Bitparallele Mustersuche in grammatikkomprimierten Daten

Omar Shhade

Bachelorarbeit
zur Erlangung des akademischen Grades Bachelor of Science (B. Sc.)

vorgelegt dem
Fachbereich Mathematik, Naturwissenschaften und Informatik der
Technischen Hochschule Mittelhessen

Gießen, September 2026



Referent: Prof. Dr. Michael Elberfeld
Korreferent: Prof. Dr. Martin Bokler

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Gießen, 01.09.2026



Kurzzusammenfassung

Die vorliegende Arbeit erweitert einen bestehenden Suchalgorithmus des Projekts Grampress, der Muster direkt in grammatikkomprimierten Daten findet. Kern der Erweiterung ist eine bitparallele Mustersuche. Sie ersetzt die bisherige Verarbeitung, die jede Position des Musters einzeln prüft. Dadurch werden viele Positionen des Musters gleichzeitig bearbeitet. Zusätzlich verarbeitet der Algorithmus Regeln mit rechten Seiten beliebiger Länge und unterstützt eine Existenzsuche, die beim ersten Treffer abbricht. Ein Vergleich beider Implementierungen auf realen Datensätzen zeigt, wie stark die Laufzeit dadurch sinkt.

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Motivation . . . 1
- 1.2 Problemstellung . . . 2
- 1.3 Zielsetzung . . . 2
- 1.4 Aufbau der Arbeit . . . 3

2 Grammatikbasierte Kompression **4**
- 2.1 Kontextfreie Grammatik . . . 4
- 2.2 Straight-Line Programme . . . 5

3 Der bestehende Suchalgorithmus **6**
- 3.1 Dateiformat . . . 6
- 3.2 Bestimmung der R- und L-Mengen der Regeln . . . 7
- 3.3 Ablauf der Suche . . . 9

4 Bitparallele Mustersuche **13**
- 4.1 Shift-And-Algorithmus . . . 14
- 4.2 Bitparallele Darstellung der R- und L-Mengen . . . 16
- 4.3 Bitparallele Berechnung der Regelinformationen . . . 18
- 4.4 Verwendung von Bitvektoren . . . 23
- 4.5 Verarbeitung von Regeln mit rechten Seiten beliebiger Länge . . . 25

5 Existenzsuche **27**
- 5.1 Early-Exit . . . 27
- 5.2 Existenzsuche mit Tiefensuche . . . 28

6 Implementierung des bitparallelen Suchalgorithmus **29**
- 6.1 Suche mit 64-Bit-Wörtern . . . 30
- 6.2 Bitvektoren mit AWINT . . . 32

7 Experimentelle Analyse **33**
- 7.1 Testumgebung und Datensätze . . . 33
- 7.2 Ergebnisse . . . 34

8 Fazit und Ausblick **37**

Literaturverzeichnis **39**

1 Einleitung

Die Menge digitaler Daten nimmt seit Jahren stetig zu. Texte, Programme und DNA-Sequenzen entstehen heute in einem Umfang, der ohne technische Hilfsmittel kaum noch zu speichern und zu übertragen ist. Kompressionsverfahren sind deshalb ein fester Bestandteil moderner Systeme. Sie verkleinern Dateien, sparen Speicherplatz und verkürzen Übertragungszeiten. Formate wie ZIP oder GZIP werden aus diesem Grund in fast allen Bereichen eingesetzt.

Die heute verbreiteten Kompressionsverfahren verändern jedoch die Form, in der eine Datei vorliegt. Nach der Kompression ist der ursprüngliche Inhalt deshalb nicht mehr unmittelbar zugänglich, denn er ist nur noch in einer kodierten Darstellung enthalten. Diese Darstellung ist zwar deutlich kleiner. Allerdings kann sie nicht wie der ursprüngliche Text gelesen oder verarbeitet werden. Wenn man auf den Inhalt einer komprimierten Datei zugreifen will, muss man die Datei deshalb zuerst dekomprimieren. Komprimierte Daten sind damit zwar gut zu speichern, aber ineffizient zu nutzen.

Besonders deutlich wird dieser Nachteil, wenn in den Daten nach einem bestimmten Muster gesucht werden soll. Klassische Suchalgorithmen arbeiten auf unkomprimiertem Text und können nicht direkt in einer komprimierten Datei suchen. Vor jeder einzelnen Suchanfrage muss die Datei deshalb vollständig dekomprimiert werden, was zusätzliche Zeit und zusätzlichen Speicher kostet. Bei großen Dateien kann der dekomprimierte Text zudem den verfügbaren Arbeitsspeicher übersteigen. Der Vorteil der Kompression bleibt damit nur erhalten, solange auf den Inhalt nicht zugegriffen wird. Aus diesem Grund gewinnen Kompressionsverfahren an Bedeutung, die eine Suche direkt auf den komprimierten Daten erlauben und eine vorherige Dekompression vermeiden.

1.1 Motivation

Damit eine Suche ohne vorherige Dekompression möglich ist, muss die komprimierte Darstellung eine Struktur besitzen, aus der sich der Aufbau des ursprünglichen Textes ableiten lässt. Genau das leistet die *grammatikbasierte Kompression*. Bei diesem Verfahren wird der Text nicht in eine kodierte Bitfolge umgewandelt, wie bei den klassischen Kompressionsverfahren, sondern durch eine Menge von Ersetzungsregeln beschrieben, die genau diesen einen Text erzeugen. Jede Regel ordnet einem Symbol eine Folge weiterer Symbole zu, und aus dem schrittweisen Ersetzen der Symbole entsteht der ursprüngliche Text. Die Kompression entsteht dadurch, dass ein Teil des Textes, der an mehreren Stellen vorkommt, nur einmal als Regel gespeichert wird. An allen diesen Stellen wird dann diese eine Regel verwendet.

Der entscheidende Vorteil dieser Darstellung liegt darin, dass die Regeln selbst lesbar bleiben. Ein Suchalgorithmus kann die Regeln der komprimierten Datei nacheinander lesen und für jede Regel bestimmen, welche Teile des gesuchten Musters in dem von ihr erzeugten Text vorkommen. Der Text muss aber dafür nicht dekomprimiert werden. Aus diesem Grund ist die grammatikbasierte Kompression für die Suche in komprimierten Daten besonders geeignet. Viele Arbeiten haben sich deshalb mit der Mustersuche auf einer solchen Darstellung beschäftigt. Lohrey gibt einen Überblick über diese Verfahren [8]. Für jede Regel werden dabei die Präfixe und Suffixe des Musters verwaltet, die am Anfang und am Ende des Textes liegen, den diese Regel erzeugt. Da für jede Regel alle Positionen eines Musters der Länge m einzeln geprüft werden, ergibt sich für eine Grammatik der Größe $|G|$ eine Laufzeit von $O(|G| \cdot m)$.

Auf unkomprimiertem Text verfolgt die Mustersuche einen anderen Ansatz. Baeza-Yates und Gonnet haben mit dem *Shift-And-Algorithmus* ein Verfahren entwickelt, das den Zustand der Suche nicht als Feld einzelner Werte, sondern als Folge von Bits eines Maschinenwortes darstellt [2]. Alle Positionen des Musters werden dadurch nicht nacheinander, sondern durch wenige Bitoperationen gemeinsam verarbeitet [11]. Für ein Muster der Länge m und eine Wortbreite von w Bits, die bei heutigen Rechnern in der Regel 64 beträgt, werden dafür $\lceil m/w \rceil$ Maschinenwörter benötigt. Bei einem Text der Länge n ergibt

sich damit eine Laufzeit von $O(n\lceil m/w \rceil)$, und für Muster mit $m \leq w$ reicht ein einziges Maschinenwort aus. Eine solche Verarbeitung wird im Folgenden als *bitparallele Mustersuche* bezeichnet.

Beide Ansätze ermöglichen die Suche, betreffen dabei aber verschiedene Teile des Problems. Die grammatikbasierte Kompression verringert die Menge der Daten, die überhaupt durchsucht werden muss, während die bitparallele Verarbeitung den Aufwand für jeden einzelnen Schritt reduziert. Die Motivation dieser Arbeit besteht darin, beide Ansätze zu verbinden. Die für jede Regel verwalteten Präfixe und Suffixe des Musters werden dafür als Bits eines Maschinenwortes dargestellt. Alle Positionen des Musters lassen sich dann gemeinsam prüfen. Der Faktor m in der Laufzeit der Suche wird dadurch durch den deutlich kleineren Faktor $\lceil m/w \rceil$ ersetzt [11].

1.2 Problemstellung

Ein Werkzeug, das die grammatikbasierte Kompression mit einer Suche auf den komprimierten Daten ermöglicht, ist *Grampress*, ein in Rust entwickeltes Kommandozeilenwerkzeug. Die Kompression und die Dekompression wurden von Graf umgesetzt [5]. Mulahusic hat das Werkzeug anschließend um eine Suche nach Mustern erweitert, die direkt auf dem komprimierten Text arbeitet [9]. Dieser Suchalgorithmus verarbeitet die Regeln in der Reihenfolge, in der sie in der Datei kodiert sind. Für jede Regel werden dabei die Präfixe und Suffixe des Musters bestimmt, die am Anfang und am Ende des Textes liegen, den diese Regel erzeugt. Ein Vorkommen des Musters wird erkannt, sobald ein solches Präfix und ein passendes Suffix zweier benachbarter Regeln zusammen das vollständige Muster ergeben. Auf diese Weise lassen sich alle Vorkommen eines Musters bestimmen, ohne dass der ursprüngliche Text entpackt werden muss.

Der Algorithmus folgt damit dem Prinzip, das auch die von Lohrey beschriebenen Verfahren verwenden [8]. Er prüft diese Präfixe und Suffixe jedoch für jede Position des Musters einzeln. Für ein Muster der Länge m und eine Grammatik der Größe $|G|$ ergibt sich dadurch die bereits genannte Laufzeit von $O(|G| \cdot m)$. Die Laufzeit wächst also mit der Länge des Musters.

Außerdem ergibt sich eine Einschränkung aus dem Dateiformat. Das ursprüngliche Format von Grampress speicherte jede Regel mit genau zwei Symbolen, und der Suchalgorithmus wurde auf dieser Grundlage entwickelt. Der Suchalgorithmus setzt diese Form deshalb voraus, da die Informationen einer Regel immer aus den Informationen zweier Symbole zusammengesetzt werden. Das Dateiformat wurde jedoch von Najjar erweitert und lässt beliebig viele Symbole pro Regel zu [10]. Für eine solche Datei muss der Suchalgorithmus angepasst werden.

Eine andere Einschränkung liegt in der Ausgabe der Suche. Der Suchalgorithmus zählt immer alle Vorkommen des Musters und verarbeitet dafür die gesamte Grammatik. Oft will man aber nur wissen, ob das Muster im Text überhaupt vorkommt. Ein einziges gefundenes Vorkommen kann diese Frage beantworten. Allerdings bietet der bestehende Suchalgorithmus eine solche Suche nicht an.

1.3 Zielsetzung

Aus diesen Einschränkungen ergeben sich drei Ziele für diese Arbeit.

Das erste Ziel besteht darin, die positionsweise Verarbeitung des bestehenden Suchalgorithmus durch eine bitparallele Verarbeitung zu ersetzen. Dafür wird das Prinzip von Baeza-Yates und Gonnet auf die Suche in grammatikkomprimierten Daten übertragen [2]. Die Präfixe und Suffixe des Musters, die der Algorithmus für jede Regel verwaltet, werden nicht mehr einzeln geprüft, sondern als Bits eines Maschinenwortes dargestellt und durch wenige Bitoperationen gemeinsam verarbeitet. Für Muster mit höchstens 64 Zeichen reicht dafür ein einziges Maschinenwort aus. Längere Muster werden auf mehrere Maschinenwörter verteilt, sodass die Suche auch für sie möglich bleibt. Der Faktor m in der Laufzeit

wird dadurch durch den kleineren Faktor $\lceil m/64 \rceil$ ersetzt, und der Vorteil gegenüber der bisherigen Verarbeitung wächst mit der Länge des Musters.

Das zweite Ziel ist die Anpassung der Suche an das erweiterte Dateiformat. Der Suchalgorithmus soll Regeln mit beliebig vielen Symbolen direkt verarbeiten können, ohne sie vorher in eine Form mit genau zwei Symbolen umzuwandeln. Eine solche Umwandlung würde zusätzliche Regeln erzeugen und damit die Grammatik vergrößern, die der Algorithmus anschließend durchlaufen muss. Stattdessen werden die Symbole einer Regel während der Suche schrittweise miteinander kombiniert.

Das dritte Ziel betrifft die Frage nach der Existenz eines Musters. Neben der bisherigen Zählsuche soll eine *Existenzsuche* umgesetzt werden, die die Verarbeitung beendet, sobald das erste Vorkommen gefunden wurde. Zusätzlich wird untersucht, ob sich diese Suche weiter beschleunigen lässt. Der bestehende Algorithmus durchläuft die Regeln ungerichtet, also ohne Rücksicht darauf, wo ein Vorkommen zu erwarten ist. Mit einer Tiefensuche [3] steht dagegen ein gerichtetes Verfahren zur Verfügung. Ob ein Vorkommen dadurch schneller gefunden wird, wird experimentell überprüft.

1.4 Aufbau der Arbeit

Die Arbeit folgt dem Aufbau, der sich aus den genannten Zielen ergibt. Abschnitt 2 beschreibt zunächst die Grundlagen der grammatikbasierten Kompression. Dazu werden zuerst kontextfreie Grammatiken erklärt und danach Straight-Line Programme, eine eingeschränkte Klasse kontextfreier Grammatiken, die genau einen Text erzeugt.

Abschnitt 3 stellt den bestehenden Suchalgorithmus von Mulahusic vor, der den Ausgangspunkt dieser Arbeit bildet. Dabei wird zuerst das bestehende Dateiformat vorgestellt, danach werden die R- und L-Mengen definiert, in denen der Suchalgorithmus die Präfixe und Suffixe des Musters in den Regeln speichert. Zum Schluss wird gezeigt, wie daraus ein Vorkommen des Musters erkannt und die Anzahl aller Vorkommen bestimmt wird.

Abschnitt 4 greift das erste Ziel auf und beschreibt die Umwandlung von dem bestehenden Suchalgorithmus in eine bitparallele Mustersuche. Zunächst wird der Shift-And-Algorithmus vorgestellt, an dem sich die bitparallele Verarbeitung orientiert. Danach wird gezeigt, wie die R- und L-Mengen als Bits dargestellt und mit Bitoperationen berechnet werden. Dabei wird zwischen Mustern unterschieden, die in ein einzelnes Maschinenwort passen, und längeren Mustern, für die mehrere Maschinenwörter benötigt werden. Am Ende des Abschnitts folgt das zweite Ziel, also die Anpassung des Suchalgorithmus an das erweiterte Dateiformat, in dem eine Regel beliebig viele Symbole enthalten kann.

Abschnitt 5 behandelt das dritte Ziel. Zuerst wird beschrieben, wie die Suche beim ersten gefundenen Vorkommen beendet werden kann. Anschließend wird die Tiefensuche als gerichtetes Verfahren untersucht. Abschnitt 6 zeigt danach, wie die beschriebenen Verfahren umgesetzt wurden. Dabei wird zwischen der Variante für kurze und der Variante für lange Muster unterschieden.

Abschnitt 7 untersucht die entwickelten Suchvarianten experimentell. Die Laufzeit der bitparallelen Implementierung wird dabei für verschiedene Musterlängen mit der Laufzeit der bestehenden Implementierung verglichen. Danach werden die Ergebnisse der Existenzsuche betrachtet. Zum Schluss wird die Laufzeit der bitparallelen Implementierung mit `zgrep` verglichen, einem verbreiteten Werkzeug für die Suche in komprimierten Dateien. Abschnitt 8 fasst die Ergebnisse zusammen und stellt mögliche Erweiterungen des Suchalgorithmus vor.

2 Grammatikbasierte Kompression

Bei der grammatikbasierten Kompression wird der Text in Form einer kontextfreien Grammatik komprimiert. Diese Grammatik macht die Suche direkt auf dem komprimierten Text möglich, ohne dass der Text vorher dekomprimiert werden muss.

Dafür wird in Abschnitt 2.1 zuerst beschrieben, was eine kontextfreie Grammatik ist, denn sie ist die Form, in der der Text komprimiert vorliegt, und liefert damit die Struktur, auf der die Suche später abläuft. Abschnitt 2.2 behandelt danach Straight-Line Programme, eine eingeschränkte Klasse der kontextfreien Grammatiken. Ihre Eigenschaften stellen sicher, dass die Grammatik genau einen Text beschreibt.

2.1 Kontextfreie Grammatik

Eine *kontextfreie Grammatik* (*engl. context-free grammar*, kurz *CFG*) ist eine Darstellung von Texten mithilfe von Ersetzungsregeln. Sie besteht aus einem 4-Tupel $G = (V, \Sigma, P, S)$, wobei die vier Bestandteile die folgende Bedeutung haben [6]:

- V ist eine endliche Menge von Symbolen, die im Verlauf der Erzeugung durch andere Symbole ersetzt werden können. Diese Symbole werden Nichtterminale genannt.
- Σ ist das Alphabet der Zielsprache, also die Symbole, aus denen der erzeugte Text besteht. Seine Symbole heißen Terminale, und es gilt $V \cap \Sigma = \varnothing$.
- P ist eine endliche Menge von Regeln. Jede Regel ordnet einem Nichtterminal eine Folge von Symbolen zu und wird als $X \to w$ geschrieben, wobei X ein Nichtterminal ist und w eine beliebig lange Folge aus Nichtterminalen und Terminalen ist. Formal gilt $P \subseteq V \times (V \cup \Sigma)^*$.
- S ist das Startsymbol, für das $S \in V$ gilt.

Terminale Σ sind Symbole, die nicht weiter expandiert werden können und den Text bilden. Nichtterminale V können zu einem oder mehreren Terminalen bzw. Nichtterminalen expandiert werden. Die Regeln P legen fest, wie die Expansion abläuft: Steht ein Nichtterminal auf der linken Seite einer Regel, so kann es durch die rechte Seite dieser Regel ersetzt werden. Ein solcher Ersetzungsschritt heißt Ableitungsschritt und wird mit „$\Rightarrow$“ notiert. Das Startsymbol bestimmt, an welcher Stelle die Ableitung beginnt.

Beispiel Gegeben sei die kontextfreie Grammatik $G = (V, \Sigma, P, S)$ mit

- $V = \{S, A, B\}$,
- $\Sigma = \{a, b, c\}$,
- $P = \{S \to aAB, A \to bc, B \to ac\}$ und
- dem Startsymbol S.

Ausgehend von S ergibt sich die Ableitung:
$S \Rightarrow aAB \Rightarrow abcB \Rightarrow abcac$

Der Ablauf einer Ableitung kann auch als Ableitungsbaum dargestellt werden. Die Wurzel des Baumes ist mit dem Startsymbol markiert. Ein *innerer Knoten* ist ein Symbol, das im Baum weiter expandiert wird. Das sind die Nichtterminale. Sei X ein innerer Knoten und sei $X \to Y_1 \dots Y_k$ eine Regel, wobei jedes Y_i ein Terminal oder Nichtterminal ist. Dann wird X über diese Regel expandiert. Die Kinder von X sind die Symbole $Y_1, \dots, Y_k$ in genau dieser Reihenfolge. Ein *äußerer Knoten* (auch *Blatt* genannt) ist dagegen ein Symbol, das nicht weiter expandiert wird. Das sind die Terminale, für die es keine Regel gibt. Die äußeren Knoten bzw. Blätter werden von links nach rechts gelesen. Daraus ergibt sich der abgeleitete Text. Abbildung 1 zeigt den Ableitungsbaum zu der obigen Grammatik.

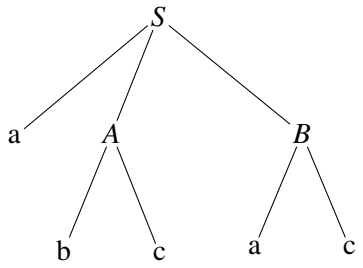

Abbildung 1: Ableitungsbaum der Grammatik G für den Text *abcac*. Die Wurzel ist das Startsymbol S und expandiert zu a, A und B. Die inneren Knoten sind A und B, da sie zu bc bzw. ac expandiert werden. Die äußeren Knoten werden von links nach rechts gelesen und ergeben damit den Text *abcac*.

2.2 Straight-Line Programme

Wie in Abschnitt 2.1 beschrieben, stellt eine kontextfreie Grammatik einen Text mithilfe von Ersetzungsregeln dar. Für die komprimierte Darstellung eines bestimmten Textes muss die Grammatik genau diesen Text erzeugen. Eine kontextfreie Grammatik mit dieser Eigenschaft heißt *Straight-Line Programm* (kurz *SLP*). Damit eine kontextfreie Grammatik ein *SLP* ist, müssen die folgenden zwei Bedingungen erfüllt sein [4]:

- Die erste Bedingung betrifft die Anzahl der Regeln jedes Nichtterminals. Für jedes Nichtterminal $X \in V$ gibt es genau eine Regel der Form $X \to w$ in P. Damit hat X bei der Expansion keine weitere Auswahl und es ist eindeutig festgelegt, wodurch X ersetzt wird. Ein Nichtterminal mit mehreren Regeln wie $X \to ab$ und $X \to ac$ ist deshalb kein *SLP*, da für X zwei verschiedene Expansionen möglich sind.
- Die zweite Bedingung ist, dass es keine Kreise gibt. Ein Kreis entsteht, wenn ein Nichtterminal $X \in V$ bei seiner Expansion direkt oder indirekt wieder zu sich selbst führt. Ein Nichtterminal X darf also nur zu Nichtterminalen expandiert werden, die selbst nicht wieder zu X führen. Die Regel $X \to aX$ ist z. B. ein direkter Kreis, da X auf der rechten Seite seiner eigenen Regel vorkommt. Bei den Regeln $X \to aY$ und $Y \to bX$ ist der Kreis indirekt, da X über Y wieder zu X führt.

Wenn beide Bedingungen erfüllt sind, dann besitzt jedes Nichtterminal X genau eine Regel für die Expansion und die Ableitung von X kommt immer zu einem Ende. Der Text, den X erzeugt, heißt die Expansion von X und wird mit v_X bezeichnet.

In einem SLP darf auch ein Nichtterminal auf der rechten Seite mehrerer Regeln vorkommen. Wenn ein bestimmter Text also mehrfach im ursprünglichen Text vorkommt, wird dafür nur ein Nichtterminal benötigt, das an allen Stellen verwendet werden kann, wo dieser Text auftritt. Dieses Prinzip macht die grammatikbasierte Kompression effizienter [5].

Beispiel Gegeben sei die kontextfreie Grammatik $G = (V, \Sigma, P, S)$ mit

- $V = \{S, A, B\}$,
- $\Sigma = \{a, b, c\}$,
- $P = \{S \to ABc, A \to aB, B \to bc\}$ und
- dem Startsymbol S.

Ausgehend von S ergibt sich die Ableitung:
$S \Rightarrow ABc \Rightarrow aBBc \Rightarrow abcbcc$

Die Grammatik im Beispiel erfüllt beide Bedingungen. Jedes Nichtterminal besitzt genau eine Regel, und kein Nichtterminal führt bei seiner Expansion direkt oder indirekt wieder zu sich selbst. Die Grammatik ist damit ein SLP mit der Expansion $v_S = \texttt{abcbcc}$. Das Nichtterminal B kommt zudem auf den rechten Seiten von S und A vor, sodass die Zeichenfolge `bc` nur einmal gespeichert wird.

3 Der bestehende Suchalgorithmus

In diesem Abschnitt wird gezeigt, wie der bestehende Suchalgorithmus arbeitet, auf dem die vorliegende Arbeit aufbaut. Zunächst wird in Abschnitt 3.1 das Dateiformat beschrieben, mit dem der Algorithmus arbeitet. Dabei wird gezeigt, wie die Regeln der Grammatik in der komprimierten Datei abgelegt sind und welche Form die Terminale, die Nichtterminale und die Startregel dort haben. Anschließend werden in Abschnitt 3.2 die sogenannten R- und L-Mengen mathematisch definiert und erläutert, wobei auch ihre jeweilige Bedeutung für die Suche erklärt wird. Diese Definitionen werden in Abschnitt 3.3 durch die konkrete Berechnung der Mengen ergänzt. Dort wird gezeigt, wie die Mengen für Terminale und für Nichtterminale bestimmt werden und wie ein Muster mithilfe dieser Mengen erkannt und gezählt werden kann. Zuletzt wird beschrieben, wie die Startregel verarbeitet wird, obwohl sie keine binäre Struktur besitzt.

3.1 Dateiformat

Der Suchalgorithmus arbeitet auf Dateien im GPS-Format. Diese Dateien enthalten einen Text, der als SLP komprimiert wurde. Für die Kompression können verschiedene Verfahren verwendet werden, z. B. *Sequitur*, *Bisection* oder *Re-Pair*. Für die Regeln der komprimierten Datei gilt dabei Folgendes [5, 9]:

- **Terminale** werden nicht durch eigene Regeln erzeugt, sondern stehen direkt auf der rechten Seite eines Nichtterminals. Da ein Byte 256 verschiedene Werte annehmen kann, werden die Zahlen von 0 bis 255 für die Terminale reserviert. Sie werden nach der ASCII-Tabelle kodiert, sodass a, b und c als 97, 98 und 99 gespeichert werden.
- **Nichtterminale** sind binäre Regeln. Sei X ein Nichtterminal, dann hat die Regel von X die Form $X \to Y_1 Y_2$, wobei Y_1 und Y_2 Terminale oder Nichtterminale sind. Die Nichtterminale werden ab der Zahl 256 durchnummeriert, sodass die erste Regel die Zahl 256 erhält, die zweite die Zahl 257 und so weiter. Der Algorithmus erkennt an der Zahl also sofort, ob es sich um ein Terminal oder ein Nichtterminal handelt. Die Regeln sind außerdem topologisch sortiert, sodass eine Regel X_i nur auf Regeln X_j mit $j < i$ verweist.
- **Startregel** muss nicht binär wie die Nichtterminale sein. Eine Startregel S kann viele Symbole auf der rechten Seite enthalten und hat die Form $S \to Y_1 \dots Y_k$. Die Startregel steht am Ende der Datei und wird im Gegensatz zu Terminalen und Nichtterminalen nicht nummeriert. Da der Suchalgorithmus binäre Regeln benötigt, wird die Startregel während der Suche schrittweise verarbeitet, indem ihre Symbole von links nach rechts kombiniert werden, d. h. das erste Symbol wird mit dem zweiten kombiniert, das Ergebnis mit dem dritten und so weiter, bis alle Symbole verarbeitet sind. Dadurch wird die binäre Form nachgebildet, ohne dass neue Regeln erzeugt werden müssen.

Beispiel Der Text `abcabcab` wird wie folgt kodiert:

Regel	**Kodierung**
$X_1 \to ab$	$[97, 98]$
$X_2 \to X_1 c$	$[256, 99]$
$S \to X_2 X_2 X_1$	$[257, 257, 256]$

Die Terminale `a` und `b` erhalten ihre ASCII-Werte 97 und 98. Das erste Nichtterminal X_1 erhält die Zahl 256 und X_2 die Zahl 257. In der Datei steht damit die Folge $[97, 98, 256, 99, 257, 257, 256]$. Da jede Regel eines Nichtterminals genau zwei Symbole hat, ist bekannt, wo eine solche Regel anfängt und wo sie endet, sodass zwischen den Regeln keine Trennzeichen benötigt werden. Die Startregel steht am Ende der Datei, sodass alle danach verbleibenden Symbole zu ihr gehören [5, 9].

3.2 Bestimmung der R- und L-Mengen der Regeln

Wie in Abschnitt 3.1 gezeigt, arbeitet der bestehende Suchalgorithmus mit binären Regeln. Sei $A \rightarrow BC$ eine solche Regel, wobei A, B und C Terminale oder Nichtterminale sind. Die Expansion von A besteht damit aus zwei Teilen, nämlich der Expansion von B und der Expansion von C. Für das Muster gibt es deshalb drei Möglichkeiten: Es kommt vollständig in der Expansion von B vor, vollständig in der Expansion von C, oder es beginnt in der Expansion von B und endet in der Expansion von C.
Im dritten Fall bildet ein Teil in der Expansion von B das Präfix des Musters. Die Positionen solcher Präfixe werden in einer sogenannten **R-Menge** gespeichert. Dann bildet ein Teil in der Expansion von C das Suffix des Musters, dessen Positionen in einer sogenannten **L-Menge** gespeichert werden. Im Folgenden werden beide Mengen genauer erläutert.

R-Mengen

Die R-Menge enthält die echten Präfixe des Musters, die am Ende einer Expansion liegen. Die Positionen laufen dabei von 1 bis $m-1$, wobei m die Länge des Musters ist. Nur diese Positionen werden betrachtet, da ein Präfix den linken Teil des Musters bildet und noch vervollständigt werden muss.

Ein solches Präfix kann auf drei Arten an der Expansion liegen. Wenn es kürzer als die Expansion ist, dann bildet es ein Suffix der Expansion. Wenn es genauso lang ist, dann deckt es die gesamte Expansion ab. Wenn es länger ist, dann geht es nach links über die Expansion hinaus.

Das *Präfix* bzw. die *R-Menge* entspricht dem Teil $p[1 \dots j]$, wobei j angibt, bis zu welcher Position das Muster mit dem Ende der Expansion übereinstimmt. Der übrige Teil $p[j+1 \dots m]$ muss rechts davon gefunden werden, um das Muster zu vervollständigen. Aus diesem Grund wird diese Menge als *R-Menge* bezeichnet.

Definition: *(R-Menge)* Sei p ein Muster der Länge m und v_i die Expansion eines Symbols der Länge n_i. Die R-Menge ist definiert als [9]:

$$R_i := \{\, j \in \{1, \dots, m-1\} \mid p[\max(1,\, j-n_i+1) \dots j] = v_i[\max(n_i-j+1,\, 1) \dots n_i] \,\}.$$

Die Position j liegt zwischen 1 und $m-1$. Durch die Maximum-Funktionen wird nur der gemeinsame Bereich von Präfix und Expansion verglichen. Dabei gibt es zwei Fälle:

- Ist $j \leq n_i$, dann ist das Präfix höchstens so lang wie die Expansion. Es wird dann $p[1 \dots j]$ mit dem Suffix $v_i[n_i - j + 1 \dots n_i]$ verglichen.
- Ist $j > n_i$, dann ist das Präfix länger als die Expansion und ragt nach links über sie hinaus. Der linke Teil des Präfixes wird deshalb weggelassen und nur $p[j - n_i + 1 \dots j]$ mit $v_i[1 \dots n_i]$ verglichen.

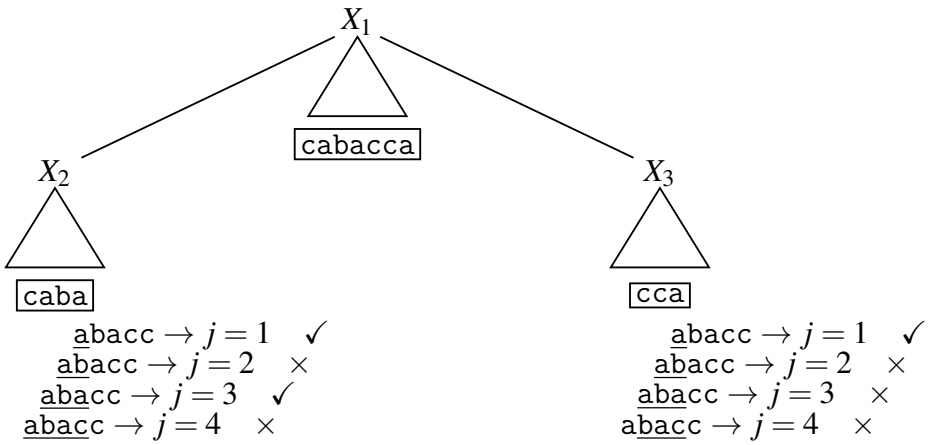

Abbildung 2: Bestimmung der R-Mengen von X_2 und X_3 für das Muster $p = \texttt{abacc}$. Für jedes j wird das Präfix $p[1 \dots j]$ mit dem Ende der jeweiligen Expansion verglichen. Der verglichene Teil des Musters ist unterstrichen. Ein Haken steht für eine Übereinstimmung, ein Kreuz für eine fehlende Übereinstimmung.

Beispiel Für das Muster $p = \texttt{abacc}$ der Länge $m = 5$ werden die ersten vier Positionen $j \in \{1,2,3,4\}$ untersucht. Abbildung 2 zeigt diese Prüfung für X_2 und X_3. Bei X_2 mit der Expansion `caba` passen zwei Positionen. Bei $j = 1$ endet die Expansion auf `a` und stimmt damit mit $p[1 \dots 1]$ überein. Bei $j = 3$ bilden die letzten drei Zeichen der Expansion den Text `aba`, was genau $p[1 \dots 3]$ entspricht. Die Positionen 2 und 4 stimmen mit der Expansion nicht überein, da `ba` $\neq$ `ab` und `caba` $\neq$ `abac`. Es ergibt sich $R_{X_2} = \{1,3\}$. Bei X_3 mit der Expansion `cca` passt nur $j = 1$, da die Expansion mit `a` endet. Die Positionen 2 und 3 passen nicht, da `ca` $\neq$ `ab` und `cca` $\neq$ `aba`. Bei $j = 4$ ist das Präfix `abac` um ein Zeichen länger als die Expansion, deshalb wird nach der Definition wegen der Max-Funktion nur $p[2 \dots 4] = \texttt{bac}$ mit `cca` verglichen. Auch hier stimmen die Zeichen nicht überein. Es ergibt sich $R_{X_3} = \{1\}$.

L-Mengen

Die L-Menge enthält die echten Suffixe des Musters, die am Anfang einer Expansion liegen. Die Positionen laufen dabei von 2 bis m. Nur diese Positionen werden betrachtet, da ein Suffix den rechten Teil des Musters bildet und links davon noch vervollständigt werden muss.

Das *Suffix* bzw. die *L-Menge* entspricht dem Teil $p[j \dots m]$, wobei j angibt, ab welcher Position das Muster mit dem Anfang der Expansion übereinstimmt. Der übrige Teil $p[1 \dots j-1]$ muss links davon gefunden werden, um das Muster zu vervollständigen. Aus diesem Grund wird diese Menge als *L-Menge* bezeichnet.

Definition: *(L-Menge)* Sei p ein Muster der Länge m und v_i die Expansion eines Symbols der Länge n_i. Die L-Menge ist definiert als [9]:

$$L_i := \{\, j \in \{2, \dots, m\} \mid p[j \dots \min(m,\, j+n_i-1)] = v_i[1 \dots \min(n_i,\, m-j+1)] \,\}.$$

Die Position j liegt zwischen 2 und m. Durch die Minimum-Funktionen wird nur der gemeinsame Bereich von Suffix und Expansion verglichen. Dabei gibt es zwei Fälle:

- Ist $m-j+1 \leq n_i$, dann ist das Suffix höchstens so lang wie die Expansion. Es wird dann $p[j \dots m]$ mit dem Präfix $v_i[1 \dots m-j+1]$ verglichen.
- Ist $m-j+1 > n_i$, dann ist das Suffix länger als die Expansion und ragt nach rechts über sie hinaus. Der rechte Teil des Suffixes wird deshalb weggelassen und nur $p[j \dots j+n_i-1]$ mit $v_i[1 \dots n_i]$ verglichen.

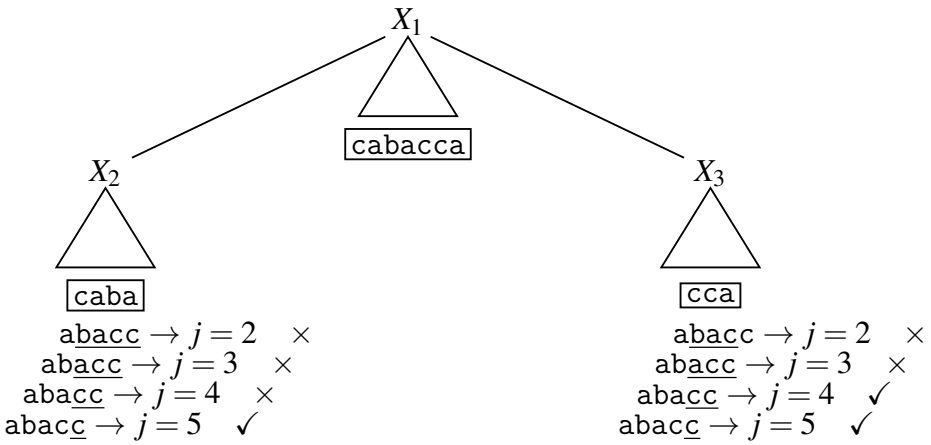

Abbildung 3: Bestimmung der L-Mengen von X_2 und X_3 für das Muster $p = \texttt{abacc}$. Für jedes j wird das Suffix $p[j \ldots m]$ mit dem Anfang der jeweiligen Expansion verglichen. Der verglichene Teil des Musters ist unterstrichen. Ein Haken steht für eine Übereinstimmung, ein Kreuz für eine fehlende Übereinstimmung.

Beispiel Für das Muster $p = \texttt{abacc}$ werden nun die letzten vier Positionen $j \in \{2,3,4,5\}$ untersucht. Abbildung 3 zeigt diese Prüfung für X_2 und X_3. Bei X_2 mit der Expansion `caba` passt nur $j = 5$, da das Suffix $p[5 \ldots 5] = \texttt{c}$ mit dem ersten Zeichen der Expansion übereinstimmt. Die übrigen Positionen passen nicht, da $\texttt{bacc} \neq \texttt{caba}$, $\texttt{acc} \neq \texttt{cab}$ und $\texttt{cc} \neq \texttt{ca}$. Es ergibt sich $L_{X_2} = \{5\}$. Bei X_3 mit der Expansion `cca` passen zwei Positionen. Bei $j = 4$ stimmt das Suffix $p[4 \ldots 5] = \texttt{cc}$ mit den ersten beiden Zeichen der Expansion überein und bei $j = 5$ stimmt $p[5 \ldots 5] = \texttt{c}$ mit dem ersten Zeichen überein. Bei $j = 3$ ist das Suffix `acc` genauso lang wie die Expansion, stimmt aber nicht mit `cca` überein. Bei $j = 2$ ist das Suffix `bacc` um ein Zeichen länger als die Expansion, deshalb wird wegen der Min-Funktion nur $p[2 \ldots 4] = \texttt{bac}$ mit `cca` verglichen. Auch hier stimmen die Zeichen nicht überein. Es ergibt sich $L_{X_3} = \{4,5\}$.

3.3 Ablauf der Suche

Die in Abschnitt 3.2 beschriebenen R- und L-Mengen bilden die Grundlage des Suchalgorithmus. Für jedes Symbol der Grammatik werden seine Informationen berechnet und gespeichert. Diese sind die R- und L-Mengen, die Länge der Expansion und die Anzahl der Vorkommen des Musters. Die Länge wird dabei benötigt, weil bei jedem Vergleich bekannt sein muss, wie viele Zeichen die Expansion enthält. Die Informationen einer Regel hängen von den Informationen ihrer Kinder ab. Der Algorithmus geht die Regeln deshalb von unten nach oben durch.

In diesem Abschnitt wird zuerst die Berechnung der Informationen für Terminale und Nichtterminale beschrieben. Danach wird gezeigt, wie mit diesen Informationen ein vollständiges Vorkommen des Musters erkannt und gezählt wird.

Berechnung der Informationen der Terminale

Ein Terminal besteht aus einem einzigen Zeichen und hat deshalb immer die Länge 1. Die R- und L-Mengen werden für jedes Terminal berechnet und als boolesche Vektoren der Länge m gespeichert. Wie bereits in Abschnitt 3.2 besprochen, hat die R-Menge den Bereich von 1 bis $m-1$ und die L-Menge den Bereich von 2 bis m.

Für die Berechnung wird das Terminal mit allen Zeichen des Musters schrittweise verglichen. Für die R-Menge wird das Terminal mit den Zeichen des Musters von der ersten Position bis zur Position $m-1$ verglichen. Wenn das Zeichen mit dem Terminal übereinstimmt, wird an dieser Position der Wert `true` gespeichert. Wenn es nicht übereinstimmt oder an der Position m liegt, dann wird `false` gespeichert. Für die L-Menge wird das Terminal auch hier mit den Zeichen des Musters verglichen, allerdings von der Position 2 bis zur Position m. Wenn das Zeichen mit dem Terminal übereinstimmt, wird an dieser Position der Wert `true` gespeichert. Wenn es nicht übereinstimmt oder an der Position 1 liegt, dann wird `false` gespeichert.

Beispiel: Sei $p = abcac$ mit $m = 5$. Das Muster besteht aus den drei Terminalen a, b und c. Die Sprache der Grammatik hat aber vier Terminale, nämlich a, b, c und d.

Muster	a	b	c	a	c
Position	1	2	3	4	5
R(a)	true	false	false	true	false
L(a)	false	false	false	true	false
R(b)	false	true	false	false	false
L(b)	false	true	false	false	false
R(c)	false	false	true	false	false
L(c)	false	false	true	false	true
R(d)	false	false	false	false	false
L(d)	false	false	false	false	false

Tabelle 1: Boolesche Vektoren der R- und L-Mengen aller Terminale für das Muster $p =$ `abcac`. Ein Eintrag ist `true`, wenn das Zeichen des Musters an dieser Position mit dem Terminal übereinstimmt und die Position im Bereich der jeweiligen Menge liegt.

Tabelle 1 zeigt die Berechnung der Vektoren für die Terminale. Für das Terminal `a` wird bei der R-Menge von der Position 1 bis zur Position 4 verglichen. Das Muster hat an den Positionen 1 und 4 das Zeichen `a`, deshalb erhalten diese beiden Positionen den Wert `true`. Die Positionen 2 und 3 erhalten den Wert `false`, da dort die Zeichen `b` und `c` stehen. Die Position 5 erhält ebenfalls `false`, da sie außerhalb des Bereichs der R-Menge liegt. Bei der L-Menge wird von der Position 2 bis zur Position 5 verglichen. Hier erhält nur die Position 4 den Wert `true`. Die Position 1 erhält `false`, obwohl das Muster dort ebenfalls das Zeichen `a` hat, weil diese Position außerhalb des Bereichs der L-Menge liegt. Für das Terminal `b` hat das Muster nur an der Position 2 eine Übereinstimmung. Diese Position liegt in beiden Bereichen, deshalb erhält `b` sowohl bei der R-Menge als auch bei der L-Menge den Wert `true` an Position 2. Alle anderen Positionen erhalten den Wert `false`. Für das Terminal `c` hat das Muster an den Positionen 3 und 5 eine Übereinstimmung. Bei der R-Menge von `c` erhält nur die Position 3 den Wert `true`, da die Position 5 außerhalb des Bereichs liegt. Bei der L-Menge von `c` liegen beide Positionen im Bereich, deshalb erhalten die Positionen 3 und 5 den Wert `true`. Das Terminal `d` kommt im Muster nicht vor, deshalb hat es an allen Positionen den Wert `false`.

Berechnung der Informationen der Nichtterminale

Die Expansion eines Nichtterminals X besteht aus mehreren Zeichen, deshalb werden seine Informationen aus den Informationen seiner Kinder berechnet. Sei $X \rightarrow YZ$ eine Regel, dann müssen die Infor-

mationen von Y und Z berechnet werden, damit man X berechnen kann. Aus diesem Grund geht der Suchalgorithmus die Regeln von unten nach oben durch. Die Länge der Expansion von X ergibt sich aus der Summe der Länge von Y und Z, es gilt $n_X = n_Y + n_Z$. Für die Berechnung der R- und L-Mengen eines Nichtterminals gibt es jeweils zwei Fälle:

R-Menge der Nichtterminale

Sei $A \to BC$ eine Regel. Die R-Menge von A enthält die Positionen, an denen ein Präfix des Musters am Ende der Expansion von A endet. Ein solches Präfix kann auf zwei Arten entstehen [9]:

- **Fall 1: Das Präfix liegt vollständig in C.** Wenn eine Position j in $R(C)$ steht, dann endet ein Präfix der Länge j am Ende der Expansion von C. Die Expansion von A endet ebenfalls mit C, deshalb endet dieses Präfix auch am Ende von A. Zusätzlich muss $j \leq n_C$ gelten. Damit wird sichergestellt, dass das Präfix wirklich in C liegt und nicht nach links über C hinausgeht. Wenn beide Bedingungen erfüllt sind, wird j in $R(A)$ übernommen.

- **Fall 2: Das Präfix überspannt die Grenze zwischen B und C.** In diesem Fall beginnt das Präfix in der Expansion von B und endet am Ende der Expansion von C. Wenn eine Position k in $R(B)$ steht und die Position $k+1$ in $L(C)$, dann liegt der Teil $p[1 \ldots k]$ am Ende von B und der Teil $p[k+1 \ldots k+n_C]$ liegt in der Expansion von C. Beide Teile bilden zusammen das Präfix. Das Präfix besteht damit aus k Zeichen aus B und der Expansion von C. Die Länge des Präfixes ist deshalb $j = k + n_C$. Die Länge ist immer größer als n_C, weil k mindestens 1 ist. Zusätzlich müssen $k \leq n_B$ und $j < m$ gelten. Damit wird sichergestellt, dass es sich noch um ein Präfix handelt und nicht um das vollständige Muster. Wenn beide Bedingungen erfüllt sind, wird j in $R(A)$ übernommen.

Beispiel: Sei $p =$ `abcac` mit $m = 5$ und seien $A \to B$`a`, $B \to C$`c` und $C \to$ `ab` drei Regeln, die Präfixe des Musters erzeugen. Die Expansion von B ist `abc` mit $n_B = 3$ und die Expansion von A ist `abca`. Das rechte Kind von A ist das Terminal `a` mit der Länge 1.

Muster	`a`	`b`	`c`	`a`	`c`
Position	1	2	3	4	5
$R($`a`$)$	true	false	false	true	false
$R(B)$	false	false	true	false	false
$L($`a`$)$	false	false	false	true	false
$R(A)$	true	false	false	true	false

Tabelle 2: Berechnung der R-Menge für die Regel $A \to B$`a`. Die ersten drei Zeilen sind Mengen der beiden Kinder, die vierte Zeile ist das Ergebnis.

Tabelle 2 zeigt die Berechnung der R-Menge von A. Für den ersten Fall wird $R($`a`$)$ betrachtet, also die R-Menge des rechten Terminals. Dieses Terminal hat die Länge $n_a = 1$. Dort haben die Positionen 1 und 4 den Wert `true`. Bei der Position 1 gilt $j \leq n_a$, also $1 \leq 1$, deshalb wird diese Position in $R(A)$ übernommen. Das Präfix `a` liegt vollständig im Terminal `a` und endet damit auch am Ende von A. Bei der Position 4 gilt $4 \leq 1$ nicht, deshalb wird diese Position in diesem Fall nicht übernommen. Das Präfix `abca` ist zu lang für das Terminal `a`. Die Positionen 2, 3 und 5 werden auch nicht übernommen, weil sie den Wert `false` haben. Für den zweiten Fall wird mit $R(B)$ und $L($`a`$)$ geprüft, ob das Muster die Grenze zwischen B und `a` überspannt. Die Position 3 in $R(B)$ hat den Wert `true`. Die Position $k+1 = 3+1 = 4$

steht in $L(\texttt{a})$ und hat auch den Wert `true`, deshalb passen beide Teile zusammen. Der Teil `abc` ist die Expansion von B und der Teil `a` ist das rechte Terminal von A. Zusammen bilden sie das Präfix `abca` mit der Länge $j = k + n_a = 3 + 1 = 4$. Weil $4 < 5$ ist, wird diese Position in $R(A)$ übernommen. Damit ergibt sich $R(A) = \{\texttt{true, false, false, true, false}\}$.

L-Menge der Nichtterminale

Sei $D \to EF$ eine Regel. Die L-Menge von D enthält die Positionen, an denen ein Suffix des Musters am Anfang der Expansion von D beginnt. Ein solches Suffix kann auf zwei Arten entstehen [9]:

- **Fall 1: Das Suffix liegt vollständig in E.** Wenn eine Position j in $L(E)$ steht, dann beginnt ein Suffix der Länge $m - j + 1$ am Anfang der Expansion von E. Da die Expansion von D ebenfalls mit E anfängt, beginnt dieses Suffix auch am Anfang von D. Zusätzlich muss $m - j + 1 \leq n_E$ gelten. Damit wird sichergestellt, dass das Suffix wirklich in E liegt und nicht nach rechts über E hinausgeht. Wenn beide Bedingungen erfüllt sind, wird j in $L(D)$ übernommen.

- **Fall 2: Das Suffix überspannt die Grenze zwischen E und F.** In diesem Fall beginnt das Suffix am Anfang der Expansion von E und endet in der Expansion von F. Wenn eine Position j in $L(E)$ steht und die Position $j + n_E$ in $L(F)$, dann liegt der Teil $p[j \ldots j + n_E - 1]$ in E und der Teil $p[j + n_E \ldots m]$ beginnt am Anfang von F. Beide Teile bilden zusammen das Suffix. Das Suffix besteht damit aus der Expansion von E und den restlichen Zeichen aus F. Die Länge des Suffixes ist immer größer als n_E, weil sich das Suffix in F fortsetzt. Wenn beide Bedingungen erfüllt sind, wird j in $L(D)$ übernommen.

Beispiel: Sei $p = \texttt{cacba}$ mit $m = 5$ und seien $D \to \texttt{a}G$, $G \to \texttt{c}H$ und $H \to \texttt{ba}$ drei Regeln, die Suffixe des Musters erzeugen. Die Expansion von G ist `cba` mit $n_G = 3$ und die Expansion von D ist `acba`. Das linke Kind von D ist das Terminal `a` mit der Länge $n_a = 1$.

Muster	`c`	`a`	`c`	`b`	`a`
Position	1	2	3	4	5
$L(\texttt{a})$	false	true	false	false	true
$L(G)$	false	false	true	false	false
$L(D)$	false	true	false	false	true

Tabelle 3: Berechnung der L-Menge für die Regel $D \to \texttt{a}G$. Die ersten zwei Zeilen sind Mengen der beiden Kinder, die dritte Zeile ist das Ergebnis.

Tabelle 3 zeigt die Berechnung der L-Menge von D. Für den ersten Fall wird $L(\texttt{a})$ betrachtet, also die L-Menge des linken Terminals. Dieses Terminal hat die Länge $n_a = 1$. Dort haben die Positionen 2 und 5 den Wert `true`. Bei der Position 5 gilt $5 - 5 + 1 \leq n_a$, also $1 \leq 1$, deshalb wird diese Position in $L(D)$ übernommen. Das Suffix `a` liegt vollständig im Terminal `a` und beginnt damit auch am Anfang von D. Bei der Position 2 gilt $5 - 2 + 1 \leq n_a$, also $4 \leq 1$ nicht, deshalb wird diese Position in diesem Fall nicht übernommen. Das Suffix `acba` ist zu lang für das Terminal `a`. Die Positionen 1, 3 und 4 von $L(\texttt{a})$ werden auch nicht übernommen, weil sie den Wert `false` haben. Für den zweiten Fall wird mit $L(\texttt{a})$ und $L(G)$ geprüft, ob das Muster die Grenze zwischen `a` und G überspannt. Die Position 2 in $L(\texttt{a})$ hat den Wert `true`. Die Position $j + n_a = 2 + 1 = 3$ steht in $L(G)$ und hat auch den Wert `true`, deshalb passen beide Teile zusammen. Der Teil `a` ist das linke Terminal von D und der Teil `cba` ist die Expansion von G. Zusammen

bilden sie das Suffix `acba`, das an der Position 2 beginnt. Bei der Position 5 gilt dies nicht, weil $5+1=6$ außerhalb des Musters liegt. Damit ergibt sich $L(D) = \{\texttt{false, true, false, false, true}\}$.

Suche und Zählung des Musters

Nachdem die R- und L-Mengen einer Regel berechnet wurden, wird geprüft, ob das Muster in dieser Regel vollständig vorkommt. Diese Prüfung wird auch *Match-Check* genannt [9]. Sei $A \to BC$ eine Regel. Ein Match liegt genau dann vor, wenn es ein $k \in R(B)$ gibt und $(k+1) \in L(C)$ steht. Zusätzlich muss auch $k \leq n_B$ und $m-k \leq n_C$ gelten. Die beiden Längenbedingungen stellen sicher, dass dieses Muster auch wirklich in der Expansion von A liegt [9].

Für den *Match-Check* wird also jede Position k aus $R(B)$ betrachtet. Wenn die Position $k+1$ in $L(C)$ steht, dann endet das Präfix $p[1\ldots k]$ am Ende von B und das Suffix $p[k+1\ldots m]$ beginnt am Anfang von C. Zusammen bilden beide Teile das vollständige Muster und ein Vorkommen wird gezählt.

Ein Muster, das länger als ein Zeichen ist, wird dank der binären Regelstruktur immer auf zwei Teile aufgeteilt, nämlich auf ein Präfix und ein Suffix. Für jede Regel werden wie gezeigt alle möglichen Präfixe und Suffixe des Musters gespeichert. Eine Regel kann deshalb bei einem Vorkommen des Musters ein Präfix bilden und bei einem anderen Vorkommen ein Suffix. Aus diesem Grund kann der Suchalgorithmus auch überlappende Vorkommen finden. Für den Text `aaaaa` und das Muster `aaa` gibt es z. B. drei überlappende Vorkommen.

Die Anzahl der Vorkommen des Musters in der Expansion eines Symbols X wird mit $occ(X)$ bezeichnet. Für eine Regel $A \to BC$ setzt sich diese Anzahl aus drei Teilen zusammen:

$$occ(A) = occ(B) + occ(C) + occ(BC)$$

Die Werte $occ(B)$ und $occ(C)$ sind die Vorkommen, die bereits in den beiden Kindern gefunden wurden. Der Wert $occ(BC)$ ist die Anzahl der Vorkommen aus dem *Match-Check*, also die Vorkommen an der Grenze zwischen B und C. Jedes Vorkommen wird auf diese Weise genau einmal gezählt, weil es genau eine Regel gibt, an der es zum ersten Mal vollständig entsteht.

Am Ende wird die Startregel verarbeitet. Wie in Abschnitt 3.1 beschrieben, hat die Startregel keine binäre Struktur wie die Nichtterminale und kann beliebig viele Symbole auf der rechten Seite haben. Sei $S \to Y_1 \ldots Y_k$ die Startregel, dann werden die Informationen ihrer Symbole von links nach rechts kombiniert. Zuerst wird Y_1 mit Y_2 kombiniert, danach wird das Zwischenergebnis mit Y_3 kombiniert und so weiter, bis alle Symbole verarbeitet sind. Jede dieser Kombinationen wird genauso wie bei den Nichtterminalen behandelt. Für jede Kombination werden also die R- und L-Mengen berechnet und ein Match-Check durchgeführt. Die dabei gefundenen Vorkommen werden zu den Vorkommen der Symbole addiert. Der Wert $occ(S)$ ist damit die Anzahl aller Vorkommen des Musters im ursprünglichen Text und ist das Ergebnis der Suche [9].

4 Bitparallele Mustersuche

In diesem Abschnitt wird die bitparallele Mustersuche beschrieben, die die Grundlage für die Erweiterung des bestehenden Suchalgorithmus bildet. Zunächst wird in Abschnitt 4.1 der Shift-And-Algorithmus vorgestellt [2]. Dieser Algorithmus bildet die Grundlage für die in dieser Arbeit entwickelte bitparallele Verarbeitung des bestehenden Suchalgorithmus. Anschließend wird in Abschnitt 4.2 beschrieben, wie die R- und L-Mengen des bestehenden Suchalgorithmus bitparallel dargestellt werden. Dabei wird zwischen Mustern unterschieden, die in ein einzelnes 64-Bit-Wort passen, und längeren Mustern, für die mehrere Bitwörter benötigt werden. Abschnitt 4.3 zeigt danach, wie die Regelinformationen und der

Match-Check mithilfe von Bitoperationen berechnet werden. In Abschnitt 4.4 wird gezeigt, wie für lange Muster Bitvektoren verwendet werden. Abschließend wird in Abschnitt 4.5 beschrieben, wie der Suchalgorithmus an das neue Dateiformat angepasst wurde. In diesem Abschnitt wird außerdem die erste Position eines Bitworts mit dem Index 1 bezeichnet und liegt in der Darstellung ganz links.

4.1 Shift-And-Algorithmus

Der *Shift-And-Algorithmus* wurde von Baeza-Yates und Gonnet entwickelt. Er ist auch unter den Namen *Bitap* bekannt [2]. Der Algorithmus sucht ein Muster in einem Text, wobei beide nicht komprimiert sind. Dafür wird für jedes Zeichen einen Bitvektor erstellt. Der Text wird danach Zeichen für Zeichen gelesen und mit diesen Bitmasken verglichen. Da eine Bitoperation alle Positionen eines Maschinenwortes gleichzeitig verarbeitet, ist der Algorithmus deutlich effizienter als ein Vergleich mit booleschen Werten [11].

Der charakteristische Vektor s_x

Vor der Suche wird für jedes Zeichen des Alphabets ein sogenannter charakteristischer Vektor s_x erstellt, wobei x ein Zeichen aus dem Alphabet ist. Dieser Vektor hat die Länge des Musters. Der Vektor eines Zeichens hat den Wert 1 an jeder Position, die diesem Zeichen im Muster entspricht. Alle anderen Positionen haben den Wert 0 [2].

Beispiel: Sei $p = \texttt{abcac}$ mit $m = 5$. Dann sind die charakteristischen Vektoren wie folgt aufgebaut:

Muster	$s_{\texttt{a}}$	$s_{\texttt{b}}$	$s_{\texttt{c}}$	$s_{\texttt{d}}$	...
a	1	0	0	0	...
b	0	1	0	0	...
c	0	0	1	0	...
a	1	0	0	0	...
c	0	0	1	0	...

Tabelle 4: Vektoren der Zeichen für das Muster $p = \texttt{abcac}$. Ein Eintrag ist 1, wenn das Muster an dieser Position das jeweilige Zeichen hat.

In Tabelle 4 wurde die Initialisierung der charakteristischen Vektoren von `a`, `b`, `c` und `d` gezeigt. Für das Zeichen `a` wird der Vektor $s_{\texttt{a}}$ erstellt. Dafür wird das Muster von der Position 1 bis zur Position 5 durchgegangen. An jeder Position wird nur geprüft, ob dort das Zeichen `a` steht. An der Position 1 steht das Zeichen `a`, deshalb erhält diese Position den Wert 1. An den Positionen 2 und 3 stehen andere Zeichen, deshalb erhalten beide Positionen den Wert 0. An der Position 4 steht wieder das Zeichen `a`, deshalb erhält diese Position den Wert 1. An der Position 5 steht ein anderes Zeichen, deshalb hat sie den Wert 0. Damit ergibt sich $s_{\texttt{a}} = \texttt{10010}$. Für das Zeichen `b` wird das Muster genauso durchgegangen. Nur an der Position 2 steht das Zeichen `b`, deshalb erhält nur diese Position den Wert 1. Alle anderen Positionen erhalten den Wert 0 und es ergibt sich $s_{\texttt{b}} = \texttt{01000}$. Für das Zeichen `c` erhalten die Positionen 3 und 5 den Wert 1. Die Positionen 1, 2 und 4 erhalten den Wert 0. Es ergibt sich $s_{\texttt{c}} = \texttt{00101}$. Das Zeichen `d` kommt im Muster an keiner Position vor. Der Vektor $s_{\texttt{d}}$ hat deshalb an allen Positionen den Wert 0. Für alle weiteren Vektoren, deren Zeichen im Muster nicht vorkommen, gilt dasselbe.

Der Zustandsvektor Z_j

Für den Vergleich mit dem Text wird ein weiterer Bitvektor Z_j verwendet. Dieser Vektor speichert nach jedem Vergleich den Stand der Übereinstimmungen. Die Position i zeigt dabei auf die Stelle des Musters, die gerade geprüft wird, deshalb hat der Vektor die gleiche Länge wie das Muster. Der Index j gibt den Schritt an, in dem sich der Algorithmus befindet. Die Anzahl der Schritte hängt nicht von der Länge des Musters ab, sondern davon, wie viele Zeichen des Textes gelesen werden müssen, bis ein Match gefunden wird. Wenn das Muster nicht gefunden werden kann oder ganz am Ende des Textes steht, dann ist die Anzahl der benötigten Schritte gleich der Länge des Textes. Für den Wert von $Z_{j+1}[i]$ gibt es drei Fälle [2]:

- **Fall 1:** An der Position $i = 1$ kann eine neue Übereinstimmung beginnen. Wenn das aktuelle Textzeichen mit dem ersten Zeichen des Musters übereinstimmt, also $p[1] = t[j+1]$ gilt, dann erhält die Position 1 den Wert 1.
- **Fall 2:** Wenn an der vorherigen Position eine Übereinstimmung zwischen dem Muster und dem Text vorliegt, dann hat der Vektor dort den Wert 1, es gilt also $Z_j[i-1] = 1$. Wenn auch an der aktuellen Position eine Übereinstimmung vorliegt, also $p[i] = t[j+1]$ gilt, dann erhält die aktuelle Position ebenfalls den Wert 1. Formal gilt also:

  $$\text{Wenn } Z_j[i-1] = 1 \text{ und } p[i] = t[j+1], \text{ dann } Z_{j+1}[i] = 1$$

- **Fall 3:** Wenn weder Fall 1 noch Fall 2 gelten, dann hat $Z_{j+1}[i]$ den Wert 0.

Ein Vorkommen des Musters wird gefunden, wenn $Z_j[m]$ den Wert 1 hat. Die Position m ist das letzte Bit des Vektors und entspricht auch der Länge des Musters. Wenn dieses Bit den Wert 1 hat, dann ist das Muster gefunden.

Ablauf des Shift-And-Algorithmus

Zu Beginn der Suche werden alle Positionen des Vektors Z mit 0 initialisiert. Danach wird der Text Zeichen für Zeichen gelesen. Für jedes gelesene Zeichen führt der Algorithmus zwei Operationen aus.

Zuerst werden alle Bits des Vektors um eine Position nach rechts verschoben. Diese Operation wird Bitshift genannt und mit $\gg$ bezeichnet. Dabei wird der Wert von der Position i auf die Position $i+1$ verschoben und die Position 1 erhält den Wert 1. Ein Vektor mit dem Wert `00000` wird durch den Bitshift also zu `10000`. Durch diese Verschiebung wird jede bisherige Übereinstimmung um eine Position verlängert, damit das nächste Zeichen verglichen werden kann. Die 1 an der Position 1, die nach jeder Verschiebung entsteht, erlaubt außerdem, dass in diesem Schritt eine neue Übereinstimmung beginnen kann.

Nach der Verschiebung des Vektors Z wird er mit dem charakteristischen Vektor des aktuellen Textzeichens durch eine UND-Verknüpfung verbunden. Eine Position behält dabei nur dann den Wert 1, wenn beide Vektoren dort den Wert 1 haben. Der Vektor Z gibt an, bis zu welcher Position die Zeichen übereinstimmen. Der charakteristische Vektor gibt an, an welchen Positionen das aktuelle Zeichen im Muster steht.

Das Ergebnis dieser Verschiebung und der UND-Verknüpfung ist der neue Vektor Z_{j+1}. Wenn dieser Vektor an der Position m den Wert 1 hat, dann wurde das Muster gefunden. Wenn nicht, dann werden beide Operationen wiederholt, bis alle Zeichen des Textes geprüft wurden oder das Muster gefunden wird.

Beispiel: Sei $p =$ `abc` mit $m = 3$ und sei der Text $t =$ `babc`.

i	Z_0	$\gg$	$s_{\texttt{b}}$	Z_1	$\gg$	$s_{\texttt{a}}$	Z_2	$\gg$	$s_{\texttt{b}}$	Z_3	$\gg$	$s_{\texttt{c}}$	Z_4
1	0	1	0	0	1	1	1	1	0	0	1	0	0
2	0	0	1	0	0	0	0	1	1	1	0	0	0
3	0	0	0	0	0	0	0	0	0	0	1	1	1

Tabelle 5: Ablauf des Shift-And-Algorithmus für das Muster $p = \texttt{abc}$ und den Text $t = \texttt{babc}$. Für jedes Textzeichen zeigt die erste Spalte den Vektor nach der Verschiebung, die zweite Spalte den charakteristischen Vektor des Zeichens und die dritte Spalte das Ergebnis der UND-Verknüpfung.

Das Beispiel in der Tabelle 5 zeigt den Ablauf für den Text `babc`. Der Vektor Z_0 hat den Wert `000` und wird durch die Verschiebung zu `100`. Das erste Textzeichen ist `b`, deshalb wird der Vektor $s_{\texttt{b}} = \texttt{010}$ für die UND-Verknüpfung verwendet. Diese ergibt $\texttt{100} \wedge \texttt{010} = \texttt{000}$, weil die beiden Vektoren an keiner Position gemeinsam den Wert 1 haben. Der Vektor Z_1 ist deshalb `000`. Beim zweiten Textzeichen `a` wird Z_1 zu `100` verschoben und mit $s_{\texttt{a}} = \texttt{100}$ verknüpft. An der Position 1 haben beide Vektoren den Wert 1, sodass $Z_2 = \texttt{100}$ ist. Damit stimmt das Muster mit dem Text am ersten Zeichen überein. Das dritte Textzeichen ist `b`. Die Verschiebung von Z_2 ergibt `110` und die Verknüpfung mit $s_{\texttt{b}}$ ergibt $Z_3 = \texttt{010}$. Damit stimmen zwei Zeichen des Musters mit dem Text überein. Am Ende kommt das Textzeichen `c`. Die Verschiebung von Z_3 ergibt `101` und die Verknüpfung mit $s_{\texttt{c}} = \texttt{001}$ ergibt $Z_4 = \texttt{001}$. Der Vektor Z_4 hat an der Position $m = 3$ den Wert 1, deshalb wurde das Muster gefunden. Das Muster beginnt an der Position $j - m + 1 = 4 - 3 + 1 = 2$ des Textes.

4.2 Bitparallele Darstellung der R- und L-Mengen

Der bestehende Suchalgorithmus speichert die R- und L-Mengen jeder Regel in einem booleschen Vektor, wie in Abschnitt 3.3 beschrieben. Diese Darstellung kostet viel Speicher, weil für jeden einzelnen booleschen Wert ein ganzes Byte reserviert wird. Vor allem aber wird jede Position des Musters einzeln geprüft, sodass der Aufwand für eine Regel mit der Länge des Musters wächst.

In diesem Abschnitt wird gezeigt, wie die Werte der R- und L-Mengen in Bits statt in booleschen Werten gespeichert werden, damit sie schneller verarbeitet werden können. Die Darstellung überträgt dabei das Prinzip des Shift-And Algorithmus [2] auf die R- und L-Mengen des bestehenden Suchalgorithmus. Jede Position des Musters entspricht genau einem Bit, sodass alle Positionen gemeinsam verarbeitet werden können. Die Bitwörter werden hier von links nach rechts geschrieben, sodass die Position 1 des Musters dem ersten Bit von links entspricht. Jede Position der Bits liegt damit an der gleichen Stelle wie in der booleschen Darstellung.

Für diese Darstellung gibt es zwei Fälle, die von der Länge des Musters abhängen. Ein Rechner kann in der Regel 64 Bit in einer einzigen Operation verarbeiten [11]. Aus diesem Grund werden zwei verschiedene Varianten benötigt, die sich im Aufbau und in der Speicherung der Daten unterscheiden, aber nach demselben Prinzip arbeiten.

Darstellung für Muster bis 64 Zeichen

Wenn das Muster höchstens 64 Zeichen hat, passen die R- und L-Mengen jeweils in ein einziges Bitwort. Jeder Wert der Mengen wird dabei durch genau ein Bit dargestellt. Sei X ein Symbol der Grammatik, R_X und L_X sind seine R- bzw. L-Mengen. Für jede Position j des Musters gilt dann:

$$R(X)[j] = 1, \text{ wenn } j \in R_X, \text{ sonst } R(X)[j] = 0$$

$$L(X)[j] = 1, \text{ wenn } j \in L_X, \text{ sonst } L(X)[j] = 0$$

Jeder Wert `true` aus der booleschen Darstellung wird also zu einer 1 und jeder Wert `false` zu einer 0.

Beispiel: Sei $p =$ `abcac` mit $m = 5$. Für die Terminale `a`, `b` und `c` ergeben sich die folgenden Darstellungen:

Muster		`a`	`b`	`c`	`a`	`c`
Position		1	2	3	4	5
R(`a`)	boolesch	true	false	false	true	false
	bitparallel	1	0	0	1	0
L(`a`)	boolesch	false	false	false	true	false
	bitparallel	0	0	0	1	0
R(`b`)	boolesch	false	true	false	false	false
	bitparallel	0	1	0	0	0
L(`b`)	boolesch	false	true	false	false	false
	bitparallel	0	1	0	0	0
R(`c`)	boolesch	false	false	true	false	false
	bitparallel	0	0	1	0	0
L(`c`)	boolesch	false	false	true	false	true
	bitparallel	0	0	1	0	1

Tabelle 6: Vergleich der booleschen und der bitparallelen Darstellung der R- und L-Mengen für die Terminale `a`, `b` und `c` des Musters $p =$ `abcac`.

In Tabelle 6 steht für jedes Symbol jeweils eine Zeile mit der booleschen Darstellung und darunter eine Zeile mit der bitparallelen Darstellung. Beide Zeilen enthalten dieselbe Information, weil an jeder Position der Wert `true` durch eine 1 und der Wert `false` durch eine 0 ersetzt wurde. Für das Terminal `a` ergibt sich in der booleschen Darstellung R(`a`) = {true, false, false, true, false} und daraus das Bitwort `10010`. Für die L-Menge ergibt sich L(`a`) = {false, false, false, true, false} und daraus das Bitwort `00010`. Für das Terminal `b` ergibt sich in beiden Mengen {false, true, false, false, false} und daraus jeweils das Bitwort `01000`. Für das Terminal `c` ergibt sich R(`c`) = {false, false, true, false, false} und daraus `00100`, sowie L(`c`) = {false, false, true, false, true} und daraus `00101`. Die bitparallele Zeile benötigt dabei nur fünf Bits statt fünf Bytes. Entscheidend ist jedoch nicht der geringere Speicherbedarf, sondern dass sich alle Positionen in dieser Darstellung durch eine einzige Bitoperation gemeinsam verarbeiten lassen.

Darstellung für Muster ab 65 Zeichen

Wenn das Muster länger als 64 Zeichen ist, dann passen die R- und L-Mengen nicht mehr in ein einziges Bitwort. Die Bits werden in diesem Fall auf mehrere Blöcke verteilt, die zusammen einen Bitvektor bilden. Jeder Block hat eine Länge von 64 Bit. Wenn das Muster länger als ein Block ist, dann wird ein weiterer Block angelegt. Die Anzahl der benötigten Blöcke ergibt sich damit aus:

$$\text{Anzahl der Blöcke} = \left\lceil \frac{m}{64} \right\rceil$$

Der letzte Block wird dabei in der Regel nicht vollständig genutzt. Wenn m ohne Rest durch 64 teilbar ist, dann sind alle Blöcke vollständig belegt. Sonst enthält der letzte Block nur m mod 64 genutzte Bits und die übrigen Bits dieses Blocks bleiben auf 0.

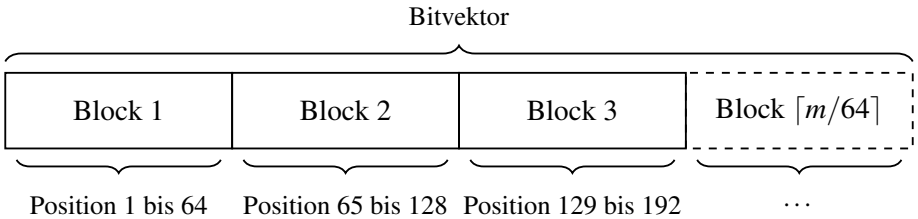

Abbildung 4: Aufteilung eines Bitvektors in Blöcke. Jeder Block hat eine Länge von 64 Bit und enthält 64 Positionen des Musters. Der erste Block liegt zwischen den Positionen 1 und 64, der zweite Block zwischen den Positionen 65 und 128 und der dritte Block zwischen den Positionen 129 und 192. Die Blöcke werden auf diese Weise fortgesetzt, bis alle Positionen des Musters abgedeckt sind. Der Block mit der Nummer $\lceil m/64 \rceil$ ist der letzte davon.

Das Prinzip der Darstellung bleibt dabei gleich. Jede Position des Musters entspricht weiterhin genau einem Bit, nur liegt dieses Bit jetzt in einem bestimmten Block. Die Position 1 liegt im ersten Block und die Position 65 liegt im zweiten Block. Der Unterschied zeigt sich erst bei den Operationen, weil eine Operation nicht mehr auf einem einzigen Bitwort ausgeführt wird, sondern auf allen Blöcken des Bitvektors. Wie diese Operationen ablaufen, wird in Abschnitt 4.4 beschrieben.

4.3 Bitparallele Berechnung der Regelinformationen

In diesem Abschnitt werden Muster mit $m \leq 64$ behandelt. Die R- und L-Mengen einer Regel liegen dabei jeweils in einem einzigen Bitwort. Für die Berechnung der Regelinformationen und für den Match-Check werden deshalb logische Operationen und Verschiebungen direkt auf diesen Bitwörtern ausgeführt. Der Unterschied zwischen einer bitparallelen Darstellung und einer booleschen Darstellung liegt darin, dass in der bitparallelen Darstellung alle Positionen der R- und L-Mengen zusammen verarbeitet werden, während in der booleschen Darstellung jede Position einzeln geprüft wird [11].

Wie in Abschnitt 3.3 gezeigt, gibt es beim Präfix zwei Fälle. Sei $A \rightarrow BC$ eine Regel. Das Präfix liegt entweder vollständig in der Expansion von C oder es überspannt die Grenze zwischen B und C. Beim Suffix gilt dasselbe aber in die andere Richtung. Das Suffix liegt entweder vollständig in der Expansion von B oder es überspannt die Grenze zwischen B und C. Bei der bitparallelen Verarbeitung werden alle Positionen der R- und L-Mengen einer Regel auf einmal verarbeitet, deshalb ist es wichtig, den Bereich der Positionen der R-und L-Mengen festzulegen. Dieser Bereich hängt von der Länge der Expansionen einer Regel ab. Für jeden der Fälle der R- und L-Mengen wird eine entsprechende Bitmaske erstellt. Eine Bitmaske ist ein Bitwort, das an bestimmten Positionen den Wert 1 hat und an allen anderen Positionen den Wert 0. Die Anzahl der benötigten Einsen in einer Bitmaske hängt von der Länge des Präfixes bzw.

des Suffixes ab. Die R- bzw. L-Menge wird dann mit dieser Maske verundet. Je nachdem, um welchen Fall es sich handelt, werden dabei nur bestimmte Positionen verarbeitet [2].

Berechnung der R-Menge

Für die R-Menge werden die beiden Fälle aus Abschnitt 3.3 nacheinander bitparallel berechnet und anschließend zusammengeführt:

- **Fall 1: Das Präfix liegt vollständig in C.**
  Wenn das Präfix vollständig in $[1 \dots j]$ liegt und $j \leq n_C$ gilt, dann ist das Präfix höchstens so lang wie die Expansion von C. Um sicherzustellen, dass das Präfix in diesem Bereich liegt, wird eine Bitmaske $low(n_C)$ erstellt. Die Maske hat an den Positionen $[1 \dots n_C]$ den Wert 1 und an den übrigen Positionen den Wert 0. Bei $n_C \geq 64$ haben alle Positionen der Bitmaske den Wert 1. Diese Maske wird mit der R-Menge einer Expansion verundet, um zu prüfen, ob das Präfix vollständig in C liegt:

  $$R_{A1} = R(C) \wedge low(n_C)$$

- **Fall 2: Das Präfix überspannt die Grenze zwischen B und C.**
  In diesem Fall werden die Positionen k gesucht. Wenn $k \in R(B)$ und $(k+1) \in L(C)$ gilt, dann beginnt das Präfix in B und endet in C. Die Positionen des Präfixes, die in B sind, werden als *Kandidaten* bezeichnet. Um zu prüfen, ob das Präfix die Grenze zwischen beiden Kindern überspannt, muss $L(C)$ um eine Stelle nach links verschoben und dann mit $R(B)$ verundet werden.

  $$Kandidat = R(B) \wedge (L(C) \ll 1)$$

  Ein Kandidat mit dem Wert 1 an der Position k bedeutet, dass $k \in R(B)$ und $(k+1) \in L(C)$ gilt. Das Präfix $p[1 \dots k]$ endet also am Ende von B und der nächste Teil $[k+1 \dots m]$ des Musters beginnt am Anfang von C. Ein Kandidat gibt nur die Länge des Teils an, der in der Expansion von B endet, also $[1 \dots k]$. Das Präfix von A besteht aber aus diesem Teil und aus der gesamten Expansion von C, deshalb hat $R(A)$ die Länge $j = k + n_C$. Damit die Kandidaten wirklich in B und nicht in C liegen, wird jede Position k um n_C Positionen nach rechts verschoben, damit die Positionen der Kandidaten hinter den Positionen der Expansion von C liegen und der neuen Länge $j = n_C + k$ entsprechen:

  $$R_{A2} = Kandidat \gg n_C$$

Um $R(A)$ zu bestimmen, werden beide Mengen aus R_{A1} und R_{A2} verodert und dann mit der Maske $low(m-1)$ verundet, weil die R-Menge nur die Positionen von 1 bis $m-1$ enthält:

$$R(A) = (R_{A1} \vee R_{A2}) \wedge low(m-1)$$

Beispiel: Sei $p = \texttt{abcac}$ mit $m = 5$ und sei $A \to \texttt{ca}$ eine Regel, die Teil eines Ableitungsbaums ist. $n_\texttt{a}$ und $n_\texttt{c}$ haben jeweils die Länge 1.

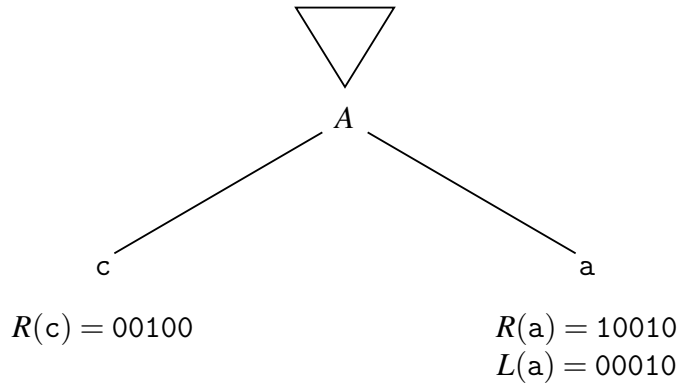

Abbildung 5: Die Regel $A \to \texttt{ca}$ mit den bitparallelen R- und L-Mengen ihrer beiden Terminale für das Muster $p = \texttt{abcac}$.

Abbildung 5 zeigt die Regel $A \to ca$. Zuerst werden die R- und L-Mengen von a und c direkt aus dem Muster berechnet. c kommt im Muster an den Positionen 3 und 5 vor. Nur die Position 3 liegt im Bereich der R-Menge. Es ergibt sich also $R(\texttt{c}) = \texttt{00100}$. Das Zeichen a kommt im Muster an den Positionen 1 und 4 vor und beide liegen im Bereich der R-Menge, deshalb ist $R(\texttt{a}) = \texttt{10010}$. Für die L-Menge von a liegt nur die Position 4 im Bereich, deshalb ist $L(\texttt{a}) = \texttt{00010}$. Um R_A zu bestimmen, müssen R_{A1} und R_{A2} berechnet werden. Für R_{A1} wird geprüft, welche Präfixe vollständig in a liegen. Die Maske ist in diesem Fall $low(n_\texttt{a}) = low(1) = \texttt{10000}$ und die Verundung mit $R(\texttt{a})$ ergibt: $R_{A1} = \texttt{10010} \wedge \texttt{10000} = \texttt{10000}$. Die Position 4 wird entfernt, weil $n_\texttt{a} = 1$ ist. Um R_{A2} zu berechnen, wird $L(\texttt{a})$ um eine Stelle nach links verschoben, und dann mit $R(\texttt{c})$ verundet. Das ergibt: $Kandidat = \texttt{00100} \wedge \texttt{00100} = \texttt{00100}$. Dieser Kandidat wird dann um $n_\texttt{a} = 1$ Position nach rechts verschoben, um R_{A2} zu berechnen: $R_{A2} = \texttt{00100} \gg 1 = \texttt{00010}$. Jetzt muss R_{A1} mit R_{A2} verodert und dann mit der Maske $low(m-1) = low(4) = \texttt{11110}$ verundet werden, um $R(A)$ zu bestimmen: $R(A) = (\texttt{10000} \vee \texttt{00010}) \wedge \texttt{11110} = \texttt{10010}$.

Berechnung der L-Menge

Die L-Menge wird nach demselben Prinzip berechnet wie die R-Menge, nur in die andere Richtung. Auch hier gibt es zwei Fälle, für die jeweils eine Bitmaske benötigt wird:

- **Fall 1: Das Suffix liegt vollständig in B.**
  Wenn das Suffix vollständig in $[j \ldots m]$ liegt und $m - j + 1 \leq n_B$ gilt, dann ist das Suffix höchstens so lang wie die Expansion von B. Ein Suffix dieser Länge beginnt frühestens an der Position $m - n_B + 1$. Um sicherzustellen, dass das Suffix in diesem Bereich liegt, wird eine Bitmaske $high(n_B)$ erstellt. Die Maske hat an den Positionen $[m - n_B + 1 \ldots m]$ den Wert 1 und an allen anderen Positionen den Wert 0. Wenn $n_B \geq m$ gilt, dann haben alle m Positionen der Maske den Wert 1. Diese Maske wird dann mit $L(B)$ verundet, um zu prüfen, ob das Suffix vollständig in B liegt:

  $$L_{A1} = L(B) \wedge high(n_B)$$

- **Fall 2: Das Suffix überspannt die Grenze zwischen B und C.**
  Wenn $j \in L(B)$ und $(j + n_B) \in L(C)$ gilt, dann beginnt das Suffix in B und endet in C. Damit beide Positionen ein Suffix bilden, müssen beide Positionen den Wert 1 haben. Um dies zu prüfen muss $L(C)$ um n_B Stellen nach links verschoben und dann mit $L(B)$ verundet werden:

  $$L_{A2} = L(B) \wedge (L(C) \ll n_B)$$

Eine Position $j \in L_{A2}$ mit dem Wert 1 bedeutet, dass $j \in L(B)$ und $(j+n_B) \in L(C)$ gilt. Das Suffix beginnt also am Anfang von B und setzt sich ab der Position $j+n_B$ in C fort. Anders als bei der R-Menge ist hier keine weitere Verschiebung nötig, weil ein Suffix immer an der Position j beginnen muss und diese Position durch die Fortsetzung in C nicht verändert wird.

Um $L(A)$ zu bestimmen, werden beide Mengen L_{A1} und L_{A2} verodert und dann mit der Maske $high(m-1)$ verundet, weil die L-Menge nur die Positionen von 2 bis m enthält:

$$L(A) = (L_{A1} \vee L_{A2}) \wedge high(m-1)$$

Beispiel: Sei p = `cacba` mit m = 5 und sei $A \rightarrow$ `ac` eine Regel, die Teil eines Ableitungsbaumes ist. $n_{\mathtt{a}}$ und $n_{\mathtt{c}}$ sind jeweils 1.

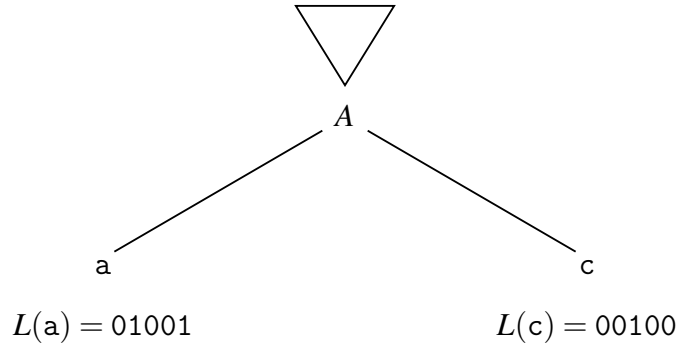

Abbildung 6: Die Regel $A \rightarrow$ `ac` mit den bitparallelen L-Mengen ihrer beiden Terminale für das Muster $p =$ `cacba`.

Die L-Mengen von `a` und `c` werden direkt aus dem Muster berechnet. `a` kommt im Muster an den Positionen 2 und 5 vor und beide liegen im Bereich der L-Menge, deshalb ist $L($`a`$)$ = `01001`. Das Zeichen `c` kommt an den Positionen 1 und 3 vor, aber nur die Position 3 liegt im Bereich der L-Menge, deshalb ist $L($`c`$)$ = `00100`. Um $L(A)$ zu bestimmen, müssen L_{A1} und L_{A2} berechnet werden. Für L_{A1} wird geprüft, welche Suffixe vollständig in `a` liegen. Die Maske ist in diesem Fall $high(n_{\mathtt{a}})$ = $high(1)$ = `00001` und die Verundung mit $L($`a`$)$ ergibt: L_{A1} = `01001` $\wedge$ `00001` = `00001`. Die Position 2 wird entfernt, weil das Suffix an dieser Position die Länge 4 hat und $n_{\mathtt{a}}$ = 1 ist. Um L_{A2} zu berechnen, wird $L($`c`$)$ um $n_{\mathtt{a}}$ = 1 Stelle nach links verschoben und dann mit $L($`a`$)$ verundet. Das ergibt: L_{A2} = `01001` $\wedge$ `01000` = `01000`. Die Position 2 ist also gültig, weil das Suffix `acba` mit dem Zeichen `a` beginnt und sich ab der Position 3 in `c` fortsetzt. Am Ende muss L_{A1} mit L_{A2} verodert und dann mit der Maske $high(m-1)$ = $high(4)$ = `01111` verundet werden, um $L(A)$ zu bestimmen: $L(A)$ = (`00001` $\vee$ `01000`) $\wedge$ `01111` = `01001`.

Match-Check

Wie in Abschnitt 3.3 gezeigt, liegt ein Match vor, wenn es eine Position k gibt, für die $k \in R(B)$ und $(k+1) \in L(C)$ gilt und zusätzlich $k \leq n_B$ und $m-k \leq n_C$ erfüllt sind. Die ersten beiden Bedingungen bilden die Kandidaten, die bereits bei der Berechnung der R-Menge bestimmt wurden. Die beiden Längenbedingungen werden dagegen erst hier geprüft. Ein Kandidat kann dabei ein Präfix darstellen oder ein Vorkommen des Musters. In diesem Abschnitt wird der Fall des Vorkommens behandelt, bei dem ein Kandidat zusätzlich die beiden folgenden Bedingungen erfüllen muss:

- **Bei $k \leq n_B$:** Um diese Bedingung zu prüfen, wird die Maske $low(n_B)$ verwendet. In dieser Maske haben die ersten Bits an den Positionen $[1 \ldots n_B]$ den Wert 1. Dieser Bereich stellt das Präfix des Musters dar.

- **Bei $m-k \leq n_C$:** Um diese Bedingung zu prüfen, wird die Maske $high(n_C+1)$ verwendet. In dieser Maske haben die letzten Bits an den Positionen $[m-n_C \ldots m]$ den Wert 1. Dieser Bereich stellt das Suffix des Musters dar.

Um zu prüfen, ob ein Kandidat diese Bedingungen erfüllt und damit ein Vorkommen des Musters darstellt und kein Präfix, wird zuerst der Kandidat wie bereits gezeigt, berechnet:

$$Kandidat = R(B) \wedge (L(C) \ll 1)$$

Eine Position mit dem Wert 1 stellt dabei einen Kandidaten dar. Danach werden die beiden Längenbedingungen zu einer einzigen Maske M verundet:

$$M = low(n_B) \wedge high(n_C+1)$$

Um zu prüfen, ob eine Position mit dem Wert 1 im Kandidaten ein Vorkommen des Musters darstellt, wird der Kandidat mit der Maske M verundet. Die Anzahl der Vorkommen ergibt sich anschließend aus der Anzahl der Bits mit dem Wert 1 im Ergebnis, die im Folgenden mit $|\cdot|$ bezeichnet wird:

$$occ(BC) = |Kandidat \wedge M|$$

Jede Position im Ergebnis, die den Wert 1 hat, stellt ein Vorkommen des Musters dar. Wenn z. B. nur eine einzige Position den Wert 1 hat, dann kommt das Muster einmal in der Regel A vor. Wenn keine Position den Wert 1 hat, dann stellt der Kandidat nur ein Präfix dar und kein Vorkommen des Musters.

Vollständiges Beispiel: Sei p = `aa` mit $m = 2$ und seien $A \rightarrow BC$, $B \rightarrow$ `aa` und $C \rightarrow$ `aa` drei Regeln. Die Expansion von A ist damit `aaaa` und es gilt $n_B = n_C = 2$.

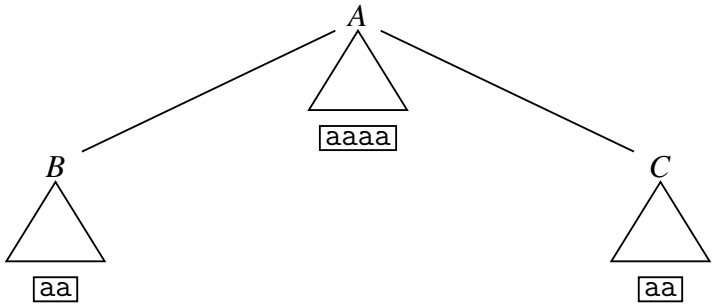

Abbildung 7: Ableitungsbaum für die Regel $A \rightarrow BC$ mit dem Muster $p =$ `aa`.

Die R- und L-Mengen des Terminals `a` werden direkt aus dem Muster berechnet. Es ergibt $R($`a`$)$ = `10` und $L($`a`$)$ = `01`.

Dann wird die Regel $B \rightarrow$ `aa` verarbeitet, wobei beide Kinder die Länge $n_{\texttt{a}} = 1$ haben. $R_{B1} = R($`a`$) \wedge low(1)$ = `10` $\wedge$ `10` = `10`. Für R_{B2} werden die Kandidaten berechnet, also $Kandidat = R($`a`$) \wedge (L($`a`$) \ll 1)$ = `10` $\wedge$ `10` = `10`. Die Verschiebung um $n_{\texttt{a}} = 1$ Position nach rechts ergibt R_{B2} = `10` $\gg 1$ = `01`. Die R-menge von B kann nun berechnet werden: $R(B) = (R_{B1} \vee R_{B2}) \wedge low(m-1)$ = (`10` $\vee$ `01`) $\wedge\ low(2-1)$ = `11` $\wedge$ `10` = `10`. Die Position 2 fällt dabei weg, weil sie außerhalb des Bereichs der R-Menge liegt.

Für die L-Menge ergibt sich $L_{B1} = L($`a`$) \wedge high(1)$ = `01` $\wedge$ `01` = `01` und $L_{B2} = L($`a`$) \wedge (L($`a`$) \ll 1)$ = `01` $\wedge$ `10` = `00`. Die L-Menge von B ist also: $L(B)$ = (`01` $\vee$ `00`) $\wedge\ high(m-1)$ = `01`.

Es wird für B geprüft, ob das Muster vorkommt. Dafür wird nochmal der Kandidat `10` verwendet. Die Maske ergibt sich aus $M = low(n_{\texttt{a}}) \wedge high(n_{\texttt{a}}+1)$ = `10` $\wedge$ `11` = `10` und die Verundung ergibt $Kandidat \wedge M$ = `10` $\wedge$ `10` = `10`. Es gilt also $occ(B) = 1$, also das Muster kommt einmal in `aa` vor. Die Regel $C \rightarrow$ `aa` liefert dieselben Werte, also $R(C)$ = `10`, $L(C)$ = `01` und $occ(C) = 1$.

Am Ende wird die Regel $A \to BC$ verarbeitet, wobei $n_B = n_C = 2$ gilt. $R_{A1} = R(C) \wedge low(2) = 10 \wedge 11 = 10$. Für R_{A2} werden die Kandidaten berechnet, also $Kandidat = R(B) \wedge (L(C) \ll 1) = 10 \wedge 10 = 10$. Die Verschiebung um $n_C = 2$ Positionen nach rechts ergibt $R_{A2} = 10 \gg 2 = 00$. Das bedeutet, dass die Eins im Kandidat von A nicht für ein Präfix steht. Die R-Menge von A kann trotzdem berechnet werden: $R(A) = (R_{A1} \vee R_{A2}) \wedge low(m-1) = (10 \vee 00) \wedge low(2-1) = 10 \wedge 10 = 10$.

Für die L-Menge ergibt sich $L_{A1} = L(B) \wedge high(2) = 01 \wedge 11 = 01$ und $L_{A2} = L(B) \wedge (L(C) \ll 2) = 01 \wedge 00 = 00$. Die L-Menge von A ist also: $L(A) = (01 \vee 00) \wedge high(m-1) = 01$.

Zuletzt wird für A geprüft, ob das Muster vorkommt. Dafür wird nochmal der Kandidat 10 verwendet. Die Maske ergibt sich aus $M = low(n_B) \wedge high(n_C + 1) = 11 \wedge 11 = 11$ und die Verundung ergibt $Kandidat \wedge M = 10 \wedge 11 = 10$. Es gilt also $occ(A) = occ(B) + occ(C) + occ(BC) = 1 + 1 + 1 = 3$, also das Muster kommt dreimal in aaaa vor.

4.4 Verwendung von Bitvektoren

In diesem Abschnitt werden Muster mit $m > 64$ behandelt. Diese Muster können nicht in einem einzelnen Bitwort dargestellt werden, da ein Rechner in der Regel nur 64 Bit auf einmal verarbeiten kann. Für Muster mit $m > 64$ werden deshalb Bitvektoren verwendet, die aus mehreren Blöcken bestehen, wobei jeder Block die Länge eines Bitwortes, also 64, hat.

Im vorherigen Abschnitt wurde die Berechnung für Muster der Länge $m \leq 64$ gezeigt. Bei längeren Mustern wird die Verarbeitung auf mehrere Blöcke erweitert. Die Funktionsweise des bitparallelen Suchalgorithmus aus Abschnitt 4.3 wird dabei weiterhin verwendet. Die Ausführung der logischen Operationen und der Verschiebungen muss jedoch an die Verarbeitung mehrerer Blöcke angepasst werden. Im Folgenden wird gezeigt, wie diese Operationen auf Bitvektoren durchgeführt werden.

Logische Operationen auf Bitvektoren

Die logischen Operationen werden bei Bitvektoren blockweise durchgeführt. Wenn zwei Bitvektoren miteinander verknüpft werden, werden die Bits mit demselben Index aus den entsprechenden Blöcken miteinander verknüpft. Seien V und W zwei Bitvektoren, sei i der Index eines Blocks in V und W, sei j der Index eines Bits innerhalb eines Blocks in V und W und sei b die Anzahl der Blöcke. Dann gilt für die beiden Operationen:

$$(V \wedge W)[i][j] = V[i][j] \wedge W[i][j] \quad \text{für alle } 1 \leq i \leq b \text{ und } 1 \leq j \leq 64$$

$$(V \vee W)[i][j] = V[i][j] \vee W[i][j] \quad \text{für alle } 1 \leq i \leq b \text{ und } 1 \leq j \leq 64$$

Jedes Bit mit dem Index j im Block i von V wird also mit dem Bit verknüpft, das im Block i von W denselben Index j hat. Das Ergebnis ist damit ein neuer Bitvektor mit b Blöcken, bei dem jedes Bit mit dem Index j im Block i den Wert der Verknüpfung von $V[i][j]$ und $W[i][j]$ hat. Eine solche logische Operation kann wie folgt dargestellt werden:

Verschiebungen auf Bitvektoren

Verschiebungen bei Bitvektoren unterscheiden sich von logischen Operationen [12]. Bei logischen Operationen bleibt ein Bit vor und nach der Operation an derselben Position. Bei Verschiebungen ändert sich dagegen die Position aller Bits. Dadurch können einige Bits ihren Block verlassen und in einen anderen Block verschoben werden. Im Folgenden wird erklärt, wie Verschiebungen bei Bitvektoren funktionieren und wie Bits von einem Block in einen anderen übertragen werden.

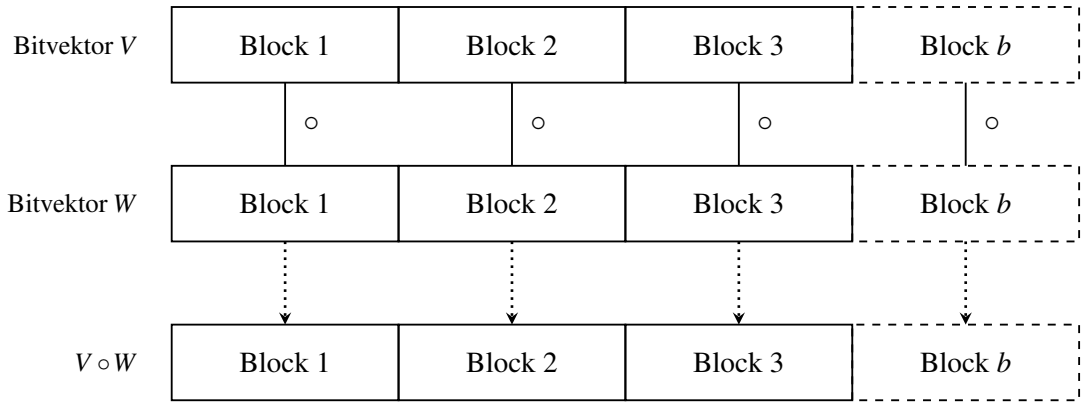

Abbildung 8: Logische Operationen auf zwei Bitvektoren V und W, wobei $\circ$ für die UND- oder ODER-Verknüpfung steht. Dabei wird jeder Block i von V mit dem entsprechenden Block i von W verknüpft. Dies wird für alle Blöcke bis zum letzten Block b durchgeführt. Das Ergebnis ist ein neuer Bitvektor, der die Werte nach der Verknüpfung der entsprechenden Blöcke enthält.

Rechtsverschiebungen von Bitvektoren

Sei V ein Bitvektor, und sei n die Anzahl der Positionen, um die V verschoben wird. Jedes Bit von V wird durch ein Paar (i, j) beschrieben, wobei $i \in \{1, \ldots, b\}$ der Index des Blocks ist, b die Anzahl der Blöcke von V ist, und $j \in \{1, \ldots, 64\}$ der Index des Bits im Block ist. Dabei gilt $n = q \cdot 64 + r$, wobei $q = \lfloor n/64 \rfloor$ angibt, um wie viele ganze Blöcke verschoben wird, und $r = n \bmod 64$ um wie viele Positionen innerhalb eines Blocks verschoben wird. Die Verschiebung besteht also aus zwei Teilen: einer Verschiebung um q ganze Blöcke und einer Verschiebung um $r < 64$ Positionen innerhalb der Blöcke. Für ein Bit (i, j) gibt es zwei Fälle:

- $j + r \leq 64$: Das Bit (i, j) bekommt die neue Position $j + r$ in seinem Block. Dieser Fall ist derselbe wie bei der Verschiebung eines einzelnen Bitworts. Der Block wird aber um q Stellen nach rechts verschoben und bekommt den Index $i + q$. Das Bit steht danach also an der Position $(i + q,\ j + r)$.
- $j + r > 64$: Das Bit befindet sich an den Positionen $65 - r \ldots 64$ und wird an die Position $(i + q + 1,\ j + r - 64)$ übertragen. Das gilt für die letzten r Positionen im Block i. Diese Bits werden im Folgenden als *Übertrag* bezeichnet.

Der Übertrag steht im Block i an den letzten r Positionen, und muss nach der Verschiebung an den ersten r Positionen des Zielblocks stehen. Damit der Übertrag von Block i in den nächsten Block übertragen werden kann, muss er also um $64 - r$ Positionen nach links geschoben werden und dann mit dem Zielblock verodert werden. Damit wird sichergestellt, dass die Bits an den Positionen $65 - r \ldots 64$ aus dem Block i jetzt die Positionen $1 \ldots r$ im Zielblock haben.

Für $n < 64$ ist $q = 0$ und $r = n$, jeder Block behält dann seinen Index und es wird nur innerhalb der Blöcke verschoben. Wenn der Zielblock außerhalb von V liegt, also wenn $i + q > b$ gilt, dann gehen die r Bits von dem letzten Block verloren. Außerdem erhalten im ersten Block die ersten r Positionen den Wert 0, weil vor dem ersten Block kein weiterer Block liegt.

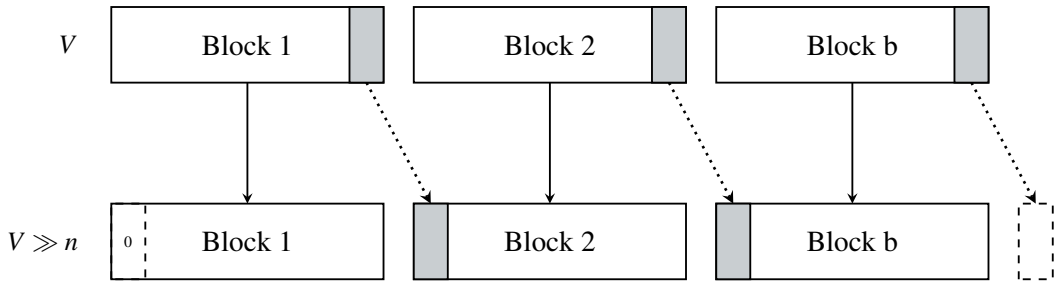

Abbildung 9: Verschiebung eines Bitvektors um n Positionen nach rechts. Die Bits eines Blockes werden innerhalb des Blockes verschoben, während die letzten r Bits eines Blockes an den Anfang des nächsten Blockes wandern. Diese Bereiche sind grau markiert. Die ersten r Positionen des ersten Blockes erhalten den Wert 0, weil vor dem ersten Block kein weiterer Block liegt. Die letzten r Bits von Block b gehen dagegen verloren, weil danach kein weiterer Block kommt.

Linksverschiebungen von Bitvektoren

Bei einer Linksverschiebung gilt dasselbe Prinzip wie bei der Rechtsverschiebung, nur in die andere Richtung. Für ein Bit (i, j) gibt es auch hier zwei Fälle:

- Wenn $j - r \geq 1$ ist, bekommt das Bit (i, j) die neue Position $j - r$ in seinem Block. Der Block wird um q Stellen nach links verschoben und bekommt den Index $i - q$. Das Bit steht danach also an der Position $(i - q,\ j - r)$.
- Wenn $j - r < 1$ ist, dann handelt es sich um einen Übertrag, und die Bits an den Positionen $1 \ldots r$ werden an die Position $(i - q - 1,\ j - r + 64)$ übertragen.

Der Übertrag muss nach der Verschiebung an den letzten r Positionen des Zielblocks stehen. Er wird deshalb um $64 - r$ Positionen nach rechts geschoben und dann mit dem Zielblock verodert. Damit haben die Bits an den Positionen $1 \ldots r$ aus dem Block i jetzt die Positionen $65 - r \ldots 64$ im Zielblock.

Die ersten r Bits von dem ersten Block gehen verloren, wenn $i - q < 1$ gilt. Außerdem werden im letzten Block die letzten r Positionen mit dem Wert 0 gefüllt, weil nach diesem Block kein weiterer Block liegt.

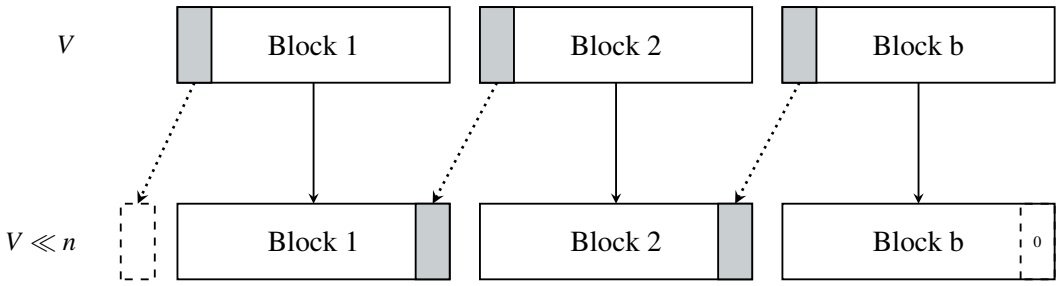

Abbildung 10: Verschiebung eines Bitvektors um n Positionen nach links. Anders als bei der Rechtsverschiebung wandern die ersten r Bits eines Blockes an das Ende des vorherigen Blockes. Die letzten r Positionen von Block b erhalten den Wert 0, die ersten r Bits des ersten Blockes gehen verloren.

4.5 Verarbeitung von Regeln mit rechten Seiten beliebiger Länge

Wie in Abschnitt 3.1 gezeigt, arbeitet der Suchalgorithmus auf Nichtterminalen mit einer binären Struktur. Diese Struktur ermöglicht eine direkte Berechnung der R- und L-Mengen, weil eine Regel immer die Form $A \to BC$ hat. In Abschnitt 3.3 wurde gezeigt, wie die Informationen von A aus den Informatio-

nen ihrer beiden Kinder bestimmt werden, also die R- und L-Mengen, die Länge der Expansion und die Anzahl der Vorkommen des Musters.

Nach der Erweiterung des Dateiformats dürfen die Nichtterminale beliebig viele Symbole auf der rechten Seite haben [13, 10]. Ein Nichtterminal X hat also nicht mehr die Form $X \to Y_1 Y_2$, sondern die Form $X \to Y_1 \dots Y_k$ mit $k \geq 2$. Die Terminale bleiben dabei unverändert. Sie stehen wie in Abschnitt 3.1 gezeigt weiterhin direkt auf der rechten Seite einer Regel und für sie sind die Zahlen von 0 bis 255 reserviert. Ab der Zahl 256 werden die Nichtterminale durchnummeriert. Sie sind weiterhin topologisch sortiert, sodass eine Regel X_i nur auf Regeln X_j mit $j < i$ verweist. Da die rechten Seiten aber keine feste Länge mehr haben, werden zusätzlich zwei reservierte Symbole verwendet [13, 10]. Vor der Kodierung einer Regel mit $k > 2$ steht der Marker M_B und nach der Kodierung der Marker M_E, während bei der Startregel nur nach der Kodierung der Marker M_E steht. Der Algorithmus kann damit zwischen einer solchen Regel und der Startregel unterscheiden. Eine Regel mit genau zwei Symbolen wird dagegen wie bisher gespeichert und braucht keine Marker.

Beispiel Der Text `abcac` wird wie folgt kodiert:

Regel	**Kodierung**
$S \to X_2 c$	$[257, 99, M_E]$
$X_1 \to ab$	$[97, 98]$
$X_2 \to X_1 ca$	$[M_B, 256, 99, 97, M_E]$

Die Startregel steht mit ihrem Marker M_E am Anfang der Datei, danach folgen die Regeln. Das erste Nichtterminal X_1 erhält die Zahl 256 und X_2 die Zahl 257. Die Regel X_1 hat genau zwei Symbole und braucht deshalb keine Marker. Die Regel X_2 hat drei Symbole und steht deshalb zwischen M_B und M_E.

Die Symbole einer Regel mit $k > 2$ werden während der Suche von links nach rechts kombiniert, genau wie die Symbole der Startregel. Zuerst werden also die ersten beiden Symbole kombiniert, danach wird dieses Zwischenergebnis mit dem dritten Symbol kombiniert, danach mit dem vierten und so weiter, bis alle Symbole der Regel verarbeitet sind. Jede dieser Kombinationen wird dabei wie eine binäre Regel behandelt, sodass für sie die R- und L-Mengen, die Länge der Expansion und die Anzahl der Vorkommen berechnet werden.

Beispiel: Sei p = `abca` mit $m = 4$ und sei $X \to$ `abca` eine Regel mit $k = 4$. Die Expansion von X ist damit genau das Muster.

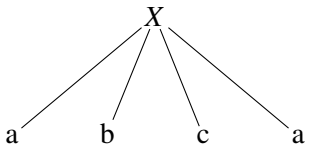

Abbildung 11: Ableitungsbaum der Regel $X \to$ `abca` mit vier Symbolen auf der rechten Seite.

Zuerst werden die R- und L-Mengen der Terminale direkt aus dem Muster berechnet. Danach werden die vier Symbole der Regel schrittweise kombiniert. Die Terminale `a` und `b` werden zum Zwischenergebnis T_2 mit der Expansion `ab` kombiniert. Die R-Menge von T_2 enthält an der Position 2 den Wert 1, weil das Präfix $p[1 \dots 2]$ = `ab` am Ende der Expansion endet. Ein Vorkommen entsteht noch nicht, weil die Expansion kürzer als das Muster ist. Das Zwischenergebnis T_2 wird mit dem Terminal `c` zu T_3 mit der Expansion `abc` kombiniert. Die R-Menge von T_3 enthält jetzt an der Position 3 den Wert 1, weil das Präfix $p[1 \dots 3]$ = `abc` am Ende der Expansion endet. Das Zwischenergebnis T_3 wird mit dem letzten Terminal `a` zu T_4 mit der Expansion `abca` kombiniert. Erst bei diesem letzten Schritt entsteht ein Vorkommen. Die Position 3 steht in der R-Menge von T_3 und die Position 4 steht in der L-Menge von `a`, sodass das Präfix `abc` am Ende von T_3 endet und das Suffix `a` am Anfang des Terminals beginnt. Beide

Teile bilden zusammen das Muster und der Match-Check liefert $occ(T_4) = 1$. Das Zwischenergebnis T_4 enthält damit die Informationen von X, also $occ(X) = 1$.

5 Existenzsuche

Bisher wurde gezeigt, wie der Algorithmus ein Muster findet und wie er seine Vorkommen zählt. Die Informationen einer Regel werden dabei aus den Informationen ihrer Kinder berechnet, sodass die Anzahl der Vorkommen von unten nach oben zusammengesetzt wird. Am Ende steht mit $occ(S)$ die Anzahl aller Vorkommen des Musters im Text.

Oft wird aber nicht die Anzahl der Vorkommen gesucht, sondern nur die Antwort auf die Frage, ob das Muster überhaupt vorkommt. Dafür muss die Grammatik nicht vollständig verarbeitet werden, weil ein einziges Vorkommen die Frage beantworten kann. In diesem Abschnitt wird zuerst gezeigt, wie der Suchalgorithmus beim ersten Treffer die Suche abbricht. Danach wird gezeigt, warum die Tiefensuche nicht die optimale Lösung bietet.

5.1 Early-Exit

Wie in Abschnitt 3.3 gezeigt, werden für jede Regel vier Informationen gespeichert: die R- und L-Mengen, die Länge der Expansion und die Anzahl der Vorkommen des Musters, die mit *occ* bezeichnet wird [9]. Der Wert *occ* einer Regel $X \to Y_1 \dots Y_k$ ergibt sich dabei aus den Vorkommen ihrer Symbole und aus den Vorkommen, die beim Match-Check an ihren Grenzen gefunden werden.

Die Existenzsuche verwendet denselben Ablauf, bricht aber ab, sobald der erste Treffer entsteht. Nach jeder verarbeiteten Regel X wird deshalb geprüft, ob $occ(X) > 0$ gilt.

- Wenn $occ(X) > 0$ ist, dann kommt das Muster in der Expansion von X vor und damit auch im Text. Die Suche wird an dieser Stelle beendet und liefert das Ergebnis `true`. Die übrigen Regeln werden nicht mehr verarbeitet.
- Wenn $occ(X) = 0$ ist, dann kommt das Muster in der Expansion von X nicht vor und die nächste Regel wird verarbeitet. Dies wird fortgesetzt, bis eine Regel mit $occ > 0$ gefunden wird oder bis die Startregel erreicht ist. Gilt am Ende $occ(S) = 0$, dann kommt das Muster im Text nicht vor und die Suche bricht nicht früher ab.

Der *Early-Exit* sorgt also dafür, dass der Suchalgorithmus so wenige Regeln wie möglich verarbeitet. Wenn aber das Muster im Text nicht vorkommt, dann werden alle Regeln durchlaufen und die Existenzsuche braucht genauso lange wie die Zählsuche.

Beispiel: Sei p = `ab` mit m = 2 und seien $B \to$ `ab` und $A \to B$`cab` zwei Regeln. Die Expansion von B ist `ab` und die Expansion von A ist `abcab`.

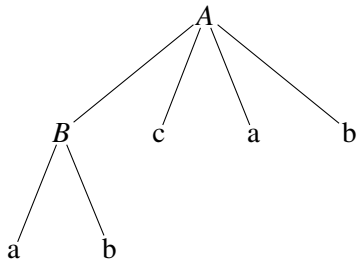

Abbildung 12: Ableitungsbaum der Regeln $A \to B$`cab` und $B \to$ `ab` für den Text `abcab`.

Zuerst werden die R- und L-Mengen der Terminale direkt aus dem Muster berechnet. Danach wird die Regel B verarbeitet, weil sie wie in Abschnitt 4.5 beschrieben vor der Regel A in der Datei steht. Bei der Berechnung ihrer Informationen ergibt sich $occ(B) = 1$, weil das Muster `ab` in der Expansion von B vorkommt. Damit gilt $occ(B) > 0$ und die Suche wird an dieser Stelle beendet. Die Regel A wird also nicht mehr verarbeitet, obwohl das Muster auch in ihrer Expansion `abcab` vorkommt. Für die Existenzsuche reicht aber das eine Vorkommen in B aus, um die Frage zu beantworten.

5.2 Existenzsuche mit Tiefensuche

Die *Tiefensuche* ist ein Verfahren, das einen Baum von oben nach unten durchläuft. Ein Teilbaum wird dabei vollständig besucht, bevor der nächste Teilbaum an die Reihe kommt [3]. Bei einem Ableitungsbaum ist die Wurzel die Startregel, sodass die Tiefensuche dort beginnt und in jedem Teilbaum über die linken Kinder bis zu den Terminalen läuft.

Für die Existenzsuche wurde ein Hybrid-Algorithmus getestet, bei dem die Tiefensuche den Weg Top-down steuert, während der Suchalgorithmus die Informationen der besuchten Regeln berechnet. Ausgehend von der Startregel wird zuerst das linke Kind betrachtet. Handelt es sich um ein Nichtterminal, dessen Informationen noch nicht berechnet wurden, dann wird dessen linkes Kind betrachtet. Handelt es sich um ein Terminal, dann werden seine Informationen berechnet und es wird zum nächsten Symbol daneben gegangen. Sobald zwei Symbole vorliegen, werden ihre Informationen kombiniert und der Match-Check wird durchgeführt. Ergibt sich dabei ein $occ > 0$, dann bricht die Suche sofort ab. Ein Nichtterminal kann außerdem an mehreren Stellen des Baumes vorkommen, deshalb werden seine Informationen nach der ersten Berechnung gespeichert und bei einem erneuten Besuch nur noch gelesen.

Der Vorteil dieses Verfahrens liegt darin, dass nur die Regeln berechnet werden, die auf dem Weg der Tiefensuche liegen. Die Reihenfolge richtet sich dabei nach dem Ableitungsbaum und nicht nach der Datei. Wenn das Muster weit unten links im Ableitungsbaum liegt, dann wird der Treffer also schon nach wenigen Regeln gefunden.

Der Einsatz der Tiefensuche auf dem Suchalgorithmus war für die Existenzsuche eines Musters aber eher negativ als positiv. Der Grund liegt darin, dass sich die Tiefensuche bei jedem Durchlauf über ein Nichtterminal erneut selbst aufruft, um festzustellen, ob die Informationen dieses Nichtterminals bereits gespeichert wurden. Auf der anderen Seite geht der Suchalgorithmus ohne Tiefensuche die Regeln in der Reihenfolge der Datei durch, liest die Informationen der Kinder und dann gibt er den Speicher einer Regel frei, wenn diese Regel nicht mehr gebraucht wird. Aus diesen Gründen ist die Existenzsuche mit Tiefensuche teurer, vor allem bei langen Mustern, weil die R- und L-Mengen bei $m > 64$ aus mehreren Blöcken bestehen. Deshalb wurde dieses Verfahren nicht übernommen und es bleibt beim Abbruch, wenn $occ(X) > 0$ gilt.

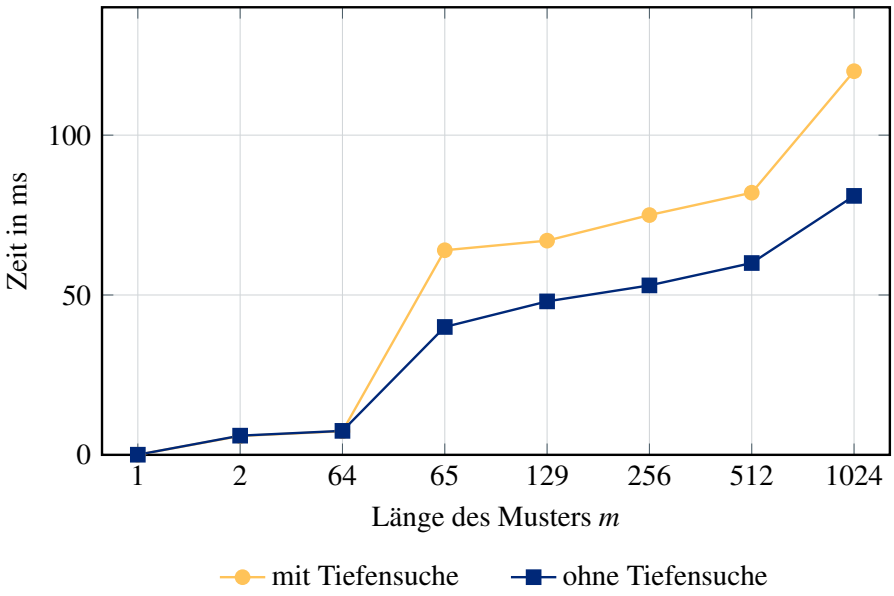

Abbildung 13: Durchschnittliche Laufzeit der Existenzsuche mit und ohne Tiefensuche für verschiedene Musterlängen (`Bible.txt`). Der Durchschnitt wurde über vier Fälle berechnet: ein Muster am Anfang, in der Mitte und am Ende des Ableitungsbaumes sowie ein Muster, das gar nicht vorkommt. Der deutliche Sprung zwischen $m = 64$ und $m = 65$ entsteht dadurch, dass ab dieser Länge Bitvektoren verwendet werden.

Wie in Abbildung 13 zu sehen ist, ist die Existenzsuche mit Tiefensuche bei fast allen Musterlängen langsamer als die ohne. Bei Mustern mit $m \leq 64$ ist der Unterschied klein, weil die Informationen einer Regel dort in einem einzigen Bitwort verarbeitet werden und eine Kombination deshalb wenig kostet. Der zusätzliche Aufwand der rekursiven Aufrufe spielt dann nur eine geringe Rolle. Bei Mustern mit $m > 64$ wächst dieser Unterschied deutlich. Der Grund ist, dass die Tiefensuche die Informationen jeder berechneten Regel behalten muss, weil ein Nichtterminal später an einer anderen Stelle des Baumes erneut besucht werden kann. Bei langen Mustern bestehen diese Informationen aus mehreren Blöcken, sodass die Tiefensuche mit zunehmender Musterlänge immer mehr Speicher belegt.

6 Implementierung des bitparallelen Suchalgorithmus

In den vorherigen Abschnitten wurde beschrieben, wie die R- und L-Mengen bitparallel dargestellt und mit logischen Operationen berechnet werden. In diesem Abschnitt wird gezeigt, wie diese Berechnung im Suchalgorithmus implementiert ist. Dafür werden für $m > 1$ zwei Varianten verwendet, die nach demselben Prinzip arbeiten, sich aber in der Darstellung der R- und L-Mengen unterscheiden. Für Muster mit einer Länge von höchstens 64 Zeichen werden die Mengen jeweils in einem 64-Bit-Wort gespeichert. Bei längeren Mustern reicht ein einzelnes Bitwort nicht mehr aus, weshalb dafür Bitvektoren verwendet werden. Für $m = 1$ wird die bitparallele Berechnung nicht benötigt, weil ein Vorkommen eines einzelnen Zeichens immer vollständig innerhalb eines Terminals liegt und nie über die Grenze zwischen zwei Symbolen hinweg entstehen kann. Es werden deshalb keine Präfixe und Suffixe des Musters verwaltet. Die Anzahl der Vorkommen ergibt sich direkt aus den Terminalen.

Welche Variante bei $m > 1$ ausgeführt wird, hängt direkt von der Länge m des Musters ab. Für $m \leq 64$ wird die Suche mit 64-Bit-Wörtern ausgeführt, während für $m > 64$ die Implementierung mit der Bibliothek `AWINT` verwendet wird [7]. Beide Varianten verarbeiten die Regeln der Grammatik von unten nach oben und kombinieren bei Regeln mit mehreren Symbolen auf der rechten Seite die Informationen

der Symbole schrittweise von links nach rechts. Zusätzlich kann die Berechnung bei einer Existenzsuche beendet werden, sobald ein erstes Vorkommen des Musters gefunden wurde.

6.1 Suche mit 64-Bit-Wörtern

In der ursprünglichen Implementierung wurden die R- und L-Mengen einer Regel jeweils als `Vec<bool>` gespeichert. Für jede Position des Musters wurde damit ein eigener boolescher Wert abgelegt. Auch die Berechnung der Informationen einer neuen Regel erfolgte positionsweise. Die Funktion `combine_infos` durchlief dafür alle m Positionen des Musters und prüfte die Bedingungen für die R-Menge, die L-Menge und den Match-Check für jede Position einzeln.

Für Muster mit $2 \leq m \leq 64$ wurde diese positionsweise Verarbeitung durch eine bitparallele Variante ersetzt. Die R- und L-Mengen werden dabei nicht mehr als boolesche Vektoren, sondern jeweils in einem einzigen Wert des Typs `u64` gespeichert. Ein `u64` besitzt 64 Bits und kann deshalb für jede Position eines solchen Musters genau ein Bit aufnehmen.

Damit dieselbe Struktur auch für längere Muster verwendet werden kann, sind die beiden Mengen nicht direkt als `u64` abgelegt, sondern über den Aufzählungstyp `BitSet`:

```
enum BitSet {
    Small(u64),
    Large(Awi),
}

struct Info {
    len: u64,
    right_set: BitSet,
    left_set: BitSet,
    occ: u64,
}
```

Die Variante `Small` enthält ein einzelnes 64-Bit-Wort und wird für $m \leq 64$ verwendet, die Variante `Large` einen Bitvektor vom Typ `Awi` für längere Muster. Welche Variante belegt wird, wird einmalig bei der Initialisierung der Terminale anhand von m festgelegt. Im Folgenden wird zunächst nur die Variante `Small` betrachtet. Die Bedeutung der gespeicherten Informationen bleibt unverändert. `len` enthält die Länge der Expansion, `right_set` die R-Menge, `left_set` die L-Menge und `occ` die Anzahl der Mustervorkommen.

Auch bei der Initialisierung der Terminale werden die Positionen direkt als Bits gespeichert. In der ursprünglichen Implementierung wurde bei einer passenden Position im booleschen Vektor der Wert `true` gesetzt. In der bitparallelen Variante wird stattdessen das entsprechende Bit gesetzt. Da beide Varianten von `BitSet` initialisiert werden müssen, geschieht das über die Hilfsfunktion `set_bit`. In der Implementierung werden die Bitpositionen von 0 an gezählt, während in den vorherigen Abschnitten von 1 an gezählt wurde. Die Position j aus der Definition entspricht daher dem Bit an Index $j-1$ im Code.

```
fn set_bit(set: &mut BitSet, i: usize) {
    match set {
        BitSet::Small(v) => *v |= 1u64 << i,
        BitSet::Large(a) => a.set(i, true).unwrap(),
    }
}

for j in 1..m {
    let k = j - 1;
    if pattern_bytes[j] == byte_val as u8 {
        set_bit(&mut info.left_set, j);
    }
    if pattern_bytes[k] == byte_val as u8 {
        set_bit(&mut info.right_set, k);
    }
}
```

Im Fall `Small` erzeugt der Ausdruck `1u64 « i` ein Bitwort, bei dem nur das Bit an der Position i den Wert 1 hat. Durch die ODER-Verknüpfung wird dieses Bit an der passenden Position gesetzt, während die anderen Bits unverändert bleiben. Nach der Initialisierung enthalten die Bitwörter damit dieselben Informationen wie die vorher verwendeten booleschen Vektoren, wie in Abschnitt 4.2 gezeigt.

Berechnung der R- und L-Mengen

Die Berechnung der R- und L-Mengen in der Funktion `combine_infos` folgt direkt den in Abschnitt 4.3 beschriebenen bitparallelen Operationen. Die dort hergeleiteten Masken, Verschiebungen sowie UND- und ODER-Verknüpfungen werden in der Implementierung direkt auf den beiden `u64`-Werten ausgeführt. Dadurch entfällt die in der ursprünglichen Version verwendete Schleife über alle Positionen des Musters.

Für den Match-Check werden zunächst die bereits bei der Berechnung der R-Menge bestimmten Kandidaten mit der entsprechenden Längenmaske verknüpft. Die Anzahl der dabei gesetzten Bits wird anschließend direkt mit `count_ones()` bestimmt. Sie zählt nur die Vorkommen, die neu über die Grenze zwischen den beiden Teilen entstehen. Die Vorkommen der beiden Kinder werden hinzuaddiert:

```
occ: left_info.occ + right_info.occ
     + (candidates & length_mask).count_ones() as u64,
```

Da jedes gesetzte Bit einem Vorkommen des Musters entspricht, können alle in einem Kombinationsschritt gefundenen Vorkommen gleichzeitig gezählt werden.

Umsetzung von Regeln mit k $\geq$ 2

Die Funktion `combine_infos` kombiniert weiterhin jeweils zwei Teile miteinander. Für die in Abschnitt 4.5 beschriebenen Regeln mit beliebig vielen Symbolen auf der rechten Seite wird diese Funktion deshalb schrittweise angewendet. Für eine Regel $X \to Y_1 \dots Y_k$ werden zunächst die Informationen von Y_1 als Zwischenergebnis `acc` übernommen. Anschließend werden die Informationen der übrigen Symbole $Y_2, \dots, Y_k$ nacheinander von links nach rechts mit `acc` kombiniert. Nach der letzten Kombination

enthält `acc` die vollständigen R- und L-Mengen, die Expansionslänge und die Anzahl der Mustervorkommen der Regel X. Dadurch kann dieselbe binäre Kombinationsfunktion auch für Regeln mit beliebig langen rechten Seiten verwendet werden. Die Informationen eines Nichtterminals werden immer noch freigegeben, nachdem alle Regeln verarbeitet wurden, die es verwenden.

Umsetzung von Early-Exit

Der in Abschnitt 5.1 beschriebene Early-Exit wird direkt nach jeder Kombination von zwei Regeln geprüft, sowohl innerhalb einer Regel als auch bei der Verarbeitung der Startregel:

```
if exist && acc.occ > 0 {
    return (acc.occ, true);
}
```

Sobald bei einer Existenzsuche ein Vorkommen gefunden wurde, wird die Berechnung damit direkt beendet. Bei der Zählsuche bleibt diese Bedingung deaktiviert und die Verarbeitung wird bis zum Ende fortgesetzt. Bei $m = 1$ wird die Suche beendet, sobald für eine Regel oder ein Symbol der Startregel ein Vorkommen gezählt wurde.

Laufzeit der 64-Bit-Variante

Die ursprüngliche boolesche Implementierung hat eine Laufzeit von $O(|G| \cdot m)$, da bei der Berechnung jeder Regelinformation alle m Positionen des Musters einzeln durchlaufen werden und $|G|$ die Größe der Grammatik, also die Summe der Längen aller rechten Regelseiten, ist [9]. Bei der 64-Bit-Variante werden diese Positionen dagegen gemeinsam durch eine konstante Anzahl von Bitoperationen verarbeitet. Da für $m \leq 64$ jede R- und L-Menge vollständig in ein Bitwort passt, benötigt eine Kombination zweier Regelinformationen konstante Zeit. Damit ergibt sich für die Verarbeitung der Grammatik eine Laufzeit von $O(|G|)$. Die Initialisierung der Terminale benötigt zusätzlich $O(m)$ Zeit, sodass die gesamte Laufzeit mit $O(|G| + m)$ angegeben werden kann. Da hier $m \leq 64$ gilt, ist dieser zusätzliche Anteil beschränkt. Bei aktiviertem Early-Exit kann die Suche früher enden. Im schlechtesten Fall, wenn kein Vorkommen gefunden wird, bleibt die Laufzeit jedoch unverändert.

6.2 Bitvektoren mit AWINT

Für Muster mit $m > 64$ werden die R- und L-Mengen mit dem Typ `Awi` (*engl. Arbitrary-width-integer*) aus der Bibliothek `AWINT` gespeichert. Ein `Awi` ist ein Bitvektor, der aus mehreren 64-Bit-Blöcken besteht und dessen Gesamtlänge an die Musterlänge angepasst werden kann [7]. Dadurch ist die bitparallele Suche nicht auf Muster mit höchstens 64 Zeichen beschränkt. Außerdem werden die benötigten logischen Operationen und Verschiebungen direkt von `AWINT` bereitgestellt. Ihre Funktionsweise wurde bereits in Abschnitt 4.4 beschrieben.

Wie bei der Variante mit 64-Bit-Wörtern werden für jedes Symbol die Länge der Expansion, die R- und L-Menge sowie die Anzahl der Mustervorkommen gespeichert. Dafür wird dieselbe Struktur `Info` verwendet, wobei die beiden Mengen nun als `Large` statt als `Small` gespeichert werden und damit beliebig viele Positionen aufnehmen können.

Da bei langen Mustern größere Bitvektoren verarbeitet werden, sollen während der Berechnung nicht für jedes Zwischenergebnis neue Vektoren angelegt werden. Dafür wird die Struktur `LargeScratch` verwendet. Sie stellt zwei temporäre Bitvektoren der Breite m bereit:

```
struct LargeScratch {
    candidates: Awi,
    tmp: Awi,
}
```

Dabei nimmt `candidates` die Kandidaten auf, die während der Kombination zweier Regelinformationen entstehen. `tmp` dient als temporärer Bitvektor für Zwischenberechnungen und Maskierungen. `LargeScratch` wird zu Beginn der Suche einmal angelegt und bei jedem Aufruf von `combine_infos` übergeben. Beide Vektoren werden dabei überschrieben, sodass für die Zwischenergebnisse kein neuer Speicher angelegt werden muss.

Laufzeit der AWINT-Variante

Bei der Variante mit `AWINT` hängt die Laufzeit einer Bitoperation wieder von der Musterlänge ab. Ein Bitvektor der Länge m besteht aus $\lceil m/64 \rceil$ 64-Bit-Blöcken, die bei den logischen Operationen und Verschiebungen verarbeitet werden müssen. Eine Kombination zweier Regelinformationen benötigt deshalb $O(\lceil m/64 \rceil)$ Zeit. Hinzu kommt die Initialisierung der Terminale mit $O(m)$. Für die gesamte Grammatik ergibt sich damit eine Laufzeit von $O(|G| \cdot \lceil m/64 \rceil + m)$. Die ursprüngliche boolesche Variante benötigt dagegen $O(|G| \cdot m)$, da jede Musterposition einzeln verarbeitet wird. Die bitparallele Verarbeitung reduziert damit die Anzahl der zu verarbeitenden Einheiten von m einzelnen Positionen auf $\lceil m/64 \rceil$ Bitblöcke. Die Wiederverwendung der Bitvektoren durch `LargeScratch` verändert die Laufzeitordnung nicht, reduziert jedoch die Anzahl neuer Speicherallokationen während der Berechnung.

7 Experimentelle Analyse

In diesem Abschnitt wird die bitparallele Implementierung experimentell untersucht und mit den anderen Suchvarianten verglichen. Zunächst werden in Abschnitt 7.1 die verwendete Testumgebung, die Datensätze und die ausgewählten Musterlängen beschrieben. Anschließend werden in Abschnitt 7.2 die gemessenen Laufzeiten ausgewertet. Dabei wird die bitparallele Implementierung zunächst mit der ursprünglichen booleschen Implementierung verglichen. Danach werden die Ergebnisse der Existenzsuche betrachtet und abschließend wird die Laufzeit der bitparallelen Implementierung mit `zgrep` verglichen.

7.1 Testumgebung und Datensätze

In diesem Abschnitt werden die Testumgebung und die verwendeten Datensätze beschrieben. Ziel der Experimente ist es, die Laufzeit der verschiedenen Suchvarianten mit verschiedenen Musterlängen zu untersuchen und die neue bitparallele Implementierung sowohl mit der ursprünglichen booleschen Implementierung als auch mit weiteren Suchvarianten zu vergleichen.

Alle Messungen wurden auf einem Rechner mit einem `Intel Core i7-10750H` Prozessor und `7.6 GB` RAM durchgeführt. Als Betriebssystem wurde `Ubuntu 24.04.3` LTS verwendet.Die Programme wurden im Release-Modus mit `rustc 1.94.1` kompiliert und ausgeführt. Alle Dateien, die *Grampress* testet, wurden mit dem Verfahren *Re-Pair* komprimiert.

Für die Experimente wurden zwei Dateien aus dem *Canterbury Corpus* verwendet [1]. Die erste Datei ist `bible.txt` mit einer Größe von etwa `4.0 MB`. Sie enthält englischen Text. Die zweite Datei ist `E.coli` mit einer Größe von etwa `4.6 MB`. Sie enthält eine DNA-Sequenz. Dadurch werden zwei verschiedene Arten von Daten getestet, die sich in ihrem Inhalt und in ihrer Struktur unterscheiden.

Die Laufzeit wurde für verschiedene Musterlängen gemessen. Folgende Musterlängen wurden verwendet:

$$m \in \{1, 2, 64, 65, 129, 256, 512, 1024, 2048, 4096\}$$

Die Längen wurden dabei so gewählt, dass verschiedene Fälle der Implementierung getestet werden. Ein Muster der Länge $m = 1$ stellt einen Sonderfall dar, da sein Vorkommen direkt gezählt werden kann. Die Musterlängen $m = 2$ und $m = 64$ werden mit einem einzelnen 64-Bit-Wort verarbeitet. Mit $m = 65$ wird der Übergang zur Verwendung von Bitvektoren betrachtet. Bei $m = 129$ werden dafür drei 64-Bit-Blöcke benötigt, wobei zwei Blöcke vollständig belegt sind und im dritten Block nur ein Bit verwendet wird. Die weiteren Musterlängen 256, 512, 1024, 2048 und 4096 werden verwendet, um zu sehen, wie sich die Laufzeit bei längeren Mustern verändert.

Die Suche wurde dabei über die Kommandozeile aufgerufen. Der folgende Aufruf sucht ein Muster in einer komprimierten Datei und gibt die Anzahl der Mustervorkommen aus:

```
./gpress --verbose --search "SUCHMUSTER" bible.txt.gps
./gpress         -vs        "SUCHMUSTER" bible.txt.gps
```

Die Option `-verbose` sorgt dafür, dass neben dem Ergebnis auch die Größe der Grammatik und die reine Suchzeit ausgegeben werden.

Die Existenzsuche wird über die zusätzliche Option `-exist` aufgerufen:

```
./gpress --exist --verbose --search "SUCHMUSTER" bible.txt.gps
./gpress            -evs            "SUCHMUSTER" bible.txt.gps
```

Dieser Aufruf gibt nur zurück, ob das Muster im Text vorkommt oder nicht. Alle anderen Informationen von `-verbose` werden auch hier ausgegeben. Nur die Anzahl der Mustervorkommen fehlt, weil sie hier nicht benötigt wird.

Für den Vergleich mit `zgrep` wurde die Datei `E.coli` zusätzlich mit `gzip` komprimiert. Da `zgrep` nur die Zeilen mit einem Treffer ausgibt, werden die einzelnen Vorkommen mit `-o` ausgegeben und anschließend mit `wc -l` gezählt:

```
zgrep -o "SUCHMUSTER" E.coli.gz | wc -l
```

7.2 Ergebnisse

In diesem Abschnitt werden die Testergebnisse gezeigt und verglichen. Zuerst wird die boolesche Implementierung mit der bitparallelen Implementierung verglichen. Danach werden die beiden Varianten der Existenzsuche betrachtet. Zum Schluss wird die bitparallele Implementierung mit `zgrep` verglichen.

Vergleich mit der booleschen Implementierung

Zunächst werden die Ergebnisse für die Datei `bible.txt` betrachtet. Die Grammatik der booleschen Implementierung enthält 81238 Regeln, während die Grammatik der bitparallelen Implementierung 81206 Regeln enthält. Der Unterschied liegt darin, dass die Regeln in der booleschen Implementierung eine binäre Struktur hatten, während sie in der bitparallelen Implementierung beliebig viele Symbole auf der rechten Seite enthalten können. Tabelle 7 zeigt die durchschnittlichen Laufzeiten für die verschiedenen Musterlängen. Jede Mustersuche wurde dabei mindestens zehnmal durchgeführt.

m	Boolesche Implementierung	Bitparallele Implementierung
1	≈ 0.72 ms	≈ 0.71 ms
2	≈ 41 ms	≈ 11.7 ms
64	≈ 99 ms	≈ 12.2 ms
65	≈ 99.4 ms	≈ 72 ms
129	≈ 148 ms	≈ 80 ms
256	≈ 229 ms	≈ 85 ms
512	≈ 390 ms	≈ 100 ms
1024	≈ 750 ms	≈ 128 ms
2048	≈ 1300 ms	≈ 171 ms
4096	≈ 2700 ms	≈ 260 ms

Tabelle 7: Durchschnittliche Laufzeiten der booleschen und der bitparallelen Implementierung für `bible.txt`.

Die Ergebnisse zeigen, dass die bitparallele Implementierung bei fast allen Musterlängen schneller als die boolesche Implementierung ist. Bei $m = 1$ gibt es einen kleinen Unterschied zwischen beiden Implementierungen, da das Muster hier direkt gezählt wird und keine R- und L-Mengen berechnet werden müssen. Bei $m = 2$ und $m = 64$ werden die R- und L-Mengen in einem einzelnen 64-Bit-Wort gespeichert. Dadurch können die Operationen direkt auf dem Bitwort ausgeführt werden.

Bei $m = 65$ wird ein Bitvektor verwendet, der aus zwei 64-Bit-Blöcken besteht. Trotzdem ist die bitparallele Implementierung weiterhin schneller als die boolesche Version. Bei $m = 129$ werden drei 64-Bit-Blöcke benötigt, wobei die Laufzeit weiterhin deutlich kleiner bleibt.

Die weiteren Ergebnisse zeigen, dass der Unterschied zwischen beiden Implementierungen größer wird, je länger das Muster ist.

Ergebnisse der Existenzsuche

Es hängt bei der Existenzsuche davon ab, wo das erste Vorkommen des Musters liegt. Ein Muster vom Anfang der unkomprimierten Datei kann nach wenigen Regeln gefunden werden, während ein Muster vom Ende fast alle Regeln benötigt. Aus diesem Grund wurde ein Muster aus der Mitte des Textes gewählt, damit ein mittlerer Fall betrachtet wird. Tabelle 8 zeigt die Laufzeiten für `bible.txt`.

m	Zählsuche	Existenzsuche
1	$\approx$ 0.71 ms	$\approx$ 0.44 ms
2	$\approx$ 11.7 ms	$\approx$ 6 ms
64	$\approx$ 12.2 ms	$\approx$ 7.5 ms
65	$\approx$ 72 ms	$\approx$ 40 ms
129	$\approx$ 80 ms	$\approx$ 48 ms
256	$\approx$ 85 ms	$\approx$ 53 ms
512	$\approx$ 100 ms	$\approx$ 60 ms
1024	$\approx$ 128 ms	$\approx$ 81 ms
2048	$\approx$ 171 ms	$\approx$ 105 ms
4096	$\approx$ 260 ms	$\approx$ 150 ms

Tabelle 8: Durchschnittliche Laufzeiten der Zählsuche und der Existenzsuche für `bible.txt`. Das Muster stammt dabei jeweils aus der Mitte der unkomprimierten Datei.

Die Ergebnisse zeigen, dass die Existenzsuche bei allen Musterlängen schneller ist. Sie braucht etwa 40 Prozent weniger Zeit als die Zählsuche, und dieser Anteil bleibt sowohl bei kurzen als auch bei langen Mustern ungefähr gleich. Der Grund dafür liegt darin, dass der Early-Exit nur die Anzahl der verarbeiteten Regeln verringert. Der Aufwand für eine einzelne Regel bleibt dagegen gleich, weil beide Varianten dieselben R- und L-Mengen berechnen. Das Muster liegt in der Mitte des Textes, deshalb wird der Treffer ungefähr in der Mitte der Grammatik gefunden und die restlichen Regeln werden nicht mehr verarbeitet. Zwischen $m = 64$ und $m = 65$ steigt die Laufzeit auch bei der Existenzsuche deutlich an, weil beide Suchvarianten ab $m > 64$ mit Bitvektoren statt mit einem einzelnen Bitwort arbeiten.

Diese Testergebnisse gelten aber nur für den mittleren Fall. Wenn das Muster im Text gar nicht vorkommt, dann werden alle Regeln verarbeitet und beide Varianten brauchen gleich lange. Die Existenzsuche wird aber nie langsamer als die Zählsuche, ihr Vorteil hängt davon ab, an welcher Stelle das erste Vorkommen liegt.

Vergleich mit zgrep

Ein verbreitetes Werkzeug für die Suche in komprimierten Dateien ist `zgrep`. Es dekomprimiert den Text zuerst vollständig und sucht danach darin. Für den Vergleich mit der bitparalleln Implementierung wurde die Datei `E.coli` verwendet. Die beiden Suchalgorithmen zählen aber nicht dasselbe, weil `zgrep` weder überlappende Vorkommen noch Vorkommen über einen Zeilenumbruch hinweg findet. Für das Muster `aa` liefert `zgrep` z. B. 255 169 Treffer, während die bitparallele Implementierung 337 820 Vorkommen zählt. Aus diesem Grund wird nur die Laufzeit und nicht die Anzahl der gefundenen Vorkommen verglichen.

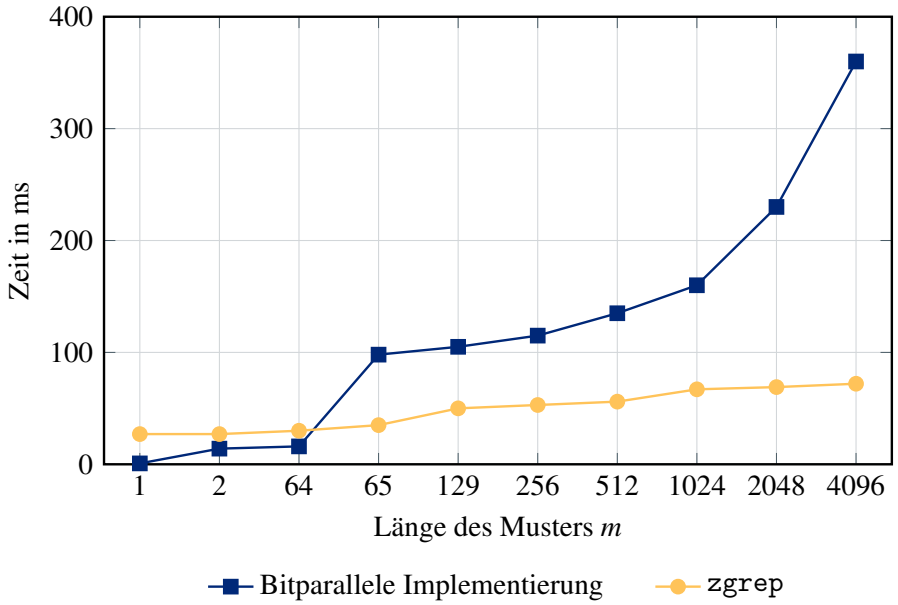

Abbildung 14: Durchschnittliche Laufzeit von der bitparallelen Implementierung und von `zgrep` für verschiedene Musterlängen (`E.coli`).

Die Ergebnisse zeigen, dass `zgrep` bei längeren Mustern schneller ist. Der Grund dafür ist, dass beide Verfahren unterschiedlich arbeiten. Da `zgrep` im unkomprimierten Text sucht, hängt die Laufzeit kaum von der Musterlänge ab und bleibt zwischen $m = 1024$ und $m = 4096$ fast gleich. Die bitparallele Implementierung sucht dagegen direkt im komprimierten Text. Ihre Laufzeit wächst mit der Musterlänge, denn je länger das Muster ist, desto mehr Blöcke werden für die Berechnung der R- und L-Mengen verwendet. Allerdings ist die bitparallele Implementierung bei $2 \leq m \leq 64$ schneller, weil dort ein einzelnes Bitwort ausreicht.

8 Fazit und Ausblick

In dieser Arbeit wurde der bestehende Suchalgorithmus von Grampress für die Suche nach Mustern in grammatikkomprimierten Daten weiterentwickelt. Ein zentrales Ziel war es, die Effizienz der Suche zu verbessern, indem die bisherige Verarbeitung der R- und L-Mengen durch eine bitparallele Verarbeitung ersetzt wurde. Außerdem wurde untersucht, ob die Existenzsuche durch eine Kombination aus der bisherigen Bottom-Up-Verarbeitung und einer gerichteten Top-Down-Verarbeitung beschleunigt werden kann. Dabei reicht es aus festzustellen, ob das Muster im Text vorkommt, ohne alle Vorkommen zu zählen. Zusätzlich sollte die bisherige Beschränkung des Suchalgorithmus auf binäre rechte Regelseiten entfernt werden.

Für die bitparallele Verarbeitung wurde die Speicherung der R- und L-Mengen angepasst, während ihre Bedeutung für die Suche unverändert geblieben ist. Bei Mustern mit bis zu 64 Zeichen werden beide Mengen in einem 64-Bit-Wort gespeichert, sodass alle Positionen des Musters gleichzeitig verarbeitet werden können. Für längere Muster werden Bitvektoren verwendet, sodass die Suche auch für Muster mit mehr als 64 Zeichen möglich bleibt. Obwohl die Laufzeit mit der Musterlänge steigt, ist die bitparallele Implementierung in den Testergebnissen schneller als die bisherige boolesche Implementierung. Die Verarbeitung der Regeln wurde auch an die neue Struktur der komprimierten Dateien angepasst. Der Suchalgorithmus kombiniert Regeln mit mehreren Symbolen auf der rechten Seite genauso, wie die Startregel im bestehenden Suchalgorithmus verarbeitet wird.

Für die Existenzsuche wurde untersucht, ob eine Kombination aus Bottom-Up- und Top-Down-Verarbeitung die Suche beschleunigen kann. Dafür wurde getestet, ob der Tiefensuche-Algorithmus eine optimale Existenzsuche bietet. Die Testergebnisse zeigten jedoch, dass dieser Hybrid-Suchalgorithmus in den meisten Fällen langsamer als der aktuelle Bottom-Up-Algorithmus ist. Aus diesem Grund wurde die Existenzsuche mit einem Early-Exit umgesetzt. Sobald ein erstes Vorkommen des Musters gefunden wird, kann die Suche beendet werden.

Ein Vergleich mit `zgrep` zeigte, dass `zgrep` bei langen Mustern schneller ist, da seine Laufzeit kaum von der Musterlänge abhängt. Der Vorteil der bitparallelen Implementierung liegt jedoch darin, dass sie direkt auf den komprimierten Daten arbeitet und den Text nicht vollständig entpacken muss.

Für zukünftige Arbeiten kann der Suchalgorithmus an mehreren Stellen erweitert werden. Eine mögliche Erweiterung ist die Ausgabe der Positionen der gefundenen Vorkommen. Der aktuelle Suchalgorithmus kann bestimmen, ob ein Muster vorkommt und wie oft es vorkommt, gibt aber keine Positionen im ursprünglichen Text zurück. In Zukunft könnte deshalb zusätzlich die Position eines einzelnen Vorkommens oder die Positionen aller Vorkommen bestimmt werden. Die Ausgabe könnte sich dabei z. B. an `grep` orientieren und neben der Position auch die jeweilige Zeile zurückgeben.

Eine weitere mögliche Erweiterung betrifft die Suchmuster selbst. Bisher wird nach einem festen String gesucht. In Zukunft könnte die Suche um Mechanismen regulärer Ausdrücke erweitert werden. Dazu gehören zum Beispiel Wildcards oder Wiederholungen. Damit könnte ein Suchmuster mehrere mögliche Strings beschreiben und die Suche wäre nicht mehr auf einen einzelnen festen String beschränkt.

Literatur

[1] R. Arnold and T. Bell. The canterbury corpus, 1997. Zugriff am: 24.August.2026. URL: https://corpus.canterbury.ac.nz/descriptions/

[2] R. Baeza-Yates and G. H. Gonnet. A new approach to text searching. Communications of the ACM, S. 74 - 82, 1992. Zugriff am: 14.Juli.2026. URL: https://dl.acm.org/doi/pdf/10.1145/135239.135243

[3] M. Elberfeld. Theoretische Informatik II. Vorlesungsskript an der Technischen Hochschule Mittelhessen im Wintersemester 2024/2025, Kapitel 9 (Durchlaufen von Graphen), S. 121 - 123, 2019.

[4] M. Ganardi. Compression by contracting straight-line programs. arXiv preprint arXiv:2107.00446, S. 5, 2021. Zugriff am: 16.August.2026. URL: https://arxiv.org/abs/2107.00446

[5] T. Graf. Implementierung und Evaluation von grammatikbasierter Kompression. Bachelorarbeit, Technische Hochschule Mittelhessen, Wetzlar, März 2026.

[6] W. H. Hesselink. Solutions of equations in languages. Formal Aspects of Computing, Bd. 22, Nr. 5, S. 538, 2010. Zugriff am: 05.August.2026. URL: https://link.springer.com/article/10.1007/s00165-009-0123-x

[7] A. Kutch. awint – Arbitrary Width Integers. Version 0.19.0. Rust crate. Zugriff am: 27.August.2026. URL: https://docs.rs/awint/latest/awint/

[8] M. Lohrey. Algorithmics on SLP-compressed strings: A survey. Groups-Complexity- Cryptology, S. 241-299, 2012. Zugriff am: 20.August.2026. URL: https://www.degruyterbrill.com/de/document/doi/10.1515/gcc-2012-0016/html

[9] B. Mulahusic. Entwurf und Implementierung eines Verfahrens zur Stringsuche in grammatikkomprimierten Daten. Bachelorarbeit, Technische Hochschule Mittelhessen, Gießen, März 2026.

[10] L. Najjar. Weiterentwicklung von Grammatik-Kodierung und-Kompression Bachelorarbeit, Technische Hochschule Mittelhessen, Hüttenberg, August 2026.

[11] G. Navarro. A guided tour to approximate string matching. ACM computing surveys (CSUR), Bd. 33, Nr. 1, S. 31-88, 2001. Zugriff am: 21.August.2026. URL: https://dl.acm.org/doi/abs/10.1145/375360.375365

[12] M. Rautiainen, V. Mäkinen and T. Marschall. Bit-parallel sequence-to-graph alignment. Bioinformatics, Bd. 35, Nr. 19, S. 3599-3607, 2019. Zugriff am: 27.August.2026. URL: https://academic.oup.com/bioinformatics/article/35/19/3599/5372677

[13] E.-H. Yang and J. C. Kieffer. Efficient Universal Lossless Data Compression Algorithms Based on a Greedy Sequential Grammar Transform, Part One: Without Context Models. IEEE Transactions on Information Theory, Bd. 46, Nr. 3, S. 755–777, 2000. Zugriff am: 26.August.2026. URL: https://ieeexplore.ieee.org/abstract/document/841161