SecGDB: Graph Encryption for Exact Shortest Distance Queries with Efficient Updates

Qian Wang[1,2(✉)], Kui Ren[3], Minxin Du[1], Qi Li[4], and Aziz Mohaisen[3]

[1] School of CS, Wuhan University, Wuhan, China
{qianwang,duminxin}@whu.edu.cn
[2] Collaborative Innovation Center of Geospatial Technology, Wuhan University, Wuhan, China
[3] Department of CSE, University at Buffalo, SUNY, Buffalo, USA
{kuiren,mohaisen}@buffalo.edu
[4] Graduate School at Shenzhen, Tsinghua University, Shenzhen, China
qi.li@sz.tsinghua.edu.cn

Abstract. In the era of big data, graph databases have become increasingly important for NoSQL technologies, and many systems can be modeled as graphs for semantic queries. Meanwhile, with the advent of cloud computing, data owners are highly motivated to outsource and store their massive potentially-sensitive graph data on remote untrusted servers in an encrypted form, expecting to retain the ability to query over the encrypted graphs.

To allow effective and private queries over encrypted data, the most well-studied class of *structured encryption* schemes are searchable symmetric encryption (SSE) designs, which encrypt search structures (*e.g.*, inverted indexes) for retrieving data files. In this paper, we tackle the challenge of designing a Secure Graph DataBase encryption scheme (SecGDB) to encrypt graph structures and enforce private graph queries over the encrypted graph database. Specifically, our construction strategically makes use of efficient additively homomorphic encryption and garbled circuits to support the shortest distance queries with optimal time and storage complexities. To achieve better amortized time complexity over multiple queries, we further propose an auxiliary data structure called *query history* and store it on the remote server to act as a "caching" resource. We prove that our construction is adaptively semantically-secure in the random oracle model and finally implement and evaluate it on various representative real-world datasets, showing that our approach is practically efficient in terms of both storage and computation.

Keywords: Graph encryption · Shortest distance query
Homomorphic encryption · Garbled circuit

1 Introduction

Graphs are used in a wide range of application domains, including social networks, online knowledge discovery, computer networks, and the world-wide web, among

A. Kiayias (Ed.): FC 2017, LNCS 10322, pp. 79–97, 2017.
https://doi.org/10.1007/978-3-319-70972-7_5

others. For example, online social networks (OSN) such as Facebook and LinkedIn employ large social graphs with millions or even billions of vertices and edges in their operation. As a result, various systems have been recently proposed to handle massive graphs efficiently, where examples include GraphLab [22], Horton [29] and TurboGraph [9]. These database applications allow for querying, managing and analyzing large-scale graphs in an intuitive and expressive way.

With the increased popularity of cloud computing, data users, including both individuals and enterprises, are highly motivated to outsource their (potentially huge amount of sensitive) data that may be abstracted and modeled as large graphs to remote cloud servers to reduce the local storage and management costs [6,15,19–21,30,31]. However, database outsourcing also raises data confidentiality and privacy concerns due to data owners' loss of physical data control. Privacy-sensitive data therefore should be encrypted locally before outsourcing it to the untrusted cloud. Data encryption, however, hinders data utilization and computation, making it difficult to efficiently retrieve or query data of interest as opposed to the case with plaintext.

To address this challenge, the notion of *structured encryption* was first introduced by Chase and Kamara [3]. Roughly speaking, a structured encryption scheme encrypts structured data in such a way that it can be privately queried through the use of a specific token generated with knowledge of the secret key. Specifically, they presented approaches for encrypting (structured) graph data while allowing for efficient neighbor queries, adjacency queries and focused subgraph queries on labeled graphs.

Despite all of these important types of queries, finding the shortest distance between two vertices was not supported. The shortest distance queries are not only building blocks for various more complex algorithms, but also have applications of their own. Such applications include finding the shortest path for one person to meet another in an encrypted social network graph, seeking the shortest path with the minimum delay in an encrypted networking or telecommunications abstracted graph, or performing a privacy-preserving GPS guidance in which one party holds the encrypted map while the other knows his origin and destination.

Recently, Meng *et al.* [24] addressed the graph encryption problem by precomputing a data structure called the *distance oracle* from an original graph. They leveraged somewhat homomorphic encryption and standard private key encryption for their construction, thus answering shortest distance queries *approximately* over the encrypted distance oracle. Although their experimental results show that their schemes are practically efficient, the accuracy is sacrificed for using the *distance oracle* (*i.e.*, only the approximate distance or even the negative result is returned). On the one hand, the distance oracle based methods only provide an estimate on the length of the shortest path. On the other hand, the exact path itself could also be necessary and important in many of the aforementioned application scenarios. Furthermore, both of the previous solutions only deal with static graphs [3,24]: the outsourced encrypted graph structure cannot explicitly support

efficient graph updates, since it requires to either re-encrypt the entire graph, or make use of generic and expensive dynamization techniques similar to [4].

To tackle the practical limitations of the state-of-the-art, we propose a new Secure Graph DataBase encryption scheme (SecGDB) that supports both exact shortest distance queries and efficient dynamic operations. Specifically, our construction addresses four major challenges. First, to seek the best tradeoff between accuracy and efficiency, we process the graph itself instantiated by adjacency lists instead of encrypting either the *distance oracle* pre-computed from the original graph or the adjacency matrix instantiation. Second, to compute the exact shortest path over the encrypted graph, we propose a hybrid approach that combines additively homomorphic encryption and garbled circuits to implement Dijkstra's algorithm [5] with the priority queue. Third, to enable dynamic updates of encrypted graphs, we carefully design an extra encrypted data structure to store the relevant information (*e.g.*, neighbor information of nodes in adjacency lists) which will be used to perform modifications homomorphically over the graph ciphertexts. Fourth, to further optimize the performance of the query phase, we introduce an auxiliary data structure called the *query history* by leveraging the previous queried results stored on the remote server as a "caching" resource; namely, the results for subsequent queries can be returned immediately without incurring further cost.

Our main contributions are summarized as follows.

- *Functionality and efficiency.* We propose SecGDB to support exact shortest distance queries with optimal time and storage complexity. We further obtain an improved amortized running time over multiple queries with the auxiliary data structure called "*query history*".
- *Dynamics.* We design an additional encrypted data structure to facilitate efficient graph updates. Compared with the state-of-the-art [3,24], which consider only static data, SecGDB performs dynamic (*i.e.*, addition or removal of specified edges over the encrypted graph) operations with $O(1)$ time complexity.
- *Security, implementation and evaluation.* We formalize our security model using a simulation-based definition and prove the adaptive semantic security of SecGDB under the random oracle model with reasonable leakage. We implement and evaluate the performance of SecGDB on various representative real-world datasets to demonstrate its efficiency and practicality.

2 Preliminaries and Notations

We begin by outlining some notations. Given a graph $G = (V, E)$ which consists of a set of vertices V and edges E, we denote its total number of vertices as $n = |V|$ and its number of edges as $m = |E|$. G is either *undirected* or *directed*. If G is undirected, then each edge in E is an *unordered* pair of vertices, and we use $\mathsf{len}(u, v)$ to denote the length of edge (u, v), otherwise, each edge in E is an *ordered* pair of vertices. In an undirected graph, $\mathsf{deg}(v)$ is used to denote the number of vertices adjacent to the vertex v (*i.e.*, *degree*). For a directed graph,

we use $\mathsf{deg}^-(v)$ and $\mathsf{deg}^+(v)$ to denote the number of edges directed to vertex v (*indegree*) and out of vertex v (*outdegree*), respectively. A shortest distance query $q = (s, t)$ asks for the length (along with the route) of the shortest path between s and t, which we denote by $\mathsf{dist}(s, t)$ or dist_q. $[n]$ denotes the set of positive integers less than or equal to n, *i.e.*, $[n] = \{1, 2, \ldots, n\}$. We write $x \xleftarrow{\$} X$ to represent an element x being uniformly sampled at random from a set X. The output x of a probabilistic algorithm $\mathcal{A}$ is denoted by $x \leftarrow \mathcal{A}$ and that of a deterministic algorithm $\mathcal{B}$ by $x := \mathcal{B}$. Given a sequence of elements $\mathbf{v}$, we refer to the i^{th} element as $\mathbf{v}[i]$ or $\mathbf{v}_i$ and to the total number of elements in $\mathbf{v}$ by $|\mathbf{v}|$. If A is a set then $|A|$ refers to its cardinality, and if s is a string then $|s|$ refers to its bit length. We denote the concatenation of n strings $s_1, \ldots, s_n$ by $\langle s_1, \ldots, s_n \rangle$, and also denote the high-order $|s_2|$-bit of the string s_1 by $s_1^{|s_2|}$.

We also use various basic data structures including linked lists, arrays and dictionaries. Specifically, a dictionary T (also known as a map or associative array) is a data structure that stores key-value pairs (k, v). If the pair (k, v) is in T, then $\mathsf{T}[k]$ is the value v associated with k. An insertion operation of a new key-value pair (k, v) to the dictionary T is denoted by $\mathsf{T}[k] := v$. Similarly, a lookup operation takes a dictionary T and a specified key k as input, then returns the associated value v denoted by $v := \mathsf{T}[k]$.

2.1 Cryptographic Tools

Homomorphic encryption. Homomorphic encryption allows certain computations to be carried out on ciphertexts to generate an encrypted result which matches the result of operations performed on the plaintext after being decrypted. In this work, we only require the evaluation to efficiently support any number of additions, and there are many cryptosystems satisfying with this property. In particular, we use the Paillier cryptosystem [27] in our construction.

In the Paillier cryptosystem, the public (encryption) key is $pk_p = (n = pq, g)$, where $g \in \mathbb{Z}^*_{n^2}$, and p and q are two large prime numbers (of equivalent length) chosen randomly and independently. The private (decryption) key is $sk_p = (\varphi(n), \varphi(n)^{-1} \mod n)$. Given a message a, we write the encryption of a as $[\![a]\!]_{pk}$, or simply $[\![a]\!]$, where pk is the public key. The encryption of a message $x \in \mathbb{Z}_n$ is $[\![x]\!] = g^x \cdot r^n \mod n^2$, for some random $r \in \mathbb{Z}^*_n$. The decryption of the ciphertext is $x = L([\![x]\!]^{\varphi(n)} \mod n^2) \cdot \varphi^{-1}(n) \mod n$, where $L(u) = \frac{u-1}{n}$. The homomorphic property of the Paillier cryptosystem is given by $[\![x_1]\!] \cdot [\![x_2]\!] = (g^{x_1} \cdot r_1^n) \cdot (g^{x_2} \cdot r_2^n) = g^{x_1+x_2}(r_1 r_2)^n \mod n^2 = [\![x_1 + x_2]\!]$.

Pseudo-random functions (PRFs) and permutations (PRPs). Let $F : \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^*$ be a PRF, which is a polynomial-time computable function that cannot be distinguished from random functions by any probabilistic polynomial-time adversary. A PRF is said to be a PRP when it is bijective. Readers can refer to [16] for the formal definition and security proof.

Oblivious transfer. Parallel 1-out-of-2 Oblivious Transfer (OT) of m l-bit strings [13,25], denoted as OT^m_l, is a two-party protocol run between a chooser

$\mathcal{C}$ and a sender $\mathcal{S}$. For $i = 1, \ldots, m$, the sender $\mathcal{S}$ inputs a pair of l-bit strings $s_i^0, s_i^1 \in \{0,1\}^l$ and the chooser $\mathcal{C}$ inputs m choice bits $b_i \in \{0,1\}$. At the end of the protocol, $\mathcal{C}$ learns the chosen strings $s_i^{b_i}$ but nothing about the unchosen strings $s_i^{1-b_i}$, whereas $\mathcal{S}$ learns nothing about the choice b_i.

Garbled circuits. Garbled circuits were first proposed by Yao [32] for secure two-party computation and later proven practical by Malkhi *et al.* [23]. At a high level, garbled circuits allow two parties holding inputs x and y, respectively, to jointly evaluate an arbitrary function $f(x, y)$ represented as a boolean circuit without leaking any information about their inputs beyond what is implied by the function output.

Several optimization techniques have been proposed in the literature to construct the standard garbled circuits. Kolensikov *et al.* [18] introduced an efficient method for creating garbled circuits which allows "free" evaluation of XOR gates. Pinkas *et al.* [28] proposed an approach to reduce the size of garbled gates from four to three entries, thus saving 25% of the communication overhead.

2.2 Fibonacci Heap

Fibonacci heap [7] is a data structure for implementing priority queues, which consists of a collection of *trees* satisfying the *minimum-heap* property; that is, the key of a child is always greater than or equal to the key of the parent. This implies that the minimum key is always at the root of one of the trees. Generally, a heap data structure supports the following six operations: $\mathsf{Make\text{-}Heap}()$, $\mathsf{Insert}(\mathsf{H}, x)$, $\mathsf{Minimum}(\mathsf{H})$, $\mathsf{Extract\text{-}MIN}(\mathsf{H})$, $\mathsf{Decrease\text{-}Key}(\mathsf{H}, x)$ and $\mathsf{Delete}(\mathsf{H}, x)$.

Compared with many other priority queue data structures including the *Binary heap* and *Binomial heap*, the Fibonacci heap achieves a better amortized running time [7].

3 System Model and Definitions

In this work, we consider the problem of designing a structured encryption scheme that supports the shortest distance queries and dynamic operations over an encrypted graph stored on remote servers efficiently.

At a high level, as shown in Fig. 1, our construction contains three entities, namely the client $\mathcal{C}$, the server $\mathcal{S}$ and the proxy $\mathcal{P}$. In the initialization stage, the client $\mathcal{C}$ processes the original graph G to obtain its encrypted form Ω_G, outsources Ω_G to the server $\mathcal{S}$ and distributes partial secret key sk to the proxy $\mathcal{P}$. The privacy holds as long as the server $\mathcal{S}$ and the proxy $\mathcal{P}$ do not collude, and this architecture of two non-colluding entities has been commonly used in the related literature [1,6,26]. Subsequently, to enable the shortest distance query over the encrypted graph Ω_G, the client generates a query token τ_q based on the query q and submits it to the cloud server $\mathcal{S}$. Finally, the encrypted shortest distance along with the path are returned to the client $\mathcal{C}$. In addition, the graph storage service in consideration is *dynamic*, such that the client $\mathcal{C}$ may add or remove edges to or from the

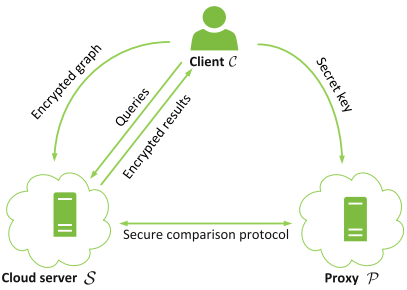


Fig. 1. System model

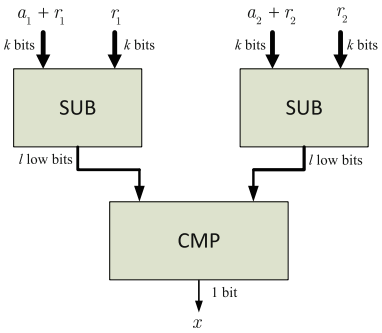


Fig. 2. The secure comparison circuit.

encrypted graph Ω_G as well as modify the length of the specified edge. To do so, the client generates an update token τ_u corresponding to the dynamic operations. Given τ_u, the server $\mathcal{S}$ can securely update the encrypted graph Ω_G.

Formally, the core functionalities of our system are listed as below.

Definition 1. *An encrypted graph database system supporting the shortest distance query and dynamic updates consists of the following five (possibly probabilistic) polynomial-time algorithms/protocols:*

$sk \leftarrow \mathsf{Gen}(1^\lambda)$*: is a probabilistic key generation algorithm run by the client. It takes as input a security parameter* λ *and outputs the secret key* sk*.*

$\Omega_G \leftarrow \mathsf{Enc}(sk, G)$*: is a probabilistic algorithm run by the client. It takes as input a secret key* sk *and a graph* G*, and outputs an encrypted graph* Ω_G*.*

$\mathsf{dist}_q \leftarrow \mathsf{Dec}(sk, c_q)$*: is a deterministic algorithm run by the client. It takes as input a secret key* sk *and an encrypted result* c_q*, and outputs* dist_q *including the shortest distance as well as its corresponding path.*

$(c_q; \sigma') \leftarrow \mathsf{DistanceQuery}(sk, q; \Omega_G, \sigma)$*: is a (possibly interactive and probabilistic) protocol run between the client and the server*[1]*. The client takes as input a secret key* sk *and a shortest distance query* q*, while the server takes as input the encrypted graph* Ω_G *and the query history* σ *(which is empty in the beginning). During the protocol execution, a query token* τ_q *is generated by the client based on the query* q *and then sent to the server. Upon completion of the protocol, the client obtains an encrypted result* c_q *while the server gets a (possibly new) query history* σ'*.*

$(\perp; \Omega'_G, \sigma) \leftarrow \mathsf{UpdateQuery}(sk, u; \Omega_G)$*: is a (possibly interactive and probabilistic) protocol run between the client and the server. The client takes as input a secret key* sk *and an update object* u *(e.g., the edges to be updated), while the server takes as*

[1] A protocol P run between the client and the server is denoted by $(u; v) \leftarrow P(x; y)$, where x and y are the client's and the server's inputs, respectively, and u and v are the client's and the server's outputs, respectively.

input the encrypted graph Ω_G. During the protocol execution, an update token τ_u is generated by the client based on the object u and then sent to the server. Upon completion of the protocol, the client gets nothing while the server obtains an updated encrypted graph Ω'_G and a new empty query history σ.

3.1 Security Definitions

As in previous SSE systems [2,4,8,14,15] we also relax the security requirements appropriately by allowing some reasonable information leakage to the adversary in order to obtain higher efficiency. To capture this relaxation, we follow [3,4,8,15] to parameterize the information by using a tuple of well-defined leakage functions (see Sect. 5). Besides, we assume that the server and the proxy are both semi-honest entities in our setting.

In the following definition, we adapt the notion of adaptive semantic security from [3,4,15] to our encrypted graph database system.

Definition 2. *(Adaptive semantic security) Let* (Gen,Enc,Dec,DistanceQuery, UpdateQuery) *be a dynamic encrypted graph database system and consider the following experiments with a stateful adversary $\mathcal{A}$, a stateful simulator $\mathcal{S}$ and three stateful leakage functions $\mathcal{L}_1$, $\mathcal{L}_2$ and $\mathcal{L}_3$:*

$\mathbf{Real}_{\mathcal{A}}(\lambda)$: *The challenger runs* $\mathsf{Gen}(1^\lambda)$ *to generate the key sk. $\mathcal{A}$ outputs G and receives $\Omega_G \leftarrow \mathsf{Enc}(sk, G)$ from the challenger. $\mathcal{A}$ then makes a polynomial number of adaptive shortest distance queries q or update queries u. For each q, the challenger acts as a client and runs* DistanceQuery *with $\mathcal{A}$ acting as a server. For each update query u, the challenger acts as a client and runs* UpdateQuery *with $\mathcal{A}$ acting as a server. Finally, $\mathcal{A}$ returns a bit b as the output of the experiment.*

$\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$: *$\mathcal{A}$ outputs G. Given $\mathcal{L}_1(G)$, $\mathcal{S}$ generates and sends Ω_G to $\mathcal{A}$. $\mathcal{A}$ makes a polynomial number of adaptive shortest distance queries q or update queries u. For each q, $\mathcal{S}$ is given $\mathcal{L}_2(G, q)$, and simulates a client who runs* DistanceQuery *with $\mathcal{A}$ acting as a server. For each update query u, $\mathcal{S}$ is given $\mathcal{L}_3(G, u)$, and simulates a client who runs* UpdateQuery *with $\mathcal{A}$ acting as a server. Finally, $\mathcal{A}$ returns a bit b as the output of the experiment.*

We say such a queryable encrypted graphs database system is adaptively $(\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3)$-semantically secure if for all probabilistic polynomial-time (PPT) adversaries $\mathcal{A}$, there exists a probabilistic polynomial-time simulator $\mathcal{S}$ such that

$$|\Pr[\mathbf{Real}_{\mathcal{A}}(\lambda) = 1] - \Pr[\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda) = 1]| \leq \mathsf{negl}(\lambda),$$

where $\mathsf{negl}(\cdot)$ *is a negligible function.*

4 Our Construction: SecGDB

In this section, we present our encrypted graph database system–SecGDB, which efficiently supports the shortest distance query and the update query (*i.e.*, to add, remove and modify a specified edge).

4.1 Overview

We assume that an original graph is instantiated by adjacency lists, and every node in each adjacency list contains a pair of the neighboring vertex and the length of the corresponding edge (*i.e.*, vertex and length pair).

Our construction is inspired by [15], and the key idea is as follows. During the initialization phase, we place every node of each adjacency list at a random location in the array while updating the pointers so that the "logical" integrity of the lists are preserved. We then use the Paillier cryptosystem to encrypt the length of the edge in each node, and use a "standard" private-key encryption scheme [16] to blind the entire node. In the shortest distance query phase, if the query has been submitted before or was a subpath of the *query history*, the result can be immediately returned to the client; otherwise, we implement the Dijkstra's algorithm with the aid of Fibonacci heap in a secure manner, and then query history is updated based on the results. To support efficient dynamic operations on the encrypted graph, we generate the relevant update token, which allows the server to add or remove the specified entry to and from the array. After finishing the updates, the query history is rebuilt for future use.

4.2 Initialization Phase

Intuitively, the initialization phase consists of Gen and Enc as presented in Definition 1. The scheme uses the Paillier cryptosystem, and three pseudo-random functions P, F and G, where P is defined as $\{0,1\}^\lambda \times \{0,1\}^* \rightarrow \{0,1\}^\lambda$, F is defined as $\{0,1\}^\lambda \times \{0,1\}^* \rightarrow \{0,1\}^*$ and G is defined as $\{0,1\}^\lambda \times \{0,1\}^* \rightarrow \{0,1\}^\lambda$. We also use a random oracle H which is defined as $\{0,1\}^* \rightarrow \{0,1\}^*$.

$\mathsf{Gen}(1^\lambda)$: Given a security parameter λ, generate the following keys uniformly at random from their respective domains: three PRF keys $k_1, k_2, k_3 \xleftarrow{\$} \{0,1\}^\lambda$ for $P_{k_1}(\cdot)$, $F_{k_2}(\cdot)$ and $G_{k_3}(\cdot)$, respectively, and (sk_p, pk_p) for the Paillier cryptosystem. The output is $sk = (k_1, k_2, k_3, sk_p, pk_p)$, where sk_p is sent to the proxy through a secure channel.

As shown in Algorithm 1, the setup procedures are done in the first five steps. From line 6 to 29, the length of the edge is encrypted under the Paillier cryptosystem and the entire node $\mathtt{N}_i$ is encrypted by XORing an output of the random oracle H. Meanwhile, the neighboring information of each node $\mathtt{N}_i$ (*i.e.*, the nodes following and previous to $\mathtt{N}_i$ in the original adjacency lists, and the corresponding positions in A_G) constitutes the dual node $\mathtt{D}_i$, and the encrypted dual node will be stored in the dictionary T_D. Generally speaking, T_D stores the pointer to each edge, and it is used to support efficient delete updates on the

Algorithm 1. Graph Enc algorithm

Input: $G = (V, E), sk$
Output: Ω_G
Set $n = |V|, m = |E|$;
Initialize an array A_G of size $m + z$;
Initialize two dictionaries $\mathsf{T}_G, \mathsf{T}_D$ of size $n+1$ and m;
Initialize a random permutation π over $[m + z]$;
Initialize a counter $\mathtt{ctr} = 1$;
for each vertex $u \in V$ **do**
  Generate $K_u := G_{k_3}(u)$;
  for $i = 1$ to $\mathsf{deg}^+(u)$ **do**
    Encrypt the length of the edge (u, v_i) under the Paillier cryptosystem $c_i \leftarrow [\![\mathsf{len}(u, v_i)]\!]_{pk_p}$
    if $i = 1$ and $i \neq \mathsf{deg}^+(u)$ **then**
      Set $\mathtt{N}_i := \langle P_{k_1}(v_i), F_{k_2}(v_i), c_i, \pi(\mathtt{ctr}+1)\rangle$;
      Set $\mathtt{D}_i := \langle \mathbf{0}, \mathbf{0}, \pi(\mathtt{ctr}), \pi(\mathtt{ctr}+1), P_{k_1}(\langle u, v_{i+1}\rangle)\rangle$;
    else if $i \neq 1$ and $i = \mathsf{deg}^+(u)$ **then**
      Set $\mathtt{N}_i := \langle P_{k_1}(v_i), F_{k_2}(v_i), c_i, \mathtt{NULL}\rangle$;
      Set $\mathtt{D}_i := \langle P_{k_1}(\langle u, v_{i-1}\rangle), \pi(\mathtt{ctr}-1), \pi(\mathtt{ctr}), \mathbf{0}, \mathbf{0}\rangle$;
    else if $i = 1$ and $i = \mathsf{deg}^+(u)$ **then**
      Set $\mathtt{N}_i := \langle P_{k_1}(v_i), F_{k_2}(v_i), c_i, \mathtt{NULL}\rangle$;
      Set $\mathtt{D}_i := \langle \mathbf{0}, \mathbf{0}, \pi(\mathtt{ctr}), \mathbf{0}, \mathbf{0}\rangle$;
    else
      Set $\mathtt{N}_i := \langle P_{k_1}(v_i), F_{k_2}(v_i), c_i, \pi(\mathtt{ctr}+1)\rangle$;
      Set $\mathtt{D}_i := \langle P_{k_1}(\langle u, v_{i-1}\rangle), \pi(\mathtt{ctr}-1), \pi(\mathtt{ctr}), \pi(\mathtt{ctr}+1), P_{k_1}(\langle u, v_{i+1}\rangle)\rangle$;
    end if
    Sample $r_i \xleftarrow{\$} \{0,1\}^\lambda$;
    Store the encrypted $\mathtt{N}_i$ in the array $\mathsf{A}_G[\pi(\mathtt{ctr})] := \langle \mathtt{N}_i \oplus H(K_u, r_i), r_i\rangle$;
    Store the encrypted $\mathtt{D}_i$ in the dictionary $\mathsf{T}_D[P_{k_1}(\langle u, v_i\rangle)] := \mathtt{D}_i \oplus F_{k_2}(\langle u, v_i\rangle)$;
    Increase $\mathtt{ctr} = \mathtt{ctr} + 1$;
  end for
  Store a pointer to the head node of the adjacency list for u in the dictionary $\mathsf{T}_G[P_{k_1}(u)] := \langle \mathtt{addr}(\mathtt{N}_1), P_{k_1}(\langle u, v_1\rangle), K_u\rangle \oplus F_{k_2}(u)$;
end for
for $i = 1$ to z **do**
  Set $\mathtt{F}_i := \langle \mathbf{0}, \pi(\mathtt{ctr}+1)\rangle$;
  if $i = z$ **then**
    Set $\mathtt{F}_i := \langle \mathbf{0}, \mathtt{NULL}\rangle$;
  end if
  Store the unencrypted $\mathtt{F}_i$ in the array $\mathsf{A}_G[\pi(\mathtt{ctr})] := \mathtt{F}_i$;
  Increase $\mathtt{ctr} = \mathtt{ctr} + 1$;
end for
Store a pointer to the head node of the `free` list in the dictionary $\mathsf{T}_G[\mathtt{free}] := \langle \mathtt{addr}(\mathtt{F}_1), \mathbf{0}\rangle$;
Output the encrypted graph $\Omega_G = (\mathsf{A}_G, \mathsf{T}_G, \mathsf{T}_D)$;

encrypted graph. After the aforementioned operations are done, the address of each head node will be encrypted and stored in the dictionary T_G, namely, T_G stores the pointer to the head of each adjacency list. The remaining z cells in the array construct an unencrypted `free` list, which is used in the add updates. To ensure the size of all the entries in A_G, T_G and T_D is identical, we should pad by a string of 0's (*i.e.*, $\mathbf{0}$). Finally, we output the encrypted graph Ω_G.

Figure 3 gives an illustrative example to construct the encrypted graph from a directed graph with four vertices v_1, v_2, v_3 and v_4 as well as five edges. All the nodes contained in the original (three) adjacency lists are now stored at random locations in A_G, and the dictionaries T_G and T_D are also shown in Fig. 3. Note that in a real encrypted graph, there would be padding to hide partial structural information of the original graph (as will be discussed in Sect. 5); we omit this padding for simplicity in this example.

4.3 Shortest Distance Query Phase

In this section, we describe the process of performing the exact shortest distance query over the encrypted graph, as summarized in Algorithm 2.

First, the client generates the query token τ_q based on a query $q = (s, t)$, and then sends it to the server. If the token has been queried before or acts as a sub-path of the *query history* σ, the server returns the result c_q ($c_q \subset \sigma$) to the client

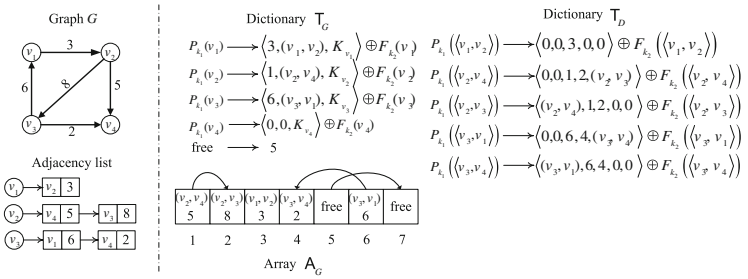


Fig. 3. An example of the encrypted graph construction.

immediately; otherwise, the server executes the Dijkstra's algorithm with the aid of a Fibonacci heap H in a private way. Concretely, the server first reads off the vertices that are adjacent to the source s and inserts to the heap H (line 14 to 22). Subsequently, each iteration of the loop from line 23 to 49 starts by extracting the vertex α with the minimum key. If the vertex α is the requested destination τ_2, the server updates the query history σ based on the newly-obtained path, computes the encrypted result c_q via reverse iteration and returns it to the client. Else, the server recovers the pointer to the head of the adjacency list for the vertex α, and then retrieves nodes in the adjacency list. Specifically, for the node $\mathtt{N}_i$, once an update of $\xi[\alpha_i]$ occurs it indicates that a shorter path to α_i via α has been discovered, the server then updates the path. Next, the server either runs $\mathsf{Insert}(\mathsf{H}, \alpha_i)$ (if α_i is not in H) or $\mathsf{Decrease\text{-}Key}(\mathsf{H}, \alpha_i, key(\alpha_i))$. It is worth noting that both the conditional statement $\xi[\alpha] \cdot c_i < \xi[\alpha_i]$ and some specific operations on the Fibonacci heap (*e.g.*, $\mathsf{Extract\text{-}MIN}$) require performing a comparison on the encrypted data. Hence we build a secure comparison protocol (see Sect. 4.3) based on the garbled circuits and invoke it as a subroutine.

Finally, the client runs $\mathsf{Dec}(c_q, sk)$ to obtain the dist_q as follows. Given c_q, the client parses it as a sequence of $\langle c_1, c_2 \rangle$ pairs, and for each pair, the client decrypts c_1 (the path) and c_2 (the distance) by using k_1 and sk_p, respectively.

Remarks. Conceptually, the history σ consists of all previous de-duplicated queried results. For a new query, the server traverses σ and checks whether the new query belongs to a record in σ. For example, let history σ consist of a shortest path from s to t (*i.e.*, $\{s, \ldots, u, \ldots, v, \ldots, t\}$), then for a new query $q = (u, v)$, the corresponding encrypted result $c_q = \{u, \ldots, v\}$ where $c_q \subset \sigma$ can be returned immediately. Note that only lookup operations (of dictionary) are required, thus making the whole process highly efficient.

Secure Comparison Protocol. We now present the secure comparison protocol which is based on the garbled circuits [12,32] for selecting the minimum of two encrypted values. This subroutine is implemented by the circuit shown in Fig. 2, and we use a CMP circuit and two SUB circuits constructed in [17] to realize the desired functionality.

Algorithm 2. DistanceQuery protocol

Input:
The client $\mathcal{C}$'s input is $sk, q = (s, t)$;
The server $\mathcal{S}$'s input is Ω_G, σ;
Output:
The client $\mathcal{C}$'s output is c_q;
The server $\mathcal{S}$'s output is σ';
$\mathcal{C}$: compute $\tau_q := (P_{k_1}(s), P_{k_1}(t), F_{k_2}(s))$;
$\mathcal{C} \Rightarrow \mathcal{S}$: output τ_q to the server;
$\mathcal{S}$: parse τ_q as (τ_1, τ_2, τ_3);
if $\mathsf{T}_G[\tau_1] = \perp$ or $\mathsf{T}_G[\tau_2] = \perp$ **then**
  $\mathcal{S} \Rightarrow \mathcal{C}$: return $\perp$ to the client;
else if $\{\tau_1, \tau_2\} \subset \sigma$ **then**
  $\mathcal{S} \Rightarrow \mathcal{C}$: return c_q to the client;
else
  $\mathcal{S}$: initialize a Fibonacci heap H $\leftarrow$ Make-Heap();
  $\mathcal{S}$: initialize two dictionaries ξ and path;
  $\mathcal{S}$: compute $\langle \mathtt{addr}_1, \mathtt{str}, K_s \rangle := \mathsf{T}_G[\tau_1] \oplus \tau_3$;
  $\mathcal{S}$: parse $\mathsf{A}_G[\mathtt{addr}_1]$ as $\langle \mathtt{N}'_1, r_1 \rangle$;
  $\mathcal{S}$: compute $\mathtt{N}_1 := \mathtt{N}'_1 \oplus H(K_s, r_1)$;
  while $\mathtt{addr}_{i+1} \neq \mathtt{NULL}$ **do**
    $\mathcal{S}$: parse $\mathtt{N}_i$ as $\langle \alpha_i, \beta_i, c_i, \mathtt{addr}_{i+1} \rangle$;
    $\mathcal{S}$: store $\mathsf{path}[\alpha_i] := \langle \tau_1, c_i \rangle$
    $\mathcal{S}$: set $\xi[\alpha_i] := c_i$ and $key(\alpha_i) := \xi[\alpha_i]$;
    $\mathcal{S}$: run Insert(H, α_i) with the $key(\alpha_i)$;
    $\mathcal{S}$: parse $\mathsf{A}_G[\mathtt{addr}_{i+1}]$ as $\langle \mathtt{N}'_{i+1}, r_{i+1} \rangle$;
    $\mathcal{S}$: compute $\mathtt{N}_{i+1} := \mathtt{N}'_{i+1} \oplus H(K_s, r_{i+1})$;
    $\mathcal{S}$: increase $i = i + 1$;
  end while
  repeat
    $\mathcal{S}$: parse Extract-MIN(H) as $\langle \alpha, key(\alpha) \rangle$;
    if $\alpha = \tau_2$ **then**
      $\mathcal{S}$: update σ' based on path;
      $\mathcal{S} \Rightarrow \mathcal{C}$: return c_q to the client;
      $\mathcal{S}$: break;
    end if
    $\mathcal{S}$: compute $\langle \mathtt{addr}_1, \mathtt{str}, K_u \rangle := \mathsf{T}_G[\alpha] \oplus \beta$;
    $\mathcal{S}$: parse $\mathsf{A}_G[\mathtt{addr}_1]$ as $\langle \mathtt{N}'_1, r_1 \rangle$;
    $\mathcal{S}$: compute $\mathtt{N}_1 := \mathtt{N}'_1 \oplus H(K_u, r_1)$;
    while $\mathtt{addr}_{i+1} \neq \mathtt{NULL}$ **do**
      $\mathcal{S}$: parse $\mathtt{N}_i$ as $\langle \alpha_i, \beta_i, c_i, \mathtt{addr}_{i+1} \rangle$;
      if $\xi[\alpha] \cdot c_i < \xi[\alpha_i]$ **then**
        $\mathcal{S}$: update $\xi[\alpha_i] := \xi[\alpha] \cdot c_i$;
        $\mathcal{S}$: set $key(\alpha_i) := \xi[\alpha_i]$;
        $\mathcal{S}$: store $\mathsf{path}[\alpha_i] := \langle \alpha, c_i \rangle$;
      end if
      if $\alpha_i \notin$ H **then**
        $\mathcal{S}$: run Insert(H, α_i) with the $key(\alpha_i)$;
      else
        $\mathcal{S}$: run Decrease-Key(H, α_i, $key(\alpha_i)$);
      end if
      $\mathcal{S}$: parse $\mathsf{A}_G[\mathtt{addr}_{i+1}]$ as $\langle \mathtt{N}'_{i+1}, r_{i+1} \rangle$;
      $\mathcal{S}$: compute $\mathtt{N}_{i+1} := \mathtt{N}'_{i+1} \oplus H(K_u, r_{i+1})$;
      $\mathcal{S}$: increase $i = i + 1$;
    end while
  until H is empty
end if

At the beginning, the server has two encrypted values $[\![a_1]\!]$ and $[\![a_2]\!]$ and the proxy has the secret key sk_p. W.l.o.g., we assume that the longest shortest distance between any pair of vertices (*i.e.*, diameter [10]) lies in $[2^l]$, namely, a_1 and a_2 are two l-bit integers. Instead of sending $[\![a_1]\!]$ and $[\![a_2]\!]$ to the proxy, the server first masks them with two k-bit random numbers r_1 and r_2 (*e.g.*, $[\![a_1 + r_1]\!] = [\![a_1]\!] \cdot [\![r_1]\!]$) respectively, where k is a security parameter ($k > l$). Then the server's inputs are r_1 and r_2, and the proxy's inputs are $a_1 + r_1$ and $a_2 + r_2$. Finally, the output single bit x implies the comparison result: if $x = 1$, then $a_1 > a_2$; 0 otherwise. Note that masking here is done by performing addition over the integers which is a form of statistical hiding. More precisely, for a l-bit integer a_i and a k-bit integer r_i, releasing $a_i + r_i$ gives statistical security of roughly 2^{l-k} for the potential value a_i. Therefore, by choosing the security parameter k properly, we can make this statistical difference arbitrarily low [12].

Packing Optimization. It is worth noting that the message space of the Paillier cryptosystem is much greater than the space of the blinded values. We can therefore provide a great improvement in both computation time and bandwidth by leveraging the packing technique. The key idea lies in that the server can send one aggregated ciphertext in the form $[\![\langle (a_{i+1} + r_{i+1}), \ldots, (a_{i+p} + r_{i+p}) \rangle]\!]$ instead of p ciphertexts of the form $[\![a_i + r_i]\!]$, where $p = \frac{1024}{k}$ (1024-bit modulus used in Paillier cryptosystem).

Algorithm 3. UpdateQuery protocol

Input:
The client $\mathcal{C}$'s input is sk, u;
The server $\mathcal{S}$'s input is Ω_G;
Output:
The client $\mathcal{C}$'s output is $\perp$;
The server $\mathcal{S}$'s output is Ω'_G, σ;

a) Adding new edges
At the client $\mathcal{C}$:
1) u contains information about newly-added edge (v_1, v_2) with the length $\mathsf{len}(v_1, v_2)$;
2) compute the update token $\tau_u := (P_{k_1}(v_1), F_{k_2}(v_1)^{|\langle \mathtt{addr},\mathtt{str}\rangle|}, P_{k_1}(\langle v_1, v_2\rangle), F_{k_2}(\langle v_1, v_2\rangle), \mathtt{N})$, where $\mathtt{N} = \langle\langle P_{k_1}(v_2), F_{k_2}(v_2), [\![\mathtt{len}]\!], \mathbf{0}\rangle \oplus H(K_{v_1}, r), r\rangle$;
$\mathcal{C} \Rightarrow \mathcal{S}$: output τ_u to the server;
At the server $\mathcal{S}$:
1) parse τ_u as $(\tau_1, \tau_2, \tau_3, \tau_4, \tau_5)$ and return $\perp$ if τ_1 is not in T_G;
2) compute $\langle \mathtt{addr}_1, \mathbf{0}\rangle := \mathsf{T}_G[\mathtt{free}]$;
3) parse $\mathsf{A}_G[\mathtt{addr}_1]$ as $\langle \mathbf{0}, \mathtt{addr}_2\rangle$;
4) update the pointer to the next `free` node $\mathsf{T}_G[\mathtt{free}] := \langle \mathtt{addr}_2, \mathbf{0}\rangle$;
5) compute $\langle \mathtt{addr}_3, \mathtt{str}\rangle := \mathsf{T}_G[\tau_1]^{|\langle \mathtt{addr},\mathtt{str}\rangle|} \oplus \tau_2$;
6) parse τ_5 as $\langle \mathtt{N}', r\rangle$ and set $\mathsf{A}_G[\mathtt{addr}_1] := \langle \mathtt{N}' \oplus \langle \mathbf{0}, \mathtt{addr}_3\rangle, r\rangle$;
7) update the pointer to the newly-added node $\mathsf{T}_G[\tau_1] := \mathsf{T}_G[\tau_1]^{|\langle \mathtt{addr},\mathtt{str}\rangle|} \oplus \langle \mathtt{addr}_3, \mathtt{str}\rangle \oplus \langle \mathtt{addr}_1, \tau_3\rangle$;
8) store $\mathsf{T}_D[\tau_3] := \langle \mathbf{0}, \mathbf{0}, \mathtt{addr}_1, \mathtt{addr}_3, \mathtt{str}\rangle \oplus \tau_4$;
9) update $\mathsf{T}_D[\mathtt{str}] := \mathsf{T}_D[\mathtt{str}]^{|\langle \mathtt{addr},\mathtt{str}\rangle|} \oplus \langle \tau_3, \mathtt{addr}_1\rangle$;
10) obtain an updated graph Ω'_G and rebuild σ;

b) Deleting existing edges
At the client $\mathcal{C}$:
1) u contains information about the existing edge (v_1, v_2) to be deleted;
2) compute $\tau_u := (P_{k_1}(\langle v_1, v_2\rangle), F_{k_2}(\langle v_1, v_2\rangle))$;
$\mathcal{C} \Rightarrow \mathcal{S}$: outputs τ_u to the server;
At the server $\mathcal{S}$:
1) parse τ_u as (τ_1, τ_2) and return $\perp$ if τ_1 is not in T_D;
2) look up in T_D and computes $\langle \mathtt{str}_1, \mathtt{addr}_1, \mathtt{addr}_2, \mathtt{addr}_3, \mathtt{str}_3\rangle := \mathsf{T}_D[\tau_1] \oplus \tau_2$;
3) compute $\langle \mathtt{addr}_4, \mathbf{0}\rangle := \mathsf{T}_G[\mathtt{free}]$;
4) free the node and set $\mathsf{A}_G[\mathtt{addr}_2] := \langle \mathbf{0}, \mathtt{addr}_4\rangle$;
5) update the pointer $\mathsf{T}_G[\mathtt{free}] := \langle \mathtt{addr}_2, \mathbf{0}\rangle$;
6) parse $\mathsf{A}_G[\mathtt{addr}_1]$ as $\langle \mathtt{N}'_1, r_1\rangle$;
7) update node $\mathsf{A}_G[\mathtt{addr}_1] := \langle \mathtt{N}'_1 \oplus \mathtt{addr}_2 \oplus \mathtt{addr}_3, r_1\rangle$;
8) update the corresponding entry $\mathsf{T}_D[\mathtt{str}_1] := \mathsf{T}_D[\mathtt{str}_1] \oplus \langle \mathtt{addr}_2, \tau_1\rangle \oplus \langle \mathtt{addr}_3, \mathtt{str}_3\rangle$;
9) update the corresponding entry $\mathsf{T}_D[\mathtt{str}_3] := \mathsf{T}_D[\mathtt{str}_3] \oplus \langle \mathtt{addr}_2, \tau_1\rangle \oplus \langle \mathtt{addr}_1, \mathtt{str}_1\rangle$;
10) obtain an updated graph Ω'_G and rebuild σ;

4.4 Supporting Encrypted Graph Dynamics

We next discuss the support of update operations over the encrypted graph, and the details are given in Algorithm 3. Here, we do not particularly consider the addition and removal of vertices, because the update of the vertex can be viewed as the update of a collection of related edges.

To add new edges, the client generates the corresponding token τ_u for an update object u and sends it to the server. After receiving τ_u, the server locates the first `free` node $\mathtt{addr}_1$ in the array A_G, and modifies the pointer in T_G to point to the second one. Later, the server retrieves the high-order useful information (without the key K_{v_1}) of the head node $\mathtt{N}_1$, stores $\mathtt{N}$ that represents the newly edge at location $\mathtt{addr}_1$ and modifies its pointer to point to the original head node $\mathtt{N}_1$ without decryption. Then, the server updates the pointer in T_G to point to the newly-added node, and finally updates the corresponding entries in the dictionary T_D. To remove the existing edges, the client generates the update token τ_u and submits it to the server. Subsequently, the server looks up in the T_D and recovers the adjacency information of the specified edge. In the following steps, the server frees the node, inserts it into the head of the `free` list and then homomorphically modifies the pointer of the previous node to point to the next node in A_G. Eventually, the server updates the related entries in the dictionary T_D. Note that modifying a specified edge can be easily achieved by removing

the "old" edge first, and adding a "new" edge with the modified length later. After the encrypted graph has been updated, the old query history is deleted and a new empty history will be rebuilt simultaneously.

4.5 Performance Analysis

The time cost of initialization phase is dominated by encrypting all the edges using Paillier cryptosystem and processing all the vertices, thus the time complexity of this part is $O(m+n)$. The generated encrypted graph, which consists of an array and two dictionaries, has the storage complexity $O(m+n)$. In the query phase, we use the Fibonacci heap to speed up the Dijkstra's algorithm, and thus we obtain an $O(n \log n + m)$ time complexity which is optimal among other priority queue optimization techniques (*e.g.*, binary or binomial heap) [7]. During the execution of the secure comparison protocol, the overheads between the server and the proxy are directly related to the number of gates in the comparison circuit. Since many expensive operations of the garbled circuits can be pushed into a pre-computation phase, most of the costs will be relieved from the query phase. By maintaining an auxiliary structure history σ at the server, we can obtain an even better amortization time complexity over multiple queries, *i.e.*, the query time for subsequent queries that can be looked up in the history are (almost) constant. Besides, it is obvious that the time complexity for both addition and removal operations on the encrypted graph are only $O(1)$.

5 Security

We allow reasonable leakage to the server to trade it for efficiency. Now, we provide a formal description of the three leakage functions $\mathcal{L}_1$, $\mathcal{L}_2$ and $\mathcal{L}_3$ considered in our scheme as follows.

- (*Leakage function* $\mathcal{L}_1$). Given a graph G, $\mathcal{L}_1(G) = \{n, m, \#\mathsf{A}_G\}$, where n is the total number of vertices, m is the total number of edges in the graph G and $\#\mathsf{A}_G$ denotes the number of entries (*i.e.*, $m+z$) in the array A_G.
- (*Leakage function*$\mathcal{L}_2$). Given a graph G, a query q, $\mathcal{L}_2(G,q) = \{\mathsf{QP}(G,q), \mathsf{AP}(G,q)\}$, where $\mathsf{QP}(G,q)$ denotes the query pattern and $\mathsf{AP}(G,q)$ denotes the access pattern, both of which are given in the following definitions.
- (*Leakage function* $\mathcal{L}_3$). Given a graph G, an update object u, $\mathcal{L}_3(G,u) = \{\mathsf{id}_v, \mathsf{id}_{new}, \mathsf{next}\}$ is for add updates, and $\mathcal{L}_3(G,u) = \{\mathsf{id}_{del}, \mathsf{next}, \mathsf{prev}\}$ is for delete updates, where id_v denotes the identifier of the start vertex in the newly edge, id_{new} and id_{del} denote the identifiers of the edges to be added and deleted, respectively. prev and next contain the neighboring information (*i.e.*, the identifiers of the neighboring edges) of the edge to be updated. If there are no nodes in A_G before and after the edge to be updated then prev and next are set to $\perp$.

Definition 3 *(Query Pattern). For two shortest distance queries* $q = (s,t), q' = (s',t')$*, define* $\mathsf{sim}(q,q') = (s = s', s = t', t = s', t = t')$*, i.e., whether each of the*

vertices in q matches each of the vertices in q′. Let $\mathsf{q} = (q_1, \ldots, q_\delta)$ *be a sequence of* δ *queries, the query pattern* $\mathsf{QP}(G, q)$ *induced by* q *is a* $\delta \times \delta$ *symmetric matrix such that for* $1 \leq i, j \leq \delta$*, the element in the* i^{th} *row and* j^{th} *column equals* $\mathsf{sim}(q_i, q_j)$*. Namely, the query pattern reveals whether the vertices in the query have appeared before.*

Definition 4 *(Access Pattern). Given a shortest distance query q for the graph G, the access pattern is defined as* $\mathsf{AP}(G, q) = \{\mathsf{id}(c_q), \mathsf{id}(c_q)', \mathsf{id}^*(c_q)\}$*, where* $\mathsf{id}(c_q)$ *denotes the identifiers of vertices in the encrypted result* c_q*,* $\mathsf{id}(c_q)'$ *denotes the identifiers of vertices contained in the dictionary* path *and it reveals the subgraph consisting of vertices reachable from the source* $(\mathsf{id}(c_q) \subset \mathsf{id}(c_q)')$*, and* $\mathsf{id}^*(c_q)$ *denotes the identifiers of the edges with one of its endpoints is the head node of retrieved adjacency lists.*

Discussion. The query pattern implies whether a new query has been issued before, and the access pattern discloses the structural information such as graph connectivity associated with the query. The leakage is not revealed unless its corresponding query has been issued. This is similar to keyword-based SSE schemes, where the leakage (*i.e.*, patterns associated with a keyword query) is revealed only if the corresponding keyword is searched. Fortunately, we can guarantee some level of privacy to the structural information with slightly lower efficiency in our setting, namely, we can add some form of noise (*i.e.*, padding carefully designed fake entries [3,4,15] to each original adjacency list) when generates the encrypted graph. Moreover, in various application scenarios where the data may be abstracted and modeled as sparse graphs (see Table 1), the leakage would not be a big problem. Fully protecting the above two patterns (also forward privacy defined in [30]) without using expensive ORAM techniques remains an open challenging problem, which is our future research focus.

Theorem 1. *If Paillier cryptosystem is CPA-secure and P, F and G are pseudo-random, then the encrypted graph query database system is adaptively* $(\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3)$*-semantically secure in the random oracle model.*

Due to the space limitation, please refer to our technical report for the proof details.

6 Experimental Evaluation

In this section, we present experimental evaluations of our construction on different large-scale graphs. The experiments are performed on separate machines with different configurations: the client runs on a machine with an Intel Core CPU with 4-core operating at 2.90 GHz and equipped with 12 GB RAM, both the server and the proxy run on machines with an Intel Xeon CPU with 24-core operating at 2.10 GHz and equipped with 128 GB RAM. We implemented algorithms described in Sect. 4 in Java, used HMAC for PRF/PRPs and instantiated the random oracle with HMAC-SHA-256. Our secure comparison protocol is built on top of FastGC [11], a Java-based open-source framework.

Table 1. The characteristics of datasets.

Dataset	Type	Vertices	Edges	Storage
Talk	directed	2,394,385	5,021,410	63.3 MB
Youtube	undirected	1,134,890	2,987,624	36.9 MB
EuAll	directed	265,214	420,045	4.76 MB
Gowalla	undirected	196,591	1,900,654	21.1 MB
Vote	directed	7,115	103,689	1.04 MB
Enron	undirected	36,692	367,662	3.86 MB

Table 2. The cost of initialization phase.

Dataset	Time (min.)	Storage (MB)			
		T_G	T_D	A_G	Total
Talk	1042.1	3.6	172.3	1460.5	1636.4
Youtube	460.6	8.93	102	874	984.93
EuAll	76.8	5.37	14.4	122	141.77
Gowalla	307.77	4.69	65.24	556.42	626.35
Vote	17.8	0.14	3.55	30.3	33.99
Enron	69.4	0.88	12.6	107	120.48

Our implementation used the following parameters: we use Paillier cryptosystem with a 1024-bit modulus, the bit length allocated for the diameter l is 16 and the bit length of each random mask is 32. Besides, the FastGC framework provides a 80-bit security level; namely, it uses 80-bit wire labels for garbled circuits and security parameter $c = 80$ for the OT extension.

6.1 Datasets

We used real-world graph datasets publicly available from the Stanford SNAP website (available at https://snap.stanford.edu/data/), and selected the following six representative datasets: *wiki-Talk*, a large network extracted from all user talk pages; *com-Youtube*, a large social network based on the Youtube web site; *email-EuAll*, an email network generated from a European research institution; *loc-Gowalla*, a location-based social network; *wiki-Vote*, a network that contains all the Wikipedia voting data; and *email-Enron*, an email communication network. Table 1 summarizes the main characteristics of these datasets.

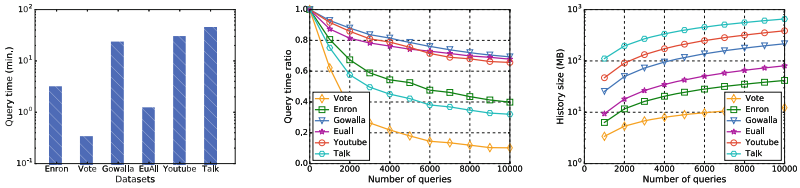
6.2 Experimental Results

Table 2 shows the performance of the initialization phase (one-time cost). As can be seen, the time to encrypt a graph ranges from a few minutes to several hours which is practical. For example, it takes only 17.4 h to obtain an encryption of the *wiki-Talk* graph including 2.4 million vertices and 5.1 million edges. Besides, we note that this phase is highly-parallelizable; namely, we bring the setup time down to just over 30 min by utilizing a modest cluster of 32 nodes. Furthermore, the storage cost of an encrypted graph is dominated by A_G with the total size ranging from 33.99 MB for *wiki-Vote* to 1.60 GB for *wiki-Talk*. We also note that our construction has less storage space requirements compared to Meng *et al.* [24] (*e.g.*, 2.07 GB for *com-Youtube* in [24], whereas our scheme takes 984.93 MB).

We first measured the time to query an encrypted graph without query history. To simulate realistic queries that work in a similar manner with [8], we choose the query vertices in a random fashion weighted according to their outdegrees. The average time at the server (taken over 1,000 random queries) is given in Fig. 4(a) for all encrypted graphs. In general, the results show that the query time ranges from 20.4 s for *wiki-Vote* to 46.4 min for *wiki-Talk*. Additionally, we can obtain an order-of-magnitude improvement in both computation

time and bandwidth by using the packing optimization presented in Sect. 4.3. The actual time for the client to generate the token and decrypt the encrypted result per each query is always less than 0.1 s which is very fast. In addition, about 1.5 KB communication overhead is required to transfer the token and the encrypted result for each query.

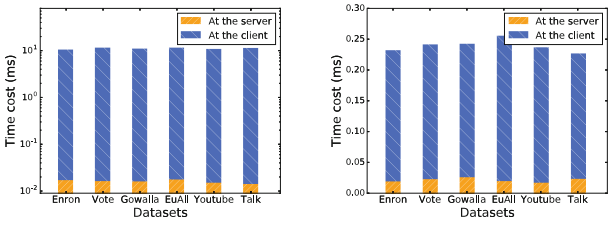
Next, the performance of the query phase with the help of history stored on the server is illustrated in Fig. 4(b) and (c), and a block of 1,000 random executions results in one measurement point in both figures. In Fig. 4(b), the y-axis represents the ratio of the average query time using history to that without using history. Generally, it reflects that the average query time decreases with the increase of the number of queries, because subsequent queries can first be answered by leveraging the history. Furthermore, as can be seen, after 10,000 queries, it obtains about 86% reduction of the query time for *wiki-Vote* compared to that without using history, *i.e.*, it only needs roughly 2.9 s to answer a subsequent query. Figure 4(c) demonstrates the increasing size of history (instantiated by HashMap in our implementation) with the increasing amount of total shortest distance queries.



(a) Query without history (b) Query with history (c) Query with history

Fig. 4. The cost of distance query phase.

Figure 5 shows the execution time (averaged over 1,000 runs) for adding and deleting an edge over all the encrypted graphs. Obviously, both addition and deletion operations are practically efficient and independent of the scale of the graphs. As shown in Fig. 5(a), the time cost at the client side is dominated by generating an encryption of the length of the edge to be updated (roughly 10 ms), while the server side has a negligible running time. Similar results can be obtained in Fig. 5(b) for the delete updates. It only needs about 0.25 ms to delete a specified edge, and the time to generate the delete token at the client side dominates the time cost of the entire process.



(a) Add updates (b) Delete updates

Fig. 5. The time cost of dynamic updates.

In addition, about 0.3 KB and tens of bytes are consumed when performing adding and deleting operations, respectively.

7 Conclusion

In this paper, we designed a new graph encryption scheme–SecGDB to encrypt graph structures and enforce private graph queries. In our construction, we used additively homomorphic encryption and garbled circuits to support shortest distance queries with optimal time and storage complexities. On top of this, we further proposed an auxiliary data structure called *query history* stored on the remote server to achieve better amortized time complexity over multiple queries. Compared to the state-of-the-art, SecGDB returns the exact distance results and allows efficient graph updates over large-scale encrypted graph database. SecGDB is proven to be adaptively semantically-secure in the random oracle model. We finally evaluated SecGDB on representative real-world datasets, showing its efficiency and practicality for use in real-world applications.

Acknowledgment. Qian and Qi's researches are supported in part by National Natural Science Foundation of China (Grant No. 61373167, U1636219, 61572278), National Basic Research Program of China (973 Program) under Grant No. 2014CB340600, and National High Technology Research and Development Program of China (Grant No. 2015AA016004). Kui's research is supported in part by US National Science Foundation under grant CNS-1262277. Aziz's research is supported in part by the NSF under grant CNS-1643207 and the Global Research Lab (GRL) Program of the National Research Foundation (NRF) funded by Ministry of Science, ICT (Information and Communication Technologies) and Future Planning (NRF-2016K1A1A2912757). Qian Wang is the corresponding author.

References

1. Boneh, D., Gentry, C., Halevi, S., Wang, F., Wu, D.J.: Private database queries using somewhat homomorphic encryption. In: Jacobson, M., Locasto, M., Mohassel, P., Safavi-Naini, R. (eds.) ACNS 2013. LNCS, vol. 7954, pp. 102–118. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38980-1_7
2. Cash, D., Jarecki, S., Jutla, C., Krawczyk, H., Roşu, M.-C., Steiner, M.: Highly-scalable searchable symmetric encryption with support for boolean queries. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013. LNCS, vol. 8042, pp. 353–373. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40041-4_20
3. Chase, M., Kamara, S.: Structured encryption and controlled disclosure. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 577–594. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17373-8_33
4. Curtmola, R., Garay, J., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. In: Proceedings of CCS 2006, pp. 79–88. ACM (2006)
5. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numer. Math. **1**(1), 269–271 (1959)

6. Elmehdwi, Y., Samanthula, B.K., Jiang, W.: Secure k-nearest neighbor query over encrypted data in outsourced environments. In: Proceedings of ICDE 2014, pp. 664–675. IEEE (2014)
7. Fredman, M.L., Tarjan, R.E.: Fibonacci heaps and their uses in improved network optimization algorithms. JACM **34**(3), 596–615 (1987)
8. Hahn, F., Kerschbaum, F.: Searchable encryption with secure and efficient updates. In: Proceedings of CCS 2014, pp. 310–320. ACM (2014)
9. Han, W.-S., Lee, S., Park, K., Lee, J.-H., Kim, M.-S., Kim, J., Yu, H.: TurboGraph: a fast parallel graph engine handling billion-scale graphs in a single PC. In: Proceedings of SIGKDD 2013, pp. 77–85. ACM (2013)
10. Harary, F.: Graph Theory. Westview Press, Boulder (1969)
11. Huang, Y., Evans, D., Katz, J., Malka, L.: Faster secure two-party computation using garbled circuits. In: Proceedings of USENIX Security 2011. USENIX (2011)
12. Huang, Y., Malka, L., Evans, D., Katz, J.: Efficient privacy-preserving biometric identification. In: Proceedings of NDSS 2011, pp. 250–267 (2011)
13. Ishai, Y., Kilian, J., Nissim, K., Petrank, E.: Extending oblivious transfers efficiently. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 145–161. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45146-4_9
14. Kamara, S., Papamanthou, C.: Parallel and dynamic searchable symmetric encryption. In: Sadeghi, A.-R. (ed.) FC 2013. LNCS, vol. 7859, pp. 258–274. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39884-1_22
15. Kamara, S., Papamanthou, C., Roeder, T.: Dynamic searchable symmetric encryption. In: Proceedings of CCS 2012, pp. 965–976. ACM (2012)
16. Katz, J., Lindell, Y.: Introduction to Modern Cryptography. CRC Press, Boca Raton (2014)
17. Kolesnikov, V., Sadeghi, A.-R., Schneider, T.: Improved garbled circuit building blocks and applications to auctions and computing minima. In: Garay, J.A., Miyaji, A., Otsuka, A. (eds.) CANS 2009. LNCS, vol. 5888, pp. 1–20. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-10433-6_1
18. Kolesnikov, V., Schneider, T.: Improved garbled circuit: free XOR gates and applications. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008. LNCS, vol. 5126, pp. 486–498. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70583-3_40
19. Lai, R.W.F., Chow, S.S.M.: Structured encryption with non-interactive updates and parallel traversal. In: Proceedings of ICDCS 2015, pp. 776–777. IEEE (2015)
20. Lai, R.W.F., Chow, S.S.M.: Parallel and dynamic structured encryption. In: Proceedings of SECURECOMM 2016 (2016, to appear)
21. Lai, R.W.F., Chow, S.S.M.: Forward-secure searchable encryption on labeled bipartite graphs. In: Gollmann, D., Miyaji, A., Kikuchi, H. (eds.) ACNS 2017. LNCS, vol. 10355, pp. 478–497. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61204-1_24
22. Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., Hellerstein, J.M.: Distributed graphlab: a framework for machine learning and data mining in the cloud. PVLDB **5**(8), 716–727 (2012)
23. Malkhi, D., Nisan, N., Pinkas, B., Sella, Y., et al.: Fairplay-secure two-party computation system. In: Proceedings of USENIX Security 2004, pp. 287–302. USENIX (2004)
24. Meng, X., Kamara, S., Nissim, K., Kollios, G.: GRECS: graph encryption for approximate shortest distance queries. In: Proceedings of CCS 2015, pp. 504–517. ACM (2015)

25. Naor, M., Pinkas, B.: Efficient oblivious transfer protocols. In: Proceedings of SODA 2001, SIAM, pp. 448–457 (2001)
26. Nikolaenko, V., Weinsberg, U., Ioannidis, S., Joye, M., Boneh, D., Taft, N.: Privacy-preserving ridge regression on hundreds of millions of records. In: Proceedings of S&P 2013, pp. 334–348. IEEE (2013)
27. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: Stern, J. (ed.) EUROCRYPT 1999. LNCS, vol. 1592, pp. 223–238. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48910-X_16
28. Pinkas, B., Schneider, T., Smart, N.P., Williams, S.C.: Secure two-party computation is practical. In: Matsui, M. (ed.) ASIACRYPT 2009. LNCS, vol. 5912, pp. 250–267. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-10366-7_15
29. Sarwat, M., Elnikety, S., He, Y., Kliot, G.: Horton: Online query execution engine for large distributed graphs. In: Proceedings of ICDE 2012, pp. 1289–1292. IEEE (2012)
30. Stefanov, E., Papamanthou, C., Shi, E.: Practical dynamic searchable encryption with small leakage. In: Proceedings of NDSS 2014 (2014)
31. Wang, Q., He, M., Du, M., Chow, S.S., Lai, R.W., Zou, Q.: Searchable encryption over feature-rich data. IEEE Trans. Dependable Secure Comput. **PP**(99), 1 (2016)
32. Yao, A.: Protocols for secure computations. In: Proceedings of FOCS 1982, pp. 160–164. IEEE (1982)