Karlsruher Institut für Technologie

Implementierung und Vergleich von Arrays mit Einträgen variabler Bitlänge

Bachelorarbeit von

Fidelio Benedict Walz

An der KIT-Fakultät für Informatik
Institut für Theoretische Informatik, Algorithm Engineering

1. Prüfer: Prof. Dr. Peter Sanders
2. Prüfer: Prof. Dr. Thomas Bläsius

1. Betreuer: Dr. Florian Kurpicz

08. Juli 2024 – 08. November 2024

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Quellen und Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

PLACE, DATE

. . . . . .

(Fidelio Benedict Walz)

Zusammenfassung

Arrays mit Einträgen variabler Bitlänge kurz VBL Arrays sind Arrays, bei denen nicht jeder Eintrag die gleiche Länge hat. Die Schwierigkeit bei der Implementierung solcher VBL Arrays ist es weiterhin Zugriff auf die einzelnen Elemente in konstanter Zeit zu gewährleisten und dabei möglichst wenig zusätzlichen Speicherplatz zu verwenden.

Ziel dieser Arbeit ist die Implementierung und der Vergleich zwischen den verschiedenen Ansätzen Sparse Sampling, Dense Sampling und Direct Addressable Codes. Dabei werden die verschiedenen Ansätze sowohl im Platzbedarf als auch in der Zugriffszeit miteinander verglichen.

In Sparse Sampling VBL Arrays werden die Elemente konkateniert abgespeichert und jedes k-te Element wird durch einen absoluten Pointer referenziert. Dabei wurden noch verschiedene Ansätze gewählt um den Speicherbedarf der absoluten Pointer und somit den Speicherbedarf für Sparse Sampling VBL Arrays weiter zu reduzieren.
Bei Dense Sampling VBL Arrays werden die Elemente ebenfalls konkateniert abgespeichert, allerdings wird für jedes Element die Position durch eine Kombination aus absoluten und relativen Pointern gespeichert. Auch hier wurden verschiedene Ansätze zum Speichern der Pointer gewählt, um weiter Platz einzusparen.
In der letzten verglichenen VBL Array Variante, Direct Addressable Codes, werden die Elemente in Teilstücke aufgeteilt und diese Teilstücke auf verschiedenen Ebenen verteilt, wobei für jede Speicherebene abgespeichert wird, welches Element noch weitere Teilstücke besitzt, um in jeder Speicherebene nur für entsprechend große Elemente einen Eintrag zu speichern.

Inhaltsverzeichnis

1. Einleitung

Das Thema dieser Arbeit ist die Implementierung und der Vergleich verschiedener Ansätze von Arrays mit Einträgen variabler Länge (VBL Array).

Die grundlegenden Datentypen in Programmiersprachen wie *char*, *short*, *int*, *long* etc. haben eine Größe, die einem Vielfachem von 8 Bits entsprechen. Was aber, wenn man Datentypen hat, die etwas mehr oder weniger Platz in Bits benötigen als das nächste oder vorherige Vielfache von Acht? Besonders bei einem Array von Elementen summiert sich dieser zusätzlich zugeteilte, aber nicht verwendete Speicher, der entsteht, da jedes Element an einem Vielfachen von Acht ausgerichtet werden muss, was einen erheblichen zusätzlichen Speicherbedarf benötigt.

Noch größere Auswirkungen hat dieses Problem, wenn die Einträge des Arrays unterschiedlich groß sind. Um alle Elemente in einem Array aus Standarddatentypen speichern zu können, muss der Datentyp des Arrays dem nächstgrößeren Datentyp des größten Elements entsprechen. Dies passiert beispielsweise wenn Zahlen effizient mit der Golomb-Rice [15] oder Elias-Delta [4] Kodierung abgespeichert werden. Hat man nun sehr viele sehr kleine Elemente und ein paar wenige Ausreißer, die sehr groß sind, verschwendet man so sehr viel Speicherplatz. Hat man z.B. 42 Elemente wovon 41 Elemente aus weniger als 8 Bits bestehen und ein Element mit mehr als 32 Bits aber weniger als 65 Bits müsste man ein Array aus long long int anlegen und hätte somit eine Verwendung zwischen $\frac{41*1+33}{42*64} = 2,75\%$ bis $\frac{41*7+64}{42*64} = 13,06\%$, was gerade in Hinsicht der immer größer werdenden Datenmengen nicht hinnehmbar ist.

Bei Arrays mit Elementen fester Größe kann dieses Problem gelöst werden, indem die Einträge im Speicher konkateniert abgespeichert werden. Die Position des i-ten Elements kann dann durch die Startposition s und die Länge der Elemente ℓ durch $s + i\ell$ berechnet werden, wodurch weiterhin konstante Zugriffszeit gewährleistet wird. Bis auf das Padding, das nach dem letzten Element hinzugefügt wird, um den zugeteilten Speicher auf ein Vielfaches von dem verwendeten Datentyp zu bringen, wird auch kein zusätzlicher Platz benötigt.

Um bei VBL Arrays konstante Zugriffszeit zu gewährleisten ist eine etwas komplexere Datenstruktur erforderlich. Genau wie bei normalen Arrays werden bei VBL Arrays die Einträge konkateniert im Speicher abgelegt. Durch die verschieden großen Einträge in dem Array kann die Position des i-ten Elements im Speicher nicht mehr wie bei Arrays mit Elementen fester Größe einfach durch $s + i\ell$ berechnet werden, da die Elemente alle eine unterschiedliche Länge haben und für die Position des i-ten Elements die Summe der Länge der vorherigen Elemente benötigt wird [14]. Ein trivialer Ansatz um weiterhin konstanten

Zugriff zu erlauben wäre ein zweites Array L der Länge n zu speichern, in dem jeder Eintrag die Summe der Länge der vorherigen Elemente speichert, wodurch die Position des i-ten Elements durch $s + L[i]$ berechnet werden kann. Dieser Ansatz ist aber sehr platzineffizient, da vor allem für Daten mit $\ell \ll w$, wobei w der Länge der Elemente in L in Bits entspricht, dadurch mehr Platz zum Speichern der Längen als zum Speichern der eigentlichen Daten benötigt wird.

Um dieses Problem besser zu lösen gibt es verschiedene Ansätze, wobei in dieser Arbeit der Fokus auf den Ansätzen Sparse- und Dense Sampling [6], Elias-Fano Coding [3, 5] und Direct Addressable Codes (DACs) [2] liegt. Sparse- und Dense Sampling sind beides Ansätze, welche Pointer auf die Einträge verwenden. Während Sparse Sampling einen Pointer auf jedes h-te Element hat, von dem dann zu dem gewünschten Element traversiert werden kann [14], verwendet Dense Sampling Absolute Pointer zu jedem h-ten Element und für die so entstandenen Blöcke relative Pointer, um zu dem gewünschten Element zu gelangen. [6]. Elias-Fano Coding ist eine effiziente Art ganzzahlige, aufsteigend sortierte und nicht doppelt vorkommende Ganzzahlen platzeffizient zu speichern, mit konstanter Zugriffszeit auf die i-te Zahl [3, 5], womit es sich gut eignet, um die Anfangspositionen der einzelnen Arrayeinträge zu speichern. Das letzte vorgestellte Konzept DACs, basiert darauf, die einzelnen Einträge in Blöcke fester Länge zu zerlegen und in Ebenen zu speichern. Ebene 1 enthält von allen Einträgen die ersten b Bits, Ebene 2 die zweiten b Bits von den Einträgen, die größer als b Bits sind. Dieses Schema wird fortgeführt, bis in der letzten Ebene kein Element fortgeführt werden muss [2]. Die Ziele dieser Arbeit sind die Recherche, Implementierung und der Vergleich dieser Ansätze, wobei nur Statische Arrays betrachtet werden. Hierzu wird jeder Ansatz in C++ implementiert und auf gegebenen Datensätzen gegeneinander getestet.

2. Theoretische Grundlagen

In diesem Kapitel werden Notationen und grundlegende Konzepte vorgestellt, welche in dieser Arbeit verwendet werden. Dabei besteht ein Array A aus $n \in \mathbb{N}$ Elementen, wobei die Länge des Elements $A[i]$ mit $i \in \{0, \ldots, n-1\}$ durch ℓ_i Bits beschrieben wird. Die Größe des Arrays ergibt sich durch $\sum_{i=0}^{n-1} \ell_i$. Die Operationen &, ˜, | und $\ll$ sind wie in C++ die Bitwise Operationen and, inverse, or und shift. Die Operation : beschreibt die Konkatenation zweier Elemente.

2.1. Variable-Byte Coding (VByte Coding)

Variable-Byte Coding (VByte Coding) [16] ist eine Kodierung, in der verschieden große Ganzzahlen x in variabel vielen Bytes gespeichert werden können. Dabei werden kleine Ganzzahlen mit weniger Bytes kodiert als große Ganzzahlen. Diese Kodierung funktioniert, in dem man die $\lfloor \log_2 x \rfloor + 1$ benötigten Bits um x zu kodieren in Blöcke von 7 Bits aufteilt. Die Ganzzahl x wird dabei Byte für Byte gespeichert, wobei in jedem Byte 7 Bit verwendet werden um ein Teil von x zu repräsentieren und das niederwertigste Bit wird in dem letzten verwendeten Byte, also den 7 niederwertigsten Bits von x, auf 0 bzw. 1 bei allen anderen Bytes gesetzt. Das Byte mit den höchstwertigsten Bits wird gegebenenfalls von vorne mit Nullen aufgefüllt um auf ein ganzes Byte zu kommen. So wird z.B. 217 (Binär 11011001) mit VByte Coding als 00000011 10110010 repräsentiert.

2.2. Huffman Codes

Huffman Codes [10] sind eine optimale verlustfreie statistische Kodierung für eine Sequenz aus einem endlichen Alphabet. Huffman Codes funktionieren, indem für häufiger vorkommende Zeichen kürzere Codes verwendet werden. Um einen Huffman Code zu konstruieren wird zunächst für jedes Zeichen des verwendeten Alphabets eine relative Häufigkeit berechnet. Mithilfe dieser relativen Häufigkeit kann ein Huffman Baum konstruiert werden, aus dem dann die Huffman Codes für die einzelnen Zeichen abgelesen werden können. Dazu werden in dieser Arbeit nur binäre Huffman Codes betrachtet. Um einen binären Huffman Code zu konstruieren, wird zuerst für jedes Zeichen einen Knoten mit der relativen Häufigkeit als Label erstellt. Um einen Huffman-Baum aus diesem Wald zu erstellen, werden die beiden Teilbäume mit den geringsten relativen Häufigkeiten an der Wurzel zu einem neuen Teilbaum kombiniert. Die neue Wurzel erhält dabei ein Label, das

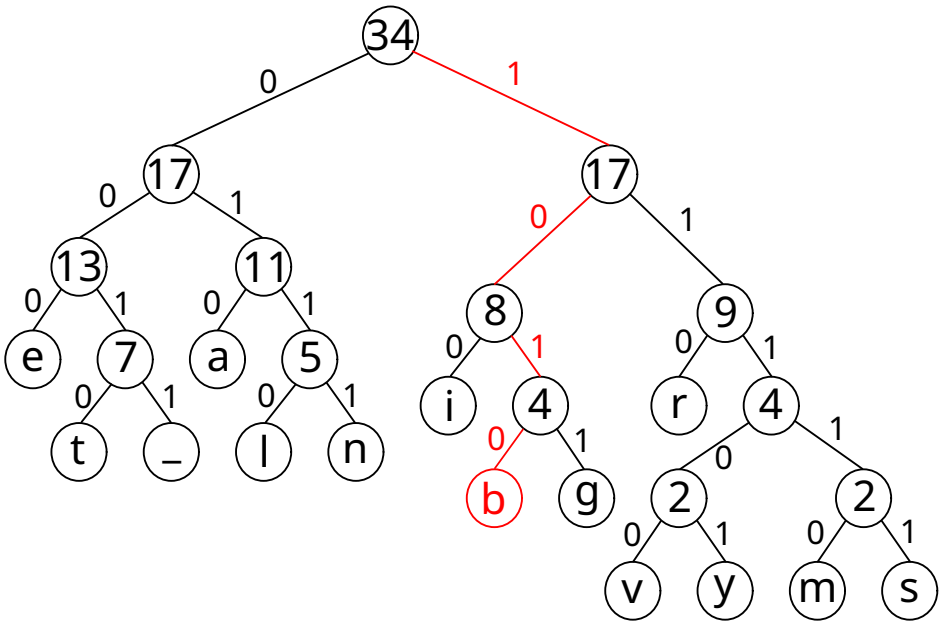

Abbildung 2.1.: Beispiel eines Huffman Baums für den Text „arrays_mit_eintraegen_variabler_bitlaenge“. Der rot markierte Pfad beschreibt den daraus resultierenden Huffman Code für den Buchstaben „b“.

der Summe der relativen Häufigkeiten der Wurzeln der beiden Kinder entspricht. Dies wird wiederholt bis alle Knoten zusammen einen Huffman Baum ergeben. Die Blätter dieses Baums entsprechen den Zeichen und die Pfade von der Wurzel zu den Blättern bilden die Huffman Codes. Dabei wird jeder Abstieg nach links mit einer 0 und jeder Abstieg nach rechts mit einer 1 assoziiert . Abbildung 2.1 zeigt beispielhaft einen Huffmanbaum für den Text „arrays_mit_eintraegen_variabler_bitlaenge“ wobei dem Buchstaben „b“ in diesem Beispiel der Huffman Code 1010 entspricht

2.3. Suffix Arrays und LCP Arrays

Ein Suffix Array SA [13] ist eine sortierte Liste aller Suffixe eines Textes T. Dabei enthält der i-te Eintrag des Suffix Arrays $SA[i]$ den Startindex des i-ten Suffixes in lexikographischer Reihenfolge.

Ein LCP Array [13] beschreibt für jeden Eintrag die Länge des größten gemeinsamen Präfixes des Suffixes i mit dem Suffix $i-1$ aus dem Suffix Array. Der j-te Wert in einem LCP Array wird durch $LCP[0] = 0$ und $LCP[j] = \max\{\ell : T[SA[j]..SA[j]+\ell) = T[SA[j-1]..SA[j-1]+\ell)\}$ für $j \geq 1$ definiert.

2.4. Elias-Gamma Codes

Elias-Gamma Codes [4] ist eine präfixfreie Kodierung der natürlichen Zahlen $x \in \mathbb{N}^+$. Um x zu kodieren wird x zuerst binär kodiert $\beta(x)$ und dann die Länge der binären Kodierung unär durch $\alpha(len(\beta(x)))$ kodiert. Dabei ist *len* eine Funktion, $len(x)$ ist die Länge von x. Da jede binäre Zahl aus $\mathbb{N}^+$ mit 1 beginnt, kann diese in der unären Längenkodierung ignoriert werden. Die Elias-Gamma Kodierung für eine Zahl $x \in \mathbb{N}^+$ ergibt sich dann durch $\gamma(x) = \alpha(\lfloor \log_2(x) \rfloor) : \beta(x)$.

2.5. Elias-Delta Codes

Elias-Delta Codes [4] sind eine Erweiterung von Elias-Gamma Codes, wobei die Länge der binären Kodierung durch Elias-Gamma Codes kodiert wird. Die Elias-Delta Kodierung für eine Zahl $x \in \mathbb{N}^+$ ergibt sich dann durch $\delta(x) = \gamma(\lfloor \log_2(x) \rfloor) : \beta(x)$.

2.6. Bitvektor

Bitvektoren mit effizienten *rank* (Abschnitt 2.6.1) und *select* (Abschnitt 2.6.2) Funktionen [8] sind essentiell für viele succincte Datenstrukturen wie succincte Baum Repräsentationen wie Level Ordered Unary Degree Sequence (LOUDS) [11] und Depth First Unary Degree Sequence (DFUDS) [1], oder auch die effiziente Kodierung von monotonen positiven Ganzzahlen mit Elias-Fano (Abschnitt 2.7) [3, 5]. Ein Bitvektor ist dabei ein Vektor aus n Bits, wobei für jedes Bit genau ein Bit im Speicher verwendet wird. Dies wird erreicht durch das Abspeichern der Bits in einem Integer Vektor W mit Wortgröße w wobei das i-te Bit durch Bitwise Operationen in W platziert wird. Das i-te Bit kann dann mit $B[i] = W[\lfloor i/w \rfloor]/2^{i \bmod w} \bmod 2$ gelesen werden.

2.6.1. Rank Funktion

Relevanter werden Bitvektoren durch die Implementierung der *rank* und *select* Funktionen. Die $rank_\alpha(i)$ Funktion mit $\alpha \in \{0, 1\}$ und $0 \leq i < n$ gibt die Anzahl der α vor der Position i aus, also $rank_\alpha(i) = \sum_{j=0}^{i-1}(B[j] = \alpha)$. Die triviale Implementierung für die *rank* Funktion ist es über B zu iterieren und α zu zählen. Die hieraus resultierende lineare Laufzeit ist aber nicht hinnehmbar.

In der klassischen Lösung [8] um ein konstante Zugriffszeit für *rank* zu garantieren wird der Bitvektor B in Blöcke der Länge $b = \lfloor \log_2(n)/2 \rfloor$ unterteilt. Hintereinander liegende Blöcke werden in Superblöcke der Größe $s = b\lfloor \log_2(n) \rfloor$ gruppiert. Für jeden Superblock wird der absolute Rank des Superblocks in R_s gespeichert. Für die Blöcke werden die relativen Ränge zu der Startposition des entsprechenden Superblocks in R_b gespeichert. $R_s[k] = rank_1(k \cdot s)$, wobei $0 \leq k \leq \lfloor n/s \rfloor$ und $R_b[x] = rank_1(x \cdot b) - rank_1(k \cdot s)$ mit $k = x/\lfloor \log_2(n) \rfloor$. R_s

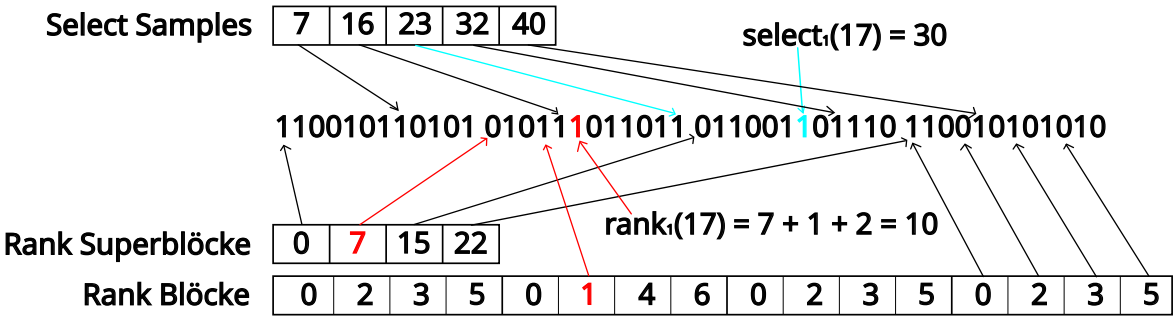

Abbildung 2.2.: Beispiel eines Bitvektors mit *rank* und *select* Unterstützung. Die Blockgröße für *rank* ist 3 Bits und die Größe der Superblöcke beträgt 4 Blöcke. Für *select* werden Select Samples gespeichert, die die Position für jede 5-te 1 beinhalten. Die Select Samples für jede 5-te 0 muss auch gespeichert werden, ist aber aus Platzgründen hier nicht abgebildet.

benötigt $O(\frac{n}{\log_2(n)})$ Bits und R_b benötigt $O(\frac{n \log_2 \log_2(n)}{\log_2(n)})$ Bits. Für jede Bitsequenz S der Länge b wird dann eine Lookuptabelle $R_p[S][j] = S.rank_1(j)$ mit $len(S) = b$ und $0 \leq j < b$ mit allen Möglichkeiten für S abgespeichert, was $O(\sqrt{n} \log_2(n) \log_2 \log_2(n))$ Bit benötigt. Da die Länge von S logarithmisch zu n ist $len(S) = O(log(n))$ kann im RAM Modell dieser Teil in konstanter Zeit auch ohne zusätzlichen Platzbedarf berechnet werden. Eine $rank_\alpha(i)$ Anfrage wird dann durch $R_s[\lfloor i/s \rfloor] + R_b[\lfloor i/b \rfloor] + R_p[S][i \bmod b]$ berechnet, wobei S die Bitsequenz des $i/b - ten$ Blocks ist. Diese Lösung verwendet $\frac{n}{\log_2(n)} + \frac{n \log_2 \log_2(n)}{\log_2(n)} + \sqrt{n} \log_2(n) \log_2 \log_2(n) = o(n)$ Bits [8].

Dieser Ansatz kann hinsichtlich des Platzbedarfs deutlich verbessert werden, indem die Blockgröße auf 512-Bits gesetzt wird, sodass ein Block exakt in eine Cachezeile passt. Die zweite Optimierung ist das verwenden der popcount Instruktion anstatt der vorberechneten Tabelle, was erheblich Platz spart ohne signifikant langsamer zu sein [17]. Die dritte Optimierung ist das Hinzufügen von super Superblöcken L der Größe $l = 2^{32}$-Bits um die Größe der Superblöcke auf 32 zu setzen und weiterhin 2^{64} Bits große Bitvektoren zu unterstützen [17].

Abbildung 2.2 zeigt exemplarisch die *rank* Funktion auf einen Bitvektor mit Blöcken der Größe 3 Bits und Superblöcke der Größe 4 Blöcke.

2.6.2. Select Funktion

Die zweite wichtige Funktion auf Bitvektoren ist die *select* Funktion. Die $select_\alpha(i)$ Funktion mit $\alpha \in \{0, 1\}$ und $0 \leq i < n$ gibt die Position des i-ten α zurück. Eine sehr einfache Methode *select* zu realisieren ist die binäre Suche über B mit Hilfe der *rank* Funktion. Dabei ist die Lösung von $select_\alpha(j)$ die Position i für die gilt: $rank_\alpha(i) = j$ und $rank_\alpha(i-1) = j - 1$. Die Zugriffszeit dieser Lösung ist allerdings $O(\log_2(n))$.

Dieser Ansatz kann weiter verbessert werden, indem man die Block Superblock Struktur von *rank* ausnutzt. Die erste binäre Suche wird dann auf dem Superblock ausgeführt, die zweite auf den Blöcken innerhalb des gefundenen Superblocks und die dritte Suche auf dem

gefundenen Block. Die Suche in einem Superblock kann neben binärer Suche auch durch sequentielle Suche auf den Blöcken oder durch direktes Zählen mit der popcount Instruktion ausgeführt werden. Durch die sehr kleinen Blöcke ist es sinnvoll auf den Blöcken keine binäre Suche zu verwenden. Stattdessen kann man entweder mittels popcount über Byteblöcke suchen und dann final Bit für Bit suchen oder direkt Bit für Bit suchen [8]. Für die klassische *rank* Implementierung ist die sequentielle Suche mit popcount in dem Superblock als auch in dem Block die schnellste Methode [8]. Durch eine andere Implementierung der *rank* Funktion durch z.B. mehr Indexebenen gibt es leichte Variationen in der Implementierung, das Prinzip bleibt aber gleich. Durch eine andere Parameterwahl für *rank* kann sich die optimale Strategie für *select* aber ändern. Es gibt auch Ansätze die das *select* Problem in konstanter Zeit lösen, diese brauchen aber deutlich mehr zusätzlichen Platz ($> 60\%$) [8], was sie im Allgemeinen nicht praktikabel macht.

Ein weiterer Ansatz um *select* zu realisieren und nicht vollständig auf binäre Suche zu setzen bietet das Sampeln jedes k-ten Elements. Es wird also für jede k-te 1 bzw. 0 die absolute Position in einem separaten Array S_1 bzw. S_0 abgespeichert. Durch dieses Sampeln kann die Suche im Voraus eingegrenzt werden. Dabei wird bei der Anfrage $select_\alpha(i)$ zuerst in $S_\alpha[i/k]$ eine untere Grenze gelesen. Ausgehend von dieser Grenze kann zuerst der korrekte Superblock und dann der korrekte Block gesucht werden [17]. Abbildung 2.2 zeigt die *select* Function mit Samples mit $k = 5$. Weitere Verbesserungen bei der Verwendung von Samples können durch Ausnutzen der super Superblöcke erzielt werden. Es wird dann für jeden super Superblock ein Select Sample Array gespeichert, wodurch diese nur 32 Bit groß sein müssen anstatt 64 Bit. Für eine *select* Anfrage wird dann zuerst eine binäre Suche auf den super Superblöcken durchgeführt. Innerhalb des super Superblocks wird wie zuvor beschrieben vorgegangen.

2.7. Elias-Fano Kodierung

Die Elias-Fano Kodierung [3, 5] beschreibt eine Kodierung um effizient n streng monotone Ganzzahlen aus einem Universum $U = [0, u)$ zu kodieren. Um die Ganzzahlen zu kodieren werden sie in eine obere und untere Hälfte geteilt. Die obere Hälfte enthält die $\lceil \log_2(n) \rceil$ höchstwertigsten Bits. Die untere Hälfte enthält die $\lceil \log_2(u) - \log_2(n) \rceil$ niederwertigsten Bits und wird in einem Array mit Elementen fester Größe $\lceil \log_2(\frac{u}{n}) \rceil$ abgespeichert. Die untere Hälfte benötigt somit $n \cdot \lceil \log_2(\frac{u}{n}) \rceil$ Bits Speicherplatz. Für die obere Hälfte wird ein Bitvektor B der Länge $2n + 1$ abgespeichert. Dabei ist $B[i + p_i] = 1$ wobei p_i die obere Hälfte des i-ten Elements ist. Um auf das i-te Element zuzugreifen kann die obere Hälfte durch $B.select_1(i + 1) - i$ gelesen werden und die untere Hälfte kann aus dem Array ausgelesen werden. Daraus lässt sich dann das i-te Element berechnen [3, 5]. Abbildung 2.3 verdeutlicht das Abspeichern von einem mit Elias-Fano kodierten Array A und den Zugriff auf das sechste Element.

Eine weitere Funktion, welche mit Elias-Fano effizient berechnet werden kann sind Predecessor Anfragen. Predecessor Anfragen sind folgendermaßen definiert: $pred(x) = \max_{y \in A}\{y \mid y \leq x\}$. Die Lösung einer Predecessor Anfrage ist also das größte Element in A, das kleiner gleich

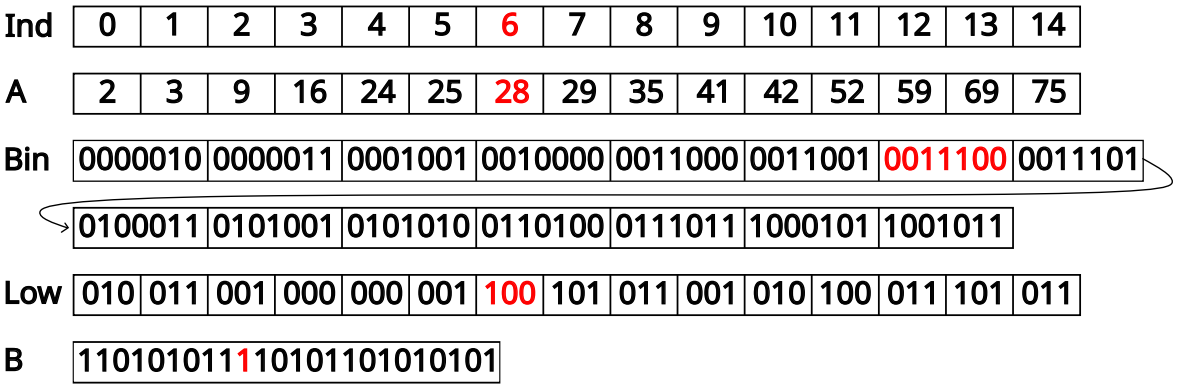

Abbildung 2.3.: Darstellung eines Arrays A in Elias-Fano mit Low als ein Array mit Elementen fester Größe mit Elementgröße 3 und den Bitvektor B zum Speichern der oberen Hälfte. Die rot markierten Einträge stellen einen Zugriff auf das 6-te Element dar. Die obere Hälfte ergibt sich aus $B.select_1(6+1) - 6 = 4 = 0011$ und die untere Hälfte aus $Low[6] = 2 = 100$ woraus sich $0011100 = 28$ rekonstruieren lässt.

der Anfrage ist. Realisiert werden diese Anfragen durch ein $select_0(x')$ mit $x' = \lceil \log_2(n) \rceil$ höchstwertigsten Bits von x um den Bereich der Anfrage zu ermitteln. Dazu werden noch alle unteren Hälften der Elemente beginnend mit x' gescannt um den Predecessor zu finden. Wird kein Element mit x' höchstwertigsten Bits kleiner gleich x gefunden kann mit $rank_1(select_0(x'))$ die Position des größten Elements mit der oberen Hälfte kleiner als x' gefunden werden. Dieses Element kann als Lösung ausgegeben werden. Successor Anfragen $suc(x) = \min_{y \in A}\{y \mid y \geq x\}$ können analog gelöst werden. Die Laufzeiten für Predecessor und Successor Anfragen sind $O(\log(U/n))$ [3, 5].

2.8. Arrays mit Elementen fester Größe

Um die Konstruktion von VBL Arrays besser zu verstehen, ist es hilfreich, das Problem von Arrays mit Einträgen fester Größe von ℓ Bits zuvor genauer zu untersuchen.

2.8.1. Problematik

Wie in der Einleitung beschrieben kann man Arrays durch Konkatenation der Einträge speichern, aber wie genau kann das erreicht werden, wenn die Daten nicht in einen standard Datentyp passen? Während Operationen wie lesen und schreiben auf ganze Bytes in vielen Programmiersprachen unterstützt werden, werden Bitsequenzen, die nicht an den Bytegrenzen ausgerichtet sind nicht nativ unterstützt. Zudem kann nur ein Vielfaches von Bytes im Speicher zugewiesen werden.

2.8.2. Konstruktion

Eine Möglichkeit um dieses Problem zu lösen ist, das Anlegen eines Integer Arrays $W[0, \lceil \frac{\ell n}{w} \rceil - 1]$, in dem die ℓn Bits in einem virtuellen Bitarray B gespeichert werden [14]. Man speichert also ein Array W aus Integern der Wortgröße w und verwendet auf diesem Array aber nicht die Integer als solche, sondern nutzt Bitwise Operationen wie $shift$, *and* und *or* um einzelne Bits aus den Integern in dem Array zu manipulieren.

Dadurch entsteht ein virtuelles Bitarray B in dem die ℓ großen Elemente konkateniert gespeichert werden können. In B wird das i-te Element aus A durch die Bits von $B[i\ell]$ bis $B[(i+1)\ell - 1]$ repräsentiert, welche in W entweder vollständig in einem Eintrag oder auf zwei Einträge verteilt sind. Wenn $\lfloor \frac{i\ell}{w} \rfloor = \lfloor \frac{(i+1)\ell-1}{w} \rfloor$ ist das Element vollständig in $W[\lfloor \frac{i\ell}{w} \rfloor]$ enthalten. Ist $\lfloor \frac{i\ell}{w} \rfloor \neq \lfloor \frac{(i+1)\ell-1}{w} \rfloor$ liegt das Element auf der Grenze zwischen zwei Einträgen in W und ist dann auf die Einträge $W[\lfloor \frac{i\ell}{w} \rfloor]$ bis $W[\lfloor \frac{(i+1)\ell-1}{w} \rfloor]$ verteilt. Abbildung 2.4 verdeutlicht die Repräsentation eines Arrays A mit $n = 7$ und $\ell = 6$.

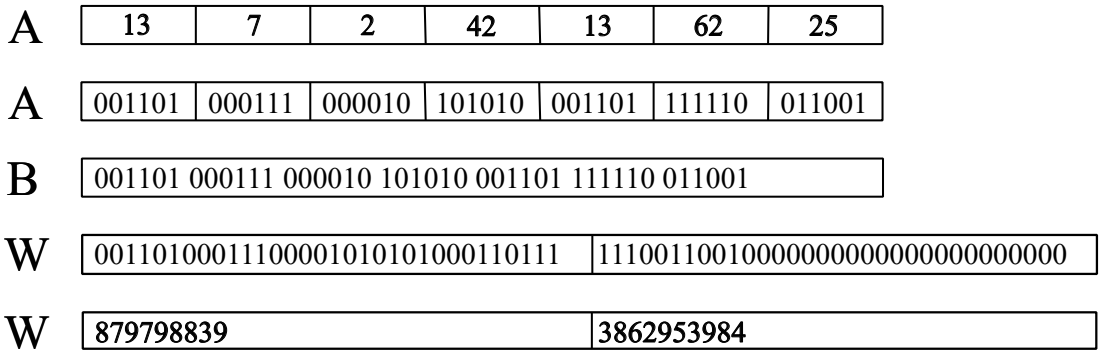

Abbildung 2.4.: Repräsentation eines Arrays A mit 7 Elementen der Größe $\ell = 6$ Bits, wobei A und W zur Verdeutlichung jeweils zweimal abgebildet sind, einmal in Binär und einmal in Dezimal.

Neben dem kompakten Abspeichern werden auch effiziente Lese- und Schreibzugriffe benötigt. Diese Arbeit beschäftigt sich nur mit statischen Arrays. Einfüge- und Änderungsoperationen werden nicht betrachtet.

In der beschriebenen Art die Arrays abzuspeichern steht das $j$te Bit des virtuellen Bitarrays B in dem Array W an der Stelle $B[j] = \lfloor W[\lfloor j/w \rfloor]/2^{w-r} \rfloor \ mod\ 2$ mit $r = j \ mod\ w + 1$ [14]. Um nun aber konstanten Lesezugriff auf $A[i]$ zu ermöglichen muss ein ganzer Block der Länge ℓ aus B gelesen werden. Genauer für das Element $A[i]$ muss $B[i\ell, (i+1)\ell - 1]$ gelesen werden. Dabei wird in zwei Fälle unterschieden:

Fall 1 das Element ist komplett in einem Eintrag in W also $\lfloor \frac{i\ell}{w} \rfloor = \lfloor \frac{(i+1)\ell-1}{w} \rfloor$:
$A[i] = B[i\ell, (i+1)\ell - 1] = \lfloor W[\lfloor \frac{i\ell}{w} \rfloor]/2^{w-r} \rfloor \ mod\ 2^{\ell}$[14]

Fall 2 das Element ist auf zwei Einträge in W verteilt also $\lfloor \frac{i\ell}{w} \rfloor + 1 = \lfloor \frac{(i+1)\ell-1}{w} \rfloor$:
$A[i] = B[i\ell, (i+1)\ell - 1] = W[\lfloor \frac{i\ell}{w} \rfloor] \ mod \ 2^{l-r} \cdot 2^r + \lfloor W[\lfloor \frac{(i+1)\ell-1}{w} \rfloor] / 2^{w-r} \rfloor$ [14]
Dabei ist r wie oben definiert mit $j = (i+1)\ell - 1$.

3. Untersuchte Arrays mit Einträgen variabler Länge

In diesem Kapitel werden die verschiedenen Implementierungsansätze von VBL Arrays detailliert beschrieben und hinsichtlich des Platzbedarfs und der Zugriffszeit analysiert.

3.1. Sampling VBL Arrays

Das platzeffizienteste VBL Array wäre ein Array A, bei dem jedes Element mit den informationstheoretisch minimalen Bits kodiert und konkateniert abgespeichert wird. Gespeichert werden die Elemente dann genau wie bei Arrays mit Elementen fester Größe (Abschnitt 2.8) [14] indem die benötigten Bits der Elemente in einem virtuellen Bitarray B gespeichert werden, welches dann in einem Integer Array W gespeichert wird. Dabei wird das Element $A[j]$ durch die Bits $B[c]$ mit $\sum_{i=0}^{j-1} \ell_i \leq c < \sum_{i=0}^{j} \ell_i$ repräsentiert. Gespeichert und darauf zugegriffen wird auf dieses Bitarray genau wie in Abschnitt 2.8.2 beschrieben.

Das macht aber den Zugriff auf die Elemente im Allgemeinen unmöglich, da nicht immer bekannt ist, wann ein Element aufhört oder ein neues Element beginnt. Wenn zum Beispiel das Array $A = [4, 7, 9]$ werden soll, wobei die Werte binär kodiert sind, erhält man auf diese Weise das Array als VBL Array kodiert: $VBLArray = 1001111001$. Dieses Array könnte aber genauso als $B = [9, 1, 1, 9]$ oder $C = [1, 0, 0, 7, 9]$ interpretiert werden.

Um weiterhin auf alle Elemente zugreifen zu können muss entweder für jedes Element bekannt sein, an welcher Position es anfängt und endet (Dense Sampling) oder es muss eine Möglichkeit geben über die Daten zu iterieren. Da das Iterieren über jedes Element von Anfang an bis zu dem gewünschten Element eine sehr schlechte worst Case Laufzeit hat , kann man hier noch für jedes k-te Element die Position speichern, um nur über maximal k Elemente iterieren zu müssen (Sparse Sampling).

Der Speicherbedarf bei dieser Art von VBL Arrays berechnet sich aus der Größe des Arrays W zusammen mit der Größe der darüber liegenden Navigationsdatenstruktur. Die Größe von W berechnet sich aus der minimalen Anzahl an benötigten Bits $\sum_{j=0}^{n-1} \ell_j$ und einem eventuellen Padding, welches am Ende benötigt wird um auf ein Vielfaches der Wortlänge w des Datentyps von W aufzufüllen. Der Speicherbedarf der grundlegenden Daten beträgt also $\lceil \frac{\sum_{j=0}^{n-1} \ell_j}{w} \rceil \cdot w$. Der Speicherbedarf für die Navigationsdatenstruktur und die Zugriffszeit hängt von der jeweiligen Navigationsdatenstruktur ab und wird in den Unterabschnitten zu Sparse- (Abschnitt 3.1.1) bzw. Dense (Abschnitt 3.1.2) Sampling analysiert.

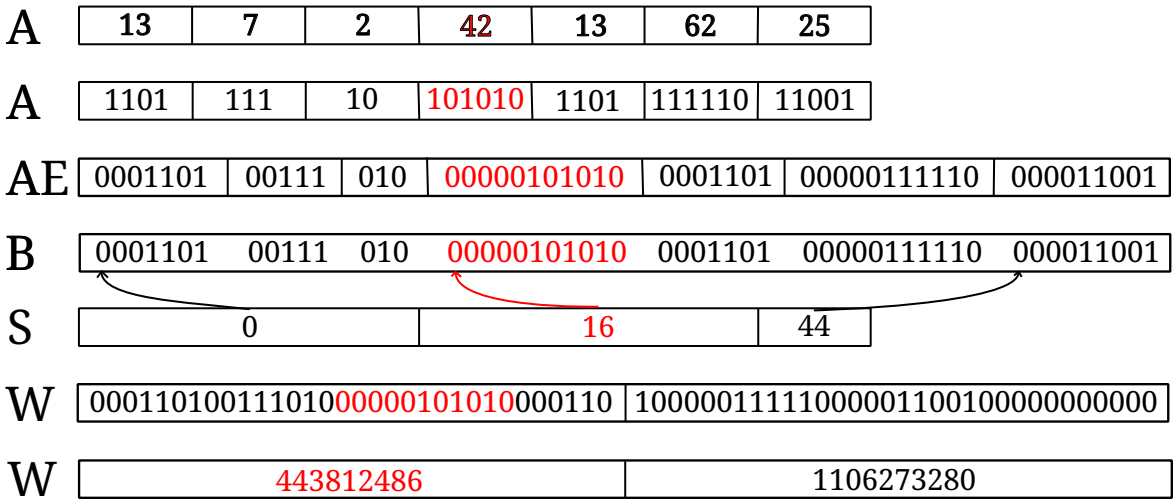

Abbildung 3.1.: Repräsentation des Arrays A als Sparse Sampling VBL Array mit $k = 3$ und Elias-Gamma Kodierung für Präfixfreiheit, wobei AE der Elias-Gamma Kodierung von A entspricht. W ist das Array, das im Endeffekt gespeichert wird. Die Sparse Samples werden in S gespeichert. Die rot markierten Elemente repräsentieren die Daten, die bei dem Zugriff auf $A[3]$ relevant sind. Die Startposition für die Suche steht an $S[3/k] = S[1] = 16$. Um ein Elias-Gamma kodiertes Element zu lesen werden von der Startposition die ersten c 0-en bis zur ersten 1 gelesen und c Bits nach der ersten 1.

3.1.1. Sparse Sampling VBL Array

Um die Möglichkeit über das VBL Array zu iterieren zu garantieren müssen die gespeicherten Daten in ihrer kodierten Form präfixfrei sein. Einige komprimierte Daten wie z.B. Huffman Codes (Abschnitt 2.2) [10] sind bereits präfixfrei und man muss dem VBL Array lediglich einen Algorithmus geben, mit dem entschieden werden kann, ob eine Bitsequenz valide ist. Andere Daten wie z.B. ein Array aus $\mathbb{N}$ sind wie in Abschnitt 3.1 beschrieben nicht per se präfixfrei, weshalb man solche vor dem Speichern zuerst mit einem präfixfreien Kode z.B. Elias-Gamma (Abschnitt 2.4) oder Elias-Delta (Abschnitt 2.5) Kodes [4] kodieren muss.

Um jetzt das j-te Element zu lesen muss vom Anfang bis zum j-ten Element über das VBL Array iteriert werden, was in einer linearen Zugriffszeit resultiert. Diese Zugriffszeit ist nicht praktikabel. Um weiterhin eine konstante Zugriffszeit zu realisieren wird bei der Konstruktion des VBL Array für jedes k-te Element die absolute Startposition in einem separatem sampling Array S gespeichert. Durch dieses zusätzliche Array kann nun jedes j-te Element durch $j \ mod \ k$ Iterationen gefunden werden. Hierzu startet man an der Position $S[\lfloor n/j \rfloor]$ und iteriert von dieser Position weitere $j \ mod \ k$ Elemente um das j-te Element zu finden. Die Zugriffszeit hierfür ist $O(k)$, da k aber eine Konstante ist, ist die Zugriffszeit wieder konstant. Abbildung 3.1 zeigt das Array $A = [13, 7, 2, 42, 13, 62, 25]$ als Sparse Sampling VBL Array wobei die rot markierten Stellen einen Zugriff auf $A[3]$ beschreiben.

Durch das Sampling wird zusätzlicher Platz benötigt um S zu speichern, wobei es einen Zusammenhang zwischen zusätzlich benötigtem Platz und Zugriffszeit gibt, welcher in der Evaluation genauer analysiert wird.

Tabelle 3.1.: Diese Tabelle Zeigt sowohl den Theoretischen Speicherbedarf, als auch die Theoretische Zugriffszeit für Sparse Sampling VBL Arrays. Dabei ist die Variable $B = \lceil \frac{\sum_{j=0}^{n-1} \ell_j}{w} \rceil \cdot w$ die Anzahl der Bits, welche für das Datenarray benötigt werden, $S = \lfloor n/k \rfloor + 1$ die Anzahl der Sparse Samples, $U = \sum_{j=0}^{n-1} \ell_j + 1$ die Universumsgröße für Elias-Fano und $A = \lfloor \log_2(\sum_{j=0}^{n-1} \ell_j) \rfloor + 1$ die minimale Anzahl an Bits, die benötigt werden um das gesamte Datenarray adressieren zu können.

	Speicherbedarf in Bits	Zugriffszeit
Sparse mit naivem Array	$B + S \cdot 64$	$O(k)$
Sparse mit fixed Array	$B + \lceil \frac{S \cdot A}{w} \rceil \cdot w$	$O(k)$
Sparse mit Elias-Fano	$B + S(2 + \log_2 \lceil \frac{U}{S} \rceil)$	$O(k)$

Die triviale Lösung um das sampling Array S zu speichern wäre ein einfaches Array aus 16, 32 oder 64 Bit Integer je nach Größe der Daten. Daraus ergibt sich aber wieder das Problem eines ineffizient genutzten Speicherplatz durch vermeidbar große Speicherblöcke pro Element in S. Um den Platz von S zu reduzieren kann man statt naiven Integer Arrays Arrays mit Elementen fester Größe(Abschnitt 2.8) [14], die von der Länge des größten Samples abhängen verwenden. Die Größe der Elemente in S beträgt dann $\lfloor \log_2(S[n-1]) \rfloor + 1$ wodurch binär kodiert das letzte Element von S exakt passt und deutlich weniger Platz verschwendet wird. Da S die Startpositionen jedes k-ten Elements in geordneter Reihenfolge enthält und jedes Element größer als null ist, ist S streng monoton wachsend wodurch sichergestellt ist, dass wenn das letzte Element passt jedes vorherige Element auch passt.

Um weiter Platz einzusparen kann ausgenutzt werden, dass S streng monoton ist, wodurch S mit Elias-Fano (Abschnitt 2.7) [3, 5] kodiert werden kann.

Da der Zugriff auf beliebige Elemente sowohl bei einfachen Arrays, Arrays mit Elementen fester Größe, als auch bei Elias-Fano konstant ist, hängt die Zugriffszeit nur von der sample Frequenz k ab. Praktisch haben die drei Methoden aber verschiedene konstante Faktoren um auf Elemente zuzugreifen, weshalb ein Unterschied erkennbar ist, dazu mehr in Kapitel 5.

Der Platzbedarf dagegen hängt sowohl von k als auch von der Art des Arrays S ab. Dabei hat S genau $\lfloor n/k \rfloor + 1$ Einträge. Beim Speichern mit einem trivialen Array ergibt es Sinn möglichst große Integer zu verwenden, um nicht die mögliche Größe des VBL Arrays zu limitieren. Verwendet man z.B. ein Array aus uint64_t, also ein Array aus 64 Bit großen Integern benötigt die Zugriffsdatenstruktur $(\lfloor n/k \rfloor + 1) \cdot 64$ Bits. Bei Arrays mit Elementen fester Größe wird die Größe der Elemente optimiert, indem für jedes Element in S nur $\lfloor \log_2(\sum_{j=0}^{n-1} \ell_j) \rfloor + 1$ Bits verwendet werden. Es werden also genau so viele Bits für jedes Element verwendet, dass das größte Element exakt in einen Arrayeintrag passt, was in einem Speicherbedarf von $\lceil \frac{((\lfloor n/k \rfloor + 1) \cdot (\lfloor \log_2(\sum_{j=0}^{n-1} \ell_j) \rfloor + 1))}{w} \rceil \cdot w$ resultiert. Wenn Elias-Fano verwendet wird kann mit optimaler Parameterwahl für lower Bits und Universumsgröße U ein Platzbedarf von $(\lfloor n/k \rfloor + 1)(2 + \log_2 \lceil \frac{U}{(\lfloor n/k \rfloor + 1)} \rceil)$ Bits erreicht werden. Da diese Arbeit sich nur mit statischen VBL Arrays beschäftigt können die optimalen Parameter immer im Voraus berechnet werden. Dabei ist $U = \sum_{j=0}^{n-1} \ell_j + 1$ und die Größe der lower Bits $low = \lfloor \log_2(n) \rfloor$ (Abschnitt 2.7) [3, 5].

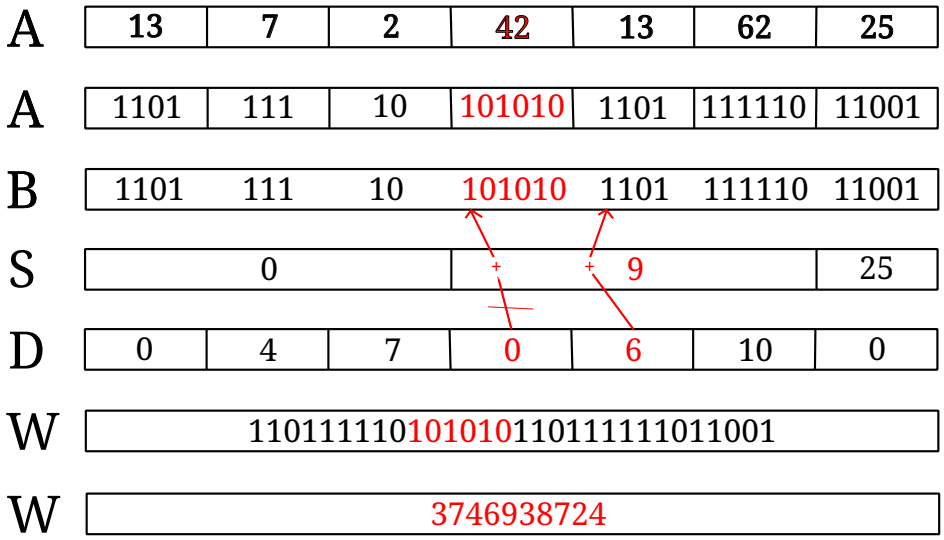

Abbildung 3.2.: Repräsentation des Arrays A als Dense Sampling VBL Array mit $k = 3$, wobei D die Dense Samples und S die Sparse Samples speichert. W ist das Array, das im Endeffekt gespeichert wird. Die rot markierten Elemente repräsentieren die Daten, die bei dem Zugriff auf $A[3]$ relevant sind. Die Startposition in B ergibt sich aus $S[\lfloor 3/k \rfloor] + D[3 \ mod \ k] = S[1] + D[0]$ und die Endposition aus $S[\lfloor 3/k \rfloor] + D[4 \ mod \ k] = S[1] + D[1]$.

3.1.2. Dense Sampling VBL Array

Um die Daten selbst nicht kodieren zu müssen aber vor allem auch um die Zugriffszeit erheblich zu verbessern kann es sinnvoll sein die Startpositionen aller Elemente zu speichern. Da die Startposition des $(j+1)$-ten Elements die Endposition des j-ten Elements ist kann man die relevanten Bits direkt lesen und die Iteration wie in Sparse Sampling fällt weg, wodurch der konstante Faktor k wegfällt.

Die einfachste Lösung dies zu realisieren wäre ein einzelnes Array, in dem alle Startpositionen gespeichert werden. Da die Elemente mit weniger oder gleich 64 Bits aber eher klein sind, ist es sinnvoll nicht für jeden Wert die absolute Position abzuspeichern, da durch die kleinen Elemente aufeinander folgende Elemente eine sehr ähnliche Startposition haben. Dieser relativ geringe Abstand zwischen den Elementen kann dann mit einem zweiten Array, einer zweiten Sampling Ebene, abgebildet werden. Man speichert also um Platz zu sparen nur für jedes k-te Element die absolute Startposition in einem Array S und für alle Elemente j dazwischen nur die relative Position zur nächst gelegenen vorherigen absoluten Startposition $S[\lfloor j/k \rfloor]$ in einem Array D. Abbildung 3.2 zeigt das Array $A = [13, 7, 2, 42, 13, 62, 25]$ als Dense Sampling VBL Array wobei die rot markierten Stellen einen Zugriff auf das dritte Element beschreiben.

Genau wie bei Sparse Sampling kann man auch bei Dense Sampling den Speicherverbrauch verbessern indem man den Platzbedarf der Sparse S und Dense D Sampling Arrays optimiert. Genau wie bei Sparse Sampling ist der erste Ansatz die Verwendung von Arrays mit Elementen fester Größe. Eine weitere Optimierung kann durch Verwendung von Elias-Fano

für die Sparse Samples erzielt werden. Für die Dense Samples kann Elias-Fano nicht sinnvoll verwendet werden, da D nicht streng monoton wachsend bzw. gar nicht monoton ist.

Um auf ein Element j zuzugreifen werden je zwei Anfragen auf S und D gestellt. Die Startposition ergibt sich aus $D[j] + S[j]$ und die Endposition aus $D[j+1] + S[j+1]$. Das Element kann dann mithilfe der Start- und Endposition genau wie bei Arrays mit Elementen fester Größe (Abschnitt 2.8) ausgegeben werden. Da ein Zugriff, unabhängig von der Speichermethode für D und S, immer aus vier Zugriffen auf die sample Arrays (jeweils Start und Endposition für Dense und Sparse Samples) und einem Zugriff auf das Datenarray, in dem die Daten konkateniert abgespeichert sind, besteht, ist der Zugriff auf ein Element offensichtlich konstant.

Der Platzbedarf der Navigationsdatenstruktur lässt sich durch den Platzbedarf von D addiert mit dem Platzbedarf von S berechnen. Dabei hat D genau $\lfloor n/k \rfloor + 1$ Elemente und S hat $n+1$ Elemente. Das „+1“ wird benötigt um für das letzte Element die Endposition zu bestimmen. Daraus ergibt sich für die VBL Arrays mit den trivialen Arrays, angenommen es werden für D uint64_t und für S uint16_t verwendet, ein Platzbedarf von $(\lfloor n/k \rfloor + 1) \cdot 64 + (n+1) \cdot 16$. Dabei darf k allerdings nicht größer als $2^{16}/64 = 2^{16}/2^6 = 2^{10} = 1024$ werden, da sonst innerhalb eines Blocks in D mit der Einschränkung, dass Elemente maximal 64 Bits groß sein dürfen nicht gewährleistet werden kann, dass jedes Element adressiert werden kann. Bei 1025 Elementen der Größe 64 Bits müsste die Startposition für das 1025-te Element noch in S stehen und wäre $1024 \cdot 64 = 65536 = 2^{16}$ was gespeichert in einem uint16_t zu einem Overflow führen würde und somit zu fehlerhaften Ergebnissen. Dieses Problem tritt bei den anderen Methoden nicht auf, da die Elementgröße von S zur Laufzeit passend zu den Daten berechnet wird. Für Arrays mit Elementen fester Größe wird der Platzbedarf für S genau wie bei Sparse Sampling durch $\lceil \frac{((\lfloor n/k \rfloor + 1) \cdot (\lfloor \log_2(\sum_{j=0}^{n-1} \ell_j) \rfloor + 1))}{w} \rceil \cdot w$ berechnet. Der Platzbedarf für D wird durch $\lceil \frac{(n+1) \cdot \lfloor \log_2(\arg\max_{0 \le x < n/k} \sum_{j=x \cdot k}^{x \cdot k - 1} \ell_j) \rfloor + 1}{w} \rceil \cdot w$ berechnet. Der Platzbedarf für Elias-Fano kann für S genau wie bei Sparse Sampling durch $(\lfloor n/k \rfloor + 1)(2 + \log_2 \lceil \frac{U}{(\lfloor n/k \rfloor + 1)} \rceil)$ mit $U = \sum_{j=0}^{n-1} \ell_j + 1$ berechnet werden und da für D kein Elias-Fano angewandt werden kann wird hier auch wieder ein Array mit Elementen fester Größe verwendet. Daraus ergibt sich die Tabelle 3.2:

3.2. Direct Addressable Codes

Im Gegensatz zu sampling VBL Arrays verwenden Direct Addressable Codes einen verschiedenen Ansatz. Direct Addressable Codes nutzen generalisiertes VByte Coding (Abschnitt 2.1) [16] um jedes Element in mehrere Teilstücke aufzuteilen. Dabei wird das Element $A[j]$ aufgeteilt in $T_{m,j} : T_{m-1,j} : \cdots : T_{1,j} : T_{0,j}$. Die einzelnen Teilstücke des V-Byte Codings werden dann in verschiedenen Ebenen gespeichert. Dabei wird jede Ebene als Array D_k abgespeichert wobei k der Tiefe der Ebene entspricht. In D_k wird für jedes Element das k-te Teilstück aller Elemente abgespeichert, die mindestens k Teilstücke haben, also $D_k = \{T_{k,j} | 0 \le j < n \text{ und } T_{k,j} \text{ existiert}\}$. Die relative Ordnung der Elemente

Tabelle 3.2.: Diese Tabelle Zeigt sowohl den Theoretischen Speicherbedarf, als auch die Theoretische Zugriffszeit für alle Sampling VBL Arrays. Dabei ist die Variable $B = \lceil \frac{\sum_{j=0}^{n-1} \ell_j}{w} \rceil \cdot w$ die Anzahl der Bits, welche für das Datenarray benötigt werden, $S = \lfloor n/k \rfloor + 1$ die Anzahl der Sparse Samples, $D = n + 1$ die Anzahl der Dense Samples, $U = \sum_{j=0}^{n-1} \ell_j + 1$ die Universumsgröße für Elias-Fano, $A = \lfloor \log_2(\sum_{j=0}^{n-1} \ell_j) \rfloor + 1$ die minimale Anzahl an Bits, die benötigt werden um das gesamte Datenarray adressieren zu können und $F = \lfloor \log_2(\arg\max_{0 \le x < n/k} \sum_{j=x \cdot k}^{x \cdot k - 1} \ell_j) \rfloor + 1$ die minimale Anzahl an Bits, die benötigt wird um für jeden Block innerhalb des Blocks jeden Eintrag adressieren zu können.

	Speicherbedarf in Bits	Zugriffszeit
Sparse mit naivem Array	$B + S \cdot 64$	$O(k)$
Sparse mit fixed Array	$B + \lceil \frac{S \cdot A}{w} \rceil \cdot w$	$O(k)$
Sparse mit Elias-Fano	$B + S(2 + \log_2 \lceil \frac{U}{S} \rceil)$	$O(k)$
Dense mit naivem Array	$B + S \cdot 64 + D \cdot 16$	$O(1)$
Dense mit fixed Array	$B + \lceil \frac{S \cdot A}{w} \rceil \cdot w + \lceil \frac{D \cdot F}{w} \rceil \cdot w$	$O(1)$
Dense mit Elias-Fano	$B + S(2 + \log_2 \lceil \frac{U}{S} \rceil) + \lceil \frac{D \cdot F}{w} \rceil \cdot w$	$O(1)$

bleibt gleich. Zusätzlich zu den eigentlichen Daten wird in jeder Ebene ein Bitvektor B_k abgespeichert, der angibt ob das j-te Element in der $k+1$-ten Ebene noch vertreten ist $B_k[j] = 1$ oder nicht $B_k[j] = 0$. Um ein Element $A[c]$ mit $0 \le c < n$ zu lesen liest man $D_0[c]$ und anschließend $B_0[c]$. Falls $B_0[c] = 0$ ist der Lesevorgang abgeschlossen und $D_0[c]$ kann ausgeben werden. Anderenfalls wird mit dem 1-Rang an der Stelle c die Postion des Teilstücks in der 1-ten Ebene bestimmt. Der nächsten Teil des Elements kann dann an der Position $D_1[c_1]$ mit $c_1 = B_0.rang_1(c)$ gelesen werden. Auch in dieser Ebene wird mithilfe des Bitvektors $B - 1$ bestimmt, ob eine weitere Ebene gelesen werden muss oder nicht. Im Allgemeinen steht ein Teilstück eines Elements in der k-ten Ebene an der Stelle $c_k = B_{k-1}.rang_1(c_{k-1})$, wobei $c_0 = c$. Das wird für jede Ebene wiederholt, bis $B_k[c_k] = 0$. Dann kann das Element ausgegeben werden durch die Konkatenation der Teilstücke: $A[c] = D_k[c_k] : D_{k-1}[c_{k-1}] : \cdots : D_1[c_1] : D_0[c_0]$.

Abbildung 3.3 zeigt das Array $A = [13, 7, 2, 42, 13, 62, 25]$ als Direct Addressable Codes, wobei die rot markierten Stellen einen Zugriff auf $A[3]$ beschreiben. Verwendet wurde ein V-Byte Coding mit Elementgröße 2. Beim Zugriff auf $A[3]$ wird zuerst in $D0[3]$ die 2 niederwertigsten Bits 10 gelesen. Da $B0[3] = 1$ muss man in der nächsten Ebene die nächsten 2 Bits lesen. Die Position in $D2$ steht an $B0.rang_1(3) = 2$. Dies wird in der nächsten Ebene wiederholt, was die Position $B1.rang_1(2) = 0$ in $D2$ ergibt. Da $B2[0] = 0$ ist der Lesevorgang abgeschlossen und das Ergebnis $D2[0] : D1[2] : D0[3] = 101010$ kann ausgegeben werden.

Direct Addressable Codes können durch das geschickte Anpassen von Teilstückgrößen in den Ebenen weiter optimiert werden. So ist z.B. ein Array mit zehn 5-Bit großen und vier 8-Bit großen Einträgen mit V-Byte Coding, mit Elementgröße von vier Bits, in zwei Ebenen unterteilt. Wenn die erste Ebene eine Größe von fünf Bits und die zweite von drei Bits besitzt, wird Speicherplatz gespart und die Zugriffszeit im Durchschnitt verbessert, da für die zehn 5-Bit großen Einträge nur eine statt zwei Ebenen gelesen werden muss. Im Allgemeinen

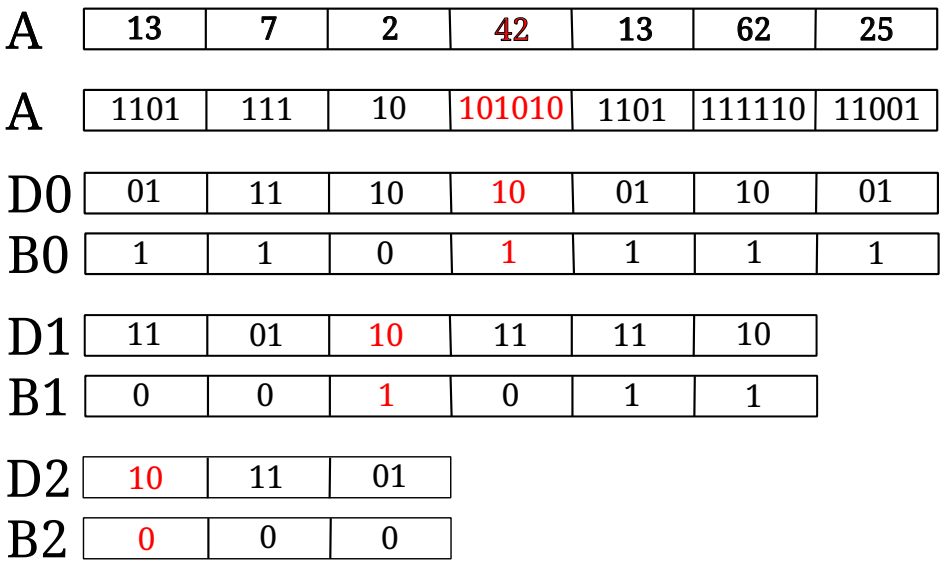

Abbildung 3.3.: Repräsentation des Arrays A als Direct Addressable Codes. Die rot markierten Elemente repräsentieren die Daten, die bei dem Zugriff auf $A[3]$ relevant sind. Um das Element $A[3]$ zu lesen wird in der ersten Ebene das $D1[3]$ gelesen. Dieses Element bildet die niederwertigsten Bits der Ausgabe. Wenn $B1[3] = 1$ muss in die darunterliegende Ebene abgestiegen werden und die Position $p = B1.rang_1(3)$ $D2[p]$ welche die Bits vor den bereits gelesenen Bits bilden abgelesen werden. Wenn $B2[p] = 1$ muss dieser Schritt für die dritte Ebene wiederholt werden. Wiederhole diesen Prozess bis $BX[B(X-1).rang_1(p)] = 0$

ist die Größe und Tiefe der Ebenen aber nicht so wie bei dem Beispiel optimal wählbar, da man immer den durch zusätzliche Ebenen gewonnene Platz gegen die verschlechterte Worst-Case Zugriffszeit abwägen muss. Die Optimierung auf Platz kann dadurch sehr viele relativ kleine Ebenen erzeugen. Um dies zu verhindern und die Worst-Case Zugriffszeit zu limitieren kann man zusätzlich die Tiefe der Ebenen limitieren. Die Zugriffszeit ist also abhängig von der Menge der Ebenen, auf die das gelesene Element verteilt ist. Der Zugriff auf ein Teilstück in einer Ebene erfolgt in konstanter Zeit, da es ein Arrayeintrag und ein Eintrag im entsprechenden Bitvektor ist. Der Platzbedarf für die DaCs hängt von der Größe und Tiefe der Ebenen und von dem Overhead der rank Funktion für den verwendeten Bitvektor ab.

Wenn V-Byte Coding mit Elementen der Größe e für jede Ebene verwendet wird, ist der Platzbedarf nach oben durch $o(en + \frac{\sum_{j=0}^{n-1} \ell_j}{e})$ beschränkt. Die Zugriffszeit beträgt dann $O(occ/e)$ wobei occ die Größe des gesuchten Elements in Bits entspricht.

Daraus ergibt sich Tabelle 3.3:

Tabelle 3.3.: Diese Tabelle Zeigt den Theoretischen Speicherbedarf, als auch die Theoretische Zugriffszeit für alle Implementierten VBL Arrays. Dabei ist die Variable $B = \lceil \frac{\sum_{j=0}^{n-1} \ell_j}{w} \rceil \cdot w$ die Anzahl der Bits, welche für das Datenarray benötigt werden, $S = \lfloor n/k \rfloor + 1$ die Anzahl der Sparse Samples, $D = n + 1$ die Anzahl der Dense Samples, $U = \sum_{j=0}^{n-1} \ell_j + 1$ die Universumsgröße für Elias-Fano, $A = \lfloor \log_2(\sum_{j=0}^{n-1} \ell_j) \rfloor + 1$ die minimale Anzahl an Bits, die benötigt werden um das gesamte Datenarray adressieren zu können und $F = \lfloor \log_2(\arg\max_{0 \le x < n/k} \sum_{j=x \cdot k}^{x \cdot k - 1} \ell_j) \rfloor + 1$ die minimale Anzahl an Bits, die benötigt wird um für jeden Block innerhalb des Blocks jeden Eintrag adressieren zu können.

	Speicherbedarf in Bits	Zugriffszeit
Sparse mit naivem Array	$B + S \cdot 64$	$O(k)$
Sparse mit fixed Array	$B + \lceil \frac{S \cdot A}{w} \rceil \cdot w$	$O(k)$
Sparse mit Elias-Fano	$B + S(2 + \log_2 \lceil \frac{U}{S} \rceil)$	$O(k)$
Dense mit naivem Array	$B + S \cdot 64 + D \cdot 16$	$O(1)$
Dense mit fixed Array	$B + \lceil \frac{S \cdot A}{w} \rceil \cdot w + \lceil \frac{D \cdot F}{w} \rceil \cdot w$	$O(1)$
Dense mit Elias-Fano	$B + S(2 + \log_2 \lceil \frac{U}{S} \rceil) + \lceil \frac{D \cdot F}{w} \rceil \cdot w$	$O(1)$
DaCs mit festen V-Byte Codes	$o(en + \frac{\sum_{j=0}^{n-1} \ell_j}{e})$	$O(occ/e)$

4. Implementierung

In diesem Kapitel wird die Implementierung der einzelnen VBL Arrays beschrieben. Dabei werden die einzelnen Designentscheidungen erläutert und begründet.

4.1. Sampling VBL Arrays

4.1.1. Datenarray

Die Grundlage von allen Sampling VBL Arrays ist das Datenarray um die Daten komprimiert zu speichern. Um dies zu erreichen werden im Speicher für jedes Element nur die notwendigen Bits, welche benötigt werden, dass keine Information verloren geht, konkateniert abgespeichert. Dabei werden im Datenarray dieser Arbeit nur Elemente der Länge $1 \leq \ell \leq 64$ Bits unterstützt.

Die Grundlage des Datenarrays bietet ein `std::vector` W aus vorzeichenlosen Integern. Dabei kann die Bit-Größe w der Integer frei zwischen uint64_t, uint32_t, uint16_t und uint8_t als Template gewählt werden. Da in einem `std::vector` jeder Eintrag ein vielfaches des verwendeten Datentyps sein muss fällt am Ende des `std::vector` ein Padding an was in Abbildung 4.1 verdeutlicht wird. Da das größtmögliche benötigte Padding durch $w - 1$ beschränkt ist fällt dieses bei größeren VBL Arrays nicht ins Gewicht. Dazu kommt, dass bei kleineren w die Wahrscheinlichkeit, dass ein Element auf der Grenze liegt deutlich erhöht ist, was eine komplexere Zugriffsoperation erfordert, weshalb in dieser Arbeit immer uint64_t als zugrunde liegender Datentyp verwendet wird. Bezogen auf die verwendeten Benchmark Daten beträgt der zusätzliche Platzbedarf von einem angenommenen worst Case Padding von 63 Bits in etwa $1,6 \cdot 10^{-9}$%, was vernachlässigbar ist.

Implementierung der Push Back Funktion.

Um das Datenarray zu befüllen wird eine `push_back` Funktion unterstützt. Diese Funktion nimmt einen den Wert x als uint64_t und die Länge des einzufügenden Elements ℓ als uint8_t. Die einzufügenden Bits sind dann die ℓ niederwertigsten Bits von x. Ein neues Element wird immer am Ende der bereits eingefügten Elemente platziert. Die Menge der bereits verwendeten Bits wird in einem uint64_t b abgespeichert. Beim Einfügen entstehen zwei verschiedene Fälle:

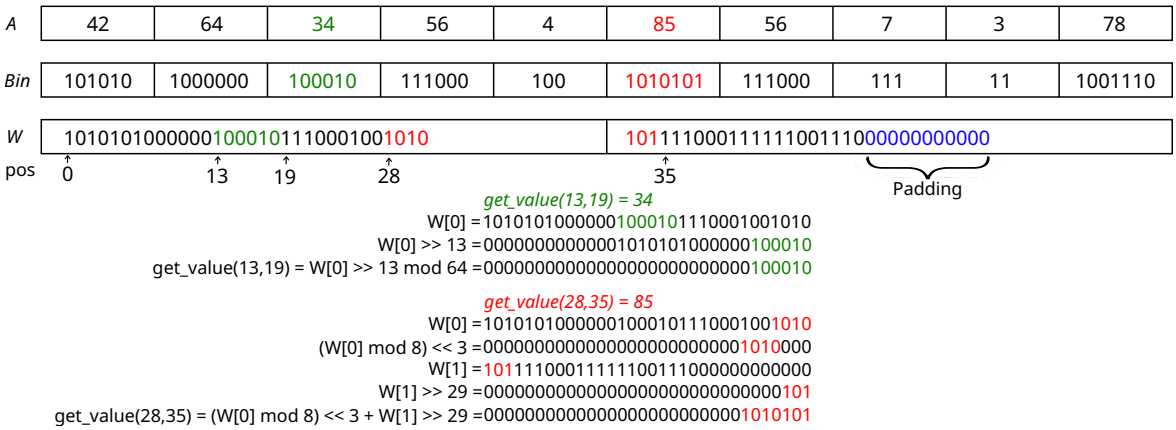

Abbildung 4.1.: Beispiel eines Datenarrays mit $w = 32$, welches das Array $A = [42, 64, 34, 56, 4, 85, 56, 7, 3, 78]$ konkateniert abspeichert. Die blau markierten Bits zeigen das benötigte 11 Bit Padding, um W auf ein vielfaches von w aufzufüllen. Die `get_value` Funktionen beschreiben zwei Zugriffe auf das Datenarray, einen innerhalb eines Blocks (grün) und einen, welcher Zugriff auf zwei Blöcke benötigt (rot)

Fall 1 Das Element ist vollständig in einem Eintrag in W enthalten also $\lfloor b/w \rfloor = \lfloor (b+\ell)/w \rfloor$. In diesem Fall wird der Eintrag in $W[\lfloor b/w \rfloor]$ geschrieben. Da W nicht mit einem Wert initialisiert wird, wird zuerst durch $W[\lfloor b/w \rfloor] = W[\lfloor b/w \rfloor] \& (2^\ell - 1 \ll r)$ der Bereich in dem x geschrieben wird auf 0 gesetzt. Danach kann der Bereich durch $W[\lfloor b/w \rfloor] = W[\lfloor b/w \rfloor] | x \ll r$ gesetzt werden. Dabei ist r durch $(b + \ell)\ mod\ w$ gegeben und gibt das benötigte Padding am Ende von W an.

Fall 2 Das Element erstreckt sich über mehrere Einträge in W also $\lfloor b/w \rfloor \neq \lfloor (b+\ell)/w \rfloor$. In diesem Fall ist das Element über die Einträge $W[\lfloor b/w \rfloor]$ bis $W[\lfloor (b+\ell)/w \rfloor]$ verteilt. Wie bei Fall 1 beschreibt $r = (b + \ell)\ mod\ w$ das benötigte Padding am Ende von W. Zuerst wird das Ende des ersten Eintrags von W wie bei Fall 1 auf 0 gesetzt und die höchstwertigsten Bits des Elements eingefügt $W[\lfloor b/w \rfloor] = W[\lfloor b/w \rfloor] \mid x >> ((\lfloor (b+\ell)/w \rfloor - \lfloor b/w \rfloor - 1) \cdot w + r)$. Die Einträge $W[\lfloor b/w \rfloor + 1]$ bis $W[\lfloor (b+\ell)/w \rfloor] - 1$ werden vollständig von dem Element verwendet und können im folgenden durch $W[pos] = x >> ((\lfloor (b+\ell)/w \rfloor - pos - 1) \cdot w + r)$ mit $pos \in [\lfloor b/w \rfloor + 1, \lfloor (b+\ell)/w \rfloor - 1]$ gesetzt werden. Die niederwertigsten Bits des Elements werden wie in Fall 1 in $W[\lfloor (b+\ell)/w \rfloor]$ gesetzt.

Implementierung der Get Value Funktion.

Die zweite Funktion die das Datenarray unterstützt, `get_value`, ist eine Zugriffsfunktion um auf Bitsequenzen des Datenarrays zuzugreifen. Dabei nimmt die Funktion zwei uint64_t, die Start- s und die Endposition e, wobei die Startposition inklusive und die Endposition exklusive ist. Die Ausgabe der Funktion ist ein uint64_t dessen $e - s$ niederwertigste Bits dem Element entsprechen. Auch in dieser Funktion wird wieder zwischen zwei Fällen unterschieden, welche in Abbildung 4.1 verdeutlicht sind:

Fall 1 Das Element ist vollständig in einem Eintrag in W enthalten also $\lfloor s/w \rfloor = \lfloor e/w \rfloor$. In diesem Fall kann das Element x durch $x = W[s/w] >> r \ mod \ 2^{e-s}$ berechnet und ausgegeben werden.

Fall 2 Das Element erstreckt sich über mehrere Einträge in W also $\lfloor s/w \rfloor \neq \lfloor e/w \rfloor$. In diesem Fall muss das Element aus den verschiedenen Teilen, welche über die Einträge $W[\lfloor s/w \rfloor]$ bis $W[\lfloor e/w \rfloor]$ verteilt sind, rekonstruiert werden. Dazu wird die Ausgabe *out* mit den höchstwertigsten Bits des Elements initialisiert $out = W[\lfloor s/w \rfloor] \ mod \ 2^{\ell-((\lfloor e/w \rfloor - \lfloor s/w \rfloor - 1)\cdot w - r)} \cdot 2^{(\lfloor e/w \rfloor - \lfloor s/w \rfloor - 1)\cdot w + r}$. Für die Teilstücke, die vollständig einen Eintrag ausfüllen, müssen diese nur an die richtige Position geshiftet werden und auf *out* addiert. $out = out + W[pos] \ll (r + (\lfloor e/w \rfloor - pos - 1)\cdot w)$ mit $pos \in [\lfloor s/w \rfloor + 1, \lfloor e/w \rfloor - 1]$. Die niederwertigsten Bits stehen in $W[\lfloor e/w \rfloor]$ und können mit $out = out + W[\lfloor e/w \rfloor] >> (w - r)$ zu *out* hinzugefügt werden, womit die Rekonstruktion abgeschlossen ist und *out* als Lösung ausgegeben werden kann.

4.1.2. Arrays mit Elementen fester Größe

Wie in Kapitel 3 beschrieben können Sampling VBL Arrays durch Arrays mit Elementen fester Größe platzeffizienter abgespeichert werden. Die Implementierung von Arrays mit Elementen fester Größe basiert in dieser Arbeit auf den Datenarrays aus Abschnitt 4.1.1. Bei der Initialisierung wird entweder die Elementgröße als Parameter übergeben und in einem uint8_t gespeichert oder es wird ein Array übergeben aus dem das größte Element gesucht wird, dessen Elementgröße berechnet und diese als Elementgröße in einem uint8_t gespeichert wird. Anschließend werden in dieser Methode alle Einträge eingefügt. Arrays mit Elementen fester Größe implementieren in dieser Arbeit im wesentlichen zwei Funktionen: `push_back` und der Zugriffsoperator.

Grundlegend speichert das Array mit Elementen fester Größe ein Datenarray (Abschnitt 4.1.1) und zusätzlich die Elementgröße g und die Länge des Arrays n. Die `push_back` Funktion fügt am Ende der Datenarrays ein neues Element der Größe g hinzu. Da bekannt ist, wie groß jedes Element ist kann die Start- s und Endposition e bei einem Zugriff auf das i-te Element durch $s = i \cdot g$ und $e = (i + 1) \cdot g$ berechnet werden und das Element kann durch `get_value(s,e)` auf das Datenarray ausgegeben werden.

4.1.3. Elias-Fano Kodierung

Für die Implementierung von Elias-Fano (Abschnitt 2.7) [3, 5] wurde zum größten Teil auf eine Bibliothek [12] zugegriffen. An dieser Version von Elias-Fano wurden in dieser Arbeit nur zwei kleine Änderungen vorgenommen.

Die erste Änderung verschiebt die lower Bits Konstante, welche beschreibt, wie viele der niederwertigsten Bits trivial abgespeichert werden, von einem Template in den Konstruktor. Diese Änderung war nötig, damit der optimale Wert für die Länge der lower Bits zur Laufzeit für den entsprechenden Datensatz berechnet werden kann.

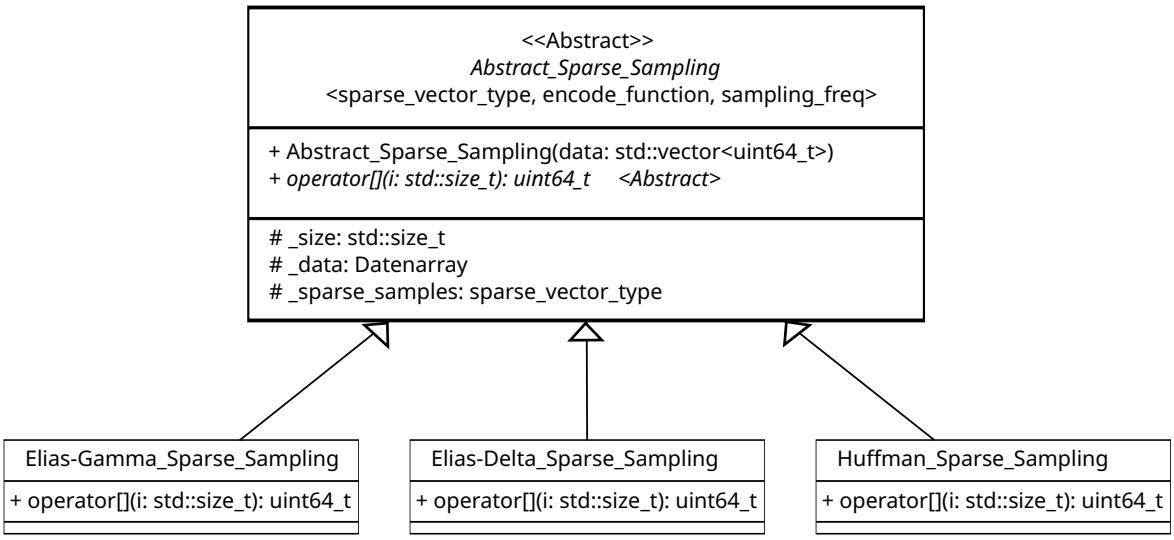

Abbildung 4.2.: Klassenstruktur der Implementierung für Sparse Sampling als UML Diagramm

Die zweite Änderung verwendet die in dieser Arbeit erstellte Implementierung von Arrays mit Elementen fester Größe anstatt von sdsl-lite [7], da diese Version die Elementgröße als Template nimmt und durch die erste Änderung diese nicht mehr zum Kompilierzeitpunkt bekannt ist.

4.1.4. Sparse Sampling

Um Sparse Sampling für beliebige präfixfreie Kodierungen zu unterstützen, und Code Dopplungen zu vermeiden, ist Sparse Sampling in eine abstrakte Oberklasse und kodierungsspezifische Unterklassen unterteilt. Die abstrakte Oberklasse bekommt dabei eine Kodierungsfunktion und die Art des sampling Arrays als Template gegeben und kann damit das VBL Array vollständig konstruieren. Die Unterklassen erben von der abstrakten Oberklasse und müssen nur den Zugriffsoperator implementieren, welcher abhängig ist von der gewählten Kodierung. In Abbildung 4.2 wird dieses Verhältnis nochmals verdeutlicht.

Die abstrakte Oberklasse ist verantwortlich für die Konstruktion und das Speichern des VBL Arrays. Dafür nimmt sie die Sampling Frequenz k, den Typ des Arrays in dem die Samples gespeichert werden sollen und die Kodierungsfunktion als Template. Um das VBL Array zu speichern werden ein Sampling Array in dem gegebenen Arraytyp, die Größe in einem uint64_t und die Daten selbst in einem Datenarray (Abschnitt 4.1.1) gespeichert. In der Konstruktion wird das zu speichernde Array als Referenz auf einen `std::vector<uint64_t>` übergeben. Über dieses Array wird dann im Konstruktor iteriert und für jedes Element zuerst die Kodierungsfunktion angewendet und dann das kodierte Element in dem Datenarray abgespeichert. Zusätzlich wird für jedes k-te Element die absolute Startposition am Ende des Sampling Arrays abgespeichert.

Die Unterklassen erweitern die Oberklasse um die Kodierungsfunktion und den Zugriffsoperator. Dabei verwendet die Unterklasse den Konstruktor der abstrakten Oberklasse

und implementiert lediglich den Zugriffsoperator selbst. Das grundlegende Prinzip des Zugriffsoperators ist bei allen Sparse Sampling VBL Arrays das Gleiche: Zuerst wird aus der gewünschten Position i und der Sampling Frequenz k das größte Sample s, welches auf ein Element vor oder auf das Element selbst zeigt, durch $s = \lfloor i/k \rfloor$ berechnet. Danach kann beginnend von diesem Element über die nächsten $i \bmod k$ Elemente iteriert werden bis in der $i \bmod k$-ten Iteration das gewünschte Element gelesen wird, welches dann dekodiert und ausgegeben werden kann. Wie über die Elemente iteriert werden kann unterscheidet sich von Kodierung zu Kodierung und wird im Folgenden für die implementierten Kodierungen Elias-Gamma, Elias-Delta und für Huffman Codes beschrieben.

Implementierung von Sparse Sampling mit Elias-Gamma.

Die einfachste Lösung für die Iteration über Elias-Gamma Codes ist für jeden Schritt beginnend von dem Ende des vorherigen Elements die nächsten 64-Bit aus dem Datenarray zu lesen und dieses zu dekodieren. Da die Länge der Elias-Gamma Codes durch die Anzahl der Nullen z vor der ersten 1 genau durch $\ell = z \cdot 2 + 1$ berechnet werden kann und die Codes bei dieser Methode nach dem Lesen aus dem Datenarray immer genau an den höchstwertigsten Bits eines uint64_t stehen, kann mit der `countl_zero` Instruktion genau z ausgelesen werden und die Startposition für das nächste Element aus der bisherigen Startposition + ℓ berechnet werden. Dabei ist `countl_zero` eine in der C++ Standardbibliothek verfügbare Instruktion, die die Anzahl der Nullen vor der ersten 1 ausgibt beginnend bei den höchstwertigsten Bits. Dies kann wiederholt werden bis zur $i \bmod k$-ten Iteration bei der nach dem Lesen von z nicht die Startposition erhöht wird sondern das Element selbst durch Lesen der Sequenz $start$ = Startposition, $end = start + \ell$ aus dem Datenarray mit `get_value(start, end)` gelesen wird. Die `get_value(start, end)` gibt dann das kodierte Element an der rechten Grenze eines uint64_t ausgerichtet aus. Da der uint64_t sowieso von links mit 0-en aufgefüllt wird und Elias-Gamma Codes nur 0-en vor der eigentlichen binären Kodierung hinzufügt, muss dieser Wert nicht weiter dekodiert werden und kann so ausgegeben werden.

Da diese Methode aber für jeden Iterationsschritt erneut auf das Datenarray zugreift und damit sehr häufig einen Speicherzugriff erzwingt ist diese Methode sehr ineffektiv. Gerade unter dem Aspekt, dass die Elemente in VBL Arrays häufig sehr klein sind werden jedes Mal die meisten geladenen Bits nicht verwendet und in der nächsten Iteration erneut um ℓ Bits nach links verschoben in den momentan betrachteten Abschnitt geladen.

Um diese Speicherzugriffe auf das Datenarray zu minimieren, werden, wie auch in Abbildung 4.3 abgebildet, in der ersten Iteration die ersten 64-Bit geladen, welche temporär in einem uint64_t $temp$ abgespeichert werden. Um weiterhin die `countl_zero` Instruktion verwenden zu können wird dieser Wert nach jedem Iterationsschritt um ℓ nach links geshiftet $temp = temp \ll \ell$. In der nächsten Iteration kann temp dann weiter verwendet werden ohne erneut auf das Datenarray zugreifen zu müssen. Um weiterhin über Sequenzen iterieren zu können, die länger als 64-Bit sind wird in jeder Iteration überprüft ob $temp == 0$, was bedeuten würde, dass das nächste Element nur zum Teil in $temp$ enthalten ist. Erst jetzt werden die nächsten 64-Bit aus dem Datenarray geladen und wieder in $temp$ abgelegt. Das

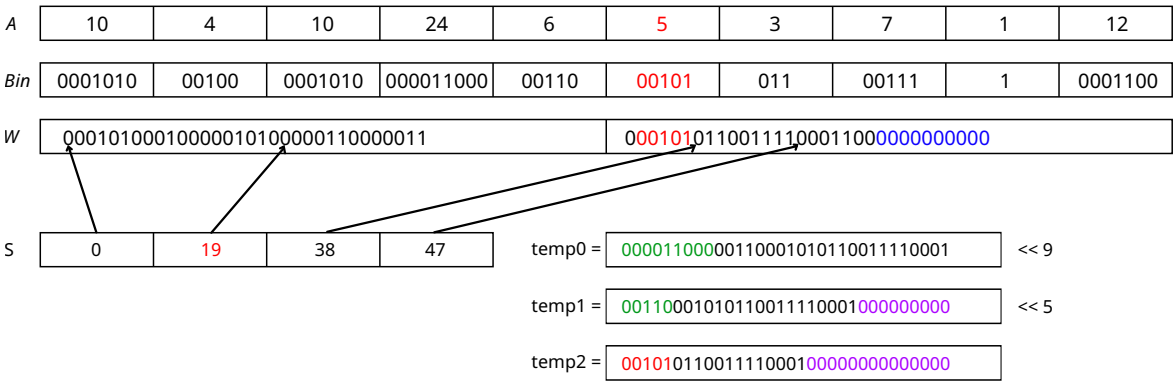

Abbildung 4.3.: Beispiel eines Sparse Sampling VBL Arrays mit $w = 32$, welches das Array $A = [10, 4, 10, 24, 6, 5, 3, 7, 1, 12]$ Elias-Gamma kodiert in W und S abspeichert. Dabei is W ein Datenarray und S das Array, welches die Sparse Samples enthält mit $k = 3$. Die blau markierten Bits zeigen das benötigte 10 Bit Padding, um W auf ein vielfaches von w aufzufüllen. Die $temp0$, $temp1$, $temp2$ Variable zeigen den Zustand der $temp$ Variable ($temp$ hier nur ein uint32_t) zum 0-ten, 1-ten und 3-ten Iterationsschritt. Die grün markierten Bits stellen das gelesene Element dar, das noch nicht dem gesuchten Element entspricht. Die lila markierten Bits markieren den Teil der $temp$ Variable, welcher durch die shift Operation aufgefüllt wurde und die rot markierten Bits das gesuchte Element.

auszugebende Element wird dann wie in der einfachen Lösung durch die errechnete Start- und Endposition gelesen und ausgegeben.

Implementierung von Sparse Sampling mit Elias-Delta.

Die Lösung für die Iteration über Elias-Delta Codes wird durch die etwas komplexere Kodierung zwar etwas komplizierter, bleibt aber im Prinzip gleich wie bei Elias-Gamma Codes.

Wie bei Elias-Gamma Codes kann durch eine $temp$ Variable, welche den momentanen Abschnitt von bis zu 64 Bit abbildet auf dem momentan iteriert wird, Zugriffe auf das Datenarray eingespart werden. Dabei ändert sich durch die hier verwendete Kodierung das Lesen eines Elements und die Bedingung, wann ein neuer 64 Bit Abschnitt in $temp$ geladen werden muss. Um ein Element zu lesen muss zuerst erneut mithilfe der `countl_zero` Instruktion die Anzahl der 0-en z bis zur ersten 1 gelesen werden. Anders als bei Elias-Gamma kann aus z aber nicht direkt die Größe des Elements berechnet werden, da z nur die Länge des Präfixes des Elements ist, welcher dann die Länge des Elements selbst binär codiert. Nachdem z gelesen wurde müssen also die nächsten $z+1$ Bits gelesen werden, welche dann als die Länge des Elements selbst c interpretiert werden. Die Länge des gesamten kodierten Elements ergibt sich dann aus $\ell = 2 \cdot z + 1 + c$.

Wie bei Elias-Gamma kann $temp$ nach dem Lesen von ℓ um ℓ nach links geshiftet werden und mit der nächsten Iteration fortgesetzt werden. Ein neuer Abschnitt in $temp$ muss geladen werden, wenn die Elias-Gamma kodierte Länge des Elements nicht mehr vollständig in $temp$ enthalten ist. Dies ist der Fall, wenn $2 \cdot z + 1 > 64 - shift$ wobei $shift$ eine zusätzlich

abgespeicherte Zahl ist, die repräsentiert, wie weit *temp* bereits nach links geshiftet wurde. In der letzten Iteration werden wie bei Elias-Gamma die Grenzen berechnet und das kodierte Element x aus dem Datenarray extrahiert. Um Elias-Delta dann zu dekodieren müssen die Bits am Anfang, die die Länge repräsentieren, noch entfernt werden. Dies kann durch $out = x \ mod \ 2^c$ erzielt werden, *out* kann dann als Ergebnis ausgegeben werden.

Implementierung von Sparse Sampling über Huffman Codes.

Im Gegensatz zu Elias-Gamma oder Elias-Delta Codes kann bei Huffman Codes die Länge eines Elements nicht durch einen simplen Algorithmus bestimmt werden.

Um überprüfen zu können, ob ein Code ein valider Huffman Code für die in dem VBL Array gespeicherten Daten ist, muss hierfür zusätzlich eine Liste an validen Huffman Codes übergeben werden. Um effizient überprüfen zu können ob eine Sequenz ein valider Huffman Code ist wird diese Liste in einem unordered_set u gespeichert. Zusätzlich wird ein aufsteigend sortiertes Array gespeichert, das die Längen der validen Codes enthält um nicht vorhandene Längen nicht überprüfen zu müssen. Da die Alphabete von Texten nur konstant groß sind, ist auch die Liste der Huffman Codes nur konstant groß was eine konstante Validierung ermöglicht.

Bei der Iteration über das Array wird dann für jeden Iterationsschritt, angefangen bei der kürzesten Länge bis zu längsten Länge, überprüft, ob die entsprechende Bitsequenz in u enthalten ist. Falls sie enthalten ist wird die Startposition erhöht, *temp* um ℓ nach links geshiftet und mit dem nächsten Iterationsschritt begonnen. Falls sie nicht enthalten ist, wird der Schritt mit der nächst größeren Länge wiederholt, bis ein valider Code gefunden wird.

Wie bei Elias-Gamma und Elias-Delta Codes wird auch hier der momentan bearbeitete Abschnitt von 64 Bits in *temp* geladen um Speicherzugriffe einzusparen. Ein neuer Abschnitt wird geladen wenn $shift + \ell > 64$ wobei $shift$ eine zusätzlich abgespeicherte Zahl ist, die repräsentiert, wie weit *temp* bereits nach links geshiftet wurde. Zusätzlich haben Huffman Codes die Besonderheit, dass die Codes sowohl mit 0-en anfangen als auch mit 0-en enden können. Da das Array immer als `std::vector<uint64_t>` übergeben wird könnte der Konstruktor nicht wissen wie groß die Elemente sind. Daher sind die Huffman Codes in dem übergebenen Array mit einer führenden 1 kodiert. Der Konstruktor kann damit genau entscheiden welche Bits für den Huffman Code relevant sind und die Kodierungsfunktion kann die führende 1 entfernen um keinen Speicher zu verschwenden.

4.1.5. Dense Sampling

Dense Sampling VBL Arrays nehmen wie Sparse Sampling VBL Arrays die Sampling Frequenz k für die Sparse Sample, den Arraytyp für die Sparse und Dense Sampling Arrays, als Template. Zusätzlich nehmen sie eine Kodierungsfunktion und eine Dekodierungsfunktion als Template, welche für Huffman Codes relevant werden, damit auch hier die führende 1

nicht mitgespeichert wird. Für Ganzzahlen wird keine weitere Kodierungsfunktion benötigt, weshalb diese in solchen Fällen die Identität darstellt.

Um ein Dense Sampling VBL Array zu konstruieren muss einmal über den gegebenen Datenvektor `std::vector<uint64_t>` iteriert werden. Dabei wird jedes Element mit der Kodierungsfunktion kodiert und dann in ein zugrunde liegendes Datenarray abgespeichert. Um die Sparse und Dense Samples korrekt zu berechnen und abzuspeichern, werden zusätzlich zwei temporäre Variablen *abs* und *rel* abgespeichert. *abs* stellt die absolute Startposition des momentan eingefügten Elements dar und *rel* die relative Startposition zu der letzten absoluten Startposition. Wird ein Element eingefügt wird für jedes Element *rel* in das Dense Sampling Array D eingefügt. Wenn $abs \bmod k = 0$ wird zusätzlich *abs* in das Sparse Sampling Array S eingefügt und *rel* auf 0 gesetzt. Nach dem letzten Element wird zusätzlich ein Sample in das Dense Sampling Array und falls $abs \bmod k = 0$ auch in S eingefügt um die Endposition des letzten Elements zu speichern.

Der Zugriffsoperator auf ein beliebiges Element i kann dann die absolute Startposition s durch $s = S[\lfloor i/k \rfloor] + D[i]$ und die absolute Endposition e durch $e = S[\lfloor (i+1)/k \rfloor] + D[i+1]$ berechnen. Das Element kann dann durch die `get_value(s,e)` Funktion aus dem Datenarray gelesen werden und mit der Dekodierungsfunktion wieder rekonstruiert und ausgegeben werden.

4.2. Direct Addressable Codes

Die Implementierung von Direct Addressable Codes wurde von der Open Source Implementierung von [2] übernommen. Da diese Implementierung in C geschrieben ist, wurde für diese Arbeit lediglich die Implementierung in C++ eingebunden.

5. Evaluation

In diesem Kapitel wird die experimentelle Evaluation der verschiedenen Implementierungen erläutert.

5.1. Benchmark Setup

Alle Benchmarks wurden auf einem Server mit folgenden Spezifikationen ausgeführt: Der Prozessor ist ein Intel i7 6700k auf 4.8 GHz übertaktet auf einem MSI Z170A M5 Mainboard mit Vengance LPX DDR4 Ram 32 GB im Dual Channel welcher mit 3200 MHZ getaktet ist. Das Betriebssystem des Servers ist ein Ubuntu Version 24.04.1. Prgrammiert wurde in c++23 mit dem Compiler gcc Version 13.2.0 mit den compile Flags: -03, -DNDEBUG, -s.

5.2. Durchgeführte Benchmarks

Um die verschiedenen Implementierungen zu vergleichen wurden je drei verschiedene Benchmarks auf 8 verschiedenen Datensätzen ausgeführt. Dabei besteht die Grundlage der Datensätze aus vier 100 MB großen Dateien, die jeweils verschiedene Daten beinhalten. DNA enthält DNA Sequenzen, Proteins enthält Aminosäuresequenzen für Proteine, English enthält einen englischen Text und Sources enthält einen Teil des Source Codes der Standardbibliotheken von C und Java [9]. Von diesen Datensätzen wurden jeweils die Huffman Codierung und das LCP-Array berechnet, mit denen dann die VBL Arrays gebenchmarkt wurden.

Der erste ausgeführte Benchmark, *yield*, misst die Performance um das ursprüngliche Array aus dem VBL Array in der richtigen Reihenfolge vollständig auszulesen.

Der zweite ausgeführte Benchmark, *k random access*, misst die Zugriffszeit für k Zugriffe auf zufällig gewählte Indexe. Dabei ist k ein fest gewählter Parameter der im Rahmen dieser Arbeit zwischen 10 und 10.000.000 mit 100.000-er Schritten liegt. Um alle VBL Arrays unter exakt gleichen Bedingungen zu testen ist der Zufall kein echter Zufall sonder Pseudozufall mit k als Seed.

Der dritte und letzte ausgeführte Benchmark, *k random access dependent*, ist eine Abwandlung von *k random access* und fügt für jeden Zugriff eine Abhängigkeit auf den vorherigen Zugriff hinzu um Pipelining in der CPU zu unterbinden.

Neben der Zugriffszeit auf die VBL Arrays ist der Platzbedarf ebenfalls relevant, weshalb in dieser Arbeit der Platzbedarf für jede Implementierung gemessen und in Verhältnis zu der Zugriffszeit gesetzt wird.

5.3. Zugriffszeit im Verhältnis zu Platzbedarf

In diesem Abschnitt wird der Platzbedarf und die Zugriffszeit in Verhältnis gesetzt und die VBL Arrays so miteinander verglichen. Dabei werden jeweils die Pareto Fronten der verschiedenen VBL Arrays zuerst für die Source LCP Instanz und darauffolgend für die Source Huffman Instanz evaluiert.

Pareto Fronten für die LCP Instanz.

Aus Abbildung 5.1 geht hervor, dass Direct Addressable Codes für LCP Arrays, also Arrays, die nicht von sich aus präfixfrei sind, die optimale Lösung sowohl für Platzbedarf als auch für die Zugriffszeit bieten. Selbst die platzoptimalste Version von Direct Addressable Codes übertrifft sowohl in Zeit- als auch in Platzbedarf alle anderen gebenchmarkten VBL Arrays.

Bei den Sparse Sampling VBL Arrays besteht ein klarer antiproportionaler Zusammenhang. Je mehr Platz das VBL Array benötigt, desto weniger Zeit wird für den Zugriff benötigt und umgekehrt. Dabei sind die schnellsten Versionen von Sparse Sampling, welche den gleichen Platzbedarf haben wie die Dense Samples, nur wenig langsamer in der Zugriffszeit.

Bei den Dense Sampling VBL Arrays zeigt sich wie erwartet kein nennenswerter Zusammenhang zwischen Zugriffszeit und Platzbedarf.

Wie in Abbildung 5.2 zu erkennen macht es keinen nennenswerten Unterschied, welche Instanz gebenchmarkt wurde, die Ergebnisse bleiben über alle Instanzen sehr ähnlich und die Interpretation der Ergebnisse bleibt gleich.

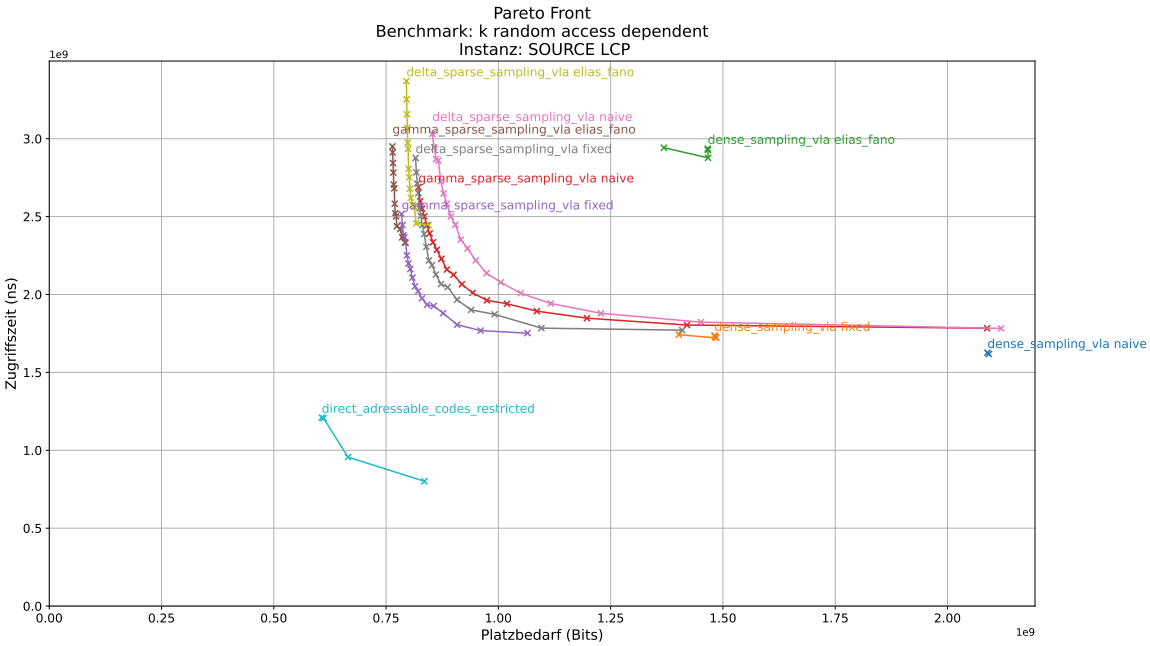

Abbildung 5.1.: Diese Abbildung zeigt die Pareto Fronten der verschiedenen VBL Arrays für den *k random access dependent* Benchmark auf die LCP Source Instanz, wobei die Zugriffszeit über den Platzbedarf abgebildet wird. Dabei sind die einzelnen Punkte in einem Plot die verschiedenen Sampling Frequenzen bzw. Levelbegrenzungen der VBL Arrays.

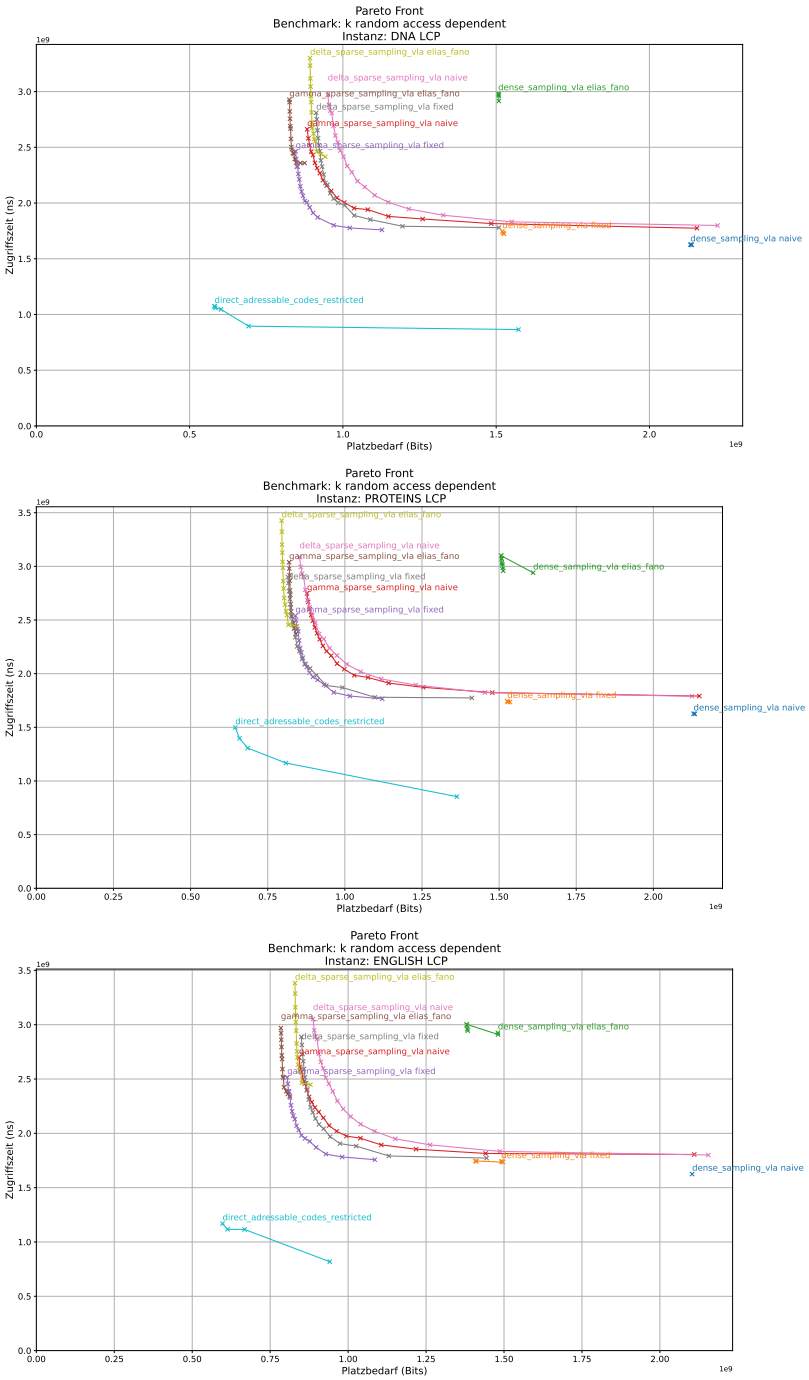

Abbildung 5.2.: Diese Abbildungen zeigen die Pareto Fronten der verschiedenen VBL Arrays für den *k random access dependent* Benchmark auf die LCP DNA, Proteins und English Instanzen, wobei die Zugriffszeit über den Platzbedarf abgebildet wird. Dabei sind die einzelnen Punkte in einem Plot die verschiedenen Sampling Frequenzen bzw. Levelbegrenzungen der VBL Arrays.

Pareto Fronten für die Huffman Instanz.

Im Gegensatz zu den LCP Instanzen zeigt sich bei den Huffman Instanzen kein klarer Sieger. Während die Direct Addressable Codes erneut die optimale Lösung für die Zugriffszeit bilden, können Sparse Sampling VBL Arrays durch das Wegfallen des zusätzlichen Platzbedarfs durch die Kodierung, Direct Addressable Codes im Platzbedarf schlagen. Abbildung 5.3 zeigt, dass die platzoptimale Version von Sparse Sampling ca. 50% weniger Platz benötigt als Direct Addressable Codes während Direct Addressable Codes ca. 4 mal so schnell ist wie Sparse Sampling.

Abbildung 5.3 demonstriert erneut den antiproportionalen Zusammenhang zwischen Zeit- und Platzbedarf für Sparse Sampling VBL Arrays. Die Dense Sampling VBL Arrays sind hier zwar deutlich schneller als die Sparse Sampling VBL Arrays können aber weder in Zugriffszeit noch in Platzbedarf mit Direct Addressable Codes mithalten.

Auch in dieser Instanz zeigen sich keine nennenswerte Zusammenhänge zwischen Zugriffszeit und Platzbedarf bei den Dense Sampling VBL Arrays.

Auch für die verschiedenen Huffman Instanzen kann in Abbildung 5.4 erkannt werden, dass es keinen nennenswerten Unterschied zwischen den verschiedenen Instanzen gibt und die Interpretation für jede Instanz gleich bleibt.

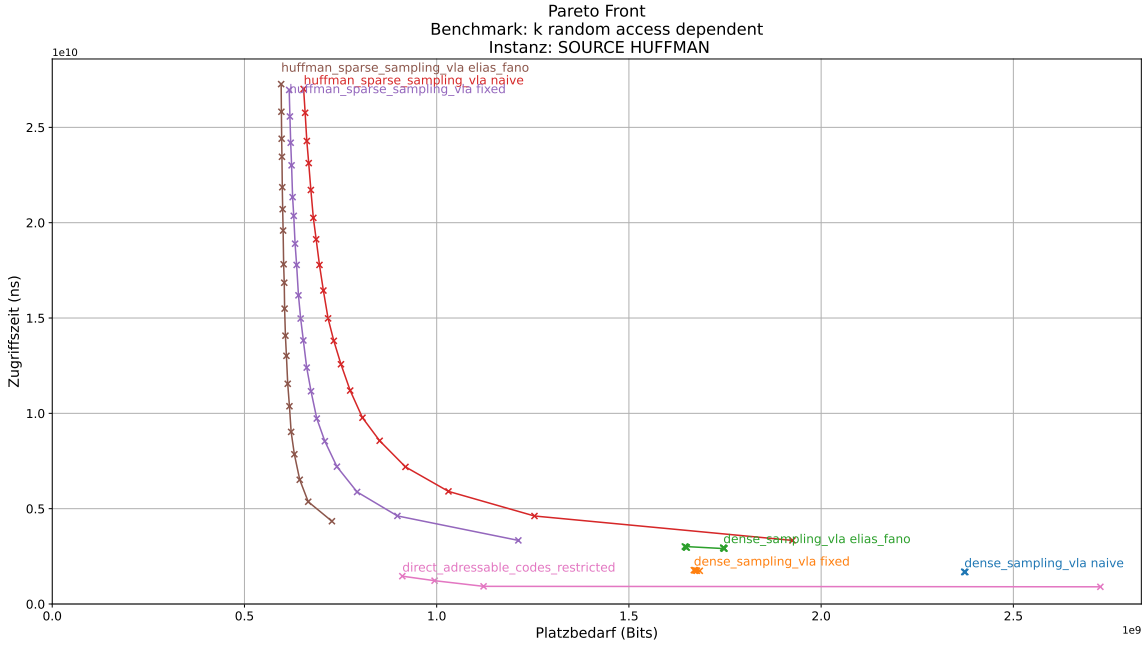

Abbildung 5.3.: Diese Abbildung zeigt die Pareto Fronten der verschiedenen VBL Arrays für den *k random access dependent* Benchmark auf die Huffman Source Instanz, wobei die Zugriffszeit über den Platzbedarf abgebildet wird. Dabei sind die einzelnen Punkte in einem Plot die verschiedenen Sampling Frequenzen bzw. Levelbegrenzungen der VBL Arrays.

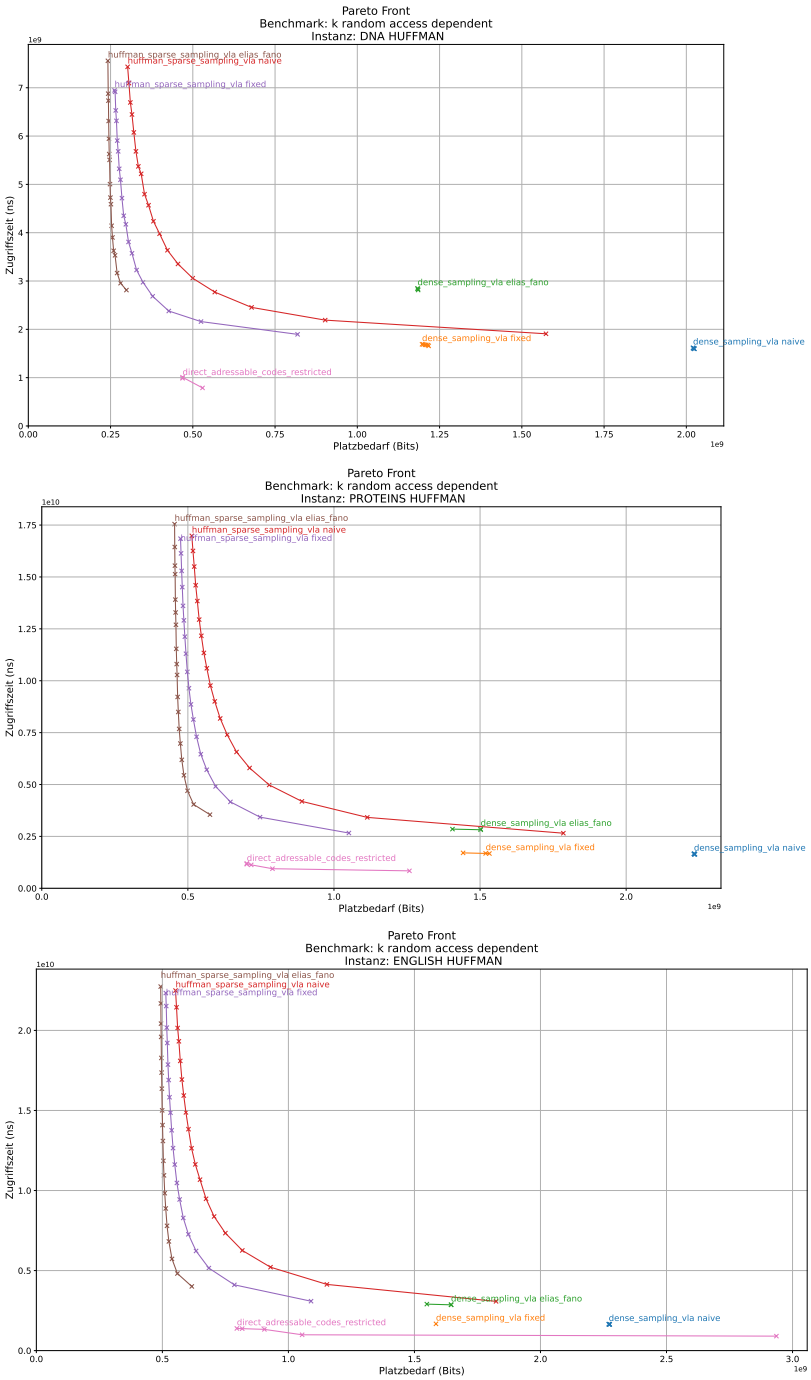

Abbildung 5.4.: Diese Abbildungen zeigen die Pareto Fronten der verschiedenen VBL Arrays für den *k random access dependent* Benchmark auf die Huffman DNA, Proteins und English Instanzen, wobei die Zugriffszeit über den Platzbedarf abgebildet wird. Dabei sind die einzelnen Punkte in einem Plot die verschiedenen Sampling Frequenzen bzw. Levelbegrenzungen der VBL Arrays.

5.4. Platzbedarf

Da der Platzbedarf und die Zugriffszeit vor allem bei Sparse Sampling VBL Arrays in direktem Zusammenhang mit der Sampling Frequenz stehen, wird in diesem Abschnitt zuerst für jede Implementierung die verschiedenen Modifikationen, wie beispielsweise die verschiedenen Sampling Frequenzen, und anschließend jeweils die beste Version hinsichtlich des Platzbedarfs mit den anderen Implementierungen verglichen. Da die Ergebnisse der verschiedenen Datensätze DNA, Proteins, Englisch und Source jeweils sehr ähnlich sind und aus jeder Benchmarkinstanz der selbe Schluss folgt wird in dieser Arbeit nur auf die Source Instanz einmal als LCP-Array und als Huffman Code eingegangen. Die Ergebnisse und Abbildungen für die anderen Instanzen können im Anhang gefunden werden.

Platzbedarf von Sparse Sampling.

Wie aus Tabelle 3.1 ablesbar ist der Platzbedarf direkt abhängig von der Anzahl der Samples. Je weniger Samples zusätzlich gespeichert werden müssen, desto weniger zusätzlicher Platz wird benötigt. Allerdings ist der Platzgewinn durch immer kleiner werdende Sampling Frequenzen k schon bei Frequenzen ab $k \geq 65$ fast vernachlässigbar wie die Abbildung 5.5 zeigt. Da die Ergebnisse für Delta kodierte VBL Arrays nahezu gleich den Gamma kodierten VBL Arrays sind, wird in dieser Arbeit nur auf die Gamma Kodierten VBL Arrays eingegangen. Um diese Arbeit zu vervollständigen und Daten nachvollziehbar zu machen sind alle relevanten Plots und Tabellen im Anhang beigefügt.

Bei einer Sampling Frequenz von $k = 65$ liegt der zusätzliche Platzbedarf sowohl von der Implementierung mit Elias-Fano als auch für Arrays mit Elementen fester Größe bereits bei 10% oder weniger für alle Instanzen. Für $k = 95$ liegt der zusätzliche Platzbedarf für die Samples bereits bei 5% oder weniger für alle Instanzen, weshalb aufgrund der immer langsamer werdenden VBL Arrays für $k > 95$ nicht weiter gebenchmarkt wurde. Des Weiteren wird aus der Abbildung 5.5 der Nachteil des Kodierens klar ersichtlich. Durch die Kodierung der Elemente werden bei einer Elias-Gamma bzw. Elias-Delta Kodierung von Anfang an 83% bzw. 90% zusätzlicher Platz benötigt um die Elemente präfixfrei zu kodieren, wogegen bei bereits präfixfreien Huffman Codes kein zusätzlicher Platz benötigt wird. Daraus folgt, dass Sparse Sampling VBL Arrays besonders bei bereits präfixfreien Codes sehr platzeffizient nahe des informationstheoretischen minimum Arrays abspeichern kann, wogegen bei Arrays, die zuerst kodiert werden müssen, ein nicht vernachlässigbarer Platzbedarf durch das Kodieren entsteht. Wie sich das im Verhältnis zu den anderen VBL Arrays auswirkt wird am Ende dieses Abschnitts betrachtet.

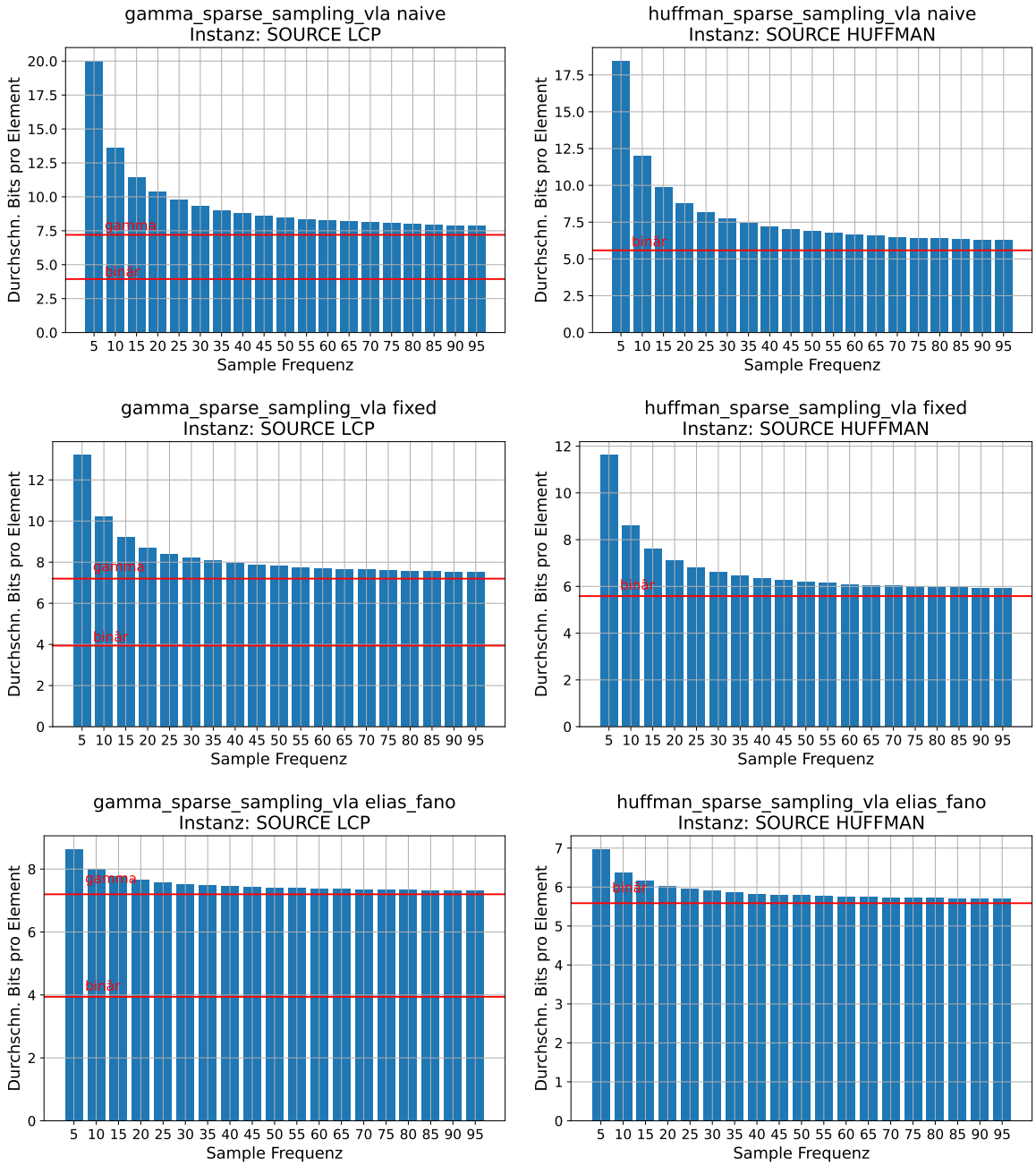

Abbildung 5.5.: Diese Abbildung zeigt sechs Säulendiagramme von sechs unterschiedlichen Versionen von Sparse Sampling. Dabei stehen auf der linken Seite die Evaluationen von der LCP Source Instanz, welche mit Elias-Gamma kodiert wurde und auf der rechten Seite die Evaluationen von der Huffman Instanz. Von Oben nach unten unterscheiden sich die Versionen in der Methode zur Speicherung der Sampling Arrays. In jedem Säulendiagramm wird die entsprechende Version mit verschiedenen Sampling Frequenzen dargestellt. Die rote Linie, welche mit binary beschriftet ist, zeigt den informationstheoretisch minimalen Platzbedarf. Die rote Linie mit gamma beschriftet zeigt den minimalen Platzbedarf für die Elias-Gamma Kodierung.

Platzbedarf Dense Sampling.

Ähnlich wie bei Sparse Sampling hängt bei Dense Sampling der Platzbedarf ebenfalls von der Sampling Frequenz k ab. Für die Implementierung mit einfachen std::vector<uint16_t> für Dense- und std::vector<uint64_t> für Sparse Samples gilt wie in Abbildung 5.6 erkennbar: Je größer k ist desto kleiner wird das VBL Array, allerdings ist deutlich erkennbar, dass der Effekt auf die Größe wesentlich geringer ist. Zudem darf k für den naiven Ansatz nicht größer als 1024 gewählt werden, da sonst nicht mehr garantiert werden kann, dass jedes Element in einem Superblock referenziert werden kann (Abschnitt 3.1.2). Für die Implementierung mit Arrays mit Elementen fester Größe und Elias-Fano lässt sich in Bezug auf den Platzbedarf nahezu kein Unterschied feststellen, da die Dense Sample, welche den wesentlichen Platzbedarf ausmachen in beiden Implementierungen mit Arrays mit Elementen fester Größe realisiert wurden. Durch ein großes k bei Dense Sampling fällt der Speicherbedarf für die Sparse Samples nicht stark ins Gewicht. Im Gegensatz zu dem naiven Ansatz wird erkennbar, dass der Speicherbedarf nicht mit steigendem k grundsätzlich abnimmt, sondern für die kleineren gebenchmarkten k sogar geringer ist. Grund dafür ist, dass für die Dense Samples Arrays mit Elementen fester Größe verwendet werden. Wenn k kleiner ist, kann für die Elemente des Dense Sample Arrays eine kleinere Elementgröße gewählt werden, was zu geringerem Platzbedarf durch die Dense Samples führt. Man erhält hierdurch allerdings mehr Sparse Samples, was aber nur für sehr kleine k Auswirkungen hätte.

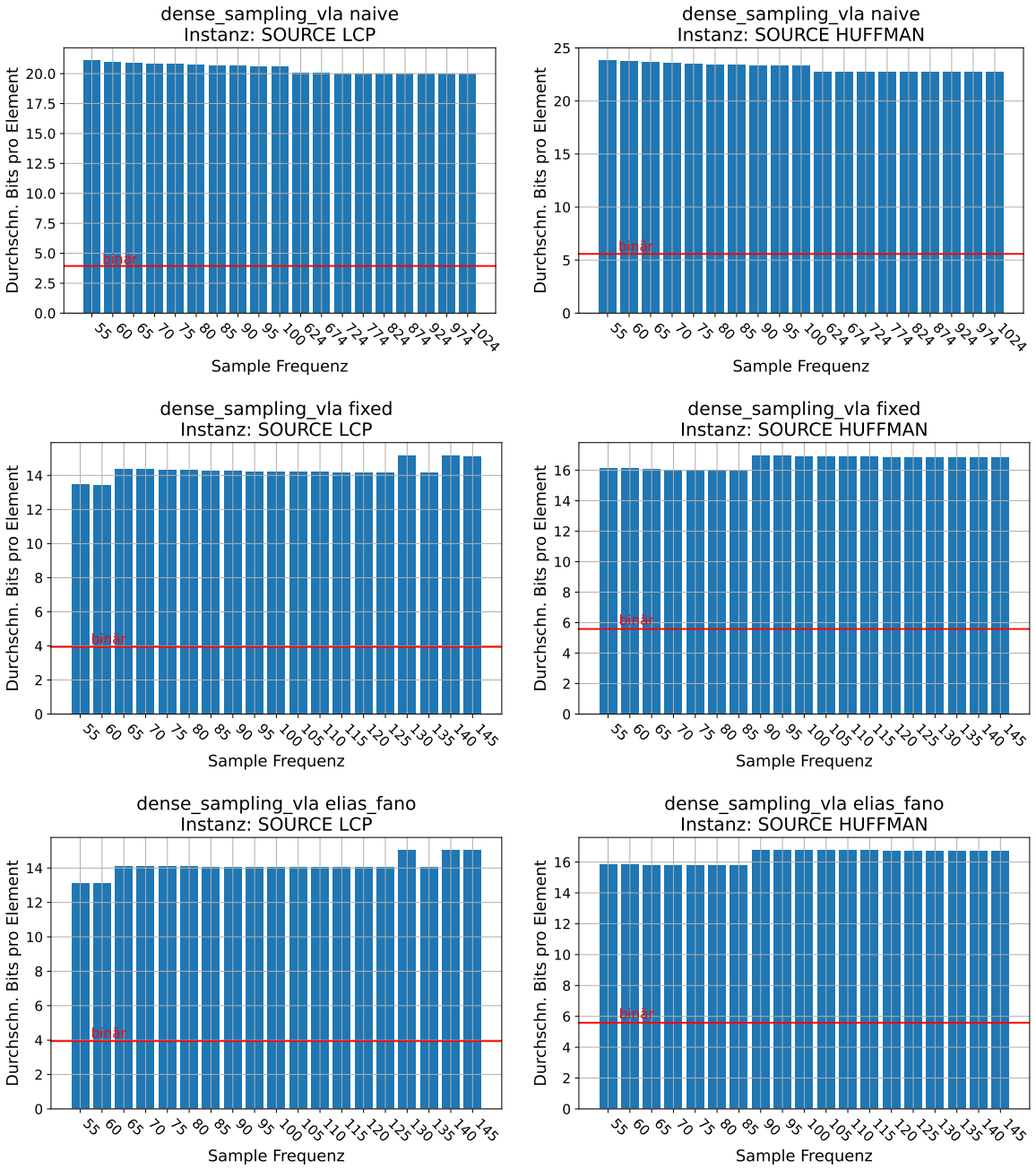

Abbildung 5.6.: Diese Abbildung zeigt sechs Säulendiagramme von drei unterschiedlichen Versionen von Dense Sampling. Dabei stehen auf der linken Seite die Evaluationen von der LCP Source Instanz und auf der rechten Seite die Evaluationen von der Huffman Instanz. Von Oben nach unten unterscheiden sich die Versionen in der Methode zur Speicherung der Sampling Arrays. In jedem Säulendiagramm wird die entsprechende Version mit verschiedenen Sampling Frequenzen dargestellt. Die rote Linie zeigt den informationstheoretisch minimalen Platzbedarf.

Platzbedarf Direct Addressable Codes.

Durch den anderen Ansatz bei Direct Addressable Codes kann hier in der platzoptimalen Version ein zusätzlicher Platzbedarf von 47% für LCP und 56% für Huffman erreicht werden. Durch die Begrenzung der Tiefe auf x Level wird dieser Platzbedarf größer. Dabei entspricht eine Begrenzung auf 1 Level einem Array mit Elementen fester Größe während die Elementgröße dem größten Element entspricht. Aus Abbildung 5.7 wird ersichtlich, dass ab 4 Level bei LCP keine und 5 Level bei Huffman keine signifikanten Änderungen im Platzbedarf erkennbar sind, woraus folgt, dass die Optimale Lösung ebenfalls nur 4 Level für LCP Verwendet. Für die SOURCE Huffman Instanz verwendet die optimale Lösung mehr als 10 Level, spart dabei aber im Vergleich zu 5 Level nur 0, 02% Platz, was vernachlässigbar ist und nachteilhaft für die Zugriffszeit, dazu mehr in Abschnitt 5.5.

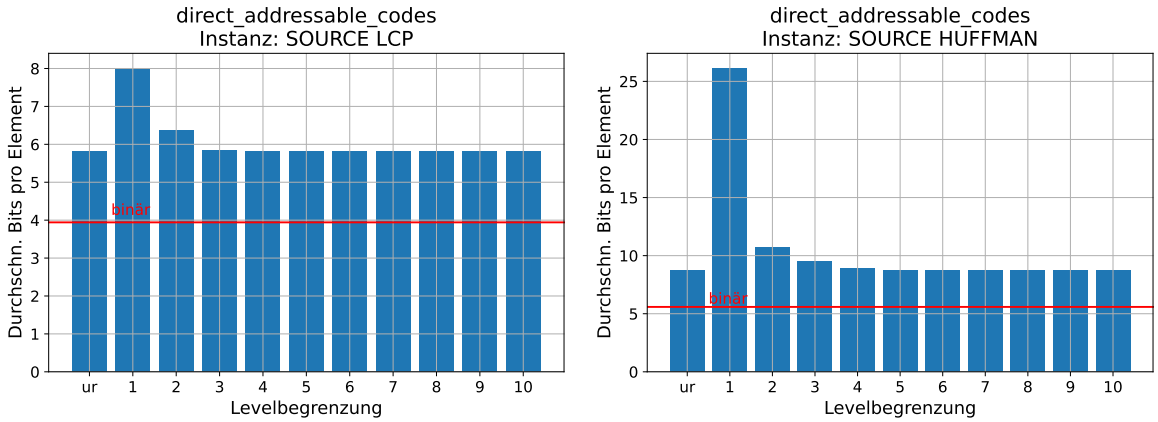

Abbildung 5.7.: Diese Abbildung Vergleicht den Platzbedarf für verschiedene Levelbegrenzungen bei Direct Addressable Codes für die LCP Source Instanz (links) und die Huffman Instanz (rechts). Die rote Linie zeigt den informationstheoretisch minimalen Platzbedarf. ur auf der x-Achse steht für unrestricted.

Vergleich des Platzbedarfs der verschiedenen VBL Arrays.

Wie in Abbildung 5.8 ersichtlich, sind Direct Addressable Codes für LCP Arrays, also Arrays, welche nicht von sich aus präfixfrei sind, die platzoptimale Lösung. Für bereits präfixfreie Huffman Codes kann mit Sparse Sampling mit Elias-Fano allerdings ein durchaus relevanter Platzvorteil erzielt werden. Wie bereits erwähnt liegt dies daran, dass Sparse Sampling für nicht präfixfreie Codes zusätzlichen Platzbedarf in die Kodierung investieren muss, was bei Direct Addressable Codes nicht notwendig ist. Bei bereits präfixfreien Codes fällt dieser Aspekt weg.

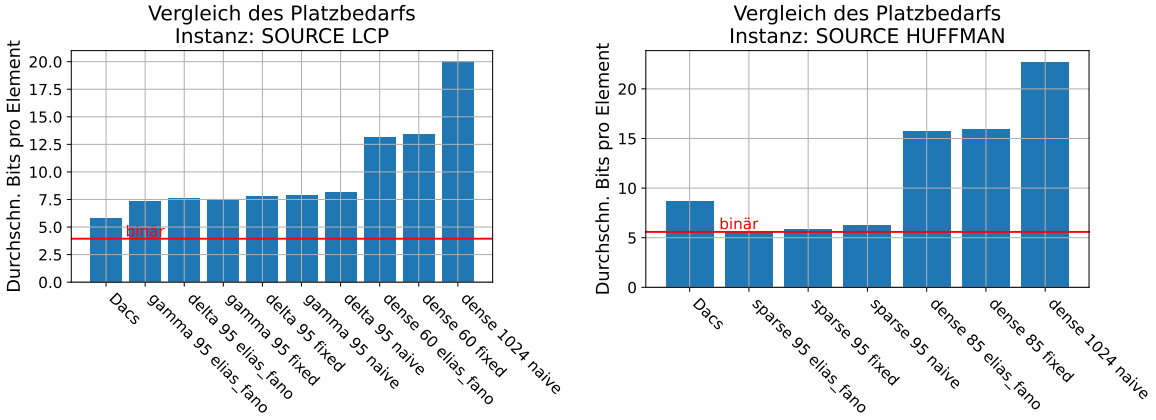

Abbildung 5.8.: Diese Abbildung Vergleicht die platzeffizientesten VBL Arrays miteinander. Auf der x-Achse steht die Art des VBL Arrays gefolgt von der Sampling Frequenz und der Speichermethode für die Sample Arrays. Die rote Linie zeigt den informationstheoretisch minimalen Platzbedarf.

5.5. Zugriffszeit

In diesem Abschnitt werden die VBL Arrays hinsichtlich der optimalen Zugriffszeit verglichen. Wie bei der Evaluation des Platzbedarfs in Abschnitt 5.4 sind die Ergebnisse für die verschiedenen Datensätze sehr ähnlich, weshalb auch hier nur auf die Source Instanzen eingegangen wird. Dazu unterscheiden sich die Benchmarks *k random access* und *k random access dependent* nur um konstante Faktoren, die relative Ordnung der VBL Arrays bleibt aber gleich, weshalb in dieser Arbeit lediglich *k random access dependent* evaluiert wird. Auch hier werden alle Abbildungen der Vollständigkeit halber im Anhang beigefügt.

Zugriffszeit Sparse Sampling.

Wie in Kapitel 3 beschrieben hängt die Zugriffszeit auf Sparse Sampling sehr stark von der Sampling Frequenz k ab. Diese Abhängigkeit ist auch in den Ergebnissen der Benchmarks reflektiert. In den Abbildungen 5.9, 5.10 und 5.11 wird der Unterschied durch die Auffächerung der verschiedenen Versionen verdeutlicht. Dabei kann den Abbildungen entnommen werden, dass die Sparse Sampling VBL Arrays mit kleinem k schneller sind als solche mit großem k. Besonders deutlich wird diese Auffächerung bei den Huffman Instanzen, welche den direkten Einfluss der Sampling Frequenz auf die Zugriffszeit zeigen. Der Unterschied zwischen den LCP Instanzen und den Huffman Instanzen liegt an der Iteration über das Array. Während für die Elias-Gamma bzw. Elias-Delta kodierten LCP Arrays die Größe des nächsten Elements direkt abgelesen werden kann, um dann das nächste Element zu lesen, muss bei der Huffman Kodierung von der kürzest möglichen Länge eines Elements bis zur größtmöglichen Länge eines Elements jede Länge gelesen und auf Validität überprüft werden. Dadurch benötigt die Sparse Sampling Implementierung für einen Iterationsschritt in einer Huffman Instanz deutlich mehr Zeit als für einen Iterationsschritt in einer LCP Instanz,

was sich durch den deutlich höheren Einfluss der Sampling Frequenz auf die Zugriffszeit zeigt.

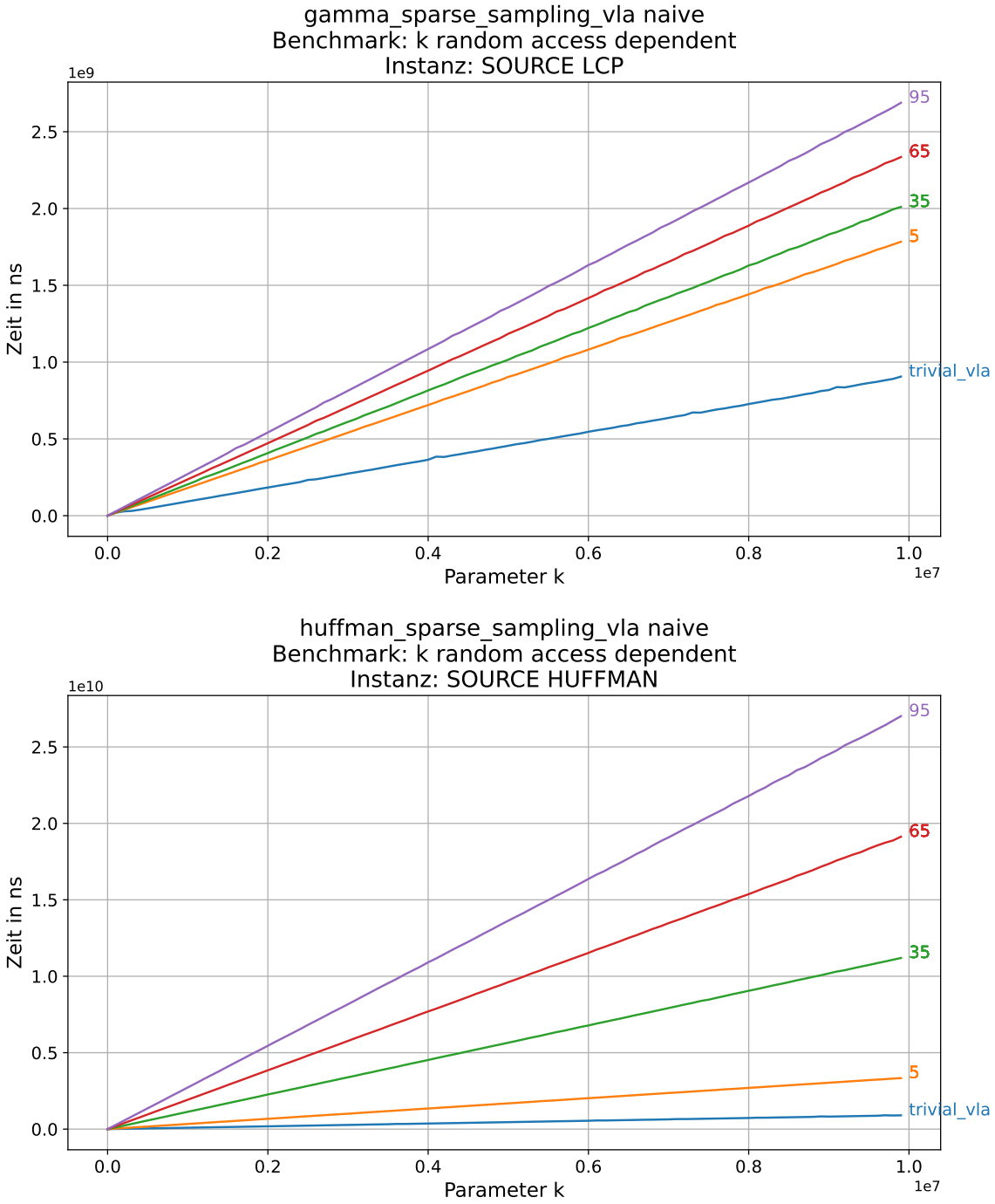

Abbildung 5.9.: Diese Abbildung zeigt zwei Graphen, welche die Zugriffszeit für den *k random access dependent* Benchmark über der Anzahl der Zugriffe auf Sparse Sampling VBL Arrays, mit trivialen Arrays als Sampling Methode, abbilden. Der obere Graph beschreibt die Zugriffszeiten auf eine Elias-Gamma Kodierte LCP Source Instanz und der rechte Graph auf die Huffman Source Instanz. Der Parameter k auf der x-Achse beschreibt die Anzahl der Zugriffe auf die Arrays und die Zahlen an den einzelnen Plots beschreiben die verschiedenen Sampling Frequenzen der Sparse Sampling VBL Arrays.

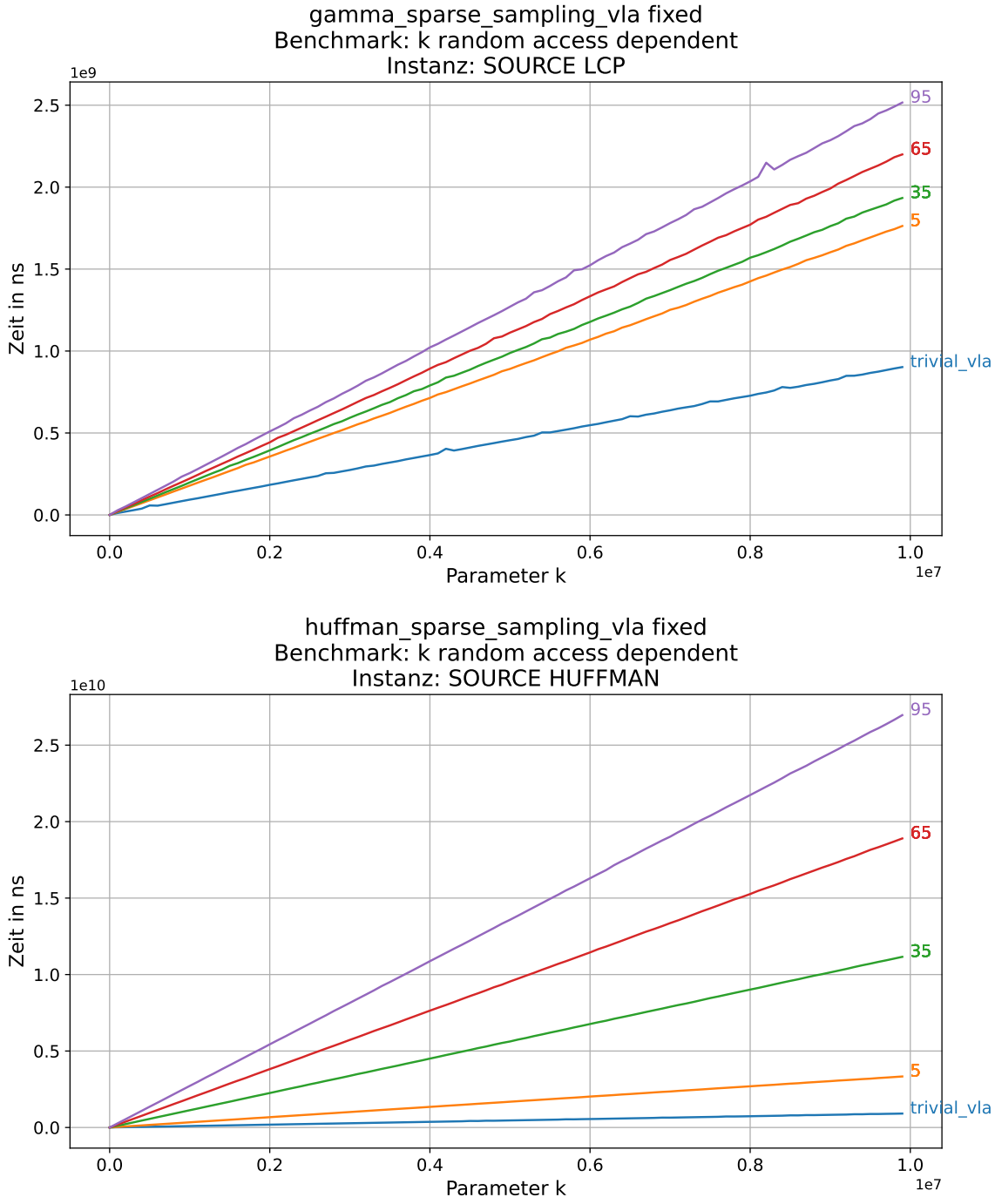

Abbildung 5.10.: Diese Abbildung zeigt zwei Graphen, welche die Zugriffszeit für den *k random access dependent* Benchmark über der Anzahl der Zugriffe auf Sparse Sampling VBL Arrays, mit Arrays mit Elementen fester Größe (Abschnitt 2.8) als Sampling Methode, abbilden. Der obere Graph beschreibt die Zugriffszeiten auf eine Elias-Gamma Kodierte LCP Source Instanz und der rechte Graph auf die Huffman Source Instanz. Der Parameter k auf der x-Achse beschreibt die Anzahl der Zugriffe auf die Arrays und die Zahlen an den einzelnen Plots beschreiben die verschiedenen Sampling Frequenzen der Sparse Sampling VBL Arrays.

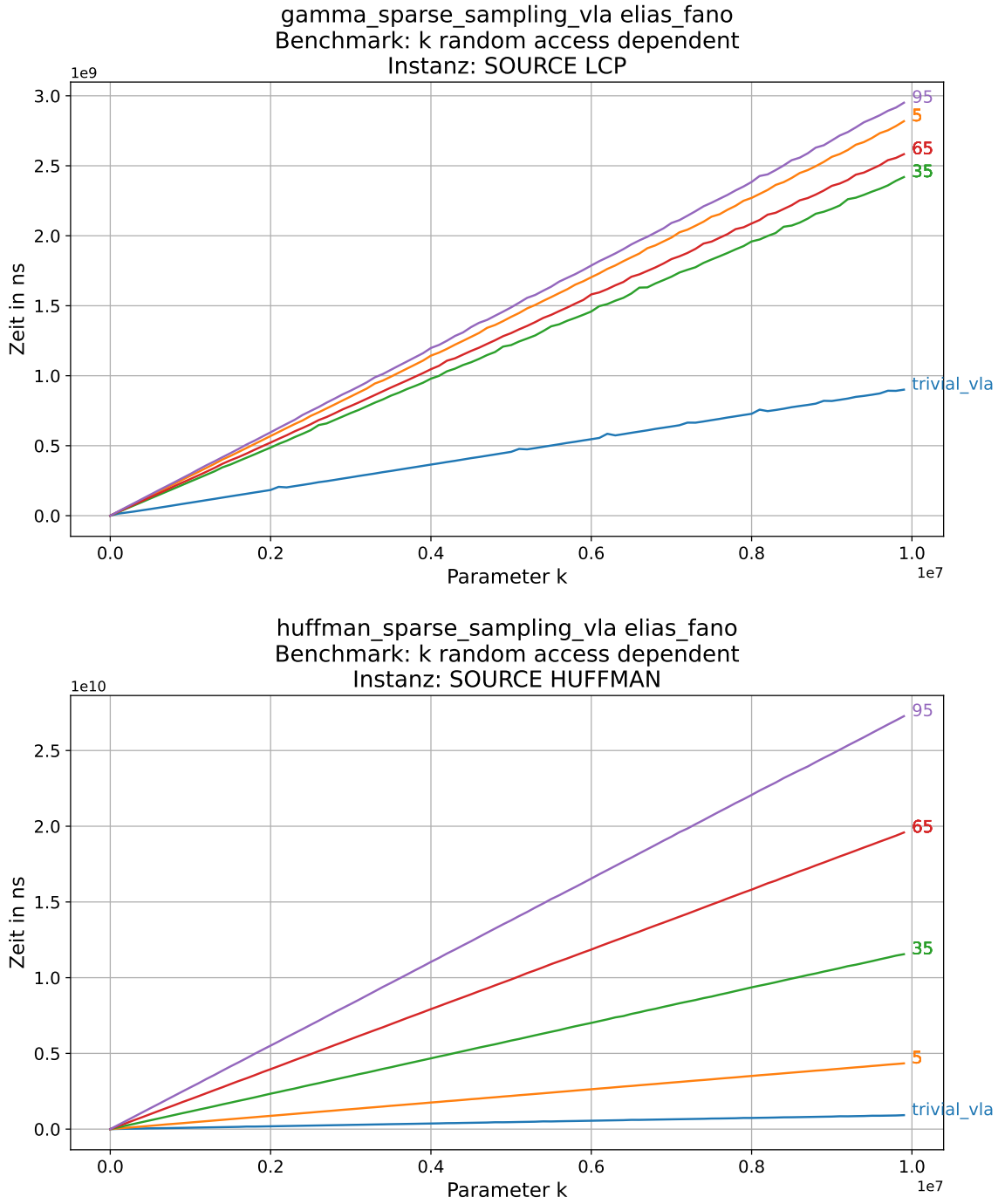

Abbildung 5.11.: Diese Abbildung zeigt zwei Graphen, welche die Zugriffszeit für den *k random access dependent* Benchmark über der Anzahl der Zugriffe auf Sparse Sampling VBL Arrays, mit einer Elias-Fano kodierten Sequenz als Sampling Methode, abbilden. Der obere Graph beschreibt die Zugriffszeiten auf eine Elias-Gamma Kodierte LCP Source Instanz und der rechte Graph auf die Huffman Source Instanz. Der Parameter k auf der x-Achse beschreibt die Anzahl der Zugriffe auf die Arrays und die Zahlen an den einzelnen Plots beschreiben die verschiedenen Sampling Frequenzen der Sparse Sampling VBL Arrays.

Zugriffszeit Dense Sampling.

Im Gegensatz zu Sparse Sampling hat die Sampling Frequenz k bei Dense Sampling nahezu keinen Einfluss auf die Zugriffszeit. Wie in den Abbildungen 5.12 und 5.13 zu sehen sind alle Versionen von Dense Sampling sehr nah beieinander. Dabei ist in der Abbildung auch klar erkennbar, dass es für Dense Sampling keinen Unterschied macht, of eine LCP Instanz oder eine Huffman Instanz gebenchmarkt wurde, was daran liegt, dass Dense Sampling für jedes Element die Start- und Endposition speichert. Was in dem Element steht ist dabei für die Zugriffszeit für *lese* und *schreib* Operationen irrelevant.

Ein deutlich sichtbarer Unterschied entsteht durch die Sampling Methode. Während Dense Sampling mit std::vector als Sampling Array ca. um einen Faktor 1.8 langsamer als der Triviale std::vector ist, ist Dense Sampling mit Arrays mit Elementen fester Größe ungefähr um Faktor 1.9 und Dense Sampling mit Elias-Fano um Faktor 3.3 langsamer.

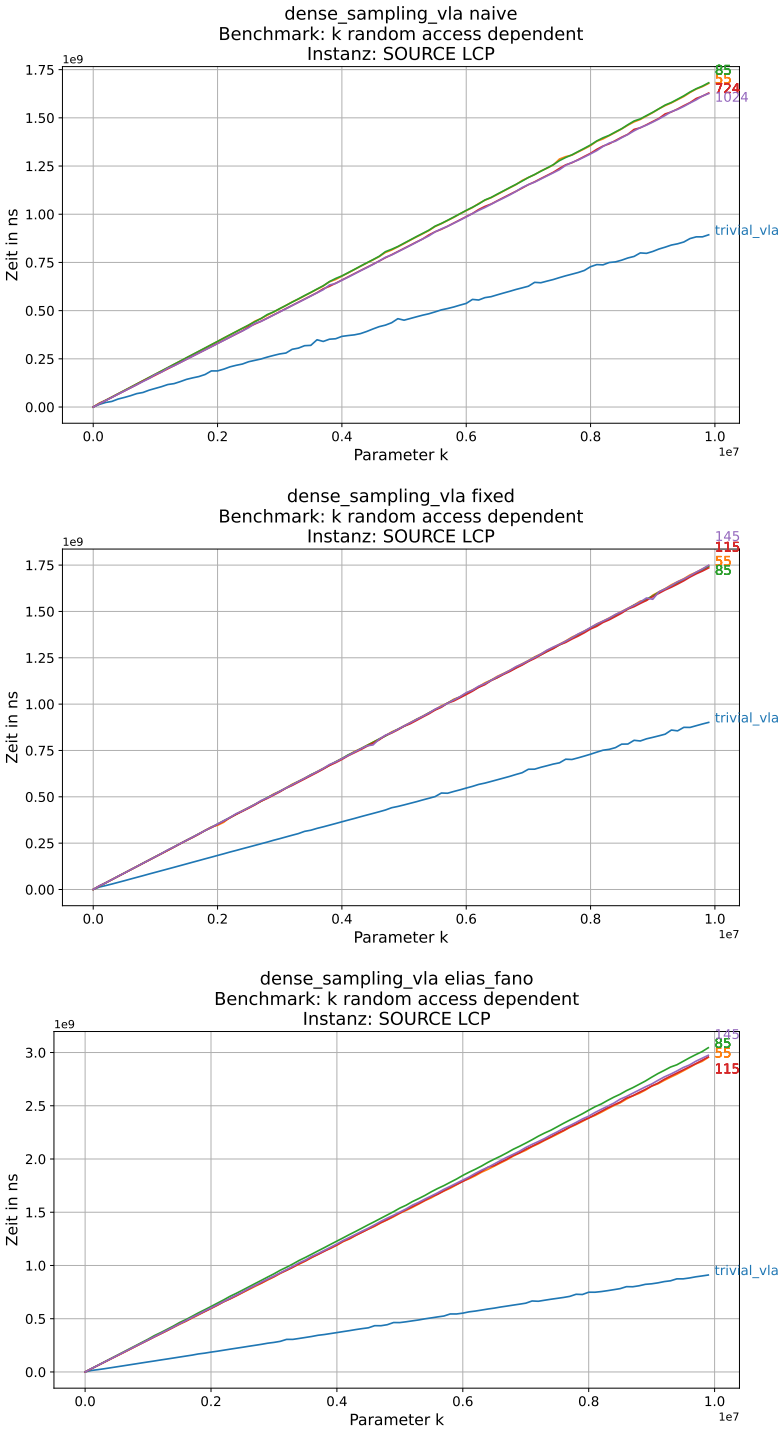

Abbildung 5.12.: Diese Abbildung zeigt drei Graphen, welche die Zugriffszeit für den *k random access dependent* Benchmark über der Anzahl der Zugriffe von verschiedenen Versionen von Dense Sampling abbilden. Die Graphen bilden Zugriffszeiten von Dense Sampling VBL Arrays, mit verschiedenen Speichermethoden für das Sample Array, auf die LCP Source Instanz ab. Der Parameter k auf der x-Achse beschreibt die Anzahl der Zugriffe auf die Arrays und die Zahlen an den einzelnen Plots beschreiben die verschiedenen Sampling Frequenzen der Dense Sampling VBL Arrays.

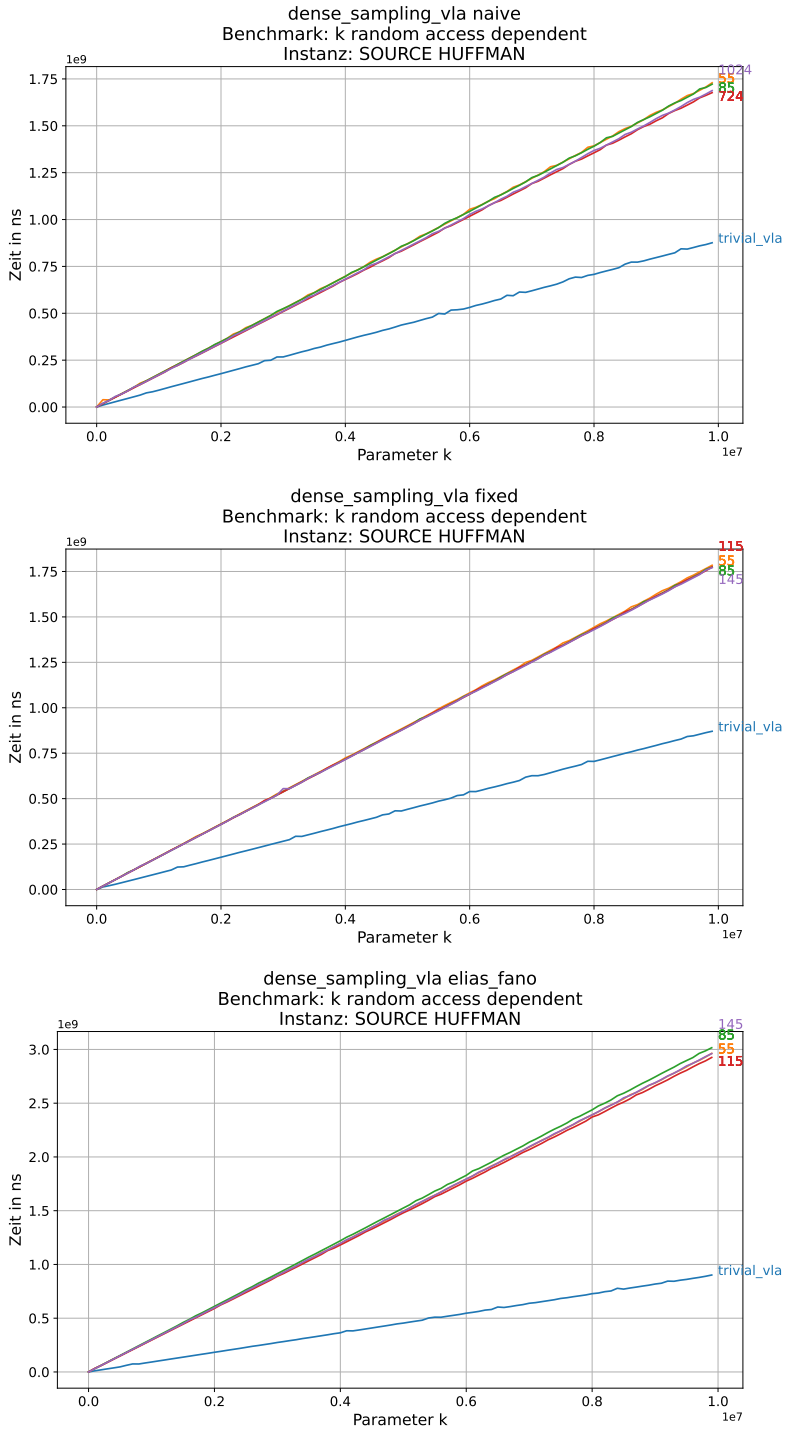

Abbildung 5.13.: Diese Abbildung zeigt drei Graphen, welche die Zugriffszeit für den *k random access dependent* Benchmark über der Anzahl der Zugriffe von verschiedenen Versionen von Dense Sampling abbilden. Die Graphen bilden Zugriffszeiten von Dense Sampling VBL Arrays, mit verschiedenen Speichermethoden für das Sample Array, auf die Huffman Source Instanz ab. Der Parameter k auf der x-Achse beschreibt die Anzahl der Zugriffe auf die Arrays und die Zahlen an den einzelnen Plots beschreiben die verschiedenen Sampling Frequenzen der Dense Sampling VBL Arrays.

Zugriffszeit Direct Addressable Codes.

Wie in Kapitel 3 beschrieben hängt die Zugriffszeit bei Direct Addressable Codes von der Menge an Ebenen ab, auf die das Element verteilt ist. Diese Abhängigkeit ist auch in den Benchmarks erkennbar. Abbildung 5.14 zeigt den deutlichen Zeitvorteil der Begrenzung der Level auf ein bzw. zwei Level. Die Abbildung zeigt aber auch, dass ab 3 Level für die Source LCP Instanz und 4 Level für die Source Huffman Instanz keine weiteren signifikanten Verschlechterungen für Versionen mit einer höheren Levelbegrenzung bzw. der platzoptimalen Version ohne Levelbegrenzung entstehen. Das liegt daran, dass wie in Abschnitt 5.4 beschrieben, für 3 bzw. 4 Level bereits eine nahezu platzoptimale Version realisiert werden kann und die weiteren Level die ohne Levelbegrenzung möglich wären, entweder nicht oder nur für sehr wenige Elemente benötigt werden.

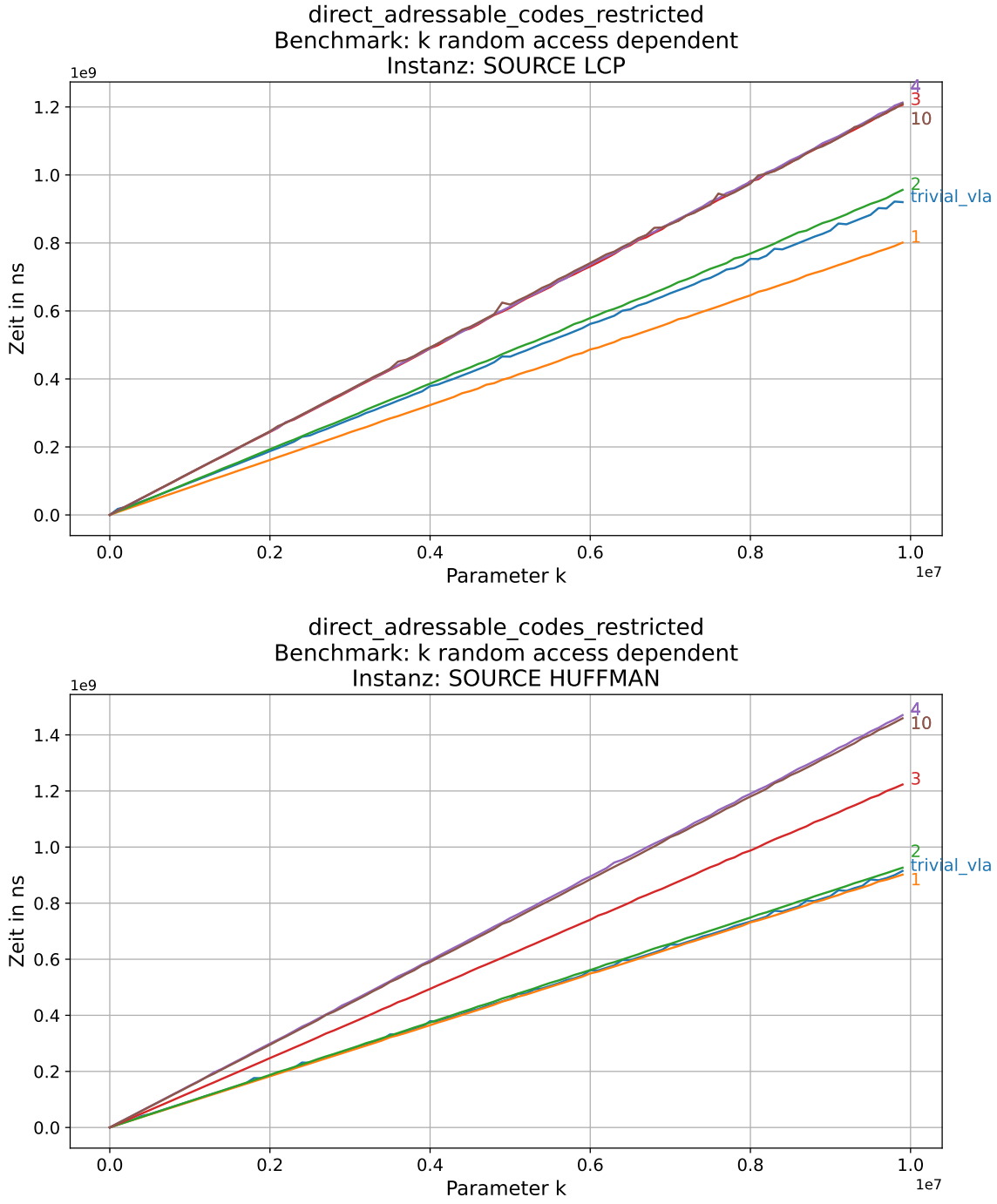

Abbildung 5.14.: Diese Abbildung zeigt zwei Graphen, welche die Zugriffszeit für den *k random access dependent* Benchmark über der Anzahl der Zugriffe von Direct Addressable Codes abbilden. Auf der linken Seite werden Zugriffszeiten von Direct Addressable Codes, auf die LCP Source Instanz abgebildet, auf der rechten Seite werden Zugriffszeiten von Direct Addressable Codes, auf die Huffman Source Instanz abgebildet. Der Parameter k auf der x-Achse beschreibt die Anzahl der Zugriffe auf die Arrays und die Legende beschreibt die Levelbegrenzung für die Direct Addressable Codes.

Vergleich der Zugriffszeit auf die verschiedenen VBL Arrays.

Wie in Abbildung 5.15 erkennbar sind sowohl die Direct Addressable Codes mit Levelbegrenzung als auch die platzoptimalen Direct Addressable Codes die zeiteffizienteste Art von VBL Arrays. Dabei macht es keinen Unterschied, ob die Huffman oder LCP Instanz betrachtet wird. Bemerkenswert ist, dass bei der LCP Instanz die schnellste Version von Sparse Sampling VBL Arrays mit den Dense Sampling VBL Arrays gleich auf sind.

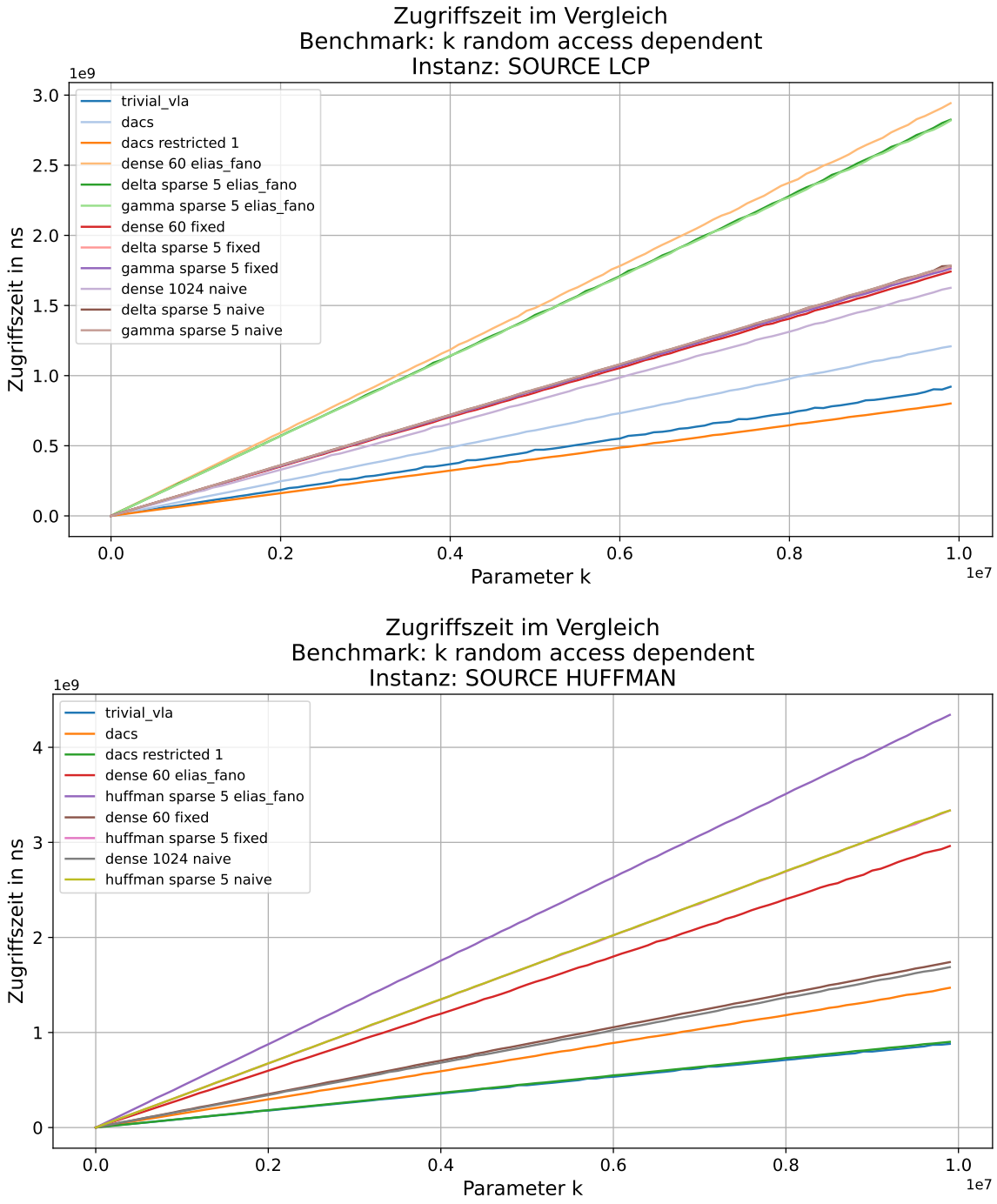

Abbildung 5.15.: Diese Abbildung zeigt zwei Graphen, welche die Zugriffszeit für den *k random access dependent* Benchmark über der Anzahl der Zugriffe von den besten Versionen der einzelnen VBL Arrays abbilden. Auf der linken Seite werden Zugriffszeiten auf die LCP Source Instanz abgebildet, auf der rechten Seite werden Zugriffszeiten auf die Huffman Source Instanz abgebildet. Der Parameter k auf der x-Achse beschreibt die Anzahl der Zugriffe auf die Arrays und die Legende beschreibt die Art des VBL Arrays gefolgt von der Sampling Frequenz bzw. der Levelbegrenzung und der Speichermethode für die Sample Arrays.

6. Fazit

In dieser Arbeit wurden drei verschiedene VBL Array Ansätze gegeneinander gebenchmarkt. Dabei wurden die Ansätze Sparse- und Dense Sampling von Grund auf mit verschiedenen Ansätzen zur Speicherung der Samples implementiert und sowohl untereinander als auch mit der open Source Implementierung von Direct Addressable Codes von [2] verglichen.

Die Ergebnisse der Benchmarks zeigen, dass für nicht präfixfreie Codes Direct Addressable Codes die beste Lösung bieten. Bei präfixfreien Codes können Sparse Sampling VBL Arrays einen Platzvorteil gegenüber Direct Addressable Codes erzielen, was in Umgebungen mit sehr wenig Speicher oder sehr seltenen Zugriffen sinnvoll sein kann.

Wenn der Hauptfokus auf der Geschwindigkeit liegt, werden Direct Addressable Codes in diesen Benchmarks nur durch triviale Arrays übertroffen, welche aber einen sehr großen zusätzlichen Speicherbedarf haben.

6.1. Zukünftige Arbeit

In zukünftiger Arbeit könnte der Platzverbrauch von Sparse und Dense Sampling weiter verbessert werden, indem eine weitere Sampling Ebene über den bereits bestehenden Sampling Ebenen hinzufügt wird, sodass die Sparse Samples in 32 Bit anstatt 64 Bit abgespeichert werden können und nur sehr wenige Samples 64 Bit benötigen.

Des Weiteren könnte der Platzbedarf der Elias-Fano Version von Dense Sampling noch weiter verbessert werden, indem für jeden Dense Sampling Abschnitt eine Elias-Fano Sequenz abgespeichert wird.

Des weiteren wurden in dieser Arbeit nur unkomprimierte VBL Arrays untersucht. In einer weiteren Arbeit kann untersucht werden, wie VBL Arrays komprimiert werden können um weiter Speicherplatz zu sparen und wie sich dabei die Zugriffszeit verhält. Diese Kompression könnte beispielsweise für Sampling VBL Arrays durch das Speichern von sich wiederholenden Muster in der Abfolge von Eintragsgrößen erreicht werden, wobei dann an den Stellen, an denen ein gespeichertes Muster passt eine Referenz auf dieses Muster gespeichert wird anstelle der Pointer auf die Elemente.

Literatur

[1] David Benoit, Erik D Demaine, J Ian Munro, Rajeev Raman, Venkatesh Raman und S Srinivasa Rao. „Representing trees of higher degree“. In: *Algorithmica* 43 (2005), S. 275–292.

[2] Nieves R. Brisaboa, Susana Ladra und Gonzalo Navarro. „DACs: Bringing direct access to variable-length codes“. In: *Information Processing & Management* 49.1 (2013), S. 392–404. ISSN: 0306-4573. DOI: https://doi.org/10.1016/j.ipm.2012.08.003. URL: https://www.sciencedirect.com/science/article/pii/S0306457312001094.

[3] Peter Elias. „Efficient storage and retrieval by content and address of static files“. In: *Journal of the ACM (JACM)* 21.2 (1974), S. 246–260.

[4] Peter Elias. „Universal codeword sets and representations of the integers“. In: *IEEE transactions on information theory* 21.2 (1975), S. 194–203.

[5] Robert Mario Fano. *On the number of bits required to implement an associative memory*. Massachusetts Institute of Technology, Project MAC, 1971.

[6] Paolo Ferragina und Rossano Venturini. „A simple storage scheme for strings achieving entropy bounds“. In: *Theoretical Computer Science* 372.1 (2007), S. 115–121.

[7] Simon Gog, Timo Beller, Alistair Moffat und Matthias Petri. „From Theory to Practice: Plug and Play with Succinct Data Structures“. In: *13th International Symposium on Experimental Algorithms, (SEA 2014)*. 2014, S. 326–337.

[8] Rodrigo González, Szymon Grabowski, Veli Mäkinen und Gonzalo Navarro. „Practical implementation of rank and select queries“. In: *Poster Proc. Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*. CTI Press und Ellinika Grammata Greece. 2005, S. 27–38.

[9] Gonzalo Navarro and others. *Pizza&Chili Corpus*. http://pizzachili.dcc.uchile.cl/. Accessed: 2024-09-29. 2005.

[10] David A Huffman. „A method for the construction of minimum-redundancy codes“. In: *Proceedings of the IRE* 40.9 (1952), S. 1098–1101.

[11] Guy Joseph Jacobson. *Succinct static data structures*. Carnegie Mellon University, 1988.

[12] Hans-Peter Lehmann. *Util*. https://github.com/ByteHamster/Util.git. Accessed: 07-09-2024. 2024.

[13] Udi Manber und Gene Myers. „Suffix arrays: a new method for on-line string searches“. In: *siam Journal on Computing* 22.5 (1993), S. 935–948.

[14] Gonzalo Navarro. „Compact data structures: A practical approach“. In: Cambridge University Press, 2016. Kap. 3.2.

[15] Robert F Rice. „Some practical universal noiseless coding techniques“. In: *Jet Propulsion Laboratory, Pasadena, California* 22 (1971), S. 1–22.

[16] Hugh E Williams und Justin Zobel. „Compressing integers for fast file access“. In: *The Computer Journal* 42.3 (1999), S. 193–201.

[17] Dong Zhou, David G Andersen und Michael Kaminsky. „Space-efficient, high-performance rank and select structures on uncompressed bit sequences“. In: *Experimental Algorithms: 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings 12*. Springer. 2013, S. 151–163.

A. Anhang

In diesem Anhang werden die in der Arbeit nicht vorgestellten Resultate der Vollständigkeit halber abgebildet.

A.1. Platzbedarf

In diesem Abschnitt wird der Platzbedarf von allen gebenchmarkten VBL Arrays auf alle Instanzen abgebildet. Dabei beschreiben die roten Linien mit binär beschriftet in den Säulendiagrammen jeweils den informationstheoretisch minimalen Platzbedarf und die roten Linien mit gamma bzw. delta beschriftet den minimalen Platzbedarf wenn die Daten Elias-Gamma bzw. Elias-Delta kodiert werden.

Sparse Sampling

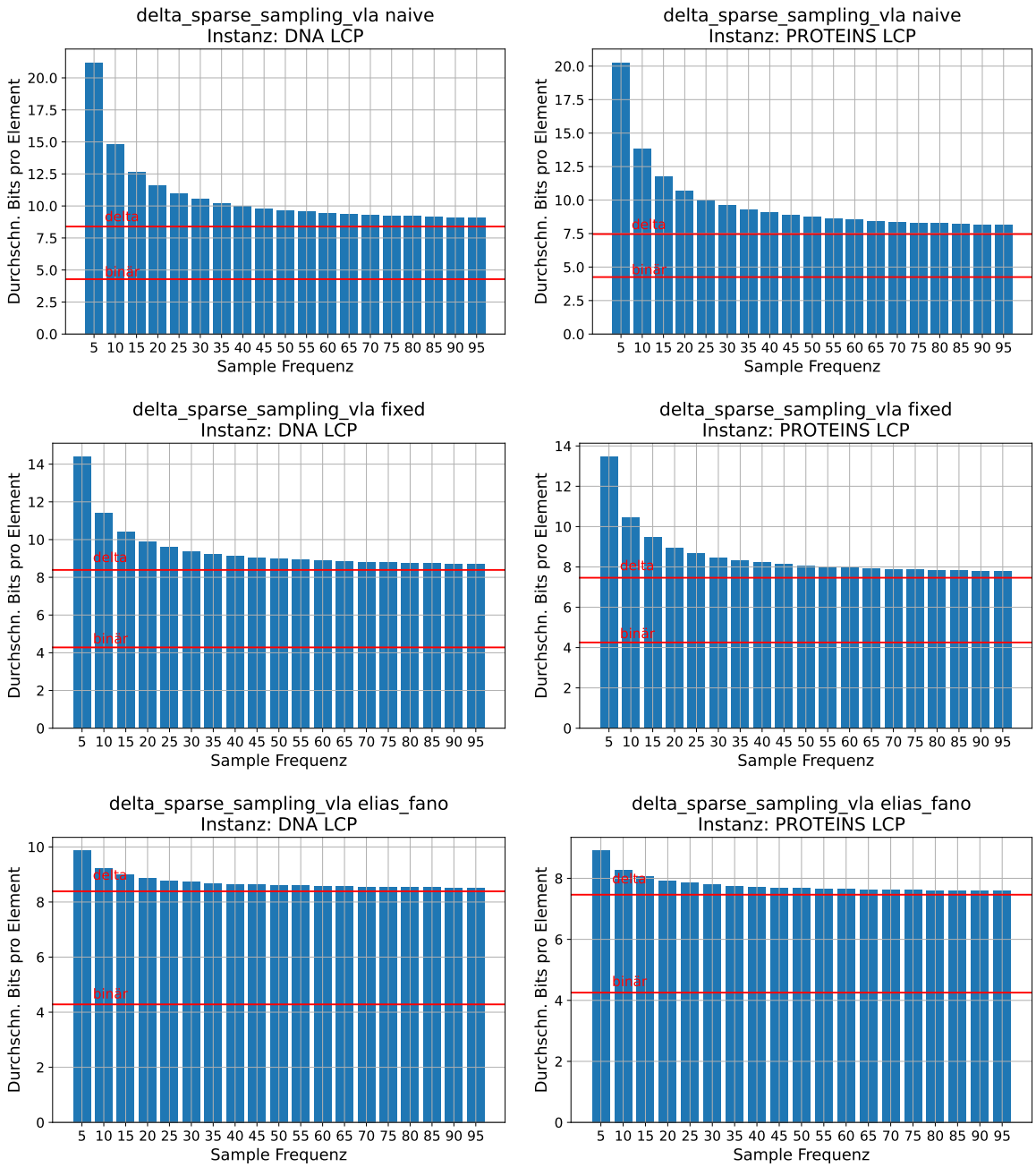

Abbildung A.1.: Platzbedarf für Sparse Sampling LCP DNA und LCP PROTEINS Elias-Delta kodiert.

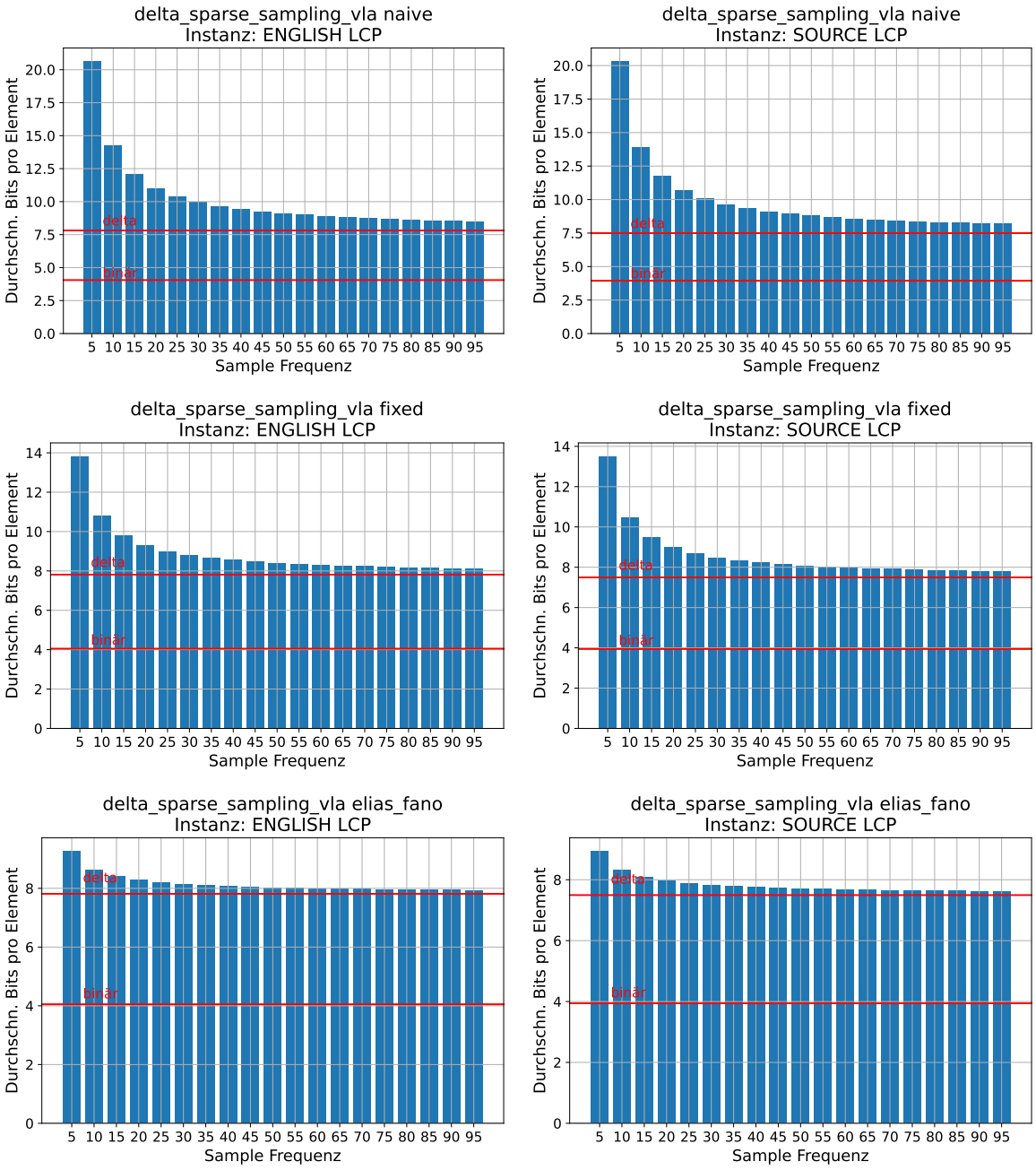

Abbildung A.2.: Platzbedarf für Sparse Sampling LCP ENGLISH und LCP SOURCE Elias-Delta kodiert.

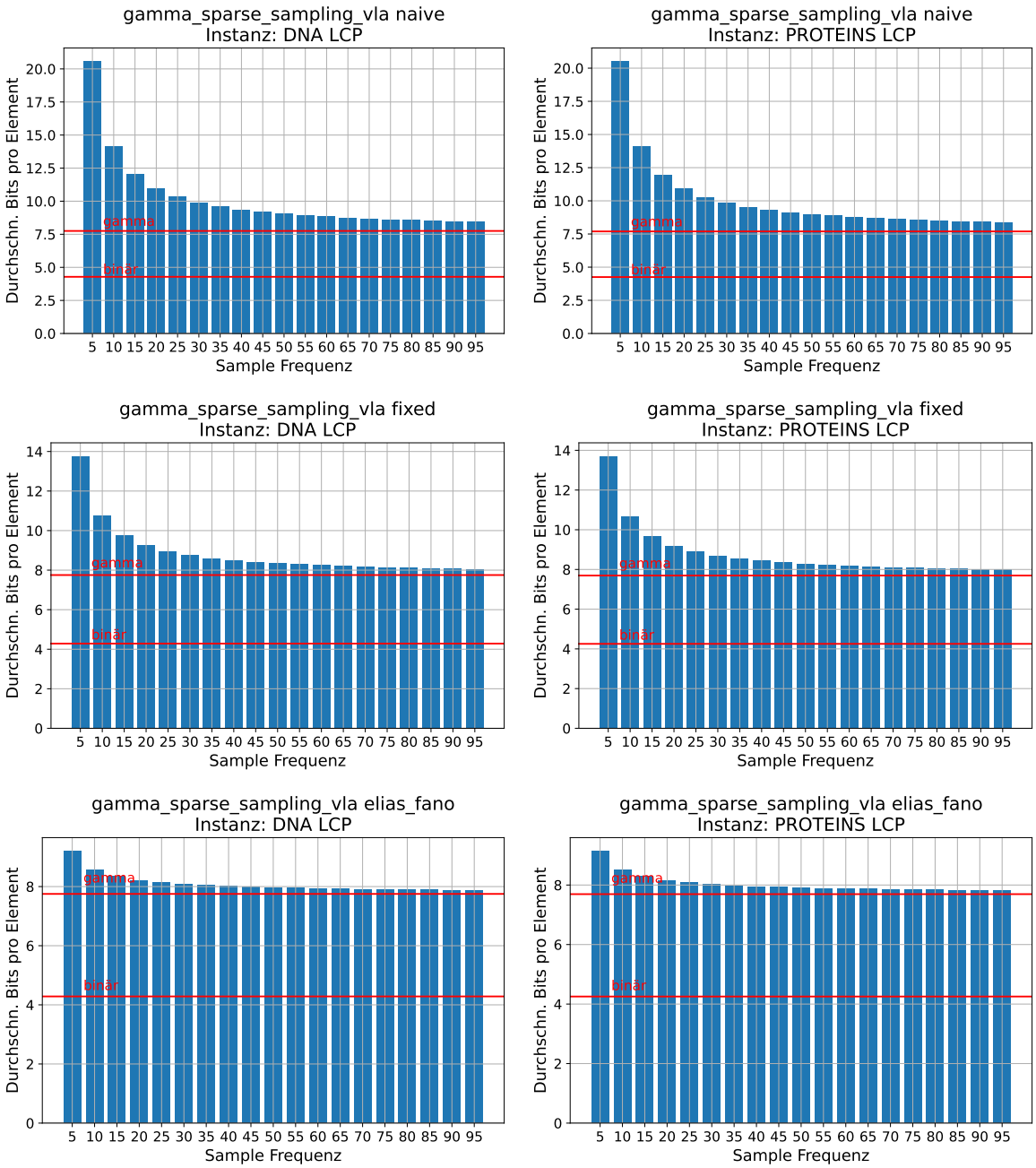

Abbildung A.3.: Platzbedarf für Sparse Sampling LCP DNA und LCP PROTEINS Elias-Gamma kodiert.

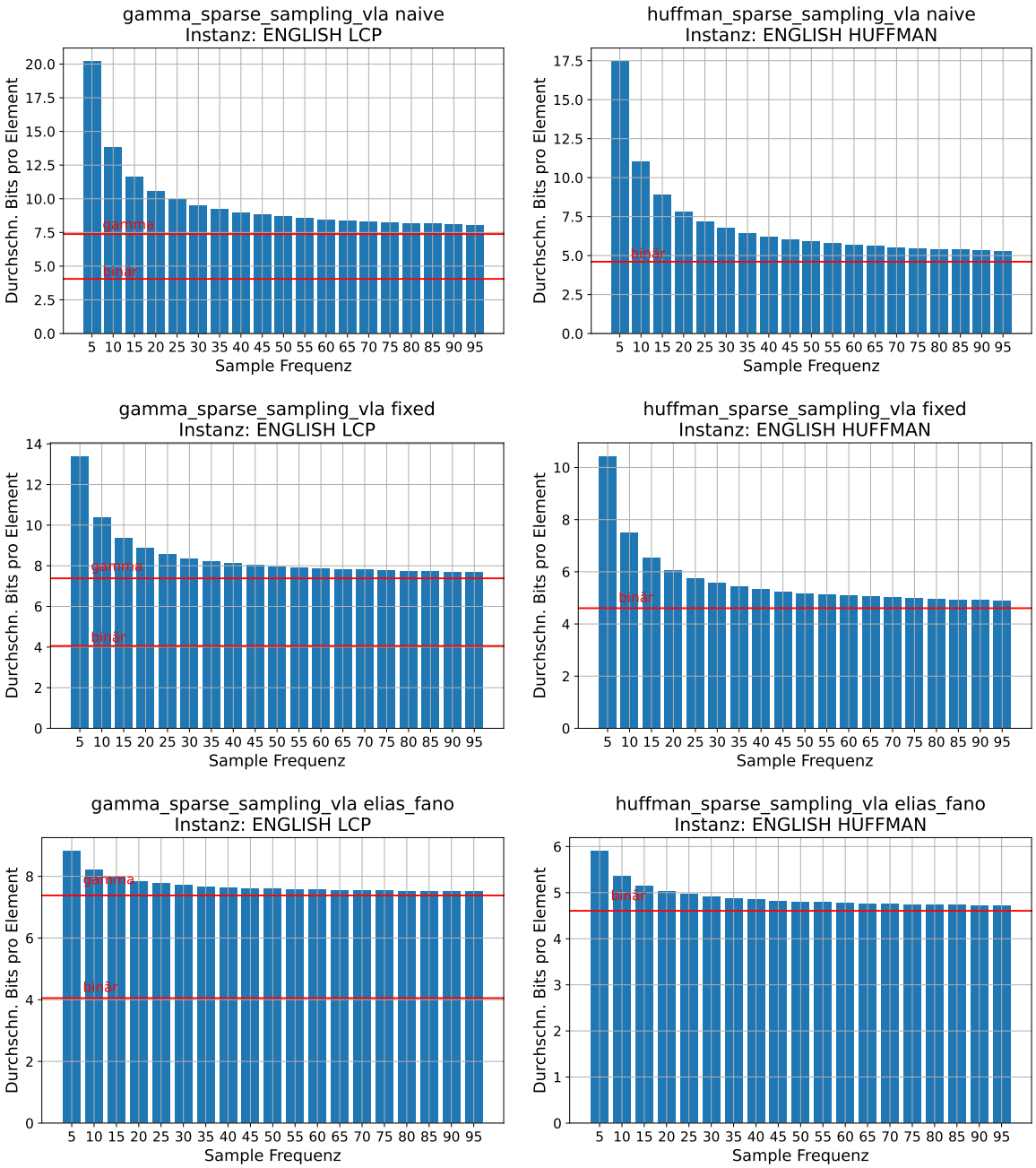

Abbildung A.4.: Platzbedarf für Sparse Sampling LCP ENGLISH Elias-Gamma kodiert und Huffman ENGLISH.

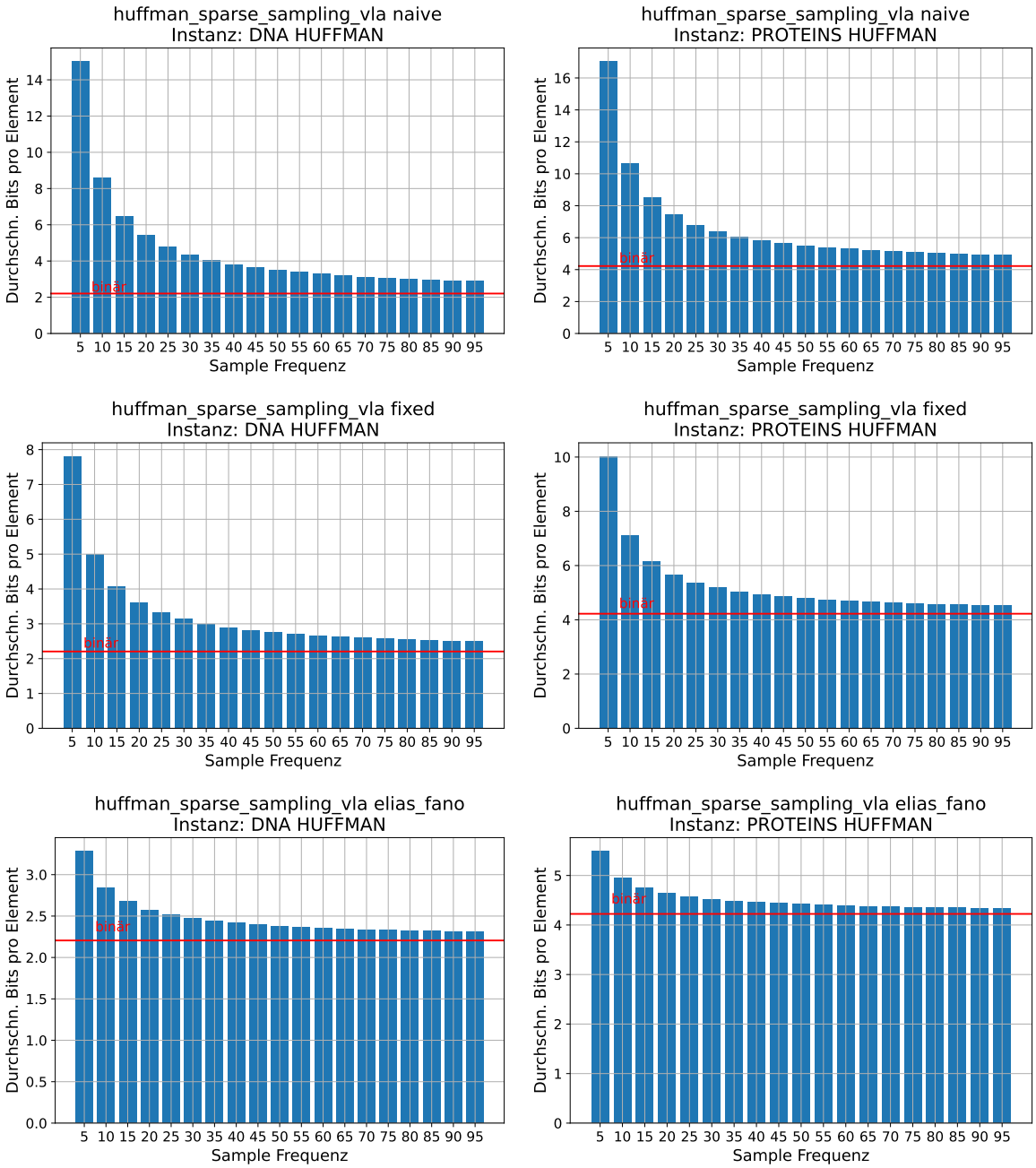

Abbildung A.5.: Platzbedarf für Sparse Sampling Huffman DNA und Huffman PROTEINS

Dense Sampling

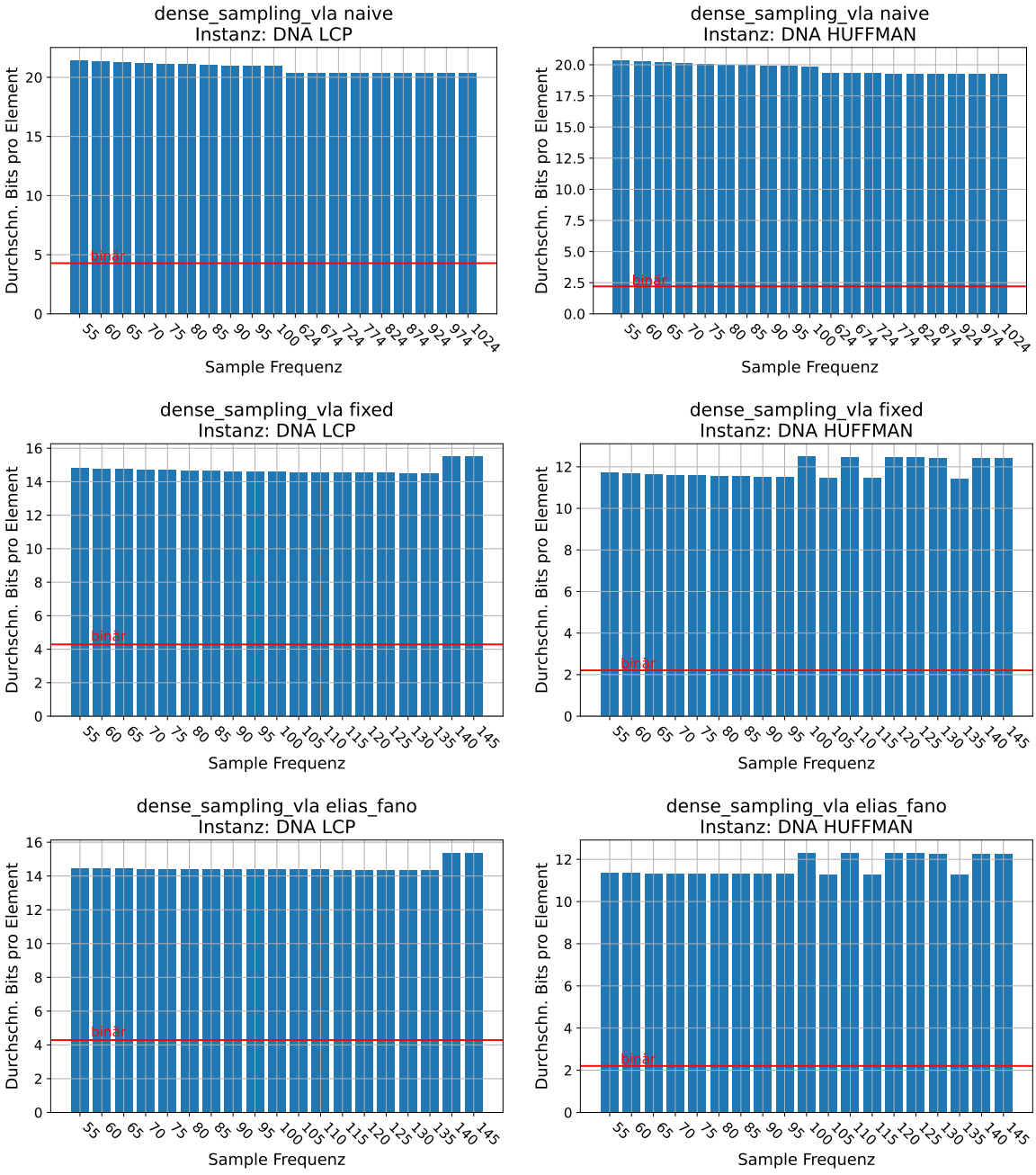

Abbildung A.6.: Platzbedarf für Dense Sampling DNA.

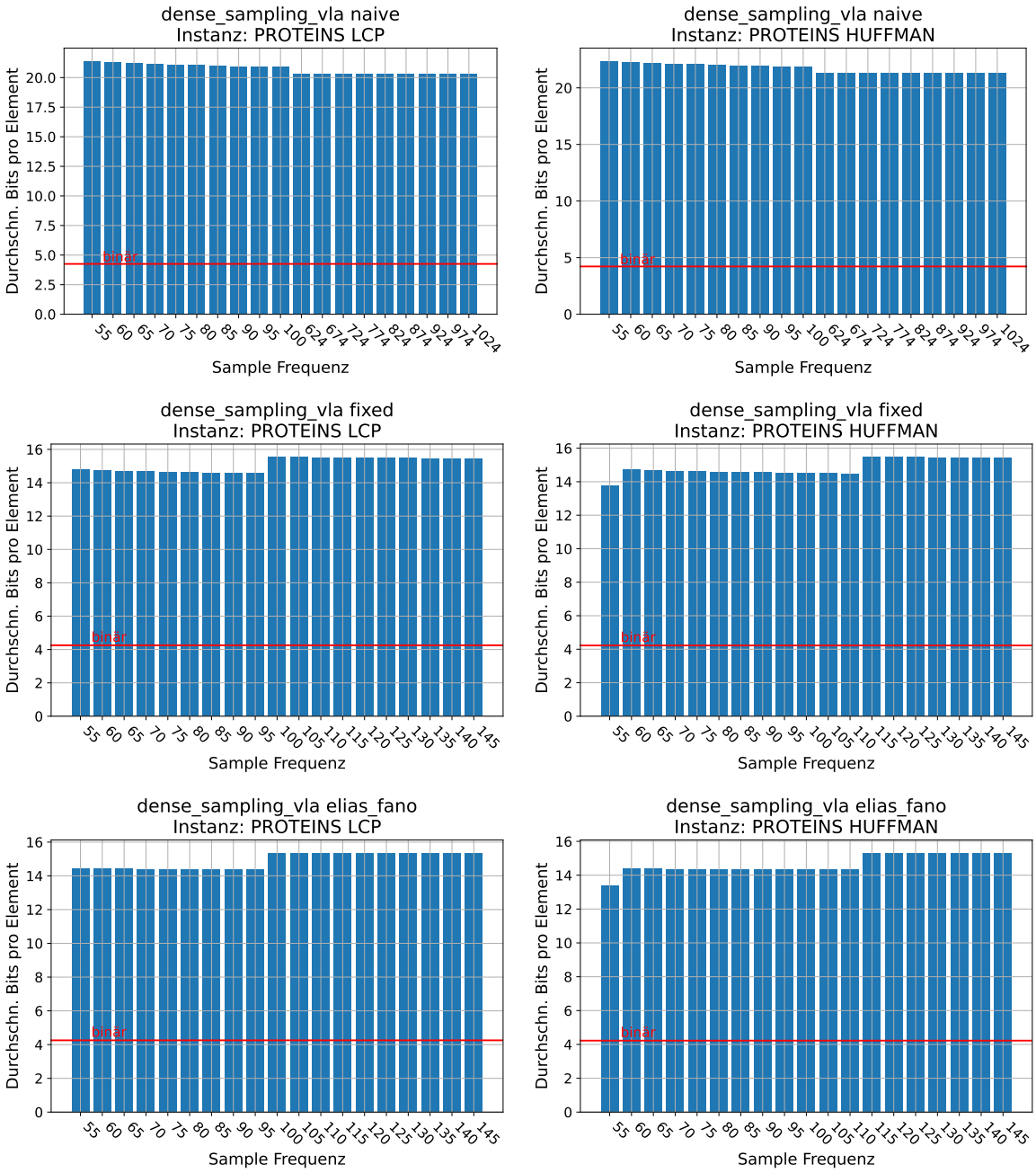

Abbildung A.7.: Platzbedarf für Dense Sampling PROTEINS.

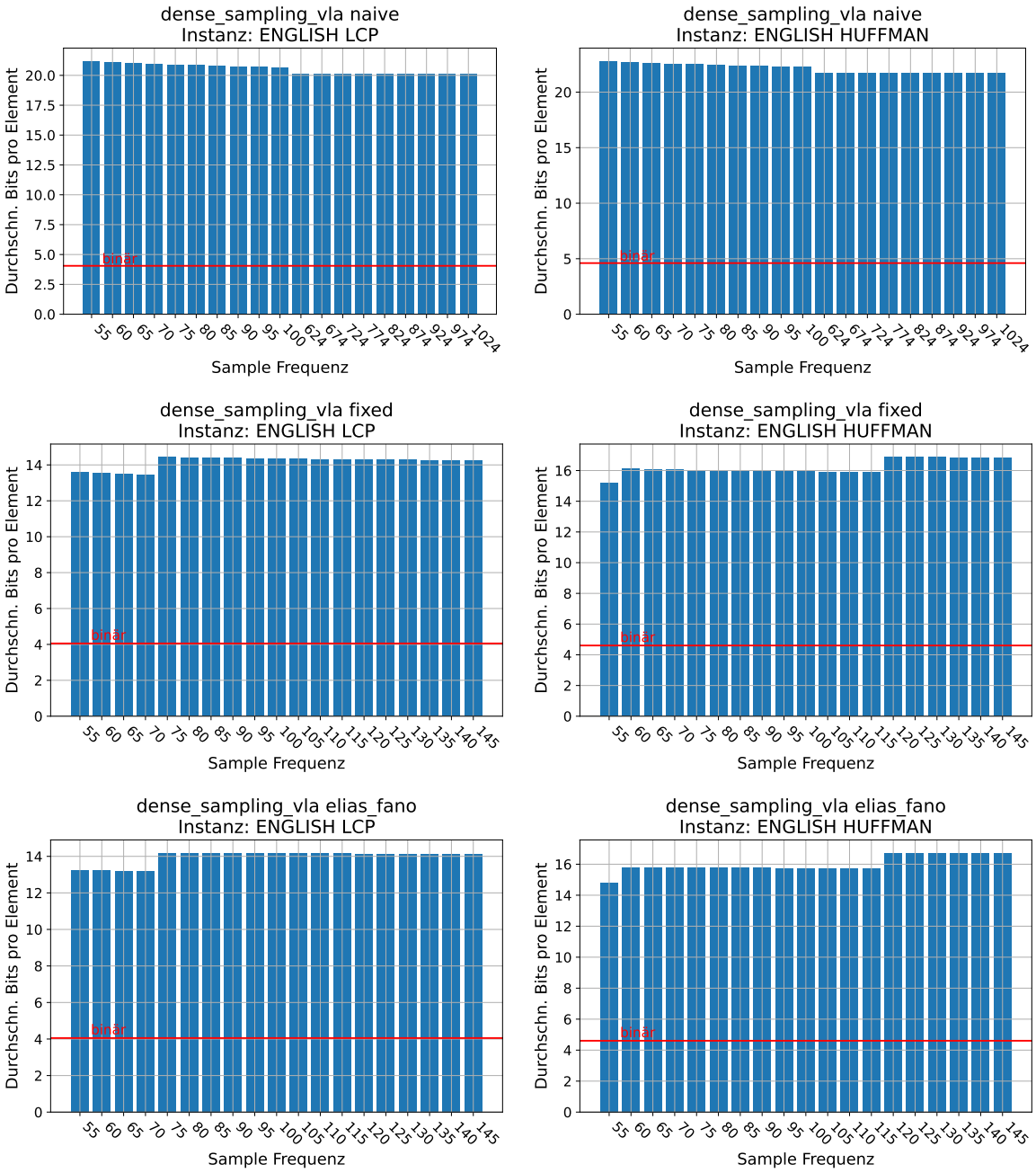

Abbildung A.8.: Platzbedarf für Dense Sampling ENGLISH.

Direct Addressable Codes

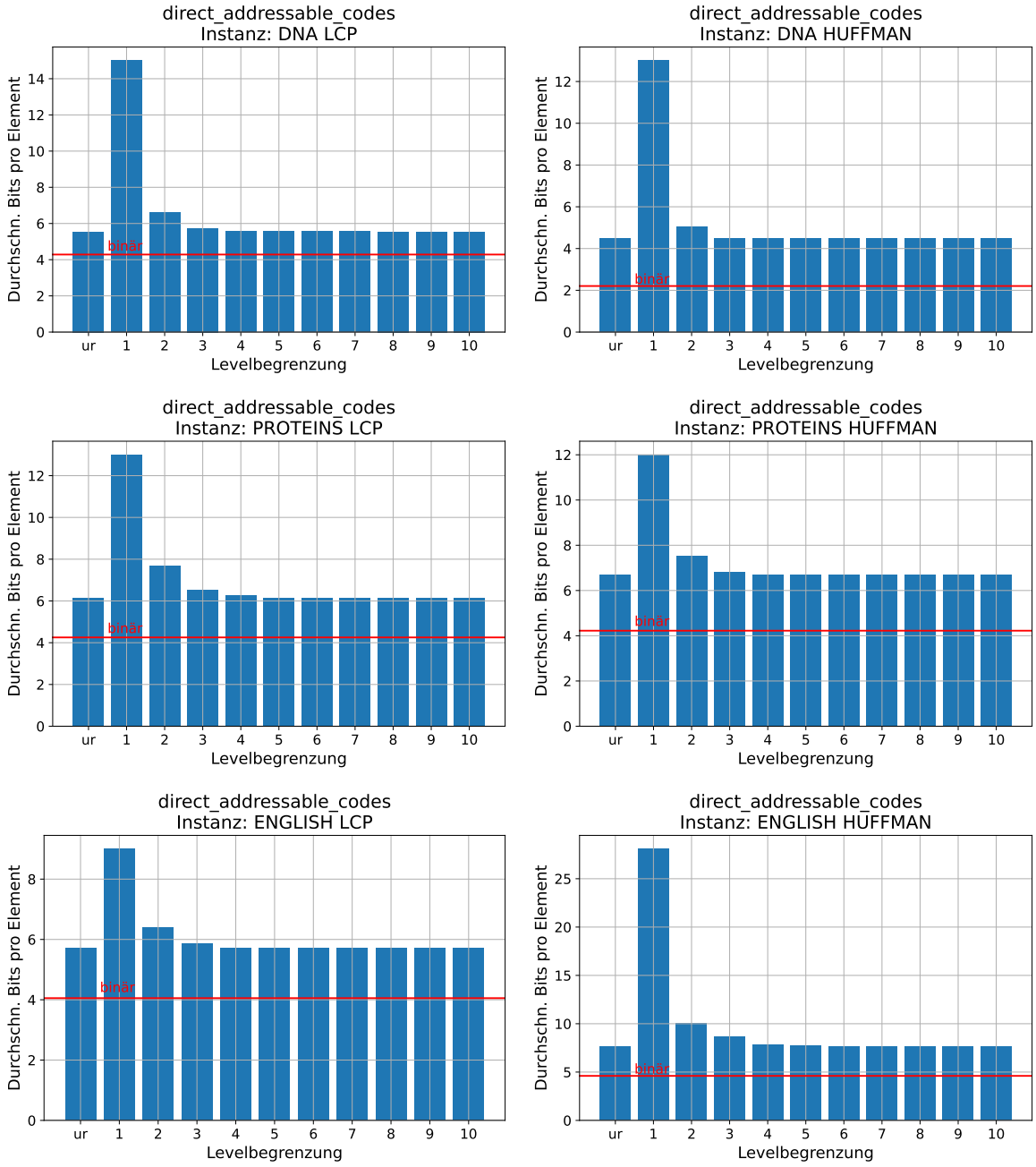

Abbildung A.9.: Platzbedarf für Direct Addressable Codes

Zugriffszeit

In diesem Abschnitt werden die Zugriffszeiten auf die VBL Arrays abgebildet, welche nicht in der Arbeit vorgestellt werden. Der Parameter k auf der x-Achse beschreibt die Anzahl der Zugriffe auf die Arrays und die Zahlen in der Legende beschreiben die verschiedenen Sampling Frequenzen der Dense Sampling VBL Arrays.

Sparse Sampling

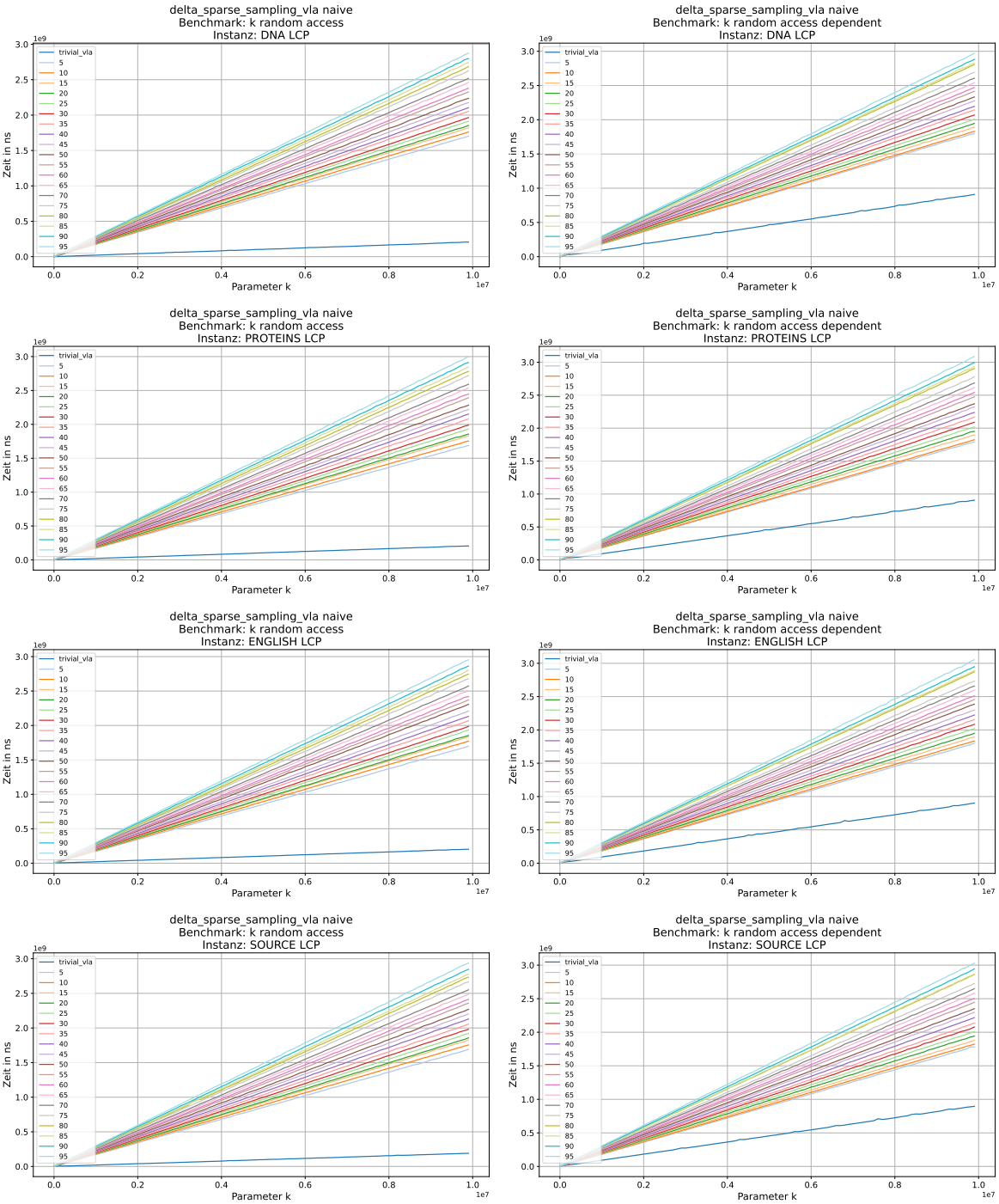

Abbildung A.10.: Zugriffszeiten Sparse Sampling für Elias-Delta kodierte Daten mit trivialen Arrays als Sampling Methode

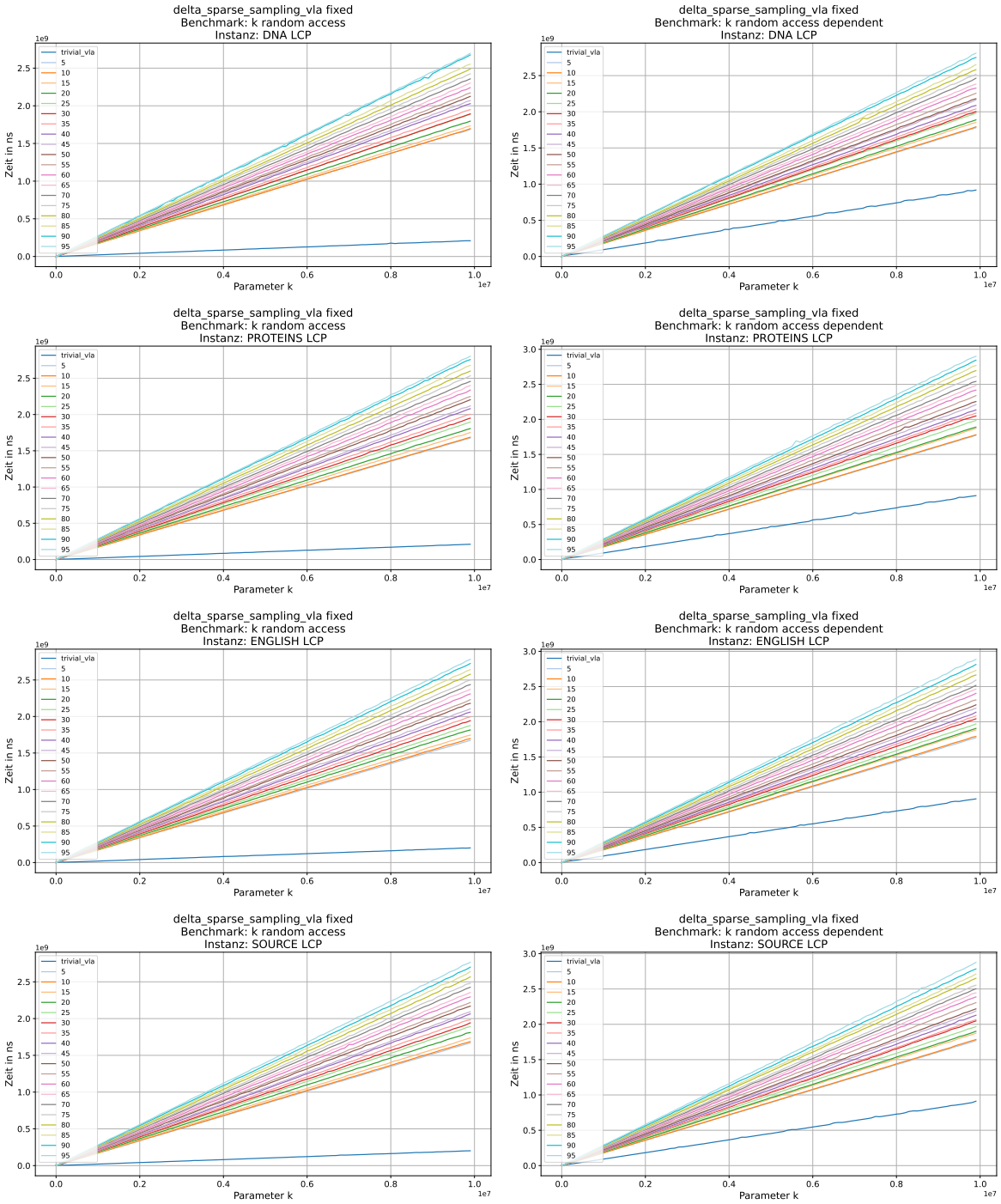

Abbildung A.11.: Zugriffszeiten Sparse Sampling für Elias-Delta kodierte Daten mit Arrays mit Elementen fester Größe als Sampling Methode

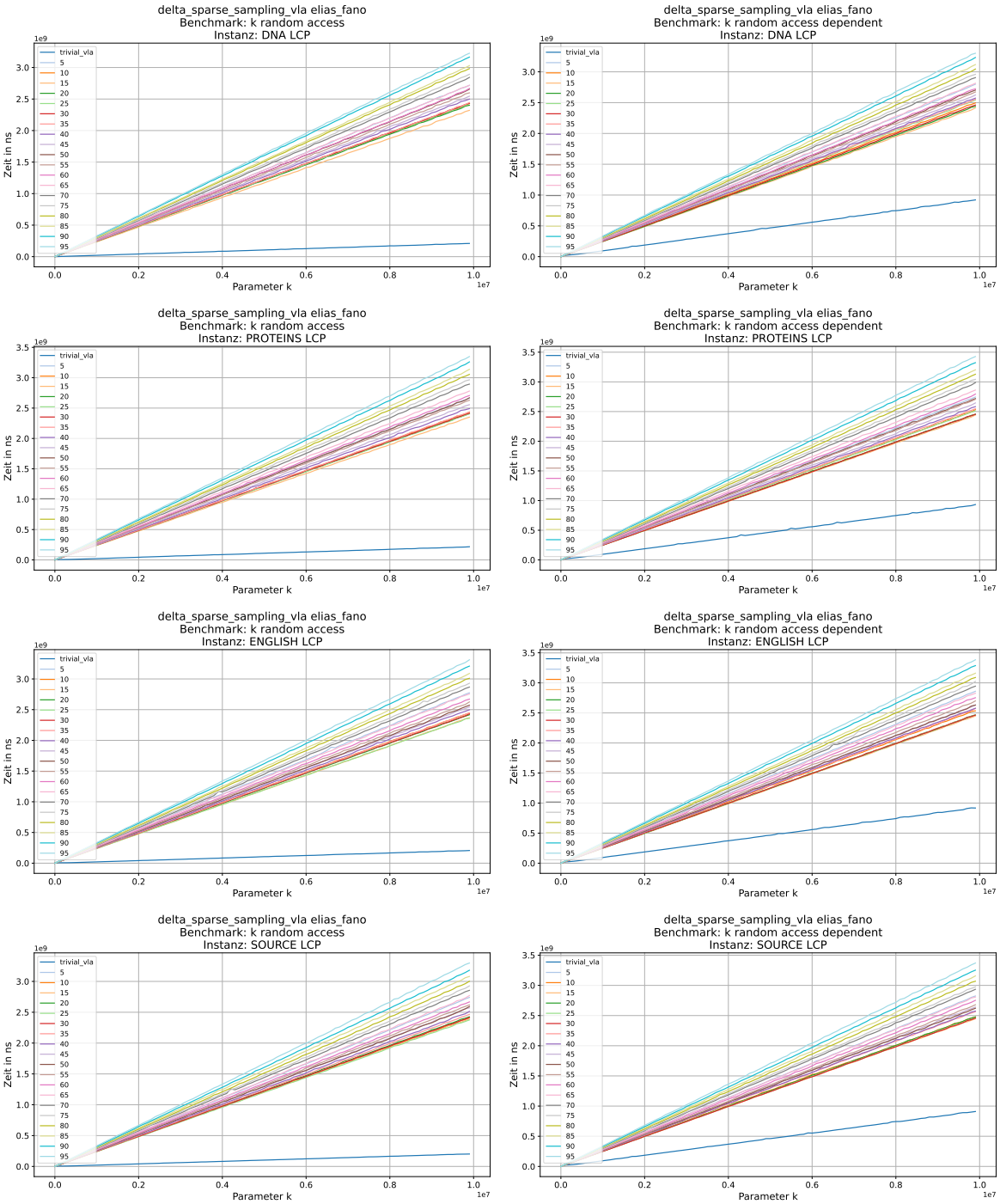

Abbildung A.12.: Zugriffszeiten Sparse Sampling für Elias-Delta kodierte Daten mit Elias-Fano als Sampling Methode

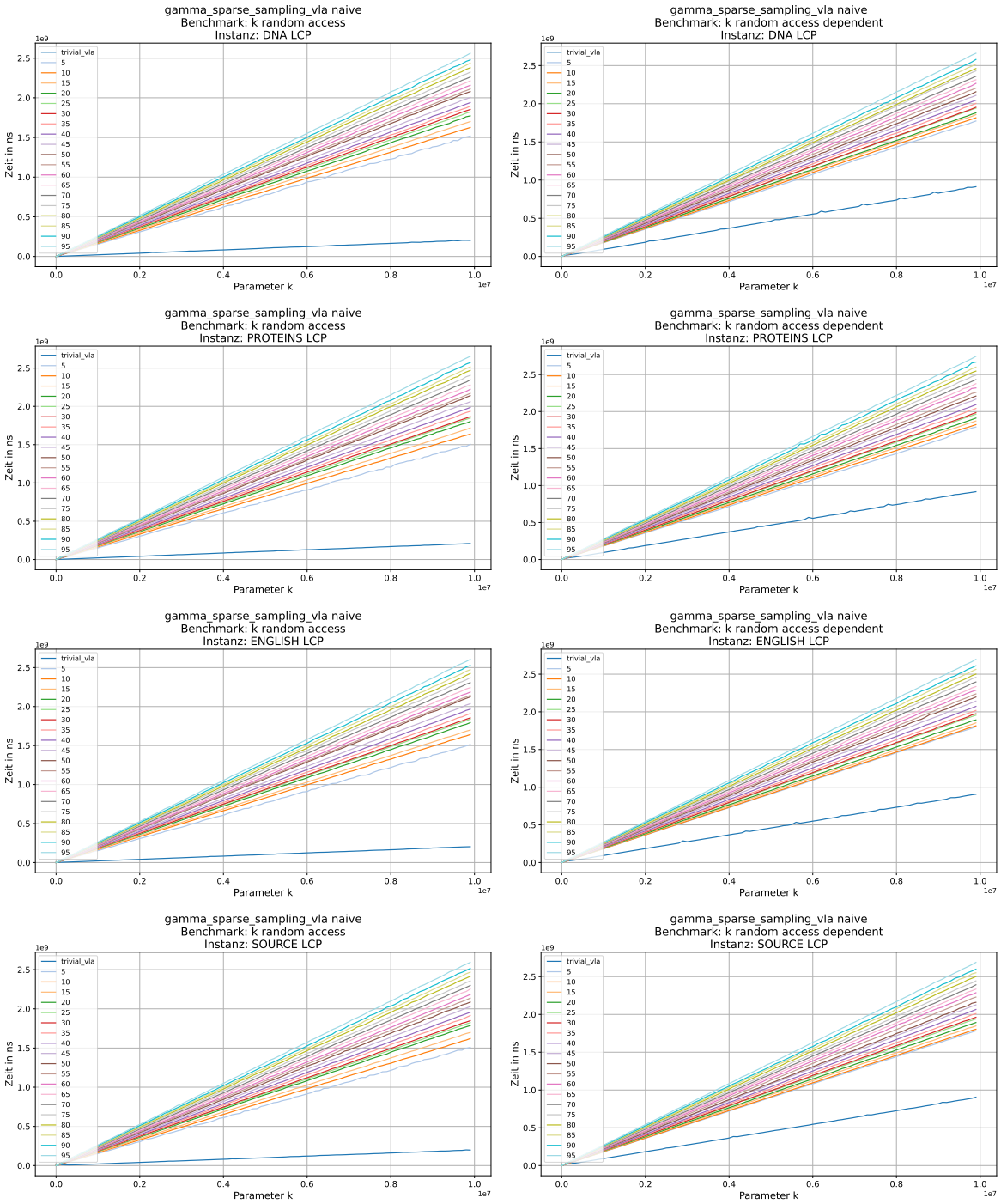

Abbildung A.13.: Zugriffszeiten Sparse Sampling für Elias-Gamma kodierte Daten mit trivialen Arrays als Sampling Methode

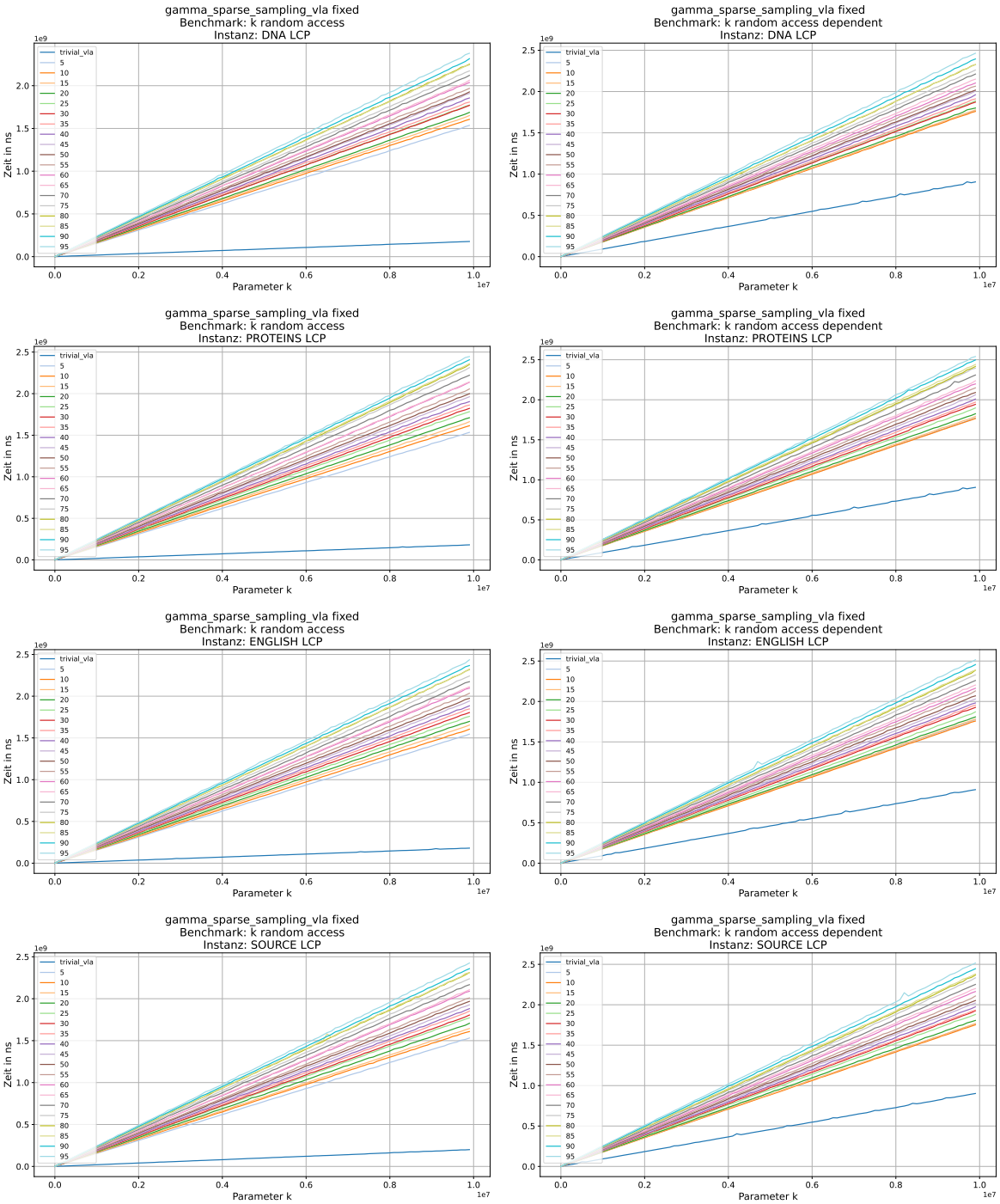

Abbildung A.14.: Zugriffszeiten Sparse Sampling für Elias-Gamma kodierte Daten mit Arrays mit Elementen fester Größe als Sampling Methode

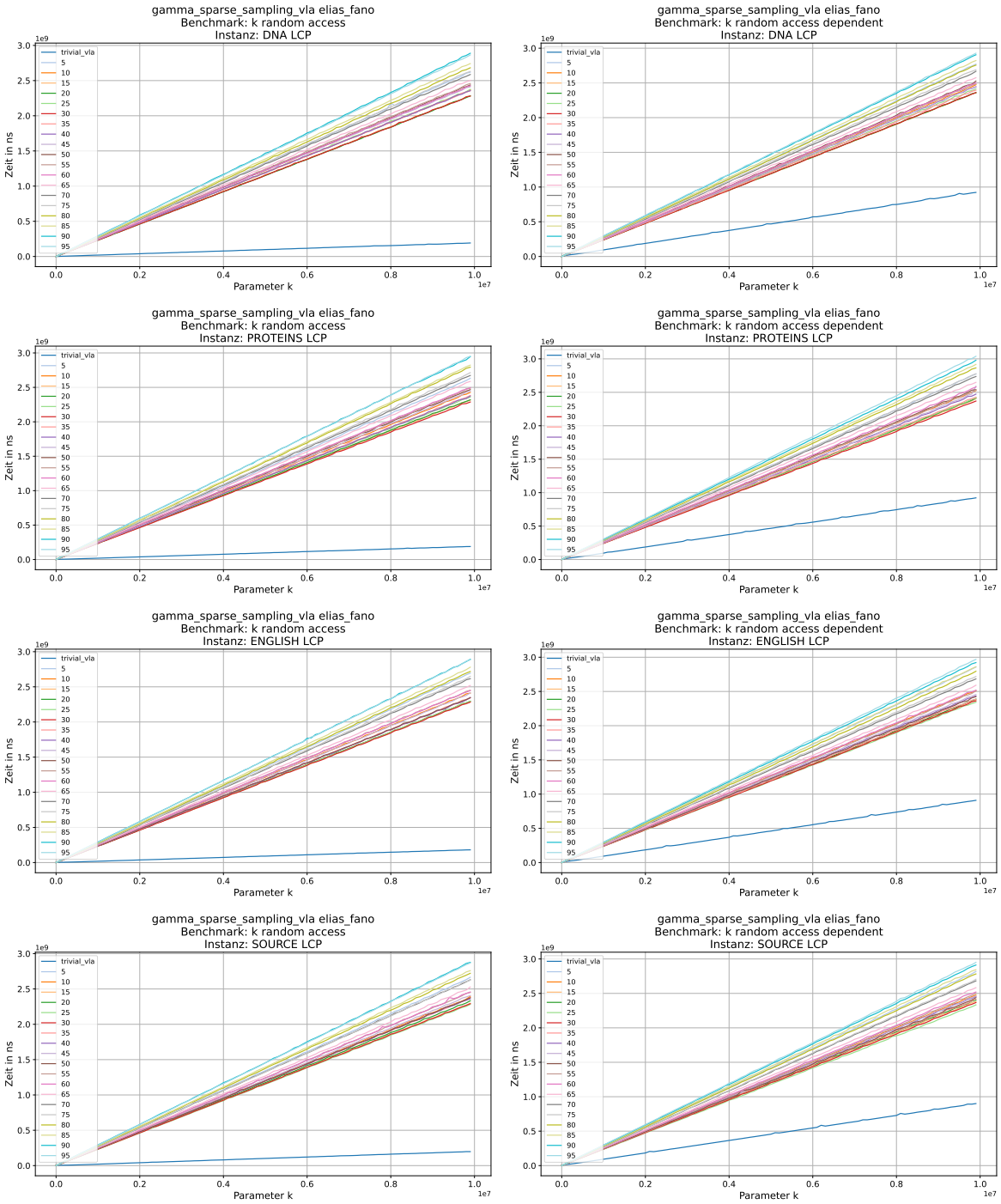

Abbildung A.15.: Zugriffszeiten Sparse Sampling für Elias-Gamma kodierte Daten mit Elias-Fano als Sampling Methode

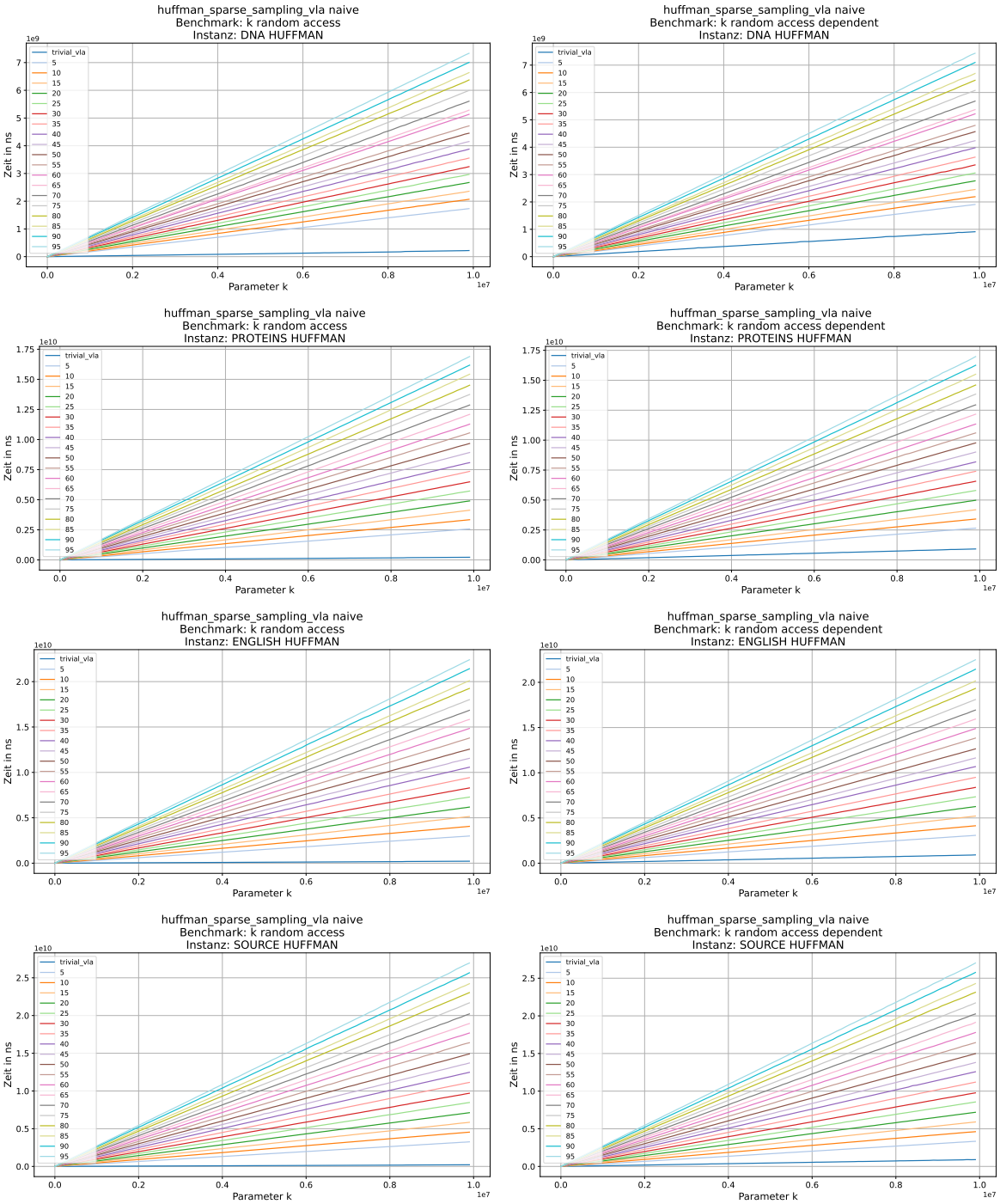

Abbildung A.16.: Zugriffszeiten Sparse Sampling für Huffman Daten mit trivialen Arrays als Sampling Methode

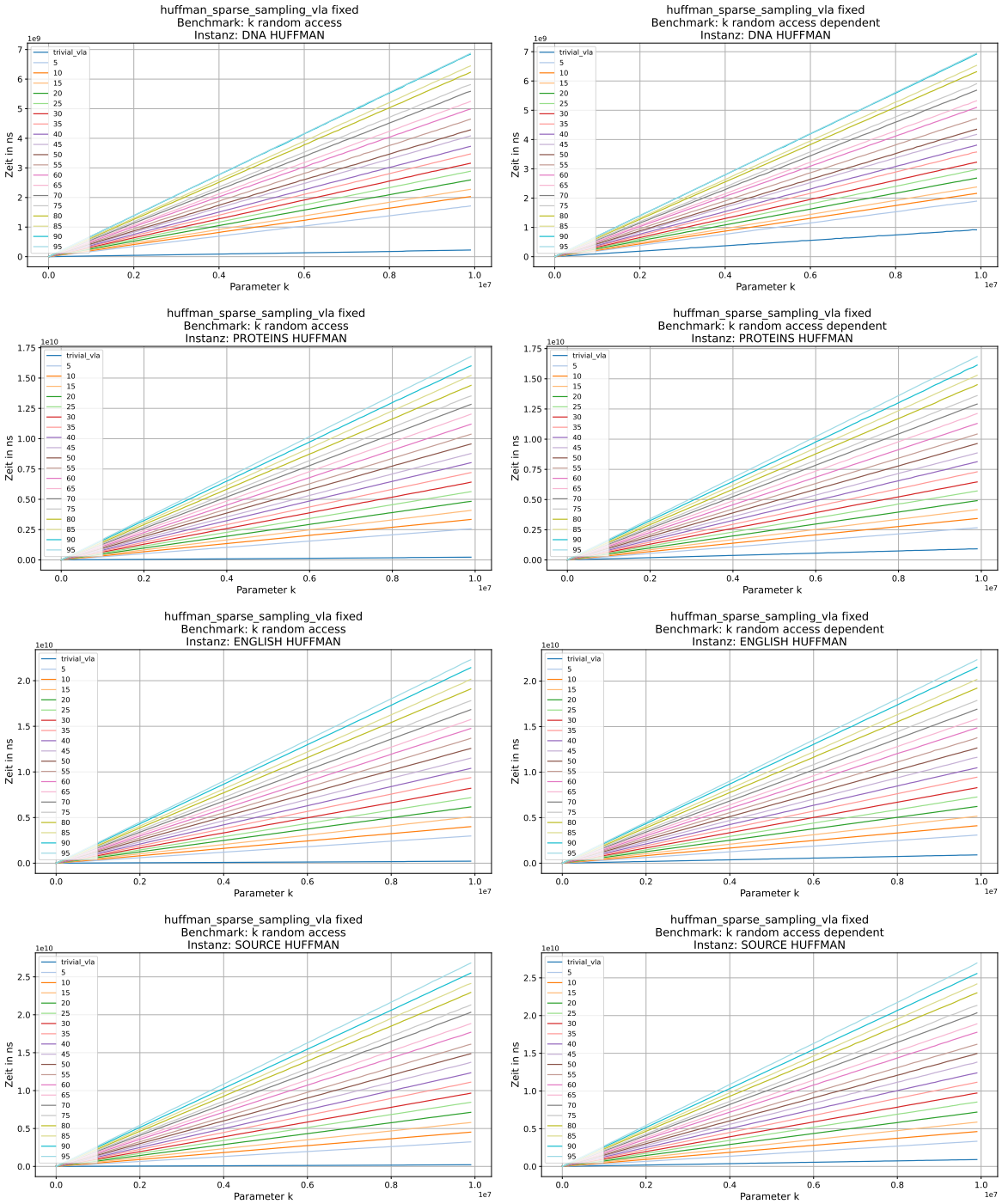

Abbildung A.17.: Zugriffszeiten Sparse Sampling für Huffman Daten mit Arrays mit Elementen fester Größe als Sampling Methode

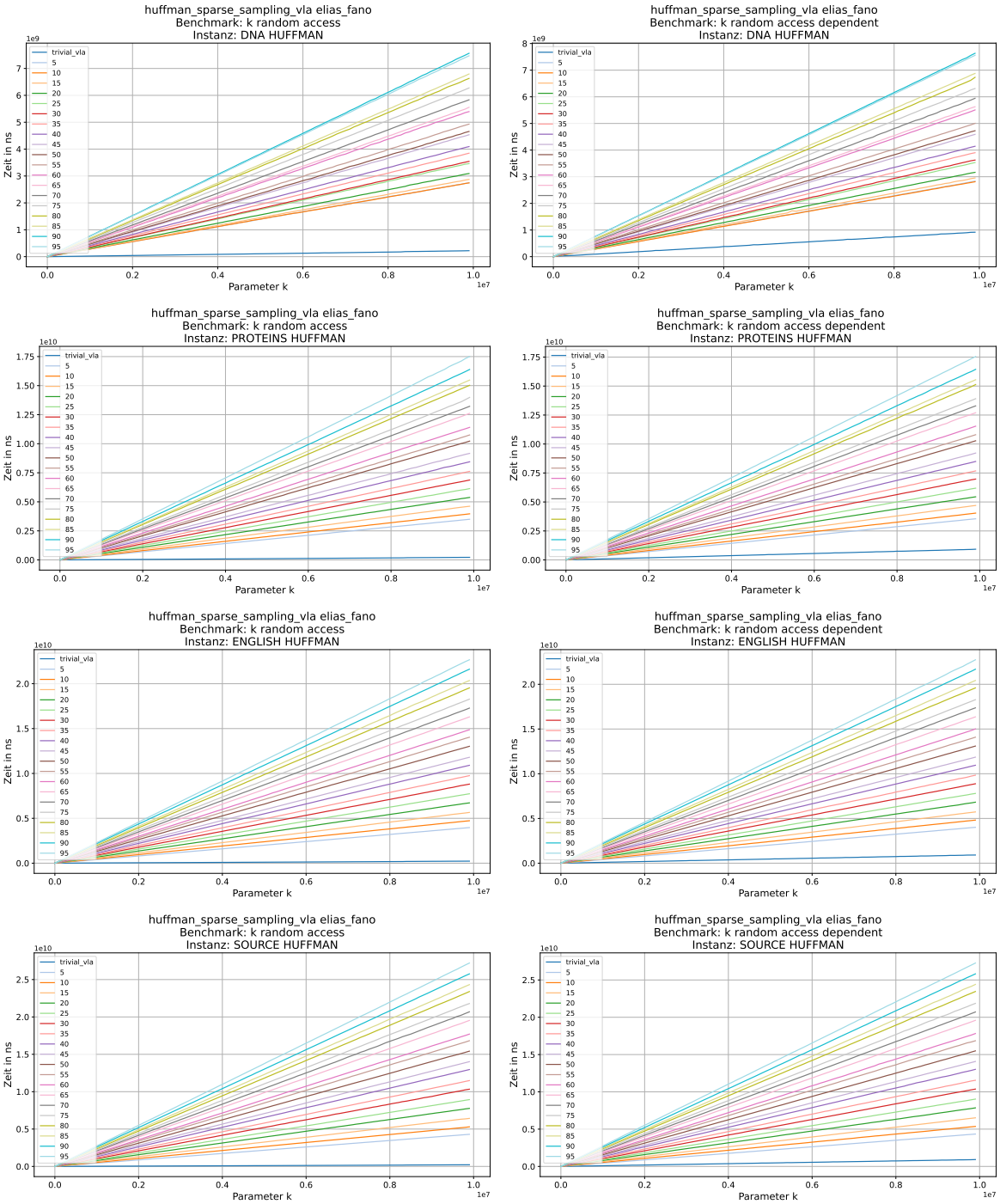

Abbildung A.18.: Zugriffszeiten Sparse Sampling für Huffman Daten mit Elias-Fano als Sampling Methode

Dense Sampling

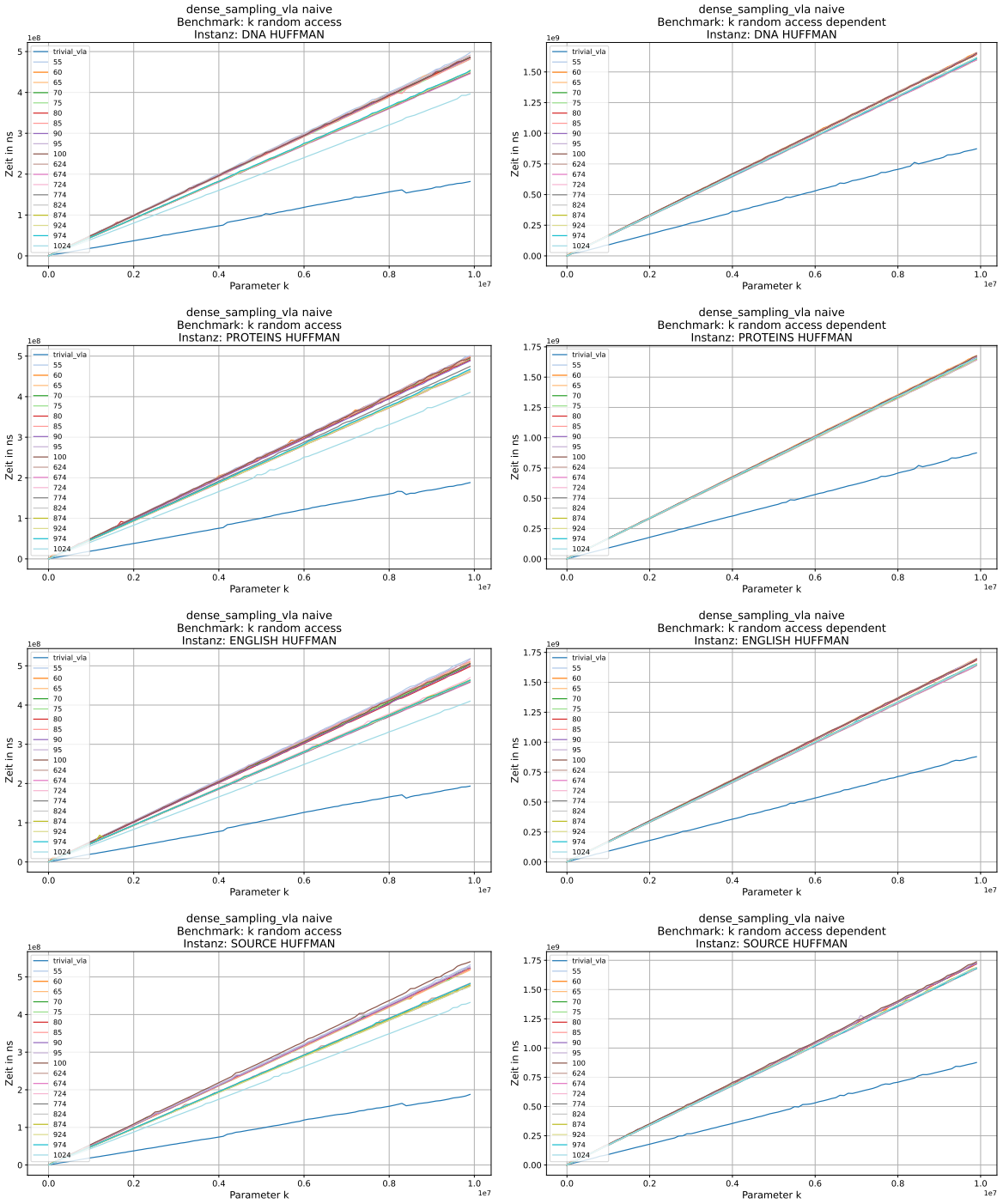

Abbildung A.19.: Zugriffszeiten Dense Sampling für Huffman Daten mit trivialen Arrays als Sampling Methode

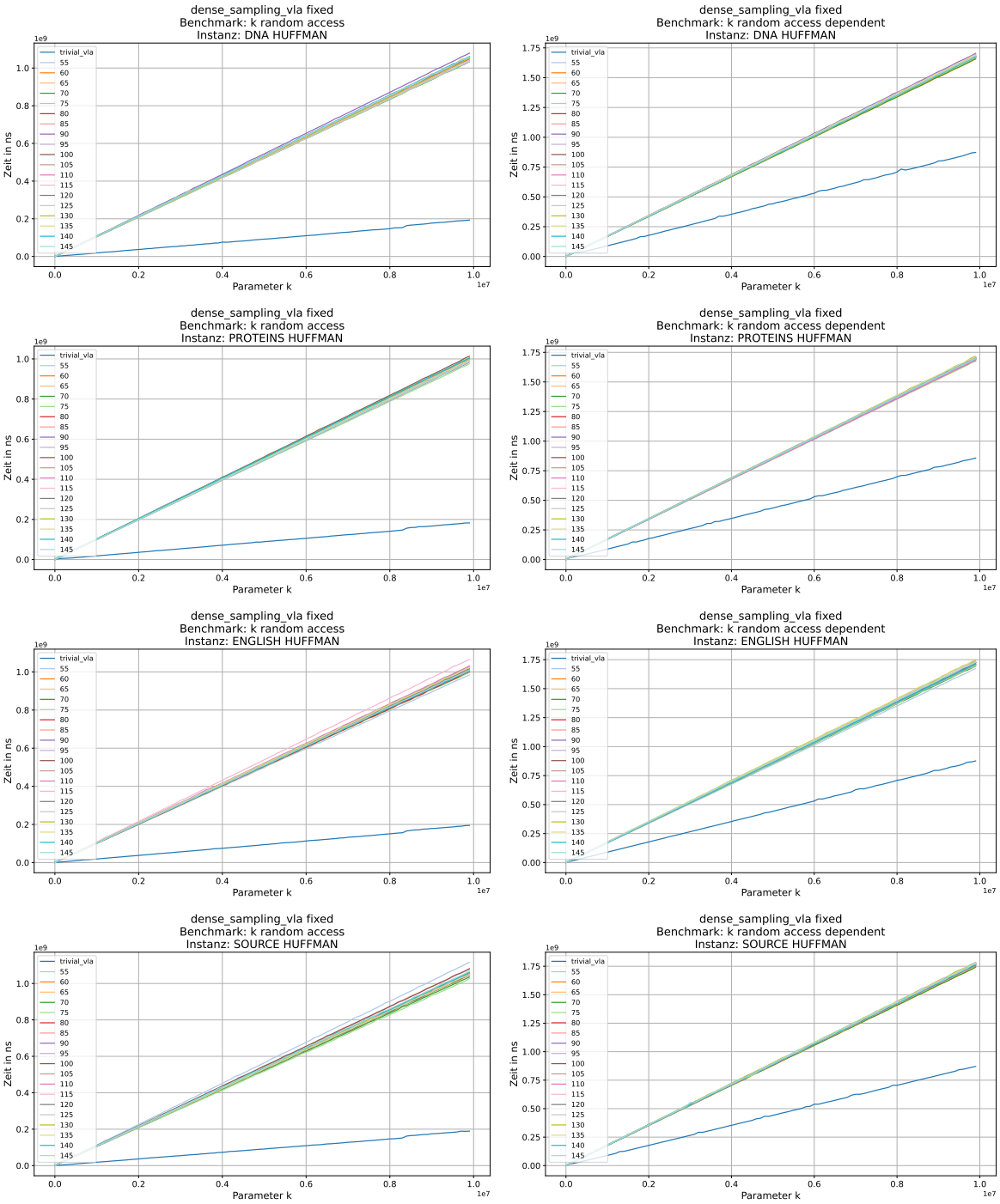

Abbildung A.20.: Zugriffszeiten Dense Sampling für Huffman Daten mit Arrays mit Elementen fester Größe als Sampling Methode

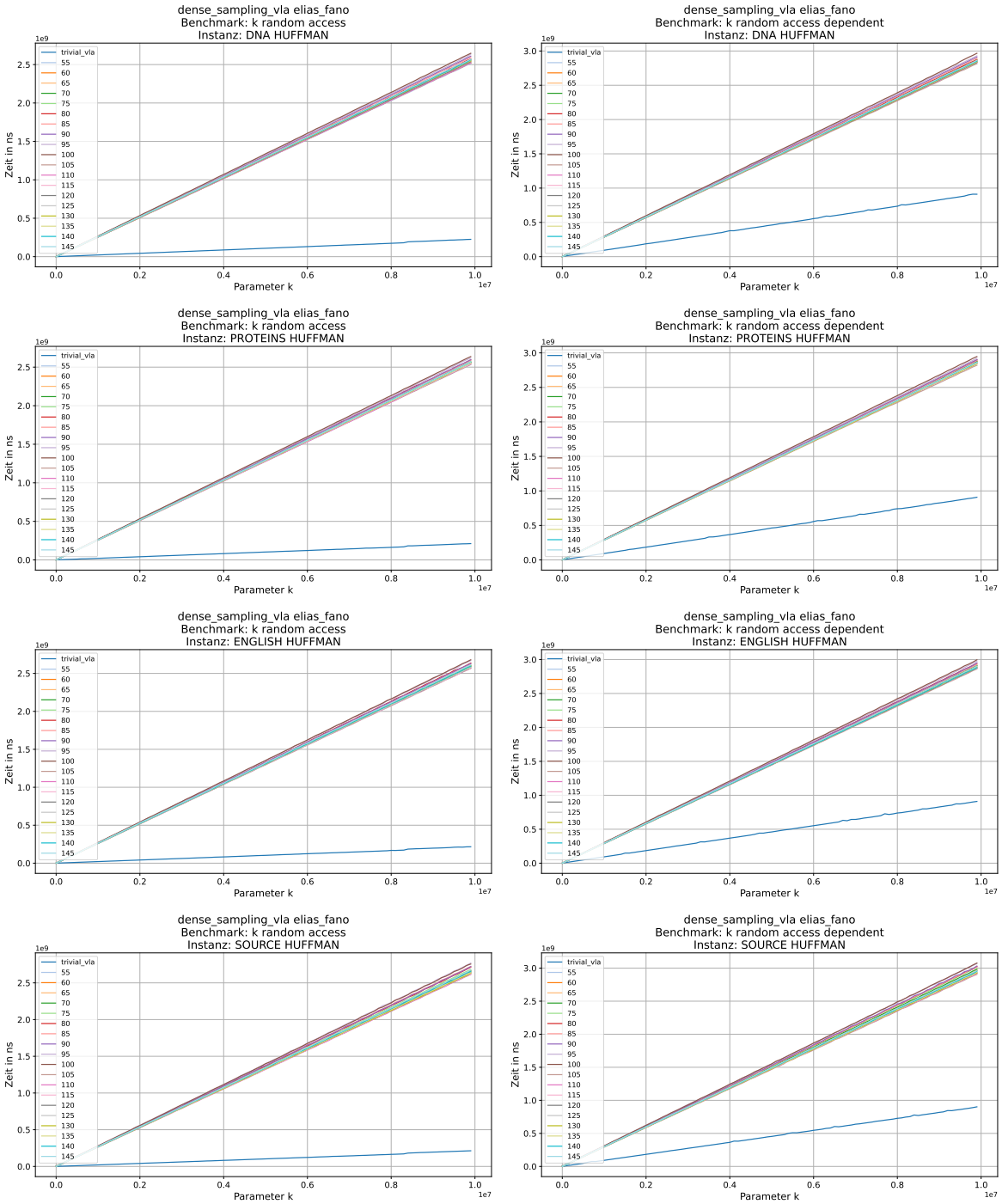

Abbildung A.21.: Zugriffszeiten Dense Sampling für Huffman Daten mit Elias-Fano als Sampling Methode

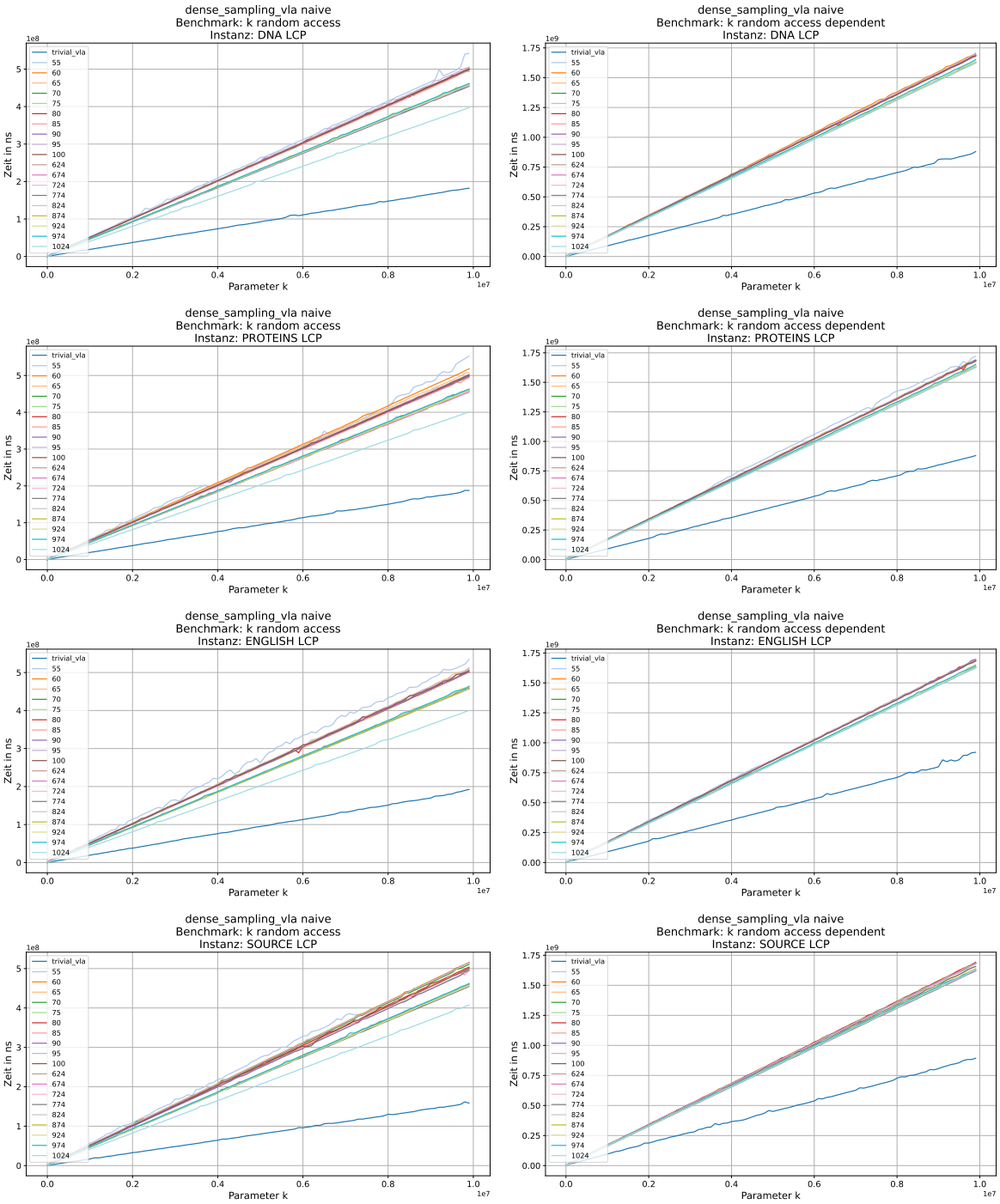

Abbildung A.22.: Zugriffszeiten Dense Sampling für LCP Daten mit trivialen Arrays als Sampling Methode

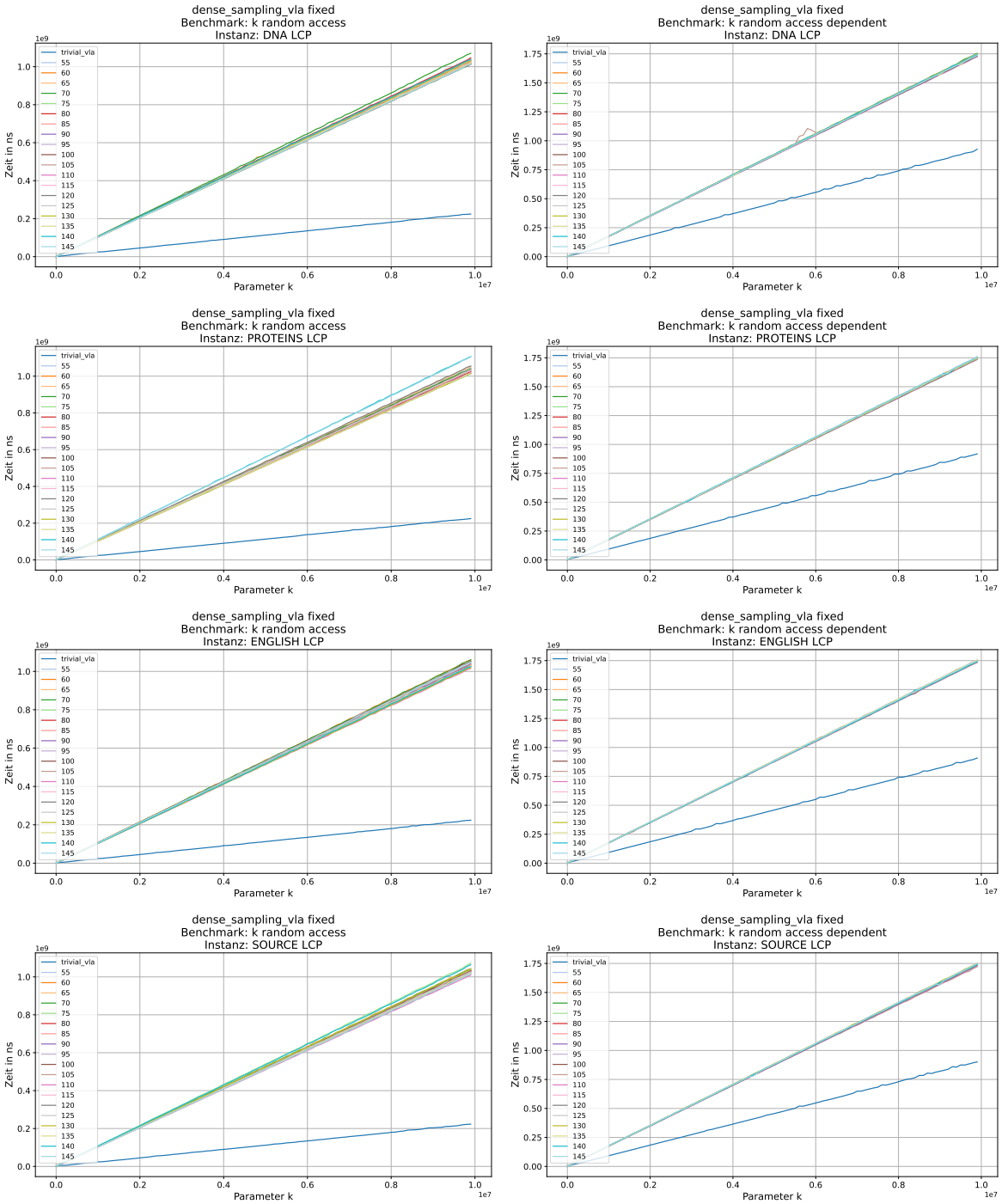

Abbildung A.23.: Zugriffszeiten Dense Sampling für LCP Daten mit Arrays mit Elementen fester Größe als Sampling Methode

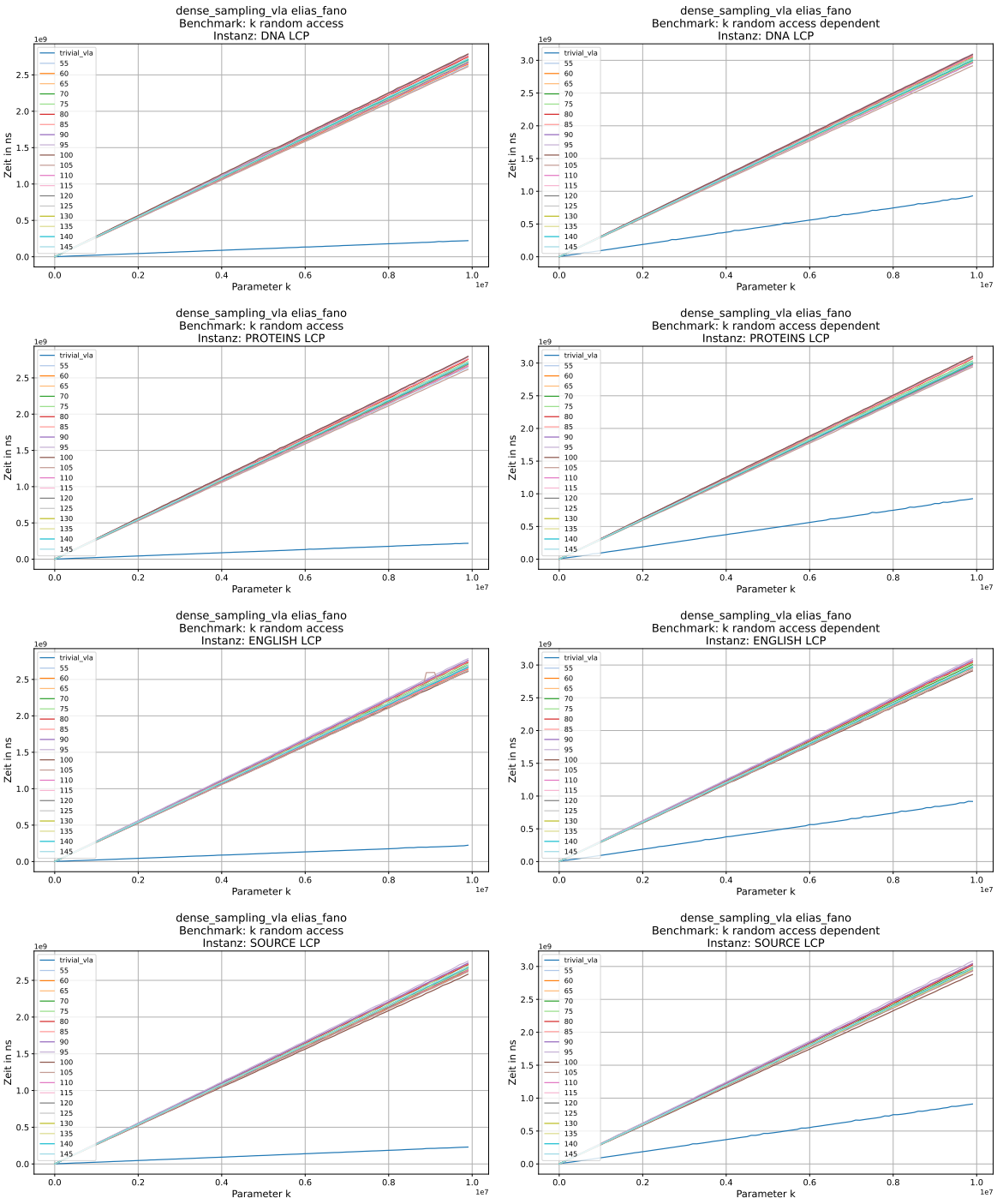

Abbildung A.24.: Zugriffszeiten Dense Sampling für LCP Daten mit Elias-Fano als Sampling Methode

Direct Addressable Codes

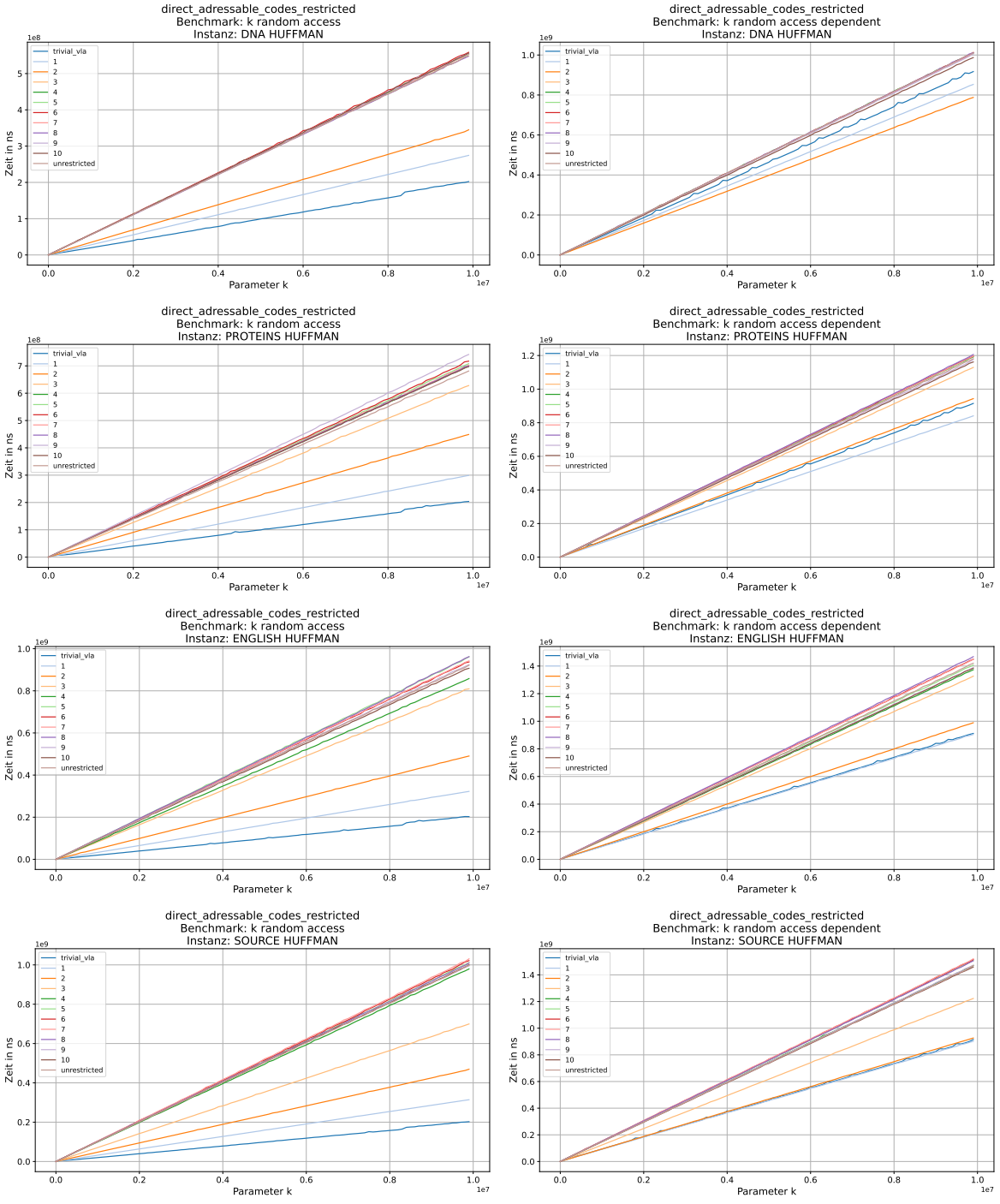

Abbildung A.25.: Zugriffszeiten Direct Addressable Codes für Huffman Daten

...

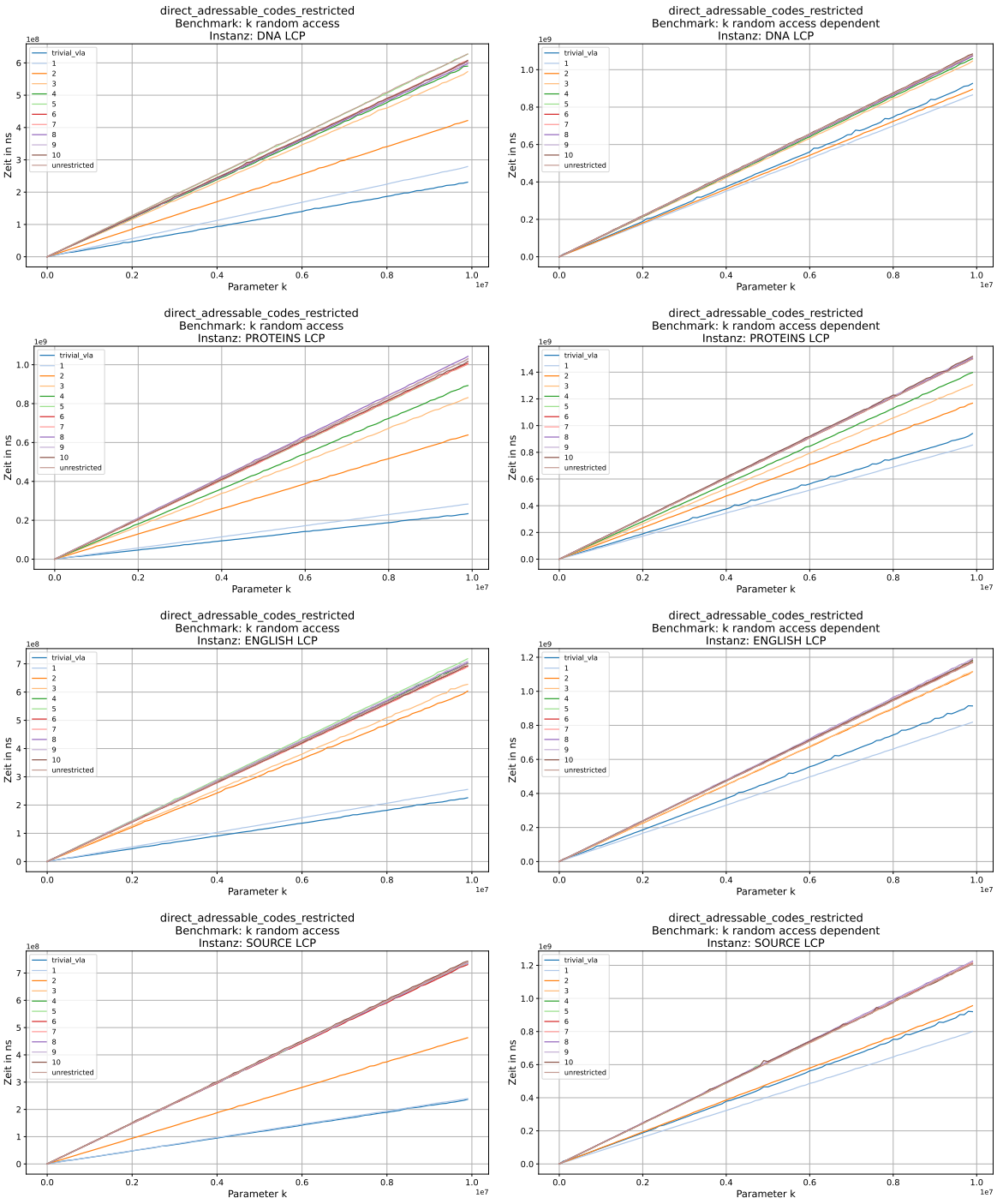

Abbildung A.26.: Zugriffszeiten Direct Addressable Codes für LCP Daten