PathDHES: Directed Hypergraph Encryption Scheme for Shortest B-Path Queries

Samuel Florin* *sflorin@mit.edu*
Andrei Marginean* *atmargi@mit.edu*
Lukas Rapp* *rappl@mit.edu*
Anastasiia Struss* *struss@mit.edu*

Abstract

We introduce PathDHES, a Directed Hypergraph Encryption Scheme designed to support efficient and privacy-preserving shortest B-path and distance paths queries on encrypted hypergraph data. Motivated by the growing use of hypergraphs in fields such as biology, operations research, and machine learning, our work extends the structured encryption paradigm to capture complex hypergraph semantics. Building on the PathGES scheme for graphs, we adapt its framework to directed hypergraphs by leveraging the Shortest B-Path Tree (SBT) algorithm. We outline efficient algorithms for B-path and distance path recovery, and propose adaptations of heavy-light decomposition to accommodate the output of SBT. Our construction is implemented and evaluated on a sample hypergraph, demonstrating the feasibility of secure and expressive queries over encrypted hypergraph-structured data.

1 Introduction

Hypergraphs, a generalization of graphs, are increasingly used to store data in fields including biology, operations research, machine learning [GLPN93, FHJ+20, FYZ+19]. In light of potential storage constraints, the outsourcing of hypergraph databases to servers may be a viable option [Ior10]. To protect this data and prevent an entire encrypted database from being downloaded during each query, structured encryption (STE) can be used [CK10]. Under STE, structured data is encrypted such that it can be efficiently queried by a user, while leaking some bounded amount of information. However, this scheme offers an efficiency speed-up over Fully Homomorphic Encryption [GKT21].

To the best of our knowledge, hypergraph encryption schemes, where encryption and queries are performed on hypergraphs, have not been explored. But, with the numerous applications of hypergraphs in fields where security might be crucial, such encryption is increasingly important [KCYS23].

1.1. Hypergraph Encryption Applications

In a directed hypergraph, edges are called hyperarcs. They have a head and a tail consisting of vertices. Due to their complexity, directed hypergraphs allow the modeling of data structures that standard graphs cannot represent: one example of this is group interactions [KCYS23] like email exchange [CM20], citation networks [YGA+21], and bitcoin transactions [RJK+17]. In email exchanges, a set of emails can be represented as a hypergraph [CM20], where vertices correspond to people and each hyperarc represents an email, with the senders in the head and the receivers in the tail. Similarly, citation networks [YGA+21] can be modeled as hypergraphs, where vertices represent researchers and each hyperarc corresponds to a citation, with the head containing the authors of the citing paper and the tail containing the authors of the cited paper.

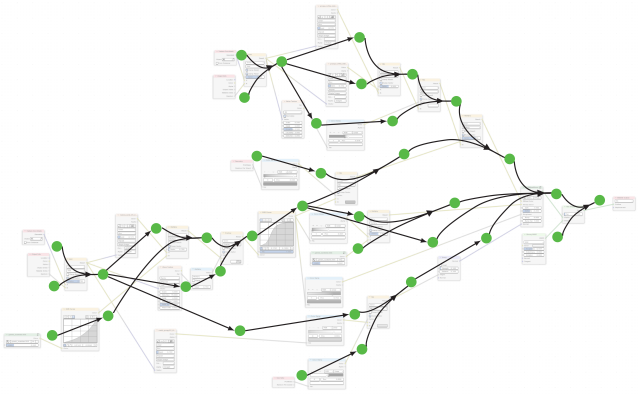

Figure 1.1: Blender Shader Nodes: Visualizes the computation of a table material as a hypergraph [Fou17].

We provide an intuitive use-case for hypergraphs. Consider a user who wants to model a complex process flow, such as material computations in the 3d rendering software, Blender. In this case, hypergraphs provide a useful and efficient way to represent these processes, far beyond the capabilities of traditional graphs. Such a representation is shown in Fig. 1.1 for table material in the 3d scene [Fou17].

Here, the vertices represent data. Vertices without incoming hyperarcs correspond to unprocessed data, such as object properties and textures. Vertices without outgoing hyperarcs store the result of the computation, representing the final material representation. Analogously, intermediate vertices encode partially processed data or temporary resources.

*Massachusetts Institute of Technology, Cambridge, MA, USA

With this view, hyperedges represent transformations or processing operations. Since each hyperedge connects several vertices from its tail to a single vertex from its head, this can be seen as input data needing to be simultaneously processed for a single operation to determine the output. Now, each operation has its own associated costs, such as processing time (i.e., CPU/GPU resources), memory usage, computational complexity, etc.

Intuitively, the cost of a vertex is the minimum processing time at which its data is available for further computation. All input vertices' data must be available to process a hyperarc and calculate the data of its output vertex. Therefore, the output vertex's processing time is determined by the maximal processing time of the input vertices plus the hyperarc's processing time. Notice that the processing time we discuss directly corresponds to the formal definition of a distance in a hypergraph, which is formally defined in Section 4.

Then, a so-called B-path between an input and an output node is just a sequence of operations that transforms initial resources into final outputs, or, in the case of Blender, a rendering pipeline between an input file and the output image. A formal definition of a B-path can be found in section 4 as well. The shortest B-path between two nodes represents the most efficient way to get the desired results from the given input. Finally, the distance path between the two nodes is the most costly sequence of individual operations contained in this shortest B-path.

This interpretation is useful for tasks beyond Blender, in which users might want to be able to recover paths contained in the hypergraph without revealing what they're searching for to the outside world. Think of healthcare treatment planning: a patient might have multiple interconnected conditions and a healthcare provider wants to find the optimal treatment combinations. However, the patient might not want to reveal their conditions, and the hospital might not want to expose their proprietary treatment protocols or let insurance companies learn about specific medical queries. Then, private hypergraph queries would allow doctors to find a way to reach the desired health outcomes given the patient's current conditions without revealing their full medical profile. In short, private hypergraph queries could provide a way to administer personalized treatment planning while maintaining strict medical privacy. This motivates the need for Hypergraph Encryption Schemes.

1.2. Structure of the Paper

In this paper, we define a hypergraph encryption scheme, with queries capturing a hypergraph-notion of SPSP. We begin by briefly discussing graph encryption schemes (Section 2) and the PathGES scheme (Section 3), which will subsequently be transformed into a hypergraph encryption scheme. Then we formally define hypergraph terminology including path generalizations i.e., B-paths and distance paths (Section 4) and present an algorithm for efficiently computing these shortest paths (Section 5).

Next, we layout the goals and definitions for a hypergraph encryption scheme (Section 6) and present our scheme, PathDHES - the generalization of PathGES (Section 7). We conclude with a description of the implementation of our scheme (Section 8).

1.3. Contributions

- Samuel Florin: Worked on adapting the framework and security definitions of a graph encryption scheme to directed hypergraphs, as well as discussing possible applications and datasets for the scheme.
- Andrei Marginean: Discussed practical applications of hypergraphs in intuitively modeling complex processes visually, and offered a concrete interpretation within the context of the Blender Shader Node example. Provided potential motivations for private path queries (both B-paths and Distance Paths) in directed hypergraphs. Organized the GitHub repository.
- Andrei Marginean and Lukas Rapp: Implemented our scheme in Python on top of the original PathGES implementation.
- Lukas Rapp and Anastasiia Struss: Worked on the formal adaptation of the PathGES encryption from graphs to directed hypergraphs, including distance path and shortest B-path encryption.
- Lukas Rapp: Generated hypergraph datasets, including the Blender Shader Node example.
- Anastasiia Struss: Provided the theoretical framework for shortest-path queries in directed hypergraphs and proposed two procedures: SBpath recovers the shortest B-path and dist_path recovers the distance path from the output of the SBT algorithm.

1.4. Future Directions

In this paper, we propose a hypergraph encryption scheme that enables private shortest B-path queries on hypergraphs. Our scheme builds on the state-of-the-art graph encryption scheme PathGES, whose security has already been evaluated.

We therefore expect that our scheme fulfills similar security guarantees. Due to the schemes' technical depth, a thorough analysis is out of this project's scope and subject to further work. Another interesting direction for further work is an analysis of our schemes' storage and search complexity on the server side, which we again expect to be similar to PathGES.

2 Graph Encryption Schemes

One type of STE is a graph encryption scheme (GES) where the structured data represents a graph and queries on the encrypted data reveal underlying properties of the graph [CK10, GKT21, MKNK15]. For example, a GES for

single-pair shortest path (SPSP) queries allows for inputs of the form (u, v) for vertices u, v, outputting the shortest path between those two vertices [KB24, LGZ+22]. Such a scheme is secure when, even if the graph is known to an adversary, only little information (and ideally no information) about the user's queries is leaked. For example, in the case of a GES for Google Maps data, with queries providing the shortest journey between a start and end destination, the road network may be public but we would want users' destinations to remain protected [FGPT24b].

Substantial research has explored secure GES for the SPSP query, including PathGES [FGPT24b], which offers a more secure scheme with similar efficiency as compared to previous schemes [GKT21].

We formalize this notion of a Graph Encryption Scheme.

Definition 2.1 *A **Graph Encryption Scheme (GES)** consists of the following algorithms:*

- $KeyGen : 1^\lambda \to K$*: probabilistically outputting a secret key* K *given a security parameter* λ*.*
- $Encrypt : (K, G) \to ED$ *encrypting a graph database* G *given a key* K*.*
- $Token : (K, q) \to tk$ *which generates a search token* tk *given a key and query* q*.*
- $Search : (ED, tk) \to resp$ *which applies the search token to the encrypted database, returning a response.*
- $Reveal : (K, resp) \to m$ *which converts a response to a plain-text message, given a secret key*

Generally, only the Search algorithm is performed by the server, with the others performed locally by the client. In this way, queries can be performed without the graph needing to be decrypted nor the query revealed to the server. A graph encryption scheme, therefore, ensures only the encrypted database and query, as well as the encrypted plaintext, are handled by the server or exchanged by the client. This structure attempts to minimize the possible leakage of a user's queries. We will formalize this notion of security when discussing the generalization of such a scheme to act on directed hypergraphs. First, we describe a state-of-the-art graph encryption scheme that will serve as the basis for the remainder of this research.

3 PathGES

PathGES is a graph encryption scheme introduced by Falzon et al. to handle shortest-path queries [FGPT24b]. Here, the query q is therefore a pair of vertices (u, v) and the message m should be the shortest path between u and v in the given graph G, with paths taken to be undirected and unweighted. This scheme was developed in response to an attack for a previous shortest-path graph encryption scheme, called GKT [GKT21], whereby information on paths with the same destination node and overlapping edges on the shortest path could be leaked [FP22].

To address this issue, the critical addition of PathGES is the heavy-light decomposition (HLD) [ST81], which can be used to partition a tree into disjoint paths. HLD is applied to the single-destination shortest path (SDSP) [FGPT24b] trees, with these paths further decomposed into smaller segments of consistent length called canonical fragments. The key cryptographic tool used in graph encryption scheme are encrypted multimaps (EMM), which can either be response-revealing, or response-hiding. A multimap simply denotes a map from labels to a set of values, rather than a single value. Here, we define these two types of EMMs and give an overview on how PathGES uses these tools to construct a graph encryption scheme for shortest-path queries [FGPT24b]. The first, which is response-hiding, maps unencrypted labels to encrypted values, whereas the second, which is response-revealing, maps encrypted labels to unencrypted values. We now formalize these definitions.

Definition 3.1 *A **response-hiding encrypted multimap** consists of the following algorithms:*

- $KeyGen : 1^\lambda \to K$*: probabilistically outputting a secret key* K *given a security parameter* λ*.*
- $Encrypt : (K, M) \to EM$ *encrypting a multimap* M *given a key* K*.*
- $Token : (K, lab) \to tk$ *which generates a search token* tk *given a key and label* lab*.*
- $Get : (EM, tk) \to resp$ *which applies the search token to the encrypted multimap, returning a response.*
- $Reveal : (K, resp) \to \{val_i\}_{i \in I}$ *which converts a response to a plain-text set of values* $\{val_i\}_{i \in I}$*, given a secret key*

Definition 3.2 *A **response-revealing encrypted multimap** consists of the following algorithms:*

- $KeyGen : 1^\lambda \to K$
- $Encrypt : (K, M) \to EM$
- $Token : (K, lab) \to tk$
- $Get : (EM, tk) \to \{val_i\}_{i \in I}$ *which uses the search token for the encrypted multimap to get a plain-text set of values* $\{val_i\}_{i \in I}$*, given a secret key*

In this sense, the definition of a response-hiding encrypted multimap looks similar to the structure of the graph encryption scheme above, whereas the response-revealing encrypted multimap processing search tokens directly on the server without using an encrypted response.

Now, PathGES, given a graph G, first precomputes the SDSP trees for each vertex, and then applies the HLD to each of these trees, which are then further fragmented into canonical fragments. Next, a response-revealing EMM M_1 maps the queried pair of vertices to some set of encrypted labels for the canonical fragments that form the shortest path between these two labels. Then, a response-hiding EMM M_2 maps these encrypted labels back to the actual fragment.

Both of these encrypted multi-maps can be hosted on the server. As such, when a user wants to query some pair of vertices, they can generate a search token for the query

and send it to the server. The server can use the encrypted multimaps to compute the encrypted fragment labels from M_1, and then the encrypted fragments themselves from M_2, which are then sent back to the user. The user can decrypt these fragments and piece them back together to get the desired shortest path.

PathGES avoids the attack to which GKT was vulnerable through this use of the heavy-light decomposition, which reduces information leakage while preserving efficiency benefits [FGPT24b]. We reap similar benefits in our similar construction of PathDHES, though this scheme must be adapted to fit the unique structure of the shortest-path problem for hypergraphs.

4 Hypergraphs

In this discussion, our immediate focus is directed hypergraphs, as this is the only type that is important for our encryption scheme. However, this does not imply that undirected hypergraphs cannot be used. The present study leaves undirected hypergraphs encryption schemes for future research directions. We mainly refer to [GLPN93] for terminology and definitions associated with hypergraphs.

Definition 4.1
- *Let V be a finite set of **vertices**, and $\mathcal{E}$ a set of ordered pairs $E = (X, Y)$, where $X, Y \subseteq V$ are subsets of vertices, then elements of $\mathcal{E}$ are called **hyperedges** (or **hyperarcs**) and $H = (V, \mathcal{E})$ is a **directed hypergraph**. $T(E) := X$ and $H(E) := Y$ are **tail** and **head** of E, respectively.*
- *The **backward star** $BS(v)$ of a vertex $v \in V$ is the set of all hyperedges, whose head contains v:*

$$\mathrm{BS}(v) = \{E \in \mathcal{E} \mid v \in H(E)\}.$$

*Analogously, the **forward star** is*

$$\mathrm{FS}(v) = \{E \in \mathcal{E} \mid v \in T(E)\}.$$

For the sake of clarity, we accompany the above definition with Example 4.2 and Figure 4.1.

Example 4.2 *Let $V = \{1, 2, \ldots, 13\}$ be a set of vertices and $\mathcal{E} = \{E_1, \ldots, E_7\}$ with*

$$\begin{aligned} E_1 &= (\{1\}, \{2, 3\}), \quad E_2 = (\{2\}, \{4, 5\}), \\ E_3 &= (\{2, 3\}, \{6, 7\}), \quad E_4 = (\{5, 6\}, \{10\}), \\ E_5 &= (\{8, 9, 10\}, \{4\}), \quad E_6 = (\{11\}, \{12\}), \\ E_7 &= (\{12, 13\}, \{11\}) \end{aligned}$$

be a set of hyperarcs. Then $H = (V, \mathcal{E})$ is a directed hypergraph.

The backward star $\mathrm{BS}(4)$ of vertex 4 consists of hyperarcs E_2 and E_5, whereas the forward star $\mathrm{FS}(4)$ is empty.

Using our hypergraph encryption scheme, one can encrypt a directed hypergraph and outsource it to an untrusted server. In the event that the user is interested in determining a shortest path between two vertices, they can create a token to be sent to the server. The server responds according to the token and, subsequently, the user is able to reconstruct the desired shortest path. But what are paths in directed hypergraphs? We are interested in two types of paths, and we will start with a simple one.

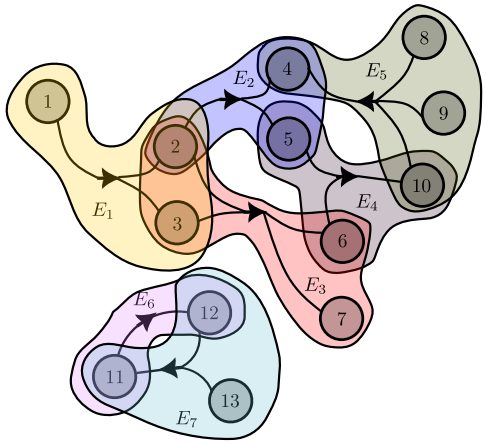

Figure 4.1: A directed hypergraph

Definition 4.3 *A **path** P_{uv} of **length** q in a directed hypergraph $H = (V, \mathcal{E})$ is a sequence*

$$u = v_1, E_{i_1}, v_2, E_{i_2}, \ldots, E_{i_q} v_{q+1} = v,$$

where
- *$v_1, \ldots, v_{q+1} \in V$ are vertices;*
- *$E_{i_1}, \ldots, E_{i_q} \in \mathcal{E}$ are hyperarcs;*
- *$u \in T(E_{i_1}), v \in H(E_{i_q})$ and $v_j \in H(E_{i_j - 1}) \cap T(E_{i_j})$ for $2 \leq j \leq q$.*

*We call P_{uv} **simple** if all hyperarcs $E_{i_1}, \ldots, E_{i_q}$ are distinct. If $v \in T(E_{i_1})$, then P_{uv} is said to be a **cycle**.*

As before, we visualize the definition and give an example.

Example 4.4 *Let H be the directed hypergraph discussed in Example 4.2. Then*

$$2, E_2, 5, E_4, 10 \quad \text{and} \quad 2, E_3, 6, E_4, 10$$

are two simple paths of length 2 from 4 to 10. Notice that

$$2, E_2, 4, E_5, 10$$

is not a path from 4 to 10. Moreover,

$$13, E_7, 11, E_6, 12$$

is a cycle, although the vertices in this path are pairwise distinct.

A B-path is a more sophisticated notion of a path in a directed graph. Here is the point where we deviate from our primary source [GLPN93] related to hypergraphs. While studying hypergraphs, we encountered a deficiency

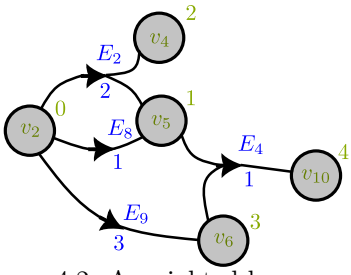

Figure 4.2: A weighted hypergraph

in the intuitive basis of the definition provided in the work [GLPN93]. Additional research led us to [NPA01], which confirmed that the initial definition given in [GLPN93] is too weak and is correct only if additional assumptions on hypergraphs are imposed. Here we use a more natural and intuitive definition of B-paths. We start by defining the B-connection using induction.

Definition 4.5 *Let u be a vertex of a directed hypergraph $H = (V, \mathcal{E})$. Then*

- *u is **B-connected** to itself;*
- *if for some $E \in \mathcal{E}$ all the vertices in $T(E)$ are **B-connected** to u, then each node $v \in H(E)$ is B-connected to u as well.*

Now we can define B-paths.

Definition 4.6 *A **B-path** from vertex u to vertex v in a directed hypergraph H is a minimal sub-hypergraph $H_{uv} = (V_{uv}, \mathcal{E}_{uv})$ of H i.e.*

- *$\mathcal{E}_{uv} \subseteq \mathcal{E}$;*
- *$u, v \in V_{uv} = \bigcup_{(X_i,Y_i)=E_i \in \mathcal{E}_{uv}} (X_i \cup Y_i) \subseteq V$;*

where v is B-connected to u.

Example 4.7 *Let H be the directed hypergraph from Example 4.2. Then $H_{1,7} = (V_{1,7} = \{1, 2, 3, 6, 7\}, \mathcal{E}_{1,7} = \{E_1, E_3\})$ is a B-path from 1 to 7.*

Thus, if we find a B-path H_{uv} from u to v in a directed hypergraph, by the very definition of B-connection, we know that if v is the head of a hyperarc E of H_{uv}, then all vertices from the tail of E are also B-connected to u.

Now we return back to [GLPN93] to discuss weighted directed hypergraphs.

Definition 4.8 *In a **weighted** hypergraph each hyperarc E is assigned a positive real value $w(E)$.*

Definition 4.9 *Let $H = (V, \mathcal{E})$ be a directed weighted hypergraph and $r \in V$ a vertex. The **distance** D_{H_r} is defined recursively for r and all $v \in V \setminus \{r\}$ B-connected to r:*

$$D_{H_r}(r) := 0;$$
$$D_{H_r}(v) := \min_{E \in \mathcal{E} \cap \mathrm{BS}(v)} \{w(E) + \max_{y \in T(E)} \{D_{H_r}(y)\}\}.$$

*A simple path between r and v, whose weight is equal to $D_{H_r}(v)$ is called a **distance path** between r and v.*

```
1   // Initialization
2   for each v in V do D(v) := ∞;
3   for each E_j ∈ E do k_j := 0;
4   Q := {r}; D(r) := 0;
5
6   // Main loop computing D and Pv
7   while Q ≠ ∅
8     select and remove v ∈ Q;
9     for each E_j in FS(v) do
10      k_j := k_j + 1;
11      if k_j = |T(E_j)| then
12        f := max_{u∈T(E_j)}{D(u)};
13        for each u ∈ H(E_j) such that
              D(u) > w(E_j) + f do
14          if u ∉ Q then
15            Q := Q ∪ {u};
16          D(u) := w(E_j) + f; Pv[u] := E_j;
17  return D, Pv;
```
Figure 5.1: Procedure SBT$(H = (V, \mathcal{E}), r \in V)$

5 SBT Procedure

Our main goal is to propose a hypergraph encryption scheme based on PathGES [FGPT24b]. Two crucial parts of PathGES are the SDSP and heavy-light decomposition (HLD) algorithms. Note that the standard SDSP algorithms are specifically designed for graphs and cannot be easily adapted for hypergraphs. In [GLPN93] we found the so-called SBT algorithm for finding minimum weight B-paths in a weighted hypergraph. By slightly extending it, it is possible to construct an alternative to the SDSP algorithm for hypergraphs.

Let $H = (V, \mathcal{E})$ be a weighted directed hypergraph and r a vertex. In the SBT procedure, the so-called **predecessor function** is used: for $v \in V \setminus \{r\}$, $Pv(v)$ points to the hyperarc $E \in \mathrm{BS}(v)$ which precedes node u in the shortest B-path from r to v.

In the initialization step of the SBT procedure, the distance value of every vertex except for the root is set to infinity. In the main loop, we will update the value of any vertex which is B-connected to r. The set Q will always contain vertices B-connected to r, whose forward star has not been examined yet. Finally, variables k_j will help us to make sure that we give distance values to the vertices from the head $H(E_j)$ only if all vertices from the tail $T(E_j)$ are already B-connected to r. We will demonstrate the functionality of the SBT procedure using an example.

Example 5.1 *Consider a directed hypergraph $H = (V, \mathcal{E})$ where $V = \{v_2, v_4, v_5, v_6, v_{10}\}$ is a set of vertices and $\mathcal{E} = \{E_2, E_4, E_8, E_9\}$ with*

$$E_2 = (\{v_2\}, \{v_4, v_5\}), \quad E_4 = (\{v_5, v_6\}, \{v_{10}\}),$$
$$E_8 = (\{v_2\}, \{v_5\}), \quad E_9 = (\{v_2\}, \{v_6\})$$

is a set of hyperarcs. The values $w(E_2) = 2, w(E_4) =$

$1, w(E_8) = 1, w(E_9) = 3$ *make* H *to a valued hypergraph. Set* $r := v_2$*. We will compute values of the predecessor function and the distance for all vertices* $v \in V$ *step by step using the SBT procedure.*

0. In the initialization step we set

$$\begin{array}{cccccc} v_i & v_2 & v_4 & v_5 & v_6 & v_{10} \\ D(v_i) & 0 & \infty & \infty & \infty & \infty \end{array},$$

$$\begin{array}{cccccc} E_j & E_2 & E_4 & E_8 & E_9 \\ k_j & 0 & 0 & 0 & 0 \end{array},$$

$$Q := \{v_2\}.$$

1. Set $u := v_2$ *and update* $Q = \{\}$.

(a) Take $E_2 \in \mathrm{FS}(u) = \{E_2, E_8, E_9\}$*. Update*

$$\begin{array}{cccccc} E_j & E_2 & E_4 & E_8 & E_9 \\ k_j & 1 & 0 & 0 & 0 \end{array}.$$

Since $k_2 = 1 = |\{v_2\}| = |T(E_2)|$*, set*

$$f := \max_{y \in T(E_2) = \{v_2\}} \{D(y)\} = D(v_2) = 0.$$

For each $y \in H(E_2) = \{v_4, v_5\}$ *holds* $D(y) = \infty$ *and* $y \notin Q$*. Hence update* $Q := \{v_4, v_5\}$ *and*

$$\begin{array}{cccccc} v_i & v_2 & v_4 & v_5 & v_6 & v_{10} \\ D(v_i) & 0 & 2 & 2 & \infty & \infty \end{array},$$

$$Pv[v_4] := Pv[v_5] := E_2.$$

(b) Take $E_8 \in \mathrm{FS}(u) = \{E_2, E_8, E_9\}$*. Update*

$$\begin{array}{cccccc} E_j & E_2 & E_4 & E_8 & E_9 \\ k_j & 1 & 0 & 1 & 0 \end{array}.$$

Since $k_8 = 1 = |\{v_2\}| = |T(E_8)|$*, set*

$$f := \max_{y \in T(E_8) = \{v_2\}} \{D(y)\} = D(v_2) = 0.$$

For each $y \in H(E_8) = \{v_5\}$ *holds* $D(y) = D(v_5) = 2 > 1 + 0 = w(E_8) + f$ *and* $y \in Q$*. Hence* Q *stays unchanged, but we update*

$$\begin{array}{cccccc} v_i & v_2 & v_4 & v_5 & v_6 & v_{10} \\ D(v_i) & 0 & 2 & 1 & \infty & \infty \end{array},$$

$$Pv[v_5] := E_8.$$

(c) Take $E_9 \in \mathrm{FS}(u) = \{E_2, E_8, E_9\}$*. Update*

$$\begin{array}{cccccc} E_j & E_2 & E_4 & E_8 & E_9 \\ k_j & 1 & 0 & 1 & 1 \end{array}.$$

Since $k_9 = 1 = |\{v_2\}| = |T(E_9)|$*, set*

$$f := \max_{y \in T(E_9) = \{v_2\}} \{D(y)\} = D(v_2) = 0.$$

For each $y \in H(E_9) = \{v_6\}$ *we have* $D(y) = \infty$ *and* $y \notin Q$*. Hence update* $Q := \{v_4, v_5, v_6\}$ *and*

$$\begin{array}{cccccc} v_i & v_2 & v_4 & v_5 & v_6 & v_{10} \\ D(v_i) & 0 & 2 & 1 & 3 & \infty \end{array},$$

$$Pv[v_6] := E_9.$$

2. Set $u := v_4$ *and update* $Q := \{v_5, v_6\}$*. Since* $\mathrm{FS(u)} = \emptyset$*, continue.*

3. Set $u := v_5$ *and update* $Q := \{v_6\}$.

(a) Take $E_4 \in \mathrm{FS}(u) = \{E_4\}$*. Update*

$$\begin{array}{cccccc} E_j & E_2 & E_4 & E_8 & E_9 \\ k_j & 1 & 1 & 1 & 1 \end{array}.$$

Since $k_2 = 1 < |\{v_5, v_6\}| = |T(E_4)|$*, continue.*

4. Set $u := v_6$ *and update* $Q = \{\}$.

(a) Take $E_4 \in \mathrm{FS}(u) = \{E_4\}$*. Update*

$$\begin{array}{cccccc} E_j & E_2 & E_4 & E_8 & E_9 \\ k_j & 1 & 2 & 1 & 1 \end{array}.$$

Since $k_2 = 2 = |\{v_5, v_6\}| = |T(E_4)|$*, set*

$$f := \max_{y \in T(E_4) = \{v_5, v_6\}} \{D(y)\} = D(v_6) = 3.$$

For each $y \in H(E_4) = \{v_{10}\}$ *we have* $D(y) = \infty$ *and* $y \notin Q$*. Hence update* $Q := \{v_{10}\}$ *and*

$$\begin{array}{cccccc} v_i & v_2 & v_4 & v_5 & v_6 & v_{10} \\ D(v_i) & 0 & 2 & 1 & 3 & 4 \end{array},$$

$$Pv[v_{10}] := E_4.$$

5. Set $u := v_{10}$ *and update* $Q = \{\}$*. Since* $\mathrm{FS}(u) = \emptyset$*, continue.*

6. Since $Q = \emptyset$*, we are done.*

The output of the procedure is

$$\begin{array}{cccccc} v_i & v_2 & v_4 & v_5 & v_6 & v_{10} \\ D(v_i) & 0 & 2 & 1 & 3 & 4 \end{array},$$

```
1  // Auxiliary function
2  Bpath_rec(H = (V, E), v ∈ V, Pv, B):
3    B:= {Pv[v]}
4    if T(Pv[v]) = {r}) then return B;
5    else
6      return B ∪ ⋃_{u∈T(Pv[v])} Bpath_rec(H, u, Pv, {});
7  // End of auxiliary function
8
9  if D(v) = ∞ then return {};
10 D, Pv := SBT(H, r);
11 return Bpath_rec(H = (V, E), v, Pv, D, {});
```

```
1  D, Pv := SBT(H, r);
2  P:=(); u:=v;
3  while u ≠ r:
4    P.add(u, Pv[u]);
5    u:=argmax_{y∈T(Pv[u])} D(y);
6  return P.add(r);
7
8
9
10
11
```

Figure 5.2: Procedures SBpath$(H = (V, \mathcal{E}), r \in V, v \in V \setminus \{r\})$ and dist_path$(H = (V, \mathcal{E}), r \in V, v \in V \setminus \{r\})$

$$Pv[v_4] := E_2, \quad Pv[v_5] := E_8,$$
$$Pv[v_6] := E_9, \quad Pv[v_{10}] := E_4.$$

But we still have not said how to obtain the shortest B-paths from r to vertices $v \in V \setminus \{r\}$. We will construct the shortest B-paths between r and v_{10} starting from v_{10} and going backwards. The predecessor function Pv says that we have to take the hyperarc E_4. This hyperarc has two vertices in the tail $T(E_4) = \{v_5, v_6\}$. They have to be included. Since $Pv[v_5] = E_8, Pv[v_6] = E_9$ and $T(E_8) = T(E_9) = \{r\}$, the shortest B-path from r to v_{10} consists of hyperarcs E_4, E_8, E_9 and all vertices from theirs heads and tails.

In the above example, we pointed out how to recover the shortest B-paths using only the predecessor function. Given a vertex $v \in V \setminus \{r\}$, we take the edge $E = Pv[v]$. Then we look at all the vertices in the tail $T(E)$. In the next step, we take the predecessors of these new vertices and iterate. We stop at the point, when the tail of the current edge is the singleton $\{r\}$. All the computed edges along the way with their tails and heads build the shortest B-path from r to v.

So far, we have not used the distance function computed by the SBT procedure at all. In fact, since the output of the procedure are the distance values from every vertex in the hypergraph to the root and the values of the predecessor function, not the shortest B-paths themselves, we have a much greater flexibility.

In the case when we are interested in a distance path between r and $v \in V \setminus \{r\}$, we use Pv and D in a different way. As before we start by taking the edge $E = Pv[v]$. Then we take a vertex from the tail $T(E)$ with the greatest distance. In the next step, we take the predecessors of this single new vertex and iterate. We stop at the point, when the tail of the current edge is the singleton $\{r\}$. All the computed edges and vertices along the way build the desired path from r to v. In the example above $r = v_2, E_9, v_6, E_4, v_{10} = v$ is a simple path of weight $4 = d(v_{10})$.

Procedures SBpath and dist_path in Figure 5.2 use the output of the SBT procedure to produce a shortest B-path between two vertices r and v and a distance path between two vertices r and v, respectively.

6 Hypergraph Encryption Scheme

We can now extend the definition of a Graph Encryption Scheme naturally to the hypergraph case, allowing users to query an encrypted directed hypergraph as defined above.

Definition 6.1 *A **Directed Hypergraph Encryption Scheme (DHES)** similarly to PathGES consists of the following algorithms:*

- $KeyGen : 1^\lambda \to K$
- $Encrypt : (K, H) \to ED$ *encrypting a hypergraph database H given a key K.*
- $Token : (K, q) \to tk$
- $Search : (ED, tk) \to resp$
- $Reveal : (K, resp) \to m$

Here, these algorithms together allow a client to perform a query on a hypergraph hosted by a potentially untrusted server.

Security for a HES is defined under the assumption of passive adversaries with access to the underlying network. We define a leakage function $\mathcal{L} = (\mathcal{L}_S, \mathcal{L}_Q)$, with $\mathcal{L}_S$ bounding the amount of information leaked during setup and $\mathcal{L}_Q$ bounding the information leaked during queries [FGPT24b, KB24]. We now define the following two games under the real-ideal paradigm, for an adversary $\mathcal{A}$, a hypergraph encryption scheme Σ, and a security parameter λ:

Real$_{\Sigma,\mathcal{A}}(1^\lambda)$: The adversary $\mathcal{A}$ picked a hypergraph H and sends it to the challenger. The challenger then produces $K \leftarrow \text{Keygen}_\Sigma(1^\lambda)$ and computes $\text{ED} \leftarrow \text{Encrypt}_\Sigma(K, H)$. ED is then sent to $\mathcal{A}$, who makes some polynomial number of adaptive queries $(q_1, \ldots, q_n)$, receiving $\text{tk} \leftarrow \text{Query}_\Sigma(K, q)$ from the challenger for each query q. After these queries, $\mathcal{A}$ outputs some bit b.

We now let Sim be a simulator and define the next game:

Ideal$_{\Sigma,\mathcal{A},\text{Sim}}(1^\lambda)$: The adversary $\mathcal{A}$ again selects a hypergraph H that is sent to the challenger. Now, given $\mathcal{L}_S(H)$, Sim constructs and sends ED to $\mathcal{A}$. Now, $\mathcal{A}$ makes some polynomial number of adaptive queries $(q_1, \ldots, q_n)$, receiving $tk \leftarrow \text{Token}_{\text{Sim}}(\mathcal{L}_Q(H, q), ED)$ from the challenger for each query q. After all of these queries, $\mathcal{A}$ again outputs some bit b.

We define Σ as being adaptively $\mathcal{L} = (\mathcal{L}_S, \mathcal{L}_Q)$-sure if

there is some simulator Sim running in probabilistic polynomial time such that, for any ppt adversary $\mathcal{A}$ and $\lambda \geq 1$, there is a negligible function $\mu(\lambda)$ such that

$$|\mathbb{P}[\mathbf{Real}_{\Sigma,\mathcal{A}}(1^\lambda) = 1] - \mathbb{P}[\mathbf{Ideal}_{\Sigma,\mathcal{A},\mathrm{Sim}}(1^\lambda) = 1]| \leq \mu(\lambda).$$

7 PathDHES

Our directed hypergraph encryption scheme builds on the core components of PathGES, specifically its multimap encryption of path fragments. This approach allows us to adopt the security analysis techniques from a state-of-the-art graph encryption scheme whose security has already been evaluated.

Moreover, due to the fact that we built our scheme on top of PathGES, we expect the search and storage complexity of PathDHES to be comparable. A thorough analysis is subject to future work.

To build our scheme on top of PathGES, we need to transform the SBT output for each root vertex $r \in V$ into a list of path fragments, which can be stored in multimaps such that the shortest B-paths can be reconstructed by combining these fragments.

PathGES fragments the output of the SDSP algorithm for every vertex r in two steps: first, the SDSP output, which is a tree with root r, is decomposed into edge-disjoint paths using the HLD. Next, these paths are further decomposed into canonical fragments. The decomposition in PathGES is essential to minimize the scheme query leakage. However, a direct application to hypergraphs is not possible because the output of the SBT algorithm cannot generally be represented as a tree, which is a requirement for the HLD algorithm.

Instead, the SBT output for a hypergraph H with vertices V and root vertex $r \in V$ can be represented as a sub hypergraph H_r of H containing the nodes $\mathcal{M} \subset V$ that are B-connected to r in the original hypergraph: To be precise, H_r is the minimal sub hypergraph of H in which every vertex $v \in \mathcal{M}$ is B-connected to r via the shortest B-path of H between r and v. H_r can easily be obtained from $Pv(\cdot)$ by backtracking every vertex in $\mathcal{M}$ to the root r. Figure 7.1 gives an example for a sub-hypergraph H_1. In this section, we present two encryption schemes that transform the SBT output to trees in a preprocessing step, enabling the fragmentation with HLD and encryption of B-path queries.

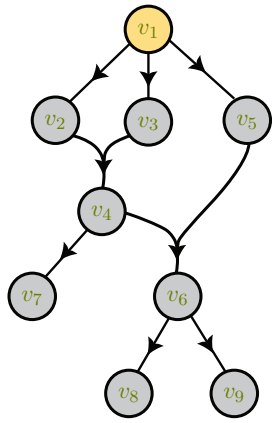

Figure 7.1: Example output of the SBT algorithm for a root vertex 1: the output of the SBT algorithm forms a sub-hypergraph H_1 of the original hypergraph H.

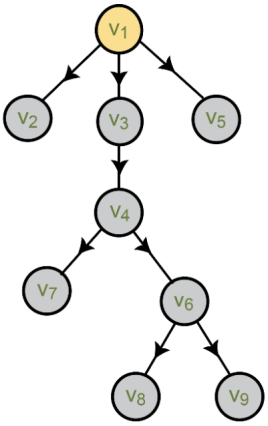

Figure 7.2: Maximal Distance Tree for sub-hypergraph H_1 from Fig. 7.1

7.1. Distance Path Queries

There are applications in which the complete shortest B-path between two vertices is not required, as one may only be interested in the edges that are responsible for the distance between the two vertices. Since the cost of a hyperarc in the distance definition (4.9) calculates the maximum over the cost of all vertices in its tail, the edges and vertices that are responsible for the distance between two vertices form a simple path. This path, which we call the **distance path**

```
1  Max_Dist_Tree_from_SBT_Output(pv, D, r) → dict
2    Keep v ∈ V with D(v) ≠ ∞ in list L;
3    Create tree T_r with r as root;
4
5    for v ∈ L
6      e ← pv(v);
7      Let max_dist be the largest weight in e.
8      Pick rightmost node u such that
           w(u) == max_dist;
9      Add (u, v) to T_r;
10   return T_r;
```

Figure 7.3: Converting sub-hypergraph into maximal distance tree

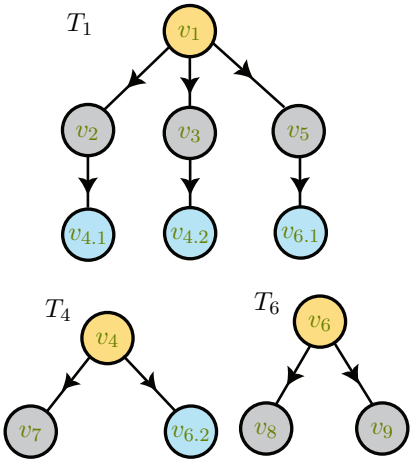

Figure 7.4: Decomposed trees for sub-hypergraph H_1 of Fig. 7.1

in definition (4.9), can be interpreted as the computational bottleneck chain for the shortest B-path (i.e. the least efficient simple path on the most efficient B-path). Optimizing the cost of any edge on this path can decrease the overall distance between the two vertices.

Preprocessing Step Our distance path encryption step requires some additional preprocessing compared to PathGES that takes the sub-hypergraph H_r for each root $r \in V$ of the SBT output and turns it into a tree T_r, which we call the **Maximal Distance Tree**. In T_r, the path from the root to any other node in the tree is the distance path from the root to that node. The pseudocode can be found in Fig. 7.3. Moreover, Fig. 7.2 gives an example of how a sub-hypergraph is converted to a maximal distance tree.

The idea of the algorithm is simple: for each node v in H_r, find the tail node u with the highest weight and add the edge (u, v) to T_r.

Encryption and Reconstruction The encryption of the distance paths is the same as in PathGES, with the one notable difference being that using every vertex as a root, we now encrypt its Maximal Distance Tree obtained from the SBT output instead of the SDSP tree. The process in Fig. 7.5 provides an overview of the encryption and query procedures. The reveal function also agrees with PathGES, and the reconstruction step is the same.

7.2. B-Path Queries

In some applications, querying only the distance paths might not be sufficient, and the user might be interested in the whole shortest B-path between two vertices. For simplicity, we limit our scheme to hypergraphs whose hyperarcs E contain only one vertex in their head, i.e., $|T(E)| = 1$, and outline later ideas on how this scheme can be generalized to arbitrary hyperarcs, which is subject to future work. For example, the hypergraph in Fig. 1.1 fulfills these requirements.

Preprocessing Step Our B-path encryption step requires an additional preprocessing step compared to PathGES that takes the sub-hypergraphs for each root $r \in V$ of the SBT output and decomposes them into trees. The pseudocode to generate this tree can be found in Fig. 7.6.

To extract trees from a sub-hypergraph H_r, we iterate through all vertices. If a vertex v is the head of a hyperarc with multiple tail vertices, the H_r contains a loop that needs to be opened. This is achieved as follows: First, the vertices in the tail of v: $T(v)$ are connected with newly introduced pseudo-vertices v_i. Second, a new tree T_v with v is created. All vertices in H_r that are reachable from v via a sequence of outgoing hyperarcs are connected to this new tree T_v.

Example 7.1 *Fig. 7.4 demonstrates the processing for the example sub-hypergraph in Fig. 7.1. Pseudo-vertices and roots of trees are visualized in blue and yellow, respectively. First, the vertices v_2, v_3 and v_5 are processed and connected via directed simple edges with v_1 in tree T_1.*

Next vertex v_4 is processed whose incoming hyperarc contains multiple vertices (v_2, v_3) in its tail. Hence, v_2 and v_3 are connected with newly introduced pseudo-vertices $v_{4,1}$ and $v_{4,2}$ in T_1 enforcing tree structure. In addition, a new tree T_4 is created with v_4 as root. The following processing of v_6, whose incoming hyperarc also has multiple tail vertices works similarly: v_5 and v_4 are connected to pseudo-vertices $v_{6,1}$ and $v_{6,2}$ of v_6, and a new tree T_6 is generated. Finally, the vertices v_7, v_8, v_9 are added.

The dictionary dict *determines the tree to which each vertex is assigned. For example, v_7 is added to T_4 because it is connected via an incoming edge to v_4, which is already part of tree $T_4 = dict(v_4)$.*

Encryption The encryption of the hypergraph is described in the pseudocode in Fig. 7.7. As in PathGES, every vertex $r \in V$ is considered as root vertex and the SBT algorithm is carried out. Our new preprocessing step in Fig. 7.6 transforms the SBT output into a list of trees of vertices and pseudo-vertices. Similar to PathGES, the HLD algo-

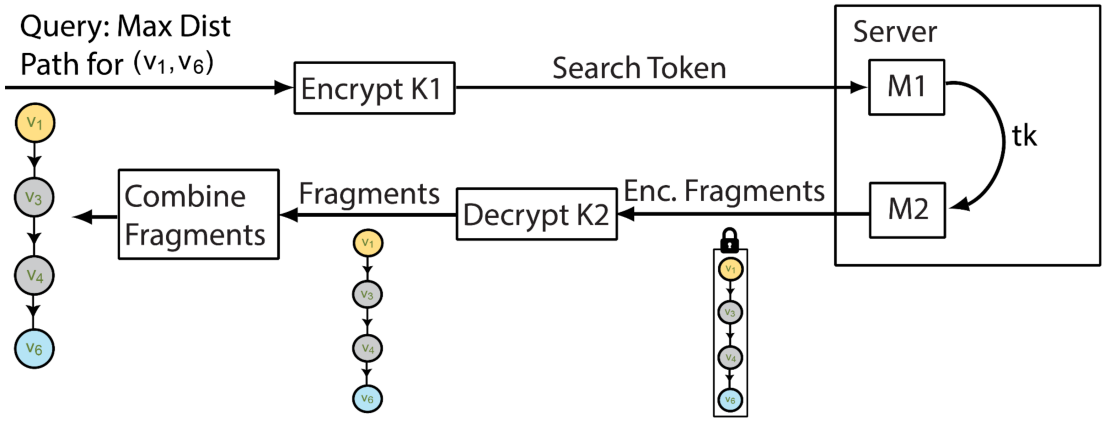

Figure 7.5: Private distance path query between nodes v_1 and v_6 in Fig. 7.1

```
1  Decompose_SBT_Output(pv, D, r) → L_T
2    Sort v ∈ V with D(v) ≠ ∞ in list L_V in
         increasing order of cost D(v);
3    Create tree T_r with r as root;
4    // Stores for each processed vertex v ∈ V to
         which tree dict(v) it belongs.
5    Init tree dictionary dict;
6    // List containing the tree decomposition.
7    Init tree list L_T;
8    dict(r) ← T_r;
9
10   for x ∈ L_V
11     e ← pv(x);
12     if |T(e)| = 1
13       Select single element u ∈ T(e);
14       Add (u, x) to tree dict(u);
15       dict(x) ← dict(u);
16     else
17       // e's tail contains multiple vertices,
             preventing tree structure
18       // → Create tree structure via pseudo
             vertices x_i and start new tree T_x for x.
19       Create tree T_x with x as root;
20       Add T_x to L_T;
21       for i-th tail u ∈ T(e)
22         // Introduce pseudo vertex x_i for vertex x.
23         Add (u, x_i) to tree dict(u);
24       dict(x) = T_x;
25   return L_T;
```

Figure 7.6: Decomposition sub-hypergraph into trees: the algorithm takes the output of the SBT algorithm (see Fig. 5.1): predecessor function Pv and vertex cost function D and the root vertex r as input and outputs a list of edge-disjoint trees that decompose the SBT output.

```
1  Encrypt(K, H)
2    Init multimaps M_1 and M_2;
3    Parse (K_1, K_2) ← K;
4    for r ∈ V
5      Compute SBT output pv, D rooted in r in H;
6      L_T ← Decompose_SBT_Output(pw, D, r);
7      // Iterate through trees in the decomposition L_T
8      for T_x ∈ L_T
9        T_x^D ← ComputeHLD(T, x);
10       for each subpath p_{u,v} ∈ T_x^D in BFS manner
11         Let ℓ be the next power of 2 > |p_{u,v}|;
12         Pad p_{u,v} to length ℓ;
13         for j ∈ [0, ⌈log_2(ℓ)⌉]
14           Let p_{u,v}^{(j)} comprise the last 2^j edges of p_{u,v};
15           M_2[(r, x, u, v, j)] ← p_{u,v}^{(j)};
16           s ← p_{u,v}^{(0)} if j = 0 else p_{u,v}^{(j)} \ p_{u,v}^{(j-1)};
17           for non-pad vertex w in s
18             // PathHES Extension
19             tk ← EMM-RH.Token(K_2, (r, x, u, v, j));
20             M_1[(w, r)] ← [tk];
21             if v ≠ x
22               M_1[(w, r)] ← M_1[(w, r)] ∪ M_1[(v, r)];
23             else if r ≠ x
24               // The start v of the current subpath
                     p_{u,v} is the root of the current
                     tree T_x.
25               // → Combine the fragment lists for
                     all pseudo-vertices x_i of x:
26               for i = 1, ..., |T(pv(x))|
27                 M_1[(w, r)] ← M_1[(w, r)] ∪ M_1[(x_i, r)];
28             Permute M_1[(w, r)];
29   Pad M_1 and M_2 to n^2 log(n) and 4n^2, respectively;
30   EM_1 ← EMM-RR.Encrypt(K_1, M_1);
31   EM_2 ← EMM-RH.Encrypt(K_2, M_2);
32 return (EM_1, EM_2);
```

Figure 7.7: Hypergraph encryption: the algorithm first decomposes a hypergraph H into edge-disjoint trees and fragments them into paths. These fragments are then stored and encrypted into two multimaps, EM_1 and EM_2, using the encryption keys $(K_1, K_2) = K$ and output.

rithm decomposes the trees into edge-disjoint paths, which are then further fragmented.

To understand how our encryption algorithm incorporates the fact that the SBT sub-hypergraph is decomposed into multiple trees, we first need to recap how PathGES generates the multimap M_1, which contains for each query (v, r), the fragment tokens that form the query's response. This multimap is constructed recursively: the token list of the query (w, r), where w is part of the subpath $p_{u,v}$ of the HLD output, is generated by copying the token list of query (v, r) plus the fragment that connects v with w.

Our algorithm builds on this idea by constructing M_1 through a two-level iteration: it iterates through all trees T_x and within each tree through subpaths $p_{u,v}$.

Unlike PathGES, it checks, for each subpath $p_{u,v}$, whether its start v is equal to the root of the current tree x. If this is the case, we do not copy the token list of the query $(v = x, r)$ but copy and merge the token lists of the queries to the pseudonodes of x: $\{(x_i, r)\}_i$. This ensures that a query (w, r) from r to a vertex w after x contains all paths that of the sub-hypergraph H_r that lead from r to x.

Example 7.2 *To illustrate this, consider the encryption of the tree composition in Fig. 7.4, where the algorithm processes subpath $P_{v_6,v_9} = (v_6, v_9)$ in tree T_6. Tree T_1 and T_4 are already fully processed and encrypted. Since the start of the subpath is the root of the current tree, multimap M_1 contains the following fragment tokens for a query from v_1 to v_9:*

$$M_1((v_9, v_1)) = M_1((v_{6,1}, v_1)) \cup M_1((v_{6,2}, v_1)) \cup \{tk\},$$

where tk is an encrypted toke for fragment (v_6, v_9). This ensures that $M_1((v_9, v_1))$ contains all fragments that lead from v_1 to v_6 enabling the reconstruction of the whole B-path after querying.

Search Token and Search The computation of the search token and the lookup of the shortest B-path on the server side are equivalent to the PathGES functions.

Reconstruct Our reveal function agrees with PathGES except for the reconstruction step, in which a list of encrypted fragments is decrypted and stored in a set P.

Note that the fragments are not sorted because of the shuffling in the encryption function. PathGES reconstructs the path by sorting the fragments in the correct order. This can be interpreted as a game of Dominoes: to reconstruct the path for a query from vertex v to u, the algorithm first selects the fragment whose start is equal to v. In the next step, it searches for a fragment whose start equals the end of the previous fragment, and so on.

In contrast, the hypergraph reconstruction algorithm must take into account that the fragments do not necessarily form a chain: this can be seen in the example in Fig. 7.9, which visualizes the fragments for the query v_1 to v_6. Fig. 7.8 shows the pseudocode of the reconstruction algorithm: generalizing the PathGES approach, the reconstruction algorithm has a list of vertices $\mathcal{C}$ that must be connected using the available fragments. For each vertex v in $\mathcal{C}$, the algorithm finds the fragments that start with vertex v and connects v with the fragments.

```
1  Reconstruct(P, v) → H
2    Initialize empty hypergraph H;
3    Add vertex v to H;
4    C ← {v};
5    while P ≠ ∅
6      Take next c from C;
7      if c is pseudo-vertex. I.e., c = "a_i" for a ∈ V
8        if a ∈ H
9          // H already contains a: merge a_i and a
10         e_{a_i} = (v, a_i) ← unique incoming edge of a_i;
11         e_a ← unique incoming hyperarc of a.
12         T(e_a) ← T(e_a) ∪ {v};
13         Remove a_i from H;
14         continue;
15       else
16         Replace a_i with vertex label a in H;
17         c ← a;
18     // Connect all fragments starting with c with c:
19     Find fragments (p_s, p_e) ∈ M ⊂ P s.t. p_s = c;
20     for (p_s, p_e) ∈ M
21       Add directed edge (c, p_s) to H;
22       C ← C ∪ {p_e};
23     Remove fragments in M from P;
24
25   return H;
```

Figure 7.8: Hypergraph Reconstruct B-Path Response: the algorithm takes an unordered list of path fragments P (e.g., left part in Fig. 7.9 and combines them to a B-path starting in vertex v.

In addition, it detects all pseudo-vertices $\{a_i\}_i$ of a vertex a and merges the incoming directed edge of all pseudo-vertices: $\{(v_{a_i}, a_i)\}_i$ into one hyperarc e with head and tail

$$H(e) = a,$$
$$T(e) = \cup_i \{v_{a_i}\}.$$

Example 7.3 *Fig. 7.9 demonstrates the reconstruction of the example response in Fig. 7.10. The algorithm starts with the open vertex $\mathcal{C} = \{v_1\}$, finds all connected fragments and connects them to v_1 and adds their end to $\mathcal{C}$: $\mathcal{C} = \{v_{4,1}, v_{4,2}, v_{6,1}\}$. The pseudo-vertices $v_{4,1}$ is replaced by the actual vertex v_4 merged with $v_{4,2}$ resulting in an incoming hyperarc from v_2 and v_3. The reconstruction of the incoming hyperarc of v_6 works in a similar way until the whole B-path from v_1 to v_6 is recovered.*

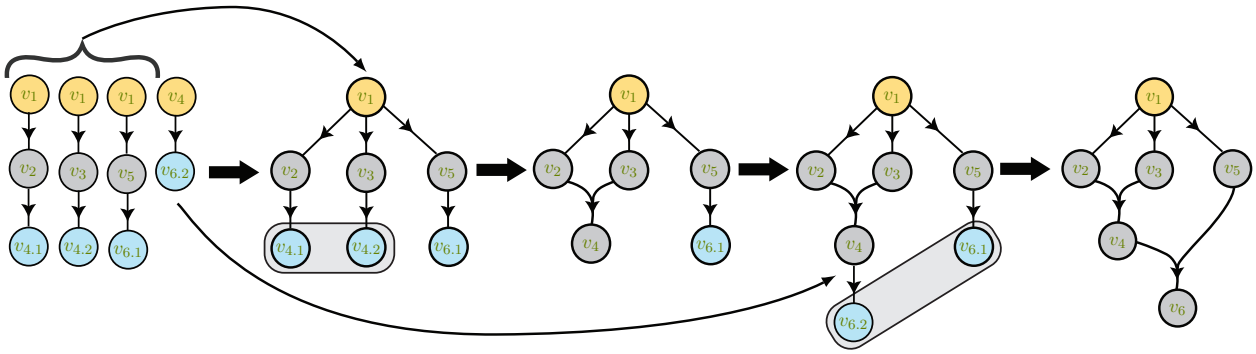

Figure 7.9: Example: Reconstruction B-Path: Visualizes the reconstruction of a B-path from an unordered set of decrypted path fragments.

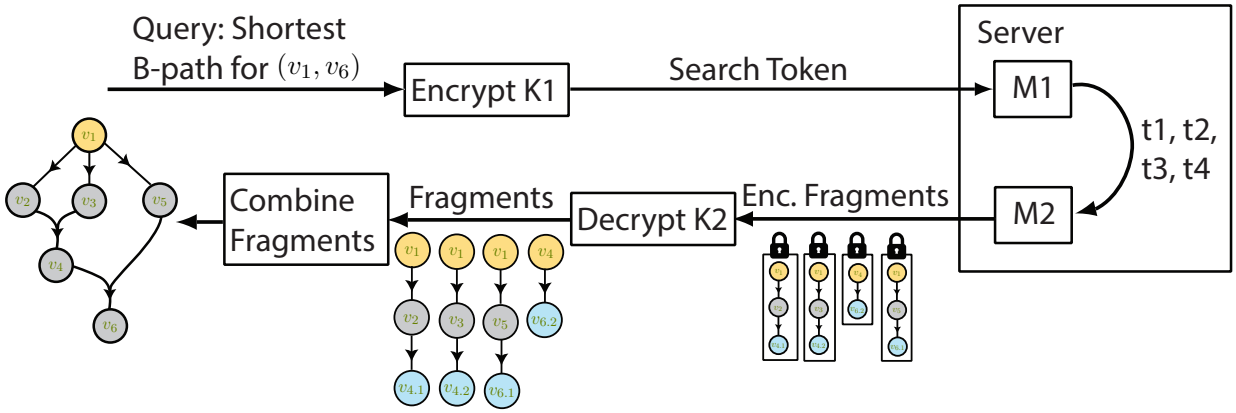

Figure 7.10: Private shortest B-path query: visualizes the steps that user and server carry out to query the shortest B-path from vertex v_1 to v_6.

Scheme The process of privately querying shortest B-paths is visualized in Fig. 7.10:

- The user generates a search token from a query specifying a start and end vertex.
- This search token is sent to the server, which retrieves the corresponding fragment token list in multimap M_1.
- For each fragment token, the server looks up the encrypted path fragments in M_2, which are sent back to the user.
- The user uses K_2 to decrypt each fragment.
- The user uses the algorithm in Fig. 7.8 to reconstruct the B-path.

8 Implementation

We implemented our hypergraph encryption scheme, PathDHES, on top of the original PathGES implementation [FGPT24a]. Our implementation is publicly available on GitHub [FMRS25]. Besides extending the functionality to PathGES, our implementation of PathDHES is backwards compatible, as one can also query paths within regular graphs using the new scheme. As such, PathDHES can be seen as a generalization, or extension, of the original scheme.

To evaluate our implementation, we include two example datasets: the example hypergraph in Fig. 7.1 and the hypergraph that we extracted from Blender's Shader Node editor (see Fig. 1.1). We use the Blender hypergraph as a proxy for a realistic hypergraph dataset of connected computational units because we did not have such a dataset. In both datasets, the weights of all hyperarcs, which represent the processing time of the corresponding computational node, is set to 1. In practical scenarios, actual processing times can be imported into our scheme.

References

[CK10] Melissa Chase and Seny Kamara. Structured encryption and controlled disclosure. In Masayuki Abe, editor, *Advances in Cryptology - ASIACRYPT 2010*, pages 577–594, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[CM20] Philip Chodrow and Andrew Mellor. Annotated hypergraphs: Models and applications. *Applied Network Science*, 5(1):9, December 2020.

[FGPT24a] Francesca Falzon, Esha Ghosh, Kenneth G. Paterson, and Roberto Tamassia. Implementation: PathGES: An efficient and secure graph encryption scheme for shortest path queries. `https://github.com/ffalzon/ges-camera`, 2024.

[FGPT24b] Francesca Falzon, Esha Ghosh, Kenneth G Paterson, and Roberto Tamassia. PathGES: An

efficient and secure graph encryption scheme for shortest path queries. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 4047–4061, 2024.

[FHJ+20] Song Feng, Emily Heath, Brett Jefferson, Cliff Joslyn, Henry Kvinge, Hugh D. Mitchell, Brenda Praggastis, Amie J. Eisfeld, Amy C. Sims, Larissa B. Thackray, Shufang Fan, Kevin B. Walters, Peter J. Halfmann, Danielle Westhoff-Smith, Qing Tan, Vineet D. Menachery, Timothy P. Sheahan, Adam S. Cockrell, Jacob F. Kocher, Kelly G. Stratton, Natalie C. Heller, Lisa M. Bramer, Michael S. Diamond, Ralph S. Baric, Katrina M. Waters, Yoshihiro Kawaoka, Jason E. McDermott, and Emilie Purvine. Hypergraph models of biological networks to identify genes critical to pathogenic viral response. ArXiv preprint: `https://arxiv.org/abs/2010.03068`, 2020.

[FMRS25] Samuel Florin, Andrei Marginean, Lukas Rapp, and Anastasiia Struss. Implementation: PathDHES: Directed hypergraph encryption scheme for shortest B-Path queries. `https://github.com/rapplu/PathDHES`, 2025.

[Fou17] Blender Foundation. Blender agent 327 barbershop demo file. `https://www.blender.org/download/demo-files/`, 2017.

[FP22] Francesca Falzon and Kenneth G. Paterson. An efficient query recovery attack against a graph encryption scheme. Cryptology ePrint Archive, Paper 2022/838, `https://eprint.iacr.org/2022/838`, 2022.

[FYZ+19] Yifan Feng, Haoxuan You, Zizhao Zhang, Rongrong Ji, and Yue Gao. Hypergraph neural networks. ArXiv preprint: `https://arxiv.org/abs/1809.09401`, 2019.

[GKT21] Esha Ghosh, Seny Kamara, and Roberto Tamassia. Efficient graph encryption scheme for shortest path queries. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, pages 516–525, 2021.

[GLPN93] Giorgio Gallo, Giustino Longo, Stefano Pallottino, and Sang Nguyen. Directed hypergraphs and applications. *Discrete applied mathematics*, 42(2-3):177–201, 1993.

[Ior10] Borislav Iordanov. Hypergraphdb: A generalized graph database. In Heng Tao Shen, Jian Pei, M. Tamer Özsu, Lei Zou, Jiaheng Lu, Tok-Wang Ling, Ge Yu, Yi Zhuang, and Jie Shao, editors, *Web-Age Information Management*, pages 25–36, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[KB24] Seyni Kane and Anis Bkakria. A privacy-preserving graph encryption scheme based on oblivious ram, 2024.

[KCYS23] Sunwoo Kim, Minyoung Choe, Jaemin Yoo, and Kijung Shin. Reciprocity in directed hypergraphs: measures, findings, and generators. *Data Mining and Knowledge Discovery*, 37(6):2330–2388, 2023.

[LGZ+22] Meng Li, Jianbo Gao, Zijian Zhang, Chaoping Fu, Chhagan Lal, and Mauro Conti. Graph encryption for shortest path queries with k unsorted nodes. In *2022 IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 89–96, 2022.

[MKNK15] Xianrui Meng, Seny Kamara, Kobbi Nissim, and George Kollios. Grecs: Graph encryption for approximate shortest distance queries. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 504–517, 2015.

[NPA01] Lars Relund Nielsen, Daniele Pretolani, and K Andersen. A remark on the definition of a b-hyperpath. *Department of Operations Research, University of Aarhus, Tech. Rep*, 2001.

[RJK+17] Stephen Ranshous, Cliff A Joslyn, Sean Kreyling, Kathleen Nowak, Nagiza F Samatova, Curtis L West, and Samuel Winters. Exchange pattern mining in the bitcoin transaction directed hypergraph. In *Financial Cryptography and Data Security: FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers 21*, pages 248–263. Springer, 2017.

[ST81] Daniel D Sleator and Robert Endre Tarjan. A data structure for dynamic trees. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 114–122, 1981.

[YGA+21] Naganand Yadati, Tingran Gao, Shahab Asoodeh, Partha Talukdar, and Anand Louis. Graph neural networks for soft semi-supervised learning on hypergraphs. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 447–458. Springer, 2021.