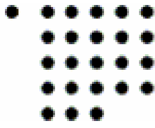


Fachhochschule Köln
University of Applied Sciences Cologne

Algorithmische Anwendungen
Wintersemester 2005/06

Projektdokumentation

String-Searching

KMP-Algorithmus
Boyer-Moore-Algorithmus

Gruppe: A-Lila
Hicham Settah
Dieter Galinowski

Stand: 25.01.2006

Inhalt:

1. Einleitung……………………………………………………………….3
2. Der Knuth-Morris-Pratt Algorithmus
   2.1 Funktionsweise…………………………………………………………...4
   2.2 Empirische Laufzeitanalyse……………………………………………7
3. Der Boyer-Moore Algorithmus
   3.1 Funktionsweise………………………………………………………….. 8
   3.2 Empirische Laufzeitanalyse……………………………………………10
4. Vorstellung der Implementierung beider Algorithmen in eine GUI………15
5. Quellenangaben…………………………………………………………17

1. Einleitung

In diesem Dokument sollen die beiden String-Suchalgorithmen von Knuth-Morris-Pratt(im weiteren Verlauf nur noch KMP genannt) und Boyer-Moore ein wenig genauer Betrachtet werden. Während der Boyer-Moore Algorithmus eigentlich als Standard bei den Suchalgorithmen bezeichnet werden kann, findet der KMP-Algorithmus in der Bioinformatik noch häufig Erwähnung. Neben einem kurzen Anschnitt zur Geschichte der Algorithmen folgt neben einer Erklärung der Funktionsweise anhand von Beispielen auch eine empirische Laufzeitanalyse, so weit uns das möglich war. Zum Abschluss folgt dann noch kurz eine Vorstellung der Java Anwendung, in der wir versucht haben die Algorithmen spielerisch leichter verständlich zu machen.

2. Knuth-Morris-Pratt Algorithmus

2.1 Funktionsweise

Jeder kennt den naiven Ansatz in einer Stringsuche. Das Ziel sollte nun also sein soweit wie möglich in dem zu suchenden String zu „springen“ ohne jedes einzelne Zeichen vergleichen zu müssen. Die Idee hinter dem KMP-Algorithmus ist nun die Informationen aus vorherigen Vergleichen wiederzuverwerten, d.h. man sollte möglichst keine bekannten Zeichen des Textes erneut vergleichen. Somit könnte man das zu Suchende Wort weiter verschieben.

Der Algorithmus funktioniert nun folgendermaßen:

1. Erzeuge eine Tabelle auf in der die eigentlichen Ränder der Präfixe abgespeichert wird woran sich dann die Sprungstellen ablesen lassen.
2. Vergleiche das zu suchende Wort mit dem Text
3. Sollte es bei den Vergleichen zu Unstimmigkeiten kommen wird aus der Tabelle die Anzahl der Stellen abgelesen um die das Suchwort verschoben wird.

Versuchen wir das ganze durch mit einem kleinen Beispiel durchzuspielen:

Text: ababababc
Suchwort: ababc

Als erstes wird die Tabelle erzeugt:

Suchwort:	E	a	b	a	b	c
Randlänge:	-1	0	0	1	2	

Wie kommt diese zu Stande?
ε entspricht hier dem leeren Wort und die Randlänge beträgt dazu laut Definition immer -1
ab ➔ keine besondere Auffälligkeit
aba ➔ eine Übereinstimmung. Somit haben wir ein Präfix mit der Größe 1.
abab ➔ 2 Übereinstimmungen. Hier haben wir somit ein größeres Präfix der Größe 2.
Das c spielt hier keine große Rolle mehr auch wenn es Teil des Präfixes wäre. Dazu später mehr.

Nachdem wir nun unsere Tabelle angelegt haben beginnen wir damit das Suchwort mit dem Text zu vergleichen:

Text	a	b	a	b	a	b	c
Suchwort:	a	b	a	b	c		
Position:	0	1	2	3	4		
- Randlänge	-1	0	0	1	2		
=Verschiebung					2		

Wie man sehen kann, stimmt der Text fast überein. An dem Buchstaben c scheitern wir jedoch. Nun passiert folgendes. Wir subtrahieren die Position(4) um die Randlänge des Präfix (2) welches dann eine Verschiebung von 2 ergibt. Hier sieht man auch warum die Randlänge von c keine Rolle gespielt hätte. Die ganze Tabelle verschiebt sich nämlich um eine Stelle nach Rechts und somit fällt der letzte Wert heraus.

Schauen wir uns nun den nächsten Schritt an:

Text	a	b	a	b	a	b	c
Suchwort:			a	b	a	b	c
Position:	0	1	2	3	4	5	6

In diesem Schritt wird die Randlängentabelle nicht benötigt, denn das gesuchte Wort wurde gefunden.

Natürlich ist die Einsparung gegenüber dem naiven Ansatz gering. Aber dies ist natürlich nur ein kleines Beispiel. In der Bioinformatik wird der Algorithmus häufig beim suchen von genetischen Übereinstimmungen verwendet. Wenn man sich die Eigenschaften einer DNS anschaut liegen die Vorteile auf der Hand. Eine DNS hat nur ein Alphabet von 4 Buchstaben, da ist die Wahrscheinlichkeit von größeren Präfixen schon recht hoch. Aber kommen wir nun zur Laufzeitanalyse.

2.2 Empirische Laufzeitanalyse

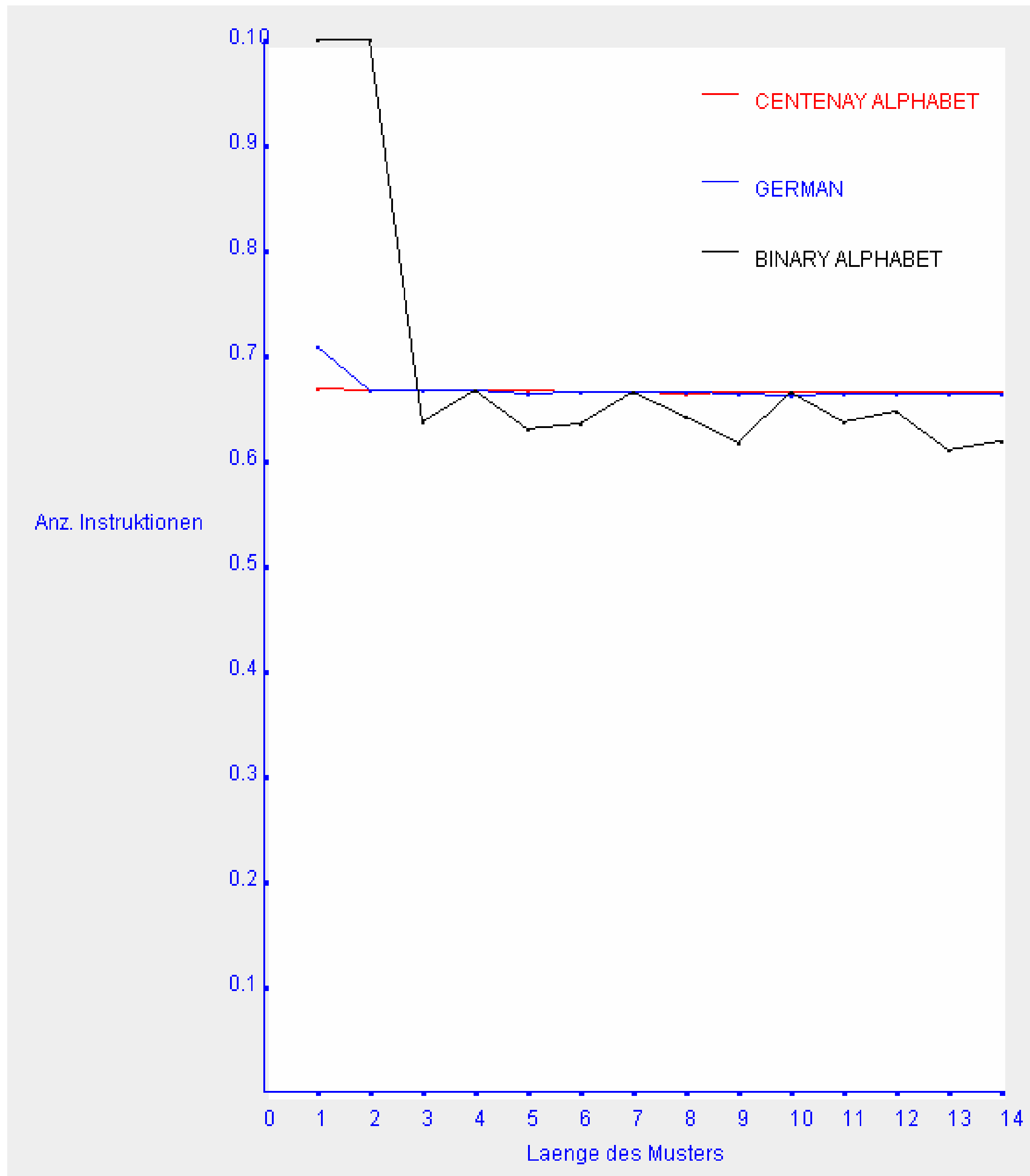
Worst Case

Verdeutlichen wir den Worst-Case an folgendem Beispiel:

Text	a	a	c	a	b	c	a	a	b
Suchwort:	a	a	b						
		a	a	b					
			a	a	b				
				a	a	b			
					a	a	b		
						a	a	b	
							a	a	b

Hier findet eine große Anzahl von Vergleichen statt. Wir haben in diesem Beispiel überhaupt keinen Vorteil aus unserer Tabelle ziehen können, da wir keine größere Anzahl von Stellen im Suchtext überspringen. Im Worst Case beträgt die Laufzeit O(n+m).

Analyse durchschnittlich zu erwartenden Verhaltens



Die Laufzeitanalyse gestaltet sich beim KMP recht schwierig. Die verschiedensten Quellen sprechen alle von einem Worst-Case oder auch Gesamtkomplexität von O(n+m).
Das interessante an diesem Algorithmus ist, das die Größe des Alphabets keine Rolle spielt. Die Laufzeit ist fast identisch, wie die Grafik bis auf einen den Anfangs“schwenker“ beim binären Alphabet.

3.1 Der Boyer-Moore Algorithmus

Die Hauptidee die hinter dem Boyer-Moore Algorithmus bezieht sich darauf das Suchwort nicht von Links nach Rechts mit dem Text zu vergleichen sondern von Rechts nach Links, man beginnt also mit dem letzten Buchstaben.
Dazu kommt die Idee das Suchwort bei einer Nichtübereinstimmung bis zu der Position zu verschieben an der der Buchstabe, als erstes von Rechts gesehen, im Suchwort vorkommt. Kommt er nicht vor kann um das gesamte Länge des Suchwortes verschoben werden.

Beispiel:
Wir suchen das Wort TEXT in dem Satz DURSUCHE DEN TEXT (Der Rechtschreibfehler wurde extra verursacht um Platz zu sparen).

Text:	**D**	**U**	**R**	**S**	**U**	**C**	**H**	**E**		**D**	**E**	**N**		**T**	**E**	**X**	**T**
Suchwort:	**T**	**E**	**X**	**T**													

Hier scheitert der Vergleich bereits am ersten Buchstaben T ≠ S. Außerdem kommt der Buchstabe S im Suchwort nicht vor somit kann um die gesamte Länge verschoben werden.

Text:	**D**	**U**	**R**	**S**	**U**	**C**	**H**	**E**		**D**	**E**	**N**		**T**	**E**	**X**	**T**
Suchwort:	**T**	**E**	**X**	**T**													
					T	**E**	**X**	**T**									

In diesem Fall ist T ≠ E jedoch kommt in diesem Fall E im Suchtext vor. Also verschieben wir das Suchwort bis zum E

Text:	**D**	**U**	**R**	**S**	**U**	**C**	**H**	**E**		**D**	**E**	**N**		**T**	**E**	**X**	**T**
Suchwort:	**T**	**E**	**X**	**T**													
					T	**E**	**X**	**T**									
							T	**E**	**X**	**T**							

T ≠ D. D kommt nicht im Suchwort vor also Komplettverschiebung

Text:	**D**	**U**	**R**	**S**	**U**	**C**	**H**	**E**		**D**	**E**	**N**		**T**	**E**	**X**	**T**
Suchwort:	**T**	**E**	**X**	**T**													
					T	**E**	**X**	**T**									
							T	**E**	**X**	**T**							
											T	**E**	**X**	**T**			

X ≠ (leer) somit wird das ganze Wort verschoben bis zur letzten Fundstelle.

Text:	**D**	**U**	**R**	**S**	**U**	**C**	**H**	**E**		**D**	**E**	**N**		**T**	**E**	**X**	**T**
Suchwort:	**T**	**E**	**X**	**T**													
					T	**E**	**X**	**T**									
							T	**E**	**X**	**T**							
											T	**E**	**X**	**T**			
														T	**E**	**X**	**T**

Der Suchbegriff wurde gefunden.

Damit nicht jedes Mal zum Überprüft werden muss ob ein Buchstabe im Suchwort vorkommt werden zwei Tabellen angelegt die dafür verwendet werden.

Die Last Tabelle speichert für jeden Buchstaben im Suchwort die sichere Verschiebedistanz. Das könnte im Extremfall jeder Buchstabe im Alphabet sein.

Die Shift Tabelle speichert für jedes Suffix des Suchwortes, das zur Nicht-Übereinstimmung mit dem Text führen kann, die sichere Verschiebedistanz.

Für unser Beispiel würden die Tabellen wie folgt aussehen.

Shift Tabelle			
T	E	X	T
3	3	3	1

Last Tabelle		
E	T	X
1	3	2

3.2 Empirische Laufzeitanalyse

Der Algorithmus von Boyer-Moore zeichnet sich durch erheblich kürzere Laufzeiten als die Algorithmen Brute-Force und Knuth-Morris-Pratt aus, da dieser auch nicht erfolgreiche Vergleiche zur Verbesserung seiner Laufzeit-Eigenschaften heranziehen kann. Durch die Nutzung zweier verschiedener Heuristiken beträgt die Abschätzung für den worst case zwar ebenfalls O(n+m), allerdings tritt dieser deutlich seltener auf. Der average case hingegen weist bei großen |Σ| und einem kurzen Pattern eine durchschnittliche Laufzeit von O(n/m) Vergleichen auf. Der zusätzliche Aufwand für die SkipTable beträgt m+|Σ| und für die NextTable entsteht der gleiche Aufwand wie für den Algorithmus von Knuth-Morris-Pratt.

Best Case

Im optimalen Fall greift der Algorithmus nur auf last-tabelle zurück und kann nach jedem Vergleich bis zum Finden des letzten Zeichens einer Fundstelle *Muster* um seine gesamte Länge *m* weiter schieben. Es ergibt sich $p = 1 / m$, wie das folgende Beispiel zeigt:

```
Text: ABCDEFGHIJKLMNOPQRSTUVWXYZZZZZ
Muster: ZZZZZ

Ablauf:
Vergleich #1 ZZZZZ
Vergleich #2      ZZZZZ
Vergleich #3           ZZZZZ
Vergleich #4                ZZZZZ
Vergleich #5                     ZZZZZ
Vergleich #6                          ZZZZZ


      Text   ABCDEFGHIJKLMNOPQRSTUVWXYZZZZZ
```

Die **Shift-Tabelle** spielt im best-case eine untergeordnete Rolle das Sie auch nur eine Sprungdistanz von m liefern kann, dies ist sehr unwahrscheinlich da mustern wie aaa...aaaa oder bb...b selten in Texten zu finden sind.

Worst case:

Wie Boyer und Moore in [BM77, S. 767] selbst feststellen, ist die Betrachtung des worst case „nontrivial" und im Laufe der Jahre wurden immer wieder vereinfachte Beweise und gesenkte obere Schranken der Komplexität geliefert. Aktueller Stand zum Zeitpunkt dieser Ausarbeitung ist eine definitiv **lineare Abhängigkeit** der durchgeführten Instruktionen O(i) (diverse Beweise) und eine maximale Anzahl Vergleiche mit oberer Schranke 3*i (Richard Cole in [Co94]), nachdem zuvor Knuth in [KMP77] 6*i erzielte und Guibas und Odlyzko in [GO80] auf 4*i gelangten.
Obwohl für den Optimalfall dargestellt wurde, dass dieser durch Verwendung der „bad character rule" herbeigeführt wird, ist **Last-Tabelle** jedoch im worst case sehr ineffizient, wie folgendes Gegenbeispiel demonstrieren soll:

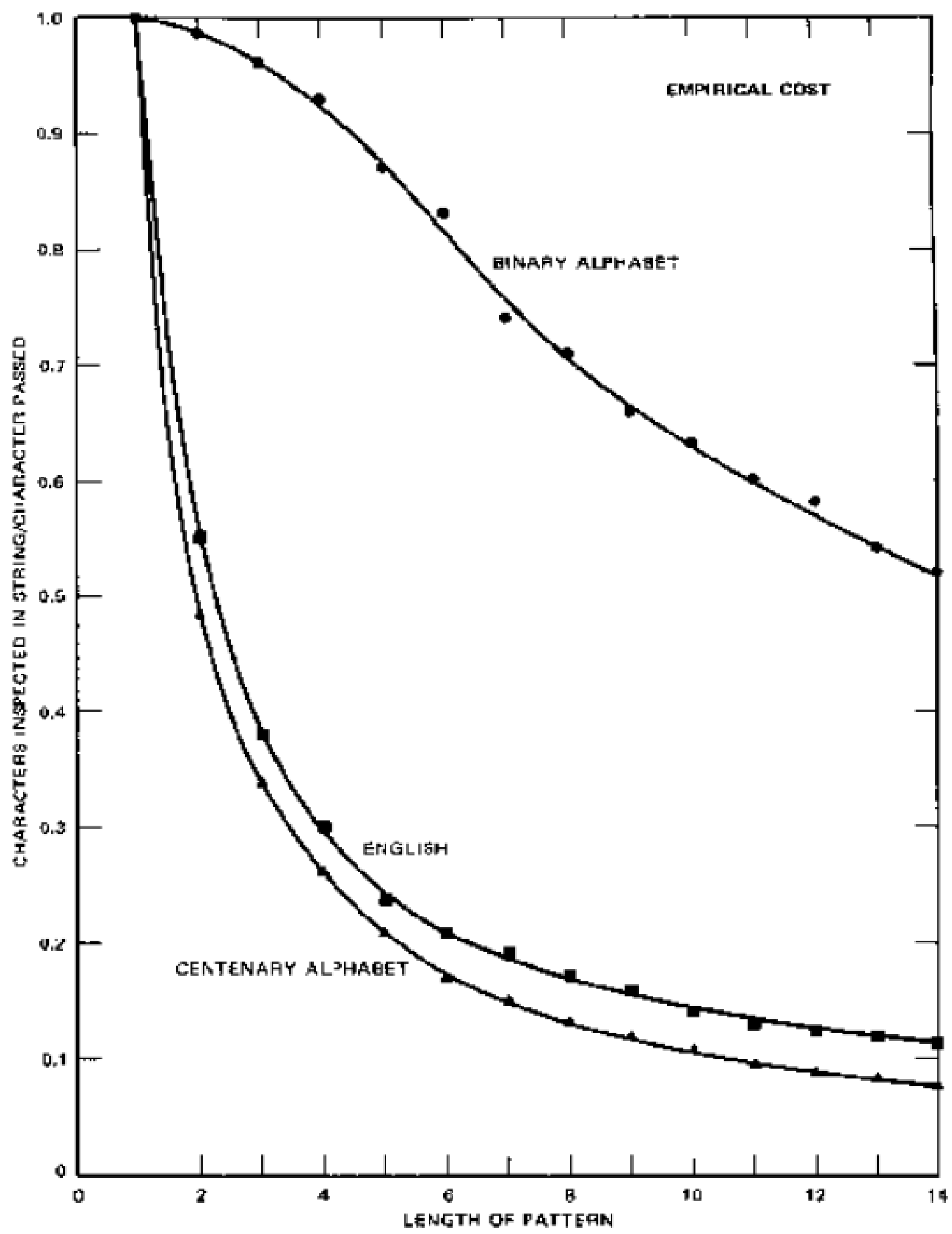
```
Text: 00000...010000
Muster: 10000

Ablauf:
Vergleichslauf #n     10000
Vergleichslauf #n-1    10000
Vergleichslauf #3       bis n-2 ...........
Vergleichslauf #2           10000
Vergleichslauf #1            10000
Text                  000000...010000
```

Der Aufwand läge hier in der Ordnung **O(n*m).**

Trotzdem hat man durch die good-suffix Regel ein Linearität des BM-Algorithmus in der Laufzeit beweisen können.

Analyse durchschnittlich zu erwartenden Verhaltens

In [BM77] haben Boyer und Moore empirische Analysen ihres Algorithmus in Bezug auf p und die durchschnittliche Anzahl **ausgeführter Instruktionen** pro Zeichen bis zum Treffen auf eine Fundstelle i, im weiteren Verlauf r genannt, veröffentlicht. Wie oben bereits aufgezeigt wurde, hängen diese Kennzahlen maßgeblich von m und a ab, so dass Boyer und Moore auch in ihren Untersuchungen drei verschiedene Alphabetgrößen und Textarten verwendeten: eine zufällige Abfolge von Nullen und Einsen, einen Auszug aus einem englischen Anleitungstext und eine zufällige Abfolge von Zeichen aus einem 100 Zeichen umfassenden Alphabet. Die zu findenden Muster wurden jeweils 300-mal zufällig aus den Texten gewonnen und hatten eine Länge zwischen 1 und 14 Zeichen. Die zu durchsuchenden Zeichenkettenlänge betrug 10.000 Zeichen. Die jeweiligen Werte für p und r wurden dann über ihre 300 Ergebnisse gemittelt. Die Implementierung zur Ermittlung von r erfolgte in PDP-10 Assembler, wobei Boyer und Moore hier auf Probleme mit der direkten Adressierung einzelner Zeichen stießen und einen weiteren Overhead von 5 Instruktionen pro 200 Zeichen in Kauf nehmen mussten. Es ist also davon auszugehen, dass der Algorithmus auf Maschinen mit frei inkrementierbaren Byte-Adressierungen noch etwas zügiger laufen wird.


EMPIRICAL COST
BINARY ALPHABET
ENGLISH
CENTENARY ALPHABET
CHARACTERS INSPECTED IN STRING/CHARACTER PASSED
LENGTH OF PATTERN
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
2
4
6
8
10
12
14

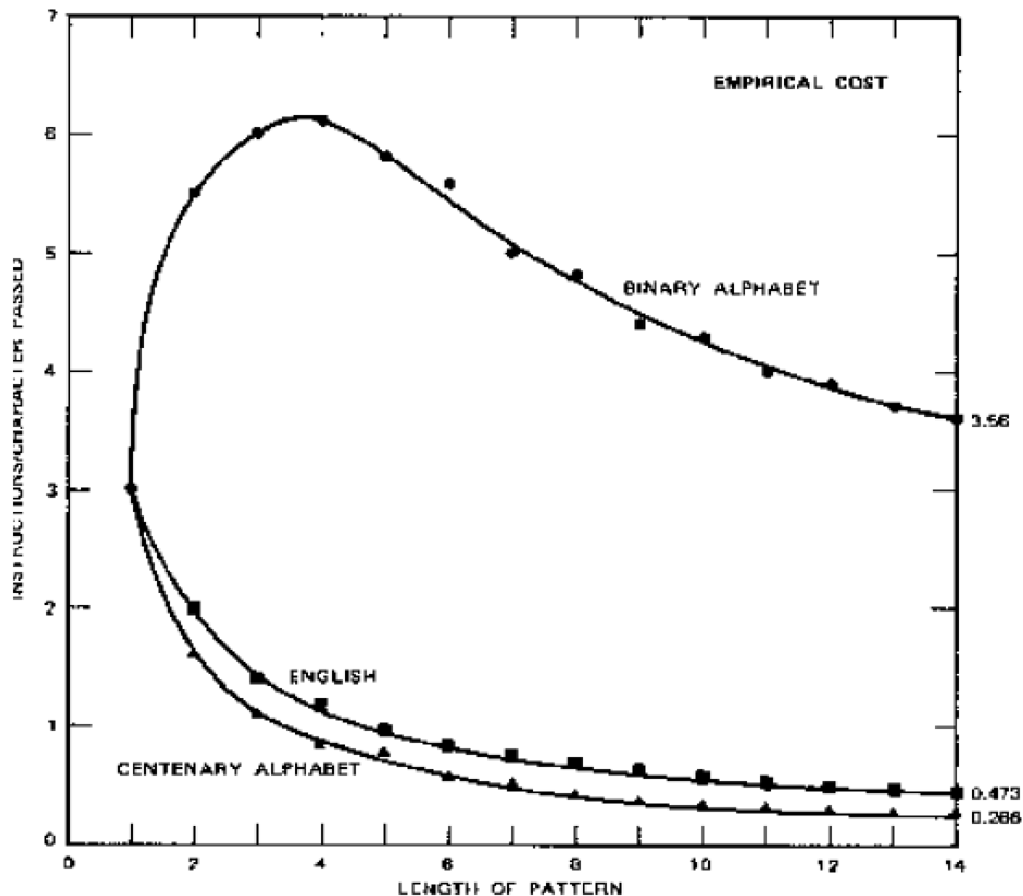


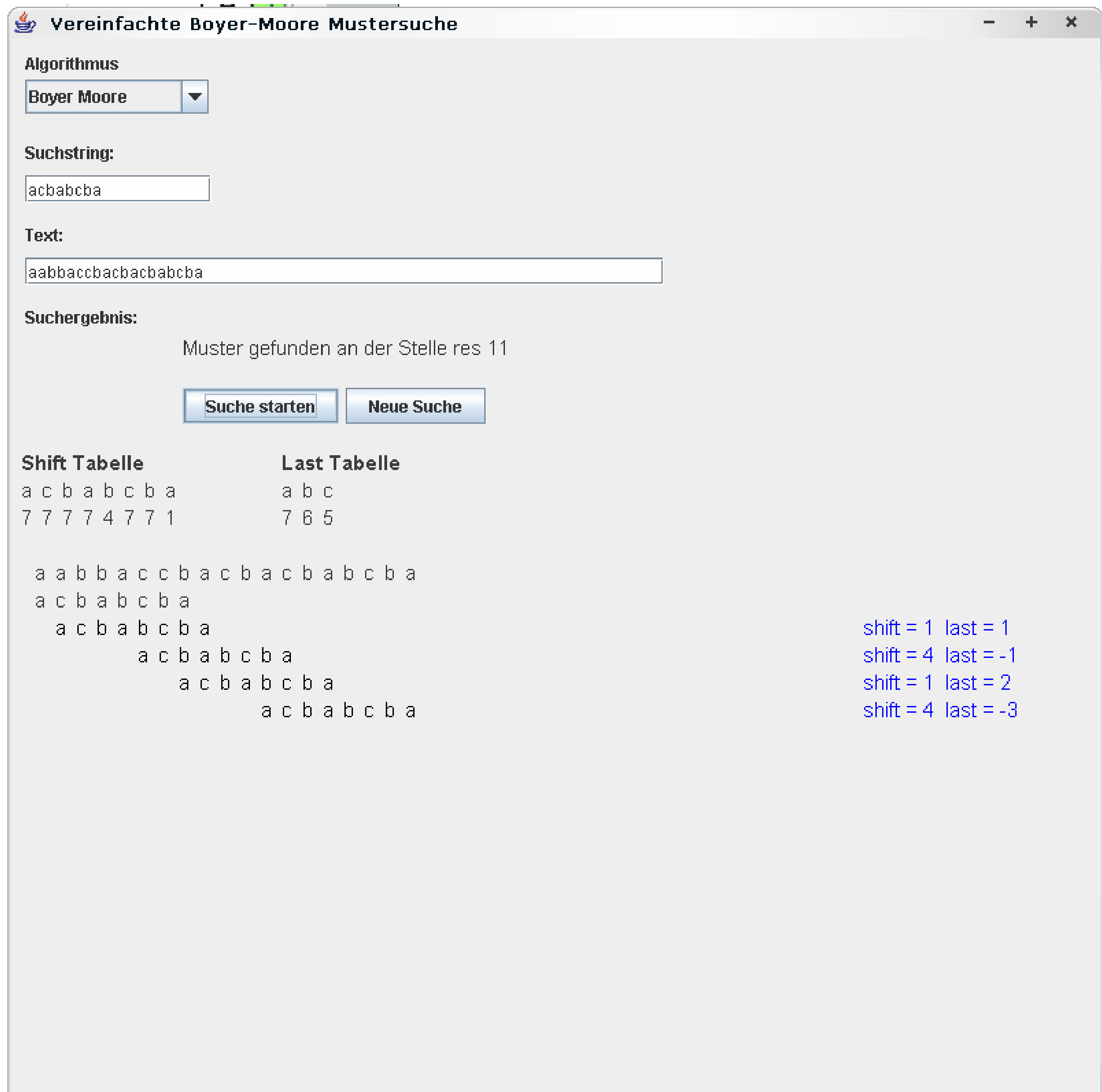
Diagramm 2: empirisch r gegen m

In Bezug auf p lässt folglich davon ausgehen, dass der Boyer-Moore Algorithmus „sublinear“ abhängig von m ist: mit steigender Musterlänge wird er immer schneller. Gleiches Verhalten lässt sich auch in Bezug auf r feststellen; hier kann jedoch weiter hinzugefügt werden, dass ab einer gewissen Musterlänge m das instruktionsgebundene r unter 1 sinkt und somit kein Algorithmus, der tatsächlich jedes Zeichen in *Text* inspiziert, schneller sein kann, wenn er für einen Vergleich mindestens eine Instruktion benötigt.
Im weiteren Verlauf führen Boyer und Moore eine analytische Untersuchung auf Basis der Annahmen durch, dass alle Zeichen in *Muster* und *Text* Realisationen unabhängig identisch verteilter (u.i.v.) Variablen sind und vergleichen die simulierten Ergebnisse mit den empirischen, wobei weitestgehend Übereinstimmungen erzielt werden, die eine vermutete „Sublinearität“, also p von unter 1 bzw. $O(t/m)$, bestätigen.
Tsung-Hsi Tsai betrachtet in seinem noch nicht publizierten Paper [Ts03] ein Modell für die probabilistische Analyse des Boyer-Moore Algorithmus mittels Markov-Ketten und findet wie auch Boyer und Moore zu einem asymptotischen Verhalten von p, das er mit empirischen Untersuchungen untermauert.

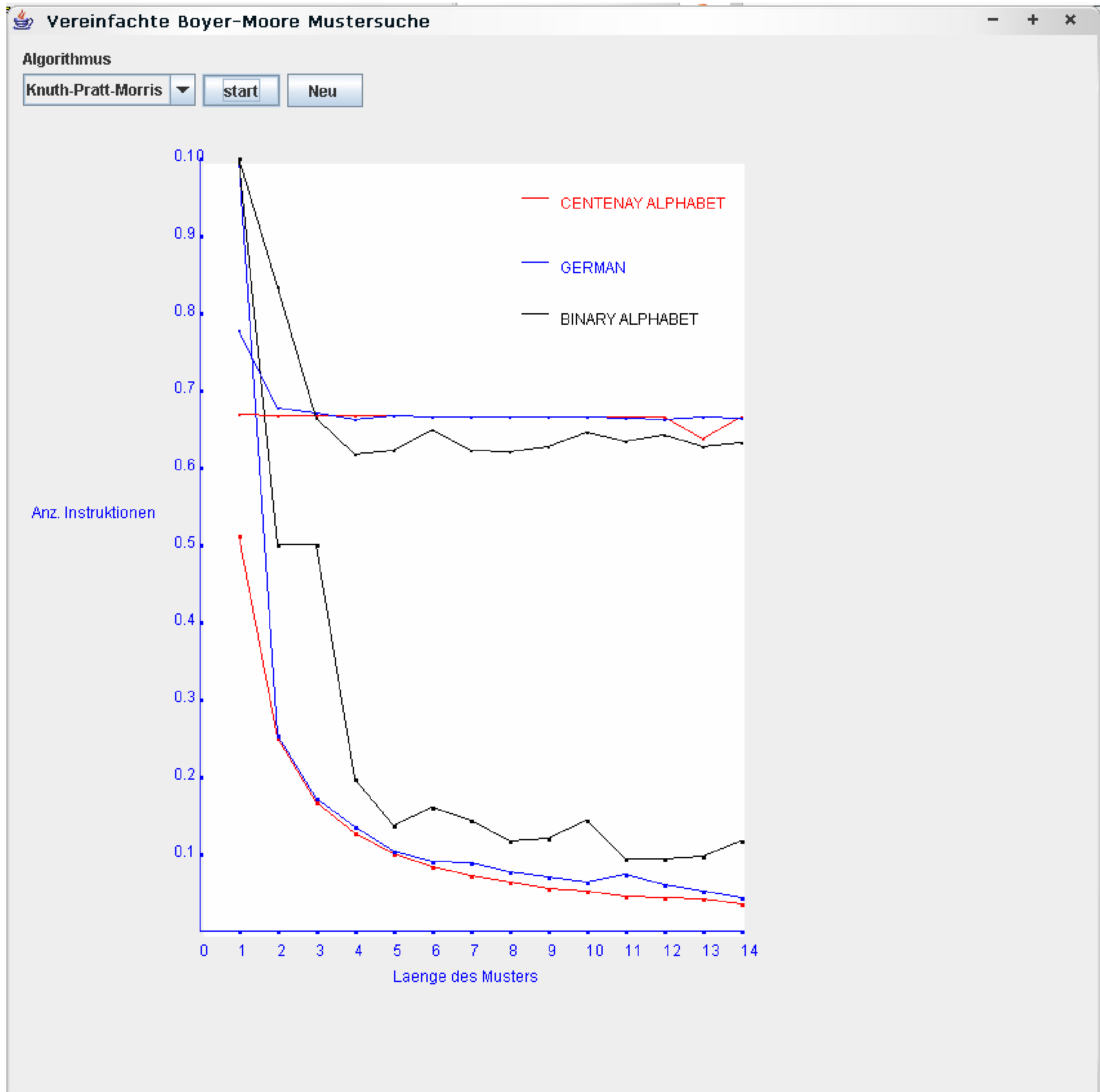
Ohne hier eine der beiden sehr abstrakten Untersuchungen detailliert auszuführen, lässt sich leicht überlegen, dass unter Verwendung eines großen Alphabets und einer hierzu relativ kleinen Musterlänge die Komplexität des Boyer-Moore Algorithmus im Rahmen von O(*i/m*), respektive O(*t/m*) liegt.
Die Wahrscheinlichkeit, bei einem Vergleich eine Übereinstimmung zu erzielen, liegt bei 1/*a*, wenn eine Gleichverteilung der Zeichen aus *A* über den Text angenommen werden kann. Für einen zweiten „Treffer" in Folge sinkt sie schon auf $1/a^2$, dann auf $1/a^3$ etc pp. Die Wahrscheinlichkeit für einen Sprung ist bei großem *a* also sehr hoch und somit die Wahrscheinlichkeit für eine sinnvolle Verwendung von $delta_2$ sehr gering, denn dies liefert tendenziell nach kurzen Übereinstimmungssequenzen kürzere Sprünge als $delta_1$.
Konzentrieren wir uns nun folglich auf $delta_1$ und die zu erwartende Sprunglänge:
Sei *v* die Anzahl verschiedener Zeichen aus *A* in *Muster*, so liegt die Wahrscheinlichkeit, dafür, dass dieser Vergleich einen $delta_1$-Wert von *m* (was ja die maximale Sprunglänge sein muss) liefert bei *(a-v)/a*. Davon ausgehend, dass *v* proportional zu *m* und somit klein ist (was bei der angenommenen Gleichverteilung unterstellt werden kann), geht *(a-v)/a* gegen 1.
Sucht man z.B. in einem ASCII-Text (*A*=256) nach einem 8 Zeichen langen Wort und geht davon aus, dass alle 8 Zeichen in diesem Wort verschieden sind, so ergebt sich als Wahrscheinlichkeit für einen Sprung der Länge *m*=8 nach einem Vergleich von ((256-1)/256) * ((256-8)/256) = 96,5%. Da die hier vorgenommenen Vereinfachungen natürlich sehr realitätsfern sind, gehen wir nun von einem effektiv genutzten Alphabet von 47 Zeichen aus (lateinische Buchstaben, Satzzeichen, Anführungszeichen und Zahlen) und *v*=5 aus. Selbst jetzt liegt die Wahrscheinlichkeit für einen Sprung von 8 Zeichen noch bei ((47-1)/47) * ((47-5)/47) = 87,5%.
Es kann also angenommen werden, dass *m* sich reduzierend als Faktor 1/(*x*m*) mit 1>x>0 auf die Komplexität auswirkt, da die Anzahl der nötigen Vergleiche durch eben diesem Faktor reduziert wird.

4. Vorstellung der Implementierung beider Algorithmen in eine GUI

Hier nun eine kurze Vorstellung des Programms, welches das Verständnis der Algorithmen verbessern soll. Es ist recht einfach Aufgebaut. Neben der Auswahl des Algorithmus kann der Suchstring und der Text eingegeben werden. Nach klicken auf Suche starten wird die Shift und/oder Last Tabelle angezeigt. Außerdem wird die Suche Schrittweise dargestellt.



Für die empirische Laufzeitanalyse wurde noch eine weitere GUI erstellt. Dort werden unterschiedlich große Alphabete mit einem Unterschiedlich langen Suchtext ausprobiert und ihre Laufzeit mit dem jeweiligen Algorithmus gemessen.



Hier entsprechen die unteren 3 Kurven dem Boyer Moore Algorithmus. Während die anderen 3 zum KMP-Algorithmus gehören.

Quellen:

Boyer-Moore-Algorithmus:
http://szugat.gmxhome.de/bioinformatics/BMA.ppt

Algorithmentheorie – Suche in Texten:
http://porta.informatik.uni-freiburg.de/lectures/Algorithmentheorie/2003-2004WS/Misc/Slides/17_Text-Suche.ppt

Textsuche:
http://ii.umit.at/teaching/ADSII05/6-ADII-Textsuche.pdf

Suche in Texten:
http://download.informatik.uni-freiburg.de/lectures/Algorithmentheorie/2004-2005WS/Misc/Slides/15_Suche_in_Texten.pdf