technische universität
dortmund

Final Report

PlaDs - Platzeffiziente Datenstrukturen (Succinct Data Structures)

Project group 628
October 25, 2020

Supervision:
Prof. Dr. Johannes Fischer
M. Sc. Patrick Dinklage
M. Sc. Jonas Ellert

TU Dortmund University
Department of Computer Science
Algorithm Engineering (11)
http://ls11-www.cs.tu-dortmund.de

Contents

Chapter 1

Introduction

The optimisation of data structures and algorithms regarding their time and space requirements has always been a crucial part in the work of a computer scientist. In 1974, Donald Knuth stated in [Knu74]:

> "In established engineering disciplines a 12 % improvement, easily obtained, is never considered marginal; and I believe the same viewpoint should prevail in software engineering. Of course I wouldn't bother making such optimizations on a one-shot job, but when it's a question of preparing quality programs, I don't want to restrict myself to tools that deny me such efficiencies."

Early computers showed huge limitations in speed of operations and available memory. Therefore, it was necessary to work with as little space as possible to realize any computation. Modern computers work much faster and can use much more space for computation, but optimising running time and memory usage is still important. Nowadays, computers often have to process huge amounts of data. For example, the processing of DNA produces sequences of characters requiring numerous gigabytes of space, but it should still be possible to analyse the genomes in reasonable time without needing significantly more additional space. Other huge data sets arise, for example, when analysing network activities.
Succinct data structures aim at storing such data sets in space near the information-theoretic lower bound and still allow time efficient operations. A simple example is the counting of set bits in a bit string up to a given position. When merely storing the bit string, this is possible in linear time. But we can find a way to answer this query in constant time with only requiring sublinear additional space (see Chapter 4). This basic data structure can then be used for further development of succinct data structures, for example representing a binary tree with n nodes requiring only $2n + o(n)$ bits of space.
One basic concept in programming is to re-use already implemented data structures and algorithms that have proven useful before. Especially during the implementation of space efficient data structures, it is necessary to use well implemented basic structures with low memory usage. Therefore, programmers often make use of libraries that contain efficient basic data structures and algorithms, and thus build a good foundation for further development. In this report, we describe our library in C++ for succinct data structures, including static and dynamic bit vectors, graphs, compressed strings and hash functions.

1.1 Project Goals

The project group is a mandatory module of the master program in computer science at TU Dortmund University. Our goal is to develop and implement a highly performant, extendable library of succinct data structures in C++. In detail, the library should

- contain data structures of different areas, including: bit vectors, graphs, strings, hash functions,
- contain data structures with space requirement near the theoretic lower bound that allow efficient operations,
- be easy to expand, such that additional data structures can be added in the future, and
- be benchmarked and evaluated thoroughly.

These goals should be achieved in two semesters. This report gives an overview of our results.

1.2 Overview

In Chapter 2, we introduce some definitions and notations that we use throughout the report. First, we explain basics about the computational model, strings, graphs and hash functions. Then, we describe some basic succinct data structures that we use repeatedly in our implementations, which are static bit vectors, AVL trees, wavelet trees, Huffman trees and gap encodings.
In Chapter 3, we introduce our library. We specify its content and structure, explain how to use it and describe the setting in which we execute our benchmarks.
In Chapters 4 to 8, we describe the data structures our library contains. For each chapter, we first describe the theoretical background of the implementations, specify the operations we support and analyse time and space requirements. Then, we describe our implementation and how it differs from the theoretical descriptions. At last, we evaluate our work by analysing our benchmarks, compare our data structures to other data structures that perform the same work and discuss opportunities to improve our implementations in the future.
In Chapter 4, we describe our implementation of static bit and integer vectors and some other structures that are basic components of our succinct data structures. Therefore, their practical time and space requirements are of utmost importance to the performance of our library.
A dynamic version of the bit vector is described in Chapter 5. Here, the content of the bit vector can change at any time so that a static support structure would need to be reconstructed regularly. Instead, we establish a tree structure that allows fast queries and fast updates.
In Chapter 6, we introduce a succinct way to store and answer some basic queries on strings. The idea is to find a grammar with special properties, a so-called straight line program, deriving only the given string. Such a grammar can then be stored space efficiently.
In Chapter 7, we describe representations for directed, unweighted graphs that support basic operations like neighbourhood queries. We analyse the GLOUDS representation which is optimised for tree-like

graphs.
The last data structure is a minimal perfect hash function, which we describe in Chapter 8. It allows fast document retrieval by utilizing fingerprints.
Finally, in Chapter 9, we summarise our results and discuss possible improvements of our library.

Chapter 2

Preliminaries

In this chapter, we introduce the basic definitions and notation that we use for descriptions of our succinct data structures. First, we cover the computational model, strings, graphs and hash functions. Afterwards, we describe bit vectors, entropy, AVL trees, wavelet trees, Huffman trees and gap encoding.

2.1 Computational Model

When discussing the space and time boundaries of data structures, we use the *word RAM* model as the basis of our results. In this model, calculations are carried out on computer words of w bits. Arithmetic instructions, such as additions, subtractions, divisions and multiplications may be performed on a word in constant time. Modulo operations (mod), as well as the floor and ceiling functions, may also be performed in constant time. The same also applies to the bitwise operations: the bitwise operations AND ($\wedge$), OR ($\vee$), XOR ($\oplus$) and the bitwise negation ($\overline{b}$) may all be performed in constant time [Nav16, p. 9]. Words are stored in a computer's random access memory. We can address any word in memory at any time and perform reading and writing in constant time. We assume that data structures are stored in RAM and are therefore accessible in their entirety. Given an arbitrary data structure of n elements, we assume $w = \Theta(\log n)$ [Nav16, p. 9]. In practice, w equals 64 bits on most modern computer architectures. Further details on the word RAM model can also be found in [CLRS09a, Sec. 2.2].

2.2 Strings

Let Σ be a finite set of characters, called *alphabet*. A *text* or *string* S of length $|S| = n$ is defined as $S = \sigma_1 \dots \sigma_n$ with $\sigma_i \in \Sigma$. If S can be represented as $S = usw$ with $u, s, w \in \Sigma^*$, the strings u, s and w are called *prefix*, *infix* (or *substring*) and *suffix* of S, respectively. For any indices i and j with $1 \leq i \leq j \leq n$, we define $S[i \dots j] = \sigma_i \dots \sigma_j$. The term $S[i] = \sigma_i$ denotes the i-th symbol in S. The *frequency* of a substring s in a string is written as $\#occ(s)$. A text S over Σ and a text T over $\hat{\Sigma}$ are considered *isomorphic* if an isomorphism $\varphi \colon \Sigma \to \hat{\Sigma}$ with $\varphi(S) = T$ exists.

2.3 Graphs

In this section, we give an introduction to graphs according to [Die06, CLRS09a, Ste07, Wil98].
A graph G is a tuple $G = (V, E)$ of two disjunctive sets V and E. The elements of V are called *vertices*. The elements of $E \subseteq V \times V$ are *edges*, which connect two vertices. Graphs represent binary relations in a clear way, take a social network as an example: we can save all friendships in the network in a table and search information about the relations between the members by sorting the table entries by names. As a graph, each member is represented by one vertex and a friendship relation is represented by an edge between the corresponding vertices. Now, we just need to go through all outgoing edges of a vertex to find the friends of a particular person.
We write an edge as a tuple $(u, v) \in E$ with $u, v \in V$. An edge e and a vertex v are *incident*, if v is one of the vertices of e, i.e. $e = (u, v)$ or (v, u). A *loop* is an edge that has only one incident vertex, i.e. (v, v).
A graph $G' = (V', E')$ is called a *subgraph* of $G = (V, E)$ iff $V' \subseteq V$ and $E' \subseteq E$.
A graph is called *undirected* if $(u, v) \in E \iff (v, u) \in E$ for every $u, v \in V$, and *directed* otherwise. For directed graphs, we distinguish between *incoming* (u, v) and *outgoing* (v, u) edges of a vertex v.
A graph $G = (V, E)$ is called *weighted* if there is a function $w : E \to \mathbb{Z}$, that assigns a number to every edge e, called the weight of e.

Neighbour: In undirected graphs, a vertex u is called a *neighbour* of v if they are connected by an edge. For directed graphs, we just consider the „outgoing neighbours“. This means u is neighbour of v if there is an edge from v to u, i.e. u neighbour of $v \iff (v, u) \in E$. The *degree* $\deg(v)$ of a vertex v is the number of its neighbours. An *isolated* vertex has no neighbours, i.e. $\deg(v) = 0$. In this project, $\deg(v)$ denotes the out-degree of vertex $v \in V$, i.e. the number of outgoing edges.

Walk: A *walk* $W = (v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k) \subseteq E$ in a graph $G = (V, E)$ is a sequence of edges where each edge starts at the end point of its predecessor. Its *length* is $|W| = k$, i.e. the number of edges. A *path* P is a walk in G, so that the visited vertices in P are pairwise distinct. The vertex v_0 is called the start point and v_k the end point of the path. We say that the path *goes from* v_0 *to* v_k. A *circle* is a walk $C = P \cup \{(v_k, v_0)\}$ of a path $P = (v_0, v_1), \ldots, (v_{k-1}, v_k)$ and the edge $(v_k, v_0) \in E$. A circle is therefore a cyclic edge sequence in G. A loop is considered as a circle of length 1. A vertex u is *reachable* by another vertex v if there exists a path with v as start point and u as end point.

Connectedness: A graph $G = (V, E)$ is *connected* if there is one vertex from which all others can be reached. We call this vertex *root*. The subgraph $C = (V_C, E_C)$ of a graph $G = (V, E)$ is called a *connected component* of G if it is connected and maximal. This means that each graph $\bar{G} = (\bar{V}, \bar{E})$ with $\bar{V} = V_C \cup \{v\}$ and $\bar{E} = E_C \cup \{(v, v')\}$ for every $v, v' \in V$, $v \notin C$, $v' \in C$ and $(v, v') \in E$ is not connected. A directed graph G is called *strongly connected* if for every pair u, v of vertices, there exists a path from u to v and from v to u.

Tree: An acyclic connected directed graph is called a *tree*. We call a connected directed graph a tree if the underlying undirected graph is acyclic. The vertices of trees are called *nodes*. The neighbours of the root are root nodes for subgraphs which are trees, too. In a tree $T = (V, E)$ we call u a *child* of v and v the *parent* of u, if u is reachable from the root by a path that contains the edge $(v, u) \in E$. The *height* of a tree is the number of edges in the longest path starting at the root. The *depth* of a node is its distance from the root, i.e. the number of nodes on the path from the root to the node.
A graph whose connected components are trees is called a *forest*. A tree that connects all vertices of a graph G is called a *spanning tree* of G.

2.3.1 Definition (k-tree-like graph). A *k-tree-like* graph is a spanning tree with a sparse number of $k \in \mathbb{N}$ additional (non-tree) edges. The value k is a yardstick of the tree-likeness of the graph [FP16].

2.3.1 Breadth-First Search

The *breadth-first search* (BFS) is an algorithm to traverse a graph. A graph traversal is a algorithmic way to consider every vertex and edge of a given graph one time. While the BFS is processing, we can scan some graph properties. For example in our project,a special graph representation is constructed while running the BFS. In the BFS, all neighbours of the current vertex are considered first and then, in turn, these neighbours are the current vertices whose neighbours are also considered. We distinguish between *undiscovered*, *discovered* and *finished* vertices. A vertex is discovered from the moment we visit it for the first time. The BFS terminates when all vertices are finished, which means that all their neighbours are discovered. We maintain a queue to save the discovered, but not yet finished vertices in the order they were discovered.
A BFS over the graph $G = (V, E)$ has a runtime of $\mathcal{O}(|V| + |E|)$. It needs $\mathcal{O}(V)$ space for the list of discovered vertices, the list of finished vertices and the queue.

2.3.2 Depth-First Search

The *depth-first search* (DFS) is another graph traversal (compare to Section 2.3.1). The algorithm recursively follows a path until a vertex is reached which either has no neighbours or whose neighbours have all already been discovered. Then, the next path is searched recursively starting from the previous-last path vertex. Just like the BFS, the DFS terminates when all vertices are finished, which means that all their neighbours are discovered.
A DFS over the graph $G = (V, E)$ has a runtime of $\mathcal{O}(|V| + |E|)$. It needs $\mathcal{O}(|V|)$ space for the lists of discovered and finished vertices.

2.4 Hash Functions

Hash functions are useful whenever we want to store data which belongs to irregularly shaped keys like URLs, user names or other strings. Basically, a hash function maps data of arbitrary size called *keys* to data of fixed size. To highlight the advantage of hash functions and hash tables, we first give an overview

about *direct-address tables* [CLRS09b, Chapter 11]. After that, we define hash functions and hash tables formally and explain why they are useful.

2.4.1 Direct-address Tables

When the universe $U = \{0, 1, \ldots, c-1\}$ of keys is small, *direct addressing* is useful. The keys in the universe U are stored in an array $T[0, 1, \ldots, c-1]$ called *direct-address table*, in which each position corresponds to a key in the universe U. The set of keys that are addressed to T is called K. If K contains no element with key $k \in U$, then the corresponding position to this key in T is empty, i.e. $T[k] = \text{nil}$. Figure 2.1 illustrates this basic idea.

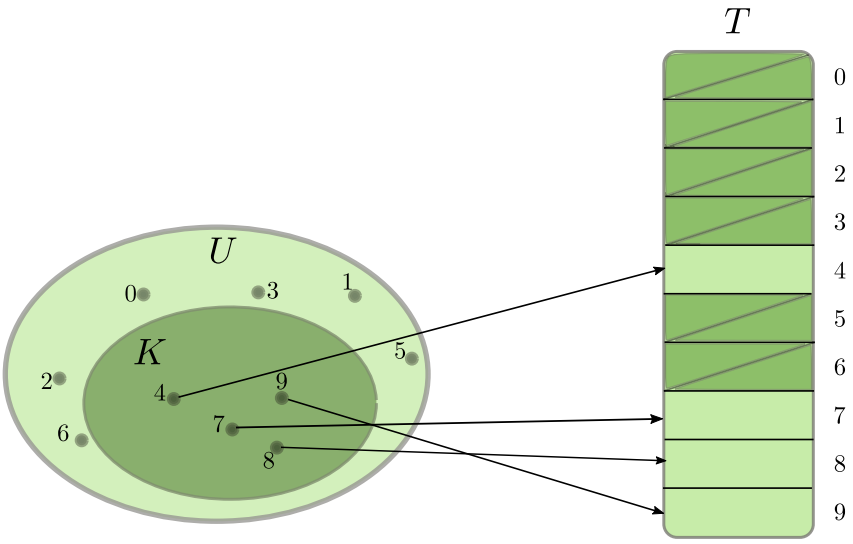


Figure 2.1: Illustration of direct addressing with universe $U = \{0, 1, \ldots, 9\}$. The set $K = \{4, 7, 8, 9\} \subset U$ of keys is addressed in the direct-address table T with 10 slots.

The direct-address table is easily constructed, since each key $k \in K$ must only be inserted into the array T at the index k, which is performed in $\mathcal{O}(1)$. It also supports search and delete operations in constant time. Nevertheless, when the universe U is large, direct-address table requires more space to be stored in the memory, since $|T| = |U|$. Moreover, when the set K of *actual keys* is much smaller in comparison to the universe U, the allocated space for T is wasted. In this case, hash functions and hash tables are more useful due to a $\mathcal{O}(K)$ space requirement.

2.4.2 Hash Functions and Hash Tables

A hash function is defined formally as a function $h : U \mapsto V$, which maps a universe of keys U to a set of values $V = \{0, 1, 2, \ldots, c-1\}$ for some integer c which is much less than $|U|$. The returned values of hash function are called *hash values* [MWHC96]. The *hash table* $T[0, 1, \ldots, c-1]$ represents the set V of values. Each position in T is called a *slot* and can store a key in the universe U. The hash function h is used to compute the slot $h(k)$ for the key $k \in U$. This action is called *hashing*. Figure 2.2 illustrates basically how hash function works. With hash functions, the arrays used for hash tables have small

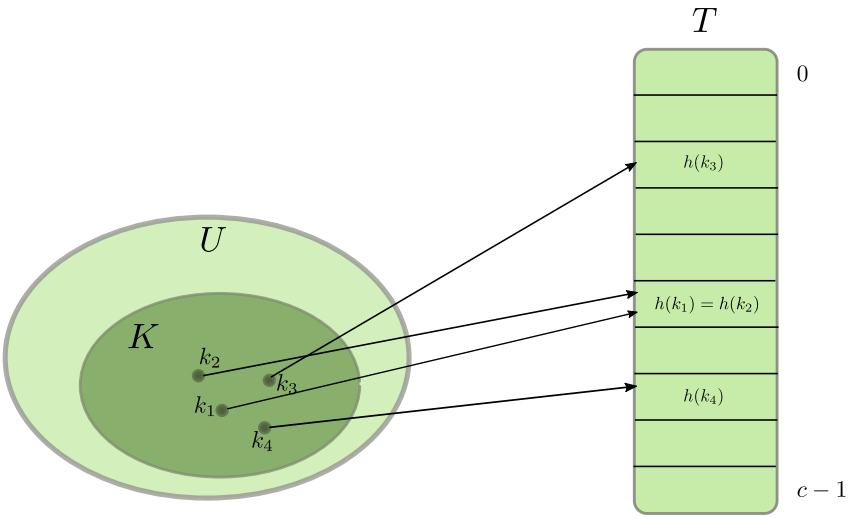


Figure 2.2: The hash function h maps the keys in K to slots of hash table T with c slots

size relative to $|U|$ and therefore the space requirement for hash tables is smaller than the direct-address tables.

Since the size of the universe U is usually larger than the size c of the hash table T, two keys in U may be mapped to the same slot in T. In Figure 2.2, two keys k_1 and k_2 have the same hash value and therefore are mapped to the same slot. This situation is called a *collision*. We take a look again at Figure 2.2. Assume that the slot $h(k_1)$ stores the key k_1, the key k_2 cannot be stored in the hash table T anymore, although there is much space left. When many collisions happen and are not resolved, many hash values cannot be stored in the hash table and many slots in the hash table will be left empty. Moreover, in applications in computer security, hash collisions can cause a *preimage attack* [RS04] or *collision attack* [RS04]. Hence, collisions should be avoided altogether, which is impossible, since $|U| > c$. To achieve a low number of collisions, we might choose a suitable hash function h which minimizes the number of same hash values for different keys. The collisions that occur still needed to be resolved. Two of many methods for resolving collisions are *chaining* (open hashing) and *open addressing* (closed hashing) which can be found in [CLRS09b, Sec. 11.2, 11.4]. By chaining method, the hash table is an array of linked lists, i.e. each slot has its own linked list. All keys with the same hash value will be stored in this linked list. In open addressing, all the keys are stored in the hash table. When some key is inserted into the hash table and the slot for its hash value is not empty, *probing* is performed until an empty slot is found. Resolving collisions often results in increasing search time in the hash table. For example, when the collisions are resolved with chaining, an unsuccessful search takes average-case time $\Theta(1+\alpha)$ [CLRS09b, Theorem 11.1].

2.4.3 Perfect Hash Functions and Minimal Perfect Hash Functions

Suppose the set K of actual keys is *static*, which means that we know this set from the beginning and it cannot be changed once the keys are stored in the hash table, we can choose a hash function h which is injective (i.e. $|K| < |V|$). In this case, no collision occurs, since distinct keys in K cannot have a

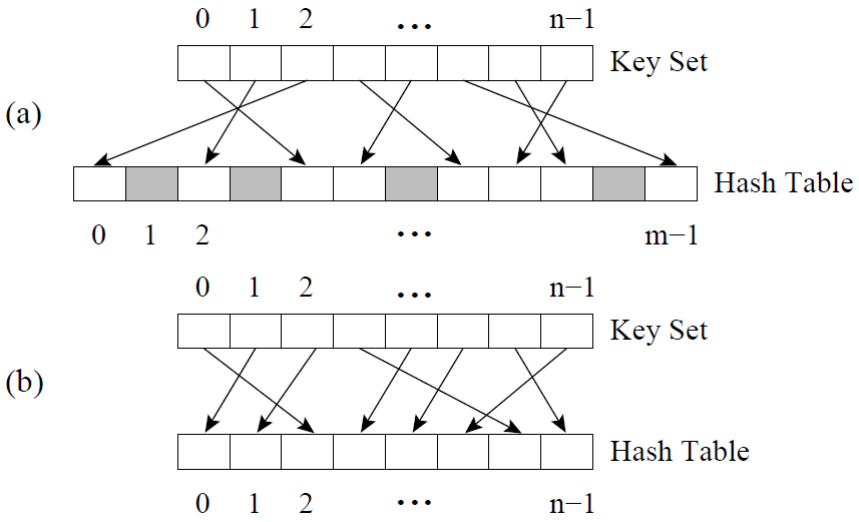


Figure 2.3: Sketch for PHF and MPHF [BKZ20]
a) *perfect hash function*: different keys are mapped to different values. There are values in the hash table that do not have corresponding keys, $|K| < |V|$
b) *minimal perfect hash function*: each key is mapped to a unique value and vice versa, $|K| = |V|$

same hash value. Therefore, collision resolving is not needed, which results in $\mathcal{O}(1)$-memory access searching. Here, h is a *perfect hash function* (PHF) [LRCP17]. A perfect hash function h is *minimal* (MPHF) if it additionally is surjective. That means h is bijective and $|K| = |V|$. Each key in K is mapped to a unique value in the value set V and vice versa. Figure 2.3 shows a simple explanation for perfect and minimal perfect hash functions.
The lower space bound for MPHFs is proven to be 1.44 bits per key [FKS82]. In theory, some algorithms can be parametrized such that the space bound is met, but in practice this is not feasible, because construction times get too long [BBD09]. Since MPHFs are typically used for massive data sets like in bioinformatics [CSM13, CLM16] or in databases [CL05], lowering the requiring space is of interest.

2.5 Bit Vector

A bit vector allows storing and retrieving individual bits arranged in a sequence. In addition to their application of managing a finite sequence of truth values, they find use as numerous support data structures throughout the PlaDs library. For a given bit vector $B \in \{0,1\}^n$, n is its number of stored bits.
We define access(B,i) as the operation that reads a bit at index i, and set(B,i,b) as the operation that sets the bit at index i to b. We also use the notation $B[i]$ as shorthand to mean access(B,i). In addition to setting and retrieving bits, we often require two additional operations from a bit vector: $rank_b$ and $select_b$. For any bit vector B, rank$_b(B,i)$ returns the number of set ($b = 1$) or clear bits ($b = 0$) in B up to and including index i in the bit vector. The operation select$_1(B,j)$ returns the position of the j-th set bit in the bit vector. If there are fewer than j total bits set in the vector, select$_1(B,j)$ returns n.

The operation select_0 behaves analogously for clear bits. If we omit b from a rank or select term, b is implicitly assumed to be 1.

These operations may easily be implemented by linearly scanning through the bit vector and counting the number of bits in $\mathcal{O}(n)$ time until the desired result is found. However, it is often worth using additional support data structures to significantly speed up the process of calculating the result. With these support data structures, it is possible to solve rank and select in $\mathcal{O}(1)$ time, with only $o(n)$ bits of extra space required [Nav16, Sec. 4.2][Gol07]. These data structures will be explored in further detail in Sections 2.5.2 and 2.5.3.

2.5.1 Access

The functionality of a bit vector may trivially be implemented by creating an array of boolean values, in which every cell of the array represents one bit. However, most programming languages allocate at least one byte (8 bits) for every cell, as this is the smallest unit of data that is easily addressable by modern processors. This is a highly inefficient use of space, which we avoid by storing a boolean value within every single bit of a word. We distribute the n bits of the bit vector among $\lceil n/w \rceil$ integers. Most modern processors have a word size of $w = 64$ bits, making it a convenient value to use in practice (see Section 2.1).

In order to retrieve the value of a bit at position i, we first determine the word it is located in by calculating $i' = \lceil i/w \rceil$. Within the corresponding word W, we find the corresponding bit at the index $i^* = i \bmod w$. This can be done using the bitwise AND ($\wedge$). In binary notation, 2^q always has only one bit set for any $q \in \mathbb{N}$. Therefore, by calculating $W \wedge 2^{(i \bmod w)}$, all bits except for the desired bit are cleared. The result is 0, if the desired bit is clear, and 1, if it is set. The word size w must be a power of two, which holds for 64.

In order to set a bit to a new value b', we determine its position in B analogously. If $b' = 0$, we clear the bit in W by setting W to $W \wedge \overline{2^{(i \bmod w)}}$. If $b' = 1$, we instead use the bitwise OR ($\vee$) to set the bit by setting W to $W \vee 2^{i \bmod w}$.

Both operations, *get* and *set*, take constant time. The resulting bit vector uses at most $w - 1$ extra bits of space compared to the n bits stored. The indexing scheme causes bits to be stored within a word least-significant-bit first. If a most-significant-bit first scheme is desired instead, one can use $2^{w-(i \bmod w)}$ instead of $2^{i \bmod w}$ to determine the position of a bit within a word.

2.5.2 Rank

In order to allow solving rank queries quickly, some support data structures need to be constructed alongside the bit vector. As $\text{rank}_0(B,i)$ can trivially be computed by calculating $(i+1) - \text{rank}_1(B,i)$, only rank_1 needs to be considered.

In order to speed up rank queries, we sample a set of results from B and store them for later retrieval. By using a two layer scheme, rank queries may be solved in constant time [Nav16, Sec. 4.2.2].

First, B is divided into *superblocks*, which span kw bits, for a given parameter $k \in \mathbb{N}^+$. For each superblock, we count the number of ones. The result of rank_1 at the beginning of each superblock is then

B	**1** 0	1 1	1 0	**0** 0	0 1	1 1	**1** 0	0 0	0 0	**1** 1
R	1			4			8			9
R'	1	2	4	0	0	2	1	1	1	1

Figure 2.4: An example sampling scheme for rank_1, with a word width $w = 2$ and $k = 3$ words in a superblock. The entries in R show how many total bits are set at the start of the superblock. The entries in R' show how many bits are set in the current block relative to the start of the superblock.

stored in an array R. We further divide every superblock into *blocks* that are w bits wide, in which we count the number of bits set relative to the start of its corresponding superblock. All blocks are further stored in another array R'. Figure 2.4 illustrates this sampling scheme.

In order to solve $\text{rank}_1(B,i)$, we first find the corresponding superblock $r = R[\lfloor i/kw \rfloor]$. We then find the corresponding block $r' = R'[\lfloor i/w \rfloor]$ and remember its value. Finally, we must determine the number of bits set from the start of the block up to i. For this, we can use the *popcount* instruction, which is supported natively in hardware by most modern processors. It quickly calculates the total amount of set bits in a word. Further details on popcount instructions, as well as software implementations of popcount, can be found in [Nav16, Section 4.2.1]. As we only need to scan the first $(i \bmod w)$ bits of the word, we first need to mask out the other $w - (i \bmod w)$ bits using a bitwise AND. We can then return the result of the operation by returning $r + r' + popcount(W \wedge (i \bmod w))$. Note that once again w must be a power of two.

Rank can be performed in constant time if popcount can be performed in constant time. If there is no constant time hardware instruction for popcount, one can pre-calculate all 2^{w-1} results for popcount and store them in a look-up table to achieve a constant time popcount [Nav16, p. 76]. The arrays R and R' introduce some space overhead. However, every cell of R and R' can be stored within $\lceil \log n \rceil$, and $\lceil \log kw \rceil$ bits respectively. Details on processing arrays with element widths not supported by hardware can be found in [Nav16, Sec. 3.1]. The space thus required for R becomes $\lceil n/kw \rceil w$ bits. R' requires $\lceil n/w \rceil \lceil \log kw \rceil$ bits. By choosing $k = w$, this space requirement becomes $o(n)$ [Nav16, p. 76].

2.5.3 Select

Whereas rank_0 could trivially be computed from rank_1, the same is not true for select. All data structures used for select_1 must be engineered, constructed and stored separately for select_0 [Nav16, p. 78]. The core idea remains: by sampling and pre-calculating solutions for select_b, one can quickly narrow the search space to a small range of bits. Let m be the total number of set bits in B. The following data structures can be defined analogously for select_0. As with rank, we use a two-stage sampling scheme for assisting with select_b, only this time we sample from the m positions of each set bit in the vector. Every superblock contains the position of every kw-th set bit in B. We again chose $k = w$, making every superblock span w^2 set bits. We then divide every superblock into blocks, each holding the position at every w-th set bit, relative to the start of the superblock. We call the arrays for the superblocks and blocks S and S' for select, respectively. A simple illustration for the sampling scheme for select is shown in Figure 2.5.

B	**1** 0 1	1 0 1	1 0 0 1	**1** 1	1 0 0 0 0 0 1	1 1	**1** 0 0
S	0			10			21
S'	0	3	6	0	2	9	0

Figure 2.5: An example sampling scheme for select_1. We sample $w = 2$ ones per block, and, and combine $k = 3$ blocks into one superblock. S contains the position of every kw-th set bit in B. For every superblock, S' contains the position of every w-th set bit relative to the start of the current superblock.

When answering a query $\text{select}_b(B, j)$, we first find $S[\lfloor j/kw \rfloor]$ to obtain the position of the corresponding superblock. To this, we add the value we find in $S'[\lfloor j/w \rfloor]$. From there, we scan linearly through the array, searching subsequently for set bits until we find the desired bit, returning its index. There will be at most $w-1$ set bits scanned in the linear search.

Rather than scanning through the words of the array bitwise, it is possible to speed up the implementation by using a *count leading zeroes* (CLZ) instruction. Similar to popcount, this type of instruction has hardware support or otherwise efficient implementations in many modern architectures. With this instruction, it is possible to find the first set bit in a word, then shift the bits right by the amount of bits returned to make the next set bit the first bit set in the word, and then repeat the process for every bit left. Using a constant *count leading zeroes* instruction, the time of the operation can be reduced to constant time for the array accesses plus $\mathcal{O}(w)$ time for scanning through at most w remaining set bits.

The space required for the support arrays is $\lceil m/kw \rceil w$ for S, and $\lceil m/w \rceil w$ for S'.

2.6 Trit Vector

A trit vector is similar to a bit vector, but each element is a *trit*, not a bit. A trit is a number of the set $\{0,1,2\}$. With trits, we can save numbers of the ternary system (base three). There are various methods for representing a trit vector and to support the operations access, rank and select in constant time.

Fischer and Peters [FP16] group five trits into one tryte. This tryte is stored in one byte. This solution needs $\lceil (n+t)/5 \rceil \cdot 8$ bits to store a trit vector with n entries and $t \leq n$ values out of $\{1,2\}$. This almost reaches the optimal space of $\lceil (n+t)\log(3) \rceil$ bits (Integer vector with size $n+t$ and width 3). Access to each trit is supported in constant time by Horner's method. Rank queries are solved by a look-up table [FP16, Mun96]. For select Fischer and Peters run binary searches on rank samples [FP16, GGMN05]. For rank and select additional $o(n+t)$ space is needed.

Our own trit vector solution is an adaption of a wavelet tree (compare to Section 2.8). Due to the fact that the alphabet of the trit vector is always $\{0,1,2\}$ we are able to store the trit vector in only two bit vectors. The first bit vector, called level 0, indicates if the trit at index i is value 0 ($\texttt{level0}[i] = 0$) or if we need to look at level 1. This other bit vector indicates if the trit is value 1 or 2. For details see Section 4.3. Level 0 has n entries and level 1 t entries. To support access, rank and select in constant time both bit vectors must offer rank and select queries (see Sections 2.5.2 and 2.5.3). In summary, our trit vector implementation needs $n+t+o(n+t)$ bits.

2.7 AVL Tree

AVL trees [Pun08, Jos10, CLRS09a] are binary search trees that are height-balanced. If n is the number of nodes, then the worst case depth of an AVL tree is $\mathcal{O}(\log n)$. This property is maintained by ensuring that for each node v in a binary tree T, the height of the left and the right subtree, h_l and h_r, differ by at most 1: $|h_l - h_r| \leq 1$.

2.7.1 Balance Factor

For each node of an AVL tree, the *balance factor* is defined as follows:

2.7.1 Definition (Balance Factor [Pun08, Jos10]). Let T be a tree and h_l and h_r the height of left and right subtrees respectively. Then the balance factor BF of every node in T is obtained as follows: $BF = h_l - h_r$.

This leads us to the following definition:

2.7.2 Definition (Balancing Rule [Pun08, Jos10]). A tree T is balanced and therefore an AVL tree, if and only if the balance factor of every node in T is in $\{0, 1, -1\}$.

With this rule, it is easy to see if a tree is no longer balanced, which happens exactly when there is a balance factor that is greater than 1 or less than -1. Each insertion or removal of a node may change the balance factor. Once the tree is unbalanced, there exist certain ways to rebalance it, which are explained next.

2.7.2 Rotations in AVL Trees

In order to restore balance in a tree, it has to be *rotated*. The four possible rotation techniques for an AVL tree are as follows:

- **Right Rotation**: A right rotation has to be performed on a node v if the height of the right subtree of v is less than the height of the left subtree of v. Additionally, the height of the left subtree of the left child v_l of v must be greater than the height of the right subtree of v_l. The left child v_l is moved into the place of v, while v becomes the right child of v_l. The node v inherits the right child of v_l as its left subtree, keeping its right subtree. The node v_l keeps its left subtree.

- **Left Rotation**: A left rotation in a node v is similar to a right rotation. In this case, the height of the left subtree of v is less than the height of its right subtree. Furthermore, the right child v_r of v has a right subtree with a height greater than the height of its left subtree. Correspondingly, v_r is moved into the place of v, while v becomes the left child of v_r. The left subtree of v_r becomes the right subtree of v, as v and v_r maintain their respective left and right subtrees.

- **Right-Left Rotation**: The right-left rotation is a combination of the two previously mentioned rotations: A right rotation of the right subtree v_r of v is applied, followed by a left rotation in v. This occurs if the height of the left child of v_r is greater than the height of its right child.

- **Left-Right Rotation**: The left-right rotation is similar to the right-left rotation. Here, the height of the right child of the left subtree v_l of v is greater than the height of its left child. In this case, a left rotation of v_l is performed, followed by a right rotation in v.

All individual rotations may be performed in $\mathcal{O}(1)$ time. Any insertion or deletion of a node may change the balance factor of the tree, possibly requiring rebalancing. Therefore, after an insertion or deletion the path from the affected node is traced back to the root and the balance factor of each node is examined. If the node is unbalanced, one of the four above mentioned rotations is performed. As the tree is always presumed balanced, up to $\mathcal{O}(\log n)$ rotations may need to be performed, resulting into a total required time of $\mathcal{O}(\log n)$ for insertions and deletions [Bra08, Section 3.1].

2.8 Wavelet Tree

The *wavelet tree* is a data structure that allows efficient rank and select operations in $\mathcal{O}(\log|\Sigma|)$ time on a string over an arbitrary alphabet Σ. The wavelet tree needs $\mathcal{O}(n\log(|\Sigma|))$ bits of space for a string of length n, which is $o(n)$ more space than the $n\lceil\log(|\Sigma|)\rceil$ bits needed to store the string explicitly, but meets the same asymptotic bound.

The wavelet tree was first designed by Grossi, Gupta and Vitta in 2003 [GGV03] and has since then been used in a number of different applications. A good overview of the data structure and its applications is given in [Nav12].

Hereafter, we only consider strings over the alphabet $\{1,\ldots,\sigma\}$, but our description can be easily transferred to arbitrary alphabets through specifying an order on the symbols.

The main idea of the wavelet tree is to construct a binary tree and to store a bit vector in every node that indicates whether the characters belong to the first or the second half of its associated alphabet.

The bit vector in the root has the same length as the original string and indicates for every position whether the symbol in the string has a value between 1 and $\lceil\sigma/2\rceil$ or between $\lceil\sigma/2\rceil+1$ and σ. Its children follow the same idea for the substrings restricted to the first and second half of the alphabet respectively.

Figure 2.6 shows an example wavelet tree for the string "342112243" over the alphabet $\{1,2,3,4\}$. The S-strings do not need to be stored in the wavelet tree but merely illustrate the idea. Only the three bit vectors need to be stored.

The tree has height $\lceil\log\sigma\rceil$ and needs $\mathcal{O}(n\log\sigma)$ bits of space to store the bit vectors.

To retrieve a symbol of the original string, one can use constant-time rank operations on the bit vectors to get the corresponding position in the respective substring. For example, to retrieve $S_{\text{root}}[7]$ we know that $B_{\text{root}}[7]=1$ and $\text{rank}_1(B_{\text{root}},7)=3$. Therefore position 7 of the original string corresponds to the third position of $S_{\text{right(root)}}$. Since $B_{\text{right(root)}}[2]=1$, we know $S_{\text{right(root)}}[2]=S_{\text{root}}[7]=4$.

This shows that we can reconstruct the original string from our wavelet tree. The construction of the tree can be implemented efficiently in a bottom-up-manner described in [FK17] using $\mathcal{O}(n\log\sigma)$ time and $2\sigma\lceil\log n\rceil$ additional bits of space for a string of length n.

Access, rank and select operations on a string over an arbitrary alphabet are defined analogously to the

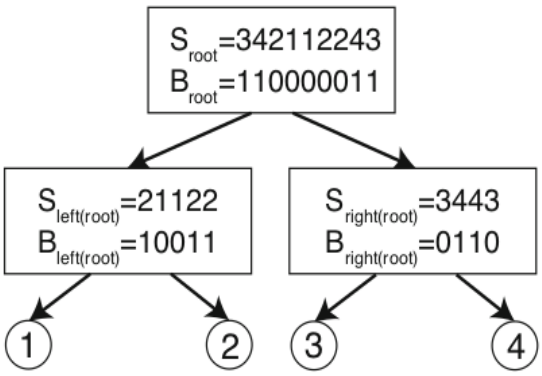


Figure 2.6: Wavelet tree for the string “342112243" [TTS13]

operations on a bit vector. For a string S, the operation access(S,i) returns $S[i]$, rank$_\tau(S,i)$ returns how often the symbol τ occurs in S up to the i-th position and select$_\tau(S,j)$ gives the position of the j-th occurrence of τ in S.

For the implementation of those operations we can make use of the efficient access, rank and select queries on bit vectors described in Section 2.5. This requires $o(n\log\sigma)$ additional space. The algorithms for the three operations are described in detail in [CNP15] and take $\mathcal{O}(\log\sigma)$ time.

Altogether, we can ensure efficient rank and select operations on a string over an arbitrary alphabet using $\mathcal{O}(n\log\sigma)$ bits for storing and $2\sigma\log n$ additional bits during the construction. However, the second space bound is not practical for large alphabets. For this case the so-called *wavelet matrix* has been developed in [CNP15] that only needs $\log\sigma\log n$ additional bits during construction and otherwise meets the time and space requirement of the wavelet tree.

2.9 Entropy Coding

A frequently used method for reducing the number of bits required to store data structures is to *compress* them. Broadly speaking, if instances of data structures are considered to be elements of a set, or represent collections of elements from a set, we wish to use fewer bits to represent more frequently occurring elements, while allocating more bits to represent less frequent elements. Let us call this set $\mathcal{U}$. To uniquely identify an element of $\mathcal{U}$, we assign a binary *code* to it. To identify any element of $\mathcal{U}$, we then need $\lceil\log|\mathcal{U}|\rceil$ bits. However, in many problem domains, symbols tend to be unevenly distributed, which we may exploit to find shorter representations for data structures.

We are only interested in coding schemes which can be reversed to unambiguously reconstruct the original data, i.e. lossless compression. This puts a lower space bound on any coding scheme, which we seek to approximate as close as possible. If the likelihood of an element occurring is given by a

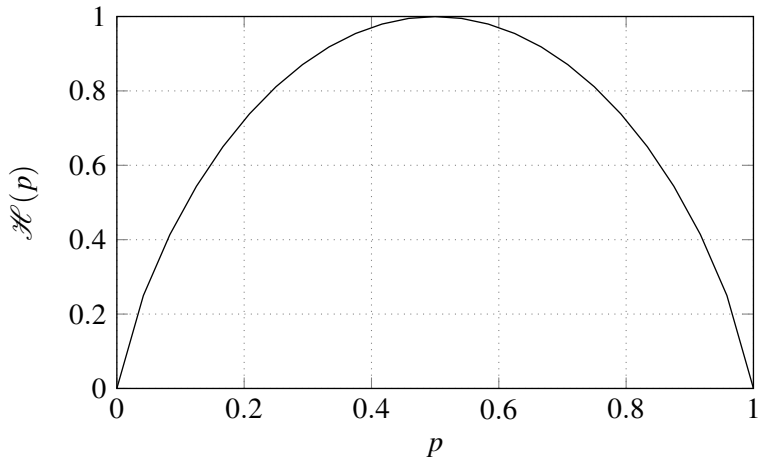


Figure 2.7: The binary entropy $\mathscr{H}(p)$ [Nav16]

probability distribution $\mathscr{P} : \mathscr{U} \mapsto [0,1]$, the minimum possible average length for every code is the *Shannon entropy* [Nav16, Sec. 2.2]:

$$\mathscr{H}(\mathscr{P}) = \sum_{u \in \mathscr{U}} \mathscr{P}(u) \cdot \log \frac{1}{\mathscr{P}(u)} \tag{2.1}$$

An optimal code for u should therefore be $\log \frac{1}{\mathscr{P}(u)}$ bits long.

2.9.1 Binary Entropy

If our goal is to compress bit vectors, $\mathscr{U}$ becomes $\{0,1\}$ and $|\mathscr{U}| = 2$. Now, 1 occurs with the probability $p = {}^{m}/_{n}$, where m is the number of ones in the bit vector and n its length. Correspondingly, 0 occurs with probability $1-p$. We can define the binary entropy function $\mathscr{H}(p)$ as follows [Nav16, Sec. 2.3]:

$$\mathscr{H}(p) = p \log\left(\frac{1}{p}\right) + (1-p)\log\left(\frac{1}{1-p}\right). \tag{2.2}$$

A graph for $\mathscr{H}(p)$ is shown in Figure 2.7. We see that if p is near 0.5, the bits in the bit vector are very evenly distributed, and compression is not possible. If the bits are very unevenly distributed, however, $\mathscr{H}(p)$ drops, and we can achieve substantial space reduction.

Note that we assume that one bit in the sequence does not influence any bits around it. The occurrence of each bit is solely governed by its probability p. We call this the *zero-order* entropy, and define the *zero-order empirical entropy* of a bit vector B as [Nav16, Sec. 2.3.1]:

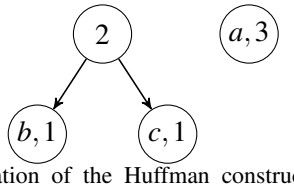
$$\mathscr{H}_0(B) = \mathscr{H}\left(\frac{m}{n}\right) = \frac{m}{n}\log\left(\frac{n}{m}\right) + \frac{n-m}{n}\log\left(\frac{n}{n-m}\right). \tag{2.3}$$

The lowest possible bound for compressing B is therefore $n\mathscr{H}_0(B)$ bits. We will apply this principle to dynamic bit vectors in Section 5.4. There we will see a scheme that compresses bits using a $\mathscr{H}_0$ entropy based compression scheme, and describe methods to encode and decode bits. Since we assume bits to be independent of each other, we are not able to compress arbitrary data structures this way by applying

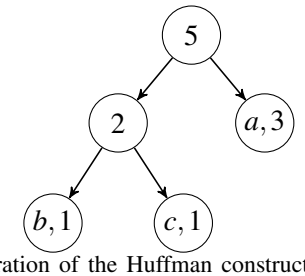
a	b	c
3	1	1

(a) Symbols (upper row) and their frequencies (lower row), input of the Huffman construction.

(b) Initialisation of the Huffman construction: the symbols saved as leaf nodes and sorted by frequencies.



(c) First iteration of the Huffman construction: nodes $(b,1)$ and $(c,1)$ are merged and sorted in again.



(d) Last iteration of the Huffman construction: nodes 2 and $(a,3)$ are merged, node 5 is the result of the algorithm.

Figure 2.8: Example of the Huffman tree construction Algorithm 2.1

such a compression scheme to their bit vector representations, as the bits are typically closely related to each other. We are still able to compress bits treated as raw data, however.

2.10 Huffman Coding

The goal of *Huffman codes* [Huf52] is to minimize the average code length. To do this, the data set is split into separate, perhaps repetitive, symbols, e.g. characters in a text. These symbols are represented in binary. The set of all symbols is called alphabet Σ. The more frequent a symbol appears in the data set, the fewer bits are used to encode the symbol. This means Huffman codes are *variable-length codes*. We represent these codes as a binary tree, whose leaves contain the symbols. To decode a single symbol, we start at the root and go left for every 0 bit and right for 1 until we reach a leaf. This leaf contains the decoded symbol. If a symbol appears more frequently than others in the data set, the path from the root to the corresponding leaf is shorter. The runtime of decoding is the length of the encoded symbol and respectively the depth of the corresponding leaf. The maximum depth of a Huffman tree with frequencies in $[1,n]$ is $\mathcal{O}(\log n)$, i.e. decoding needs $\mathcal{O}(\log n)$ time. A Huffman tree consists of exactly $|\Sigma|$ leaves and $|\Sigma|-1$ internal nodes [CLRS09a, Nav16].

In Algorithm 2.1, a greedy Huffman tree construction algorithm is given, requiring the symbols and their frequency. A list initially contains the leaves of the Huffman tree (see Lines 1 to 3). In each iteration, the algorithm merges the two subtrees of the list with the smallest frequencies into a new subtree. Its frequency is the sum of the frequencies of the two subtrees (compare to Line 6). The merge result is sorted in the list by its new frequency (loop from Line 7 to 9 updates the insert position, Line 10 to 12 is the insertion itself). This is repeated until the list contains only a single element, the root node of the Huffman tree (Line 5). Figure 2.8 gives an example. The algorithm needs $\mathcal{O}(|\Sigma|\log(|\Sigma|))$ time.

Input: $S[1,|\Sigma|]$, with $S[i].s$ a distinct symbol and $S[i].f$ its frequency
Output: Huffman tree with internal node fields l (left) and r (right) and leaf field s (symbol)

1: Sort $S[1,|\Sigma|]$ in ascending order of $S[i].f$
2: Create list L linked by field $L.next$, with its i-th element
3: $L.s \leftarrow S[i].s$, $L.f \leftarrow S[i].f$, $L.l \leftarrow$ null, $L.r \leftarrow$ null
4: $I \leftarrow L$ (the first list element)
5: **while** $L.next \neq$ null **do**
6: Create list element $N.l \leftarrow L, N.r \leftarrow L.next, N.f \leftarrow N.l.f + N.r.f$
7: **while** $I.next \neq$ null **and** $I.next.f \leq N.f$ **do**
8: $I \leftarrow I.next$
9: **end while**
10: $N.next \leftarrow I.next$
11: $I.next \leftarrow N$
12: $L \leftarrow L.next.next$
13: **end while**
14: **return** L (is a single element, the root of the tree)

Algorithm 2.1: Construction of a Huffman tree with alphabet $\Sigma = [1,|\Sigma|]$ [Nav16, Algorithm 2.2]

2.11 Gap Encoding

In general, we need at least $m \log n$ bits to store an array of length m over $\{1,\ldots,n\}$. However, if we want to store an array D that is non-decreasing, meaning $D[1] \leq D[2] \leq \cdots \leq D[m]$, we can use a technique that requires $n+m$ bits of space. The *gap encoding* of D is a bit vector of the following form:

$$B_D := 0^{D[1]}10^{D[2]-D[1]}1\ldots0^{D[m]-D[m-1]}1.$$

For example, the gap encoding of $\{1,2,4,4,6\}$ is 01010011001. The set bits are separating the indices and the value $D[i]$ can be retrieved by counting the clear bits up to the $(i-1)$th one, meaning $D[i] = \text{rank}_0(B_D, \text{select}_1(B_D, i-1))$. In Section 2.5 we have seen that we can answer those rank and select queries in constant time with $o(n+m)$ and $\mathcal{O}((n+m)\log\log(n+m)/\log(n+m))$ bits of additional space, respectively. Therefore, gap encoding improves the space requirement for a non-decreasing array while still ensuring constant access time.

Chapter 3

Project PlaDs

3.1 Overview

In this chapter we elaborate on our library PlaDs. We give an overview of its content and structure. We also explain how to use it and describe the environment in which we execute our benchmarks to evaluate our implementations.
The aim of our project PlaDs is to build a library of data structures that need only a small amount of space while supporting fast operations. These data structures are *(dynamic) bit vectors*, *graphs*, *grammar-compressed strings* and *hash functions*. More specifically, we construct the following data structures:

- **Dynamic bit vectors (DBV)**: After being constructed, the size of conventional static bit vectors cannot be changed any more. Moreover, the operations rank and select require not only the original bit vector, but also other support structures. Once the contents of the original bit vector change, these structures must be reconstructed, which affects the performance of the operations. To solve this problem, we provide dynamic bit vectors. This data structure offers both fast updates and fast queries.

  We implemented the *dynamic bit vector* introduced in [Nav16]. Apart from a bit vector, this data structure consists of an AVL tree, enabling fast query as well as fast update operations. These operations include rank, select, get, insert, delete and set. In order to achieve not only time efficiency, but also space efficiency, we implemented another variant of the data structure: the compressed dynamic bit vector is based on the $\mathscr{H}_0$ entropy and allows space reduction for both very sparse and very dense bit vectors.

- **Grammar-compressed strings (GCS)**: Strings are one of the most important data types in many applications. Because strings are often too long to be stored explicitly in the memory, we need a way to compress them without losing information while allowing fast operations. In our library we provide special grammars for this purpose.

  We implemented a grammar compression for strings, that enables us to access individual parts of the original text without decompressing it. Our compression is based on straight line programs

Dynamic bit vectors	• `include/plads/bit_vector_dynamic/bit_vector_dynamic.hpp`
Grammar-compressed strings	• `include/plads/gcs/grammar_compressed_string.hpp` • `include/plads/gcs/util/succinct_slp.hpp` for succinct storing of SLPs
Graphs	• `include/plads/glouds/glouds.hpp` for GLOUDS without pre-calculation of the optimal start points of connected components • `include/plads/graph/glouds_startpoint.hpp` for GLOUDS with optimal start points
Minimal perfect hash functions	• `include/plads/mphf/mphf_fire.hpp` for FiRe • `include/plads/mphf/mphf_fipha.hpp` for FiPHa

Table 3.1: Overview of the main implementation classes of PlaDs

(SLPs). We use the RePair algorithm implemented by Navarro [1] to construct a SLP from the original text. This SLP is then stored succinctly with the SLP compression described in [TTS13]. Our implementation supports efficient access, size and lce operations on the original text.

- **Graphs**: Graphs are very useful for computer science because they model binary relations. We implement different methods to encode graphs. Operations like neighbour queries with the help of the space efficient encoding should be in constant time.

  We implement two versions of GLOUDS [FP16]. This solution is optimised for tree-like graphs. One version is faster, the other one needs less space.

- **Minimal perfect hash functions (MPHF)**: *Minimal perfect hash functions* are the most desirable hash functions, as described in Chapter 2. In this project we attempt to build data structures from which files can be retrieved with the help of minimal perfect hash functions.

  We implemented at first the FiRe data structure [MSSZ14] which produces a method for *file retrieving*. The second implemented data structure FiPHa [MSSZ14] is a optimised version of FiRe regarding the construction time and space. FiPHa is also a minimal perfect hash function.

3.1.1 Overview of the Project's Structure

The data structures are implemented header-only. All implementation files of the mentioned data structures are located in the folder `include/plads`. There is a folder for each implemented data structure. For example, the implementation for FiRe is the file `mphf_fire.hpp` in the folder `include/plads/mphf`. The main implementation class for each data structure is listed in Table 3.1.
All the test files are located in the folder `test` under the main folder of the PlaDs library. The tested data structure is indicated in the name of the test file.

[1] See `https://users.dcc.uchile.cl/~gnavarro/software/repair.tgz`

Namespaces

All data structures of the project have the root namespace `plads`. Each field has its own namespace that is included in the namespace `plads`. This structure results in nested namespaces for the data structures. For example, the implementations for graphs can be accessed under the namespace `plads::graph`.

External Libraries

The used submodules and external libraries can be found in the folder `deps` under the main folder. Here we have data structures that are included in our benchmarks. We also use the submodule malloc_count[2] to measure the space requirement of the data structures' constructions and the implemented operators for the benchmark. The submodule is already integrated in the file CMakeLists in the main folder. Whenever we need a hash map, we use `tsl::robin_map`[3] instead of `std::unordered_map`. It answers queries a lot faster and needs similar amount of space as `std::unordered_map`. We also used the implementation of RePair for SLPs (see Section 6.2).

3.1.2 Instruction for Using the Library

System Requirement

Our project is written in `C++17`. The library will only work for CMake version 3.4.1 or later. For compiling, `gcc` is needed with a version later than 8.3.

Linking the Library

Our library is under the GNU General Public License and therefore can be linked in other projects. There are two different ways to link our library to other project.

1. *Clone or download the git-Repository*: One developer can create a directory `lib` in the source folder of his project. After that, the library plads can be cloned or downloaded into this `lib` folder. In particular, the following commands would be called respectively under the source folder of the project: `mkdir lib`
   `git clone http://flint-v3.cs.tu-dortmund.de/plads/plads.git plads` (for cloning only)

2. *Using git submodules and CMake*: The folder `lib` should be created beforehand. Then, the submodule can be added inside `lib` directory with `git submodule add http://flint-v3.cs.tu-dortmund.de/plads/plads.git lib/plads`. The library `plads` will be cloned and put into `lib/plads`.

   We also need to configure the `CMakeLists.txt` file, so that the project can recognize the submodule later. We can define a variable `EXTERNAL_LIB_HEADERS` for the external library `plads`

[2] `https://github.com/bingmann/malloc_count/`
[3] `https://github.com/Tessil/robin-map`

by adding `set(EXTERNAL_LIB_HEADERS lib/plads)` into the `CMakeLists.txt` file. Then, `EXTERNAL_LIB_HEADERS` will be added to the list of directories where C++ compiler should be looking for header files.

```
target_include_directories(
executable_name PRIVATE
${EXTERNAL_LIB_HEADERS}
)
```

The advantage of git submodules is that the changes of the submodules will be tracked and cloned automatically when `cmake ..` is called.

Using the Implemented Data Structures

Since all implementation files of the data structures are located in the folder `include/plads`, the necessary data structure can be easily included in other classes with the keyword `include`. For example, assume that the library `plads` is stored in `lib` directory and `lib` is already defined as a `include`-path in CMakeFile of the project, the following command must be inserted to use FiRe data structure:

```
#include <plads/include/mphf/mphf_fire.hpp>
```

Compile

A folder `build` under the source folder should be created before the compilation. In the created folder, the following commands would be called to compile the library:

1. `cmake ..`
2. `make`

The test files for the data structures can be found under the path `build/tests` after the compilation of the whole library. The test for any data structure will be executed when the corresponding output file is called.

3.2 Benchmark

In this section, we describe the environment and execution of our benchmarks that we use to evaluate our implementations.

3.2.1 Environment of the Benchmark

Our benchmark is executed entirely on the *Lido3-Cluster*[4] of the TU Dortmund. There are two types of nodes for the users: the 2-socket node and the 4-socket node that are listed in Table 3.2.

[4] `https://www.itmc.tu-dortmund.de/cms/de/dienste/hochleistungsrechnen/lido3/index.html`

Nodes	CPU Type	Max Cores	Max RAM/Nodes	Remark
cstd01	2x Intel Xeon E5-2640v4	20	64GB	
cstd02 or ib_1to1	2x Intel Xeon E5-2640v4	20	64GB	non-blocking
cquad01	4x Intel Xeon E5-4640v4	48	256GB	
cquad02	4x Intel Xeon E5-4640v4	48	1024GB	

Table 3.2: List of nodes which do not have GPU on the Lido3-Cluster

The used computer nodes on the *Lido3-Cluster* have at least 64GB of RAM and are equipped either with *Intel Xeon E5-2640v4* or *Intel Xeon E5-4640v4*. Each node contains local storage with at least 2TB. Each 2-socket node has 2.4GHz frequency and 25MB L3 Cache. Each 4-socket node has 2.1GHz and 30MB L3 Cache. More detailed informations about the cluster can be found on `https://www.lido.tu-dortmund.de/cms/de/LiDO3/lido3kurz.pdf`.
We use the node `cstd01` for the benchmarks of DBV, GCS and MPHF. The benchmarks for Graph are executed on node `cquad01`. Since these data structures have different functions, it is not important that they are executed on different nodes.

3.2.2 Description of the Experiments

For each of our data structures we measure the required time and space for the construction of the data structure and for the operations which are implemented. We use the class `plads::util:timer` for the measurement of the time requirement with the use of `std::chrono` in the header. As for the measurement of the required space we have the class `plads::util::spacer` with the usage of malloc_count in the header. After the compilation, the output files for the benchmark are in the corresponding path in the folder `build`.
For different data structures, we have different benchmarks using particular sets of test data. The full information about the benchmarks can be found in the chapter of the specific data structure. Moreover, the result of each benchmark can be found there, too. We use *sqlplotools*[5] to present the results in diagrams.

3.2.3 Execution of the Benchmark

All the files for the benchmarks can be found in the folder `src/`. After compilation, the executable benchmarks are located in the `build` directory. A new result file is created for each time we run the benchmark. This result file is the input for *sqlplottools*.

[5] `https://github.com/bingmann/sqlplot-tools`

Chapter 4

Basic Data Structures

In this chapter, we describe our implementation of the basic data structures static bit vector, static integer vector, trit vector and permutation. We need those data structures in many cases throughout our library. Therefore, an efficient implementation is important for us.

4.1 Static Integer Vector

The class `int_vector` is a space-optimised version of a `std::vector<uint64_t>`. The vector has the parameters size n, which is the number of integers in the vector, and width w, the number of bits used to save these integers. The integers are stored as a bit sequence divided in 64-bit-blocks. Therefore, an `std::vector<uint64_t>` is used. An integer vector of length n supports the following operations:

- `size()`: returns n, the number of integer values in the vector.
- `resize`(s,w): sets the size of the `std::vector<uint64_t>` to the needed number of 64-bit-blocks. For s entries represented with w bits, $s \cdot w$ bits and thus $\lceil s \cdot w/64 \rceil$ words are needed.
- `rebuild`(s,w): changes s and w of the integer vector, respectively the underlying `std::vector<uint64_t>`, and retains the values already entered. This requires copying all values once.
- `set`(i,b): sets the value at index i to integer value $b \in \mathbb{N}_0$ with $0 \leq i < n$ and $b \leq 2^w - 1$.
- `get`(i): returns integer value at index i with $0 \leq i < n$.

For `get` and `set`, we calculate which blocks the bits are located in that encode the integer that was asked for. The bit sequence that is the binary number of the integer at index i could be at the end of a 64-bit-block, at its beginning, or crossing a block border, i.e. a prefix of the number could be at the end of a block and the suffix at the beginning of the next block. Bitwise operators are used to realize the access in constant time. When n is the number of entries and w is the number of bits needed to save the highest entry, the size of the underlying vector is $n \cdot w$ bits plus at most 63 additional bits (number of blocks needs to be rounded up). All functions are executed in constant time.

operator	function	example
x `&` y	x and y	`0101 & 1001 = 0001`
x `|` y	x or y	`0101 | 1001 = 1101`
$\sim x$	not/ invert x	$\sim$ `1001 = 0110`
x `«` y	shift left/ multiply with 2^y	`1 « 4 = 10000`
x `»` y	shift right/ divide integer by 2^y	`1001 » 1 = 100`
$x?y:z$	if condition x is true, evaluate to y else z	`0 ? 2 : 1 = 1`

Table 4.1: Overview bitwise operators [Gee, cpl]

4.2 Static Bit Vector

Static bit vectors are vectors over the alphabet $\{0,1\}$ with a fixed, unchangeable length. In preparation for the project group, we ran a competition for the most efficient implementation of a bit vector. The target was that the operations `set`, `get`, `rank` and `select` should be done in a minimum constant time. The basic ideas for the data structures that realize this are given in Section 2.5. For the library PlaDs, we choose the fastest implementation in the competition by Patrick Dinklage, one of the supervisors of the project group. In the following, the implementation and the time and space requirement is analysed in detail. The basic idea of all functions is to divide the data into 64-bit-blocks in order to optimize the data for the 64-bit-processor. Furthermore, bitwise operators (see Table 4.1) are used to minimize the runtime.

4.2.1 Set and Get

The basic bit vector B of length n supports three operations (compare to Section 2.5.1):

- `size`(): returns n, i.e. the number of bits of B,
- `set`(B,i,b): sets the bit at index i with $0 \leq i < n$ to the value $b \in \{0,1\}$, and
- `get`(B,i): returns the bit at index i with $0 \leq i < n$.

The implementation uses a `std::vector<uint64_t>` of length $\lceil n/64 \rceil$. Each entry of the vector stores 64 bits of the bit vector. We call this one 64-bit-block. To enable the access to the bits for reading and writing, we convert an index in the bit vector to the corresponding block and index in this block. The function `block`(i) calculates the index $\lfloor i/64 \rfloor = \lfloor i/2^6 \rfloor = i \gg 6$ of the block that includes the i-th bit in the bit vector. The function `offset` calculates the index $i\%64$ of the i-th bit within its 64-block. With these functions, `get`(B,i) and `set`(B,i,b) convert the input index i in the bit vector B into the block in the `std::vector<uint64_t>` and the index in this block. We can `get` the k-th bit of a block with an AND operation of the block and 2^k (binary number where only index k is 1, shift left, see Table 4.1). The result is 2^k if $k = 1$, otherwise 0. To `set` the k-th bit in a block to the value $b = 1$, we use an OR operation for the block and the bit mask 2^k, because after that the k-th bit is 1, no matter what value k had. `Set`ting the k-th bit in a block on value $b = 0$ is possible by an AND-Operation with a bit mask, where all bits except

the k-th are set. The representation of a bit vector of length n as a `std::vector<uint64_t>` needs at most 63 additional bits (number of blocks needs to be up rounded), which is negligible in practice. The constructor needs linear time to copy the values of a given `std::vector<bool>` into the optimised block structure. By using bitwise operators, access, reading and writing take no additional space and can be executed in constant time.

4.2.2 Rank

To count the bit vector entries of a certain value, we create the rank bit vector, which is a data structure in addition to the basic bit vector. A rank bit vector B of length n supports two operations (compare to Section 2.5.2):

- $\texttt{rank}_1(B,i)$: calculates the number of set bits up to and including index i with $0 \leq i < n$, and
- $\texttt{rank}_0(B,i)$: calculates the number of clear bits up to and including index i with $0 \leq i < n$.

The implementation uses a pointer on the bit vector, for which rank queries shall be supported. As described in Section 2.5.2, we create two data structures to support rank: blocks and superblocks. Both are saved as an integer vector (see Section 4.1). Each entry in the integer vector of the superblocks saves the number of ones from index 0 until the next 64 blocks, i.e. $64 \cdot 64 = 4096$-bit-blocks of the bit vector. Each entry in the integer vector of the blocks saves the number of ones from the beginning of the corresponding superblock up to the next 64-bit-block of the bit vector. First, to construct the blocks and superblocks, the number of needed blocks and superblocks and the maximal possible value of each block, called the width of an entry, is calculated. Given the number and width of the entries, an optimised integer vector is built by the function `resize(size,width)`. To fill the support data structures, we iterate once over the bit vector to count and save the read ones. As explained in Section 2.5.2, $\texttt{rank}_1(B,i)$ is solved by finding the corresponding superblock $r = R[\lfloor i/4096 \rfloor]$, the corresponding block $r' = R'[\lfloor i/64 \rfloor]$ and calculating the sum $r''$of bits set from the start of the block up to i. Then, $\texttt{rank}_1(B,i) = r + r' + r''$. Similar to the access for reading and writing (see Section 4.2.1), we calculate the superblock and previous block by bitwise operators in constant time, if the processor supports `__builtin_popcountll`[Fre]. The GCC compiler function counts the ones in a 64-bit-block up to and including a given index in very small constant time. The function $\texttt{rank}_0(B,i)$ is trivially computed by calculating $(i+1) - \texttt{rank}_1(B,i)$. The superblock data structure is an integer vector with size $\lceil n/4096 \rceil$ and width $\log\lceil n-1 \rceil$. The block data structure is an integer vector with size $\lceil n/64 \rceil$ and width 12 (for $4096 = 2^{12}$). The construction time is $\mathcal{O}(n)$ because of the iteration over the whole bit vector. By using the support data structures, it is possible to count the number of set or clear bits in $\mathcal{O}(1)$ time. If `set` operations are executed on the corresponding bit vector, the built rank support data structures are no longer valid.

4.2.3 Select

To choose the k-th bit vector entry of a certain value, we implement the select bit vector, a data structure in addition to the basic and rank bit vector. A select bit vector B supports two operations (compare to Section 2.5.3):

- $\texttt{select}_1(B,k)$: returns the index of the k-ths bit which is set to 1 with $1 \leq k \leq$ number of set bits in B, and
- $\texttt{select}_0(B,k)$: returns the index of the k-ths bit which is set to 0 with $1 \leq k \leq$ number of clear bits in B.

The implementation uses a pointer to the bit vector for which select operations shall be supported. As described in Section 2.5.3, we create two data structures to support select for each possible value $b \in \{0,1\}$. This is because, unlike rank, we are not able to reconstruct $\texttt{select}_0$ from $\texttt{select}_1$. The data structures for 0 and 1 works the same way. Therefore, we have a template class with the value $b \in \{0,1\}$ for which the structure shall work as template parameter. The select class generates two select objects out of the template, one for $b = 0$ and one for $b = 1$. In the following we explain the structure of the template class with the parameter b. The select data structure consists of two data structures, the blocks and the superblocks. Both are saved as an integer vector (see Section 4.1). The i-th superblock encodes the index where $i \cdot \texttt{superblock\_size}$ bits are set on b with $\texttt{superblock\_size} = (\max\{\log\lceil n-1\rceil, 1\})^2$. The j-th block encodes the index where $j \cdot \texttt{block\_size}$ bits from the start of the corresponding superblock are set on b with $\texttt{block\_size} = \max\{\log\lceil n-1\rceil, 1\}$. One superblock consists of $\log n$ blocks. To construct the support data structure we iterate over all 64-bit-blocks of the bit vector, for which select operations shall be supported. We use the GCC compiler function `__builtin_popcountll` [Fre] to count the ones at the 64-bit-block until including a given index in very small constant time. This allows us to decide which 64-bit-block contains the index that is stored next in our support data structures. As can be clearly seen in the example of Figure 2.5, the first block in the new superblock always has the entry 0. Therefore our implementation omits this entry. To `select` the k-th index set to b, we search for the right superblock and block. Therefore, we set two local variables on the highest possible superblock and block and narrow them down to the right ones linear in step of the (super-)block sizes. This takes $\mathcal{O}(w)$ time. From this point we search forward directly in the bit vector the right index. The `gcc` compiler function $\texttt{\_\_builtin\_ctzll}(v)$ [Fre] returns the number of trailing zeros in a 64-bit-block v in very small constant time. Shifting left (see Table 4.1) by the number of zeros achieves that the leftmost bit in the result 64-bit-number is the first (leftmost) value set to 1 in the original one. Now we call `__builtin_ctzll` again to get the second value set to 1 and so on. For $b = 0$, we invert the 64-bit-block, so the zeros are ones and vice versa and we do the same procedure. The data structure for superblocks is an integer vector with size $\lceil n/(\max\{\log\lceil n-1\rceil, 1\})^2\rceil$ and width $\max\{\log\lceil n-1\rceil, 1\}$. The block data structure is an integer vector with size $\lceil n/(\max\{\log\lceil n-1\rceil, 1\})\rceil$ and width $\max\{\log\lceil n-1\rceil, 1\}$. The construction time is $\mathcal{O}(n)$ because there are two nested loops where the outer one iterates over all 64-bit-blocks of the bit vector and the inner one iterates over a part (worst case all) of the bits current 64-bit-block. By using the support data structures, it is possible to select the index where a given number of reached ones

or zeros in $\mathcal{O}(\log n)$ time. If `set` operations are executed on the corresponding bit vector, the built select support data structures are no longer valid.

4.3 Static Trit Vector

A trit vector is similar to the bit vector but the elements are out of $\{0,1,2\}$. The trit vector T supports two operations (compare to Section 2.6):

- `size`(): returns the length, i.e. the number of trits of the trit vector,
- `access`(T,i): return the value at index i with $0 \leq i <$ `size`(),
- `rank`$_b(T,i)$: calculates the number of occurrences of the value $b \in \{0,1,2\}$ up to and including index i with $0 \leq i <$ `size`(), and
- `select`$_b(T,i)$: calculates index of the i-th occurrence of $b \in \{0,1,2\}$ with $0 \leq i <$ `size`(). If value b occurs less than i times, `size`() is returned.

The basic idea for our implementation is a wavelet tree (see Section 2.8). In a trit vector the alphabet is always $\{0,1,2\}$. Due to this fact we are able to optimise the wavelet tree: we only need two levels of the wavelet tree, with just one bit vector in each level. The first bit vector is called *level 0* and has the length of the trit vector. If `level0`$[i] = 0$ for an index i, there is zero at index i of the trit vector. If `level0`$[i] = 1$, there is one or two in the trit vector. To decide if a bit on level 0 encodes one or two, we use *level 1*. The length of level 1 corresponds to the number of ones and twos in the trit vector. If `level1`$[j] = 0$ for an index j, the j-th one in level 0 encodes a trit with value one, otherwise it encodes two. For example the trit vector $T = 1012020$ is encoded as `level0` $= 1011010$ and `level1` $= 0011$. Similar to access, rank and select operations on wavelet trees (compare to Section 2.8), these operations can be solved in constant time by rank and select queries on the bit vectors. Worst case, there is a rank or select operation on level 0 that got a call of a rank or select on level 1 as parameter. Our trit vector implementation needs n bits for level 0 where n is the length of the trit vector and $o(n)$ to support rank and select queries and $t + o(t)$ bits for level 1 where $t \leq n$ is the number of ones and twos in the trit vector. We need at most $2 \cdot 63$ additional bits, because the two bit vectors in our implementation use 64-bit blocks.

4.4 Permutation

A permutation of a set $M \subset \mathbb{N}_0$ is an arbitrary fixed arrangement of the elements of M. We want to store an arbitrary permutation π of the set $\{0,1,\ldots,n-1\}$ and offer the following operations:

- `read`(π,i): returns $\pi[i]$, i.e. the value at index i in π, and
- `inverse`(π,k): returns $\pi^{-1}[i]$, i.e. the index i in π with $\pi[i] = k$.

Our implementations get the permutation as `std::vector<uint64_t>` and store it in a naive or succinct way to support the required operations.

Naive Representation for Permutations

A permutation can be stored without any support data structures. In this case $\pi[i]$ is answered by direct access to the permutation. But for $\pi^{-1}[i]$ a linear search on the whole permutation is necessary.
To reduce runtime, our naive representation stores the permutation itself and in addition the inverse. Therefore, two `int_vectors` are used with $s = n$ and $w = \log n$ (compare to Section 4.1) where n is number of elements in the permutation. The representation needs maximal $\mathcal{O}(n \log n)$ bits. The operations read and inverse require $\mathcal{O}(1)$ time, because just a single request to the corresponding vector is necessary.

Succinct Representation for Permutations

If we only need the permutation and its inverse there is a simple data structure [Nav16] to reduce space compared to the naive approach above. We still have to store the permutation, but we do not need to store every inverse any more. First observe that to compute `inverse`(π, k) we can call $i_0 = k$ and compute $i_1 = \pi[i_0], i_2 = \pi[i_1] = \pi[\pi[i_0]]$, and so on, until $i_l = \pi[i_{l-1}] = k$. The inverse is i_{l-1}. In the worst case we need to check every number and need linear time. If we look at one cycle, we can pre-compute shortcuts every b elements (see Figure 4.1). If we want to find the inverse we follow the cycle until we can take a shortcut. Then we look for i_{b-1} from there. This guarantees, that we only need b steps until we find the inverse. Navarro suggests to store a bit vector, that stores whether there is a shortcut from a given number in the permutation. The shortcuts are stored in an array and the correct index in this array can be computed with a rank query on the bit vector.
This data structure can be constructed by three iterations over the given permutation. One iteration to copy the input permutation into the first integer vector, the second iteration to calculate the shortcut positions and set these in the bit vector and in the last iteration we fill the integer vector of the shortcuts. This takes $\mathcal{O}(n)$ time where n is number of elements in the permutation. The read operation is solved by direct access on the permutation in $\mathcal{O}(1)$ time. The runtime to calculate the inverse depends on the distance b between the shortcuts, it is $\mathcal{O}(b)$ time.
Our implementation, called `permutation_inverse`, offers to choose the shortcut distance b by an integer class template parameter. We store the permutation itself as `int_vector` with size n and width $\log n$ (compare to Section 4.1). The static bit vector to mark the indices with a shortcut has n entries and needs $o(n)$ additional bits to support rank_1 queries (compare to Section 4.2.2). The positions of the shortcuts are stored as `int_vector` with size $s \leq \lfloor n/b \rfloor$ and width $w = \log n$ for a shortcut distance of b. The size s of the shortcuts vector is equal to the real number of shortcuts, worst case this is $\lfloor n/b \rfloor$.
If the input permutation is the identity, i.e. $\pi[i] = i$ for every index i, we just use the default constructor and reduce to construct the succinct representation. A global boolean variable mark this case, so the operations can evaluate if they return $\pi[i] = i$ or respectively $\pi^{-1}[k] = k$.

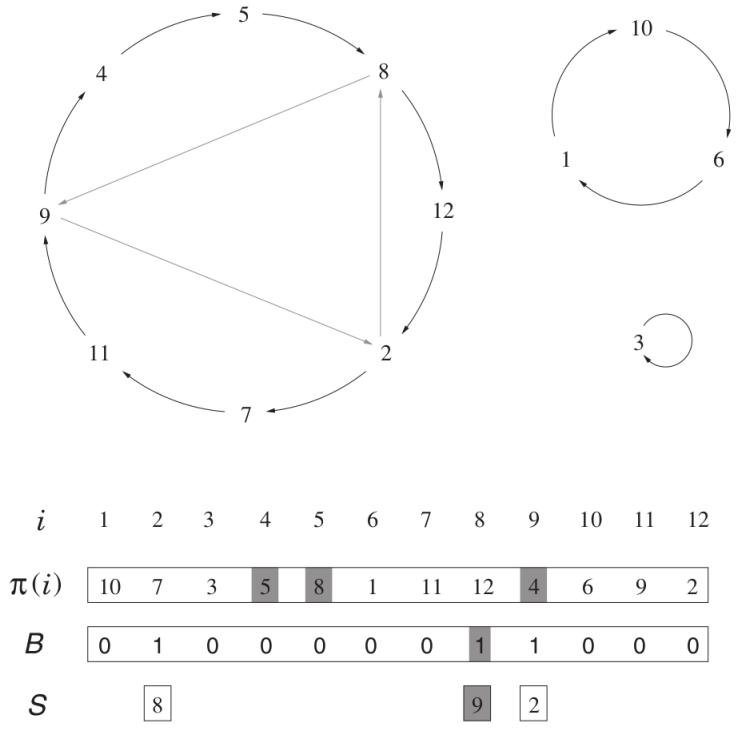


Figure 4.1: Data structure that can calculate $\pi[i]$ in constant time and $\pi^{-1}[i]$ in $O(b)$ time. A shortcut is stored every three elements in a cycle, so $b = 3$. It stores the permutation $\pi[i]$, a bit vector B and the shortcuts S, (Figure taken from [Nav16].)

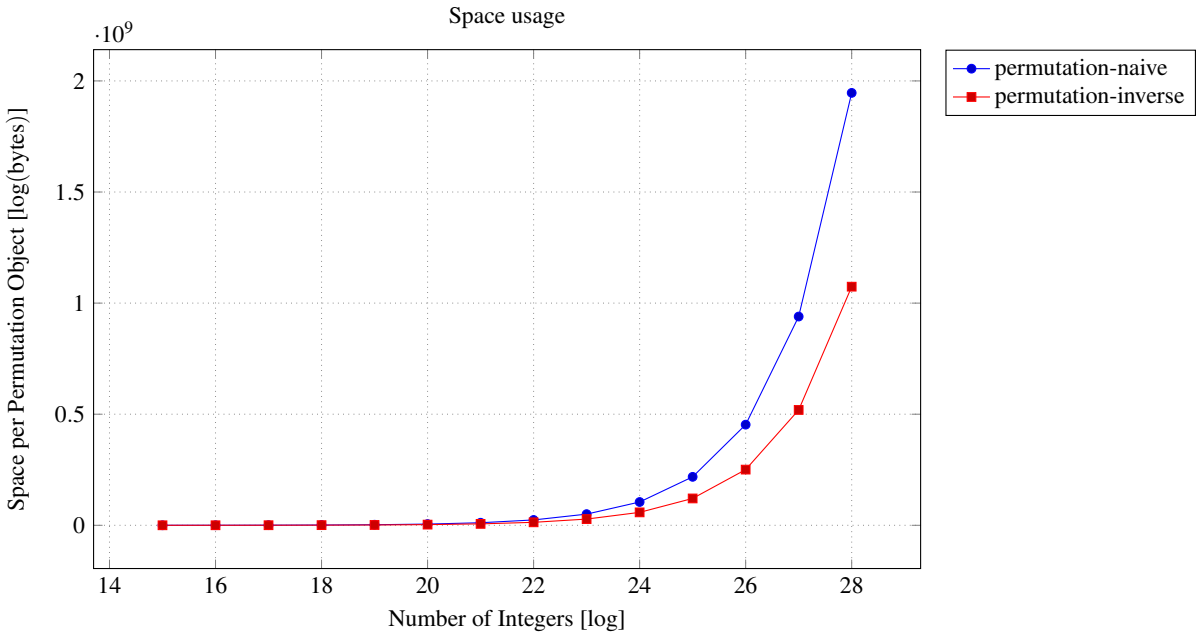


Figure 4.2: Benchmark results for the space requirement of our permutation implementations

Comparison

We benchmark our implementations of both representations by generating a `std::vector<uint64_t>` of size $n = 2^k$ with $15 \leq k \leq 28$. The vector is filled with values from 0 to $n-1$ and shuffled by `std::random_shuffle`. To benchmark the read and inverse operation we call these functions one million times for random indices. We decided to choose $b = 16$ for the shortcut distance of the succinct representation to yield constant runtime for the inverse.

Figure 4.2 shows the comparison of our naive implementation (two integer vectors) and `permutation_inverse` (shortcuts) space-wise. The space requirement of both grows quadratic. In Figure 4.3, we see that the naive representation requires significantly less construction time. In the theoretical worst case comparison both need linear construction time. But while permutation-naive just iterates over the input permutation and copies the content to two integer vectors, permutation-inverse iterates three times to copy the permutation, to fill the bit vector and to fill the shortcut vector. The differences in the running times of the read operations are not significant (compare to Figure 4.4). The succinct permutation needs to calculate the inverse instead of just one access operation by the naive one. But both yield a low constant running time even for high sizes (compare to Figure 4.5).

Therefore, we decide to use `permutation_inverse<16>` if a permutation data structure is needed.

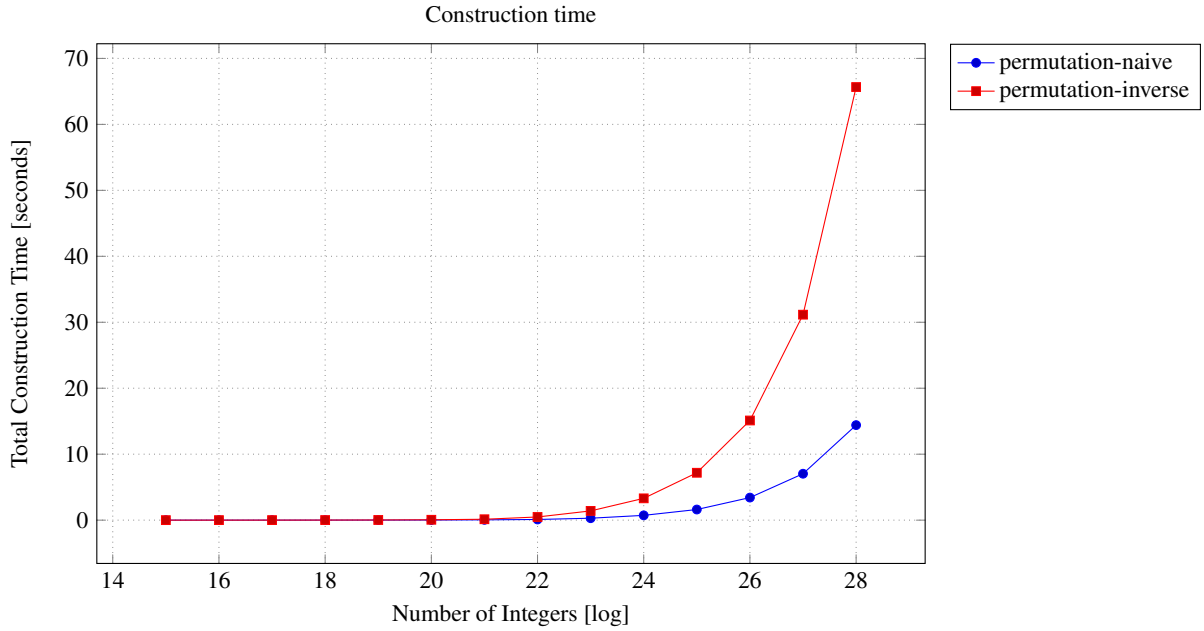


Figure 4.3: Benchmark results for the construction time (in seconds) of our permutation implementations

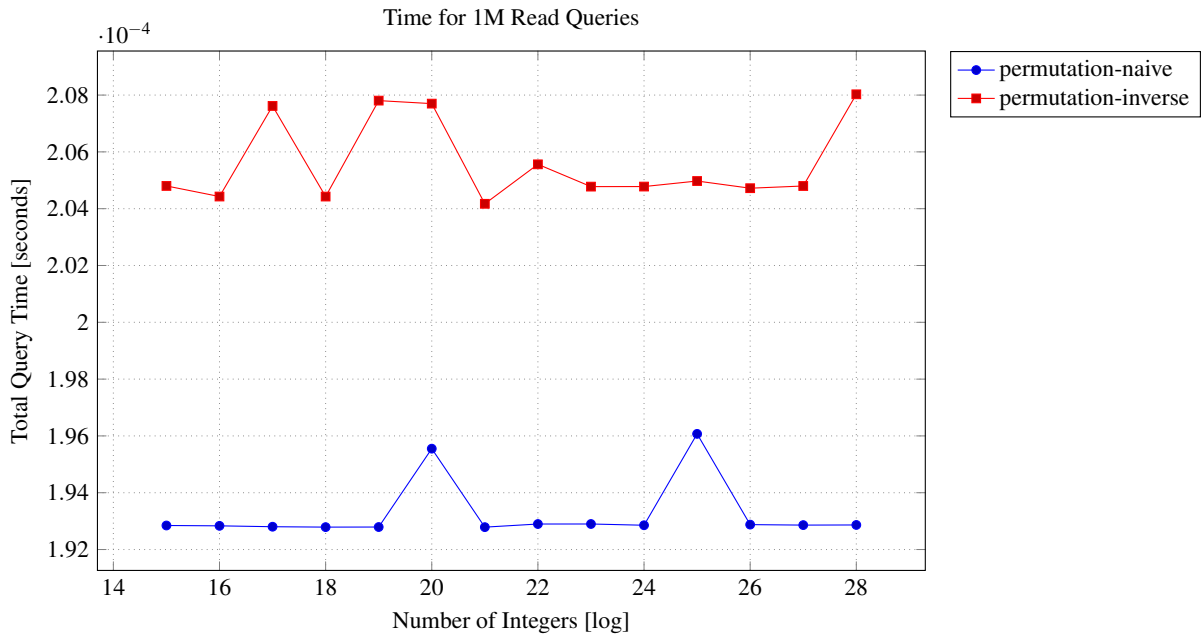


Figure 4.4: Benchmark results for runtime (in seconds) for one million read queries of our permutation implementations

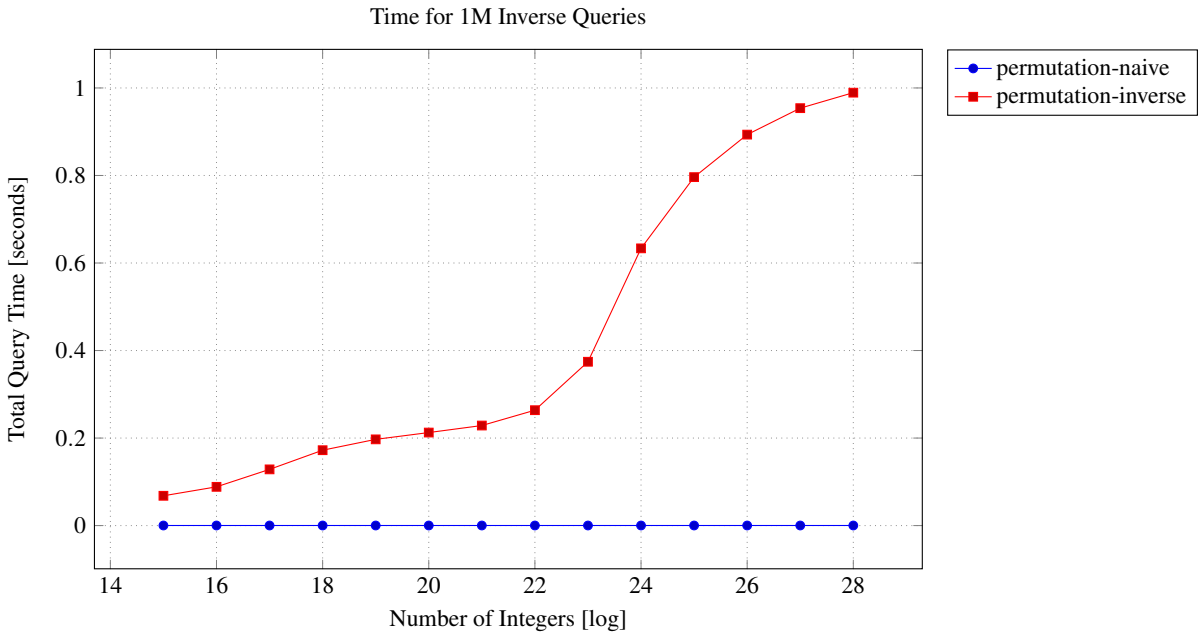


Figure 4.5: Benchmark results for runtime (in seconds) for one million inverse queries of our permutation implementations

Chapter 5

Dynamic Bit Vector

In Section 2.5, we saw how to store bit vectors supporting rank and select operations in a compact manner. In order to allow fast rank and select queries, some additional support structures need to be constructed on top of the bit vector. These support structures become invalidated whenever the contents of the bit vector change, requiring reconstruction. Additionally, as the bit vector is stored as a contiguous array of integers in memory, its size cannot easily be changed once constructed. We rectify these problems by using an alternate representation of bit vector, Dynamic Bit Vectors. They were presented in [Nav16, Section 12.1] and use a binary tree structure to allow for both fast queries and fast updates. Every leaf stores a pointer L to an array of bits. From left to right, the bit arrays of the individual leaves form the complete bit vector. Every leaf may store up to $\Theta(w^2)$ bits, where w refers to the number of bits in a computer word. Ordinarily, w is a 64-bit word in modern computer systems (see Section 2.1). Each inner node of the tree contains two attributes, *num* and *ones*. The attribute *num* holds the total number of bits contained in the leaves of the left subtree of the node. Similarly, *ones* contains the number of bits in the left subtree that are set. These attributes are used for traversal and must be kept up to date as the bit vector is modified. The extra space incurred by the tree nodes on top of the n bits of the bit vector is $\mathcal{O}(n/w)$ bits. As each inner node holds four attributes, each taking w bits of space, the space incurred by a node is $4w$ bits. There are $\mathcal{O}(n/w^2)$ nodes in total. In order to achieve fast navigation of the tree nodes, we use a balanced search tree, specifically an AVL tree (see Section 2.7). This ensures finding a leaf can always be done in logarithmic time.
If only set operations are required, i.e. no insertion or deletion, the size of a bit vector will always stay the same. In this case, it is possible to store all bits in a simple array. Instead of a binary tree, a heap can be used for the node attributes, so that the pointers to the child nodes do not need to be stored any more. This avoids the need to store pointers to the left and right subtrees of a node, saving memory space [CLRS09a, Chapter 6.1].

5.1 Construction

A dynamic bit vector can trivially be constructed by starting with an empty tree, in which bits are repeatedly inserted into. We will see that insertions on this data structure can be performed in logarithmic

Input: index i
Output: $B[i]$

```
1: v ← T.root
2: while v is not a leaf do
3:     if i ≤ v.num then
4:         v ← v.left
5:     else
6:         i ← i − v.num
7:         v ← v.right
8:     end if
9: end while
10: return v.L[i]
```

Algorithm 5.1: Operation Access for a dynamic bit vector [Nav16].

time, resulting in a required construction time of $\mathcal{O}(n \log n)$. It is much faster to construct a dynamic bit vector B from an existing bit array B' by splitting B' into $\mathcal{O}(n/w^2)$ leaves, and constructing the tree structure bottom up. To accomplish this, we first create the leaves from the given bit array, and then repeatedly join two adjacent nodes under a common parent, until we reach the root. For each node v, the correct values for the attributes *num* and *ones* can be determined recursively. The method for obtaining the attributes is shown in Equation (5.1) and Equation (5.2). For a bit array pointed to by L, $num(L)$ returns its size in bits. Correspondingly, $ones(L)$ returns the number of set bits in L. We also refer to an attribute a of a node v with $v.a$.

$$num(v) = \begin{cases} v.left.num + num(v.left.right) & \text{if } v \text{ is not a leaf} \\ num(v.L) & \text{otherwise} \end{cases} \tag{5.1}$$

$$ones(v) = \begin{cases} v.left.ones + ones(v.left.right) & \text{if } v \text{ is not a leaf} \\ ones(v.L) & \text{otherwise} \end{cases} \tag{5.2}$$

5.2 Queries

All queries are handled in a similar manner. For a dynamic bit vector B, *access(B, i)* returns the value of the bit at index i. The operations $\text{rank}_b(B,i)$ and $\text{select}_b(B,i)$ are analogous to static bit vectors. Unlike with static bit vectors, no separate data structures are required to support select_0 and select_1. For $\text{access}(B,i)$, we start at the root v. If $v.num$ is less than or equal to i, we descend into the left subtree. Otherwise, we subtract $v.num$ from i and descend into the right subtree, skipping the bits in the left tree. Once a leaf is reached, the bit is read with the methods discussed in Section 2.5.1. The procedure is outlined in Algorithm 5.1. The binary search tree guarantees $\mathcal{O}(\log n)$ time for traversal, and the bit array can be read in constant time, making access operations require $\mathcal{O}(\log n)$ time total.

Input: index i
Output: $\text{rank}_1(B,i)$

```
1:  v ← T.root
2:  ones ← 0
3:  while v is not a leaf do
4:      if i ≤ v.num then
5:          v ← v.left
6:      else
7:          i ← i − v.num
8:          ones ← ones + v.ones
9:          v ← v.right
10:     end if
11: end while
12: return ones + rank_1(v.L, i)
```

Algorithm 5.2: Operation Rank for a dynamic bit vector [Nav16].

In the case of $\text{rank}_0(B,i)$, it can once again be computed as $i - \text{rank}_1(B,i)$. For rank_1, we traverse the tree as with access, but maintain a counter of set bits named *ones*. If we descend into a right subtree, we skip the ones from the left subtree, so we add *v.ones* to *ones* to compensate. *v.L* is processed as it would be in the static case: we can process the array one word at a time using *popcount* operations. As the array contains $\Theta(w^2)$ bits, it can be processed in $\mathcal{O}(w)$ steps, making the entire operation take $\mathcal{O}(w)$ time. Algorithm 5.2 shows rank_1 in detail.

The operation $\text{select}_b(B,j)$ is performed slightly differently. We maintain a counter *pos*, which represents the total number of bits skipped in the left subtrees. If $j \leq$ *v.ones*, that means the desired bit must be in the left subtree. Otherwise, the desired bit is in the right subtree, so *v.num* is added to *pos*, and *v.ones* is subtracted from j. For select_0, *v.num* $-$ *v.ones* conveniently equals the number of clear bits in the left subtree, so *v.num* $-$ *v.ones* is used for deciding which subtree to descend into. Accordingly, j is set to $j - ($*v.num* $-$ *v.ones*$)$ instead. Once at a leaf, we look for the j-th 1 or 0. This can once again be done in $\mathcal{O}(w)$ time using *popcount* operations: we scan *v.L* word-wise until we come across the word that contains the desired bit. The final word may then be scanned bitwise, taking a maximum of $\mathcal{O}(w)$ steps. The pseudocode for select_b is shown in Algorithm 5.3.

5.3 Update Operations

Updating a bit vector means deleting, inserting or flipping a bit. Since insertions increase the size of a leaf, we have to pay special attention to the maximum allowed leaf size. As aforementioned, every leaf may store up to $\Theta(w^2)$ bits. We allow the leaf size to be less than $2w^2$. Regarding deletions that reduce the leaf size, we require every leaf to contain at least $w^2/2$ bits. This does not need to be adhered if the whole tree consists of only one root node.

Input: index j
Output: $\text{select}_{0|1}(B, j)$

1: $v \leftarrow T.root$
2: $pos \leftarrow 0$
3: **while** v is not a leaf **do**
4: **if** $j \leq \{(v.num - v.ones) \mid v.ones\}$ **then**
5: $v \leftarrow v.left$
6: **else**
7: $j \leftarrow j - \{(v.num - v.ones) \mid v.ones\}$
8: $pos \leftarrow pos + v.num$
9: $v \leftarrow v.right$
10: **end if**
11: **end while**
12: $p \leftarrow \text{select}_{0|1}(v.L, j)$
13: **return** $pos + p$

Algorithm 5.3: Operation Select for a dynamic bit vector[Nav16].

5.3.1 Inserting a Bit

Given a bit b and an index i for the position, the procedure of inserting can be described as follows: first of all, we determine the correct leaf to insert into and the position within that leaf as we would in an access operation. Whenever the left subtree of a node v is visited, the *num* attribute of the node has to be increased by one. If the handled bit is set, *ones* has to be increased by one, too. Arriving at the leaf into which b is to be inserted, the size of the corresponding bit vector has to be checked. If by inserting the new bit the maximum leaf size of $2w^2$ is reached, the respective bit vector has to be split into two halves. The current leaf gets converted into an inner node and becomes the parent node of two new leaves: the left child represents the first half of the split bit vector and the right child the second half. The attributes *num* and *ones* must be updated accordingly. The last step of this procedure is checking whether the tree structure is a balanced AVL tree. If the balance factor is not an element of the set $\{1, 0, -1\}$, then the tree has to be rebalanced. Here it is important to adjust the values of *num* and *ones* correctly. If v_2 is the right child node of v_1, then performing a left rotation requires the following updates on v_2, which now represents the new root: $v_2.num \leftarrow v_2.num + v_1.num$ and $v_2.ones \leftarrow v_2.ones + v_1.ones$. For a right rotation, let v_1 be the left child node of v_2. The attributes of v_2 are changed as follows: $v_2.num \leftarrow v_2.num - v_1.num$ and $v_2.ones \leftarrow v_2.ones - v_1.ones$.
Traversing and rebalancing the tree is done in $\mathcal{O}(\log n)$ time. When inserting a bit into a leaf, all bits after the insertion point in the leaf must be shifted to the right by one. Since bits are shifted word-wise, inserting a bit at position i requires $\mathcal{O}(w)$ time. As $w \geq \log n$, we have an upper bound of $\mathcal{O}(w)$. The following gives an overview of the described steps. Hereby the de-/incrementation of both attributes *num* and *ones* is not included.

1. If i is less than or equal to *num*, move down to the left node. Otherwise move down to the right node. Repeat this step recursively until a leaf is reached.

2. After the insertion of the bit b at position i, two cases have to be considered:

   - **Case 1:** The size of the leaf node is less than $2w^2$. In this case nothing has to be done since the required maximum size is not violated.
   - **Case 2:** The size of the leaf node is equal to $2w^2$. In this case, the node has to be split into two leaves that now represent the left and right child of the current node.

3. Rebalance the tree using one of the rotations mentioned in Section 2.7.2 if the condition described in Definition 2.7.2 is no longer satisfied. Since a recursive approach is used, every node is examined after exiting the recursion.

An example of the procedure for inserting a bit is visualized in Figure 5.1.

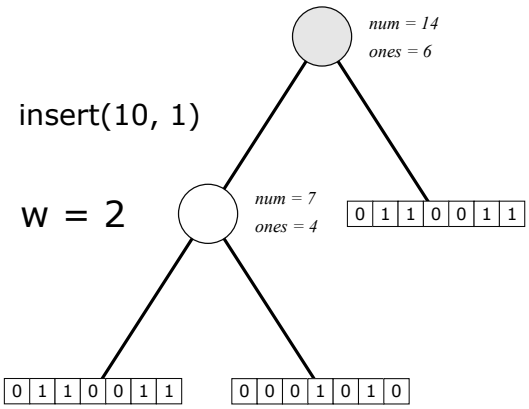
5.3.2 Deleting a Bit

To delete a bit at position i, the tree is traversed as in the insert operation. If after deletion the leaf size equals $w^2/2-1$, so that the minimum leaf size of $w^2/2$ is no longer met, a bit needs to be *stolen* from another leaf or leaves are merged. The attributes *num* and *ones* must be decremented accordingly. It is only clear after the bit has been determined whether a set or a clear bit is deleted and the corresponding attributes can only be adjusted afterwards. As in the case of an insertion, the delete operation is implemented using a recursive algorithm. Thus, the attributes can be adapted when leaving the recursion.
The tree has to be rebalanced if necessary. Like the tree traversal, this is done in $\mathcal{O}(\log n)$ time. As aforementioned, bits are shifted word-wise. Hence, the overall time complexity again corresponds to $\mathcal{O}(w)$. Given this information, the procedure of the process can be explained:

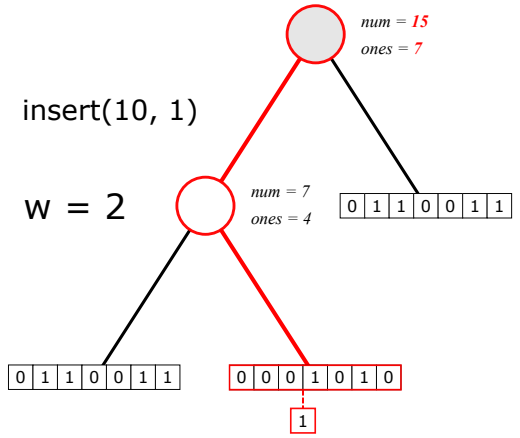
1. If i is less than or equal to *num*, move down to the left node. Otherwise move down to the right node. Repeat this step recursively until a leaf is reached.

2. When deleting a bit at position i, two cases have to be considered:

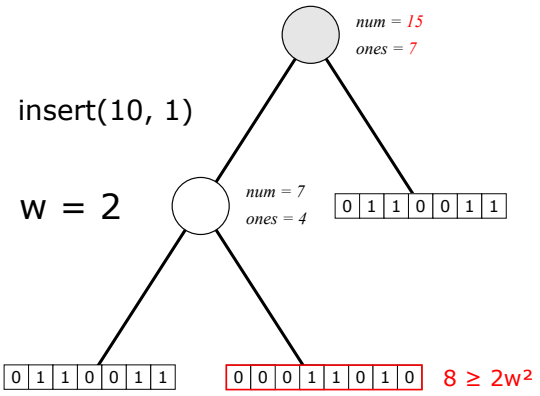
   - **Case 1:** The size of the leaf node is greater than $w^2/2$. In this case, remove the bit at position i.
   - **Case 2:** The size of the leaf node is equal to $w^2/2$ and removing a bit results in a size of $w^2/2-1$. If the current leaf node is a left child, move to the next leaf positioned on the right side. If the current leaf node is a right child, move to the next leaf positioned on the left. Here again, two cases are considered:
     - **Case 2a:** The found leaf, which stores a pointer $v'.L$, has a size greater than $w^2/2$ and we are able to steal a bit from it. The position to be stolen from depends on the position of our original leaf node storing $v.L$. If $v.L$ is a left child, the first bit of $v'.L$ is removed. On the other hand, if $v.L$ is a right child, the last bit of $v'.L$ is removed and inserted into $v.L$.



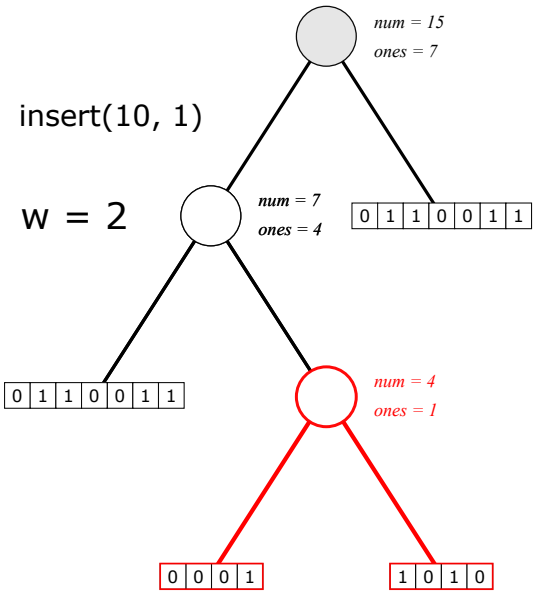
(a) Inserting the bit 1 at position 10.



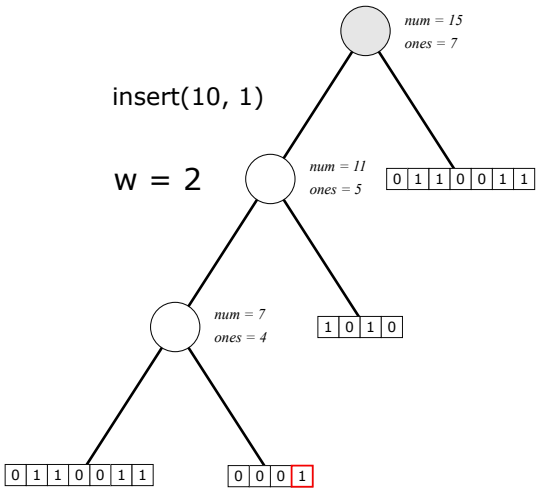
(b) Go left ($10 < num{=}15$), then right ($10 > num{=}7$)



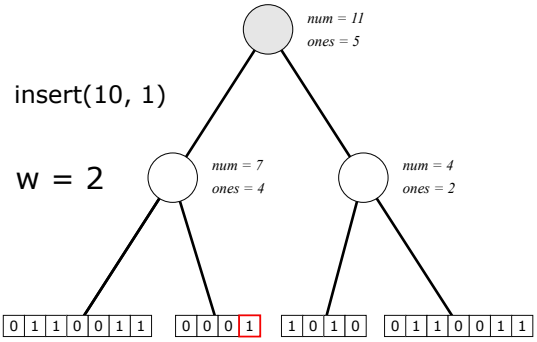
(c) After inserting the bit, the maximum leaf size is reached.



(d) The leaf node is split into two leaves.



(e) After splitting, balance has to be restored.



(f) Now the tree is balanced.

Figure 5.1: The procedure of inserting a bit into a tree structure.

- **Case 2b:** The found leaf node, which stores a pointer $v'.L$, has a size of $w^2/2$. In this case, stealing a bit from it is not possible and our leaf, storing $v.L$, has to be merged with it by inserting the contents of $v.L$ into $v'.L$. Since $v.L$ has a size of $w^2/2-1$, the final size of the newly created leaf will not exceed $2w^2$. After merging the two leaves, the leaf node storing $v.L$ is deleted.

3. Rebalance the tree using one of the rotations mentioned in Section 2.7.2 if the condition described in Definition 2.7.2 is no longer satisfied. Since a recursive approach is used, every node is examined after exiting the recursion.

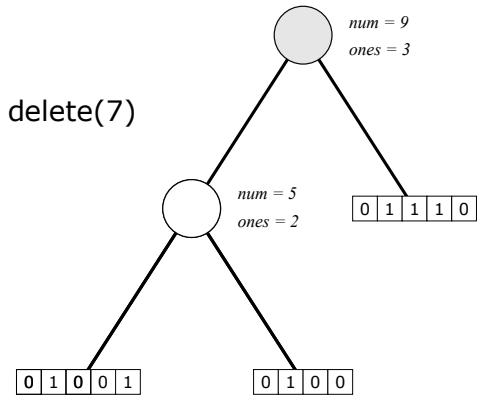
In Figure 5.2, an example of the procedure for deleting a bit is visualized.

5.3.3 Flipping a Bit

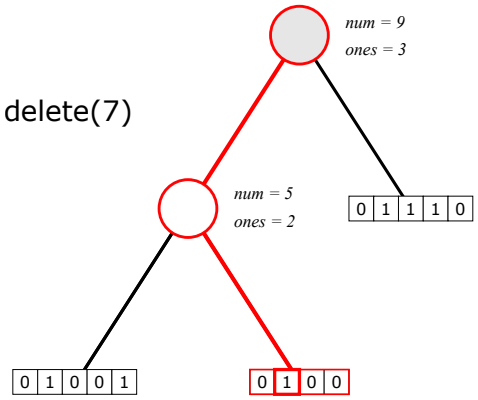
The update operation of setting a bit can easily be performed by deleting a bit and then inserting the new bit at this position. This can be done in $\mathcal{O}(w)$ time.
However, it is more efficient to change the bit in one operation and to adjust the attributes accordingly. If the handled bit is changed from clear to set, the attribute *ones* has to be incremented. On the other hand, changing a set bit to a clear bit means decrementing *ones*. As previously shown in the delete operation, the way in which the node attributes have to be modified becomes only apparent after finding the required bit. Therefore, the recursive approach demonstrated in the pseudocodes of *bitset* and of *bitclear* that are listed in Algorithms 5.4 and 5.5 is used.
When calling bitset(v,i), a tree node v is traversed until the leaf which holds position i is reached (lines 1-11). This again is done in $\mathcal{O}(\log n)$ time. In the case that the bit at position i is set, `false` is returned, as no bit had to be changed (lines 12-13). Otherwise, the bit is set to one and `true` is returned (lines 15-16). In this case, the attribute *ones* is incremented by one (line 5).
The procedure of bitclear(v,i) is similar to that of bitset: if the bit at position i is a clear bit, `false` is returned (lines 12-13). Otherwise, the bit is changed from set to clear and `true` is returned (lines 15-16). The attribute *ones* then is decremented by one (line 5).
Since setting and reading a bit requires only constant time, the overall time complexity of the two operations corresponds to $\mathcal{O}(\log n)$.

5.4 Compressed Dynamic Bit Vectors

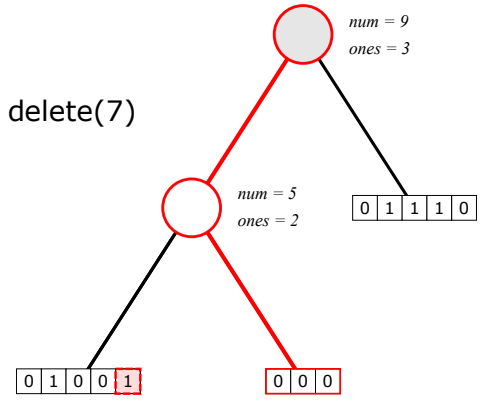
In this section we examine another variant of dynamic bit vectors, presenting different methods that allow reducing space usage. This idea of *compression* is based on the zero-order entropy $\mathcal{H}_0$ mentioned in Section 2.9 and presented in [Nav16, Section 4.1.1]. We start by explaining the basic concept of compressing bit vectors and finish by applying it to our data structure.



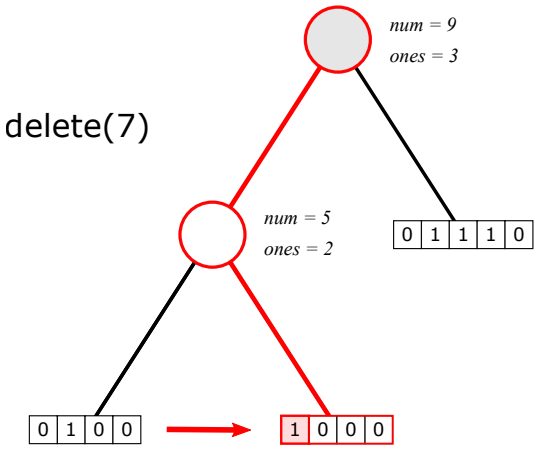
(a) Deleting the bit at position 7.



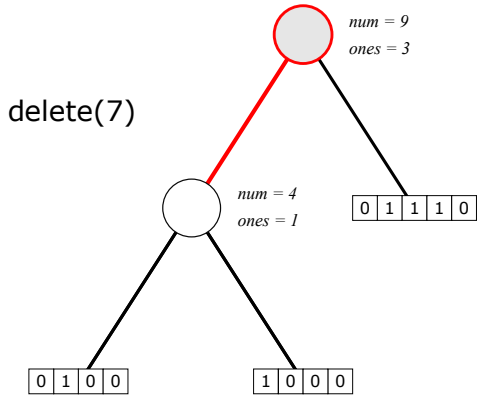
(b) Go left ($7 < num{=}9$), then right ($7 > num{=}5$)



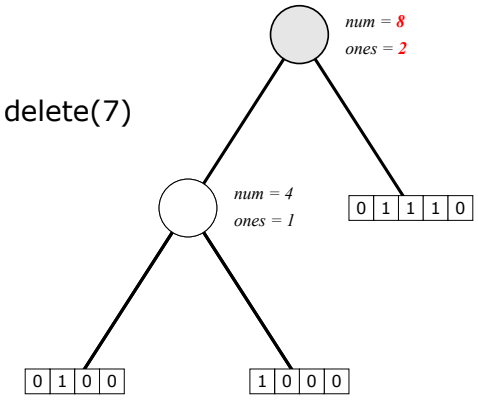
(c) After deleting the bit, the minimum leaf size is violated.



(d) Therefore a bit from the left neighbor ist stolen.



(e) After stealing the bit, *num* and *ones* are updated.



(f) The attributes *num* and *ones* of the root node are updated. No balancing is required.

Figure 5.2: The procedure of deleting a bit from a tree structure.

Input: tree node v and index i

Output: sets the bit at index i below v to one and returns if it changed

```
1: if v is not a leaf then
2:    if i ≤ v.num then
3:       flipped ← bitset(v.left, i)
4:       if flipped then
5:          v.ones ← v.ones + 1
6:          return flipped
7:       end if
8:    else
9:       return bitset(v.right, i − v.num)
10:   end if
11: end if
12: if v.L[i] = 1 then
13:   return false
14: end if
15: v.L[i] ← 1
16: return true
```

Algorithm 5.4: Operation Bitset for a dynamic bit vector [Nav16].

5.4.1 Bit Vectors and Zero-Order Compression

Compression in our case affects only the leaves of our tree structure. Therefore, we are concerned with transforming the leaf bit vectors in such a way that space reduction is guaranteed. However, for this to be achieved, certain requirements concerning the values of the bit vector must be satisfied: as already stated in Section 2.9, the lowest possible bound for compressing a bit vector B corresponds to $n\mathscr{H}_0(B)$ bits. In the case that B contains about as many ones as zeroes, $n\mathscr{H}_0(B)$ will be approximately n and space as well as time will be increased. Consequently, the method described in this section is only recommended if the following condition is met with respect to a bit array: $\#ones \ll \#zeroes$ or $\#zeroes \ll \#ones$.

Transformation of the Bit Vector

To begin, a bit vector B is divided into *blocks* B_i of length b each, where $0 \leq i < \lceil n/b \rceil$. The number of ones in a block B_i is called the *class* c_i of the block. The number of all possible representations of a block is described by the *binomial coefficient* $\binom{b}{c_i}$. Thus, each class c_i has $\binom{b}{c_i}$ elements.

5.4.1 Example. *Given a block length of $b = 8$, class $c = 1$ has eight elements, since $\binom{8}{1} = 8$. There are eight possible representations of the block:* 00000001, 00000010, 00000100, 00001000, 00010000, 00100000, 01000000, 10000000.

Apart from the class c_i of a block, there is another important number: the *offset* o_i, with $o_i < \binom{b}{c_i}$, serves as an identifier, allowing to identify a block among all possible block variants belonging to a class c_i.

Input: tree node v and index i
Output: sets the bit at index i below v to zero and returns if it changed

```
1:  if v is not a leaf then
2:      if i ≤ v.num then
3:          flipped ← bitclear(v.left, i)
4:          if flipped then
5:              v.ones ← v.ones − 1
6:              return flipped
7:          end if
8:      else
9:          return bitclear(v.right, i − v.num)
10:     end if
11: end if
12: if v.L[i] = 0 then
13:     return false
14: end if
15: v.L[i] ← 0
16: return true
```

Algorithm 5.5: Operation Bitclear for a dynamic bit vector [Nav16].

Each offset o_i requires $\left\lceil log\binom{b}{c_i} \right\rceil$ bits to store. The more balanced the ratio between the number of ones and the number of zeroes is, the larger the offset becomes. A ratio of 0.5 represents the worst case.

Encoding and Decoding Methods

In order to obtain the class and offset representations of a bit block, we employ a method to convert each block of the bit vector. However, it is also essential to be able to reconvert the classes and offsets into the previous form of the block.
The two algorithms for *encoding* bit blocks and *decoding* classes and offsets are shown in Algorithm 5.6 and Algorithm 5.7. Both algorithms make use of a global two-dimensional table K, which consists of precomputed binomial coefficients and is defined as follows:

$$K[i][j] = \binom{i}{j} \; \forall \, i, j \; . \; 0 \leq i \leq b \wedge 0 \leq j \leq b \tag{5.3}$$

The space used by table K adds up to b^2 integers, the time required by each algorithm meets $\mathcal{O}(b)$.
The idea behind Algorithm 5.6 can be explained as follows: since it suffices to count the ones in the given block in order to obtain the classes (line 4), the focus lies on computing the offset values. Offsets are assigned to classes in increasing order of the block integers. There exist $\binom{b-1}{c}$ bit blocks that start with zero, followed by $\binom{b-1}{c-1}$ blocks that start with one. Using this information, we start with setting offset o to zero (line 7) and check if the first bit of the block corresponds to one (line 11). If this is the

Input: bit array $B[1,b]$ (an integer), block length b
Output: pair (c,o) which encodes B

1: $c \leftarrow 0$
2: **for** $j \leftarrow 1$ to b **do**
3: **if** $B[j] = 1$ **then**
4: $c \leftarrow c+1$
5: **end if**
6: **end for**
7: $o \leftarrow 0$
8: $c' \leftarrow c$
9: $j \leftarrow 1$
10: **while** $0 < c' \leq b - j$ **do**
11: **if** $B[j] = 1$ **then**
12: $o \leftarrow o + K[b-j][c']$
13: $c' \leftarrow c' - 1$
14: **end if**
15: $j \leftarrow j+1$
16: **end while**
17: **return** (c,o)

Algorithm 5.6: Operation Encode for a dynamic bit vector [Nav16].

case, we add $\binom{b-1}{c}$ to offset o, skipping over all offset values that belong to blocks which start with zero (line 12). If the first bit of the block corresponds to zero, we do not perform any modifications on o.
Algorithm 5.7 employs the mechanism of encode, but in the reverse direction. If the condition $o \geq \binom{b-1}{c}$ (line 4) is not true, then the block starts with zero and a decremented b is used for the next iteration step to consider the succeeding bits. On the other hand, if $o \geq \binom{b-1}{c}$ holds, then the current block starts with one and lines 5-7 are executed, setting the bit to one (line 5) and reducing the offset (line 6) as well as the class (line 7) to continue with the rest of the block.

Encoding and Decoding in Constant Time

In order to decode an offset in constant time, a lookup table T_c can be used storing all b block integers that belong to class c, as well as their corresponding offset values. Table T consists of all tables T_c regarding a block length b. As table T_c contains $\binom{b}{c}$ entries, the overall number of entries of T add up to $\sum_{c=0}^{b}\binom{b}{c} = 2^b$. Consequently, depending on the value chosen for b, storing every single table T_c can increase space usage.
In Table 5.1 an excerpt of table T with respect to a block length of $b = 4$ is shown, demonstrating the two tables T_0 and T_1 . Here the rows represent the classes and the columns represent the offsets, such that each line stands for a table T_c. For example, obtaining the corresponding block of class $c = 1$ and offset $o = 2$ would mean calling $T[1][2] = 0100$.

Input: class c, offset o, block length b
Output: bit array B (an integer) decoding (c, o)

```
1:  B ← 0
2:  j ← 1
3:  while c > 0 do
4:      if o ≥ K[b − j][c] then
5:          B[j] ← 1
6:          o ← o − K[b − j][c]
7:          c ← c − 1
8:      end if
9:      j ← j + 1
10: end while
11: return B
```

Algorithm 5.7: Operation Decode for a dynamic bit vector [Nav16].

		offset				
		0	1	2	3	⋯
class	0	0000				
	1	0001	0010	0100	1000	
	⋮	⋯	⋯	⋯	⋯	⋯

Table 5.1: Table T containing tables T_0 and T_1, with $b = 4$

For encoding an offset in constant time, a table similar to table T can be used, storing the blocks as keys to identify their associated offsets.

5.4.2 Application

When applying the compression scheme to our data structure, the attributes *num* and *ones* remain unchanged. Traversing the tree until a leaf is reached is still accomplished in $\mathcal{O}(\log n)$ time, the process of rebalancing the AVL tree continues to be executed. Each leaf vector is divided into blocks. As a result, the leaves of our compressed data structure no longer consist of bit vectors, but of class components c and offset components o describing the blocks of a bit vector. When trying to access or modify the tree, decoding a pair (c, o) is always necessary. By this means, accesses and modifications of the data structure like getting or deleting a bit can still be performed.
The minimum and maximum leaf sizes are still defined in terms of bits represented, not physical bits used. Consequently, the space used corresponds to $\mathcal{O}(n/w) + n\mathcal{H}_0(B) + o(n)$ bits.

class c	0	1	2	3	4	5	6	7	8
L[c]	**0**	**3**	**5**	**6**	**7**	**6**	**5**	**3**	**0**

Table 5.2: Table L for a block length of $b = 8$

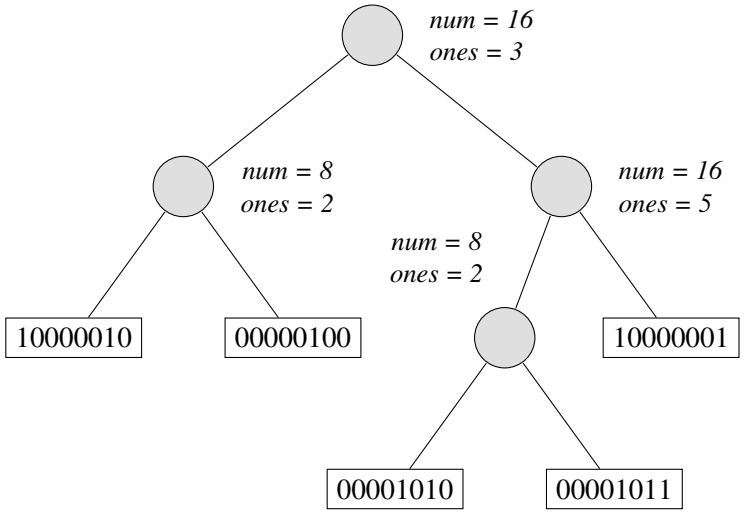


Figure 5.3: Example for an uncompressed dynamic bit vector with
$B = 10000010\ 00000100\ 00001010\ 00001011\ 10000001$ [Nav16].

As shown in Section 5.4.1, the amount of bits required for representing an offset plays a crucial role in achieving compression, as it can vary significantly. Here a lookup table L can be employed containing the number of bits required for representing an offset o_i in dependence of a class c_i:

$$|o_i| = L[c_i] = \left\lceil log \binom{b}{c_i} \right\rceil . \tag{5.4}$$

Table 5.2 shows table L for a block length of $b = 8$ and all corresponding classes $0 \leq c \leq b$. Obviously, the value of $L[c]$ tends to be larger the closer c is to the middle.
If the block length b corresponds to $\Theta(w)$ bits, a leaf is divided into $\mathcal{O}(w) = \mathcal{O}(b)$ blocks. Get, rank and select operations require traversing the $\mathcal{O}(w)$ class elements and decoding only one offset element in $\mathcal{O}(w)$ time. On the other hand, performing the update operations insert and delete is less efficient. When inserting or deleting a bit at a certain position, we must decode all the following blocks, update the position and finally re-encode all the decoded blocks. As this is done bitwise, the required time corresponds to $\mathcal{O}(w^2)$.

Figure 5.3 illustrates an example for an uncompressed dynamic bit vector, while Figure 5.4 shows the compressed variant of the same tree.

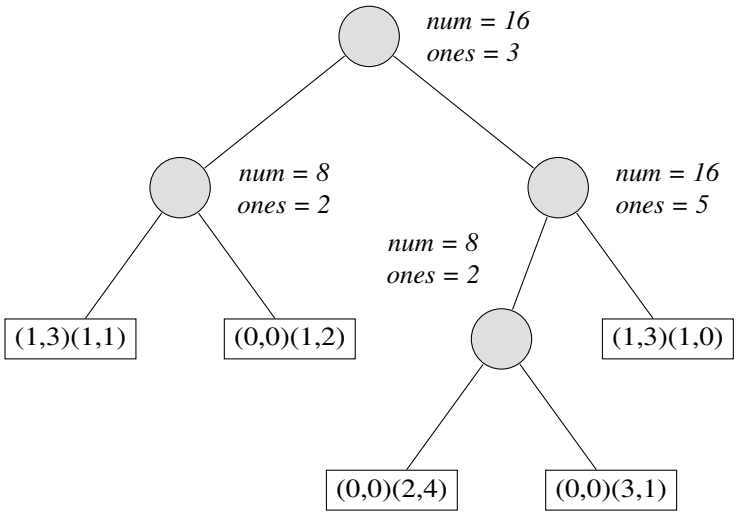


Figure 5.4: Example for a compressed dynamic bit vector, with
$B = 10000010\ 00000100\ 00001010\ 00001011\ 10000001$ and $b = 4$ [Nav16].

5.5 Implementation

We use an AVL tree (see Section 2.7) to ensure that the binary search tree remains balanced. Each leaf may hold no more than $2w^2 - 1$ bits and must hold at least $w^2/2$ bits. As a leaf grows above its maximum size with repeated insertions, we split the leaf as described in Section 5.3. We also perform a tree rotation if necessary. Correspondingly, two adjacent leaves are merged if they are both smaller than $w^2/2$ bits. As w is 64 bits on modern systems, this puts the maximum and minimum leaf sizes at 8191 and 2048 bits, respectively.

5.5.1 Construction

Trees are constructed in a bottom-up manner, as described in Section 5.1. We first divide a static bit vector into leaves that are of size w^2 bits. If the final leaf contains less than w^2 bits, it is merged with the previous leaf to ensure that no leaf is smaller than the minimum leaf size. Let $z = \lfloor n/w^2 \rfloor$ be the number of leaves that make up the new bit vector. To ensure that the tree is balanced, the tree is constructed in a heap-like structure, where the first $2(z - 2^{\lfloor \log z \rfloor})$ leaves are placed one level lower than the remaining leaves. Then, any two adjacent leaves are merged by creating a common predecessor one layer above. This process repeats until we arrive at the root.
It is also possible to construct a tree based on given pre-built leaves. Here we start at the bottom, recursively building the tree until we reach the top. However, the condition must be satisfied for the total number of given leaves to be equal to a power of two, so that the construction results in a full tree.

5.5.2 Uncompressed Leaves

For space usage, uncompressed leaves are managed by a custom-built class that is internally backed by a `std::vector<uint64_t>`. Every bit we store is assigned to one bit of a `uint64_t`. When reading

or writing a bit, we use the methods discussed in Section 2.5.1, first finding the correct word in the `std::vector` and then finding and reading the correct bit in the word. Insertions are handled by finding the affected bit using the same procedure, then inserting the bit into the word. All subsequent words are shifted to the right by one bit, and an extra word is appended to the `std::vector` if necessary. Deletions are handled analogously, shifting to the left and truncating the vector if necessary. This ensures insertions and deletions are carried out in $\mathcal{O}(w)$ time. For rank and select operations, we process the vector word-wise, using the *popcount* and *count leading zeroes* instructions discussed in Sections 2.5.2 and 2.5.3 which also results in $\mathcal{O}(w)$ time.

5.5.3 Compressed Leaves

In the case of compressed bit vectors, the data structure for the leaf changes: For maintaining the sequence of classes, we use a `int_vector` (see Section 4.1). For offsets, we use a purpose-built class called `var_vector`. A `var_vector` is a (static) bit vector which permits setting or reading up to b bits at once at any bit position. It too is backed by a `std::vector<uint64_t>`.

When encoding or decoding, we make use of a global lookup table consisting of binomial coefficients $\binom{i}{j}$, where $0 \leq i \leq b$ and $0 \leq j \leq b$ with b being the size of each block (see table K of Equation (5.3)). We choose a block size of 64.

For access operations, we must decode a block from the class and offset vectors of a leaf. The correct class in the class vector can be straightforwardly computed by dividing the bit position within the leaf by the block size. Finding the right offset however requires determining the correct bit position in the offset vector, which can only be computed by iterating through all classes up to the desired class.

Access operations therefore require $\mathcal{O}(w)$ time to carry out. Further, if a bit is changed, the class and offset of its block change, which means all subsequent offsets in the offset vector need to be rewritten into the vector at a new bit position. In order to obtain the number of bits used for an offset, we implement the table of Equation (5.4) as a global lookup table.

With insertions, we decode all blocks from the insertion point into a temporary bit vector, which is processed word-wise in $\mathcal{O}(w)$ time, as in the uncompressed case. The same procedure applies to deletions. As any block requires $\mathcal{O}(w)$ time to decode, the total runtime of insertions and deletions is $\mathcal{O}(w^2)$. However, for certain very unbalanced blocks, we store pre-calculated results in global lookup tables, reducing the decoding and encoding time for those blocks to $\mathcal{O}(1)$. Specifically, if a block contains less than 3 set bits, or more than 61 set bits, we delegate to a lookup table. As these blocks also compress well, this is a fitting combination. For decoding in constant time, we employ table T described in Section 5.4.1, for encoding in constant time, we use a table similar to T, but consisting of blocks and their corresponding offsets. In contrast to the rest of the work, we do not use a data structure of `std` for the latter table, but represent it with a `robin_hood::unordered_map` (see Section 3.1.1). In this way we ensure an even more efficient time.

To solve rank, we can read the number of bits in a block directly from its class, leaving us with decoding the final block and counting its bits using a *popcount* operation. Similarly, to solve select, we can quickly iterate through blocks by summing the number of ones or zeroes from their respective class indices, then

decoding the final word, similar to the uncompressed case. As a result, both rank and select require $\mathcal{O}(w)$ time.

5.5.4 Further Remarks on Space Usage

The use of `std::vector<uint64_t>` incurs some space overhead, as a `std::vector` may store additional attributes which are not strictly required for our purposes: In typical implementations, a vector holds three attributes of 8 bytes each, amounting to 24 bytes per `std::vector`. We also maintain a count of the bits in the leaf. Further, each node stores its own height to allow for testing if the AVL tree needs balancing. Additionally, we always store distinct pointers for the left and right children as well as a pointer to a leaf, regardless of whether a node represents a leaf or not. The attributes *num*, *ones*, *left* and *right* are unused if the node is a leaf, and *L* is unused otherwise, incurring more space overhead. We deem the trade-off to be acceptable, although further space savings could be achieved with using separate inner node and leaf types, as well as using a specialized data structure instead of `std::vector`. For compressed vectors, the space required is governed by the class vector and the offset vector. As both use a `std::vector`, the space overhead described above applies to them as well. Both vectors must also manage some attributes of their own. In total, the `int_vector` requires 48 bytes of storage, and the `var_vector` requires 32 bytes of storage. We investigate the space requirements of both data structures in detail in Section 5.6.

5.6 Benchmarks

When testing the data structures, we focus on measuring the construction time as well as on testing the efficiency of the functions delete, insert, get, set, rank and select. For all operations, we measure the throughput as bits processed per second. We measured all operations on bit vectors of different sizes, starting at a minimum of 2^{16} bits, and reaching 2^{34} bits as a maximum size.

We compare our data structures against Nicola Prezza's *DYNAMIC* library [Pre17], which is freely available online[1]. DYNAMIC offers two kinds of dynamic bit vectors which we compare against. One, *Succinct_Bitvector*, is based on searchable partial sums, while the other, *Gap_Bitvector*, is based on a gap encoding (see Section 2.11) of set bits in the bit vector. Both data structures offer functionality for all the standard operations, i.e. set, get, insert, delete, rank and select, making them suitable to compare to our bit vectors.

We refer to the ratio of set bits to the number of total bits in a bit vector as its *fill rate*. Unless otherwise noted, we assume an evenly distributed vector, i.e. a fill rate of 0.5. As mentioned in Section 5.5, for compressed bit vectors, we make use of pre-calculated lookup tables for very sparse and very dense blocks. Because of this, the performance of compressed bit vectors is highly subject to the fill rate. We will therefore examine the performance of the compressed bit vector for different fill rates in more detail in Section 5.6.4. We will also compare its performance to that of the gap bit vector, as it also exhibits different performance characteristics with different fill rates.

[1] `https://github.com/xxsds/DYNAMIC`

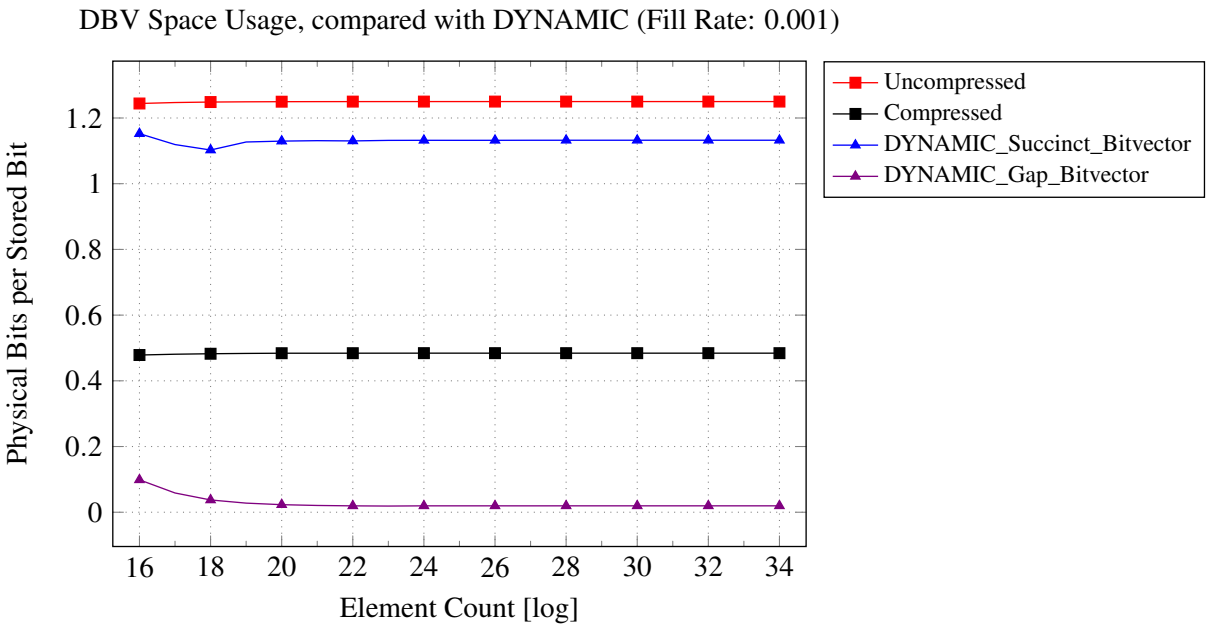


Figure 5.5: The space usage of the four tested bit vectors for a fill rate of 0.001. On the x axis, the size of the bit vector is displayed, while the y axis shows the number of bits required to encode one bit of information. For example, a factor of 2 would mean that to store 10 bits of information, the data structure requires 20 bits of memory. We see that for such a sparsely populated bit vector, the compressed bit vectors massively outperform the static succinct bit vector and our uncompressed bit vector.

5.6.1 Construction

First we examine the overall space usage of the different bit vectors. The resulting plots are displayed in Figures 5.5 to 5.7 for the fill rates of 0.001, 0.5 and 0.999, respectively. We note that for a low fill rate of 0.001, the gap bit vector of Nicola Prezza performs extremely well, which is expected, as it has only few bits to encode. Overall, it requires only 0.02 physical bits to encode 1 bit of information. Our compressed bit vector still offers a significant improvement over the other two implementations, as it uses 0.48 physical bits per stored bit for such a sparse arrangement. Nicola Prezza's succinct bit vector performs slightly better than our bit vector, using 1.13 physical bits per stored bit, whereas our uncompressed bit vector requires 1.25 physical bits per stored bit.

With an even distribution of bits, the performance of the compressed bit vector and Prezza's gap bit vector degrade strongly, while the other two data structures are not affected by the change in fill rate. The compressed bit vector now requires 1.42 physical bits of space per stored bit, whereas the gap bit vector now requires 4.62 physical bits per stored bit. All in all, we can see that our compressed bit vector still offers a reasonable space usage for such an evenly distributed bit vector, whereas the performance of the gap bit vector is unacceptable.

If the bit vector is very densely filled, the performance of the gap bit vector degrades even more. For a fill rate of 0.999, it requires 5.46 physical bits of space, whereas our compressed bit vector once more offers excellent space usage. However, if the distribution of bits is known in advance, the bit vector can easily be inverted to take advantage of the gap bit vector's improved performance again.

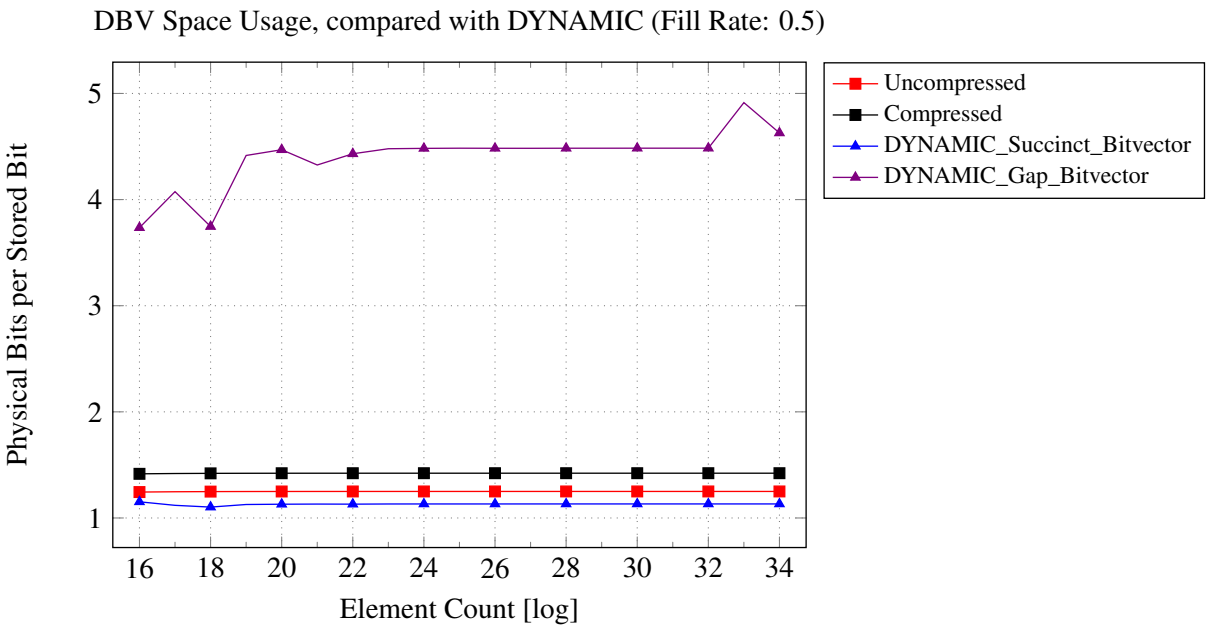


Figure 5.6: The space usage of the four tested bit vectors for a fill rate of 0.5, analogous to Figure 5.5. For such very evenly distributed data, the compressed bit vector and the gap bit vector perform worse than the other two bit vectors.

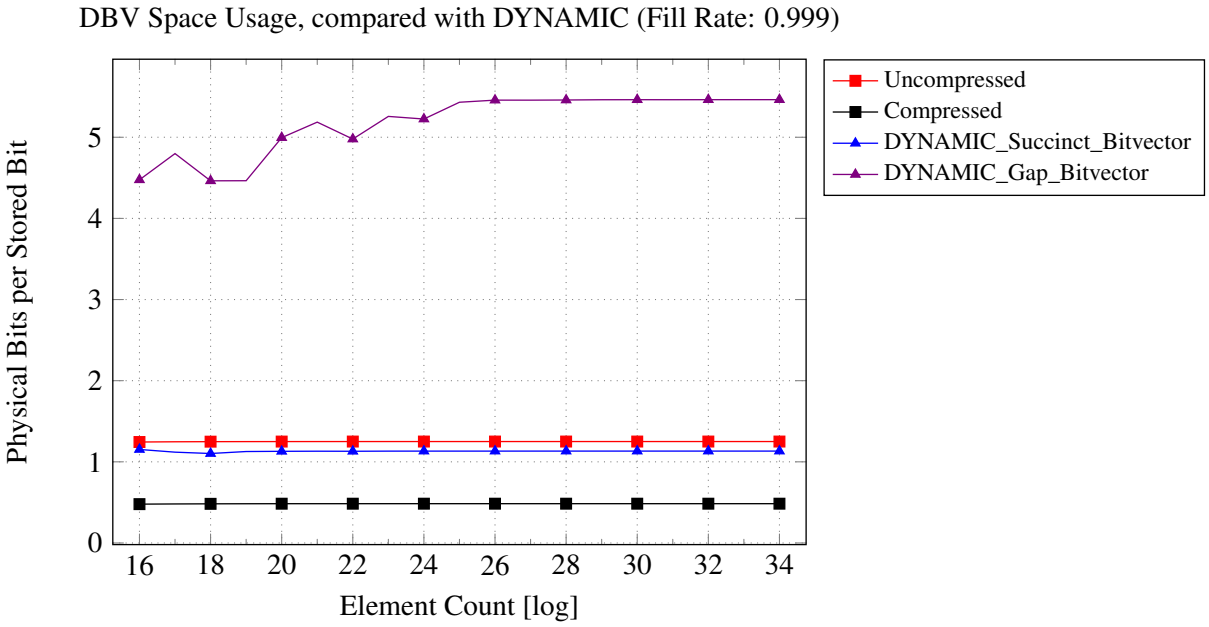


Figure 5.7: The space usage of the four tested bit vectors for a fill rate of 0.999, analogous to Figure 5.5. The gap bit vector performs worst, though if the distribution is known beforehand, the bits can simply be inverted. The compressed bit vector benefits from a very uneven distribution, allowing it to use the least space of all vectors.

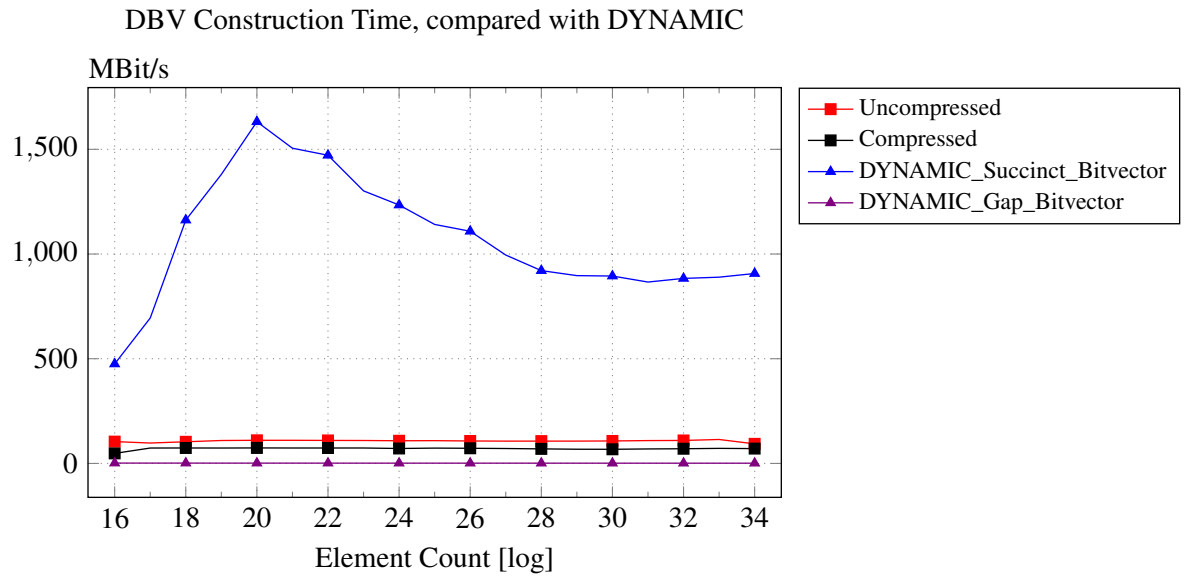


Figure 5.8: The construction time for the four bit vectors, measured by their throughput. The *y* axis shows bits per second.

We now turn our attention to the construction time of the bit vectors. The results are shown in Figure 5.8. For a size of 2^{34} bits, the uncompressed bit vector can be constructed at 93.50 MBit/s, while construction of the compressed bit vector is somewhat slower at 70.81 MBit/s. The succinct bit vector can be constructed by appending one word of 64 bits at a time, making it by far the fastest of the four data structures to construct, at 906.97 MBit/s. The gap bit vector exhibits by far the weakest performance. It can be constructed at just 1.05 MBit/s. In summary, the succinct bit vector takes significant advantage of inserting multiple bits at a time, whereas the gap bit vector is not fast to construct whatsoever. Our bit vectors perform much better than the gap bit vector, but lag behind the succinct bit vector's word-wise construction.

5.6.2 Access

For get, the results are shown in Figure 5.9. We can see that performance of the uncompressed bit vector decreases as the number of bits increase. Overall, performance bottoms out at 831.31 KBit/s. Prezza's succinct bit vector performs better, and even at 2^{34} bits, still manages to achieve 1.42 MBit/s. Both the compressed bit vector and the gap bit vector perform worse than the other two. The compressed bit vector performs better than the gap bit vector however, achieving 514.97 KBit/s, while the gap bit vector accomplishes only 253.72 KBit/s. We also note that the difference between the compressed and the uncompressed vector is very large for smaller inputs, but gradually becomes smaller as the number of bits increases. This is probably because processing a leaf takes long in the compressed bit vector compared to navigating the tree, whereas for larger trees, the cost of finding the right leaf becomes larger for both vectors.

The results for rank_1 are shown in Figure 5.10. We see similar performance characteristics to get, albeit the total performance is somewhat lower. Unlike with get, the uncompressed bit vector performs fastest

for small input sizes, but is overtaken by the succinct bit vector at 2^{18} bits. The uncompressed bit vector exhibits a throughput of 977.13 KBit/s at 2^{34} bits, whereas the compressed bit vector can only achieve 622.04 KBit/s. The succinct bit vector once again performs better at 1.36 MBit/s, while the gap bit vector once again performs worst, at only 419.50 KBit/s.
For select, the performance for evenly distributed bit vectors is very similar between select_0 and select_1. The results are visualized in Figure 5.11. For small inputs, the uncompressed bit vector performs best, but as the input size approaches 2^{34}, the data structures begin to perform closer to each other: The uncompressed bit vector and the succinct bit vector are nearly equal to each other, at 752.23 and 762.99 KBit/s, respectively. The compressed bit vector is slightly worse, at 555.69 KBit/s, and once more, the gap bit vector performs worst of all at 430.14 KBit/s.

5.6.3 Modify

We will now look at operations which modify the bit vector. First off, the results for set are shown in Figure 5.12. For smaller inputs, the uncompressed bit vector performs best, but it is again overtaken by the succinct bit vector: At a size of 2^{34} bits, the uncompressed bit vector exhibits a throughput of 844.20 KBit/s, while the succinct bit vector accomplishes 1.08 MBit/s. The compressed bit vector still achieves 448.54 KBit/s, whereas the gap bit vector significantly lags behind at only 7.20 KBit/s.
The results for insert are shown in Figure 5.13. The uncompressed bit vector performs much better here than the others, but its performance once more evens out at higher input sizes. All in all, at 2^{34} bits, the uncompressed bit vector can perform insertions at 470.94 KBit/s, followed by the succinct bit vector at 349.72 KBit/s. The gap bit vector is next, at 227.85 KBit/s. The weakest performance is from the compressed bit vector, at 26.81 KBit/s. This is likely caused by the need to decompress and recompress each block in a leaf.
In the case of deletion, the bit vectors of DYNAMIC perform notably worse. At 2^{34} bits, the performance of the uncompressed bit vector is even better than with insertions, at 550.72 KBit/s. The compressed bit vector performs similar to before, at 27.65 KBit/s, also surpassing the throughput of insert. The gap bit vector struggles with deletion, at 7.52 Kbit/s. Worse yet, the succinct bit vector can only delete 43 bits per second.
In summary, for access queries Prezza's succinct bit vector performs slightly better than our uncompressed bit vector, while also having a slightly lower space requirement. The compressed bit vector typically performs worse, whereas the gap bit vector performs worse than any of the others, with the exception of insertions, where the compressed bit vector performs worse. While the succinct bit vector can be constructed very fast thanks to its word-wise insertion, the gap bit vector struggles, while our two bit vectors are in the middle. Both the succinct bit vector and the gap bit vector have very poor performance for deletions, making our bit vectors better at erasing for large vector sizes. As mentioned, all performance benchmarks were done assuming a fill rate of 0.5. Next up, we will examine the performance of the compressed dynamic bit vector and the gap bit vector in more detail for different fill rates.

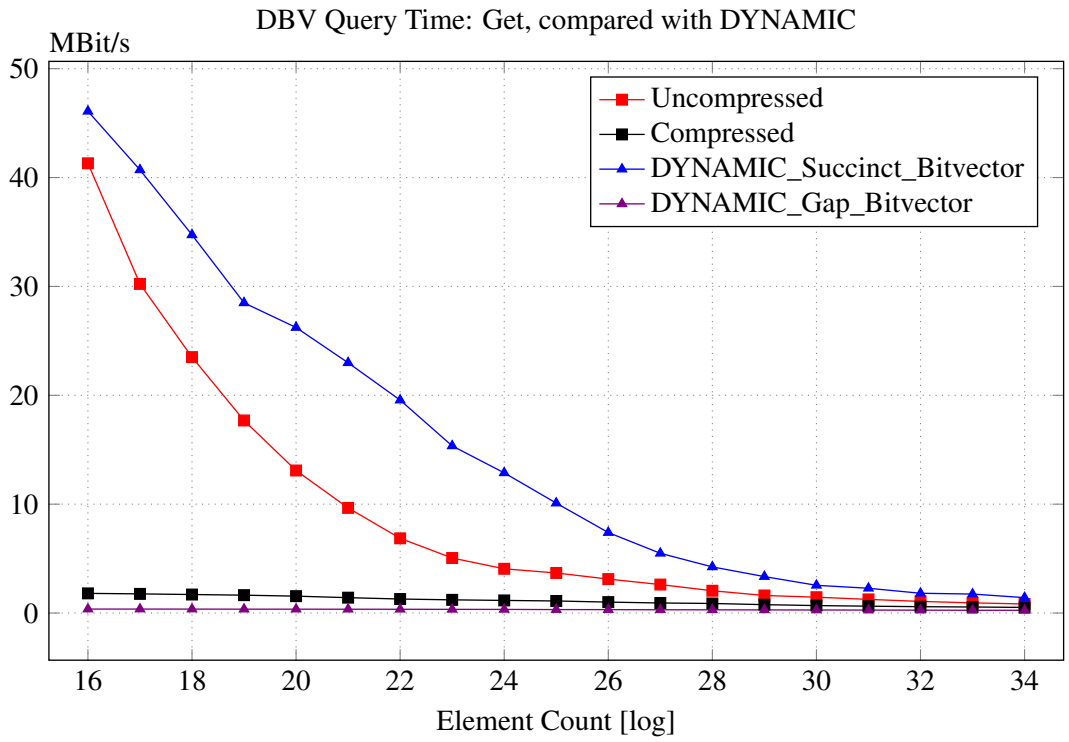


Figure 5.9: The query time for get for the four bit vectors, measured by their throughput. The *y* axis shows bits per second.

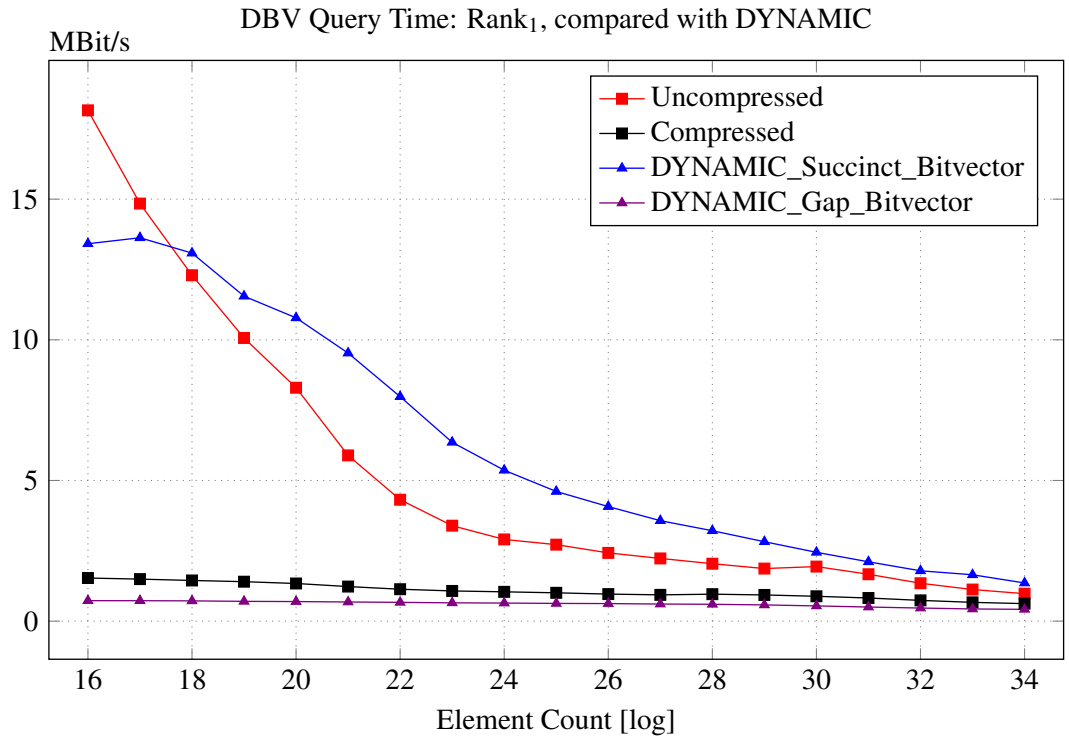


Figure 5.10: The query time for rank$_1$ for the four bit vectors, measured by their throughput. The *y* axis shows bits per second.

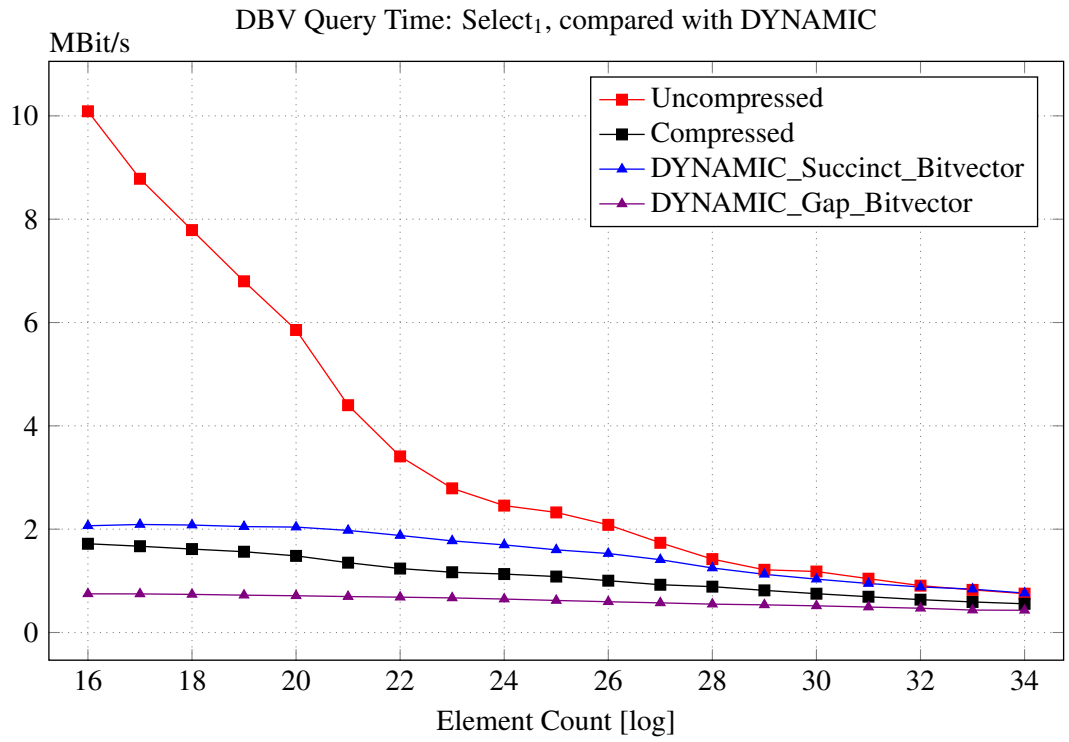


Figure 5.11: The query time for select$_1$ for the four bit vectors, measured by their throughput. The *y* axis shows bits per second.

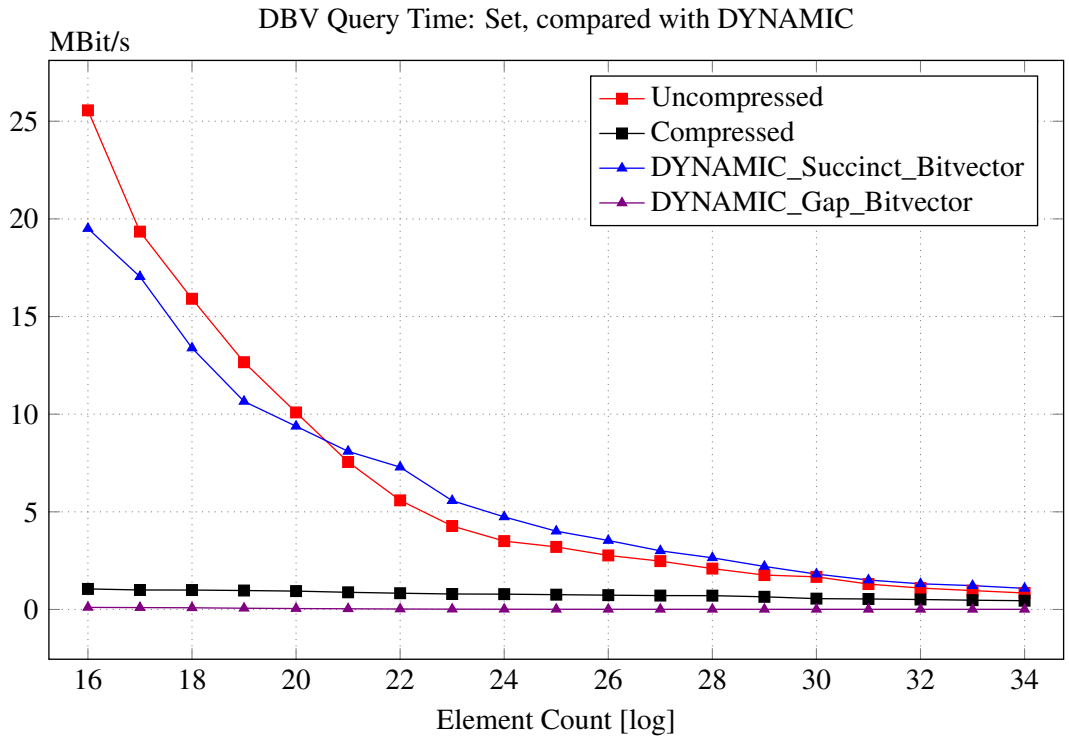


Figure 5.12: The query time for set for the four bit vectors, measured by their throughput. The *y* axis shows bits set per second.

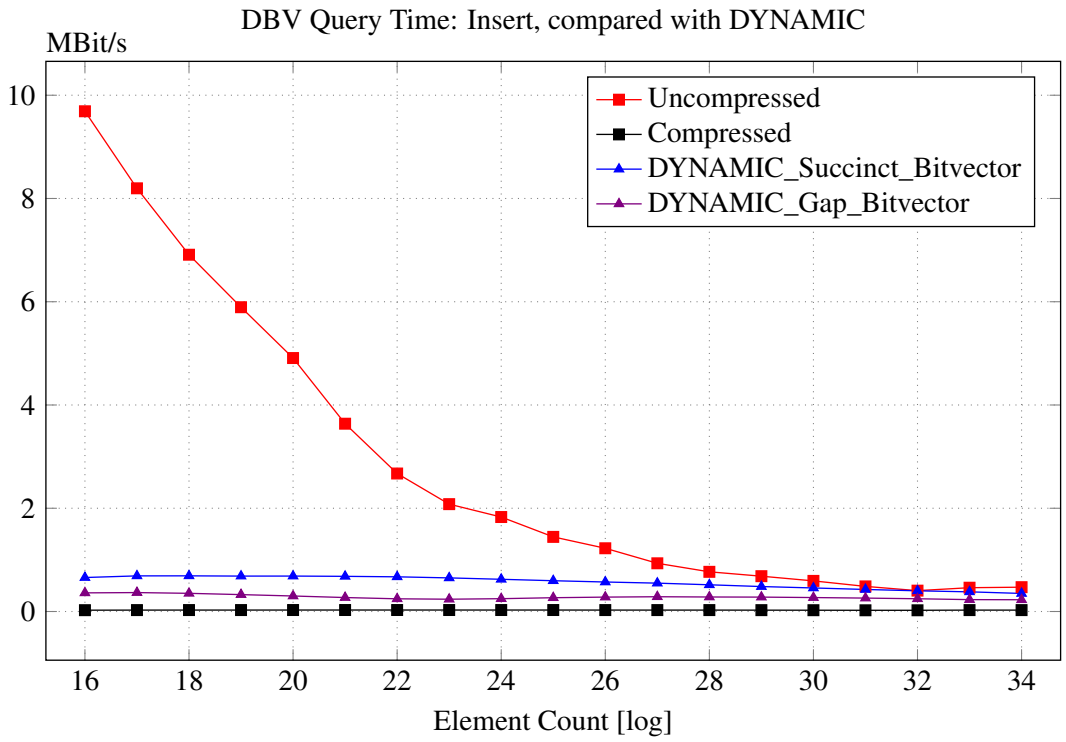


Figure 5.13: The query time for insert for the four bit vectors, measured by their throughput. The *y* axis shows bits added per second.

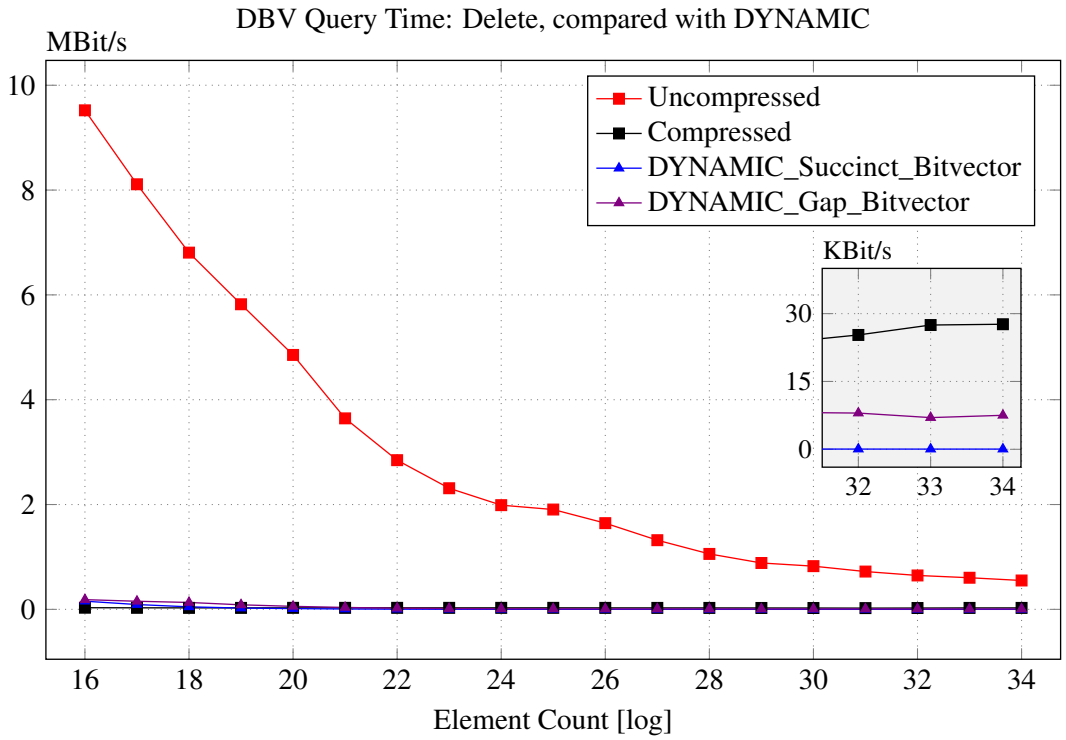


Figure 5.14: The query time for delete for the four bit vectors, measured by their throughput. The *y* axis shows bits removed per second.

5.6.4 Performance of Compressed Bit Vectors and Gap Bit Vectors

In this section, the performance of the compressed dynamic bit vector is compared to the performance of *DYNAMIC*'s gap bit vector for fill rates of 0.001, 0.999 and 0.5. Since there are methods of the gap bit vector that behave differently for different fill rates, it is chosen as an object of comparison.

Figure 5.15 illustrates the construction time of the compressed bit vector compared to the construction time of the gap bit vector. For all fill rates, the compressed vector performs considerably better than the gap bit vector, reaching a peak at a throughput of 212.12 MBit/s for fill rate 0.001 and 204.02 MBit/s for fill rate 0.999. A balanced distribution of bits leads to a throughput of about 71 MBit/s. The strong performance of the first two fill rates can be explained as follows: as aforementioned, we employ lookup tables for certain values of class c, enabling decoding and encoding in constant time. Hence, when constructing our data structure, time performance depends on the number of set bits.

On the other hand, the overall construction time of the gap bit vector does not change significantly for fill rates of 0.999 and 0.5. Here, the performance of both cases does not exceed 1.7 MBit/s. The curve of fill rate 0.001 shows a better performance with a maximum throughput of 21.35 MBit/s at a size of 2^{16} bits.

Figure 5.16 visualizes the results for rank_0. As in the previous case, the compressed bit vectors with fill rates of 0.001 and 0.999 perform remarkably better than those containing evenly distributed bits. Nonetheless, as the size of the bit vector increases, the performance degrades, exhibiting a throughput of about 900 KBit/s at a size of 2^{34} after reaching a throughput of 4.33 MBit/s (fill rate 0.001) and 4.26 MBit/s (fill rate 0.999) at a size of 2^{16}.

The gap bit vector clearly reveals a poorer performance. However, the course of the curves is similar to the compressed vector: both sparse and dense bit vectors reach a peak at a size of 2^{16} with 2.71 MBit/s for a fill rate of 0.001 and 2.42 MBit/s for a fill rate of 0.999. At size 2^{34}, fill rate 0.001 leads to 644.95 KBit/s and fill rate 0.999 shows 377.29 KBit/s. As in the case of the compressed bit vector, the performance worsens for a fill rate of 0.5.

Figure 5.17 presents the results for select_0, while Figure 5.18 illustrates the results for select_1.

In the case of compressed bit vectors, select_0 yields the best performance for densely populated vectors with a throughput of 6.26 MBit/s at a size of 2^{16}. On the other hand, select_1 performs best for sparsely populated vectors at size 2^{16} with a throughput of 7.21 MBit/s. The larger the vector size becomes, the lower the throughput is. Both operations show similar results for a fill rate of 0.5: at size 2^{16} select_0 leads to 1.67 MBit/s and select_1 yields 1.72 MBit/s, at size 2^{34} select_0 yields 553.05 KBit/s and select_1 leads to 555.69 KBit/s.

Generally, the compressed vector again shows an overall better performance than the gap bit vector for both operations select_0 and select_1. For a fill rate of 0.001, the gap bit vector reaches a peak of 2.72 MBit/s for select_0 and 3 MBit/s for select_1 at the minimum vector size. With increasing size, the throughput decreases, and the curve of 0.001 levels out at a size of 2^{18}. Only for the operation select_1 and a fill rate of 0.999 the gap bit vector performs better than the compressed vector at sizes 2^{16} (17.73 MBit/s), 2^{18} (11.64 MBit/s) and 2^{19} (9.54 MBit/s).

The results for insert (Figure 5.19) once again display the slow performance of compressed vectors for evenly distributed bit vectors with an overall throughput of about 28 KBit/s. In contrast, fill rates 0.001 and 0.999 peak at a vector size of 2^{23} and 2^{24}, respectively, yielding a throughput of 463.05 KBit/s and 387.01 KBit/s.
For a fill rate of 0.5, the performance of the gap bit vector is similar to fill rates of 0.001 and 0.999 with a throughput of 360.86 KBit/s at size 2^{16}. Also, the gap bit vector performs better than the compressed vector for a balanced distribution of bits. Regarding very dense and very sparse bit vectors, the compressed vector yields better results for sizes 2^{21} to 2^{28}.
The query time for delete is visualized in Figure 5.20. Both data structures show the poorest performance for a fill rate of 0.5. For a fill rate of 0.999, the compressed vector consistently displays a better performance than the gap bit vector.
To conclude, in all cases the compressed vector achieves a lower throughput for a fill rate of 0.5. The same holds for the gap bit vector, except for construction and insertion, where it does not reveal any significantly slower performance for evenly distributed vectors.

5.6.5 Space Usage of Compressed Dynamic Bit Vectors

Now we move on to analyse the space usage of compressed dynamic bit vectors regarding different values for block size b and compare the results against our uncompressed data structure.
Figure 5.21 presents the results for a bit vector of 2^{24} bits, or 16 megabits. Apart from a block size of $b = 64$, which we use in our implementation, we measure the space usage for $b = 32$ and $b = 16$.
While the size of the uncompressed bit vector approximately corresponds to 21 megabits of physical space, the used space of the compressed bit vector varies in dependence of the values for the $\mathscr{H}_0$ entropy. For all three block lengths, the required size reaches its minimum when the block is either very sparsely or very densely populated. Here, block length $b = 64$ shows the best results with a size of about 7.86 megabits at an entropy of $\mathscr{H}_0 = 0$ and $\mathscr{H}_0 = 1$. Block lengths $b = 32$ and $b = 16$ yield similar sizes requiring 9.17 and 11.27 megabits respectively for memory storage.
As the entropy converges towards a value of $\mathscr{H}_0 = 0.5$, space usage of the compressed bit vector increases significantly. The course of the curves of the block lengths is similar to the binary entropy visualized in Figure 2.7.
Among all three block lengths, the block consisting of 64 bits leads to the overall best results. Thus, the larger the value of b, the higher the space reduction is.

5.7 Further Work

As seen in the benchmark plots, the compressed bit vector shows deviating results, yielding an overall good performance for dense or sparse blocks, but performing less well for an even distribution of bits. A potential improvement could be allowing encoding and decoding in constant time even for a fill rate of 0.5. This can be accomplished by including all possible values for class c in our global lookup tables. However, as this will place a significant load on the memory, ways need to be found to improve time

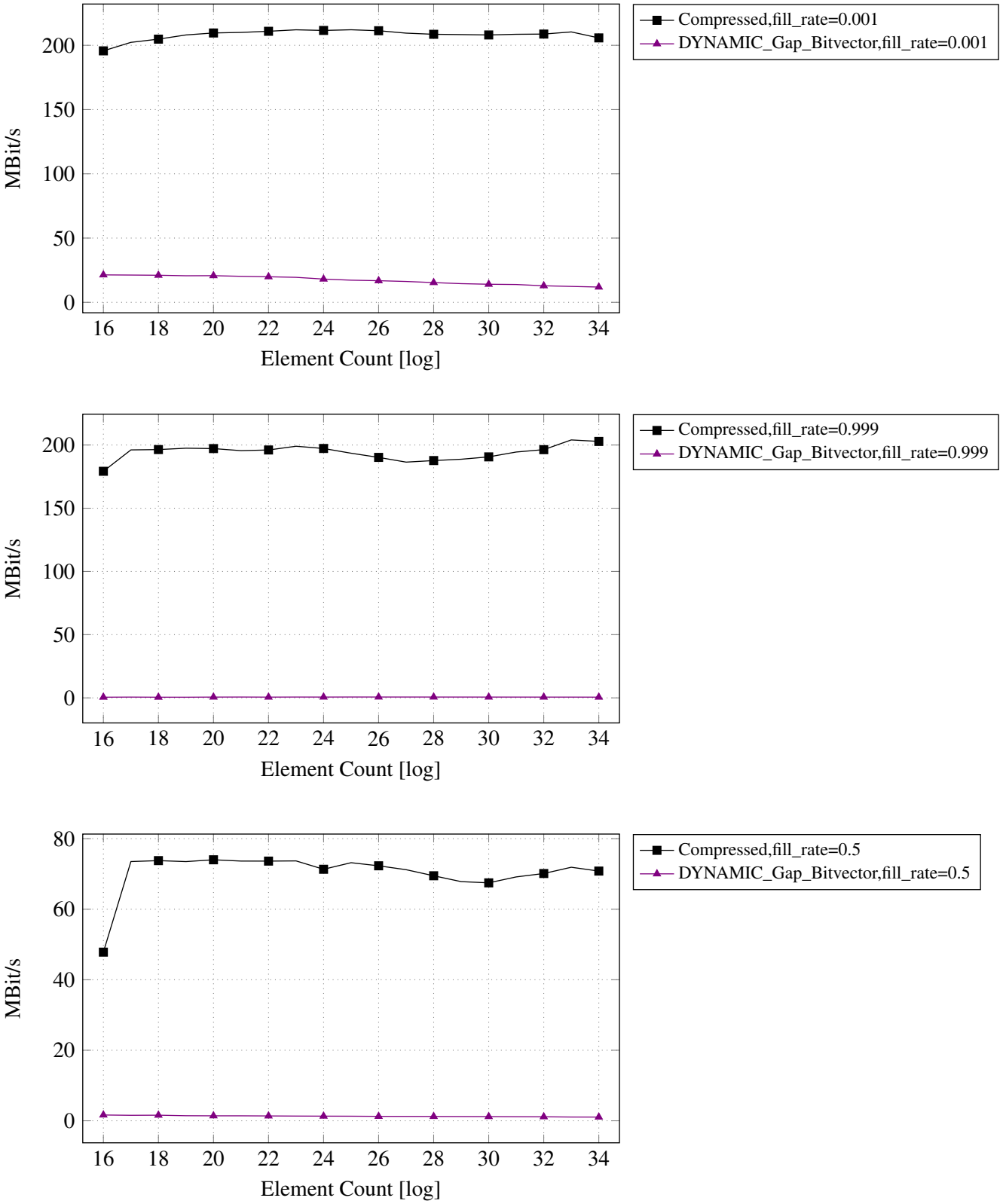


Figure 5.15: The construction time of the compressed dynamic bit vector and the gap bit vector regarding different fill rates. The *y* axis shows bits per second. For all fill rates, the compressed vector performs better than the gap bit vector.

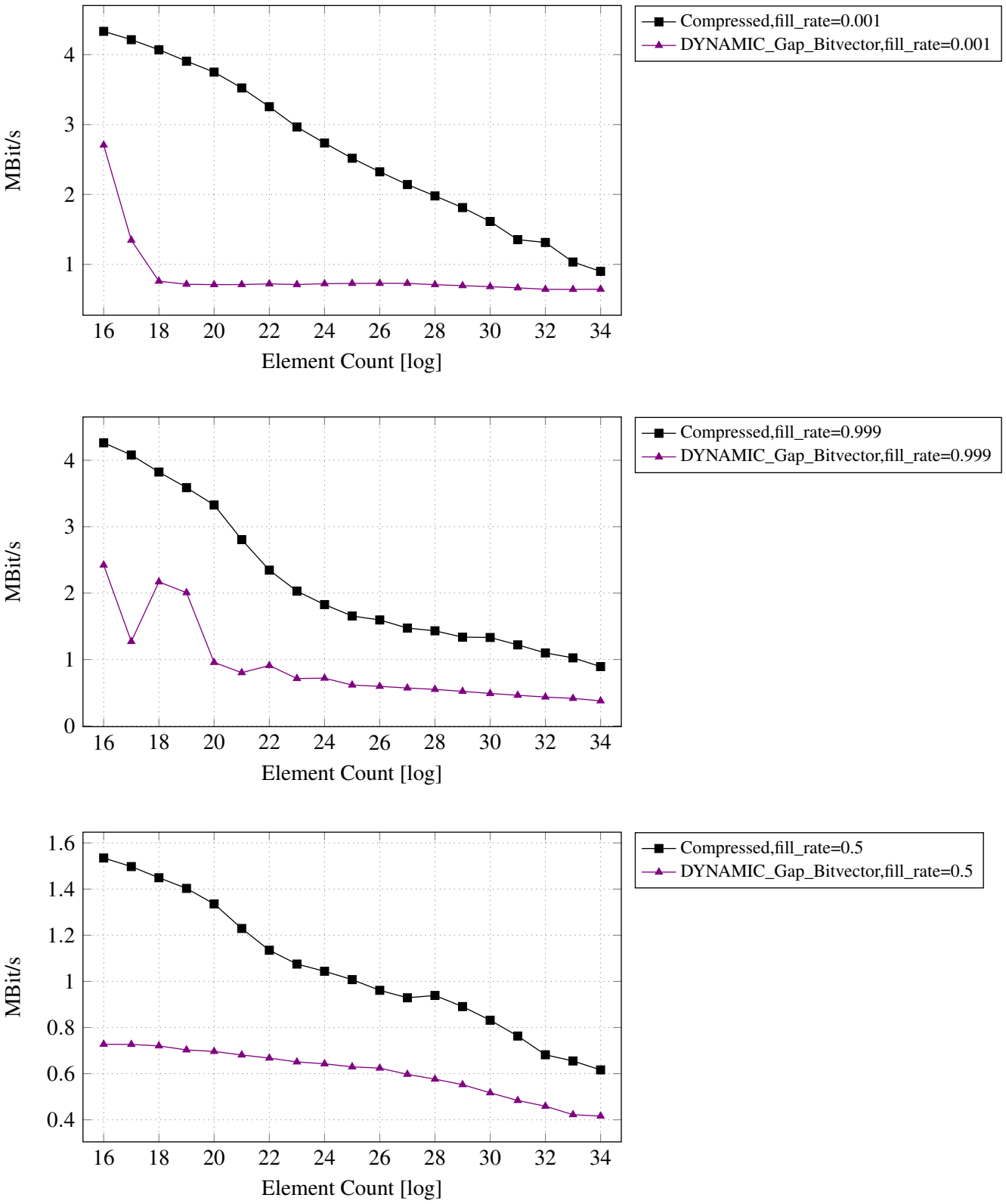


Figure 5.16: The query time for rank_0 of the compressed bit vector and the gap bit vector regarding different fill rates. The *y* axis shows bits per second. The measured throughput for fill rates 0.001 and 0.999 shows better results than the throughput for a fill rate of 0.5.

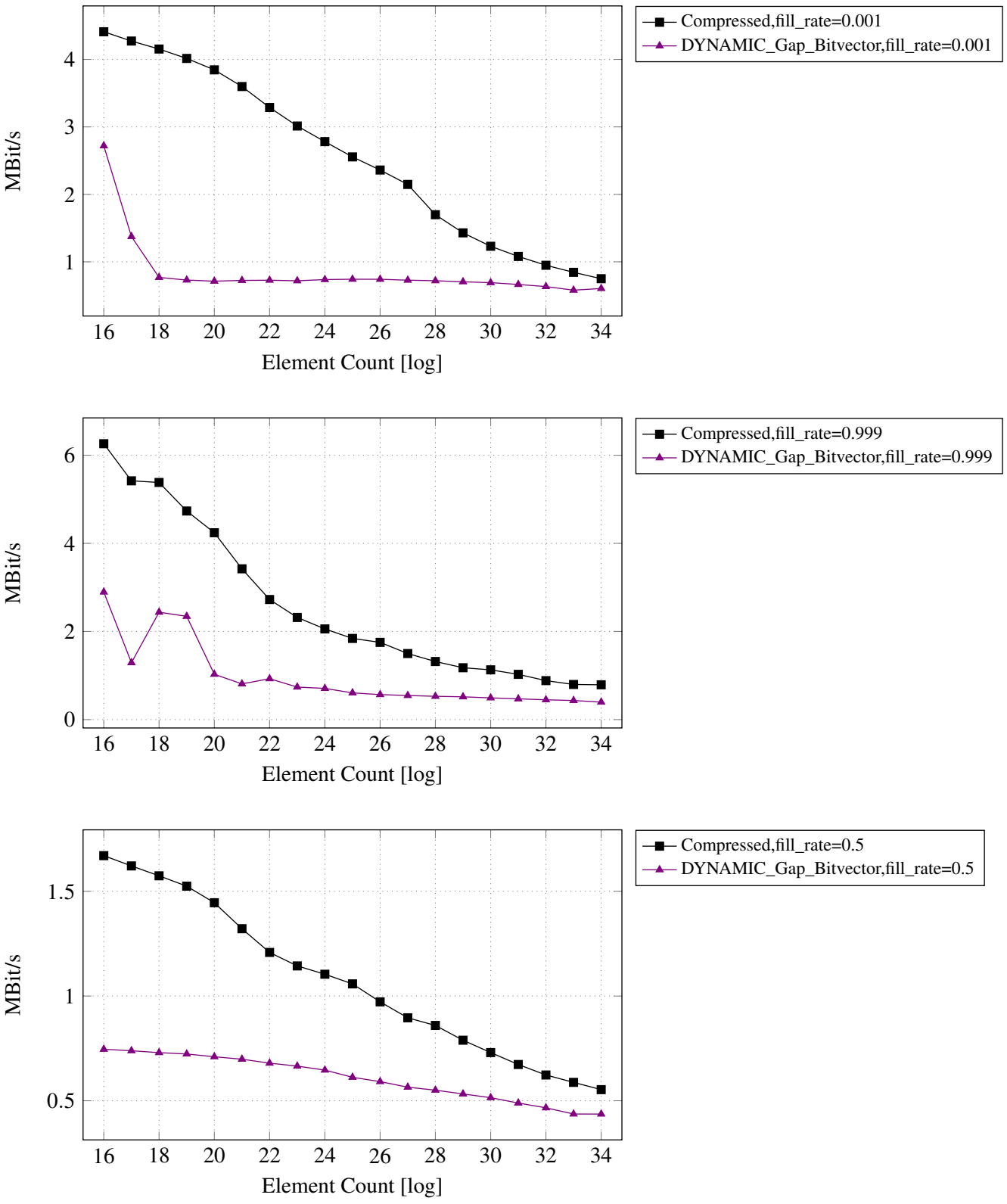


Figure 5.17: The query time for select_0 of the compressed bit vector and the gap bit vector regarding different fill rates. The *y* axis shows bits per second. For all fill rates, the compressed vector shows a better performance than the gap bit vector.

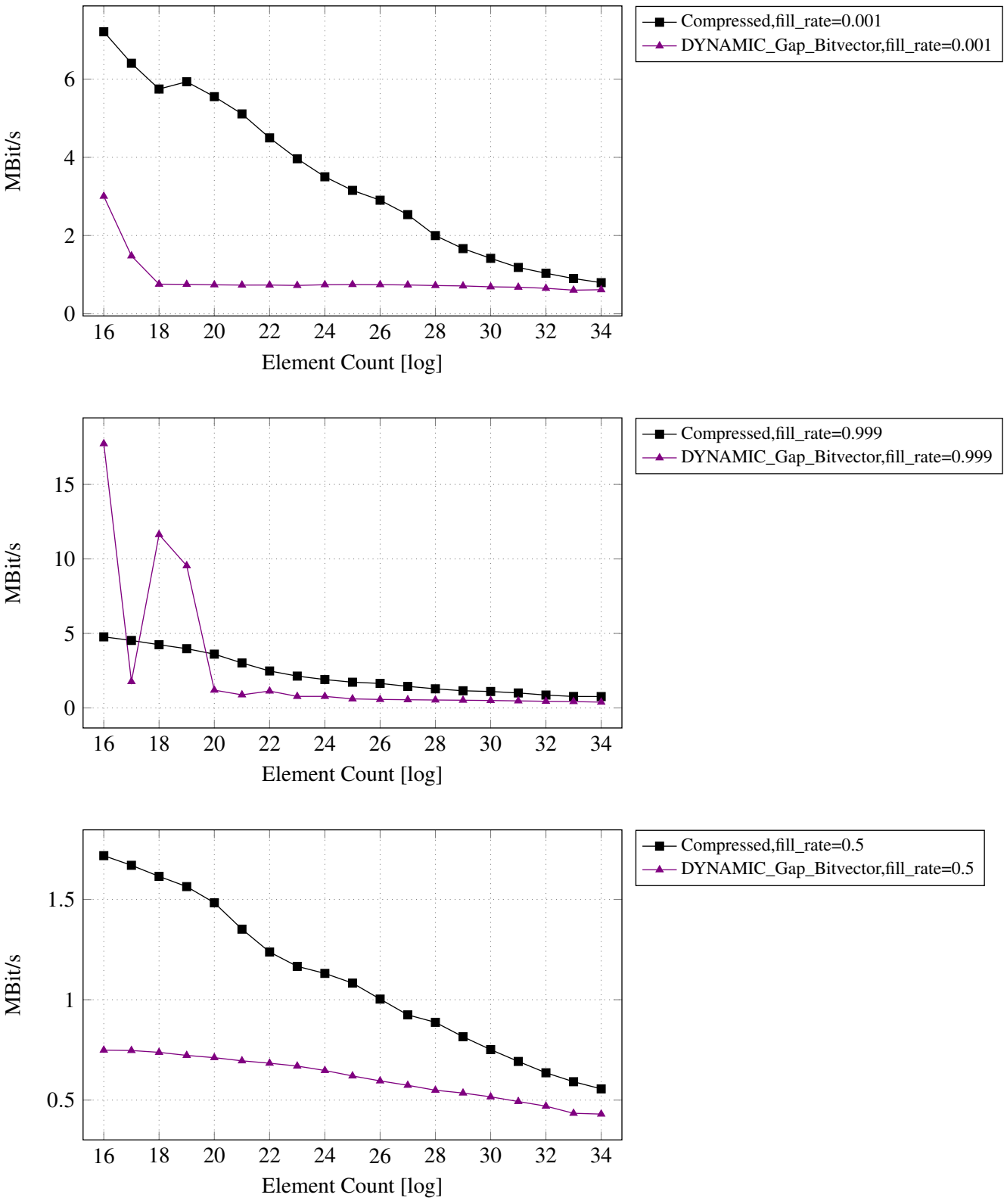


Figure 5.18: The query time for select_1 of the compressed bit vector and the gap bit vector regarding different fill rates. The *y* axis shows bits per second. For fill rates of 0.5 and 0.001 the compressed vector performs better than the gap bit vector. For a fill rate of 0.999 the gap bit vector shows a better performance at sizes 2^{16}, 2^{18} and 2^{19}.

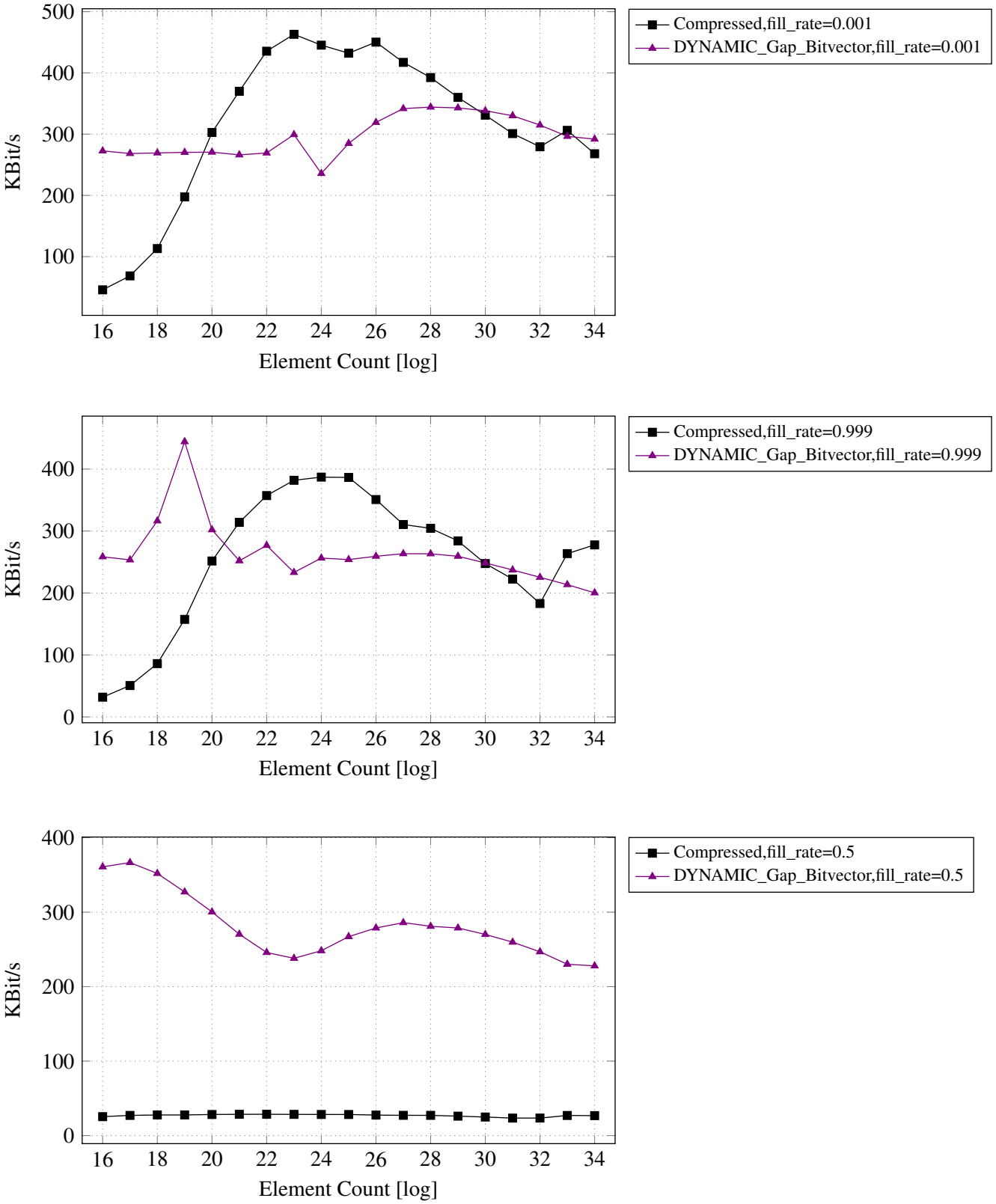


Figure 5.19: The query time for insert of the compressed bit vector and the gap bit vector regarding different fill rates. The *y* axis shows bits per second. The performance of the compressed vector clearly drops for a fill rate of 0.5, while a balanced distribution of bits does not worsen the performance of the gap bit vector.

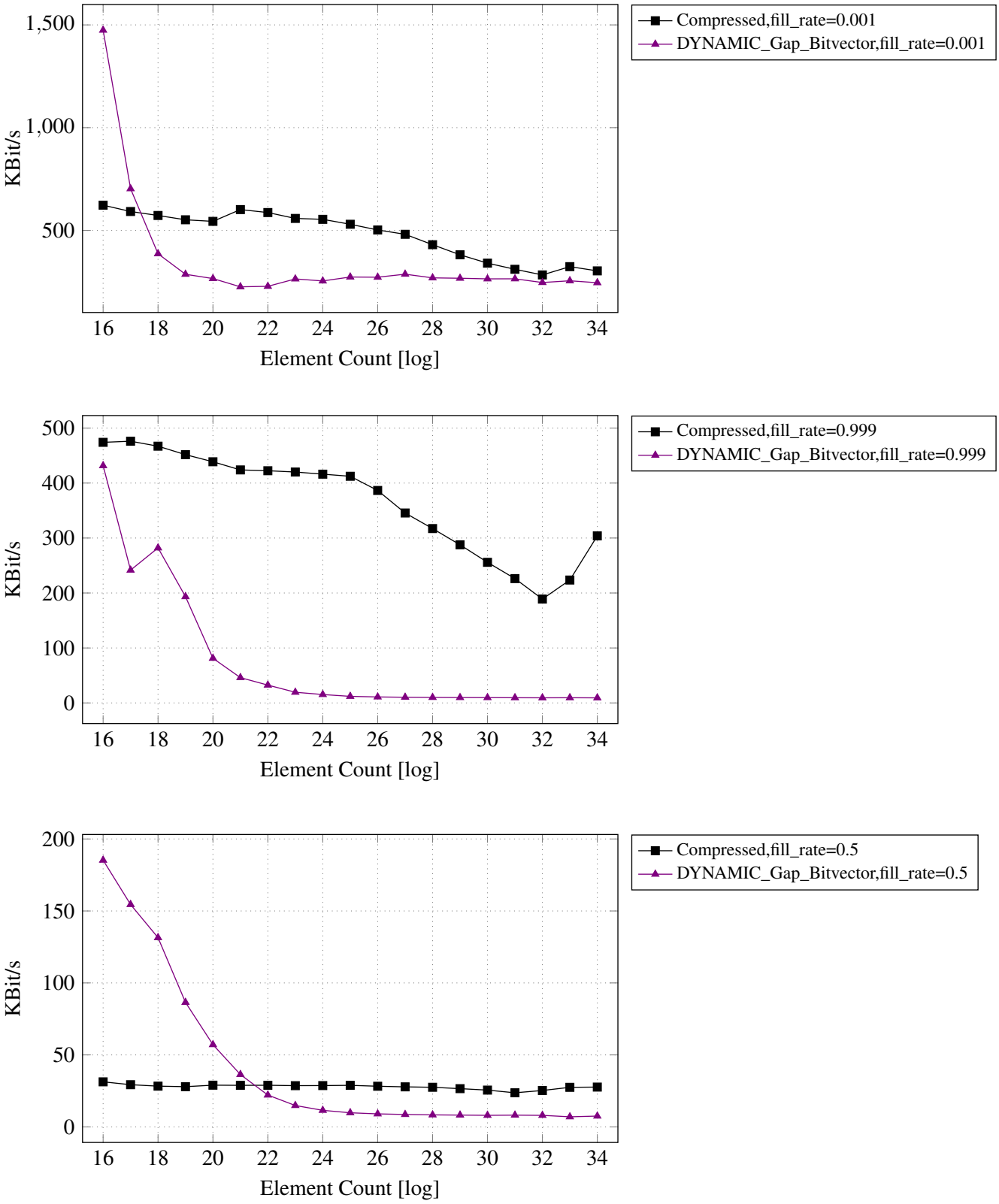


Figure 5.20: The query time for delete of the compressed bit vector and the gap bit vector regarding different fill rates. The *y* axis shows bits per second. For a fill rate of 0.999, the compressed vector performs better than the gap bit vector. Both data structures show the poorest performance for a fill rate of 0.5.

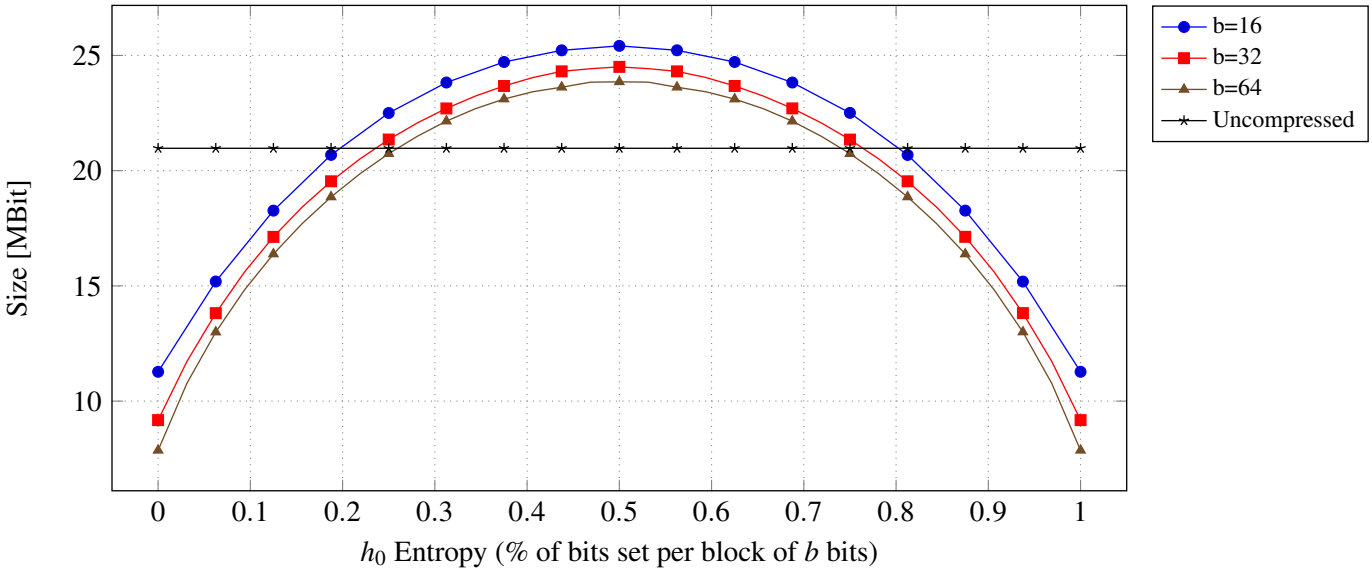


Figure 5.21: The space usage of the compressed and uncompressed dynamic bit vector for a vector size of 16 megabits. The size of the compressed bit vector is measured using different values for block size b. The x axis describes the $\mathscr{H}_0$ entropy, which is the percentage of set bits per block of length b. The y axis displays the number of megabits used for memory storage. The compressed bit vector is more space efficient than the uncompressed bit vector as long as the bit block is either very sparsely or very densely populated.

efficiency without increasing space usage.

Another important point to mention is the result visualized in the very last plot, Figure 5.21. Apparently, the larger the value for block length b, the less space will be used. Since we store various lookup tables in the background, limits were set on determining a suitable value for b. The block length of 64 could be replaced by a higher number as long as possibilities are found for representing the employed global lookup tables in a space efficient way.

The overall strong performance of the uncompressed bit vector degrades as the number of bits increases. Thus, finding a way to traverse the tree structure more efficiently could be a good starting point for a further optimization. Since we only used an AVL tree in our implementation, which requires rebalancing the tree, other data structures for representing a tree storing dynamic bit vectors can be tested.

Chapter 6

Grammar-Compressed Strings

A computer program often has to work on long sequences of symbols. In many applications, for instance analysing parts of DNA, these strings are too long to be stored explicitly, but it is often still needed to access any part of the string and to be able to answer certain queries. Therefore, the compression of strings has been a vivid research topic in the past.
In this chapter, we present a succinct compression of strings based on grammars. We first describe how to construct a special type of grammar, the so-called *straight line program*, representing a given string. Then, we show a way to store such a grammar space optimally.
In Section 6.3, we describe how some basic operations on strings can be realized using our compression. The operations that our representation of a string S supports are:

- `size`(): returns the size of S,
- `access`(i): returns $S[i]$,
- `contains`(s): returns whether s is a substring of S, and
- `lce`(i, j): returns the largest k with $S[i \dots (i+k)] = S[j \dots (j+k)]$, computing the longest common extension of i and j.

Afterwards, in Section 6.4, we elaborate on some details of our implementation and in Section 6.5, we see the results of our benchmarks. We analyse in which cases our implementation is useful in practice and what could be improved in further research.

6.1 Preliminaries

We use some other terms and operations concerning strings that need to be defined. We call a substring with more than one occurrence *repeat*. If any word $w \in \Sigma^*$ (see Section 2.2) is added to the substring s in the form of ws or sw, we call that a left or right *extension*. The *maximal extension* of a substring is the substring that can not be extended any further without decreasing the frequency of the substring. If all other extensions occur less often, this maximal extension is called a *maximal repeat*. For the text $T =$ `abracadabra` the substring `abra` is a maximal repeat, `br`, however, is not, because it can be

extended to the left (`abr`) or right (`bra`) side without decreasing the frequency.
A big part of our work relies on grammar compression. Therefore, we first discuss grammars. In the following, we use the definitions from [TTS13]. A *context free grammar* (CFG) is a 4-tuple G with $G = \{\mathscr{X}, \Sigma, X_{\text{start}}, R\}$. The set $\mathscr{X}$ is an ordered set of *variables*, which we use to define R, a finite set of binary relations called *production rules*. The set Σ is the ordered finite alphabet storing the characters of the text and $X_{\text{start}} \in \mathscr{X}$ is the *start variable* from which the whole text can be (re-)constructed. The production rules define how the variables are substituted and have the form $X \to \alpha$ with $X \in \mathscr{X}$ and $\alpha \in (\mathscr{X} \cup \Sigma)^*$. A tuple $\hat{G} = \{\hat{\mathscr{X}}, \hat{\Sigma}, \hat{R}, \hat{X}_{\text{start}}\}$ is called *subgrammar* of G if $\hat{\mathscr{X}} \subseteq \mathscr{X}, \hat{\Sigma} \subseteq \mathscr{X} \cup \Sigma$ and $\hat{R} \subseteq R$.
Grammar compression is the name for a CFG that produces exactly one given text and is therefore lossless and deterministic. To ensure determinism, for every variable v at most one production rule of the form $v \to \alpha$ exists. The language $L(G)$ generated by the grammar contains only the text and therefore $|L(G)| = 1$ holds. From now on, we assume that all grammars are deterministic and only contain rules of the form $X_i \to \sigma$ with $\sigma \in \Sigma$ or $X_i \to X_{j_1} X_{j_2} \dots X_{j_n}$, where $i > j_k$ for all $1 \leq k \leq n$, $n \in \mathbb{N}$.
We use grammars to efficiently compress texts. The quality of such a compression is determined by the size of the grammar, meaning the number of its variables and symbols on the right sides of its production rules. An important specialization of context free grammars are *straight line programs* (SLPs):

6.1.1 Definition. Straight line programs are grammars with the following properties:

1. $\mathscr{X} = \{X_1, \dots, X_n\}$ is a countable set of variables,
2. for every variable X_i, exactly one rule of the form $X_i \to \alpha$ with $\alpha \in (\mathscr{X} \cup \Sigma)^*$ exists,
3. $|L(G)| = 1$, and
4. every rule has the form $X_i \to \sigma$ for any $\sigma \in \Sigma$ **or** $X_i \to X_j X_k$ with $1 \leq j, k \leq i \leq |\mathscr{X}|$.

Straight line programs can be represented as graphs that can be constructed as follows: we initialize $\mathscr{X} \cup \Sigma$ as the set of vertices. For every rule $X_i \to X_j X_k$, two edges (X_i, X_j) (called a *left edge*) and (X_i, X_k) (*right edge*) are added to the graph for the left and right side of the rule. For every rule following the pattern $X_i \to \sigma$ we add the edge (X_i, σ). The vertices from Σ are connected to an additional vertex s. If all edges of this graph are inverted (meaning we replace an edge (i, j) with the edge (j, i)) and then remove all left edges or all right edges respectively, the resulting graph is a tree with s as the root.
Figure 6.1 shows a SLP for the string `abab` and its representation as a graph.
A naive approach to store a SLP, called *dictionary*, uses an array D of length $2n$. For a rule $X_i \to X_j X_k$ in the SLP, $D[2i-1] := j$ and $D[2i] := k$ are set. For $X_i \to \sigma$ for $\sigma \in \Sigma$, we set $D[2i-1] = D[2i] = 0$. This approach uses $2n\lceil \log n \rceil$ bits of space. For an index k, the rule $X_k \to X_i X_j$ can then be found in constant time. Of course, if the string uses an alphabet that does not have the form $\{1, \dots, m\}$, we need to store the original symbols and their corresponding rule numbers explicitly to restore the original string.
We further use rank and select queries on strings over arbitrary alphabets, as defined in Section 2.8.

SLP:

$X_1 \to a$,

$X_2 \to b$,

$X_3 \to X_1X_2$,

$X_4 \to X_2X_3$,

$X_5 \to X_1X_4$

stored in array (dictionary):

1	2	3	4	5	6	7	8	9	10
0	0	0	0	1	2	2	3	1	4

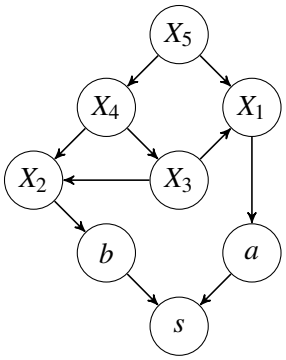


Figure 6.1: A straight line program for the string `abab` and its representations as graph and dictionary

6.2 Construction

In this section, we will describe how to construct a SLP representing a given string and how to store that SLP efficiently.

6.2.1 Constructing the SLP

As first step, the given string has to be transformed into a CFG that is deterministic and lossless. We are using RePair [LM00] to achieve this in linear time. RePair looks for the most common pair of symbols in the given text $S \in \Sigma^*$ and creates a grammar $G = \{\mathscr{X}, \Sigma, X_{\text{start}}, R\}$. The set of rules R is filled with the following steps:

1: Replace every symbol $\sigma \in \Sigma$ with a variable X_σ and add a new rule $X_\sigma \to \sigma$ to R.

2: Find the most frequent pair $p = X_{p_1}X_{p_2}$ in S.

3: Replace every occurrence of p with a new variable X and add a new rule $X \to X_{p_1}X_{p_2}$ to R.

4: If the maximum frequency of all pairs in the text generated by step 3 is 1, terminate and add the rule $X_{\text{start}} \to$ (current text) to R. Else return to step 2.

RePair works in $\mathscr{O}(n)$ expected time and $5n + 4|\Sigma|^2 + 4|R| + \lceil\sqrt{n+1}\rceil - 1$ words of space, where n is the length of the source text, $|\Sigma|$ the cardinality of the source alphabet Σ and $|R|$ the cardinality of the final dictionary, the set of rules R.

Since RePair selects only pairs and only creates one rule at a time, all rules in R follow the conditions for SLPs set in property 4 of Definition 6.1.1. Step 1 in RePair corresponds directly to property 2. Since only the source text should be possible to recreate with the generated grammar, property 3 is also fulfilled. The first rule however, the start variable, violates the SLP property 4. We stop compressing the text after no more pairs can be found with a frequency of at least two and the given text is not likely to be fully compressed. Therefore, there is a rest left over that contains more than two variables. This means we have to compress the text further and we do that by modifying our start variable.

A start variable constructed by RePair can have the form $X_{start} = X_1 \dots X_n$ for any given n. Since our

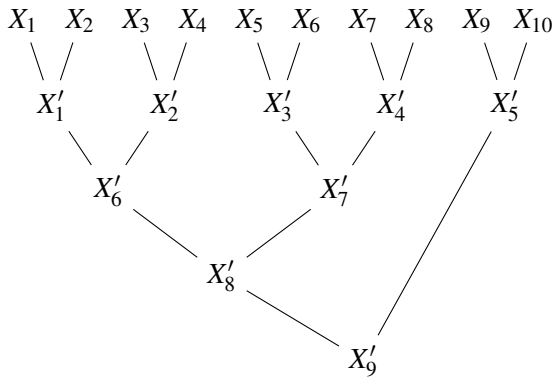


Figure 6.2: The tree structure of the series of rules used to transform a start variable of length 10 into a SLP

definition of a SLP needs a start variable from $\mathscr{X}$, we have to create new rules on top of the grammar to represent the start variable. To reduce the height of the derivation tree, we take pairs of variables to make into a new rule in the form of $X_i^{(1)} \to X_i X_{i+1}$, starting at X_1 until X_n is reached. Then, the next level of rules is constructed: $X_i^{(2)} \to X_i^{(1)} X_{i+1}^{(1)}$. On levels with an odd number of variables, the last variable will be carried over to the next level that again has an odd amount. This makes the tree of rules slightly unbalanced in practice, but creating new rules or empty variables to change the length of input start variable to a power of two would create a big overhead in the number of rules and does not improve the derivation time of the start variable. The whole process terminates when only one variable is left, the new start variable. These new rules follow the constraints of SLPs since we always take pairs and can therefore be stored efficiently, as we will see later. An example tree is shown in Figure 6.2. The variable X'_5 has no partner on its level and is carried over until only X'_8 is left. The new start variable is then X'_9 with the first rule $X'_9 \to X'_8 X'_5$.

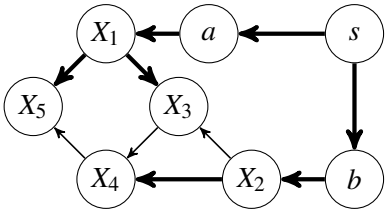
6.2.2 Succinct Representation of the SLP

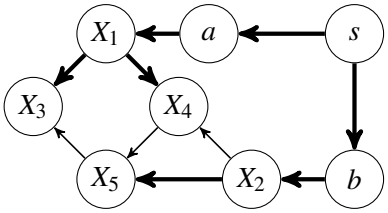
In this subsection, we describe how to store SLPs space efficiently. The data structure is developed in [TTS13]. The authors also prove a lower bound for storing a SLP: an SLP with n variables requires at least $2n + \log n! + o(n)$ bits to be represented. The shown compression asymptotically meets this lower bound.

The main idea of the succinct representation is to store the dictionary (the representation of a SLP in an array) using less space.

The first step is to rename the variables $X_1, \ldots, X_n$. For that, we consider the subgraph that consists of all inverted non-right edges of our graph representation from Section 6.1. We rename the variables according to their occurrence in a breadth-first search of this subgraph starting in s. This process is illustrated in Figure 6.3 for the example SLP of Section 6.1. During a breadth-first search from s, the variables are reached in the order X_1, X_2, X_5, X_3, X_4. Therefore we rename X_5 to X_3, X_3 to X_4 and X_4 to X_5. The grammar and dictionary have to be changed accordingly.

After renaming, the dictionary fulfils the property $D[1] \leq D[3] \leq \cdots \leq D[2n-1]$. That means the values in the array form a monotonically increasing sequence, if we omit the even indices. With a gap encoding





$X_1 \to a, X_2 \to b,$
$X_3 \to X_1X_2, \quad X_4 \to X_2X_3,$
$X_5 \to X_1X_4$

$X_1 \to a, X_2 \to b,$
$X_4 \to X_1X_2, \quad X_5 \to X_2X_4,$
$X_3 \to X_1X_5$

1	2	3	4	5	6	7	8	9	10
0	0	0	0	1	2	2	3	1	4

1	2	3	4	5	6	7	8	9	10
0	0	0	0	1	5	1	2	2	4

Figure 6.3: On the left, the original SLP, its dictionary and the graph from Figure 6.1 are shown. The graph is depicted after inverting its edges. The highlighted edges form a tree with root s that contains all non-right edges. On the right, the SLP, dictionary and modified graph are shown, after the variables are renamed according to their occurrence in a BFS starting in s.

original dictionary:

D:

1	2	3	4	5	6	7	8	9	10
0	0	0	0	1	5	1	2	2	4

omitted odd indices:

D':

1	2	3	4	5
0	0	5	2	4

$s_1 = \{1,2,4\}$ (monotonically increasing),
$s_2 = \{3,5\}$ (monotonically decreasing)

Figure 6.4: The original array is reduced to its even indices. The indices of the resulting array are divided into two monotonic subsequences.

(see Section 2.11), we can store that half of the dictionary using $3n-1$ bits of space, as the dictionary contains $2n$ values and the largest value is smaller than n.

Hereafter, we omit the uneven indices from D, defining $D'[i] \coloneqq D[2i]$, so that D' has length n. This new array is not necessarily a monotonic sequence, but we follow a similar idea. We divide the indices $1,\ldots,n$ into a partition $\mathscr{S} = \{s_1,\ldots,s_k\}$ so that for every s_i, its associated values in D' are a monotonic subsequence of D'.

Figure 6.4 shows an example partition. We omit the uneven indices and get a new array D' of half length. The sequence s_1 is monotonically increasing, because $D'[1] \leq D'[2] \leq D'[4]$ holds. Along the same lines, s_2 is monotonically decreasing, because $5 = D'[3] \geq D'[5] = 4$ holds. This partition is by no means unique, we could as well choose $s'_1 = \{1,2,4,5\}$ and $s'_2 = \{3\}$ as a partition. For our string

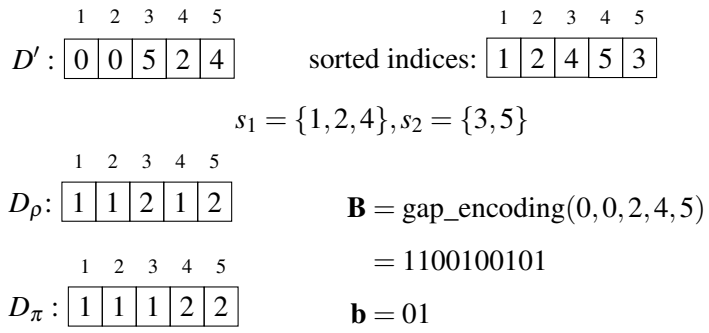


Figure 6.5: Our succinct representation for the example from Figure 6.4.

representation, we rely on the ability to find a partition that consists of as few sequences as possible. How to compute such a partition was one challenge for our implementation. If given such a partition we can represent the remaining array D' space efficiently.

Our data structure is a tuple $(D_\rho, D_\pi, \mathbf{B}, \mathbf{b})$. The array D_ρ encodes which indices of D' are included in which sequence in $\mathscr{S}$. That means, $D_\rho[i] = k$ holds if and only if $i \in s_k$. The array D_π denotes a permutation of D_ρ, ordered by the values of D'. That means if $D'[l_1] \leq D'[l_2] \leq \cdots \leq D'[l_n]$ holds, then $D_\pi[i] = D_\rho[l_i]$. Both D_ρ and D_π are arrays over the alphabet $\{1,\ldots,n\}$. We use wavelet trees (see Section 2.8) to store them, because we later need efficient rank and select queries to reconstruct the dictionary from our representation. The bit vector $\mathbf{B}$ gap encodes the array D', sorted by values in ascending order. Lastly, $\mathbf{b}$ is a bit vector that indicates for every sequence in $\mathscr{S}$ whether it is monotonically increasing or monotonically decreasing. In other words, $\mathbf{b}[k] = 0$ iff s_k is monotonically increasing. We store both bit vectors as described in Section 4.2, to ensure efficient rank and select queries in this case as well.

In Figure 6.5, the encoding for our previous example is illustrated. The array D' only stores the even indices of its original version. The monotonic sequences are the ones from Figure 6.4. The property $D_\rho[1] = D_\rho[2] = D_\rho[4] = 1$ holds, because those indices belong to s_1 and analogously $D_\rho[3] = D_\rho[5] = 2$, because 3 and 5 belong to s_2. The array D_π matches $D_\rho[1], D_\rho[2], D_\rho[4], D_\rho[5], D_\rho[3]$. The bit vector $\mathbf{B}$ encodes D', in ascending order, using gap encoding and $\mathbf{b}$ indicates that the indices in s_1 correspond to a monotonically increasing subsequence of D' and the indices in s_2 correspond to a monotonically decreasing subsequence of D'.

We describe how to reconstruct the original dictionary from this representation in Section 6.3.

Intuitively, this construction needs less space than the naive approach of just storing the dictionary because we only store the indices of the sequences explicitly. If we find a partition with few sequences, those indices are much smaller than the values from D'. The explicit values in D' are only stored implicitly in the gap encoding and therefore take much less space than in the naive approach. Nevertheless, it is clear that our space requirement is dependent of our ability to find appropriate monotone subsequences. The exact space needed is analysed in [TTS13] and is $2n \log |\mathscr{S}|(1 + o(1))$. In [BF98] it is shown that we can find a partition with $\mathscr{S} \leq 2\sqrt{n}$ for every SLP. Asymptotically, this space requirement meets the proven lower bound of $2n + \log n! + o(n)$ bits, as both bounds are in $\Theta(n \log n)$.

6.3 Operations

In this section, we discuss how to realize the intended operations `access`, `size`, `contains` and `lce` for our grammar compressed string. The main component of our data structure is the succinct representation of a dictionary. Since operations require access to rules of the SLP, we have to discuss first how any rule of the SLP can be reconstructed in short time.
Then we will discuss how we realised the operations in theory, and which additional components our data structure needs to ensure efficient operations.

6.3.1 Accessing Rules of the SLP from the Succinct Storage

First, we show how our grammar representation simulates the original dictionary D. To compute the value $D[p]$, where p is even, we first determine to which set s_k of our partition $\mathscr{S}$ the index p belongs with $k = D_\rho[p]$. Then we determine whether s_k corresponds to a monotonically increasing sequence ($\mathbf{b}[k] = 0$) or to a monotonically decreasing sequence ($\mathbf{b}[k] = 1$).
In the first case, we count how often the value k occurs in D_ρ up to index p and therefore determine how many smaller indices also belong to s_k. Afterwards, we use D_π to see how many values in D are smaller than $D[p]$ and with that knowledge, we can read $D[p]$ from $\mathbf{B}$. Altogether, we compute:

$$k = D_\rho[p] \text{ and } D[p] = \text{rank}_0(\mathbf{B}, \text{select}_1(\mathbf{B}, \text{select}_\text{k}(D_\pi, \text{rank}_\text{k}(D_\rho, p))))$$

In the case that s_K corresponds to a monotonically decreasing subsequence, we also want to count how many smaller indices belong to s_k. In this case however, we first compute how many indices belong to s_k altogether and then subtract the number of indices larger than p. All in all, in this case we compute:

$$k = D_\rho[p] \text{ and } D[p] = \text{rank}_0(\mathbf{B}, \text{select}_1(\mathbf{B}, \text{select}_\text{k}(D_\pi, \text{rank}_\text{k}(D_\rho, n) + 1 - \text{rank}_\text{k}(D_\rho, p)))).$$

Since the components are represented as wavelet trees and bit vectors, we can ensure that the needed rank and select queries can be executed runtime efficiently. In [TTS13] it is stated that we can achieve $\mathscr{O}(\log\log n)$ time to access an even value in D'.
To get the values of D for the odd indices, we merely have to read the value from the gap encoding as described in Section 2.11 and therefore compute:

$$D[p] = \text{rank}_0(ge, \text{select}_1(ge, \lfloor p/2 \rfloor)),$$

where ge is the gap encoding of the odd indices. This needs a rank and a select query on a static bit vector and is therefore possible in constant time.
Altogether, we need $\mathscr{O}(1 + \log\log n)$ time to reconstruct any rule from the SLP.

6.3.2 The Access Operation

A naive approach to get the symbol at a specific position in the original text is to retrieve the text from the SLP (fully or up to the desired index). This can be achieved step by step through replacing a variable X_i with α iff $X_i \to \alpha$ is a rule of the SLP, starting with the start variable. This process always terminates

and returns the original text. Since the length of the string can be exponential in the size of the SLP, the derivation can take exponential time in the size of the SLP and is therefore not practical.
To make `access` faster, especially for high indices, we concede some extra space usage. For every variable in the SLP, we store the length of the string it produces. Then we can find the symbol we need without deriving the rest of the text. The extra space needed is $\mathcal{O}(|D|\log(|T|))$ bits, one number $\leq |T|$ for every rule of the SLP. The time needed to find the wanted rule is $\mathcal{O}(\log|D|)$, since we only need to compare the needed index with the saved string lengths of each rules' left and right side. Since the dictionary and our rules are built similar to a binary tree structure, the number of comparisons is $\mathcal{O}(\log|D|)$. After finding the right rule we only have derive a single rule for which we only need $\mathcal{O}(1)$ extra time. How the array of string lengths is built and how `access` is managed in detail is shown in Section 6.4.6 and Section 6.4.7 respectively.

6.3.3 The Other Operations

The `size` operation can be easily realised in constant time by storing the size of the input string explicitly. In theory, this takes $\mathcal{O}(\log|T|)$ additional space for a text T.
The `lce` and `contains` operations are realised naively using the `access` operation.
For `lce(i,j)`, we reconstruct the text starting from the indices i and j until they differ. This takes time $\mathcal{O}(|D|\cdot l)$ where $l \leq |T|$ is the longest common extension.
The `contains(s)` operation can be solved in constant time if $|s| = 1$, since every symbol stored in the SLP representation is contained in the original text. However, if s is longer, we follow the naive approach and reconstruct the original text until s is found or the whole string is reconstructed.

6.4 Implementation

In this section we describe some challenges and properties of our implementation.

6.4.1 Building the SLP

To build a SLP, we firstly need to generate a grammar for the given text. To manage this, we used Navarro's implementation of RePair[1]. It works in linear time and could be incorporated into our project fairly easily. However, we had to make some adjustments to the RePair implementation. Navarro's implementation works on files with a command line input. While leaving the core functionality the same, we changed it so RePair uses input text directly and works on that. Since the output is also saved into a file, we removed the whole file creation and extracted the necessary results directly. We get a start variable, a dictionary for all symbols and a list of all pairs from RePair. Using the pairs and the old start variable we can then build a new start variable as described in Section 6.2.

[1] See `https://users.dcc.uchile.cl/~gnavarro/software/repair.tgz`

6.4.2 Finding Monotone Sequences

We saw that finding a fitting partition into monotone subsequences is of utmost importance to the performance of our succinct construction. In [BF98], it is shown that a partition into no more than $2\sqrt{n}$ sets exists for any sequence of length n. Sadly, the problem of finding such a partition is NP-hard.
Since our dictionaries can get really big, time efficiency is a crucial factor in the computation of sequences for us. Nevertheless, if too many sequences are computed, the succinct storing will have no advantage over storing the SLP explicitly.
Therefore, we chose a relatively simple greedy algorithm in our implementation. In Section 6.5, we will see that it computes a satisfyingly small partition.
The pseudocode can be found in Algorithms 6.1 and 6.2. The main idea is to always start at the first index and then greedily compute a decreasing and an increasing sequence simultaneously. This is done by always adding the next fitting index to a sequence. The longer sequence is then put into our partition and its indices can be removed. Algorithm 6.1 is used to repeatedly call Algorithm 6.2 in line 2, add the computed partition to the set in line 3 and store which indices from D are not partitioned in the set R.
In detail, Algorithm 6.2 computes two monotone sequences, S_{inc}, which is increasing, and S_{dec}, which is decreasing, that both begin with the first unpartitioned index that is given as input i. The variables *last_value_inc* and *last_value_dec* are used to store the current last values in the sequences. We need those to decide whether another value continues one of the sequences. The variables are initialized in lines $1-3$, so that both sequences contain i and the last value is the corresponding value in D. The loop from line 4 is used to iterate through all indices from i to the end of the array that were not partitioned when the algorithm was called. If the value at the current index continues the current increasing sequence (line 5), S_{inc} and *last_value_inc* are updated (lines 6-7). Similarly, the index is potentially added to S_{dec} (lines 9-11). After iterating over all indices, the longer sequence is returned. If both have equal length the increasing sequence is returned instead(lines 14-17).

Input: array D of length n
Output: partition of indices $1,\ldots,n$ in which the corresponding values from D form a monotone subsequence for every set

1: Set $R = \{1,\ldots,n\}$
2: **while** D has indices that are not in the partition ($|R| > 0$) **do**
3: Compute greedy monotone subsequence S starting in first unpartitioned index (see Algorithm 6.2)
4: Add S to partition
5: Remove indices in S from R
6: **end while**

Algorithm 6.1: Greedy algorithm for calculating monotone subsequences

In our current implementation we can use a parameter to restrict the maximal length of a subsequence. In Section 6.5 we will see which value works best in practice. The runtime of the algorithm depends on the size of D and of the number of calculated sequences, which makes its analysis a bit complicated. In the worst case, our array D contains the values 1 to n (where n is even) and has the following form:

Input: array D of length n, index i, set R containing active indexes from D
Output: subset of indices $\{i,\ldots,n\}$ where the corresponding values in D form a monotone subsequence

1: Initialize S_{inc} and S_{dec} with $\{i\}$
2: $last\_value\_inc \leftarrow D[i]$
3: $last\_value\_dec \leftarrow D[i]$
4: **for each** $j \in \{i+1,\ldots,n\}$ with $j \in R$ **do**
5: **if** $(D[j] \geq last\_value\_inc)$ **then**
6: Add j to S_{inc}
7: $last\_value\_inc \leftarrow D[j]$
8: **end if**
9: **if** $(D[j] \leq last\_value\_dec)$ **then**
10: Add j to S_{dec}
11: $last\_value\_dec \leftarrow D[j]$
12: **end if**
13: **end for**
14: **if** $|S_{inc}| \geq |S_{dec}|$ **then**
15: return S_{inc}
16: **end if**
17: return S_{dec}

Algorithm 6.2: Greedy algorithm for calculating a monotone subsequence starting at a given index

$$1, n, 2, n-1, 3, n-2, \ldots, \tfrac{n}{2}, \tfrac{n}{2}+1.$$

For this array, the first iteration of our algorithm computes an increasing sequence of length 2, because the second index contains the maximal value. The decreasing sequence is even shorter and only contains the first index, as the first value is the minimum. Therefore, the first two indices will be extracted and as the new maximal and minimal values are again at the beginning of the remaining array, the algorithm repeatedly only computes sequences of length 2. Therefore, the whole array is partitioned into $n/2$ monotonically increasing sequences, even though the optimal partition is really simple and only contains two sequences. As the whole array is iterated $n/2$ times, the computation time is in $\mathcal{O}(n^2)$. Even though these bounds seem impractical, we will see in Section 6.5 that this algorithm shows good results in practice.
We have also implemented some alternative algorithms (mostly small modifications of the greedy algorithm) which have all shown less useful for our needs (some are compared in Section 6.5).
The algorithms we implemented are the following:

- A simple linear time algorithm that only creates subsequences of consecutive indices. This creates at most $\lceil n/2 \rceil$ sequences. The practical runtime of this approach is really good but the number of sequences is not small enough for our means (see Section 6.5).

- An algorithm that divides the sequence in d sequences of size n/d and computes a partition for every part with our greedy implementation. The motivation is that the number of sequences only increases by a constant factor, while the original sequence is iterated over less often.

- An approach that works like the greedy algorithm, but does only add an index to a subsequence if its value is within a certain interval of the last added value. The motivation is that the greedy algorithm may produce short subsequences if the original sequence contains really large and really small values at small indices. However, this approach did not show better results than the greedy algorithm, as the number of sequences actually increased.

- Our first approach was to repeatedly find the longest monotone subsequence. Starting at the first position, we add a new sequence for every index that does not fit in our current subsequences. For indices that fit in one or more sequences, we add a new sequence for every fitting subsequence that additionally contains the current index. This algorithm creates a lot of sequences in every iteration and therefore its runtime is really high, even for small sequences. Therefore, we discarded this idea early on.

6.4.3 Renaming the Variables

To rename the variables in the dictionary, we use a breadth-first search (BFS, see Section 2.3.1) on the graph representation of the SLP. In our implementation, the BFS is done directly on the dictionary without changing its representation. We use a `std::queue<uint64_t>` to implement the queue of processed vertices and a `bit_vector` that indicates for every variable if it has already been renamed. The new names for the variables are stored in a `std::vector<uint64_t>`.

We implemented the search naively using the dictionary. This is space efficient, because we only need additional space for the queue and the search results. Before the BFS, we store the reversed dictionary in a `std::unordered_map<uint64_t,std::vector<uint64_t>>` (The vector is needed because a variable can occur on the right side of multiple rules). Therefore we are able to find the left side of the rules, meaning all indices j with $D[j] = i$, in expected constant time during the BFS. Therefore, the time needed for the BFS is $\mathcal{O}(n)$ for n variables, because the graph representation has exactly two edges per variable. The time needed to build the hash table is also in $\mathcal{O}(n)$, as we only have to take each edge into account once.

6.4.4 Creating D_π

The array D_π corresponds to D_ρ sorted with respect to the values from D'. To realise that, we use `std::pair<uint64_t,uint64_t>` to connect the values $D'[i]$ and $D_\rho[i]$ for every index i. We store the pairs in a `std::vector` of pairs and sort this with `std::sort` with respect to the copied values from D'. Afterwards, we have to extract the values from D_ρ in the new order to store them in D_π and construct the wavelet tree.

6.4.5 Wavelet Tree

We use the wavelet tree from Section 2.8 to store the arrays D_π and D_π over the alphabet $\{1,\dots,n\}$ from Section 6.2. Our implementation concatenates the bit vectors in the nodes of a layer and therefore, we store one bit vector of length n for every layer of the tree. The wavelet tree itself is then stored as a `std::vector<bit_vector>`. We construct the tree bottom-up, as described in [FK17]. The implementations of the access, rank and select operations use the algorithms from [CNP15].

6.4.6 Calculating Rule Lengths

To make the access to our data structure fast, we calculate the length of the string each rule produces and store them as a `std::vector<size_t>` We can do this bottom-up from the rules that only produce a single symbol. After filling the array with these values, the string lengths of the rules can be calculated as follows: for a rule $X_a \to X_b X_c$ it is $\texttt{string\_length}(X_a) = \texttt{string\_length}(X_b) + \texttt{string\_length}(X_c)$. Since the variables were renamed in one of the previous steps (Section 6.4.3), it can be possible that the string length of a variable X_c was not calculated yet. If this happens, we go into a subroutine to calculate the string length of X_c and, recursively, all other variables it may need that have not yet been calculated. To make sure that every variable is only calculated once, each result of the main- and subroutine is saved inside our array and can be used for other calculations.

6.4.7 Operations

To realize the needed operations on our data structure, we used the idea from Section 6.3 to use the string lengths.
We implemented the operations in the following way:

- `size()`: returns the length of the string `s`,
- `access`(i): finds the rule that produces $s[i]$ and return the symbol,
- `contains`(s'): iterates through `s` and look whether `s'` is a substring, and
- `lce`(i, j): iterates through `s` starting from indices i and j and compare values until they differ.

The `size` operation takes constant time as we store the string size in a member variable of the main class.
For the `access` operation we can go through the string lengths array. To find a position i in the string, we take the first rule of the grammar and check if i is greater than the string length of the left side of the rule. If this is the case, we go into the variable on the right side and repeat the process while decreasing i by the length of the left side of the rule. If we reach a rule that produces only a single symbol we return that. This method needs logarithmic time depending on the number of rules.
For `contains`, we can separate queries for single symbols and longer substrings. For single symbols, we can check the symbol list we store whether the wanted symbol is contained. For longer substrings, we use the `access` function to return symbols and check them against the substring one by one.
The `lce` operation uses a similar approach by using `access` to compare two symbols at a time.
To summarize, we compromised on the need of faster operations by using more space. The approach of reconstructing the whole string would save some extra space here, but is a lot slower, as the string has to be derived for every operation. The improved `access` operation is the main part we were trying to improve and we will see in the results later how big the time improvements and space trade off are.

6.5 Benchmarks

We implemented benchmarks to evaluate different parts of our data structure, namely:

- the computation of monotone subsequences of a sequence,
- the succinct way to store a SLP from [TTS13] and how efficiently we can reconstruct the original rules, and
- our text compression based on SLPs with the operations `access`, `size`, `lce` and `contains`.

All benchmarks were executed on the Lido3-Cluster, as described in Section 3.2.

6.5.1 Calculating Monotone Sequences

In Section 6.4.2, we described our approaches of calculating a partition of monotone subsequences. The goal of this benchmark was to find the approach that is most practical for our needs. The benchmark generates a number of random sequences of `uint64_t` values with a given length. Then a partition was computed with the algorithms “easy_sequences”, “greedy_sequences” (with a parameter for the maximal length of a subsequence) and “greedy_partition_sequences” (with a parameter for the number

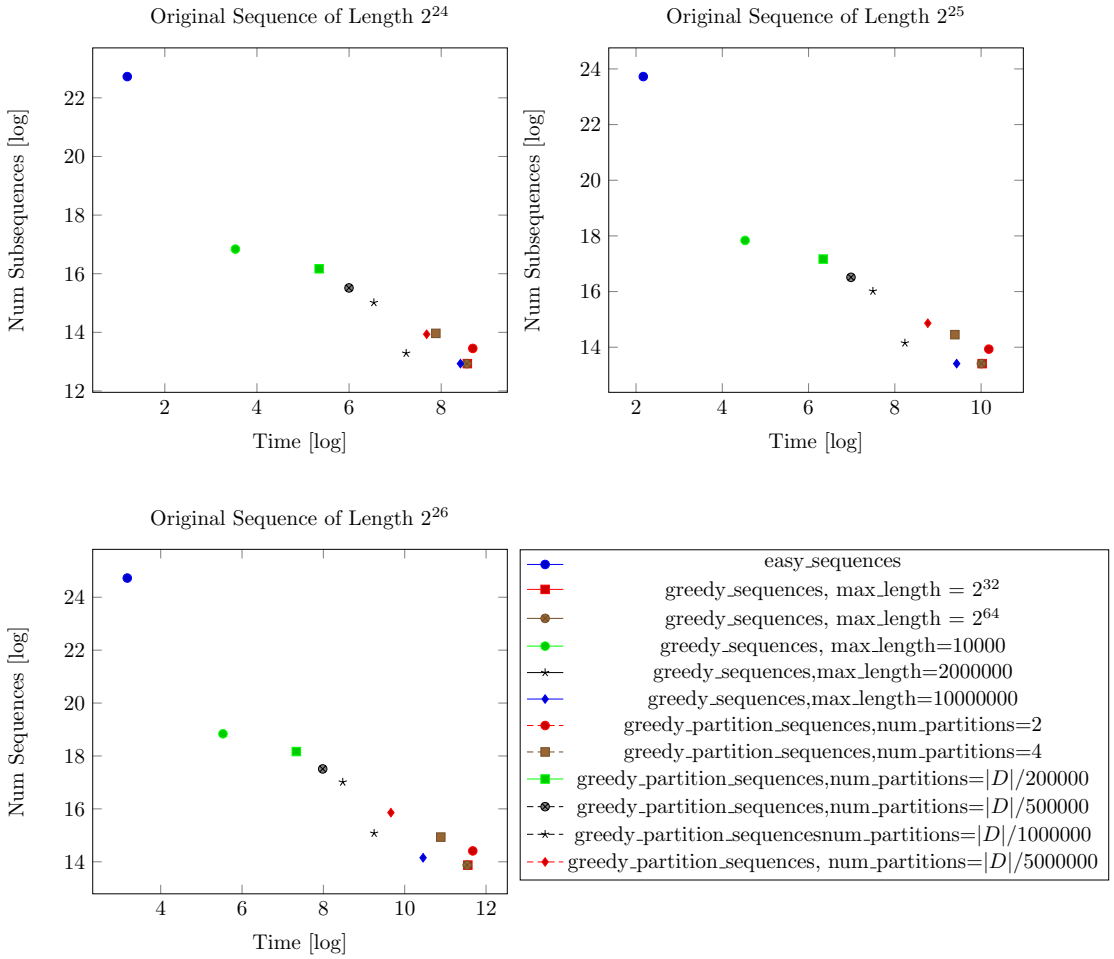


Figure 6.6: Benchmark results for the calculation of monotone subsequences. Results for random sequences of length between 2^{24} and 2^{26} were plotted in one graph, respectively. On the x axis, the time needed for computation is plotted on a logarithmic scale. The y axis shows the number of calculated sequences on a logarithmic scale.

of parts the original sequence is divided into). The other approaches showed to be way less efficient regarding both time and the number of subsequences. Therefore, we decided to not include them in the benchmark results, as they could not be executed on large sequences. The number of generated sequences, iterations and length can be given as input to the benchmark. We executed the benchmark on sequences of length up to 2^{26} and with multiple parameters. For our goal of implementing a succinct grammar representation, the most important aspect is that the number of computed subsequences is as low as possible. Also, the time needed for computation should be reasonable.

The results of this benchmark are plotted in Figure 6.6. While the "easy_sequences" is really fast, the computed number of sequences is way too high to be of practical use for us. The results of the "greedy_partition_sequences" turn out to be worse than the standard greedy algorithm, both in time and space usage. We therefore decided to use the "greedy_sequences" algorithm for the construction of our SLP representation. Altogether, its results seem satisfying, and much better than the worst case described in Section 6.4.2. As the number of calculated sequences is our most pressing aspect, we decided to take 2^{64} as parameter in practice.

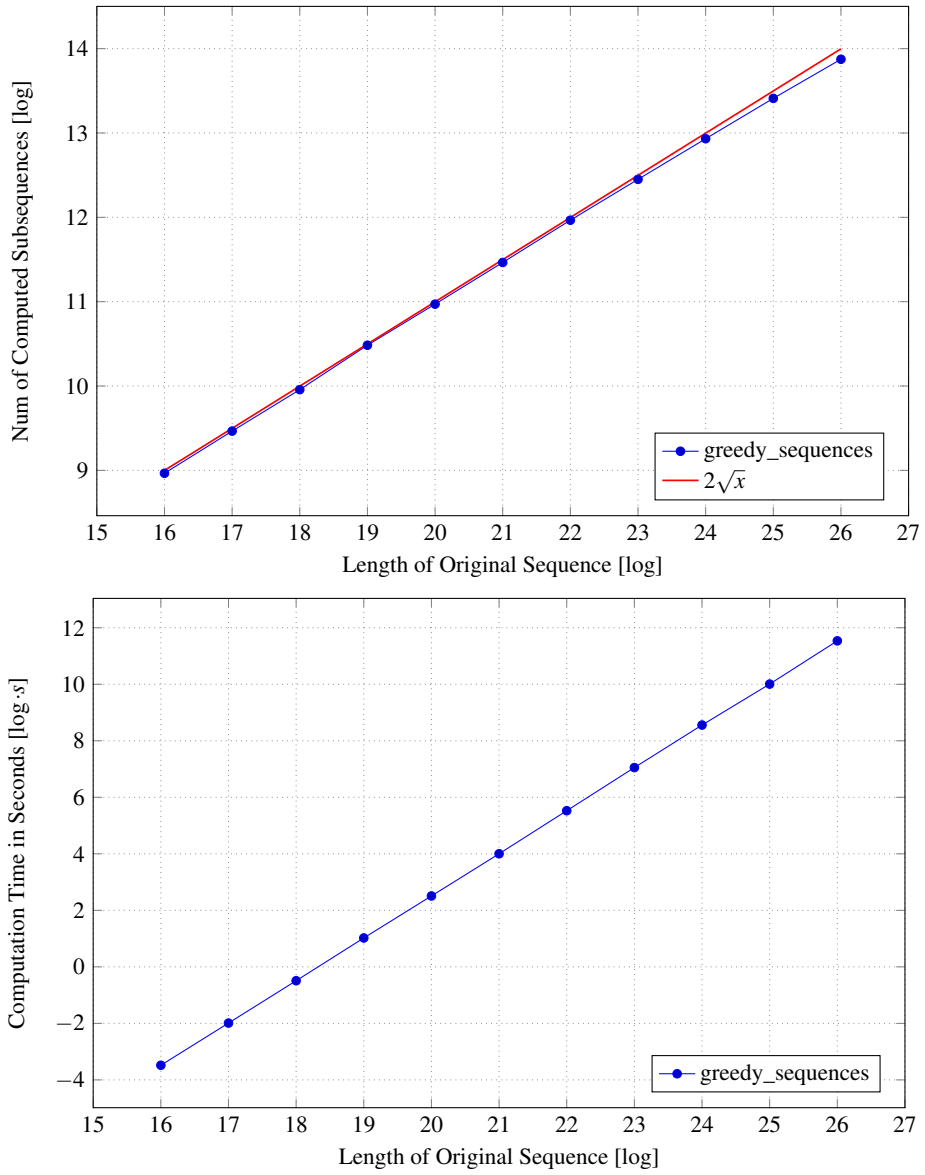


Figure 6.7: Benchmark results for the calculation of monotone subsequences using the greedy_sequences algorithm. Results for random sequences of length between 2^{10} and 2^{26} are shown. On the x axis, the lengths of the original sequences are shown on a logarithmic scale. On the y axis we see the number of computed subsequences and the runtime on a logarithmic scale, respectively. We use the function $2\sqrt{x}$ as comparison for the number of calculated sequences.

In Figure 6.7 more detailed results for greedy_sequences are shown. We can see that the number of calculated sequences seems to be really close to the bound of $2\sqrt{n}$ that was shown in [BF98]. This is satisfactory as a result, even though one has to consider that a partition into $2\sqrt{n}$ sequences is not an optimal solution, but merely a proven upper bound. The second plot shows that the computation time is linear in the length of the original sequence. Both, number of sequences and computation time turn out to be much more practical than the worst case analysis in Section 6.4.2 indicated.
Unfortunately, the dictionaries in our data structure can get really long, so that even a linear-time computation of the partition will still take up a huge part of construction.

6.5.2 Succinct Storing of SLPs

One part of our implementation is the succinct data structure for SLPs (see Section 6.2). The goal of this structure is to store a SLP in a succinct manner. The space usage should be as low as possible while still allowing to reconstruct a rule from the SLP in little time. In Section 6.2 and Section 6.3, we saw that the theoretical space usage for a SLP of size n is $2n\log m(1+o(1))$ bits with $m \leq 2\sqrt{n}$ and that a rule can be reconstructed in $\mathcal{O}(\log\log n)$ time. It is clear that the space usage can only fulfil this bound with an optimal algorithm for computing monotone sequences. Therefore it is probably not possible to implement the data structure space-optimally and with a practical construction time. We compare our succinct data structure to the naive approach of storing the dictionary in a `std::vector<uint64_t>`.
For our benchmark, we generate a number of SLPs from random texts of given length using the RePair algorithm. Since we use our succinct storing in our library only for dictionaries generated by RePair, it was our goal to see how well the succinct storing stores those SLPs. Since random texts are not likely to be very repetitive we can assume that a big part of the generated SLPs are constructed by our algorithm that splits the starting variable, as described in Section 6.2. After generating the SLPs, our data structure is constructed, but only the parts described in Section 6.2 and then the additional data structure, which contains the lengths of the string per variable and is needed for the `access` operation. This means that the construction time here does not include the RePair algorithm and the construction of the naive dictionary is merely a copy instruction. After construction, a number of `get_rule` operations is executed on the naive and succinct data structure.
We generate strings and corresponding SLPs for every size of the form 2^x, the maximal x can be given as a parameter to the benchmark, as well as the number of generated SLPs per size, the number of `get_rule` operations performed per SLP, and which type of dictionary should be constructed (‘naive”,“succinct”,“succinct_lengths” or “all”).
The results of this benchmark are shown in Figures 6.8 to 6.10.
In Figure 6.8 the space usage of our dictionary implementation is shown. The naive dictionary constantly takes 16 bytes of space per rule, as every rule is stored as two 64 bit integers. We can see that the succinct dictionary is useful for dictionary sizes between 2^{11} and 2^{21}. For larger dictionaries, the naive approach seems to be more space efficient. The succinct dictionary that additionally stores an array of lengths of course takes exactly 8 additional bytes per rule. This takes up more space than the naive dictionary for every dictionary size, but makes up with a better runtime for the `access` operation. The graph

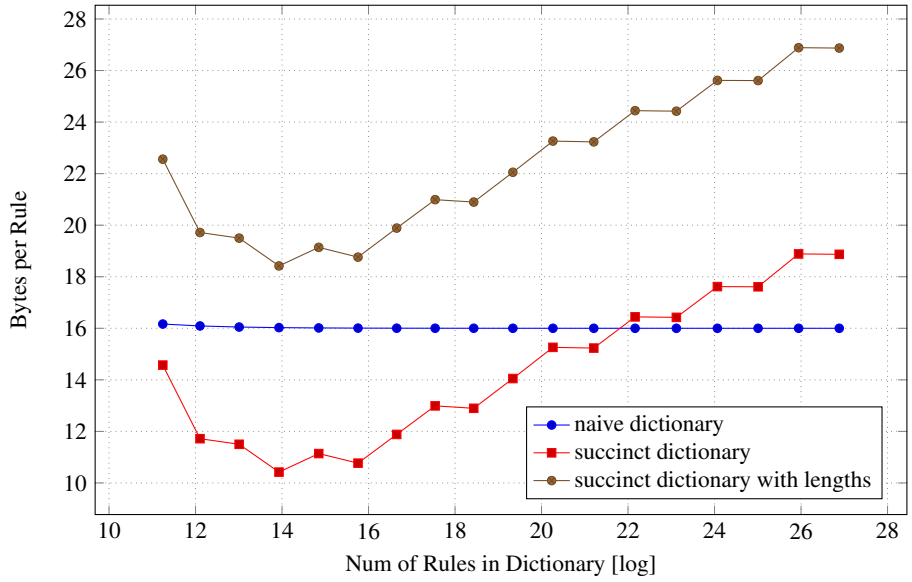


Figure 6.8: Benchmark results for the space usage of a naive dictionary and our succinct data structure, with and without the additional string lengths array needed for a fast access of a grammar compressed string. The input are SLPs generated by RePair on strings of lengths 2^{13} to 2^{30}. The x axis shows the number of rules in the randomly generated dictionary (in powers of two). The y axis displays the number of bytes that are needed as space per rule.

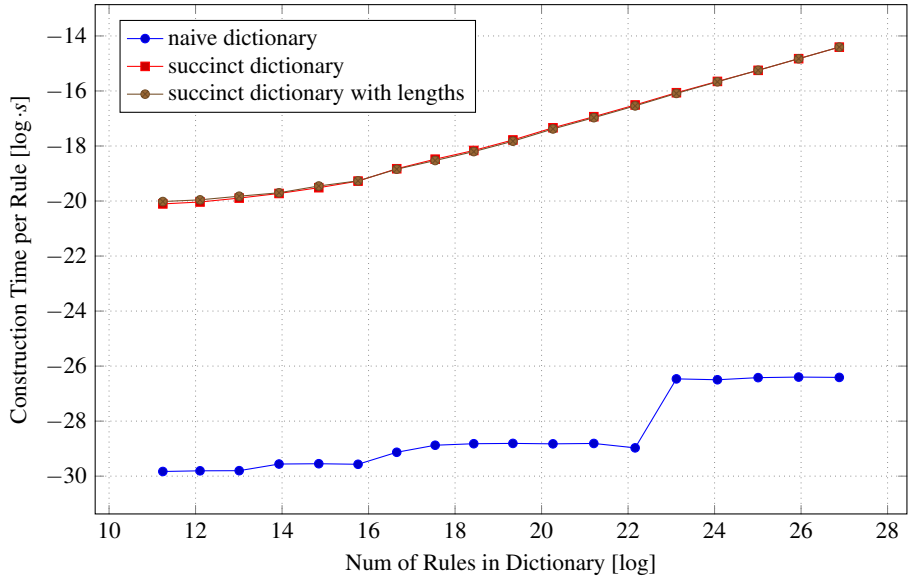


Figure 6.9: Benchmark results for the construction time of a naive dictionary and our succinct data structure, with and without the additional string lengths array needed for a fast access of a grammar compressed string. The input are SLPs generated by RePair on strings of lengths 2^{13} to 2^{30}. The x axis shows the number of rules in the randomly generated dictionary (in powers of two). The y axis has the time in seconds that is needed for construction per rule.

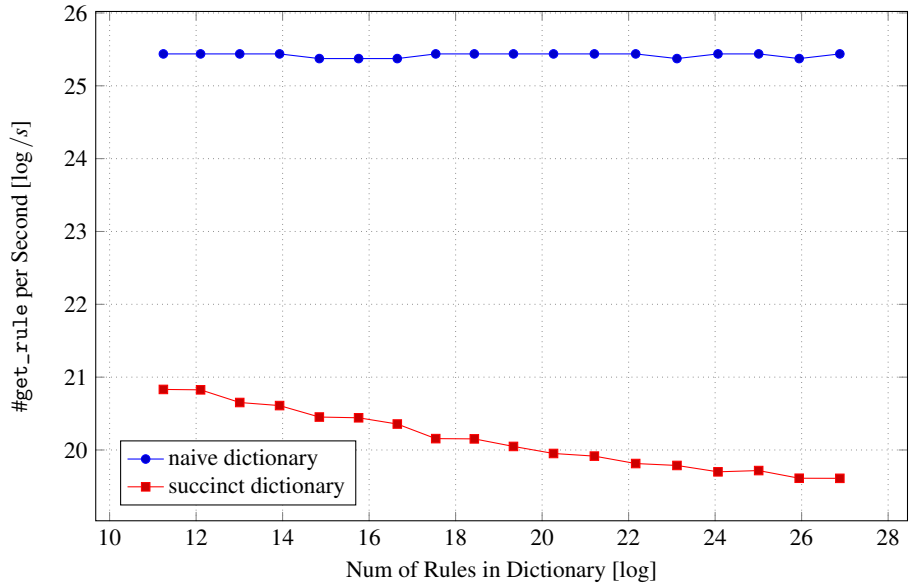


Figure 6.10: Benchmark results for the time usage of the `get_rule` operation on a naive dictionary and on our succinct data structure. The input are SLPs generated by RePair on strings of lengths 2^{13} to 2^{30}. The x axis shows the number of rules in the generated dictionary (in powers of two). The y axis shows the number of `get_rule` operations that can be executed per second (in powers of two).

for our succinct implementation shows some interesting behaviour. The higher space usage for small dictionaries seems to be comprehensible, as we store bit vectors with additional structures for rank and select queries. These are likely to take up the main part of the space for really small dictionaries. As for the larger dictionaries, it seems interesting that the space usage per rule only increases every second point. This behaviour corresponds to the number of layers in the wavelet tree. For example, dictionaries of sizes 2^{22} and 2^{23} always produced wavelet trees with 13 layers, while those of size 2^{24} needed 14 layers. This explains the sudden increase in space. In turn, the number of layers in the wavelet tree depends on the number of calculated monotone subsequences, as the number of subsequences is the alphabet size of the wavelet tree. Here, we were able to observe a similar behaviour. The number of calculated sequences increases more for every second dictionary size. As we were not able to observe this behaviour when benchmarking our algorithm on random sequences (see Section 6.5.1) we assume, that the special form of SLPs and our algorithm to split the starting variable induced these results.
In Figure 6.9 we can see and compare the construction time of the naive and our succinct dictionary. For the naive dictionary, this is still copying the dictionary. Therefore, it is clear that our succinct dictionary will need significantly more time to construct. The construction time for the naive dictionary jumps between 2^{22} and 2^{23}. Most likely, this is the point where the dictionary cannot fully be stored in the cache. As we used 25 MB caches for the execution of our benchmarks and need $(8 = 2^3)$ bytes for the naive storing of a rule in the dictionary, this seems reasonable. We see that the computation of the lengths does not increase the construction time of our succinct data structure significantly. The overall construction time for large sizes seems to correspond to the time needed for computing the partition into

sequences as seen in Figure 6.7. This was predictable, since this is the only part of construction that takes more than linear time in the size of the dictionary.
In Figure 6.10 we see the throughput for `get_rule` operations. For the naive dictionary, this takes constant time, as we only have to read two values from an array. It is clear that the `get_rule` operation of the succinct dictionary is not able to achieve constant time. Nevertheless, the plot shows that our implementation corresponds to the theoretical $\mathcal{O}(\log \log n)$ bound, where n is the size of the dictionary.

6.5.3 Grammar Compressed Strings

To evaluate our succinct string compression, we take into account the space usage, construction time and operation time. The goal is to compress the text as much as possible, while still allowing efficient operations.
We compare our data structure to the naive approach to merely storing the whole text without any compression as a `std::vector<char>`. In this approach all operations were implemented naively, achieving constant runtime for `access` and `size`. The operations `contains` and `lce` have a linear time bound. In our succinct data structure we also follow the naive approach for `contains` and `lce`. We therefore will not be able to exceed the operation time of the naive approach, but try to get as near as possible to these bounds. As discussed in Section 6.4.7, in our succinct implementation we realised the `access` operation utilizing the string lengths array, resulting in a theoretic logarithmic time bound.
Of course, our main goal is to achieve a profoundly better space usage than the naive approach and therefore accept the worse time bounds for construction and operation.
We tested on the texts of the Pizza and Chili Corpus [2], that were made to have reference texts for benchmarks on text indexing and other applications on texts. The corpus features highly repetitive texts, which are of special interest to us, as RePair is designed to efficiently compress those in particular.
We used the following standard texts:

- *pitches*, pitch values of various MIDI files,
- *sources*, source code of linux and gcc distributions,
- *proteins*, protein sequences,
- *dna*, gene DNA sequences,
- *english*, a concatenation of english text files, and
- *XML*, bibliography information on computer science journals.

For the *english* text, we had to use a prefix of 1024 MB, since the size of the full text exceeds the maximal value of a 32-bit signed integer. Unfortunately, the implementation of RePair that we used does not work on strings of that length.
Since our data structure is working best on repetitive texts, we also considered the texts from the repetitive corpus of Pizza and Chili, both artificial and real:

[2] http://pizzachili.dcc.uchile.cl

real	*cere, coreutils, einstein.de.txt, einstein.en.txt, Escherichia Coli, influenza, kernel, para, world leaders*
artificial	*fib41, rs.13, tm29*

Text	Size (bytes)	Alphabet size	Compressibility
SOURCES	210,866,607	230	28.01%
PITCHES	55,832,855	133	30.59%
PROTEINS	1,184,051,855	27	49.71%
DNA	403,927,746	16	32.46%
ENGLISH	2,210,395,553	239	45.07%
XML	294,724,056	97	21.01%

Table 6.1: Statistics about the used data. The compressibility shown is a measure of the size compressed text divided by the original size. Method used here is Gzip (www.gzip.org) with parameter -1. The other texts had no such data available.

Some insight into the used data is given in Table 6.1. Firstly, we test the construction time and space requirement of our data structure. In the case of space usage, we take a look on the total space needed for our data structure **after** construction and compare it to the naive version and the explicit storing of the SLP as a dictionary. A small dictionary size means that RePair compresses the text well and produces a small starting variable. Our succinct representation aims at compressing the dictionary and storing it as small as possible, while still allowing efficient operations. To measure the time constraints, we take a more detailed look into the construction process and measure each major step in building our data structure (see Section 6.2). For all of these steps, we will need to look at a median over all the different files to eliminate any inaccuracies the texts may cause by themselves.
After construction, the benchmark can make a more detailed analysis of space usage and measures the space used by the components of our data structure. Afterwards, we execute the operations on the text, both for the naive string and our succinct version.
To execute our benchmark directly, it needs three command line inputs. The first is the file path of the file you want to execute the benchmark on. Then, the next four arguments specify the number of times the operations (in the order `access`,`size`,`lce`,`contains`) are executed. The last input is the number of iterations that the succinct data structure is constructed with detailed space measurement. If you do not want the extra space benchmark, simply choose "0" as the last argument. The executable can be found at `src/benchmark_gcs/benchmark_gcs.o` after building the library.

Space Usage

Firstly, we will analyse the space usage of our implementation. In Figure 6.11, we can see the total space usage of our grammar compressed string after construction for every test file in the fourth column. These numbers do include the space needed for the array of lengths used for our implementation of the `access` operation. The compression ratio compared to the length of the original file is shown in the fifth

text	length text	#rules in dictionary	space gcs (Bytes)	compression% gcs	time total (s)	#access per s	#size per s	#lce per s
fib41	267,914,296	84	2,890	0.0	33.73	253,164.56	2,717,391.30	69,681.56
rs.13	216,747,218	182	7,282	0.0	24.43	99,745.65	2,732,240.44	46,231.02
tm29	26,8435,456	164	5,770	0.0	31.76	142,592.33	2,890,173.41	40,406.49
einstein.en.txt	467,626,544	326,444	7,834,067	1.68	180.91	9,216.72	2,949,852.51	4,595.77
einstein.de.txt	92,758,441	125,460	2,385,533	2.57	34.16	11,942.22	3,021,148.04	5,869.75
kernel	257,961,616	2,255,000	59,869,256	23.21	129.17	2,340.40	2,808,988.76	1,066.25
cere	461,286,644	6,433,188	141,642,397	30.71	173.60	6,382.15	3,086,419.75	2,156.08
world_leaders	46,968,181	601,846	15,426,457	32.84	14.12	10,546.24	2,724,795.64	4,337.61
coreutils	205,281,778	3,950,630	82,261,284	40.07	121.42	1,601.97	2,594,033.72	451.21
influenza	154,808,555	3,060,874	69,000,143	44.57	56.42	10,342.65	2,739,726.03	4,728.52
para	429,265,758	8,483,054	237,636,341	55.36	200.76	5,219.49	2,785,515.32	2,442.54
Escherichia_Coli	112,689,515	7,227,166	150,914,983	133.92	76.66	6,975.71	2,590,673.58	2,247.53
dblp.xml	296,135,874	26,249,572	587,888,913	198.52	391.77	7,227.57	2,699,055.33	3,601.63
sources	210,866,607	32,469,760	707,429,918	335.49	1,111.78	6,242.06	2,923,976.61	3,007.02
english.1024MB	1,073,741,824	173,233,416	4,583,756,957	426.90	10,553.80	4,055.45	2,680,965.15	360.43
proteins	1,184,051,855	254,664,362	5,534,722,027	467.44	7,307.33	4,148.70	2,724,795.64	1,795.76
dna	403,927,746	85,207,846	2,155,965,384	533.75	1,391.26	6,955.60	2,906,976.74	3,399.95
pitches	55,832,855	18,695,494	532,349,005	953.47	875.71	6,556.22	2,808,988.76	3,168.86

Figure 6.11: Summary of benchmark results for the test text files. For every text, the original length of the text file is shown in the second column. The third and fourth columns show the number of rules in the dictionary that is computed with the RePair algorithm, and the space in bytes used by our grammar compression. The compression ratio in the fifth column was computed as $\frac{\text{space gcs}}{\text{length text}} \cdot 100$. The sixth column shows the construction time needed for the grammar compression, including the RePair algorithm. The last three columns contain information about the operations `access`, `size` and `lce`. For every text, the number of operations that can be executed per second is shown.

column. It is clear to see that the quality of our compression profoundly differs depending on the original text. In general, it shows good results for repetitive texts (the first twelve texts in the table). Especially the artificial repetitive texts can be compressed really well. For them, the grammar compressed string needs less than 0.1% of the original file size. The texts *einstein.de.txt* and *einstein.en.txt* also can be compressed well, with 2.57% and 1.68%, respectively. The other repetitive texts show a compression ratio between 23% and 55.36%. The non-repetitive texts do not show satisfying results altogether. For *pitches*, our grammar compression does need nearly ten times as much space as the original text. We can explain these results with a look at the dictionary sizes. It is clear that our compression heavily relies on the RePair algorithm and the size of its resulting dictionary. While this dictionary is really small for the artificial repetitive files, the non-repetitive files produce dictionaries that are nearly half as large as the original string and therefore will naturally need more space to be stored. Here it is to be said that the large dictionaries are probably not a result of the RePair algorithm itself, meaning that RePair may produce a way smaller grammar compression with a long start variable. The splitting of this start variable as described in Section 6.2 then produces a huge SLP and dictionary.

In Figure 6.12, we can see this connection in more detail. Here, we plot the space usage of our grammar compression against the size of the dictionary computed by RePair. The data points represent our test texts. We see that the dictionary size and space usage correspond approximately linearly. We also see that only some of our text files produce a dictionary with a length between 2^{11} and 2^{21}. This is the interval for which our succinct storing of SLPs seems to be practical, as seen in Section 6.5.2.

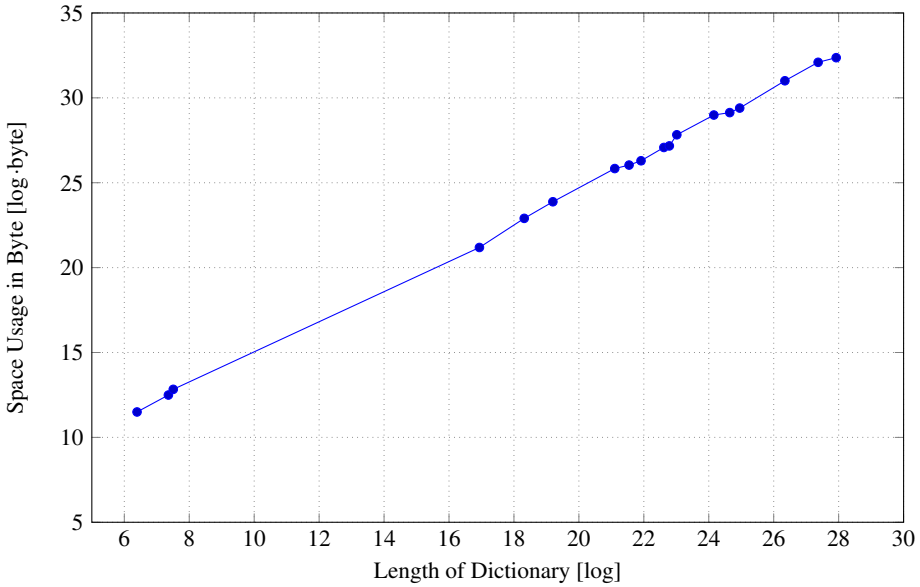


Figure 6.12: The plot connects the dictionary size with the space usage of our grammar compression for the test text files. The x axis shows the length of the dictionary, that the RePair algorithm computed, on a logarithmic scale. On the y axis the space usage in bytes of our compression is plotted, also on a logarithmic scale. The data points represent our texts.

In Figure 6.13, the size of our compression is illustrated a bit more. We compare it with our grammar compression that does not include the lengths array, as well as with a naive storing of the dictionary computed by RePair. The space is measured relatively to the original string size. The plot shows the space needed for one byte of the original text. Therefore, smaller values are better in practice. We mostly see the results that were already stated above. Our compression is really useful for the artificial repetitive text, as well as for the `einstein` texts. The other repetitive texts at least show some practical use. For the non-repetitive texts, we can see that even the naive dictionary does not compress the texts. As our implementation with the lengths array, that ensures a faster `access` operation than the naive dictionary, does need more space than the naive dictionary in all cases, it is not surprising that we do not achieve compression either. We also see that our succinct storing of the dictionary, even without the additional length array, is seldomly useful for the test files. It only shows some small improvement for *Escherichia_Coli, cere, einstein.de.txt, einstein.en.txt, influenza, kernel* and *world_leaders*. Here we can again conclude that these are the only files where the size of the dictionary is in the useful interval.
To analyse the space usage better and to see which parts have room for improvement, we plotted the space usage in more detail for the non-repetitive texts in Figure 6.14. Here, we see the parts of our data structure and their space requirement in bytes on a logarithmic scale. We see that the array of lengths does take the most space for all the texts. Still, we actually assumed that the space requirement for this array would be a lot higher compared to the other parts, as the array of lengths is always half the size of the original dictionary. However, for these big dictionaries produced by the non-repetitive texts this does not seem to matter much, as the succinct storing of the dictionary does not work well in general. Furthermore, we see that the wavelet trees and the gap encodings have the highest space usage. This

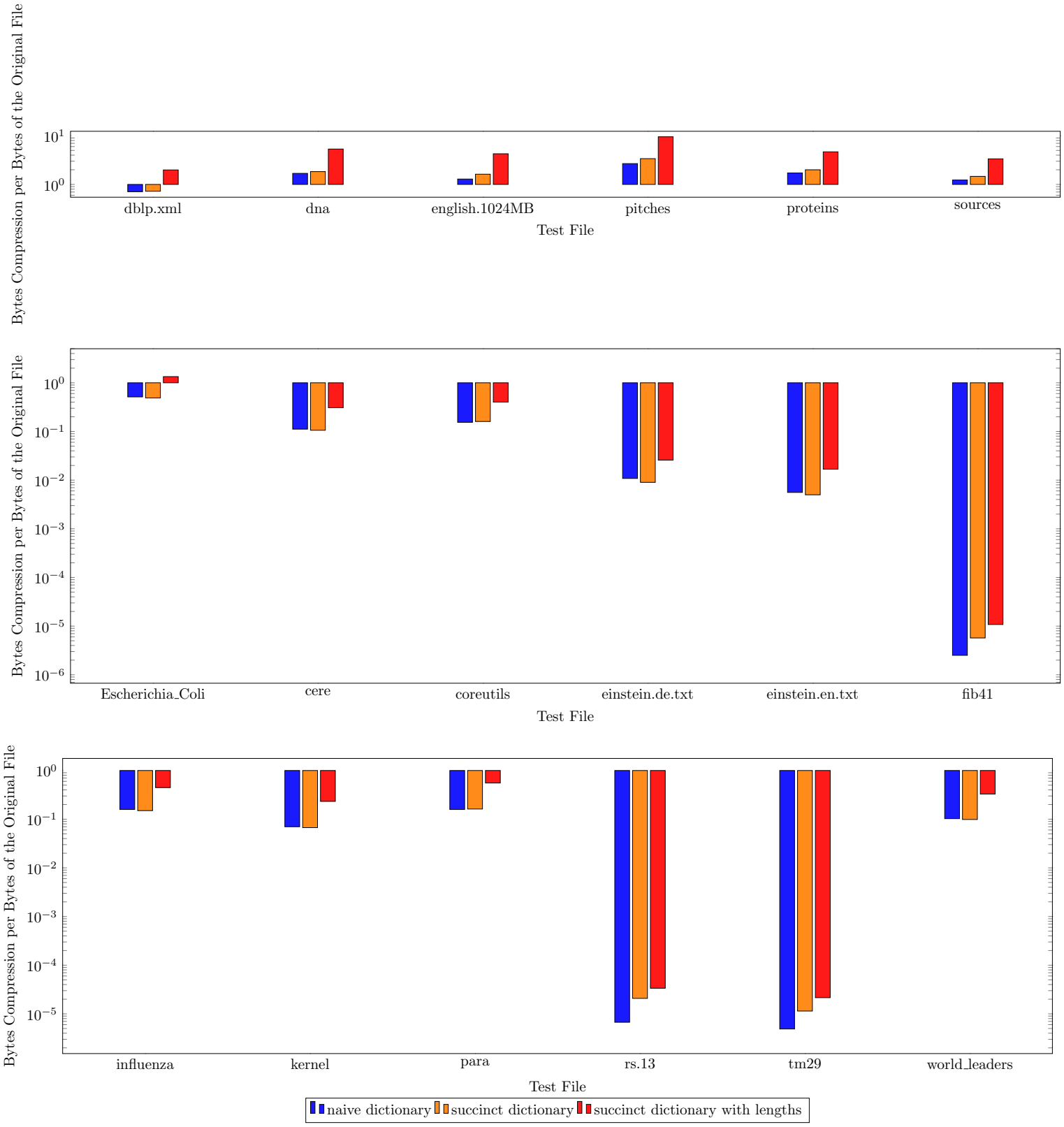


Figure 6.13: The plots show the space usage of the naive dictionary and our grammar compression (with and without lengths array) for our text files. On the *x* axis are the test files. The *y* axis shows the bytes that are needed to store one byte of the original text on a logarithmic scale.

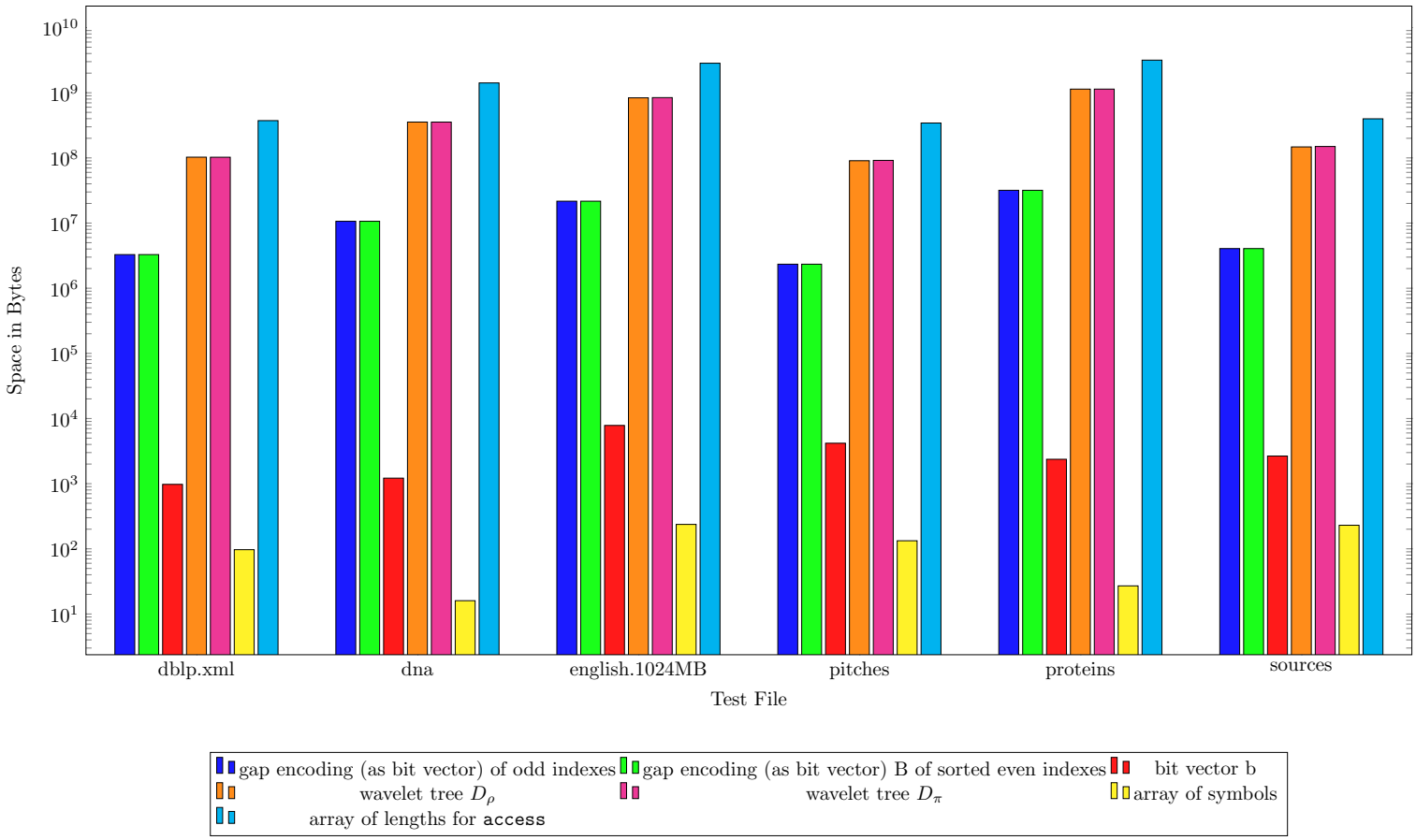


Figure 6.14: The plot shows the space usage of the parts of our grammar compression for the non-repetitive test files. The test files are on the *x* axis. The *y* axis shows the space in bytes on a logarithmic scale.

does seem reasonable when viewing from a theoretical point of view. For *dblp.xml*, the wavelet trees need about 4 bytes per dictionary entry. Considering that the wavelet tree additionally stores the rank and select data structures for every inner bit vector, as described in Section 2.5, this seems reasonable.

Construction Time

In this section, we will take a closer look on the construction time of our data structure. In Figure 6.11, we can see the total construction time needed in the sixth column. It ranges between 14.12 seconds for *world_leaders* and 10,553.8 seconds for *english.1024MB*. We saw in Section 6.5.2 that the construction time of the succinct dictionary grows in accordance with the size of the dictionary. This can be seen in the table, as the non-repetitive texts, which imply huge dictionaries, also need a long time for construction. Still, we see that the construction time also depends on the text size and the text itself, as the RePair algorithm generally does take more time for longer and less repetitive texts.
In Figure 6.15, we see the this in a bit more detail. In general, RePair does need more time for larger texts and this explains why *world_leaders* has the smallest construction time overall. But wee see that the dots corresponding to the *fib41, tm29* and *rs.13* show a significantly lower runtime, as these texts can be compressed best.

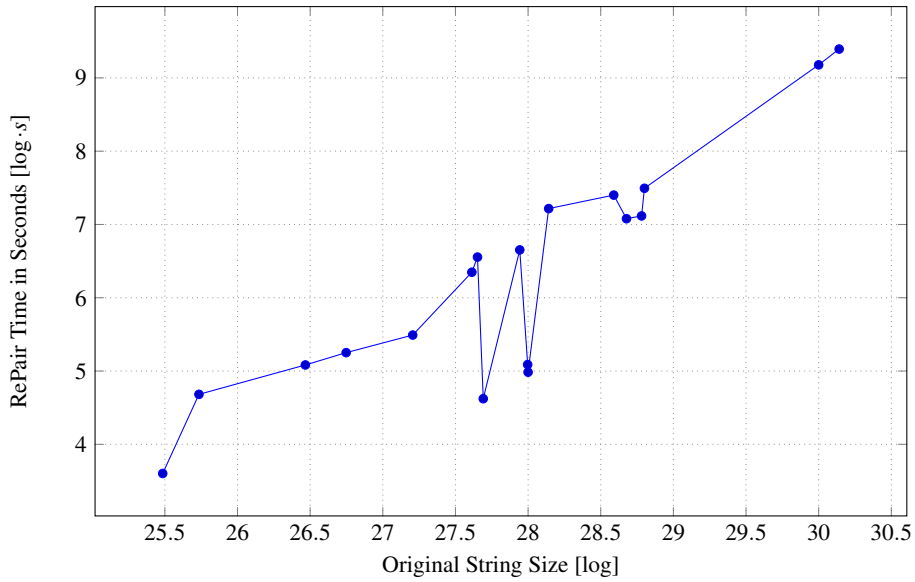


Figure 6.15: The plot shows the time of the RePair algorithm based on the size of the original text. On the *x* axis the original string size is shown on a logarithmic scale. The *y* axis shows the runtime of the RePair algorithm on a logarithmic scale. The data points correspond to the test texts.

To analyse the construction time in more detail, we take a look at Figure 6.16. Here, we see for every test file how much time the individual parts of construction take up. The observations we can make here correspond to the previous analysis. For the artificial repetitive text, where the dictionary is very small, the RePair algorithm nearly accounts for all of the construction time. For the non-repetitive texts the runtime of RePair is overshadowed by the dictionary construction, even though we saw in Figure 6.15 that RePair takes significantly longer for those text files. The longest part of construction is always the computing of monotone subsequences. Especially for the large dictionaries from *english.1024MB* and *proteins*, the time nearly reaches 10^4 seconds, meaning over 2 hours. The breadth first search on the dictionary, that is needed to sort the odd indices to be monotonically increasing, does also take more time for larger dictionaries. Since the runtime should be linear, this seems reasonable. The BFS, as well as the construction of the wavelet trees, still need less time than RePair in any case.

Operations

Now, we will look at the time requirements of our implementation of the operations `access`, `size` and `lce`. We do not show details for `contains` here. As we implemented this operation naively, in the worst case (when a text does not contain the string) this operation has to reconstruct the whole string from the succinct dictionary. As we implemented `contains` using the `access` method, this method turned out to be really inefficient and of no practical use. Therefore, we will disregard this operation for now, and will leave the implementation of a more efficient `contains` operation as a possibility for further research.
For the `access` operation, we can take a look at the seventh column of Figure 6.11. Here, we can see the mean number of `access` operations that can be executed per second on the succinct storing of our test files. Here, the succinct storing includes the array of lengths. The naive implementation that

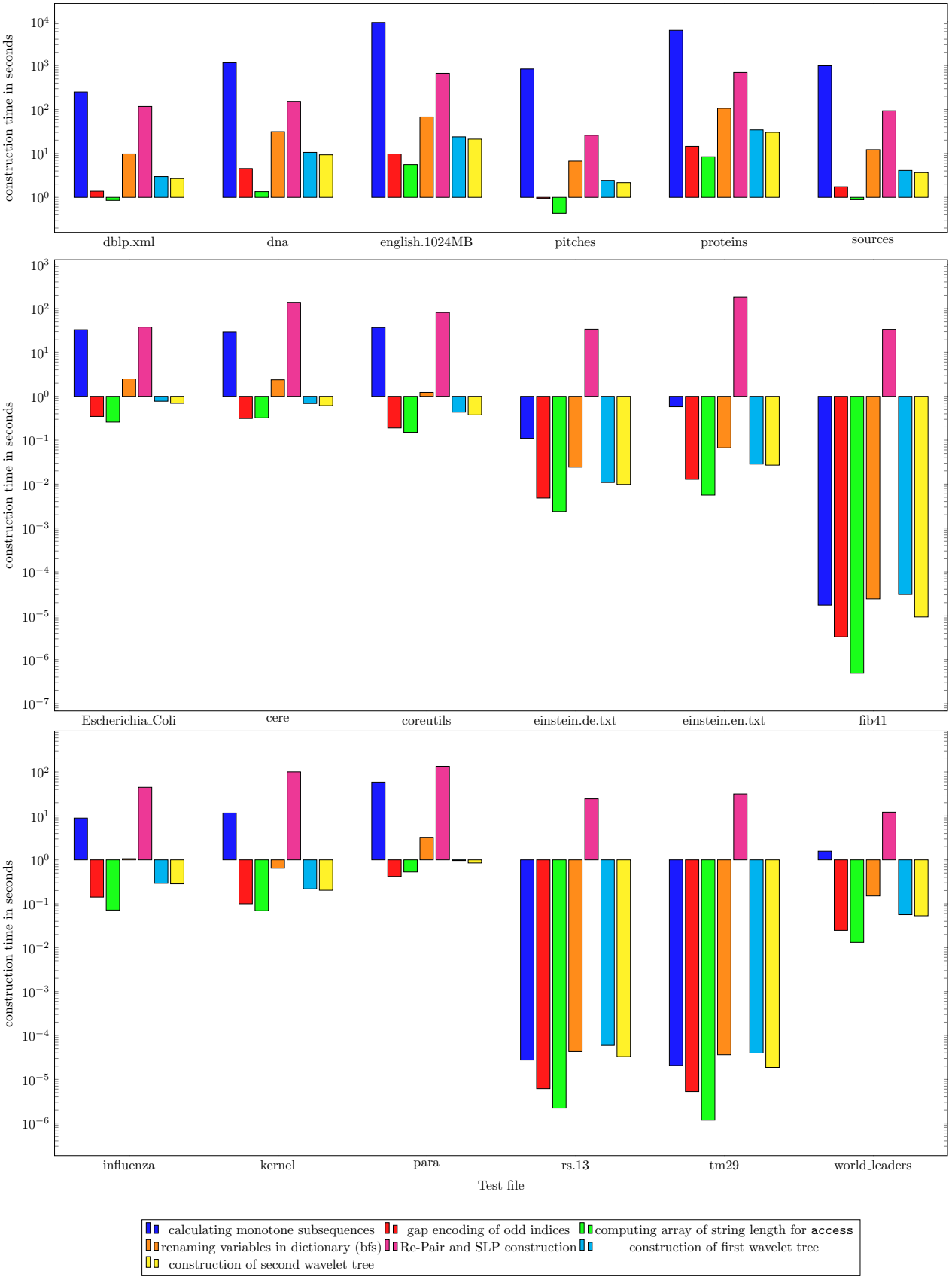


Figure 6.16: Detailed running times of the parts of construction of our grammar compression. On the x axis are the test files. The y axis shows the runtime in seconds on a logarithmic scale.

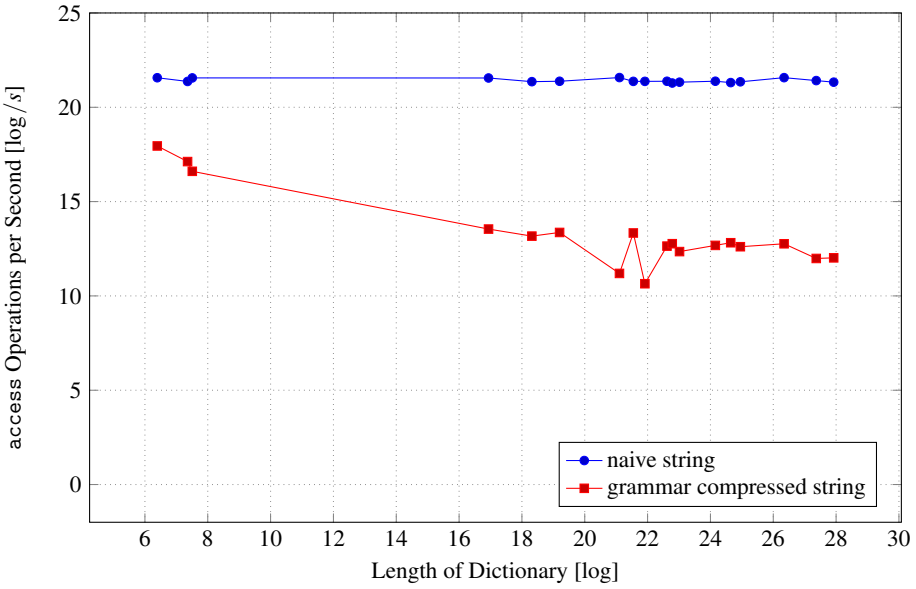


Figure 6.17: The plot compares the `access` operation of the naive string and our grammar compression. The x axis has the length of the dictionaries that RePair constructs on a logarithmic scale. The y axis shows the number of operations per second on a logarithmic scale.

reconstructs the whole string instead of individual symbols turned out to be of no practical use. We see that the number of `access` operations per second shows a similar behaviour to the space usage and the construction time: for the artificial repetitive text the number is highest. The real repetitive texts and the non-repetitive texts show worse numbers, but the `access` operation still shows practical results. This can also be seen in Figure 6.17. In this figure, the number of `access` operations per second is plotted against the size of the dictionary. We can see that the graph is logarithmic. This corresponds to our theoretical run time from Section 6.4.7. In particular, for our largest dictionaries, the number of `access` operations is still satisfactorily high at about 10^{12} per second. In Figure 6.17 we also compare the naive string with our compression. As the `access` operation for the naive string does only read from a `std::vector` and therefore needs constant time, it is clear that a compression will most likely not reach similar results. Nevertheless, for the artificial repetitive texts, we at least get relatively close and our run time gets close to constant for large dictionaries.

The `lce` operation does mainly rely on the `access` operation in our implementation. As expected, the `lce` operation shows a similar behaviour in our benchmarks. In Figure 6.11, the mean number of operations per second can be found in the last column. For the non-repetitive texts, this number is in general about half the number of `access` operations per second. For very repetitive texts, like *fib41*, it is a smaller portion. We expected this, since the implementation of `lce` does more `access` operations when the symbols on the two given indices (and following indices) are the same. This is more likely for repetitive files. The plot Figure 6.19 therefore looks really similar to the corresponding plot for the `access` operation. The curve is just a bit flatter in the beginning, as the texts get less repetitive with larger dictionary size. The comparison to the naive dictionary also yields the same results. The naive

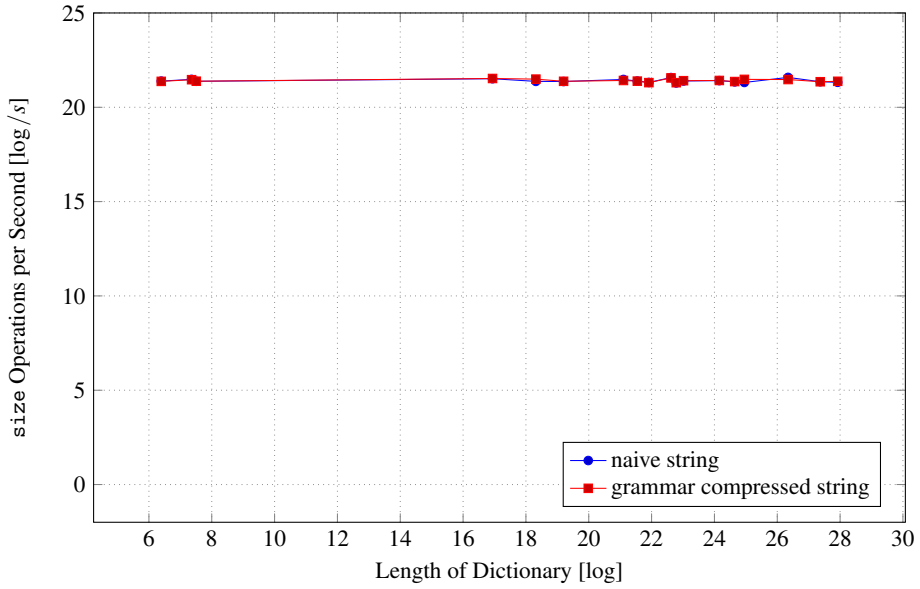


Figure 6.18: The plot compares the `size` operation of the naive string and our grammar compression. The x axis shows the length of the dictionaries that RePair constructs on a logarithmic scale. The y axis shows the number of operations per second on a logarithmic scale.

implementation also relies on the `access` operation and therefore shows the same behaviour as the naive string does in Figure 6.17.

For the `size` operation, we can show good results. As shown in Figure 6.11 in the eight column, the number of `size` operations that can be executed per second is really high, up to three million. In Figure 6.18, we can see that these numbers correspond to the results for the naive string. This is explained fairly easy, as we store the size of the original string explicitly and therefore only have to return it.

6.6 Summary and Further Work

In conclusion, we managed to build a data structure that is able to store texts succinctly in regards to space and within a reasonable runtime. This applies to mostly smaller texts and texts that can be compressed into small grammars. If the grammar is small, we barely need any space compared to the original text. On bigger grammars our implementation gets worse, in some cases even worse than just storing the dictionary as an array, but it is still feasible for use.

A big challenge for the construction of the data structure was finding a good partition of monotone sequences, which we managed to do fairly well, considering that the problem is NP-complete. Still, we believe our algorithm can surely be improved to make the overall compression better regarding space and time.

We also made several operations on the original text possible through our data structure. While we needed some extra space to make the access to single symbols inside the text possible in a reasonable time frame, the running times, again especially for the smaller grammars, were quite good. The fact that

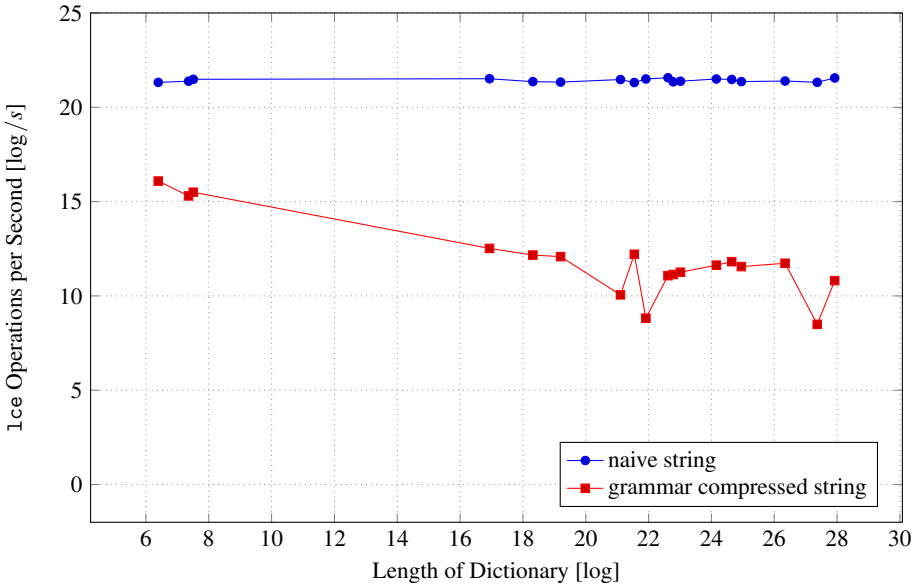


Figure 6.19: The plot compares the `lce` operation of the naive string and our grammar compression. The x axis has the length of the dictionaries that RePair constructs on a logarithmic scale. The y axis shows the number of operations per second on a logarithmic scale.

our `access` function is only really competitive on smaller texts or grammars also shows some scalability issues that we did not manage to fix. Something we also leave for further work is the implementation of the `contains` method, which we were not able to fully realize as we wanted to. Here, a possible solution is building a separate structure for pattern matching inside a grammar, which would however also increase the needed space again. Another point of optimization is the use of the RePair algorithm. Replacing it with another grammar compression could yield smaller grammars and therefore a better compression. A big problem in this case is however our reliance on straight line programs. Any modified grammars, for instance using more than two variables on the right side of a rule, would require heavy modifications of our work.

Chapter 7

Succinct Representations for Graphs

We analyse different methods to construct data structures for graphs. Our target is to support operations on graphs in the same asymptotical time as on naive graph structures with a succinct space requirement.

The data structures we consider shall support the following operations for a given directed, unweighted graph $G = (V, E)$, where the vertices are named by numbers in $[0, \dots |V| - 1]$:

- `size()`, `number_of_nodes()`: returns the number of vertices, i.e. $|V|$.
- `number_of_edges()`: returns the number of edges, i.e. $|E|$.
- `degree`(v): returns the out-degree of $v \in V$.
- `get`(v, i): returns the i-th (outgoing) neighbour of $v \in V$ with $0 \leq i < |V|$ (the neighbour list is sorted increasing by their names).
- `is_neighbour`(u, v): returns true iff there is an edge from u to v with $u, v \in V$.

We implement GLOUDS (Section 7.1), a succinct graph data structure introduced by Fischer and Peters [FP16] for tree-like graphs (Definition 2.3.1). As comparable representations we also implement the adjacency list (Section 7.3.1) and adjacency matrix (Section 7.3.2).

7.1 Representing Tree-Like Graphs with GLOUDS

Fischer and Peters [FP16] designed a data structure optimised for tree-like graphs (Definition 2.3.1). A spanning tree of the given graph can be transformed to a bit vector (LOUDS representation [Jac89], Section 7.1.1), with additional information to store the whole graph.

7.1.1 LOUDS

The *level order unary degree sequence* (LOUDS) [Jac89] encodes a given ordered tree T as a bit vector B. We start at the root and traverse T through a breadth-first search (BFS, see Section 2.3.1). For each

discovered node, we add as many 1 bits as the current node has neighbours to the end of B. If the current node is finished, we write 0 [FP16]. So, let k be the number of neighbours of a node, we add 1^k0 to B. An example for the construction of LOUDS can be found in Section 7.1.3. With rank and select queries on B we can identify the corresponding children and the parent of a given node in the BFS spanning tree. The nodes are named by the time step at which they were discovered by the BFS, i.e. the node that is discovered fourth has the name 4. To find a parent p of node i, we can induce $p = \text{rank}_0(B, \text{select}_1(B,i))$. To find the children of node i, we locate the i-th 0 via $\text{select}_0(B,i)$ and then simply iterate through the following bits until we hit another 0 entry [FP16].
With this technique, we are able to represent a tree with n nodes by a bit vector with $2 \cdot n + o(n)$ bits. The relations of the nodes, for example the parent of a node, can be calculated in $\mathcal{O}(1)$ time.

7.1.2 GLOUDS

Fischer and Peters [FP16] extend LOUDS to a data structure for storing arbitrary graphs. This data structure is called *graph level order unary degree sequence* (GLOUDS). It uses LOUDS to encode the spanning tree of a graph. This is enhanced to represent the non tree edges as well.

The following variables are used for GLOUDS:

- let G be a directed graph,
- n the number of vertices in G,
- m the number of edges in G,
- $c \leq n$ the number of connected components in G,
- $k = \max\{m + c - n, 0\}$ the number of edges in G that need to be added to the spanning tree of G to reconstruct G, i.e. the number of non-tree-edges, and
- $h \leq \min\{n, k\}$ the number of vertices in G that have more than one incoming edge.

At the start, we choose a vertex r as the start point of the breadth-first search. We call r the root or start point of its connected component (= all vertices that are reachable by r). If not all vertices are reachable by r, we add a super root sr and connect sr to each required root. We encode sr as 1^c0, where c is the number of connected components.
Through the BFS, we create and encode a spanning tree T of the graph with LOUDS (see Section 7.1.1). If a vertex v that is adjacent to the current vertex u is discovered, but not finished yet, v is added to T as a child of u. These discovered, but not finished, child vertices are named *shadow nodes* [FP16]. Since 0 or 1 encode nodes and edges of the spanning tree, we expand the possible inputs with 2 to encode shadow nodes. So, GLOUDS uses a trit vector B (Section 2.6) instead of a bit vector. During the BFS, B is filled like LOUDS with the adaption that shadow nodes are encoded by 2. An example is given in Section 7.1.3. The trit vector for G needs $n + m + c + 1$ trits. To support various operations on the encoded graph, we additionally construct an array H to store the shadow nodes in the order they were

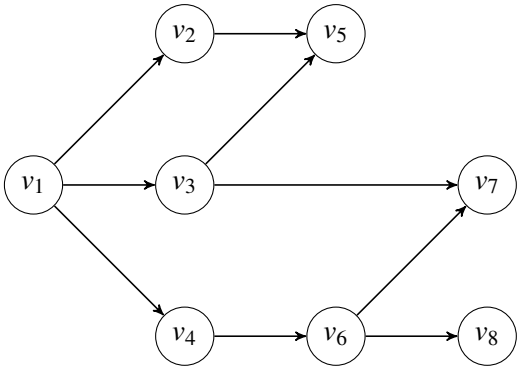


Figure 7.1: Example graph

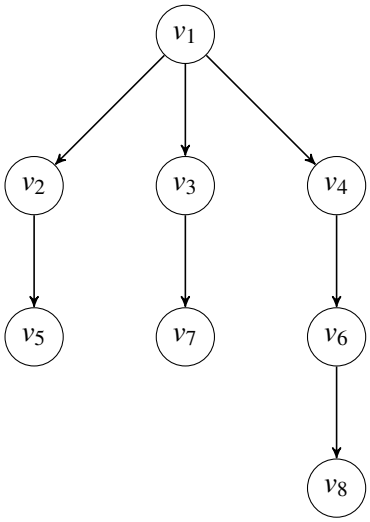


Figure 7.2: Spanning tree of the graph given in Figure 7.1

discovered. With rank and select queries on B and access to H we are able to reconstruct all information of the graph, for example neighbourhoods, in constant time.

7.1.3 Example Construction

Now we demonstrate via the graph G in Figure 7.1 how to represent a graph with GLOUDS. The spanning tree T of G is given through Figure 7.2 which is encoded by LOUDS. Figure 7.3 shows the procedure of the BFS over G which is starting at the super root sr. Shadow nodes, vertices that have been discovered in the BFS multiple times, are highlighted by dotted lines.

The LOUDS representation of the spanning tree in Figure 7.2 is the following for each level:

sr-level	10
v_1-level	1110
v_2, v_3, v_4-level	10 10 10
v_5, v_7, v_6-level	0 0 10
v_8-level	0

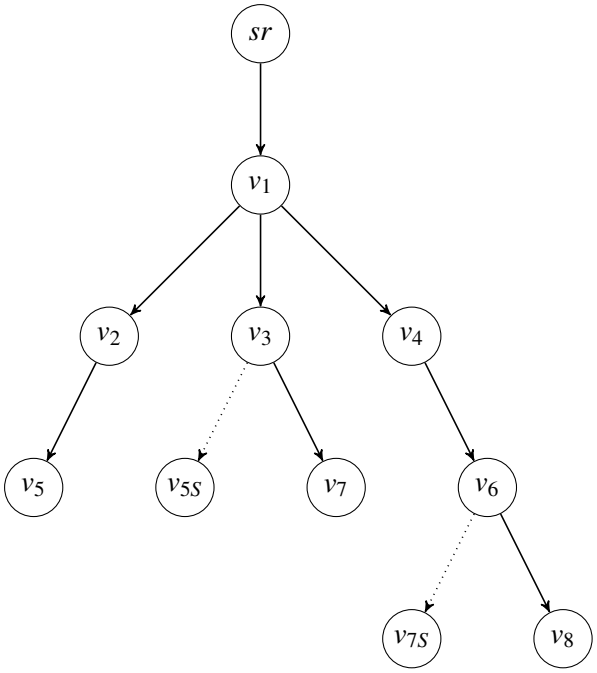


Figure 7.3: Visualisation of the BFS over the graph in Figure 7.1 starting at *sr*, the vertices v_5 and v_7 are listed twice, once as 'real' vertices and once as 'shadow' nodes

Which constructs the bit vector $B = 10\ 1110\ 10\ 10\ 10\ 0\ 0\ 10\ 0$.

The GLOUDS representation of the graph in Figure 7.1, whose construction is visualized in Figure 7.3, is the following:

sr-level	10
v_1-level	1110
v_2, v_3, v_4-level	10 210 10
v_5, v_7, v_6-level	0 0 210
v_8-level	0

Which constructs the trit vector $B' = 10\ 1110\ 10\ 210\ 10\ 0\ 0\ 210\ 0$. The array H contains the names of shadow nodes in order of their discovery during the BFS. For this example the resulting array is $H = \{v_5, v_7\}$.

7.1.4 Space Requirements

As mentioned in Section 7.1.2, n is the number of vertices, m the number of edges, k the number of shadow nodes and $c \leq n$ the number of connected components. By using a lookup table, Fischer and Peters [FP16] need $(n+m+c) \cdot (\log(3) + o(1))$ bits to store the trit vector. Supporting rank and select queries needs additional $o(n+m)$ bits. The support list of shadow nodes H can be stored as a wavelet tree (Section 2.8) with additional $k\log(n)$ bits plus $o(k\log n)$ bits to support select queries. Due to the fact that the amount h of different shadow nodes could be smaller than k, Fischer and Peters use a more

succinct way to store H with just $h\log(n)+k\log(h)+o(h+k\cdot\log(h))+\mathcal{O}(\log(\log(n)))$ bits (for details see [FP16]).
Summed up, the representation of Fischer and Peters requires $(2n+m)\cdot\log(3)+h\cdot\log(n)+k\cdot\log(h)+o(m+k\cdot\log(h))+\mathcal{O}(\log(\log(n)))$ bits.

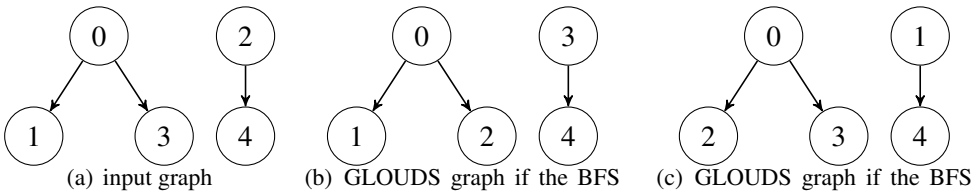
7.2 Implementation

GLOUDS works for any directed, unweighted graph, even though it is optimised for tree-like graphs. The given graph does not have to be connected. We allow loops and isolated vertices.
Our implementation creates a GLOUDS representation of a graph which is isomorphic to the input graph. This is because the representation names of the vertices according to the order they were discovered during the BFS. Our implementation has a boolean template parameter that allows to choose whether the names of the input graph (`glouds<true>`) or the isomorphic GLOUDS graph (`glouds<false>`) should be used for input and output. A detailed explanation for this and our solution is given in Section 7.2.3.
Fischer and Peters [FP16] start the BFS at the super root. The super root refers to the optimal start points, or roots, of the connected components (see Section 7.1.2). The optimal set of start points is defined as a minimum set of vertices from which each vertex of the graph can be reached. Since we allow an arbitrary naming of the vertices the optimal start points cannot be given by the input. This implies a pre-calculation of this minimal set of start points. We implement two versions of GLOUDS:

- `glouds_startpoint` calculates the optimal start points by a depth-first search (DFS, see Section 2.3.2) before running the BFS. This takes $\mathcal{O}(m+n)$ additional time for n vertices and m edges. This solution is presented in Section 7.2.2.

- `glouds` starts the BFS at the vertex named 0 and always chooses the next (minimal name number) undiscovered vertex as root for the next connected component. This reduces time, but generates a possible non-minimal set of start points. Additionally it is necessary to store the names of the start points. For more details see Section 7.2.1.

The different versions produce a different isomorphism to the input graph. The differences are shown in Figure 7.4.
All of our GLOUDS implementations have the following common data structures: Our solution for the trit vector is an adaption of a wavelet tree, instead of the lookup table used in Fischer and Peters [FP16] (for more details see Section 4.3). We store the list of shadow nodes as integer vector of size m and width $\log n$ (compare to Section 4.1).
Both versions require a graph which can be represented by its adjacency matrix (any binary vector (bit or boolean), compare to Section 7.3.2) or adjacency list (`std::vector<std::vector<uint64_t>>`, compare to Section 7.3.1). The constructor methods of both GLOUDS versions are using the adjacency list because of its faster neighbourhood queries. The matrix input will be converted to a list. Converting methods in quadratic time are offered by `utils_for_adjacency`.



(a) input graph

(b) GLOUDS graph if the BFS starts and restarts always at the smallest undiscoverd vertex

(c) GLOUDS graph if the BFS starts at the super root which is connected to the optimal start points

Figure 7.4: Example to show the different BFS order of the different GLOUDS versions and the corresponding naming of the vertices

Each of our graph implementations store the number of vertices and the number of edges in `uint64_t` member variables. So, `number_of_nodes()` and `number_of_edges()` are answered in $\mathcal{O}(1)$ time. The values are calculated while constructing the representation: the number of vertices can be calculated in $\mathcal{O}(1)$ time. The number of edges must be calculated in $\mathcal{O}(n)$ time for an adjacency list and $\mathcal{O}(n^2)$ for an adjacency matrix (explanations given in Sections 7.3.1 and 7.3.2). Methods for these calculations are offered by `utils_for_adjacency`.

7.2.1 GLOUDS Implementation with Non-Optimal BFS Start Points

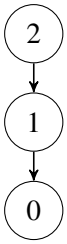
Our first version of GLOUDS starts the BFS directly at any start point (we choose the vertex named 0). During the BFS, the start point of the next connected component is the undiscovered vertex with a minimal name number. Due to this procedure, the set of start points could be not minimal.
To implement `get`, we need the vertex names of the start points. We store these in a `std::vector<uint64_t>`. The super root has no function in this version, so it does not exist here.
The GLOUDS representation `glouds` needs:

- a `trit_vector` of length $n+m$
- an `int_vector` for the shadow nodes with the size m and width $\max\{\log\lceil n-1\rceil, 1\}$, and
- a `std::vector<uint64_t>` with length c' for the start points,

where n is the number of vertices, m the number of edges and $c' \leq n$ is the non-minimal number of start points corresponding to the BFS. It holds $k = \max\{m+c'-n, 0\}$ for the number of shadow nodes, but in this version the start points arise while running the BFS. Therefore, c' is not known at the initialisation of the shadow node list. We choose the size m because for the worst case each edge might generate a shadow node.

Construction

The constructor builds the trit vector and the list of shadow nodes like GLOUDS (see Section 7.1). In addition, the list of start points must be constructed.



optimal start points: $\{2\}$, BFS order is $2,1,0$
start points by BFS: $\{0,1,2\}$, BFS order is $0,1,2$

Figure 7.5: Worst-case for GLOUDS without pre-calculation the start points

To reduce construction time, the trit vector constructor receives the already calculated levels of the wavelet tree, named level 0 and 1 (compare to Section 4.3). While traversing the given graph, both levels must be calculated. First, we create two basic bit vectors (see Section 4.2.1) for each level of the trit vector of the GLOUDS encoding (without super root). These will be filled while running a BFS over the graph. The length of level 0 is $n+m$ for the input graph with n vertices and m edges. This occurs, because we list each vertex and its corresponding neighbours. The length of level 1 is m since we need to clarify if a neighbour from level 0 is encoded as a shadow node or regular vertex.
Now we execute the BFS. It starts at vertex 0 and traverses all vertices that are reachable from 0. After this, we start at the next (smallest number) undiscovered vertex for the next connected component. The first time we visit a vertex (vertex becomes discovered), we start to enlist all neighbours for this vertex in the trit vector. This means we iterate over its neighbours and write 1 for undiscovered neighbours and 2 for discovered (shadow nodes). If a vertex v is finished, we write 0. We call the block in the trit vector between two zeros *neighbour information* of the corresponding vertex v (whose zero completes the block), because this block describes the outgoing edges from v. For example, the encoding of the graph in Figure 7.3 is $B' = 10\ 1110\ 10\ 210\ 10\ 0\ 0\ 210\ 0$. The neighbour information of vertex v_3 is 21 which starts behind the third zero and ends at the fourth.
Figure 7.5 depicts an example graph showing why this order of the BFS is not optimal: The BFS starts at vertex 0 and the connected component of that vertex is completed. Then it advances to the next smallest undiscovered vertex, and all other vertices of the connected component of 1 (just 0) are already finished. For 2 as a next start point, it is the same, so the BFS counts three instead of one connected components. This means, for the worst case, the list of start points receives $n-1$ additional unnecessary entries.
The whole representation is constructed while running the BFS. Therefore, the construction time is $\mathcal{O}(n+m)$ where n is the number of vertices and m is the number of edges.
The construction needs additional space for a `std::queue<uint64_t>` of length $\mathcal{O}(n)$ and a basic `bit_vector` with length n to mark if a vertex is discovered (we do not need to know if a vertex is finished in our use-case). Since we store shadow nodes by their GLOUDS name in the shadow node list, we need to store the local isomorphism, no matter if the class template parameter is `true` or `false` (with mapping or not). If the isomorphism is just needed locally (`false`) we choose a `int_vector` with size n and width $\max\{\log\lceil n-1\rceil, 1\}$. Otherwise, we need a `std::vector<uint64_t>` because our permutation implementations require this input (see also Sections 4.4 and 7.2.3).

Graph Operations

With the trit vector we are able to reconstruct all information about the neighbourhoods of the vertices. For this we usually have to calculate the neighbour information section of vertex v. This section ends at the v-th zero and starts at the $v-1$-th zero. The starting and ending position of the neighbour information can be calculated by combining select queries on the trit vector.
The $\texttt{degree}(v)$ of a vertex $v \in V$ is the length of its neighbour information section, so it can be calculated by two select queries in constant time.
For $\texttt{get}(v,i)$, we have to distinguish between tree and shadow nodes, because for tree nodes we need to recalculate the name number of the vertex while shadow nodes can be read out from the list of shadow nodes. If the i-th neighbour information of v is 1, we need to recalculate the name number of a tree node. This number depends on the order the BFS visits the vertices. This is the only reason why we need to save the start points. Via a binary search over the start points we calculate the number of the start point which v belongs to. A binary search is directly possible because the list of start points always contains ascending values. Therefore the case that the i-th neighbour information is a 1 takes $\mathcal{O}(\log c') = \mathcal{O}(\log n)$ time for n vertices and $c' \leq n$ start points. For example in the graph G of Figure 7.3 with the encoding $B = 1110\ 10\ 210\ 10\ 0\ 0\ 210\ 0$: $\texttt{get}(2,1)$ searches the second entry in the neighbour information of vertex v_3, called vertex 2 by the GLOUDS naming, i.e. $B[8] = 1$. The start point list of G just contains vertex v_1, i.e. vertex 0. So, the name of the second neighbour of v_3 is vertex $0 + \text{rank}_1(B,8) - 1 = 0 + 5 - 1 = 4$, i.e. vertex v_5. If the i-th neighbour information of v is 2, the neighbour is a shadow node and we look in the list of shadow nodes H. To get the right index in H, we need one rank query, so the second case is solved in constant time. Again for G of Figure 7.3: $\texttt{get}(5,0)$ searches for the first entry in the neighbour information of vertex v_6, called vertex 5 by the GLOUDS naming, i.e. $B[14] = 2$. The list of shadow nodes is $H = \{v_5, v_7\} = \{4,6\}$. So, $\texttt{get}(5,0)$ returns $H[\text{rank}_2(B,14) - 1] = H[2] = 6$, i.e. vertex v_7.
The operation $\texttt{is\_neighbour}(u,v)$ iterates over the neighbour information section of v and calls $\texttt{get}(u,i)$ for all neighbours of u ($i = \{0,1,\ldots,\deg(u)-1\}$) to test if any neighbour of u is equivalent to v. Neighbour queries can be answered with $\mathcal{O}(\deg(u) \cdot f)$ time, where $\deg(u) \in \mathcal{O}(n)$ and f is the runtime for $\texttt{get}$. The worst case of a neighbour query is $\mathcal{O}(n \log n)$ time.

7.2.2 GLOUDS Implementation with Pre-Calculating BFS Start Points

In this version we calculate the optimal start points before running the BFS. This requires an additional graph traversal, for which we choose a DFS.
The GLOUDS representation $\texttt{glouds\_startpoint}$ needs:

- a $\texttt{trit\_vector}$ of length $n + m + c + 1$, and
- an $\texttt{int\_vector}$ for the shadow nodes with size k and width $\max\{\log\lceil n-1\rceil, 1\}$,

where n is the number of vertices, m the number of edges, $k = \max\{m + c - n, 0\}$ the number of shadow nodes and $c \leq n$ is the minimal number of start points.

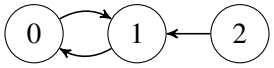


Figure 7.6: Example graph to explain the necessity of storing all possible start points

Construction

First we run a DFS to calculate the optimal start points. Our approach is to choose an arbitrary vertex r as a start point and mark every vertex which can be reached by r. Then we choose any next undiscovered vertex r'. If r is reachable by r', all vertices reachable by r are reachable by r'. The connected component of r is then a subgraph of the connected component of r'. The vertex r is not a start point any longer. The DFS maintains a static integer vector SP with $SP[i] = n$ iff vertex i is undiscovered (because no vertex has name n). Otherwise, $SP[i]$ is the start point of the connected component which contains i. If a connected component has multiple possible start points we have to store all these possibilities during the DFS. We illustrate this in Figure 7.6: we start at vertex 0 and define 0 as a start point of the first connected component, $SP[0] = 0$. We discover vertex 1, reachable by 0, $SP[1] = 0$. Then we need to restart the DFS at vertex 2 as a start point of the next connected component, $SP[2] = 2$. From 2 we just reach the already discovered vertex 1 and terminate the recursive search. To detect if the current connected component includes another previously discovered component, we either have to check over all reachable vertices (that would be too much time) or mark all potential start points of a component. The reason why there are several possible start points in our example is that the edges between the vertices 0 and 1 make a circle. Therefore, we store a static bit vector R with $R[i] = 1$ iff vertex i is on a circle which includes the start point $SP[i]$. To maintain R, we hand over the start point of the current connected component at each recursive step to the DFS. After the recursive DFS step, we update the entry of the actual vertex v: if $R[u] = 1$ for a neighbour u of v follows $R[v] = 1$. If we detect that a connected component C with start point s_C contains another component C' by reaching any possible start point $s_{C'}$ of C', we set $SP[SP[s_{C'}]] = s_C$ and $R[s_{C'}] = 0$. As a consequence all other possible start points of C' are unmarked as start point. The vertex $SP[SP[s_{C'}]]$ is the first found start point of C' and thus all other possible start points, that have been found later, refer to $SP[SP[s_{C'}]]$, which is updated now. At the end of the DFS, the vertices i with $SP[i] = i$ yield the minimal start point set. Therefore, we just need to iterate over SP one time and add these vertices to the optimal list of start points. The generated start points list is used as the neighbour list of the super root sr (add sr to the input adjacency list).
Now we start the BFS at sr. All vertices are reachable by the super root because sr is adjacent to every start point. Filling the trit vector levels and shadow node list works the same way as the BFS without pre-calculating the start points.
The construction time is $\mathcal{O}(n+m)$ where n is the number of vertices and m edges.
The BFS needs the same additional local space as the BFS of our implementation without pre-calculating the start points. The `int_vector` for SP, which is used by the DFS to store the start point of each vertex, has size n and width $\max\{\log\lceil n-1\rceil, 1\}$. The `bit_vector` for R to mark if a vertex is a possible start point has length n. We use a `std::vector<uint64_t>` with size $\mathcal{O}(n)$ to store the optimal start point list.

Graph Operations

The operations `degree`(v) and `is_neighbour`(u,v) have not changed. The big advantage of this version is that the changed BFS order (compare to Figure 7.4) results in the following equivalence: the i-th one in the trit vector encodes the first discovery of vertex $i-1$. So, the name of a vertex in the neighbour information section of another vertex can be reconstructed by one rank query. Therefore, we are able to implement the `get`(v,i) operation in constant time and do not need to store the names of the start points in an additional integer vector.

7.2.3 Isomorphism

The GLOUDS representation is isomorphic to the input graph. This is due to the fact that for GLOUDS the vertices are named by their number as they are processed by the BFS. In Figure 7.4 is an example given where the input and constructed graphs are not identical. The isomorphism to reconstruct the names of the original graph from the GLOUDS graph with non-optimal start points is $\{0 \mapsto 0, 1 \mapsto 1, 2 \mapsto 3, 3 \mapsto 2, 4 \mapsto 4\}$.
We have two options to deal with this problem, these options can be chosen by a boolean class template parameter:

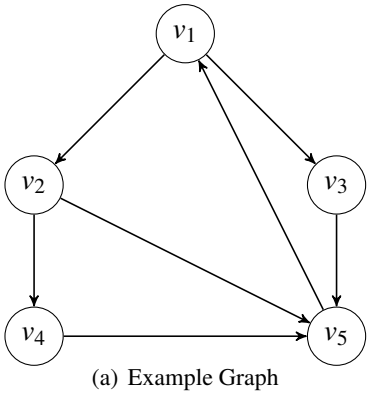
- We allow an arbitrary naming of the graph. For get or neighbour queries we have to use the GLOUDS names of the vertices. Alternatively, the input and output vertices are renamed before or respectively after, outside of GLOUDS. Functions to rename the vertices for all graph implementations are offered by the class `plads::graph::mapping`. This option can be chosen by using `false` as a template parameter.

- Otherwise, the GLOUDS implementation itself renames the input and output. Therefore, the isomorphism must be stored. This option can be chosen by using `true`.

The isomorphism is a permutation of the integers in $[0,n]$ (see Section 4.4). We choose the succinct permutation `permutation_inverse` by Navarro [Nav16] with parameter $b = 16$.
The operation `degree`(v) needs one read query on the permutation to get the real input name. For `get`(v,i) the input and output vertex have to be converted. The `is_neighbour`(u,v) operation calls `get` and needs one additional reading on the permutation. Every read and inverse operation on the permutation needs $\mathcal{O}(1)$ additional time.

7.3 Other Representations for Graphs

In the following section, we introduce the *adjacency list* and *adjacency matrix* of graphs which are used as a benchmark for the comparison.



(a) Example Graph

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

(b) Matrix representation

Figure 7.7: Example of the adjacency matrix representation of a graph

7.3.1 Adjacency List

The *adjacency list* of a graph is a vector of integer vectors where all neighbours of each vertex are listed. So, $\text{adj}_l[i]$ returns a list of the (outgoing) neighbours of vertex i where $\text{adj}_l[i][j]$ is the j-th neighbour of i.

We implement the adjacency list as `std::vector<std::vector<uint64_t>>` of length n for n vertices. The inner vector $\text{adj}_l[i]$ has length $\deg(i) \in \mathcal{O}(n)$.
The number of vertices is the length of the outer vector. The number of edges is the sum of the length of all inner vectors. A method of `utils_for_adjacency` calculates this sum in $\mathcal{O}(n)$ time. The number of vertices and the number of edges are stored as `uint64_t` member variables. The degree of a vertex v is equal to the length of the corresponding inner vector. I.e. $\deg(v) = |\text{adj}_l[v]|$ can be calculated in constant time. Neighbour queries are answered by an iteration over all neighbours (inner vector) of the given vertex. The operation `is_neighbour` takes $\mathcal{O}(\deg(u)) \in \mathcal{O}(n)$ time. For `get`(v,i) we return $\text{adj}_l[i][j]$ in constant time. This operation is the benefit of this representation.

7.3.2 Adjacency Matrix

For the *adjacency matrix* representation, the basic idea is that all outgoing edges from a vertex are listed in a binary matrix. So, to rebuild the graph $G = (V,E)$ out of the matrix, we have the equivalence

$$\text{adj}_m[i][j] = 1 \iff (i,j) \in E,$$

i.e. there is an edge from the i-th vertex in the list to the j-th, where i is the row and j the column of the matrix with $i, j \in [0,n)$ for n vertices. See Figure 7.7 for an example.

In our implementation the matrix is stored as static bit vector (see Section 2.5.1). Its length is n^2 bits where n is the number of vertices. A `std::vector<bool>` can also be used as an input and will be converted.

The equation to rebuild the graph is

$$\text{adj}_m[i \cdot n + j] = 1 \iff (i, j) \in E.$$

The number of vertices is the length of the adjacency matrix, i.e. $\sqrt{|\text{adj}_m|}$. To calculate the number of edges we count all 1 entries in the whole matrix. This is offered by `utils_for_adjacency` and costs $\mathcal{O}(n^2)$ time. The number of vertices and the number of edges are stored as `uint64_t` member variables. The operation `degree`(v) is solved in linear time because $\deg(v) = \sum_{i=0}^{n-1} = \text{adj}_m[v \cdot n + i]$ holds. For `get`(v, i) we iterate over the row of outgoing edges of v and stop at the i-th one. The reached index corresponds to the i-th neighbour. So, `get`(v, i) needs $\mathcal{O}(n)$ time. The adjacency matrix is very efficient for neighbourhood queries because it offers direct access to every outgoing edge. This gives us `is_neighbour`$(u, v) =$ `true` iff $\text{adj}[u \cdot n + v] = 1$ in constant time.

7.4 Results and Further Work

In this section, we compare our implementations of the different graph representations. Table 7.1 summarizes the space needed for the representations. The number of edges in a directed graph is $0 \leq m \leq n^2$, but for a k-tree-like graph it is $m = n - 1 + k$ for a small number $k \in \mathbb{N}_0$. Table 7.2 gives an overview of the space and time requirement of the constructors of the different representations. A summary of the running times of the three operations is given in Table 7.3. Here we see that our solution with precalculating the optimal start points is asymptotically always at least as good as the other representations except the adjacency matrix for neighbourhood queries.

7.4.1 Benchmark Results

In this section, we present the results of our benchmarking and discuss these in the context of the theoretically determined time and space requirements. For our benchmarks, we construct graphs of different sizes with a given parameter k. Hereby, k denotes a factor with which the total amount of edges that will be added to any given graph in relation to the number of its vertices is calculated. The amount of edges m is described through $m = n \cdot \left(1 + \frac{k}{100}\right)$.
For each edge that has to be added, we generate two random numbers, that correspond to a start vertex and a end vertex which are not yet connected. Since we benchmark only undirected graphs , we also add the corresponding edge from the end vertex to the start vertex. Note that with an increasing k we move further away from a tree-like graph. However our k is not the same as the *k-tree-like graphs* in [FP16] since our k only is used for calculating the amount of edges that we add. While this approximates to a tree-like structure, we do not guarantee that all vertices will be connected to a single graph (isolated vertices might still occur, but less frequently with increasing k). The graph is then represented by its adjacency list and used as an input for the different representations we implement. We compare the following representations:

- graph-list for the adjacency list (see Section 7.3.1),

representation	space in bits
GLOUDS by [FP16]	$(n+m+c)\cdot(\log(3)+o(1))$ (trit vector) $h\log(n)+k\log(h)+o(h+k\log(h))+\mathcal{O}(\log(\log(n)))$ (shadow node list) $=\mathcal{O}(n)$
GLOUDS with non-opt. start points	$(m+n)+m+o(2m+n)$ (trit vector) $m\cdot\log n$ (shadow node list) $\mathcal{O}(c')$ (start point list for $c'\geq c$ start points) $=\mathcal{O}(n\log n)$
GLOUDS with opt. start points	$(m+n+c+1)+(m+c)+o(2m+n+2c+1)$ (trit vector) $k\cdot\log n$ (shadow node list) $=\mathcal{O}(n)$
adjacency list	$\mathcal{O}(n)$
adjacency matrix	$\mathcal{O}(n^2)$

Table 7.1: Theoretical worst case comparison of space requirement of the discussed graph representations for graphs with n vertices, m edges and $c\in\mathcal{O}(n)$ connected components. The summing up (in $\mathcal{O}$-notation) is for k-tree-likeness with a small constant k from which $m\in\mathcal{O}(n)$ follows (for details see Sections 7.1 to 7.3). If GLOUDS stores the isomorphism $\mathcal{O}(n\log n)$ additional bits are needed to store a object of `permutation_inverse<16>` (compare to Section 4.4).

representation	construction time	additional construction space in bits
GLOUDS with non-opt. start points	$\mathcal{O}(m+n)$ (BFS)	$\mathcal{O}(n)$ (queue) $\mathcal{O}(n)$ (discovered)
GLOUDS with opt. start points	$\mathcal{O}((m+n)\cdot n)$ (DFS) $\mathcal{O}(m+n)$ (BFS)	$\mathcal{O}(n)$ (start point of each vertex) $\mathcal{O}(n)$ (identify possible start points) $\mathcal{O}(n)$ (super root neighbours) $\mathcal{O}(n)$ (queue) $\mathcal{O}(n)$ (discovered)
adjacency list	$\mathcal{O}(1)$ (set reference)	–
adjacency matrix	$\mathcal{O}(n^2)$ (convert)	–

Table 7.2: Theoretical worst case comparison of construction time and space of the discussed graph representations for graphs with n vertices and m edges if the input graph is represented as adjacency list (for details see Sections 7.1 to 7.3). If GLOUDS stores the isomorphism $\mathcal{O}(n)$ additional construction time is needed to construct the succinct permutation (see Section 4.4). Otherwise, additional $\mathcal{O}(n)$ bits of construction space is needed to store the isomorphism locally.

representation	degree	get(v,i)	is neighbour(u,v)
GLOUDS with non-opt. start points	$\mathcal{O}(1)$	$\mathcal{O}(\log c')$ $= \mathcal{O}(\log n)$	$\mathcal{O}(\deg(u) \log c')$ $= \mathcal{O}(n \log n)$
GLOUDS with opt. start points	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\deg(u))$ $= \mathcal{O}(n)$
adjacency list	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\deg(u))$ $= \mathcal{O}(n)$
adjacency matrix	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$

Table 7.3: Theoretical worst case comparison of running time of the discussed graph representations for graphs with n vertices, m edges, $c \in \mathcal{O}(n)$ optimal and $c' \geq c$ non-optimal start points (for details see Sections 7.1 to 7.3). If GLOUDS maps the isomorphic vertex names itself $\mathcal{O}(1)$ additional time for reading and inverting is needed (see Section 4.4).

- graph-matrix for the adjacency matrix (see Section 7.3.2),
- glouds-true for the GLOUDS representation without pre-calculating the optimal start points and with storing of the isomorphism (see Sections 7.2.1 and 7.2.3),
- glouds-false for the GLOUDS representation without pre-calculating the optimal start points and without storing of the isomorphism (see Sections 7.2.1 and 7.2.3),
- glouds-startpoints-true for the GLOUDS representation with pre-calculating the optimal start points and with storing of the isomorphism (see Sections 7.2.2 and 7.2.3), and
- glouds-startpoints-false for the GLOUDS representation with pre-calculating the optimal start points and without storing of the isomorphism (see Sections 7.2.2 and 7.2.3).

The size of our graphs n is a power of two, starting with 2^{15} going up to 2^{28} vertices.
We measure a total of three steps for a given factor of $k = 1$, $k = 30$ and $k = 100$. First we evaluate the time it takes to finish a constructor call with a specific amount of vertices in the graph. Then we investigate the space in bytes to save one object. After that, we show a Pareto front to show which implementation yields the best results for 2^{18} vertices. Then later on, we measure the time it takes to call the various operations one million times on each input graph. This includes get, degree and is_neighbour operations. For each method, we construct a query input beforehand, which is then used to execute the function call one million times.
We noticed that adding edges to the corresponding graph did not influence the construction time of graph-matrix and graph-list. Since we merely copy the input for graph-list, the time needed stems from said copy operation. As far as graph-matrix is concerned, the given adjacency list as input needs to be converted to a adjacency matrix. This results in quadratic construction time. Therefore, after benchmarking all classes, we decided to exclude graph-matrix results for graphs with more vertices than 2^{19} since we could see that it lacks behind on both required space and calculation time.

The construction times are sown in Figures 7.8 to 7.10. Although graph-matrix and graph-list do not require any specific calculation, it appears that the size of the input object seems to be the main factor of how fast these classes can construct one object. Since graph-matrix performed slow, we decided not to list any results bigger than 2^{19} for this class. However, graph-list outperforms any GLOUDS implementation because the construction is just copying the reference of the input adjacency list. As far as our various GLOUDS implementations are concerned, we can see that if we add the construction of data structure to store the isomorphism (here `permutation_inverse<16>`, see Sections 4.4 and 7.2.3) it naturally adds more time to the constructor calculation by a linear factor (see also Figure 4.3). In addition to that, we see that our DFS to locate the best start points adds significantly to the runtime. While the runtime of the GLOUDS constructors increases with higher k values, the respective increments are linear. This incremental effect however does not appear for graph-matrix and graph-list.
For the space usage, we can deduce that the required space grows accordingly to the increase of vertices (Figures 7.11 to 7.13). We can see that all GLOUDS implementations outperform graph-matrix and graph-list. Furthermore, as far as the GLOUDS implementations are concerned, we can see that performing the DFS to calculate the optimal start points yields a reduction of space usage for the object later on. This is due the fact that the amount of shadow nodes is kept to a minimum, and we hence do not encode these vertices more than required. In addition, pre-calculating the start points saves the space of a list of start points, which is needed to realise the get function in glouds-true/false. This space reduction superiors the space reduction of glouds-true/false by leaving out the super root. This improvement however diminishes if we add more edges to the input graph. We explain this as follows: the more edges a graph contains, the more likely it is that there are fewer connected components. Therefore for a high parameter k, the advantage of glouds-startpoint-true/false, that we do not need to store a list of start points, might have less influence, because the number of non-minimal start point becomes smaller for a more connected graph. Figure 4.2 shows the comparison of a naive implementation and a succinct implementation of permutation which is used to store the isomorphism. We can see that if we decide to store the isomorphism, we automatically add a specific amount of space to every GLOUDS version. Although we use an improved version of permutation, it still takes some space, which results in the difference space requirement between glouds-false and glouds-true, and respectively glouds-startpoint-false and glouds-startpoint-true.

While graph-list shows a constant runtime for get queries, graph-list requires constant time, while the GLOUDS implementations require almost constant runtime (Figures 7.14 to 7.16). Increasing the amount of edges has next to no impact on the runtime of get queries for all classes. However, if we store the isomorphism, the runtime for get increases significantly because we need to include the mapping process as well. For the additional running times arising from the additional mapping of the input and later the output see Figures 4.4 and 4.5.
The degree function behaves almost similar to the get function in terms of runtime. As seen through Figures 7.17 to 7.19 the increasing of k has no significant impact on the running times themselves. Overall

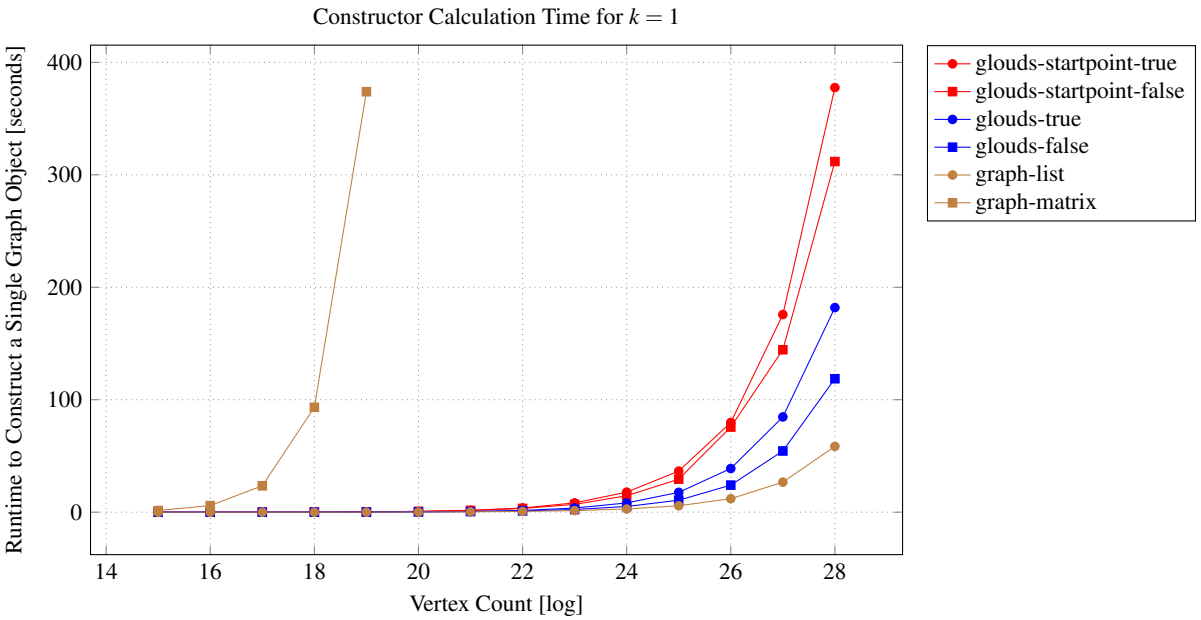


Figure 7.8: Construction time (in seconds) required to construct a single object for the various listed implementations with $k = 1$.

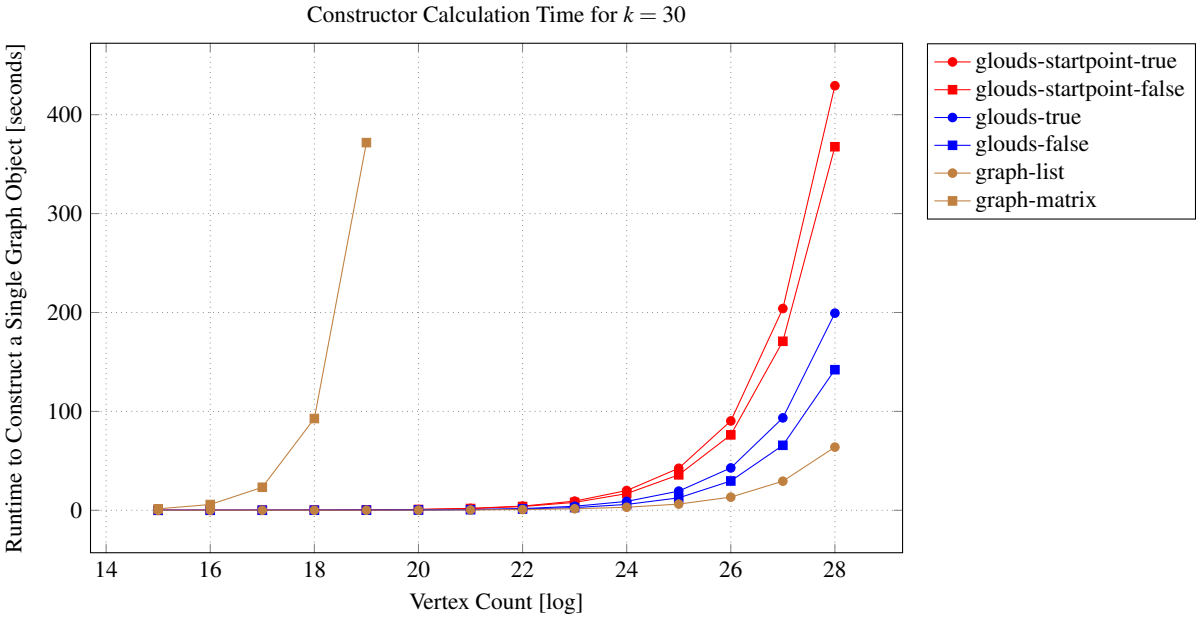


Figure 7.9: Construction time (in seconds) required to construct a single object for the various listed implementations with $k = 30$.

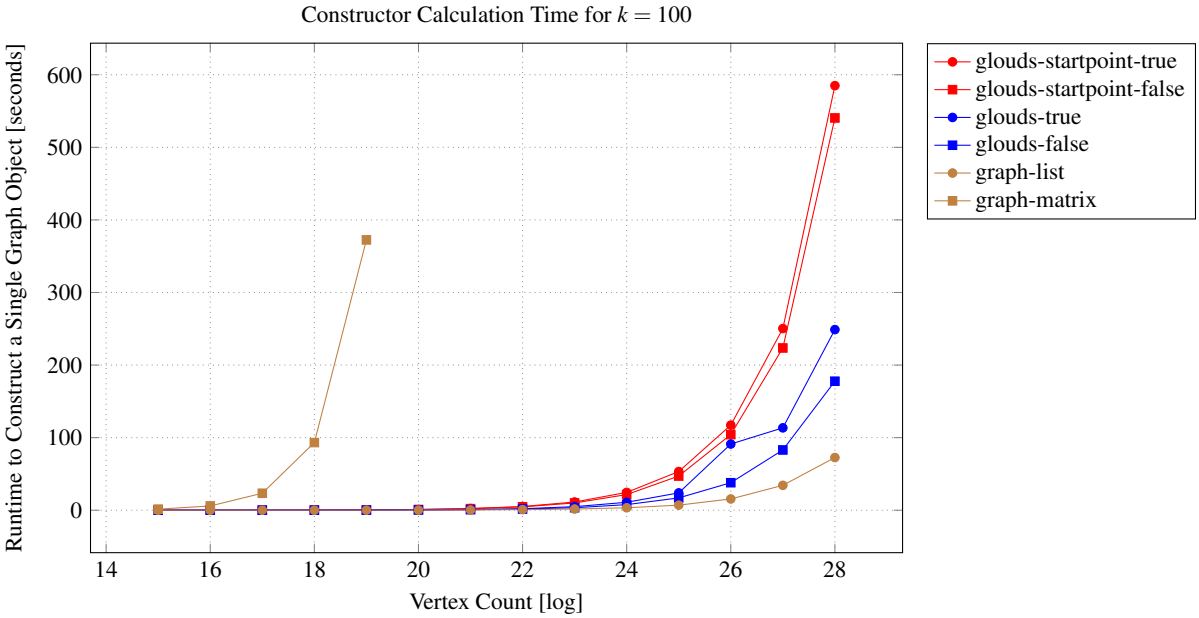


Figure 7.10: Construction time (in seconds) required to construct a single object for the various listed implementations with $k = 100$.

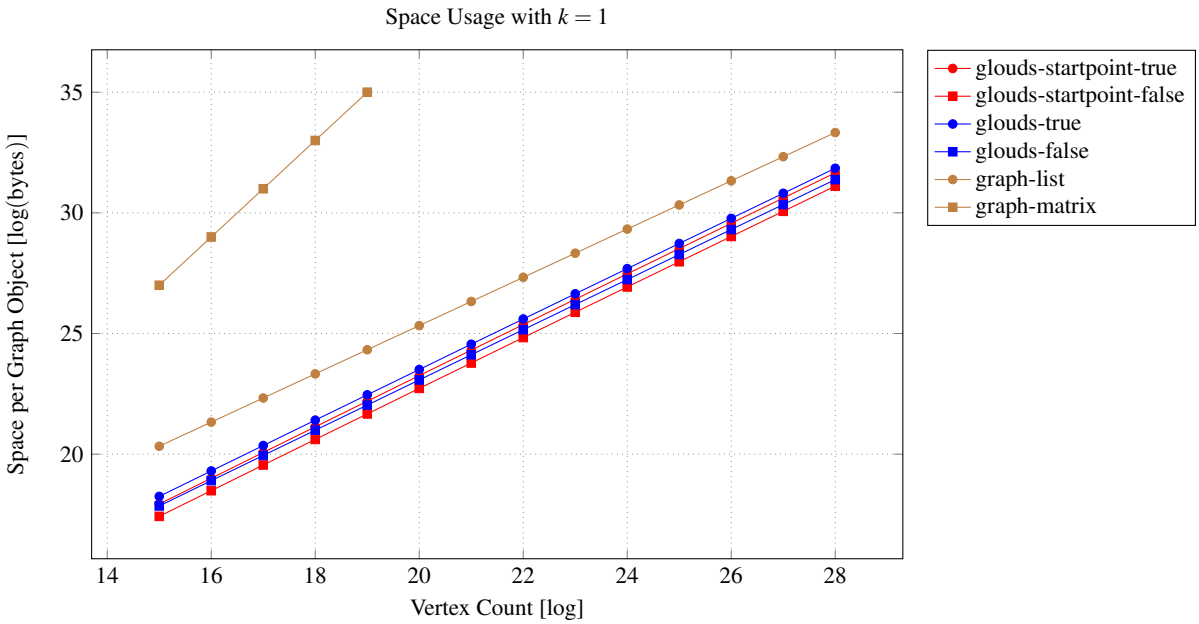


Figure 7.11: Amount of space (in bytes) it took to store each object with $k = 1$.

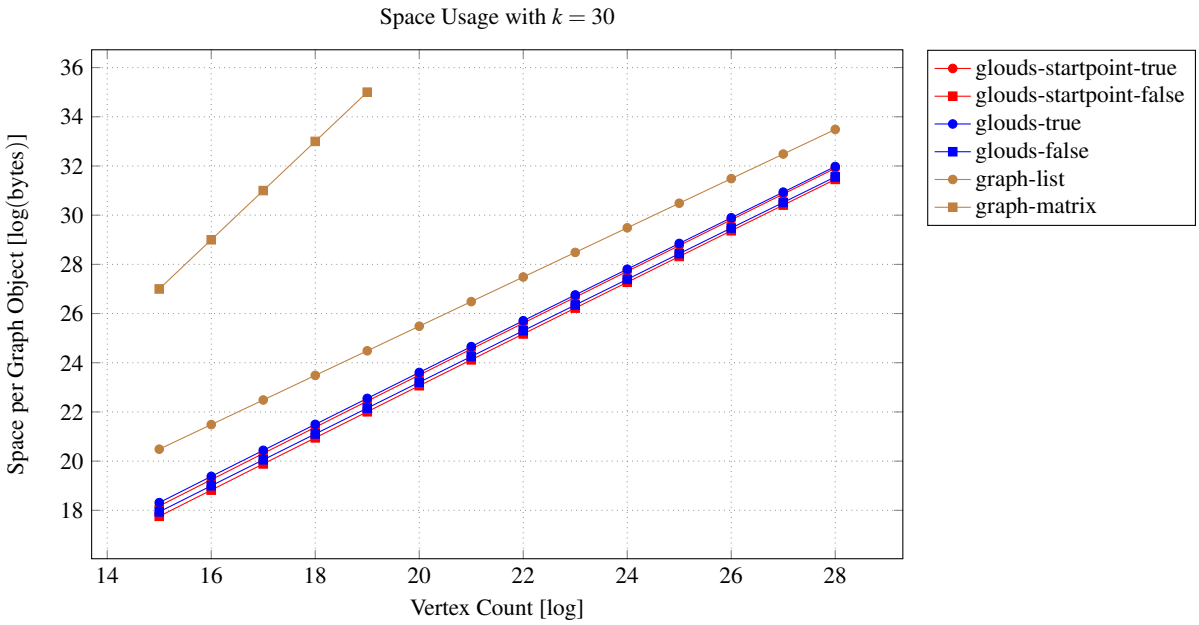


Figure 7.12: Amount of space (in bytes) it took to store each object with $k = 30$.

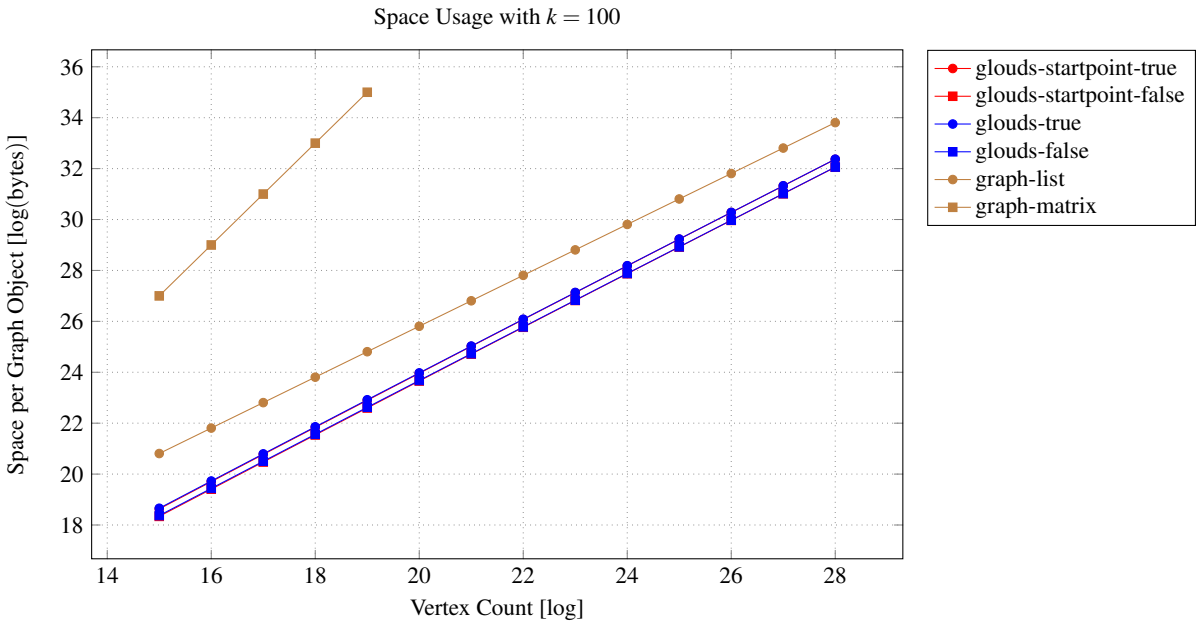


Figure 7.13: Amount of space (in bytes) it took to store each object with $k = 100$.

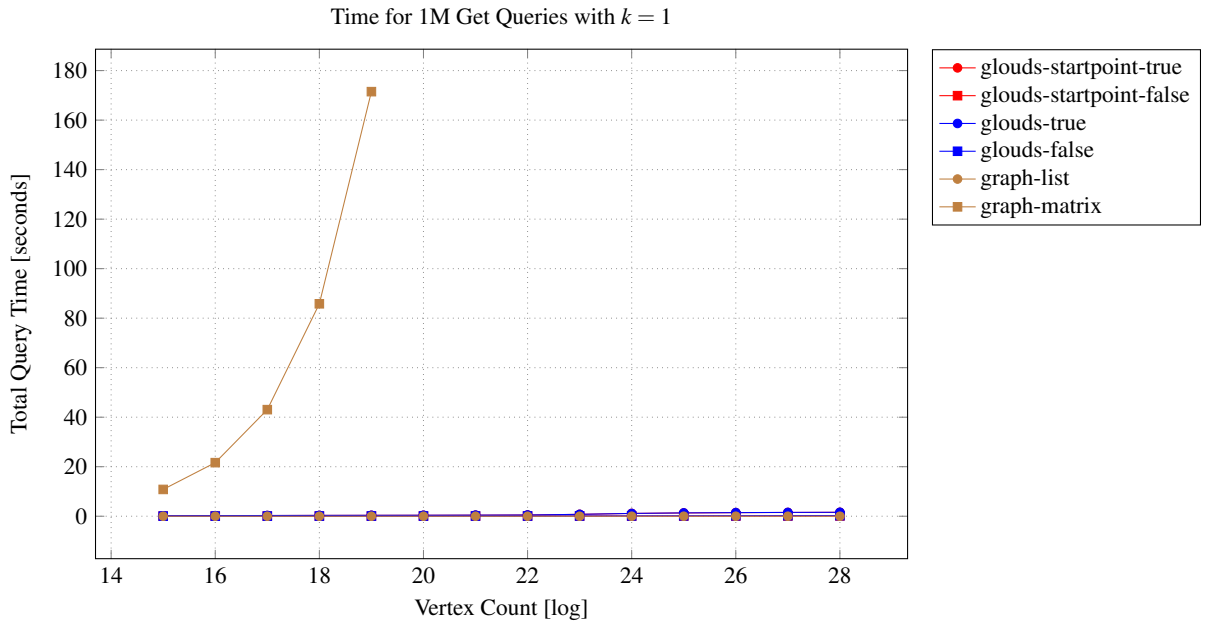


Figure 7.14: Runtime (in seconds) required to execute one million get queries on the respective objects for $k = 1$. Figures 7.15 and 7.16 exclude graph-matrix, because this figure demonstrates how poorly graph-matrix performs in comparison to the other methods.

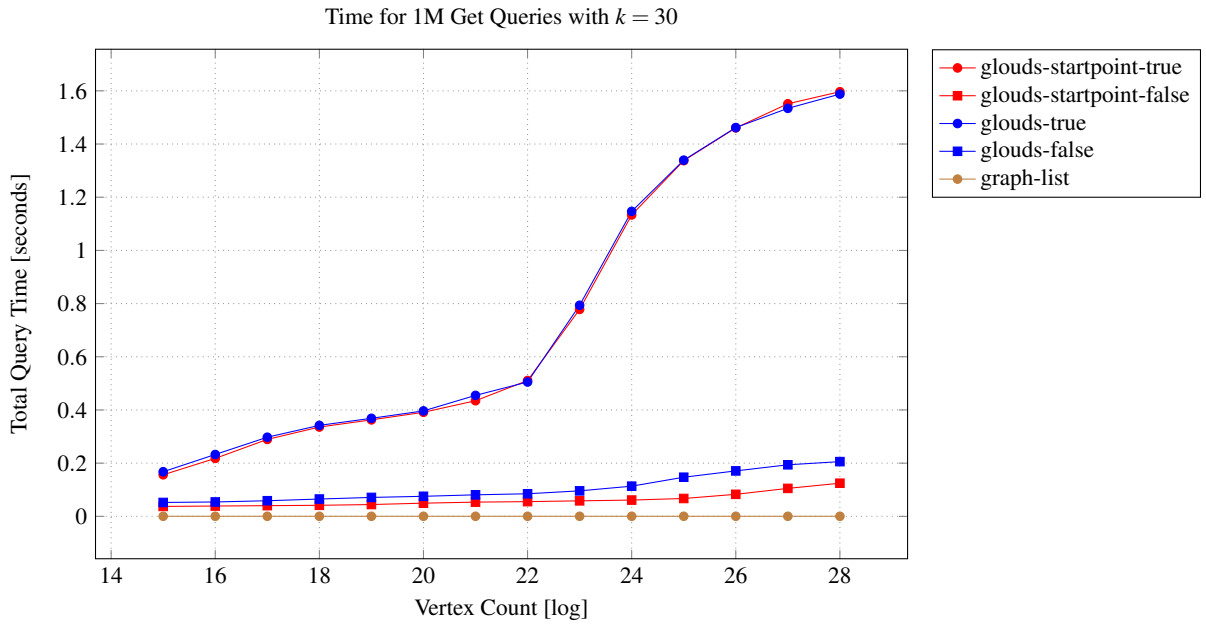


Figure 7.15: Runtime (in seconds) required to execute one million get queries on the respective objects for $k = 30$. We excluded graph-matrix to take a closer look on how the GLOUDS implementations perform in comparison to graph-list (for graph-matrix see Figure 7.14).

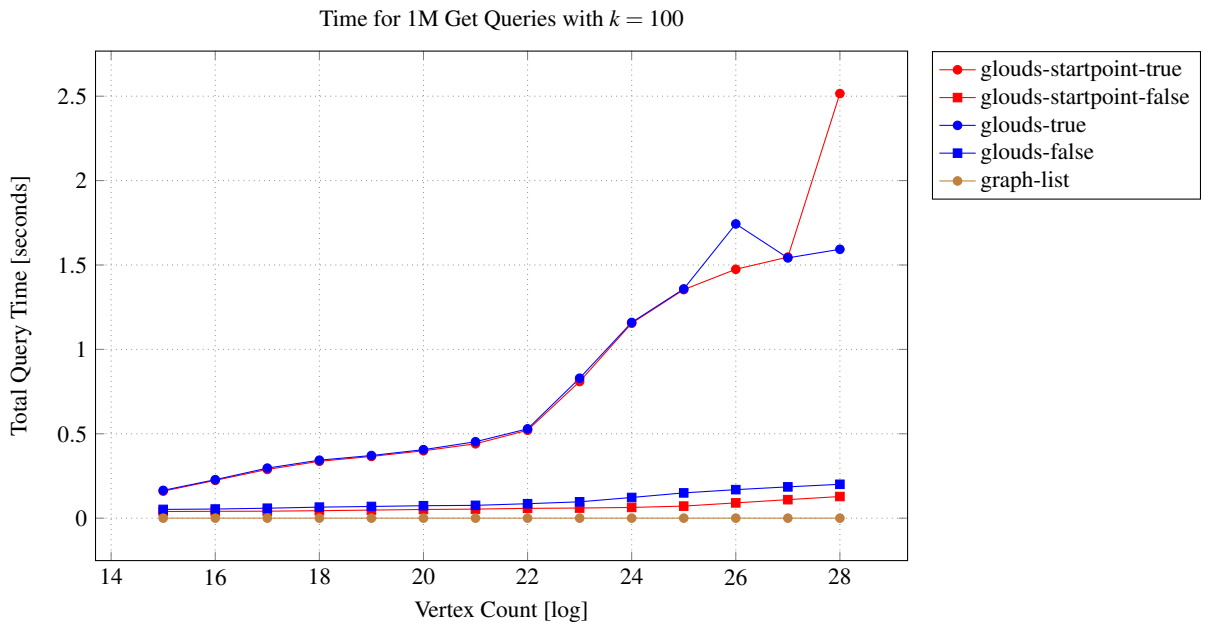


Figure 7.16: Runtime required to execute one million get queries on the respective objects for $k = 100$. We exclude graph-matrix because it performs rather poorly than the other representations (compare to Figure 7.14).

the runtime is linear with an added factor once we include the isomorphism.

Since the is_neighbour function operates on all data structures rather quick, even when increasing k, we decided to only show the time it took for the case of $k = 1$ in Figure 7.20. All classes show a very low constant runtime with only minor peaks in between.

To conclude the benchmarks, we show in Figure 7.21 that glouds-false beats the other implementations' runtime and space requirement for the construction of an object while also delivering an almost constant runtime for each method. However depending on the amount of edges that the graph contains, we showed that for a low amount of edges of any given graph, glouds-startpoint-false can save some additional space, while requiring a longer runtime for the construction of the object itself.
In addition to that, our benchmark showed that once we store the isomorphism through glouds-true or glouds-startpoint-true, we not only require a lot more space, but also incur slower running times. So, whenever the isomorphism does not have to be stored, glouds-false and glouds-startpoint-false offer benefits concerning space and runtime.

7.4.2 Further Work

Our trit vector uses a wavelet tree adaption. Fischer and Peters [FP16] use a lookup table. It is still left to implement this solution and to evaluate in which case which approach is more efficient. We store the list of shadow nodes with our integer vector implementation (see Section 4.1). Fischer and Peters [FP16] decided to use a wavelet tree (see Section 2.8). The length of the string to be encoded is the number of shadow nodes k. The size of the alphabet is $h \leq \min\{n,k\}$ where h is the number of the n vertices

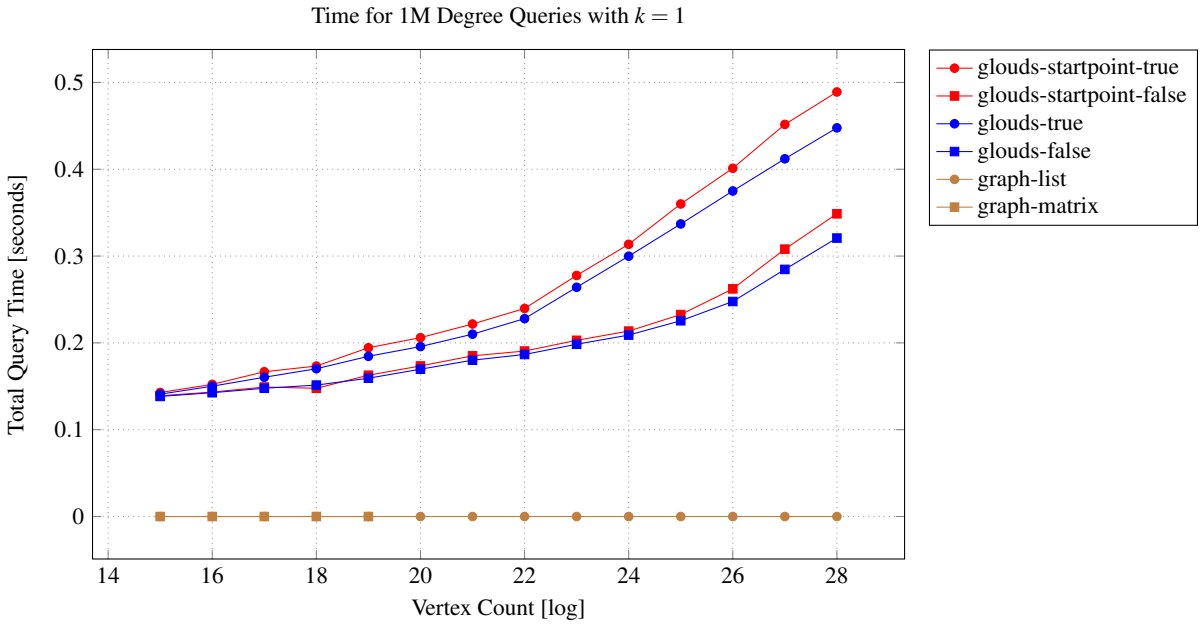


Figure 7.17: Total time (in seconds) it took to execute one million degree function calls for $k = 1$.

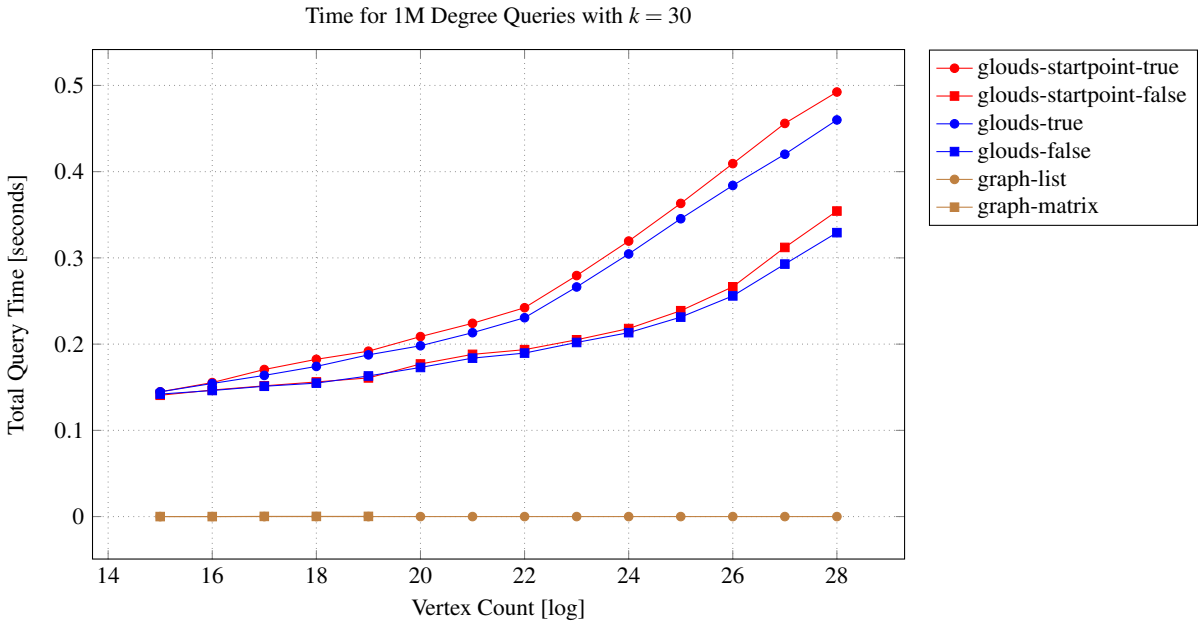


Figure 7.18: Total time (in seconds) it took to execute one million degree function calls for $k = 30$.

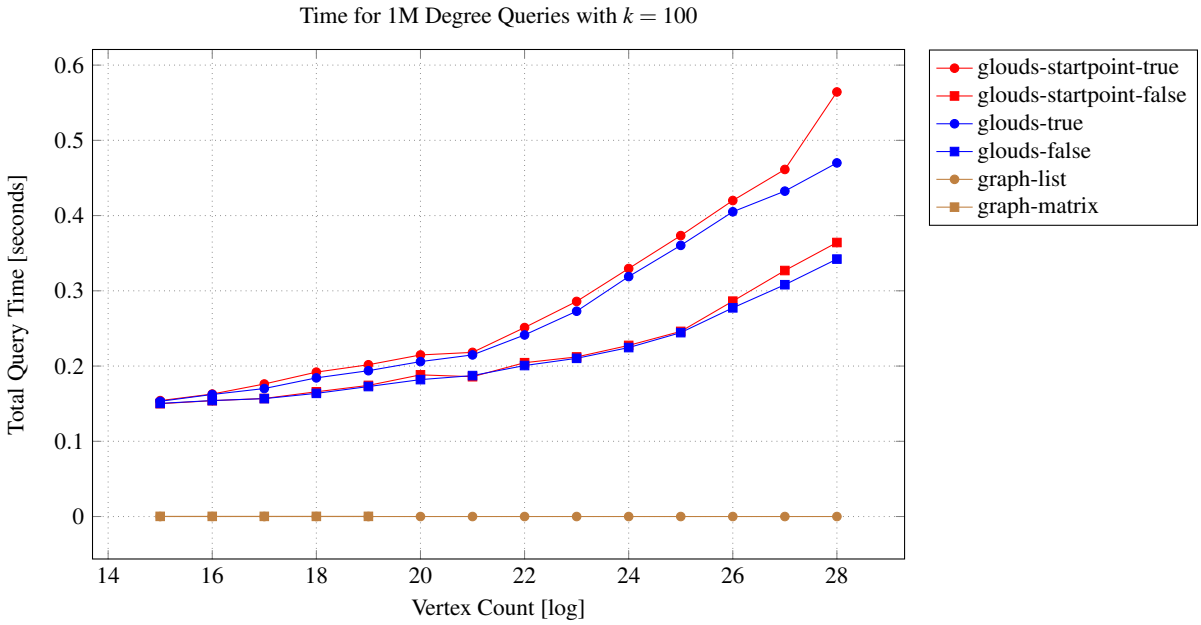


Figure 7.19: Total time (in seconds) it took to execute one million degree function calls for $k = 100$.

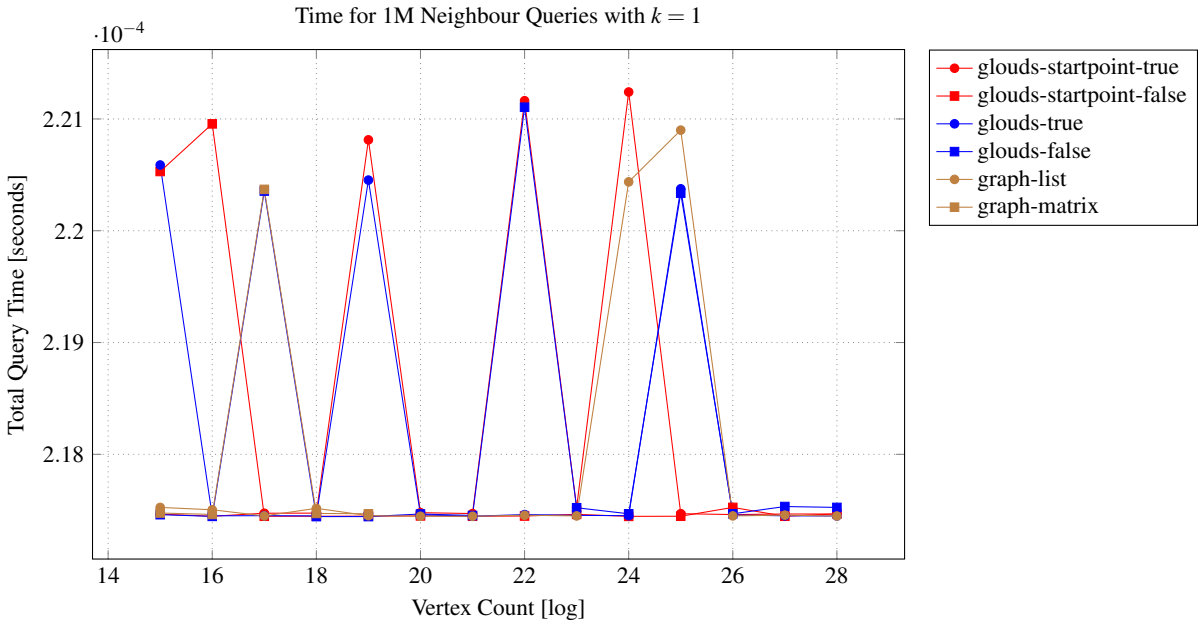


Figure 7.20: Total time (in seconds) it took to execute one million is_neighbour function calls for $k = 100$.

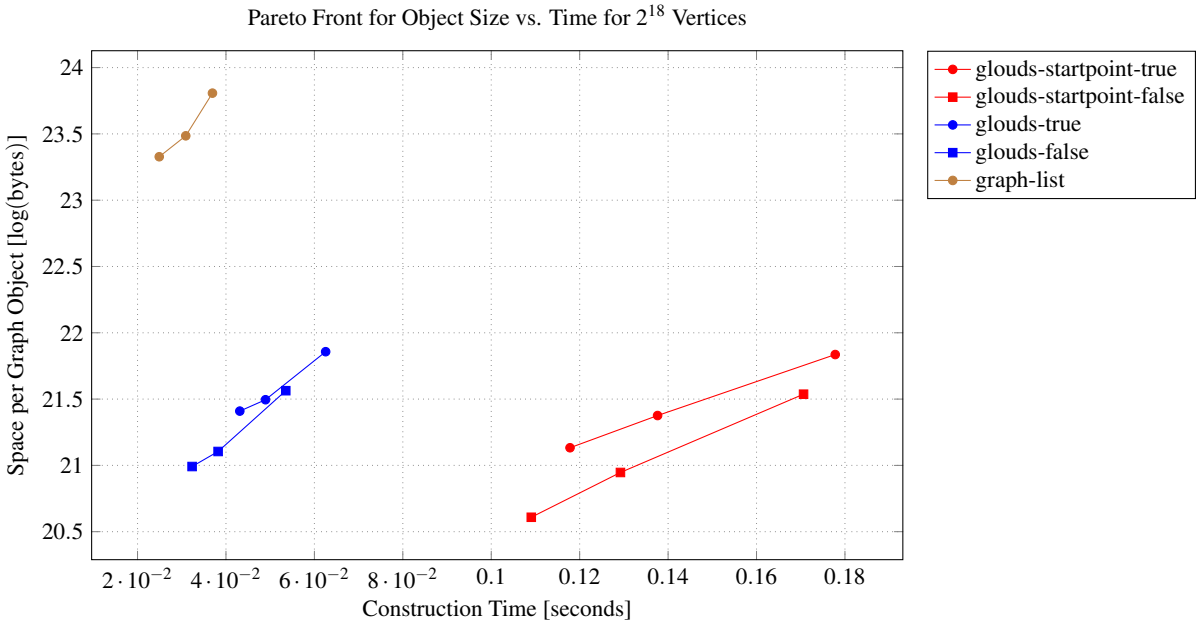


Figure 7.21: The Pareto front of the four respective objects shows the time versus space requirement. Each graph contains three vertices to represent $k = 1$, $k = 30$ and $k = 100$.

that have more than one incoming edge. So, our implementation needs $k \cdot \log n$ bits, while a wavelet tree needs $\min\{k \cdot \log(h), h \cdot log(k)\}$ bits.

In preparation for the project group, we got to know further techniques to encode trees and graphs. For trees, there is the very popular *parenthesis technique*. For this method the tree is traversed with a depth-first search. When a node is entered for the first time, an opening parenthesis or 0 is written. When it is entered for the second time, a closing parenthesis or 1 is written [Nav16, Chap. 8]. In this way, a tree can be converted to a bit vector of the length $2n$ where n is the number of nodes. Methods to determine the parent or children of a given node can be realized by rank and select operations on this bit vector in constant time.
This approach is transferred by Turán [Tur84] to planar graphs. Turán traverses the spanning tree and stores it in the parenthesis representation. In addition, for each edge of the graph which is not included in the spanning tree, the time of the first and second discovery of this edge during the traversal is stored. Thus Turán achieves a representation with $4m$ bits for a planar graph with m edges. This representation answers neighbourhood queries in $\mathcal{O}(f(m))$ time for a function $f(m) \in \omega(m)$ [FSG+16, Tur84].

The mentioned techniques are optimized for special graphs (trees, tree-like, planar graphs). Further a data structure could be developed that selects the best representation according to the properties of the input graph.

Chapter 8

Retrieval and Perfect Hashing

A very common problem is to store a set of keys associated with information. This is solved by a retrieval data structure (often also called dictionary). The naive solution is to save every key-value pair in an array, leading to $\mathcal{O}(n)$ access. If we want faster queries, we can use the two common approaches, search trees and hash tables. Besides this, we can also solve the problem with (minimal) perfect hash functions (see Section 2.4).
In this chapter we will cover two data structures introduced by Mueller et al. [MSSZ14]. FiRe (Fingerprint retrieval) solves the retrieval problem and FiPHa (Fingerprint based hashing) is a (minimal) perfect hash function. We implemented these two data structures and benchmarked them against state-of-the art implementations which provide solutions to the same problem.

8.1 Retrieval

Given a set S of n keys from a universe U, a retrieval data structure stores information that is linked to these keys. For a key k and information v it can:

- $insert(k,v)$: insert a key with its information
- $delete(k)$: delete the key and the information associated with it
- $assign(k,v)$: assign a new value to a key
- $lookup(k)$: return the value associated with k

If there is no need to manipulate the key set, *insert* and *delete* are not needed. There are two simple solutions to the problem. We can store each key-value pair in a node and store all nodes in a search tree, allowing $\mathcal{O}(\log n)$ time operations when a total order on the key set exists. The other solution is a hash table (see Section 2.4.2), which allows average constant time operations.

8.2 Related Work

There are many different ideas for building minimal perfect hash functions. FiPHa is benchmarked against three state-of-the-art implementations. These are outlined here.

8.2.1 Linear Systems

To explain the idea of random linear systems, we show how to represent a function as a linear system [GOV16]. We start by looking at the special case where we map every key k_i to a value $v_b \in F_2$, where F_2 denotes the field with two elements. We shorten every key to m bits beforehand by using a hash function $h_\theta : U \to F_2^m$ from a hash family. Now we want to find a vector w so that for each i the following holds:

$$h_\theta(k_i)^T w = v_i,$$

where $h_\theta(k_i)^T$ is the transpose of $h_\theta(k_i)$. We can now stack the row vectors $h_\theta(k_i)^T$ into a matrix H and the values v_i into the vector v, so we get:

$$Hw = v$$

The next step is to generalize to the case where $v_i \in F_2^b$, v becomes a $n \times b$ matrix and w becomes a $m \times b$ matrix. If we look at a set of 4 keys with $h_\theta(k_0) = 010,\ h_\theta(k_1) = 011,\ h_\theta(k_2) = 000,\ h_\theta(k_3) = 111$ and associated values $v_0 = 00,\ v_1 = 01,\ v_2 = 10,\ v_3 = 11$, we get

$$h_\theta(k_i) * w^T = v_i. \text{ Here: } \begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix} * \begin{pmatrix} w_{00} & w_{01} \\ w_{10} & w_{11} \\ w_{20} & w_{21} \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{pmatrix}$$

If we find a solution, we can store the function by storing θ and w. Majewski, Wormald, Havas, and Czech [HMWC93] introduced a construction for static functions using the connection between linear systems and hyper graphs using the framework above. Chazelle, Kilian, Rubinfeld and Tal[CKRT04] proposed this idea independently and noted that this can also be used to build MPHFs. The theoretic size of this is $2.46 + o(1)$ bits per key. Genuzio, Ottaviano, and Vigna [GOV16] made several improvements to this idea and present a data structure that achieves $2.24 + o(1)$ bits per key.

8.2.2 CHD

Belazzougui, Botelho, and Dietzfelbinger [BBD09] introduced an idea called CHD (compressed hash-and-displace), which splits the keys into buckets with expected size λ and tries random hash functions until there are no collisions in a bucket. For each bucket the index of the hash function is stored. It can be parametrized to produce greater buckets, but finding a hash function without collisions becomes much harder. Because this idea is used in RecSplit (see Section 8.2.3), it will be explained in more detail later. In theory, 1.44 bits per element can be reached. In practice, this data structure does not manage to go below 2 bits due to infeasible construction time. Figure 8.1 illustrates this idea.

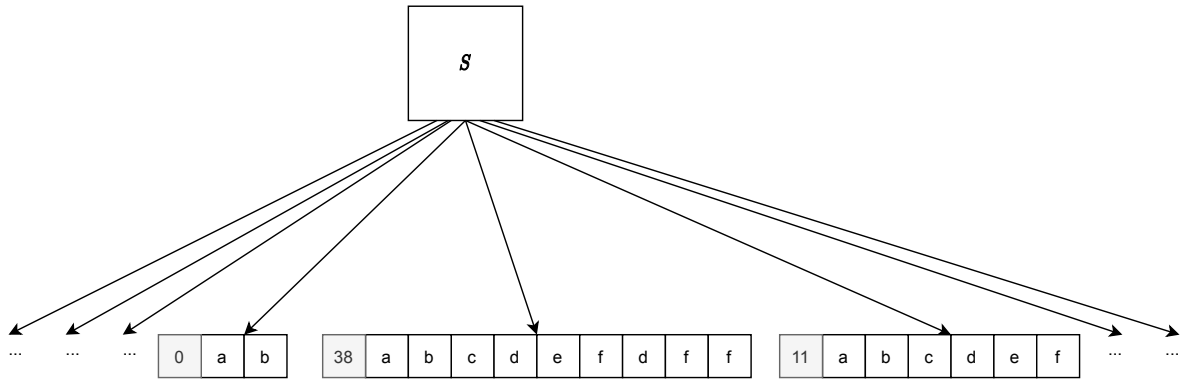


Figure 8.1: CHD for keys S. Three buckets are shown. The index of the collision free hash function is saved in every bucket.

8.2.3 BooMPHF

BBHash [LRCP17] is a MPHF library based on fingerprinting that aims to handle large data sets. While slightly larger (about 3 bits per key) than MPHFs from other libraries, it has a faster construction and query time. Because FiRe and FiPHa share the same properties, this comparison is especially useful to evaluate our implementation. BBHash also works in parallel.

8.2.4 RecSplit

The idea of RecSplit [EGV20] is to split the set of keys into small sets, for which we can find a MPHF more easily. The first step is to partition the keys into manageable buckets of expected size $b \in \mathbb{N}^+$ by using a random hash function $g : S \to \lfloor n/b \rfloor$. These buckets contain a subset of the keys which then can be *split recursively* into smaller subsets until we reach a leaf size l. For these small sets we can calculate a minimal perfect hash function by brute force using universal hashing. The two parameters b and l lead to different data structure sizes, construction times and hash query times. To calculate a hash value of a key we have to find the bucket of the key. The hash value is then calculated by adding the number of keys in previous buckets and the value of the MPHF inside the bucket (see Figure 8.2).
Our FiRe data structure stores the perfect hash value of the keys in a way that $\mathcal{O}(1)$ time queries can return this hash.

8.3 Fingerprint Retrieval (FiRe)

A *perfect hash function* maps keys uniquely to a small range of IDs. This can be used to build a structure that stores data linked to keys, which in general is called a *retrieval data structure*. If the keys are too long, e.g. in the case of websites, we can use short *fingerprints*[KR87] instead. Although fingerprinting is a well known method for indexing structures, it has not been applied to perfect hashing or retrieval. Mueller et al.[MSSZ14] propose a simple retrieval data structure that maps keys to buckets that are capable of storing keys with several different fingerprints. This strategy may lead to the problem that keys in the same *bucket* have the same fingerprint and therefore cannot be distinguished. To solve this

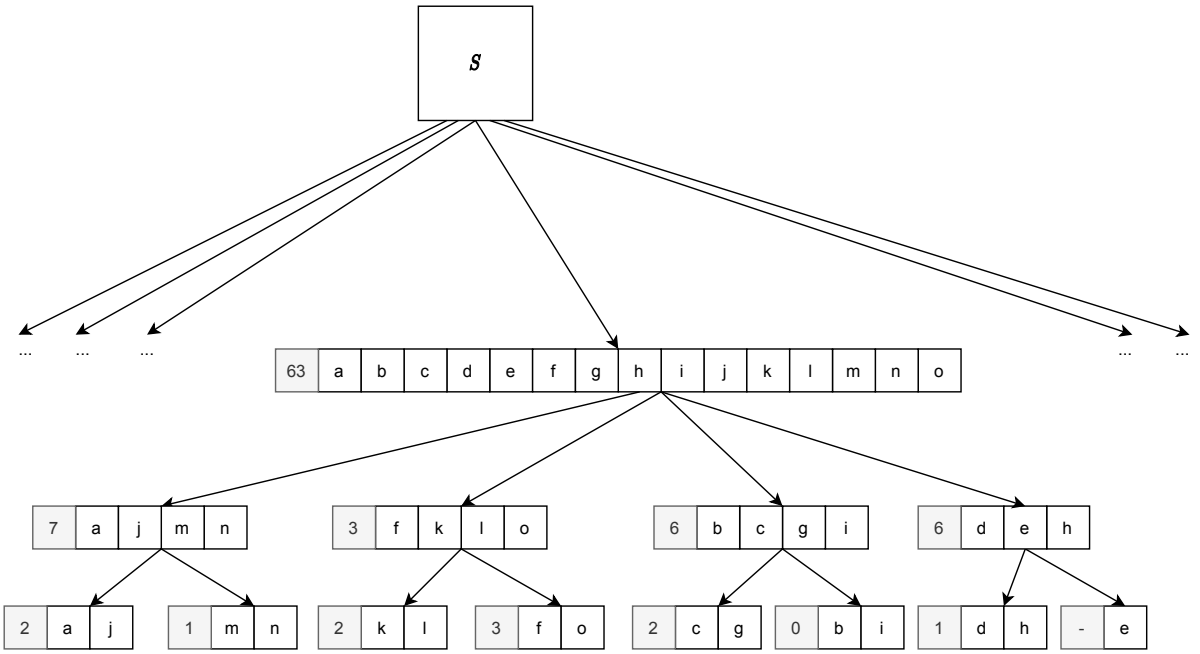


Figure 8.2: Splitting tree that illustrates how RecSplit works. We calculate the hash value by going into the correct bucket. We take the information that 145 keys have been stored in previous buckets. Now we use the hash function with index 63 and see that the key is in the third split. Because the splitting strategy is shared globally we know that we skip 4 keys per split. Now we use the hash function with index 5 and move to the first split. In this we get the value 1 from the hash function with index 1. We now return $145+2*6+0*2+1=158$.

problem, an overflow data structure is created to store all colliding keys. The overflow data structure has the same structure as the original structure but is created with a different hash function. It stores not only the colliding elements but also the elements that do not fit in an overfull bucket. We can have many levels of overflow data structures. The elements are pushed recursively to the data structure until the size of the last overflow data structure is small enough to change to a more expensive naive data structure. Here we explain how the FiRe data structure is constructed and how data can be retrieved with FiRe. We introduce our implementation shortly and show the result of our benchmark as well as possible improvements.

8.3.1 Structure of FiRe

Our goal is to map a set of n keys to unique values in $\{0,...,n-1\}$. For that we say that the first key has a hash value of 0, the second key has a hash value of 1 and so on. To construct this minimal perfect hash function, the first level of a FiRe data structure has $m = n/b$ buckets for some parameter $b \in \mathbb{N}^+$. In each bucket we can store the hash value of up to a keys for some parameter $a \in \mathbb{N}^+$. To remember which value belongs to which key, we save the fingerprint of the key for every cell. This is accomplished by a bit vector of size k for some parameter $k \geq a$. If $rank_1(B,y) = z$ for a bucket B, then the value in cell z corresponds to the key with fingerprint y. To calculate the fingerprint of a key, we use a function $h_f : U \to \{1,...,k\}$. The correct bucket for a key is obtained from a function $h_B : U \to \{1,...,m\}$. We can calculate both correct bucket and fingerprint with one function $h \to \{1,...,km\}$. If two keys collide

(i.e. $h(k_1) = h(k_2)$), we are not able to distinguish them in the first level. In this case we build the next level from these elements. It is also possible that too many keys map to a bucket. In this case we can choose some elements and also put them in the next level. To solve collisions, every layer uses a different hash function. Collisions in the second level can be passed to the next level and so on. After enough repetitions or in the case only few elements are left, we can store the remaining keys in a naive data structure, e.g. a list of $(key, value)$ tuples.

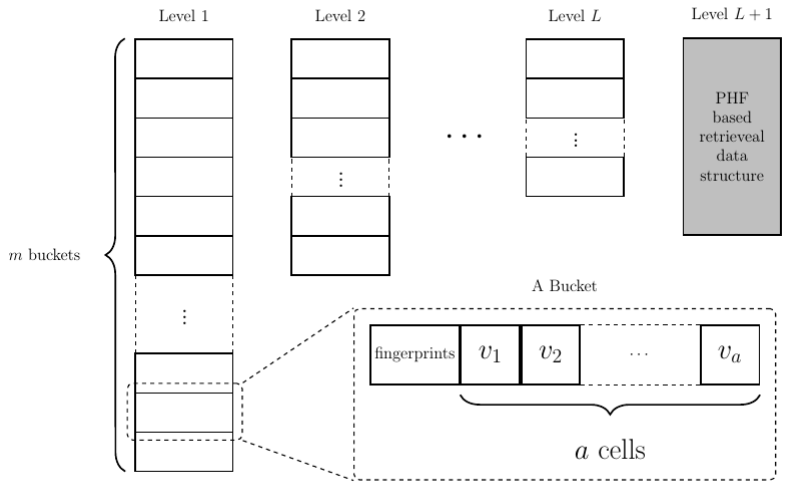


Figure 8.3: Illustration of the FiRe data structure [MSSZ14]

The number of the buckets in the successive levels of FiRe decreases as we add more layers. For L as the number of levels in FiRe, tuning the parameters (a,k,b,L) will be the main task for the implementation.

8.3.2 Hash Query

To get the hash value of the key, we have to check at each level whether the bit for the fingerprint is set. We know that the first level for which this is true contains the correct fingerprint, because no collision happened here. If nothing is found, the method will return a special value indicating that the key is not in the data structure. Keys that are not in S could have the same hash as a valid key. In this case a hash value will be output, even though the key does not exist.

8.3.3 Implementation

In our implementation, we build the data structure by constructing each level separately. The size of a bucket is set and the number of buckets is decided by the number of elements. First, we look at every element and remember which cell it would occupy (i.e. the fingerprint). For every bucket we save a list with all elements that belong to the bucket. If two elements would occupy the same cell, we memorize that a collision happened there. After considering every element, we sort the lists by their elements' fingerprints and push the first a elements that did not collide into each bucket. Elements that overflow are added to the next level.

Theoretical parameters	Template parameters
L	`t_levels`
b	`t_b`
k	`t_fp`
a	`t_cells`

Table 8.1: Corresponding template parameters to theoretic parameters

Usually we are able to store much more than half of the remaining elements per level, which means that the level size decreases exponentially. Because this is not guaranteed, we only build up to thirty levels. We also stop building levels if there are less than 1024 elements left. The remaining elements are saved in a list that stores all remaining $(key, hash)$ tuples (see Section 8.3.1). Searching for a key means going through the list and returning the hash value as soon as we find the correct key.
Hash queries are answered by checking if the fingerprint bit is set in the corresponding bucket. If not, we look in all the other layers and the fallback list. If it is set, we do a naive rank query and return the hash value in the cell at index determined by the rank query. The main task of the implementation is to find parameters which achieve low space overhead and fast query times. To this end, some combinations of the parameters will be tested with a benchmark. For conveniently evaluating different combinations of parameters, we used the template parameters listed in Section 8.3.3.
Besides these template parameters, we need also a maximal number of levels in the data structure which can be controlled with `t_levels`. When the number of levels in the data structure reaches this boundary, all of the remaining keys will be pushed to the naive data structure.
We tuned the template parameters step by step.

Bucket Size

A main advantage of FiRe is the low amount of cache misses. In order to achieve that, we need to construct buckets that fit into one cache line. Cache misses correspond to the number of levels that have to be scanned. To achieve few cache misses we tune the number of fingerprints per bucket k and the number of cells per bucket a. Because we use naive rank queries to scan which fingerprint belongs to which cell, more fingerprints mean that we have to do longer queries. In order to find good parameters, we built a full FiRe level, where every bucket is full, and tested the speed of queries for different a and k.
We choose $a = 30, k = 127$, so that k is a prime and a bit vector of size k fits into two 64-bit integers.

Level Size

Now that we have a fixed bucket size, we have to decide how many buckets we use per level. Because fewer buckets result in more collisions, the more buckets the data structure has, the fewer levels it requires. Since the hash query time is dependent on the number of levels, this will lead to better query

times. Nevertheless, the buckets will not be filled as well as possible in this case. So the number of buckets in each level must be controlled through parameters. In order to find good combinations of parameters, we constructed a FiRe data structure from random numbers and tuned the parameter b, which determines the number of buckets $m = n/b$. Here we looked at the space overhead and the query time.
We choose $b = 54$, because it is the fastest among the ones that are close to 8 bytes per element.

Template parameters	Default value
`t_levels`	30
`t_b`	54
`t_fingerprints`	127
`t_cells`	30

Table 8.2: Default values for the template parameters in FiRe

8.4 Fingerprint Based Perfect Hashing (FiPHa)

FiPHa is a optimized version of FiRe regarding space usage. In FiPHa, we do not store any information about the key in the bucket besides the fingerprint information. This means that the cells in FiRe are discarded and no bits are required to store the calculated values of the keys. Without the presence of values, there is no collision in FiPHa. Therefore, retrieving data from this data structure can be seen as perfect hashing.

8.4.1 Structure of FiPHa

In particular, the fingerprint for each key is calculated with the hash function h_f which is introduced in Section 8.3.1. If the fingerprint $h_f(x)$ of a key x is a valid fingerprint in the bucket determined by $h_b(x)$, the fingerprint can be used to define the injective function $h_p(x) = ah_b(x) - a + \text{rank}_1(B,x)$. If it is not a valid fingerprint, we use the information $h_p(x)$ for the next level and add into it the offset am. Besides these slight changes, the structure of FiPHa is analogous to FiRe.

8.4.2 Implementation

The implementation of FiPHa can be adapted from the implementation of FiRe introduced in Section 8.3.3. In comparison to the structure explained above, there is little difference in the actual implementation from FiPHa. We use the fingerprints directly as a replacement for the bucket and each key x is allocated to the corresponding fingerprint with $h_b(x) = x \mod p$ where p is the number of fingerprints in the level. Essentially, these fingerprints act as buckets in this implementation. Therefore, the structure for buckets in FiRe is discarded. Only the structure for level remains. The function h_p is

defined as $h_p(x) = n' + \text{rank}_1(B, h_b(x)) - 1$ where n' is the number of keys added to the level before x and $\text{rank}_1(B, h_b(x))$ indicates rank_1 at the position (i.e. fingerprint) $h_b(x)$.
Like FiRe, the levels are constructed separately. We use the template parameters `t_levels` for the maximal number of levels in the data structure and `t_b`/`t_fp` for the number of fingerprints in each level which is calculated together with the number of remaining keys. These parameters are currently set as constants and can be changed for optimization. First, we determine the corresponding fingerprint for each key in the level. If the fingerprint is already set, the key is collected in a list of overflow elements and will be pushed to the next level.
When there are less than 1024 elements left, the construction of the levels is stopped and all the remaining keys will be pushed into a naive data structure (see Section 8.3.3).

Hash query

To calculate the hash value of a key x, we go through all the levels in the data structure. In each level, we must check if the fingerprint $h_b(x)$ is set. The fingerprint information $h_b(x)$ will be used to determine the hash value $h_p(x)$.

Level size

Since the buckets were discarded in our implementation for FiPHa, we only need to search for a good level size, which is also the number of fingerprints in this level. We tried to determine good parameters for the level size in a similar way to FiRe.
The current values for the template parameters can be found in Section 8.4.2

Template parameters	Default value
`t_levels`	30
`t_b`	6
`t_fp`	4

Table 8.3: Default values for the template parameters in FiPHa

8.5 Benchmark

We compare our FiRe- and FiPHa implementation against MPHFs BBHash from Limasset et. al. and RecSplit from Emmanuel Esposito, Thomas Mueller Graf and Sebastiano Vigna [EGV20]. In [LRCP17], the authors claimed that BBHash has a faster construction time, but a slightly bigger space overhead than state-of-the art libraries. BBHash does not save the hash values because they are regained by a rank query over a large bit vector. This is a big advantage over FiRe, because the hash values take up several bytes per element, while the additional bit vectors only need a few bits per element.

In RecSplit, the set of keys is partitioned in buckets and each bucket forms a tree which defines an independent MPHF. The authors analysed these trees exactly for efficient search by brute-force, for fast lookup and efficient storing space. It is claimed that RecSplit has a better balance between lookup speed and space usage, which is a problem for FiRe and FiPHa, since construction time and lookup is very slow with the space usage of 2.58 bits per key.
We measured construction time by constructing a MPHF from sets with up to 2^{29} keys (see Figure 8.4). Then we measured the space consumption per element for different sized key sets (see Figures 8.5 and 8.6). After that, we looked at hash query times for different sized key sets (see Figure 8.7). At the end, we also sketched the ratio between construction time and construction space of the observed MPHFs (see Figure 8.8). To confirm that the query results are correct, we compared the sum of all hash queries.

8.5.1 Construction Time

We only examine the construction time of the observed MPHFs with sets of keys that contain more than 2^{21} keys. For smaller key sets, all other MPHFs are constructed faster than RecSplit, which has only a throughput of 2×10^6 keys per second. Nevertheless, the construction time of RecSplit does not change much when the key set gets larger, while the construction time of other MPHFs is strongly affected by the size of the key set. With a key set that contains more than 2^{26} keys, FiRe and BBHash are constructed slower than RecSplit. The construction of FiPHa up to 2^{29} keys is still the fastest one. However, all observed MPHFs are constructed as fast as each other when the key set becomes larger.

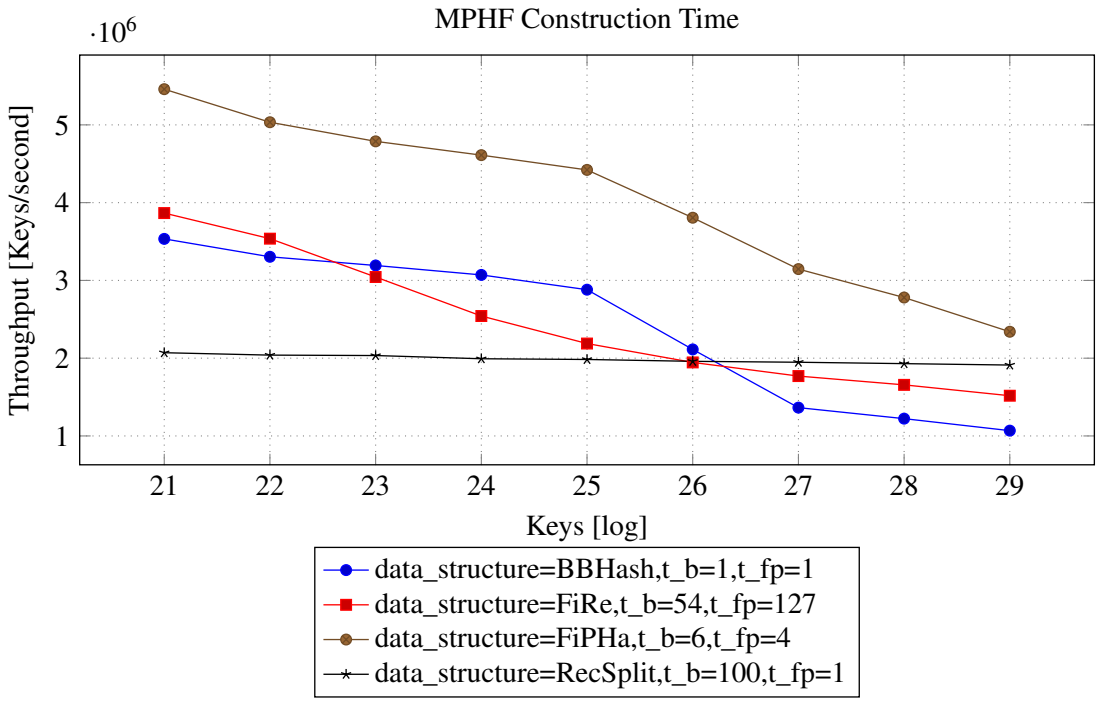


Figure 8.4: Construction time for random sets of keys by different MPHFs. The construction time for RecSplit is quite stable, while the construction time of other MPHFs drops strongly with larger key sets.

8.5.2 Space Overhead by Construction

Although BBHash needs more time than FiPHa to be constructed (see Figure 8.4), both of them have quite similar space overhead for different key sets, which leads to the result that our implementation of FiPHa is even slightly better than BBHash. Our FiRe implementation needs more than 4 bytes per key due to storing the values, as 32 bits alone are needed to store the hash value. If these values can be discarded, the space usage for FiRe would be better. This is shown in Figure 8.6. RecSplit is still the best MPHF among the others regarding space overhead. This MPHF uses only about 2 bits per key in its construction.

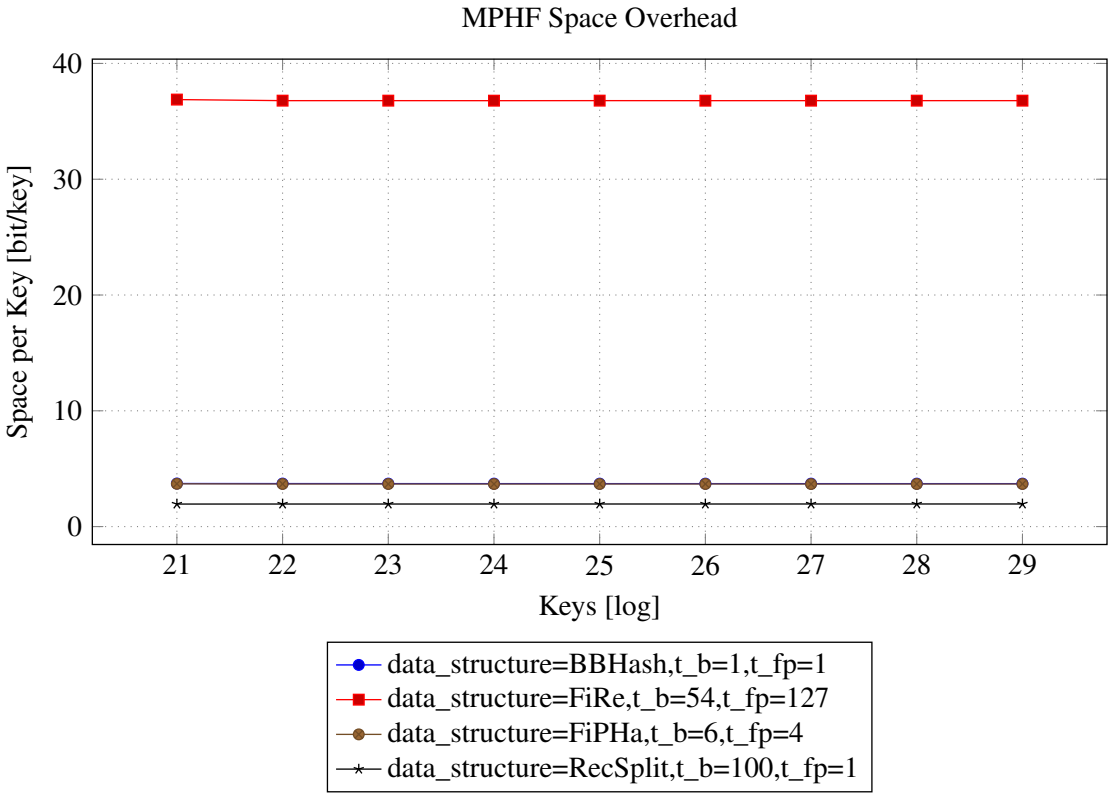


Figure 8.5: Space overhead for random sets of keys. This diagramm is dominated by the space overhead of FiRe, since each hash value alone is stored with 32 bits.

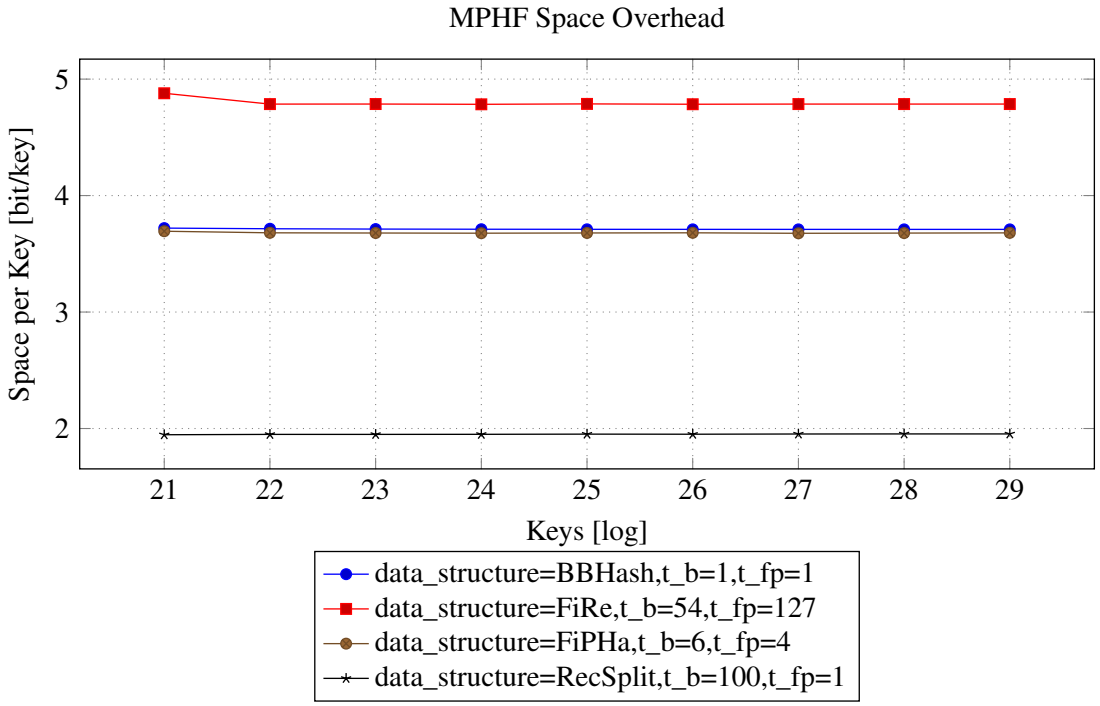


Figure 8.6: Enlarged diagram from Figure 8.5 with the assumption that the values in FiRe are discarded. Here, FiRe requires only about 5 bits per key. BBHash does not need to store the hash values, which is a big advantage. FiPHa is as good as BBHash, even a little bit better. The space optimization of RecSplit, which uses only about 2 bits per key, can be seen clearly.

8.5.3 Hash Query Time

For hash query time, FiRe seems to be faster with smaller sets of keys. With more than 2^{22} keys, hash query time for FiRe becomes slower, since more levels are added to the data structure. From 2^{22} keys up to 2^{26} keys, FiPHa and BBHash are faster than FiRe regarding the hash query time. For a key set which is smaller than 2^{26} keys, FiPHa has the fastest hash query time with about 1.5×10^7 hash queries per second. RecSplit is not the best one for small key sets, but hash queries with RecSplit are faster than with other MPHFs. For a key set which contains more than 2^{26} keys, RecSplit is the better choice, since about 10^7 hash queries are performed per second, while the fastest hash query time among other MPHFs can reach only 80% of this speed.

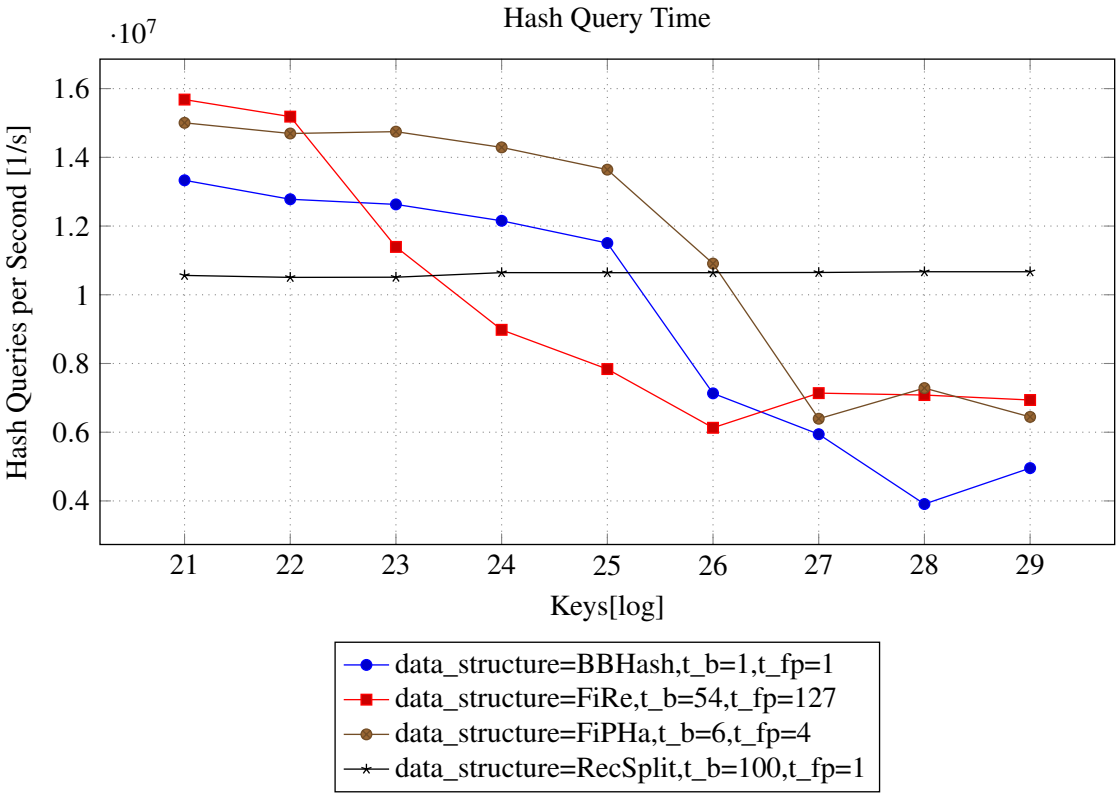


Figure 8.7: Query time for random sets of keys. For bigger key sets, our FiRe- and FiPHa implementation is as fast as BBHash and faster than RecSplit.

8.5.4 Ratio between Construction Time and Construction Space Overhead

In order to highlight the relation between space and time consumption of the construction of the observed MPHFs, we also sketched a Pareto front for the ratio of these properties in Figure 8.8. For each data structure, this ratio is shown for key sets with a size from 2^{26} to 2^{29} keys. The four markers for each data structure correspond to the four sizes of key set in the order 2^{26}, 2^{27}, 2^{28} and 2^{29}. Assuming that FiRe does not use 32 bits for each hash value, we can see that the space consumption for the construction of all MPHFs does not change much for different key sets. However, in order to keep the space usage low, FiPHa needs more time to be constructed. Although this problem in construction time seems not to be apparent in FiRe, the construction of our FiRe implementation still requires more space than the other MPHFs. This diagram shows once again that RecSplit is still the best choice for large key sets, since space and time consumption of the construction of RecSplit is not strongly affected by the size of key set.

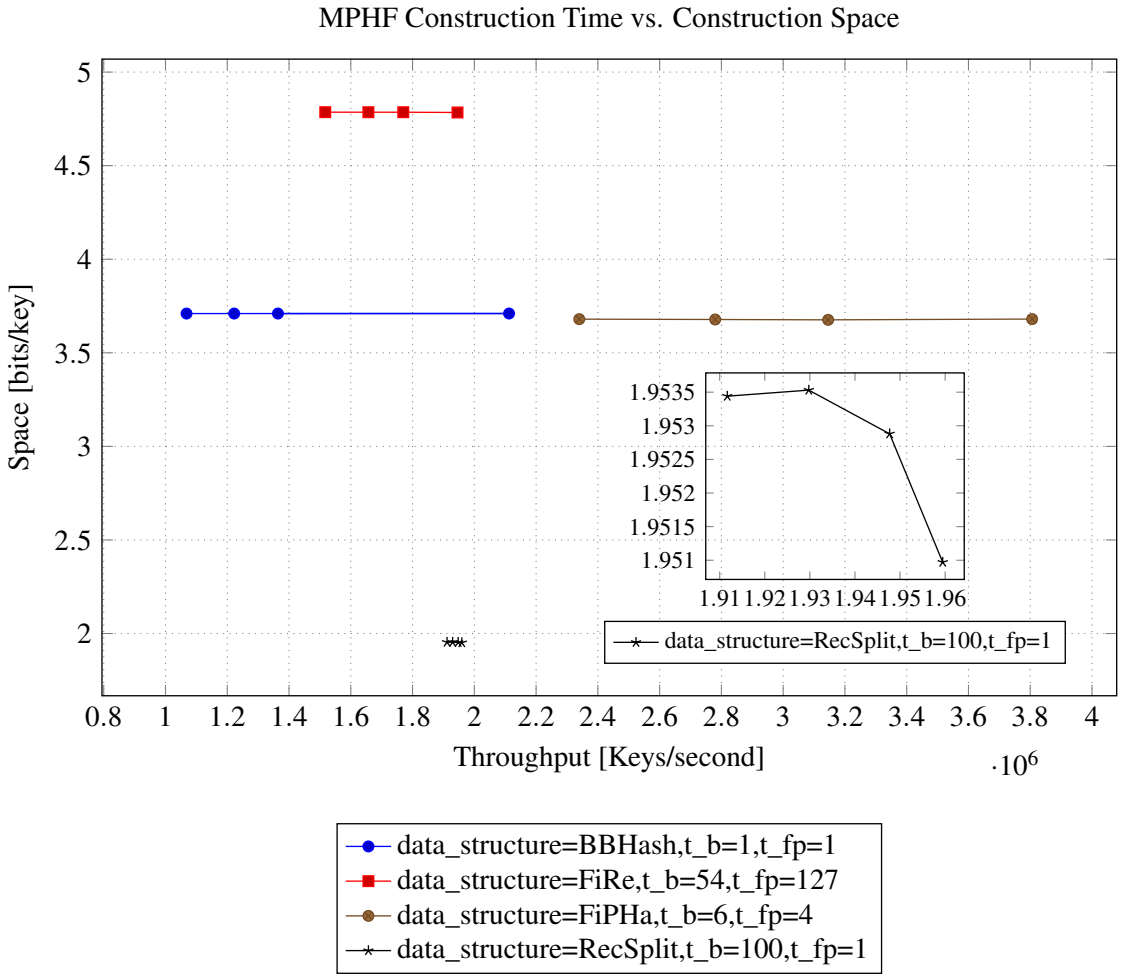


Figure 8.8: Ratio between Construction Time and Construction Space Overhead

8.6 Results and Further Work

The FiRe and FiPHa data structures in [MSSZ14] are competitive in build time and query time, but not in space usage. In our implementations, the space usage is not yet optimal like in the literature, since we want to keep a balance between construction time, query time and space usage.

In the data structure FiRe, the values of the keys are stored in the data structure. Therefore, it is like a hash table and the space overhead is quite high in comparison to other data structures like BBHash or RecSplit. FiPHa is an optimised version of FiRe and has a similar approach to BBHash. In addition, we see that in the benchmark results, the hash query time, build time and the space usage of both structures are not very different. RecSplit is still the best data structure with regard to the space usage.

It is surely possible to reduce the space needed by choosing better hash functions or parameters. However, there are no optimal parameters and we must accept a trade-off between time and space.

Chapter 9

Summary and Further Work

In this report we gave an overview of the work we did during the year of our project group. We have developed a library for succinct data structures including static and dynamic bit vectors, graphs, grammar-compressed strings and minimal perfect hash functions. All implemented methods work correctly and build a good basis for further improvements. We developed benchmarks to evaluate our data structures and compared them to other existing implementations. Here we give a more detailed overview of the implemented data structures and discuss possible improvements.

- **Dynamic bit vectors**: We implemented dynamic bit vectors that allow dynamically inserting and removing elements in addition to answering queries. We accomplish this by storing bits within the leaves of an AVL tree. We also implemented a compressed variant of the bit vector which uses the $\mathscr{H}_0$ entropy of the bits to reduce space usage. Bits are grouped into blocks of 64 and are then assigned a class and an offset which may be stored in fewer physical bits than 64, provided the block has many more ones than zeroes or vice versa.

  Leaves have a maximum size of $2w^2$ bits, so if a leaf grows too large after an insertion, the leaf must be split in two, creating a new inner node. Correspondingly, if a leaf becomes too small after a deletion, two leaves are merged. Operations are performed in $\mathscr{O}(w)$ time, with the exception of insertions and deletions on compressed bit vectors, which require $\mathscr{O}(w^2)$ time, due to needing to recompress the blocks of a leaf. We compared the two data structures to a succinct bit vector implementation and a gap bit vector implementation from Nicola Prezzas DYNAMIC library. Overall, the performance of our data structures with regard to space usage and run time are satisfactory, although further space savings could be achieved by storing less redundant attributes in a tree node.

- **Grammar-compressed strings**: We built a data structure that is able to compress texts into grammars and stores them succinctly while also enabling operations on the string. It performs well on highly compressible or small texts, but is able to handle larger texts as well.
  We rely on the RePair [LM00] algorithm to find a small grammar and this could be a first target for optimization. A big challenge for the construction of the data structure was also finding a good partition of monotone sequences, which we managed to do fairly well, but we believe this

can surely be improved to make the overall compression better regarding space and time.
While we needed some extra space to make the access to single symbols inside the text possible in a reasonable time frame, the running times, again especially for the smaller grammars, were quite good. The fact that our `access` function is only really competitive on smaller texts or grammars also shows some scalability issues that we did not manage to fix. Something we also leave for further work is the implementation of the `contains` method, which we were not able to fully realize as we wanted to. Here, a possible solution is building a separate structure for pattern matching inside a grammar, which would however also increase the needed space again.

- **Graphs**: We implement GLOUDS [FP16] by Fischer and Peters. This representation is optimised for tree-like graph, but it works for any directed, unweighted graph. For GLOUDS we developed an own solution for storing a trit vector and implement a succinct representation of permutations by Navarro [Nav16]. We implement two versions of GLOUDS. The first one calculates the optimal start points for the breath-first search which encode the graph. This is solved by a depth-first search. The other version chooses the start points by increasing names which is possible not optimal. This reduce the construction runtime, but we need additional space and a slower runtime for neighbour queries. We benchmark our implementations for arbitrary graphs with a number of edges which is linear in the number of vertices. Both GLOUDS versions overtop the space requirement of the naive representations, the adjacency matrix and the adjacency list. For get and degree queries we did not beat the adjacency list, but all requested operations need a low constant running time even for large graphs.
  Fischer and Peters discuss some further optimisations, like using a wavelet tree or look-up-table. While GLOUDS is optimised for tree-like graphs there exist different other data structure optimised for other types of graphs, e.g. trees or planar graphs. A possible extension of our library would be to implement more of these data structures and then automatically decide for a given graph which of these structures is the best fit.

- **Minimal perfect hash functions**: We implemented FiRe and FiPHa and included default parameters that worked well on our test system. The implementation from FiPHa is adapted from FiRe with some changes explained in Section 8.4. FiRe needs quite a lot of space for storing the hash values, while FiPHa does not store the hash values in the data structure, but they can be calculated with the fingerprint information. With small space requirement (see Figure 8.6), FiPHa is a succinct data structure. Our implementation is up to now comparative with BBHash but not so good as RecSplit for large key sets, which has a quite stable construction time, construction space overhead and hash queries time. For small key sets, both data structures FiRe and FiPHa have fast construction and hash queries. Features, that could be implemented next are a parallel construction or dynamic data structures that can insert and delete keys.

Bibliography

[BBD09] Djamal Belazzougui, Fabiano C. Botelho, and Martin Dietzfelbinger. Hash, displace, and compress. In *ESA*, volume 5757 of *Lecture Notes in Computer Science*, pages 682–693. Springer, 2009.

[BF98] Reuven Bar-Yehuda and Sergio Fogel. Partitioning a sequence into few monotone subsequences. *Acta Informatica*, 35(5):421–440, 1998.

[BKZ20] Fabiano Botelho, Yoshiharu Kohayakawa, and Nivio Ziviani. An approach for minimal perfect hash functions for very large databases. 03 2020.

[Bra08] Peter Brass. *Advanced Data Structures*. Cambridge University Press, 2008.

[CKRT04] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. The bloomier filter: an efficient data structure for static support lookup tables. In *SODA*, pages 30–39. SIAM, 2004.

[CL05] Chin-Chen Chang and Chih-Yang Lin. Perfect hashing schemes for mining association rules. *Comput. J.*, 48(2):168–179, 2005.

[CLM16] Rayan Chikhi, Antoine Limasset, and Paul Medvedev. Compacting de bruijn graphs from sequencing data quickly and in low memory. *Bioinform.*, 32(12):201–208, 2016.

[CLRS09a] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

[CLRS09b] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[CNP15] Francisco Claude, Gonzalo Navarro, and Alberto Ordóñez Pereira. The wavelet matrix: An efficient wavelet tree for large alphabets. *Information Systems*, 47:15–32, 2015.

[cpl] cplusplus.com. Operators. `http://www.cplusplus.com/doc/tutorial/operators/`. Accessed: 2020-02-19.

[CSM13] Yupeng Chen, Bertil Schmidt, and Douglas L. Maskell. A hybrid short read mapping accelerator. *BMC Bioinform.*, 14:67, 2013.

[Die06] Reinhard Diestel. *Graphentheorie*. Springer, 3rd edition, 2006.

[EGV20] Emmanuel Esposito, Thomas Mueller Graf, and Sebastiano Vigna. Recsplit: Minimal perfect hashing via recursive splitting. In *ALENEX*, pages 175–185. SIAM, 2020.

[FK17] Johannes Fischer and Florian Kurpicz. Fast and simple parallel wavelet tree and matrix construction. *Computing Research Repository*, abs/1702.07578, 2017.

[FKS82] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with O(1) worst case access time. In *FOCS*, pages 165–169. IEEE Computer Society, 1982.

[FP16] Johannes Fischer and Daniel Peters. GLOUDS: representing tree-like graphs. *Journal of Discrete Algorithms*, 36:39–49, 2016.

[Fre] Free Software Foundation, Inc. GNU Manual: Using the GNU Compiler Collection (GCC). `https://gcc.gnu.org/onlinedocs/gcc/`. Accessed: 2020-03-17.

[FSG+16] Leo Ferres, José Fuentes Sepúlveda, Travis Gagie, Meng He, and Gonzalo Navarro. Fast and compact planar embeddings. *Computing Research Repository*, abs/1610.00130, 2016.

[Gee] GeeksforGeeks. Bitwise Operators in C/C++. `https://www.geeksforgeeks.org/bitwise-operators-in-c-cpp/`. Accessed: 2020-02-19.

[GGMN05] Rodrigo González, Szymon Grabowski, Veli Mäkinen, and Gonzalo Navarro. Practical implementation of rank and select queries. In *In Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA'05) (Greece*, pages 27–38, 2005.

[GGV03] Roberto Grossi, Ankur Gupta, and Jeffrey S. Vitter. High-order entropy-compressed text indexes. In *Symposium on Discrete Algorithms*, pages 841–850. ACM/SIAM, 2003.

[Gol07] Alexander Golynski. Optimal lower bounds for rank and select indexes. *Theoretical Computer Science*, 387(3):348–359, 2007.

[GOV16] Marco Genuzio, Giuseppe Ottaviano, and Sebastiano Vigna. Fast scalable construction of (minimal perfect hash) functions. In *SEA*, volume 9685 of *Lecture Notes in Computer Science*, pages 339–352. Springer, 2016.

[HMWC93] George Havas, Bohdan S. Majewski, Nicholas C. Wormald, and Zbigniew J. Czech. Graphs, hypergraphs and hashing. In *WG*, volume 790 of *Lecture Notes in Computer Science*, pages 153–165. Springer, 1993.

[Huf52] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[Jac89] Guy Jacobson. Space-efficient static trees and graphs. In *Foundations of Computer Science*, pages 549–554. IEEE Computer Society, 1989.

[Jos10] Brijendra Kumar Joshi. *Data Structures and Algorithms in C++*. Tata McGraw Hill Education, 2010.

[Knu74] Donald E. Knuth. Structured programming with go to statements. *ACM Computing Surveys*, 6(4):261–301, 1974.

[KR87] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987.

[LM00] Niklas J. Larsson and Alistair Moffat. Off-line dictionary-based compression. *Proceedings of the IEEE*, 88(11), 2000.

[LRCP17] Antoine Limasset, Guillaume Rizk, Rayan Chikhi, and Pierre Peterlongo. Fast and scalable minimal perfect hashing for massive key sets. In *SEA*, volume 75 of *LIPIcs*, pages 25:1–25:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[MSSZ14] Ingo Müller, Peter Sanders, Robert Schulze, and Wei Zhou. Retrieval and perfect hashing using fingerprinting. In *Symposium on Experimental Algorithms*, volume 8504 of *Lecture Notes in Computer Science*, pages 138–149. Springer, 2014.

[Mun96] J. Ian Munro. Tables. In *FSTTCS*, volume 1180 of *Lecture Notes in Computer Science*, pages 37–42. Springer, 1996.

[MWHC96] Bohdan S. Majewski, Nicholas C. Wormald, George Havas, and Zbigniew J. Czech. A family of perfect hashing methods. *Comput. J.*, 39(6):547–554, 1996.

[Nav12] Gonzalo Navarro. Wavelet trees for all. In *Combinatorial Pattern Matching*, volume 7354 of *Lecture Notes in Computer Science*, pages 2–26. Springer, 2012.

[Nav16] Gonzalo Navarro. *Compact Data Structures - A Practical Approach*. Cambridge University Press, 2016.

[Pre17] Nicola Prezza. A framework of dynamic data structures for string processing. In *International Symposium on Experimental Algorithms*. Leibniz International Proceedings in Informatics (LIPIcs), 2017.

[Pun08] Aniruddha Puntambekar. *Analysis And Design Of Algorithms*. Technical Publications Pune, 2008.

[RS04] Phillip Rogaway and Thomas Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In *FSE*, volume 3017 of *Lecture Notes in Computer Science*, pages 371–388. Springer, 2004.

[Ste07] Angelika Steger. *Diskrete Strukturen: Band 1: Kombinatorik, Graphentheorie, Algebra*. Springer, 2nd edition, 2007.

[TTS13] Yasuo Tabei, Yoshimasa Takabatake, and Hiroshi Sakamoto. A succinct grammar compression. *Computing Research Repository*, abs/1304.0917, 2013.

[Tur84] György Turán. On the succinct representation of graphs. *Discrete Applied Mathematics*, 8(3):289–294, 1984.

[Wil98] Robin J. Wilson. Introduction to graph theory. *The Mathematical Gazette*, 82(494):343–344, 1998.