Karlsruher Institut für Technologie

Komprimierte Bitvektoren mit Rank und Select Anfrageunterstützung

Bachelorarbeit von

Tobias Paweletz

an der Fakultät für Informatik
Institut für Theoretische Informatik, Algorithm Engineering (ITI)

Erstgutachter:	Prof. Dr. rer. nat. Peter Sanders
Zweitgutachter:	Prof. Dr. Carsten Sinz
Betreuender Mitarbeiter:	Dr. rer. nat. Florian Kurpicz
Zweiter betreuender Mitarbeiter:	M.Sc. Hans-Peter Lehmann

14. November 2022 – 14. März 2023

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Änderungen entnommen wurde.

Karlsruhe, 14.03.2023

. . . . . .

(Tobias Paweletz)

Zusammenfassung

Rank und Select Datenstrukturen sind grundlegende Bausteine für andere komprimierte Datenstrukturen. Allerdings basieren diese meistens auf unkomprimierte Bitvektoren und benötigen deshalb meistens mindestens so viel Platz wie der unkomprimierte Bitvektor. Vor dieser Arbeit hat sich gezeigt, dass eine Huffman-codierte Eingabe den Platzbedarf des Bitvektors deutlich minimieren kann, allerdings hat dies auch zu einem deutlich erhöhten Zeitbedarf von Rank und Select Anfragen geführt.
Das Ziel dieser Arbeit ist es, die Huffman-codierte Eingabe zu indizieren, sodass besonders schnell auf einzelne Codewörter zugegriffen werden kann, damit die Anfragen beschleunigt werden. Die Indizierung des Huffman-Codes konnte das Verhältnis zwischen Anfragezeit zu Platzmehrbedarf stark verbessern, aber es existierten alternative Datenstrukturen, die über die Eingaben sowohl schneller als auch platzeffizienter waren. Allerdings existiert auch eine alternative Codierung, die in Kombination mit den implementierten Datenstrukturen den Platzbedarf so weit minimieren konnte, dass diese für manche Eingaben und v. a. für Select Anfragen pareto-optimal wurden.

Inhaltsverzeichnis

1 Einleitung

Rank und Select Datenstrukturen sind das Brot und Butter komprimierter Datenstrukturen. Diese werden häufig verwendet, um Baum bzw. Graph Codierungen [7, Kap. 8], Elias-Fano Codierungen [11] oder Permutationen [7, Kap. 5] zu codieren. Eine Rank Datenstruktur kann die Anzahl aller gesetzten Bits vor einem Index ausgeben und eine Select Datenstruktur kann die Position des i-ten gesetzten Bits ausgeben. Es existieren mehrere Datenstrukturen, die nur Rank oder Select Anfragen in konstanter Zeit lösen können [12] [10], allerdings benötigen diese viel Platz. Der Platzbedarf kann minimiert werden, indem ein passender Kompromiss zwischen Platz und Anfragezeit gewählt wird. Alternativ können Rank und Select Datenstrukturen so kombiniert werden, dass nur noch eine Datenstruktur gespeichert werden muss. Die aktuell schnellste Implementierung, die beide Möglichkeiten verwendet, ist *pasta-flat* in der SIMD Konfiguration, welche einen Platzmehrbedarf von weniger als 4 % benötigt [5].

Meine Arbeit beschäftigt sich damit, die Datenstrukturen von Pasta um die Anwendung auf komprimierte Bitvektoren zu erweitern. Dies kann durch unterschiedliche generische Datenstrukturen erreicht werden, die den komprimierten Bitvektor indizieren, wie mit den „Dense-Pointer“ [7, Kap. 3.2.2], „Sampled-Pointer“ [7, Kap. 3.2.1] oder Direct Access Codes [7, Kap. 3.4.2].

Zu Beginn wird in dem Kapitel 2 „Grundlagen“ über die benötigten grundlegenden Datenstrukturen und Codierungen aber auch über grundlegende Funktionsweisen der CPU aufgeklärt. Anschließend werden in dem Kapitel 3 „VLA Varianten und Platzanalyse“ Indizierungsmöglichkeiten von Codierungen sowie deren Platzanalysen behandelt. In dem Kapitel 4 „Implementierung“ wird erklärt, wie der Code optimiert und implementiert wurde. Schließlich werden in dem Kapitel 5 „Evaluation“ die Implementationen untereinander und mit anderen Codierungen verglichen. Abschließend wird in Kapitel 6 ein Fazit gezogen und Strategien, die in einer zukünftigen Arbeit evaluiert werden könnten, vorgestellt.

2 Grundlagen

2.1 Codierung für Zahlen unterschiedlicher Länge

Im der restlichen Arbeit wird strikt zwischen Codewörtern und Zahlen unterschieden. Eine Zahl ist nun ein Codewort für das gilt, dass beliebig viele weitere äquivalente Zahlen existieren, die sich nur durch die Anzahl an führenden 0-en unterscheiden. [1]
Es wird angenommen, dass $x > 0$ für jede Zahl x gilt. Diese Unterscheidung wird deshalb benötigt, da die Annahme aus Abschnitt 2.4, dass die führenden Nullen entfernt werden können, nicht zwingend gilt. Beispielsweise dürfen diese bei Codewörtern wie den Huffman-Codes [3] nicht entfernt werden.

> Beispiel: Falls 0101 eine (binäre) Zahl ist, gilt $101 \equiv 0101 \equiv 0 \cdots 0101$.
>
> Falls 0101 ein Codewort aber keine Zahl ist, so gilt mindestens $101 \not\equiv 0101$ oder $0101 \not\equiv 0 \cdots 0101$.

2.1.1 Unär Codierung

Die unär Codierung für eine Zahl x ist gegeben durch $u(x) = 0^{x-1} \cdot 1$.[7, kap. 2.7]
Hier ist $i \cdot j$ die Konkatenation von i und j und für $y > 0$ gilt $0^y = 0 \cdot 0^{y-1}$ für $y = 0$ gilt $0^0 = \epsilon$, wobei ϵ das leere Wort ist.

Diese ist vorteilhaft, falls man nur sehr kleine Zahlen codieren möchte, da $u(x)$ exakt x Bits Speicherplatz benötigt.

> Beispiel: $u(1) = 1, u(2) = 01, u(6) = 000001$

2.1.2 Elias-Gamma Codierung

Die Elias-Gamma Codierung einer Zahl x kann durch $\gamma(x) = u(|x|) \cdot [x]_{|x|-1}$ gebildet werden.[7, kap. 2.7] Hierbei ist $|x|$ die Länge der binären Darstellung (ohne führende Nullen) von x sowie $[x]_l$ gibt die l Bits der Zahl x mit dem niedrigsten Stellenwert zurück.

Diese Codierung benötigt exakt $2\lfloor \log_2(x) \rfloor + 1$ bzw. $2|x| - 1$ Bits.

[1] Auch wenn man formal von einer Darstellung einer Zahl sprechen würde, ist diese Definition intuitiv. So hat beispielsweise die Zahlendarstellung „9“ die gleiche Bedeutung wie die Zahlendarstellung „09“.

Beispiel: $\gamma(1) = 1, \gamma(2) = 01 \cdot 0, \gamma(3) = 01 \cdot 1, \gamma(6) = 001 \cdot 10$

2.1.3 Elias-Delta Codierung

Die Elias-Delta Codierung einer Zahl x kann durch $\delta(x) = \gamma(|x|) \cdot [x]_{|x|-1}$ gebildet werden.[7, kap 2.7] Es gilt, dass die Elias-Delta Codierung asymptotisch optimal ist, da der Platzmehrbedarf in $O(\log(|x|))$ liegt. Falls $|x| \geq 6$ gilt, ist die Elias-Delta Codierung kleiner als die Elias-Gamma Codierung.

Diese Codierung benötigt exakt $2\lfloor \log_2 \lfloor \log_2(x) \rfloor + 1 \rfloor + \lfloor \log(x) \rfloor + 1$ bzw. $2\lfloor \log_2(|x|) \rfloor + |x|$ Bits.

Beispiel: $\delta(1) = 1, \delta(2) = 010 \cdot 0, \delta(3) = 010 \cdot 1, \delta(6) = 011 \cdot 10$

2.2 Rank und Select Datenstrukturen

Rank und Select Datenstrukturen sind Datenstrukturen, die einen Bitvektor B der Länge $|B| \in \mathbb{N}$ mit den Werten $B[1], ..., B[|B|]$ um die $rank_x(i)$ bzw. $select_x(i)$ Funktionalität erweitern.

$rank_x(i)$ gibt die Anzahl der Vorkommen von x innerhalb des Bitvektors vor dem Index i aus. Es gilt also

$$rank_x(i) := \sum_{j=0}^{i} \mathbb{1}_{\{B[j]=x\}} \text{ für } x \in \{0,1\}, i \in [0, |B|].$$

Man kann zusätzlich auch die $rank_0(i)$ durch $rank_1(i)$ berechnen, mit $rank_0(i) = i - rank_1(i)$ $select_x(i)$ gibt den Index aus, an dem x zum i-ten mal vorkommt. Es gilt also

$$select_x(i) = \min\{j \in [0, |B|] \subseteq \mathbb{N} \mid rank_x(j) = i\} \text{ für } x \in \{0,1\}, i \in [0, rank_x(|B|)].$$

Gewissermaßen kann man $select_x(i)$ als Inverse von $rank_x(i)$ interpretieren, da $i = rank_x(select_x(i))$ gilt (allerdings nicht vice versa[2]).

Beispiel: Sei $B = [0100\ 1101\ 0011\ 1011]$ mit $|B| = 16$
So ist: $rank_1(0) = rank_1(1) = 0, rank_1(6) = 3, rank_1(16) = 9;$
$select_1(0) = 0, select_1(1) = 2, select_1(2) = 5, select_1(9) = 16$

[2] aufgrund der Injektivität von *select* und der Surjektivität von *rank*

2.3 Pasta-flat

Die in meiner Arbeit verwendete und erweiterte Datenstruktur ist *pasta-flat*. [5] Die ursprüngliche Datenstruktur arbeitet mit unkomprimierten Bitvektoren und fügt diesem 3 weitere Arrays hinzu, das Array L_0, das Array L_1 und das Array L_2.

Zunächst wird der Bitvektor in 2^{44} Bit Blöcke aufgeteilt. Für jeden dieser Blöcke wird die Anzahl der gesetzten Bits in einem L_0 gespeichert.
Anschließend wird jeder 2^{44} Bit Block in $2^{12} = 4096$ Bit Blöcke aufgeteilt. Die Anzahl der gesetzten Bits werden in L_1 gespeichert. Zuletzt wird jeder 2^{12} Bit Blöcke in 8 $2^9 = 512$ Bit Blöcke aufgeteilt, und in L_2 gespeichert.
Um $select_i(x)$ Anfragen effizient beantworten zu können, ist in einem weiteren Array jedes 8192-te gesetzte Bit abgespeichert.

$rank_i(x)$ Anfragen werden in konstanter und $select_i(x)$ in linearer (aber praktisch sehr schneller) Zeit unterstützt.

2.4 Variable Bit Arrays

Hier gilt als Annahme, dass Variable Bit Arrays ausschließlich Zahlen abspeichern können.[3] Diese Annahme wird sich allerdings später ändern.
Ein Variable Bit Array (VLA) ist ein Array, welches Zahlen mit variabler Bitlänge in diesem mit möglichst wenig Platzmehrbedarf abspeichert. Die Benutzung eines Variablen Bit Arrays ist insbesondere dann sinnvoll, wenn viele kurze Zahlen und wenig lange Zahlen abgespeichert werden.

Analog zu normalen Arrays implementieren VLA's die Funktion $access(i)$, welches die Zahl an Index i ausgibt.

2.4.1 Sampled-Pointer

Das Sampled-Pointer VLA mit einer Samplingrate k[7, kap. 3.2.1] besteht aus zwei Arrays, dem Code-Array [C] und dem Sample-Array [S].

Das Code-Array speichert die Codierungen der Zahlen aus der Eingabe. Für die Codierung muss gelten, dass man nur durch das bekannte Startbit des Codewortes die Zahl und das Ende des Codewortes (bzw. den Beginn des nachfolgenden Codewortes) extrahieren kann. Mögliche Codierungen, die diese Voraussetzung erfüllen, sind beispielsweise die oben beschrieben Elias-Gamma Codierung und die Elias-Delta Codierung.

Um das Sample-Array bestimmen zu können, müssen zunächst die Codewörter aus dem Codearray in $\frac{n}{k}$ Blöcke unterteilt werden. Jeder Block, außer der letzte, besteht aus exakt k Codewörtern. Das Sample-Array speichert für jeden Block das Startbit des ersten Codewortes.

Um auf ein Codewort an dem Index i zuzugreifen, muss nur das Startbit des $\frac{i}{k}$ Blockes durch das Sample-Array herausgefunden werden und anschließend über die nächsten $i \bmod k$ Codewörter gescannt werden.

[3] Siehe in Abschnitt 2.1 für die verwendete Definition von Zahlen.

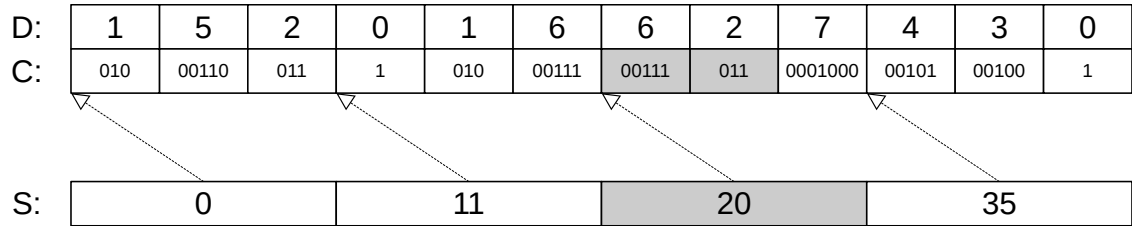


Abbildung 2.1: Sampled-Pointer VLA der Sequenz D, mit dem zugehörigen Code-Array C, welches Elias-Gamma codiert wurde und dem Sample-Array S. Graue Felder sind Felder auf die zugegriffen werden muss, wenn *access[7]* aufgerufen wird.

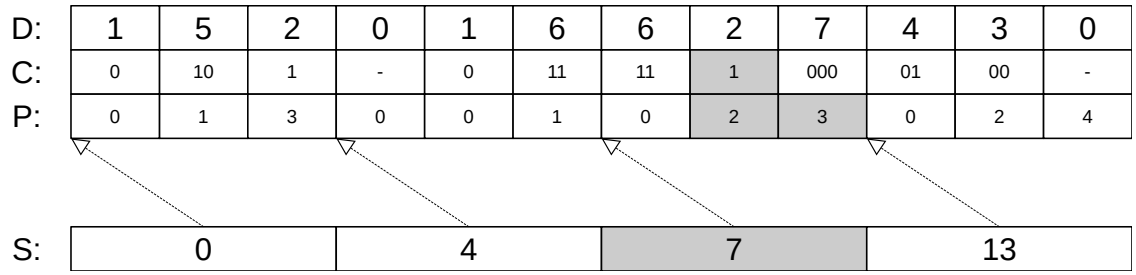


Abbildung 2.2: Dense-Pointer VLA der Sequenz D, mit dem zugehörigen Code-Array C, dem Pointer-Array P und dem Sample-Array S. Graue Felder sind Felder auf die zugegriffen werden muss, wenn *access[7]* aufgerufen wird. Für die Übersichtlichkeit, speichert der erste Wert eines Blockes in P eine 0.

2.4.2 Dense-Pointer

Das Dense-Pointer VLA mit der Samplingrate k[7, kap. 3.2.2] besteht aus drei Arrays, dem Code-Array [C], dem Pointer-Array [P] und dem Sample-Array [S].

Das Code-Array speichert für jede Zahl der Eingabe nur die Bits nach dem führenden 1 Bit der Zahl. Das führende 1 Bit muss deshalb nicht mit gespeichert werden, da jede Zahl eines besitzt und dies redundant ist.

Nun wird das Code-Array analog zum Sampled-Pointer VLA in $\frac{n}{k}$ Blöcke bestehend aus k Codewörtern aufgeteilt.
Das Pointer-Array speichert für jedes Codewort die Bitposition des Startbits abhängig von dem Startbit des zugehörigen Blockes.
Das Sample-Array speichert analog zum Sampled-Pointer VLA für jeden Block die Position des Startbits.

Um auf eine Zahl an Index i zuzugreifen, muss nur auf das Startbit des Blockes analog wie beim Sampled-Pointer bestimmt werden, anschließend bestimmt man den Offset und die Länge des Codewortes durch zwei Zugriffe auf das Pointer-Array an den Indices i und $i + 1$. Dies ist in Abbildung 2.2 zu sehen.

D:	1	5	2	0	1	6	6	2	7	4	3	0	
C:	1	11	00	0	1	000	000	00	001	10	01	0	
B:	1	10	10	1	1	100	100	10	100	10	10	1	1

Abbildung 2.3: Elias-Fano VLA der Sequenz D, mit der Zugehörigen Codierung C und dem Bitvektor der Select-Datenstruktur B. Graue Felder sind Felder auf die zugegriffen werden muss, wenn *access[7]* aufgerufen wird.

2.4.3 Elias-Fano

Das Elias-Fano VLA[7, kap. 3.4.3] darf man nicht mit der Elias-Fano Codierung[11], die streng monoton steigende Zahlenfolgen codiert, verwechseln.

Das Elias-Fano VLA besteht aus einer Select Datenstruktur und einem Code-Array [C].

Das Code-Array speichert die Zahlen analog wie das Code-Array des Dense-Pointer VLAs, nur mit dem Unterschied, dass vorher jede Zahl um 1 inkrementiert werden muss, da jede Zahl mindestens 2 Bit lang sein muss. Diese muss 2 Bit lang sein, da nur die Bits hinter dem führenden Bit abgespeichert werden.

Der Bitvektor [B], auf dem die Select Datenstruktur aufgebaut wird, hat nur an den Positionen ein 1 Bit, an denen ein Startbit eines Codewortes im Code-Array liegt, sowie ein 1 Bit am Ende.

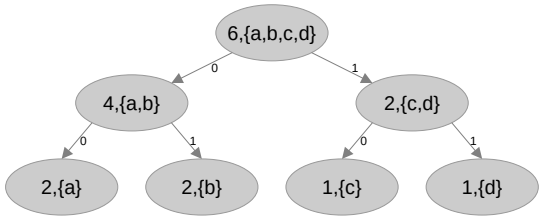
Um eine Zahl an Index i zu extrahieren, muss nur der Offset durch $select_1(i)$ und die Länge durch $select_1(i+1)$ bestimmt werden. Dies ist in Abbildung 2.3 zu sehen.

2.5 Huffman-Codierung

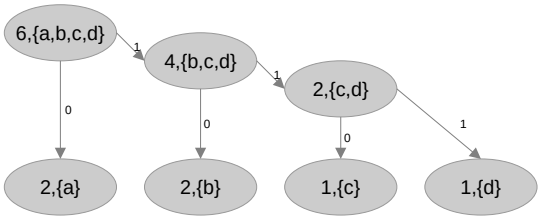
Eine Huffman-Codierung ist eine Codierung, die abhängig von einem Quellalphabet A, einem festen Text T und einer absoluten Häufigkeit, die durch $P_T(x) = \sum_{i=0}^{|T|-1}(\mathbb{1}_{\{T[i]=x\}})$ gegeben ist, eine binäre Codierung mit variabler Länge pro Codewort erstellen kann. Diese Codierung hat die Eigenschaft, dass sie präfixfrei ist. Das heißt, dass ein Codewort c_i kein Präfix eines anderen Codewortes c_j sein kann. Diese Eigenschaft bewirkt, dass für eine beliebige Konkatenation von präfixfreien Codewörtern aus einer Menge eine Umkehrabbildung von der Konkatenation auf die ursprüngliche Menge existiert.

Im Allgemeinem, aber für diese Arbeit nicht weiter relevant, kann man den Text T durch eine relativen Häufigkeit $p_A(x), x \in A$ ersetzen. Im schlechtesten, wenn auch unwahrscheinlichsten, Fall kann der erzeugte Code, durch einen speziell gewählten Text, ähnlich (lang) zu einer unärcodierung sein.[4] Dies kann ein Problem bereiten, denn die Codewortlänge eines Zeichens könnte so größer sein, als eine Computerwortlänge und man könnte in diesem Fall nur in linearer Zeit ein Codewort einlesen. Allerdings ist dies praktisch unmöglich erreichbar, da der Text länger als 16 Terabyte groß sein müsste, damit ein Zeichen nicht in ein 64 Bit Wort passt[7, kap. 2.6.2]. Ein Huffman-Baum/Codierung kann durch den Algorithmus 1 gebildet werden. Hierbei gilt für den Knoten $v = (v_0, v_1) \in V$,

[4] gdw. $P_T(x_i) \geq 2P_T(x_{i-1}), x_i \in A$ wobei x_i nach Größe der Wahrscheinlichkeit sortiert ist.



	Huffman-Codewort
a	00
b	01
c	10
d	11



	Huffman-Codewort
a	0
b	10
c	110
d	111

Abbildung 2.4: Mögliche (Kanonische) Huffman-Bäume sowie deren zugehörige Huffman-Tabellen für den Text: „aabbcd“

dass v_1 die Menge der von dem Knoten v abgedeckten Zeichen ist und v_0 die absolute Häufigkeit der Zeichen in v_1 innerhalb des Textes ist. Für die Kante $e = (e_0, (e_1, e_2)) \in E$ ist $e_0 \in \{0, 1\}$ das Kantenlabel und $(e_1, e_2) \in V \times V$ die Kante von e_1 nach e_2. Ein Code eines Zeichens kann z. B. durch eine Tiefensuche gefunden werden. Sei z. B. $[e^1, e^2, ..., e^n]$ mit $e^i = (e_0^i, (e_1^i, e_2^i)) \in E$ die Kantenliste eines Pfades von der Wurzel zu einem Blatt (also wenn $|v_1| = 1, (v_0, v_1) \in V$), so ist der zu dem Zeichen gehörige Huffman-Code $e_0^1 \cdot e_0^2 \cdot ... \cdot e_0^n$.

Algorithmus 1 Erstellung eines Huffman-Baumes [3]

Eingabe: $A = [x_0, ..., x_n], P_T(x_i)$
Ausgabe: $G = (V, E)$ ▷ G ist ein Baum mit Knotenmenge V und Kantenmenge E.

1: $V' = V = \{(P_T(x_i), \{x_i\}) \mid \forall i \in [0, n]\}, E = \{\}$
2: **while** $|V'| > 1$ **do**
3: $n = (n_0, n_1) \in \{(i_0, j_0) \in V' \mid \forall (i_1, j_1) \in V' : i_0 \leq i_1\}$
▷ extrahiere 1. Minimales Element aus V
4: $m = (m_0, m_1) \in \{(i_0, j_0) \in V' \setminus \{n\} | \forall (i_1, j_1) \in V' \setminus \{n\} : i_0 \leq i_1\}$
▷ extrahiere 2. Minimales Element aus V
5: $nm \leftarrow (n_0 + m_0, n_1 \cup m_1)$
6: $V' \leftarrow (V' \cup \{nm\}) \setminus \{m, n\}$ ▷ $|V'| \leftarrow |V'| - 1$
7: $V \leftarrow V \cup \{nm\}$
8: $E \leftarrow E \cup \{(0, (nm, n)), (1, (nm, m))\}$
9: **end while**
10: **return** G=(V,E)

Beispiel: In der Abbildung 2.4 sieht man mögliche Ausführungen des Algorithmus 1 auf dem Text „aabbcd“. Im ersten Schritt werden die Knoten $(1, \{c\})$, $(1, \{d\})$ verwendet, und der Knoten $(2, \{c, d\})$ gebildet.

Es fällt auf, dass der Algorithmus 1 im Allgemeinen nichtdeterministisch ist, da es mehrere Möglichkeiten gibt, n und m zu wählen, falls es mehrere minimale Elemente gibt.

Eine beliebte Form eines Huffman-Baumes, ist ein sogenannter Kanonischer Huffman-Baum. Diese hat die Eigenschaft, falls man die Huffman-Codeswörter als binäre Zahlen interpretiert, dass die Zahlen monoton steigend sind, für die nach der Häufigkeit sortierten Buchstaben. Es gilt also auch, dass die Tiefe der Blätter von „links nach rechts“ gelesen monoton steigend ist. Die Bäume in Abbildung 2.4 sind beide kanonisch.

2.6 Treap mit Lazy Propagation

Ein Treap ist eine Datenstruktur, welche sowohl die Eigenschaft eines binären Suchbaumes, als auch die Heap-Eigenschaft erfüllt. [9] Die Eigenschaft des binären Suchbaumes wird über die zu speichernden Werte gebildet und die Heap-Eigenschaft wird über Zufallswerte gebildet.

Im Prinzip kann jede Funktion, die ein Treap verändern soll, durch die *merge* und die *split* Funktion abgeleitet werden, welche eine erwartete Laufzeit von $O_{exp}(\log(n))$ aufweisen. Die *merge* Funktion nimmt als Eingabe zwei Treaps und bildet einen neuen. Es muss gelten, dass alle Werte der beiden Treaps sortiert sind und alle Werte des ersten Treaps kleiner als alle Elemente des zweiten Treaps sind.
Die *split* Funktion nimmt als Eingabe einen Treap und einen Wert x und gibt zwei Treaps aus. Dieser Treap besitzt alle Werte x_0, für die $x_0 \leq x$ gilt, und der andere besitzt alle Werte x_1, für die $x_1 > x$ gilt.

Ein Treap kann durch Lazy Propagation alle Werte in einem Intervall durch eine Update-Funktion ändern. Diese Update-Funktion sollte je nach Verwendungszweck spezielle Eigenschaften erfüllen, um die Laufzeit nicht zu verändern oder die Eigenschaft des binären Suchbaums zu zerstören. Lazy Propagation benötigt zusätzlichen Speicherplatz, denn jeder Knoten (inklusive der Blätter) in dem Treap benötigt einen zusätzlichen Wert. Aus diesem Wert müssen die Parameter der Funktion abgeleitet werden können.

Später wird ein Treap nicht als ein Suchbaum verwendet sondern als eine spezielle Form einer Liste. Diese Liste muss nur iteriert werden und Intervalle in der Liste verändert werden, was durch den Treap mit einer besseren erwarteten Laufzeitgarantie einhergeht.

2.7 Speicherhierarchie

Die für meine Arbeit wichtigen Speicherelemente sind die drei Cachestufen L1, L2 und L3, sowie der Hauptspeicher. Wie man in der Tabelle 2.1 sehen kann, wird die Latenzzeit mit Zunahme der Entfernung von der CPU immer größer, dafür nimmt allerdings auch der Speicherplatz immer weiter zu. Der Cache funktioniert so, dass bei einer Anfrage auf eine

Cacheart	Latenz	Speicherplatz
Register	«1ns	1KB
L1 Cache	1ns	48KB
L2 Cache	3ns	512KB
L3 Cache	10ns	8MB
RAM	74ns	16GB

Tabelle 2.1: Speicherhierarchie des Computers mit groben Latenzzeiten. [Hier: i7-1065G7 (Ice Lake), Cycles wurden umgerechnet in ns [4]].

Speicherzelle ein Cache antwortet, falls diese Speicherzelle in diesem enthalten ist. Man kann also davon ausgehen, falls eine Speicherzelle häufig verwendet wird, dass diese sich im Cache befindet und deshalb sehr effizient auf diese zugegriffen werden kann. Da die Latenzzeit teilweise sehr langsam ist, sollte dies so gut wie möglich wegoptimiert werden. Dies kann auf unterschiedliche Möglichkeiten erreicht werden.

2.7.1 Sequentielle Zugriffe

Auf die Daten sollte sequentiell zugegriffen werden, da hier die CPU das Anfragemuster erkennen kann und die Daten im Vorhinein in den Cache laden kann. [2, kap. 9.9]

2.7.2 Vorladen von Daten

Falls im Vorhinein bekannt ist, auf welche Adressen zugegriffen werden muss, kann man diese explizit mit „Prefetch“ Befehlen vorladen. Allerdings ist dies nur hilfreich, falls die Zugriffe keinen Mustern folgen. [2, kap. 9.11]

2.7.3 Aufteilung von Daten in Blöcke

Da der Cache in Blöcken organisiert ist, ist es sinnvoll Cachespeicherplatz zu sparen, indem die Daten auch in Blöcke aufgeteilt werden. Außerdem kann man zusätzliche Informationen in einen Block reinschreiben, die später benötigt werden. Diese Information werden dann automatisch mitgeladen, wenn der Block geladen wurde. Dies wurde in *pasta-flat* [5] verwendet, damit direkt das L2 Array mit dem L1 Array geladen wird.

2.8 Sprungvorhersager (Branch-Predictor)

CPUs arbeiten häufig mehrere Befehle parallel in einer Pipeline ab. Allerdings kann ein Programm auch bedingte Sprünge bzw. „if“ Statements beinhalten, was die parallele Abarbeitung der Befehle verhindert, da die CPU nicht im Vorhinein bestimmen kann, welcher Pfad genommen werden soll. Dieses Problem kann auf diverse Arten gelöst werden, z. B. könnten CPUs einfach solange warten, bis der Pfad bestimmt wurde. Allerdings sind die meisten Pfade so vorhersehbar, dass meistens eine andere Strategie verwendet wird. Hierfür wird von dem Branch-Predictor der Pfad abhängig von der Vergangenheit bestimmt

und die Befehle des bestimmten Pfades werden parallel mit ausgeführt. Dies hat den Nachteil, falls der Sprung falsch vorhergesagt wurde, dass alle vorher ausgeführten Befehle zurückgesetzt werden müssen, insbesondere die Befehle, die in Variablen geschrieben haben.

3 VLA Varianten und Platzanalyse

Im Grundlagenkapitel wurden VLAs für Zahlen als Eingabe behandelt. Nun sollen diese nicht nur Zahlen, sondern auch Codewörter als Eingabe akzeptieren können.
Diese Aufgabe ist trivial, da aus jedem Codewort c eine Zahl konstruiert werden kann, indem ein 1 Bit mit dem Codewort konkateniert wird. Diese Funktion ist konkret definiert durch $z(c) := 1 \cdot c$. Außerdem ist es für ein Huffman-Codewort auch nicht möglich dies besser zu lösen, da das vorderste Bit eines Codewortes sowohl eine 0 als auch eine 1 sein kann.

Da es in dieser Bachelorarbeit darum geht, Huffman-Codierungen in einem VLA abzuspeichern, kann der Platzbedarf weiter minimiert werden, da die Codewörter einer Huffman-Codierung immer präfixfrei sind.

3.1 Alternative nicht präfixfreie Codierung

Allerdings kann in den Erweiterungen, die in Unterabschnitt 3.2.1, Unterabschnitt 3.3.1 und Unterabschnitt 3.4.1 beschrieben werden, die Huffman-Codierung durch eine nicht präfixfreie Codierung ersetzt werden. Diese wird gebildet, indem zuerst die Wörter nach der Häufigkeit sortiert werden. Anschließend wird den Wörtern abhängig von der absoluten Häufigkeit ein Codewort zugewiesen. Das am häufigsten vorkommende Wort wird das leere Wort ϵ zugewiesen, dem zweithäufigsten die 0, usw. [7, kap. 4.1.2]

> Beispiel: Sei $[(8, a), (5, c), (3, b), (2, f), (1, o)]$ die sortierte Liste mit Wörtern, sei $g(x)$ die Funktion, die ein Wort x auf das zugehörige Codewort abbildet. So ist $g(a) = \epsilon$, $g(c) = 0$, $g(b) = 1$, $g(f) = 00$ und $g(o) = 01$.

3.2 Sampled-Pointer Erweiterungen

3.2.1 Triviale Erweiterung

Die triviale Erweiterung ist durch die Transformation der Codewörter in der Eingabe auf Zahlen durch $z(c)$ gegeben. Die Codewörter müssen nicht präfixfrei sein, da diese Erweiterung nicht diese Eigenschaft ausnutzt. Deshalb ist die in Abschnitt 3.1 beschriebene Codierung hierfür optimal. Nun kann der Platz abhängig der gewählten Codierung, also der Elias-Gamma oder der Elias-Delta Codierung bestimmt werden.

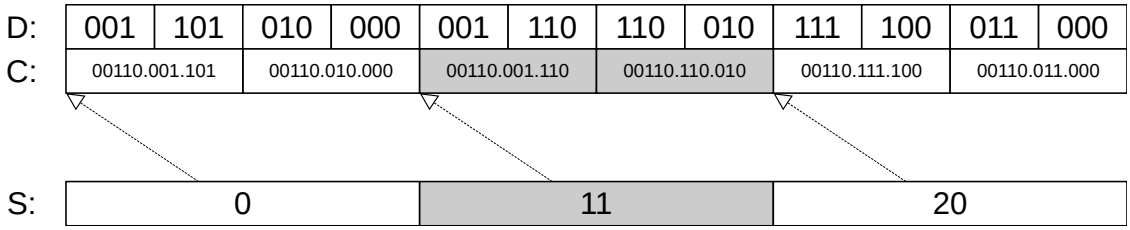


Abbildung 3.1: Sampled-Pointer VLA mit konkatenierten Codewörtern des präfixfreien Codes D und $k_1 = k_2 = 2$, mit dem zugehörigen Code-Array C, welches Elias-Gamma codiert wurde. Die Grenzen der Codewörter und dem Mehrbedarf der Elias-Gamma Codierung wurden mit Punkten kenntlich gemacht. S ist das Sample-Array. Graue Felder sind Felder auf die Zugegriffen werden muss, wenn *access[7]* aufgerufen wird.

3.2.1.1 Platzbedarf Sampled-Pointer mit Trivialer Erweiterung

Sei k die Samplingrate der Sampled-Pointer Codierung. So besitzt diese Struktur n/k Blöcke und es wird pro Block ein Bit-Pointer, welcher auf das Startbit des Blockes zeigt, benötigt. Die Länge in Bits, die ein Pointer benötigt, beträgt $\log(\sum_{i=0}^{n}(|c_i|))$.[1]

Der Platzbedarf des Code-Arrays ist abhängig von der gewählten Codierung. Sei $|c_{avg}|$ die mittlere Codelänge und n die Größe der Eingabe.
Falls das Code-Array Elias-Gamma codiert wurde, so konvergiert der Platzbedarf nach $n \cdot (2|c_{avg}| + 1)$ Bits.
Falls das Code-Array Elias-Delta Codiert ist, so konvergiert der Platzbedarf nach $2\lfloor \log_2(|c_{avg}|+1)\rfloor + 1 + |c_{avg}|$ Bits.

Falls $|c_{avg}|+1 \leq 2\lfloor(\log(|c_{avg}|+1)\rfloor+1$ gilt, gilt dass die Elias-Gamma Codierung höchstens gleichlang wie die Elias-Delta Codierung ist. Diese Ungleichung ist für $|c_{avg}| \leq 4$ erfüllt.

3.2.2 Erweiterung durch konkatenierte Codewörter

Zunächst wird die Samplingrate k in die Samplingraten k_1 und k_2 so aufgeteilt, dass $k = k_1 k_2$ gilt. Anschließend wird die Eingabe in $\frac{n}{k_2}$ Blöcke der Länge k_2 aufgeteilt und alle Codewörter in einem Block konkateniert. Da die Huffman-Codewörter präfixfrei sind, können die Codewörter in einem Block immer noch decodiert werden. Nun können die Blöcke der Länge k_2 in k_1 Blöcke zusammengefasst werden. Dies wird in Abbildung 3.1 dargestellt. Das Sample-Array speichert immer die Position des Startbits eines großen Blockes.

Diese Blöcke sollten am besten Elias-Delta codiert werden[2], da die Elias-Gamma Codierung keinen Platzvorteil liefert.

[1] Allerdings kann man auch eine hinreichend große Bitlänge, wie 64 Bit, bestimmen, da der Platzbedarf der Pointer mit $4 - 20$ % verhältnismäßig klein ist, und für große Eingaben ist dies höchstens doppelt so groß.
[2] Natürlich gibt es noch weitere sinnvolle Codierungen, diese werden hier allerdings nicht behandelt.

3.2.2.1 Platzbedarf Konkatenierte Codewörter

Sei $|c_{avg}|$ wieder die durchschnittliche Länge eines Codewortes.

So gilt, dass im Durchschnitt jeder Block $2\lfloor \log_2(k_2(|c_{avg}| + 1))\rfloor + 1 + (k_2|c_{avg}|) = 2\lfloor \log_2(|c_{avg}| + 1)\rfloor + 2\lfloor \log_2(k_2)\rfloor + 1 + (k_2|c_{avg}|)$ Bits benötigt.

Das wiederum bedeutet, dass der durchschnittliche, durch die Codierung verursachte, Platzmehrbedarf pro Codewort von $2\lfloor \log_2(|c_{avg}| + 1)\rfloor$ auf $\frac{2\lfloor \log_2(|c_{avg}|+1)\rfloor}{k_2} + \frac{2\lfloor \log_2(k_2)\rfloor}{k_2} \leq \frac{2\lfloor \log_2(|c_{avg}|+1)\rfloor}{k_2} + 1$ verbessert wurde.

3.3 Dense-Pointer Erweiterungen

Die Dense-Pointer VLAs können hauptsächlich durch Variationen des Pointer-Arrays erweitert werden. Zunächst muss entschieden werden, ob das Pointer-Array die Offsets oder die Längen der Wörter abspeichert.

Falls die Offsets abgespeichert werden, kann der Offset des ersten Wortes in einem Block kostenlos abgespeichert werden, da dieser immer 0 ist und benötigt somit weniger Platz, allerdings muss auf das Sample-Array zugegriffen werden, falls das letzte Codewort eines Blockes decodiert werden soll.

Die Variation kann entweder eine Array-artige Abspeicherung beibehalten und nur die Codierung der Zahlen verändern oder eine andere Datenstruktur verwenden, um die Offsets bzw. Längen abzuspeichern.

3.3.1 Triviale Erweiterung

Die triviale Erweiterung muss analog zu der trivialen Erweiterung der Sampled-Pointer die einzelnen Codewörter der Eingabe in Zahlen transformieren. Diese Codierung nutzt auch nicht die Eigenschaft präfixfreier Codewörter aus. Angenommen, es existieren genau m unterschiedliche Codewörter und es wird mit der in Abschnitt 3.1 beschriebenen Codierung codiert, so gilt, dass die maximale Codewortlänge $|c_{max}| = \log(m)$ ist. Hiermit kann man in der Platzanalyse in dem Unterunterabschnitt 3.3.2.1, die hier aufgrund der Redundanz weggelassen wird, stärkere Annahmen über den Platzbedarf des Pointer-Arrays treffen. Für Huffman-Codes gilt dies nicht, hier könnte die maximale Codewortlänge bis zu $|c_{max}| = m - 1$ Bits benötigen.

3.3.2 Erweiterung durch alternative Array-Codierung

Die Längen-Codierung speichert für jedes Codewort c_i die Länge des Codewortes $|c_i|$ in dem Pointer-Array an Index i. Es gilt also $P1[i] = |c_i|$

Die (relative) Offset-Codierung speichert für jedes Codewort c_i den Offset relativ zu dem vorhergehenden Codewort c_{i-1} in dem Pointer-Array an Index i. Dieser Offset ist für das erste Codewort eines Blockes immer 0. Dies ist äquivalent dazu, dass das Codewort c_i die Länge des vorhergehenden Codewortes speichert.
Es gilt $P1'[i] = |c_{i-1}|$, gdw. $i \mod k \neq 0$.

Die Laufzeit von *access* ist für die Offset-Codierung und die Länge-Codierung linear abhängig von der Blockgröße, da alle Werte bis zu dem gesuchten Index aufsummiert werden müssen.

Es kann entweder über die Offset-Codierung oder die Längen-Codierung ein binärer Suchbaum aufgebaut werden, der als ein Array codiert ist. Dieser kann aufgebaut werden, indem für ein Index i innerhalb eines Blockes die größte Zweierpotenz 2^x die i teilt, gesucht wird. Der Block speichert an Index i die Summe der $2^x - 1$ vorhergehenden Werte. Sei k wieder die Samplingrate, so gilt $P2[i] = \sum_{x=1+i-\max\{y\in\mathbb{N}|(i \mod k) \mod 2^y=1\}}^{i} P1[x]$.
Eine *access* Anfrage auf den Suchbaum funktioniert analog wie eine Intervall-Anfrage auf ein Segment-Baum (Segment Tree [1]). Das heißt, dass für einen Index i die Summe aller Werte von 0 bis i so berechnet werden kann, indem i auf die Summe $i = \sum_{j\in M} j$, wobei jedes $j \in M$ eine Zweierpotenz ist, aufgeteilt wird. Nun gilt $\sum_{j=0}^{i}(P1[j]) = \sum_{j\in M} P2[j]$.
Die Laufzeit von *access* ist logarithmisch über die Blockgröße, da
$|M| = \lceil \log(i) \rceil \in O(\log(k))$.[3] Durch den Beweis über die Platzanalyse in Unterunterabschnitt 3.3.2.1 fällt auf, dass angenommen wird, dass dieses Array ein VLA ist. Allerdings kann der Offset und die Länge einer Zelle des Arrays durch eine Lookup Tabelle oder durch eine Formel bestimmt werden. Es kann bewiesen werden, dass der Offset der Zelle an Index i durch die Formel $i(|c_{max}| + 1) - popcount(i)$ und die Länge der Zelle an Index i durch die Formel $|c_{max}| + ctz(i)$, wobei $|c_{max}|$ die Länge des größten Codewortes ist, $popcount(i)$ die 1 Bits von i zählt und $ctz(i)$ die unteren 0 Bits von i zählt. Dies folgt aus der Platzanalyse in Unterunterabschnitt 3.3.2.1 und daraus, dass ein so codiertes Array der Länge 2^j exakt $2^j - 1$ Bits zusätzlich benötigt. [4]

Außerdem kann entweder über die Offset-Codierung oder die Längen-Codierung der Offset bzw. die Länge relativ zum Startbit des Blockes abgespeichert werden. Dies ist über eine Präfixsumme über den Block berechenbar. Es gilt also $P3[i] = \sum_{x=i-(i \mod k)}^{i} P1[x]$.
Die Laufzeit von *access* ist nun konstant.
Eine Darstellung der sechs unterschiedlichen Codierungen ist in Abbildung 3.2 gegeben.

3.3.2.1 Platzanalyse

Da die Platzanalyse für die Offset-Codierung und den darauf basierenden Codierungen analog ist zu der Platzanalyse der Längen-Codierung und den darauf basierenden Codierungen, wird hier nur die Platzanalyse der Längen-Codierung und den darauf basierenden Codierungen behandelt.

Sei $|c_{max}| := \max_{0\le i<N}(|c_i|)$ die Länge des längsten Codewortes, wobei N die Anzahl der gespeicherten Codewörter ist, so benötigt die Längen-Codierung exakt $N(\log_2(|c_{max}|)+1)$ Bits, die Längen-Codierung als binärer Suchbaum höchstens $N(\log_2(|c_{max}|)+2)$ Bits und als Präfixsumme über die Längen-Codierung höchstens $N(\log_2(|c_{max}|) + 1 + \frac{1}{k}\sum_{i=0}^{\lfloor \log_2(k)\rfloor}(i2^i))$ Bits.

[3] Diese Codierung kann auch als ein Suchbaum verwendet werden, falls der Index für eine Präfixsumme bestimmt werden soll. Dies könnte für Select Anfragen sinnvoll sein.

[4] Anm: Mit dieser Codierung könnte man eine Rank und Select Datenstruktur generieren, die beide Funktionen in $O(\log(n))$ unterstützt und exakt $2n - O(\log(n))$ Bits benötigt.

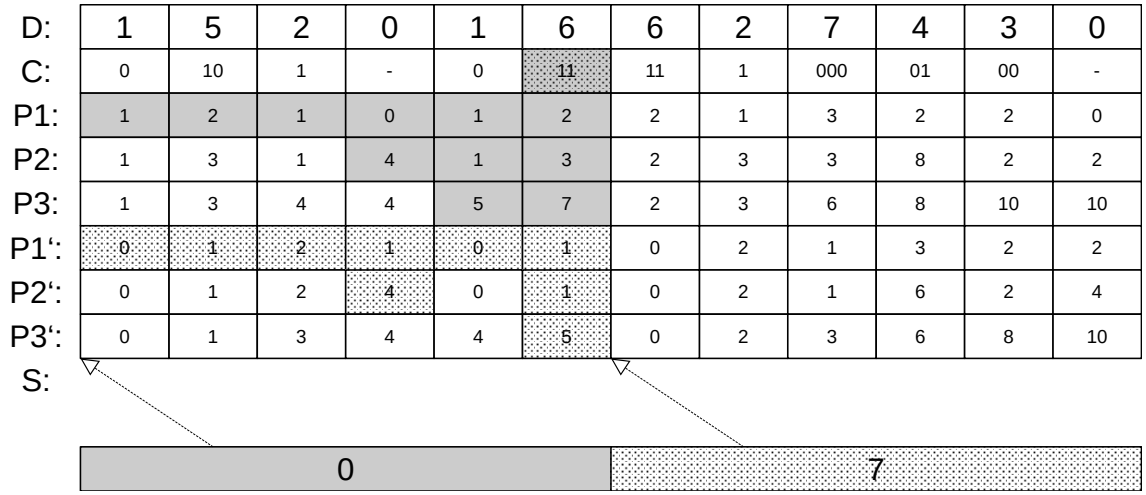


Abbildung 3.2: Dense-Pointer VLA mit den alternativen Codierungen des Pointer-Arrays mit der Eingabe D und dem zugehörigen Code-Array C. S ist das Sample-Array. $P1$ speichert das Längen-Codierung, $P2$ speichert den binären Suchbaum über die Längen-Codierung, $P3$ speichert die Präfixsumme über die Längen-Codierung, $P1'$ speichert die Offset-Codierung, $P2'$ speichert den binären Suchbaum über die Offset-Codierung und $P3'$ speichert die Präfixsumme über die Offset-Codierung. Graue bzw. gemusterte Felder sind Felder auf die Zugegriffen werden muss, wenn *access[7]* aufgerufen wird.

Die Größe des Pointer-Arrays mit Längen-Codierung kann hergeleitet werden dadurch, dass das Pointer-Array exakt N Zahlen speichert und für jede Zahl gilt, dass diese zwischen 0 und $|c_{max}|$ liegt. Also benötigt jede Speicherzelle in dem Pointer-Array $\log_2(|c_{max}|) + 1$ Bits.

Die Größe des Pointer-Arrays mit dem Suchbaum über die Längen-Codierung zu bestimmen, kann durch die Größe des Pointer-Arrays mit Längen-Codierung hergeleitet werden. Da für den Index i, welcher durch 2^x für ein $x \in \mathbb{N}$ teilbar ist, mindestens die Summe der $2^x - 1$ vorhergehenden Werte speichert, gilt, dass die Speicherzelle mindestens $\log_2(2^x\ |c_{max}|)+1 = \log_2(2^x)+\log_2(|c_{max}|)+1 = x+\log_2(|c_{max}|)+1$ Bit benötigt. Außerdem gilt, falls dieser Index durch 2^x, für $x > 0$, teilbar ist, dass dieser auch durch 2^{x-1} teilbar ist und ein Block der Länge k höchstens $\frac{k}{2^x}$ Indizes enthält, die durch 2^x teilbar sind. Hieraus folgt direkt die Ungleichung

$$k(\log(|c_{max}|) + 1) + \sum_{i=1}^{\log(k)} (\frac{k}{2^i}) \leq k(\log(|c_{max}|) + 1) + (k \sum_{i=1}^{\infty} (\frac{1}{2^i})) = k(\log(|c_{max}|) + 1) + k,$$

welche den Speicherplatz in Bits für einen Block abschätzt. Das Pointer-Array besteht aus $\frac{N}{k}$ Blöcken, also benötigt das Pointer-Array $\frac{N}{k}(k(\log(|c_{max}|) + 1) + k) = N(\log(|c_{max}|) + 2)$ Bits.

Die Größe der Präfixsumme über die Längen-Codierung folgt direkt dadurch, dass für einen Index i innerhalb das Blockes der Länge k der gespeicherte Wert höchstens $i|c_{max}|$

groß ist. Also benötigt die Speicherzelle $\lfloor \log_2(i|c_{max}|)\rfloor + 1 = \lfloor \log_2(i)\rfloor + \lfloor (|c_{max}|)\rfloor + 1 = \lfloor \log_2(2^{\lfloor \log_2(i)\rfloor})\rfloor + \lfloor (|c_{max}|)\rfloor + 1$ Bits.

Man kann sehen, dass alle Indices zwischen $2^{\lfloor \log_2(i)\rfloor}$ und $2^{\lfloor \log_2(i)+1\rfloor} - 1$ gleich viel Speicherplatz benötigen. Hieraus folgt direkt der Platzbedarf von $N(\log_2(|c_{max}|) + 1 + \frac{1}{k}\sum_{i=0}^{\lfloor \log_2(k)\rfloor}(i2^i))$ Bits über das gesamte Pointer-Array.

Falls man das Pointer-Array Offset-codiert abspeichert, so kann man mit jeder Codierung mindestens $\frac{N}{k}|c_{max}|$ Bit sparen. Dies ist möglich, weil das erste Element eines Blockes in dem Pointer-Array dann immer 0 ist.

3.3.3 Erweiterung durch alternative Datenstruktur

Sei k wieder die Samplingrate und N die Größe der Eingabe. So kann jeder Block in dem Pointer-Array durch eine Elias-Fano Codierung[11] ausgetauscht werden. Dies ist möglich, da jeder Block eine streng monoton steigende Zahlenfolge codiert.

3.3.3.1 Platzanalyse

Im Allgemeinen gilt, dass eine Elias-Fano Codierung $2n + n\lceil \log_2(\frac{u}{n})\rceil$ Bits benötigt, wobei n die Anzahl der Elemente in der Sequenz ist und u das größte Element in der Sequenz ist.

Sei $|c_{min}|$ die minimale Codewortlänge einer Huffman-Codierung. Nun kann die untere Schranke des Platzbedarfs bestimmt werden, da die Elias-Fano Codierung eine Sequenz aus k Elementen speichert, in der das größte Element mindestens $k|c_{min}|$ groß ist.

Also ist die Elias-Fano Codierung eines Blockes insgesamt mindestens
$2k + k\lceil \log_2(\frac{k|c_{min}|}{k})\rceil = 2k + k\lceil \log_2(|c_{min}|)\rceil = k(2 + \lceil \log_2(|c_{min}|)\rceil)$ Bit lang.

Daraus folgt, falls $|c_{min}|+2 < |c_{max}|$ gilt, dass die Elias-Fano Codierung um einen linearen Faktor platzeffizienter sein kann. Allerdings hat die Elias-Fano Codierung auch einen relativ hohen konstanten Platzverbrauch, für kleine k, da einige zusätzliche Informationen gespeichert werden müssen z. B. Pointer.

Für größer werdende k konvergiert der Platzbedarf gegen $k(2 + \lceil \log_2(|c_{avg}|)\rceil)$ Bit pro Block, wobei $|c_{avg}|$ die durchschnittliche Codewortlänge ist, da die durchschnittliche Codewortlänge der Wörter in einem Block nach $|c_{avg}|$ konvergiert.

3.4 Elias-Fano Erweiterungen

3.4.1 Triviale Erweiterung

Die triviale Erweiterung transformiert analog zu der trivialen Erweiterung in den Sampled-Pointer VLA die einzelnen Codewörter in Zahlen.

3.4.1.1 Platzanalyse

Die Platzanalyse ist analog zu der Sampled-Pointer VLA mit Elias-Gamma Codierung. Denn der zusätzliche Speicherplatz der Elias-Gamma Codierung wird in einem externen Bitvektor ausgelagert. Der Unterchied ist, dass der Bitvektor pro Codewort noch ein Bit mehr benötigt.

3.4.2 Erweiterung durch Huffman-Codierung

Der Bitvektor des Elias-Fano VLA kann Huffman-codiert werden, um den zusätzlich benötigten Speicherplatz zu minimieren. Falls man diesen Bitvektor naiv Huffman-codiert kann man wenige Aussagen über den Platzbedarf treffen und im schlechtesten Fall könnte dieser inklusive der Huffman-Tabelle sogar größer sein, als der ursprüngliche Bitvektor.

Zunächst wird eine Samplingrate k eingeführt und die Codewörter in Blöcke der Größe k aufgeteilt. Außerdem sei $|c_{min}|$ die Länge des minimalen Codewortes. Nun hat der Bitvektor eine 1 am Start eines Blockes und nicht am Start eines Codewortes.

Anschließend wird der Bitvektor als ein Array (Längen-Array [L]) an Codewörtern der Länge $|c_{min}|$ uminterpretiert und dieses Array wird Huffman-codiert.

Diese Aufteilung hat den Vorteil, dass es exakt $|c_{min}| + 1$ unterschiedliche Codewörter auftreten können, denn ein Codewort c_i kann entweder 0 sein oder 2^i für ein $i \in [0, |c_{min}|)$. Man kann davon ausgehen, dass für zwei verschiedene Codewörter $x \neq 0$, $y \neq 0$ $\mathbb{P}(x) \approx \mathbb{P}(y)$ gilt. Für $k \geq 2$ und ein Codewort a gilt $\mathbb{P}(a = 0) \geq \mathbb{P}(a \neq 0)$, da $P(a = 0) \geq \frac{1}{2}$ gilt. Dies gilt, da auf jedes Codewort ungleich 0 mindestens ein Codewort gleich 0 folgt.

Hieraus kann für $k \geq 2$ folgende Huffman-Codierung für ein Codewort c hergeleitet werden:

$$c \mapsto \begin{cases} 1 \cdot [\log_2(c)]_{\log(|c_{min}|)}, & \text{falls } x \neq 0 \\ 0, & \text{sonst} \end{cases}$$

Für $k = 1$ ist die Codierung $c \mapsto [\log_2(c)]_{\log(|c_{min}|)}$, wobei $[a]_b$ die Zahl a in einem Wort der Länge b speichert. Beispielsweise ist $[3]_5$ gleich 00011. Man kann sehen, dass dieses Konstrukt sowohl präfixfrei ist, als auch die restlichen Eigenschaften der Huffman-Codierung erfüllt. Die Eigenschaft, dass dies eine Huffman-Codierung ist, macht die Platzanalyse in Unterunterabschnitt 3.4.2.1 leicht.

Nun wird eine äquivalente Codierung gebildet. Hierfür benötigt man einen weiteren Bitvektor B' und ein Längen-Array L'. Der Bitvektor B' speichert genau dann eine 1, wenn das zugehörige Codewort ungleich „$0 \cdots 0$“ ist, sonst speichert dieser eine 1. Da jeder Block exakt eine 1 besitzt und diese immer innerhalb des ersten Codewortes liegt, benötigt das Längen-Array L' für jeden Block genau einen Eintrag. Dieses Array speichert den Offset der 1 innerhalb des ersten Codewortes und da es genau $|c_{min}|$ unterschiedliche mögliche Offsets existieren, benötigt jede Zelle exakt $\log_2(|c_{min}|)$ Bits.

Der Offset des i-ten Blocks kann durch $|c_{min}|select_1(i) + (|c_{min}| - L'(i))$ bestimmt werden. Diese Codierung ist äquivalent zu der oben genannten Huffman-Codierung, da B' das erste Bit aller Codewörter speichert und die in L' gespeicherten Offsets äquivalent zu $[\log_2(c)]_{\log(|c_{min}|)}$ sind.

In Abbildung 3.3 wird diese Codierung dargestellt.

3.4.2.1 Platzanalyse

Für eine Samplingrate $k \geq 2$ gilt für jeden Block, dass dieser mindestens $k - 1$ Codewörter besitzt die 0 sind, und ein Codewort, das ungleich 0 ist. Jedes Codewort welches 0 ist, wird um exakt $\frac{1}{|c_{min}|}$ und jedes Codewort ungleich 0 wird um $\frac{\log_2(|c_{min}|)+2}{|c_{min}|}$ komprimiert. Hieraus folgt, dass die Huffman-Codierung höchstens $|L| + \frac{|L|\log_2(|c_{min}|)+2}{k}$, für $|c_{min}| \geq 2$, Bits

D:	3	5	2	4	7	5	6	2	7	4	3	6
C:	01	11	00	10	001	11	000	00	001	10	01	000
B:	10	00	00	10	000	00	100	00	000	10	00	000
L:	10	00	00	10	00	00	01	00	00	00	01	00	00	00
L':	1			1			0			0		
B':	1	0	0	1	0	0	1	0	0	0	1	0	0	0

Abbildung 3.3: Elias-Fano VLA mit Huffman-Codierung mit der Samplingrate $k = 3$. Hier ist $|c_{min}| = 2$. Graue Felder sind Felder auf die zugegriffen werden muss, wenn *access[7]* aufgerufen wird.

benötigt, wobei $|L| = \frac{|B|}{|c_{min}|}$ die Länge des Längen-Arrays ist. Insgesamt ist die Benutzung dieser Strategie sinnvoll, falls ein hinreichend großes $|c_{min}|$ vorliegt. Falls $|c_{avg}| \geq 2|c_{min}|$ gilt, könnte sogar $k = 1$ sinnvoll sein. Man darf dies nicht mit Längen-Codierung von Dense-Pointer verwechseln, da der Platzmehrbedarf dieser Codierung asymptotisch gleich groß ist wie der Bitvektor B.

4 Implementierung

4.1 Codewort Extraktion

In dieser Arbeit werden Algorithmen implementiert, die branchless Wörter mit einer variablen Bitlänge aus einem Bitarray extrahieren sowie Elias-Gamma codierte Wörter extrahieren. In beiden Fällen darf die Länge des Wortes 64 Bit nicht überschreiten. Das Bitarray wird immer als ein Array aus 64 Bit Wörtern gespeichert. Zusätzlich benötigt man einen Bit-Pointer, welcher ein Startbit codiert, da normale Pointer nur auf Byte-Ebene funktionieren. Dieser besteht aus einem Offset, um das Bit in dem 64 Bit Wort zu finden, und dem Index für das richtige 64 Bit Wort. Da 2^{64} Bit ungefähr zwei Millionen Terabyte entspricht und jede Eingabe deutlich kleiner ist, ist es hinreichend, die Bit-Pointer als eine 64 Bit Zahl zu implementieren. Hierfür wird der Offset in den letzten 6 Bit und der Index des 64 Bit Wortes in den vorderen 58 Bit abgespeichert. Der Index kann über einen Rechtsshift um 6 und der Offset durch ein bitweises Und um $0x3f$ bestimmt werden. Da diese beiden Funktionen essentiell sind und sehr häufig aufgerufen werden, wurden diese so stark wie möglich zu beschleunigen.

4.1.1 Wort Extraktion

Die Herausforderung die Wort-Extraktion branchless zu gestalten liegt dabei, dass ein Wort, in höchstens zwei 64 Bit Maschinen-Wörtern aufgeteilt, gespeichert ist. Der Trick hierbei ist es, beide 64 Bit Maschinen-Wörter in ein 64 Bit Maschinen-Wort so zu verschmelzen, dass die beiden Teilwörter kontinuierlich abgespeichert sind und das Wort an dem obersten Bit des Maschinen-Wortes beginnt. Der verwendete Algorithmus folgt direkt aus der Annahme, dass ein Wort immer mindestens bis zu der rechten Grenze des ersten Maschinen-Wortes reicht. Auch wenn diese Annahme fast immer falsch ist, ist diese sinnvoll, da in dem falschen Fall nur unbenötigte Bits verändert werden. Zuletzt müssen die unteren unbenötigten Bits mit einem Rechtsshift ausmaskiert werden. Nun ist das untere Bit des Codewortes am unteren Bit des Maschinen-Wortes ausgerichtet und die Extraktion ist abgeschlossen. Dies ist in 2 beschrieben.

Algorithmus 2 Codewort Extraktion

Eingabe: $M_1, M_2, word\_length, offset$ ▷ M_i sind Maschinen-Wörter
Ausgabe: $word$

1: $word \leftarrow (M_1 \operatorname{lsh} offset) \operatorname{or} (M_2 \operatorname{rsh}(64 - offset))$
▷ „lsh“ ist Linksshift, „rsh“ ist Rechtsshift, „or“ ist bitweises-oder.
2: $word \leftarrow word \operatorname{rsh}(64 - word\_length)$
3: **return** $word$

Beispiel: Seien nun c_1 bzw. c_2 die Bits des zu extrahierenden Wortes, x_1 bzw. x_2 die unwichtigen Bits und w_1 bzw. w_2 die Wörter des Bitarrays. Die Wortlänge ist in diesem Beispiel aufgrund der Übersichtlichkeit 6 Bit.
Beispiel mit Wort nur in w_1

1. $w_1 = [x_1, x_1, c_1, c_1, x_1, x_1]; w_2 = [x_2, x_2, x_2, x_2, x_2, x_2]$
2. $w_1 \leftarrow [c_1, c_1, x_1, x_1, x_2, x_2]$
3. $w_1 \leftarrow [0, 0, 0, 0, c_1, c_1]$

Beispiel mit Wort in w_1 und w_2

1. $w_1 = [x_1, x_1, x_1, c_1, c_1, c_1]; w_2 = [c_2, c_2, x_2, x_2, x_2, x_2]$
2. $w_1 \leftarrow [c_1, c_1, c_1, c_2, c_2, x_2]$
3. $w_1 \leftarrow [0, c_1, c_1, c_1, c_2, c_2]$

4.1.2 Elias-Gamma Extraktion

Eine Elias-Gamma Extraktion funktioniert analog zu der normalen Wortextraktion, nur mit dem Unterschied, das man zwei Wörter extrahieren muss und die Länge des zweiten Wortes durch das erste Wort bestimmt wird und die Länge des ersten Wortes unbestimmt ist.

Der Konkatenierungsschritt bleibt hier identisch, da nach diesem Schritt die obersten benötigten Bits direkt am Wortanfang stehen. Die Wortlänge kann man nun durch das Zählen der obersten 0-Bits herausfinden. Dies lässt sich in C++ 20 durch std: : countl_zero durch eine einzelne Hardware-Instruction lösen. [1] Interressant hierbei ist, dass die branchless Variante langsamer ist, da das zweite Wort in drei 64 Bit Wörtern liegen kann und die Transformation von drei 64 Bit Wörtern in ein 64 Bit Wort arbeitsintensiv ist. Deshalb wurde ein Branch hinzugefügt, der überprüft, ob das Wort höchstens 32 Bit lang ist, da hier das Wort nur in den ersten beiden 64 Bit Wörtern liegt. Dieser Branch macht zusätzlich Sinn, da dieser bei Huffman-Codewörter sehr gut vorhergesagt werden kann, denn entweder sind alle Wörter länger als 32 Bit oder die meisten Wörter sind kürzer als 32 Bit.

[1] Falls man über die LZCNT Instruction verfügen kann und hiermit compiliert.

4.2 Huffman-Tabelle

4.2.1 Erzeugung der Huffman-Tabelle

Um einen kanonischen Huffman-Baum bestimmen zu können, wird eine Liste benötigt, die für jedes Wort die Länge des codierten Wortes abspeichert. Diese Liste wurde durch einen Treap mit Lazy Propagation ersetzt. Durch dieses Ersetzen, kann nicht nur die Allokationen auf eine konstante Anzahl reduziert werden, sondern auch die asymptotische Laufzeit von $O(n^2)$ auf $O_{exp}(n\log(n))$ verbessert werden. Die Laufzeit liegt in $O_{exp}(n\log(n))$, da eine Liste nicht schneller als in $O(n\log(n))$ sortiert werden kann, ein Treap in $O_{exp}(\log(n))$ verschmolzen werden kann sowie aufgrund der Lazy Propagation alle Elemente in einem Treap in konstanter Laufzeit inkrementiert werden können. Die Laufzeit der Sortierung ist wichtig, da die Priority-Queue in $O(n)$ initialisiert wird und deshalb die Extraktion des kleinsten Elementes niemals schneller als in $\Omega(\log(n))$ möglich ist, da ansonsten die untere Schranke der Sortieralgorithmen gebrochen würde. Der Algorithmus baut zu Beginn für jedes Zeichen einen Baum und eine Priority Queue, welche die Auftrittswahrscheinlichkeit und einen Pointer auf einen Treap eines Zeichens speichert. Die Liste ist nach der Auftrittswahrscheinlichkeit sortiert.

Algorithmus 3 Bestimmung der Länge der Codewörter

Eingabe: $pq = [(freq_0, t\_ptr_0), ..., (freq_n, t\_ptr_n)]$
▷ $freq_x$ ist Auftrittswahrscheinlichkeit, t_ptr_x ist Pointer auf zugehörigen treap
Eingabe: $treaps = [t_0, t_1, ..., t_n]$
▷ Jedes t_i ist ein Baum bestehend aus einem Knoten und null Kanten.
Ausgabe: $Treap$
▷ speichert die Struktur eines Huffman-Baumes sowie die Länge aller Codewörter

1: **while** $|pq| > 1$ **do**
▷ $O(n)$ Iterationen, da in jeder Iteration exakt ein Element gelöscht wird
2: $(freq_0, t_0) \leftarrow pq.pop(), (freq_1, t_1) \leftarrow pq.pop()$ ▷ $O(\log(n))$
3: $t_3 = t_1.merge(t_2)$ ▷ $O_{exp}(\log(n))$ und keine Allokation
4: **for all** $x \in t_3$ **do** ▷ insgesamt in $O(1)$ durch Ausnutzen der Struktur
5: $x.length+ = 1$
6: **end for**
7: $pq.push(freq_0 + freq_1, t_3)$
8: **end while**
9: $(freq_{res}, t_{res}) \leftarrow qp.pop()$
10: **return** t_{res}

4.2.2 Abspeicherung der Huffman-Tabelle

Es ist möglich, die Huffmantabelle zum Decodieren in einer Lookup-Tabelle abzuspeichern. Das Komplizierte hierbei ist allerdings die Berechnung des korrekten Indizes durch die Funktion f.

Es muss hierbei gelten: $table[f(c_w)] = w$, wobei *table* ein Array der uncodierten Wörter ist, c_w das codierte Wort und w das uncodierte Wort ist. Es macht zusätzlich Sinn, *table* als ein Array von Blöcken zu betrachten, wobei in jedem Block B alle Codewörter einer Länge $|c_w|$ abgespeichert werden. Nun muss $table[|c_w|][f'(c_w)] = w$ gelten, wobei $table[|c_w|]$ der Block aller Codewörter der Länge $|c_w|$ ist. Da in einer kanonischen Huffman-Codierung alle Codewörter einer Länge eine aufsteigende Folge bilden, also $c_i + 1 = c_{i+1}$ gilt, muss man nur das minimale Codewort aller Codewörter einer Länge zusätzlich abspeichern, um ein Codewort zu decodieren. Die finale Funktion lautet also $table[|c_w|][c_w - \min(c_B)] = w$. Um Cachemisses in dem L2 bzw. L3 Cache zu minimieren, wurden alle Blöcke sowie die Lookuptabelle, die $\min(c_B)$ und $table[|c_w|]$ speichert, in einem zusammenhängendem Speicherbereich abgespeichert.

Im schlechtesten Fall, in dem jedes Codewort in dem Text exakt einmal vorkommt, kann die Huffman-Tabelle bis auf einen konstanten Faktor gleich groß wie der Text sein.

4.3 Sampled-Pointer

Die Sampled-Pointer sind analog wie in Unterabschnitt 3.2.1 implementiert worden. Die Bit-Pointer in dem Sample-Array sind jeweils 64 Bit-Pointer und das zu Grunde liegende Code-Array wurde Elias-Gamma Codiert.

4.4 Dense-Pointer

Es wurde die Dense-Pointer Strategie, wie in Unterabschnitt 3.3.2 beschrieben, implementiert, das zu jedem Codewort die Länge des Wortes in dem Pointer-Array an dem selben Index abgespeichert wird. Dies hat den Nachteil, dass das Suchen eines Elementes nur in $O(k)$, wobei k die Samplingrate ist, möglich ist. Allerdings wird dies dadurch ausgeglichen, dass fast immer auf konsekutiv abgespeicherte Werte zugegriffen wird, da *pasta-flat* auf der untersten Ebene (also in dem L2 Array) auf 512 Bit Blöcken operiert und auf diesen die Bits zählen muss.

Außerdem wurden die Zugriffe auf das Pointer-Array beschleunigt. Die Zugriffe erfolgen unter der Annahme, dass die Größe einer Speicherzelle des Pointer-Arrays weniger als 32 Bit beträgt. Hieraus folgt, dass ein Wort sich in exakt zwei aneinanderliegenden 32 Bit Speicherzellen befinden kann.

Da das ursprüngliche Array 64 Bit Speicherzellen besitzt, und als ein 32 Bit Array uminterpretiert wird, ist es abhängig von der Endianess des Betriebssystems, an welchen beiden Indices die Wörter existieren. Im trivialen Fall, also der Big-endianess gilt, dass das zweite Wort immer rechts von dem ersten Wort ist. Im Falle von Little-endianess muss ein Index i erst in big-endianess umgewandelt werden, was durch ein i xor 1 geschieht.

4.5 Elias-Fano

Der Elias-Fano VLA benutzt als Select Datenstruktur *pasta-flat*. Diese bestimmt allerdings nur das Start-Bit eines Codewortes, denn die Start-Bits der nachfolgenden Codewörter wird durch eine Wort-Extraktion und einem *std::countl_zero* bestimmt.

5 Evaluation

Es wurde gegen verschiedene Rank und Select Datenstrukturen getestet. Hiervon basierten zwei auf unkomprimierten Bitvektoren, vier auf komprimierten Bitvektoren und den drei in dieser Arbeit beschriebenen Bitvektoren, die Huffman-codiert oder alternativ codiert wurden.

- *dense-pointer*: Die in Unterabschnitt 3.3.1 beschriebene Struktur mit Huffman-Codierung.
- *dense-pointer-alt*: Die in Unterabschnitt 3.3.1 beschriebene Struktur mit der alternativen Codierung.
- *elias-fano*: Die in Unterabschnitt 3.4.1 beschriebene Struktur mit Huffman-Codierung.
- *elias-fano-alt*: Die in Unterabschnitt 3.4.1 beschriebene Struktur mit der alternativen Codierung.
- *sampled-pointer*: Die in Unterabschnitt 3.2.1 beschriebene Struktur mit Huffman-Codierung.
- *sampled-pointer-alt*: Die in Unterabschnitt 3.2.1 beschriebene Struktur mit der alternativen Codierung.
- *pasta-popcount-flat-compressed*: Eine Rank und Select Datenstruktur basierend auf einem huffman-codierten Bitvektor, die ähnlich wie Sampled-Pointer implementiert wurde. Der einzige Unterschied ist, dass die Codewörter ohne einer Codierung, wie Elias-Gamma, abgespeichert werden. Die Blockgröße der Huffman-Codierung wird so gewählt, dass der komprimierte Bitvektor minimal groß ist.
- *pasta-popcount-flat*: Eine Rank und Select Datenstruktur basierend auf einem unkomprimierten Bitvektor.
- *sdsl-rank-v5*: Eine Rank und Select Datenstruktur basierend auf einem unkomprimiertem Bitvektor.
- *sdsl-rrr*: Eine auf [8] basierende Struktur. Diese komprimiert jedes Zeichen asymptotisch optimal.
- *sdsl-sd*: Eine Rank und Select Datenstruktur optimiert für besonders dünnbesetzte Bitvektoren
- *sdsl-hyb-rank*: Eine Rank und Select Datenstruktur die Blöcke unterschiedlich codiert.

Im Folgenden wird immer eine zufällige und eine nachteilige Verteilung in den Darstellungen verwendet, mit einem Füllgrad von 1, 5, 10 Prozent an gesetzten Bits.

In der zufälligen Verteilung ist die Wahrscheinlichkeit, dass ein Bit gesetzt ist für alle Bits identisch. In der nachteiligen Verteilung wird der Bitvektor in zwei Blöcke aufgeteilt. In dem vorderen ist die Wahrscheinlichkeit, dass ein Bit gesetzt ist exakt 1 Prozent und in dem anderen 99 Prozent. Diese Eingabe kann für bestimmte Select-Datenstrukturen suboptimal sein, da die Dichte stark variiert. Ausserdem ist die nachteilige Verteilung gut komprimierbar. Es wurde auf einem PC mit einer Intel I7 11700 CPU und 64 GB DDR4 Ram getestet. Die Programme wurden immer mit dem Compiler „gcc“ und den Parametern „-march=native -O3“ [1] compiliert.

5.1 Größe komprimierter Bitvektoren

Die Komprimierung der Datenstrukturen wird in den Diagrammen in Abbildung 5.1 dargestellt.
Da interessanterweise die meisten platzeffizientesten Varianten der Datenstrukturen auch die schnellsten Varianten sind, wird nicht der Durchschnitt sondern das Minimum des komprimierten Bitvektors über alle Varianten einer Datenstruktur dargestellt.[2] Dieses Phänomen wird in den Diagrammen, die die benötigte Zeit abhängig von dem Komprimierungsgrad darstellen, in Abbildung 5.2 und Abbildung 5.3 ersichtlich. Dies könnte daran liegen, dass die komprimiertesten Bitvektoren für diese Eingabegröße eine Blockgröße von 64 Bit besitzen und deshalb tendenziell weniger Blöcke pro Anfrage decodiert werden müssen. In dem Abschnitt 2.3 wird erklärt, dass das L2 Array der *pasta-flat* Datenstruktur eine Blockgröße von 512 Bit verwendet, also muss für die Blockgröße von 64 Bit 8 Wörter decodiert werden und bei einer Blockgröße von 8 Bit 64 Wörter decodiert werden.

Es fällt auf, dass der Huffman-codierte Bitvektor *pasta-popcount-flat-compressed* für hinreichend große Eingaben den wenigsten Platz benötigt. Der *elias-fano-alt* benötigt meistens am zweitwenigsten Platz. Dies liegt daran, dass dieser ein bisschen mehr als doppelt so viel Platz wie die alternative Codierung benötigt, denn die Rank und Select Datenstruktur benötigt über den Bitvektor wenig Platz. Der *sdsl-rrr* Bitvektor benötigt am drittwenigsten Platz, denn sowohl die Huffman-Codierung als auch die in *sdsl-rrr* verwendete Codierung komprimiert die Eingabe aufgrund der Verteilung asymptotisch optimal [8]. Dies liegt daran, dass für eine Füllrate $f = \frac{m}{n}$, wobei m die Anzahl der gesetzten Bits und n die Eingabegröße in Bits ist, asymptotisch nicht besser als $n\mathbf{H}(\frac{m}{n})$ komprimiert werden kann unter der Annahme, dass eine zufällige Verteilung verwendet wird, wobei $\mathbf{H}$ die Shannon-Entropy ist [7, kap. 2.3]. Für schlecht komprimierbare Bitvektoren ist *dense-pointer-alt* platzeffizient, da hier die durchschnittliche Codelänge maximal wird und diese pro Codewort zusätzlich nur $O(\log(|c_{max}|))$ Platz benötigt.

Es wurde auch die Komprimierungsrate für Bitvektoren, die durch *pac-hash* [6] generiert wurden, getestet, allerdings waren diese zu klein und ähnlich zu Abbildung 5.2. Allerdings

[1] „-march=native“ impliziert auf dieser Architektur die zusätzlichen Optionen „-msse2 -mlzcnt“, die benötigt werden.

[2] Die größte Ausnahme bildet der *sdsl-rrr* Bitvektor. Dies kann man gut an den unteren sechs Plots in 5.2 erkennen.

Datenstruktur	gemessene Größe	relative Größe
dense-pointer	2.551.676.254	1,433
elias-fano	3.499.429.080	1,966
pasta-popcount-flat-compressed	1.780.316.976	1,000
sampled-pointer	3.761.043.004.	2,113

Tabelle 5.1: Die Tabelle betrachtet die gemessene Größe und die Größe eines VLAs relativ zu *pasta-popcount-flat-compressed*. Die gemessenen Größen beziehen sich auf eine Eingabe mit einer Füllrate von 25 % und einer zufälligen Verteilung. Die Eingabegröße beträgt $16384 \cdot 16^5$ Bits, was ungefähr zwei Gigabyte entspricht.

kann man die Eingaben mit nachteiliger Verteilung als eine Eingabe mit einem Muster sehen, diese können gut Huffman-codiert werden. Deshalb konnte bei diesen Eingaben *dense-pointer-alt* besser komprimiert werden als *sdsl-rrr*.

Da die alternative Codierung sehr wenig Platz benötigt, wäre es überlegenswert das Sample-Array von *sampled-pointer-alt* und *dense-pointer-alt* anders zu codieren. Dies könnte beispielsweise Elias-Fano [3][11] codiert werden.

Meine Implementierungen mit Huffman-Codierung benötigen nach den in den Abschnitten 3.2.1.1, 3.3.2.1 und 3.4.1.1 verwendeten Beweisen mehr Platz als die ursprüngliche Eingabe, die in diesem Fall ein Huffman-codierter Bitvektor ist. Dies stimmt auch mit den real gemessenen Größen in 5.1 überein, da für das Elias-Fano VLA und das Sampled-Pointer VLA vorausgesagt wurde, dass diese ungefähr doppelt so viel Platz benötigen.

5.2 Rank und Select Geschwindigkeit

Die Geschwindigkeit der *rank* Anfragen wird in den Diagrammen 5.2 und die Geschwindigkeit der *select* Anfragen wird in der Abbildung 5.3 dargestellt. Die Geschwindigkeit der *rank* Anfragen wurden gegenüber *pasta-popcount-flat-compressed* relativ stark verbessert, aber alle von mir implementierten Varianten sind trotzdem langsam. Die Geschwindigkeit der *select* Anfragen sind allerdings sehr nah an der Geschwindigkeit des *sdsl-rrr* Bitvektors. Da vor allem die Anfragegeschwindigkeiten optimiert wurden, wird es vermutlich mit diesen Datenstrukturen nicht möglich sein deutlich schneller als *sdsl-rrr* zu sein. Es konnte gezeigt werden, dass aufgrund der alternativen Codierung die VLAs Pareto-optimal sind. Die *sampled-pointer* sind vermutlich so ineffizient, da die Extraktion der Elias-Gamma Codes langsamer ist als das Suchen des nächsten Bits auf einem externen Bitvektor, wie es die *elias-fano* implementiert haben.

5.3 Konstruktionszeit

Die Konstruktionszeit relativ zu der Eingabegröße wird in der Tabelle 5.2 dargestellt.
Es wurde immer die die maximale Konstruktionszeit verwendet, damit die Varianten, die

[3] hier ist nicht das Elias-Fano VLA gemeint.

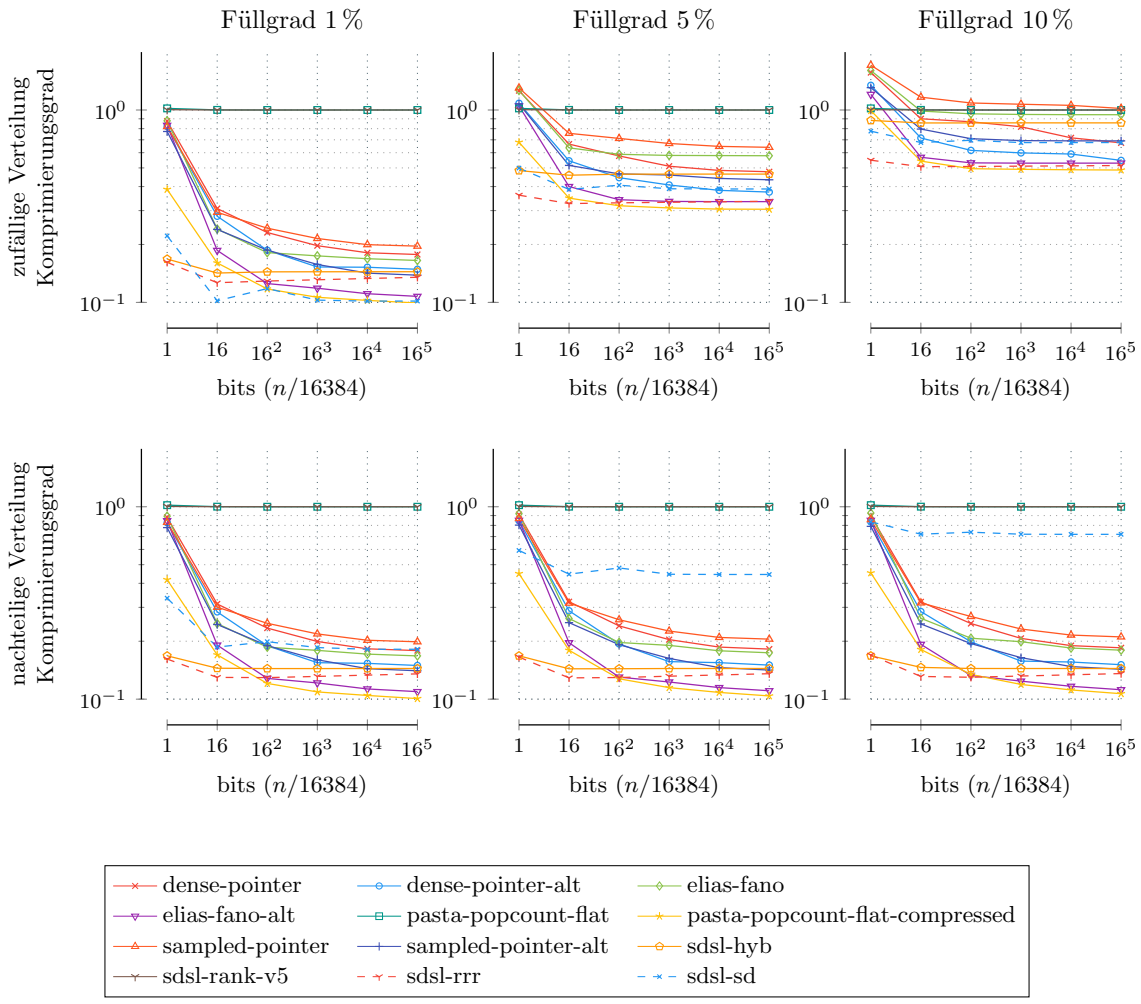


Abbildung 5.1: Die minimale komprimierte Größe der Bitvektoren abhängig von der Eingabegröße

eine Huffman-Codierung verwenden, vergleichbar sind. Denn es wird bei den Huffman-Codierungen jeweils zuerst der Platzbedarf des komprimierten Bitvektors für alle Blockgrößen getestet und anschließend die minimale Blockgröße verwendet.

Da die Anzahl der unterschiedlichen Codewörter mit dem Füllgrad korelliert, kann man zusätzlich den Zeitbedarf für unterschiedlich große Alphabete erkennen. Der Zeitbedarf der Huffman-Codierung konnte in dieser Arbeit für größere Alphabete stark minimiert werden, unter der Verwendung der in Abschnitt 4.2.1 beschriebenen Verbesserung.

Datenstruktur	1%	5%	10%	25%
dense-pointer	2.3633	4.7550	20.0098	32.0636
dense-pointer-alt	1.7966	5.5182	12.9096	21.4545
elias-fano	1.4753	4.7125	20.0173	29.6027
elias-fano-alt	1.4724	3.7198	12.7815	21.3833
pasta-popcount-flat	0.0069	0.0069	0.0069	0.0069
pasta-popcount-flat-compressed	1.1332	9.1902	40.7562	86.5908
sampled-pointer	1.3892	4.7339	20.0026	30.0615
sampled-pointer-alt	1.3743	3.7565	12.7775	21.2890
sdsl-hyb	0.6198	0.9203	1.0844	0.1335
sdsl-rank-v5	0.0172	0.0172	0.0172	0.0171
sdsl-rrr	0.5518	1.0877	1.5426	2.6523
sdsl-sd	0.1509	0.4405	0.8509	2.2726

Tabelle 5.2: Maximale Konstruktionszeit abhängig von der Eingabegröße in Nanosekunden pro Bit. Die Tabelle betrachtet nur die Konstruktionszeit für zufällig verteilte Eingaben mit der Größe $16384 \cdot 16^5$ Bits, was ungefähr zwei Gigabyte entspricht und für Füllgrade von 1%, 5%, 10%, 25%.

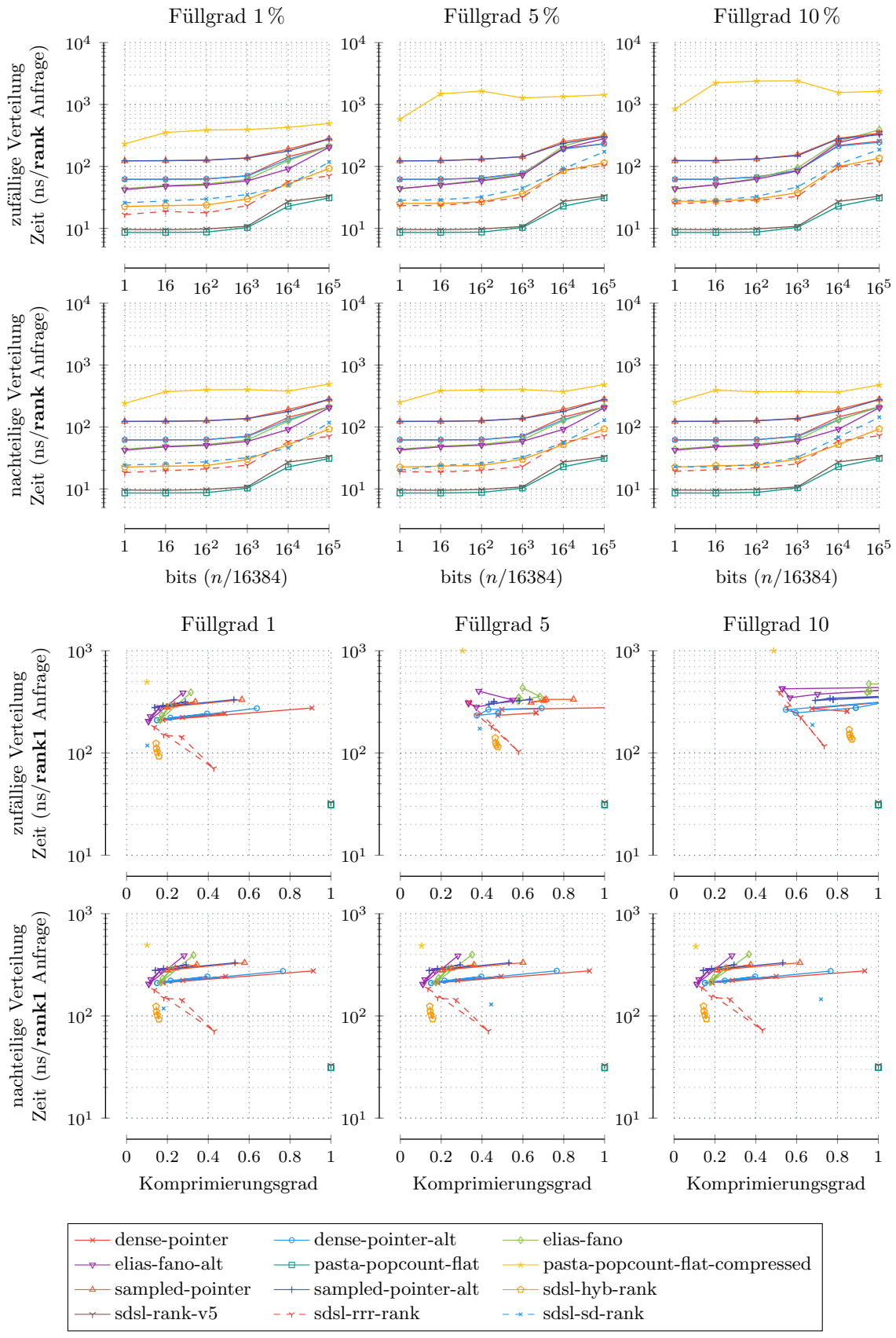


Abbildung 5.2: Oben: Minimale *rank* Zeit abhängig von der Eingabegröße. Unten: Verhältnis von *rank* Zeit zu Komprimierungsrate.

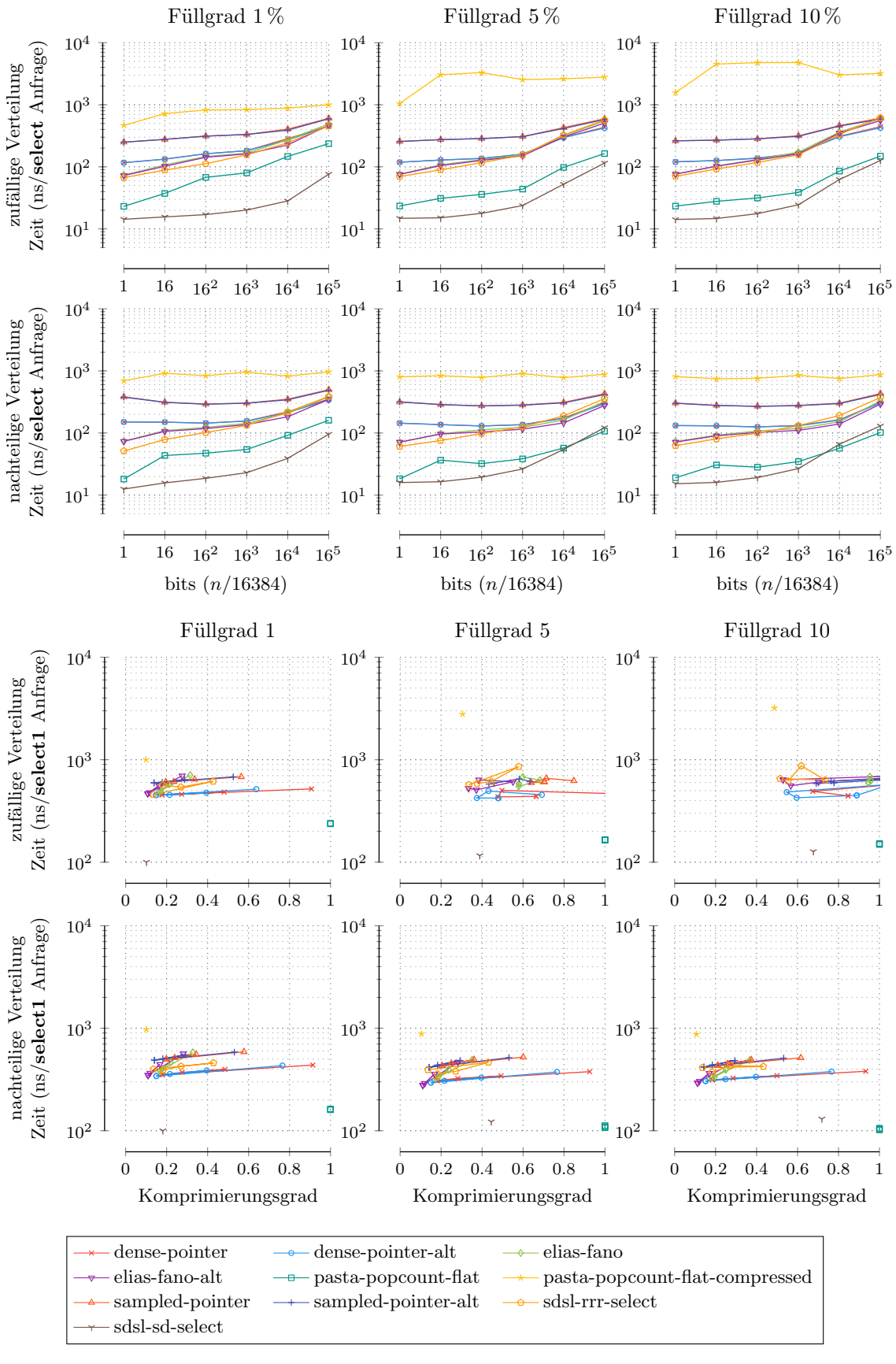


Abbildung 5.3: Oben: Minimale *select* Zeit abhängig von der Eingabegröße. Unten: Verhältnis von *select* Zeit zu Komprimierungsrate.

6 Fazit

Es wurden verschiedene Möglichkeiten vorgestellt, komprimierte Bitvektoren zu indizieren. Außerdem wurde zu diesen Möglichkeiten Variationen vorgestellt und der Platzbedarf dieser theoretisch hergeleitet.

Einige dieser Möglichkeiten wurden implementiert und mit bereits existierenden Implementationen verglichen. Es hat sich herausgestellt, dass die neuen Indizierungen in Kombination einer Rank und Select Datenstruktur das Verhältnis von Anfragezeit und Platzbedarf der ursprünglichen Implementation minimieren konnten. Falls man zusätzlich die in Abschnitt 3.1 definierte alternative Codierung verwendet, so konnten die neuen komprimierten Rank und Select Datenstrukturen auf bestimmten Eingaben Pareto-optimal sein.

6.1 Zukünftige Arbeit

In einer zukünftigen Arbeit könnten Verbesserungen zu der Decodierung von Huffman-Codierungen untersucht werden, da die ursprünglich Implementation bitweise die Huffman-Codewörter decodiert. Außerdem könnten alternative und große Eingaben mit Mustern untersucht werden, da diese optimal durch eine Huffman-Codierung komprimierbar sind, allerdings könnten diese Eingaben in der Realität nur selten vorkommen. Des Weiteren könnten die in den Kapiteln 3.2.1, 3.3.2, 3.3.3 und 3.4.2 vorgestellten Varianten verglichen werden, um den Platzbedarf weiter zu minimieren. Allerdings dürften die Anfragezeiten dieser Varianten zunehmen, weshalb die in [8] vorgestellte Datenstruktur trotzdem effizienter wäre oder im Vergleich sogar Pareto-optimal ist. Des Weiteren können die Elias-Fano VLAs umgebaut werden, sodass es ähnlich zu den Dense-Pointer VLAs ein Pointer-Array besitzt und dieses mit dem Bitvektor indiziert wird.

Literatur

[1] Mark de Berg u. a. „Computational Geometry“. In: *Computational Geometry: Algorithms and Applications*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000. ISBN: 978-3-662-04245-8. DOI: 10.1007/978-3-662-04245-8_1. URL: https://doi.org/10.1007/978-3-662-04245-8_1.

[2] Agner Fog. *Optimizing software in C++*.

[3] David Huffman. „A method for the construction of minimum-redundancy codes“. In: *Resonance* 11 (Feb. 2006), S. 91–99. DOI: 10.1007/BF02837279.

[4] *Intel Ice Lake*. 2020. URL: https://www.7-cpu.com/cpu/Ice_Lake.html.

[5] Florian Kurpicz. *Engineering Compact Data Structures for Rank and Select Queries on Bit Vectors*. 2022. DOI: 10.48550/ARXIV.2206.01149. URL: https://arxiv.org/abs/2206.01149.

[6] Florian Kurpicz, Hans-Peter Lehmann und Peter Sanders. „PaCHash: Packed and Compressed Hash Tables“. In: *CoRR* abs/2205.04745 (2022). DOI: 10.48550/arXiv.2205.04745.

[7] Gonzalo Navarro. *Compact Data Structures: A Practical Approach*. Cambridge University Press, 2016. DOI: 10.1017/CBO9781316588284.

[8] Rajeev Raman, Venkatesh Raman und Srinivasa Rao Satti. „Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets“. In: *ACM Transactions on Algorithms* 3.4 (Nov. 2007), S. 43. DOI: 10.1145/1290672.1290680. URL: https://doi.org/10.1145%5C%2F1290672.1290680.

[9] Raimund Seidel und Cecilia R Aragon. „Randomized search trees“. In: *Algorithmica* 16.4-5 (1996), S. 464–497.

[10] Sebastiano Vigna. „Broadword Implementation of Rank/Select Queries“. In: *Experimental Algorithms*. Hrsg. von Catherine C. McGeoch. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, S. 154–168. ISBN: 978-3-540-68552-4.

[11] Sebastiano Vigna. *Quasi-Succinct Indices*. 2012. DOI: 10.48550/ARXIV.1206.4300. URL: https://arxiv.org/abs/1206.4300.

[12] Dong Zhou, David G. Andersen und Michael Kaminsky. „Space-Efficient, High-Performance Rank & Select Structures on Uncompressed Bit Sequences“. In: *Proc. 12th International Symposium on Experimental Algorithms (SEA)*. Juni 2013.