Danmarks Tekniske Universitet

Bachelor Project 2026

Implementation of Compact Data Structures for Collections of Sets

Sebastian Gloerfelt-Tarp
s235236

Contents

Abstract

Storing a collection of sets compactly, while still supporting the basic queries access, rank and select, is a recurring problem in computer science. The naive approach stores each set on its own, independently, which costs the worst-case entropy $H(\mathcal{S})$ and ignores any subset relationships between the sets. The paper "Compact Data Structures for Collections of Sets" proposes containment entropy $L(\mathcal{S}) \leq H(\mathcal{S})$, which encodes each set relative to its smallest superset rather than relative to the whole universe. This thesis implements two variants in C. The first is a Baseline compressed ranked bitvector that stores each set independently, and the second is a contracted containment hierarchy in which every level bitvector is stored with that same Baseline compression. The two are then compared on two contrasting synthetic collections, one densely nested and one of independent random sets. On the nested data the hierarchy uses about 47% less space than the Baseline, while on the independent data the two stay within one percent of each other. That space saving comes at a cost in speed, as the hierarchy is roughly 1.3 to 1.7 times slower per query, so it trades query time for space on data that is nested.

1 Introduction

Collections of sets appear throughout computer science, and storing them compactly is a practical problem that is seen quite often. This can be seen in, for example, the adjacency-list representation of graphs [2], in large collections of sparse bitvectors and perhaps in genomic search indexes. As the number of sets and the size of the universe grow, the space needed to store the collection becomes the limiting factor, and the way in which the sets are encoded starts to matter quite a bit. This thesis studies one of these types of encoding. The aim is to implement and evaluate the containment entropy data structure introduced in "Compact Data Structures for Collections of Sets" [1], and to compare it against a Baseline solution built from the techniques described in "Compact Data Structures" [5].

1.1 Motivation

The simplest way to store a collection of sets is to store each set on its own. A single set of n_i elements from a universe of size u can be represented in $\lg \binom{u}{n_i}$ bits, which is its information-theoretic lower bound, and storing every set independently in this way costs in the worst case [1]

$$H(\mathcal{S}) = \sum_i \lg \binom{u}{n_i}$$

bits in total. Many systems do exactly this, and for sets that have no relation to one another it is essentially the best that can be done, at least as far as the scope of this assignment. On the other hand, when the sets are related, there is room for improvement from the worst-case entropy. In many real collections, the sets overlap heavily and many similarities can be derived. The real value is when one set can be fully contained in another. Encoding each set independently then pays again and again for the elements that the sets have in common. This wasted space is precisely the shared structure that a more clever encoding can exploit, and it is the motivation for representing the collection by how its sets relate to each other instead of treating them as if they were unrelated.

1.2 Problem statement

The goal of this thesis is to implement and evaluate the containment-entropy data structure, which targets wasted space directly by storing each set relative to a parent rather than relative to the whole universe. When a set S_i is stored relative to a parent $p(S_i)$ that contains it, its cost falls from $\lg \binom{u}{|S_i|}$ to $\lg \binom{|p(S_i)|}{|S_i|}$, and summing over the collection gives the containment entropy [1]

$$L(\mathcal{S}) = \sum_i \lg \binom{|p(S_i)|}{|S_i|} \leq H(\mathcal{S}).$$

The inequality holds because every parent is at most as large as the universe, so the structure can never cost more than independent storage and should cost considerably less whenever containment in one another is actually present. This raises two claims that then need to be checked in practice rather than only on paper. The first is that the

predicted saving, $L(\mathcal{S}) \leq H(\mathcal{S})$, actually works on representative data. The second is that the contraction step used when building the hierarchy actually reduces its depth, since that depth is largely what decides how expensive a query operation is.

1.3 Research questions

These goals lead to three concrete research questions that this project will attempt to answer. The first, will the containment entropy $L(\mathcal{S})$ create a measurable improvement over the independent cost $H(\mathcal{S})$ on real data. The second asks whether the contracted hierarchy actually shortens retrieval in practice, in other words, whether creating a bound on the depth of the hierarchy translates into faster queries. Thirdly, how the implementation compares, in both space and query time, against a Baseline built from a single compressed bitvector, so that the cost of the containment representation can be compared fairly and accurately.

1.4 Contributions

To answer these questions, this thesis provides a compressed bitvector that serves as the Baseline against the containment structure, where time and space are evaluated and compared accordingly. It creates an implementation in C of the contracted containment hierarchy, in which each individual bitvector is stored with a compressed representation that supports rank and select queries in constant time [1, 5]. It creates a hierarchical structure based on an inverted list, which finds each set's parent without having to brute-force a pairwise subset check between every pair of sets. Finally, there is an empirical evaluation using synthetic datasets that are generated with relatively controlled depth and density, so that the behaviour of the structure can be evaluated across a range of conditions.

1.5 Thesis structure

The remainder of the thesis is organised as follows. Section 3 provides the necessary background, covering bitvectors and the rank and select operations defined on them, the entropy measures used to deduce the space, and the containment hierarchy itself. Section 4 describes the implementation of both the Baseline and the containment structure in detail. Sections 5 and 6 present the experimental setup and the results obtained from it. Sections 7 and 8 then discuss what those results mean and draws a conclusion.

2 Background

This section's purpose is to lay the foundation and structure for understanding the implementation of the algorithms and data structures. It will introduce the theory behind the approaches and the general idea behind them and how they are stored.

2.1 Bitvectors

A bitvector $B[1..u]$ is an array of u bits, where every position holds either a 0 or a 1. A bitvector can be used to represent a set of positive integers. For instance, the set $S = \{1, 3, 5, 7\}$ in a universe $\mathcal{U} = \{1, 2, \ldots, 8\}$ is represented by the bitvector

$$B = 10101010,$$

where $B[i] = 1$ when $i \in S$.

With the aforementioned bitvector, three relevant operations are standard [5]:

- $access(B, i)$ returns the bit $B[i]$.
- $rank(B, i)$ counts the number of 1s in $B[1..i]$.
- $select(B, k)$ returns the position of the k-th 1, reading from left to right.

A bitvector with n ones in a universe of u positions can always be stored explicitly in u bits, but this is quite wasteful when n is much smaller than u. This is because a lot of space would be used for 0s representing the absence of elements

in that given set. As an example, the set $S = \{1, 8\}$ in a universe of size 8 is represented by

$$B = 10000001,$$

where six of the eight bits store nothing but the absence of elements.

The theoretical minimum for storing a set represented by bits is $\lg \binom{u}{n}$ bits, the logarithm of the number of distinct n-element subsets of a universe of size u. Compressed representations come close to this bound and still support all three operations in constant time, using a small amount of overhead space on top [5].

2.2 Compressed bitvector representation

Storing a bitvector in its plain form is simple but wasteful, as the example above showed. A better option is to compress the bitvector so that it takes space close to the theoretical minimum of $\lg \binom{u}{n}$ bits, while still answering access, rank and select quickly [5]. The Baseline implementation in this thesis does exactly this, using a block based combinatorial encoding.

Splitting into blocks

The first step is to cut the bitvector into fixed size blocks of b bits each [5, 6]. Each block is then described by two numbers. The first is the class c, which is simply the number of 1s in the block. The second is the offset, which says which specific block this is among all blocks that share the same class. There are $\binom{b}{c}$ different ways to place c ones in a block of b bits, so the offset is just the index of the pattern in that list. Knowing the class and the offset together is enough to rebuild the exact block.

Finding the offset

To turn a block into its offset, list every pattern that has the same class in increasing numeric order and read off the position of the block, counting from 0. Take a block size of $b = 4$ and the block 1010. It has two 1s, so its class is $c = 2$. The $\binom{4}{2} = 6$ patterns with exactly two 1s, in increasing order, are

$$0011,\ 0101,\ 0110,\ 1001,\ 1010,\ 1100.$$

Our block 1010 is the fifth in this list, so its offset is 4 because it is 0-indexed. The pair (class 2, offset 4) fully identifies the block. In practice the list is not written out. A small precomputed table of binomial coefficients, which is Pascal's triangle, allows the encoder to add up the right values and compute the offset directly, and the same table lets the decoder turn a (class, offset) pair back into the original block.

Why this saves space

The class is a small number between 0 and b, and the offset needs only

$$\left\lceil \lg \binom{b}{c} \right\rceil$$

bits, because that is all it takes to tell apart the $\binom{b}{c}$ patterns of class c. Blocks that are all 0s or all 1s have just one possible pattern and so need no offset bits at all. For the block 1010 above, the six patterns need only $\lceil \lg 6 \rceil = 3$ offset bits instead of the 4 raw bits, and the gap grows for larger blocks. Summed over the whole bitvector, the offset storage approaches the lower bound of $\lg \binom{u}{n}$ bits [5].

The stored arrays

A few arrays hold the encoded data and make the queries fast [5]. The first two are enough to rebuild the bitvector, and the rest only exist to keep queries fast.

- $C[\,]$: one entry per block, holding that block's class.
- $O[\,]$: all the offsets packed back to back, where the offset of a block of class c takes $L[c]$ bits.

- $L[c]$: a small lookup giving the number of offset bits for class c, that is $\lceil \lg \binom{b}{c} \rceil$.
- $R[\,]$: rank checkpoints, the running number of 1s seen so far, sampled once every k blocks.
- $P[\,]$: position checkpoints, the bit position in $O[\,]$ where the next block's offset begins, sampled once every k blocks.
- $S[\,]$: the positions of all 1s in order, used to answer select directly.

The checkpoints in $R[\,]$ and $P[\,]$ let a query jump close to the block it needs instead of scanning from the start, which is what keeps rank and access constant-time. Here, k is the sampling rate, which is a fixed number in the implementation.

Worked example

Take the 24-bit vector

$$B = 1010\ 1100\ 0001\ 1111\ 0100\ 0110,$$

with block size $b = 4$ and one checkpoint every $k = 2$ blocks. Cutting it into six blocks and reading off each class and offset gives:

block	bits	class c	offset	offset bits ($L[c]$)
1	1010	2	4	100
2	1100	2	5	101
3	0001	1	0	00
4	1111	4	0	(none)
5	0100	1	2	10
6	0110	2	2	010

Here $L[2] = 3$ and $L[1] = 2$, while class 4 is all 1s and needs no offset bits. Collecting the columns gives the stored arrays:

$$C = [\,2,\ 2,\ 1,\ 4,\ 1,\ 2\,]$$
$$O = 100\ 101\ 00\ 10\ 010$$
$$S = [\,1,\ 3,\ 5,\ 6,\ 12,\ 13,\ 14,\ 15,\ 16,\ 18,\ 22,\ 23\,]$$

O is just the offset bit strings written one after another and S lists the positions of the twelve 1s in B. With $k = 2$ the checkpoints are taken at blocks 1, 3 and 5. Before block 1 no 1s have been seen and O is empty. Before block 3 there are $2 + 2 = 4$ earlier 1s and $3 + 3 = 6$ earlier offset bits, so the next offset begins at position 7. Before block 5 there are $2 + 2 + 1 + 4 = 9$ earlier 1s and $3 + 3 + 2 = 8$ earlier offset bits, so the next offset begins at position 9. This gives

$$R = [\,0,\ 4,\ 9\,]$$
$$P = [\,1,\ 7,\ 9\,],$$

where the positions in P are counted from 1. From these arrays the original bitvector can be rebuilt block by block, and rank, access and select are all answered in constant time.

Space analysis

It is worth checking that the two data arrays really do reach the bound of $\lg \binom{u}{n}$ bits. There are $\lceil u/b \rceil$ blocks in total, so the analysis runs over that many blocks.

Start with the class array C. Each entry is a number between 0 and b, so it fits in $\lceil \lg(b+1) \rceil$ bits, and the whole array takes

$$\lceil u/b \rceil \cdot \lceil \lg(b+1) \rceil = \tfrac{u}{b} \lg b + O\big(\tfrac{u}{b}\big) \text{ bits.}$$

A larger block size b means fewer blocks and so a smaller C, but it also makes each block slower to decode, so b is a trade-off.

The offset array O is also interesting. Block i has class c_i and so uses $\lceil \lg \binom{b}{c_i} \rceil$ bits, and the total number of bits in O is

$$\sum_{i=1}^{\lceil u/b \rceil} \left\lceil \lg \binom{b}{c_i} \right\rceil .$$

We bound this in three steps.

Each ceiling rounds up by less than one bit, and there are $\lceil u/b \rceil$ blocks, so dropping all of them adds at most $\lceil u/b \rceil$:

$$\sum_{i=1}^{\lceil u/b \rceil} \left\lceil \lg \binom{b}{c_i} \right\rceil < \sum_{i=1}^{\lceil u/b \rceil} \lg \binom{b}{c_i} + \lceil u/b \rceil.$$

Using $\lg a + \lg b = \lg(ab)$, the sum of logarithms becomes the logarithm of the product:

$$\sum_{i=1}^{\lceil u/b \rceil} \lg \binom{b}{c_i} = \lg \prod_{i=1}^{\lceil u/b \rceil} \binom{b}{c_i}.$$

The key fact is

$$\prod_{i=1}^{\lceil u/b \rceil} \binom{b}{c_i} \leq \binom{u}{n},$$

where $n = \sum_i c_i$ is the total number of 1s. Take these two blocks first:

$$\binom{b}{c}\binom{b}{c'} \leq \binom{2b}{c+c'}.$$

The right side counts all the ways to choose $c + c'$ ones among $2b$ positions. The left side counts only those ways that put exactly c ones in the first b positions and exactly c' ones in the second b positions. That is a subset of all the choices on the right, so the left side can only be smaller or equal. Applying this step block by block across the whole bitvector turns the product into $\binom{u}{n}$. Putting the three steps together,

$$\sum_{i=1}^{\lceil u/b \rceil} \left\lceil \lg \binom{b}{c_i} \right\rceil < \lg \binom{u}{n} + \lceil u/b \rceil,$$

so the offset array reaches the worst-case minimum $\lg \binom{u}{n}$ plus the lower-order term $\lceil u/b \rceil$.

In total, adding the two arrays together with the small lookup tables L and the Pascal table K, which take $O(wb^2)$ bits for word size w, the whole structure uses

$$\lg \binom{u}{n} + \tfrac{u}{b} \lg b + O\big(\tfrac{u}{b} + wb^2\big) \text{ bits.}$$

This is the optimal worst-case space $\lg \binom{u}{n}$ plus lower-order terms, which is exactly what we wanted from a compressed representation [5].

This block based structure works efficiently when n is a great part of u. When n is much smaller than u, however, the bitvector is very sparse, a representation that stores only the positions of the 1s can be a better fit, but that case is not the focus of this thesis [5].

2.3 Worst-case entropy

A direct and inefficient method to store a collection of sets $\mathcal{S} = \{S_1, \ldots, S_s\}$ for a universe of size u is to encode each set as its own bitvector. If S_i has size n_i, its lower bound on space is $\lg \binom{u}{n_i}$ bits, and summing across the whole collection gives the worst-case entropy [1]

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

$H(\mathcal{S})$ is the lower bound on space when the sets are considered as independent, each set can be any of $\binom{u}{n_i}$ subsets of the universe. In order to go below $H(\mathcal{S})$, the encoding has to take advantage of the structure shared between the sets. This means utilising that some sets are subsets of others and can thus be stored more efficiently, which is what is done in containment entropy.

2.4 Containment entropy

When one set in the collection is contained inside another, storing the two sets unrelated and independent is wildly inefficient. If $S_i \subset S_j$, then S_i does not need to be described as a subset of the full universe $\mathcal{U}$. Instead, it only needs to be represented as a subset of S_j. There are only

$$\binom{|S_j|}{|S_i|}$$

possible subsets of S_j with size $|S_i|$, which is at least as many or fewer than

$$\binom{u}{|S_i|}$$

possible subsets of the full universe. Containment entropy capitalises on this by assigning each set S_i a parent $p(S_i)$, the smallest set in the collection that contains S_i. If no proper superset exists, then S_i is assigned the universe $\mathcal{U}$ as its parent [1].

Each set is then stored as a bitvector of length $|p(S_i)|$, where a 1 in position j means that the j-th element of the parent also belongs to S_i. To exemplify this, take the three sets

$$S_1 = \{1, 3, 5, 7\}, \quad S_2 = \{1, 5, 7\}, \quad S_3 = \{5, 7\},$$

in the universe $\mathcal{U} = \{1, 2, \ldots, 8\}$, with $S_3 \subset S_2 \subset S_1$.

Stored as plain universe relative bitvectors of length 8, the three sets make up the tree

$$\begin{array}{c} \mathcal{U} \\ | \\ S_1 = 10101010 \\ | \\ S_2 = 10001010 \\ | \\ S_3 = 00001010 \end{array}$$

Now to actually utilise their similarity through being subsets, encoding each child relative to its parent, where the bit in position j corresponds to the j-th 1 in the parent:

$$\begin{array}{c} \mathcal{U} \\ | \\ S_1 = 10101010 \\ | \\ S_2 = 1011 \\ | \\ S_3 = 011 \end{array}$$

The total length drops from 24 to 15 bits.

Due to this type of structure, summing the lower bound across the collection gives [1]

$$L(\mathcal{S}) = \sum_{i=1}^{s} \lg \binom{|p(S_i)|}{|S_i|}. \tag{2}$$

$L(\mathcal{S})$ is never larger than $H(\mathcal{S})$, and is strictly smaller as soon as any set has a superset in the collection [1]. The collection of sets together with their parent assignments forms a tree like structure where the universe is like a shared root. Consequently we get the following inequality:

$$L(\mathcal{S}) \leq H(\mathcal{S}). \tag{3}$$

Given this tree like structure, in order to make it as efficient as possible, each bitvector is turned into the corresponding compressed bitvector representation as seen in the section above.

2.5 Select

The select operation on a set S takes an index k and returns the universe value of the k-th smallest element of S. Take the example, if $S = \{5, 7\}$ then $select(S, 2) = 7$, because 7 is the second smallest value in the set.

On a single compressed bitvector this is constant time. The stored array $S[\,]$ already lists the positions of the 1s in order, so the k-th 1 is read straight out of it in constant time. The complexity that arises in the containment structure is that a set's bitvector is not measured against the universe but against its parent, so the k-th 1 of S is only a position inside that parent and not the universe value. To turn it into an actual value, the position has to be carried up the hierarchy one level at a time until the root is reached.

That is precisely the task the retrieval algorithm in the following section performs, so select on a set is carried out by $retrieve(S, k)$. Each level does a single constant-time bitvector select to turn the current index into a position in the parent, and at the root which is the universe it returns the value directly. This query therefore costs $O(\text{depth})$, which the contracted hierarchy brings down to $O(\log(u/|S|))$.

2.6 Retrieval

The problem is the following, given a set in a containment entropy structure, we cannot directly read off the value of its elements from the bitvector or compressed bitvector alone, because each set is stored relative to its parent [1]. This algorithm therefore retrieves the value of an element in a set S_i recursively.

$Retrieve(S, k)$ returns the universe value of the k-th element of S by translating the index up through the hierarchy one level at a time. At each level, $select(S, k)$ turns the k-th 1 of S into its position inside the parent $p(S)$, and that position becomes the index for the next call. The recursion stops at the root, where the parent is the universe and $select(\mathcal{U}, k)$ gives the universe value directly.

Algorithm 1 Retrieve the universe value of the k-th element of a set

```
function RETRIEVE(S, k)
    if S = U then
        return select(U, k)
    end if
    j ← select(S, k)
    return RETRIEVE(p(S), j)
end function
```

The function walks up the hierarchy, doing one constant time select per level, so the whole query takes $O(\text{depth})$ time.

The following is a worked example furthering on the previous example, with $S_3 \subset S_2 \subset S_1$ over $\mathcal{U} = \{1, \ldots, 8\}$, stored relative to their parents as

$$S_1 = 10101010, \quad S_2 = 1011, \quad S_3 = 011.$$

$retrieve(S_3, 2)$ is called, querying for the universe value of the 2nd element of $S_3 = \{5, 7\}$, which should come out as 7. The recursion unfolds as follows.

- $retrieve(S_3, 2)$: the 1s of $S_3 = 011$ sit at positions 2 and 3, so $select(S_3, 2) = 3$. Recurse with $retrieve(S_2, 3)$.
- $retrieve(S_2, 3)$: the 1s of $S_2 = 1011$ sit at positions $1, 3, 4$, so $select(S_2, 3) = 4$. Recurse with $retrieve(S_1, 4)$.

- *retrieve*$(S_1, 4)$: the 1s of $S_1 = 10101010$ sit at positions $1, 3, 5, 7$, so *select*$(S_1, 4) = 7$. Recurse with *retrieve*$(\mathcal{U}, 7)$.
- *retrieve*$(\mathcal{U}, 7)$: this is the base case at the root, where *select*$(\mathcal{U}, 7) = 7$ returns the universe value directly.

The recursion returns 7, which matches the 2nd element of $S_3 = \{5, 7\}$. The query used one select per level across the three levels, which further demonstrates the O(depth) cost.

2.7 Rank

The rank operation on a set S_i takes a value k from the universe and returns how many elements of S_i are at most k. As a small example, if $S_i = \{5, 7\}$ and $k = 6$, the answer is 1, because only the element 5 is at most 6.

The difficulty lies in that the bitvector for S_i is not measured against the universe. It is once again relative to the parent $p(S_i)$, so a 1 in position j means that the j-th element of the parent also belongs to S_i. The positions in that bitvector therefore count elements of the parent, not values in the universe. We cannot simply call *rank*(S_i, k), because k is a universe value while the bitvector expects a position counted in the parent's elements. The cutoff k first has to be translated into the system that S_i actually uses.

Rank is the counterpart of retrieval, and we implement it here on top of the same log-time navigation up the parent chain that the structure provides [1]. This translation has the same recursive shape as retrieval, but it runs both up and down the recursion in the steps as follows.

- The cutoff k travels up the hierarchy unchanged, from S_i towards the root. None of the sets along the way can interpret a raw universe value either, so each one simply passes k to its parent.
- The recursion stops at the universe. There *rank*$(\mathcal{U}, k)$ counts how many universe elements are at most k. This number is the position of k measured against the universe, and it is the first value that actually means something.
- Each level receives a position p from the level above and calls *rank*(S_i, p). This counts how many of S_i's own elements fall within the first p elements of its parent, which is the cutoff expressed in S_i's coordinate system. That count is then handed to the level below.
- The value is then returned at the very bottom in the original set which is the number of elements of S_i that are at most k.

Algorithm 2 Count the elements of a set that are at most a universe value k

```
function RankSet(S, k)
    if S = U then
        return rank(U, k)
    end if
    p ← RankSet(p(S), k)
    return rank(S, p)
end function
```

Each level does one rank query, so the whole operation costs O(depth) time, the same as retrieval.

To see it in action, the three nested sets are used, with $S_3 \subset S_2 \subset S_1$ in the full universe $\mathcal{U} = \{1, \ldots, 8\}$, stored relative to their parents as

$$S_1 = 10101010, \quad S_2 = 1011, \quad S_3 = 011.$$

Calling *rank*$(S_3, 6)$, querying how many elements of $S_3 = \{5, 7\}$ are at most 6, which should come out as 1. The recursion first walks up to the root, then brings the answer back down.

- Going up, S_3 passes 6 to S_2, which passes 6 to S_1, which passes 6 to the universe.
- At the root, *rank*$(\mathcal{U}, 6) = 6$, since the universe holds all of $1, \ldots, 8$ and six of them are at most 6.
- Back in S_1, we take *rank*$(S_1, 6)$. The first 6 bits of $S_1 = 10101010$ are $1, 0, 1, 0, 1, 0$, which hold three 1s, so the result is 3.

- Back in S_2, we take $rank(S_2, 3)$. The first 3 bits of $S_2 = 1011$ are $1, 0, 1$, which hold two 1s, so the result is 2.
- Back in S_3, we take $rank(S_3, 2)$. The first 2 bits of $S_3 = 011$ are $0, 1$, which hold one 1, so the result is 1.

The recursion returns 1, which matches the single element 5 of $S_3 = \{5, 7\}$ that is at most 6. As with retrieval, the query touched one level at a time, giving the $O(\text{depth})$ cost.

2.8 Contracted hierarchy

The idea behind the contracted hierarchy is to shorten the depth of the tree structure in containment entropy. As it stands, every set finds its smallest superset, which results in a large chain of parents and with a possible great depth. This is a problem because as mentioned before retrieval and rank largely depend on the depth given their $O(\text{depth})$ time.

The main concept behind the contracted hierarchy is to find the highest ancestor of $p(S_i)$ with size $\leq 2|S_i|$ [1]. This caps the hierarchy depth at $O(\log u)$. The reason is that two consecutive steps up the contracted chain more than double the set size, the contracted parent has size at most $2|S_i|$, while the next ancestor above it must be larger than $2|S_i|$, since otherwise it would have been chosen instead. A size that keeps doubling can only do so $\lg u$ times before it reaches the universe, so the chain has at most $O(\log u)$ levels.

This alteration does not change how many bitvectors are stored, since each set still has exactly one parent, but it does cost a little extra space, because a set encoded against a larger contracted parent gives a denser relative bitvector. That overhead is bounded rather than free, since the contracted parent is at most twice the size of the set.

The result of this change allows for better query times for retrieval and rank, resulting in their new running time, $O(\log(u/|S|))$ [1]. Since each step at least doubles the set size, a query that starts from a set of size $|S|$ reaches the universe of size u after about $\lg(u/|S|)$ steps, and each step does only constant work.

As an example, take eight nested sets $S_1 \subset S_2 \subset \cdots \subset S_8$ with sizes $|S_j| = j$, in a large universe. In the normal hierarchy each set points at the next one up, creating the long chain

$$S_1 \to S_2 \to S_3 \to \cdots \to S_8 \to \mathcal{U},$$

so a query starting from S_1 has to pass through seven sets. Under contraction, S_1 points at the highest ancestor of size at most $2|S_1| = 2$, which is S_2, then S_2 points at the highest of size at most 4, which is S_4, and S_4 points at the highest of size at most 8, which is S_8. The chain from S_1 shortens to

$$S_1 \to S_2 \to S_4 \to S_8 \to \mathcal{U},$$

This shortens the depth from seven steps to three, while still pointing each set at a parent maximally twice its size.

3 Implementation

The implementation is where the brunt of the problem lies. For it is where the theory meets the practice in a way that provides the user with the desired result. A number of design choices have been made in the implementation to work the way that it does and this section will highlight the pipeline and inner workings of the practical code written.

3.1 Overview

There are two programs in question, both are written in C and compiled with the gcc command. The respective programs are Baseline.c and CEB.c (CEB: Containment Entropy with Baseline encoding). The implementations are composed as such, the Baseline encoding keeps the general structure of the sets as explained previously, however every set is stored independently as a single compressed bitvector that offers rank and select support in $O(1)$ time [5]. The other program, CEB, unlike the Baseline, alters the structure and creates a hierarchical tree that stores the bitvector relative to its parent $p(S_i)$ [1]. It then uses the Baseline encoding to encode the relative bitvectors in order to still be able to perform efficient rank and select operations.

Memory was also an area where a lot of solutions could be implemented, but for the sake of this project it is allocated statically with fixed upper bounds. The current program's upper bounds are set using variables with the

MAX_ELEMENT_VALUE = 1024 and the MAX_SETS = 128 for the largest collection of sets. This was designed to keep memory handling simple and predictable and avoid the troubles with dynamic allocation.

Additionally, in order to make the implementation solid and structurally sound for comparison, the containment program reuses the Baseline bitvector code through a shared header, so the two implementations encode and query bitvectors in exactly the same way.

Another remark in the way CEB builds its tree, it does not simply attach every set to its smallest superset, as that can create a long and thin chain of parents that is slow to climb. Instead it uses contracted parent assignment, where the parent of a set S_i is the highest ancestor whose size is still at most $2|S_i|$ [1]. This keeps the tree relatively shallow and caps the number of levels a query has to walk through, while the size bound guarantees that the relative bitvectors do not grow too large.

Finally, the whole experiment is written in the CEB.c file, which builds and compares both representations on the same data. Its `main` sweeps the experimental configurations and calls a single driver function, `run_config`, once for each, generating the data, building both structures, measuring space and time, and checking every answer against a brute force scan, all of which happens inside that one function, so the whole flow of the experiment can be followed from a single place. Baseline.c is kept separately as a small standalone demo that encodes one example set and prints the resulting arrays, so the block encoding can be inspected on its own. This is also the case with CEBTemp.c allowing for easier readability with the isolated Containment Entropy structure with the Baseline encoding. This allows the algorithms and data structure to be simplistic and easily readable as a whole.

3.2 Baseline implementation

3.2.1 Idea

The core idea behind the Baseline implementation is to create a simple and intuitive implementation of the Baseline which involves encoding the sets into independent bitvectors made up of different stored arrays such that the operations can be performed in a timely manner while the space complexity remains manageable.

More precisely, the goal is to store a bitvector close to the lower bound $\lg \binom{u}{n}$ bits [5]. This method involves the aforementioned stored arrays storing relevant bits that describe the original set in tandem with combinatorial logic, this allows queries to skip directly to the block they need. This is what allows for the constant time queries.

3.2.2 Block encoding

The first step of the Baseline is to split the bitvector into fixed sized blocks of b bits each. Here the bitvector is the vector of a set in a universe of size u, it has length u and a 1 at position i when the value i belongs to the set. The blocks make up the vector from left to right, and because of this, the experiment pads each bitvector with trailing 0s so that its length is a multiple of b, the padding only adds empty all zero blocks and does not change the set. The implementation uses $b = 4$ for the worked examples below, but as discussed at the end of this section the value of b is a parameter that strongly affects both the space and the speed of the structure. Each block is then described by two numbers, the class and the offset, and these two numbers together are enough to rebuild the block exactly.

The class c is simply the number of 1s in the block. The offset says which of the equally heavy blocks this particular one is. If a block has class c, then there are exactly $\binom{b}{c}$ different blocks with that same number of 1s. If we list all of those patterns in order of increasing numeric value and number them from 0, the offset is the position of our block in that list. The pair (c, o) is therefore a complete description, since the class tells us which list to look in and the offset tells us where in that list the block sits.

Because there are $\binom{b}{c}$ patterns to tell apart, the offset can be stored in $L[c] = \lceil \lg \binom{b}{c} \rceil$ bits, which is the fewest bits that can distinguish them. A block that is all 0s or all 1s has only one possible pattern, so it needs no offset bits at all. This is where the saving comes from, since very light and very heavy blocks cost far less bits than the naive b bits per block.

Rather than building the ordered list explicitly, the encoder function uses the combinatorial number system, which computes the offset directly with the help of a precomputed table of binomial coefficients $K[\,][\,]$, where $K[n][k] = \binom{n}{k}$ [5]. This table is just Pascal's triangle and is filled once at startup. Walking the block from left to right, every time a 1 is met at position j the encoder adds $K[b-j][t]$ to the offset, where t is the number of 1s that still remain to be placed,

including the current one. The walk touches each of the b positions exactly once, so encoding takes $O(b)$ time, which is constant for a fixed block size [5].

Example: encoding the block 1010. This is exactly what the `encode` function does. Here $b = 4$ and the leftmost bit is position 1. The function first counts the 1s to set the class $c = 2$, then sets $o = 0$ and a counter $t = c$ for the 1s still to be placed. Reading each bit with `bit_read` and adding $K[b-j][t]$ on every 1 (the line `*o_out += K[b-j][ctemp]` in the code), it walks the positions:

- Position 1, bit 1: add $K[3][2] = \binom{3}{2} = 3$, so $o = 3$, then $t = 1$.
- Position 2, bit 0: skip.
- Position 3, bit 1: add $K[1][1] = \binom{1}{1} = 1$, so $o = 4$, then $t = 0$.
- Position 4, bit 0: skip.

The block is therefore described by the class 2 and the offset 4. As a check, the six class 2 patterns in increasing value order are

$$0011, \quad 0101, \quad 0110, \quad 1001, \quad \underline{1010}, \quad 1100,$$

numbered 0 to 5, and 1010 sits at index 4. The offset is then stored in $L[2] = \lceil \lg 6 \rceil = 3$ bits, namely the bit string 100.

Algorithm 3 Encoding a block into a class and an offset

```
c ← number of 1s in B
o ← 0
t ← c
for j ← 1 to b do
    if B[j] = 1 then
        o ← o + K[ b − j ][ t ]
        t ← t − 1
    end if
end for
return (c, o)
```

Decoding runs the same idea backwards and turns a (class, offset) pair back into the block. Starting with an empty block, at each position j the decoder checks whether the remaining offset is at least $K[b-j][c]$. If it is, that position must hold a 1, so the decoder sets the bit, subtracts that term from the offset, and lowers the class, otherwise the position is a 0. Like encoding, it goes through each position once and so also runs in $O(b)$ time [5].

Example: decoding the class 2 and offset 4. This is what the `decode` function does. It starts with an empty block `B` and, at each position, compares the remaining offset against $K[b-j][c]$. When the offset is at least that large it sets the bit with `bit_set`, subtracts $K[b-j][c]$ from the offset, and lowers c:

- Position 1: compare $o = 4$ with $K[3][2] = 3$. Since $4 \geq 3$, set a 1, leave $o = 1$, lower the class to 1.
- Position 2: compare $o = 1$ with $K[2][1] = 2$. Since $1 < 2$, leave a 0.
- Position 3: compare $o = 1$ with $K[1][1] = 1$. Since $1 \geq 1$, set a 1, leave $o = 0$, lower the class to 0.
- Position 4: compare $o = 0$ with $K[0][0] = 1$. Since $0 < 1$, leave a 0.

The recovered block is

$$1010,$$

exactly the original.

Algorithm 4 Decoding a class and offset back into a block

```
B ← 0
for j ← 1 to b do
    if o ≥ K[b − j][c] then
        B[j] ← 1
        o ← o − K[b − j][c]
        c ← c − 1
    end if
end for
return B
```

The block size b is a main parameter of the Baseline, and clearly demonstrates a trade-off between space and time. A bitvector of length u is cut into u/b blocks, and the class array stores one class per block using $\lceil \lg(b+1) \rceil$ bits each, so the class array alone costs about $(u/b)\lceil \lg(b+1) \rceil$ bits. A small block size means many blocks and therefore a long class array, together with more checkpoints, this inflates the space on top of the entropy. A large block size means few blocks and a short class array, but the offsets grow, the per block work of encoding and decoding grows with b since both are $O(b)$, and the offset values and table entries must still fit inside a word size, which here is 64 bits and so bounds how large b can be. There is also a small rounding waste of up to about one bit per block from the ceiling in $L[c]$, and this waste shrinks as there are fewer blocks.

The ideal block size is therefore the one that keeps the class array and checkpoint overhead small without making the offsets and the query work too large. In theory a block size on the order of $\lg u$ keeps the offsets within a single word while keeping the trade-off low. Which value is best in practice depends on the data and the use case. Rather than assuming a block size of $\lg u$ the experiment covers several block sizes, from $b = 4$ upward, and reports the bits per element and the query time for each. The worked examples above use $b = 4$ for readability, since the blocks are small enough to follow by hand.

3.2.3 Stored arrays

The representation of a single bitvector is spread across a handful of arrays, all held in one `Bitvector` struct and filled by `bitvector_build`. The pair (C, O) is enough to rebuild the original bitvector, while the remaining arrays exist only to keep queries fast. Each array is produced by a specific function:

- $C[\,]$ (class array): the class of every block, one small integer per block, written with $\lceil \lg(b+1) \rceil$ bits each. `bitvector_build` fills it by calling `encode` on each block which returns the class as the block's number of 1s.
- $O[\,]$ (offset array): all the offsets packed one after another with no padding between them, so block i occupies exactly $L[C[i]]$ bits and consecutive blocks can have different widths. The offset for each block also comes from `encode`, and `bits_write` appends it to $O[\,]$ using $L[c]$ bits.
- $L[c]$ (width lookup): the offset width for class c, equal to $\lceil \lg \binom{b}{c} \rceil$. It is computed once by `createL`, which counts the bits needed to hold $\binom{b}{c} - 1$.
- $K[\,][\,]$ (Pascal's triangle table): the binomial coefficients that `encode` and `decode` read. It is filled once by `populateK` using the following structure $K[i][j] = K[i-1][j-1] + K[i-1][j]$.
- $R[\,]$ (rank checkpoints): the accumulated number of 1s seen so far, sampled by `bitvector_build` once every `SUPERBLOCK_FREQ` blocks.
- $P[\,]$ (position checkpoints): the bit position inside $O[\,]$ where the corresponding sampled block's offset begins. Because the blocks in $O[\,]$ have different widths, $P[\,]$ is what lets a query jump straight into $O[\,]$.
- $S[\,]$ (select array): the positions of all 1-bits in increasing order, filled by `CreateS` in a second pass so that select is a single lookup.

In actuality, these arrays are filled in one go by `bitvector_build`. Before the main loop the function calls `populateK` once to build K and `createL` once to build L for the chosen block size, since both are fixed lookups that the rest of the work reads from. It then keeps two running counters, `o_pos`, the next free bit position in O, which starts at 1, and

`rank_count`, the number of 1s seen so far, which starts at 0. The loop visits the blocks in order, with block i starting at bit $i \cdot b + 1$ of the input, and does three things per block. First, whenever i is at a k'th checkpoint it stores the current counters as a checkpoint, writing `rank_count` into $R[i/k]$ and `o_pos` into $P[i/k]$. This is why R and P always begin with the pair $(0, 1)$ for block 0. Secondly, it calls `encode` on the block to get the class c and offset o, stores c in $C[i]$, and, when the class needs offset bits (where $L[c] > 0$), calls `bits_write` to add o into O at position `o_pos` using exactly $L[c]$ bits, then moves `o_pos` by $L[c]$. Thirdly, it adds the class to `rank_count`, because the class is the block's number of 1s. After the loop the final value of `o_pos` records the total offset length, and a single call to `CreateS` goes through the bitvector once more and writes the position of every 1 into S. The whole construction is therefore one linear pass to fill C, O, R and P, plus one more linear pass for S.

Example: three nested sets. To make the arrays concrete, and to set up the same data for the containment entropy section, take a universe of values 1 to 20, a block size $b = 4$, and a checkpoint spacing of $k = 4$, so each 20-bit vector is five blocks and is sampled at blocks 0 and 4. For this example, there are three sets, each a subset of the one before it:

$$S_1 = \{1, 3, 7, 9, 10, 11, 12, 17, 18, 19\}, \qquad S_2 = \{3, 10, 12, 17, 19\}, \qquad S_3 = \{3, 12, 19\}.$$

For $b = 4$ the width lookup is the same for all of them, since $\binom{4}{0} = \binom{4}{4} = 1$ need no offset bits, $\binom{4}{1} = \binom{4}{3} = 4$ need 2 bits, and $\binom{4}{2} = 6$ needs 3 bits:

$$L = [\,0,\ 2,\ 3,\ 2,\ 0\,] \quad \text{for classes } 0, 1, 2, 3, 4.$$

At first S_1, it is written as a bitvector of length 20 and cut into five blocks

$$\underbrace{1010}_{c=2}\ \underbrace{0010}_{c=1}\ \underbrace{1111}_{c=4}\ \underbrace{0000}_{c=0}\ \underbrace{1110}_{c=3}.$$

This is exactly what `bitvector_build` walks over. For each block it calls `encode`, which returns the class (the block's number of 1s) and the offset, and then `bits_write` appends that offset to $O[\,]$ using $L[c]$ bits:

- 1010 is class 2, offset 4, stored in 3 bits as 100,
- 0010 is class 1, offset 1, stored in 2 bits as 01,
- 1111 is class 4, offset 0, stored in 0 bits (nothing),
- 0000 is class 0, offset 0, stored in 0 bits (nothing),
- 1110 is class 3, offset 3, stored in 2 bits as 11.

While walking the blocks, `bitvector_build` also samples the checkpoints at blocks 0 and 4 (every $k = 4$ blocks), before block 0 no 1s have been seen and the offset is empty, and by block 4 the first four blocks have $2 + 1 + 4 + 0 = 7$ ones and filled five offset bits, which become the second entries of R and P (positions in O are counted from 1). A final pass by `CreateS` records the member positions into S. Collecting everything, the stored arrays for S_1 are

$$\begin{aligned} &C = [\,2,\ 1,\ 4,\ 0,\ 3\,], && O = 1000111,\\ &R = [\,0,\ 7\,], && P = [\,1,\ 6\,],\\ &S = [\,1, 3, 7, 9, 10, 11, 12, 17, 18, 19\,]. \end{aligned}$$

The same procedure on S_2 gives the blocks

$$\underbrace{0010}_{c=1}\ \underbrace{0000}_{c=0}\ \underbrace{0101}_{c=2}\ \underbrace{0000}_{c=0}\ \underbrace{1010}_{c=2},$$

where the non-empty offsets are $0010 \to 01$, $0101 \to 001$, and $1010 \to 100$. The stored arrays are

$$\begin{aligned} &C = [\,1,\ 0,\ 2,\ 0,\ 2\,], && O = 01001100,\\ &R = [\,0,\ 3\,], && P = [\,1,\ 6\,],\\ &S = [\,3, 10, 12, 17, 19\,]. \end{aligned}$$

Finally S_3 gives the blocks

$$\underbrace{0010}_{c=1}\ \underbrace{0000}_{c=0}\ \underbrace{0001}_{c=1}\ \underbrace{0000}_{c=0}\ \underbrace{0010}_{c=1},$$

where each non-empty block is class 1 with offsets $0010 \to 01$, $0001 \to 00$, and $0010 \to 01$. The stored arrays are

$$C = [\,1,\ 0,\ 1,\ 0,\ 1\,], \qquad O = 010001,$$
$$R = [\,0,\ 2\,], \qquad P = [\,1,\ 5\,],$$
$$S = [\,3, 12, 19\,].$$

These three sets are nested, $S_3 \subset S_2 \subset S_1$. This also means that the union of these three sets is S_1. That is the structure the containment entropy section exploits in later sections, rather than having to store each set against the full universe of 20 values, it stores S_1 as the universe and then each subset against the one that contains it, which is where the space savings come from.

3.2.4 Construction pipeline

Putting the pieces together, building the structure for one bitvector is a short pipeline that `bitvector_build` runs from start to finish:

1. Call `populateK` to fill the Pascal table K and `createL` to fill the width lookup L for the chosen block size.
2. For each b-bit block in turn, call `encode`, which reads the block's bits with `bit_read` and returns its class and offset, store the class in C, and when the class needs offset bits ($L[c] > 0$) call `bits_write` to add the offset into O at the current position `o_pos` using $L[c]$ bits.
3. Every k blocks, record the accumulated count for 1s `rank_count` into R and the current offset position `o_pos` into P so later queries can jump in without starting at the beginning.
4. After the block loop, call `CreateS`, which loops through the bitvector with `bit_read` and writes every position for 1s into S for select.

Steps 2 and 3 share the same single pass inside `bitvector_build`, and step 4 is one more pass inside `CreateS`, so the whole construction is linear with the length of the bitvector.

3.2.5 Query operations

The fast query times result from this efficient structure. They allow for constant time rank, select and access. The following three functions are the query operations of the Baseline, and each one is explained below and then shown on the set $S_1 = \{1, 3, 7, 9, 10, 11, 12, 17, 18, 19\}$ from the stored arrays example, whose arrays were

$$C = [\,2, 1, 4, 0, 3\,], \quad O = 1000111, \quad L = [\,0, 2, 3, 2, 0\,], \quad R = [\,0, 7\,], \quad P = [\,1, 6\,], \quad S = [\,1, 3, 7, 9, 10, 11, 12, 17, 18, 19\,].$$

Rank. `bitvector_rank`(i) returns how many 1s lie in positions 1 to i. It does this in the following way.

Firstly, it jumps to the nearest checkpoint before position i. That checkpoint is number, $is = \lfloor (i-1)/(b \cdot k) \rfloor$, and from it the function reads two stored values, namely, the count of 1s seen so far, $r = R[\texttt{is}]$, and a pointer to where the next offset begins, $p = P[\texttt{is}]$. These two values are what let the query skip everything in front of the checkpoint.

Secondly, it goes through the few whole blocks between that checkpoint and the block holding position i. For each one it looks up the block's class in $C[\,]$, which is simply its number of 1s, adds that to r, and moves p forward by $L[c]$ bits so the pointer keeps pointing at the start of the next block's offset. Since checkpoints are only k blocks apart, this loop repeats at most k times, a fixed amount.

Thirdly, it handles the block that actually contains position i. It reads that block's offset out of $O[\,]$, starting at p and taking $L[c]$ bits, using `bits_read_simple`, and passes the result to `decode`, which rebuilds the original block. It then reads just the bits of that block before the relevant index, again with `bits_read_simple`, and counts their 1s with `popcount`. Adding this final count to r gives the answer. Each step does a fixed amount of work, so the whole query runs in $O(1)$ time.

The time is constant and the guarantee is with respect to the size of the data, neither the length of the bitvector nor the number of 1s it holds changes how many operations a rank performs, because the checkpoints let the query skip straight to a fixed neighbourhood of blocks rather than scan from the front. It is not constant in the block size b. The two steps that touch raw bits, decoding the block that contains i and reading its leading bits, both do work that grows with b, so a larger block would make the rank call slower even though the cost stays flat in the size of the input. This is the trade-off behind the block size experiments later in the thesis. Logically, a larger b packs the offsets more tightly and lowers the space, but it lengthens every rank, which is why the measured rank time should increase with b whereas the bits per element fall.

Example: `bitvector_rank`(7) on S_1. With $b = 4$ and $k = 4$ the checkpoint index is `is` $= \lfloor 6/16 \rfloor = 0$, so the function starts from $r = R[0] = 0$ and $p = P[0] = 1$. Position 7 lies in block 1 (the second block), so the loop walks only block 0, its class is $C[0] = 2$, giving $r = 2$ and advancing the pointer by $L[2] = 3$ to $p = 4$. The containing block is block 1 with class $C[1] = 1$, its offset is read from O at bits 4 to $4 + L[1] - 1 = 5$, namely $01 = 1$, and `decode` rebuilds the block

$$\texttt{decode}(1, 1) = 0010.$$

Position 7 is the third bit of that block, and bits 1 to 3 are 001, which holds one 1. The result is therefore

$$r + \texttt{popcount}(001) = 2 + 1 = 3,$$

matching the members $\{1, 3, 7\}$ that are at most 7.

Select. `bitvector_select`(j) returns the position of the j-th 1. Because `CreateS` already stored every 1-position in order in $S[\,]$, the function is simply the lookup $S[j - 1]$ (the array is zero-indexed), which is a single access and therefore $O(1)$. This keeps the select implementation as simple as possible.

Example: `bitvector_select`(5) on S_1. The function is the single lookup

$$\texttt{select}(5) = S[4] = 10.$$

The fifth smallest member of S_1 is indeed 10, since the members in order are $1, 3, 7, 9, 10, \ldots$

Access. `bitvector_access`(i) returns the single bit at position i, that is, whether the value i is in the set. It reuses the same idea as rank without the counting, it finds the block that contains position i, uses the nearest checkpoint in $P[\,]$ and then adds the widths $L[c]$ of the blocks in between to land on that block's offset inside $O[\,]$, reads the offset with `bits_read_simple`, calls `decode` to rebuild the block, and finally returns the bit at position $((i - 1) \bmod b) + 1$ within it with `bit_read`. Like rank it only goes over a constant number of blocks, so it is $O(1)$.

Example: `bitvector_access`(2) on S_1. Position 2 lies in block 0, whose class is $C[0] = 2$. Its offset is read from O at bits 1 to $L[2] = 3$, namely $100 = 4$, and `decode` rebuilds the block

$$\texttt{decode}(2, 4) = 1010.$$

Position 2 is the second bit of that block, which is 0, so $\texttt{access}(2) = 0$ and the value 2 is not in S_1.

3.2.6 Running, measuring and checking the experiment

Due to the nature of this algorithm and data structure, code simplicity is highly prioritised, so the Baseline is measured as part of the same driver that runs the containment experiment, `run_config` in CEB.c. On each configuration the code generates the collection in memory and calls `build_baseline`, which stores every set as its own compressed bitvector against the shared universe using `bitvector_build`. It then times the Baseline's queries on that same data. This is done to measure how the block size affects the structure, `main` goes through several values of b and calls `run_config` once for each, so a full run clearly illustrates the whole space and time picture across the different block sizes at once.

The main focal point is the space. To create a very comparable figure, the space is reported as bits per element and is computed by `payload_bits`. For each set the stored payload is made of two parts. The first is the class array,

which holds one class per block. A class is the number of 1s in a block of b bits, so it can take any of the $b+1$ values $0, 1, \ldots, b$, the different possibilities are what need $\lceil \lg(b+1) \rceil$ bits per block. This is exactly the width the helper `class_bits` returns. The second part is the offsets, which occupy `o_pos` $-$ 1 bits in total, since `o_pos` is left pointing just past the last offset that was written. These two parts are summed over all sets and divided by the total number of 1s across all sets to give the final bits-per-element figure. The arrays R, P and S are left out of this count, so the number reflects only the part of the structure that actually approaches the entropy, a different count is then also made to include them to get a total picture.

Precisely what is being measured is the logical size of the encoding, which is the number of bits the class and offset data actually occupy for the given input. It is not the same as the program's memory, because the arrays are declared at fixed upper bounds, so the `Bitvector` struct has the same size regardless of how many elements a set has. A real deployment and possible future work would pack the payload into exactly these bits, but the fixed size arrays are a simplification and the reported space is what such a packed representation would use.

Alongside the space, the experiment also records timing. This is to ensure that we can compare the solutions using this as well. In general it is a trade-off between time and space, where we cannot decrease one without increasing the other, thus it is relevant to measure both. Build time and per query times are measured with `clock_gettime(CLOCK_MONOTONIC)`, and the helper `ms` turns the intervals into milliseconds, so the output can give both the total time and the average time per query.

Finally, every run is checked for correctness rather than trusted blindly. Before any set is made parent-relative, `record_brute_force` records each set's true members, and `make_queries` turns that record into the expected answer for every query. Every answer the Baseline returns is then compared against its expected value, and the run reports how many matched, so a run is only meaningful if all of them do.

3.2.7 Role in this thesis

The Baseline serves as the foundation and the point of comparison for the whole project. By itself, it already does something useful by storing each set in space close to the worst-case entropy $H(\mathcal{S})$ while still answering rank, select and access in constant time. However, its main role is to be the reference that the containment hierarchy is measured against. Because the containment program reuses this exact same block encoding to store its relative bitvectors, any difference seen in the experiments comes from the hierarchy and not from a change in how the bitvectors themselves are encoded. This makes the comparison fair. The Baseline shows what it costs to store every set independently, and the gap between that and the containment entropy solution, measured both in bits per element and in query speed, is what reveals if exploiting containment actually pays off. The rest of this chapter builds the containment implementation on top of this same foundation.

3.3 Containment entropy implementation

3.3.1 Idea

The containment implementation idea is rather simple, when the sets in a collection overlap heavily or nest inside one another, storing each one independently against the full universe repeats a lot of the same information [1]. The Baseline pays $\lg \binom{u}{n_i}$ bits for every set S_i, where u is the size of the universe, even when S_i sits entirely inside another set that has already been stored. The containment idea is to store each set not against the whole universe but against a set that contains it, called its parent $p(S_i)$. Because the parent is usually much smaller than the universe, so far fewer bits are needed to single out the child inside it.

Concretely, a set is stored as a parent relative bitvector, walking through the 1s of the parent in order, it records a 1 wherever that element is also in the child and a 0 otherwise. Its length is therefore the size of the parent, not the size of the universe. The three nested sets from the previous section make this concrete:

$$S_1 = \{1, 3, 7, 9, 10, 11, 12, 17, 18, 19\}, \quad S_2 = \{3, 10, 12, 17, 19\}, \quad S_3 = \{3, 12, 19\}.$$

Their union is S_1, so S_1 is the root and acts as the universe for the others. To store S_2 relative to S_1, we walk the ten elements of S_1 in order and write a 1 for each that is also in S_2 and a 0 otherwise, five of them are, giving the ten-bit string 0100101101. Storing S_3 relative to S_2 works the same way over the five elements of S_2, three of which

lie in S_3, giving 10101. Each relative bitvector therefore has one bit per element of its parent and is exactly as long as the parent is large, ten bits for S_2 and five for S_3.

The saving comes from the size of the relation. By S_3 inside its parent S_2 costs only $\lg \binom{5}{3} \approx 3.3$ bits, whereas describing the same set directly against a universe of 20 values would cost $\lg \binom{20}{3} \approx 10.2$ bits. Summed over the whole collection this is the containment entropy

$$L(\mathcal{S}) = \sum_i \lg \binom{|p(S_i)|}{|S_i|} \;\le\; \sum_i \lg \binom{u}{n_i} = H(\mathcal{S}),$$

since every parent is at most as large as the universe, so each term on the left is no larger than its match on the right. The universe vector itself is stored once and shared by all sets.

It is important that the sets must be arranged into a hierarchy so that every set has a containing parent, which is what the construction pipeline builds. Second, a query can no longer read its answer from a single bitvector, because a set knows its members only as positions inside its parent. It therefore starts at a set and walks up the hierarchy, translating positions from one level to the next with the same constant time rank and select until it reaches the universe and recovers the real value. This is why each node keeps a parent pointer and the query operations are recursive. Each set is still encoded with the same Baseline block encoding as above, so the per-bitvector representation is unchanged, only what each bitvector is measured against differs.

3.3.2 Data structures

This section describes how the containment entropy structure is implemented in the program. The design follows the theory closely, so that each idea from the previous sections has a direct part in the code and one can be read as a translation of the other. Practically, this means a small amount of structs, each of which holds one piece of the structure: the shared universe, the individual sets and the links between them, and a helper structure that makes building those links efficient. They are described one at a time below.

The first piece is the universe. It is a single shared bitvector that records every element appearing in any of the sets, and nothing more. It is built simply by taking the bitwise OR of all the input sets, so a position holds a 1 when at least one set contains that value. Because every set is later measured against this universe, it is stored only once and shared by all of them rather than copied into each set. In the code the `Universe` struct holds this bitvector in compressed form together with its size, which is just the number of 1s in the OR. Reusing the three nested sets from before makes this concrete. The universe is the OR of S_1, S_2 and S_3, and since $S_3 \subset S_2 \subset S_1$ the OR contributes nothing beyond S_1 itself. In the universe $\{1, \ldots, 20\}$ the shared bitvector is therefore

$$\mathcal{U} = 1010\,0010\,1111\,0000\,1110$$

which has ten 1s, so its size is 10. Because the sets are nested, this union is exactly S_1, the largest set. Had they not been nested, the smaller sets would contribute elements beyond S_1, so the universe would generally be larger.

Each individual set is stored in a `SetNode`. This struct holds three things: the set's size, its bitvector in compressed form, and a `parent_index` that records which other set, or the universe, it sits under in the hierarchy. The parent index is what turns the list of sets into a tree, and it is filled in during construction. The bitvector itself goes through two stages. While the collection is being built it is universe relative, meaning its positions refer to the universe. Afterwards it is then transformed into the parent-relative form that the containment entropy bound relies on, where its positions refer only to the smaller parent set. Keeping the size and parent alongside the bitvector means a single node carries everything a query operation needs to go up the hierarchy.

The `SetCollection` ties everything together into one object. It stores the shared universe, an array holding all of the `SetNode`s, and a count of how many sets there are. The construction pipeline fills it in, and the query operations read from it, so a set is always referred to by its index into this SetCollection array.

Building the hierarchy means finding, for each set, which other sets contain it. Doing this by comparing every set against every other set would cost a number of comparisons that grows with the square of the number of sets, thus it is avoided by using an `InvertedList`. This structure holds one entry per universe element, and each entry lists the

indices of all the sets that contain that particular element. To find the sets that contain a given set, it is then enough to look up the entries for that set's own elements and intersect them, since a set containing it must appear in every one of those entries. This replaces the pairwise search with a few list intersections and is what keeps construction efficient as the collection grows.

A set that sits directly under the universe, rather than under another set, has no real parent in the array, so its `parent_index` is set to the value `PARENT_IS_ROOT`, which is the universe. Several of the steps above need the bits in plain, uncompressed form while they work, so the program keeps a few scratch buffers for that purpose during construction. Once each bitvector has been compressed into its final form these buffers are released by `scratch_free`, leaving only the compact structure in memory.

3.3.3 Construction pipeline

The construction pipeline is what turns a plain list of input sets into the finished hierarchy. It runs once, before any queries are made. It is a sequence of small functions that each have their part in the construction as a whole. The steps follow the same order in the code, and the rest of this section walks through them one at a time.

The pipeline begins by building the shared universe. `build_universe` ORs all of the input sets together into one bit array and counts the 1s, which gives the universe and its size. Next, `load_sets` copies each input set into its own scratch buffer, counts its 1s to record the set's size, and marks every set's `parent_index` with the root, since no parents have been worked out yet. With the sets loaded, `sort_sets_by_size_desc` sorts them from largest to smallest using an insertion sort [3]. This ordering matters for the later steps, when a set's bitvector is rewritten relative to its parent, the parent must still be available in its original form, and processing the sets in size order is what guarantees this.

The core of the pipeline is working out who each set's parent is, and this is where the inverted list earns its place [4]. `build_inverted_list` fills in the structure described above, giving every universe element a list of the sets that contain it. `build_hierarchy` then uses it, for each set it looks up the inverted-list entries of that set's own elements and intersects them. A set that contains all of those elements must appear in every one of those entries, so the intersection is exactly the collection of sets that contain the set in question. Removing the set itself leaves its supersets, and the smallest of these is chosen as the parent. A set that turns out to have no proper superset keeps the universe as its parent.

Example: the inverted list of the three nested sets. Reusing S_1, S_2 and S_3 from before makes this concrete. Their elements are

$$S_1 = \{1, 3, 7, 9, 10, 11, 12, 17, 18, 19\}, \quad S_2 = \{3, 10, 12, 17, 19\}, \quad S_3 = \{3, 12, 19\},$$

so in the universe $\{1, \ldots, 20\}$ only ten values appear in any set at all. For each of those the inverted list records which sets contain it:

$$\begin{aligned} 1 &\to \{S_1\} & 11 &\to \{S_1\} \\ 3 &\to \{S_1, S_2, S_3\} & 12 &\to \{S_1, S_2, S_3\} \\ 7 &\to \{S_1\} & 17 &\to \{S_1, S_2\} \\ 9 &\to \{S_1\} & 18 &\to \{S_1\} \\ 10 &\to \{S_1, S_2\} & 19 &\to \{S_1, S_2, S_3\} \end{aligned}$$

To find the parent of $S_3 = \{3, 12, 19\}$, the program looks up the entries of its three elements. All three are $\{S_1, S_2, S_3\}$, so their intersection is $\{S_1, S_2, S_3\}$. Dropping S_3 itself leaves the supersets $\{S_1, S_2\}$, and the smaller of these is S_2, so the parent of S_3 is S_2. The same procedure on $S_2 = \{3, 10, 12, 17, 19\}$ intersects the entries $\{S_1, S_2, S_3\}$, $\{S_1, S_2\}$, $\{S_1, S_2, S_3\}$, $\{S_1, S_2\}$ and $\{S_1, S_2, S_3\}$ down to $\{S_1, S_2\}$, and after dropping S_2 the only superset left is S_1, so the parent of S_2 is S_1. Finally S_1 has no superset at all, so its parent is the universe.

The advantage of the inverted list is clearly shown in the running time. Without it, finding the parents means testing each set against every other set to see which ones contain it. With m sets that is on the order of m^2 subset tests, and each test has to compare membership across the universe, so the total work grows like $m^2 \cdot u$. For the three sets here that is $3 \times 2 = 6$ subset tests. With that being said it climbs extremely fast, a collection of one hundred sets already needs close to ten thousand of them. The inverted list replaces this inefficiency with a single pass that, for each universe element, notes which sets contain it, which costs on the order of $m \cdot u$. Each set's parent then comes

from intersecting the entries of its own elements, so the work scales with how many sets genuinely share elements rather than with the number of pairs. For the three sets this is just one pass through to build the list with three short lookups, quite a bit fewer than the six pairwise tests, and because m^2 grows far faster than m the gap widens rapidly as the collection gets bigger. For example, at one hundred sets the linear pass does on the order of one hundred units of work where the pairwise check does on the order of ten thousand.

Once every set has a parent, `contract_parents` shortens up the shape of the tree. Starting from a set's assigned parent it walks upward as long as the next ancestor is still no larger than twice the set's own size, and reassigns the parent to the highest ancestor that is within the bound. This stops the hierarchy from forming long thin chains and caps its depth at $O(\log u)$, while the size bound keeps the relative bitvectors from growing too large [1]. With the tree established, `build_relative_bitvectors` rewrites each set's bitvector so that its positions count only inside its parent, keeping a bit for each position where the parent has a 1 and dropping the rest. The sets are handled from smallest to largest so that, at the moment a child is rewritten, its parent is still expressed in terms of the universe and can be read correctly.

The pipeline finishes with `compress_all`, which packs each parent relative bitvector into the Baseline encoder, producing a compressed `Bitvector` for the universe and for every set. With the compact form, the scratch buffers are no longer needed and are freed, leaving only the finished hierarchy in memory.

Lastly, the construction pipeline for the construction of Containment Entropy using the Baseline encoding is as such:

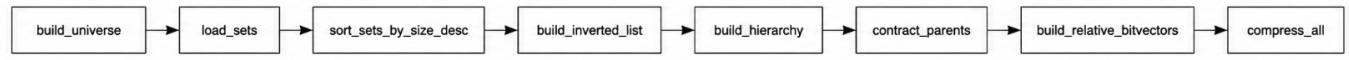

Figure 1: Construction Pipeline

3.3.4 Query operations

A set in the hierarchy stores its members only as positions inside its parent, so no single bitvector holds a query's answer directly. Every query therefore traverses the parent chain, using the constant-time Baseline rank and select at each level to translate a position from one set's coordinate system into the next [1, 5]. Two operations are provided: `retrieve`(S, k) returns the value of the k-th smallest element of S, and `rank_set`(S, k) returns the number of elements of S that are at most k. Unlike the Baseline examples above, where each query exercised a single operation in isolation, these two chain the Baseline's own `bitvector_select` and `bitvector_rank` once per level, so the queries are built directly on the same encoding rather than on a new one. They are shown below on the three nested sets $S_1 = \{1, 3, 7, 9, 10, 11, 12, 17, 18, 19\}$, $S_2 = \{3, 10, 12, 17, 19\}$ and $S_3 = \{3, 12, 19\}$, whose relative bitvectors are 1111111111 for the root S_1, 0100101101 for S_2 and 10101 for S_3.

Retrieve. `retrieve`(S, k) ascends the hierarchy. At a set that is not the root it calls `bitvector_select` on the set's own bitvector to map its k-th element to a position in the parent, then recurses one level up with that position. At the root the position is a universe coordinate, and a final `bitvector_select` on the universe bitvector returns the actual value. Each step is one $O(1)$ Baseline select.

Example: `retrieve`$(S_3, 2)$. The 2nd 1 of S_3's bitvector 10101 sits at position 3, so the search becomes the 3rd element of S_2. The 3rd 1 of 0100101101 sits at position 7, so it becomes the 7th element of the root S_1. The 7th 1 of 1111111111 is position 7, and a select on the universe returns its 7th smallest value:

$$\texttt{select}_{S_3}(2) = 3, \quad \texttt{select}_{S_2}(3) = 7, \quad \texttt{select}_{S_1}(7) = 7, \quad \texttt{select}_{\text{universe}}(7) = 12.$$

The universe values in order are $1, 3, 7, 9, 10, 11, 12, \ldots$, so the answer is 12, which is indeed the 2nd smallest member of $S_3 = \{3, 12, 19\}$.

Rank. `rank_set`(S, k) works in the opposite direction. It first climbs to the root carrying the value k unchanged, where a `bitvector_rank` on the universe converts k into the number of universe elements that are at most k. It then descends, and at each level a `bitvector_rank` on that set's own bitvector narrows the count handed down by its parent

to the count that falls inside the set. A safeguard returns 0 early if the cutoff ever lands before the set begins. Each step is one $O(1)$ Baseline rank.

Example: `rank_set`$(S_3, 12)$. At the root, ranking the universe counts the values at most 12, namely $1, 3, 7, 9, 10, 11, 12$, giving 7. Descending, S_1 ranks its bitvector 1111111111 over the first 7 positions and keeps 7. S_2 ranks 0100101101 over the first 7 positions, whose 1s sit at $2, 5, 7$, keeping 3, and S_3 ranks 10101 over the first 3 positions, whose 1s sit at $1, 3$, keeping 2:

$$\mathsf{rank}_{\text{universe}}(12) = 7, \quad \mathsf{rank}_{S_1}(7) = 7, \quad \mathsf{rank}_{S_2}(7) = 3, \quad \mathsf{rank}_{S_3}(3) = 2.$$

So two elements of S_3 are at most 12, which matches $\{3, 12\}$ directly.

In both operations every step is a single Baseline rank or select, each $O(1)$, and the number of steps equals the depth of the queried set in the hierarchy. Contraction caps that depth at $O(\log u)$, so a query runs in $O(\log u)$ time in the worst case rather than the flat $O(1)$ of the Baseline [1, 5]. This logarithmic factor is the price the containment representation pays for storing each set relative to a small parent instead of the full universe, and the experiments show whether the space saved is worth that cost.

3.3.5 Running, measuring and checking the experiment

The whole experiment for the containment program lives in a single function, `run_config`, which `main` calls once per configuration after one time setup. It generates a collection, builds both representations on that same collection, measures their space and speed, and checks every query answer against a known correct result, so the entire experiment can be followed from one place, just as in the Baseline program.

To make the space comparison fair, the function also builds the sole Baseline on the very same data. Each set is stored on its own as a compressed bitvector over the shared union universe, which is the same domain the hierarchy uses at its root, so the two space figures are measured against the same universe and are directly comparable. The space itself is reported as bits per element for both representations, using the same payload count as the Baseline section described, namely the class array plus the packed offsets and nothing else. For the hierarchy the function sums the payload of the universe bitvector together with the payload of every set bitvector, for the Baseline it sums the set of bitvectors independently. From these two totals it also prints the percentage of space saved.

Alongside space, the run reports the shape and the speed of the structure. Hierarchy depth is summarised as the maximum and the mean number of nodes to the root, found simply by following parent pointers. Build time is measured with `clock_gettime` on the monotonic clock wrapped around the construction pipeline, from `load_sets` through `compress_all`, so it captures the whole cost of arranging the hierarchy and compressing every bitvector. The per-query cost of `retrieve` and `rank_set` is timed over many random queries and reported as an average, which gives more accurate data.

Finally, correctness is checked against a brute force pass through rather than trust in the code. Just before the bitvectors are rewritten to be parent-relative, while they still hold each set's true members in universe coordinates, the function records every set's element list with `record_brute_force`. Each `retrieve` answer is then compared against the recorded element it should return, and each `rank_set` answer against a direct count over the recorded members, and the run reports how many of each passed. Because the synthetic data is generated from a fixed seed, a failure is fully reproducible, which makes the checks useful while developing as well as a guarantee that the reported timings come from a structure that actually answers correctly.

3.4 Design choices

A handful of deliberate choices were made for the containment implementation. The first is to store every level of the hierarchy with the same shared Baseline bitvector rather than inventing a separate encoding for relative sets. This keeps the codebase small, since there is only one rank and select to write and test, and it lifts the per-level cost of a query to $O(1)$ for free [5]. The second is to build the hierarchy from the inverted list instead of testing every pair of sets, which, as shown earlier, replaces a number of comparisons that grows with the square of the collection size by a single linear pass and a few short lookups. The third is the choice of an insertion sort to order the sets by size. A faster sort would matter for very large collections, but with at most 128 sets the cost isn't that great. Additionally, the simple algorithm is easier to read and to be confident in. The last choice is to make contraction a switch rather than

fixed. This allows the experiment to build the hierarchy both with it and without it and measure its effect directly. Contraction costs a little extra space, because relative bitvectors against a larger parent are slightly denser, but in return it bounds the hierarchy depth and therefore the worst-case query time, which is what the structure is built around, so it is left on for every experiment except the one that isolates its effect [1].

3.5 Synthetic data generation

The test collections are produced inside the program itself, so no external data file is needed and the exact same data is rebuilt on every run. The randomness comes from a small seeded xorshift generator [7], and each configuration is run under a fixed set of seeds, which makes every collection reproducible from one run to the next. This matters both for debugging, since a failing case can always be recreated, and for fair measurement, since the Baseline and the hierarchy are always timed on the same data.

Two kinds of collection are generated, one favourable to the hierarchy and one not. `generate_nested` builds a deep chain, the first set is a random subset of the value range, and every later set is a random subset of the one above it, kept at between 93 and 98 percent of its parent's size. Because each set is by construction a subset of the previous one, the collection nests, giving the hierarchy something real to exploit. `generate_independent` instead draws every set directly from the value range at random, each with a size between a third and a half of the range, so any containment between two sets would only arise by chance. This second collection is the unfavourable case, where the hierarchy has almost no containment to compress against, and the two together bracket the behaviour of the structure from its best case to its worst.

3.6 Implementation limitations

There are two main limitations of the implementation. The first is scale, and it follows from the decision to allocate memory statically. The fixed array sizes cap the test data at a largest universe of `MAX_ELEMENT_VALUE = 1024` and a largest collection of `MAX_SETS = 128`, so the experiments speak to this range rather than to arbitrarily large inputs. The second is data. The collections are synthetic and generated to bracket the two extremes, a deeply nested chain and a set of independent random sets, rather than drawn from a real workload. They are enough to show where the hierarchy helps and where it does not, but not how the structure behaves on real collections, whose containment is likely to be somewhere in the middle of the two extremes.

4 Experimental Setup

This section describes the setup used in the experiments that compare the Baseline and containment entropy approaches. The aim is to document the environment and parameters clearly enough that the results can be reproduced.

4.1 Hardware and software

All experiments were run on a 2022 MacBook Air (model Mac14,2) with an Apple M2 chip and 8 GB of memory. The M2 has eight CPU cores, four performance cores and four efficiency cores, where the performance cores run at up to 3.49 GHz. The 8 GB of unified memory is LPDDR5 clocked at 6400 MT/s, which gives about 100 GB/s of memory bandwidth.

The machine ran macOS Tahoe 26.2 (build 25C56). The C code was compiled with Apple Clang 15.0.0 (clang-1500.3.9.4), which is the compiler that the `gcc` command invokes on macOS. The build commands use no explicit optimisation, so the compiler's default settings were used.

4.2 Datasets

Every collection in the experiments is synthetic, which lets the universe size, the set count and the set densities be fixed exactly. `CEB.c` produces the data on the fly at the start of each run, so there is nothing to download or preprocess beforehand. Each collection holds 64 sets over the value range $\{1, \ldots, 512\}$, and two contrasting shapes are used throughout.

The nested shape, built by `generate_nested`, is a single chain. Its first set takes 480 of the 512 values at random, and each set after it is a random subset of it sized at 93 to 98 percent of the previous set. Every set therefore lies inside the one before it, so containment is dense from top to bottom, which is the favourable case the hierarchy is designed to take advantage of. The independent shape, built by `generate_independent`, takes the opposite approach. Each of its sets is drawn straight from the value range with a size between a third and a half of it, so two sets share elements only by accident. With almost no containment to compress against, this is the unfavourable case. The two cases make for the extremes on both sides, from a best case rich in containment to a worst case with essentially none.

A seeded xorshift generator creates the randomness [7]. Each configuration is repeated across the eight seeds $\{1, 7, 42, 101, 1337, 2024, 90210, 555\}$, so re-running the code reproduces the identical collections and the reported figures can be averaged over those seeds. Doing a pass through of the block size over $\{4, 8, 16, 32, 64\}$ and switching contraction off and on then gives 2 datasets $\times$ 8 seeds $\times$ 5 block sizes $\times$ 2 contraction settings = 160 different configurations in total.

4.3 Metrics

Four quantities are measured. The first is space, reported as bits per element which is the total number of bits used by the stored structure divided by the total number of 1-bits across all sets. This allows the terms of different sizes to be directly comparable.

The second is hierarchy depth. It is recorded both as the maximum over all sets and as the mean. The maximum bounds the worst-case query cost, since a query walks from a set up to the root, while the mean gives a sense of the typical cost. Comparing the depth before and after contraction shows how much contraction impacts the hierarchy and query operations.

The third is build time, which is the time to construct the structure. For the containment variation, this covers the construction pipeline from `load_sets` through `compress_all`.

The fourth is query latency, the average time to answer a single `retrieve` or `rank_set` query. Query time is averaged over many random queries so that the per-query figure is accurate, relevant and trustworthy.

4.4 Procedure

For each of the 160 configurations both structures are built from the same input, so that any difference in space or speed is not due to differing data. Timing uses `clock_gettime` with the `CLOCK_MONOTONIC` clock, which is not affected by adjustments to the system wall clock and is therefore suitable for measuring the short durations recorded in these experiments. Each query figure is the average over the many queries answered in that configuration, and every configuration is then run under all eight seeds, so each number reported in the results is the mean across those seeds with its standard deviation shown as an error bar.

Correctness is checked alongside the timing. Just before each set's bitvector is rewritten to be parent-relative, while it still holds the set's elements directly, `record_brute_force` records those members, and `make_queries` turns that record into the expected answer for every query. Every answer returned by the Baseline and by the containment hierarchy, for both `retrieve` and `rank_set`, must be fully identical with its expected value. Only after this check passes are the timing figures considered correct and can have an impact.

4.5 Reproducibility

All code is available in the repository at `https://github.com/sebgt11/Bachelor`, so that the entire pipeline can be repeated from scratch. The experiment is compiled and run with `gcc -o CEB CEB.c` followed by `./CEB`, which generates the data in memory, runs every configuration, and writes the raw numbers to `results.csv`. The figures and the summary table are then produced by `python3 plot_results.py`, which reads that file. The sole Baseline is also present with a demo, run using `gcc Baseline.c -o Baseline` and `./Baseline`. The same goes for a `CEBTemp.c` where it is to demonstrate a simple version of the containment entropy with Baseline encoding rather than data collection. It is compiled and run with `gcc -o CEBTemp CEBTemp.c` followed by `./CEBTemp`. With all these steps, the code and results should be reproducible and simple to understand, adding that the seeds also provide the same outputs.

5 Results

All numbers below come from the same experiment, two synthetic collections are used throughout. The first is nested, it is a deep chain of 64 sets where each set is a near-copy of the one above it, so containment is everywhere and the structure should shine. The second, called independent, is 64 random sets of similar size with no deliberate containment, and it is meant to be the unfavourable case where the hierarchy buys nothing. The value range is $\{1, \ldots, 512\}$, the block size is fixed at $b = 16$, and contraction is on. Every figure is averaged over the eight seeds $\{1, 7, 42, 101, 1337, 2024, 90210, 555\}$ and the error bars are the standard deviation across those seeds. Each of the 160 configurations was checked against a brute-force reference and all of them answered every `retrieve` and `rank_set` query correctly, so the timings that follow come from a structure that is known to be correct.

5.1 Space against the entropy bounds

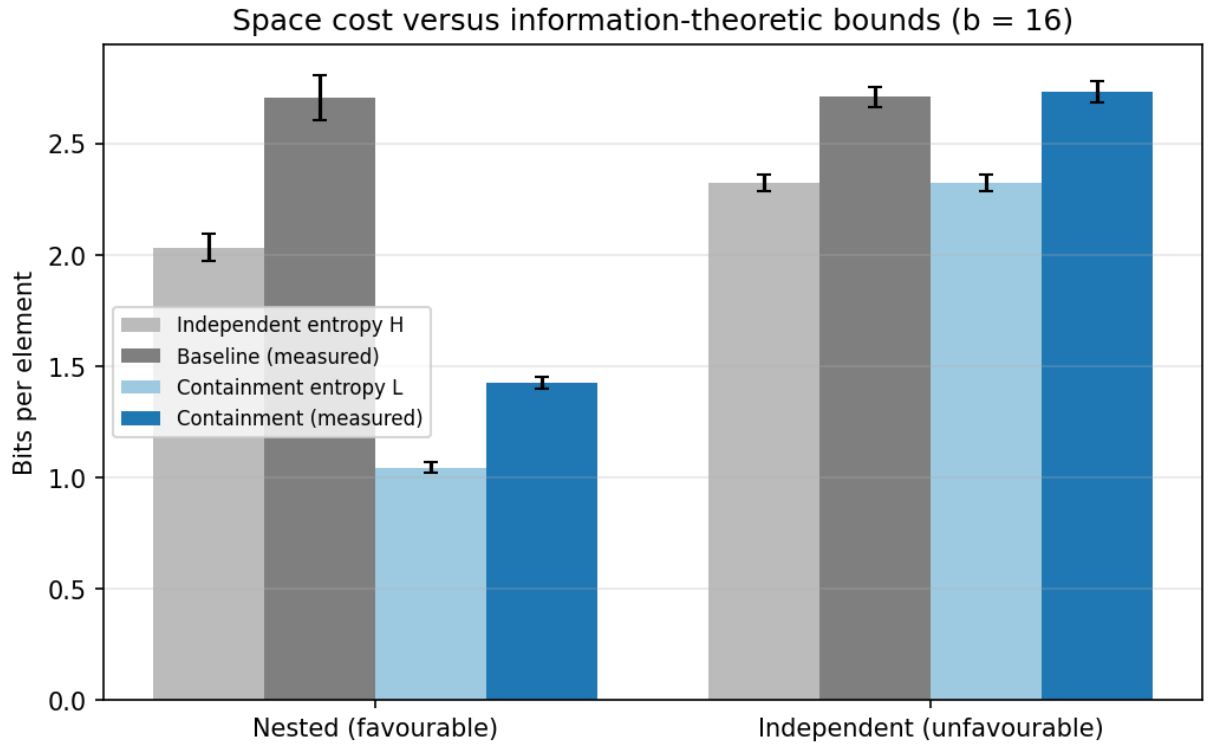

Figure 2: Measured bits per element for the Baseline and the containment hierarchy, shown next to the independent entropy $H(\mathcal{S})$ and the containment entropy $L(\mathcal{S})$, on both datasets at $b = 16$.

Figure 2 shows two dataset groups, where each group is made up of four bars: the worst case entropy H, the measured Baseline, the theoretical containment entropy L, and lastly the measured containment hierarchy. The two groups are differentiated by their datasets, one being favourable and the other unfavourable, and the difference lies in how each dataset is structured. The favourable dataset is made up of highly similar sets, meaning that many sets have subsets and/or supersets, which allows for the exploitation of the containment entropy structure. On the other hand, the independent dataset, which is largely unfavourable, is made of very distinct sets, disallowing the utilisation of the containment entropy efficiency among similar sets.

On the nested data the two theoretical bounds can be seen to be far apart, where $H = 2.03$ bits per element while $L = 1.04$. This means that the containment entropy is roughly half the independent entropy when the sets nest within one another. To further support this, it can be seen that the measured bars on the nested data are Baseline 2.71 and containment 1.43 bits per element, which is a 47% reduction in space from using the hierarchical structure.

Each measured bar sits a little above its own entropy bound. The Baseline at 2.71 is about 0.7 bits above H, and the containment at 1.43 is about 0.4 bits above L. This gap is not a failure of the bound but the fixed overhead of the block coding, namely the per block class together with the checkpoint samples that keep the queries fast. The ordering on the nested data is then exactly what the theory predicts, $L\,(1.04) <$ containment $(1.43) < H\,(2.03) <$ Baseline (2.71), with the measured containment cost landing neatly between the two bounds.

The independent data tells the opposite story. Here the two bounds coincide at $H = L = 2.32$ bits per element, because with no containment to exploit the containment entropy falls back onto the independent entropy. The measured bars reflect this directly, Baseline 2.71 and containment 2.73, so the hierarchy is about 0.9% larger than the Baseline on this data. That small loss is the price of the parent pointers and the relative bitvectors. Across both datasets the error bars are small, only a few hundredths of a bit per element, so both the wide separation on the nested data and the near tie on the independent data are stable across all eight seeds.

5.2 Space including the query index

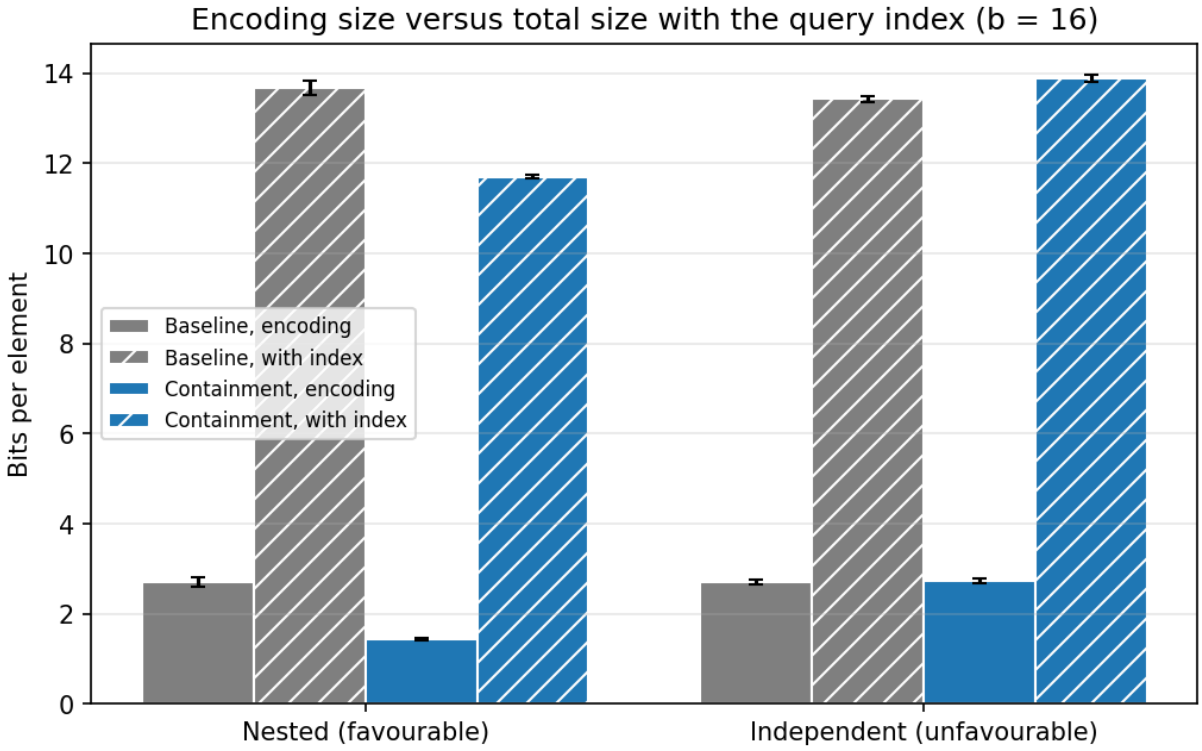

Figure 3: Encoding size next to total size with the rank and select index counted, for the Baseline and the containment hierarchy, on both datasets at $b = 16$.

Figure 2 counts only the encoding itself, the class array and the packed offsets, which is the part that the entropy bounds describe. The structure also keeps a query index, namely the rank checkpoints R, the offset pointers P and the select array S, which is needed to answer queries at the speeds reported later but carries no information beyond what the encoding already holds. This figure adds that index back in. For each dataset it shows four bars, the Baseline and the containment hierarchy each given twice, once as the bare encoding and once with the index counted at the same packed bit widths used for the data, so the realistic memory can be compared directly against the bare encoding.

The first thing the figure makes clear is that the index makes up most of the data. On the nested data the Baseline encoding is 2.71 bits per element but its total rises to 13.68, and the containment encoding is 1.43 but its total rises to 11.70, so in both cases the index adds roughly 10 to 11 bits per element on top of the encoding. Almost all of that is the select array, which stores one full position for every element of every set, about nine bits each over a value

range of 512, with the rank and offset checkpoints adding only a little more. This is a deliberately simple select, a single array lookup, and a compact select index would shrink this term substantially. The figure therefore shows the memory of this particular implementation rather than a lower bound on what a general indexed structure costs.

With the index included the containment hierarchy still uses less total space than the Baseline on the nested data, 11.70 against 13.68 bits per element, but the saving falls from the 47% seen on the encoding alone to about 14%. The reason is that the index is close to the same size for both methods, so it dilutes the data saving. Interestingly, the hierarchy's index is even slightly smaller per element than the Baseline's, because its relative bitvectors index positions inside small parents, which need fewer bits than positions inside the full universe.

On the independent data the picture matches the encoding only result but is slightly worse. The Baseline total is 13.43 bits per element and the containment total is 13.88, so the hierarchy is about 3.4% larger once the index is counted, against the 0.9% it was larger on the encoding alone. With no containment to exploit the relative bitvectors save nothing, and the hierarchy still has to carry the extra universe bitvector and its index, so the small loss on the unfavourable data grows slightly rather than decreasing. The error bars are small throughout indicating that the losses are stable across the eight seeds.

The conclusion is that the containment hierarchy's space advantage is genuine but it lives in the encoding, which is the part the entropy bounds speak to. The heavy and unoptimised select index is added, which dilutes the advantage to a 14% total saving on favourable data and a small loss on unfavourable data, because the index is a shared cost that neither method avoids. A different select index would potentially let more of the encoding saving show through in the total, that would be left for future work.

5.3 Effect of parent contraction

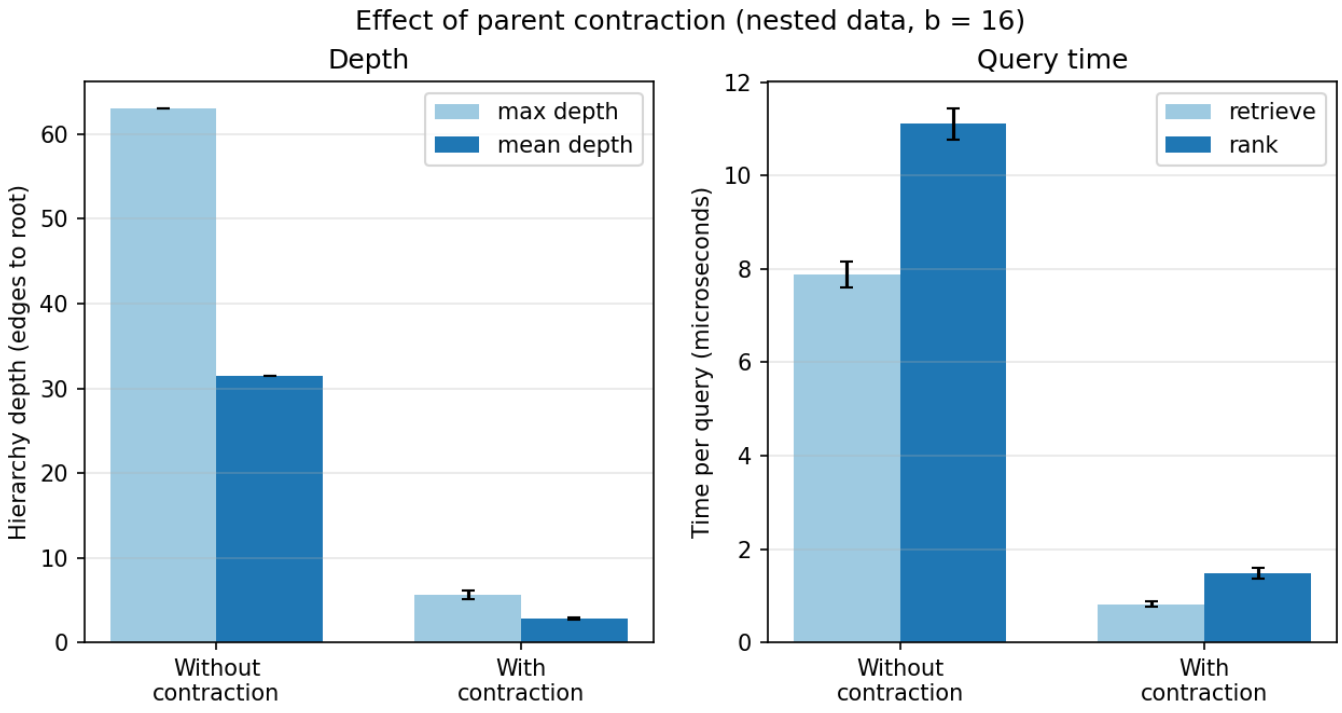

Figure 4: Hierarchy depth (left) and per-query time (right) on the nested data with contraction off and on, at $b = 16$.

The left graph of the figure reports the hierarchy depth, given as both the maximum and the mean number of edges from a set up to the root, while the right panel reports the average time of a `retrieve` and a `rank_set` query. Both graphs are shown twice, once with contraction off and once with it on, so the effect of contraction can be read and compared directly.

Without contraction the nested data forms a single long chain, since every set is attached straight to its smallest superset, which is the set directly above it. The maximum depth is therefore 63 and the mean depth is 31.5, meaning the deepest set sits 63 levels from the root and the average set sits about halfway down the tree. The maximum depth has zero spread across the seeds, always exactly 63, because the length of the chain is fixed by the number of sets rather than by the random seed due to the nested generation logic. Turning contraction on changes the picture quite a bit, the maximum depth falls to 5.6 and the mean depth to 2.83. Capping a parent at twice the child's size flattens the deep tree into a more shallow tree.

This flattening can be directly seen in the query time graph. Without contraction the queries are slow because they have to walk the whole chain, so a retrieve takes 7.88 μs and a rank takes 11.10 μs. With contraction the same queries take 0.82 μs to retrieve and 1.48 μs to rank, which is roughly 10× faster for retrieve and about 7× faster for rank. In both, rank is the slower of the two, 11.10 against 7.88 μs without contraction and 1.48 against 0.82 μs with it, because rank performs a `bitvector_rank` decode at every level it visits whereas retrieve resolves each level through the constant time select lookup. The depth bars carry almost no error, since the maximum depth is fixed and the mean depth varies by only about 0.13, while the time bars carry a little more spread on the long uncontracted walks, where the rank standard deviation is about 0.33 μs from hardware performance.

However, the improvement is not free and it comes at a cost. The matching space saving falls from 75% without contraction down to 47% with it, because a relative bitvector measured against a larger contracted parent is denser and therefore costs more bits. The shorter depth, and the faster queries that come with it, are therefore a trade-off with a decent amount of extra space, which is exactly the trade contraction is designed to make.

5.4 Query time against the Baseline

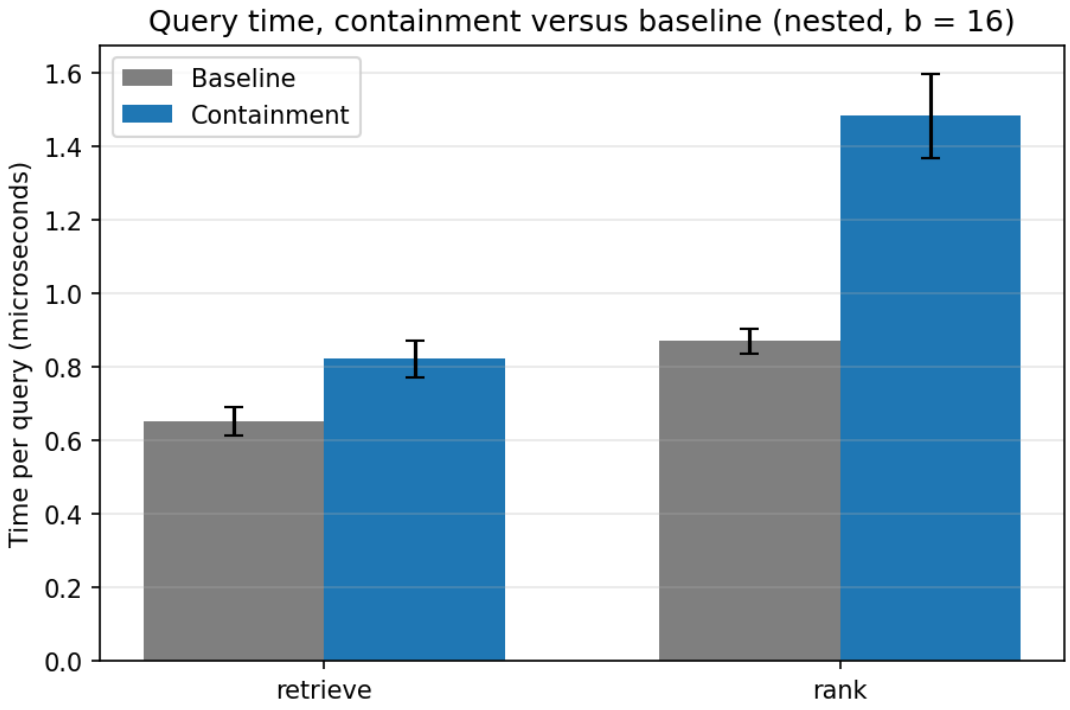

Figure 5: Per-query time of the containment hierarchy and the Baseline on the nested data, at $b = 16$ with contraction on.

This figure groups the two operations, retrieve and rank, and shows a Baseline bar next to a containment bar for each, so the cost of the hierarchy can be compared against the Baseline directly on the same data.

On retrieve the Baseline takes 0.65 μs while the containment hierarchy takes 0.82 μs, making the hierarchy about 1.3×

slower. On rank the Baseline takes 0.87 μs and the hierarchy 1.48 μs, a gap of about 1.7×. So even with contraction turned on the containment structure is slower than the Baseline on both operations. This however is explainable logically, the Baseline answers every query from a single bitvector at depth 1, whereas the hierarchy still has to walk the mean 2.83 levels of the contracted tree before it reaches the universe. Within each, rank again costs a little more than retrieve, 1.48 against 0.82 μs for the containment structure and 0.87 against 0.65 μs for the Baseline, this is the result of the same decode versus lookup seen in the last section.

It is worth keeping in mind the scale of this slowdown. All of the absolute times lie between about 0.65 and 1.5 μs, so the hierarchy's cost is a small constant factor on queries that are already very fast. The containment bars sit clearly above the Baseline bars for both operations and the error bars do not close that gap, so the slowdown is consistent across all eight seeds.

5.5 Construction time

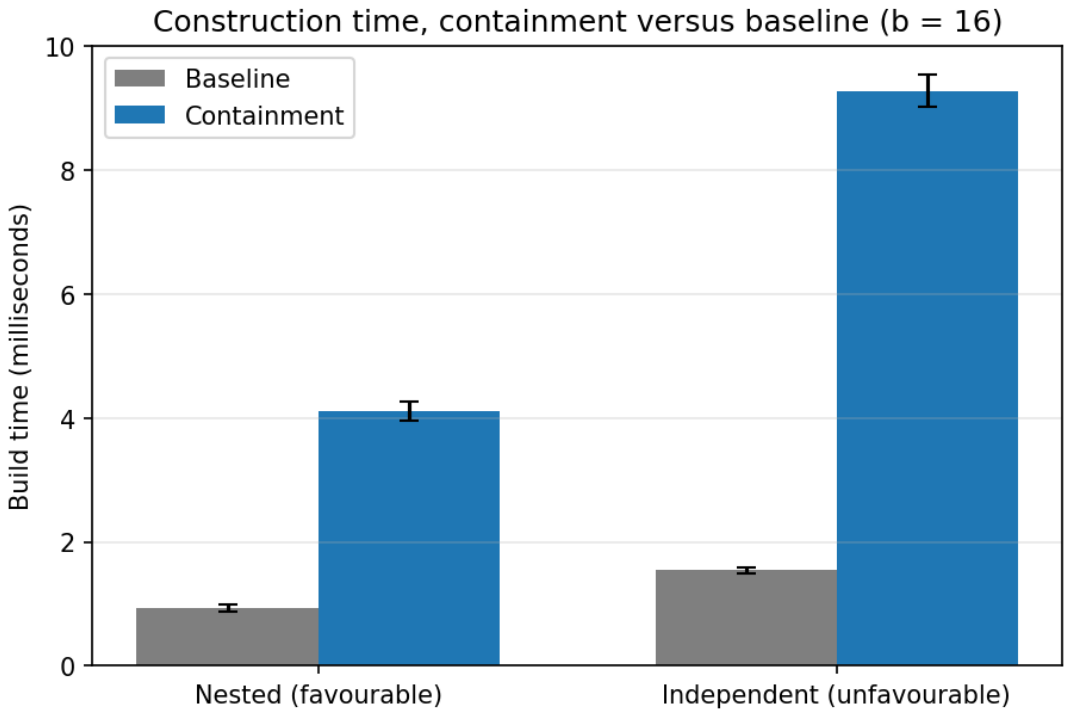

Figure 6: Construction time of the containment hierarchy and the Baseline on both datasets, at $b = 16$ with contraction on.

This figure groups the two datasets and shows a Baseline build bar next to a containment build bar for each, measuring the one off or pre-compute cost of constructing each representation.

On the nested data the Baseline builds in 0.93 ms and the containment hierarchy in 4.12 ms, so the hierarchy costs about 4.4× as much to build. On the independent data the gap widens sharply, with the Baseline at 1.54 ms and the hierarchy at 9.28 ms, about 6× as much. The hierarchy is more expensive on both datasets because it does a great deal of extra work that the Baseline never does, it builds the inverted list, searches for each set's smallest superset, contracts the parents, and rewrites every bitvector relative to its parent, all on top of the same compression that the Baseline also performs.

The reason the penalty is so much larger on the independent data, 9.28 ms against 4.12 ms, comes down to density. The independent sets are dense, with sizes between about a third and a half of the universe, so the inverted list intersection that drives the hierarchy search touches many more set element pairs, and it does all of that work and results in subset pairs. The Baseline alone is also slower on the independent data than on the nested data, 1.54 against

0.93 ms, for the same density reason, but it is the gap between the two methods that really widens on the unfavourable data. Even so, every build finishes in under 10 ms for 64 sets, so construction is cheap in absolute terms. The figure is about the relative cost of arranging the hierarchy, a cost that is paid once at build time and never again at query time. The largest spread is on the containment build for the independent data, with a standard deviation of about 0.25 ms, while the other three bars are tighter.

5.6 Effect of the block size

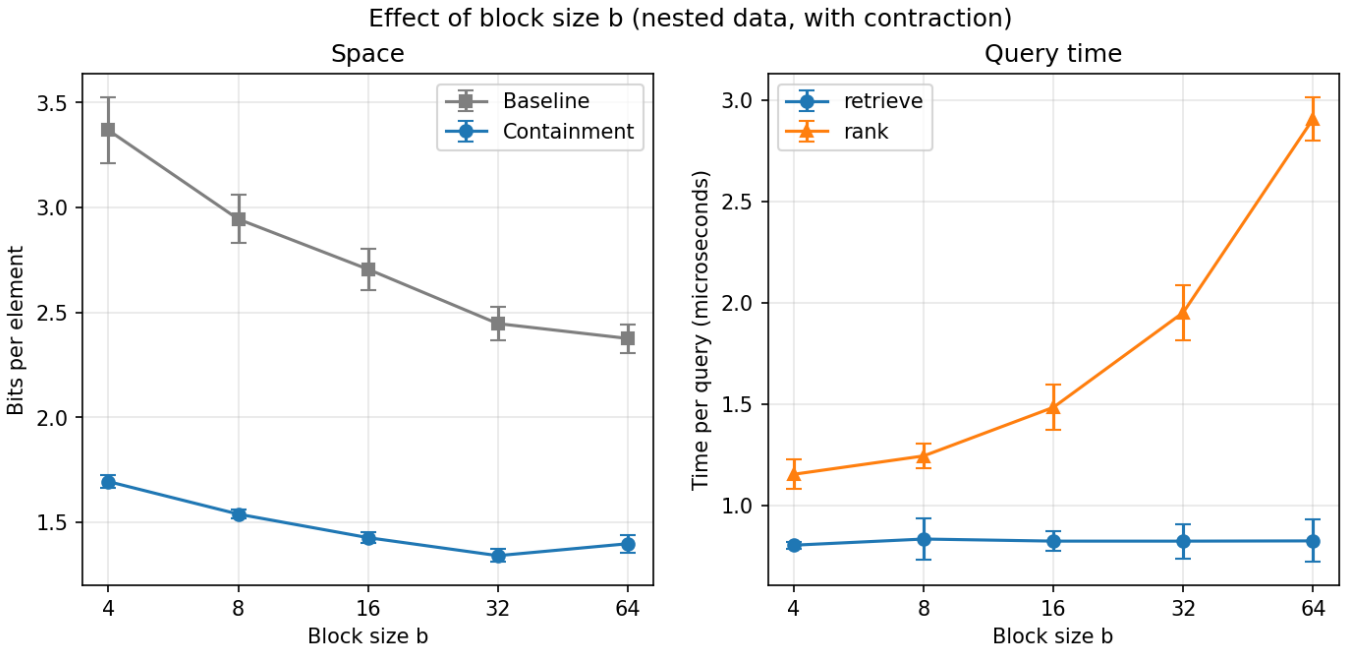

Figure 7: Space (left) and per-query time (right) of the containment hierarchy as the block size b varies, on the nested data with contraction on.

The final figure isolates the effect of the block size b. The left graph plots bits per element against the block size for both the Baseline and the containment hierarchy, while the right panel plots retrieve and rank time against the same block size, with $b \in \{4, 8, 16, 32, 64\}$ drawn on a logarithmic axis.

Both space curves fall as b grows, because a larger block packs the class and offset of each element more compactly. For the containment hierarchy the space goes 1.69 at $b = 4$, 1.54 at $b = 8$, 1.43 at $b = 16$, 1.34 at $b = 32$, and 1.40 at $b = 64$ bits per element, so it reaches its minimum at $b = 32$ and then ticks slightly back up at $b = 64$. The Baseline space goes 3.37, 2.95, 2.71, 2.45 and 2.38 bits per element, decreasing across the whole range without turning back up. Throughout, the containment curve sits roughly 1.0 to 1.7 bits per element below the Baseline, so the space advantage of the hierarchy holds at every block size tested.

The query times behave very differently from one another. Retrieve time is essentially flat, between about 0.80 and 0.83 μs for every block size, because retrieve resolves through select, which is a single array lookup and does not depend on the block size at all. Rank time, on the other hand, climbs steeply with the block size, going 1.15 at $b = 4$, 1.24 at $b = 8$, 1.48 at $b = 16$, 1.95 at $b = 32$ and 2.91 μs at $b = 64$, more than doubling from the smallest block to the largest. Rank climbs because every call decodes a full b-bit block and reads up to b bits, so its per call work grows with b, exactly the dependence previously mentioned in the implementation chapter.

The two graphs therefore pull in opposite directions, space wants a large block, with its best value at $b = 32$, while rank time wants a small block. In other words, $b = 32$ is the space-optimal block size in these experiments, but not the best overall operating point once query time is considered. Around $b = 16$ the structure keeps almost all of the space saving, 1.43 bits per element against the 1.34 minimum, while holding rank near 1.5 μs, quite a bit under the

2.91 μs seen at $b = 64$. This balance between the two trade-offs is what motivated the block size of $b = 16$ for every other experiment in this chapter.

6 Discussion

6.1 Answering the research questions

The first question was whether the containment entropy $L(\mathcal{S})$ offers a measurable improvement over the independent cost $H(\mathcal{S})$ on representative data, and not just as a worst-case bound. The answer is yes on hierarchical data and no on disjoint data. On the nested collection the measured containment entropy is about 1.04 bits per element against an independent cost of about 2.03, so the bound is roughly halved, and the actual encoding follows it down, 1.43 bits per element for the hierarchy against 2.71 for the Baseline, resulting in a saving of 47%. On the independent collection the two bounds are closer at about 2.32, and the measured sizes match that, 2.73 against 2.71, a loss of under one percent. So $L \leq H$ always holds, and the gap is largest exactly where containment is densest, which is the behaviour the theory predicts [1]. Figure 2 makes this visible, the encoded size tracks its entropy bound on both datasets, so the saving is set by how much containment the data holds rather than by the encoding.

The second question was whether the contracted hierarchy shortens retrieval in practice, whether bounding the depth means faster queries. Here the two move together. Without contraction the nested hierarchy is a single chain of maximum depth 63 and mean depth 31.5, with contraction the maximum drops to about 5.6 and the mean to 2.83, comfortably under the $O(\log u) \approx 9$ ceiling [1]. Query time follows the same path, uncontracted, a retrieve and a rank cost about 7.88 and 11.10 microseconds, while contracted they cost 0.82 and 1.48, a definitive improvement. Bounding the depth is therefore a key part in making this data structure viable. Figure 4 puts the two side by side, where the drop in depth and the drop in query time line up almost exactly, confirming that depth is what drives the query cost.

The third question was how the implementation compares, in both space and query time, against a Baseline built from a single standalone ranked bitvector. The containment hierarchy wins on space on hierarchical data but pays for it in time. It is about 1.3 to 1.7 times slower per query than the Baseline, 0.82 and 1.48 microseconds against 0.65 and 0.87, and it is 4.4 to 6 times slower to build, 4.12 milliseconds against 0.93 on the nested data and 9.28 against 1.54 on the independent data, because it walks a parent chain on every query and constructs a hierarchy the Baseline never forms. Because the trade-off is between space and speed, the better choice is largely determined by how nested the data is. Figures 5 and 6 show the price of that win, the hierarchy sits above the Baseline on both, so it only pays off when the space saved on nested data outweighs the speed given up.

6.2 When containment helps

Containment helps whenever sets are nested. In that case each child is cheap to describe relative to its parent and L falls well below H. The nested collection is the clear illustration, using 47% less space than the Baseline with contraction on and 75% less with it off. The benefit grows with depth and how the child relates to its parent. In the synthetic data generator, every child keeps between 93 and 98% of its parent, but a shallower or looser real hierarchy would still be expected to benefit, just by less.

The structure this exploits appears naturally in several kinds of data. Versioned or append-only collections, where each revision is the previous one plus a few changes, are nested by construction. The same goes for category trees, where a child label's members are a subset of its parent's. In all of these the containment the hierarchy needs is present which is what makes them the natural targets for this representation.

6.3 When containment does not help

Containment does not help when sets are mostly disjoint or independent, because then any containment is accidental, the hierarchy becomes a flat list, and L falls back to the worst-case entropy H. Additionally, the indexing and extra overhead makes the containment entropy slightly worse than H. The hierarchy spends effort organising sets that have nothing in common with no reward. Even on data it suits, the structure is the wrong choice when queries dominate and space is cheap, since it is consistently slower per query and slower to build than the standalone Baseline. For a read-heavy workload on a collection that fits comfortably in memory, the simpler Baseline is the better engineering choice.

Moreover, once the shared query index is counted rather than the encoding alone, the nested space saving shrinks from 47% to 14%, because the select array, which stores one full position per one-bit, is a fixed per-element cost that the parent-relative structure does not reduce. The encoding saving is genuine, and it is what the entropy theory is about. But a real deployment also has to store the index, and both methods pay that same index cost, so once it is counted some of the saving disappears.

6.4 Cost of contraction

Contraction gives up a little space to make the hierarchy much shallower and faster, and on nested data that trade is worth it. With contraction off the nested encoding is smaller, 0.68 against 1.43 bits per element, because each set is then stored against its immediate parent rather than a larger ancestor, so the relative sets are tighter. The uncontracted chain is 63 levels deep and its queries take seven to ten times as long. The roughly 0.75 extra bits per element that contraction costs are the price of a structure that is fast to query. That overhead is small and bounded, and the build-time difference between the two is tiny next to the gap between the hierarchy and the Baseline. It could be argued that the real cost of contraction is the encoding it gives up, not the time it adds.

6.5 Design choices and alternatives considered

A few decisions shaped the implementation. Every node in the hierarchy reuses the same compressed-bitvector code as the Baseline. This keeps the codebase small and gives constant-time rank and select at each level instead of needing a separate format per node. Select is from a stored array of positions rather than by scanning blocks, which is the simplest version that works. It is also the largest part of the query index, so a sparse or sampled select is the obvious way to win back the space the total-size comparison loses, and that is left as future work. The parent links are built from an inverted index, which avoids the slow $O(s^2 \cdot u)$ check of every pair of sets, and the sets are ordered with an insertion sort, which is cheap at these sizes and easy to check.

Two further choices are worth naming. Contraction is a switch that the experiment sweeps rather than a fixed setting, so both the contracted and uncontracted hierarchies are built and measured, which is what makes the trade-off between depth and encoding above visible. Contraction is the recommended default, since the uncontracted version, though smaller, is too deep to query efficiently [1]. The block encoding spends $\lg \binom{b}{c}$ bits per block, this size approaches the entropy bound and the same one used in the paper [5], and it is this choice that lets the measured bits per element be compared directly against L and H. Finally, the data is synthetic and built in memory, which gives full control over the hierarchy's depth and density and makes every run reproducible. All of these choices allow for the simple implementation seen in this paper.

6.6 Threats to validity

The synthetic data is the main threat. The nested and independent collections are deliberately a best case and a worst case, and real data would sit somewhere between the two, so the 47% saving should be read as the top of a range rather than a typical figure.

The reported space is the logical size of the encoding, that is, the class and offset bits the input actually needs, not how much memory the program uses. Because the arrays are allocated at fixed upper bounds, the program's memory does not shrink with the data, and the containment variation even uses more structs than the Baseline because it also stores the universe. So the saving shows up in the size of the representation, which is what the entropy theory is about, and not in the program's memory use.

The encoding only and total size comparison are both reported to differentiate between what is actual encoding of the original data and what is used to make queries faster. The encoding figure measures what the entropy theory predicts, while the total figure, which also counts the rank, offset position and select index, measures what a real deployment would actually store. The summary is that the hierarchy's advantage is genuine at the level the theory addresses but is diluted by a shared index that this work did not set out to shrink. Lastly, all timings come from one machine and a dataset small enough to fit in cache, so the absolute microsecond numbers will not carry over to other setups. What stays stable is the ratio between the methods, and it is those ratios that the conclusions are based upon.

7 Conclusion and Future Work

This thesis set out to implement and evaluate the containment entropy data structure and to measure it against a straightforward Baseline. Both were written in C and share the same compressed bitvector encoding, so any difference between them comes from the hierarchy and not from the way the bitvectors are stored underneath. The Baseline keeps every set on its own as a single compressed bitvector, while the containment variation arranges the sets into a contracted hierarchy and stores each one relative to a parent that contains it. Running both over the same synthetic collections, across eight seeds, five block sizes and the two contraction settings, gave a consistent illustration of where the idea pays off and where it does not.

The central claim was tested and it held up. The containment entropy $L(\mathcal{S})$ was never larger than the worst-case entropy $H(\mathcal{S})$ on any dataset tested, and on hierarchical data the gap was substantial. On the nested collection L came to about 1.04 bits per element against an H of about 2.03, roughly half, and the measured encoding tracked that prediction down to 1.43 bits per element for the hierarchy against 2.71 for the Baseline, a saving of 47%. On the independent collection the two bounds met at about 2.32, the measured sizes broke even, and the hierarchy carried a small overhead instead of a saving. The structure therefore delivers the actual theorised savings exactly where containment is present and costs a little where it is not.

Contraction also worked as expected. With an uncontracted structure the nested hierarchy formed a single chain of depth 63, but with contraction the maximum depth dropped to about 5.6 and the mean to 2.83, both definitely under the $O(\log u) \approx 9$ ceiling. Furthermore, it cut a retrieve query from about 7.88 to 0.82 microseconds and a rank query from 11.10 to 1.48, which is what makes the structure usable for queries at all. Set against the Baseline, the containment hierarchy is smaller on hierarchical data but the slower one, about 1.3 to 1.7 times slower per query and several times slower to build, so the comparison comes down to a trade of space for speed where the outcome depends on how nested the data is.

The saving is the size of the encoding itself, namely the class and offset data. Once the shared query index is counted as well, and in particular the select array that stores one position per one-bit, the nested saving shrinks from 47% to about 14%. This once again goes to dilute the advantage. Nonetheless, there is an advantage, and shrinking the index is the clearest opening for further work.

7.1 Future work

The most direct next step is to evaluate the structure on real nested data rather than synthetic best and worst cases, for example genomic indexes, web-graph adjacency lists, or versioned document collections, where the containment the hierarchy needs arises by design. The second is to replace the explicit select array with a sampled or sparse select, which is the single largest part of the query index and therefore the obvious place to recover the space the total-size comparison gives back. Together, these would give a most realistic insight into how well the solutions would perform against one another.

8 Use of AI

This section documents where AI tools were used in the course of the project.

For the writing, AI was used as a light editing aid. It helped tighten loose sentences, correct punctuation, and point out places where an explanation was unclear.

For the main C code, the use was minimal. The implementation was written by myself, and AI was mainly used when tracking down bugs and explaining why they were happening, rather than for writing the core data structures.

In terms of coding, AI had a heavier influence in two supporting areas. The first was the plotting code that turns the raw results into the figures in this report, where it assisted with creating the graphs and the practicalities of handling CSV files. The second was the synthetic data generators, where it helped build the functions that create the nested and independent collections used in the experiments and write them out to CSV files.

References

[1] Jarno N. Alanko, Philip Bille, Inge Li Gørtz, Gonzalo Navarro, and Simon J. Puglisi. Compact Data Structures for Collections of Sets. Open Access Series in Informatics (OASIcs), 2025. https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.Grossi.6.

[2] Paolo Boldi and Sebastiano Vigna. The WebGraph Framework I: Compression Techniques. https://dl.acm.org/doi/epdf/10.1145/988672.988752, 2004. Accessed: June 2026.

[3] GeeksforGeeks. Insertion Sort Algorithm. https://www.geeksforgeeks.org/dsa/insertion-sort-algorithm/, 2026. Accessed: June 2026.

[4] GeeksforGeeks. Inverted Index. https://www.geeksforgeeks.org/dbms/inverted-index/, 2026. Accessed: June 2026.

[5] Gonzalo Navarro. *Compact Data Structures: A Practical Approach*. Cambridge University Press, Cambridge, United Kingdom, 2016. ISBN: 978-1-107-15238-0. https://www.cambridge.org/core/books/compact-data-structures/68A5983E6F1176181291E235D0B7EB44.

[6] Daisuke Okanohara and Kunihiko Sadakane. Practical Entropy-Compressed Rank/Select Dictionary. https://arxiv.org/pdf/cs/0610001, 2007. Accessed: June 2026.

[7] Wikipedia. Xorshift. https://en.wikipedia.org/wiki/Xorshift, 2026. Accessed: June 2026.