Compact Data Structures for Collections of Sets

Jarno N. Alanko
Department of Computer Science, University of Helsinki, Finland

Philip Bille
Technical University of Denmark, Lyngby, Denmark

Inge Li Gørtz
Technical University of Denmark, Lyngby, Denmark

Gonzalo Navarro
Department of Computer Science, University of Chile, Santiago, Chile
Center for Biotechnology and Bioengineering (CeBiB), Santiago, Chile

Simon J. Puglisi
Department of Computer Science, University of Helsinki, Finland

Abstract

We define a new entropy measure $L(\mathcal{S})$, called the *containment entropy*, for a set $\mathcal{S}$ of sets, which considers the fact that some sets can be contained in others. We show how to represent $\mathcal{S}$ within space close to $L(\mathcal{S})$ so that any element of any set can be retrieved in logarithmic time. We extend the result to predecessor and successor queries and show how some common set operations can be implemented efficiently.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases Compressed data structures, entropy of sets, data compression

Digital Object Identifier 10.4230/OASIcs.Grossi.2025.6

Category Research

Funding *Jarno N. Alanko*: Funded by the Helsinki Institute for Information Technology (HIIT).
Philip Bille: Danish Research Council grant DFF-8021-002498.
Inge Li Gørtz: Danish Research Council grant DFF-8021-002498.
Gonzalo Navarro: Basal Funds FB0001 and AFB240001, ANID, Chile.
Simon J. Puglisi: Academy of Finland grant 339070.

Acknowledgements This work was initiated at the NII Shonan Meeting no. 187 on "Theoretical Foundations of Nonvolatile Memory."

1 Introduction

We consider the problem of representing a collection of sets $\mathcal{S} = \{S_1, \ldots, S_s\}$ from a universe $\mathcal{U}$ of size u while supporting basic queries, including retrieving the kth element and predecessor and successor queries. The goal is to store the sets compactly while supporting fast queries. This problem has important applications, including the representation of postings lists in inverted indexes and adjacency-list representations of graphs.

To measure space, we often consider the *worst-case entropy* defined as $H(\mathcal{S}) = \sum_{i=1}^{s} \lg \binom{u}{|S_i|}$ as a natural information-theoretic worst-case lower bound on the number of bits needed to store $\mathcal{S}$. Using standard techniques [6, 5, 9], we can store $\mathcal{S}$ within $H(\mathcal{S}) + O(n + s \log n)$ bits and support retrieval (i.e., accessing any kth element of any set) in constant time.

In this paper, we propose a finer measure of entropy for $\mathcal{S}$ that can take advantage of the fact that some sets may be subsets of others. If $S_i \subset S_j$, we can encode S_i using $\log \binom{|S_j|}{|S_i|}$ bits, by indicating which elements in S_j are also in S_i. We show how to construct a hierarchy from $\mathcal{S}$ such that children are subsets of their parent. This leads to a new notion of entropy called the *containment entropy*, defined as

$$L(\mathcal{S}) = \sum_{i=1}^{s} \lg \binom{|p(S_i)|}{|S_i|},$$

where $p(S_i)$ is the parent of S_i (see Section 3). It is easy to see that the containment entropy is a finer notion of entropy since $L(\mathcal{S}) \leq H(\mathcal{S})$. Our main result is that we efficiently represent $\mathcal{S}$ in space close to its containment entropy while supporting queries efficiently.

▶ **Theorem 1.** *Let $\mathcal{S}$ be a set of s sets of total size n, elements of which are drawn from a universe of size u. We can construct a data structure in $O(sn \log n)$ time that uses $L(\mathcal{S}) + O(n + s \log n)$ bits of space and supports retrieval, predecessor, and successor queries on any set $S \in \mathcal{S}$ in time $O(\log(u/|S|))$.*

Thus, compared to the above-mentioned standard data structures, we replace the worst-case entropy $H(\mathcal{S})$ with $L(\mathcal{S})$ in our space bound in Theorem 1.

We also obtain several corollaries of Theorem 1. We show how to implement the common operations of set intersection, set union, and set difference. By combining Theorem 1 with techniques from Barbay and Kenyon [2] we obtain fast implementations of these operations in terms of the *alternation bound* [2] directly on our representation. We also show how to apply Theorem 1 to efficiently store a collection of bitvectors while supporting access, rank, and select queries.

Technically, the result is obtained by constructing a tree structure, where each node represents a set, and children represent subsets of their parent. We then represent each set as a sparse bitvector that indicates which of the elements of the parent are present in the child. This leads to a simple representation that achieves the desired space bound. Unfortunately, if we directly implement a retrieval query on this representation, we have to traverse the path from the set S to the root leading to a query time of $\Omega(h)$, where h is the height of the tree. We show how to efficiently *shortcut* paths in the tree without asymptotically increasing the space, yielding the $O(\log(u/|S|))$ time. We then extend these techniques to handle predecessor and successor queries in the same time, leading to the full result of Theorem 1.

Relation with wavelet trees. We note that the idea is reminiscent of wavelet trees [7], which represent a sequence of symbols as a binary tree where each node stores a bitvector. In the root's bitvector, 0s/1s mark the sequence positions with symbols in the first/second half of the alphabet, and left/right children recursively represent the subsequences formed by the symbols in the first/second half of the alphabet. Nodes handling single-symbol subsequences are wavelet tree leaves. If we define $\mathcal{S}$ as the sets of positions in the sequence holding the symbols of each subalphabet considered in the wavelet tree, then our data structure built on $\mathcal{S}$ has the same shape of the wavelet tree. The wavelet tree uses less space, however, because it corresponds to a particular case where the maximal subsets of each set partition it into two: while the wavelet tree can be compressed to the zero-order entropy of the sequence [7], $L(\mathcal{S})$ doubles that entropy.

2 Basic Concepts

A *bitvector* $B[1..u]$ is a data structure that represents a sequence of u bits and supports the following operations:

- $access(B, i)$ returns $B[i]$, the ith bit of B.
- $rank_b(B, i)$, where $b \in \{0, 1\}$, returns the number of times bit b occurs in the prefix $B[1..i]$ of the bitvector. We assume $rank_b(B, 0) = 0$.
- $select_b(B, j)$ returns the position in B of the jth occurrence of bit $b \in \{0, 1\}$. We assume $select_b(B, 0) = 0$ and $select_b(B, i) = n + 1$ if b does not occur i times in B.

Note that $rank_0(B, i) = i - rank_1(B, i)$. By default we assume $b = 1$. It is possible to represent bitvector B using $u + o(u)$ bits and solve all the operations in $O(1)$ time [3, 8]. If n is the number of 1s in B, it is possible [6, 5, 9] to represent B using

$$n \lg \frac{u}{n} + 2n \quad = \quad \lg \binom{u}{n} + O(n + \log u)$$

bits,[1] so that $select_1$ is implemented in constant time, while $access$ and $rank_b$ are solved in time $O(1 + \log \frac{u}{n})$. Operation $select_0$ can be solved in time $O(\log n)$ by binary search on $select_1$. This is called the *sparse bitvector representation* in this paper.

We note that $\lg \binom{u}{n}$ is the entropy of the set of positions where B contains n 1s, or equivalently, the entropy of the sets of n elements in a universe of size u. We will indistinctly speak of the operations $access$, $rank$, and $select$ over sets S on the universe $[1..u]$, referring to the corresponding bitvector B where $B[i] = 1$ iff $i \in S$. For example, $select(S, k)$ is the kth smallest element of S.

3 Containment Entropy

Let $\mathcal{S} = \{S_1, S_2, \ldots, S_s\}$ be a set of s distinct sets, and $\mathcal{U} = S_1 \cup S_2 \cup \cdots \cup S_s$ be their union. For simplicity we assume $\mathcal{U} = [1..u]$, so $u = |\mathcal{U}|$. Let $n_i = |S_i|$ and $n = \sum_{i=1}^{s} n_i$ be the total size of all sets; note $n \geq u$.

A simple notion of entropy for $\mathcal{S}$ is its *worst-case entropy*

$$H(\mathcal{S}) \quad = \quad \sum_{i=1}^{s} \lg \binom{u}{n_i}.$$

It is not hard to store $\mathcal{S}$ within $H(\mathcal{S}) + O(n + s \log n)$ bits, by using the sparse bitvector representation of Section 2, which stores each S_i within $\lg \binom{u}{n_i} + O(n_i + \log u)$ bits, and offers constant-time retrieval of any element of S_i via $select(S_i, k)$. We use $O(s \log n)$ additional bits to address the representations of the s sets.

We now propose a finer measure of entropy for $\mathcal{S}$, which exploits the fact that some sets can be subsets of others. If $S_i \subset S_j$, we can describe S_i using $\lg \binom{n_j}{n_i}$ bits, which indicate which elements of S_j are also in S_i. We define a tree structure whose nodes are the sets S_i, and the parent of S_i, $p(S_i)$, is any *smallest* S_j such that $S_i \subset S_j$. If S_i is not contained in any other set, then its parent is the tree root, which represents $\mathcal{U}$. We then define the following measure of entropy, which we call the *containment entropy*.

[1] The formula on the left is zero if $n = 0$.

Algorithm 1 Retrieving the kth element of a set S in a hierarchy.

Input : Set S and rank k of the element to be retrieved, with $1 \leq k \leq |S|$.
Output : The kth smallest element in S.

```
1 function retrieve(S, k)
2     if S is the root then return k
3     else return retrieve(p(S), select(S, k)) // use c(S) on contracted hierarchy
```

▶ **Definition 2.** *Let $\mathcal{S}$ be a set of sets S_i. Its* hierarchy *has $\mathcal{U} = \cup_i S_i$ at the root, and the parent $p(S_i)$ of S_i is any smallest set containing S_i (or $\mathcal{U}$ if no such set exists). Let $p_i = |p(S_i)|$. The* containment entropy *of $\mathcal{S}$ is*

$$L(\mathcal{S}) \quad = \quad \sum_{i=1}^{s} \lg \binom{p_i}{n_i}.$$

Clearly $L(\mathcal{S}) \leq H(\mathcal{S})$ because $p_i \leq u$ for every i. Note that $L(\mathcal{S})$ is arguably the optimal space we can achieve by representing sets as subsets of others in $\mathcal{S}$, but we could obtain less space if other arrangements were possible. For example, if many sets are subsets of both S_i and S_j, it could be convenient to introduce a new set $S_i \cap S_j$ in the hierarchy, even if it is not in $\mathcal{S}$. An advantage of $L(\mathcal{S})$ is that it is easily computed in polynomial time from $\mathcal{S}$, whereas more general notions like the one that allows the creation of new sets may be not.

4 A Containment Entropy Bounded Representation with Fast Retrieval

It is not hard to represent $\mathcal{S}$ within space close to $L(\mathcal{S})$: we can use the sparse bitvector representation of Section 2 to store each S_i relative to its parent $p(S_i)$, within $\lg \binom{p_i}{n_i} + O(n_i + \log p_i)$ bits, which add up to $L(\mathcal{S}) + O(n + s \log u)$ bits. The problem is that now $select(S_i, k)$ gives the position of the kth element of S_i within those of $p(S_i)$, not within $\mathcal{U}$, and thus $select(S_i, k)$ is not directly the identity of the kth element of S_i. In order to obtain the identity of the element, we must compute $select(p(S_i), select(S_i, k))$ and so on, consecutively following the chain of ancestors of S_i until the root $\mathcal{U}$; see Algorithm 1. This may take time proportional to the height of the hierarchy, which can be up to $O(s)$. We now show how to reduce this time to logarithmic, by introducing *shortcuts* in the hierarchy.

▶ **Definition 3.** *The* contracted hierarchy *of $\mathcal{S}$ has $\mathcal{U}$ as its root, and the parent $c(S_i)$ of S_i is the highest ancestor S_t of $p(S_i)$ in the hierarchy of $\mathcal{S}$ such that $n_t \leq 2n_i$. If no such ancestor exists, then $c(S_i) = p(S_i)$.*

We now prove a couple of relevant properties of this contracted hierarchy.

▶ **Lemma 4.** *The depth of node S_i in the contracted hierarchy of $\mathcal{S}$ is $O(\log(u/n_i))$. The height of the contracted hierarchy is $O(\log u)$.*

Proof. By definition, the grandparent of node S_i, if it exists, has size $> 2n_i$; thus the path towards the root (whose size is u) cannot be longer than $2 \lg(u/n_i)$. An obvious consequence is that the height of the tree cannot be longer than $2 \lg u$. ◀

If we change $p(S)$ to $c(S)$ in Algorithm 1, then, the retrieval time becomes $O(\log(u/|S|))$. We now show that the space is not asymptotically affected by contracting the hierarchy.

▶ **Lemma 5.** *The contracted hierarchy of $\mathcal{S}$ can be represented within $L(\mathcal{S}) + O(n + s \log n)$ bits.*

Algorithm 2 Finding the position of the predecessor of k in the set S of a hierarchy.

Input : Set S and element k of the universe, $1 \leq k \leq u$.
Output : The position of the predecessor of k in S.

```
1 function predecessor(S, k)
2     if S is the root then return k
3     else return rank(S, predecessor(c(S), k))
```

Proof. We start from our representation that uses $L(\mathcal{S})+O(n+s\log n)$ bits, and show how it changes when we replace $p(S_i)$ by $c(S_i)$. In case $p(S_i) \neq c(S_i)$, it holds that $|c(S_i)| < 2|p(S_i)|$, because $S_i \subset p(S_i)$ and thus $|c(S_i)| \leq 2|S_i| < 2|p(S_i)|$. Then, changing $p(S_i)$ to $c(S_i)$ increases the size of the representation from $n_i \lg \frac{p_i}{n_i} + 2n_i$ to at most $n_i \lg \frac{2p_i}{n_i} + 2n_i = n_i \lg \frac{p_i}{n_i} + 3n_i$. Thus, the total increase produced by changing all $p(S_i)$ to $c(S_i)$ is bounded by n; we still use $L(\mathcal{S}) + O(n + s\log n)$ bits. The parent pointers that allow climbing paths also fit in $O(s\log n)$ bits. ◀

In summary, we have the following result.

▶ **Lemma 6.** *A set $\mathcal{S}$ of s sets of total size n and universe size u can be represented within $L(\mathcal{S}) + O(n + s\log n)$ bits so that any element of any set S can be retrieved in time $O(\log(u/|S|))$.*

5 Other Operations

5.1 Predecessor and Successor in a Set

Given an element identifier k and a set S_i, the *predecessor* is the largest $v \leq k$ such that $v \in S_i$. Conversely, the *successor* is the smallest $v \geq k$ such that $v \in S_i$. We can find the predecessor in $O(\log u)$ time, as follows. We start at the node S_i and walk up the path to the root $\mathcal{U}$, so as to determine the nodes in the path. In the return from the recursion, we start at the position $v \leftarrow k$ in $\mathcal{U}$, and whenever returning from a node S to its child S', we compute $v \leftarrow rank(S', v)$, which gives the number of elements from S' up to the element v in S, that is, the predecessor of v among the elements of S'. By the time we return to S_i again, v is the position in S_i of the predecessor of k; see Algorithm 2. We then find out the identity of v in $\mathcal{U}$ using Algorithm 1. To find the successor, we use $v \leftarrow 1 + rank(S', v-1)$ instead.

Operation *rank* takes time $O(1 + \log(|S|/|S'|))$ in the sparse bitvector representation of S', as seen in Section 2. Therefore, the sum of the times along the path to S_i telescopes to $O(\log(u/n_i))$. This yields the following result.

▶ **Lemma 7.** *On the same representation of Lemma 6, we can compute the predecessor and successor of any element of any set S in time $O(\log(u/|S|))$.*

5.2 Set Operations

These operations are useful to compute set operations, by mimicking any standard algorithm that traverses both ordered sets. In particular, we can implement an intersection algorithm that is close to the alternation complexity lower bound δ [2]: any algorithm that can find the successor of any element in either list in time t, can intersect both lists in time $O(\delta t)$. Predecessor and successor on the complement of a set can also be solved in time $O(t)$, by using $rank_0$ instead of *rank* (and managing the base case accordingly).

▶ **Corollary 8.** *On the representation of Lemma 6, we can compute the intersection between any two sets S_i and S_j in time $O(\delta \log u)$, where δ is the alternation complexity of both sets. We can also compute their union in time $O(|S_i \cup S_j| \log u)$. The difference $S_i \setminus S_j$ can be implemented in time $O(\delta' \log u)$, where δ' is the alternation complexity of S_i and S_j^c.*

5.3 Back to Bitvectors

Returning to the bitvector semantics, our results allow us store a set of sparse bitvectors $B_i[1..u]$, representing subsets S_i of a universe $\mathcal{U}$ of size u, so that operations $access(B_i, k)$, $rank(B_i, k)$ and $select(B_i, k)$ take time $O(\log(u/n_i))$. To implement $access(B_i, k)$, we return 1 if the predecessor of k in S_i is k, or else 0. To implement $rank(B_i, k)$, we compute the position v of the predecessor of k in the ordered set S_i and return $rank(S_i, v)$. To implement $select_1(B_i, k)$, we retrieve the kth element of S_i, and for $select_0$ we do binary search on $select_1$.

▶ **Corollary 9.** *A set of s bitvectors $B_i[1..u]$ can be represented in $L(\mathcal{S}) + O(n + s \log u)$ bits, where $\mathcal{S}$ is the set of sets $S_i = \{k,\ B_i[k] = 1\}$, n_i is the number of 1s in B_i and $n = \sum_i n_i$. This representation supports operations access, $rank_b$, and $select_1$ on any B_i in time $O(\log(u/n_i))$, and $select_0$ in time $O(\log n_i \log(u/n_i))$.*

This is to be compared with storing each bitvector directly [9], which gives total space $H(\mathcal{S}) + O(n + s \log u)$ and supports *access* and $rank_b$ in the same time $O(\log(u/n_i))$, and $select_b$ in the better times $O(1)$ for $b = 1$ and $O(\log n_i)$ for $b = 0$.

6 Construction

We can build the hierarchical representation by adding one set S at a time, *in decreasing order of size*, to ensure it is correct to insert S as a leaf. To insert S in $\mathcal{S}$, we first find a smallest set $S' \in \mathcal{S}$ such that $S \subset S'$. The sets that contain S form a connected subgraph of the hierarchy that includes the root. So we can traverse the hierarchy from the root, looking for the lowest nodes that contain S, and retain the smallest of those. For each hierarchy node S_i to check, we take each element of S and verify that it exists in S_i via a predecessor query, which takes time $O(\log(u/|S_i|)) \subseteq O(\log(u/|S|))$, because $|S| \leq |S_i|$. We then find a smallest set S' containing S in time $O(s'|S| \log(u/|S|))$, where $s' \leq n/|S|$ is the current number of sets in $\mathcal{S}$ and n is its final sum of set sizes.

We then insert S in the hierarchy by setting $p(S) = S'$. To find $c(S)$, we find the highest ancestor S'' of S' whose size is $|S''| \leq 2|S|$ (or let $S'' = S'$ if no such ancestor exists), and set $c(S) = S''$. Finally, we build the representation of S relative to S'', in time $O(|S|)$ [9].

Note that, when we find the ancestor S'', we traverse the upward path defined by the parent function $p(\cdot)$, not $c(\cdot)$. We still use $c(\cdot)$ during construction to answer the predecessor queries, to determine inclusion of S, in logarithmic time.

▶ **Lemma 10.** *The representation of Lemma 6 can be built in time $O(sn \log n)$.*

Combining Lemmas 6, 7, and 10 we have shown Theorem 1.

7 Concluding Remarks

We have described the containment entropy, a new entropy measure for collections of sets, which is sensitive to subset relationships between sets. To our knowledge, this idea has not before been considered in the vast literature on efficient set representations that has emerged

primarily from efficiency concerns in information retrieval systems [11]. One could consider dictionary-based compression of sets (see, e.g., [10, 4]) as implicitly capturing some aspect of subset relationships, but the representations and analysis we have described here consider subset relationships explicitly.

An interesting direction for future work is to explore the practicality of these hierarchical set representations in various application contexts. An immediate concern is that of hierarchy construction. Our initial experiments with a collection of sets taken from the genomic search engine Themisto [1] applied to a set of 16 thousand bacterial genomes (over 80GB of data) indicate that, even for large set collections, hierarchy construction is tractable and scales almost linearly with the total length of the sets in practice. On a collection of 10.55 million sets of average size 3,607 (10.52 million of which were subsets of at least one other set) of more than 38 billion elements in total, we were able to find the smallest super set of every set in less than 40 hours in total, using just 16 threads of execution. The resulting representation used just 0.18 bits per element, compared to the 0.32 bits per element used by Themisto's representation, which selects between different set representations based on set density.

References

1 Jarno N. Alanko, Jaakko Vuohtoniemi, Tommi Mäklin, and Simon J. Puglisi. Themisto: a scalable colored k-mer index for sensitive pseudoalignment against hundreds of thousands of bacterial genomes. *Bioinformatics*, 39(Supplement-1):260–269, 2023. doi:10.1093/BIOINFORMATICS/BTAD233.

2 J. Barbay and C. Kenyon. Alternation and redundancy analysis of the intersection problem. *ACM Transactions on Algorithms*, 4(1):1–18, 2008. doi:10.1145/1328911.1328915.

3 D. R. Clark. *Compact PAT Trees*. PhD thesis, University of Waterloo, Canada, 1996.

4 F. Claude and G. Navarro. Fast and compact web graph representations. *ACM Transactions on the Web*, 4(4):article 16, 2010.

5 P. Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM*, 21:246–260, 1974. doi:10.1145/321812.321820.

6 R. Fano. On the number of bits required to implement an associative memory. Memo 61, Computer Structures Group, Project MAC, Massachusetts, 1971.

7 R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.

8 J. I. Munro. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 37–42, 1996.

9 D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 60–70, 2007.

10 Giulio Ermanno Pibiri, Matthias Petri, and Alistair Moffat. Fast dictionary-based compression for inverted indexes. In *Proc. 12th ACM International Conference on Web Search and Data Mining (WSDM)*, pages 6–14. ACM, 2019. doi:10.1145/3289600.3290962.

11 Giulio Ermanno Pibiri and Rossano Venturini. Techniques for inverted index compression. *ACM Computing Surveys*, 53(6):125:1–125:36, 2021. doi:10.1145/3415148.