GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Johann Wolfgang Goethe-Universität Frankfurt am Main

# Implementierung eines Pattern-Matching-Algorithmus von SLP-komprimierten Texten nach der Lifshits-Methode

**Diplomarbeit an der**
**Professur Künstliche Intelligenz und Softwaretechnologie**
**Prof. Dr. Manfred Schmidt-Schauß**

vorgelegt von
**Bianca Mihaela Borsan**

# Danksagung

Ich bedanke mich recht herzlich beim Herrn Prof. Dr. Manfred Schmidt-Schauß für das interessante Diplomarbeitsthema und für die ausgezeichnete Betreuung.

# Erklärung der Urheberschaft

Hiermit bestätige ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen Quellen oder Hilfsmittel als die in dieser Arbeit angegebenen verwendet habe.

Ort, Datum　　　　　　　　Vorname Nachname

# Zusammenfassung

Es ist ein klassisches Problem zu entscheiden, ob ein Muster (pattern) P in einem Text T vorkommt und es gibt viele Algorithmen, mit denen sich dieses Problem in Linearzeit lösen lässt. Ob das Muster als Teilwort in dem Text vorkommt, lässt sich in Linearzeit mit konstantem zusätzlichem Speicher entscheiden.

Um Speicherplatz einzusparen, liegen Wörter häufig in komprimierter Form vor. Der komprimierte Text muss schnell zugänglich sein und ohne explizite Dekomprimierung zu verarbeiten sein.

Dies führt uns zu der Frage, wie schwierig das Matching-Problem ist, wenn der Text $T$ und das Muster $P$ in komprimierter Form vorliegen. (Fully-Compressed Pattern Matching). Es wird angenommen, dass die beiden Texte in Form einer SLP Grammatik (Straight-Line Program) vorliegen. Eine SLP Grammatik hat die besondere Eigenschaft, dass Produktionen nur von der Form $X_i = X_l X_r$ sind, wobei die Indizes auf der rechten Seite kleiner als $i$ sind. Vor allem letztere Eigenschaft begünstigt die Anwendung der dynamischen Programmierung für verschiedenste Problemstellungen. Es ist bereits bekannt, dass sich Fully-Compressed Pattern Matching in Polynomialzeit lösen lässt.

In der vorliegenden Diplomarbeit wird der effiziente Algorithmus von Lifshits [4] vorgestellt und implementiert. Dieser löst das Problem in $O(n^2m)$ Zeit, wobei $n$ die Größe der SLP-Grammatik von $T$ und $m$ die Größe der SLP-Grammatik von $P$ ist. Hier ist anzumerken, die nicht-komprimierten Texte exponentiell in $m$ bzw. $n$ sein können. Somit ist dieser Ansatz asymptotisch deutlich besser als eine Lösung in Linearzeit auf dem dekomprimierten Text.

Der Kernteil des effizienten Algorithmus ist eine Datenstruktur, genannt AP-Tabelle, welche wichtige Informationen über Vorkommen von Teilstrings von $P$ in Teilstrings von $T$ enthält. Die Tabelle hat die Größe $(nm)$ und jeder Eintrag wird mit $O(1)$ Speicher kodiert. Die Einträge der Tabelle werden mit dynamischer Programmierung berechnet.

Da die Berechnung der AP-Tabelle recht komplex ist, ist deren Implementierung auch fehleranfällig. Zu Vergleichszwecken wird deshalb in dieser Arbeit auch eine naive, auf den dekomprimierten Strings arbeitende, Implementierung benutzt. Obwohl diese nur für Grammatiken mit bis zu 40 Produktionen anwendbar ist, dient diese zur Überprüfung der Einträge der AP-Tabelle. Weiterhin war es Hilfreich für das Debugging einige Hilfsmethoden sowohl effizient als auch ineffizient aber einfach zu implementieren.

Am Ende wurden $10^6$ Testläufe mit zufälligen SLP Grammatiken gemacht, bei denen jeweils die AP-Tabelle der effizienten mit der ineffizienten aber sicheren Implementierungen verglichen wurden. Die Ergebnisse beider Ansätze waren immer übereinstimmend.

Für einen experimentellen Vergleich der Laufzeit wurde eine Beispielgrammatik mit exponentieller Stringlänge aus [4] für eine allgemeine Anzahl an Variablen implementiert. Beide Verfahren wurden für $3, \ldots, 42$ Variablen getestet. Bis zu etwa 30 Variablen war die ineffiziente Variante unter einer Millisekunde, wohingegen die effiziente Variante etwa $5ms$ gebraucht hat pro Durchlauf. Bei mehr Variablen hat sich die Laufzeit der ineffizienten Variante fast immer verdoppelt und erreichte bei 42 Variablen mehrere Sekunden. Die effiziente Variante hat niemals mehr als $8ms$ gebraucht. Ein Testen mit mehr als 42 Variablen war nicht möglich aufgrund des hohen Speicherverbrauchs der ineffizienten Variante.

Zur Entscheidung des FCPM Problems wurde eine rekursive Funktion implementiert, welche die Einträge der Tabelle benutzt. Weiterhin wurden rekursive Funktionen implementiert, welche die Position des ersten und letzten Vorkommens und die Gesamtanzahl Vorkommen von $P$ in $T$ berechnen.

# Inhaltsverzeichnis


**Abbildungsverzeichnis** XI

**Tabellenverzeichnis** XIII

**1 Einführung** 1

**2 Begriffe und Definitionen** 5
- 2.1 Grammatik-basierte Kompression von Texten . . . 6
- 2.2 Implementierung der SLP Grammatik . . . 7
- 2.3 Präprozessing-Werte . . . 9

**3 Ein neuer Fully Compressed Pattern-Matching-Algorithmus nach Lifshits-Methode** 11
- 3.1 Pattern Matching via AP-Table . . . 12
- 3.2 Präprozessing . . . 14
- 3.3 Ein ineffizienter Ansatz . . . 15
- 3.4 Effiziente Berechnung . . . 17
  - 3.4.1 Zeilen und Spalten mit $|P_i| = 1$ bzw. $|T_j| = 1$ . . . 17
  - 3.4.2 Tabelleneinträge $(i, j)$ mit $|P_i| > 1$ und $|T_j| > 1$. . . . 18
  - 3.4.3 Realisierung der lokalen Suche . . . 24
- 3.5 FCMP anhand der AP-Tabelle . . . 28

**4 Testumgebung und Ergebnisse** 31
- 4.1 GUI Anwendung . . . 31
  - 4.1.1 package action . . . 33
  - 4.1.2 package converter . . . 34
  - 4.1.3 package worker . . . 35
  - 4.1.4 package gui . . . 36
- 4.2 Testläufe . . . 38
  - 4.2.1 GUI Tests . . . 39
  - 4.2.2 Grammatik Tests . . . 40
  - 4.2.3 Laufzeit Tests . . . 43

**Literaturverzeichnis** XV

# Abbildungsverzeichnis


2.1 Dekomprimierung des Textes aus SLP als Baum . . . . . . . . . . . . . . . . 6
2.2 Cut-Position für eine Nicht-Terminal Produktion . . . . . . . . . . . . . . . 10

3.1 Arithmetische Progression aller Vorkommen eines Patterns in einem Text 12
3.2 arithmetische Progression als Eintrag für die AP-Table . . . . . . . . . . . 13
3.3 Berechnung der Einträge für die AP-Tabelle . . . . . . . . . . . . . . . . . . 18
3.4 Positionen von $P_i$ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
3.5 Berechnung $P_r$ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
3.6 Berechnung Continental und Seaside Endings von $P_r$ . . . . . . . . . . . . 23
3.7 Local Search . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26

4.1 Controller . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
4.2 Manager . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
4.3 Action . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
4.4 Gui . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
4.5 Application Frame . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

# Tabellenverzeichnis

2.1 Array für die Produktionen . . . . . . . . . . . . . . . . . . . . . . . . . 8

# Kapitel 1

# Einführung

Wie kann man Daten speichern, so dass ein schneller Zugang zu den benötigten Informationen verschaffen und gleichzeitig der Speicherplatz maximal verringert wird?
Für die zuerst genannte Aufgabenstellung eigneten sich Datenstrukturen, die eine schnelle Verarbeitung der Anfragen ermöglichen, wie balancierte Bäume, Suffixarrays.
Für die Verringerung vom Platzbedarf werden Archivierungsalgorithmen angewendet. Im Folgenden werden wir einige Methoden zur Textkomprimierung betrachten, die verlustfrei (looseless) arbeiten. Das heisst, aus dem komprimierten Text kann der Originaltext vollständig rekonstruiert werden.

Jacob Ziv und Abraham Lempel veröffentlichten 1977 und 1978 neue Algorithmen (LZ77 [1], LZ78[2]) zur verlustfreien Datenkompression. Die Modellierung dieser Algorithmen ist die Tabellen-basierte Kompression.
Im Jahre 1983, entwickelte Terry Welch eine schnellere Variante des Lempel-Ziv Algorithmus, die heute so genannte LZW [11] (Lempel-Ziv-Welch)-Komprimierung. Zur Laufzeit wird aus der Eingabe Muster gewonnen, die, sofern noch nicht bekannt sind, ins Wörterbuch ("dictionary") eingetragen werden.
Der Eingabestring wird zwar schneller verarbeitet, "wobei Einschränkungen bei der Suche nach der Suche des Präfixes des Reststrings in bereits verarbeiteten String gemacht werden, z.B. LZ78 und LZW. Ein Vorteil von LZ77 ist, dass eine logarithmische Kompression von $O(\log n)$ möglich ist, während bei LZ78 und LZW die Kompression maximal bis $O(\sqrt{n})$ ist".[10]

Die Forschungen begannen mit Algorithmen der Suche nach expliziten angegebenen Substring in komprimierten Texten. Sehr schnell ging man von der Sicht der Kompressionsalgorithmen zum theoretischen Kompressionmodell der Straight-Line Programme (SLP) über.
SLP ist eine kontextfreie Grammatik, die genau einen Text/String generiert. Genauer beschreiben wir die SLPs in Kapitel 2.

Rytter[9] zeigte, dass sich ein komprimierter Text mithilfe LZ-Kompressionsalgorithmen (LZ-Familie, RLE, Wörterbuch-basiert) schnell und ohne bedeutenden Größenzuwachs in ein SLP vergleichbarer Größe überführen lässt, welches dasselbe Wort komprimiert.

Jeder LZ-komprimierte Text kann mit einem $O(\log N/n)$ zusätzlichen Speicherplatz in $\mathcal{O}(n \log(N/n))$ Zeit in einem als SLP komprimierten Text transformiert werden.

Nun stelt sich die Frage, welche Operationen können auf komprimierte Texte (ohne komplette Dekomprimierug) effizient ausgeführt werden?
Folgende drei fundamentale Fragenstellungen ergeben sich:

1. **Gleichheitsproblem**: prüfe, ob zwei als SLP-komprimierte Texte gleich sind
2. **Fully Pattern Matching Teilwortproblem**: prüfe, ob ein komprimierter Text Substring von einem anderen komprimierten Text ist. Falls ja, finde die Stelle des ersten Eintritts und die Anzahl aller Vorkommen.
3. **Hamming Distanz**: bestimmme die Hamming-Distanz zwischen zwei als SLP-komprimierten Texten gleicher Länge

Folgende Komplexität ist für voll-komprimiertes Pattern Matching Problem auf SLP-Strings bekannt:

Das Gleichheitsproblem wurde erstmals im Jahre 1994 in Zeit $O(n^4)$ gelöst, wo n die Summe der Größen der beiden SLPs ist.[8]
Gleichheit komprimierter Texte kann für den Vergleich verschiedener Archivkopien von einen und denselben System angewendet werden.

Fully Pattern Matching Teilwortproblem erhielt ein Jahr später, in 1995, eine erste polynomielle Lösung [6]. Myazaki, Shinohara und Takeda veröffentlichten 1997 für die Suche komprimierter Substrings in komprimierten Texten einen Algorithmus, der in $O(n^2 m^2)$ Zeit läuft, wobei n, m die Größen der komprimierten SLPs sind [7].
Es ist bereits bekannt, dass sich voll komprimiertes Pattern Matching (FCPM) in Polynomialzeit lösen lässt [3]. "Es ist sogar so, dass man eine (komprimierte) Darstellung aller Positionen eines solchen Matchings in polynomieller Zeit ausrechnen kann"[10].
FCPM kann bei Software Verifikation und bei der Suche nach Mediadaten (vorausgesetzt das Audio/Video Pattern auch in komprimierter Form vorliegend) verwendet werden.
Im Jahre 2000 wurde für einen Spezialfall der SLP-Klasse, das FCPM-Problem in Zeit O(nm) gelöst [5].

Über die Komplexität des komprimierten Hamming-Distanz Problem ist nichts bekannt. Als Anwendungsberiech dafür ist Bioinformatik bekannt, als auch die Suche nach Mediadaten.

In dieser Diplomarbeit soll die Antwort auf die Frage gegeben werden, wie schnell das voll komprimierte Pattern Matching (FCPM) durchgeführt werden kann und wie ein Algorithmus aussieht, der die bestmögliche Laufzeit erreicht.

Ein zentraler Punkt dieser Diplomarbeit ist die Idee, eine Grammatik-basierte Kompression bei der Lösung des voll-komprimierten Pattern Matching-Problems zu nutzen. Ein neuer Algorithmus für SLP-komprimierte Texte wird vorgestellt, der in Zeit $O(n^2m)$ läuft. n, m sind dabei die Größen der beiden komprimierten SLPs-Strings.

Als Grundlage für diese Diplomarbeit dient der Artikel "Processing Compressed Texts: A Tractability Border"[4].
Weitere Quellen, die zusätzlich herangezogen wurden, werden in den jeweiligen Kapiteln dieser Arbeit erwähnt.

Im Kapitel 2 wird das abtrakte SLP-Modell eingeführt und die SLP Grammatik-basierte Kompression vorgestellt bzw. implementiert. Ein naiver Algorithmus zur Lösung des Fully Pattern Matching Teilwortproblems wird vergleichend vorgestellt.

In Kapitel 3 wird ein neuer FCPM Algorithmus nach Lifshits Methode vorgestellt. Für jedes Textproblem auf einem als SLP-komprimierten String ergeben sich folgende zwei Fragen:

1. Gibt es ein polynoimieller Algorithmus dafür?
2. Wenn die Antwort "Ja" ist, was ist die genaue Komplexität des Problems?

Dabei erreicht der neue effiziente FCPM Algorithmus nach Lifshits Methode die untere Schranke von $O(n^2m)$ Zeit.

In Kapitel 4 wird die GUI-Anwendung vorgestellt. Eine ausführlichere Beschreibung der Klassen für Prä- und Post-Prozessing, sowie aller GUI-Komponente, wie: Menü, Panele, Ein-Ausgabegabefelder. Die Hauptkomponenten und deren Kommunikation über Managers werden auch mittels UML Klassendiagrammen dargestellt.

# Kapitel 2

# Begriffe und Definitionen

Grosse Text- bzw. Bilddaten werden häufig aufgrund hohem Platzbedarf komprimiert abgespeichert, mit Hilfe einer der LZ-Algorithmen oder einer kontextfreien Grammatik. Sucht man in solch einem komprimierten Text nach einem komprimierten Pattern, so ist es üblich, dass beide zuerst dekomprimiert werden, um danach einen der Pattern-Matching Algorithmen anzuwenden.
Aus Effizienzgründen wäre vorteilhafter die Suche, ob P ein Substring von T ist, in der komprimierten Form durchzuführen.

**Rytters Theorem**[9] hat zwei wichtige Eigenschaften:

- **SLP** hat fast die gleiche Größe wie der ursprüngliche kompriemierte Text
- **Transformation** ist schnell.

Anders gesagt:

1. Von einem **SLP**-String S kann in Polynomialzeit das LZ77(val(S)) errechnet werden.
2. Von einem **LZ77val(w)** kann in Polynomialzeit ein SLP-String S mit val(S) = w ausgerechnet werden

wobei val(S) den von S erzeugten String über $\Sigma$ bezeichnet.

Durch die einfache Beschreibungsart eignet sich die kontextfreie Grammatik bzw. das SLP-Modell besser zum Entwurf von Algorithmen.
Im Folgenden werden nötige Grundbegriffe nun formal eingeführt.

## 2.1 Grammatik-basierte Kompression von Texten

Ein straight-line program (SLP) ist eine kontextfreie Grammatik, die genau einen String (Wort) erzeugt.

**Definition**: Sei Σ ein Alphabet. Ein straight line program (SLP) G ist eine kontextfreie Grammatik zu Σ, wenn die Produktionsregeln von der Form sind:

- **Xi->a** wobei a $\epsilon$ Σ Terminal
- **Xi->XpXq** mit i > p, q ; wobei $X_1$,. . . ,$X_n$ Nichtterminale

Betrachten wir zum Beispiel, den String S = **abaababaabaab**, der aus folgender SLP erzeugt wurde:

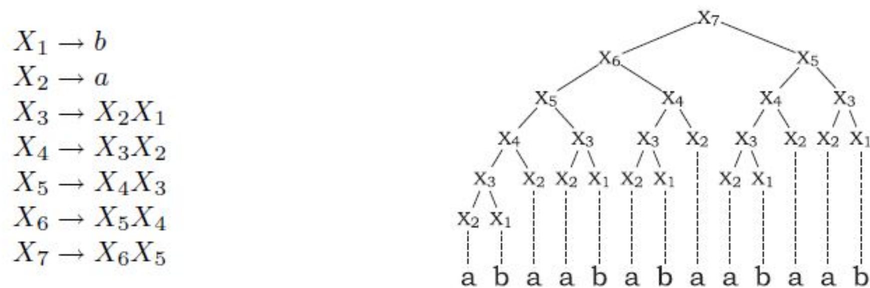


Abbildung 2.1: Dekomprimierung des Textes aus SLP als Baum

Der String, der von SLP erzeugt wird, ist eindeutig und wird vom letzten Nichterminal Xn abgeleitet. Die Anzahl der Variablen bestimmt die Größe der SLP Grammatik. Abhängig von der Wortlänge ergibt sich eine logarithmische Anzahl von Produktionen. Wenn die Größe der SLP Grammatik, n ist, kann damit einen String der Länge $2^n$ erzeugt werden. Das bedeutet, dass die SLPs exponentiell kleiner sind als die Strings, die sie erzeugen!

In 2003 zeigte Rytter[9] dass einen als LZ-komprimierten String S effizient in einem als SLP-komprimierten String überführt werden kann, der höchstens $O(/log|S|)$ grösser ist, als der ursprüngliche LZ-kompriemierte Text ist.
Die Straight-Line Programs (SLP) erlauben einen exponentiellen Faktor zwischen der SLP Grammatik Größe und der urspünglichen Wortlänge.

## 2.2 Implementierung der SLP Grammatik

Die Grammatik benötigt ein Array (1,...,n) für die Produktionen:
`Production[1],Production[2],...,Production[höchstIndex]`.

Dabei sind die Produktionen:

**Production[1],Production[2],...,Production[terminal]**
**Production[1],Production[2],...,Production[nonTerminal]**

Produktionen auf Terminale. bzw. auf Nicht-Terminale.

Der Konstruktor hat einen SLP-String als Eingabeparameter. Es wird jedes Element des übergebenen Strings gelesen und an das Semikolon ";" in einzelnen Produktionen gesplittet:

```
public Grammar(String s) {
String[] Prod = s.split("; ");
for (int i = 0; i < Prod.length; i++) {
String[] Prod2 = Prod[i].split("->");
int pnr = Integer.parseInt(Prod2[0].replace("X", ""));
```

Dabei ist hindex der gröÿte Index einer Variable und Productions[1,..,hindex] speichert die jeweiligen Produktionen:

```
if (i == 0) {
this.Productions = new Production[pnr + 1];
this.hindex = pnr;}
```

Wenn die Produktion in der Form $X_i$->$X_aX_b$ vorkommt, ist eine Nicht-Terminal Produktion und ihre linke und rechte Seite werden in einem Integer Array kodiert, mit i->a und i->b:

```
if (Prod2[1].contains("X")) {
String[] Prod3 = Prod2[1].split("X");
int a = Integer.parseInt(Prod3[1]);
int b = Integer.parseInt(Prod3[2]);
Productions[pnr] = new Production(a, b);}
```

| Prod | |
|---|---|
| $X_1$ | "b" |
| $X_2$ | "a" |
| $X_3$ | 2 - 1 |
| $X_4$ | 3 - 2 |
| $X_5$ | 4 - 3 |
| $X_6$ | 5 - 4 |
| $X_7$ | 6 - 5 |
| ... | ... |
| $X_n$ | (n-1) - (n-2) |

Tabelle 2.1: Array für die Produktionen

Die zwei Produktionen Arrays dienen zur Berechnung der Präprozessingwerte für die AP-Ttabelle und lassen sich wie folgt definieren:

- ein **Integer Array**, welcher die Nichtterminal-Produktionen der Form $X_i$->$X_aX_b$ kodiert, mit i>a und i>b
- ein **Char Array**, welcher die Terminal-Produktionen der Form $X_i$=a kodiert

Die Klasse **Production** beschreibt eine Produktion, wie folgt:

- **int links** Index der linken Variable
- **int rechts** Index der rechten Variable
- **char terminal** Falls Produktion $X_i$ ->"a"steht hier der Buchstabe a
- **boolean toTerminal** = Produktion auf Terminal mit

$$toTerminal = \begin{cases} true & \text{wenn Produktion auf Terminal}(X_i- > a) \\ false & \text{wenn Produktion auf Nicht-Terminal}(X_i- > X_leftX_right) \end{cases}$$

- **String text** = welchem Teilstring die Variable entspricht (Voraussetzung für die triviale Lösung)

Jeder Variable ihren String zuzuweisen ist ineffizient, da die Strings exponentiell wachsen.

## 2.3 Präprozessing-Werte

Wir führen zunächst eine Reihe notwendiger Begriffe ein:

- **Position** (engl. position) - ein Punkt zwischen zwei aufeinander folgenden Buchstaben Für einen Text $a_1, \ldots, a_n$ sind die Positionen 0, ..., n, wobei "0" vor dem ersten Buchstaben und "n" nach dem letztem Buchstaben liegt.
- **Vorkommen** (engl. occurrance) - steht sowohl für den entsprechenden Substring, als auch für seine Startposition
- **Berührung** (engl. touch) - ein Substring berührt eine vorgegebene Position, wenn der Substring an der vorgegebenen Position beginnt/endet oder wenn sich die Position innerhalb des Substrings befindet.

Um die eingeführten Begriffe zu erläutern wird ein Beispiel angegeben. Sei $P =$'aba' und $T =$'abaababa' mit $|P| = 3$ und $|T| = 8$. Die Positionen im String $T$ sind $0, \ldots, 8$.

Wir sagen, dass $P$ in $T$ an den Positionen $0, 3, 5$ vorkommt. Sei nun die Position 5 vorgegeben. Die Vorkommen von $P$, welche die Position 5 berühren starten bei 3 und 5. Für eine beliebige Position $i$ berührt ein Vorkommen von $P$ diese, genau dann wenn $P$ eine Startposition im Intervall $[i - |P|, i]$ hat, was äquivalent dazu ist, dass die Endposition im Intervall $[i, i + |P|]$ liegt.

Seien $P_1, \ldots, P_m$ und $T_1 \ldots T_n$ die Nicht-Terminale der SLPs, die $P$ bzw. $T$ erzeugen. Wir bezeichnen mit $P_i$ bzw. $T_j$ sowohl die Nicht-Terminale als auch den durch diese erzeugten String. Im Folgenden führen wir den Begriff der **Cut-berührenden** Vorkommen eines Strings $P_i$ in einem String $T_j$ ein. Dazu wird zuerst für jedes Nichtterminal die spezielle Cut-Position definiert.

Die **Cut-Position** ist:

$$Cut\!-\!Position = \begin{cases} Startposition, \text{wenn einbuchstabige Texte, Bsp: } X_i =' a' \\ Mergeposition, \text{wenn mehrbuchstabige Texte, Bsp: } X_i = X_r X_s \end{cases}$$

Eine Merging-Position würde für eine Nicht-Terminal Produktion der Form $X_i = X_r X_s$ zwischen den beiden Nicht-Terminalen sein und hat den Wert $|X_r|$ (Länge von $X_r$). Die Abbildung 2.2 verdeutlicht dies.

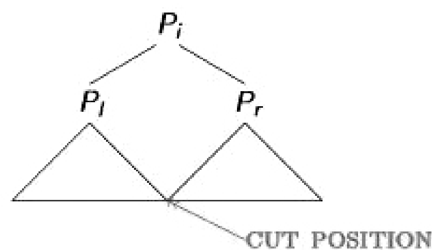


Abbildung 2.2: Cut-Position für eine Nicht-Terminal Produktion

Betrachten wir nun die vorherige Beispielgrammatik (Abb.2.1). Wir benutzten das Symbol '|' um die Merge-Position innerhalb eines Strings darzustellen. Die Cut-Position für $X_6$ =**abaab|aba** aus unserem Beispiel ist 5. Für $X_1$ ='a' und $X_2$ ='b' ist die Cut-Position jeweils 0.

Für zwei Variablen $P_i$ und $T_j$ sind die **cut-berührenden** Vorkommen von $P_i$ in $T_j$ alle Vorkommen, die auch den Cut von $T_j$ berühren.
In unserem Beispiel haben die cut-berührenden Vorkommen von $P_i = X_4$ ='aba' in $T_j = X_6$ ='abaab|aba' die Startpostionen 3 und 5. Beachte, dass $X_4$ auch an Position 0 vorkommt, jedoch nicht cut-berührend ist. Wenn wir im Folgenden sagen, dass $P_i$ an Position $\alpha$ in $T_j$ vorkommt, bezeichnen wir damit die Startposition $\alpha$ von $P_i$.

# Kapitel 3

# Ein neuer Fully Compressed Pattern-Matching-Algorithmus nach Lifshits-Methode

In diesem Kapitel behandeln wir Implementierungen für das FCPM Problem. Die Entscheidungsversion des Fully Pattern-Matching Problems (FCPM) lautet:

**Eingabe**: ein als SLP $P$ komprimiertes Muster und ein als SLP $T$ komprimierter Text

**Ausgabe**: Ja/Nein (kommt P als Teilwort in T vor?)

Andere Variationen des Fully Pattern-Matching Problems sind:

- die Suche nach dem ersten Vorkommen des Patterns
- das Zählen aller Vorkommen eines Patterns
- das linkeste Vorkommen des Patterns
- das rechteste Vorkommen des Patterns

Wir präsentieren eine Implementierung des neuen und effizienten Algorithmus für FCPM von Lifshits[4]. Zusätzlich verfolgen wir einen einfachen (naiven) Ansatz für FCPM, welcher die zwei Grammatiken zuerst auswertet und die notwendigen Informationen durch lineare Suche extrahiert. Der naive Ansatz hat den Nachteil, dass sowohl Laufzeit als auch Speicherbedarf exponentiell in der Größe der Grammatiken sein können. Dies disqualifiziert den naiven Ansatz für größere Grammatiken, jedoch dient dieser Ansatz zumindest bei kleinen Grammatiken zur Überprüfung der Korrektheit der effizienten (und komplizierteren) Variante.

Der neue und effiziente Algorithmus löst die zuletzt erwähnte Aufgabenstellung des Fully Pattern-Matching Problems in Zeit $O(n^2m)$, wobei $m$ die Größe der Grammatik des Patterns $P$ und $n$ die Größe der Grammatik des Textes $T$ ist. Der Kernteil des effizienten

Algorithmus besteht aus der Berechnung einer Hilfsdatenstruktur, der so genannten **AP-Tabelle**. Diese enthält alle für die ersten Variationen nötigen Informationen.

Wir beginnen die Beschreibung des neuen Algorithmus mit der Einführung der speziellen Datenstruktur (AP-Tabelle). Anschließend beschreiben wir, wie diese Datenstruktur mit Hilfe dynamischer Programmierung berechnet wird. Beim Ausfüllen der Tabelle wird eine Hilfsmethode, die so genannte Lokale Suche, benutzt. Diese Hilfsmethode wird als drittes vorgestellt.

**AP-Tabelle** Die AP-Tabelle enthält Informationen über cut-berührende Vorkommen von allen $P_i$'s in den $T_j$'s. Eine präzisere Definition der AP-Tabelle wird im folgenden Unterkapitel gegeben. Jedoch soll hier schon angemerkt werden, dass die cut-berührenden Vorkommen von $P_i$ in $T_j$ platzsparend (mit maximal drei ganzen Zahlen) codiert werden können. Weiterhin lassen sich aus den Einträgen der AP-Tabelle schnell das FCPM Problem lösen. Wenn wir wissen wollen ob $P$ in $T$ vorkommt müssen wir nur überprüfen ob es mindestens ein cut-berührendes Vorkommen von $P_m = P$ in $T_1, T_2, \ldots, T_n$ gibt.

## 3.1 Pattern Matching via AP-Table

Die AP-Tabelle beinhaltet Informationen für alle Paare $P_i$ und $T_j$, an welchen Positionen $P_i$ in $T_j$ cut-berührend auftritt. Lemma 1 besagt, dass jeder Eintrag mit Hilfe einer einzigen arithmetischen Progression kodiert werden kann. Eine arithmetische Progression ist eine Folge von Zahlen $a_1, \ldots a_k$ mit der Eigenschaft dass die Differenz zwischen zwei direkt aufeinander folgenden Zahlen konstant ist (Beispiel: $1, 3, 5, 7, 9, 11$).

**Lemma 1 ([4])**: Alle Vorkommen eines Patterns P in einem Text T, die eine fixe Position (wie cut position) berührt, bilden eine **eine einzige** arithmetische Progression.

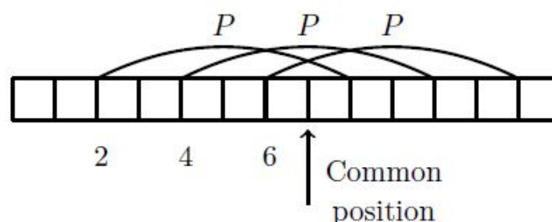


Abbildung 3.1: Arithmetische Progression aller Vorkommen eines Patterns in einem Text

Folglich definieren wir die AP-Tabelle (Tabelle der arithmetischen Progressionen).

Für jeden $1 \leq i \leq m, 1 \leq j \leq n$ ist der AP[i,j]-Eintrag eine arithmetische Progression aller Vorkommen von $P_i$ in $T_j$, welche die Cut-Position von $T_j$ berührt. Jede arithmetische Progression kann als Tripel dreier Integers modelliert werden:

**arpr = (start, step, count)**

Die Klasse **arpr** kodiert die arithmetische Progression und ist darauf zugeschnitten, als Eintrag für die AP-Table zu dienen. Eine nichtleere Folge sieht folgendermaßen aus:

$(start, start + step, start + 2 * step, ..., start + (count - 1) * step)$

- **start**: Starposition des Vorkommens
- **step**: Differenz / Delta
- **count**: Gesamtanzahl der Elemente / Vorkommen

Falls die arithmetische Progression leer ist bedeutet das, dass $P_i$ den Cut von $T_j$ nicht berührt. Wir kodieren die leere arithmetische Progression auf zwei Arten, je nachdem warum es keine cut-berührenden Vorkommen von $P_i$ in $T_j$ gibt. Es kann folgende zwei Gründe geben:

- **toosmall**: $P_i$ ist größer als $T_j$, also ist $T_j$ zu klein und $P_i$ kann kein Teilstring von $T_j$ sein ($\phi 1$)
- **empty**: $P_i$ ist kleiner als $T_j$, jedoch berührt $P_i$ den Cut von $T_j$ nicht ($\phi 2$)

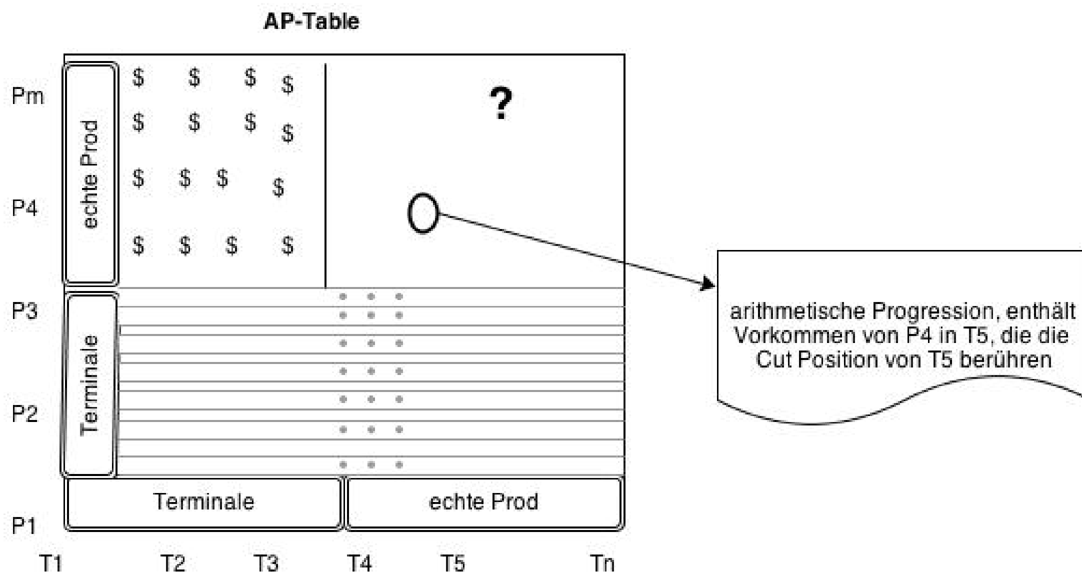


Abbildung 3.2: Arithmetische Progression als Eintrag für die AP-Table

Wir nehmen wieder als Beispiel die Grammatik aus Abb. 2.1 und wählen diese Grammatik sowohl für $T$ als auch für $P$.

Eintrag $(4, 6)$ der AP-Tabelle entspricht den cut-berührenden Vorkommen von $X_4$ ='aba' in $X_6$ =**abaab|aba** und wird durch die arithmetische Progression $(4, 2, 2)$ codiert (entspricht der Folge $4, 6$).

Der Eintrag $(4, 3)$ entspricht den Vorkommen von $X_4$ =aba in $X_3$ =**a|b**. Da $X_4$ aus drei Terminalen besteht und $X_3$ eine Länge von zwei hat, kann es keine Vorkommen von $X_4$ in $X_3$ geben, somit codieren wir den Eintrag mit $\phi 1$.

Anders ist es in dem Fall von Eintrag $(1, 5)$ mit $X_1$ =b und $X_5$ = **aba|ab**. Hier ist $|X_5| \geq |X_1|$, $X_1$ kommt in $X_5$ vor, aber nicht cut-berührend. Somit codieren wir den Eintrag $(1, 5)$ mit $\phi 1$.

Im Folgenden gehen wir darauf ein, wie die Einträge der AP-Tabelle Schritt für Schritt berechnet werden. Zur Berechnung der AP-Tabelle sind folgende Schritte notwendig:

1. Präprozessing der Grammatiken: Länge, Cut-Position, erster und letzter Buchstabe von $P_1, \dots, P_m$ und $T_1, \dots T_n$ werden berechnet

2. Berechnung aller Zeilen und Spalten der AP-Tabelle , mit $|P_i| = 1$ bzw. $|T_j| = 1$.

3. Für alle $P_i$ und $T_j$ mit $|P_i| > 1$ und $|T_j| > 1$ berechne die $AP[i, j]$-Einträge, vom kleinsten zum größten Pattern, vom kleinsten zum größten Text.

## 3.2 Präprozessing

Das Präprozessing ist eine Vorverarbeitung der jeweiligen Grammatiken und liefert notwendinge Informationen zur Berechnung der AP-Tabelle. Für jede Variable $P_i$ bzw. $T_j$ brauchen wir die Cut-Position sowie die Länge, den ersten und letzten Buchstaben des zu der Variablen entsprechenden Strings.

Es werden mit Hilfe von dynamischer Programmierung die Arrays length, cutposition, FL (first letter) und LL (lastletter) initialisiert und mit Hilfe der Dynamischen Programmierung berechnet. Es ist zu beachten, dass wir hierzu die Variablen nicht auswerten. Für Variablen $X_i$, mit $X_i = X_r X_s$ nutzen wir folgende rekursive Zusammenhänge:

- **Länge**: $\text{length}(X_r) = \text{length}(X_r) + \text{length}(X_s)$
- **linkester Buchstabe**: $\text{FL}(X_i) = \text{FL}(X_r)$
- **rechtester Buchstabe**: $\text{LL}(X_i) = \text{LL}(X_s)$

Falls $X_i$ von der Form $X_i = c$ ist, wobei $c$ ein Terminal ist, können wir die Einträge direkt berechnen:

- **Länge**: length($X_i$) =1
- **linkester Buchstabe**: FL($X_i$) =c
- **rechtester Buchstabe**: LL($X_i$) =c

Man beachte, dass durch die Eigenschaft der SLP gilt: Aus $X_i = X_r X_s$ folgt, dass $i > r$ und $i > s$. Somit reicht es aus, die Arrays nach aufsteigendem Index zu bearbeiten, denn bei der rekursiven Berechnung von $X_i$ sind die Einträge für $X_r$ und $X_s$ bereits berechnet. Die Methode Preprocess() der Klasse Grammatik sieht folgermaßen aus:

```
void Preprocess(){
     for (int i=1; i<=hindex; i++){
       Production p= Productions[i];
       if (p.toTerminal){
          p.length=1;
          p.cutpos=0;
          p.firstl=p.lastl=p.terminal;
       }
       else{
         p.length=Productions[p.left].length+
                     Productions[p.right].length;
         p.cutpos=Productions[p.left].length;
         p.firstl=Productions[p.left].firstl;
         p.lastl=Productions[p.right].lastl;
       }
    }
}
```

Dabei ist hindex der größte Index einer Variable und Productions[1,..,hindex] speichert die jeweiligen Produktionen.

## 3.3 Ein ineffizienter Ansatz

Zunächst behandeln wir einen einfachen Algorithmus zur Berechnung der Tabelleneinträge. Die triviale Methode `computableIneff()` ist nur benutzbar, falls die Variablen mit der Methode `evaluate()` berechnet wurden.

Die Methode **evaluate()** der Klasse Grammatik weist jeder Variable ihren String zu:

```
public void Evaluate(){
 for (int i = 1; i <= hindex; i++){
   Production p = Productions[i];
   if (p.toTerminal)
      p.text = new String("" + p.terminal);
   else {
      p.text = Productions[p.left].text + Productions[p.right].text;
```

Die Methode **computableIneff()** der AP-Table benutzt die Strings von $P_i$ bzw. $T_j$ (durch Evaluate() vorberechnet). Es benutzt weiterhin die (durch Präprozessing berechnete) Cut-Position des Strings $T_j$, die Länge des Patterns $P_i$ um die Startposition (startpos) des ersten möglichen Cut-berührenden Vorkommens von $P_i$ zu ermitteln.

```
String pi = P.Productions[i].text;
String tj = T.Productions[j].text;
int cutj = T.Productions[j].cutpos;
int leni = P.Productions[i].length;
int startpos = cutj - leni;
```

Das Pattern $P_i$ wird mit jedem Teilstring des Strings $T_j$, welcher an Position startpos anfängt verglichen. Variable startpos wird in jeder Iteration um eine Position nach rechts verschoben. Vorkommen von $P_i$ mit Startpositionen im Intervall $[cut - |P_i|, cut]$ werden in der arithmetischen Progression aufgenommen. Am Ende gilt die arithmetische Progression als Eintrag für die AP-Tabelle:

```
for (int pos = startpos; pos <= cutj; pos++){
  if (tj.substring(pos).startsWith(pi)){
   if (occurences==0){start=pos;}
   if (occurences==1){delta=pos-start;}
   occurences++;
  }
}
table[i][j] = new arpr(start, delta, occurences);
```

Die Methode `computetableIneff()` ist ineffizient, weil die jeder Variable ihren String zuweist und Strings exponentiell wachsen. Somit erreicht diese Variante eine Laufzeit und einen Speicherbedarf von $O(2^n + 2^m)$. Der Vorteil dieses Ansatzes liegt in seiner Einfachheit. Dieser wurde bei der Implementierung des nun folgenden effizienten, jedoch deutlich komplexeren, Algorithmus als Checkroutine verwendet. Sprich, liefern beide Algorithmen dasselbe Ergebnis?

## 3.4 Effiziente Berechnung

Als nächstes wird die effiziente Methode `computetableEff()` aus [4] vorgestellt. Diese benutzt nur die Präprozessing Methode der Grammatiken und benötigt keine Evaluierung der Variablen.

### 3.4.1 Zeilen und Spalten mit $|P_i| = 1$ bzw. $|T_j| = 1$

Als erster Schritt werden die Zeilen $(i, *)$ mit $|P_i| = 1$ und Spalten $(*, j)$ mit $|T_j| = 1$ berechnet, also Variablen mit einer Terminalproduktion. Dies leistet die Methode `computeRowsAndColumns()`, welche im folgenden vorgestellt wird. Beachte, dass bei dieser Berechnung maximal zwei cut-berührende Vorkommen von $P_i$ in $T_j$ gefunden werden können.
Man unterscheidet folgende Fälle:

**1.Fall $|P_i|$=1, $|T_j|$=1** bzw. beide Terminal Produktionen.

Wenn es sich um das gleiche Terminal handelt, also $P_i = T_j$ wird die arithmetische Progression arpr $=(0, 0, 1)$ als Eintrag für die AP-Tabelle ermittelt. Andernfalls wird $\phi 2$ als AP-Tabelleneintrag ermittelt, d.h. $P_i$ berührt den Cut von $T_j$ nicht.

```
if (Tj.terminal == Pi.terminal)
    table2[i][j] = new arpr(0, 0, 1);
else table2[i][j] = new arpr(2);
```

**2.Fall $|P_i|$=1, $|T_j|$>1** bzw. $T_j$=$T_rT_s$ ist eine Nicht-Terminal Produktion.

Da das Pattern $P_i$ aus einem Terminal besteht kann es nur direkt links oder direkt rechts vom Cut von $T_j = T_rT_s$ Cut-berührend vorkommen. Dies kann man anhand zweier Vergleiche der vorberechneten Präprozessingwerte (last letter, first letter) durchführen. Das Pattern $P_i$ muss einmal mit dem letztem Buchstabe der linkesten Seite $T_r$ (last letter) und einmal mit dem ersten Buchstabe der rechten Seite $T_s$ (first letter) verglichen werden. Die Abbildung 3.3 verdeutlicht dies.

$AP[i, j] =$ {

**Endet Tr mit "c"?**

```
if (T.Productions[Tj.left].lastl == Pi.terminal)
  aux = new arpr(Tj.cutpos - 1, 0, 1);
```

**Startet Ts mit "c"?**

```
if (T.Productions[Tj.right].firstl == Pi.terminal){
 aux = new arpr(Tj.cutpos, 0, 1);
```

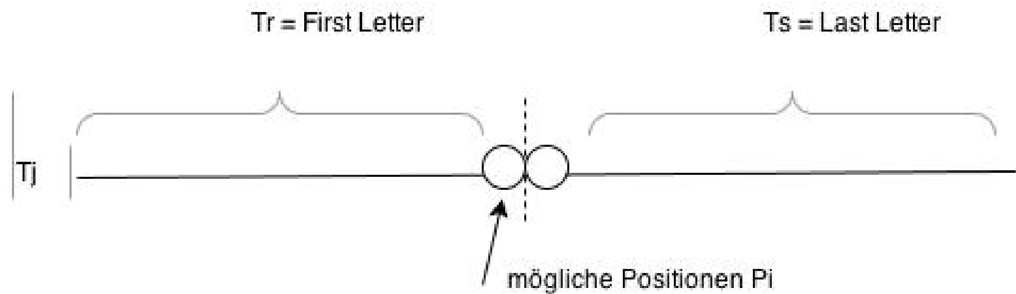


Abbildung 3.3: Berechnung der Einträge für die AP-Tabelle

**3. Fall** $|P_i|$**>1,** $|T_j|$**=1** bzw. $P_i$ ist eine Nicht-Terminal Produktion.

Pattern $P_i$ kann kein Teilstring von $T_j$ sein, somit auch keiner der den Cut berührt! Hierfür wird $\phi 1$ ermittelt, d.h. $T_j$ zu klein.

**Laufzeit** Sowohl der Vergleich beider einbuchstabiger Terminal Produktionen, als auch der Vergleich mit $T_j$ als mehrbuchstabige Nicht-Terminal Produktion ergeben eine arithmetische Progression `arpr` mit höchstens zwei Elementen. Folglich verbraucht die Berechnung jedem Tabelleneintrag konstante Zeit $O(1)$. Wir erhalten also als Gesamtlaufzeit der Methode `computeRowsAndColumns()` eine Laufzeit von $O(nm)$.

Alle anderen Einträge werden mit dynamischer Programmierung mit Hilfe der effizienten Funktion `computeEff(i,j)` berechnet.

### 3.4.2 Tabelleneinträge $(i, j)$ mit $|P_i| > 1$ und $|T_j| > 1$.

Nach der Befüllung aller einbuchstabigen $P_i$-, $T_j$- Zeilen und Spalten wird die AP-Tabelle in der lexikographischen Reihenfolge der Paaren (i,j) befüllt. Dieser Teil bildet den Hauptteil des Algorithmus aus [4].

Die effiziente Methode `computeEff(i,j)` berechnet die Tabelleneinträge mithilfe dynamischer Programmierung, wenn sowohl $P_i$ als auch $T_j$ Nichtterminale sind. Im folgenden Code ist P.terminal bzw. T.terminal der höchste Index einer Terminalproduktion.

```
for (int i = P.terminal + 1; i <= P.hindex; i++){
  for (int j = T.terminal + 1; j <= T.hindex; j++){
    this.computeEff(i,j);
```

**Ein einfacher Fall** Da beide Nichtterminal Produktionen sind, muss zuerst folgendes überprüft werden: Falls $|P_i|>|T_j|$, kann $P_i$ kein Teilstring von $T_j$ sein, somit auch keiner der den Cut schneidet! Somit haben wir $\phi 1$, d.h. $T_j$ zu klein, als AP-Tabelleneintrag. Die Werte $|P_i|$ und $|T_j|$ haben wir durch das Präprozessing gegeben.

**Zusammensetzung von $P_i$** Im folgenden gehen wir davon aus, dass $|P_i| \leq |T_j|$. Sei $P_i$ von der Form $P_i=P_rP_s$. Falls $|P_r|>=|P_s|$ ist, so ist `ComputeLS(i,j)` anzuwenden, wobei LS für LongShort steht. Andernfalls wird ComputeSL(i,j) angewendet, wobei SL für ShortLong steht. Es wird die Methode `ComputeLS(i,j)` vorgestellt. Die Methode `ComputeSL(i,j)` funktioniert analog. Diese Methode sucht also cut-berührende Vorkommen von $P_i$ in $T_j$, wobei $P_r$ (der linke Teil von $P_i$) mindestens so groß ist wie $P_s$ (der rechte Teil von $P_i$).

Zuerst folgende Überlegungen, wie wir solche Vorkommen von $P_i$ finden.

1. In welchen Intervallen kann $P_i$ in $T_j$ mit Cutberührung vorkommen? Abbildung 3.4 zeigt die Position von $P_i$, falls $P_i$ direkt auf dem Cut beginnt.

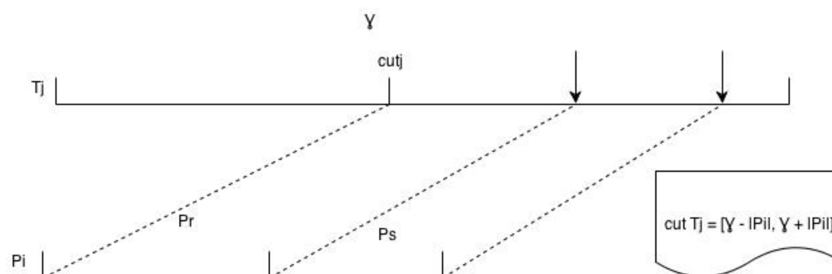


Abbildung 3.4: Positionen von $P_i$

2. Jedes Vorkommen von $P_i$ impliziert ein Vorkommen von $P_r$ gefolgt von $P_s$.

**Grobe Idee** Die grobe Idee um cut-berührende Vorkommen von $P_i$ zu finden ist folgende: Suche $P_r$ Vorkommen, welche Anfänge von cut-berührenden $P_i$ Vorkommen sein können. Sei nun $\gamma$ der Cut von $T_j$. Somit sind interessante Startadressen von $P_r$ im Intervall $[\gamma - |P_i|, \gamma]$. Anschließend werden gefundene Vorkommen von $P_r$ daraufhin untersucht ob diese direkt von $P_s$ gefolgt werden.

**LocalSearch** Doch wie suchen wir nach Vorkommen von $P_r$ und $P_s$? Wir wenden eine Hilfsmethode `LocalSearch(i,j,a,b)` an. Wir gehen erst später auf die Implementierung dieser Funktion ein, hier wird aber aufgezählt, was diese Methode leistet:

1. Sucht alle Vorkommen von $P_i$ in $T_j$, welche eine Startposition $\geq a$ haben und vor Position $b$ enden.

2. Einschränkung: Das Suchintervall ist höchstens $3 \cdot |P_i|$ lang ($b - a \leq 3|P_i|$).

3. Das Ergebnis besteht aus maximal zwei aufeinander folgenden arithmetischen Progressionen.

4. Die Methode benutzt zum ermitteln der Vorkommen die Einträge $(I, J)$ aus der AP-Tabelle mit der Eigenschaft $I \leq i$ und $J \leq j$.

Die Tatsache, dass sich das Ergebnis der LocalSearch Methode mit maximal zwei arithmetischen Progressionen darstellen lässt, folgt aus der Beschränkung der Intervallänge beim Suchen. Betrachtet man die zwei Positionen $a + (a - b)/3$ und $a + 2(a - b)/3$, so folgt aus $b - a \leq |P_i|$, dass jedes Vorkommen von $P_i$ innerhalb dieses Intervalls mindestens eine dieser Positionen berührt. Aufgrund von Lemma 1 erhalten wir, dass die Vorkommen mit maximal zwei arithmetischen Progressionen darstellbar sind.

Die eigentliche dynamische Programmierung findet in dieser Methode statt. Beachte, wenn der Tabelleneintrag $(i, j)$ berechnet wird, wird LocalSearch für die lokale Suche von $P_r$ und $P_s$ in $T_j$ angewendet. Da sowohl $r < i$ als auch $s < i$ wurden alle Tabelleneinträge die notwendig für LocalSearch sind bereits berechnet.

Weiterhin ist anzumerken, dass die Parameter für das Suchintervall sich nicht auf die Startpunkte beziehen.

**Suche von $P_r$** Man erinnere sich, dass wir annehmen, dass $P_i = P_r P_s$ und $P_r \geq P_s$ und $\gamma$ die Cut-Position von $T_j$ ist. Cut-berührende Vorkommen von $P_i$ implizieren, dass $P_r$ frühestens bei $\gamma - |P_i|$ beginnt und spätestens bei $\gamma + |P_r|$ endet. Die Abbildung 3.5 verdeutlicht dies. Die Methode `computeLS(int i, int j)` sucht zuerst Vorkommen von $P_r$ mit LocalSearch im Intervall $[\gamma - |P_i|, \gamma + |P_r|]$.

Bei der Anwendung von LocalSearch müssen wir darauf achten, dass das Suchintervall nicht zu groß ist. Es hat die Länge

$$|P_i| + |P_r| = 2|P_r| + |P_s| \leq 3|P_r|$$

und ist somit höchstens dreimal so groß wie der gesuchte String $P_r$.

Da es sich bei dem Ergebnis der LocalSearch nur um potenzielle Anfänge von $P_i$-Vorkommen handelt, muss noch überprüft werden ob diese jeweils direkt von einem $P_s$ gefolgt werden. Genauer gesagt, müssen wir die Enden der gefundenen $P_r$ Vorkommen daraufhin untersuchen, ob diese Startpunkte für $P_s$ Vorkommen sind.

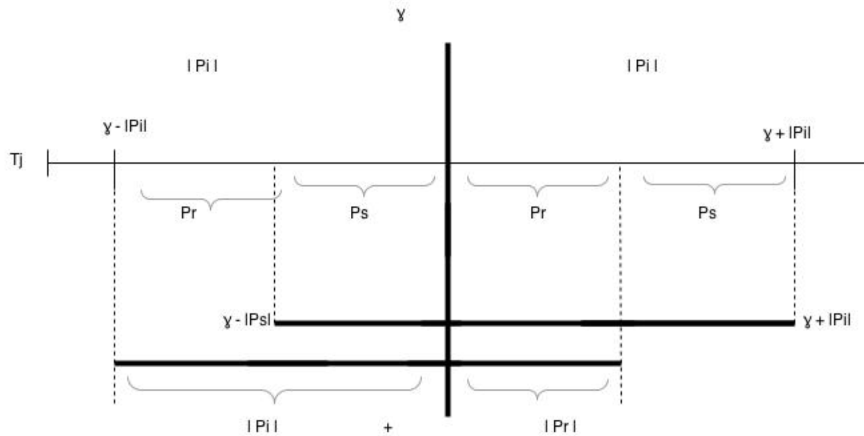


Abbildung 3.5: Berechnung $P_r$

Zuerst überlegen wir uns, dass interessante $P_s$ Vorkommen im Intervall $[\gamma-|P_s|, \gamma+|P_i|]$ beginnen und enden. Eine Anwendung von LocalSearch ist jedoch nicht ohne weiteres möglich, da die Länge des Suchintervalls folgende ist:

$$|P_s| + |P_i| = 2|P_s| + |P_r|$$

In diesem Fall gilt nicht, dass die Länge höchstens so groß ist wie $3|P_s|$, da $|P_s| \leq |P_r|$.

Es wird daher eine Methode `processLSarpr(...)` benutzt, auf deren Implementierung später eingegangen wird. Dieser Methode wird eine arithmetische Progression mit Startpunkten von $P_r$ Vorkommen übergeben und diese liefert eine arithmetische Progression mit den Enden der $P_r$ Vorkommen, die Startpunkte von $P_s$ Vorkommen darstellen. Es liefert uns also Startpunkte von $P_i$ (jedoch um $|P_r|$ versetzt).

Die Methode `processLSarpr(...)` wird auf beide arithmetischen Progressionen von $P_r$ Vorkommen angewendet und die Ergebnisse um $|P_r|$ nach links verschoben, damit wir Startpunkte von $P_i$ erhalten. Zum Schluß werden die (maximal) zwei arithmetischen Progressionen von $P_i$ vorkommen zu einer zusammengefasst und dienen als Eintrag für die AP-Tabelle an der Stelle $(i, j)$.

Eine vereinfachte Version des Codes für den Fall, dass die Suche nach $P_r$ zwei arithmetische Progressionen ergeben hat:

```
public void computeLS(int i, int j) {
DblArpr prArpr=LocalSearch(pr,j,cutj-Pi.length, cutj+Pr.length);
  arpr Res1=processLSarpr(prArpr.arpr1, pr, ps, i, j);
  Res1.shift(-Pr.length);
  arpr Res2= processLSarpr(prArpr.arpr2, pr, ps, i, j);
  Res2.shift(-Pr.length);}
  Res=Res1.merge(Res2);
```

```
table[i][j]=Res;
```

#### 3.4.2.1 Suche nach $P_s$

Die Methode `processLSarpr(arpr, pr, ps, i, j)` ist eine Hilfsmethode zur Berechnung der Tabelleneinträge $(i, j)$, welche die cut-berührenden Einträge von $P_i$ in $T_j$ repräsentiert, für den Fall $P_r \geq P_s$.

Für den anderen Fall steht die Methode `processSLarpr(arpr, pr, ps, i, j)` zur Verfügung und arbeitet analog.

Es wird eine arithmetische Progression übergeben, welche Vorkommen von $P_r$ repräsentiert. Das Ziel dieser Methode ist es, alle Vorkommen von $P_r$ zu eliminieren, die nicht direkt von $P_s$ gefolgt werden, und somit auch keine Vorkommen von $P_i = P_r P_s$ darstellen. Die $P_r$ Vorkommen sind aus dem Intervall $[\gamma - |P_i|, \gamma + |P_r|]$ (s. Abb 3.6) und wir müssen herausfinden, welche Vorkommen von $P_s$ sich im Intervall $[\gamma - |P_s|, \gamma + |P_i|]$ befinden. Die direkte Anwendung der lokalen Suche ist nicht erlaubt weil das Suchintervall nicht durch $3|P_s|$ beschränkt ist.

**Continental und Seaside** Eine wichtiger Kniff aus [4] ist die Unterteilung der $P_r$ Vorkommen in zwei Kategorien: *continental* und *seaside*. Dabei betrachten wir die *Enden* der $P_r$ Vorkommen, welche man mit einem Shift von $+|P_r|$ erreicht.

Sei $x$ die Endposition des letzten $P_r$-Vorkommens in der arpr. Alle Vorkommen deren Endposition einen Abstand von mindestens $|P_s|$ von $x$ haben sind *continental* Vorkommen. Alle Vorkommen mit Abstand kleiner als $|P_s|$ werden *seaside* genannt.

Die Methode `processSLarpr(...)` spaltet zuerst die arpr der Startpunkte von $P_r$ in die zwei arprs seaside und continental. Im folgenden ist ein Auszug aus dem Code zum Berechnen der continental Enden der Klasse arpr, für den Fall, dass die arpr nicht leer ist.

```
arpr continental(int prs,int pss)    if (this.count == 1)
      return new arpr(0, 0, 0);
   int border = this.lastEl() - pss;
   return new arpr(start + prs, step, (border - start) / step + 1);}
```

Diese Methode liefert eine leere arpr, falls die ganze Progression nur ein Vorkommen hat, denn dann muss dieses Seaside sein. Andernfalls wird zuerst die Position des letzten Elementes ermittelt und $|P_s|$ davon abgezogen. Der resultierende Wert entspricht der

Grenze, ab wann Vorkommen als Continental klassifiziert wird. Mit Hilfe der Startposition, Grenzposition und der Schrittweite der Progression wird die Anzahl der Continental Elemente. Man beachte, dass wir die Klassifizierung anhand der Startpunkte von $P_r$ durchgeführt haben. Um die Enden zu bekommen wird am Ende noch ein Shift der Startposition um $+|P_r|$ durchgeführt.

Als Beispiel diene die arpr=$(2, 2, 5)$, $|P_r| = 5$ und $|P_s| = 3$. Wir haben die Startpositionen von $P_r$ : $2, 4, 6, 8, 10$, es dementsprechend die Enden $7, 9, 11, 13, 15$. Das Anwenden der Methode continental() erhalten wir als Rückgabe $7, 9, 11$ bzw. die arpr=$(7, 2, 3)$. Bei der Berechnung des Ergebnisses, ist border= $15 - 3 = 12$.

Die Berechnung der Seaside Enden wird analog durchgeführt und liefert beim voherigen beispiel die arpr=$(13, 2, 15)$.

```
if (this.count == 1)
   return new arpr(this.start + prs, this.step, 1);
int border = this.lastEl() - ps;
int c = (border - start) / step + 1;
return new arpr(start + c * step + prs, step, count - c);
```

Beachte, dass beide Codefragmente nur Auszüge sind, und einige Speziallfälle zur besseren Lesbarkeit entfernt wurden.

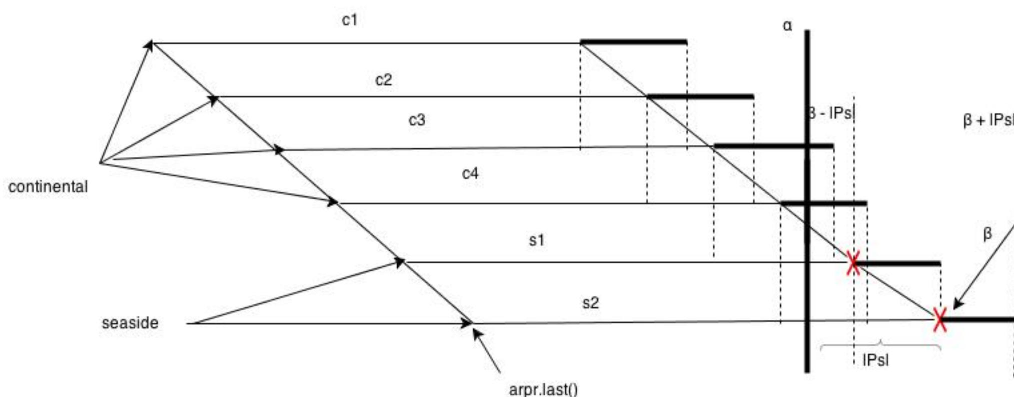


Abbildung 3.6: Berechnung Continental und Seaside Endings von $P_r$

**Bearbeitung der Continental Enden** Continental-Vorkommen haben den Vorteil, dass sie alle den gleichen Suffix der Länge $|P_s|$ haben (s. Abb. 3.6), somit reicht es aus, ein Continental-Ende daraufhin zu überprüfen, ob dieser direkt von $P_s$ gefolgt wird. Sei das letzte Continental-Ende an Position $y$.

Wir suchen mit `LocalSearch(...)` das Vorkommen von $P_s$ in $T_j$ im dem Suchintervall $[y, y+|Ps|]$.Haben wir $P_s$ an dieser Position gefunden, dann stellen alle Continental-Vorkommen auch Vorkommen von $P_i$ dar. Andernfalls verwerfen wir alle Continental-Vorkommen. Man beachte, dass die Länge des Suchintervalls die Länge des gesuchten Strings hat, und somit LocalSearch anwendbar ist.

**Bearbeitung der Seaside Enden** Die Seaside Enden liegen im Intervall $[x - |P_s|, x + |P_s|]$, wobei $x$ die Position des letzten Continental Endes ist. Somit können wir $P_s$ in diesem Intervall mit LocalSearch suchen, da das Suchintervall die Länge $2|P_s|$ hat. Obwohl LocalSearch im Allgemeinen zwei arpr Objekte zurückgibt, ist es in diesem fall nur eines. Grund dafür ist, dass jedes Vorkommen von $P_s$ die Position $x$ berührt.

Mit der arpr Methode `intersect(..)` werden nur noch die Seaside-Vorkommen von $P_r$ behalten, welche direkt von den mit `LocalSearch(...)` gefundenen $P_s$ Vorkommen gefolgt sind. Die Methode `intersect(...)` gibt den Schnitt zweier arpr Objekte, also alle Punkte, die in beiden arithmetischen Progressionen vorkommen.

**Zusammenfassen der Ergebnisse** Es wurden zuerst die Enden der $P_r$ Vorkommen in seaside und continental aufgesplittet. Die continental Vorkommen sind entweder alle Vorkommen von $P_i$ oder keines. Bei den Seaside Vorkommen wird eine Teilmenge gefiltert. Die zwei resultierenden arpr von $P_i$ Vorkommen werden am Ende vereinigt. Dies geht aufgrund von Lemma 1 und wird über die arpr Methode `merge()` gemacht. Wichtig hierbei ist, dass die $arpr_2$ eine echte Fortsetzung der ersten $arpr_1$ ist.

### 3.4.3 Realisierung der lokalen Suche

Die lokale Suche ist eine Hilfsmethode zum Berechnen der Einträge in der AP-Tabelle. Wir benutzen für den Namen der Methode die Abkürzung $LS(i, j, a, b)$. Dabei sind $i$ und $j$ Variablenindizes und $a, b$ Intervallgrenzen. Die Methode soll folgende Eigenschaften.

1. Sucht alle Vorkommen von $P_i$ in $T_j$, welche eine Startposition $\geq a$ haben und vor Position $b$ enden.
2. Einschränkung: Das Suchintervall ist höchstens $3 \cdot |P_i|$ lang ($b - a \leq 3|P_i|$).
3. Das Ergebnis besteht aus maximal zwei aufeinander folgenden arithmetischen Progressionen.
4. Die Methode benutzt zum ermitteln der Vorkommen die Einträge $(I, J)$ aus der AP-Tabelle mit der Eigenschaft $I \leq i$ und $J \leq j$.

Eigenschaft 3 ist nur möglich aufgrund von Eigenschaft 2, da es in diesem Fall zwei Positionen gibt, so dass jedes Vorkommen von $P_i$ in $T_j$ mindestens eine der beiden Positionen berührt.

**Ineffiziente Version** Zuerst wird wieder eine ineffiziente Version vorgestellt. Diese erfordert wie bei der ineffizienten Version der Berechnung der AP-Tabellen Einträge, dass die Gramatik ausgewertet wird, und somit exponentielle Laufzeit und Platz in Anspruch nehmen kann. Diese Version benutzt keine dynamische Programmierung, somit ist Punkt 4 nicht zu beachten. Der Zweck dieser ineffizienten Variante ist die Verifizierung der effizienten Variante, welche im Anschluß vorgestellt wird.

Es wird der $P_i$ in $T_j$ zuerst an der Anfangsposition $a$ gesucht. Anschließend wird in jeder Iteration die Anfangsposition um 1 erhöht bis diese den Wert $b - |P_i|$ erreicht.

Immer wenn ein Vorkommen von $P_i$ gefunden wird, wird diese Position in ein DblArpr Objekt *ret* hinzugefügt. Die DblArpr Klasse codiert maximal zwei arithmetische Progressionen. Mit der Methode $add(x)$ kann eine neue Zahl hinzugefügt werden.

Die $add()$ Operationen müssen jedoch mit Zahlen in aufsteigender Reihenfolge ausgeführt werden, wie es jedoch hier der Fall ist. Die $add()$ Methode versucht die erste arithmetische Progression zu erweitern. Falls dies nicht mehr möglich ist, startet diese eine zweite arithmetische Progression. Dies funktioniert nur falls das Suchintervall höchstens $3^*|P_i|$ groß ist, sonst sind eventuell mehr als zwei arithmetische Progressionen notwendig. Am Ende der Iterationen wird das DblArpr Objekt *ret* zurückgegeben.

```
public DblArpr LocalSearch2(int i, int j, int a, int b){
  DblArpr ret = new DblArpr();
  String pi = P.Productions[i].text;
  String tj = T.Productions[j].text;
  if (alpha < 0)
      alpha = 0;
  for (int pos = alpha; pos <= beta - pi.length; pos++){
    if (tj.substring(pos).startsWith(pi)){
    ret.add(pos);}}
  return add;
```

**Effiziente Methode** Für die Effiziente Variante haben wir die Strings von $P_i$ und $T_j$ nicht zur Verfügung. Es werden stattdessen die Einträge $(I, J)$ verwendet, mit $I \leq i$ und $J \leq j$.

Sei $T_j$ von der Form $T_j = T_xT_y$ und $\gamma$ der Cut von $T_j$. Die grobe Idee besteht aus folgendem rekursiven Zusammenhang, dass Vorkommen von $P_i$ in $T_j$ wie folgt unterteilt werden können (s. Abb. 3.7):

1. alle, die die Cutposition von $T_j$ berühren
2. alle zwischen $a$ und dem Cut von $T_j$ befinden sich in $T_x$
3. alle zwischen dem Cut von $T_j$ und $\beta$ befinden sich in $T_y$

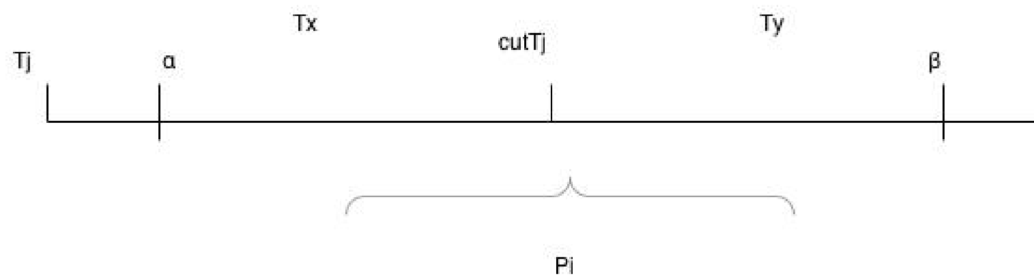


Abbildung 3.7: Local Search

Die Methode $LS(...)$ überprüft zuerst die Intervallgrenzen, ob diese zwischen $[0, \ldots, |T_j|]$ liegen. Falls nicht, werden diese angepasst.

```
DblArpr LS( int i, int j, int a, int b){
if (a<0) a=0;
if (b>T.Productions[j].length) b=T.Productions[j].length;
DblArpr ret= new DblArpr();
```

Als nächstes wird die rekursive Funktion crawling(...) aufgerufen, welche die gesuchten Vorkommen in einer Liste von arpr Objekten abspeichert.

```
ArrayList<arpr> arprL= new ArrayList<arpr>();
 crawling(i,j,alpha,beta, arprL, null, 0, false);
  for (int k=0; k<arprL.size(); k++){
   arpr aux=(arpr)arprL.get(k);}
   ret=new DblArpr(arprL);
   return ret;}
```

Der Parameter Arraylist $L$, arpr *parent* und boolean *left* geben an, wo die gefundenen Vorkommen als arpr in der Arrayliste L eingefügt werden. Weiterhin gibt der shift Parameter den additiven Term, der zu den Vorkommen dazuaddiert wird, vor dem Einfügen in die Liste.

Nachdem die crawling Funktion fertig ist, liegen alle gesuchten Vorkommen in Liste $L$. Diese Liste wird dann durch den speziellen Konstruktor von DblArpr(ArrayList <arpr> L) zu einem Objekt der Klasse DblArpr komprimiert und ist der Rückgabewert der Methode.

```
DblArpr( ArrayList<arpr> L ){
for(int i=0; i<L.size();i++){
arpr aux=(arpr) L.get(i);
 if (aux.empty) continue;
if (no==0){
arpr1= new arpr();
    arpr1.add(aux);
  no++;
continue;}
if (no==2){
 if(arpr2.add(aux)) continue;}}}
```

Bedingung ist, dass alle arpr Objekte aus der ArrayList aufeinanderfolgende Zahlenfolgen beschreiben. Weiterhin müssen sich alle Zahlen durch maximal zwei arpr-Objekte darstellen lassen.

**Crawling** Im Folgenden wird die Funktionsweise der rekursiven Funktion `crawling(...)` beschrieben:

1. Finde alle Vorkommen von $P_i$ in $T_j$, die den Cut von $T_j$ berühren (diese haben wir in der AP-Tabelle) und beschränke diese auf dem Intervall $[a, b]$ mit der `IntervalChop(..)` Funktion der Klasse `arpr`. Sei $T_j = T_l T_r$, und Cut-Position von $P_j = \gamma$. Alle anderen Vorkommen befinden sich links vom Cut, also in $P_l$ oder rechts davon in $P_r$.
2. rekursiver Aufruf der Funktion in $T_l$, wobei $P_i$ zwischen $[a, min(b, \gamma - 1)]$ gesucht wird. Da wir das linke Teilintervall suchen, setzen wir left=true, shift bleibt unverändert und als parent geben wir die arpr aus Punkt 1. an. Der rekursive Aufruf wird sein Ergebnis links von dem des Vateraufrufs abspeichern. Weiterhin sind die gefundenen Positionen $T_l$ identisch mit $T_j$.
3. rekursiver Aufruf der Funktion in $T_r$, wobei $P_i$ zwischen $[\min(a, \gamma + 1), b]$ gesucht wird. Der rekursive Aufruf wird sein Ergebnis links von dem des Vateraufrufs abspeichern. Da wir das rechte Teilintervall suchen, setzen wir left=false, $shift = shift + \gamma$ und als parent geben wir die arpr aus Punkt 1. an. Der Parameter $shift$ ist wichtig, für den rechten Aufruf, da dieser erlaubt Positionen aus $T_j$ in Positionen in $T_r$ umzurechnen. So entsprich etwa die Cut postion $\gamma$ in $T_j$ der Position 0 in $T_r$.

Die rekursiven Aufrufe in 2. und 3. finden nur dann statt, wenn das Suchintervall $\geq= |P_i|$ ist. Nach Aufruf der crawling(...) Funktion stehen alle gesuchten Vorkommen in der ArrayListe L vom Typ arpr, welche mehr als zwei Objekte des Typs arpr enthalten kann.

## 3.5 FCMP anhand der AP-Tabelle

Nachdem die AP-Tabelle berechnet wurde, können folgende Problemstellungen mit wenig Aufwand gelöst werden.

1. **Substring(i,j)**: Kommt $P_i$ in $T_j$ vor?
2. **FirstOcc(i,j)**: Position des ersten Vorkommens von $P_i$ in $T_j$ (Ausgabe -1, falls $P_i$ gar nicht vorkommt
3. **LastOcc(i,j)**: Position des letzten Vorkommens von $P_i$ in $T_j$ (Ausgabe -1, falls $P_i$ gar nicht vorkommt
4. **Occurences(i,j)**: Gibt an, wie oft $P_i$ in $T_j$ vorkommt

Wenn wir uns nur für $P$ und $T$ interessieren so müssen wir die Funktionen mit den Parametern $i = m$ und $j = n$ aufrufen. Somit erhalten wir zum Beispiel eine Antwort auf die Frage, ob $P$ in $T$ vorkommt, mit dem Aufruf **Substring**$(m, n)$.

**Substring Funktion** Diese Funktion nutzt folgenden rekursiven Zusammenhang wenn $P_i$ kleiner als $T_j$ ist und $T_j = T_l T_r$:

$P_i$ kommt in $T_j$ genau dann vor, wenn der Tabelleneintrag $(i, j)$ nichtleer ist oder wenn $P_i$ in $T_l$ oder in $T_r$ vorkommt. Falls $T_j$ aus einem Buchstaben besteht, so kommt $P_i$ in $T_j$ vor, falls der Eintrag $(i, j)$ nichtleer ist.

Der Code zu dieser Funktion ist:

```
boolean SS(int i, int j){

if (T.Productions[j].length<T.Productions[i].length)
      return false;
if (T.Productions[j].toTerminal)
      return !this.table2[i][j].empty;
if (this.table2[i][j].count>0)
      return true;
else
```

```
   return SS(i,T.Productions[j].left) ||
                     SS(i,T.Productions[j].right);
}
```

**FirstOccurence und LastOccurence** Es wird nur die Methode FirstOccurence (FO) beschrieben, da LastOccurence analog dazu funktioniert.

Falls $T_j$ aus einem Buchstaben besteht, so wird überprüft ob der Eintrag der AP-Tabelle an Position $(i, j)$ nicht leer ist. Je nachdem wird entweder 0 zurückgegeben oder $-1$, wobei letzteres dafür steht, dass es gar kein Vorkommen gibt.

Im Fall $T_j = T_l T_r$ wird erst rekursiv in $T_l$ gesucht. Falls etwas gefunden wurde, so wird das Ergebnis ausgegeben. Andernfalls wird das erste Element aus dem Tabelleneintrag $(i, j)$ zurückgegeben, falls dieses nichtleer ist. Wurde hier auch nichts gefunden, so wird noch in $T_r$ rekursiv gesucht. Falls dort etwas gefunden wurde, muss vor der Rückgabe $|T_l|$ aufaddiert werden. Beachte, dass Position 0 in $T_r$ der Position $|T_l|$ in $T_j$ entspricht.

Im Falle von LastOccurence ist die Reihenfolge umgekehrt, also von rechts nach links. Der code von FirstOccurennce lautet:

```
int FirstOcc(int i, int j){
     if (T.Productions[j].length<T.Productions[i].length)
        return -1;
     if (T.Productions[j].toTerminal) {
       if(!this.table2[i][j].empty) return table2[i][j].start;
       else return -1;}

   int leftocc=FirstOcc( i, T.Productions[j].left);
   if (leftocc!=-1) return leftocc;

   if(this.table2[i][j].count>0) return table2[i][j].start;

   int rightocc=FirstOcc( i, T.Productions[j].right);
   if (rightocc!=-1) return T.Productions[j].cutpos+rightocc;
   else return -1;
}
```

**Anzahl der Vorkommen** Die rekursive Funktion Occurences(i,j) zählt die Anzahl der Vorkommen von $P_i$ in $T_j$. Für $T_j = T_l T_r$ ist die Anzahl der Vorkommen in etwa die Anzahl der Vorkommen in $T_l$ und in $T_r$ und die Anzahl der Cut-berührenden Vorkommen. Man muss jedoch folgendes beachten: berührt $P_i$ den Cut ganz links, dann kommt es auch

im linken Teil von $T_j$ vor, und wir müssen dafür sorgen, dass dieses Vorkommen nicht doppelt gezählt wird. Das gleiche passiert falls $P_i$ den Cut ganz rechts berührt.

```
int Occurences(int i, int j){
if (T.Prod[j].length<T.Prod[i].length)
    return 0;
if (T.Prod[j].toTerminal) {
   if(!this.table[i][j].empty) {
      return 1;
   } else return 0; }
int cutocc=table2[i][j].count;
if (table2[i][j].start==T.Prod[j].cutpos-T.Prod[i].length)
cutocc--;
if (table2[i][j].lastEl()==T.Prod[j].cutpos) cutocc--;
return ( cutocc+ Occurences( i, T.Prod[j].left )
         +Occurences( i, T.Prod[j].right) ) ;
}
```

# Kapitel 4

# Testumgebung und Ergebnisse

Im Rahmen dieser Diplomarbeit erfolgte die Java-Implementierung eines Pattern-Matching-Algorithmus von SLP-komprimierten Texten nach der Lifshits-Methode.

## 4.1 GUI Anwendung

Um Programme zu starten wird eine entsprechende Properties Datei (*.properties) über die Menüauswahl, Converter -> Program -> program -> [dateiname].properties eingelesen. Diese Datei enthält den zu ladenden Klassen-/Programmnamen, sowie einige Laufzeitparameter, die das zugehörige ManagerPanel definieren, sowie die Animation der Statusleiste an- oder abschalten.
Der leichteren Zuordnung wegen werden Properties-Dateien entsprechend der Programmklasse benannt.
Wenn die java Datei zum Beispiel `AlgorithmusTask.java` heisst,sollte die properties Datei `Algorithmus.properties` heissen.

Wenn die Klasse im `converter` Package liegt, muss ihr Inhalt so aussehen:

**clazz** : borsan.grammar.converter.AlgorithmusTask - Klassenname mit Packageangabe

**manager** : Panel, welches die JTextArea enthält, aus der die Eingabe gelesen wird

**progress**: true

Der Parameter 'manager' steuert zu welchem Panel der Task gehört bzw. zu welchem Manager (converter oder crawler).
Der Parameter 'progress' steuert über die boolesche Werte 'true' oder 'false', ob der Progressbar animiert wird.

Der Unterschied zwischen den verschiedenen packages ist, dass die Klassen im converter package auf das obere Textfeld zugreifen und die Klassen aus dem crawler Textfeld auf das mittlere große Textfeld. zugreifen.

Converter-Klassen dienen normalerweise dem Pre-Processing, Crawler-Klassen hingegen dem Post-Processing.

1. **Input**: - Eingabe SLP-Strings aus oberer TextArea einlesen - komprimierte, durch Rautezeichen getrennte SLP-Strings parsen
2. **Process**: - Preprocessing für die Berechnung der AP-Table - Computing für die Berechnung der Terminal-Produktionen
3. **Output**: - Ergebnisse der Zwischenwerte vom Pre-Processing - Ergebnisse der Werte vom Post-Processing: Entscheidung Ja / Nein

Das Hauptanwendungsfenster(AppFrame.java) wird über die Klasse AppStart initialisiert und geöffnet. Der Anwender gibt nun die beiden, durch Raute separierten SLP-Strings in das obere Textfeld ein. Nach Auswahl eines entsprechenden Programms über das Menü(Converter -> Program -> program) wird dieses geladen und sofort ausgeführt. Sollte der Anwender noch keine SLP Strings eingegeben haben wird eine entsprechende Fehlermeldung ausgegeben

Einmal ausgewählt, kann man den gleichen Prozess immer wieder durch die Run-Taste wiederholen und eventuell die Eingaben in der Textarea verändern, bis man 'New' klickt oder ein anderes Program wählt.
Das Projekt (project) grammar lässt sich in fünf Pakete (package) unterteilen:

- **action** : enthällt alle Aktion-Klassen (Ereignisse, die durch Menüeinträge- oder Schaltflächen ausgelöst werden)
- **converter** : enthällt den Konverter (i.A. Klassen für das Prä-Prozessing)
- **gui** : enthällt alle GUI-Komponente (Menü, Panele, Ein-Ausgabegabefelder)
- **lib** : enthält die Hauptklassen für die Analyse in Form einer Bibliothek (library)
- **worker** : enthält die Superklasse(SwingWorkerTask.java) für alle Task Prozess-Klassen sowie einige Standardableitungen zum Laden und Speichern von Dateien

Zusätzlich zum Paket `grammar` enthält das Projekt `SLPLifshits` weitere Verzeichnisse:

- **briefing** : enthällt das Paper
- **converter** : enthällt die .properties Datei(en)
- **help** : enthällt die Html Viewer Datei
- **uml** : enthällt alle UML Klassendiagramme

Eine .bat Datei `starter` ist hinterlegt, um die Anwendung außerhalb der IDE zu starten.

### 4.1.1 package action

Die wichtigste Klasse `AbstractManagerAction` leitet von `javax.swing.AbstractAction` ab.
Über den eindeutigen String `referrerComponentKey` werden abgeleitete Klassen mit einem der Mananger Panels assoziiert. Die gleiche Aktion kann verschiedene Events, die mit verschiedenen Manager Panels interagieren, auslösen.

```
//key of ManagerPanel to lookup in component registry in GuiManager type
     private String referrerComponentKey;
//manager panel to this component key
    private IManagerPanel managerPanel;
```

Beispiel um mittels referrerComponentKey Zugriff auf ein ManagerPanel zu erhalten:

```
   managerPanel = GuiManager.getComponent(getReferrerComponentKey())
```

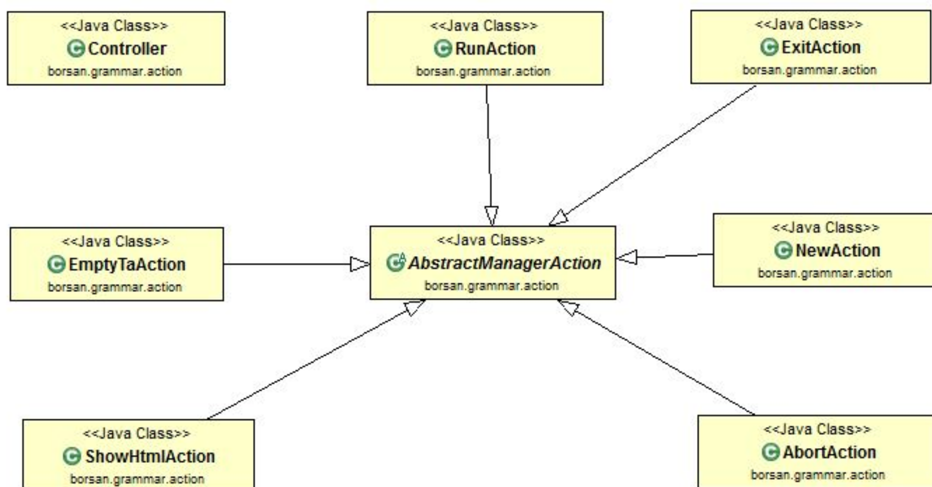


Abbildung 4.1: Controller

Folgende Aktionen werden für die JButton or JMenuItem Events verwendet:

- **NewAction** : Aktion wird ausgeführt beim Klick auf den 'New'-Knopf eines Managers (löscht alle zugehörigen Textfelder und entlädt das aktuelle Programm)
- **EmptyTaAction** : Aktion wird ausgeführt beim Klick auf den 'Empty'-Knopf eines Managers (löscht nur die zugehörigen Textfelder)
- **RunAction** : Aktion wird ausgeführt beim Klick auf den 'Run'-Knopf eines Managers (startet oder wiederholt ein aktuell geladenes Programm)
- **ExitAction** : Aktion wird ausgeführt, um die Applikation zu beenden

- **AbortAction** : Aktion wird ausgeführt beim Klick auf den 'Abort'-Knopf eines Managers (stoppt ein aktuell laufendes Programm)
- **ShowHtmlAction**: Aktion wird ausgeführt beim Klick auf den Menüeintag 'Help' (lädt die entsprechende Html-Hilfedatei und zeigt sie an)

Jede Action-Klasse besitzt eine componentKey, ID des Manager Panels oder des Menüeintrags, welche die Aktion ausgelöst hat.

### 4.1.2 package converter

Die wichtigste Klasse `SLPAnalyzerTask` leitet von SwingWorkerTask bzw. SwingWorker ab und läuft damit innerhalb eines eigenen, asynchronen Threads. Dies stellt sicher, dass die GUI während des Programmdurchlaufs nicht blockiert ('einfriert') und Aktionen wie, zum Beispiel 'Abort' weiterhin ausgeführt werden können.
Der Inhalt des über das ManagerPanel zugeordneten Textfelds wird vom Programm eingelesen. Sollte die Eingabe leer sein wird eine entsprechende Fehlermeldung ausgegeben.
Die Instanz des SLPAnalyzerTask wird am ManagerPanel gespeichert, damit man ihn ggf. mit geänderten Eingabeparametern, über die Run-Schaltfläche wiederholen kann, ohne das Programm erneut zu laden.

Alle abgeleiteten Task-Klassen überschreiben die Methode `public Void doInBackground()` der die eigentliche Arbeit, innerhalb eines separaten Threads stattfindet:

```
public Void doInBackground() { {
if (null!=getThrowable()) {;
if (getThrowable().getMessage().equals(IConst.STR_NULL)) {;
    AppFrame.showQuickInfo;
("\nAchtung\nEs sind keine Daten im Textfeld unter\n'"+;
  IConst.CONVERTER_TITLE+"' vorhanden.\n"+;
  JOptionPane.WARNING_MESSAGE);
```

Hiermit bekommt ein Task Zugriff auf ein ManagerPanel:

```
public SLPAnalyzerTask(String managerId,boolean blRunProgress,...)
   super(managerId, blRunProgress);
  // SLP-String von der GUI holen;
slpString = GuiManager.getComponent(managerId).getTextArea().getText
```

in diesem Fall auf den in der Textarea enthaltenen SLP-Eingabestring.

Hiermit speichert ein Task sich selbst an einem ManagerPanel womit das ManagerPanel dann Zugriff auf den Task erhält um ihn zum Beispiel abzubrechen oder neuzustarten.

```
 // Task in der Registry speichern damit man ihn wiederholen kann
GuiManager.getComponent(managerId).setWorkerTask(this);}
```

Der Task wird im ManagerPanel in dieser Variablen gespeichert:

```
// aktueller Task
private ISWCancel currentTask;
```

Alle Tasks implementieren ISWCancel.
Im Folgenden werden die Unterschiede der Hauptkomponenten erklärt und mögliche Anwendungen aufgezeigt.

### 4.1.3 package worker

SwingWorkerTask leitet von `javax.swing.SwingWorker` ab.

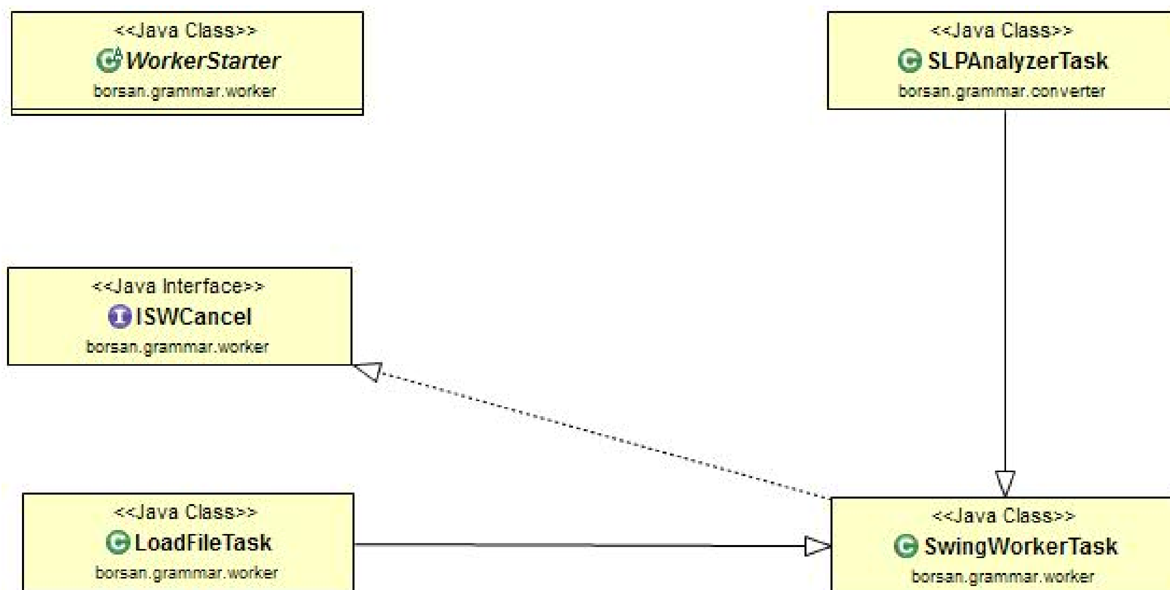


Abbildung 4.2: Manager

`SwingWorkerTask` ist die Superklasse für alle asynchronen Tasks der Anwendung: LoadFileTask, SwingWorkerTask, SLP AnalyzerTask. Damit werden alle Progamme als asynchrone Threads implementiert und laufen parallel. Diese Basisklasse steuert die Progressleiste, den Gui Update vor und nach dem Prozess, das Abbrechen von Tasks.

```
public class SwingWorkerTask extends SwingWorker<Void, Void>
   implements ISWCancel { {;
   // zugehöriger Manager;
    protected String managerId;
   // Listener für Threadstatus;
     protected PropertyChangeListener stateListener;
  // optionales Ergebnis;
     protected Object processResult;
```

Die eigentliche Hauptfunktion `public Void doInBackground()` wird von den ableitenden Klassen überschrieben:

```
 public Void doInBackground() {
   return null;}
```

### 4.1.4 package gui

`ManagerPanel` erbt von `javax.swing.JPanel` und implementiert IManagerPanel.

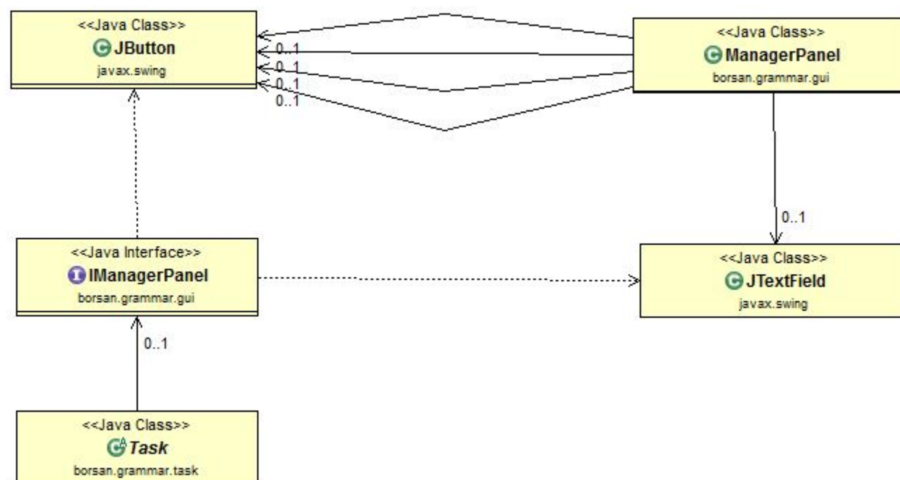


Abbildung 4.3: Action

Ein ManagerPanel stellt Eingabe- bzw. Ausgabe Komponenten zur Verfügung, auf die die asynchronen Tasks zugreifen können. Es beinhaltet zudem Schaltflächen durch die ein Programm gestartet oder zurückgesetzt werden kann.

```
// ID in Task Registry für aktuellen Prozess {
   private String currentProcessId;
// Manager ID in Registry
   private String managerId;
// aktueller Task
   private ISWCancel currentTask;
```

Am ManagerPanel wird der aktuell ausgewählte Task gespeichert um das gleiche Programm mit geänderten Eingabeparametern neu zu starten, ohne es wieder aus dem Menü auswählen zu müssen.

```
// HashMap für temporäre process data
   private static volatile HashMap<String, Object> store;
static { // nur einmal beim Laden der Klasse initialisieren
store = new HashMap<String, Object>();}
```

Die Klasse `AppStart` initialisiert AppFrame welche von `javax.swing.JFrame` ableitet und das Hauptanwendungsfenster zur Verfügung stellt.

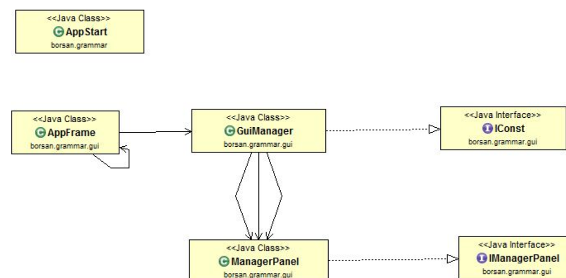


Abbildung 4.4: Gui

Um die GUI aufzubauen werden Methoden der GuiManager Klasse aufgerufen, die die entsprechenden Komponenten initialisieren. Die Klasse beinhaltet außerdem eine Registry(java.util.HashMap), in der alle ManagerPanel unter einer eindeutigen String-ID gespeichert werden.

Programme erhalten über diese Registry, mittels der String-ID, Zugriff auf entsprechende Komponenten wie Ein- oder Ausgabefelder, Dialoge für Dateiauswahl, User Information und User Eingaben anzuzeigen.

```
// HashMap für die Hauptkomponenten
 private static Map<String, Component> componentRegistry = null;
```

die als HashMap initialisiert wird und alle ManagerPannels abspeichert.

```
componentRegistry = new HashMap<String, Component>();
```

HashMap deshalb weil wir unter einer String-ID eine Komponente speichern und sie anhand dieser ID auch wieder holen wollen:

```
slpString = GuiManager.getComponent(managerId).getTextArea().getText
```

ManagerID ist die eindeutige String-ID.

## 4.2 Testläufe

Das Programm mit dem Namen textbfBorsanGui-Safe liegt auf der der Diplomarbeit beigelegten CD-ROM.
Es ist als eine auf den meisten Betriebssystemen[1] ausführbare Runnable jar-Datei vorhanden.

Nach der Ausführung des Programms öffnet sich die GUI-Anwendung:

Abbildung 4.5: Application Frame

[1]Zur Ausführung der jar-Datei muss die Java Runtime Environment 5 oder 6 installiert sein

### 4.2.1 GUI Tests

Im Menü wählt man den Converter-Menüeintrag aus um ein Programm für die Verarbeitung der Eingabe auszuführen oder den Help-Menüeintrag, um die Dokumentation des Programms anzuschauen.

Die Abbildung zeigt drei Panels.Aus dem oberen liest ein Programm die Eingabeparameter ein. Im mittleren Panel werden Log-Ausgaben des Programms angezeigt und im dritten, unteren Laufzeitfehler- und Ausnahmen.
In dem Converter-Panel gibt man die zwei SLP-Strings ein, getrennt durch eine Raute, dann aus Menüeintrag `Converter` das `program SLPAnalyzer.properties` auswählen und ausführen.
In dem Crawler-Panel werden die Zwischen- und die Endergebnisse ausgegeben.
In dem Process-Log Panel wird standard Logging eines Prozesses ausgegeben.

Folgende Möglichkeiten bieten sich für erneute Eingaben an:

- **Empty** - leert das zugehörige Textfeld für erneute Parametereingabe
- **Run** - den gleichen Prozess wiederholen, vorausgesetzt ein Programm wurde zuvor ausgewählt
- **New** - zugehörige Textfelder leeren und aktuelles Programm entladen
- **Abort** - bricht einen gerade laufenden Prozess/Task ab (JButton ist nur aktiv, während eine Programmdurchlaufs)

Wurde einmal ein Programm ausgewählt, kann man den gleichen Prozess immer wieder durch die 'Run'-Taste wiederholen und ggf. die Eingaben in der Textarea verändern, bis man die 'New'-Taste klickt.
Einmal die 'New'-Taste klickt, muss man erneut aus dem Menüeintrag Converter das program SLPAnalyzer.properties auswählen und ausführen. Alternativ ist es auch möglich direkt ein anderes Program über das Menü auszuwählen, ohne die 'New'- Taste zu klicken.
Da die Prozesse derzeit sehr kurz sind, kann man kaum/gar nicht sehen, dass Abort-Button aktiviert ist.Die Prozesse sind zu schnell beendet.
In das Process Log wird das standard Logging eines Prozesses geschrieben. Diese Informationen können über seine 'Empty'-Taste ebenfalls wieder gelöscht werden.

### 4.2.2 Grammatik Tests

Da die effiziente Implementierung der AP-Tabelle fehleranfällig ist, wurde diese mit der ineffizienten Version verglichen. Zum Testen wurden zwei Grammatiken mit 20 Variablen zufällig erstellt. Beide haben die Terminalproduktionen $X_1 =' a'$ und $X_2 =' b'$. Für $i > 2$ wurde die Produktion $X_i = X_r X_s$ erstellt, wobei $r$ und $s$ zufällig und gleichverteilt aus der Menge $\{1, \ldots, i-1\}$ gezogen. Es wurden $10^6$ Wiederholungen des Tests durchgeführt, bei denen keine Fehler auftraten.
Um die eingeführten Testläufe zu erläutern, wird ein konkretes Beispiel angegeben.

| **P:** |
| --- |
| X7–>X6X5 abaababaabaab |
| X6–>X5X4 abaababa |
| X5–>X4X3 abaab |
| X4–>X3X2 aba |
| X3–>X2X1 ab |
| X2–>a a |
| X1–>b b |

| **T:** |
| --- |
| X7–>X6X5 abaababaabaab |
| X6–>X5X4 abaababa |
| X5–>X4X3 abaab |
| X4–>X3X2 aba |
| X3–>X2X1 ab |
| X2–>a a |
| X1–>b b |

Die Präprozessing-Werte beider Grammatiken werden ermittelt:

**Berechnung der Präprozessing-Werte**

P:
Laenge: 13 CutPos: 8 FL, LL a b
Laenge: 8 CutPos: 5 FL, LL a a
Laenge: 5 CutPos: 3 FL, LL a b
Laenge: 3 CutPos: 2 FL, LL a a
Laenge: 2 CutPos: 1 FL, LL a b
Laenge: 1 CutPos: 0 FL, LL a a
Laenge: 1 CutPos: 0 FL, LL b b

T:
Laenge: 13 CutPos: 8 FL, LL a b
Laenge: 8 CutPos: 5 FL, LL a a
Laenge: 5 CutPos: 3 FL, LL a b
Laenge: 3 CutPos: 2 FL, LL a a
Laenge: 2 CutPos: 1 FL, LL a b
Laenge: 1 CutPos: 0 FL, LL a a
Laenge: 1 CutPos: 0 FL, LL b b

Die AP Table-Einträge für Terminal |Pi|=1 lassen sich wie folgt berechnen:

**Berechne AP-Einträge für Terminal |Pi|=1**

|Pi|=1
Berechne table[1,1] P1=b: T1=b table[1,1]=(0,-,1)
Berechne table[1,2] P1=b: T2=a table[1,2]=(N,-,-)
Berechne table[1,3] P1=b: T3=T2T1 T3.cut=1, T2.LL=a, T1.FL=b table[1,3]=(1,-,1)
Berechne table[1,4] P1=b: T4=T3T2 T4.cut=2, T3.LL=b, T2.FL=a table[1,4]=(1,-,1)
Berechne table[1,5] P1=b: T5=T4T3 T5.cut=3, T4.LL=a, T3.FL=a table[1,5]=(N,-,-)
Berechne table[1,6] P1=b: T6=T5T4 T6.cut=5, T5.LL=b, T4.FL=a table[1,6]=(4,-,1)
Berechne table[1,7] P1=b: T7=T6T5 T7.cut=8, T6.LL=a, T5.FL=a table[1,7]=(N,-,-)
Berechne table[2,1] P2=a: T1=b table[2,1]=(N,-,-)
Berechne table[2,2] P2=a: T2=a table[2,2]=(0,-,1)
Berechne table[2,3] P2=a: T3=T2T1 T3.cut=1, T2.LL=a, T1.FL=b table[2,3]=(0,-,1)
Berechne table[2,4] P2=a: T4=T3T2 T4.cut=2, T3.LL=b, T2.FL=a table[2,4]=(2,-,1)
Berechne table[2,5] P2=a: T5=T4T3 T5.cut=3, T4.LL=a, T3.FL=a table[2,5]=(2,1,2)
Berechne table[2,6] P2=a: T6=T5T4 T6.cut=5, T5.LL=b, T4.FL=a table[2,6]=(5,-,1)

Die AP Table-Einträge für Terminal |Tj|=1 lassen sich wie folgt berechnen:

**Berechne AP-Einträge für Terminal |Tj|=1**

|Tj|=1
Berechne table[1,1] T1=b: P1=b table[1,1]=(0,-,1)
Berechne table[2,1] T1=b: P2=a table[2,1]=(N,-,-)
Berechne table[3,1] T1=b: |P3|>1 table[3,1]=(S,-,-)
Berechne table[4,1] T1=b: |P4|>1 table[4,1]=(S,-,-)
Berechne table[5,1] T1=b: |P5|>1 table[5,1]=(S,-,-)
Berechne table[6,1] T1=b: |P6|>1 table[6,1]=(S,-,-)
Berechne table[7,1] T1=b: |P7|>1 table[7,1]=(S,-,-)
Berechne table[1,2] T2=a: P1=b table[1,2]=(N,-,-)
Berechne table[2,2] T2=a: P2=a table[2,2]=(0,-,1)
Berechne table[3,2] T2=a: |P3|>1 table[3,2]=(S,-,-)
Berechne table[4,2] T2=a: |P4|>1 table[4,2]=(S,-,-)
Berechne table[5,2] T2=a: |P5|>1 table[5,2]=(S,-,-)
Berechne table[6,2] T2=a: |P6|>1 table[6,2]=(S,-,-)
Berechne table[7,2] T2=a: |P7|>1 table[7,2]=(S,-,-)

Am Ende haben wir folgende Einträge der AP-Tabelle:

**Effiziente Tabelle**

(0,-,1)(N,-,-)(1,-,1)(1,-,1)(N,-,-)(4,-,1)(N,-,-)
(N,-,-)(0,-,1)(0,-,1)(2,-,1)(2,1,2)(5,-,1)(7,1,2)
(S,-,-)(S,-,-)
(S,-,-)(S,-,-)
(S,-,-)(S,-,-)
(S,-,-)(S,-,-)
(S,-,-)(S,-,-)

### 4.2.3 Laufzeit Tests

Um einen Eindruck für die Laufzeit der effizienten Variante vs ineffiziente zu erhalten wurde die Grammatik aus [4] für eine beliebige Anzahl von Produktionen erweitert und implementiert. Diese hat die Terminalproduktionen $X_1 =' a'$ und $X_2 =' b'$ und für $i > 2$ gilt $X_i = X_{i-1}X_{i-2}$. Eine besondere Eigenschaft dieser Grammatik ist, dass die Länge der Strings exponentiell in der Anzahl der Variablen wächst.

Beide Verfahren wurden für $3, \ldots, 42$ Variablen getestet. Bis zu etwa 30 Variablen war die ineffiziente Variante unter einer Millisekunde, wohingegen die effiziente Variante etwa $5ms$ gebraucht hat pro Durchlauf. Bei mehr Variablen hat sich die Laufzeit der ineffizienten Variante um Faktor ca. 1.5 erhöht und erreichte bei 42 Variablen mehrere Sekunden. Die effiziente Variante hat niemals mehr als $8ms$ gebraucht. Ein Testen mit mehr als 42 Variablen war nicht möglich aufgrund des hohen Speicherverbrauchs der ineffizienten Variante.

Die Zeitmessungen für 31 bis 42 Variablen sind in folgender Tabelle aufgelistet:

| **Anzahl Prod** | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Eff. Var. in ms** | 6 | 5 | 6 | 6 | 8 | 8 | 6 | 7 | 6 | 6 | 6 | 8; |
| **Ineff. Var. in ms** | 5 | 12 | 26 | 41 | 66 | 127 | 225 | 342 | 603 | 1021 | 1422 | 2413 |

# Literaturverzeichnis


[1] A. Lempel J. Ziv. *A Universal Algorithm for Sequential Data Compression.* IEEE Transactions on Information Theory, 1977. pp. 337-343.

[2] A. Lempel J. Ziv. *Compression of Individual Sequences Via Variable-Rate Coding.* IEEE Transactions on Information Theory, 1978. pp. 530-536.

[3] W. Rytter M. Karpinski and A. Shinohara. *Pattern-matching for strings with short descriptions.* Verlag Springer, 1995. Process Combinatorial Pattern Matching, volume 637 of Lecture Notes in Computer Science.

[4] Yury Lifshits. *Processing Compressed Texts: A Tractability Border.* CPM, 2007. pp. 228-240.

[5] M. Takeda M. Hirao, A. Shinohara and S. Arikawa. *Fully compressed pattern matching algorithm for balanced straight-line programs.* IEEE Computer Society, 2000. In SPIRE'00.

[6] W. Rytter M. Karpinski and A. Shinohara. *Pattern-matching for strings with short descriptions.* Springer-Verlag, 1995. In CPM'95, LNCS 937.

[7] A. Shinohara M. Miyazaki and M. Takeda. *An improved pattern matching algorithm for strings in terms of straight line programs.* Springer-Verlag, 1997. In CPM'97, LNCS 1264.

[8] W. Plandowski. *Testing equivalence of morphisms on context-free languages.* Springer-Verlag, 1994. In ESA'94, LNCS 855.

[9] W. Rytter. *Application of Lempel-Ziv factorization to the approximation of grammar-based compression.* Theor. Comput. Sci., 2003. pp. 302(1-3):211-222.

[10] Manfred Schmidt-Schauß. *Automatische Deduktion.* http://www.ki.informatik.uni-frankfurt.de/lehre/SS2012/AD/skript/AD.pdf, Sommersemester 2012. Teil I Deduktionssysteme: Grundlagen.

[11] Terry A. Welch. *A Technique for High Performance Data Compression.* IEEE Computer vol. 17 no. 6, 1984. pp. 530-536.