Computational Bounds on Hierarchical Data Processing with Applications to Information Security

Roberto Tamassia Nikos Triandopoulos

rt@cs.brown.edu nikos@cs.brown.edu

Department of Computer Science
Brown University

July 7, 2005

Abstract

Motivated by the study of algorithmic problems in the domain of information security, in this paper, we study the complexity of a new class of computations over a collection of values associated with a set of n elements. We introduce *hierarchical data processing (HDP)* problems which involve the computation of a collection of output values from an input set of n elements, where the entire computation is fully described by a directed acyclic graph (DAG). That is, individual computations are performed and intermediate values are processed according to the hierarchy induced by the DAG.

We present an $\Omega(\log n)$ lower bound on various computational cost measures for HDP problems. Essential in our study is an analogy that we draw between the complexities of any HDP problem of size n and searching by comparison in an order set of n elements, which shows an interesting connection between the two problems. In view of the logarithmic lower bounds, we also develop a new randomized DAG scheme for HDP problems that provides close to optimal performance and achieves cost measures with constant factors of the (logarithmic) leading asymptotic term that are close to optimal. Our lower bounds are general, apply to all HDP problems and, along with our new DAG construction, they provide an interesting –as well as useful in the area of algorithm analysis– theoretical framework.

We apply our results to two information security problems, *data authentication through cryptographic hashing* and *multicast key distribution using key-graphs* and get a unified analysis and treatment for these problems. We show that both problems involve HDP and prove logarithmic lower bounds on their computational and communication costs. In particular, using our new DAG scheme, we present a new efficient authenticated dictionary with improved authentication overhead over previously known schemes. Moreover, through the relation between HDP and searching by comparison, we present a new skip-list version where the expected number of comparisons in a search is $1.25 \log_2 n + O(1)$.

1 Introduction

Deriving lower bounds on the complexity of computational problems is an important, often difficult, task. On one hand, lower bounds put limits on the efficiency that one can hope for, on the other hand, when lower bounds asymptotically match upper bounds derived by known algorithmic constructions, one proves the optimality of these algorithms. In this case, exact analysis and the study of constant factors often replace asymptotic analysis for the need to explore the best possible

algorithmic efficiency for the problem in study. Finally, lower bounds proofs can often provide a characterization of the problem and give an explanation of the computational difficulty that is intrinsic in it.

In this paper, we walk along these lines. We present a unified analysis and design of algorithms and data structures for two important, and seemingly unrelated, algorithmic problems in the area of information security: (i) *data authentication through cryptographic hashing*, i.e., the authentication of membership queries in the presence of data replication at untrusted directories, and (ii) *multicast key distribution using key-graphs*, i.e., the distribution of cryptographic keys by the controller of a dynamic multicast group. We provide logarithmic lower bounds on various time and space cost measures for these problems. In view of these lower bounds, we develop new efficient data structures for these problems and give an accurate analysis of their performance, taking into account constant factors in the leading asymptotic term.

Our unified approach is based on the definition of an abstract and generic class of computational problems, where a directed acyclic graph (DAG) describes the computation of a collection of output values from an input set of n elements. In particular, we introduce *hierarchical data processing (HDP)* problems and study their computational bounds. An HDP problem is related to maintaining, in a systematic way, a dynamic collection of elements along with an associated set of values. Updates of elements generally result in updates of the associated values and queries on elements return subsets of the associated values. In either case, for a problem of this type, the computations that are carried out after an update or a query are all performed sequentially, according to an associated hierarchy, which in turn is expressed by means of a DAG. Accordingly, various cost measures related with the time or space complexity of these computations depend on certain structural properties of the underlying DAG. We define several cost measures for subgraphs of a DAG that characterize the time and space complexity of queries and update operations in an HDP problem. We prove $\Omega(\log n)$ lower bounds on these cost measures using a reduction from the problem of searching by comparisons in an ordered set. Furthermore, in view of these logarithmic lower bounds, we design a new randomized DAG scheme for HDP problems that is based on a variation of the skip-list. Our new DAG scheme achieves computational efficiency which is very close to the optimal with respect to the leading logarithmic terms.

We motivate the study of the class of HDP problems by two seemingly different algorithmic problems related with applications in the area of information security. Both problems share the property that the involved computations are performed according to the hierarchy induced by a DAG, that is, they involve some type of hierarchy-based data processing. The first application is the model of *hash-based authenticated data structures* that has been recently proposed for data authentication in distributed and untrusted environments. In this model, queries on a data set are answered by untrusted entities and not by the source and owner of the data set, in a way where each answer contains information that can been used to provide an cryptographic proof about the validity of the answer. We focus on *authenticated dictionaries*, where membership queries are asked about a set, and consider the case where authentication is achieved by hierarchically applying cryptographic hashing over the data set. The second application is the problem of *multicast key distribution using key-graphs*. Here, a dynamic group of users share a set of keys so that they can securely implement multicast transmissions by means of private-key encryption. Using key-graphs, any update in the group memberships results in changing and securely redistributing to the users some subset of the keys.

We obtain new results for these two information security problems by first showing that they can be modeled by a HDP problem and by appropriately applying to their domain the general lower bounds and our new DAG scheme. For data authentication through cryptographic hashing, we model (the design of) any authenticated dictionary as an HDP problem where only various costs

related to authentication, the *authentication overhead*, are considered. Using our framework, we prove that any hash-based authenticated dictionary incurs logarithmic on its size authentication cost in the worst case and we present a new authenticated dictionary with authentication cost closer to the theoretical optimal. Similarly, we show that a broad class of multicast key distribution protocols can be viewed as HDP problems and we thus prove that any such protocol has a logarithmic on the size of the group time or communication complexity in the worst case.

Interestingly, the proof of our lower bounds is based on a reduction from the problem of searching by comparison in an ordered set of size n to an HDP problem of the same size, which establishes a relationship between these two types of problems. This relationship further characterize the computational difficulty of the security problems that we study. Additionally, through this reduction, our new DAG scheme provides us with a new skip-list version where searches have expected cost that is closer to the theoretically optimal. Our study of HDP not only offers a unified treatment of the two information security problems in consideration, but also provides an new interesting theoretical framework in problem analysis and algorithm and data structure design. Indeed, the class of HDP problems can in principle be used to model any hierarchy-based data processing problem in terms of lower bounds and efficient constructions.

In the rest of the section we briefly describe the class of problems that we study, we summarize our contributions in the analysis of the complexity of HDP problems and in data structure design and we present an overview of the new results that we get by applying our framework in the two algorithmic problems in the domain of information security. We also review previous work.

1.1 Hierarchical Data Processing

We introduce an abstract and generic class of problems, the *hierarchical data processing* (HDP) problems. This class of problems models computations over a dynamic set of n elements, where all computations are lead by the hierarchy induced by a directed acyclic graph (DAG), so that various costs depend on certain structural properties of this underlying DAG. These problems share the following characteristics. Associated with the elements is a structured collection of values, organized according to the DAG. As elements change over time, the values are accordingly updated. Additionally, queries on elements are issued, where typically the answer of a query is a subset of the associated values.

More specifically, in an HDP problem a dynamic collection of elements, maintained by update operations, is associated with a set of values, which in turn is organized by means of a DAG. Data processing involves the computation and update of the set of values after every update in the elements. In addition, queries on elements are answered, again, by accessing and processing data values through traversals of the DAG. All operations on elements and associated values involve processing of data according to the hierarchy that the DAG induces. Additionally, the complexity of all computations and all costs parameters related to the problem depend on specific properties and the structure of the DAG in use.

Previous Work. We are not aware of any previous systematic study of the class of HDP problems.

Our Results. We define several cost measures for subgraphs of a DAG that characterize the space and time complexity of queries and update operations in an HDP problem. For a problem of size n, we relate each of these cost measures to the number of comparisons performed in the search of an element in an ordered set of size n. Through this reduction, we prove an $\Omega(\log n)$ lower bound on the space and time complexity of query and update operations in any HDP problem. We also show that for this class of problems trees are optimal DAG structures compared with general DAGs. In view of the logarithmic lower bound and the optimality of tree DAGs for HDP problems, we also design a new randomized DAG, called *multi-way skip-list DAG scheme*, which is based on

a variation the skip-list data structure. We give a detailed analysis of the cost measures of our DAG scheme taking into account the constant factor on the leading asymptotic term and we show that it achieves structural properties that are close to optimal. Accordingly, we get that multi-way skip-lists, when used to lead the computations of any HDP problem, achieve efficiency in terms of time and space complexities. Hierarchical data processing problems and their computational bounds are presented in Section 2. In Section 3, we present and analyze our multi-way skip-list DAG scheme.

1.2 Data Authentication and Authenticated Data Structures

Data structures are designed to organize a collection of data, so that searching in the collection and answering queries about the data are performed efficiently. Implicitly, the owner and the user of the data structure are assumed to be the same entity. An important security problem arises when this assumption is abandoned. For instance, with the advent of Web services and pervasive computing, a data structure can be controlled by an entity different than the owner or the user of the data. Additionally, data replication applications achieve computational efficiency by caching data at servers near users, but they present a major security challenge. Namely, how can a user verify that the data items replicated at a server (e.g., the answer to a query) are the same as the original ones generated by the data source? The naive approach of directly applying traditional message authentication techniques (like digital signatures or message authentication codes) in this data authentication problem either fails or lacks efficiency and scalability. For instance, digitally signing the answer to any query of a data structure is not viable for the set of possible answers can be unbounded, especially for dynamic data sets that evolve over time.

Authenticated data structures (ADSs) capture exactly this, non-conventional (and closer to today Internet's reality) new data structuring paradigm. A data structure is controlled by an entity that is *not* the creator of the data. That is, not the data source but rather an entity *not trusted* to a user issuing queries answers these queries. ADSs solve the security problem of data authentication in these untrusted distributed environments and receive more and more attention. They support *authenticated queries*: they allow the user to verify the validity of the answer, i.e., either accept the answer as authentic or reject it.

In particular, an ADS is a distributed model of computation where a *directory* answers queries on a data structure on behalf of a trusted *source* and provides to the *user* a cryptographic proof of the validity of the answer. The source signs a *digest* (i.e., a cryptographic summary) of the content of the data structure and sends it to the directory. Ideally, the digest has $O(1)$ size. This signed digest is forwarded by the directory to the user together with the proof of the answer to a query. To verify the validity of answer, the user computes the digest of the data from the answer and the proof, and compares this computed digest against the original digest signed by the source.

Cost parameters for an ADS include the space used (for the source and directory), the update time (for the source and directory), the query time (for the directory), the digest size, the proof size and the verification time (for the user). In the important class of *hash-based authenticated data structures*, the digest of the data set is computed by *hierarchical hashing*, i.e., by hierarchically applying a cryptographic hash function over the data set.

Previous Work. Early work on ADSs was motivated by the *certificate revocation* problem in public key infrastructure and focused on *authenticated dictionaries*, on which membership queries are issued. The *hash tree* scheme introduced by Merkle [24] can be used to implement a static authenticated dictionary. A hash tree T for a set S stores cryptographic hashes of the elements of S at the leaves of T and a value at each internal node, which is the result of computing a cryptographic hash function on the values of its children. The hash tree uses linear space and has

$O(\log n)$ proof size, query time and verification time. A dynamic authenticated dictionary based on hash trees that achieves $O(\log n)$ update time is described in [28]. A dynamic authenticated dictionary that uses a hierarchical hashing technique over skip-lists is presented in [13]. This data structure also achieves $O(\log n)$ proof size, query time, update time and verification time. Other schemes based on variations of hash trees have been proposed in [3, 9, 19]. The software architecture and implementation of an authenticated dictionary based on skip-lists is presented in [15]. A distributed system realizing an authenticated dictionary and an empirical analysis of its performance in various deployment scenarios are described in [10]. The authentication of distributed data using web services and XML signatures is investigated in [31] and *prooflets*, a scalable architecture for authenticating web content based on authenticated dictionaries, are introduced in [36].

An alternative approach to the design of authenticated dictionary, based on the *RSA accumulator*, is presented in [14]. This technique achieves constant proof size and verification time and provides a tradeoff between the query and update times. For example, one can achieve $O(\sqrt{n})$ query time and update time. In [1], the notion of a *persistent authenticated dictionary* is introduced, where the user can issue historical queries of the type "was element e in set S at time t". A first step towards the design of more general ADSs (beyond dictionaries) is made in [8] with the authentication of relational database operations and multidimensional orthogonal range queries. In [22], a general method for designing ADSs using hierarchical hashing over a search graph is presented. This technique is applied to the design of static ADSs for pattern matching in tries and for orthogonal range searching in a multidimensional set of points. Efficient ADSs supporting a variety of fundamental search problems on graphs (e.g., path queries and biconnectivity queries) and geometric objects (e.g., point location queries and segment intersection queries) are presented in [16]. This paper also provides a general technique for the design of ADSs that follow the *fractional cascading* paradigm. Work related to ADSs includes [4, 7, 11, 20, 21]. Related to ADSs is also recent work on zero knowledge sets and consistency proofs [25, 29] that model data authentication in a more adversarial environment, where the data source is not considered trusted per-se. These schemes use significantly more computational resources than ADSs.

No previous study on the cost of ADSs has been made. As we will see later, from our study we get the following. The computational overhead incurred by an ADS over a non-authenticated data structure consists of: (1) the additional space used to store authentication information (e.g., signatures and hash values) in the data structure and in the proof of the answer, and (2) the additional time spent performing authentication computations (e.g., computing signatures and cryptographic hashes) in query, update and verification operations. Since cryptographic operations such as signatures and hashes are orders of magnitude slower than comparisons and a single hash value is relatively long, the authentication overhead dominates the performance of an ADS. All the existing hash-based authenticated dictionaries have logarithmic query, update and verification cost and logarithmic proof size.

We thus address the following question: can the authenticated version of a data structure (i.e., authenticating membership queries) be more efficient then the non-authenticated one (i.e., answering membership queries)[1]? Considering only dictionaries, the existence of tree-based schemes, schemes with logarithmic authentication costs (proof size, verification time, update/query time) where hashing is performed according to the search tree (data structure) (e.g., [28, 13]), shows that answering authenticated membership queries is at most as expensive as answering non-authenticated membership queries using search trees. But can we do better? Already in the introduction of authenticated dictionaries, Naor and Nissim [28] posed as an open problem the question of whether

[1]Clearly, for the question to make sense, we completely separate the notions of authenticating and answering membership queries.

one can achieve sublogarithmic authentication overhead for dictionaries. We answer this question negatively for hash-based ADSs.

Our Results. We present the first study on the cost of ADSs, focusing on dictionaries. We model a hash-based dictionary ADS as an HDP problem. We consider a very general authentication technique where hashing is performed over the data set in *any possible* way and where *more than one* digests of the data structure are digitally signed by the source. Applying our results from Section 2 in this domain, we prove the first nontrivial lower bound on the authentication cost for dictionaries. In particular, we show that in any hash-based authenticated dictionary of size n where the source signs k digests of the data set, any of the authentication costs (update/query time, proof size or verification time) is $\Omega(\log \frac{n}{k})$ in the worst case. Thus, the optimal trade-off between signature cost and hashing cost is achieved with $O(1)$ signature cost and $\Omega(\log n)$ hashing cost. In this case, we show that hash-based authenticated dictionaries of size n incur $\Theta(\log n)$ complexity.[2] We also show that among all DAGs, trees are optimal when used for hashing. Also, our skip-list structure from Section 3 can be used to implement an efficient authenticated dictionary, where certain cost parameters are reduced with respect to previous constructions. Our results on authenticated dictionaries are described in Section 4.

1.3 Multicast Key Distribution

Multicast key distribution (or multicast encryption) is a model for realizing secrecy in multicast communications among a dynamic group of n users. To achieve secrecy, one needs to extend the conventional point-to-point encryption schemes to the multicast transmission setting. Namely, the users share a common secret key, called *group-key*, and encrypt multicast messages with this key, using a secret-key (symmetric) encryption scheme. When changes in the multicast group occur (through additions/deletions of users), in order to preserve (forward and backward) security, the group-key needs to be securely updated.

In general, a *group controller* (physical or logical entity) is responsible for distributing an initial set of keys to the users. Each user possesses his own secret-key (known only to the controller), the group-key and a subset of other keys. Upon the insertion/removal of a user into/from the group, a subset of the keys of the users are updated. Namely, new keys are encrypted by some of the existing keys so that only legitimate users in the updated group can decrypt them. The main cost associated with this problem is the number of messages that need to be transmitted after an update. Additional costs are related to the computational time spent for key encryptions and decryptions.

Previous Work. Many schemes have been developed for multicast key distribution. We focus on the widely studied *key-graph* scheme, introduced in [39, 40], where constructions are presented for key-graphs realized by balanced binary trees such that $O(\log n)$ messages are transmitted after an update, where n is the current number of users. Further work has been done on key-graphs based on specific classes of binary trees, such as AVL trees, 2-3 trees and dynamic trees. See, e.g., [17, 12, 34]. In [6], the first lower bounds are given for a restricted class of key distribution protocols, where group members have limited memory or the key distribution scheme has a certain structure-preserving property. In [37], an amortized logarithmic lower bound is presented on the number of messages needed after an update. The authors prove the existence of a series of $2n$ update operations that cause the transmission of $\Omega(n \log n)$ messages. Recently, a similar amortized logarithmic lower bound has been shown in [26] for a more general class of key distribution protocols, where one can

[2] Interestingly, the use of a different cryptographic technique, namely the use of one-way accumulators in [14], achieves constant proof size and verification time, but involves more expensive update costs and more expensive primitive operations (exponentiations) with respect to the efficient cryptographic hashing.

employ a pseudorandom generator to extract (in a one-way fashion) two new keys from one key and one can perform multiple nested key encryptions. Pseudorandom generators for this problem were first described in [5], where the number of messages are decreased from $2\log n$ to $\log n$.

Our Results. We show that the multicast key distribution problem using key-graphs is an HDP problem. Applying our results from Sections 2 and 3 to this domain: (*i*) we perform the first study of general key-graphs (other than trees) and show that trees are optimal structures; (*ii*) we prove an exact worst-case logarithmic lower bound on both the communication cost (number of messages) and the computational cost (cost due to encryption/decryption) of any update operation, the first of this type; and (*iii*) we present a new scheme (tree DAG) that achieves costs closer to the theoretical optimal. Note that we give the first lower bounds on the encryption/decryption costs and that our lower bound proof is more generic since it depends on *no certain series* of update operations. In essence, we present the first *exact*, *worst case* logarithmic lower bound for the communication cost of the multicast key distribution problem. All of the previously known lower bounds are *amortized*, i.e., they prove the existence of a sequence of updates that include an expensive (of at least logarithmic cost) one. In contrast, we prove the existence of a single update of at least $\lfloor \log n \rfloor$ communication cost for any instance of the problem. Our lower bound holds also for protocols that use pseudorandom generators or multiple encryption, as in the model studied in [26]. These results are described in Section 5.

1.4 Skip-Lists

The skip-list, introduced in [32, 33], is an efficient randomized data structure for dictionaries. It is well known that skip-lists correspond to some tree (search) structure, so they constitute an optimal DAG structure for HDP problems.

Previous Work. In [32, 33] it is shown that the expected number of comparisons for a search in a skip-list is $(\log_2 n)/(p\log_2 \frac{1}{p}) + O(1)$, where p is a probability parameter. In the same work, an improved – in terms of number of comparisons – skip-list version gives $\frac{1-p^2}{p\log_2 \frac{1}{p}}\log_2 n + O(1)$ expected comparisons for a search. We are not aware of any improved skip-list based scheme with better logarithmic constant.

Our Results. Through the relation between HDP and searching by comparison we obtain a new version of skip-lists, where the expected number of comparisons in a search is $1.25\log_2 n + O(1)$, which is closer to the theoretically optimal up to an additive constant term. In particular, for p being the probability parameter of the skip-list, our skip-list version reduces the expected number of comparisons down to $\frac{(1-p)(1+p^2)}{p\log_2 \frac{1}{p}}\log_2 n + O(1)$ [3]. Details for the new multi-way skip-list DAG and the corresponding new skip-list version are presented in Section 6.

1.5 Organization of the Paper

The rest of the paper is organized as follows. In Section 2, we introduce the problems of hierarchical data processing, study their complexity, prove lower bounds on various associated costs and show that tree DAGs are optimal when used for HDP problems. In view of these results, we focus on tree DAGs and in Section 3 we design and analyze our new multi-way skip-list DAG scheme for HDP problems that is based on skip-lists and achieves performance close to optimal. In Section 4 we apply our results to data authentication problem through hashing and in Section 5 to the problem of multicast key distribution using key-graphs. Finally, in Section 6 we present the new improved skip-list version. We conclude and discuss open problems in Section 7.

[3] For $p = \frac{1}{2}$, the logarithmic constant of the expected number of comparisons for a search drops from 1.5 to 1.25.

2 Hierarchical Data Processing and its Theoretical Limits

In this section, we define several structural cost measures for subgraphs of a DAG and prove lower bounds them. Such cost measures are related to the computational complexity of operations in a class of problems that we call *hierarchical data processing* problems. Our lower bounds are naturally translated to complexity results of this type of problems.

2.1 DAG Scheme

Before we introduce our new concepts, we define some graph notation. Let $G = (V, E)$ be a directed acyclic graph. For each node v of G, $indeg(v)$ denotes the in-degree of v, i.e., the number of incoming edges of v, and similarly, $outdeg(v)$ denotes the out-degree of v, i.e., the number of outgoing edges of v. A *source* of G is a node v such that $indeg(v) = 0$. A *sink* of G is a node v such that $outdeg(v) = 0$. We denote with $V_{source} \subset V$ the set of *source* nodes of G and with $V_{sink} \subset V$ the set of sink nodes of G. For any edge $e = (u, v)$ in E, node u is a *predecessor* of v and node v is a *successor* of u. A directed path in G connecting node u to some other node is called *trivial*, if every node in the path other than u has in-degree 1. A subgraph H of G is said to be *weakly connected*, if it is connected when one ignores edge directions, *non-trivial*, if it contains no trivial paths, and *complete*[4], if all edges in G connecting nodes of H are edges of H. An edge in G connecting nodes of H that is not an edge of H is called a *missing* edge of H and by $\bar{H}$ we denote the complete graph that we take if we add in H all of its missing edges. For any node v in a DAG G, G_v denotes the subgraph in G whose nodes are connected with v through directed paths that start at v, i.e., they are successor nodes of v in the transitive closure of G and whose edges belong in these directed paths. We say that subgraph G_v is *reachable* from node v. Note that for any node v, graph G_v is a complete graph (having no missing edges).

Definition 1 (DAG scheme). *A* DAG scheme Γ *is a quadruple* (G, S, n, k)*, where* $G = (V, E)$ *is a directed acyclic graph without parallel edges,* $S \subset V$ *is a set of* special *nodes and* n *and* k *are integers such that: (i)* $|V_{source}| = n$*; (ii)* $|V|$ *is bounded by a polynomial in* n*; and (iii)* $|S| = k$*,* $S \supset V_{sink}$ *and* $S \cap V_{source} = \emptyset$*.*

That is, G has n source nodes and $poly(n)$ nodes in total; G contains a subset of k non-source nodes, called special nodes, that includes all the sink nodes of G.

2.2 Cost Measures of a DAG Scheme

We first define three *structural cost measures* for any weakly connected subgraph of a DAG.

Definition 2 (Structural cost measures for subgraphs). *Let* $H = (V_H, E_H)$ *be a weakly connected subgraph of a DAG* G*. We define the following with respect to* G*:*

1. *The* node size $size(H)$ *of* H *is the number of nodes in* H*, i.e.,* $size(H) = |V_H|$*;*
2. *the* degree size $indeg(H)$ *of* H *is the sum of the in-degrees (with respect to* G*) of the nodes of* H*, i.e.,* $indeg(H) = \sum_{v \in H} indeg(v)$*;*
3. *the* combined size $comb(H)$ *of* H *is the sum of its node and degree sizes, i.e.,* $comb(H) = size(H) + indeg(H)$*;*
4. *the* boundary size $bnd(H)$ *of* H *is the number of edges of* G *that enter nodes of* H *but are not in* H*.*

Whenever it is not clear from the context with respect to which DAG G a structural cost measure is defined, we use subscripts; e.g., $indeg_H(\cdot)$ denotes the degree size with respect to graph H.

[4]The used term is not confusing: cliques are not defined for DAGs.

Using the above structural cost measures of subgraphs of DAGs, we define three *cost measures* for a DAG scheme Γ.

Definition 3 (Cost measures of DAG scheme). *Given a DAG scheme $\Gamma = (G, S, n, k)$, let s be a source node of G. Let P_s^t denote the set of directed paths connecting node s to node t in G. The* associated path *π_s of s is a directed path in G_s that starts at s, ends at a node of S and has the* minimum *combined size among all such paths, i.e., $comb(\pi_s) = \min_{u \in S, p \in P_s^u} comb(p)$. We define the following cost measures for Γ:*

1. *the* update cost *$\mathcal{U}(\Gamma)$ of Γ is $\mathcal{U}(\Gamma) = \max_{s \in V_{source}} comb(G_s)$, i.e., the maximum, over all source nodes in V_{source}, combined size of the subgraph G_s reachable from s;*
2. *the* query cost *$\mathcal{Q}(\Gamma)$ of Γ is $\mathcal{Q}(\Gamma) = \max_{s \in V_{source}} comb(\pi_s) = \max_s \min_{u \in S, p \in P_s^u} comb(p)$, i.e., the maximum, over all source nodes in V_{source}, combined size of the associated path π_s of s;*
3. *the* sibling cost *$\mathcal{S}(\Gamma)$ of Γ is $\mathcal{S}(\Gamma) = \max_{s \in V_{source}} bnd(\pi_s)$, i.e., the maximum, over all source nodes in V_{source}, boundary size of the associated path π_s of s.*

Note that the associated path of a source node s (which is not necessarily unique) is generally not the minimum boundary size path from s to a node in S, because for general graphs minimum combined size does not imply minimum boundary size. In our study, sibling costs defined as above are thus linked to query costs; we justify this choice at the end of the section. For trees, this asymmetry disappears.

The following lemma states some useful facts about the structural cost measures of subgraphs of a DAG and the cost measures of a DAG scheme.

Lemma 1. *Let $\Gamma = (G, S, n, k)$ be any DAG scheme with update cost $\mathcal{U}(\Gamma)$, query cost $\mathcal{Q}(\Gamma)$ and sibling cost $\mathcal{S}(\Gamma)$, H be any weakly connected subgraph of G, $\bar{H}$ be the complete subgraph that corresponds to H and p be any directed path. We have:*

1. *$comb(H) = \sum_{v \in H}(1 + indeg(v))$ and $bnd(p) = 1 + indeg(p) - size(p)$;*
2. *$bnd(\bar{H}) \leq bnd(H)$ and $bnd(\bar{H}) \leq indeg(H) - size(H) + k$;*
3. *$comb(H) > indeg(H) \geq size(H) - 1$ and $indeg(H) \geq bnd(H)$;*
4. *$\mathcal{U}(\Gamma) \geq \mathcal{Q}(\Gamma) > \mathcal{S}(\Gamma)$.*

Proof. (1) Both expressions are derived by inspecting the definition of degree and boundary sizes.

(2) Clearly $bnd(\bar{H}) \leq bnd(H)$, because the missing edges of H contribute to its boundary size. Let now $pred(H, v)$ denote the number of predecessors of node v in G that are not nodes of H and $\overline{pred}(H, v)$ denote the number of predecessors of v in H (i.e., the in-degree of v in H). Since $\bar{H}$ is a complete subgraph and $V_{\bar{H}} = V_H$, we have that $bnd(\bar{H}) = \sum_{v \in V_H} pred(H, v) = \sum_{v \in V_H}[indeg(v) - \overline{pred}(H, v)] = indeg(\bar{H}) - \sum_{v \in V_H} \overline{pred}(H, v)]$. But $indeg(\bar{H}) = indeg(H)$ and, if V_{Sink}^H denotes the set of sink nodes in subgraph H with respect to their out-degree in H, then $\sum_{v \in V_H} \geq size(\bar{H}) + V_{Sink}^{\bar{H}}$, since each non-sink node in $\bar{H}$ contributes one in the sum: the edge that connects it to a successor node of $\bar{H}$. Since $size(\bar{H}) = size(H)$ and $V_{Sink}^{\bar{H}} = V_{Sink}^H \leq k$, we finally have that $bnd(\bar{H}) \leq indeg(H) - size(H) + k$.

(3) By definition and since $size(H) > 0$ we have that $comb(H) > indeg(H)$. Also, consider any spanning tree of the weakly connected subgraph H; all of its $size(H) - 1$ edges belong in H when they are appropriately given a direction. Thus, $indeg(H) \geq size(H) - 1$. (This implies that $comb(H) \geq size(H)$, although by definition for weakly connected graphs we take that $comb(H) > size(H)$.) Also note that $indeg(H) = bnd(H) + |E_H|$ for any subgraph H in G and $|E_H| \geq 0$.

(4) Note that the associated path π_s of any source node of G is a subgraph of the subgraph G_s of G that is reachable from s. Thus, $comb(G_s) \geq comb(\pi_s)$ for any source node s of G and

accordingly, $\mathcal{U}(\Gamma) \geq \mathcal{Q}(\Gamma)$. Similarly, since for any subgraph H of G we have from (3) that $comb(H) > bnd(H)$, we take that for any $s \in V_{source}$ it holds that $comb(\pi_s) > bnd(\pi_s)$. Thus, if s^* is the source node in G with the associated path π_s of maximum boundary size, we have that $\mathcal{Q}(\Gamma) \geq comb(\pi_{s^*}) > bnd(\pi_{s^*}) = \mathcal{S}(\Gamma)$. □

Note that, by the above lemma, for any DAG scheme, the update cost is no less than the query cost and the query cost is no less than the sibling cost. This fact will be used for the lower bound derivation: it suffices to focus only on the smallest of the costs of a DAG scheme, i.e., its sibling cost. Note that from Lemma 1 and in the *worst case*, the combined, degree and boundary sizes of the associated paths of source nodes of DAG scheme $\Gamma = (G, S, n, k)$, but also of the subgraphs reachable from the source nodes of G, are each lower bounded by cost measure $\mathcal{S}(\Gamma)$ of Γ.

2.3 Hierarchical Data Processing Problems

Our motivation for introducing DAG schemes is that they model an abstract class of computational problem where a DAG G holds a collection of n input *elements* (stored at source nodes) and a collection of output *values* of size that is bounded by a polynomial on n (stored at non-source nodes) that are computed using the DAG. *Query operations* on elements return a collection of values. *Update operations* modify the DAG G and the input elements, causing corresponding changes to the set of values. Computations are performed sequentially and hierarchically, according to the hierarchy induced by the underlying DAG G. The computational cost (time, space, or communication complexity) of query and update operations can be expressed as the combined, degree or boundary size of a subgraph (usually G_s or π_s, for a source node s of G), where every node v in the subgraph contributes to the cost an amount proportional to $indeg(v)$. Generally, any computational cost measure for a problem in this class is completely characterized by structural cost measures of subgraphs of DAG G. We refer to such problems as *hierarchical data processing (HDP) problems*. More formally:

Definition 4 (Hierarchical Data Processing problems). *The class of* hierarchical data processing *problems contain computational problems Π with the following characteristics.*

1. *The* input *of Π is a set of* elements $\mathcal{E} = \{el_1, ..., el_n\}$ *of size n.*
2. *The* output *of Π is a collection of* values $\mathcal{V} = \{val_1, ..., val_t\}$ *where t is bounded by a polynomial on n.*
3. *Associated with Π is a DAG scheme (G, S, n, k), such that elements in $\mathcal{E}$ and values in $\mathcal{V}$ are stored at source and respectively non-source nodes of G.*
4. *Given a subset $\mathcal{E}' \subseteq \mathcal{E}$ of elements, problem Π involves a* computation $\mathcal{C}_{\mathcal{E}'}$ *that performs some type of* data processing. *Computations in Π are triggered by some* operation*: either an* update *of an element in $\mathcal{E}$ or a* query *about a subset $\mathcal{E}' \subseteq \mathcal{E}$ of elements.*
5. *Any computation $\mathcal{C}$ in Π is fully characterized by an* associated *subgraph H of G. In particular:*
   - *The associated subgraph H of computation $\mathcal{C}_{\mathcal{E}'}$ depends on the elements in $\mathcal{E}'$, graph G and the set of special nodes of G. In particular, H is a weakly connected and complete subgraph of G that includes the source nodes hosting elements in $\mathcal{E}'$ and* at least one *of the nodes in S (special nodes of G).*
   - *Computation $\mathcal{C}$ is performed* sequentially *at* steps*, where each node v of H corresponds to a step of $\mathcal{C}$, and* hierarchically *according to the hierarchy induced by H, where a step corresponding to node v can be executed only after all steps corresponding to the predecessor nodes of v have been executed.*

- *The execution of any step of computation $\mathcal{C}$ corresponding to node v of H contributes to its computational cost an amount* proportional *to indeg*(v). *That is, a step at node v has time, space or communication complexity cost of* $\Theta(indeg(v))$.
- *Computations in* Π *involve one (or more) of the following types of data processing of the collection $\mathcal{V}' \subseteq \mathcal{V}$ of values stored at the nodes of the associated subgraph H, where data processing is performed sequentially and hierarchically according to DAG H: (i) the update of values in $\mathcal{V}'$, (ii) the output of values in $\mathcal{V}'$ or of values in a subset $\mathcal{V}'' \subset \mathcal{V}'$ of them or (iii) the evaluation of a function on values in $\mathcal{V}'$ or evaluations of functions on the values in $\mathcal{V}'$ or on the values in $\mathcal{V}'' \subset \mathcal{V}'$.*

We note that the DAG scheme that is associated with an HDP problem is *not* part of the input of the problem. That is, it is not given in advance. The DAG scheme is part of the algorithmic and data structuring technique that is used for the solution of the problem. Naturally, we ask which particular DAG scheme or DAG schemes of which type achieve optimal performance in terms of the complexity of the computations performed. By definition, any computation in an HDP problem is performed sequentially and hierarchically according to an associated subgraph of DAG G. Additionally, any such computation incurs computational costs that are fully described by the structural cost measures of the associated subgraph H. Thus, the cost measures of the DAG scheme associated to a HDP problem capture the intrinsic worst case computational complexity of this problem.

In the rest of the section we derive results that reveal the inherent computational limits that exist in any HDP problem and, furthermore, that characterize the optimal DAG scheme structure for these problems.

2.4 Sibling Cost and Search by Comparisons

We first show that the cost measures for a tree-based DAG scheme are related to the number of comparisons in a *search tree* derived from the scheme. By the above discussion and Lemma 1, focusing on the sibling cost suffices. For completeness, we first give some definitions.

Definition 5 (Directed Trees). *A* directed tree *is a DAG resulting from a rooted tree when its edges are assigned directions towards the root of the tree. Then, parent-child relation is defined among neighboring nodes, leaves correspond to source nodes and internal nodes correspond to non-source nodes.*

Definition 6 (Search Trees). *Let $(X, \preceq)$ be a totally ordered set of size n, drawing values from universe $\mathscr{U}$, where $\preceq$ is a binary relation, referred in this paper as* greater or equal.

- *Given any element $y \in \mathscr{U}$, we say that we* locate *y in X if we find the* predecessor *(element) $pr(y)$ of y in X, if it exists, defined to be $pr(y) = \{\max_{\preceq} x | x \in X \wedge x \preceq y\}$, i.e., the maximum (with respect to relation $\preceq$) element $x \in X$ such that $x \preceq y$. Locating an existing element of X in X corresponds to* finding *the element itself.*
- *A* leaf-based search tree *for $(X, \preceq)$ is a rooted tree such that: (i) the tree has exactly $|X| = n$ leaves, each storing an element in X, and the internal tree nodes are assigned with $n-1$ elements from X, (ii) given any element $y \in \mathscr{U}$, $pr(y) \in X$ can be located by searching in the tree, where at each tree node the search is lead by* comparisons, *i.e., evaluations of relation $\preceq$ on pairs of elements consisting of y and an element that is assigned at this tree node.*

Lemma 2. *Let $(X, \preceq)$ be a totally ordered set with n elements drawn from universe $\mathscr{U}$ and let $\Delta = (T, S, n, 1)$ be a DAG scheme, where T is a directed tree. We can build from T a search tree T' for X by storing the elements of X at the leaves of T and assigning tuples of elements in X to internal tree nodes of T, such that using T' any element $y \in \mathscr{U}$ can be located with $bnd(\pi_s)$*

comparisons, with s being the source node of T where the search ends. Accordingly, element $x \in X$ stored at source node s of T can be found in X using T' with $bnd(\pi_s)$ comparisons $\preceq$.

Proof. Assume that tree T is non-trivial. For each internal node of tree T, we fix a *left-right* ordering of its children. Using this ordering, we consider the topological order of T that corresponds to a postorder traversal of tree T and traverse the nodes of T according to this topological order. That is, any node in T is visited after all of its children nodes have been visited and according to the left-right ordering (with respect to the visit of its siblings in T). As we encounter leaves of T we store at them elements of X, one element at each leaf, selecting elements from X in increasing order. Next, we perform the following element assignment for each internal node in T, using again the topological order. Each non-source node v with predecessor nodes $u_1, \ldots, u_\ell$, listed according to the corresponding left-right node ordering, is assigned the ordered $(\ell - 1)$-tuple of elements $x_1^v, \ldots, x_{\ell-1}^v \in X$, where for $1 \leq i \leq \ell - 1$, x_i^v is the maximum element with respect to relation $\preceq$ that has been stored at the source nodes of the subtree in T having as root node u_i.

Tree T along with the elements stored at leaves and the assigned elements at internal tree nodes is a leaf-based search tree T' for set X. First, it is easy to see, using induction, that the number of assigned elements to internal nodes of T is $n - 1$ (only the maximum according to $\preceq$ relation element in X is not assigned to any node). Suppose we search for element y in the universe $\mathscr{U}$ where elements of X are drawn from. Observe that, while being at non-source node v, elements $x_1^v, \ldots, x_{\ell-1}^v$ can be used to decide to which node, among nodes $u_1, \ldots, u_\ell$, to advance our search as follows. For relation $\succ$ being the complement of relation $\preceq$, we advance the search at node u_1 if $y \preceq x_1^v$, at node u_ℓ if $y \succ x_{\ell-1}^v$, or otherwise at node u_i, where i is the unique integer value $1 \leq i \leq \ell - 2$ such that $y \succ x_i^v$ and $y \preceq x_{i+1}^v$. While visiting a source node s storing element x_s, we simply report as the predecessor of y in X element x_s. Besides, with respect to the correctness of the above searching procedure, we see that because of the way elements in X are stored at leaves and assigned to internal nodes of T, T' satisfies the following desired (search tree) property: at any node, elements stored in subtrees of children nodes that are to the right in the left-to-right ordering are larger (with respect to relation $\preceq$ on elements in X) than elements stored in subtrees of children nodes that are to the left in this left-to-right ordering. Moving to new node u_{i+1}, we correctly reduce the search space to the ordered subset of X $\{suc(x_i^v), \ldots, x_{i+1}^v\}$, where $suc(x)$ denotes the *successor* of x in X (defined to be the unique element x' in X such that $x = pr(x')$). Note that if a search to locate y ends at a source node s storing element x_s, x_s is the predecessor of y, and if $y \succ x_s$, then y and x_s are the same elements. That is, with one additional comparison at a leaf node we can test equality (whether the predecessor of the located element is the element itself).

Consider the search path p_s, when searching for an element $y \in \mathscr{U}$, that ends at source node s in T', that is the path connecting the root of T' to node s. Since for DAG scheme $\Delta = (T, S, n, 1)$, k (the number of special nodes) equals one, the root of T is the special node and thus p_s is the unique associated path π_s of s, i.e., $p_s = \pi_s$. Let v be a non-source node of this path. We have that by performing $indeg(v) - 1$ comparisons at node v we can advance our search to the correct node among the children of v. When we enter a source node, not comparison is needed. Thus, the total number of comparisons $\preceq$ performed when searching to locate y in X is equal to $\sum_{v \in \pi_s | indeg(v) > 0} (indeg(v) - 1) = 1 + \sum_{v \in \pi_s} (indeg(v) - 1) = 1 + indeg(\pi_s) - size(\pi_s) = bnd(\pi_s)$. Accordingly, since locating an existing element in X corresponds to finding the element itself, we can find element $x_s \in X$ stored at source node s of T using search tree T' by performing $bnd(\pi_s)$ comparisons.

If T contains trivial paths, the proof is as above with only one difference with respect to any nodes with in-degree 1. At every node w with in-degree 1, we assign the element assigned to its

child and when searching in the tree we perform no comparison $\preceq$ at w, but rather immediately advance our search at the child node. (That is, at a node v assigned with only one search key, we perform a comparison only when $indeg(v) = 2$.) The number of comparisons is again $bnd(\pi_s)$. □

Lemma 2 draws a direct analogy between the sibling cost of any tree-based DAG scheme and the number of comparisons performed in a search tree corresponding to the DAG scheme. We use this analogy as the basis for a reduction from searching by comparisons to any computational procedure of a HDP problem with cost that is expressed by the sibling cost of a tree-based DAG scheme.

Theorem 1. *Any DAG scheme $\Delta = (T, S, n, 1)$ such that T is a directed tree has $\Omega(\log n)$ update, query and sibling costs.*

Proof. It follows from Lemma 2 and the well-known $\Omega(\log n)$ lower bound on searching for an element in an ordered sequence in the comparison model (see for instance [18]). Namely, this fundamental result states that *any algorithm* for finding an element x in a list of n entries, by *comparing* x to list entries, *must perform* at least $\lfloor \log n \rfloor + 1$ comparisons for some input x. Comparisons are simply evaluations of binary relations on pair of elements. Obviously the above statement applies also to any search tree built for a totally ordered set X of size n using binary relation $\preceq$ on the elements of X. To see why, observe that a search tree can be viewed as an index structure for searching an ordered list of n elements, represented as the leaves of the search tree according to postorder tree traversal; additionally, any search path to an element of X in the tree completely describes the sequence of comparisons performed in order to locate this element. Consequently, applying the above to the search tree T' of Lemma 2, we get that for any DAG scheme $(T, S, n, 1)$, T being a directed tree, there exists a source node s such that the boundary size $bnd(\pi_s)$ of the associated path π_s of s is at least $\lfloor \log n \rfloor$, thus there exists a source node s such that $bnd(\pi_s)$ is $\Omega(\log n)$. Then by definition, we get that for any DAG scheme $\Delta = (T, S, n, 1)$ it holds that $\mathcal{S}(\Delta)$ is $\Omega(\log n)$. We finish the proof, by noting that from Lemma 1, for any DAG scheme $\Delta = (T, S, n, 1)$ we have that $\mathcal{S}(\Delta) < \mathcal{Q}(\Delta) \leq \mathcal{U}(\Delta)$ and that, since T is a directed tree its query cost $\mathcal{Q}(\Delta)$ is equal to its update cost $\mathcal{U}(\Delta)$. Thus, $\mathcal{Q}(\Delta) = \mathcal{U}(\Delta)$ and both are $\Omega(\log n)$. □

Remark 2.1. In any DAG scheme $(T, S, n, 1)$, where T a directed tree, there is a unique path connecting source node s to the unique special node in S (the root of T). Thus, the associated path π_s is also the minimum boundary size path from s to the root.

2.5 Optimality of Tree Structures

Next, we show that trees have optimal cost measures among all possible DAG schemes. We start by showing that for DAG schemes $(G, S, n, 1)$ having one special node, optimal costs are achieved when G is a directed tree.

Theorem 2. *Let $\Gamma = (G, S, n, 1)$ be a DAG scheme. There exists a DAG scheme $\Delta = (T, S, n, 1)$ such that T is a directed tree and $\mathcal{U}(\Delta) \leq \mathcal{U}(\Gamma)$, $\mathcal{Q}(\Delta) \leq \mathcal{Q}(\Gamma)$, and $\mathcal{S}(\Delta) \leq \mathcal{S}(\Gamma)$.*

Proof. DAG scheme Γ has only one special node, the unique sink node of G, so associated paths of the source nodes of G are paths from a source node to the sink node of G. we fix a topological order $t(G)$ of G. We define DAG T to be the union of all minimum combined cost directed paths π_s for all source nodes in G, where ties in computing paths π_s are broken using a consistent rule according to the topological order $t(G)$. It is easy to see that the union is a directed tree: if two paths π_{s_1} and π_{s_2} from source nodes s_1 and s_2 cross at node v and meet again at node u (u may be the sink node of G), this contradicts either the fact that each path has minimum combined cost or

the tie breaking rule. For instance, if subpath $p_{v,u}(s_2)$ of π_{s_2} does not coincide subpath $p_{v,u}(s_1)$ of π_{s_2}, either $comb(p_{v,u}(s_1)) \neq comb(p_{v,u}(s_2))$, in which case, one of the two paths π_{s_1} and π_{s_2} is not optimal (it is not of minimum combined cost), or $comb(p_{v,u}(s_1)) = comb(p_{v,u}(s_2))$, in which case, the tie breaking rule was violated.

By definition, it holds that $\mathcal{U}(\Delta) \leq \mathcal{U}(\Gamma)$, since for any source node s the reachable from s subgraph T_s in T (which is simply the corresponding leaf-to-root path in T) is a subgraph of the reachable from s subgraph G_s in G. With respect to query and sibling cost, it is easy to see that, since for any source node s the associated path π_s stays the same in graphs G and T but $|E_G| \geq |E_T|$, it holds that $indeg_T(\pi_s) \leq indeg_G(\pi_s)$. Thus, $\mathcal{Q}(\Delta) \leq \mathcal{Q}(\Gamma)$, and $\mathcal{S}(\Delta) \leq \mathcal{S}(\Gamma)$, as stated. □

Remark 2.2. The directed tree T in the proof above, by construction, consists of minimum combined size paths from source nodes in G to the unique special node and thus T is an optimal tree with respect to the structural cost measure of combined size. However T is not necessarily optimal with respect to the structural cost measure of boundary size, because, for general graphs, minimum combined cost does not imply minimum boundary cost. Thus, although tree T is a minimum combined size tree in G and it is a tree such that $\mathcal{S}(\Delta) \leq \mathcal{S}(\Gamma)$, T may not be a minimum boundary size tree in G. In other words, the minimum combined size and minimum boundary size trees in G do not necessarily coincide. This does not affect any of our results.

Finally, we examine how allowing more than one special nodes in a DAG scheme affects its cost measures. We have that a DAG scheme Γ with k special nodes achieve minimum cost measures when the roots of at most k *distinct trees* are the only special nodes in Γ.

Lemma 3. *Let $\Gamma = (G, S_G, n, k)$ be a DAG scheme. There exists a DAG scheme $\Phi = (F, S_F, n, \ell)$ such that F is a forest of $\ell \leq k$ directed trees, $S_F \subseteq S_G$ and additionally $\mathcal{U}(\Phi) \leq \mathcal{U}(\Gamma)$, $\mathcal{Q}(\Phi) \leq \mathcal{Q}(\Gamma)$, and $\mathcal{S}(\Phi) \leq \mathcal{S}(\Gamma)$.*

Proof. The proof is similar to the proof of Theorem 2. Consider the union F of the minimum combined size paths $\pi_1, \dots, \pi_n$ in G from source nodes $s_1, \dots, s_n$ to a special node, where ties break according to a well-defined and consistent rule (e.g., using a topological order of G). The resulting subgraph F of G is a forest: two paths never cross, but they only meet to same special node, and additionally, no path connecting two distinct special nodes exists in F. As in the proof of Theorem 2, since (i) for any source node s the reachable from s subgraph F_s in F is a subgraph of the reachable from s subgraph G_s in G, (ii) $|E_F| \leq |E_G|$, and (iii) the minimum combined size paths (from source nodes to special nodes) are the same in G and F, we have that $\mathcal{U}(\Phi) \leq \mathcal{U}(\Gamma)$, $\mathcal{Q}(\Phi) \leq \mathcal{Q}(\Gamma)$, and $\mathcal{S}(\Phi) \leq \mathcal{S}(\Gamma)$, as desired. □

2.6 Lower Bounds for Hierarchical Data Processing

The following theorem summarizes the results of this section with respect to the cost measures of any DAG scheme.

Theorem 3. *Any DAG scheme $\Gamma = (G, S, n, k)$ has $\Omega(\log \frac{n}{k})$ update, query and sibling costs.*

Proof. It follows directly from Theorems 1 and 2 and Lemma 3. First, by Lemma 3 and Theorem 2 we get that the best (lowest) cost measures of DAG scheme Γ are achieved when G is a forest F of at most k trees. In this case, since these trees are minimum combined cost trees, the update, query and sibling cost of G are defined by the tree $T^* \in F$ having the maximum complexity in terms of the cost measure of sibling cost. Moreover, this cost measure depends of the number of source nodes $V_{source}(T^*)$ of T^*. We know that $|V_{source}(T^*)|$ is $\Omega(\frac{n}{k})$. If $\Phi = (F, S_F, n, \ell)$, $\ell \leq k$, is the DAG scheme of Lemma 3 and $\Delta = (T^*, S, |V_{source}(T^*)|, 1)$ is the DAG scheme that corresponds to

tree T^*, we have that $\mathcal{S}(\Gamma) \geq \mathcal{S}(\Phi) = \mathcal{S}(\Delta)$, which from Theorem 1 is $\Omega(\log \frac{n}{k})$. By Lemma 1 we get also that $\mathcal{U}(\Gamma)$ is $\Omega(\log \frac{n}{k})$ and that $Q(\Gamma)$ is $\Omega(\log \frac{n}{k})$. □

The above results form the basis for a reduction from searching by comparisons to any computation of an HDP problem. Essentially, we draw an analogy between searching by comparisons and the sibling cost of any DAG scheme. The above theorem can be directly combined with the definition of the class of HDP problems to give the following general result about the complexity of any problem in the class.

Theorem 4. *All computations of any hierarchical data processing problem* Π *associated with DAG scheme* $\Gamma = (G, S, n, k)$ *have* $\Omega(\log \frac{n}{k})$ worst case *time, space or communication complexity.*

Proof. By definition, an HDP problem Π involves computations that correspond to associated subgraphs of G. Recall that a computation $\mathcal{C}$ is associated to a subgraph H of G and involves some type of data processing and is performed sequentially and hierarchically according to graph H, such that: (i) computations are performed in steps, one step per node of H and steps are executed one after the other, (ii) computational step corresponding to node u of H can be executed only when the computational steps corresponding to all predecessor nodes of u in H has been executed and (iii) the computational step corresponding to node u of H has time, space or communication complexity that is proportional to $indeg(u)$, that is, the complexity is $\Theta(indeg(u))$. Also recall that computation (and graph H) depends on a subset $\mathcal{E}'$ of the input elements $\mathcal{E}$. Namely, graph H is the complete subgraph of G that contains the source nodes storing elements in $\mathcal{E}'$ and at least one special node.

Let $\mathcal{C}_{\mathcal{E}'}$ be any computation of problem Π that depends on subset $\mathcal{E}'$, with H being its associated subgraph of G. Given the above setting with respect to the computational model for HDP problems, it follows that computation $\mathcal{C}$ has time, space or communication complexity C_H proportional to $\sum_{v \in V_H} \Theta(indeg(v))$, thus $C_H = \Theta(\sum_{v \in V_H} indeg(v)) = \Theta(indeg(H)))$. Accordingly $C_H = \Omega(bnd(H))$, since $indeg(H) \geq bnd(H)$ for any H (Lemma 1). We have concluded that any computation $\mathcal{C}_{\mathcal{E}'}$ with associated graph H has time, space or communication complexity $C_H = \Omega(bnd(H))$. To complete the proof, simply consider $\mathcal{E}' = x_{s^*}$, where x_{s^*} is the input element stored at the source node s^* that defines the sibling cost $\mathcal{S}(\Gamma)$ of DAG scheme Γ, i.e., the source node for which the minimum combined cost path to a special node u_{sp} is the maximum over all other source nodes, and let H contain only this special node $u_{sp} \in S$. The associated path π_{s^*} of s^* is then a subgraph of H. Thus, $bnd(H) \geq bnd(\pi_{s^*}) = \mathcal{S}(\Gamma)$ and from Theorem 3 $\mathcal{S}(\Gamma)$ is $\Omega(\log \frac{n}{k})$, so $C_H = \Omega(\log \frac{n}{k})$. It follows that for any HDP problem Π associated with DAG scheme Γ, there exists a computation with $\Omega(\log \frac{n}{k})$ time, space or communication complexity. □

Remark 2.3. Regarding the definition of the sibling cost of a DAG scheme Γ with respect to the minimum combined size associated paths, we note that this choice is twofold. First, we have seen that $\mathcal{S}(\Gamma)$ serves our purposes, since it *strictly* bounds the query and update costs of Γ and provides us with worst case lower bounds for the computational costs of the update and query operations in any HDP problem. Second, in the case where the cost of a computation in an HDP problem is characterized by the boundary size of some source-to-special node path, this path typically corresponds to a minimum combined cost path. We will see such a case in Section 4.

We have established logarithmic lower bounds for the complexity of computations related to HDP problems, where we have further characterized the tree-based DAG schemes as the optimal structures for these problems. The connection between problems in this class and DAG schemes is illustrated in Sections 4 and 5, where we model two information security problems as HDP problems and translate the above results to their domain.

3 A New DAG Scheme Based on Skip-Lists

In view of the logarithmic lower bounds and the optimality of tree structures for HDP problems, in this section, we describe a new tree-based DAG scheme $\Delta = (T, S, n, 1)$, which we call *multi-way skip-list DAG scheme*, that is based on and defined with respect to skip-lists [32, 33], which are randomized data structures, equivalent to balanced trees, upon which dictionaries can be built. We study the performance of Δ in terms of the node size $size(\cdot)$, the degree size $indeg(\cdot)$ and also the boundary size $bnd(\cdot)$, where we show that all these cost measures have low expected values.

3.1 Skip-Lists and Bridges

We briefly describe the notation that we will use. A *skip-list* with *probability parameter* p is a set of lists $L_1, ..., L_h$, where L_1 stores all the element of a totally ordered set $(X, \preceq)$ of size n, sorted according to $\preceq$, where elements are drawn from universe $\mathscr{U}$, and, for each i, each of the elements of list L_i is independently chosen to be contained in L_{i+1} with probability p. Lists are viewed as *levels* and we consider all elements of the same value that are stored in different levels to form a *tower*. That is, a tower consists of nodes of lists that store the same copied element. The *level* of a tower is the highest level of the tower (or the level of its top element). Each node of a tower has a forward pointer to the successor element in the corresponding list and pointer to the element one level below it. A *header* tower that stores sentinel element $-\infty$, representing the minimum with respect to $\preceq$ value in $\mathscr{U}$, is included in the skip-list as the left-most tower of level one more than the maximum level in the skip-list. A node of the skip-list is a *plateau* node if it is the top node of its tower. Furthermore, we introduce the notion of a *bridge* and also define relative concepts.

Definition 7 (Skip-List Notation). *In a skip-list:*

- *a* bridge b *is a sequence of towers of the same level, where no higher tower is interfering them and the plateau nodes of the towers are all reachable in a sequence using forward pointers;*
- *the* bridge size $|b|$ *of bridge* b *is the number of towers in the bridge (i.e., the size of the sequence); the* bridge size *of a tower is the size of the bridge that the tower belongs to;*
- *a* child bridge *of* b *is a bridge that is contained under* b *and to which a tower of* b *is connected through forward pointers;*
- *the* plateau towers *of a tower* t *are the towers whose plateau nodes can be reached by* t *using one forward pointer.*

3.2 Directed Tree T

We now describe the new DAG scheme $(T, S, n, 1)$, the skip-list DAG, where T is a directed tree with n leaves (source nodes) and one special node, the root r of T, thus $S = r$. For simplicity, we use the term skip-list DAG to refer to both the DAG scheme $(T, r, n, 1)$ and the directed tree T. The skip-list DAG is defined with respect to a skip-list with probability parameter p. In what follows, by *list node* we refer to a node of the skip-list and by *DAG node* to a node of skip-list DAG T. Any edge (v, u) in T is assumed to be directed from node v to node u. To facilitate our description we define an operation on DAG nodes, which, given existing DAG nodes in T, creates new nodes and edges in T: if $v, v_1, \ldots, v_l$ are nodes in T, then operation $New(v, v_1, \ldots, v_l)$ creates in T new nodes $u_1, \ldots, u_l$ and new edges $(v_1, u_2), \ldots, (v_{l-1}, u_l), (v_l, v)$ and $(u_1, u_2), (u_2, u_3), \ldots, (u_{l-1}, u_l), (u_l, v)$, where DAG node u_1 is a *source* node in T. So, operation $New(\cdot)$, in fact, creates and appropriately connects it in T a directed path from source node u_1 to the existing in T node v, where existing nodes in T $v_1, \ldots, v_l$ are sibling nodes in the path. Finally, to better understand the connection between the skip-list DAG and the skip-list and also for our analysis, we consider that each DAG node is

attached to some list node in the skip-list.

The notion of a bridge is essential in skip-list DAG. For each bridge b in the skip-list, a corresponding node $v(b)$ is created in T. We call $v(b)$ the DAG node of b. In essence, $v(b)$ is connected in T with the DAG nodes of all the child bridges of b. This allows us to define DAG T with respect to a skip-list in a recursive way: first all bridges in the skip-list are identified and the DAG node for the outer bridge, corresponding to the header tower of the skip list, is created, then, given that the DAG node $v(b)$ of a bridge b is created, using operation $New(\cdot)$, it is connected with paths in T to the newly created DAG nodes of the child bridges of b (see Figure 1). We next explain how this connection is performed.

First, suppose that the size of b is one, i.e., b is simply a tower t (see Figure 1(a)). Let $t_1, ..., t_l$ be the plateau towers of t in increasing order with respect to their level. Note that the level of t_l is less than the level of t. If plateau tower t_i belongs in bridge b_i, then let $v(b_1), ..., v(b_l)$ be the corresponding DAG nodes of the bridges. Then we perform operation $New(v(b), v(b_1), \ldots, v(b_l))$ where $v(b)$ is the DAG node of b. We attach the new nodes of T created from this operation as follows: source node u_1 is attached to the lowest list node of the tower, node u_i, $2 \leq i \leq l$, is attached to the list node of t at the level of bridge b_{i-1} and the DAG node $v(b)$ of b is attached to the list node of t at the level of bridge b_l. Note that DAG node u_1 is the basis in the recursion. If the size of b is more than one, say k, then, let $t_1, \ldots, t_k$ be the towers of b (see Figure 1(b)). First, for each such tower t_i, we create a new DAG node $v(t_i)$, $1 \leq i \leq k$. For tower t_k, we consider its, say l, plateau towers $t_{k1}, \ldots, t_{kl}$ and perform operation $New(v(t_k), v(b_{k1}), \ldots, v(b_{kl}))$, where $b_{k1}, \ldots, b_{kl}$ are the child bridges of b that plateau towers $t_{i1}, ..., t_{il}$ belong in and where the new source node u_1 created by this operation corresponds to the element that is stored in tower t_k. Moreover, for each tower t_i, $i < k$, of, say $l+1$, plateau towers, we consider its l lowest plateau towers $t_{i1}, \ldots, t_{il}$, that is, for $i < k$, tower t_{i+1} is *omitted* from this sequence (only in this case, t_{i+1} is not considered to be a plateau tower of t_i). Let $b_{i1}, \ldots, b_{il}$ be the child bridges b that plateau towers $t_{i1}, ..., t_{il}$ belong in. Then for tower t_i, $i < k$, we perform operation $New(v(t_i), v(b_{i1}), \ldots, v(b_{il}))$, where the new source node u_1 created by this operation corresponds to the element that is stored in tower t_i. Finally, we add the following k new edges in T: edge $(v(t_i), v(b))$ for $1 \leq i \leq k$. Newly created nodes are attached to list nodes in a similar way as before. We attach the DAG nodes created when considering tower t_i, $1 \leq i \leq k$, to the list nodes of the tower at the level of the bridge they are connected to, source nodes to the corresponding lowest-level list nodes and DAG node $v(b)$ of bridge b to the top left-most list node of b.

By this recursive definition, it is easy to see that, indeed, T is a directed tree; the root and unique sink node of T corresponds to the first created DAG node r for the highest bridge of the skip-list, the one consisting of the left-most header tower. The leaves and source nodes of T correspond to the lowest-level list where n elements of set X are stored in the skip-list. Thus, skip-list DAG is a $(T, r, n, 1)$ DAG scheme (see also Figure 2). Note that since our new DAG scheme is defined with respect to a skip-list, it is a *randomized* DAG scheme.

3.3 Cost Measures of Skip-List DAG

We now analyze the cost measures of the DAG scheme $\Delta = (T, r, n, 1)$ according to the definitions in Section 2.2. From Theorem 2 and Lemma 3, it suffices to study only the case where there is only one special node in T, the root of T. Also, we know that for the update , query and sibling cost measures of Δ it holds $\mathcal{U}(\Delta) = Q(\Delta) > \mathcal{S}(\Delta)$ and that all are lower bounded $\log n$. Accordingly, it suffices to concentrate our analysis to the associated path π_s of a source node s in T. This path is not only the minimum combined size path from s to the root but also the minimum boundary size path of s. Since Δ is a randomized DAG scheme, we study its *expected* cost measures, and

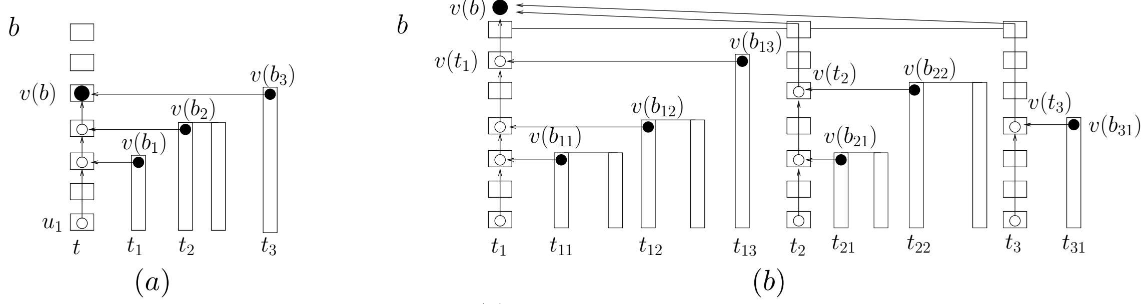

Figure 1: Skip-list DAG T: DAG node $v(b)$ of bridge b is recursively connected to the DAG nodes of the child bridges, for the case where (a) the size of bridge b is one or (b) the size of b is more than one. Circle nodes are DAG nodes of T and square nodes are list nodes of skip-list. A DAG node is attached to the list node in which it is contained.

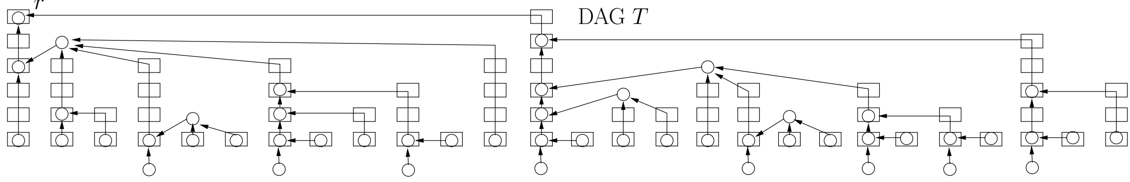

Figure 2: The skip-list DAG T of a skip-list. T is a directed tree.

since these measures can be expressed using the structural metrics node size $size(\cdot)$, degree size $indeg(\cdot)$ and boundary size $bnd(\cdot)$ (see Lemma 1), we are interested in studying the *expected node size* $E(size(\pi_s))$, the *expected degree size* $E(indeg(\pi_s))$ of path π_s and the *expected boundary size* $E(bnd(\pi_s))$ of path π_s, i.e., the expected number of nodes in the path, the expected total number of the predecessor nodes of nodes in the path and the expected total number of sibling nodes in the path π_s.

In the next Theorem we prove that our new DAG scheme achieves cost measures that are close to the theoretical optimal value of $\lfloor \log n \rfloor + 1$. Note that this value is theoretically optimal also for randomized DAG schemes, since Lemma 2 holds true for any DAG scheme, even randomized, because only its structural properties matter for the definition of the corresponding search tree.

Theorem 5. *With respect to a skip-list with probability parameter p, the skip-list DAG scheme $\Delta = (T, r, n, 1)$, for any fixed source node s of T and the corresponding source to root path π_s, has the following expected performance:*

1. $E[size(\pi_s)] \leq 2(1-p)\log_{\frac{1}{p}} n + O(1)$;
2. $E[indeg(\pi_s)] \leq (1-p)\frac{(1+p)^2}{p}\log_{\frac{1}{p}} n + O(1)$;
3. $E[bnd(\pi_s)] \leq \frac{(1-p)(1+p^2)}{p}\log_{\frac{1}{p}} n + O(1)$ *and*
4. $E[size(T)] \leq (1 + pq^2 + pq + \frac{p}{q(2-pq^2)})n$, *where* $q = 1-p$.

Proof. In the proof, we use a worst case scenario, assuming that the skip-list has *infinite* size to the left or to the right, that is, it is unbounded to one direction. This allows us to actually compute upper bounds of expected values of random variables related to our analysis.

We first compute the expected size of a bridge b in the skip-list. Consider the left-most tower t of b and its top skip-list node u. Assuming an infinite to the right skip-list, u has a forward pointer to the skip-list node v, the next node in list of u. With probability $1-p$ node v is the top node of its tower t_v, thus t_v belongs in b and with probability p tower t_v is of higher level, thus t_v is not a

tower of b. If $|b|$ denotes the size of bridge b, then $|b| = 1+Y$, where Y is a geometrically distributed random variable with parameter p, which counts the number of failures before the success occurs, where $\Pr[\text{success}] = p$. Thus, $E[|b|] = 1 + \frac{1-p}{p} = \frac{1}{p}$ and on average $1/p$ towers are consecutive having the same height.

(1), (2) & (3) We use a backward analysis as in [32, 33]. Given a skip-list, we consider the corresponding skip-list data structure storing a totally ordered set X of size n, where elements in X can be found by traversing the list nodes of the skip-list. We consider traveling backwards on the search path π in the skip-list data structure for element x stored at source node s. It is easy to see, that, given the way with which DAG nodes of T are assigned to list nodes of the skip-list, path π_s in T, the leaf-to-root path in T from source node s of T, is *contained* in path π. Consequently, as we travel backwards (starting from s) along the search path π, we compute the structural metrics $size(\pi_s)$, $indeg(\pi_s)$ and $bnd(\pi_s)$ of path π_s in T. Assuming a worst case analysis, where π reaches level $\log_{\frac{1}{p}} n$, we split π in two parts: subpath π_1 and subpath π_2. Path π_1 takes us to level $\log_{\frac{1}{p}} n$ and path π_2 completes the backward search up to the first skip-list node of π. Accordingly, in our analysis we partition π_s into subpaths π_{s_1} and π_{s_2} corresponding to subpaths π_1 and respectively π_2 of π. Obviously, since $\pi_s = \pi_{s_1} \cup \pi_{s_2}$, $size(\pi_s) = size(\pi_{s_1}) + size(\pi_{s_2})$, $indeg(\pi_s) = indeg(\pi_{s_1}) + indeg(\pi_{s_2})$ and $bnd(\pi_s) = bnd(\pi_{s_1}) + bnd(\pi_{s_2})$.

The node, degree and boundary sizes of π_{s_2} are all on average constant, i.e., $E[size(\pi_{s_2})] = O(1)$, $E[indeg(\pi_{s_2})] = O(1)$ and $E[bnd(\pi_{s_2})] = O(1)$, because the skip-list size above level $\log_{\frac{1}{p}} n$ is on average (and with high probability) *constant*. Indeed, subpath π_2 can be further partitioned into the set L of nodes that we reach in our backward traversal moving leftward and the set U of nodes that we reach moving upward. We have that $|L| \leq Y$, where Y is a random variable counting the number of towers in the skip list with level $\log_{\frac{1}{p}} n$ or higher and that $Y \sim Bin(n, \frac{1}{np})$, since with probability $p^{(\log_{\frac{1}{p}} n)-1} = \frac{1}{np}$ a tower has level $\log_{\frac{1}{p}} n$ or higher. So, $E[L] \leq \frac{1}{p} = O(1)$. Also, we have that $|U| \leq Z$, where Z is the size of the highest skip-list tower above level $\log_{\frac{1}{p}} n$ and that $Z \sim G(1-p)$, i.e., geometrically distributed with parameter $1-p$ (where we use the memoryless property of geometric distribution). So $E[U] \leq \frac{p}{1-p} = O(1)$. We conclude that the size of π_2 is on average at most $E[L] + E[U] = O(1)$. Since every node in π_{s_2} has constant in-degree (2 or $E[|b|] = O(1)$ for a bridge b), we conclude that the node, degree and boundary sizes of π_{s_2} are all $O(1)$.

(1) For the node size of π_{s_1}, we assume an infinite skip-list to the left, that is, no header tower is present (and thus a worst case analysis is in place). Let $C_k(t)$ be a random variable counting the node size $size(\pi)$ counted so far when k upwards moves remain to be taken in part π_1 of path π and we are performing the t-th step. Then if we move up $C_k^t = X_U + C_{k-1}^{t+1}$, otherwise $C_k^t = X_L + C_k^{t+1}$, where X_U, X_L are 0-1 random variables that count if a DAG-node is encountered when moving up or left respectively. Then we have that $\Pr[X_U = 1] = p(1-p)$, because with probability $1-p$ the node that the forward pointer points to is a plateau node and with probability p the node that we move to is not a plateau node (i.e., we count a DAG-node created by applying operation $New(\cdot)$). Also we have that $\Pr[X_L = 1] = p + p(1-p)$, because with probability p we left a bridge and with probability $(1-p)$ the bridge has size more that one (i.e., we count a node created by operation $New(\cdot)$ and possibly a bridge DAG node). Observe that we count DAG nodes of bridges of size 1 when moving up. Since we have an infinite skip list, $C_k^t \sim C_k^{t+i} \sim C_k$ for any $i > 0$, where C_k is a

random variable distributed as C_k^t. Thus, using conditional expectation

$$\begin{aligned}
E[C_k] &= E[E[C_k|\text{move}]] \\
&= E[\Pr[\text{up}]C_k|\text{up} + \Pr[\text{left}]C_k|\text{left}] \\
&= E[p(X_U + C_{k-1}) + (1-p)(X_L + C_k)] \\
&= p(E[X_U] + E[C_{k-1}]) + (1-p)(E[X_L] + E[C_k]) \\
&= p(p(1-p) + E[C_{k-1}]) + (1-p)(p + p(1-p) + E[C_k]),
\end{aligned}$$

which gives $E[C_k] = E[C_{k-1}] + 2(1-p)$, and finally we get that $E[C_k] = 2(1-p)k$. So, $E[size(\pi_s)] \leq E[C_k] = 2(1-p)\log_{\frac{1}{p}} n + O(1)$.

(2) Similarly, regarding the degree size of π_{s_1} and, again, assuming an infinite skip-list to the left, let $C_k(t)$ be a random variable counting the degree size $indeg(\pi)$ counted so far when k upwards moves remain to be taken and we are performing the t-th step. Then if we move up $C_k^t = X_U + C_{k-1}^{t+1}$, otherwise $C_k^t = X_L + C_k^{t+1}$. Here X_U and X_L are random variables that count the number of predecessor DAG nodes that we have to add when moving up or left respectively. We have that $E[X_U] = 2p(1-p)$ because with probability $p(1-p)$ we count two predecessors after moving up (a node created by operation $New(\cdot)$ has in-degree two). Also $E[X_L] = p(2 + \frac{1-p^2}{p})$, because with probability p we have just left a bridge moving left and, thus, we count $2 + Y$ predecessors. I.e., a node created by operation $New(\cdot)$ has two predecessors and Y is a random variable counting the in-degree of a possible bridge DAG node that must be encountered. Observe that $Y = 0$ unless the bridge has size at least 2 and we compute $E[Y] = (1-p)(2 + \frac{1-p}{p}) = \frac{1-p^2}{p}$. Using conditional expectation as above, we finally get $E[indeg(\pi_s)] \leq (1-p)(2p+2+\frac{1-p^2}{p})\log_{\frac{1}{p}} n + O(1) = (1-p)\frac{(1+p)^2}{p}\log_{\frac{1}{p}} n + O(1)$.

(3) For the boundary size of π_{s_1} and assuming an infinite skip-list to the left, let $C_k(t)$ be a random variable counting the degree size $bnd(\pi)$ counted so far when k upwards moves remain to be taken and we are performing the t-th step. As before, if we move up $C_k^t = X_U + C_{k-1}^{t+1}$, otherwise $C_k^t = X_L + C_k^{t+1}$, where X_U and X_L are random variables that count the number of sibling DAG nodes in π_{s_1} that we have to add when moving up or left respectively. We have that $E[X_U] = p(1-p)$ because with probability $p(1-p)$ we count one sibling DAG node after moving up (a node created by operation $New(\cdot)$ has in-degree two: one node is in p_s and one is a sibling node). Also $E[X_L] = p(1 + \frac{1-p}{p}) = 1$, because with probability p we have just left a bridge moving left and, thus, we count $1 + Y$ sibling nodes, one sibling of the node created by operation $New(\cdot)$ and Y siblings (a random variable) corresponding to the possible bridge DAG node that we just left. We have that $E[Y] = (1-p)(1 + \frac{1-p}{p}) = \frac{1-p}{p}$. Using conditional expectation as before, we finally get $E[bnd(\pi_s)] \leq (1-p)(\frac{1+p^2}{p})\log_{\frac{1}{p}} n + O(1)$.

(4) By construction, $V_T = V_{source}(T) \cup B \cup N$, where $V_{source}(T)$ is the set of the n source nodes in T, B the set of bridge DAG nodes and N the set of the non-source, non-bridge, DAG nodes (created by operation $New(\cdot)$ or being predecessors of a DAG node of a bridge). Let B_1 denote the set of DAG nodes in B that are assigned to list-nodes of level 1 and let $B_{>1} = B - B_1$. Similarly, let N_1 denote the set of DAG nodes in N that are assigned to list-nodes of level 1 and let $N_{>1} = N - N_1$. We can compute that $E[|B_1|] \leq p(1-p)^2 n$ using the union bound, since with probability $p(1-p)^2$ the lowest plateau tower of a tower is the left most tower of a bridge of level 1 and of size at least two. Similarly, we can compute that $E[|N_1|] \leq p(1-p)n$, again by applying the union bound and noticing that with probability $p(1-p)$ a non-source DAG node is created by operation $New(\cdot)$. Now, let M denote the set of list nodes in the skip-list that have not been assigned a DAG node,

which we call *empty* nodes, then for the set $K_{>1}$ of list nodes in the skip-list of level 2 or more, we have that $K_{>1} = M \cup B_{>1} \cup N_{>1}$, where in this formula DAG nodes are treated like the list nodes they are assigned to. We next pair up DAG nodes in $B_{>1} \cup N_{>1}$ to *distinct* empty list nodes in M. Nodes in $B_{>1}$ are paired up with probability 1. Consider any node u in $B_{>1}$ corresponding to bridge b, with $|b| \geq 2$. Node u can be paired with any of the empty list nodes of the top level of b. Any DAG node u in $B_{>1}$ corresponding to bridge b, with $|b| = 1$, can be paired with the empty list node one level up in its tower. Any DAG node u in $N_{>1}$ can be paired up with the empty list node $l(u)$ on its left, if it not paired with a node in $B_{>1}$. The probability that u can not be paired up with $l(u)$ is $\lambda = p(1-p)^2$ ($l(u)$ is the top-right node of a bridge of size 2), an *independent* event from any other node in $N_{>1}$ not being paired up with its empty node of the left. Thus, we have that, if $M = M_1 \cup M_2 \cup M_3$, where M_1, M_2, M_3 are disjoint sets containing the empty nodes paired up with nodes in $B_{>1}$, in $N_{>1}$ and respectively with no DAG nodes, then $E[|M_1|] = E[|B_{>1}|]$ and $E[|M_2| = (1-\lambda)E[|N_{>1}|]]$. Using this, we get that $E[|K_{>1}|] = 2E[|B_{>1}|] + (2-\lambda)E[|N_{>1}|] + E[|M_3|]$ and the bound $E[|B_{>1}|] + E[|N_{>1}|] \leq \frac{E[K_{>1}]}{2-\lambda}$. Putting all together, we finally get the upper bound for $E[size(T)] = E[|V_T|]$. Note that $E[size(T)] < \frac{n}{1-p}$, the expected size of a skip-list. □

3.4 Comparison with other DAG Schemes

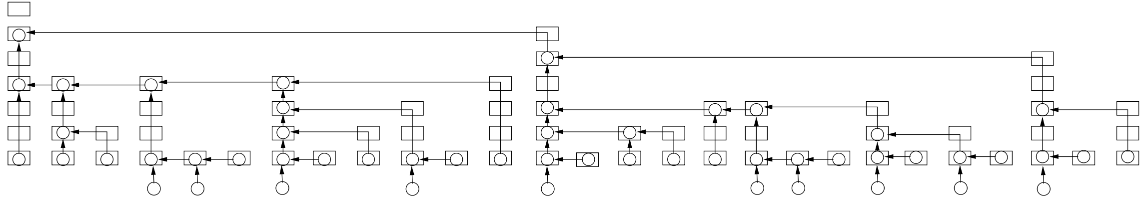

Figure 3: The DAG scheme Δ that corresponds to an improved version (with respect to the number of comparisons) of standard skip list [33].

We now compare our multi-way skip-list with red-black trees and the standard skip-list in terms of the cost measures of the underlying search DAGs. Figure 3 shows that search DAG that corresponds to an improved version of the standard skip-list appeared in [33]. For these trees and for any fixed element stored at a source node s and the corresponding to path π_s in DAG T, we compare the corresponding expected values for the node size $size(\pi_s)$, degree size $indeg(\pi_s)$ and boundary size $bnd(\pi_s)$ of path π_s and the node size $size(T)$ of T. Note that by Lemma 2, $comb(\pi_s)$ corresponds to the number of comparisons performed in the search structure and that $size(T)$ corresponds to the total number of decision nodes in search tree T.

	$E[size(\pi_s)]$	$E[indeg(\pi_s)]$	$E[bnd(\pi_s)]$	$E[size(T)]$
red-black tree	$\log n$	$2\log n$	$\log n$	$2n$
standard skip-list	$1.5\log n$	$3\log n$	$1.5\log n$	$2n$
multi-way skip-list	$\log n$	$2.25\log n$	$1.25\log n$	$1.9n$

Table 1: Comparison of three DAG schemes for $p = 0.5$; $size(\pi_s)$: node size of search path π_s, $indeg(\pi_s)$: number of predecessors of nodes on search path π_s, $comb(\pi_s)$: number of predecessors of nodes on search path π_s that do not belong on the path, or number of comparisons performed by search structure, $size(T)$: total number of nodes of DAG or total number of decision nodes in search tree; all numbers correspond to expected values of the corresponding cost parameters.

Table 1 summarizes the comparison results. For red-black trees, expectation corresponds to the

average search path, which has size $c \log n$ for a constant c which is very close to 1 (see [35]), whereas for skip-lists it corresponds to the random skip-list structure. To simplify the comparison, we choose parameter $p = \frac{1}{2}$ for the two skip-lists. The multi-way skip-list DAG has better performance (achieves lower constants) when compared with the standard skip-list. On the other hand, we observe an interesting trade-off on the performance that reb-black tree and multi-way skip-lists achieve: in multi-way skip-lists the combined and boundary sizes are larger but the node size of the tree is less.

4 Data Authentication Through Hashing

In this section, we apply our results of Sections 2 and 3 in *hash-based data authentication*, that is data authentication based on cryptographic hashing. We focus on *authenticated dictionaries*, where membership queries on sets are authenticated. We show that this problem is a hierarchical data processing problem. Applying our results to authenticated dictionaries, we get a logarithmic lower bound on the authentication cost for any authentication scheme that uses cryptographic hashes and a new authenticated dictionary based on skip lists with authentication cost closer to optimal.

4.1 Authenticated Data Structures

Authenticated data structures provide a model of computation where an untrusted *directory* can answer queries issued by a *user* on a data structure on behalf of a trusted *source* but provide a proof of the validity of the answer to the user. The data source ideally signs only a single *digest* of the data. For *data authentication through hashing* a hash function is systematically used to produce this digest. On a query, along with the answer, the signed digest and some information that relates the answer to this digest are also given to the user and these are used for the answer verification.

In particular, the model involves a structured collection X of objects, the *source*, the *directory*, and the *user*. A repertoire of *query operations* and optional *update operations* are assumed to be defined over X. The role of each party is as follows. The *source* holds the original version of X. Whenever an update is performed on X, the source produces *update authentication information*, which consists of a signed time-stamped statement about the current version of X. The *directory* maintains a copy of X. It interacts with the source by receiving from the source the updates performed on X together with the associated update authentication information. The directory also interacts with the user by answering queries on X posed by the user. In addition to the answer to a query, the directory returns *answer authentication information*, which consists of (*i*) the latest update authentication information issued by the source; and (*ii*) a *proof* of the answer. The *user* poses queries on X, but instead of contacting the source directly, it contacts the directory. However, the user trusts the source and not the directory about X. Hence, it verifies the answer from the directory using the associated answer authentication information. The data structures used by the source and the directory to store collection X, together with the algorithms for queries, updates, and verifications executed by the various parties, form what is called an *authenticated data structure*.

Authenticated Dictionary We focus on the dictionary problem where we want to answer membership queries about a set of objects in the previous authentication model. Let X be a data set owned by the source that evolves through update operations insert and delete. Membership queries exists are issued on X. A (multivariate extension of a) cryptographic hash function h is used to produce a *digest* of set X, which is signed by the source. In our study, we actually consider a more general model where more than one digests are produced and signed by the source. These digests

are computed through a *hashing scheme* over a directed acyclic graph (DAG) that has k *signature* nodes $t_1, \ldots, t_k$ and stores the elements of X at the source nodes (see [16]). Each node u of G stores a *label* or *hash value* $L(u)$ such that if u is a source of G, then $L(u) = h(e_1, \ldots, e_p)$, where $e_1, \ldots, e_p$ are elements of X, else (u is not a source of G) $L(u) = h(L(w_1), \ldots, L(w_l), e_1, \ldots, e_q)$, where $(w_1, u), \ldots, (w_l, u)$ are edges of G and $e_1, \ldots, e_q$ are elements of X (p, q and l are some constant integers). We view the labels $L(t_i)$ of the sink nodes t_i of G as the digests of X, which are computed via the above DAG G.

The authentication technique is based on the following general approach. The source and the directory store identical copies of the data structure for X and maintain the same hashing scheme on X. The source periodically signs the digests of X together with a time-stamp and sends the signed time-stamped digests to the directory. When updates occur on X, they are sent to the directory together with the new signed time-stamped digests. In this setting, the update authentication information has $O(k)$ size. When the user poses a query, the directory returns to the user (1) a signed time-stamped digest of X, (2) the answer to the query and (3) a proof consisting of a small collection of labels from the hashing scheme (or of data elements if needed) that allows the recomputation of the digest. The user validates the answer by recomputing the digest, checking that it is equal to the signed one and verifying the signature of the digest; the total time spent for this process is called the *answer verification time*. Security (against the possibility that the user verifies a, forged by the directory, proof for a non-authentic answer), typically follows from the properties of the signature scheme and the hash function.

Authentication Overhead Now we study the performance overhead due to authentication-related computations in an authenticated dictionary based on a hashing scheme (the analysis is valid for any ADS). This overhead, called *authentication overhead*, consists of *time overhead* for the (i) maintenance of the hashing scheme after updates, (ii) generation of the answer authentication information in queries, and (iii) verification of the proof of the answer; *communication overhead*, defined as the size of the answer authentication information; *storage overhead*, given by the number of hash values used by the authentication scheme; and *signature overhead*, defined as the number of signing operations performed at the source (and thus the number of signatures sent by the source). Even with the most efficient implementations, the time for computing a hash function is a few orders of magnitude larger than the time for comparing two basic numerical types (e.g., integers or floating-point numbers). Thus, the rehashing overhead dominates the update time and the practical performance of an ADS is characterized by the *authentication overhead*, which depends on the hash function h in use and the mechanism used to realize a multivariate hash function from h.

4.2 Cryptographic Hash Functions

The basic cryptographic primitive for an ADS is a *collision-resistant hash function* $h(x)$ that maps a bit string x of arbitrary length to a hash value of fixed length (e.g., 128 or 160 bits), such that collisions (i.e., distinct inputs that hash to same value) are hard to find. We refer to h simply as *hash function*. Generic constructions of hash functions are modeled by *iterative computations* [38] based on a *compression function* $f(z, y)$ that maps a string z of N bits and a string y of B bits to an output string of N bits. The input string x is preprocessed (using a *padding rule*) into a string y whose length is a multiple of B. Let $y = y_1 \| y_2 \| ... \| y_k$, where each y_i has length B and let z_0 be a public *initial value* of N bits. Then $h(x) = z_k$, where z_k is given by the following iterative application of function f: $z_1 = f(z_0, y_1), z_2 = f(z_1, y_2), \ldots, z_k = f(z_{k-1}, y_k)$.

Lemma 4. *There exist constants c_1 and c_2 such that, given an input string x of size ℓ, the iterative computation of a hash function $h(x)$ takes time $T(\ell) = c_1 \ell + c_2$.*

Note that the constants in Lemma 4 depend on the compression function in use and, thus, they may depend on some security parameter k. Still, the dependency of the hashing time on the input length is *linear*. This is a very general assumption that holds for any collision resistant function. For instance, for hash functions based on block ciphers or on algebraic structures and modular arithmetic (e.g. based on discrete logarithm using Pedersen's commitment scheme [30]) or custom designed hash function (e.g, SHA-1).

Multivariate Hash Functions Let $h(x)$ be a hash function. In order to realize a hashing scheme, we extend h to a multivariate function using *string concatenation*. Namely, we define $h_C(x_1, ..., x_d) = h(x_1 \| ... \| x_d)$. There exist alternative realizations of a multivariate hash function. For example, one may use h_C for $d = 2$ and use a binary hash tree for $d > 2$. The following lemma states that, without loss of generality, we can restrict our analysis to the concatenation hash function h_C.

Lemma 5. *Any realization of a d-variate function $h(x_1, ..., x_d)$ can be expressed by iterative applications of h_C expressed through a hashing scheme G.*

4.3 Cost of Data Authentication Through Hashing

Let G be any hashing scheme used to implement an hash-based authenticated dictionary for set X of size n, where the hash values stored in k nodes of G have been digitally signed by the source. A node with signed hash value is called a *signature* node. Hashing scheme G along with the k signature nodes can be viewed as a DAG scheme $\Gamma = (G, S, n, k)$, where S is the set of k signatures nodes in G and there are exactly n source nodes in G corresponding to elements in X. Each cost parameter of the authentication overhead, (update, query or verification time overhead and storage overhead), is expressed as some structural metric of a subgraph H of the hashing scheme G. In general, the node size $size(H)$ corresponds to the number of *hash operations* that are performed at some of the three parties (source, directory or user) and the degree size $indeg(H)$ corresponds to the total number of hash values that participate as operands in these hash operations. In particular, each cost parameter of the authentication overhead depends *linearly* on $size(H)$ and $indeg(H)$ for some subgraph H of G.

We have for any authenticated dictionary implemented in the model of data authentication through hashing.

Lemma 6. *Let $\Gamma = (G, S, n, k)$ be any hashing scheme used to implement a hash-based authenticated dictionary for set X, where special nodes are signature nodes. Let s be a source node of G storing element $x \in X$, G_s be the subgraph of G that is reachable from s, and π_s the associated path of s. We have:*

1. *an update operation on element x has update time overhead that is bounded by $comb(G_s)$;*
2. *a query operation on element x has verification time overhead that is lower bounded by $comb(\pi_s)$ and communication overhead that is lower bounded by $bnd(\pi_s)$;*
3. *the storage overhead is $size(G)$;*
4. *all involved computations are performed sequentially and hierarchically according to the hierarchy induced by G, where at any node v of G computations have time or space or communication complexity proportional to $indeg(v)$.*

Proof. For an update operation (insert or delete) on an element at source node s in G, the hash values stored in the nodes of the reachable from s subgraph G_s of G need to be updated. Updating the hash value of node u of G_s using the concatenation multivariate hash function (Lemma 5

justifies this choice) takes time $c_1 indeg(u) + c_2$ (Lemma 4), thus the update operation is performed hierarchically in time $c_1 indeg(G_s) + c_2 size(G_s)$, which is $\Omega(comb(G_s))$. For a query operation exists the query element is first located in X and suppose it is found in X [5]. Providing the proof to the user involves collecting a set of hash values that can be used to recompute a data digest. For this reason, a path of minimum authentication cost from source node s storing the query element is found and this path is exactly the associated path π_s of s, the minimum combined size path from s to a signature node. Regarding the size of proof that is sent to the user, we note that in order the user to compute the hash value that is stored at node u of G, exactly $indeg(u) - 1$ hash values need to be sent by the directory. Thus the proof consists of $bnd(\pi_s)$ labels and the communication overhead is $\Omega(bnd(\pi_s))$. In addition, the user verifies this proof by hierarchically hashing the hash values of the proof along path π_s in time $c_1 indeg(\pi_s) + c_2 size(\pi_s)$, which is $\Omega(comb(G_s))$. Finally, for the storage overhead, clearly $size(G)$ hash values are stored in the data structure at the source and the directory. □

Thus, hash-based authentication of membership queries is a hierarchical data processing problem, where operations insert/delete are related to the update cost and operation exists to the query cost of the hashing scheme in use. Theorems 1, 2 and 3 suggest that, for the dictionary problem, signing more than one hash values does not help and tree hashing structures are optimal. Finally, these theorems and the fact that signature nodes are signed periodically also translate to the following.

Theorem 6. *In the data authentication model through hashing, any hashing scheme with k signature nodes that implements an authenticated dictionary of size n has*

- $\Omega(\log \frac{n}{k})$ *worst-case update and verification time overheads;*
- $\Omega(\log \frac{n}{k})$ *worst-case communication overhead; and*
- $\Omega(k)$ *signature overhead.*

Finally, Lemma 6 suggests the use of our skip-list DAG to implement an authenticated dictionary.

Theorem 7. *There exists a skip-list authenticated dictionary of size n and probability parameter p that achieves the following expected performance. For any fixed element in the skip-list and constants c_1 and c_2 that depend on the hash function h in use, the expected hashing overhead of an update or verification operation is upper bounded by $(1-p)(2c_2 + \frac{(1+p)^2}{p} c_1) \log_{\frac{1}{p}} n + O(1)$, the expected communication cost is upper bounded by $(1-p)(\frac{1+p^2}{p}) \log_{\frac{1}{p}} n + O(1)$ and the expected storage overhead is upper bounded by $(1 + pq^2 + pq + \frac{p}{q(2-pq^2)})n$, where $q = 1-p$.*

In particular, by our experimental value of $1.41n$, when $p = \frac{1}{2}$, for the node size $size(G)$ of the skip-list DAG G, such an implementation benefits low storage cost: on average only $1.41n$ hash values are stored.

5 Multicast Key Distribution Using Key-Graphs

In this section, we apply results from Sections 2 and 3 in multicast key distribution using key-graphs. We prove that key-trees are optimal compared to general key-graphs and derive logarithmic lower bounds for involved computational and communication costs. In contrast to previous *amortized* logarithmic lower bounds on the communication cost that any protocol requires [37, 26], we proof

[5] Without loss of generality we consider only positive answers. Standard techniques to authenticate negative answers have similar authentication overhead.

exact worst case lower bounds and our proof is more general, since it does not depend on any series of update operations. Even though general graphs were initially defined for this model [40], only tree DAGs have been studied. We prove the optimality of tree DAGs.

5.1 Multicast Key Distribution

The problem refers to the confidentiality security problem in multicast groups. A *group* consists of a set of n *users* and a *group key controller*. Private key cryptography is used to transmit *encrypted* multicast messages to all the users of the group. These messages are encrypted using a *group key* available to all the current users of the group. The security problem arises when updates on the group are performed, i.e., when users are added or removed from the group. The goal is to achieve *non-group confidentiality*, i.e., only members of the group can decrypt the multicast messages, and *forward (backward) secrecy*, i.e., users deleted from (added in) the group can not decrypt messages transmitted in the future (past). No collusion between users should break any of the above requirements.

5.2 Key-Graphs

In this model, the group controller, a trusted authority and, conventionally, not member of the group, is responsible for distributing secret keys to the users and replacing (updating) them appropriately after user updates (additions/removals) in the group. The idea is that a set of keys, known to the controller, is distributed to the users, so that a key is possessed by more than one user and a user possesses more than one keys. In particular, any user is required to have a *secret key* that no other user knows and all users possess a *group key*, which is used for secure (encrypted) transmissions. In this model, on any user update, a subset of the keys needs to be updated to preserve the security requirements. Some keys are used for securely changing the group key as needed and for updating previous keys that have to be replaced. That is, new keys are encrypted with existing valid keys or with other previously distributed new keys.

Two extreme, trivial and inefficient solutions in this model are the following: each user possesses only one secret to other users key, where on any user removal from the group $n-1$ messages must be transmitted; alternatively, one key for all possible subsets of group users is used, but now the number of keys grows exponentially. *Key-graphs* [39, 40] provide a framework to implement this idea. A key-graph models the possession of keys by users and the computations (key encryptions at the controller, key decryptions at the users) and message transmissions that need to be performed after any update. A key-graph is a single-sink DAG G that the group controller and users know and that facilitates group updates. Source nodes in G correspond to users and store their individual secret keys and all non source nodes correspond to keys that can be shared among many users. A user possesses *all* and *only* the keys that correspond to the subgraph G_s of G that is reachable from its source node s. On any update of the user corresponding to s, these keys have to change (the group key at the root is always among them) to achieve forward and backward secrecy. A new keys is distributed by being sent encrypted by an old or previously distributed key.

Cost parameters The cost parameters associated with key distribution using key-graphs after an update are: (i) the computational cost at the controller, the *encryption cost*, for encrypting all new keys and thus producing the messages for transmission, and the computational cost at a user, the *decryption cost*, for decrypting received messages and updating her keys, (ii) the *communication cost*, the number of transmitted messages, and (iii) the total number of keys stored at the key controller or a user. We can view a key-graph G as DAG scheme $\Gamma = (G, S, n, 1)$, where S consists of the unique sink node of G, called *group* node and n source nodes correspond to the users.

Each cost parameter of key distribution is expressed as some structural metric of a subgraph of G. The node size corresponds to keys stored at users and also to key generations and the degree size corresponds to the number of encryptions and decryption performed during the update. In particular, each cost parameter depends linearly on $size(H)$ and $indeg(H)$ for some subgraph H of G.

Definition 8 (Reduced Key-graphs). *Let $\Gamma = (G, S, n, 1)$ be a key-graph scheme and let v be a node in G. The* support *$Sup(v)$ of v is the set of source nodes of G that can reach v through directed paths, i.e., the set of users that possess the key stored at v. Note that $s \in Sup(v)$ if and only if $v \in G_s$. Let $U = \{u_1, ..., u_k\}$ be a set of nodes of G and let $T \subseteq V_{source}$ be a set of source nodes of G. We say that set U* spans *set T if*

$$\bigcup_{1 \leq i \leq k} Sup(u_i) = T.$$

A node v is said to be safe *if v is a source node or if $indeg(v) > 1$ and any node set that spans $Sup(v)$ and does not include v has size at least $indeg(v)$. Key-graph scheme Γ is said to be* reduced*, if all nodes in G are safe.*

Lemma 7. *Let $\Gamma = (G, S, n, 1)$ be a reduced key-graph scheme used for the multicast group management problem. Then we have:*

1. *an update operation on a user that corresponds to a source node s has communication cost at least $indeg(G_s)$ and encryption cost at least $comb(G_s)$;*
2. *the key-graph stores $size(G)$ keys in total; and*
3. *all involved computations are performed sequentially and hierarchically according to the hierarchy induced by G, where at any node v of G computations have time or space or communication complexity proportional to $indeg(v)$;*

Proof. (1) By the definition of a key-graph scheme, all the keys stored at nodes of G_s (i.e., $size(G_s)$ keys) need to be updated. Consider the key stored at a node $v \in G_s$. We now show that at least $\ell = indeg(v)$ messages must be broadcasted by the controller in order for this key to be updated. Either $\ell = 0$ or $\ell \geq 2$. If $\ell = 0$, then v is a source node and we need no broadcast message to update its key. If $\ell \geq 2$, then all and only the users that correspond to nodes of $Sup(v)$ need to be able to decrypt the broadcasted messages of the encrypted new key of v. Suppose that fewer than ℓ messages suffice in updating the key of v. This implies that there exists a set of nodes $U = \{u_1, ..., u_k\}$ of G such that:

- U spans $Sup(v)$;
- U does not contain v; and
- $k < \ell$.

This is a contradiction since Γ is reduced and thus v is safe, i.e., $Sup(v)$ cannot be spanned by a set not containing v of size less than ℓ. In particular, we have that $Sup(u)$ is spanned by the set of predecessor nodes $w_1, ..., w_\ell$ of v. Thus, in order to update all the nodes of G_s, the controller needs to broadcast at least $indeg(G_s)$ messages. Additionally, when sending t messages to update the key of a node v, the encryption cost is proportional to $1 + t$, since, one (new) key generation and t encryptions of this key are performed in order to create the messages. It follows that the encryption cost is lower bounded by $comb(G_s)$.

(2) By the definition of a key-graph scheme, $size(G)$ keys are stored. □

Thus, multicast key distribution using reduced key-graphs is a HDP problem, where the overhead of an update in the group is related to the update cost of the underlying DAG scheme. By

studying more carefully reduced key-graph schemes and using Theorems 1 and 2, we now prove the main result of the section.

Theorem 8. *For a multicast key distribution problem of size n using key-graphs, in the worst case, an update operation in the group requires at least $\lfloor \log n \rfloor + 1$ communication cost and $\Omega(\log n)$ encryption and decryption costs. Also, key-tree structures are optimal over general key-graphs.*

Proof. We describe a transformation $R_v(\cdot)$ on key-graphs that given a non-safe node v in G performs changes on G (edge deletions and possibly node deletions occur). Let $\Gamma = (G, S, n, 1)$ be a key-graph scheme. Transformation R_v can be applied to G only if v is a non-safe node of G such that all nodes in G_v other than v are safe in G. (Recall that G_v is the subgraph of G that is reachable from node v through directed paths.) Let v be a non-safe node in G satisfying the above property. Let $W = \{w_1, ..., w_d\}$ be the set of predecessor nodes of v, where $d = indeg_G(v)$, $d > 0$. Transformation R_v is defined as a series of steps; we consider the following two cases, depending on the in-degree d of v.

Case I (replacement) $d \geq 2$

1. since v is not safe, we can find a minimum-size set of nodes $U = \{u_1, ..., u_\ell\}$ such that U spans $Sup(v)$, U does not contain v, and $\ell < d$; note that set U may contain one or more predecessor nodes of v;
2. edges $(w_1, v), ..., (w_d, v)$ are removed from G (this step may cause some predecessors of v to become sink nodes);
3. edges $(u_1, v), ..., (u_\ell, v)$ are added to G (note that edges removed by the previous step may be reinserted by this step);
4. while G has a sink node z distinct from the special node (the unique node of set S), remove z and its incoming edges.

Case II (contraction) $d = 1$

1. let w be the predecessor of v and let $z_1, ..., z_m$ be the successor nodes of v;
2. if $m > 0$, edges $(v, z_1), ..., (v, z_m)$ are removed from G and edges $(w, z_1), ..., (w, z_m)$ are added to G;
3. edge (w, v) and node v are removed from G.

Note that, in case I, R_v possibly makes node v safe (if $\ell > 1$) and that, in case II, R_v deletes node v. Also no cycle is introduced in step 3 of case I since node v cannot reach any node u_i through a directed path, or otherwise, u_i, a node of G_v, is not safe, a contradiction regarding the precondition in applying R_v on G. Obviously, no cycle is introduced in case II.

Let $G' = R_v(G)$ be the graph that results from applying transformation R_v to graph G. We have the following:

- the set of nodes of G' is included in the set of nodes of G, and thus the number of nodes of G is greater than or equal to the number of node of G';
- the set of source nodes of G' is the same as the set of source nodes of G, and thus both G and G' have n source nodes;
- R_v deletes at least one edge, thus G' has fewer edges than G;
- in case I, v is not deleted and $Sup_G(v) = Sup_{G'}(v)$; in case II, v is deleted and $Sup_G(v) = Sup_{G'}(w)$.

Let $\Gamma' = (G', S, n, 1)$ be the corresponding transformed scheme after transformation R_v has taken place. Given any key-update algorithm A for scheme Γ, we now derive a corresponding transformed key-update algorithm A' for scheme Γ'. Transformed algorithm A' is defined as follows.

- To update the key of a node z of G' that is distinct from v, algorithm A' mimics algorithm A. Namely, it performs exactly what A performs when it updates the key of node z in G.
- Regarding node v, algorithm A' either broadcasts $\ell = indeg_{G'}(v) > 0$ messages (case I), encrypting the new key for node v using the keys stored at the predecessor nodes $u_1, ..., u_\ell$ of v in G', or it broadcasts no message if v is not a node of G' (case II). Note that v is not a source node, otherwise, since any source node is safe, transformation R_v would not have been applied.

We now show that for any source node s, the communication cost of algorithm A for updating s in Γ is greater than or equal to the communication cost of algorithm A' for updating s in Γ', i.e., the number of messages sent by A is greater than or equal to the number of messages sent by A'. We consider the following two cases.

- If $s \notin Sup_G(v)$, then $G_s = G'_s$, so algorithms A and A' have exactly the same executions and send the same number of messages.
- If $s \in Sup_G(v)$, then,
  - for case I, G'_s has fewer edges than G_s since transformation R_v reduces the in-degree of v, i.e., $indeg_{G'}(v) < indeg_G(v)$. Also, we have $indeg(G'_s) < indeg(G_s)$ and $size(G'_s) \leq size(G_s)$, i.e., G'_s has no more nodes than G_s. Algorithm A sends at least as many messages as algorithm A' does. Let k be the number of messages that algorithm A sends to update the key at v and k' be the number of messages that algorithm A' sends for the same task. We have $k' = indeg_{G'}(u) = \ell$ and $k \geq \ell$, or otherwise, if $k < \ell$, then there would be a set of nodes of size smaller than ℓ in G that spans $Sup_G(v) = Sup_{G'}(v)$ and thus, transformation R_v was not performed correctly, a contradiction.
  - for case II, G'_s has fewer edges than G_s since transformation R_v deletes one edge. We also have $indeg(G'_s) < indeg(G_s)$ and $size(G'_s) < size(G_s)$, i.e., G'_s has fewer nodes than G_s, since v is deleted. Algorithm A sends more messages than algorithm A' does: A sends at least one message to update the key of v, but A' sends no message.

Starting with key-graph scheme $\Gamma = (G, S, n, 1)$, while the current graph has non-safe vertices, we apply the above transformation repeatedly, yielding a series of graphs $G = G_0, G_1, G_2, \ldots$ and corresponding schemes $\Gamma = \Gamma_0, \Gamma_1, \Gamma_2, \ldots$, where $G_{i+1} = R_{v_i}(G_i)$, for some vertex v_i of G_i for which transformation R_{v_i} is applicable on G_i. That is, v_i is a non-safe node of G_i such that all nodes in $(G_i)_{v_i}$ other than v_i are safe in G_i. Recall that $(G_i)_{v_i}$ is the subgraph of G_i that is reachable from node v_i through directed paths. Since each transformation reduces the number of edges and does not add any new nodes and never makes a safe node non-safe, there exists a finite sequence of q transformations, indexed by nodes $v_0, ..., v_{q-1}$, resulting in a scheme Γ_q whose graph G_q is reduced (i.e., it has no non-safe nodes).

Given a key management algorithm A for Γ, let $A_0 = A$ and, for $i = 0, \ldots, q-1$, let A_{i+1} be the algorithm for Γ_{i+1} obtained by transforming algorithm A_i for Γ_i. By repeating the argument above, we have that, given a source node s, for $i = 0, \ldots, q-1$, for the update of s, algorithm A_i in Γ_i has communication cost greater than or equal to the communication cost of algorithm A_{i+1} in Γ_{i+1}.

Let s^* be a source node whose update has largest communication cost for algorithm A_q in Γ_q. For $i = 0, \ldots, q$, we define c_i to be the communication cost of algorithm A_i for Γ_i in the update of node s^*. We have that $c_i \geq c_{i+1}$ for $i = 0, \ldots, q-1$. Also, since graph G_q is reduced, by Lemma 7 and Theorem 1, we have that $c_q \geq \lfloor \log n \rfloor + 1$. Thus, we obtain that $c_0 \geq \lfloor \log n \rfloor + 1$. We conclude that for any key-management algorithm for a scheme $\Gamma = (G, S, n, 1)$, there is an update whose communication cost is at least $\lfloor \log n \rfloor + 1$.

Recall that the encryption and decryption costs are each lower bounded by the communication cost, thus, we get that for any instance of the multicast key distribution problem of size n using key graphs, there exists an update that has $\Omega(\log n)$ encryption and decryption costs. Finally, Theorem 2 implies that the best reduced key-graphs are the key-trees. □

We observe that by the proof of Theorem 8, we can view the multicast key distribution problem as a HDP problem.

6 A New Skip-List Version

From Lemma 2 and Theorem 5, we have a version of the skip-list data structure for searching in a totally ordered set $(X, \preceq)$ with expected number of comparisons $\preceq$ close to the theoretical optimal $\lfloor \log n \rfloor + 1$, up to an additive constant term. Our DAG scheme and the new skip-list version can be viewed as a *multi-way* extension of the skip-list data structure, in the same way multi-way trees (e.g., B-trees, 2-4 trees) are extension to binary search trees. We call the new skip-list version *multi-way skip-list.*

Theorem 9. *There is a* multi-way *version of a skip-list for set X of size n and probability parameter p, where the expected number of comparisons performed while searching in X for any fixed element is at most $\frac{(1-p)(1+p^2)}{p \log_2 \frac{1}{p}} \log_2 n + O(1)$, or $1.25 \log_2 n + O(1)$ for $p = \frac{1}{2}$.*

Proof. It follows from Lemma 2 and Theorem 5. Since for DAG scheme $\Delta = (T, r, n, 1)$, T is a directed tree, from Lemma 2 we get a transformation of T into a search tree T' for set X, where interior nodes of T' are assigned with elements from X. It follows that every element x_s in X stored at source node s of T' can be located with $bnd(\pi_s)$ comparisons, which completes the proof. □

Regarding the above new skip-list version, the idea is to use our skip-list DAG scheme to create a *search tree structure* in the skip-list for searching elements of X. The multi-way skip-list data structure is a appropriately modified version of the regular skip-list data structure, such that the search tree T' of Theorem 9 is implicitly represented in the skip-list and searches are performed according to search tree T'. That is, we can keep the simplicity in creating and updating a skip-list, but we can save element comparisons by using our skip-list DAG. Note that for bridges of size $k \geq 2$, by keeping the appropriate elements that advance the search in this bridge, $k-1$ (instead of k) comparisons are needed. Skipping most of the details, to implement this skip-list version, elements that are used in a bridge b to advance the search in a child bridge of b must be stored in the *entrance* (top-left) list node of the bridge. Elements are inserted and deleted as in the regular skip-list data structure. The update operations should only maintain the appropriate elements at the entrance list nodes of the bridges traversed by the update; this can be easily achieved but appropriately modifying the search procedures.

We note that also in the regular skip-list any search corresponds to a path in a search tree. This tree-like interpretation of skip-lists is known in the literature (e.g., see [27]). Our result provides a new tree-like interpretation of skip-list with closer to optimal search performance. In particular, with respect to the expected number of comparison in a search, the multi-way skip-list version achieves the best known performance for a skip-list. Indeed, in [32, 33] it is shown that the expected number of comparisons for a search in a skip-list with probability parameter p is $(\log_2 n)/(p \log_2 \frac{1}{p}) + O(1)$. In the same work, an improved – in terms of number of comparisons – skip-list version gives $\frac{1-p^2}{p \log_2 \frac{1}{p}} \log_2 n + O(1)$ expected comparisons for a search. We are not aware of any improved skip-list based scheme with better (smaller) logarithmic constant. Our multi-way skip-list version reduces

the expected number of comparisons down to $\frac{(1-p)(1+p^2)}{p\log_2\frac{1}{p}}\log_2 n+O(1)$. For instance, when $p=\frac{1}{2}$, the expected number of comparisons when searching in the (improved) standard skip-list achieves is $1.5\log n+O(1)$, whereas, using the new multi-way skip-list the expected number of comparisons for searching an element is $1.25\log n+O(1)$, that is, the logarithmic constant of the expected number of comparisons for a search drops from 1.5 to 1.25. Interestingly, multi-way skip-lists have a more explicit tree structure than standard skip-lists, thus in fact, they constitute a new *randomized* search tree (see, e.g.,[2, 23]).

7 Conclusions and Future Work

In this paper, we introduced the concept of hierarchical data processing. This concept defines a new class of problems which models computations on elements and associated values that share certain properties. Computations are carried out hierarchically, according to the structure of an associated with the problem directed acyclic graph. Moreover, all costs that are related to the computations are fully characterized by the associated DAG, where in principle these costs depend on certain structural properties of the graph. Hierarchical data processing constitutes an interesting new theoretical framework for analyzing computational problems and designing efficient data structuring techniques.

We have proved logarithmic lower bounds for several cost measures related to hierarchical data processing, by studying structural measures of any DAG and by relating these measures to the number of comparison performed in a search structure. Overall, we have drawn an analogy between hierarchical data processing problems and searching by comparison. This connection not only serves as the basis for our theoretical results on the complexity of hierarchical data processing problems, but also provides us with an interesting and useful explanation of the computational difficulty that is intrinsic in them. We also proved the optimality of tree structures for any hierarchical data processing problem.

In view of the logarithmic lower bounds and the optimality of trees, we have also designed and analyzed a new randomized tree-like DAG. This DAG enjoys certain nice properties with respect to the cost measures related to the complexity of hierarchical data processing problems. Through the computational equivalence between hierarchical data processing and searching by comparison, we also get a new version of skip-list, where the expected number of comparisons of a search if closer to the theoretical optimal. This skip-list version improves the performance of previous versions.

We further apply our framework of hierarchical data processing to model two information security problems. We prove logarithmic lower bounds and efficient constructions for authenticated dictionaries in the model of data authentication through cryptographic hashing as well as for multicast key distribution using key-graphs. We show how these problems involve hierarchical data processing and accordingly get new results. We believe that more applications and more problems can be modelled as hierarchical data processing problems. Our framework not only provides a unified treatment of two interesting and seemingly unrelated problems, but also provides a general tool in studying other computational problems.

As future work on the subject, we plan to further investigate hierarchical data processing. Regarding the generality of our framework we wish to apply our framework in modelling more problems. Moreover, the trade-off between two important computational measured related to hierarchical data processing which was observed for two important data structures, needs further investigation. We would like to be able to find how other data structures behave with respect to this trade-off and also find data structures that achieve the best performance in this range. Additionally, we wish to study in more detail the performance of multi-way skip-lists. We have

implemented multi-way skip-lists and wish to compare them both theoretically and experimentally in terms of performance with other randomized search trees (e.g., treaps) or other multi-way trees (e.g, 2-3 or 2-4 trees). We leave as an open problem the derivation of non-trivial bounds on new cost measures of DAG schemes that characterize the complexity of different class of problems.

References

[1] A. Anagnostopoulos, M. T. Goodrich, and R. Tamassia. Persistent authenticated dictionaries and their applications. In *Proc. Information Security Conference (ISC 2001)*, volume 2200 of *LNCS*, pages 379–393. Springer-Verlag, 2001.

[2] C. Aragon and R. Seidel. Randomized search trees. In *Proc. 30th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 540–545, 1989.

[3] A. Buldas, P. Laud, and H. Lipmaa. Accountable certificate management using undeniable attestations. In *ACM Conference on Computer and Communications Security*, pages 9–18. ACM Press, 2000.

[4] J. Camenisch and A. Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *Proc. CRYPTO*, 2002.

[5] R. Canetti, J. Garay, G. Itkis, D. Micciancio, M. Naor, and B. Pinkas. Multicast security: A taxonomy and some efficient constructions. In *INFOCOMM'99*, pages 708–716, 1999.

[6] R. Canetti, T. Malkin, and K. Nissim. Efficient communication - storage tradeoffs for multicast encryption. In *Advances in cryptology (EUROCRYPT'99), LNCS 1592*, pages 459–474, 1999.

[7] P. Devanbu, M. Gertz, A. Kwong, C. Martel, G. Nuckolls, and S. Stubblebine. Flexible authentication of XML documents. In *Proc. ACM Conference on Computer and Communications Security*, pages 136–145, 2001.

[8] P. Devanbu, M. Gertz, C. Martel, and S. G. Stubblebine. Authentic data publication over the internet. *Journal of Computer Security*, 11(3):291 – 314, 2003.

[9] I. Gassko, P. S. Gemmell, and P. MacKenzie. Efficient and fresh certification. In *Int. Workshop on Practice and Theory in Public Key Cryptography (PKC '2000)*, volume 1751 of *LNCS*, pages 342–353. Springer-Verlag, 2000.

[10] M. T. Goodrich, J. Lentini, M. Shin, R. Tamassia, and R. Cohen. Design and implementation of a distributed authenticated dictionary and its applications. Technical report, Center for Geometric Computing, Brown University, 2002. Available from `http://www.cs.brown.edu/cgc/stms/papers/stms.pdf`.

[11] M. T. Goodrich, M. Shin, R. Tamassia, and W. H. Winsborough. Authenticated dictionaries for fresh attribute credentials. In *Proc. Trust Management Conference*, volume 2692 of *LNCS*, pages 332–347. Springer, 2003.

[12] M. T. Goodrich, J. Z. Sun, and R. Tamassia. Efficient tree-based revocation in groups of low-state devices. In *Advances in Cryptology – CRYPTO 2004*, Lecture Notes in Computer Science. Springer-Verlag, 2004.

[13] M. T. Goodrich and R. Tamassia. Efficient authenticated dictionaries with skip lists and commutative hashing. Technical report, Johns Hopkins Information Security Institute, 2000. Available from `http://www.cs.brown.edu/cgc/stms/papers/hashskip.pdf`.

[14] M. T. Goodrich, R. Tamassia, and J. Hasic. An efficient dynamic and distributed cryptographic accumulator. In *Proc. of Information Security Conference (ISC)*, volume 2433 of *LNCS*, pages 372–388. Springer-Verlag, 2002.

[15] M. T. Goodrich, R. Tamassia, and A. Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing. In *Proc. 2001 DARPA Information Survivability Conference and Exposition*, volume 2, pages 68–82, 2001.

[16] M. T. Goodrich, R. Tamassia, N. Triandopoulos, and R. Cohen. Authenticated data structures for graph and geometric searching. In *Proc. RSA Conference—Cryptographers' Track*, pages 295–313. Springer, LNCS 2612, 2003.

[17] J. Goshi and R. E. Ladner. Algorithms for dynamic multicast key distribution trees. In *Proc. of the twenty-second Annual Symposium on Principles of Distributed Computing (PODC 2003)*, pages 243–251. ACM, 2003.

[18] D. Knuth. *The art of computer programming.* Addison-Wesley, 1973.

[19] P. C. Kocher. On certificate revocation and validation. In *Proc. Int. Conf. on Financial Cryptography*, volume 1465 of *LNCS*. Springer-Verlag, 1998.

[20] P. Maniatis and M. Baker. Enabling the archival storage of signed documents. In *Proc. USENIX Conf. on File and Storage Technologies (FAST 2002)*, Monterey, CA, USA, 2002.

[21] P. Maniatis and M. Baker. Secure history preservation through timeline entanglement. In *Proc. USENIX Security Symposium*, 2002.

[22] C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, and S. G. Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39(1):21–41, 2004.

[23] C. Martínez and S. Roura. Randomized binary search trees. *Journal of the ACM*, 45(2):288–323, 1998.

[24] R. C. Merkle. A certified digital signature. In G. Brassard, editor, *Proc. CRYPTO '89*, volume 435 of *LNCS*, pages 218–238. Springer-Verlag, 1989.

[25] S. Micali, M. Rabin, and J. Kilian. Zero-Knowledge sets. In *Proc. 44nd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 80–91, 2003.

[26] D. Micciancio and S. Panjwani. Optimal communication complexity of generic multicast key distribution. In *Advances in Cryptology — EUROCRYPT 2004*, volume 3027 of *Lecture Notes in Computer Science*, pages 153–170. Springer Verlag, 2004.

[27] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, New York, NY, 1995.

[28] M. Naor and K. Nissim. Certificate revocation and certificate update. In *Proc. 7th USENIX Security Symposium*, pages 217–228, Berkeley, 1998.

[29] R. Ostrovsky, C. Rackoff, and A. Smith. Efficient consistency proofs for generalized queries on a committed database. In *Proc. 31th International Colloquium on Automata, Languages and Programming (ICALP)*, 2004.

[30] T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In J. Feigenbaum, editor, *Advances in Cryptology – CRYPTO '91*, volume 576 of *Lecture Notes in Computer Science*, pages 129–140. Springer Verlag, 1992.

[31] D. J. Polivy and R. Tamassia. Authenticating distributed data using Web services and XML signatures. In *Proc. ACM Workshop on XML Security*, 2002.

[32] W. Pugh. Skip list cookbook. Technical Report CS-TR-2286, Dept. Comput. Sci., Univ. Maryland, College Park, MD, July 1989.

[33] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

[34] O. Rodeh, K. P. Birman, and D. Dolev. Using AVL trees for fault tolerant group key management. *International Journal on Information Security*, pages 84–99, 2001.

[35] R. Sedgewick. *Algorithms in C++.* Addison-Wesley, Reading, MA, 1992.

[36] M. Shin, C. Straub, R. Tamassia, and D. J. Polivy. Authenticating Web content with prooflets. Technical report, Center for Geometric Computing, Brown University, 2002. `http://www.cs.brown.edu/cgc/stms/papers/prooflets.pdf`.

[37] J. Snoeyink, S. Suri, and G. Varghese. A lower bound for multicast key distribution. In *Proc. INFOCOMM*, 2001.

[38] D. R. Stinson. *Cryptography: Theory and Practice, Second Edition.* CRC Press Series, 2002.

[39] D. M. Wallner, E. G. Harder, and R. C. Agee. RFC 2627 – Key management for multicast: issues and architecture, Sep. 1998.

[40] C. K. Wong, M. Gouda, and S. S. Lam. Secure group communications using key graphs. *IEEE/ACM Transactions on Networking*, 8(1):16–30, 2000.