Compressed Set Representations based on Set Difference

Travis Gagie[1,3], Meng He[1], Gonzalo Navarro[2,3]

[1] Dalhousie University, Canada, mhe@cs.dal.ca,Travis.Gagie@dal.ca
[2] Dept. of Computer Science, University of Chile, gnavarro@dcc.uchile.cl
[3] Center for Biotechnology and Bioengineering (CeBiB)

Abstract. We introduce a compressed representation of sets of sets that exploits how much they differ from each other. Our representation supports access, membership, predecessor and successor queries on the sets within logarithmic time. In addition, we give a new MST-based construction algorithm for the representation that outperforms standard ones.

1 Introduction

The goal of compact data structures is to represent combinatorial objects in space close to their compressibility limit, so that the representation can efficiently answer a desired set of queries on the objects, without the need to decompress them [32]. Given the incomputability of Kolmogorov's absolute notion of compressibility, the notion of compressibility limit varies depending on the application and on the kind of regularities one expects to exploit from the data.

In this paper we focus on representing a set $\mathcal{S}$ of finite sets, each drawn from a totally ordered universe $\mathcal{U}$ of elements, and what we aim to exploit is the fact that some sets may be similar to others, that is, their symmetric difference, $S \triangle S'$, may be small. In such a case, we can represent S in terms of S' in space proportional to $|S \triangle S'|$, by listing which elements we have to insert into or delete from S' in order to obtain S (or, symmetrically, we can represent S' from S).

Let $\Delta(\mathcal{S})$ be the size of a representation of $\mathcal{S}$ based on encoding symmetric differences. Our goal in this paper is to show that a representation of $\mathcal{S}$ in space $O(\Delta(\mathcal{S}))$ can provide efficient general access to the compressed data. Concretely, we focus on providing the following functionality:

Membership: Determine if some element $x \in \mathcal{U}$ belongs to some set $S \in \mathcal{S}$.
Access: Obtain the ith smallest element of some set $S \in \mathcal{S}$, for any $1 \leq i \leq |S|$.
Rank: Count the number of elements $\leq x$ that exist in $S \in \mathcal{S}$, for some $x \in \mathcal{U}$.
Predecessor and successor: Find the largest element $\leq x$ and the smallest element $\geq x$ in a set $S \in \mathcal{S}$.

Those fundamental queries enable many operations on sets of sets, such as for example union and intersection of sets in $\mathcal{S}$. Our representation supports the fundamental queries in time $O(\log |\mathcal{U}|)$ or less, which is a low overhead anyway incurred in many cases, even with explicit (uncompressed) representations.

Sets of sets arise in many applications. For example, the rows of a Boolean matrix can be viewed as the characteristic vectors of sets, in which case our membership query corresponds to accessing individual matrix cells, while access (as well as predecessor/successor) corresponds to collecting the 1s in a row. Since a graph can be represented as its Boolean adjacency matrix, we can also view it as a set of sets, in which case membership corresponds to asking for the existence of individual edges and access to traversing the neighbors of a node. Our structures then offer both dense and sparse representation functionality and can be used to run a variety of algorithms, such as matrix multiplication and graph traversals, while they are represented within space $O(\Delta(\mathcal{S}))$. This idea has a rich history of developments and applications, which we defer to Section 5.

A close predecessor of our work is the so-called "containment entropy" [3], which represents sets as subsets of others: if $S \subset S'$, then they can represent S by telling which elements of S' it contains. They define a compressibility measure corresponding to the best such representation, and show that within that space they can support the five queries mentioned above on each set S in time $O(\log(|\mathcal{U}|/|S|))$. This can be called "deletion compressibility": it specifies what to delete from S' to obtain S. Our measure $\Delta(\mathcal{S})$ is coarser, in the sense that it measures the number of elements and not the exact number of bits to represent them, but it is more general because it allows insertions and deletions.

We start by defining the simpler concept of "insertion compressibility", $\mathcal{I}(\mathcal{S})$, where we describe S' via the $|S' \setminus S|$ elements we must add to S to obtain S'. This is the dual of deletion compressibility: in terms of number of elements, $|S' \setminus S|$, they differ only because the deletion compressibility starts from the set of all elements, $\mathcal{U}$, whereas the insertion compressibility starts from the set $\emptyset$.

We then define the more powerful "set-difference compressibility", $\Delta(\mathcal{S}) \leq \mathcal{I}(\mathcal{S})$, as the size of the minimum arrangement where sets can be defined in terms of others by specifying insertions and deletions to make, starting from basic sets $\emptyset$ and $\mathcal{U}$. We solve the five basic queries within time $O(\log|\mathcal{U}|)$ and space $O(\Delta(\mathcal{S}))$.

The five queries, both on the structures of space bounded by $O(\mathcal{I}(\mathcal{S}))$ and $O(\Delta(\mathcal{S}))$, are efficiently supported on top of the so-called "tree extraction" framework [29, 28], relatively directly in the case of insertion compressibility, and requiring new ideas in the case of set-difference compressibility. In particular, we show how to access the ith smallest element among the ancestors of some node, when some nodes in the path can "delete" symbols that exist upward in the path. This solution can have independent interest.

As a byproduct, we give improved results for the MST-based construction of the compressed matrix representation of Alves et al. [6], which multiplies it by a dense vector in time $O(\Delta(\mathcal{S}))$. This is a crucial aspect to make this representation usable, up to the point that previous works [13] preferred to use approximate MSTs. We also show how to compute insertion compressibility, $\mathcal{I}(\mathcal{S})$, which can be used directly to improve the time to compute the containment entropy [3].

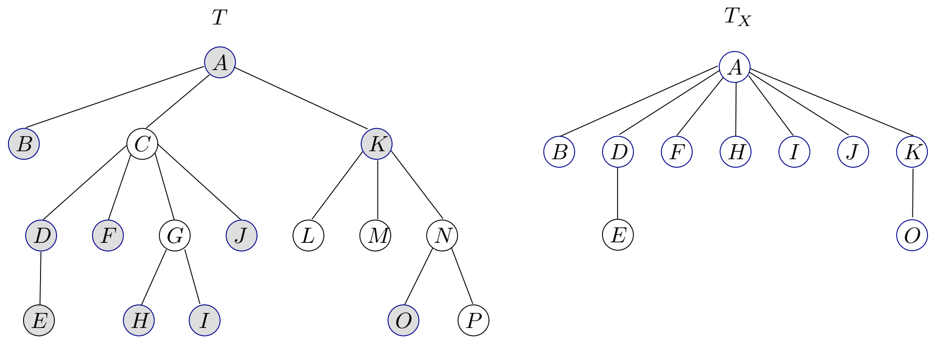

Fig. 1. An example of tree extraction, in which the original tree T is shown on the left, $X = \{A, B, D, E, F, H, I, J, K, O\}$ is the set of shaded nodes, and the extracted tree T_X is shown on the right.

2 Preliminaries and the Tree Extraction Framework

Throughout this paper, we adopt the word RAM model with ω-bit words. As we heavily use the tree extraction framework [29, 28] to obtain our results, we first present it in this section. Tree extraction, a key process of this framework, works as follows: Given a tree T and a subset, X, of its nodes including the root, we construct a new tree by deleting the nodes of T that are not in X. Whenever a node v is deleted, its children are inserted in its place as the children of v's parent, preserving the original left-to-right order among nodes. Figure 1 shows an example.

He, Munro and Zhou [28] use tree extraction to represent a labeled tree to support labeled navigational operations. Given an ordinal tree $\mathcal{T}$ with $|\mathcal{T}|$ nodes, each with a label from universe $\mathcal{U} = \{1, 2, \ldots, u\}$, they define a subtree $\mathcal{T}_\alpha$ for each $\alpha \in \mathcal{U}$ by using on $\mathcal{T}$ the tree extraction process described in the previous paragraph, for the set X_α consisting of the nodes labeled α, their parents and the root. Thus, $\sum_{\alpha \in \mathcal{U}} |\mathcal{T}_\alpha| = O(|\mathcal{T}|)$. The nodes of the trees $\mathcal{T}_\alpha$ are marked with values 0 or 1, so that nodes marked 1 correspond to nodes of $\mathcal{T}$ labeled α. He et al. show how to compute, in $O(\log \log_\omega u)$ time, a mapping $f_{\mathcal{T}}(v, \alpha) = v'$, where v is a node from $\mathcal{T}$ and v' is a node from T_α, so that the number of nodes labeled α from v to the root of $\mathcal{T}$ equals the number of nodes labeled 1 from v' to the root of $\mathcal{T}_\alpha$. Their representation is built in $O(|\mathcal{T}| \log u)$ time and occupies $|\mathcal{T}| \lg u + O(|\mathcal{T}|) + o(|\mathcal{T}|) \cdot \lg u$ bits, or $O(|\mathcal{T}|)$ words, of space. It supports these operations in time dominated by the cost to compute $f_{\mathcal{T}}$:

$\texttt{parent}_\alpha(v)$**:** the lowest ancestor of v labeled α (or $\perp$ if there is none).
$\texttt{rank}_\alpha(v)$**:** the number of ancestors of v labeled α.
$\texttt{select}_\alpha(v, i)$ **:** the ith highest ancestor of v labeled α.

In a later paper, He, Munro and Zhou [29] extend the concept of a *wavelet tree* [27, 31] to support more sophisticated operations called *path queries*. They partition the universe $\mathcal{U} = \mathcal{U}_{1,u}$ into subuniverses $\mathcal{U}_{a,b} = \{a, a+1, \ldots, b\}$ via successive halving: starting from $a = 1$ and $b = u$, they partition $\mathcal{U}_{a,b}$ into $\mathcal{U}_{a,m}$ and $\mathcal{U}_{m+1,b}$, where $m = \lfloor (a+b)/2 \rfloor$. They store trees $\mathcal{T}_{a,b}$ labeled with 0 and 1, where each node $v \in \mathcal{T}_{a,b}$ is labeled 0 if the label of v belongs to $\mathcal{U}_{a,m}$ and 1 if it belongs to $\mathcal{U}_{m+1,b}$. Starting with $\mathcal{T}_{1,u} = \mathcal{T}$, they use tree extraction to define $\mathcal{T}_{a,m} = (\mathcal{T}_{a,b})_0$ and $\mathcal{T}_{m+1,b} = (\mathcal{T}_{a,b})_1$. The partition ends at the trees of the form $\mathcal{T}_\alpha = \mathcal{T}_{\alpha,\alpha}$, for $\alpha \in \mathcal{U}$. Because the universe is just {0,1}, they store each tree $\mathcal{T}_{a,b}$ using $O(|\mathcal{T}_{a,b}|)$ bits, so that the total hierarchical partition uses $O(|\mathcal{T}| \log u)$ bits, or $O(|\mathcal{T}|)$ space. It is built in time $O(|\mathcal{T}| \log u)$. This arrangement can be used to answer the following queries:

Path counting: Given $v \in \mathcal{T}$ and labels $a \leq b$, counts the number of ancestors of v with label between a and b.

Path selection: Given $v \in \mathcal{T}$ and an integer i, finds the ancestor of v with the ith smallest label.

This hierarchical partitioning of the universe enables $O(\log u)$ time support for path queries. He et al. then improve the query time to $O(\log_\omega u)$ by decomposing the universe into a sublogarithmic number of subuniverses each time [29]. Their improved approach achieves succinct space simultaneously: when $u \leq \omega^\epsilon$ for some constant $\epsilon \in (0,1)$, the space cost is $|\mathcal{T}|(\lg u + 2) + o(|\mathcal{T}|)$ bits; otherwise, it uses $|\mathcal{T}| \lg u + O(\frac{|\mathcal{T}| \lg u}{\lg \lg |\mathcal{T}|})$ bits.

3 Warm-up: Insertion Compressibility

We start with a weaker compressibility definition that only captures the fact that some subsets can be (large) subsets of others, and thus can be used to represent the containing subset within little space. This definition, as explained, is the dual of the "containment entropy" [3], which focuses on representing the contained set in terms of the containing one.

Definition 1. *Let $\mathcal{S}$ be a set of sets. For each $S \in \mathcal{S}$, let $p(S)$ be a largest strict subset of S in $\mathcal{S}$, or $\emptyset$ if no such subset exists. Then the* insertion compressibility *of $\mathcal{S}$ is*

$$\mathcal{I}(\mathcal{S}) \quad = \quad \sum_{S \in \mathcal{S}} |S \setminus p(S)| \quad = \quad \sum_{S \in \mathcal{S}} |S| - |p(S)|.$$

To represent a set of sets $\mathcal{S}$ within $O(\mathcal{I}(\mathcal{S}))$ space, we define a so-called "insertion graph".

Definition 2. *Let $\mathcal{S}$ be a set of sets, with $p(S)$ according to Def. 1. The* insertion graph *of $\mathcal{S}$ is a weighted directed graph with nodes $\mathcal{S} \cup \{\emptyset\}$ and an edge from each set $S \in \mathcal{S}$ to $p(S)$, with weight $|S| - |p(S)|$.*

It is easy to see that the insertion graph is equivalent to a tree rooted at $\emptyset$, whose weights add up to $\mathcal{I}(\mathcal{S})$; see the left of Figure 2. By storing $S \setminus p(S)$ at each node S of the graph, we spend $O(\mathcal{I}(\mathcal{S}))$ space and can reconstruct $\mathcal{S}$.

We will use a slightly different representation, in the form of a labeled tree, still of size $O(\mathcal{I}(\mathcal{S}))$, in order to efficiently support the five queries on $\mathcal{S}$.

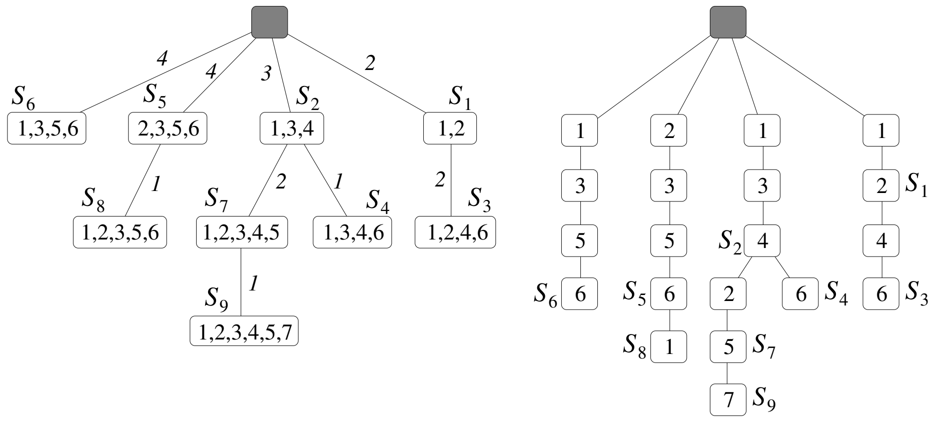

Fig. 2. On the left, an insertion graph for sets $\mathcal{S} = \{S_1, \ldots, S_9\}$, with the weights written as slanted values on the edges. It holds $\mathcal{I}(\mathcal{S}) = 20$. On the right, a corresponding insertion tree with 21 nodes. We write S_i besides its corresponding node $v(S_i)$.

Definition 3. *Let $\mathcal{S}$ be a set of sets, with $p(S)$ according to Def. 1. The* insertion tree *of $\mathcal{S}$ is defined as follows.*

- *The root of the tree is the empty set, and is called $v(\emptyset)$.*
- *Every set $S \in \mathcal{S}$ is a node, which we call $v(S)$, the node associated with S (but there may be other tree nodes that are not associated with any sets).*
- *Let $I(S) = S \setminus p(S)$. Then the tree has a chain of $|I(S)|$ edges from $v(p(S))$ to $v(S)$, each labeled with a distinct element of $I(S)$, in arbitrary order.*

The insertion tree of $\mathcal{S}$ clearly has $1 + \mathcal{I}(\mathcal{S})$ nodes; see the right of Figure 2. We represent it in $O(\mathcal{I}(\mathcal{S}))$ space using the tree extraction framework. More precisely, we represent the insertion tree using the labeled tree structure of [28] to enable $O(\log\log_\omega |\mathcal{U}|)$-time support of $\texttt{parent}_\alpha$, $\texttt{rank}_\alpha$ and $\texttt{select}_\alpha$, and we additionally represent it using the labeled tree structure of [29] to support path counting and path selection queries in $O(\log_\omega |\mathcal{U}|)$ time. These two structures use $2|\mathcal{I}(\mathcal{S})| \lg u + O(|\mathcal{I}(\mathcal{S})|) + o(|\mathcal{I}(\mathcal{S})|) \cdot \lg u$ bits in total, which is within $O(\mathcal{I}(\mathcal{S}))$ words of space. We now describe how the basic operations can be supported on any set S.

Membership To find out whether $x \in S$, we locate $v(S)$ (which we can directly associate with S within the space budget). We then reduce the query to the primitive $\texttt{parent}_x(v(S))$ [28] (which finds the closest ancestor of $v(S)$ labeled x), returning true iff there is one. This takes time $O(\log\log_\omega |\mathcal{U}|)$.

Access To access the ith smallest element of S, we retrieve the ith smallest symbol on an edge in the path from $v(S)$ to the root. This corresponds to a path selection query in our insertion tree, requiring time $O(\log_\omega |\mathcal{U}|)$.

Rank A rank for x in S corresponds to counting the number of labels of ancestors of $v(S)$ that are at most x. This is solved with a path counting query with interval $[1, x]$ in $O(\log_\omega |\mathcal{U}|)$ time.

Predecessor/Successor To find the predecessor of x in S we compute the rank i of x in S. If $i = 0$, then x has no predecessor in S; otherwise we obtain it by accessing the ith smallest element of S. For successor, if the predecessor of x is not x, we access and return instead the $(i+1)$st smallest element of S, or return that there is no successor if i is the depth of $v(S)$ in the insertion tree. This entire process uses $O(\log_\omega |\mathcal{U}|)$ time.

The insertion tree is easily built from the insertion graph in time $O(\mathcal{I}(\mathcal{S}))$; the costly part is to find $p(S)$ for every $S \in \mathcal{S}$. Let $s = |\mathcal{S}|$, $n = \sum_{S \in \mathcal{S}} |S|$, and $u = |\mathcal{U}|$. We can first sort the sets by increasing size in time $O(s \log s)$, and insert them one by one as nodes in the insertion graph (which initially contains only the node $\emptyset$) with an edge towards $p(S)$. To insert S, we sort its elements (which induces total time $O(n \log u)$) and then traverse the current graph in DFS order from the node $\emptyset$, finding the largest sets $S' \subset S$. Checking such containments takes time $O(|S| + |S'|)$ with a merge-like algorithm. This process takes time $O(\sum_{S,S'}(|S| + |S'|)) = O(sn)$. If we store the sets $S \setminus p(S)$ along this same process, the insertion graph is built in additional time $O(\mathcal{I}(\mathcal{S})) \subseteq O(n)$. From the insertion graph, we build the insertion tree in time $O(\mathcal{I}(\mathcal{S}))$ and the tree extraction data structure in additional time $O(n \log u)$.

Tree extraction assumes that $\mathcal{U} = [1..u]$. If this is not the case, we collect all the n elements, sort them in $O(n \log u)$ time, and assign to the u distinct values integers in $[1..u]$. Queries and answers can be translated in constant time.

Theorem 1. *On a word RAM with ω-bit words, a set of s sets $\mathcal{S}$, over a universe of size $u = |\cup_{S \in \mathcal{S}} S|$, can be represented within $O(\mathcal{I}(\mathcal{S}))$ space so that access, rank, predecessor and successor queries can be carried out in time $O(\log_\omega u)$ and membership in time $O(\log \log_\omega u)$. If $n = \sum_{S \in \mathcal{S}} |S|$, then the data structure can be built in time $O(n \log u + sn)$.*

In the next section we give, as a byproduct, an improved construction time; see Theorem 3.

4 Symmetric-Difference Compressibility

We now give our definition of symmetric-difference (or *symdiff* for short) compressibility, which allows expressing a set by both inserting in or removing elements from some other set. This is strictly stronger than both insertion and deletion compressibility.

Definition 4. *A* symdiff graph *on a set of sets $\mathcal{S}$ is a weighted directed graph on the nodes $\mathcal{S} \cup \{\emptyset, \mathcal{U}\}$, where $\mathcal{U} = \cup_{S \in \mathcal{S}} S$. There is exactly one edge leaving from each set $S \in \mathcal{S}$. The target node of that edge is called $p(S)$ and the weight of the edge is $|S \triangle p(S)|$, the size of the symmetric difference. Further, there is a path from every $S \in \mathcal{S}$ to $\emptyset$ or to $\mathcal{U}$. The* weight *of a symdiff graph is the sum of the weights of its edges.*

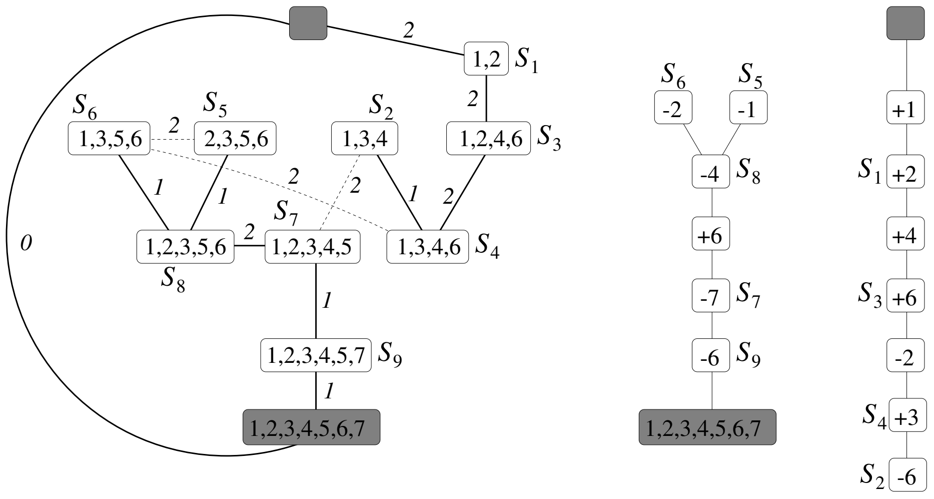

Fig. 3. On the left, a symdiff graph of minimum weight between the same sets of Figure 2, yielding $\Delta(\mathcal{S}) = 13$. To build it, it is sufficient to consider the full graph edges with weights up to $\ell = 2$. Among these edges, those not belonging to the MST (i.e., the symdiff graph of minimum weight) are dashed. On the right, the indel trees, rooted at $\mathcal{U}$ (upside down) and at $\emptyset$, that represent $\mathcal{S}$ in space $O(\Delta(\mathcal{S}))$. We write $+x$ and $-x$ to indicate elements x to add or to delete, respectively, from the parent node.

Note that a symdiff graph can be regarded as two rooted trees, one rooted at $\emptyset$ and the other rooted at $\mathcal{U}$; see the left of Figure 3. Given a symdiff graph for $\mathcal{S}$, we can represent at each node S only those elements in $S \setminus p(S)$ and those in $p(S) \setminus S$, so that we can reconstruct S from $p(S)$. The total space incurred is the weight of the two trees. This is a slight variation over the model of Alves et al. [6], which use only one tree rooted at $\emptyset$.

Definition 5. *The* symdiff compressibility *of a set of sets $\mathcal{S}$, $\Delta(\mathcal{S})$, is the minimum weight of a symdiff graph on $\mathcal{S}$.*

We now describe a representation that uses $O(\Delta(\mathcal{S}))$ space and supports the five basic queries in the almost same times as with insertion compressibility.

As explained, we will represent the two trees, storing at each node the symmetric difference with its parent node. We will later modify this tree and add more structures in order to support efficient queries on the sets.

4.1 Construction

Analogously to what Alves et al. [6] do on a slightly different model, we can build the lightest symdiff graph as the minimum spanning tree (MST) over the full graph with nodes $S \in \mathcal{S} \cup \{\emptyset, \mathcal{U}\}$ and edge weights $w(S, S') = |S \triangle S'|$. The only exception is that, in order to obtain the best pair of trees according to our model, we set $w(\emptyset, \mathcal{U}) = 0$. We first show the correctness of this construction.

Lemma 1. *The weight of the described MST is $\Delta(\mathcal{S})$ and the two trees that achieve it are obtained by removing the edge between $\emptyset$ and $\mathcal{U}$.*

Proof. Every MST contains the lightest edge, which is the one between $\emptyset$ and $\mathcal{U}$. Removing this edge disconnects the MST into two trees, one rooted at $\emptyset$ and one rooted at $\mathcal{U}$. The sum of the weights of the two trees is the same of the MST, as we removed a zero-cost edge. On the other hand, any pair of trees rooted at $\emptyset$ and $\mathcal{U}$ can be turned into a spanning tree of the same cost by adding the zero-cost edge between $\emptyset$ and $\mathcal{U}$. It follows that the cost of the MST is $\Delta(\mathcal{S})$. $\square$

We can build the MST in time $O(s^2)$ once all the edge weights are computed as in Section 3: increasingly sort the elements of all sets in time $O(n \log u)$, and compute all the edge weights in time $O(sn)$. The total construction time is then $O(n \log u + sn)$ as for insertion compressibility. This is only slightly better than the complexity obtained by Alves et al. [6]: $O(sn + s^2 \log s)$ if the sets come already sorted. We now show that a more refined construction time is possible.

We first show that, once the sets are sorted, one can compute $S \triangle S'$ in time $O(|S \triangle S'|)$ after an $O(n \log u)$-time preprocessing of $\mathcal{S}$. We map the elements of $\mathcal{U}$ to the interval $[1..u]$ in time $O(n \log u)$ if necessary, as before. Let $S_1, \ldots, S_s$ be the mapped sets. We then build a *text* $T(\mathcal{S}) = S_1\$_1 S_2\$_2 \cdots S_s\$_s$, which is a string where all the mapped sets S_i are appended a unique terminator symbol $\$_i = u + i$ and concatenated. Since $|T(\mathcal{S})| = n + s = O(n)$, we can build the suffix tree [7, 19] of $T(\mathcal{S})$ in $O(n)$ time [23], with support for constant-time level ancestor queries [11]: $lca(u, v)$ is the deepest ancestor of suffix tree leaves u and v, and its (stored) string depth is the length of the longest common prefix of the suffixes denoted by those leaves. By storing the suffix tree leaf corresponding to each suffix $T(\mathcal{S})[i..]$, and the position where each string $S_i \in \mathcal{S}$ starts in $T(\mathcal{S})$ (all of which takes $O(n)$ space) we can, in constant time, compute $lcp(S[p..], S'[q..])$, that is, how many equal symbols follow from any $S[p..]$ and any $S'[q..]$.

With this tool we can compare S and S' as follows. We start from $p, q := 1$ and find the longest common prefix $lcp(S[p..], S'[q..]) = \ell$ in constant time. This means that the first position where the strings differ is $S[p + \ell] \neq S'[q + \ell]$. The smaller of both symbols is the first element of $S \triangle S'$. If the smaller element is $S[p + \ell]$ we set $p := p + \ell + 1$ and $q := q + \ell$; if it is $S'[q + \ell]$ we set $p := p + \ell$ and $q := q + \ell + 1$. We then continue extracting the elements of $S \triangle S'$ one by one. When both p and q reach terminators $\$_*$, we are done. Note that we can use this technique in iterator mode, finding the next difference at each call, so that we determine in time $O(k)$ whether the symmetric difference exceeds k.

Lemma 2. *After k calls to this procedure on S and S', if both p and q point to terminators $\$_*$, then $|S \Delta S'| < k$. Otherwise, there are k elements of $S \Delta S'$ listed in $S[..p-1]$ and $S'[..q-1]$, so $|S \Delta S'| \geq k$.*

Proof. This holds after $k = 0$ calls because $p = q = 1$ and it cannot be $S = S' = \emptyset$. Now assume the proposition holds after $k - 1$ calls. There is a kth call only if not both p and q point to terminators $\$_*$. The kth call then finds $\ell = lcp(S[p..], S'[q..])$, meaning there is no element of $S \Delta S'$ in $S[p..p + \ell - 1]$

or $S[q..q+\ell-1]$, but $S[p+\ell] \neq S'[q+\ell]$. Because the elements of S and S' are increasingly sorted, this means that the smallest of $S[p+\ell]$ and $S'[q+\ell]$ is in $S\Delta S'$, or both are terminators $\$_*$. Assume $S[p+\ell] \in S\Delta S'$; the case $S'[q+\ell] \in S\Delta S'$ is analogous. The algorithm sets $p := p+\ell+1$ and $q := q+\ell$, which restablishes the property because now there are $(k-1)+1$ elements of $S\Delta S'$ in $S[..p-1]$ and $s'[..q-1]$. If, on the other hand, both $S[p+\ell]$ and $S'[q+\ell]$ are terminators $\$_*$, this means there are no more elements in $S\Delta S'$ and therefore $|S\Delta S'| = k-1 < k$. □

We adapt Prim's algorithm, which grows a set V (the nodes already attached to the MST), by taking at each step a node out of V' (the nodes not yet in the MST) and moving it into V. Initially we set $V := \{\emptyset, \mathcal{U}\}$, with the zero-cost edge connecting them in the MST, and $V' = \mathcal{S}$. Unlike the classic algorithm, which knows all the weights from the beginning, we will start with all edge weights set to $+\infty$ for Prim's algorithm (except $w(\emptyset, \mathcal{U}) = 0$, which is already in the MST), and will discover the true weights incrementally, from smallest to largest.

We will maintain all the edges whose weights are yet unknown in a *bag* (initially of size $\binom{s+2}{2} - 1$). For each edge (S, S') in the bag, we initialize its iterator at $p, q := 1$, which completes the iteration $k = 0$.

Now we start the iterations $k = 1$ onwards. In the kth iteration, we advance the iterators once in all the edges (S, S') of our bag. For those edges where both p and q end up pointing to terminators $\$_*$, we know that the edge weight is $w(S, S') = k-1$, because by Lemma 2 it was $\geq k-1$ and it is $< k$. So we set the corresponding weight for Prim's algorithm and remove the edge from the bag.

The invariant is that after the kth iteration we have defined all the edge weights that are less than k (note that those edges may or may not connect nodes from V and V', so not all of them are immediately useful for Prim). We can then run some steps of Prim, until the next lowest weight to include in the MST is $+\infty$. At that point we suspend Prim's algorithm and move on to the next iteration, the $(k+1)$th, where we find all the weights equal to k. The next lemma proves the correctness of this approach.

Lemma 3. *Prim algorithm runs correctly by using only the edges of weight up to k before using any heavier edge.*

Proof. Let $E = S_k \cup L_k$ be the graph edges, with $S_k = \{e \in E,\ w(e) < k\}$ being the edges with weight less than k. At any point in Prim's algorithm, it chooses the edge $e_{\min} = \operatorname{argmin}\{w(e),\ e \in E \cap (V \times V')\}$, that is, the lightest one connecting V and V', and includes $e_{\min}$ in the MST (and one of its extremes in V). This can be rewritten as $e_{\min} = \operatorname{argmin}\{w(e),\ e \in (S_k \cup L_k) \cap (V \times V')\}$. Since $w(e) < k \leq w(e')$ for all $e \in S_k$ and $e' \in L_k$, it follows that $e_{\min} = \operatorname{argmin}\{w(e),\ e \in S_k \cap (V \times V')\}$ if $S_k \cap (V \times V') \neq \emptyset$. It is therefore correct to run Prim only on the edges S_k discovered up to iteration k as long as it finds candidate edges of weight less than k. □

For the analysis, note that, if ℓ is the heaviest weight included in the MST, then we have incremented each weight $w(S, S')$, along the algorithm, $\min(\ell, |S \triangle S'|)$

times. The total time is then $O(\sum_{S,S'} \min(\ell, |S \triangle S'|))$, which is bounded both by $O(s^2\ell)$ and by $O(sn)$ (the latter because $|S \triangle S'| \leq |S| + |S'|$).

Theorem 2. *A set of sets $\mathcal{S}$ can be represented within $O(\Delta(\mathcal{S}))$ space. If $\mathcal{S}$ has s sets, the sum of the sizes of its sets is n, they contain u distinct elements in total, and the maximum weight in a minimum-weight symdiff graph of $\mathcal{S}$ is ℓ, then that graph can be built in time $O(n \log u + \sum_{S,S' \in \mathcal{S}} \min(\ell, |S \triangle S'|)) \subseteq O(n \log u + \min(s^2\ell, sn))$.*

Improving the computation of $\mathcal{I}(\mathcal{S})$. A similar approach can speed up the construction of the insertion graph of Section 3. For every new set S, instead of fully computing the distance towards all the preceding sets S', we create a bag with those and use the same iterations from $k = 0$ until finding the first value $k-1$ associated with some set S' such that $S' \subset S$ and $|S \setminus S'| = k-1$. We then set $p(S) := S'$. The computation using the pointers p and q is a bit different, because whenever we find the distinct element to occur in S' we must discard S' because $S' \not\subset S$. Overall, we spend time $O(s \cdot |S \setminus p(S)|)$ to add S to the tree.

Theorem 3. *The data structure of Theorem 1 can be built in time $O(n \log u + s \cdot \mathcal{I}(\mathcal{S})) \subseteq O(n \log u + sn)$.*

By symmetry, this construction can be used to compute the containment entropy [3], which is based on deletions instead of insertions, and has $\mathcal{U}$ instead of $\emptyset$ at the root. Their original construction takes time $O(sn \log n)$ [3].

4.2 Supporting the operations

We will use the following data structure to represent $\mathcal{S}$ in space $O(\Delta(\mathcal{S}))$. See the right part of Figure 3.

Definition 6. *Let $\mathcal{S}$ be a set of sets, with $p(S)$ according to Def. 4. The* indel trees *of $\mathcal{S}$ are two trees defined as follows.*

- *The root of one tree, called $v(\emptyset)$, represents the empty set, and the root of the other, called $v(\mathcal{U})$, represents $\mathcal{U}$.*
- *Every set $S \in \mathcal{S}$ is a node, which we call $v(S)$, in the same tree of $v(p(S))$ (but there may be other tree nodes that are not associated with any sets).*
- *Let $I(S) = S \setminus p(S)$ and $D(S) = p(S) \setminus S$. Then the tree where $v(S)$ belongs has a chain of $|I(S)| + |D(S)|$ edges from $v(p(S))$ to $v(S)$, each labeled with a distinct element of $I(S)$ or $D(S)$, in arbitrary order. Those labels $x \in I(S)$ are written $+x$ and those $x \in D(S)$ are written $-x$.*

It is clear that the two indel trees of a minimum-weight symdiff graph have $\Delta(\mathcal{S}) + 2$ nodes in total. Again we represent either tree using the labeled tree data structures [28, 29] to support labeled operations and path queries, and this time the labels are from the set $[-u..u]$. These two structures then use $2|\Delta(\mathcal{S})| \lg u + O(|\Delta(\mathcal{S})|) + o(|\Delta(\mathcal{S})|) \cdot \lg u$ bits in total, which is within $O(\Delta(\mathcal{S}))$ words of space. We now show how the set queries are carried out using operations on labeled trees.

Membership To find whether $x \in S$, we first locate $v(S)$. We then compute $u^+ = \texttt{parent}_{+x}(v(S))$ and $u^- = \texttt{parent}_{-x}(v(S))$ [28]. If both exist, then $x \in S$ iff the depth of u^+ is larger than that of u^- (i.e., the last edit operation involving x was an insertion). If only u^+ exists, then $x \in S$. If only u^- exists, then $x \notin S$. If none exists, then $x \in S$ iff $v(S)$ descends from $v(\mathcal{U})$. As for insertion compressibility, this process takes $O(\log\log_\omega |\mathcal{U}|)$ time.

Access We show in Section 4.3 how to solve this operation in time $O(\log |\mathcal{U}|)$ with $O(\Delta(\mathcal{S}))$ words of additional space.

Rank The rank of x in S is found by counting the number of labels of ancestors of $v(S)$ in the range $[1..x]$, minus the number of labels of ancestors of $v(S)$ in the range $[-x..-1]$. If $v(S)$ descends from $v(\mathcal{U})$, we add x to this difference, because the elements $[1..u]$ are tacitly assumed to exist at the root. This computation is done with two path counting queries, in $O(\log_\omega |\mathcal{U}|)$ time.

Predecessor/Successor This is solved exactly as for insertion compressibility. As they use the access operation, their time is $O(\log |\mathcal{U}|)$.

Theorem 4. *On a word RAM with ω-bit words, a set of s sets $\mathcal{S}$, over a universe of size $u = |\cup_{S\in\mathcal{S}} S|$, can be represented in $O(\Delta(\mathcal{S}))$ space so that membership queries can be performed in time $O(\log\log_\omega u)$, rank can be performed in time $O(\log_\omega u)$, and access, predecessor and successor in time $O(\log u)$. If $n = \sum_{S\in\mathcal{S}} |S|$, then, once a minimum-weight symdiff graph is constructed (see Theorem 2), this structure can be built in $O(n \log u)$ extra time.*

4.3 Accessing with positive and negative values

We now show how to access the ith smallest element of a set S. Among the two indel trees representing $\mathcal{S}$, let $\mathcal{T}$ be the one containing the node $v = v(S)$. Let x be the element of S whose rank is i, that is, the element we are looking for. Then, because negative values $-x$ can only occur if x is already present in the set that an ancestor of v represents, the rank i of x is the number of ancestors of v with labels in $[1..x]$ minus the number of ancestors of v within labels in $[-x..-1]$) if $\mathcal{T}$ is rooted at $v(\emptyset)$; otherwise, it is this difference plus x. Thus, it is easy to use rank operations from v to binary search for x in the indel tree in time $O(\log u \log_\omega u)$. We will, instead, exploit the structure of the hierarchy described in Section 2 to do the binary search in time $O(\log u)$.

To achieve this, more preprocessing is required. For either indel tree $\mathcal{T}$, we will use the hierarchical binary universe partitioning described in Section 2 to form two hierarchies, one for positive values $[1..u]$ starting at a tree $\mathcal{T}^+$ and another for negative values $[-u..-1]$ (seen as $[1..u]$) starting at a tree $\mathcal{T}^-$. Both trees $\mathcal{T}^+$ and $\mathcal{T}^-$ are constructed from $\mathcal{T}$ via tree extraction (with labels $\{+,-\}$), using function $f_\mathcal{T}$. Figure 4 illustrates the process. Since, for either indel tree, we perform $O(\lg u)$ levels of partitioning, and, for each level, we construct a 0/1-labeled tree which occupies $O(|\Delta(\mathcal{S})|)$ bits, the extra space cost incurred here is $O(|\Delta(\mathcal{S})| \lg u)$ bits, or $O(|\Delta(\mathcal{S})|)$ words.

The main idea of our query algorithm is to start with ranges $[a..b] := [1..u]$ and, as we descend down both hierarchical partitions, shrink the difference between a and b, while maintaining the invariant that i is between the rank of a in

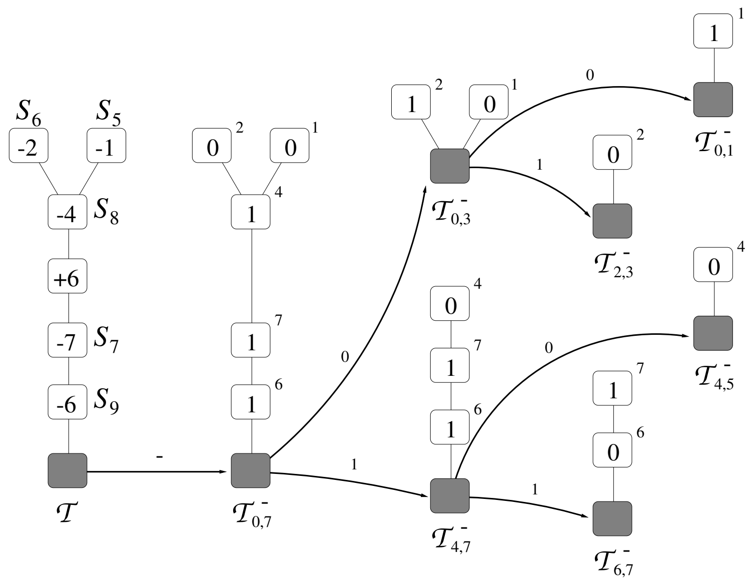

Fig. 4. The hierarchical extraction process for $\mathcal{T}^- = \mathcal{T}^-_{0,7}$, starting from the (upside down) tree $\mathcal{T}$ rooted at $\mathcal{U}$ of Figure 3. The rightward bold arrows lead from $\mathcal{T}^-_{a,b}$ to $\mathcal{T}^-_{a,m}$ (with label 0) and to $\mathcal{T}^-_{m+1,b}$ (with label 1). We show the labels of the trees $\mathcal{T}^-_{a,b}$ inside the nodes, and the original symbols in small font near the boxes.

S and the rank of b in S. To describe this algorithm in detail, we first consider the case in which $\mathcal{T}$ is rooted at $v(\emptyset)$. Letting $v = v(S) \in \mathcal{T}$, we start from $v^+ = f_{\mathcal{T}}(v, +) \in \mathcal{T}^+_{1,u}$ and $v^- = f_{\mathcal{T}}(v, -) \in \mathcal{T}^-_{1,u}$. In this process, $v^+ \in \mathcal{T}^+_{a,b}$ and $v^- \in \mathcal{T}^-_{a,b}$ will be the current nodes in both hierarchical partitions. We compute $s^+ = \mathtt{rank}_0(v^+)$ in $\mathcal{T}^+_{a,b}$ and $s^- = \mathtt{rank}_0(v^-)$ in $\mathcal{T}^-_{a,b}$. Note that $\mathtt{rank}_0$ is computed in constant time because the alphabet, $\{0, 1\}$, of $\mathcal{T}^{\pm}_{a,b}$ is of size two. Observe that, in the path of $\mathcal{T}$ between $v(\emptyset)$ and v, the number of nodes representing the insertion of an element must be either equal to or exactly one more than the number of nodes representing its deletion. It then follows that S has $s^+ - s^-$ elements in $[a..m]$, where $m = \lfloor (a + b)/2 \rfloor$. Therefore, if $s^+ - s^- \geq i$, then $x \in [a..m]$. In this case, we set $b := m$, $v^+ := f^+_{a,b}(v^+, 0) \in \mathcal{T}^+_{a,m}$, and $v^- := f^-_{a,b}(v^-, 0) \in \mathcal{T}^-_{a,m}$, where $f^{\pm}_{x,y}$, the mapping function of the tree $\mathcal{T}^{\pm}_{x,y}$, is again computed in constant time because the trees have only two labels. Otherwise, we set $a := m + 1$, $v^+ := f^+_{a,b}(v^+, 1)$, $v^- := f^-_{a,b}(v^-, 1)$, and $i := i - (s^+ - s^-)$. We iterate until we reach $a = b$, when we return the answer $x := a = b$.

When the tree root is $\mathcal{U}$ instead of $\emptyset$, we modify the above procedure by using $s^+ - s^- + (b - a + 1)$ instead of $s^+ - s^-$. As we spend $O(1)$ time on each level of $\mathcal{T}^+$ and $\mathcal{T}^-$, which have $O(\log u)$ levels, this process uses $O(\log u)$ time.

5 Applications and Previous Work

There are various applications where it is natural to encode sets of sets by their symmetric differences. An example is inverted indices in natural language text collections, which store the sets of documents where each word occurs. The phenomenon that semantically correlated words tend to appear in about the same sets of documents has been observed long ago [34][9, Ch. 3], and for example is used to detect word associations in NLP. Another example is the adjacency lists of web graphs and social networks, where the symmetric differences between such lists have been used to compress the graphs with high success [16, 15, 14]. This idea has been extended to arbitrary Boolean matrices (which in particular can be the adjacency matrices of graphs).

Elgohary et al.'s papers [21, 22] have contributed to a burst of research [1, 6, 8, 10, 20, 24, 30, 35] on compressing matrices and manipulating them in compressed form. The compression methods used by Elgohary et al. and the researchers that followed them usually treat the rows of the matrices as sequences, however — possibly after re-ordering the columns and/or rows — and thus have difficulty taking advantage of repetitions of a pair of values in two columns when they are separated by columns whose contents vary.

In contrast, over thirty years ago Bookstein and Klein [17] proposed compressing collections of bitvectors as collections of sets: we can express one set in terms of a similar one by recording their symmetric difference, and we can find the best way to express a collection this way by considering the complete graph whose vertices are the sets and whose edges are weighted by the cardinalities of the symmetric differences of their endpoints, and building a spanning forest of rooted trees that minimizes the sum of the cardinalities of the sets at the roots (which we store explicitly) and the total weights of the edges in the forest.

Bookstein and Klein's idea was reinvented several times [2, 12, 13, 18], but it seems Björklund and Lingas [13] were the first to observe that we can multiply matrices in time bounded in terms of the weights of their forests (the cardinalities of the sets at the roots and the total weights of the edges). To see why, consider that if we have already multiplied a row $\boldsymbol{u}$ of binary matrix $\mathbf{A}$ by a column $\boldsymbol{v}$ of binary matrix $\mathbf{B}$ and we know the symmetric difference of $\boldsymbol{u}$ and another row $\boldsymbol{u}'$ of $\mathbf{A}$, viewed as sets, then we can compute the product of $\boldsymbol{u}'$ and $\boldsymbol{v}$ by multiplying the elements in that symmetric difference by the corresponding elements in $\boldsymbol{v}^\top$ and adjusting the product $\boldsymbol{u} \cdot \boldsymbol{v}^\top$ appropriately. This observation easily generalizes also to non-binary matrices.

In the case of adjacency matrices of webgraphs with the rows sorted by URL, neighbouring rows are more likely to be similar. Boldi and Vigna [16, 15, 14] took advantage of this tendency and considered only edges between rows close in that ordering, in order to speed up the construction of spanning forests that usually still have low weight in practice for webgraphs. They limited the possibilities of compression in order to provide a reasonable extraction time for the adjacency lists (i.e., the nonempty cells in a row). Grabowski and Bieniecki [26] optimized their heuristic and Francisco et al. [25] observed that the compression could be used to speed up matrix-vector multiplication. They were all apparently aware of some earlier work but not Bookstein and Klein's nor Björklund and Lingas's papers in particular.

In the case of coloured de Bruijn graphs (CDBGs) for bioinformatics, the sets of colours of neighbouring vertices are more likely to be similar. Almodaresi et al. [5] took advantage of this tendency when compressing CDBGs for use in their tool Mantis [33] by considering only edges between sets of colours associated with neighbouring vertices, and encoding the resulting color set on the minimum spanning tree of the induced graph.

Alanko et al. [3] recently proposed compressing a collection of sets by finding, for each set S in the collection, the smallest superset S' of it in the collection and making S' the parent of S in a tree. They path-compress this tree to make its height logarithmic in the size of the universe. They encode S as a bitvector indicating which elements of S' are in S, which allows them to support queries such as predecessor and successor on the sets in logarithmic time. They tested this idea on CDBGs in the tool Themisto [4] with 16 thousand bacterial genomes and found it improved compression compared to Themisto's default (0.18 bits per element instead of 0.32 bits), although they did not report query times.

6 Conclusion and Further Work

We have introduced a measure $\Delta(\mathcal{S})$ of the space needed to optimally represent a set $\mathcal{S}$ of sets by indicating which elements from each set differ from those of some other set. Such representation has been used already in various applications, which demonstrates its practical interest. Our contributions are (1) formalizing the measure; (2) giving an improved algorithm to compute $\Delta(\mathcal{S})$ based on Prim's MST construction with edge costs computed incrementally; (3) designing an $O(\Delta(\mathcal{S}))$-space representation, based on tree extraction, that supports the most fundamental operations on the sets of $\mathcal{S}$ (membership, access, rank, predecessor, successor) in time at most $O(\log u)$, where u is the universe size.

Our adaptation of Prim (2) is of independent interest: it assumes that (integer) edge costs c can be computed incrementally: at any moment we know the cost is at least c and, in $O(1)$ time, we can determine whether it is c or greater. If the maximum cost of an MST edge is ℓ, then the total cost of our Prim's variant is the sum, for every edge of cost c, of $\min(c, \ell)$.

Another result of independent interest, in (3), is an $O(\log u)$-time extension of the known algorithm to find the ith smallest label among the ancestors of a labeled tree node. In the extension, some nodes can be marked as "deleting" a label that occurs upwards. We leave for future work to achieve $O(\log_\omega u)$ time for this extension, which is the cost of the basic version with no deletion marks. This would make the times of the five basic queries within $O(\Delta(\mathcal{S}))$ space asymptotically the same as those we obtain within the larger space $O(\mathcal{I}(\mathcal{S}))$.

We also leave for future work to obtain succinct space, $\Delta(\mathcal{S}) \lg u + o(\Delta(\mathcal{S}) \lg u)$ bits, not just $O(\Delta(\mathcal{S}))$ words. The data structures of tree extraction [28, 29] we build on are indeed succinct, but we use several copies/versions of them. It is easy to collapse all those into one if we aim at $O(\log u)$ for all query times, but achieving succinctness without sacrificing query times is more challenging.

We can also use more refined notions of compressibility. For example, as it is evident from Figure 2, one could reduce the number of nodes in the insertion tree by factoring paths; say, the paths for S_6 and S_5/S_8 could start with a shared part with the elements 3 and 5. This corresponds to adding to $\mathcal{S}$ an extra subset $\{3, 5\}$. It is possible to define a more refined form of $\mathcal{I}(\mathcal{S})$ (and of $\Delta(\mathcal{S})$) where one can optimally add subsets. While all our machinery to query the insertion-tree-based representation would work verbatim, it is not clear that the optimal sets to add can be found in polynomial time; approximations can be of interest.

On the other hand, it is possible to introduce more operations apart from insertions and deletions of elements, for example set complements, or taking the union or intersection of two sets. It is not clear, in this case, if one could efficiently support the query operations on those representations, apart from the possible hardness of building them optimally.

Acknowledgements

Funded in part by Basal Funds FB0001 and AFB240001, ANID, Chile. TG also supported by NSERC Discovery Grant RGPIN-07185-2020. MH also supported by NSERC Discovery Grant RGPIN-05953-2024. GN also supported by Fondecyt Grant 1-230755, ANID, Chile. Thanks to Rob Patro for helpful discussions.

References

1. Abboud, A., Backurs, A., Bringmann, K., Künnemann, M.: Impossibility results for grammar-compressed linear algebra. Advances in Neural Information Processing Systems **33**, 8810–8823 (2020)
2. Adler, M., Mitzenmacher, M.: Towards compressing web graphs. In: Proc. Data Compression Conference (DCC). pp. 203–212. IEEE (2001)
3. Alanko, J., Bille, P., Gortz, I., Navarro, G., Puglisi, S.: Compact data structures for collections of sets. In: From Strings to Graphs, and Back Again: A Festschrift for Roberto Grossi's 60th Birthday. OASIcs, vol. 132. Schloss Dagstuhl - Leibniz Center for Informatics (2025), article 6
4. Alanko, J.N., Vuohtoniemi, J., Mäklin, T., Puglisi, S.J.: Themisto: A scalable colored k-mer index for sensitive pseudoalignment against hundreds of thousands of bacterial genomes. Bioinformatics **39**(Supplement_1), i260–i269 (2023)
5. Almodaresi, F., Pandey, P., Ferdman, M., Johnson, R., Patro, R.: An efficient, scalable, and exact representation of high-dimensional color information enabled using de Bruijn graph search. Journal of Computational Biology **27**(4), 485–499 (2020)
6. Alves, J.N.F., Moustafa, S., Benkner, S., Francisco, A.P., Gansterer, W.N., Russo, L.M.S.: Accelerating graph neural networks using a novel computation-friendly matrix compression format. In: Proc. IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 1091–1103 (2025)
7. Apostolico, A.: The myriad virtues of subword trees. In: Combinatorial Algorithms on Words. pp. 85–96. NATO ISI Series, Springer-Verlag (1985)

8. Arroyuelo, D., Gómez-Brandón, A., Navarro, G.: Evaluating regular path queries on compressed adjacency matrices. The VLDB Journal **34**(1), article 2 (2025)
9. Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval. Addison-Wesley, 2nd edn. (2011)
10. Baunsgaard, S., Boehm, M.: Aware: workload-aware, redundancy-exploiting linear algebra. Proceedings of the ACM on Management of Data **1**(1), 1–28 (2023)
11. Bender, M., Farach-Colton, M.: The level ancestor problem simplified. Theoretical Computer Science **321**(1), 5–12 (2004)
12. Bharat, K., Broder, A., Henzinger, M., Kumar, P., Venkatasubramanian, S.: The connectivity server: Fast access to linkage information on the web. Computer networks and ISDN Systems **30**(1-7), 469–477 (1998)
13. Björklund, A., Lingas, A.: Fast boolean matrix multiplication for highly clustered data. In: Proc. 7th International Workshop on Algorithms and Data Structures (WADS). pp. 258–263 (2001)
14. Boldi, P., Rosa, M., Santini, M., Vigna, S.: Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In: Proc. 20th International Conference on World Wide Web (WWW). pp. 587–596 (2011)
15. Boldi, P., Santini, M., Vigna, S.: Permuting Web and social graphs. Internet Mathematics **6**(3), 257–283 (2009)
16. Boldi, P., Vigna, S.: The WebGraph framework I: Compression techniques. In: Proc. 13th International Conference on World Wide Web (WWW). pp. 595–602 (2004)
17. Bookstein, A., Klein, S.T.: Compression of correlated bit-vectors. Information Systems **16**(4), 387–400 (1991)
18. Buchsbaum, A.L., Caldwell, D.F., Church, K.W., Fowler, G.S., Muthukrishnan, S.: Engineering the compression of massive tables: an experimental approach. In: Proc. 11th Annual ACM Symposium on Discrete Algorithms (SODA). pp. 175–184 (2000)
19. Crochemore, M., Rytter, W.: Jewels of Stringology. World Scientific (2002)
20. Donenfeld, D., Chou, S., Amarasinghe, S.: Unified compilation for lossless compression and sparse computing. In: Proc. IEEE/ACM International Symposium on Code Generation and Optimization (CGO). pp. 205–216 (2022)
21. Elgohary, A., Boehm, M., Haas, P.J., Reiss, F.R., Reinwald, B.: Compressed linear algebra for large-scale machine learning. The VLDB Journal **27**(5), 719–744 (2018)
22. Elgohary, A., Boehm, M., Haas, P.J., Reiss, F.R., Reinwald, B.: Compressed linear algebra for declarative large-scale machine learning. Communications of the ACM **62**(5), 83–91 (2019)
23. Farach-Colton, M., Ferragina, P., Muthukrishnan, S.: On the sorting-complexity of suffix tree construction. Journal of the ACM **47**(6), 987–1011 (2000)
24. Ferragina, P., Manzini, G., Gagie, T., Köppl, D., Navarro, G., Striani, M., Tosoni, F.: Improving matrix-vector multiplication via lossless grammar-compressed matrices. Proceedings of the VLDB Endowment **15**(10), 2175–2187 (2022)
25. Francisco, A.P., Gagie, T., Köppl, D., Ladra, S., Navarro, G.: Graph compression for adjacency-matrix multiplication. SN Computer Science **3**, article 193 (2022)
26. Grabowski, S., Bieniecki, W.: Tight and simple web graph compression for forward and reverse neighbor queries. Discrete Applied Mathematics **163**, 298–306 (2014)
27. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA). pp. 841–850 (2003)
28. He, M., Munro, J.I., Zhou, G.: A framework for succinct labeled ordinal trees over large alphabets. Algorithmica **70**(4), 696–717 (2014)

29. He, M., Munro, J.I., Zhou, G.: Data structures for path queries. ACM Transactions on Algorithms **12**(4), 53:1–53:32 (2016)
30. Marinò, G.C., Furia, F., Malchiodi, D., Frasca, M.: Efficient and compact representations of deep neural networks via entropy coding. IEEE Access **11**, 106103–106125 (2023)
31. Navarro, G.: Wavelet trees for all. Journal of Discrete Algorithms **25**, 2–20 (2014)
32. Navarro, G.: Compact Data Structures – A practical approach. Cambridge University Press (2016)
33. Pandey, P., Almodaresi, F., Bender, M.A., Ferdman, M., Johnson, R., Patro, R.: Mantis: A fast, small, and exact large-scale sequence-search index. Cell Systems **7**(2), 201–207 (2018)
34. Salton, G., Lesk, M.E.: Computer evaluation of indexing and text processing. Journal of the ACM **15**(1), 8–36 (1968)
35. Tosoni, F., Bille, P., Brunacci, V., De Angelis, A., Ferragina, P., Manzini, G.: Toward greener matrix operations by lossless compressed formats. IEEE Access (2025)