is constructed by the removal of all direct edges and addition of edges from vertex u to $\overline{v}$ if $\delta_G(u, v) \geq 2$. We show the edges related with vertices $\overline{e}$ and $\overline{k}$ in G_o in Fig.3(c). This graph is a non-2-radius graph since two adjacent vertices e and k in the original graph co-exist in the same d-radius outsourced graph, which violates the first condition of a d-radius graph. Attackers can observe that the graph is strongly connected, thus, they can know that there is a direct edge (with any weight) or a path with cost no larger than 2 between e and k in the original graph. Based on the triangle inequality over G_o, attackers even infer that the edge weight is no smaller than 4!

2.3 Shortest Distance Computation using d-Radius Graph

We transform an original edge-weighted graph G into a set of outsourced d-radius graphs $\mathcal{G}_o = \{G_o^1, \ldots, G_o^{|\mathcal{G}_o|}\}$ which will be deployed on the cloud server, together with a link graph G_l on the client side. An edge in G_l takes the form of $(u, \overline{v})$, which maintains the relationship between vertices in G to an appearance of v in an outsourced graph. The edge can be also expressed in the form of $(u, G_o.\overline{v})$ to specify that the appearance $\overline{v}$ is in the outsourced graph G_o. The weight of an edge $(u, \overline{v})$, $w(u, \overline{v})$, in G_l is equal to $\delta_G(u, v)$. In particular, $w(u, \overline{u}) = 0$. We may use u, $\overline{u}$ and $G_o.\overline{u}$ interchangeably in the following.

Now, we give an important property to characterize the outsourced graphs $\mathcal{G}_o$ and the link graph G_l.

DEFINITION 2. ($\geq d$)***-Shortest Distance Equivalent Graph.*** *Let $G = (V, E)$ and $G' = (V', E')$ be two graphs, $V \subseteq V'$. G' is a $(\geq d)$-shortest distance equivalent graph of G if $\delta_G(u, v) = \delta_{G'}(u, v)$ for any vertex pair (u, v) with $\delta_G(u, v) \geq d$, $u, v \in V$.*

In order to use outsourced graphs $\mathcal{G}_o$ and the link graph $G_l = (V_l, E_l)$ to compute shortest distances, we require the union of $\mathcal{G}_o$ and G_l to be a $(\geq d)$*-shortest distance equivalent graph* of the original graph G. For graphs $G_0 = (V_0, E_0)$ and $G_1 = (V_1, E_1)$, their union result is $G = (V_0 \cup V_0, E_0 \cup E_1)$.

Given two vertices u and v in G, the outsourced graph $\mathcal{G}_o$ and the link graph $G_l = (V_l, E_l)$, the shortest distance can be computed as follows:

$$len = \min_{\substack{\overline{x}, \overline{y} \,\in\, V_o, G_o = (V_o, E_o) \in \mathcal{G}_o \\ (u, \overline{x}),\, (v, \overline{y}) \,\in\, E_l}} w(u, \overline{x}) + \delta_{G_o}(\overline{x}, \overline{y}) + w(\overline{y}, v)$$

$$\delta_G(u, v) = \min\{len, w\}, \text{where } w \text{ is } w(u, v) \text{ in } G_l \quad (1)$$

Notice that we only utilize the outsourced graphs and the link graph to compute the shortest distance from u to v with $\delta_G(u, v) \geq d$. For the shortest distance less than d, we can compute it in the original graph G. Thus, the complete shortest distance computing can be described as follows: *given two vertices u and v, Dijkstra's algorithm runs on the original graph to find whether u can reach v within d. If no path can be found, we begin to rewrite the distance query from u to v into multiple distance queries against the outsourced graphs $\mathcal{G}_o$. We locate $u.Edges$ for u's edges and $v.Edges$ for v's edges in the link graph G_l. For each pair of edges $e_u = (u, \overline{x}) \in u.Edges$ and $e_v = (v, \overline{y}) \in v.Edges$, if $\overline{x}$ and $\overline{y}$ are in the same outsourced graph $G_o \in \mathcal{G}_o$, a distance query from $\overline{x}$ to $\overline{y}$ is issued in G_o. We then combine the returned results from the outsourced server with the distance information ($w(u, \overline{x})$, $w(\overline{y}, v)$) in G_l to yield* len *in Equation (1). Due to the security reason or the optimization purpose, for two vertices with distance no less than d, the link graph G_l may materialize their relationship by directly linking them with an edge. Thus, the shortest distance is computed by choosing the minimum between* len *and $w(u, v)$ in G_l (if exists), as illustrated in Equation (1).*

2.4 Optimization Problem

To sum up, given a graph $G = (V, E)$ and d, the graph transformation produces outsourced graphs $\mathcal{G}_o = \{G_o^1, \ldots, G_o^{|\mathcal{G}_o|}\}$ and a local link graph G_l which achieve the following objectives:

1. Each outsourced graph $G_o \in \mathcal{G}_o$ is a d-radius graph;
2. The union of $\mathcal{G}_o$ and G_l is a $(\geq d)$-shortest distance equivalent graph of G;
3. The space cost of G_l and the cost of the shortest distance computation on the client side are minimized.

In the following sections, we study how to solve this problem efficiently.

3. GRAPH TRANSFORMATION WITH EXACT DISTANCE ANSWERING

In this section, we first give a naïve approach to show the problem complexity, then we give a greedy algorithm to transform a graph for exact shortest distance answering. Finally we analyze our method.

3.1 A Naïve Approach

We observe that the two optimization targets (in Objective 3) are along the same line with one another and not in conflict. For instance, if we minimize the space cost G_l, the computational cost over G_l tends to be minimized. In the following, we will focus on minimizing the space cost of G_l. In addition, as discussed above, for any pair of vertices (u, v) with distance less than d, its distance can be discovered on the original graph. Thus, only those vertex pairs whose distances are no less than d need to be considered in the transformation. Formally, our graph transformation can be reduced to a problem on minimizing G_l as follows:

DEFINITION 3. ***Minimizing*** G_l***.*** *Given a graph $G = (V, E)$ and d, we seek a set of d-radius graphs $\mathcal{G}_o$ from G and a link graph G_l such that for each pair of vertices (u, v) in graph $G = (V, E)$ with $\delta_G(u, v) \geq d$,*

1. *there exists an outsourced graph $G_o \in \mathcal{G}_o$, $(u, G_o.\overline{x}) \in E_l$ and $(G_o.\overline{y}, v) \in E_l$, where $\delta_G(u, v) = \delta_G(u, x) + \delta_G(x, y) + \delta_G(y, v)$;*
2. *or there exists an edge $(u, v) \in E_l$ in G_l.*

Our objective is to minimize G_l, or the number of edges in G_l.

In order to solve this problem (minimizing G_l), we may consider a straightforward brute-force approach. Basically, we can try to enumerate all candidate configurations, where each candidate configuration consists of a set of outsourced graphs and a link graph, and can answer any distance query from u to v with $\delta_G(u, v) \geq d$ (Equation (1)); then we compute the space cost of G_l in each candidate configuration. Finally, the optimal solution is the configuration (a set of outsourced graphs with a link graph) with the minimal G_l.

Now, let us look at the number of candidate configurations in the brute-force approach. We note that the vertices in an outsourced graph are actually the sub-set of the vertices in the original graph. Although not all sub-sets of vertices of original graph can produce valid d-radius graphs, the total number of different outsourced graphs (d-radius graphs) can still be $O(2^n)$ in the worst case. In addition, the maximal vertices in the shortest path from u to v will be $O(n)$ in the worst case, which indicates that each outsourced graph from the total $O(2^n)$ ones has the potential to be used in the distance preserving for (u, v). Based on these factors, we can observe the total number of different configurations is exponential in terms of the number of vertices in the original graph. This clearly makes the brute-force approach too expensive to be feasible.

We can also observe the relationship between the minimizing G_l problem and the set cover problem [?]. The set cover problem is described as: *given a ground set U, and a candidate family $\mathcal{S}$ consisting of the subsets of U, the goal is to find the minimal number of candidate sets, denoted as $\mathcal{C} \subseteq \mathcal{S}$, whose union is U*. We can transform an outsourced graph G_o into a candidate set S in $\mathcal{S}$ and transform a vertex pair p into an element e in the ground set U. Specifically, the candidate set S consists of all vertex pairs whose distance can be answered via G_o. Given this, we may be inclined to adopt a set-cover approach [?] to solve our problem. However, the difficulty is that in our problem, the outsourced graphs and consequently the candidate sets are not given in advance. And the vertices in an outsourced graph must satisfy the d-radius constraint. Moreover, the optimization target in our problem is to minimize the space cost of G_l, which is hard to be coded as the goal in the set cover problem.

3.2 Fast Greedy Method

The naïve solution above reveals the massive search space in the optimal solution to our graph transformation problem. In this part, we design a fast greedy method to produce a reasonable graph transformation plan.

Basic Idea. The main problem with the naïve approach lies in the fact that we have to enumerate all possible outsourced graphs. To avoid producing all outsourced graphs, we wish to construct only the needed ones. Intuitively, the outsourced graph which can answer more distance queries will be constructed first, since in such a case, the edges in the link graph have higher chances to be reused and thus the space cost of the link graph can be reduced. This idea is similar to the greedy idea used in the set cover problem, which selects the sub-set with the maximal number of elements first.

We then make a heuristic restriction on edges in G_l in order to reduce the search space for the outsourced graphs. Recalling Equation (1), the shortest distance from u to v can be computed via a sub-path from $\overline{x}$ to $\overline{y}$ in an outsourced d-radius graph. Taking the sub-path from u to $\overline{x}$ as an example, we require either the distance from u to x to be less than a threshold or u to be adjacent to x in the original graph. This restriction can rule out the cases where edges in the link graph have large weights. Note that the restriction does not contradict our above intuitions. A sub-path from u to x which can be used in more distance queries always has a lower weight. For simplicity, the threshold on the maximal edge weight

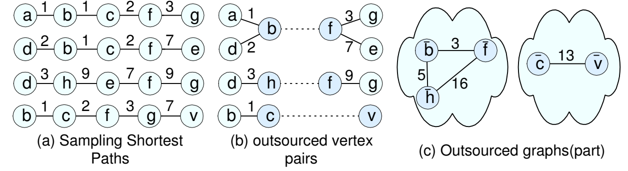


Figure 4: Outsourced Graph Generation

in the local graph has the same setting as d in d-radius graph in the following.

Under the restriction, we give the following notation on the candidate outsourced vertex pairs. The first condition is the same as Equation (1), which means the shortest distance from u to v can be answered via x and y. The second condition is the restriction on edges in the link graph. The third condition corresponds to the requirements of d-radius.

DEFINITION 4. ***Candidate Outsourced Vertex Pair.*** *Given a pair of vertices (u, v) in graph $G = (V, E)$ with $\delta_G(u, v) \geq d$ and $(u, v) \notin E$, if there exists another pair of vertices (x, y):*

1. $\delta_G(u, v) = \delta_G(u, x) + \delta_G(x, y) + \delta_G(y, v)$;
2. $\delta_G(u, x) < d$ *or* $(u, x) \in E$, $\delta_G(y, v) < d$ *or* $(y, v) \in E$;
3. $\delta_G(x, y) \geq d$ *and* $(x, y) \notin E$.

then (x, y) is the candidate outsourced vertex pair for pair (u, v).

We illustrate candidate outsourced vertex pairs for the shortest paths in Fig.4(a). Take the shortest path between node a and node g as an example. For d=2, the candidate outsourced vertex pairs include (a, f), (a, g), (b, f), and (b, g). We also note that some vertex pairs, such as (b, f) in Fig.4(b), lie on the shortest paths between multiple vertex pairs and thus can be used to answer multiple distance queries. In order to make the outsourced graph answer more distance queries, we prefer to select such vertex pairs for the outsourced graph. Also, note that $(\overline{c}, \overline{v})$ cannot be put in the d-radius graph which already contains $(\overline{b}, \overline{f})$, since $\delta_G(b, c) < 2$. In such a case, we have to put them into different outsourced graphs.

Another problem with the naïve approach is the large intermediate space required. We need to preserve the distance for $O(n^2)$ vertex pairs in the graph transformation. On a relatively large graph, the representation of these vertex pairs alone will easily exceed the memory limit. Thus, we need to extend the greedy framework to use a pipeline approach to generate the outsourced graphs and their link graph. That is, we iteratively enumerate vertex pairs until the number of vertex pairs reaches a threshold, and then use these partial vertex pairs to guide the construction of one outsourced graph. Once the outsourced graph is constructed, the vertex pairs which have been preserved will be removed from memory. Then we load a new set of vertex pairs for the next outsourced graph, until all distances between vertex pairs have been preserved.

Greedy Graph Transformation. We present our algorithm in Algorithm 1. We initialize the shortest path set, outsourced graphs and the link graph in line 1. Then we attempt to preserve the distances of these shortest paths with newly constructed outsourced graphs. In the iteration from line 2 to line 11, when the total paths in memory exceed *MPMem* in line 5, we build a vertex sequence L with a benefit function and invoke Algorithm 2 to generate an outsourced graph according to L in line 7. *MPMem* is used to control the maximal space cost used in the graph transformation. The paths which have been preserved are removed from P in line 10.

Algorithm 1: Greedy Graph Transformation

Input: graph $G = (V, E)$, d, *MPMem* for threshold on maximal paths in memory
Output: outsourced graphs $\mathcal{G}_o$ and link graph G_l.
1 Initialize an empty shortest path set P, $\mathcal{G}_o$, and G_l;
2 **while** *there remains shortest paths not handled* **do**
3 Locate a remaining shortest path p with $len(p) \geq d$, add p into P;
4 Enumerate all candidate outsourced vertex pairs for p;
5 **if** $sizeof(P) >$ *MPMem* **then**
6   Build a vertex sequence L with the pair based benefit function;
7   $G_o \leftarrow OutGraph(G, d, L)$;
8   $\mathcal{G}_o \leftarrow \mathcal{G}_o \cup G_o$;
9   Build edges in G_l from vertices in G to these in G_o;
10   Remove the paths from P exactly answered by G_o;
11   Re-enumerate candidate vertex pairs in P;
12 $removed \leftarrow \infty$;
13 **while** $removed > |G_o|$ **do**
14 Generate G_o as line 6 to line 11;
15 $removed \leftarrow$ the number of removed paths by G_o;
16 For each $p \in P$, build an edge e with ending vertices of p, and add e into G_l;
17 return $\mathcal{G}_o$ and G_l.

Now we discuss the benefit function used in the greedy method. For each shortest path, we enumerate all possible candidate outsourced vertex pairs. We have two kinds of benefit functions. The first function is based on the vertex frequency. That is, we simply count the occurrences of x and y for each candidate outsourced vertex pair (x, y) separately. The second one is based on vertex pair frequency. We record the frequency of all possible vertex pairs, and sort vertex pairs in terms of their frequency. Note that L may contain duplicate vertices in such a case. Since two vertices x and y having high frequencies does not imply that (x, y) can be used to compute more shortest distance queries, Algorithm 1 applies the vertex pair based benefit function in line 6. Our experimental results show that the latter function works better than the former.

As for the termination condition of the greedy method, we can continue generating new outsourced graphs and pruning path set P until P is empty. The algorithm will stop since each outsourced graph can be used to preserve at least one remaining distance. However, when the iteration from line 2 to line 11 stops, those vertex pairs whose distances can be "easily" answered by outsourced graphs have been removed. In order to reduce the transformation cost, we can terminate the construction of outsourced graphs when the number of nodes in a newly generated outsourced graph G_o is larger than the number of remaining vertex pairs whose distances can be answered via G_o. Since the remaining paths are recorded in G_l in line 16, the union of $\mathcal{G}$ and G_l is still a $(\geq d)$-shortest distance equivalent graph of the original graph.

Discovering Single Outsourced Graph. The next key problem is how to use the heuristic information collected from the greedy method to guide the construction of a outsourced graph. We now show that each d-radius graph can be easily constructed from a distance aware cluster cover.

DEFINITION 5. ***Distance Aware Cluster Cover.*** *Let $G = (V, E)$ be an original graph. A cluster cover $\mathcal{C}$ is a set of clusters, each cluster $C_d(x) \in \mathcal{C}$ having a center $x \in V$. A distance aware cluster cover meets the following requirements:*

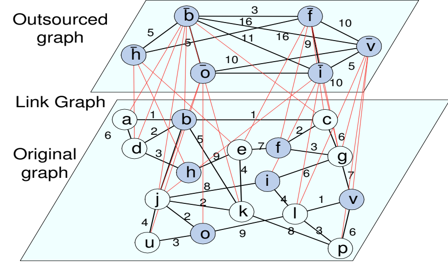


Figure 5: Link Graph and One Outsourced Graph

1. *For any vertex $u \in V$, u belongs to at least one cluster $C_d(x)$ where u is directly connected to x or $\delta_G(u, x) < d$;*

2. *For any two clusters $C_d(x)$ and $C_d(y)$, $\delta_G(x, y) \geq d$ and $(x, y) \notin E$.*

Intuitively, the centers of clusters can be used as the outsourced vertices in a d-radius graph. For example, the vertices in Fig.3(a) are the cluster centers in a cluster cover. For a vertex u in a cluster $C_d(x)$, we build an edge from u to its cluster center x, and such an edge is actually in the link graph G_l. Given a graph G, d, and a vertex sequence L which encodes the heuristic knowledge, an outsourced graph can be constructed efficiently with Algorithm 2.

Algorithm 2: $OutGraph(G, d, L)$

Input: graph $G = (V, E)$, d, a vertex sequence L.
Output: Outsourced Graph $G_o = (V_o, E_o)$.
1 Initialize empty $G_o = (V_o, E_o)$;
2 **while** *there exists uncovered vertex in L* **do**
3 Pick the top uncovered vertex x from L as a cluster center, and $V_o \leftarrow V_o \cup \{x\}$;
4 Create a cluster $C_d(x)$ containing 1-neighbor of x and vertex u with $\delta_G(u, x) < d$ with Dijkstra's search, and mark them *covered*;
5 For any two cluster centers x and y, build the necessary edge between them;
6 Return G_o.

In Algorithm 2, we select the first uncovered vertex from L as the cluster center and build its cluster, since L is sorted in terms of node benefit and the vertices closer to the head of L have higher benefits. The cluster centers will be the vertices in the outsourced graph G_o. The other vertices in a cluster can be discovered with Dijkstra's algorithm and are labeled as *covered* so they will not be selected as cluster centers . Notice that this strategy also ensures that the minimal distance between cluster centers is no smaller than d. In line 5, we build edges between any two vertices in the outsourced graph. The edge weight equals their shortest distance in the original graph discovered by Dijkstra's algorithm. Given three vertices x, y, and z, if $\delta_G(x, y) + \delta_G(y, z) = \delta_G(x, z)$, the edge from x and z need not be constructed.

Figure 5 shows one outsourced graph and its link graph produced by Algorithm 1 on the graph (Fig.2(a)). The outsourced graph is the same as Fig.3(a). It can be constructed by Algorithm 2. Lines with red color are the edges in the link graph G_l. In order to make the figure clear, we omit the weight of these edges.

3.3 Analysis of Graph Transformation

In this part, we analyze the time cost and correctness of the graph transformation algorithm, the impact of d, and the overhead distribution between the cloud server and client side. We use the following symbols. The meanings of n_o, n_l, n, m are given in Table 1. x is the total number of outsourced graphs, which is related to d and the graph's features. b is the maximal number of a vertex's edges to an outsourced graph.

The time cost of graph transformation in Algorithm 1 includes the enumeration of all shortest paths, the sorting of candidate outsourced vertex pairs, and the generation of all outsourced graphs. The enumeration of all shortest paths requires $O(n(m + n \log n))$. Since we need to sort at most $O(n^2)$ candidate outsourced vertex pairs in the benefit computation before the construction of each of x outsourced graphs, the total sorting cost is $O(xn^2 \log n)$. The single outsourced graph generation in Algorithm 2 includes the vertex selection cost and edge building cost. The vertex selection requires $O(n_o n_l^2)$, since there are $O(n_o)$ clusters and each requires $O(n_l^2)$ for local shortest path discovery. In the edge building for an outsourced graph, we need $O(n_o)$ times shortest distance discovery in the original graph in the worst case. Thus, a single outsourced graph construction takes $O(n_o(m+n \log n) + n_o n_l^2)$=$O(n_o(m+n \log n))$. With all factors considered, the total time cost of graph transformation in Algorithm 1 is $O(n(m+n \log n) + xn^2 \log n + xn_o(m + n \log n))$. A minor extension to Algorithm 1 is to cache the computed shortest distances which can be used in the edge building in outsourced graphs. In this way, the time complexity can be reduced to $O(n(m + n \log n) + xn^2 \log n)$.

Now, we show that Equation (1) can yield correct shortest distances over the union of $\mathcal{G}_o$ and G_l produced by Algorithm 1. Algorithm 1 has enumerated all shortest paths P with their distances no smaller than d. From the construction rules, we know that a shortest path $p \in P$ can be removed from P only when p can be exactly answered by one outsourced graph in $\mathcal{G}_o$ or p is stored in the link graph G_l. Therefore, the union of $\mathcal{G}_o$ and G_l from Algorithm 1 is a $(\geq d)$-shortest distance equivalent graph of the original graph. At the same time, Equation (1) enumerates all possible paths from u to v using $\mathcal{G}_o$ and G_l. Therefore, Equation (1) can produce the correct results.

From the above discussion, we can notice that d is a key factor to adjust the security strength and overhead on the client side. A larger d leads to fewer outsourced vertices, which indicates more information is hidden in a "coarser" outsourced graph. At the same time, a larger d results in more vertices in one cluster in the cluster cover, which shows that the client side requires more space cost to store the link graph and more time cost in the local path searching. In addition, since a single "coarser" outsourced graph preserves fewer shortest distances than a "finer" one, it needs more outsourced graphs along with the link graph to meet the requirement of $(\geq d)$-shortest distance equivalent graph, which also results in a higher graph transformation time cost.

	Client Side	Cloud Server
Space	$O(m + xn_o n_l)$	$O(xn_o^2)$
Query Time	$O(n_l^2 + xb^2)$	$O(xb^2 n_o^2)$

Table 2: Overhead Distribution with Graph Outsourcing

Now we discuss the overhead distribution after graphs are outsourced in Table 2. As for the space cost, the client side needs $O(m+xn_o n_l)$ to store the original graph and the link graph, while the cloud server requires $O(xn_o^2)$ space to store all outsourced graphs. As for the time cost of shortest distance query answering, the client side needs time $O(n_l^2)$ for the local search and time $O(xb^2)$ for result merging with Equation (1). The cloud server takes $O(xb^2 n_o^2)$ time to compute the rewritten queries. Note that the cloud server can build indices [21, 1, 7] and perform parallel processing over x graphs to lower the query evaluation cost significantly. Compared with $O(m + n \log n)$ time cost without graph outsourcing, the client side saves much time cost with the aid of the cloud server. The final experimental results also show the effectiveness of graph outsourcing.

4. GRAPH TRANSFORMATION WITH APPROXIMATE DISTANCE ANSWERING

In this section, we relax the 2-nd objective on $(\geq d)$ equivalent shortest distance graph to handle large graphs. We first propose a method to transform graph with approximate distance answering, and then discuss heuristic methods in the outsourced graph construction.

4.1 Average Additive Error Guided Graph Transformation

Although the graph transformation method discussed above can answer shortest distances exactly, the transformation requires enumerating all shortest paths and computing the edges inside outsourced graphs, which makes the method unsuitable for large graphs.

Graph transformation with approximate distance answering on large graphs thus becomes an important research problem, since approximate shortest distances are good enough in many applications [22, 12, 21]. This then raises the issue of how many outsourced graphs are needed to achieve good distance values, if we use random vertex sequences for Algorithm 2.

The quality of the approximate distance from u to v can be measured by $\alpha\delta_G(u, v) + \beta$ [22], where α is the multiplicative error, and β is the additive error. There are many studies on approximate distance answering. A pre-computed data structure is proposed to achieve α in $[1, 2k-1]$, where k can adjust the space and time cost in pre-computation [22]. Kleinberg *et al.* achieve $\alpha = 1 + \phi$ and $\beta = 0$, given enough random beacons and the triangle inequality rule [12].

In this part, we attempt to achieve $\alpha = 1$ and a given average additive error $\overline{\beta}$ for all shortest distance queries. For any distance query $q \in Q$ from u and v, a path p_q is discovered for q using $\mathcal{G}_o$ and G_l. Average additive error $\overline{\beta}$ can be defined as $\frac{\sum_{q \in Q} \beta_q}{|Q|}$, where β_q = $len(p_q) - \delta_G(u, v)$. The rationale of averaged addition error is to get acceptable results with a limited number of outsourced graphs. The average additive error can be useful when a large number of shortest distance queries are evaluated in graph analysis. In addition, since our outsourced graph is generated randomly, the worst case of additive error can be lowered along with the average one simultaneously with a high possibility.

We present our graph transformation method in Algorithm 3 to achieve the given additive error $\overline{\beta}$. The basic idea is to repeatedly construct randomized outsourced graphs until the estimated additive error avg is less than $\overline{\beta}$. The estimated additive error avg is initialized in line 2 and is adjusted after a new outsourced graph is constructed in the iteration from line 3 to line 10.

In line 4, we build an outsourced graph with Algorithm 2. Since the exact edge weight computation between outsourced vertices takes $O(n_o)$ times Dijkstra's search, where n_o is the number of vertices in an outsourced graph, we relax the exact edge building as follows. We select l vertices from n_o total outsourced vertices, and we build the full shortest path trees for these l vertices in the original graph with Dijkstra's algorithm. The shortest path tree can

Algorithm 3: Average Additive Error Guided Outsourced Graph Construction

Input: Graph G, d, additive error threshold $\overline{\beta}$, s for the number of sampling queries, l for the number of full shortest path trees.
Output: Outsourced Graphs $\mathcal{G}_o$.
1 Initialize three empty query lists Q_0, Q_1 and Q_2 with length s;
2 $avg \leftarrow \infty$;
3 **while** $avg > \overline{\beta}$ **do**
4   $G_o \leftarrow OutGraph(G, d, L)$ with modified edge building based on l full shortest path trees;
5   $\mathcal{G}_o \leftarrow \mathcal{G}_o \cup \{G_o\}$;
6   Collect the average number n_o of vertices in G_o and the average number n_l of vertices in cluster;
7   Remove queries from Q_0 and Q_1 which do not meet the requirement of Q_0 and Q_1;
8   Add new queries into Q_0 and Q_1 with total s queries;
9   Evaluate queries over G with Dijkstra's algorithm and Equation (1) to obtain l_0, l_1 and l_2;
10   $avg \leftarrow pct(Q_0)l_0$+$pct(Q_1)l_1$+$pct(Q_2)l_2$;
11 Return $\mathcal{G}_o$.

ensure that the path in the tree is also the shortest path in the original graph. Then, we build edges for any two outsourced vertices $\overline{x}$ and $\overline{y}$ when $\overline{x}$ is the lowest ancestor of $\overline{y}$ in the shortest path tree. We observe that our relaxed edge building method is similar to that of the landmark index [21]. However, the landmark index only records relationships from vertices to the root of the shortest path tree. Hence, the outsourced graph with this relaxed edge building strategy can yield results more precisely than the landmark index when using the same number of shortest path trees.

When outsourced graphs $\mathcal{G}_o$ are generated randomly and the relaxed edge building strategy is used, the result of Equation (1) may be not the shortest distance. Given two vertices u and v, the additive error of distance computed by Equation (1) comes from the error in the shortest distance between cluster centers in the outsourced graph (due to relaxed edge building strategy) and the deviation of their cluster centers from their shortest path.

The next key problem is how to estimate the average additive error avg from existing outsourced graphs. In order to compute avg more precisely, we put a shortest distance query from u to v into one of 4 categories, namely Q_l, Q_0, Q_1, and Q_2, according to the relationships from u and v to current outsourced graphs $\mathcal{G}_o$. Then avg can be calculated with the proportion and average additive error of each category. Specifically, if the shortest distance $\delta_G(u, v)$ is smaller than d, q belongs to Q_l. q can be answered exactly with local Dijkstra's search in the original graph. If neither u or v has been outsourced into any $G_o \in \mathcal{G}_o$, q is in Q_0, as illustrated in Fig.6(a). Even if the outsourced graph returns the correct distance between x and y, where x is the cluster center for u and y is the cluster center for v, the maximal additive error for a query in Q_0 is $4d_m$, where $d_m = max(d, w_{max})$, and w_{max} is the maximal edge weight. If either u or v has been selected into an outsourced $G_o \in \mathcal{G}_o$, q belongs to Q_1, as illustrated in Fig.6(b). Suppose that the outsourced graph returns the correct distance, the maximal additive error for a query in Q_1 is $2d_m$. If both u and v are selected in one $G_o \in \mathcal{G}_o$, q is in Q_2, as illustrated in Fig.6(c). The additive error of q now comes from the error in the distance computation in outsourced graphs. From the above discussion, we know that the additive error for a query in Q_l is 0, and the average additive error for queries in Q_0 is larger than that for queries in Q_1, which is larger than that for queries in Q_2.

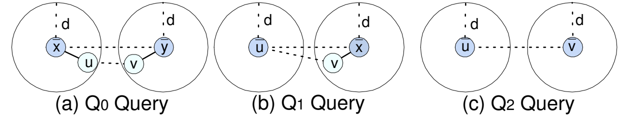


Figure 6: Query Categories

The proportion of each query category can be computed with two statistics. The average number of vertices in a cluster, n_l, and the average number of vertices in an outsourced graph, n_o, are collected in line 6 in Algorithm 3. Suppose that the total number of outsourced graphs is x. The proportion $pct(Q_l)$ for Q_l is $\frac{n_l}{|V|}$. The increase of x does not affect $pct(Q_l)$. The proportion $pct(Q_0)$ for Q_0 is $(1 - \frac{n_0}{|V|})^{2x}(1 - \frac{n_l}{|V|})$. A vertex is not selected as the cluster center for outsourcing under the probability $(1 - \frac{n_0}{|V|})$. $pct(Q_0)$ reveals the probability that neither of two vertices is selected after x times. As the number of outsourced graphs increases, $pct(Q_0)$ decreases dramatically. The proportion $pct(Q_2)$ for Q_2 is $(1-(1-\frac{n_0}{|V|}^2)^x)(1-\frac{n_l}{|V|})$, where $(1-\frac{n_0}{|V|}^2)$ is the probability that two vertices are not selected as cluster centers simultaneously. $pct(Q_2)$ increases with the increase of the number of outsourced graphs. With three of the four proportions defined, the proportion for Q_1 is simply the remainder: $pct(Q_1)$=1-$pct(Q_l)$-$pct(Q_0)$-$pct(Q_2)$.

Now, we compute l_0, l_1, and l_2, which are the average additive errors for queries in Q_0, Q_1, and Q_2 respectively. It is not a trivial task, since these values are related to the local structure inside the cluster and the global structure between clusters. We can know that these additive distance errors decrease monotonically with the increase of outsourced graphs. In this paper, we employ a sampling method to estimate these additive distance errors. That is, we randomly build three query lists for query categories Q_0, Q_1, and Q_2. These query lists are initialized in line 1, and the queries in the list for Q_0 and Q_1 may be adjusted after a new outsourced graph is generated in line 7 and line 8. l_0, l_1, l_2 are then computed as the average difference between the exact shortest distances from Dijkstra's algorithm and the distance with Equation (1) for all queries in corresponding query categories.

Finally, the average additive error can be computed in line 10 with

$$pct(Q_0)l_0 + pct(Q_1)l_1 + pct(Q_2)l_2, \tag{2}$$

where l_0, l_1, and l_2 are averaged additive errors for queries in Q_0, Q_1, and Q_2 respectively. We do not mention the query category Q_l since the additive error for the query in Q_l is 0. One may wonder why we cannot produce the average additive error directly from the sampling queries. This is due to the fact that the proportions of query categories are changed with more outsourced graphs. In addition, $l_2 < l_1$, $l_1 < l_0$. Thus, Equation (2) can estimate the additive error more precisely.

The time cost of the client side in Algorithm 3 mainly consists of the evaluation cost of sampling queries and the generation cost of outsourced graphs. Let n, m, n_l, n_o have the meanings illustrated in Table 1. l is the number of full shortest path trees constructed for each outsourced graph. x is the number of outsourced graphs, which is related with d, the additive error bound $\overline{\beta}$, and the graph's features. s is the number of sampling queries. The evaluation of sampling queries takes $O(xs(m + n\log n))$ time cost in the worst case, and the outsourced graph generation takes

$O(x(l(m+n\log n)+n_o n_l^2))$ time cost, where $O(l(m+n\log n))$ is the cost for l times Dijkstra's algorithm in edge building, and $O(n_o n_l^2)$ is the cost in outsourced vertex selection. Then the time complexity in Algorithm 3 is $O(x(t(m+n\log n)+n_o n_l^2))$, where $t = max(s, l)$.

4.2 Heuristic Construction Rules

Algorithm 3 achieves the desired additive error with outsourced graphs constructed randomly. Another extension is to introduce heuristic rules in the outsourced graph construction. From Equation (1), it is feasible for outsourced graphs to produce more precise results when they contain more shared shortest sub-paths. Existing work on landmark indices also shows that the heuristic rules work better than the random ones [21].

The basic idea behind heuristic construction is to make it more probable for a vertex to be selected as a cluster center if it is located on more shortest paths. Thus more shortest distance queries can be answered via outsourced graphs generally. In our paper, we design two heuristic rules. The degree based outsourced graph construction attempts to select vertices with the higher degree as the cluster centers; the cluster based method selects the vertex x with the largest number of vertices in cluster $C_d(x)$ each time. In order to make the outsourced graph construction in Algorithm 2 be aware of these heuristic values, we sort the vertices in sequence L with the heuristic values, since the vertices nearer to the top of L have more chances to be outsourced in Algorithm 2.

The duplication of vertices in different outsourced graphs is another important concern in the heuristic construction method. The same vertex sequence will produce the same outsourced graph using Algorithm 2. In order to avoid constructing duplicate graphs, we make an extension to the outsourced graph construction method in Algorithm 2 with an introduction of a percentage $k(0 < k < 1)$ and a function $f(x)$, where x is the number of outsourced graphs, $f(1) = k$ and $f(i) < f(i + 1)$. In the first outsourced graph construction, rather than always choosing the vertex with the maximal benefit value among the uncovered ones in vertex sequence L, we select a vertex randomly from uncovered vertices with top-k benefit values. In the following x-th outsourced graph generation, k is enlarged by $f(x)$ such as $f(x + 1) = 2f(x)$ until $k \geq 1$. Thus after multiple rounds of outsourced graph construction, the heuristic values have been extensively exploited, and the outsourced graph construction retreats to the random method, which focuses on the distribution of the outsourced vertices in outsourced graphs.

5. EXPERIMENTAL RESULTS

In this section, we implement the graph transformation with the exact and approximate distance answering, and conduct extensive experiments on both real and synthetic datasets.

5.1 Experimental Setup

Measures. We focus on the following measures related to the graph transformation: the transformation time cost, the space cost of the link graph $|G_l|$ (the number of edges in G_l), and the average additive error achieved by outsourced graphs. In addition, in order to show the effectiveness of graph outsourcing in shortest distance computing, we define a local overhead ratio $r_l = t_l/t_f$, where t_l is the time cost to discover the shortest distance with Equation (1) on the client side, and t_f is the time cost used by Dijkstra's algorithm on the client side.

Implementation Details and Competitors. We implement our methods in Java with JDK 1.6. The maximal runtime memory of JVM is set to 1.55GB. All experiments are carried out on 1.8 GHz AMD processor running Windows Server 2003. The shortest distance computation in the outsourced server is simulated in the same machine. When one outsourced graph is constructed, we store it into a relational database so we can support multiple outsourced graphs. The time cost in accessing the relational database is not included in the time reported.

In addition, we implement the edge anonymization method with all-pair shortest path preserving (denoted as *LP*) in the technical report version for [24]. As discussed above, *LP* only anonymizes edge weights, and then the transformed graph cannot counter the structural attack. Here, we want to compare their work in terms of the graph transformation time cost, the space cost and the local overhead ratio. Just as in [24], we also use LPsolver 5.5 [1] to solve the rules generated.

Datasets. We use five graph datasets to test our methods, including three real graphs named *Gnut08*, *DBLP* and *Bay*, and two synthetic graphs named *Random* and *Power*.

Gnut08[2] is a directed gnutella P2P network. *DBLP* is extracted from a recent snapshot of DBLP dataset[3]. We select the records after 2004. *Bay* dataset[4] describes the road network in the San Francisco Bay Area. The edge weights in *Gnut08* and *DBLP* graph are assigned randomly in the range [1,100]. Each edge weight in Bay is divided by 30 so that the average edge weight 54 in Bay is similar to that of the other graphs.

Dataset	# of Vertices	# of Edges
Gnut08	6,301	20,777
DBLP5k	5,000	20,663
DBLP200k	200,000	847,433
Bay	321,270	400,086
Random$x$kNyd	1k-100k	yk-100yk
Power$x$kNyd	50k-200k	50yk-200yk

Table 3: Statistics of Graph Datasets

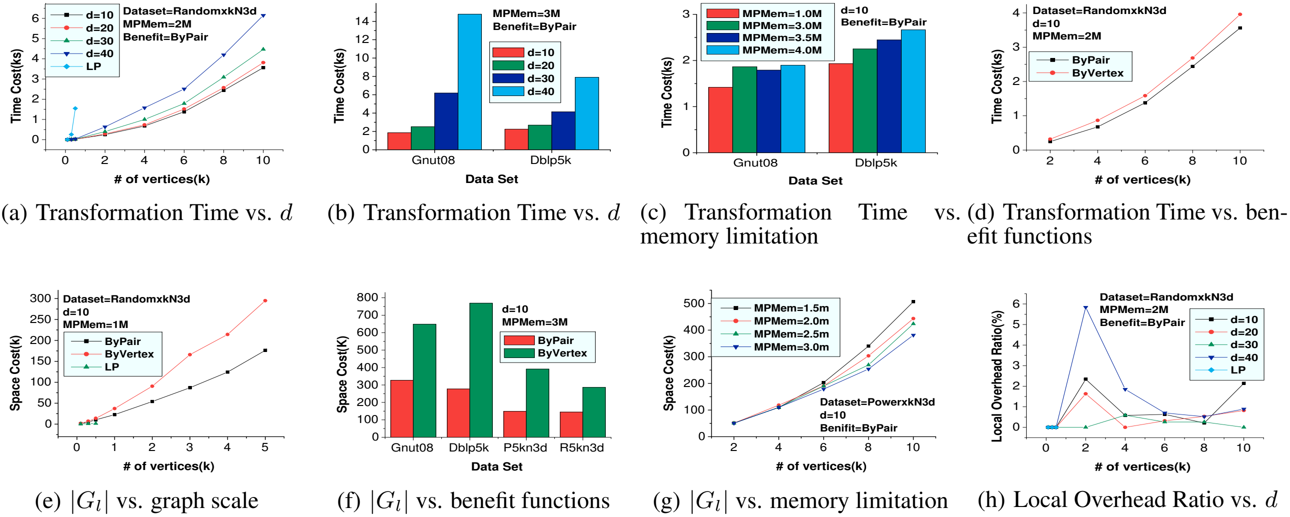
The *Random* dataset of graphs are generated as follows. Let n and m be the number of vertices and edges respectively, we randomly select the source and target vertex for m times among n vertices. The *Power* graph set is generated using *Barabasi Graph Generator v1.4*[5]. It can create graphs in which the distribution of outdegrees obeys a power law. The weights of edges in *Random* and *Power* graph are assigned randomly in [1,100].

Some statistics about these graphs are summarized in Table 3. Our synthetic graphs have the suffix xNyd, where x is the number of vertices and y is the average degree. For example, *Random100kN3d* represents a *Random* graph with 100k vertices and an average degree of 3.

5.2 Graph Transformation with Exact Answer

In this section, we study the impact of different factors on the measures of graph transformation with exact shortest distance answering. Specifically, we are interested in: i) What is the impact of d on the transformation overhead, the size of the link graph and local overhead ratio? ii) We have discussed two benefit functions. Can the vertex-pair based function *ByPair* reduce the overhead compared to the vertex based function *ByVertex*? iii) Does

[1] http://lpsolve.sourceforge.net/5.5/
[2] http://snap.stanford.edu/data/p2p-Gnutella08.html
[3] http://dblp.uni-trier.de/xml/
[4] http://www.dis.uniroma1.it/ challenge9/data/
[5] http://www.cs.ucr.edu/ ddreier/barabasi.html



(a) Transformation Time vs. d (b) Transformation Time vs. d (c) Transformation Time vs. memory limitation (d) Transformation Time vs. benefit functions

(e) $|G_l|$ vs. graph scale (f) $|G_l|$ vs. benefit functions (g) $|G_l|$ vs. memory limitation (h) Local Overhead Ratio vs. d

Figure 7: Experimental Results on Graph Transformation with Exact Answers

the threshold *MPMem* on maximal paths in memory in Algorithm 1 impact on the effectiveness of the greedy method?

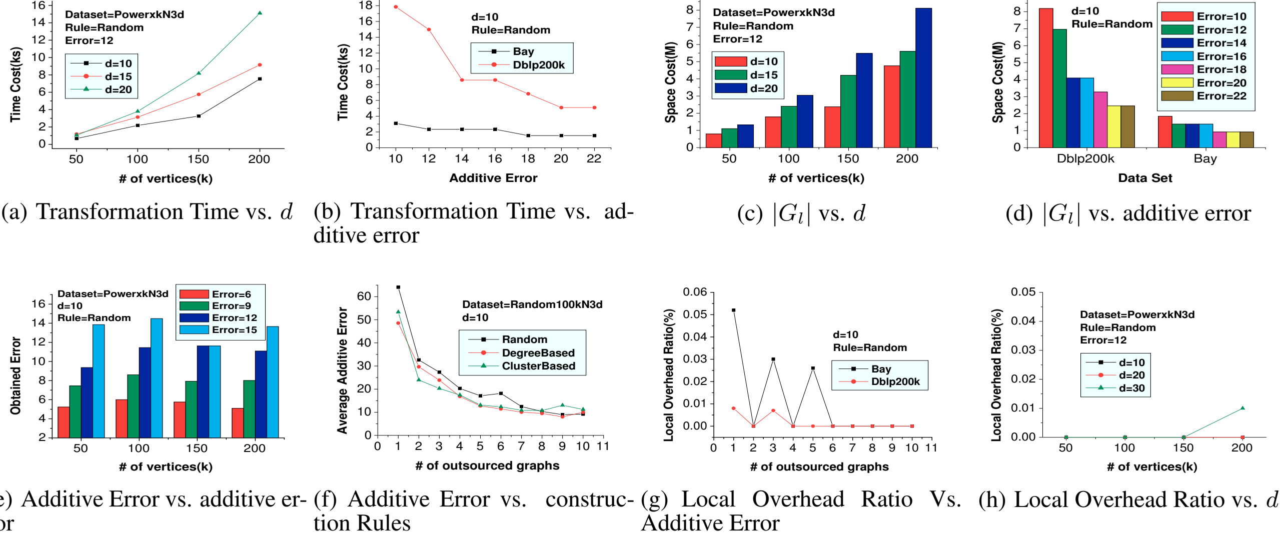
Transformation Time Cost. Figure 7(a) and Figure 7(b) show the transformation time cost with respect to d on *Random* graphs and real data respectively. The graph data used in Fig.7(a) is *Random$x$kN3d*, where x is the number of vertices in X-axis. We observe that the time cost of graph transformation also increases with the increase of d. The main reason lies in the fact that when d is larger, there are fewer vertices in each d-radius outsourced graph, and thus more outsourced graphs are required to achieve $(\geq d)$-shortest distance equivalent graph. We also find *LP* method does not scale well. *LP* produces $O(dn^2)$ rules for a graph to achieve all-pairs shortest path preservation, where d is the average degree, and n is the number of vertices. For example, a *Random* graph with only 600 vertices and average degree 3 generates 1,071,608 rules. LPSolver cannot handle this many rules and terminates. On smaller graphs with the number of vertices less than 500, we also notice that the time used by *LP* rises rapidly.

Figure 7(c) studies the impact of the threshold *MPMem* (maximal paths in memory) on the transformation time cost over two real graphs. Our algorithm introduces *MPMem* to control the total intermediate space used in the greedy method. Intuitively, the greedy method can produce more reasonable outsourced graphs based on more shortest paths with a larger *MPMem*. In addition, when the number of shortest paths exceeds *MPMem*, another new outsourced graph has to be generated inevitably. Thus, the increase of *MPMem* will reduce the graph transformation time cost in general. We also notice that when *MPMem* is large enough, its increase is not effective any more, since all computed shortest paths can be put into the memory in such a case.

Figure 7(d) compares the effects of the two benefit functions, *ByVertex* and *ByPair*, on the transformation time cost on *Random* graphs. As discussed before, the case that two vertices x and y have a higher frequency does not indicate that the vertex pair (x, y) can answer more shortest distances. Thus, *ByPair* method is a more reasonable function which can produce fewer outsourced graphs and then lower the transformation time cost, as shown in Fig.7(d). We also notice that the reduction of transformation time with the *ByPair* function is not significant. This is because we need to enumerate all shortest paths in two cases, which dominates the entire time cost.

Size of Link Graph. Figure 7(e) summarizes the space cost of the link graph on *Random* graphs, for various graph sizes. Obviously, a larger original graph results in a larger link graph. We also obtain the size of the link graph generated by *LP*. Actually, the *LP* method needs to record the anonymized edge weight for each edge in space cost $O(m)$, where m is number of edges. Although *LP* consumes less space cost than our method, the main drawback of *LP* lies in its scalability. We cannot run it on a large graph. Figure 7(f) presents the impact of different benefit functions on the space cost of the link graph across real graphs, *Power* graphs (*P*) and *Random* graphs (*R*). All datasets clearly show the effectiveness of the *ByPair* function. The space cost of link graphs with *ByPair* is nearly 1/3 to 1/2 that of link graph using *ByVertex*. As explained above, *ByPair* is a more effective benefit function than *ByVertex* to produce fewer outsourced graphs. Figure 7(g) shows the space cost of link graph on *Power* graphs varying *MPMem* from 1.5M (M is for million) to 3M. It also verifies our earlier claim. When *MPMem* is sufficient to store shortest paths, such as in graphs with fewer than 6k vertices, the adjustment of *MPMem* is not effective. When the graph is larger and only a smaller proportion of all paths can be loaded into memory, *MPMem* is more useful in reducing the space cost of the link graph.

Local Overhead Ratio. Figure 7(h) illustrates the local overhead ratio for various d values on the *Random* graph set. We randomly generate 100 shortest distance queries, and compute the average local overhead ratio. As shown in the figure, the local overhead ratio is lower than 0.06 in all test cases. In other words, the outsourced server carries out the bulk of the computational work for shortest distance discovery. The client side handles the local shortest distance search for distances smaller than d and the merging of results in Equation (1). Therefore, the decrease of d can further reduce the time cost on the client side. We believe the curve in the figure is due to the randomization in the time recording, since all local time cost is nearly zero. As for the *LP* method, it takes $O(l)$ time cost to recover the shortest distance, where l is the total edges in the shortest path. The time cost of *LP* over small graphs is also nearly zero.



(a) Transformation Time vs. d (b) Transformation Time vs. additive error (c) $|G_l|$ vs. d (d) $|G_l|$ vs. additive error

(e) Additive Error vs. additive error (f) Additive Error vs. construction Rules (g) Local Overhead Ratio Vs. Additive Error (h) Local Overhead Ratio vs. d

Figure 8: Experimental Results on Graph Transformation with Approximate Answers

5.3 Graph Transformation with Approximate Answer

In this section, we study the impact of different factors on the measures of graph transformation with approximate shortest distance answering. Specifically, we are interested in: i) What is the impact of d on the transformation overhead and local overhead ratio? ii) Can the given additive error bound be achieved with our method? iii) What is the impact of different heuristic construction rules on the additive error? In all experiments, the number of full shortest path trees in each outsourced graph is set to 50, and the number of sampling queries for each category is 20.

Transformation Time Cost. Figure 8(a) illustrates the transformation time cost across the *Power* graph set, varying d from 10 to 20, when the average additive error $\overline{\beta}$ is set to 12. Outsourced graphs are generated in a random way (denoted by *Random*). A larger d indicates each outsourced graph preservers fewer shortest paths. In order to meet the requirement of the given additive error, more outsourced graphs are needed, which leads to more transformation time.

Figure 8(b) presents the impact of different additive errors on the transformation time cost across two real datasets. We still use the random method to construct outsourced graphs. From Fig.8(b), we can see that a larger additive error leads to less transformation time cost, since in such a case, fewer outsourced graphs are needed.

Size of Link Graph. Figure 8(c) summarizes the size of the link graph for various d on the *Power* graphs with the given average additive error 12. It is no surprise that the size of the link graph goes up with the increase of d. As we have explained in the transformation time cost, a larger d corresponds to more outsourced graphs and then more edges in the link graph to outsourced graphs. Figure 8(d) illustrates the space cost on the real datasets with different additive errors. It clearly shows that the increase of additive error results in a decrease in the size of the link graph. The reason is similar to the explanation we gave along with Fig.8(b). A larger additive error reduces the number of the outsourced graphs used, indicating fewer edges in the link graph.

Additive Error. Figure 8(e) studies whether we can achieve the additive error as expected. After we generate the outsourced graphs with Algorithm 3, we evaluate 100 shortest path queries with Equation (1) and Dijkstra's algorithm, and test whether their average additive error is the same as that specified. We observe that the two values (in X and Y axis) are very close. In other words, our graph transformation method can achieve the specified additive error quite well, which also illustrates the effectiveness of the average additive error estimation strategy in Algorithm 3.

Figure 8(f) compares the average additive errors on the same number of outsourced graphs constructed with different heuristic rules over *Random* graphs. We implement the degree based (denoted by *DegreeBased*) and the cluster based (denoted by *ClusterBased*) method for constructing outsourced graphs. The average additive errors are obtained with 100 queries on outsourced graphs. The results in Fig.8(f) show that among all heuristic construction rules, the cluster based method can produce outsourced graphs with lowest additive error when the number of outsourced graphs is relatively small, for example, less than 4. When more outsourced graphs are generated, three construction methods produce the similar answers, since the distribution of outsourced vertices is more important in such a case.

Local Overhead Ratio. Figure 8(g) and Figure 8(h) illustrate the local overhead ratio on the real datasets and *Power* graphs. We obtain the local overhead ratio as that in Fig.7(h). In all cases, the local time cost used in shortest distance answering is nearly zero. By combining the results in Fig.7(h), we can know that the local overhead ratio scales very well in terms of the graph size. As shown in Table 2, the client side requires $O(n_l^2 + xb^2)$ time cost for shortest distance computation, while n_l, x and b are nearly constant with respect to the graph size. At the same time, the graph size has a significant impact on Dijkstra's algorithm in the distance discovery. Therefore, the local overhead ratio declines sharply with the increase of graph size.

5.4 Summary

To sum up, from the experimental results, we can draw the following conclusions: i) The increase of d, although strengthening security of outsourced graphs, drives up the transformation time cost, the space cost of the link graph and overhead of the query

answering on the client side; ii) Our graph transformation with exact distance answering scales much better than the existing method. Our graph transformation with approximate distance answering achieves the given additive error quite well and can handle large graphs. iii) In all test cases, the local overhead ratio is very low and even goes down with the increase of graph size. Such results illustrate the effectiveness of graph outsourcing.

6. CONCLUSION AND FUTURE WORKS

In this paper, we study how to utilize cloud computing to efficiently compute shortest distance in large graphs without compromising their sensitive information. We define a new security model called 1-neighborhood-d-radius. Our purpose is to reduce the space cost and shortest distance evaluation cost on the client side while satisfying both security and utility requirements. We devise a greedy method to transform graphs with exact shortest distance answering, and develop a fast transformation method to support approximate distance answering within the given average additive error bound.

This work can be extended in several interesting directions. First, we will study how other graph queries such as reachability query can be computed in the cloud within our framework. In general, our framework can be useful for graph operations whose evaluation cost mainly comes from a "global search", yet local information can be easily merged into the results. Second, we will investigate stronger security standards over outsourced graphs. For instance, we can add noise on the edge weight or node degree on the condition that the shortest paths can be preserved in the outsourced graph. Third, incremental graph outsourcing is desirable for dynamic graphs. At the same time, incremental outsourcing should not lead to information leakage in outsourced graphs.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their helpful comments. We would like to thank Victor E. Lee for helping with the paper revision. The NSFC supported Gao via 60873062 and 61073018, and supported Jin via 61003167. The research grants Council of the Hong Kong supported Yu via 419008 and 419109. The NSF supported Jin via IIS-0953950. National science and technology major program supported Wang via 2010ZX01042-001-003-05 and 2010ZX01042-002-002-02

7. REFERENCES

[1] A.V.Goldberg and C.Harrelson. Computing the shortest path: search meets graph theory. In *SODA*, pages 156–165, 2005.

[2] B.Zhou and J.Pei. Preserving privacy in social networks against neighborhood attacks. In *ICDE*, pages 506–515, 2008.

[3] C.Li, M.Hay, V.Rastogi, G.Miklau, and A.McGrego. Optimizing linear counting queries under differential privacy. In *PODS*, pages 123–134, 2010.

[4] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, pages 269–271, 1959.

[5] D.J.Abadi. Data management in the cloud: Limitations and opportunities. *IEEE Data Eng. Bull. (DEBU)*, 32(1):3-12, 2009.

[6] Cynthia Dwork, Frank Mcsherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography Conference*, 2006.

[7] E.Cohen, E.Halperin, H.Kaplan, and U.Zwick. Reachability and distance queries via 2-hop labels. In *SODA*, pages 937–946, 2002.

[8] G.Cormode, D.Srivastava, S.Bhagat, and B.Krishnamurthy. Class-based graph anonymization for social network data. *PVLDB*, 2(1):766-777, 2009.

[9] G.Cormode, D.Srivastava, T.Yu, and Q.Zhang. Anonymizing bipartite graph data using safe groupings. *PVLDB*, 1(1):833-844, 2008.

[10] H.Hacigümüs, B.R.Iyer, and S.Mehrotra. Providing database as a service. In *ICDE*, pages 29–40, 2002.

[11] J.Cheng, A.W.Fu, and J.Liu. K-isomorphism: privacy preserving network publication against structural attacks. In *SIGMOD*, pages 459–470, 2010.

[12] J.M.Kleinberg, A.Slivkins, and T.Wexler. Triangulation and embedding using small sets of beacons.. *J. ACM (JACM)*, 56(6), 2009.

[13] K.Liu and E.Terzi. Towards identity anonymization on graphs. In *SIGMOD*, pages 93–106, 2008.

[14] L.Backstrom, C.Dwork, and J.M.Kleinberg. Wherefore art thou r3579x?: anonymized social networks, hidden patterns, and structural steganography. In *WWW*, pages 181–190, 2007.

[15] L.Liu, J.Wang, J.Liu, and J.Zhang. Privacy preservation in social networks with sensitive edge weights. In *SDM*, pages 954–965, 2009.

[16] L.Zou, L.Chen, and M. TamerÖzsu. Distancejoin: Pattern match query in a large graph database. *PVLDB*, 2(1):886-897, 2009.

[17] L.Zou, L.Chen, and M. TamerÖzsu. K-automorphism: A general framework for privacy preserving network publication. *PVLDB*, 2(1):946-957, 2009.

[18] M.Hay, C.Li, G.Miklau, and D.Jensen. Accurate estimation of the degree distribution of private networks. In *ICDM*, pages 169–178, 2009.

[19] M.Hay, G.Miklau, D.Jensen, D.F.Towsley, and P.Weis. Resisting structural re-identification in anonymized social networks. *PVLDB*, 1(1):102-114, 2008.

[20] M.L.Yiu, Y.Lin, and K.Mouratidis. Efficient verification of shortest path search via authenticated hints.. In *ICDE*, pages 237–248, 2010.

[21] M.Potamias, F.Bonchi, and C.Castillo. Fast shortest path distance estimation in large networks. In *CIKM*, pages 867–876, 2009.

[22] M.Thorup and U.Zwick. Approximate distance oracles. In *STOC*, pages 183–192, 2001.

[23] N.Jing, Y.Huang, and E.A.Rundensteiner. Hierarchical encoded path views for path query processing: An optimal model and its performance evaluation. *TKDE*, 10(3):409–432, 1998.

[24] S.Das, M.Egecioglu, and A.E.Abbadi. Anonymizing weighted social network graphs. In *ICDE*, pages 904–907, 2010.

[25] S.Jung and S.Pramanik. An efficient path computation model for hierarchically structured topographical road maps. *TKDE*, 14(5):1029-1046, 2002.

[26] S.Nath, H.Yu, and H.Chan. Secure outsourced aggregation via one-way chain. In *SIGMOD*, pages 31–44, 2009.

[27] X.Ying and X.Wu. Randomizing social networks: a spectrum preserving approach. In *SDM*, pages 739–750, 2008.