Universität Hannover

Diplomarbeit
Mehrdimensionale Mustersuche

Autor: Tobias Müller
Matrikelnummer: 1762402

Prüfer: Prof. Dr. R. Parchmann und Dr. J. Specht

Datum: 15.9.2005

Erklärung

Hiermit versichere ich, diese Arbeit
selbstständig verfasst und nur die
angegebenen Quellen benutzt zu haben.

____

(Tobias Müller)

Inhaltsverzeichnis

1 Einleitung **5**

2 Algorithmus von Amir, Benson und Farach **9**

2.1 Zweidimensionale Perioden 9
2.2 Grundidee des Algorithmus 13
2.3 Vorverarbeitung des Suchmusters 16
2.3.1 Der Algorithmus von Main und Lorentz 16
2.3.2 Berechnung der Witness-Tabellen 20
2.4 Bearbeitung des Textes 24
2.4.1 Eindimensionaler Konsistenz-Algorithmus 26
2.4.2 Zweidimensionaler Konsistenz-Algorithmus 32
2.5 Verifizierung der Fundstellen 42
2.6 Résumé 48

3 Idee von Crochemore, Gąsienic, Plandowski und Rytter **49**

3.1 Vorüberlegungen und Begriffe 49
3.2 Suche nach nichtperiodischen Mustern 52
3.3 Suche nach periodischen Mustern 60
3.4 Résumé 61

4 Algorithmus von Kärkkäinen und Ukkonen **63**

4.1 Der generelle Algorithmus 64
4.2 Quadratische Vorlagen 69
4.2.1 Laufzeitverhalten 73
4.3 Lineare Vorlagen 75
4.3.1 Laufzeitverhalten 79
4.4 Résumé 80

5 Laufzeitvergleich der vorgestellten Algorithmen **81**

6 Résumé und Aussicht **95**

1 Einleitung

Die eindimensionale Zeichenkettensuche, also die Suche nach einer bestimmten Zeichenkette in einem Text, ist weitgehend erforscht. Für diesen Themenbereich existiert eine Vielzahl effektiver Algorithmen. In den letzten Jahren wird jedoch die Suche nach mehrdimensionalen Mustern immer wichtiger. Beispiele für deren Anwendung finden sich in der Bilderkennung im industriellen Bereich, in vielen Bereichen der Bioinformatik und besonders in den letzten Jahren in der Verarbeitung von biometrischen Daten zur Authentifikation von Benutzern.

Diese Arbeit zeigt anhand der zweidimensionalen Mustersuche einige Ideen und Algorithmen auf, die sich auch auf höhere Dimensionen erweitern lassen. Die Überführung der bekannten eindimensionalen Algorithmen auf mehrdimensionale Anwendungen ist leider nur bedingt möglich. In den meisten Fällen können sie nur dazu benutzt werden um ein n-dimensionales Problem in ein n-1-dimensionales Problem zu überführen. Auf diese Weise gelang es auch Baker[8] und Bird[9] unabhängig von einander einen ersten zweidimensionalen Suchalgorithmus zu entwickeln. Sie generierten dabei aus dem gesuchten Muster einen AC-Automaten[2], der für jede Spalte einen bestimmten Endzustand erreicht. Die Endzustände aller Spalten ergeben damit hintereinander geschrieben eine Zeichenkette s, welche das Muster eindeutig identifiziert. Benutzt man den Automaten über dem Text, so kann dieser genauso in eine Matrix von Endzuständen überführt werden. Findet sich s in dieser Matrix wieder, so ist auch das gesuchte Muster im Ausgangstext gefunden. Dieser Algorithmus lässt sich bereits in einer Zeit von $O(Nlog\ c)$ realisieren.

Da hier jedoch ein speziell auf eindimensionale Probleme angepasster Algorithmus zweckentfremdet wird, ist zu erwarten, dass er nicht in der selben effektiven Laufzeit arbeitet wie ein auf n Dimensionen spezialisiertes Verfahren. Ein solches Verfahren setzt eine genaue Untersuchung der Eigenschaften von höher dimensionalen Mustern voraus. Vorreiter auf diesem Gebiet sind Amir, Benson und Farach[3], die mit ihren Ideen bis heute alle gängigen Algorithmen beeinflussen. Ein erster von ihnen entwickelter spezialisierter Algorithmus wird im ersten Abschnitt dieser Arbeit vorgestellt.

Amir, Benson und Farach verfolgen dabei einen optimistischen Ansatz und gehen von einer potentiell großen Anzahl von Fundstellen aus. Im zweiten Abschnitt dieser Arbeit wird eine genau entgegengesetzte Idee von Crochemore, Gąsienic, Plandowski und Rytter präsentiert[10]. Sie nehmen pessimistisch eine minimale Anzahl von Fundstellen an. Anstatt nun diejenigen Orte zu suchen, an denen diese Fundstellen liegen, verfolgen sie die Idee durch einige gezielte Vergleiche die Vielzahl an Positionen im Text zu bestimmen, an denen das

gesuchte Muster nicht gefunden wird. Für diese Suche entwickeln sie einige Ideen für komplexe dynamische Verfahren. Diese Ideen wurden nie zu einem kompletten Algorithmus umgesetzt, sie bilden jedoch als Gedankenmodell die Grundlage für einige später entwickelte Verfahren.

Im dritten Abschnitt wird eines dieser Verfahren vorgestellt. Kärkkäinen und Ukkonen geben einen Algorithmus an, der ebenfalls den pessimistischen Ansatz verfolgt und im Vorfeld nicht alle, aber eine große Anzahl von Kandidaten als Fundstellen ausschließt[18]. Da nur wenige Kandidaten verbleiben ist es vertretbar, diese im Anschluss mit einem trivialen Algorithmus exakt zu untersuchen. Dabei präsentieren die Autoren einen generellen Algorithmus, der auf viele Arten spezialisiert werden kann. Zwei dieser Spezialisierungen werden genauer untersucht.

Die Arbeit endet mit einem praktischen Vergleich der Algorithmen bei realer Implementierung. Hierbei wird sich der Algorithmus von Kärkkäinen und Ukkonen als derjenige herausstellen, der in den meisten Fällen die besten Laufzeiten liefert.

In dieser Arbeit wird davon ausgegangen, dass in einer quadratischen Matrix T der Größe $n \times n$ eine ebenfalls quadratische Untermatrix P der Größe $m \times m$ gesucht wird. T wird im Folgenden als Text bezeichnet, die Untermatrix P als (Such-)Muster (engl. Pattern). Der obere linke Eintrag von P ist der Ursprung von P, die Position welche sich in T unterhalb des Ursprungs von P befindet wird dementsprechend auch als Ursprung von P in T bezeichnet.
Mit Σ bzw. σ wird die Größe des Alphabets benannt, welches dem Text bzw. dem Muster zu Grunde liegt.
Die Teilzeichenkette einer Zeichenkette s, welche vom i-ten bis zum j-ten Zeichen reicht, wird mit $s[i, j]$ oder $s[i...j]$ bezeichnet, s_i steht abkürzend für $s[i, i]$.
Einzelne Zeichen in einer Matrix T werden mit $T[i, j]$ notiert. Dies meint das j-te Zeichen in der i-ten Zeile.
Mit $T[0 : 3, 1 : 4]$ bzw. $T[0...3, 1...4]$ wird die Untermatrix von $T = (t)_{ij}$ mit $0 \leq i \leq 3,\ 1 \leq j \leq 4$ bezeichnet, also die Untermatrix von T welche ihre linke obere Ecke bei $T[0, 1]$ und ihre rechte untere Ecke bei $T[3, 4]$ hat.

Die Zählung der Zeilen und Spalten beginnt in dieser Arbeit grundsätzlich bei 0. Als Kandidat wird eine Position im Text bezeichnet, an der die gesuchte Untermatrix gefunden werden könnte. Es handelt sich also um eine potentielle Fundstelle. Alternativ können beide Begriffe anstelle der Position auch für einen $m \times m$ großen Bereich an dieser Position stehen. Die jeweilige Bedeutung ist aus dem Kontext ersichtlich.
Einfachere Algorithmen werden in Pseudocode angegeben, bei komplizierteren Verfahren wird eine besser verständliche, verbale Beschreibung benutzt.

Die Bezeichnungen orientieren sich zum größten Teil an denen aus der englischen Literatur, um die angegebenen Algorithmen und Ideen leichter mit weiteren Verfahren vergleichen zu können. Ferner wurden die meisten englischen Fachbegriffe als solche benutzt und nur einige wenige übersetzt.

Die Einschränkung auf quadratische Texte und Muster vereinfacht die Algorithmen in einigen Fällen erheblich. Die Erweiterung auf beliebige rechteckige Texte ist jedoch in jedem Fall durch eingefügte Fallunterscheidungen oder veränderte Grenzen möglich.
Will man zweidimensionale unregelmäßige Muster suchen, so können diese vorher auf rechteckige Muster aufgefüllt werden. Dabei ist darauf zu achten, dass die Füllzeichen in den Algorithmen bei jedem Vergleich wie Wildcards behandelt werden, die Vergleiche also je nach Art des Algorithmus immer erfolgreich sind oder immer fehlschlagen.

2 Algorithmus von Amir, Benson und Farach

Für eine effiziente Umsetzung ihres Algorithmus benutzen Amir, Benson und Farach einige Ideen und Begriffe aus einer vorangegangen Arbeit von Amir und Benson[3] über Perioden in zweidimensionalen Mustern. Die wichtigsten Begriffe hieraus sollen deshalb zunächst als Grundlage für diesen und weitere Algorithmen erläutert werden.

2.1 Zweidimensionale Perioden

Die Idee für die Definition von Perioden führt auf den eindimensionalen Fall zurück. Eine Zeichenkette w ist periodisch, wenn das längste Präfix p von w, welches zugleich dessen Suffix ist, ein Länge von mindestens $p/2$ hat. Zum Beispiel ist $w = abcabcabcab$ periodisch, da $p = abcabcab$ das längste Präfix und zugleich Suffix von w und $2 * |p| > |w|$ ist. Anschaulich kann man zwei Kopien von w übereinander legen, diese um $|w| - |p| = 3$ verschieben und erhält im Bereich der Überlappung eine Übereinstimmung der Zeichen.

Diese Definition kann leicht auf den zweidimensionalen Fall übertragen werden. Der Präfix einer $m \times m$-Matrix M sei eine Untermatrix, welche einen der Eckpunkte von M enthält. Ein dazugehöriger Suffix von M ist eine Untermatrix, welche den diagonal gegenüberliegenden Eckpunkt beinhaltet.

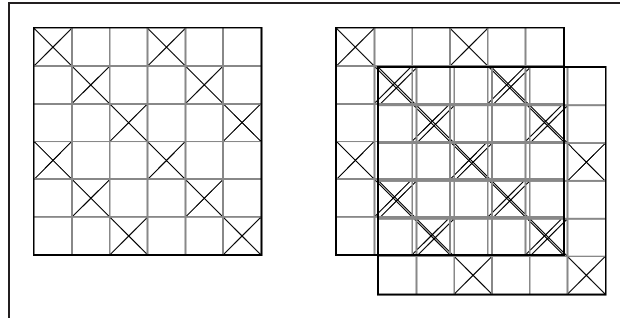


Abbildung 1: Ein Matrix und zwei sich überlappende Matrizen

M ist periodisch, wenn das größte Präfix P von M, welches gleichzeitig Suffix ist, mehr als halb so groß wie M ist, wenn also $2 * dim(P) > dim(M)$ gilt. Anschaulich führt dies wie im eindimensionalen Fall dazu, dass man zwei Matrizen übereinanderlegen und so verschieben kann, dass sie im überlappenden Bereich keinerlei Unterschiede aufweisen (Abbildung 1).

Nun sollen 4 verschiedene Klassen eingeführt werden, in welche sich die Perioden einteilen lassen.

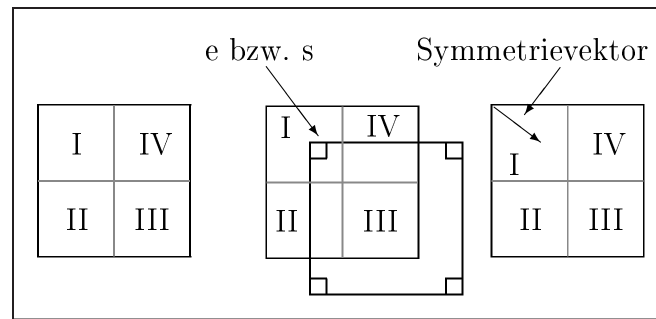


Abbildung 2: Quadrant-I Symmetrie

Hierfür wird zunächst die Matrix wie in Abbildung 2 in 4 Quadranten unterteilt, welche beginnend ab dem oberen linken Quadranten gegen den Uhrzeigersinn mit I bis IV markiert werden. Legt man zwei identische Kopien der Matrix übereinander, so sind alle übereinander liegenden Zeichen der beiden Matrizen identisch, die Matrizen sind in *Deckung*. Verschiebt man die obere Matrix so, dass die beiden Matrizen wieder in Deckung sind (sich also im überlappenden Bereich nicht unterscheiden), so liegt genau eine Ecke e der oberen Matrix über einem Zeichen s der unteren Matrix. Da es sich um zwei identische Matrizen handelt, sind e und s dabei in beiden Matrizen zu finden. s wird als *Source* (Ursprung) bezeichnet. Liegt e im ersten Quadranten der zugehörigen Matrix (handelt es sich also um $M[0,0]$), so ist die Matrix *Quadrant I symmetrisch* und s wird auch genauer als *Quadrant-I-Source* bezeichnet (vergleiche Abbildung 2). Entsprechend kann sie auch *Quadrant II, III oder IV symmetrisch* sein.

Der *Symmetrievektor* ist der Vektor von e nach s innerhalb der unteren Matrix. Er kann als Vektor zwischen den Matrizen interpretiert werden, der angibt, bei welcher Verschiebung sich die Matrizen wieder in Deckung befinden. Ist die Matrix periodisch, so wird dieser Vektor auch als *Periode* der Matrix bezeichnet. Je nachdem, in welchem Quadranten der Vektor startet, handelt es sich um einen *Quadrant I, II, III, oder IV - Symmetrievektor*. Die Länge eines Symmetrievektors sei definiert als das Maximum seines horizontalen und vertikalen Anteils. Ist diese Länge $< \frac{m}{2}$, so ist der Vektor *periodisch*. Anschaulich also dann, wenn er nicht über die horizontale bzw. vertikale Mitte der Matrix hinausreicht, also eine Verschiebung der oberen Matrix um weniger als die halbe Breite (bzw. Höhe) ausreicht, um die beiden Matrizen wieder in Deckung zu bringen.

Es ist offensichtlich, dass eine Matrix mehr als einen Symmetrievektor $\vec{v}$ besitzen kann. Wenn es möglich ist, dass man durch Verschiebung der oberen Matrix um $\vec{v}$ die beiden Matrizen in Deckung bringt, so muss dies auch durch eine Verschiebung der unteren Matrix um $-\vec{v}$ möglich sein, da es sich augen-

scheinlich um denselben Vorgang handelt. Zu jedem Symmetrievektor $\vec{v}$ gibt also einen zweiten Vektor $-\vec{v}$, welcher entgegengesetzt im diagonal gegenüberliegenden Quadranten startet. Aufgrund dieser Symmetrie werden im Folgenden nur noch Quadrant-I und -II symmetrische Vektoren betrachtet, also nur noch eine Verschiebung der oberen Matrix nach rechts oben oder rechts unten.

Zur Klassifikation der Perioden betrachtet man die so genannten Basisvektoren eines jeden Quadranten. Der *Basisvektor in Quadrant I* ist aus der Menge der kürzesten Quadrant-I-Vektoren derjenige mit dem betragsmäßig geringstem vertikalen Anteil ("der Waagerechteste"). Der *Basisvektor in Quadrant II* ist aus der Menge der kürzesten Quadrant-II-Vektoren derjenige mit dem betragsmäßig geringstem horizontalen Anteil ("der Senkrechteste").

Seien im Folgenden (falls existent) $\vec{v}_I$ und $\vec{v}_{II}$ die Basisvektoren des ersten und zweiten Quadranten. Es lassen sich nun für eine Matrix M vier Arten von zweidimensionalen Perioden unterscheiden.

- non-periodic (nichtperiodisch, Abbildung 3):
  M hat keine periodischen Vektoren.

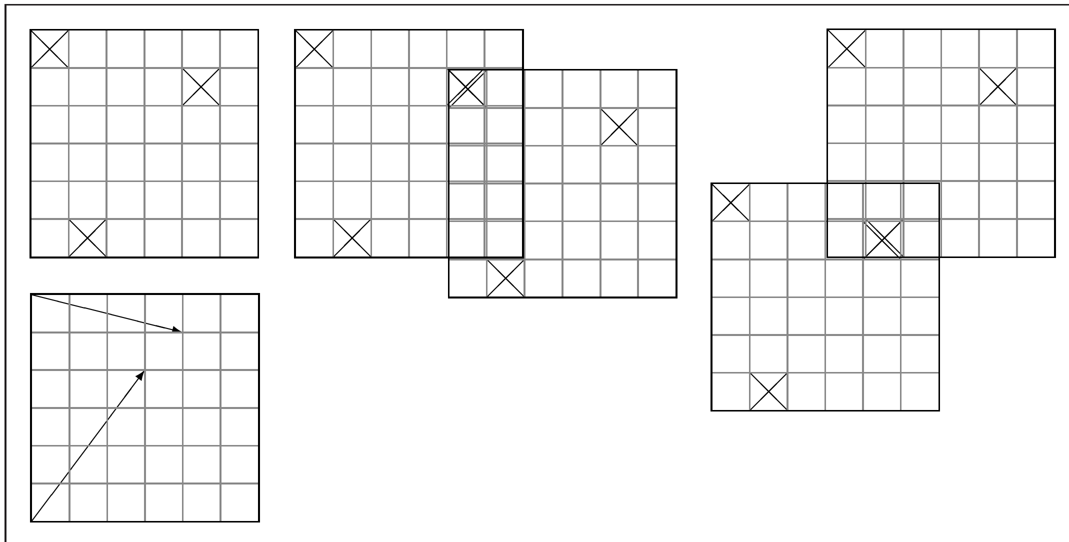


Abbildung 3: non-periodic Matrix

- lattice periodic (gitterperiodisch, Abbildung 4 auf der nächsten Seite):
  Der erste und der zweite Quadrant haben beide einen periodischen Basisvektor.
  Alle Source-Punkte liegen auf einem von den Basisvektoren aufgespannten Gitter, ganauer:
  Befindet sich der Punkt $(0,0) + i\vec{v}_I + j\vec{v}_{II}$ mit $i, j \in \mathbb{N}$ im ersten Qua-

dranten, so ist er ein Quadrant-I-Source. Weitere Quadrant-I-Sources existieren im ersten Quadranten nicht.
Die äquivalente Aussage gilt für Quadrant-II-Sources im zweiten Quadranten mit dem Gitter $(m-1,0)+i\vec{v}_I+j\vec{v}_{II}$.

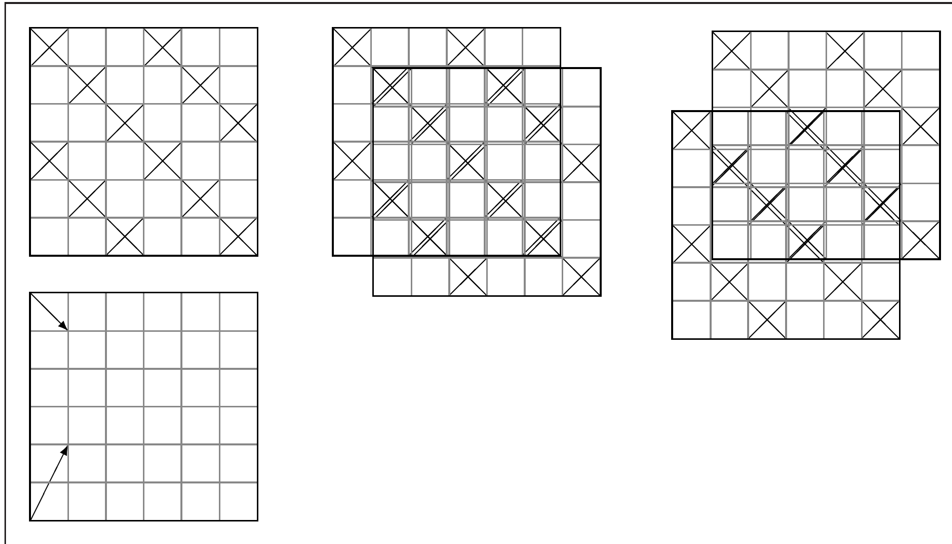


Abbildung 4: lattice periodic Matrix

- line periodic (geradenperiodisch, Abbildung 5 auf der nächsten Seite):
  Genau einer der beiden Quadranten I und II hat einen periodischen Basisvektor. Alle Source-Punkte im Quadranten mit dem periodischen Basisvektor liegen auf einer Geraden.
  Ist z.B. $\vec{v}_I$ der periodische Basisvektor, dann ist der Punkt $(0,0)+i\vec{v}_I$ mit $i \in \mathbb{N}$ wenn er sich im ersten Quadranten befindet ein Quadrant-I-Source und es existieren im ersten Quadranten keine weiteren Quadrant-I-Source-Punkte.

- radiant periodic (strahlenperiodisch, Abbildung 6 auf Seite 14):
  Dieser Fall ist nahezu identisch mit dem geradenperiodischen Fall, mit dem Unterschied, dass es weitere Source-Punkte gibt, welche alle auf Strahlen liegen, die in derselben Ecke wie der periodische Basisvektor beginnen. Die genaue Position hängt dabei von der zugrunde liegenden Matrix ab, ein Source-Punkt liegt jedoch nie auf einem Punkt, der sich als Linearkombination der Basisvektoren ergibt.

Die Ermittlung der Symmetrievektoren (und damit auch der Basisvektoren) kann mit Hilfe von zwei Witness-Tabellen (Zeugentabellen, Abbildung 7 auf Seite 14) erfolgen.
Diese werden im Folgenden mit `TOP-WITNESS`, bzw. `BOTTOM-WITNESS` bezeichnet.

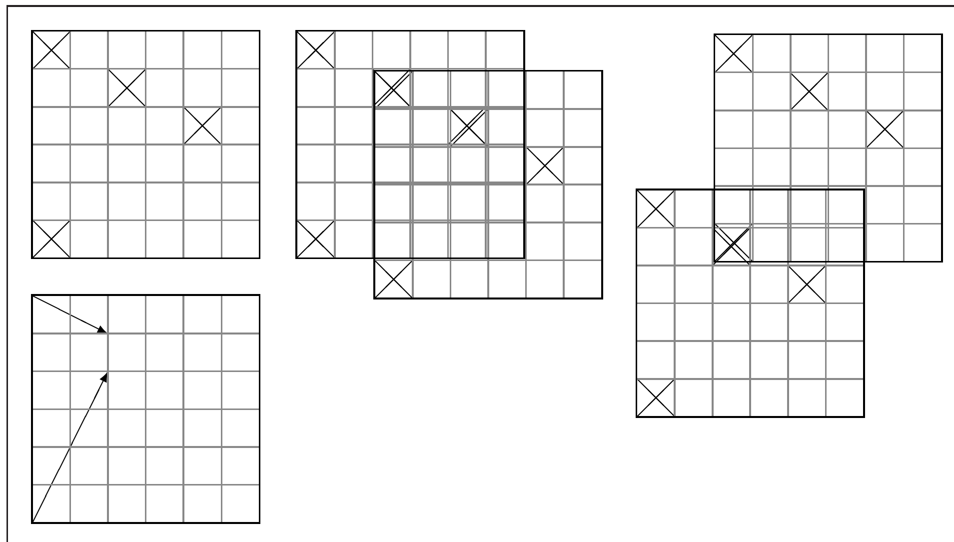


Abbildung 5: line periodic Matrix

Legt man wieder zwei Kopien der $m \times m$-Matrix M übereinander und verschiebt die obere Matrix um $i < m$ Zeichen nach unten und um $j < m$ Zeichen nach rechts, dann ergibt sich einer der folgenden Fälle:

- Die Matrizen sind nicht in Deckung. In diesem Fall gibt es in der oberen Matrix mindestens ein Zeichen $M[r, c]$, welches die untere Matrix überlappt und nicht mit dem Zeichen der unteren Matrix übereinstimmt. Das Zeichen $M[r, c]$ ist dann ein "Zeuge" dafür, dass sich die Matrizen nicht decken. Es kann mehr als einen Zeugen geben, die Wahl des Zeugen ist beliebig.

- Die Matrizen sind in Deckung.

In der Tabelle `TOP-WITNESS` wird nun für jeden Wert $0 \leq i, j < m$ entweder die Position eines Zeugen relativ zur verschobenen Matrix, oder – falls die Matrizen bei einer Verschiebung um (i, j) in Deckung sind – der Wert (m, m) als Platzhalter eingetragen. Die Symmetrievektoren im ersten Quadranten sind somit genau diejenigen (i, j), bei denen der Pseudozeuge (m, m) eingetragen wurde.
Verschiebt man die obere Matrix nach oben, so ergibt sich äquivalent die Tabelle `BOTTOM-WITNESS`, welche die Symmetrievektoren des zweiten Quadranten enthält.

2.2 Grundidee des Algorithmus

In einer schnellen Lösung des zweidimensionalen Suchproblems würde man für jede Position des Suchmusters im Text vergleichen, ob dieser dort mit dem Suchmuster übereinstimmt. Man geht also davon aus, dass jede Position auch

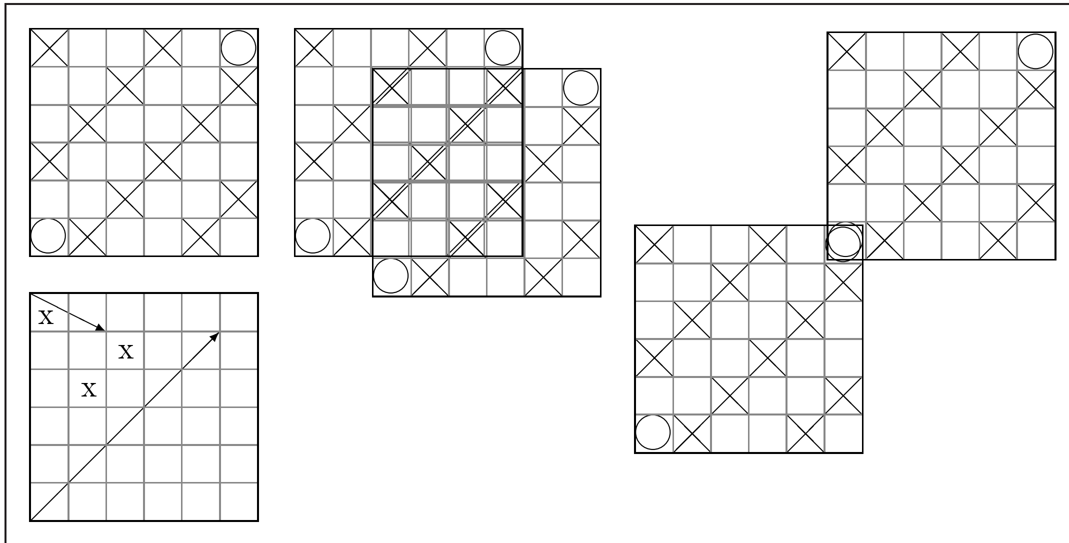


Abbildung 6: radiant periodic Matrix mit weiteren markierten Source-Punkten

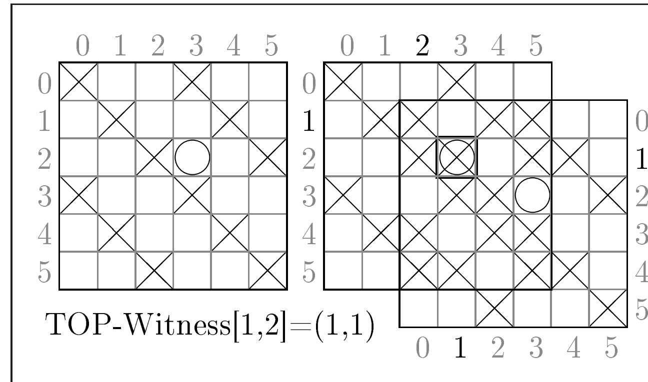


Abbildung 7: Witness-Eintrag für $(i, j) = (1, 2)$ und $(r, c) = (1, 1)$

eine mögliche Fundstelle sein könnte.
Amir, Benson und Farach machten jedoch die Beobachtung, dass nicht alle Positionen im Text als Fundstellen in Frage kommen[5]. Überlappen sich die Suchmuster an zwei Positionen und sind sie im Bereich der Überschneidung nicht identisch, so können nicht beide Positionen auch Fundstellen sein. Es müssen also nicht an beiden Positionen Vergleiche des Suchmusters mit dem darunter liegenden Text durchgeführt werden.
In diesem Fall kann durch eine vorher für das Suchmuster konstruierte Witness-Tabelle (Zeugentabelle) in konstanter Zeit eine Position bestimmt werden, an dem sich die beiden Suchmuster unterscheiden. Durch den Vergleich dieser Position (des sogenannten Zeugen) mit dem unter den Mustern liegenendem Text kann dann ermittelt werden welches der beiden Suchmuster nicht mit dem Text übereinstimmt und somit als Fundstelle ausgeschlossen werden kann. Dieses Suchmuster braucht nicht weiter betrachtet zu werden.

Mit dieser Idee ist es möglich vor der eigentlichen Suche die Anzahl der möglichen (und damit zu untersuchenden) Positionen stark einzuschränken, da sich die Muster hier in den überlappenden Bereichen in Deckung befinden müssen. Da sich in den möglichen Positionen die Fundstellen im Bereich der Überschneidung nicht unterscheiden, kann außerdem vermieden werden, ein Zeichen im Text mehrfach, also mit Zeichen aus verschiedenen Fundstellen, zu vergleichen, da bereits verglichene Textpositionen markiert und im weiteren Verlauf übersprungen werden können. In der endgültigen Suche werden somit nur maximal n^2 Vergleiche benötigt.

2.3 Vorverarbeitung des Suchmusters

2.3.1 Der Algorithmus von Main und Lorentz

Zur Vorverarbeitung des Suchmusters stellen Amir, Benson und Farach in ihrer Arbeit[5] einen einfachen Algorithmus vor, in dem sie versuchen das zweidimensionale Problem auf ein Problem der eindimensionalen Zeichenkettensuche zurückzuführen.
Die Grundvoraussetzung ist eine Idee aus einer Arbeit von Main und Lorentz[20]:

Algorithmus A: `max_prefix(W,T)`
Der Algorithmus `max_prefix` erhält als Eingabe eine Zeichenkette $T = t_0 t_1 ... t_{n-1}$ der Länge n und eine Zeichenkette $W = w_0 w_1 ... w_m$ der Länge m. Hieraus wird eine Tabelle `max_prefix[0..n-1]` erzeugt. Dabei ist `max_prefix`[i] die Länge des längsten Präfixes von W, welches in der Zeichenkette T beim Zeichen t_i beginnt (siehe Abbildung 8).

⊚

W =	A	B	A	B	A	B	C	
T =	C	A	B	A	A			
lppattern =	7	0	4	0	2	0	0	
lptext =	0	3	0	1	1			
max_prefix =	0	3	0	1	1			

Abbildung 8: max_prefix

Main und Lorentz geben einen zweigeteilten Algorithmus an, welcher dieses Problem in einer Zeit von $O(n)$ löst.
Im ersten Algorithmus `lppattern` wird zunächst `max_prefix(W,W)` berechnet. Aus der hieraus erhaltenen Tabelle wird mit dem leicht abgeänderten Verfahren `lptext` dann die endgültige Tabelle zu `max_prefix(W,T)` ermittelt (siehe Abbildung 9 auf der nächsten Seite).

Da die Teilalgorithmen beide auf derselben Idee, einer Variation des Algorithmus von Knuth, Morris und Pratt[17] beruhen, werden sie hier nur am Beispiel von `lppattern` beschrieben.
Der Algorithmus baut die Tabelle vom ersten zum letzten Eintrag auf, es ist nicht nötig innerhalb des Algorithmus noch einmal einen bereits ermittelten Wert abzuändern. Trivialer Weise ist der Wert von `lppattern[0]` immer m und wird daher in den folgenden Überlegungen nicht weiter betrachtet. Angenommen `lppattern[1]`...`lppattern[i-1]` wurden schon berechnet und es soll nun ein Wert für `lppattern[i]` ermittelt werden. Sei außerdem ange-

nommen es wurde bereits jenes k im Bereich $1 \leq k < i$ notiert, welches den Wert von $k +$ `lppattern`$[k]$ maximiert. Ist nun i kleiner als $k +$ `lppattern`$[k]$, so kann der bereits bekannte Wert von `lppattern[i-k]` zur Berechnung von `lppattern[i]` benutzt werden:

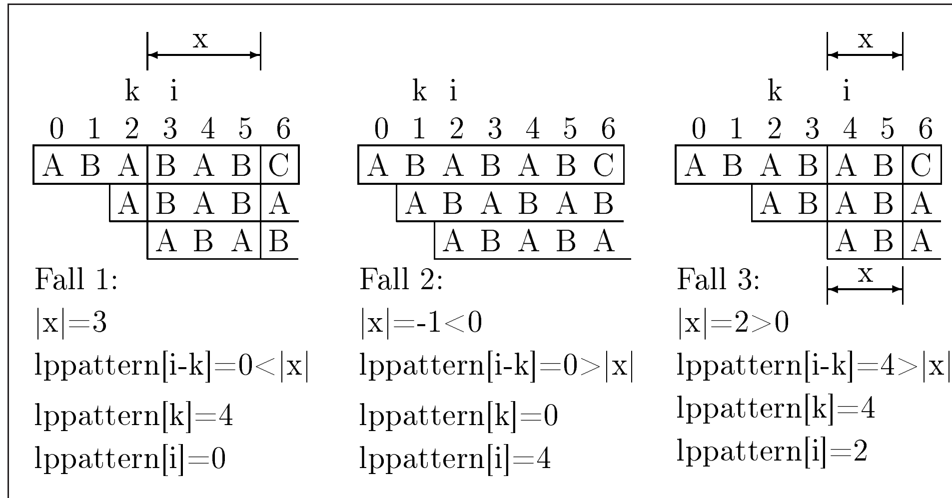


Abbildung 9: Die Fälle von `lppattern`

Befindet sich i unterhalb dieser Schranke, dann ist die Teilzeichenkette $x := w_i ... w_{k+\texttt{lppattern[k]}-1}$ mit der Zeichenkette $w_{i-k} ... w_{\texttt{lppattern[k]}}$ identisch.
Im Folgenden meint $|x|$ die Länge der Zeichenkette x. Liegt das Ende der Zeichenkette vor deren Anfang, so ist $|x|$ negativ!
Ist `lppattern`$[\texttt{i} - \texttt{k}] < |x|$, so gilt folglich `lppattern`$[\texttt{i}] =$ `lppattern`$[\texttt{i} - \texttt{k}]$ (Fall 1).
Ist andererseits `lppattern`$[\texttt{i} - \texttt{k}] \geq |x|$, so muss `lppattern`$[\texttt{i}]$ mindestens so groß wie $|x|$ sein, in diesem Fall können bei positivem x also $|x|$ Vergleiche eingespart werden (Fall 2,3).

Algorithmus B: `lppattern(W)`

W sei eine Zeichenkette der Länge m mit Endzeichen, jeder Vergleich mit w_m führt somit zum Misserfolg.
Es wird die Tabelle `lppattern[0]`...`lppattern[m-1]` wie oben beschrieben ausgegeben.

```
/* Initialisierung */
lppattern[0]:=m;
/* Bestimmung von lppattern[1] */
j:=0;
while(w[j] == w[j+1])
  j:=j+1;
```

```
lppattern[1]:=j;
k:=1;
/* Bestimmung der restlichen Einträge */
for(i:=2 to m-1) {
  |x|:=k+lppattern[k]-i;
  if(lppattern[i-k] < |x|) {
    lppattern[i]:=lppattern[i-k];   /* Fall 1 */
  } else {
    if(j >= k+lppattern[k])
      j:=0                          /* Fall 2 */
    else
      j:=|x|;                       /* Fall 3 */
    while(w[j] == w[j+i])
      j:=j+1;
    lppattern[i]:=j;
    k:=i;
  }
}
```

⊚

Anmerkung:

- Der erste Eintrag enthält trivialer Weise immer die Gesamtlänge von W.
- Der zweite Eintrag muss in jedem Fall vor der Bestimmung der restlichen Einträge komplett "ausgezählt" werden, da es noch keine Vorinformationen gibt.
- In Fall 2 ist $|x|$ negativ. Es sind also keine Vorinformationen vorhanden.
- In Fall 3 beginnt die "Zählung" erst ab $|x|$, da bis hierhin bereits "ausgezählt" wurde.

Theorem 1:
Der Algorithmus `lppattern` benötigt eine Zeit von $O(m)$.

Beweis:
In jedem Durchlauf der inneren `while`-Schleife wurde das Zeichen $w[j+i]$ noch nie erfolgreich mit einem anderen Zeichen aus W verglichen. Da es nur m Zeichen in W gibt, kann es folglich auch insgesamt nur m Durchläufe mit positivem bzw. negativem Test geben.
Also ergibt sich insgesamt für die äußere `for`-Schleife und damit für den Algorithmus die angegebene Laufzeit. □

Durch kleine Änderungen des Algorithmus erhält man `lptext`:

Algorithmus C: `lptext(W,T,lppattern)`

W bzw. T seien Zeichenketten der Länge m bzw. n mit Endzeichen, jeder Vergleich mit w_m bzw. t_n führt somit zum Misserfolg.
`lppattern` sei die vom gleichnamigen Algorithmus erzeugte Tabelle.
Es wird die Tabelle `lptext[0]`...`lptext[n-1]` ausgegeben, diese entspricht der gesuchten Tabelle `max_prefix`.

```
/* Bestimmung von lptext[0] */
j:=0;
while(w[j] == t[j])
  j:=j+1;
lptext[0]:=j;
k:=0;
/* Bestimmung der restlichen Einträge */
for(i:=1 to n-1) {
  |x|:=k+lptext[k]-i;
  if(lppattern[i-k] < |x|) {
    lptext[i]:=lppattern[i-k];
  } else {
    if(j >= k+lptext[k])
      j:=0;
    else
      j:=|x|;
    while(w[j] == t[j+i])
      j:=j+1;
    lptext[i]:=j;
    k:=i
  }
}
```

◎

Theorem 2:

Der Algorithmus `lptext` benötigt eine Zeit von $O(n)$.

Beweis:

Analog zum Beweis der Laufzeit von `lppattern`. □

Theorem 3:
Der Algorithmus `max_prefix` lässt sich in einer Zeit von $O(n)$ realisieren.

Beweis:
Für den Algorithmus von Main und Lorentz folgt eine Gesamtlaufzeit von $O(m + n)$. Da im Allgemeinen $m > n$ gilt, genügt es im Algorithmus `lppattern` die ersten n Elemente zu berechnen, da Einträge größer n nicht mehr für den Algorithmus `lptext` benötigt werden.
Hierdurch lässt sich die Laufzeit auf $O(2n) = O(n)$ senken. □

2.3.2 Berechnung der Witness-Tabellen

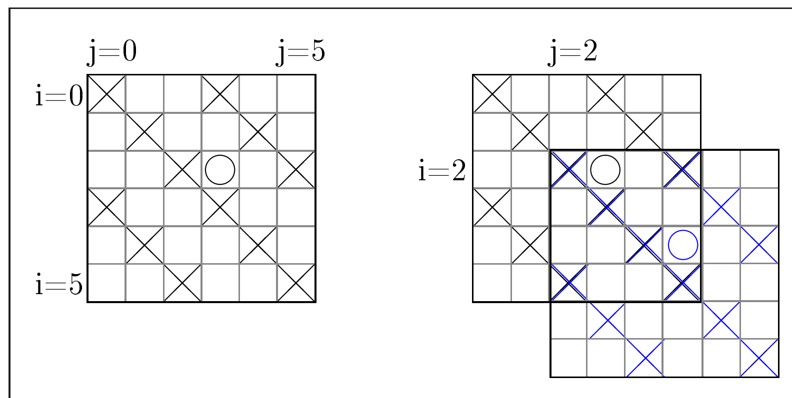


Abbildung 10: Einzelnes Muster und Verschiebung um $(i, j) = (2, 2)$

Man stelle sich vor, zwei identische Muster lägen wie in Abbildung 10 übereinander. Im Adhoc-Verfahren würde man nun alle übereinander liegenden Zeichen miteinander vergleichen, in diesem Fall wären das 16 Vergleiche unter der Annahme, die Muster würden übereinstimmen.
Schafft man es jedoch ohne großen Aufwand zuerst die überlappenden Zeilen (anstelle der einzelnen Zeichen) miteinander zu vergleichen und bestimmt erst bei Nichtübereinstimmung die Fehlerstelle innerhalb der entsprechenden Zeile, so käme man im vorliegenden Beispiel unter gleicher Annahme auf nur maximal 4 Zeilen- und 4 Einzelvergleiche.

Der folgende Algorithmus leistet dies:

Algorithmus D: `Witness-Berechnung`

Gegeben sei ein Muster P der Dimension $n \times n$. Die Spalten seien mit $j = 0...m-1$ und die Zeilen mit $i = 0...m-1$ durchnummeriert.

Für jede Spalte $j = 0...m-1$:

1. Sei w_i das Präfix der Zeile i von P mit der Länge $m - j$. Dann ist W diejenige Zeichenkette, die durch aneinander hängen von $w_0...w_{m-1}$ entsteht. Hierbei seien die w_i nicht als einzelne Zeichenketten, sondern als jeweils ein Zeichen zu interpretieren.

2. Sei t_i das Suffix der Zeile i von P mit der Länge $m - j$. Dann ist T diejenige Zeichenkette, die durch aneinander hängen von $t_0...t_{m-1}$ entsteht. Hierbei seien die t_i nicht als einzelne Zeichenketten, sondern als jeweils ein Zeichen zu interpretieren.

3. Bestimme nun die Tabelle `max_prefix` für die Präfixe von W innerhalb von T. Beachte hierbei, dass die w_i und t_i jeweils als Zeichen und nicht als Zeichenkette zu betrachten sind!

4. Für jede Zeile i von P:

   (a) Wenn $\texttt{max_prefix}[i] = m - i$ gilt, dann ist eine Position gefunden, an der die Muster übereinstimmen und es wird $TOP - WITNESS[i, j] := (m, m)$ gesetzt.

   (b) Ansonsten:
   Sei l die Länge des gemeinsamen Präfixes von $T_{i+\texttt{max_prefix}[i]}$ und $W_{\texttt{max_prefix}[i]}$, nun als Zeichenketten und nicht mehr als einzelne Zeichen interpretiert.
   Dann gilt $TOP - WITNESS[i, j] = (\texttt{max_prefix}[i], l)$.

Zur Berechnung der $BOTTOM - WITNESS$ benutzt man eine nahezu identischen Algorithmus. Hier werden W und T mit $W = w_{m-1}w_{m-2}...w_0$ und $T = t_{m-1}t_{m-2}...t_0$ initialisiert. Ist eine Fehlerstelle gefunden, so gilt $BOTTOM - WITNESS[i, j] = (m - \texttt{max_prefix}[i] - 1, l)$.

⊚

Erklärung:
Das Muster wird zu Beginn des Algorithmus zweimal erstellt und deckungsgleich übereinander gelegt. In jedem Durchlauf wird es anschließend um je eine Position weiter nach rechts verschoben (j wird um eins erhöht).
Die w_i und die t_i sind jeweils diejenigen Abschnitte des Muster, welche sich ohne vertikale Verschiebung (d.h. $i = 0$) überlappen.

Aus diesen Zeilenabschnitten werden nun die Zeichenketten W und T gebildet und `max_prefix` berechnet. In Abbildung 10 auf Seite 20 würden sich so diese Ketten ergeben:

$i =$	0	1	2	3	4	5
$W =$	`X  X`	`X`	`XO`	`X  X`	`X`	`X`
$T =$	`X`	`X`	`XO X`	`X`	`X`	`X  X`
`max_prefix`$[i] =$	0	0	0	0	0	1

Nun wird das Suchmuster nach unten verschoben (d.h. i erhöht). Dabei gelangen die letzten i Zeilen aus W und die ersten i Zeilen aus T aus dem Bereich der Überdeckung heraus.
`max_prefix` gibt an, wie viele der ersten Zeilenabschnitte von $t_i, t_{i+1}, \ldots, t_{m-1}$ mit den Zeilenabschnitten $w_0, w_1, \ldots, w_{m-1}$ übereinstimmen.
Gilt `max_prefix`$[i] = m - i$, so stimmt der Suffix von T ab t_i mit W überein, die beiden Muster sind also an der aktuellen Position deckungsgleich. Nach der Definition der Witness-Tabelle wird somit TOP-WITNESS[i,j] auf (m,m) gesetzt.
Ansonsten ist `max_prefix`$[i]$ die erste Zeile, ab der die sich überlappenden Bereiche der beiden Muster nicht mehr übereinstimmen und kann als horizontaler Anteil für diese Position in die Witness-Tabelle eingetragen werden. Der vertikale Anteil ergibt sich genau zur Position l des ersten Zeichens, welches innerhalb dieser Zeile im Bereich der Überdeckung vom zweiten Muster abweicht.

Sei zum Beispiel $j = 2$ und $i = 2$ (siehe Abbildung 10 auf Seite 20).
Dann gilt `max_prefix`$[i] = 0 \neq 4 = m - i$. Also gibt es gleich bei Zeile 0 im verschobenen Suchmuster eine Abweichung. Der überlappende Anteil dieser Zeile ist “XO X”, der korrespondierende Anteil des zweiten Suchmusters ist “X X”. Der gemeinsame Präfix dieser Zeichenfolgen ist “X”, sie unterscheiden sich also ab Position $l = 1$. Somit ist TOP-WITNESS[2,2]=(0,1).

Die Berechnung der Bottom-Witness erfolgt nach dem gleichen Prinzip. Da hier jedoch das zweite Suchmuster nach oben verschoben wird ergeben sich die im Algorithmus genannten Abweichungen.

Theorem 4:
Der Algorithmus benötigt eine Zeit von $O(m^2 log\sigma)$.

Beweis:
Der Algorithmus `max_prefix` benötigt eine Zeit von $O(m)$. Hier wird allerdings davon ausgegangen, dass der Vergleich von zwei Zeichen in konstanter Zeit erfolgt.
Um diese Voraussetzung zu erfüllen, kann man zu Beginn des Witness-

Algorithmus einen Suffix-Baum aus den aneinander gehängten Reihen des Suchmusters erstellen[23] und diesen für die Suche von kleinsten gemeinsamen Vorfahren (*least common ancestor,* LCA) vorbereiten. Der LCA zweier Zeichenketten ist gleichzeitig deren längstes gemeinsames Präfix[19], ist er genauso lang wie die Ketten, so sind diese folglich identisch.
Der Aufbau eines solchen Suffix-Baums kann in $O(m^2 log\sigma)$ erfolgen. Die Vorbereitung auf LCA-Abfragen ist in einer linearen Zeit möglich, die Abfrage der Vorfahren geschieht dann wie benötigt in konstanter Zeit[16].
Die Suche nach der Länge des längsten gemeinsamen Präfix (=LCA) im vierten Schritt kann mit diesem Verfahren ebenfall in konstanter Zeit erfolgen.

Insgesamt wird der Suffix-Baum nur einmal aufgebaut, der Algorithmus `max_prefix` wird für jede der m-Spalten je einmal benötigt.
Die Witness-Berechnung wird einmal für die Tabelle TOP-WITNESS und einmal für die Tabelle BOTTOM-WITNESS durchgeführt.
Es folgt die angegebene Laufzeit. □

Anmerkung:
Insgesamt ergeben sich für das Muster aus Abbildung 10 auf Seite 20 die folgenden Witness-Tabellen:

$TOP - WITNESS:$

	0	1	2	3	4	5
0	$(\mathbf{6},\mathbf{6})$	$(0,0)$	$(2,0)$	$(0,0)$	$(0,0)$	$(0,0)$
1	$(0,0)$	$(1,2)$	$(0,0)$	$(0,0)$	$(\mathbf{6},\mathbf{6})$	$(0,0)$
2	$(0,0)$	$(0,0)$	$(0,1)$	$(0,0)$	$(0,0)$	$(\mathbf{6},\mathbf{6})$
3	$(2,3)$	$(0,0)$	$(0,0)$	$(\mathbf{6},\mathbf{6})$	$(0,0)$	$(0,0)$
4	$(0,0)$	$(\mathbf{6},\mathbf{6})$	$(0,0)$	$(0,0)$	$(\mathbf{6},\mathbf{6})$	$(0,0)$
5	$(0,0)$	$(0,0)$	$(\mathbf{6},\mathbf{6})$	$(0,0)$	$(0,0)$	$(\mathbf{6},\mathbf{6})$

$BOTTOM - WITNESS:$

	0	1	2	3	4	5
0	$(\mathbf{6},\mathbf{6})$	$(5,1)$	$(5,0)$	$(2,0)$	$(5,1)$	$(5,0)$
1	$(5,1)$	$(5,0)$	$(3,1)$	$(5,1)$	$(5,0)$	$(\mathbf{6},\mathbf{6})$
2	$(5,0)$	$(4,2)$	$(5,1)$	$(5,0)$	$(\mathbf{6},\mathbf{6})$	$(4,0)$
3	$(5,3)$	$(5,1)$	$(5,0)$	$(5,0)$	$(5,1)$	$(5,0)$
4	$(5,1)$	$(5,0)$	$(\mathbf{6},\mathbf{6})$	$(5,1)$	$(5,0)$	$(\mathbf{6},\mathbf{6})$
5	$(5,0)$	$(\mathbf{6},\mathbf{6})$	$(5,1)$	$(5,0)$	$(\mathbf{6},\mathbf{6})$	$(\mathbf{6},\mathbf{6})$

2.4 Bearbeitung des Textes

Die eigentliche Suche läuft in zwei Schritten ab. Zunächst werden von allen Positionen im Text, an denen das gesuchte Muster liegen könnte, diejenigen gesucht, welche paarweise "konsistent" sind.
Konsistenz bedeutet, dass die Kandidaten in allen Positionen, in denen sie sich überlappen, identisch sind, sich also in Deckung befinden.
Im zweiten Schritt werden dann unter den verbliebenen Kandidaten diejenigen bestimmt, welche tatsächlich die gesuchten Textstellen sind.
Die Konsistenz soll nun mit den Begriffen der Witness-Theorie definiert werden:

Definition 1:

Sei T ein Text und P ein Suchmuster. P habe seinen Ursprung an der Position $T[i, j]$ im Text und sei vollständig im Text enthalten.
Dann wird diese Position von P in T <u>Kandidat</u> (in T) genannt und mit (i, j) bezeichnet.

Im Folgenden werden sowohl diese gesuchten Muster im Text als Ganzes, als auch die Positionen an denen ihr Ursprung im Text liegt, als "Kandidaten" bezeichnet. Die jeweilige Bedeutung ist dem Kontext zu entnehmen.

Definition 2:

Seien (r, c) und (x, y) zwei Kandidaten in T der Art, dass $c \leq y$, dann sind (r, c) und (x, y) <u>konsistent</u>, wenn gilt:

1. Fall $(r \leq x,\ x - r < m)$:
   $TOP - WITNESS[x - r, y - c] = (m, m)$
2. Fall $(r > x,\ r - x < m)$:
   $BOTTOM - WITNESS[r - x, y - c] = (m, m)$

In den Fällen $(r \leq x,\ x - r \geq m)$ und $(r > x,\ r - x \geq m)$ finde keine Überschneidung der Muster statt. In diesem Fall ist die Konsistenz trivialer Weise gegeben.
Sind (r, c) und (x, y) konsistent, so wird dies mit $(r, c) \sim (x, y)$ bezeichnet, ansonsten mit $(r, c) \nsim (x, y)$.

Anmerkung:

- $(|x - r|, y - c)$ ist genau die Verschiebung der beiden Muster untereinander. Ist diese gleich (m, m), so gibt es nach Definition der Witness-Tabellen keine Position innerhalb der Überlappung dieser Muster, an denen die Zeichen nicht übereinstimmen.

- Die Voraussetzung $c \leq y$ kann immer erfüllt werden, gilt sie nicht, so werden die Kandidaten vertauscht. Der Grund dieser Bedingung sind die Symmetrieüberlegungen, welche zur Einschränkung der Betrachtung von 4 auf 2 Quadranten führten (siehe Seite 11 bzw. [4]).

Offensichtlich ist es von Vorteil in möglichst wenigen Schritten möglichst viele inkonsistente Kandidaten zu eliminieren. Der Algorithmus nutzt hierzu die Transitivität der Konsistenz aus:

Lemma 1: *top-transitiv*
Für alle $1 \leq r_1 \leq r_2 \leq r_3 \leq n$ und für alle $1 \leq c_1 \leq c_2 \leq c_3 \leq n$ gilt:
Gilt $(r_1, c_1) \sim (r_2, c_2)$ und $(r_2, c_2) \sim (r_3, c_3)$,
dann gilt auch $(r_1, c_1) \sim (r_3, c_3)$.

Beweis:
Angenommen unter den Voraussetzungen des Satzes gelte $(r_1, c_1) \not\sim (r_3, c_3)$.
Dann gibt es ein $x \leq m - r_3 + r_1$ und ein $y \leq m - c_3 + c_1$ der Art, dass $P[x, y] \neq P[x + r_3 - r_1, y + c_3 - c_1]$. $(\star)$
Es gilt außerdem $r_3 \geq r_2$.
Hieraus folgt $x + r_3 \geq r_2$ und $m \geq x + r_3 - r_1 \geq r_2 - r_1$.
Andererseits gilt $m \geq y + c_3 - c_1 \geq c_2 - c_1$.
Da $(r_1, c_1) \sim (r_2, c_2)$ folgt $P[x+r_3-r_1, y+c_3-c_1] = P[x+r_3-r_2, y+c_3-c_2]$.
Mit der gleichen Überlegung folgt $P[x, y] = P[x + r_3 - r_2, y + c_3 - c_2]$ aus $(r_3, c_3) \sim (r_2, c_2)$.
Insgesamt ergibt sich somit $P[x, y] = P[x + r_3 - r_1, y + c_3 - c_1]$.
Dies ist ein Widerspruch zu $(\star)$. Es folgt die Behauptung. □

Lemma 2: *bottom-transitiv*
Für alle $1 \leq r_1 \leq r_2 \leq r_3 \leq n$ und für alle $1 \leq c_3 \leq c_2 \leq c_1 \leq n$ gilt:
Gilt $(r_1, c_1) \sim (r_2, c_2)$ und $(r_2, c_2) \sim (r_3, c_3)$,
dann gilt auch $(r_1, c_1) \sim (r_3, c_3)$.

Beweis:
Analog zum vorherigen Lemma. □

2.4.1 Eindimensionaler Konsistenz-Algorithmus

Zunächst soll ein Algorithmus vorgestellt werden, der die Positionen aller konsistenten Kandidaten ermittelt, welche in einer gegebenen Spalte ihren Ursprung haben.

Algorithmus E: `col_cons_check(c)`

Gegeben sei ein Suchmuster P mit berechneter Witness-Tabelle und ein Text T. Das Suchmuster habe die Größe m×m, der Text habe die Größe n×n.
Sei c die Spalte von T, für die eine doppelt verkettete Liste S konsistenter Positionen von T bestimmt werden soll.
S sei mit der Position $(n-m, c)$ initialisiert.

Für alle Zeilen $r = n - m - 1...0$:

1. (x, c) sei der letzte Eintrag in S.
2. Überprüfe (r, c) und (x, c) auf Konsistenz.
3. Bei Konsistenz füge (r, c) am Ende von S ein.
4. Ansonsten:
   (a) Ermittle mit Hilfe der Witness-Tabellen ein Zeichen welches in (r, c) und (x, c) nicht übereinstimmt.
   (b) Vergleiche diese Zeichen mit dem korrespondierenden Zeichen im Text.
   (c) Stimmt (x, c) an dieser Stelle nicht mit dem Text überein, dann lösche (x, c) aus S.
   (d) Ist S nun leer, dann füge (r, c) ein. Wenn nicht, dann vergleiche (r, c) an dieser Stelle mit dem Text. Stimmen die Zeichen überein, dann fahre mit demselben r, aber dem neuen letzten Element von S bei Schritt 1 fort.

Gib S zurück.

◎

Erklärung:
Das Suchmuster wird am "unteren Ende" der Spalte c im Text positioniert. Da das Muster m und der Text n Zeichen hoch ist, entspricht dies genau dem Wert $n-m$. Da ein einzelner Kandidat zu sich selbst trivialer Weise konsistent ist, wird S entsprechend mit $(n-m, c)$ initialisiert.
Ein Muster wird nun an die letzte Position aus S gelegt. Ein zweites Muster

wird eine Zeile darüber positioniert und in jedem Schritt (also bei jeder Erniedrigung von r) um eine Zeile weiter nach oben verschoben.
Sind die beiden Muster konsistent, so ist das zweite Muster wegen der Transitivität der Konsistenz auch mit allen anderen Elementen aus S konsistent. Es kann also in S eingefügt werden.
Anderenfalls ist mindestens eines der Muster an einer Position, in der es keine Fundstelle im Text repräsentiert.
Durch die Witness-Tabelle ist eine Position bekannt, an der sich die beiden Muster in ihrem überlappenden Bereich unterscheiden. Die entsprechenden Zeichen werden mit dem Text an der entsprechenden Stelle verglichen.
Passt der Text nicht zum ersten Muster, so wird dessen Position aus S gelöscht, da es sich in keinem Fall um eine Fundstelle handeln kann. Damit ist jedoch nicht automatisch der zweite Kandidat mit den anderen Einträgen aus S inkonsistent. Mit diesem Kandidaten und der neuen letzten Position aus S wird nun bei Schritt 1 fortgefahren, um seine Konsistenz mit den übrigen Kandidaten aus S zu testen.
Passt das Zeichen aus dem ersten Muster zum Text, so kann der neue Kandidat (r, c) keine Fundstelle im Text sein. Das zweite Muster wird nun eine weitere Zeile nach oben verschoben (r wird verringert).

Wenn S im Laufe dieses Verfahrens keine Elemente mehr enthält, so wird es mit dem Muster an der aktuellen Position (r, c) initialisiert. Dieses ist trivialer Weise mit sich selbst konsistent, die Eigenschaft von S nur konsistente Elemente zu enthalten bleibt damit erhalten.

Anmerkung:

- Sind zwei Kandidaten inkonsistent, so können nicht beide eine Fundstelle im Text darstellen. Durch den Abgleich mit dem Text über die Witness-Tabellen wird in jedem Fall ein Kandidat entfernt, welcher keine Fundstelle ist. Auf diese Weise wird nie eine Fundstelle gelöscht!

- Die Konsistenz der Positionen wird mit Hilfe von Definition 2 auf Seite 24 bestimmt.
  Da in diesem Algorithmus immer $y = c$ und $r \leq x$ gilt, reduziert sich die Abfrage auf:
  $$(r, c) \sim (x, c) \Leftrightarrow TOP - WITNESS[x - r, 0] = (m, m)$$
  Dabei ist besonders zu beachten, dass die Kandidaten trivialer Weise konsistent sind, wenn $x - r \geq m$.

- Im Fall $(r,c) \not\sim (x,c)$ ergeben sich die nicht korrespondieren Zeichen zu:
  $P[i,j]$ für das Zeichen im Muster an der Position (x,c)
  $P[(x-r)+i, 0+j]$ für das Zeichen im Muster an der Position (r,c)
  $T[x+i, c+j]$ für das Zeichen im Text
  mit $(i,j) := TOP-WITNESS[x-r, 0]$

- Sind zwei Kandidaten bei diesem Algorithmus um mehr als m Zeilen voneinander entfernt, so überschneiden sie sich nicht und sind trivialer Weise konsistent. Der untere Kandidat wird also in S gelangen und ist mit jedem weiteren (höherem) Kandidaten ebenso trivial konsistent. Sämtliche Konsistenztests werden sich in diesem Fall nur noch zwischen Positionen oberhalb des unteren Kandidaten vollziehen.

- Die doppelt verlinkte Liste ist notwendig um die Laufzeit des Algorithmus zu senken.

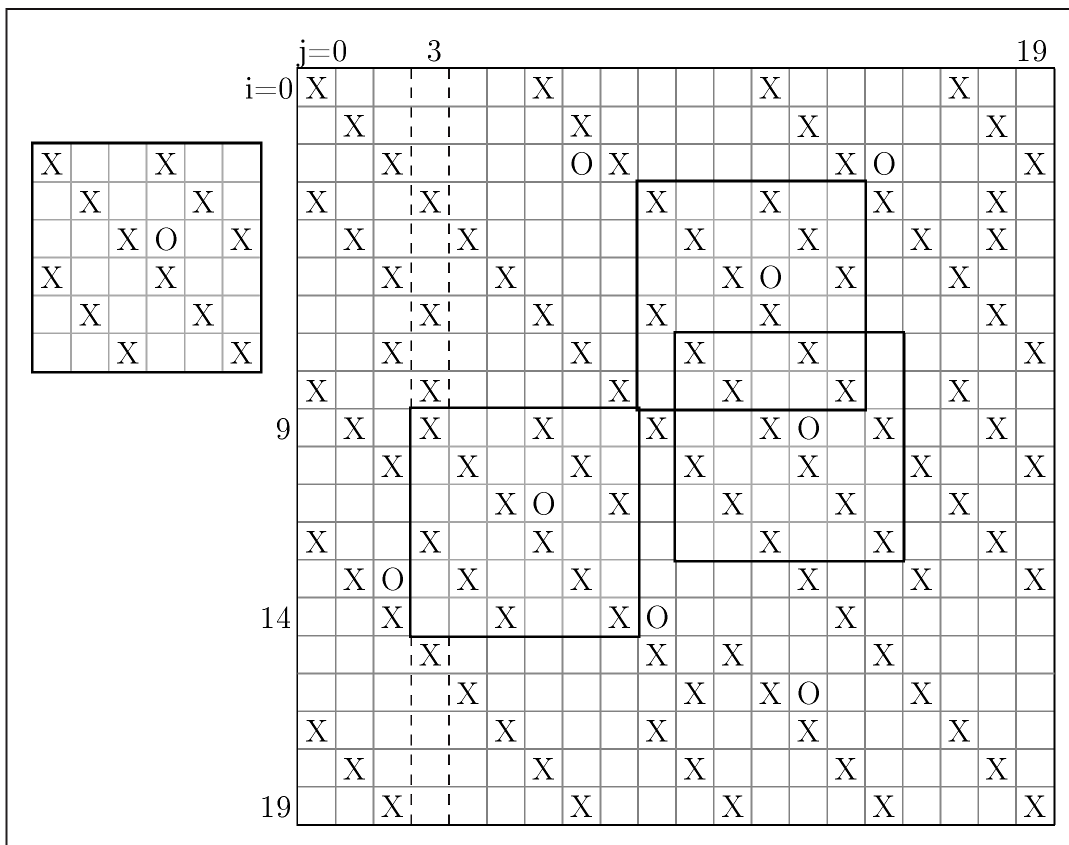


Abbildung 11: Beispielmuster und -text

Beispiel 1:
Beispielsweise soll nun S für Spalte 3 bestimmt werden:
Sei $c = 3$. Seien P und T wie in Abbildung 11 auf der vorherigen Seite gegeben.
Die Witness-Tabellen von P wurden bereits berechnet (Seite 23).
$n = 20$, $m = 6$, $\Rightarrow m - n = 20 - 6 = 14$. Also $S := (m - n, c) = (14, 3)$.

$\underline{r = 13}$ $(= m - n - 1)$:
$(r, c) = (13, 3)$, $(x, c) = (14, 3)$
$TOP - WITNESS[x - r, 0] = TOP - WITNESS[1, 0] = (0, 0) =: (i, j)$
$\Rightarrow (r, c) \not\sim (x, c)$
korrespondierendes Witness-Zeichen in (x, c): $P[i, j] = P[0, 0] =$“X”
korrespondierendes Zeichen in (r, c): $P[(x - r) + i, 0 + j] = P[1, 0] =$“ ”
korrespondierendes Zeichen im Text: $T[x + i, c + j] = T[14, 3] =$“ ”
Das Zeichen aus dem Muster bei (x, c) und das Zeichen im Text stimmen nicht überein. Lösche (x, c) aus S.
S ist nun leer, füge (r, c) ein und fahre mit dem nächsten r fort.
$S = (13, 3)$

$\underline{r = 12}$:
$(r, c) = (12, 3)$, $(x, c) = (13, 3)$
$TOP - WITNESS[1, 0] = (0, 0) =: (i, j) \Rightarrow (r, c) \not\sim (x, c)$
korrespondierendes Witness-Zeichen in (x, c): “X”
korrespondierendes Witness-Zeichen in (r, c): “ ”
korrespondierendes Zeichen im Text: “ ”
Lösche (x, c) aus S.
S ist nun leer, füge (r, c) ein und fahre mit dem nächsten r fort.
$S = (12, 3)$

$\underline{r = 11}$:
$(r, c) = (11, 3)$, $(x, c) = (12, 3)$
$TOP - WITNESS[1, 0] = (0, 0) =: (i, j) \Rightarrow (r, c) \not\sim (x, c)$
korrespondierendes Witness-Zeichen in (x, c): “X”
korrespondierendes Witness-Zeichen in (r, c): “ ”
korrespondierendes Zeichen im Text: “X”
Das Zeichen aus dem Muster bei (x, c) und das Zeichen im Text stimmen überein, das Zeichen aus dem Muster bei (r, c) stimmt dementsprechend nicht mit dem Text überein. Fahre mit dem nächsten r fort.
$S = (12, 3)$

$\underline{r = 10}$:
$(r, c) = (10, 3)$, $(x, c) = (12, 3)$
$TOP - WITNESS[2, 0] = (0, 0) =: (i, j) \Rightarrow (r, c) \not\sim (x, c)$
korrespondierendes Witness-Zeichen in (x, c): “X”

korrespondierendes Witness-Zeichen in (r, c): “ ”
korrespondierendes Zeichen im Text: “X”
Fahre mit dem nächsten r fort.
$S = (12, 3)$

<u>$r = 9$:</u>
$(r, c) = (9, 3)$, $(x, c) = (12, 3)$
$TOP - WITNESS[3, 0] = (2, 3) =: (i, j) \Rightarrow (r, c) \not\sim (x, c)$
korrespondierendes Witness-Zeichen in (x, c): “O”
korrespondierendes Witness-Zeichen in (r, c): “ ”
korrespondierendes Zeichen im Text: “ ”
Lösche (x, c) aus S.
S ist nun leer, füge (r, c) ein und fahre mit dem nächsten r fort.
$S = (9, 3)$

Bei den folgenden Durchläufen ändert sich der Inhalt von S nicht, deshalb nun weiter bei $r = 3$:

$S = (9, 3)$
<u>$r = 3$:</u>
$(r, c) = (3, 3)$, $(x, c) = (9, 3)$
Nun müsste der Witness-Eintrag für [6,0] ermittelt werden. Da das Muster jedoch nur 6x6 Zeichen groß ist, ist dieser Wert nicht definiert, trivialer Weise gilt hier jedoch $(r, c) \sim (x, c)$.
Also wird (r, c) am Ende von S eingefügt.
$S = (9, 3) \rightarrow (3, 3)$

Dieser Wert bleibt bis zum letzten Schritt erhalten:
$S = (9, 3) \rightarrow (3, 3)$
<u>$r = 0$:</u>
$(r, c) = (0, 3)$, $(x, c) = (3, 3)$
$TOP - WITNESS[3, 0] = (2, 3) =: (i, j) \Rightarrow (r, c) \not\sim (x, c)$
korrespondierendes Witness-Zeichen in (x, c): “O”
korrespondierendes Witness-Zeichen in (r, c): “ ”
korrespondierendes Zeichen im Text: “ ”
Lösche (x, c) aus S. S ist nicht leer, weiter bei Schritt 1.
$S = (9, 3)$

$(r, c) = (0, 3)$, $(x, c) = (9, 3)$
Trivialer Weise sind nun wiederum (r, c) und (x, c) konsistent.
Also wird (r, c) am Ende von S eingefügt.
$S = (9, 3) \rightarrow (0, 3)$

Also sind für $c = 3$ in diesem Beispiel $(9, 3)$ und $(0, 3)$ zwei konsistente Kandidaten.
Später wird sich $(9, 3)$ als eine Position des Musters im Text herausstellen. ⌟

Theorem 5:
Der Algorithmus `col_cons_check` ist korrekt und benötigt eine Zeit von $O(n)$.

Beweis:
Die Korrektheit ist aus den Erklärungen in Verbindung mit Lemma 1 auf Seite 25 ersichtlich.
Zur Laufzeit:
S kann in konstanter Zeit initialisiert werden. Jeder Konsistenztest erfolgt über die Witness-Tabellen in konstanter Zeit. Für jede Zeile r in der Schleife gibt es höchstens einen erfolgreichen Test auf Konsistenz. Für jeden negativen Test wird entweder der aktuelle Kandidat (r, c) oder der oberste Kandidat aus S entfernt. Da die Anzahl der Kandidaten durch n beschränkt ist (streng genommen $n - m$), folgt die Gesamtlaufzeit zu $O(n)$. □

2.4.2 Zweidimensionaler Konsistenz-Algorithmus

Nachdem es nun möglich ist innerhalb einer Spalte vom Text T alle konsistenten Kandidaten des Musters P zu ermitteln, muss der Algorithmus erweitert werden, um nur jene Kandidaten zu erhalten, welche auch horizontal betrachtet paarweise konsistent sind.

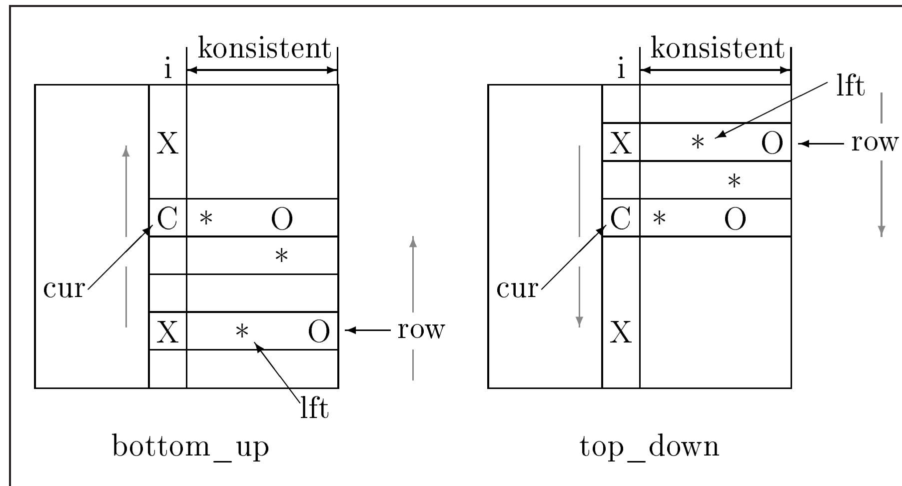


Abbildung 12: Bottom Up und Top Down

Die Idee:
Es sei die Situation von Abbildung 12 gegeben.
Es gelte folgende Invariante für $j > i$:

$$P(j) \equiv \text{ die nicht gelöschten Kandidaten in den Spalten } j, ..., n-1 \text{ sind paarweise konsistent}$$

Die Kandidaten, welche sich im Text auf der rechten Seite der aktuellen Spalte i befinden, sind also alle konsistent.
Nun soll die Spalte i hinzugenommen werden. Die Kandidaten innerhalb dieser Spalte (X bzw. C) sind bereits mit dem Algorithmus `col_cons_check` als vertikal konsistent identifiziert.
Nun muss überprüft werden, ob diese Kandidaten auch mit allen Kandidaten des rechten Bereichs konsistent sind. Aufgrund der Transitivität der Konsistenz gilt:
Ist ein Kandidat C aus der zu überprüfenden Spalte mit dem linken Kandidaten $*$ einer Zeile aus dem schon verifizierten Bereich konsistent, so gilt diese Eigenschaft für alle Kandidaten O aus dieser Zeile. Es genügt also jeweils der Vergleich mit einem Element einer Zeile. Im Algorithmus wird hierzu das Element benutzt, "welches am weitesten links" liegt, also den kleinsten Spaltenindex besitzt.

Sind C und $*$ nicht konsistent, so kann nach einem Abgleich mit dem zugrunde liegenden Text (wie schon bei `col_cons_check`) entweder der neue Kandidat C oder aber $*$ entfernt werden.
Wird $*$ entfernt, so muss der Test mit dem nächsten Element aus dieser Spalte wiederholt werden. Wird C entfernt, so kann man mit dem nächsthöheren Kandidaten X in der aktuellen Spalte fortfahren.
Nach diesem Prinzip kann gewährleistet werden, dass jeder Kandidat aus der neuen Spalte mit jedem Kandidaten aus dem schon konsistenten Bereich **unterhalb** seiner eigenen Position konsistent ist. Diese Überlegungen führen zum Algorithmus `bottom_up`.
Es fehlt noch die Konsistenz der neuen Elemente mit allen darüber liegenden Kandidaten im konsistenten Bereich. Dies leistet der Algorithmus `top_down`, der im Gegensatz zu `bottom_up` die neuen Kandidaten und die Zeilen aus dem konsistenten Bereich von oben nach unten durchläuft.

Es folgt nun der eigentliche Algorithmus zur Ermittlung aller konsistenten Kandidaten in einem zweidimensionalen Text. Dieser durchläuft den Text von rechts nach links und fügt jede neue Spalte wie beschrieben mit Hilfe von `bottom_up` und `top_down` und unter Einhaltung der Invariante P hinzu.

Algorithmus F: `cons_check`

Für alle $i = 0...n-m$ sei $C_i := col_cons_check(i)$ die Liste aller Kandidaten in der Spalte i.
Für alle $j = 0...n-m$ initialisiere eine leere doppelt verkettete Liste R_j zur Aufnahme von Kandidaten in der Zeile j.
Sortiere alle Kandidaten aus der Liste C_{n-m} in die Listen R_j ein.

Für alle $i = n - m - 1...0$:

1. `bottom_up(i)`
2. `top_down(i)`
3. Sortiere alle überlebenden Kandidaten aus C_i entsprechend ihrer Zeilenkomponente in die Listen R_j ein.

◎

Anmerkung:

- Die Spaltenkomponente von C_i ist trivialer Weise immer i, die Zeilenkomponente von R_j immer j.
- Durch `bottom_up(i)` und `top_down(i)` werden, wie oben beschrieben, alle Kandidaten eliminiert, welche nicht konsistent zu den über bzw. unter ihnen befindlichen Zeilen sind.

- In den R_j befinden sich durch diesen Algorithmus nur Kandidaten, welche rechts der aktuellen Spalte i liegen.
- Spalten und Zeilen größer als $n - m$ bleiben unberücksichtigt, da hier keine vollständige Überdeckung durch das Muster möglich ist und daher Kandidaten an diesen Positionen keine Fundstellen sein können. Sie sind nicht mehr vollständig im Text enthalten.
- Am Ende dieses Algorithmus enthalten die C_i und die R_j nur noch paarweise konsistente Kandidaten.

Algorithmus G: `bottom_up(c)`

Der Algorithmus greift auf die Listen C_i und R_j von `cons_check` zu.
Sei cur ein Zeiger auf den untersten Eintrag von C_c.
Sei $row = n - m$ die letzte verglichene Zeile.
Solange cur existiert:

1. Sei lft ein Zeiger auf den Kandidaten in R_{row} mit der kleinsten Spaltenkomponente.
2. Sind cur und lft konsistent, oder ist R_{row} leer, dann fahre mit der nächsten Zeile fort: $row := row - 1$.
3. Sind sie nicht konsistent, dann
   (a) finde eine Position (i, j) an der die Muster nicht übereinstimmen (aus der Witness-Tabelle)
   (b) vergleiche die entsprechenden Zeichen aus cur und lft mit dem korrespondierenden Zeichen im Text um herauszufinden, welches dieser beiden Muster in jedem Fall nicht als Fundstelle in Frage kommt.
      i. Kommt lft nicht in Frage, dann entferne diesen Eintrag aus den entsprechenden Listen C_i und R_j.
      ii. Kommt cur nicht in Frage, dann entferne cur aus C_c, setze cur auf den nächsthöheren Eintrag in C_c.
4. Wenn row kleiner als die Zeilenkomponente von cur ist (also row "über" cur), dann setze cur auf den nächsthöheren Eintrag von C_c. Gibt es keinen höheren Eintrag, so existiert cur nicht und der Algorithmus bricht ab.

⊚

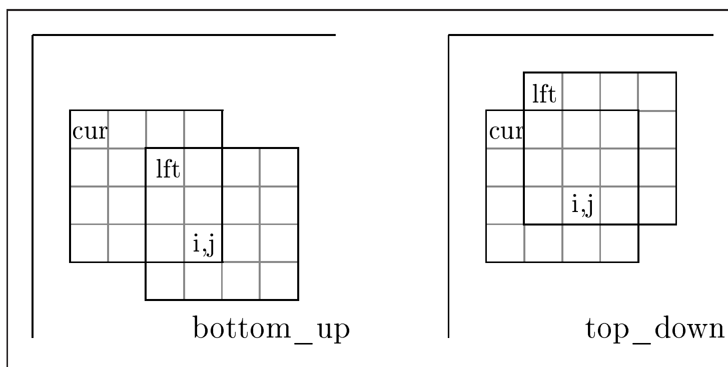


Abbildung 13: Vergleiche bei Inkonsistenz

Anmerkung:

- Die räumlichen Begriffe sind immer als Bezug des Kandidaten auf seine Position im Text zu interpretieren.

- In den R_j befinden sich durch diesen Algorithmus nur Kandidaten, welche rechts der aktuellen Spalte i liegen. lft liegt also auch immer rechts von cur.

- Der Konsistenztest erfolgt nach Definition 2 auf Seite 24. Da sich cur immer links von lft befindet vereinfacht sich die Abfrage:
  $$(c_r, c_c) \sim (l_r, l_c) \Leftrightarrow TOP-WITNESS[l_r - c_r, l_c - c_c] = (m, m)$$
  mit $(c_r, c_c) := cur$ und $(l_r, l_c) := lft$.

- Im Falle einer Inkonsistenz werden die folgenden Zeichen verglichen (vergleiche Abbildung 13):
  $P[i + (l_r - c_r), j + (l_c - c_c)]$ für das Zeichen aus dem Muster bei cur
  $P[i, j]$ für das Zeichen aus dem Muster bei lft
  $T[i + l_r, j + l_c]$ für das Zeichen im Text
  mit $(i, j) := TOP-WITNESS[l_r - c_r, l_c - c_c]$

- Wird ein Element aus R_{row} gelöscht, dann wird beim nächsten Schleifendurchgang das nächste Element aus row mit cur verglichen. Dies wird dadurch erreicht, dass in diesem Fall cur nicht weiter gesetzt wird.

- Ist lft konsistent zu cur, dann ist die ganze Zeile zu cur konsistent (Transitivität!).

- Ist eine Zeile aus dem konsistenten Bereich mit cur konsistent, dann ist sie auch mit jedem Element aus C_c oberhalb von cur konsistent (Transitivität!). Aus diesem Grund muss jede Zeile nur einmal als konsistent zu einem C_c oberhalb dieser Zeile identifiziert werden. Deshalb wird selbst bei Löschung von cur row nicht zurückgesetzt.

- Vorsicht Falle: Je nach Implementierung kann es notwendig sein das nächste *cur* zu bestimmen **bevor** man das aktuelle *cur* wegen Inkonsistenz löscht!

Der Algorithmus `top_down` funktioniert im Prinzip genauso und sei nur der Vollständigkeit halber und ohne Kommentare hier angegeben:

Algorithmus H: `top_down(c)`
Der Algorithmus greift auf die Listen C_i und R_j von `cons_check` zu.
Sei *cur* ein Zeiger auf den obersten Eintrag von C_c.
Sei $row = 0$ die letzte verglichene Zeile.

Solange *cur* existiert:

1. Sei lft ein Zeiger auf den Kandidaten von R_{row} mit der kleinsten Spaltenkomponente.
2. Sind *cur* und lft konsistent, oder ist R_{row} leer, dann fahre mit der nächsten Zeile fort: $row := row + 1$.
3. Sind sie nicht konsistent, dann
   (a) finde eine Position (i, j) an dem die Mustern nicht übereinstimmen (aus der Witness-Tabelle)
   (b) vergleiche die entsprechenden Zeichen aus *cur* und lft mit dem korrespondierenden Zeichen im Text um herauszufinden, welches dieser beiden Muster in jedem Fall nicht als Fundstelle in Frage kommt.
      i. Kommt lft nicht in Frage, dann entferne diesen Eintrag aus den entsprechenden Listen C_i und R_j.
      ii. Kommt *cur* nicht in Frage, dann entferne *cur* aus C_c, setze *cur* auf den nächsttieferen Eintrag in C_c.
4. Wenn *row* größer als die Zeilenkomponente von *cur* ist, dann setze *cur* auf den nächsttieferen Eintrag von C_c.
   Gibt es keinen tieferen Eintrag so existiert *cur* nicht und der Algorithmus bricht ab.

◎

Anmerkung:
Es gelten äquivalente Überlegungen wie zu `bottom_up`. Da sich jedoch lft immer oberhalb von *cur* befindet, muss für den Konsistenzvergleich
$BOTTOM-WITNESS[c_r - l_r][l_c - c_c] =: (i, j)$ betrachtet werden.

Im Falle einer Inkonsistenz sind dann folgende Zeichen zu vergleichen (Abbildung 13 auf Seite 35):

$P[i-(c_r-l_r), j+(l_c-c_c)]$ für das Zeichen aus dem Muster bei *cur*
$P[i,j]$ für das Zeichen aus dem Muster bei *lft*
$T[i+l_r, j+l_c]$ für das Zeichen im Text

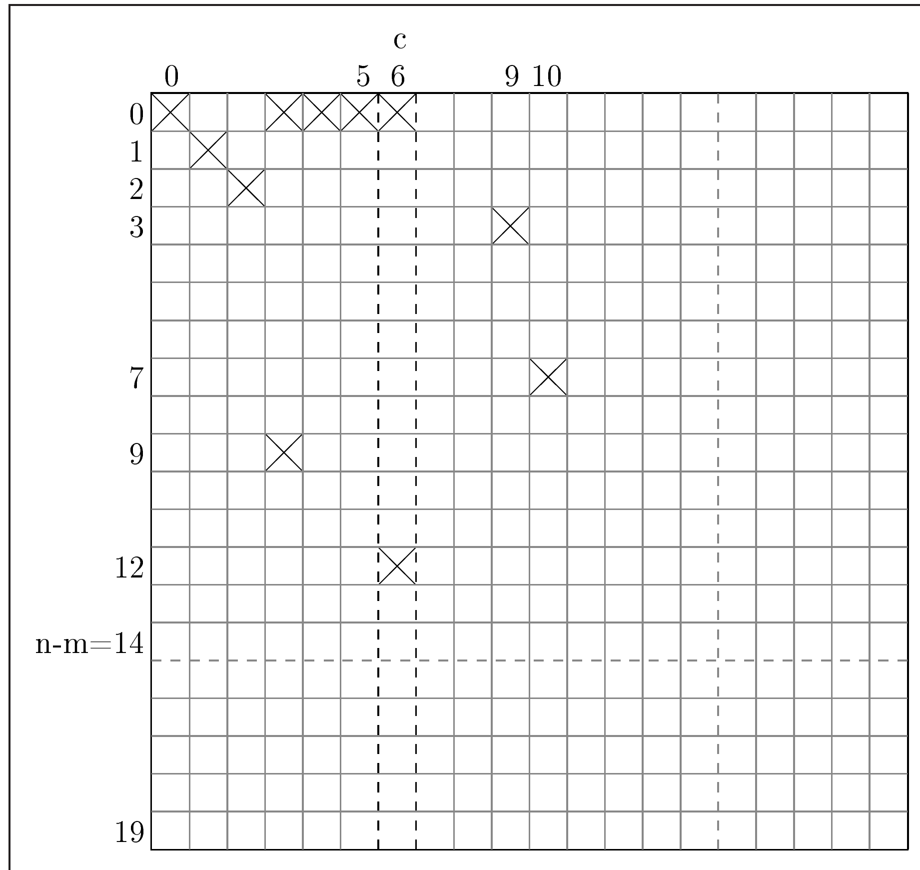


Abbildung 14: Der Fall c=6

Beispiel 2:
Für die Suche im Beispiel 11 auf Seite 28 ergeben sich zu Beginn des Algorithmus die folgenden Listen:

$C_0 \to (0,0)$	$C_1 \to (1,1)$	$C_2 \to (2,2)$
$C_3 \to (0,3) \to (9,3)$	$C_4 \to (0,4)$	$C_5 \to (0,5)$
$C_6 \to (0,6) \to (12,6)$	$C_7 \to (1,7)$	$C_8 \to (2,8)$
$C_9 \to (3,9)$	$C_{10} \to (0,10) \to (7,10)$	$C_{11} \to (0,11)$
$C_{12} \to (0,12)$	$C_{13} \to (1,13)$	$C_{14} \to (2,14)$

$R_2 \to (2,14)$
Alle anderen Listen sind leer.

Angenommen die ersten Durchläufe des Algorithmus sind bereits erfolgt, als nächstes soll Spalte 6 bearbeitet werden. Es liegt die Situation aus Abbildung 14 auf der vorherigen Seite vor, in den Listen sind die folgenden Werte enthalten:

$C_0 \to (0,0)$	$C_1 \to (1,1)$	$C_2 \to (2,2)$
$C_3 \to (0,3) \to (9,3)$	$C_4 \to (0,4)$	$C_5 \to (0,5)$
$C_6 \to (0,6) \to (12,6)$	$C_9 \to (3,9)$	$C_{10} \to (7,10)$
$R_3 \to (3,9)$	$R_7 \to (7,10)$	

1. Phase: `bottom_up`

$cur \to (12,6)$
$row := n - m = 20 - 6 = 14$
R_{14} ist leer, setze row herab $\to R_{13}$ leer $\to \cdots \to R_{11}$.
Nun steht $row = 11$ oberhalb von cur (Zeile 12). cur wird also auf den nächsthöheren Eintrag von C_6 in der aktuellen Spalte gesetzt:
$cur \to (0,6)$
R_{11} leer $\to \cdots \to R_7$.
R_7 ist nicht leer, der am weitesten links liegende (und einzige) Eintrag von R_7 ist $(7,10)$, setze $lft := (7,10)$.
Prüfe auf Konsistenz: $(0,6) \overset{?}{\sim} (7,10)$
Die Kandidaten sind trivialer Weise konsistent, fahre mit der nächsten row fort: $row = 6$.
R_6 leer $\to \cdots \to R_3$.
$lft := (3,9)$
Prüfe auf Konsistenz: $(0,6) \overset{?}{\sim} (3,9)$
Die Kandidaten sind konsistent, fahre mit der nächsten row fort: $row = 5$.
Alle weiteren R_{row} sind leer, bei $row := -1$ bricht der Algorithmus ab, da hier row wieder über cur liegt. Nun müsste der nächste cur aus C_6 gewählt werden, es gibt jedoch keinen weiteren Eintrag in C_6.

2. Phase: `top_down`

$cur \to (0,6)$
$row := 0$
R_0 leer $\to R_1$.
Damit ist row unterhalb von cur, setze cur auf den nächsttieferen Eintrag von C_6:
$cur \to (12,6)$
R_1 leer $\to \cdots \to R_3$.
R_3 ist nicht leer, der am weitesten links liegende (und einzige) Eintrag von R_3 ist $(3,9)$, setze $lft := (3,9)$.
Prüfe auf Konsistenz: $(12,6) \overset{?}{\sim} (3,9)$
Die Kandidaten sind konsistent, fahre mit der nächsten row fort: $row = 4$.
R_7 leer $\to \ldots \to R_7$.

$lft := (7, 10)$
Prüfe auf Konsistenz: $(12, 6) \overset{?}{\sim} (7, 10)$
Die Kandidaten sind konsistent, fahre mit der nächsten row fort: $row = 8$.
Alle weiteren R_{row} sind leer, bei $row := 13$ bricht der Algorithmus ab, da hier row unterhalb von cur liegt. cur müsste nun auf den nächsten Eintrag von C_6 gesetzt werden, dieser existiert jedoch nicht.

3. Phase: Einsortierung

Da beide Kandidaten in C_6 konsistent waren und nicht gelöscht wurden, werden sie nun in die R_j einsortiert. Die Listen C_i bleiben unverändert, die R_j ergeben sich nun zu:
$R_0 \rightarrow (0, 6)$
$R_3 \rightarrow (3, 9)$
$R_7 \rightarrow (7, 10)$
$R_{12} \rightarrow (12, 6)$

Fahre nun mit dem nächsten i fort:
$i := 5$

1. Phase: `bottom_up`

$cur \rightarrow (0, 5)$
$row := 14$
R_{14} leer $\rightarrow \cdots \rightarrow R_{12}$.
$lft := (12, 6)$
Prüfe auf Konsistenz: $(0, 5) \overset{?}{\sim} (12, 6)$
Die Kandidaten sind konsistent, fahre mit der nächsten row fort: $row = 11$.
R_{11} leer $\rightarrow \cdots \rightarrow R_7$.
$lft := (7, 10)$
Prüfe auf Konsistenz: $(0, 5) \overset{?}{\sim} (7, 10)$
Die Kandidaten sind konsistent, fahre mit der nächsten row fort: $row = 6$.
R_6 leer $\rightarrow \cdots \rightarrow R_3$.
$lft := (3, 9)$
Prüfe auf Konsistenz: $(0, 5) \overset{?}{\sim} (3, 9)$
Die Kandidaten sind nicht konsistent!
Überprüfe mit Hilfe der Witness-Tabelle ob cur oder lft nicht mit dem Text übereinstimmen, cur stellt sich als nicht übereinstimmend heraus, lft passt an dem in der Witness-Tabelle gegebenem Zeichen zum Text.
cur wird aus C_5 entfernt.
Nun wird cur auf den nächsthöheren Eintrag in C_5 gesetzt, da es keinen gibt wird der Algorithmus beendet.

In diesem Fall war der Kandidat bei cur ungültig. Da dieser noch nicht in einem vorangegangenen Schritt in die R_{row}-Listen übertragen wurde, muss er auch nur aus der Liste C_{cur} gelöscht werden. Wäre der Kandidat bei lft ungültig gewesen, so hätte dieser aus den C_{cur}- und R_{row}-Listen gelöscht werden müssen. Im nächsten Durchlauf des Algorithmus wäre dann der nächste Kandidat innerhalb von R_{row} mit C_{cur} verglichen worden.
Ein Beispiel für solch einen Fall ist der Vergleich von $(0, 12)$ mit $(2, 14)$ bei Verarbeitung von Spalte 12.

2. Phase: `top_down`
Da C_5 durch die letzte Phase nun keinen Kandidaten mehr enthält gibt es kein gültiges cur und der Algorithmus bricht sofort ab.

3. Phase: Einsortierung
Da keine neuen Kandidaten gefunden wurden (C_5 ist nun leer!) findet keine Einsortierung von neuen Elementen in R_i statt. Es ist aber zu beachten, dass mittlerweile $(0, 5)$ aus C_5 gelöscht wurde, diese Liste ist nun also leer.

Die nächsten Schritte sollen nicht näher gezeigt werden, nach Ablauf des gesamten Algorithmus verbleiben die folgenden paarweise konsistenten Kandidaten in den beiden Listen:
$(0, 6), (2, 2), (3, 9), (7, 10), (9, 3), (12, 6)$ ⌟

Theorem 6:
Der Algorithmus ist korrekt und benötigt eine Zeit von $O(n^2)$.

Beweis:
Wie in Algorithmus `col_cons_check` werden nur Kandidaten gelöscht, welche nach einem Abgleich mit dem Text nicht als Fundstelle in Frage kommen. Es werden also explizit nie Fundstellen gelöscht.

Um zu zeigen, dass nach Ablauf des Algorithmus nur paarweise konsistente Kandidaten in R_j (und somit auch in C_i) verbleiben, seien zunächst (r_1, c_1) und (r_2, c_2) zwei willkürlich gewählte Kandidaten aus R_j mit oBdA $c_1 < c_2$.
1. Fall: $r_1 \leq r_2$
Beweis per vollständiger Induktion:
Es gilt die Invariante $P(c_1)$.
Annahme: Nach Bearbeitung der Spalte $c_1 + 1$ gilt $P(c_1 + 1)$.
Die Kandidaten innerhalb der Spalte $c_1 + 1$ sind aufgrund von Theorem 5 auf Seite 31 paarweise konsistent. Sei (r_2, c') der Kandidat mit der kleinsten Spaltenkomponente aus R_{r_2} nach Bearbeitung von Spalte c_1, der Art, dass $c' > c_1$.

Nach Lemma 1 auf Seite 25 genügt es zu zeigen, dass $(r_1, c_1) \sim (r_2, c')$, da $(r_2, c') \sim (r_2, c_2)$.
Sei (r', c_1) der letzte Kandidat, mit dem (r_2, c') innerhalb von `bottom_up(`c_1`)` verglichen wurde.

Behauptung: $r' \geq r_1$ und $(r', c_1) \sim (r_2, c')$.
Beweis:
Angenommen $(r', c_1) \not\sim (r_2, c')$. Dann wird entweder (r', c_1) oder (r_2, c') von der Liste der Kandidaten gelöscht.
Wird (r_2, c') entfernt, so muss nun der Kandidat mit der nächsthöheren Spaltenkomponente in R_{r_2} mit (r', c_1) verglichen werden. Dies widerspricht der Annahme, dass kein Element aus c_1 mit einem Kandidaten aus R_{r_2} verglichen wurde, der eine größere Spaltenkomponente als (r_2, c') hat.
Wird (r', c_1) eliminiert, dann muss nun (r_2, c') mit dem Kandidaten aus c_1 mit der nächstniedrigeren Zeilenkomponente (also dem nächsthöheren Kandidaten) verglichen werden. Dies widerspricht jedoch der Annahme, dass (r', c_1) der letzte Kandidat in c_1 war, mit dem (r_2, c') verglichen wurde.

Um zu zeigen, dass $r' \geq r_1$ gilt, genügt die folgende Überlegung:
Wäre $r_1 > r'$, dann wäre niemals (r_2, c') mit (r', c_1) verglichen worden ohne vorher (r_1, c_1) auf Konsistenz mit (r_2, c') zu testen. Da jedoch keiner dieser Kandidaten gestrichen wurde, müssten sie konsistent gewesen sein.
Dann wären jedoch (r_2, c') und (r', c_1) nie verglichen worden. □

Damit ist gezeigt, dass $(r_1, c_1) \sim (r', c_1)$, $(r', c_1) \sim (r_2, c')$, $(r_2, c') \sim (r_2, c_2)$ gilt und außerdem $r_1 \leq r' \leq r_2$ und $c_1 \leq c' \leq c_2$ gelten muss. Mit Hilfe von Lemma 1 auf Seite 25 ist damit der Beweis für diesen Fall erbracht.

2. Fall: $r_1 > r_2$
Der Beweis erfolgt analog mit der Prozedur `top_down` anstelle von `bottom_up` und Lemma 2 auf Seite 25 anstelle von Lemma 1 auf Seite 25.

Die Komplexitätsuntersuchung erfolgt mit ähnlichen Argumenten wie im Beweis zu Theorem 5 auf Seite 31.
In `bottom_up` und `bottom_down` führt jeder Vergleich zur Entfernung eines Kandidaten oder zu einer Verschiebung des *cur*-Zeigers. Da die obere Schranke für die Anzahl von Kandidaten bei n^2 liegt (genauer: $(n-m)^2$), kann es auch höchstens n^2 Löschungen pro Prozedur geben. Eine Verschiebung von *cur* kann höchstens $O(n)$ mal pro Prozedur erfolgen (Anzahl möglicher Kandidaten pro Spalte), jede Prozedur wird $O(n)$ mal aufgerufen (mögliche Anzahl von Spalten). Insgesamt ergibt sich also die behauptete Laufzeit zu $O(n^2)$. □

2.5 Verifizierung der Fundstellen

Nun befinden sich in den Listen R_j (und genauso in C_i) nur noch Textstellen, welche paarweise konsistent sind. Die letzte Aufgabe des Algorithmus ist es nun auf effiziente Weise diejenigen Kandidaten zu ermitteln, welche auch wirklich mit dem Text übereinstimmen.
Die Konsistenzeigenschaft führt zu folgender Erkenntnis:
Wird ein Zeichen im Text von zwei Kandidaten überdeckt, so genügt es diese Stelle mit einem der Kandidaten zu vergleichen. Das Ergebnis gilt dann analog für beide Kandidaten, da sie an der betroffenen Stelle über dem Text übereinstimmen.
Zur Übersichtlichkeit wird zuerst nur der eigentliche Algorithmus vorgestellt. Im Anschluss werden zur genaueren Beschreibung zwei Prozeduren präsentiert, welche eine effiziente Implementierung gewährleisten.

Algorithmus I: `verify`

1. Markiere jedes Zeichen $T[r, c]$ im Text mit einem Koordinatenpaar $< i, j >$, welches angibt, mit welchem Zeichen im Suchmuster $P[i, j]$ es verglichen werden muss.

2. Vergleiche jedes Textelement an der markierten Position mit dem gesuchten Muster und markiere es bei Übereinstimmung mit "wahr", ansonsten mit "falsch".

3. Markiere jeden Kandidaten, welcher innerhalb seiner Überlappung mit dem Text ein "falsch" enthält mit einer "Defekt"-Marke.

4. Entferne jeden Kandidaten mit einer "Defekt"-Marke an seinem Ursprung aus der Liste der Kandidaten.

Nun enthält die Kandidatenliste alle Fundstellen des Musters im Text.

⊚

Anmerkung:

1. Textstellen, welche von keinem Kandidaten überlappt werden, erhalten in Schritt 1 keinerlei Markierung

2. Die Markierung in Schritt 1 ist nicht eindeutig, da ein Zeichen im Text zu verschiedenen Kandidaten (sogenannte "relevante Kandidaten") gehören kann. Durch die Konsistenz der Kandidaten untereinander ist es jedoch irrelevant auf welchen sich die Markierung bezieht, da es keinen Einfluss auf das zu vergleichende Zeichen in Schritt 2 hat.

3. Zur Markierung in Schritt 1 wird die Prozedur `abf_wave_mark` benutzt.

4. Schritt 1 und Schritt 2 können zusammengefasst werden, indem schon bei Markierung eines Textelements der Vergleich aus Schritt 2 erfolgt.

5. Die Markierung im 3. Schritt erfolgt mit der Prozedur `abf_wave_disc`.

6. Ein Kandidat gilt in Schritt 4 nur dann als zum Entfernen markiert, wenn sein Basiselement mit einer entsprechenden Marke versehen ist.

7. Der 3. und 4. Schritt kann wiederum zusammengefasst werden, indem schon in `abf_wave_disc` die entsprechenden Kandidaten entfernt werden.

Zur Markierung der Zeichen im Text mit den entsprechenden Koordinaten eines Zeichens im Suchmuster wird ein Prinzip benutzt, welches die Autoren "The Wave" nennen[5]. Der Name kommt daher, dass dieses Prinzip seinen Ursprung im Sport hat, wenn die Zuschauer die "Welle" bilden, in dem die Zuschauerreihe, welche links neben einer stehenden Reihe sitzt, sich hinstellt und die stehende Reihe sich daraufhin wieder setzt. Diese Welle setzt sich dann durch das ganze Stadion fort.
Im folgenden Algorithmus wird nun das Muster spaltenweise (zeilenweise) von links nach rechts und oben nach unten durchlaufen. Der Wert eines jeden Zeichens hängt direkt vom Zeichen ab, welches eine Spalte (Zeile) vorher in derselben Zeile (Spalte) bearbeitet wurde.
Hierdurch werden die Markierungen über den Text verbreitet (Abbildung 15). Dabei wird dafür gesorgt, dass die Ausbreitung der "Welle" außerhalb der Kandidaten, oder bei Überschreitung der Grenze zu einem neuen Kandidaten, verhindert wird.

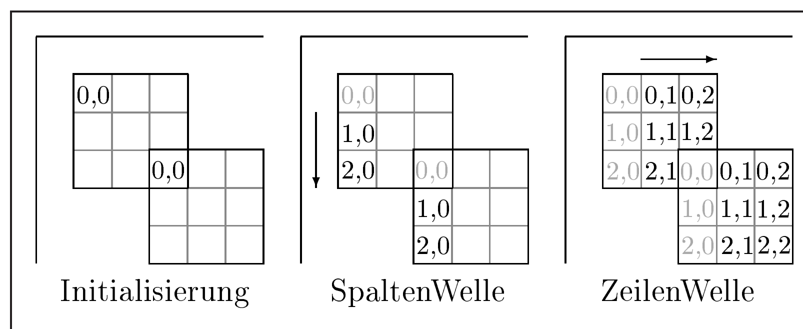


Abbildung 15: `wave_mark`

Algorithmus J: `wave_mark`

1. Initialisierung: Markiere für jeden Kandidaten dessen Ursprung im Text mit $< 0, 0 >$.

2. SpaltenWelle:
   Für alle Spalten c:
      Für alle Zeilen $r := 0...n-1$:
         Besitzt $T[r, c]$ noch keine Marke und ist $T[r-1, c]$ definiert
         und hat die Marke $< i, j >$ und gilt $i < m - 1$,
         dann markiere $T[r, c]$ mit $< i + 1, j >$.

3. ZeilenWelle:
   Für alle Zeilen r:
      Für alle Spalten $c := 0...n-1$:
         Besitzt $T[r, c]$ noch keine Marke und ist $T[r, c-1]$ definiert
         und hat die Marke $< i, j >$ und gilt $j < m - 1$,
         dann markiere $T[r, c]$ mit $< i, j + 1 >$.

⊚

Anmerkung:
Die Funktionsweise ist mit Hilfe von Abbildung 15 auf der vorherigen Seite einsichtig. Im ersten Schritt werden alle Marken vertikal nach unten verteilt. Im zweiten Schritt findet dann die horizontale Verteilung nach rechts statt.
Dabei wird ausgehend vom Ursprung des Kandidaten an der entsprechenden Stelle im Text jeweils die relative Position der Zeichen zum Text erhöht, bis das "Ende" des Kandidaten erreicht ist, also bis $j >= m$ bzw. $i >= m$.

Um Speicher zu sparen kann der Algorithmus auch dahingehend geändert werden, dass man nur jeweils eine Spalte im Speicher behält und den Text spaltenweise von links nach rechts und innerhalb der Spalte von oben nach unten durchläuft. Unterhalb der aktuellen Position enthält die Spalte dabei die Einträge der vorhergehenden Spalte. Ist die aktuelle Position in den C_i enthalten, so wird sie mit $(0, 0)$ markiert, ansonsten wird eine gültige Markierung oberhalb oder links von der Position gesucht und deren Markierung entsprechend der Welle erweitert. Sollte dabei sowohl die Position oberhalb als auch die Position links der aktuellen Position gültig sein, so wählt man diejenige mit den kleineren Markierungen. Ist auf diese Weise ein Index für die aktuelle Position gefunden, so kann der Vergleich und somit die Markierung mit "wahr" oder "falsch" erfolgen.

Die Markierung der zu entfernenden Kandidaten geschieht mit einer zweiten Welle, dieses Mal von unten nach oben und rechts nach links (Abbildung 16). Damit sich keine Fehlerstelle in einem gültigen Kandidaten ausbreitet, wird ein Zähler mitgeführt. Dieser stellt sicher, dass sich jeder Fehler nur um m Stellen, also maximal über die Breite einer Fundstelle hinweg, ausbreitet. Damit kann er nur dann den Ursprung eines Kandidaten erreichen, wenn er innerhalb der Grenzen dieses Kandidaten lag und nur in diesem Fall wird der Kandidat für ungültig erklärt.
Die Reihenfolge der Spalten in der Spaltenwelle, bzw. der Zeilen in der Zeilenwelle ist für die Funktion des Algorithmus dabei irrelevant.

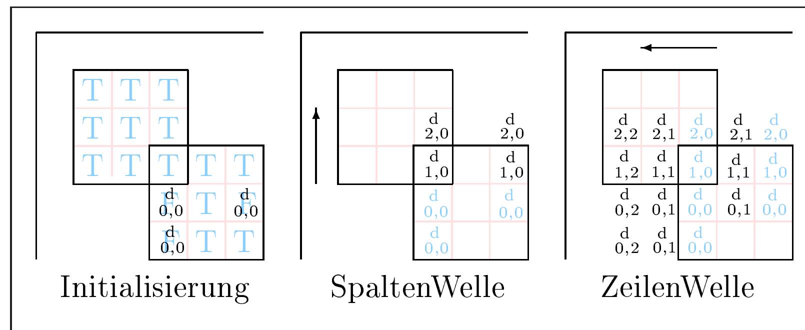


Abbildung 16: `wave_disc`

Algorithmus K: `wave_disc`

1. Initialisierung: Markiere jede Stelle mit einer "Fehler"-Markierung mit $d < 0, 0 >$ als Defekt.

2. SpaltenWelle:
   Für alle Spalten c:
   Für alle Zeilen $r := n - 1...0$:
   Besitzt $T[r, c]$ noch keine Marke und ist $T[r + 1, c]$ definiert und hat die Marke $d < i, j >$ und gilt $i < m - 1$, dann markiere $T[r, c]$ mit $d < i + 1, j >$.

3. ZeilenWelle:
   Für alle Zeilen r:
   Für alle Spalten $c := n - 1...0$:
   Besitzt $T[r, c]$ noch keine Marke und ist $T[r, c + 1]$ definiert und hat die Marke $d < i, j >$ und gilt $j < m - 1$, dann markiere $T[r, c]$ mit $d < i, j + 1 >$.

⊚

Anmerkung:

1. Die Initialisierung geschieht zweckmäßiger Weise direkt bei der Markierung der Fehlerstellen in der Prozedur `verify`, bzw. unter Berücksichtung von Anmerkung 4 zu diesem Algorithmus in `wave_mark`.
2. Die Reihenfolge der Spalten in der Spaltenwelle, bzw. der Zeilen in der Zeilenwelle ist für die Funktion des Algorithmus dabei wieder irrelevant.

Theorem 7:
Der Algorithmus `verify` ist korrekt und benötigt eine Zeit von $O(n^2)$.

Beweis:
Der Richtigkeit des grundsätzlichen Algorithmus ist klar. Der einzige anzuzweifelnde Aspekt ist, ob die "Welle" in den beiden Unterprozeduren jeweils die richtigen Kandidaten markiert.
Sei (r, c) ein Kandidat, welcher die Textposition $T[r+i, c+j]$ enthält. Dann kann j als *Spaltenabstand* und i als *Zeilenabstand* zwischen dem Ursprung von (r, c) im Text und $T[r+i, c+j]$ angesehen werden.
Kandidaten mit einem minimalen Zeilenabstand (Spaltenabstand) werden als *zeilennahe* (*spaltennahe*) Kandidaten bezeichnet. Ist ein Kandidat zugleich spaltennah und zeilennah zu $T[r, c]$ und sind diese Abstände im Betrag kleiner als m (der Kandidat enthält $T[r, c]$), dann handelt es sich um einen *nahen* Kandidaten.
Als *Distanz* zwischen $T[r, c]$ und einem Kandidaten wird das zugehörige Wertepaar aus Zeilenabstand und Spaltenabstand bezeichnet.

Behauptung: Das Zeichen im Suchmuster, mit welchem $T[r, c]$ im Text von `wave_mark` markiert wird, ist $< i, j >$, wobei $(r - i, c - j)$ der nahe Kandidat von $T[r, c]$ ist.

Beweis: Per Induktion über den Spaltenabstand des nahen Kandidaten.
Für einen Spaltenabstand von 0 (der Kandidatenursprung des nahen Kandidaten liegt in derselben Spalte wie das zu markierende Zeichen im Text $T[r, c]$) sichert die Spaltenwelle, dass $< i, j >$ die Distanz zwischen $T[r, c]$ und seinem nahen Kandidaten darstellt.
Angenommen, dass für jedes Zeichen im Text mit einem Spaltenabstand von d zu seinem nahen Kandidaten, das Paar $< i, j >$ die Distanz zu seinem nahen Kandidaten angibt, so ist leicht einzusehen, dass die Zeilenwelle die korrekte Markierung aller Zeichen im Text mit einem Spaltenabstand von $d + 1$ zum nahen Kandidaten sicherstellt. □

Der Beweis der Richtigkeit von `wave_disc` verläuft analog.

Laufzeitverhalten:
Die Prozeduren `wave_mark` und `wave_disc` benötigen eine Zeit von je $O(n^2)$. Sie werden in `verify` jeweils nur einmal aufgerufen. Da auch jeder weitere Schritt in `verify` leicht in einer Zeit $O(n^2)$ zu implementieren ist, folgt dies als Gesamtlaufzeit. □

2.6 Résumé

Sowohl die Suche nach konsistenten Kandidaten, als auch die endgültige Verifizierung der gefundenen Positionen benötigt eine Zeit von $O(n^2)$ und ist damit insbesondere wie gewünscht unabhängig vom verwendeten Alphabet und außerdem linear über die Anzahl der Zeichen im Text $N := n * n$.
Durch die Konstruktion der Witness-Tabellen in einer Zeit von $O(m^2 log\sigma)$ kann jedoch leider keine wie von den Autoren versprochene alphabetunabhängige Gesamtlaufzeit von $O(n^2)$ erzielt werden.
Mit anderen (komplizierteren) Verfahren zur Witness-Bestimmung kann dieser Makel ausgeräumt werden.

Ein Verfahren haben die Autoren selbst unter dem bezeichnenden Titel "The Truth, the Whole Truth, and Nothing but the Truth"[6] veröffentlicht. Sie benutzen hierbei die Funktion

$$P'[i,j] = \begin{cases} 1 & \text{, wenn } P[i,j] = P[0,0] \\ 0 & \text{, wenn } P[i,j] \neq P[0,0] \end{cases}$$

um aus dem gesuchten Muster P ein Muster P' zu erzeugen, welches über einem binären Alphabet definiert ist. Durch $log\ 2 = 1$ ist es möglich für P' in einer Zeit von $O(m^2)$ eine Witness-Tabelle zu erzeugen.

Je nach Art der Periode von P' kann nun dessen Witness-Tabelle auf die Witness-Tabelle von P überführt werden.
Diese Überführung benötigt im schlimmsten Fall eine Laufzeit von $O(m^2)$ unabhängig vom gewählten Alphabet. Die benutzten Algorithmen sind jedoch leider sehr komplex und werden zum Teil von den Autoren nur im Ansatz oder als Idee wiedergegeben. Sie können somit als theoretischer Beweis für die Existenz eines alphabetunabhängigen Algorithmus angesehen werden, in der Praxis finden diese Ideen jedoch vermutlich keine direkte Verwendung.

Eine weitere Methode wird von Crochemore und Rytter vorgeschlagen[12]. Sie benutzen einen veränderten Algorithmus von Galil und Park[15]. Hier wird mit einer nicht radiant-periodischen Untermatrix von P begonnen und diese rekursiv an ihren Rändern erweitert. Bei dieser Erweiterung kann dann je nach Art der momentan für die Untermatrix vorliegenden Periodizität die Witness-Tabelle generiert werden. Insgesamt ist auf diese Weise ein auch praktisch anwendbarer Algorithmus möglich, welcher eine Zeit von $O(m^2)$ zum Aufbau der Witness-Tabelle von P benötigt.

In den abschließenden Untersuchungen wird sich jedoch zeigen, dass die Vorverarbeitungsphase auch mit einfachen Algorithmen nur einen sehr kleinen Teil der Gesamtlaufzeit ausmacht und der Algorithmus selbst ohne Vorverarbeitung nur in seltenen Fällen eine akzeptable Laufzeit liefert.

3 Idee von Crochemore, Gąsienic, Plandowski und Rytter

Crochemore, Gąsienic, Plandowski und Rytter beziehen sich auf die von Amir, Benson und Farach geschaffenen Grundlagen, haben in ihrer Arbeit [10] aber einen vollkommen anderen Ansatz gewählt. Sie zeigen die Idee für einen Algorithmus auf, welcher aufgrund seiner Komplexität eher theoretischer Natur ist und präsentieren dementsprechend auch nur einen unvollständigen algorithmischen Unterbau.
Da die Ideen und Ansätze jedoch von vielen anderen Autoren aufgegriffen wurden und als Grundlage weiterer Algorithmen dienen, sollen sie hier näher beschrieben werden. Ein aus diesen Ideen entstandener Algorithmus von Kärkkäinen und Ukkonen [18] wird im nachfolgenden Kapitel vorgestellt.

3.1 Vorüberlegungen und Begriffe

Wie schon im vorangegangenen Kapitel werden als Perioden solche Verschiebungsvektoren bezeichnet, die zwei identische Muster bei Verschiebung um diesen Vektor in den überlappenden Bereichen in Deckung bringen.

Definition 1:

Sei $\alpha = (\alpha_1, \alpha_2)$ ein zweidimensionaler Vektor. Dann ist die Größe von α definiert durch $|\alpha| := max(|\alpha_1|, |\alpha_2|)$.
Sei $N := n' \times n''$ ein rechteckiger Bereich. Ist $|\alpha_1| \leq c{*}|n'|$ und $|\alpha_2| \leq c{*}|n''|$, dann ist α ein kurzer Vektor bezüglich N.

Jede Komponente eines Vektors ist also betragsmäßig höchstens c mal so groß wie die korrespondierende Seite des Rechtecks. Im Folgenden wird c weiterhin als Konstante mit einem Wert von $\frac{1}{8}$ benutzt.

Definition 2:

Besitzt ein Muster eine kurze Periode, so ist das Muster periodisch.

Die Autoren wollen unnötige Vergleiche vermeiden um die Laufzeiten des Algorithmus zu minimieren. Hierzu vergleichen sie zuerst nur einige bestimmte Punkte des Suchmusters mit den entsprechenden Punkten im Text. Erst wenn diese *Probe* (engl. *Sample*) erfolgreich war wird das ganze Suchmuster verglichen.
Je mehr Punkte die Probe enthält, umso größer ist die Chance aufgrund des ersten Vergleichs eine mögliche Fundstelle auszuschließen. Gleichzeitig steigt dadurch jedoch auch die Anzahl der benötigten Vergleiche.
Das Problem ist also eine Probe zu finden, welche möglichst charakteristisch für das gesuchte Muster ist und zeitgleich möglichst wenig Punkte enthält.

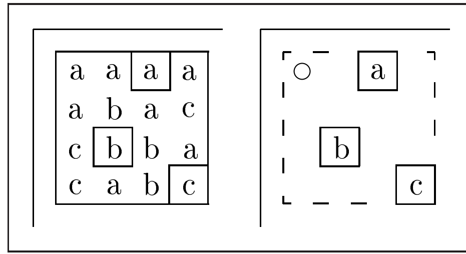


Abbildung 17: Suchmuster und Probe

Definition 3:

Eine Probe S zu einem Suchmuster P ist eine Menge von Positionen mit zugehörigen Zeichen aus P.
Die Position einer Probe ("$\circ$" in Abbildung 17) innerhalb eines Texts ist die Position der linken oberen Ecke des zur Probe gehörenden Suchmusters im Text.

Definition 4:

Stimmen alle Elemente einer Probe S an der Position x mit dem Text überein, so stimmt das Suchmuster P partiell mit dem Text T überein.
Schreibweise: $PartiellerFund(x, S)$.

Nun kommt eine weitere Idee der Autoren zum Tragen. Stimmt an einer Position x die Probe S mit dem Text T überein, so läßt sich ein rechteckiger Bereich $\mathcal{F}$ im Text bestimmen, in dem sich (gegebenenfalls mit Ausnahme von x) keine Position einer Fundstelle befindet. Dieser Bereich wird als *Field of Fire* von S (kurz: $FoF(S)$) bezeichnet. Dabei muss x nicht notwendigerweise in $\mathcal{F}$ liegen. Die relative Position von x bezüglich $\mathcal{F}$ wird als *Handle* von S bezeichnet.

Definition 5:

Mit $bedecktVon(T, \mathcal{F}, x)$ werden mit Ausnahme von x diejenigen Positionen im Text T bezeichnet, welche von $\mathcal{F}$ überdeckt werden.

Zusammengefasst gilt also:
Gilt $PartiellerFund(x, S)$, so gibt es keine Fundstelle des
Musters P an den Positionen $bedecktVon(T, \mathcal{F}, x)$ im Text.

Das Problem besteht an dieser Stelle darin ein entsprechendes FoF zu jeder Probe S zu finden. Hat man dieses Problem gelöst, so ist es möglich mit relativ wenigen Vergleichen ($|S|$ viele) eine (zumeist große) Anzahl von Kandidaten

($|\mathcal{F}|$ viele) auszuschließen. Da das FoF nur von S abhängt, kann dieser Ausschluss in konstanter Zeit erfolgen.

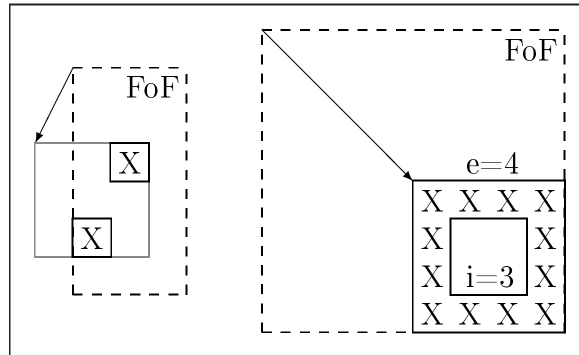


Abbildung 18: Kleine Probe und Rahmenprobe

Die Probe kann eine beliebige Untermenge des Suchmusters sein, im Folgenden werden jedoch nur zwei Arten von Proben betrachtet (siehe auch Abbildung 18):

Definition 6:

Eine kleine Probe besteht aus zwei Positionen des Suchmusters mit den dazugehörigen Symbolen. Das zugehörige FoF $\mathcal{F}$ ist ein rechteckiger Bereich der Größe $c*m' \times c*m''$. Das Handle von S muss sich dabei nicht zwangsläufig innerhalb von $\mathcal{F}$ befinden.

Definition 7:

Ein Rahmen S von P besteht aus Positionen eines $e \times e$ großen quadratischen Bereichs des Suchmusters P und den dazugehörigen Symbolen, mit Ausnahme eines im Zentrum von S befindlichen Bereichs (dem Loch) der Größe $(i-1) \times (i-1)$. Er wird beschrieben durch ein Wertepaar (e, i) und mit $\theta(e, i)$ bezeichnet.
Die Dicke des Rahmens ist definiert als $\phi(\theta) := e - i + 1$.
Eine Rahmenprobe S von P ist ein Rahmen von P mit einem FoF $\mathcal{F}$.
$\mathcal{F}$ ist ein quadratischer Bereich der Größe $c * \phi(\theta) \times c * \phi(\theta)$. Das Handle von S liegt im Zentrum von $\mathcal{F}$.

Es ist offensichtlich, dass die Effizienz eines entsprechenden Algorithmus mit wachsender Anzahl von Zeichen in der Probe sinkt und mit wachsender Größe des FoF steigt. So gesehen könnte man meinen, die kleine Probe wäre in den meisten Fällen von Vorteil. Durch die Position des Handles kann jedoch oft ein Rahmensample bessere Ergebnisse liefern.

3.2 Suche nach nichtperiodischen Mustern

Gąsienic, Plandowski und Rytter haben für den eindimensionalen Fall einen sehr einfachen Algorithmus implementiert [14].
Sei p ein eindimensionales, nichtperiodisches Suchmuster, $pre[p]$ das Präfix von p mit Länge $\frac{2}{3}|p|$ und $suf[p]$ das Suffix von p mit Länge $\frac{2}{3}|p|$. Dann sind entweder $pre[p]$, $suf[p]$ oder beide nichtperiodisch.

Sei $next[p] := \begin{cases} pre[p] & \text{, falls } pre[p] \text{ nichtperiodisch} \\ suf[p] & \text{, sonst} \end{cases}$

und $zoomseq[p] = (p, next[p], next[next[p]], ...)$. Dann ist $zoomseq$ eine so genannte *Zoom-Sequenz*, eine Folge von ähnlichen regelmäßigen Objekten mit abnehmender Größe. Jedes Element lässt sich dabei aus seinem Vorgänger in konstanter Zeit und mit konstantem Speicherverbrauch berechnen. Da jedes Element entweder Präfix oder Suffix seines Vorgängers ist, genügt ein Bit zur Speicherung eines jeden Elementes von $zoomseq$, insgesamt ist der Speicheraufwand dieser Liste somit logarithmisch zur Listenlänge.
Während der Suche vergleicht man nun zuerst das letzte und damit kleinste Element der Liste mit dem Text. Bei Übereinstimmung vergleicht man mit dem nächst größeren Element und fährt bis zur Auffindung von p fort. Durch die sukzessive Verwendung von Präfixen und Suffixen kann man bei Nichtübereinstimmung das Suchmuster um die Länge des zuletzt übereinstimmenden Elements über dem Text verschieben.

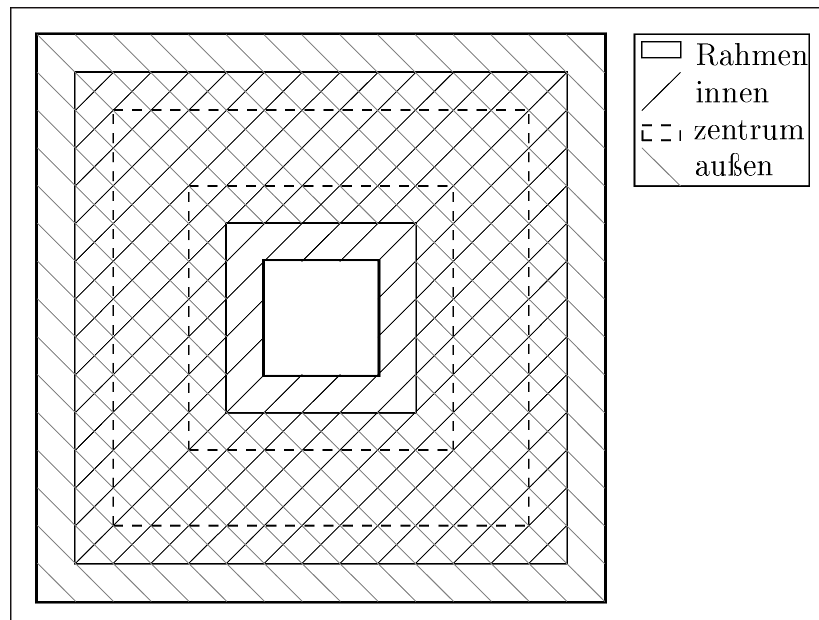


Abbildung 19: *innen*, *aussen* und *zentrum*

Um dieses Verfahren auf zweidimensionale Muster zu übertragen muss zuerst eine Datenstruktur gefunden werden, welche für P den Präfixen und Suffixen von p im eindimensionalen Fall entspricht. Hierfür lassen sich Rahmen von P verwenden. Präfixe und Suffixe sind hierbei der innere und der äußere Bereich des Rahmens, das ganze P entspricht einem Rahmen mit leerem Loch, also mit $i = 1$ (siehe Abbildung 19 auf der vorherigen Seite).

Definition 8:

Sei $\theta := \theta(e, i)$ ein Rahmen mit der Dicke $t := \phi(\theta) = e - i + 1$. Dann seien die folgenden Unterrahmen definiert:
$innen(\theta) := \theta(e - 2c * t, i)$
$zentrum(\theta) := \theta(\frac{e+i}{2} + c * t, \frac{e+i}{2} - c * t)$
$aussen(\theta) := \theta(e, i + 2c * t)$

Beispiel 1:

Sei $\theta(e, i) = \theta(30, 7)$. Dann ist die Dicke $t = \phi(\theta) = e - i + 1 = 30 - 7 + 1 = 24$. Die Probe ist also ein $e \times e = 30 \times 30$ Einheiten großes Quadrat, in dem ein $i - 1 \times i - 1 = 6 \times 6$ großer Innenteil fehlt.
$innen(\theta) = \theta(e - 2ct, i) = \theta(30 - 2\frac{1}{8}24, 7) = \theta(24, 7)$. Dies ist ein 24×24 Einheiten großes Quadrat, in dem ein 6×6 großer Innenteil fehlt.
$zentrum(\theta) = \theta(\frac{e+i}{2} + ct, \frac{e+i}{2} - ct) = \theta(21.5, 15.5)$. Dies ist ein $21,5 \times 21,5$ Einheiten großes Quadrat, in dem ein $14,5 \times 14,5$ großer Innenteil fehlt.
$aussen(\theta) = \theta(e, i + 2ct) = \theta(30, 13)$. Dies ist ein 30×30 Einheiten großes Quadrat, in dem ein 12×12 großer Innenteil fehlt.

Es ist deutlich die Lage wie in Abbildung 19 auf der vorherigen Seite zu erkennen. *innen* und *aussen* überlappen sich in einem grossen Bereich, in welchem mit Abstand zu den Grenzen von *aussen* und *innen* der *zentrum*-Rahmen liegt. ⌟

Es fehlt die Definition der Periodizität für solche Rahmen, sie ist der bekannten Definition für Matrizen jedoch sehr ähnlich.

Definition 9:

Zwei Punkte x, y innerhalb eines Rahmens θ sind gradlinig verbunden in θ, wenn die Verbindungsgerade von x und y ganz in θ liegt.

In P sind also alle Punkte paarweise gradlinig verbunden, da P einem Rahmen mit $i = 1$ (also ohne "Loch") entspricht.

Definition 10:

Der Vektor α ist eine Periode des Rahmens θ, wenn für alle gradlinig in θ verbundenen Punkten $x \in P$, $x + \alpha$ auch $P[x] = P[x + \alpha]$ für jede Position x aus dem zum Rahmen gehörendem Muster P gilt.
Der Rahmen θ ist periodisch, wenn er eine Periode mit einer Länge kleiner als $c * \phi(\theta)$ besitzt.

Ist der Rahmen θ nichtperiodisch, dann ist er eine Rahmenprobe für P mit einen FoF $\mathcal{F}$ der Größe $c\phi(\theta) \times c\phi(\theta)$ und einem Handle im Zentrum von $\mathcal{F}$.

Um den eindimensionalen Algorithmus zu übertragen muss nur noch gezeigt werden, dass für jedes nichtperiodische P entweder $innen(P)$, $aussen(P)$ oder beide wieder nichtperiodisch sind. Das folgende Lemma leistet dies:

Lemma 1:

(a) Sei $innen(\theta)$ periodisch und $x,y \in \theta$ zwei in $innen(\theta)$ gradlinig verbundene Punkte. Sei außerdem $x - y$ bezüglich θ ein kurzer Vektor. Dann gibt es zwei gradlinig verbundene Punkte $x', y' \in zentrum(\theta)$, so dass $P[x] = P[x']$, $P[y] = P[y']$ und $x - y = x' - y'$ gilt.

(b) Ist θ ein nichtperiodischer Rahmen, dann ist mindestens einer der beiden Rahmen $innen(\theta)$ und $aussen(\theta)$ nichtperiodisch.

Beweis:

Zu (a): Es sei π eine kurze Periode von $\theta' := innen(\theta)$. Sei außerdem L eine Gerade durch x und y. Sei π nicht parallel zu L. Transponiert man nun x und y über ein Vielfaches von π möglichst nah an den äußeren Rand von θ', dann werden x, y auf die gesuchten x', y' geschoben.
Ist π parallel zu L, so verfährt man analog.
Zu (b): Ist $innen(\theta)$ nichtperiodisch, dann ist die Aussage erfüllt. Wenn nicht, dann sei angenommen, dass $aussen(\theta)$ ebenfalls periodisch ist.
Folglich hat $\theta' := innen(\theta)$ einen kurzen Periodenvektor π und $\theta'' := aussen(\theta)$ besitzt einen kurzen Periodenvektor α. Da θ nichtperiodisch ist, existiert ein gradlinig verbundenes Paar von Positionen x, y in θ mit $x - y = \alpha$ und $P[x] \neq P[y]$. Da x und y auch in θ'' liegen könnten, seien diese oBdA so gewählt, dass x und y in θ'' nicht gradlinig verbunden sind. Da sich θ' θ'' in einem sehr großen Bereich überlappen, impliziert dies, dass sich x und y in θ' nah am inneren Rand von θ'' befinden. Mit (a) können x und y auf zwei Positionen x' und y' in $zentrum(\theta)$ transponiert werden, so dass gilt $P[x] = P[x']$ und $P[y] = P[y']$. Durch die Konstruktion der drei Unterrahmen gilt immer $zentrum(\theta) \subseteq \theta''$. Da $P[x'] = P[x] \neq P[y] = P[y']$ brechen x' und y' in θ'' die Periodizität α, wenn sie gradlinig verbunden sind. In

zentrum(θ) sind diese Punkte jedoch bezüglich θ'' immer gradlinig verbunden, da hier alle Punkte weit genug vom Loch von θ'' entfernt sind. Es gilt also $x', x' + \alpha \in \theta''$ und x' und $x' + \alpha$ sind gradlinig in θ'' verbunden, es gilt jedoch auch $P[x'] \neq P[x' + \alpha]$. Dies widerspricht der Annahme, dass α eine Periode von θ'' ist.
Damit können *innen*(θ) und *aussen*(θ) nicht beide periodisch sein. □

Mit Hilfe von Lemma 1 auf der vorherigen Seite (b) kann nun wie im eindimensionalen Fall eine Funktion *Zoomseq* definiert werden:

Definition 11:
$Zoomseq(P) = (\theta_1, \theta_2, ..., \theta_k)$ mit $\theta_1 = P$ und

$$\theta_{i+1} = \begin{cases} innen(\theta_i) & \text{, falls } innen(\theta_i) \text{ ist nichtperiodisch} \\ aussen(\theta_i) & \text{, falls } aussen(\theta_i) \text{ ist nichtperiodisch} \end{cases}$$

für $1 \leq i < k$.
Die Dicke der einzelnen Rahmen $\phi(\theta_i)$ nimmt somit wie im eindimensionalen Fall die Länge der Zeichenketten stetig ab.
Das k ist so zu wählen, dass $\phi(\theta_{i+1}) < \phi(\theta_i)$ für $1 \leq i < k$.

Da P nicht periodisch ist, existiert nach Lemma 1 auf der vorherigen Seite (b) *Zoomseq*(P). Im Folgenden kann deshalb oBdA davon ausgegangen werden, dass diese Funktion schon bestimmt wurde.
Außerdem führt diese Eigenschaft zu einer weiteren wichtigen Beobachtung: in einem Ausschnitt vom Text der Größe $\frac{c}{2}m \times \frac{c}{2}m$ kann es höchstens eine Fundstelle des Musters geben. Es ist also möglich den Text gleichmäßig in nicht überlappende Segmente dieser Größe aufzuteilen und diese einzeln auf jeweils eine Fundstelle zu untersuchen.
Weiterhin sei bereits eine kleine Probe S für P bekannt.
Dass man davon oBdA ausgehen kann, wird später gezeigt.

Der nun folgende Algorithmus durchläuft ein Segment $\mathcal{S}$ von rechts nach links und von unten nach oben. Die erste untersuchte Position befindet sich also unten rechts in $\mathcal{S}$, die letzte oben links. Im Folgenden wird mit $\mathcal{A}$ der "aktive" Bereich des Segments bezeichnet, der alle möglichen Fundstellen enthält, im "inaktiven" Bereich befinden sich Positionen, welche bereits als Fundstellen ausgeschlossen werden konnten. Zu Beginn des Algorithmus ist $\mathcal{A} = \mathcal{S}$, der inaktive Bereich ist leer. Mit $next(x, \mathcal{A})$ wird die in der erläuterten Reihenfolge nächste auf x folgende aktive Position bezeichnet. Ist x bereits das "letzte" Element, oder ist $\mathcal{A}$ leer, so gilt $next(x, \mathcal{A}) = NULL$.

Für jede Position x wird zuerst überprüft ob ein partieller Fund von S vorliegt. Ist dies der Fall, so wird das vom Durchmesser größte Element θ_i von *Zoomseq*(P) bestimmt, welches ebenfalls partiell an dieser Position gefunden

wird. Ist dies P selbst, so ist x eine gesuchte Fundstelle, eine weitere kann es in diesem Segment nicht geben und der Algorithmus kann abgebrochen werden. Ansonsten können entsprechend dem FoF von θ_i einige Positionen aus $\mathcal{A}$ ausgeschlossen werden, zusätzlich werden die Positionen vor x entfernt. Da an dieser Position ebenfalls kein partieller Fund von S vorlag können dies auch keine Fundstellen von P sein.
Um die ausgeschlossenen Bereiche effizient speichern zu können, werden die folgenden Notationen eingeführt:

Definition 12:
Der Abstand dist zweier Punkte (x_1, x_2) und (y_1, y_2) ist definiert als $dist((x_1, x_2), (y_1, y_2)) := max\{|x_1 - y_1|, |x_2 - y_2|\}$.

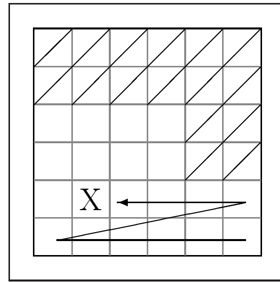


Abbildung 20: $\Gamma(x, 2)$

Definition 13:
In der Menge $\Gamma(x, r)$ liegen alle Positionen $z \in \mathcal{S}$ oberhalb und rechts der Position x, für die gilt $dist(x, z) > r \geq 0$ (siehe Abbildung 20).

Definition 14:
Der Radius eines FoF der Größe $f \times f$ ist $max\{\left\lfloor \frac{f}{2} \right\rfloor, 1\}$.

Der folgende Algorithmus implementiert die Suche nach einer Fundstelle von P innerhalb eines Segments $\mathcal{S}$ von T, unter der Annahme, dass P nichtperiodisch ist und $Zoomseq(P)$ bzw. die kleine Probe S zu P schon bestimmt wurde.

Algorithmus A: `SamplingZooming`

$\mathcal{A} = \mathcal{S}$
Sei x die untere rechte Ecke von $\mathcal{S}$.
Solange x nicht NULL
Gilt $partiellerFund(x, S)$, dann
Sei $j := min\{i \geq 1 : partiellerFund(x, \theta_i)\}$
Ist $j = 1$, dann ist x eine Fundstelle,
der Algorithmus kann beendet werden.
Sonst setze $r := radius(FoF(\theta_j))$.
$\mathcal{A} := \Gamma(x, r)$
$x := next(x, \mathcal{A})$

⊚

Lemma 2:

Der Algorithmus `SamplingZooming` findet eine Position von P im Text in einer Zeit von $O(N)$ mit einem Speicheraufwand von $O(1)$.

Beweis:

Da die Größe von $\mathcal{S}$ konstant ist, hängen die Gesamtkosten für die Bestimmungen von `partiellerFund(S)` linear von der Größe des Segments $\mathcal{S}$ ab. Ins Gewicht fällt deshalb im Wesentlichen die Berechnung von j. Diese kann durch die sukzessive Bestimmung von `partiellerFund` an der Position x mit den Rahmen $\theta_k, \theta_{k-1}, \ldots, \theta_j$ realisiert werden. Die Zoom-Sequenz der θ_i kann dabei durch eine logarithmische Anzahl von Bits beschrieben und somit in einem einzelnen Integer-Wert gespeichert werden. Hierdurch kann ein konstanter Speicheraufwand zur Bestimmung von j erzielt werden.
Die eigentliche Bestimmung von `partiellerFund` geschieht jeweils mit einem naiven Algorithmus, sie hängt damit zeitlich proportional von der Größe von θ_j ab. Die zeitlichen Kosten betragen also $O(\phi(\theta_j)m)$. $\phi(\theta_j)$ ist wiederum proportional zum Radius r des FoF von θ_j. Zusammenfassend kann also $j := min\{i \geq 1 : partiellerFund(x, \theta_i)\}$ mit einem konstanten Speicheraufwand und in einer Zeit $O(r * m)$ bestimmt werden.
Im Verlauf des Algorithmus werden verschiedene x $(x_1, x_2, \ldots, p)$ und entsprechend verschiedene Radien r $(r_1, r_2, \ldots, r_p)$ betrachtet. Somit benötigt der Algorithmus insgesamt eine Zeit von $\sum_{i=1}^{p} r_i m$. Dabei liegt jedes x_i oberhalb und rechts seines Vorgängers und es gilt $dist(x_i, x_{i+1}) \geq r_i$, die Summe der r_i ist also durch m beschränkt. Die $\sum$ kann deshalb durch m^2 abgeschätzt werden, die Laufzeit ist somit hinsichtlich der Größe des gesuchten Musters linear.
Es gibt $O(\frac{n^2}{m^2})$ Segmente $\mathcal{S}$ in T. Da jedes in einer Zeit von $O(|\mathcal{S}|)$ verarbeitet werden kann, ergibt sich eine Gesamtlaufzeit für den gesamten Text, welche linear zu dessen Größe ist. □

Die für den Algorithmus getroffene Annahme, dass $Zoomseq(P)$ bestimmt wurde ist aufgrund von Lemma 1 auf Seite 54 unproblematisch, die Annahme der Existenz einer kleinen Probe S jedoch nicht. Der Algorithmus kann aber leicht erweitert werden um auf diese Annahme verzichten zu können.

Definition 15:

Mit $\underline{truncate(P)}$ wird die $(1-c)m \times m$ große Untermatrix von P bezeichnet, welche aus den unteren $(1-c) * m$ Zeilen von P besteht.
Mit "breiten Proben" werden nichtperiodische Untermatrizen von P bezeichnet, welche durch die Entfernung der obersten Zeilen von P entstehen. Die Anzahl der zu entfernenden Zeilen ist nicht festgelegt. Die Bezeichnung "breite Probe" folgt aus der Tatsache, dass die Probe mehr Spalten als Zeilen enthält.

Ist $truncate(P)$ nichtperiodisch, so handelt es sich hierbei folglich ebenfalls um eine breite Probe.

Lemma 3:

Ist P nichtperiodisch und $truncate(P)$ periodisch, so besitzt P eine "kleine Probe" S.

Mit Hilfe von *truncate* kann aus P eine Zoom-Sequenz $\mathcal{Z}$ von nichtperiodischen Untermatrizen von P bestimmt werden:
$\mathcal{Z} = (P = P_0, P_1, \ldots, P_k)$ mit $P_{i+1} = truncate(P_i)$ für $1 \leq i < j$ und einem k der Art, dass $P_k = truncate(P_j)$ periodisch ist oder nur noch aus einer Zeile besteht.

Der nun folgende Algorithmus unterteilt den Text in Segmente mit einer Größe von $\frac{c}{2}m \times \frac{c}{2}m$. Da die breiten Proben P_i so breit sind, dass sie ein FoF mit mindestens $\frac{c}{2}m$ Spalten besitzen, ist sichergestellt, dass der aktive Bereich im Falle eines partiellen Fundes genau aus einer gewissen Anzahl von vollständigen Zeilen am Anfang des aktuellen Segments besteht. In diesem Fall können also genau h Zeilen bei der weiteren Suche übersprungen werden, mit $h :=$Anzahl der Zeilen von P_i.
Die einzelnen Kandidaten im jeweils aktiven Bereich werden in derselben Reihenfolge wie bei `SamplingZooming` durchlaufen. Der Algorithmus benutzt dabei die folgenden Funktionen:
`findNext`$(\mathtt{x}, \mathtt{P}', \mathcal{S})$ sucht ab (und inklusive) der Position x im aktiven Bereich nach der nächsten Untermatrix $P' \in \{P = P_0, P_1, \ldots, P_k\}$ innerhalb des Segments $\mathcal{S}$ und gibt deren Position zurück. Wird P' nicht gefunden, so gibt sie NULL zurück. `findNext` wird mit Hilfe des Algorithmus `SamplingZooming` bestimmt.

`shiftUp`$(\mathtt{x}, \mathtt{s}, \mathcal{S})$ gibt den ersten Punkt derjenigen Zeile in $\mathcal{S}$ zurück, welche s Zeilen über x liegt. Liegt diese Zeile außerhalb von $\mathcal{S}$, so wird NULL zurückgegeben.

Algorithmus B: `GenerelleNichtperiodischeSuche`

x :=erste Position in $\mathcal{S}$ (unten rechts).
solange x nicht $NULL$ ist
  $x := findNext(x, P_k, \mathcal{S})$
  $j := min\{i \geq 1 : partiellerFund(x, P_i)\}$
  Ist $j = 1$, dann ist x eine Fundstelle von P in T, der
    Algorithmus kann abgebrochen werden.
  h :=Anzahl der Zeilen von P_j
  $x := shiftUp(x, \frac{ch}{2}, \mathcal{S})$

⊚

Die in einem Durchlauf erbrachte Arbeit des Algorithmus ist proportional zur Anzahl der durch `shiftUp` übersprungenen Zeilen. Daher ist der Zeitaufwand für die Untersuchung eines Segments proportional zu dessen Größe. Zusammen mit der Untersuchung von `SamplingZooming` ist damit gezeigt, dass ein nichtperiodisches Muster in einem Text mit linearem Zeitaufwand und konstantem Speicher gefunden werden kann.

3.3 Suche nach periodischen Mustern

Ein nichtperiodisches Muster ist entweder geraden-, strahlen-, oder gitterperiodisch (siehe Seite 11). Je nach der Art der Periodizität kann man die Suche auf eine Suche nach nichtperiodischen Mustern zurückführen. Exemplarisch soll an dieser Stelle der schwierigste Fall, die Behandlung von geradenperiodischen Mustern erläutert werden (siehe Abbildung 5 auf Seite 13). Sei P ein

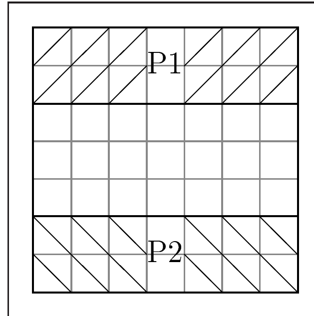


Abbildung 21: Zusammengesetztes Muster D

geradenperiodisches Suchmuster und π die kürzeste Periode von P. Die Richtung von π sei $\alpha(\pi) = \frac{x}{y}$ mit oBdA $|\alpha(\pi)| \geq 1$. Sei $P1$ der Bereich von P, welcher aus dessen oberen $3|\pi| =: h$ Zeilen besteht, $P2$ der Bereich von P, welcher aus den unteren h Zeilen von P besteht (Abbildung 21). Fasst man $P1$ und $P2$ als ein Muster D zusammen, so ist D nichtperiodisch. Dies bedeutet auch, dass jeder Nicht-Nullvektor $v := (x, y)$ mit $|x| < ch$ und $|y| < cm$, welcher keine Periode von P ist, auch keine Periode von $P1$ und $P2$ sein kann.
Man kann D auch als Probe für P in S ansehen. Demnach besitzt D ein FoF $\mathcal{F}$ der Größe $ch \times cm$ mit einem handle im Zentrum von $\mathcal{F}$. Die Suche nach einem solchen zusammengesetzten nichtperiodischem Muster unterscheidet sich im wesentlichen nicht von der bekannten herkömmlichen Suche nach nichtperiodischen Mustern.

Man könnte sich also für die geradenperiodische Suche einen Algorithmus in der folgenden Art vorstellen:
Als erstes wird der Text T gleichmäßig in Segmente mit einer Größe von $c * \frac{m}{2} \times c * \frac{m}{2}$ Zeichen unterteilt. Jedes dieser Segmente teilt man dann wiederum in Untersegmente mit einer Größe von $c\frac{h}{2} \times c\frac{h}{2}$ auf. Da D ein FoF der Größe $ch \times cm$ besitzt, kann in jedem Untersegment nur noch maximal eine Fundstelle von D (und somit auch maximal eine Fundstelle von P) liegen.
In der bekannten Reihenfolge von unten nach oben und von rechts nach links wird nun jedes Segment betrachtet und in jedem Untersegment nach D gesucht. Findet man D in einem Untersegment eines Segments an einer Position x und ist diese Position gleichzeitig eine Fundstelle von P im Segment, so prüft man nach, ob die Verbindungsgerade zwischen dieser und der vorangegangenen

Fundstelle in Richtung einer Periode von P verläuft. Ist dies nicht der Fall, so ist x keine gültige Fundstelle und die Suche kann im nächsten Untersegment fortgesetzt werden. Ansonsten versucht der Algorithmus einen möglichst dicken Außenrahmen an dieser Position zu bestimmen, er beginnt damit naiv zu prüfen, ob bei x der Außenrahmen von P mit einer Dicke h mit dem Text übereinstimmt. Wenn nicht kann mit der Suche im nächsten Untersegment fortgefahren werden.
Stimmt der Aussenrahmen bei einer Dicke von h mit dem Text überein, so kann seine maximale Dicke r bestimmt werden, wobei Vergleiche vermieden werden können, welche zur letzten gefundenen Position von P im Text gehören. Ist x keine Fundstelle von P, so können die nächsten $t := max(0, \frac{r-h}{\pi})$ Untersegmente bei der Suche nach D übersprungen werden, ansonsten wird x als letzte Fundstelle gespeichert.

Durch diese grobe Überlegung eines möglichen Algorithmus wird klar, dass prinzipiell eine lineare Laufzeit möglich ist, da jedes Symbol nur einmal verglichen wird und die Zeit, welche zur Suche von D extra benötigt wird, durch die Sprünge um t Untersegmente wieder eingespart werden kann.

3.4 Résumé

Die Ideen des Algorithmus sind sehr innovativ und führen zu einem theoretischen Algorithmus mit einer linearen Laufzeit bei geringem Speicheraufwand. In den Details ist er jedoch noch unvollständig und viele der guten Ansätze würden bei einer Implementierung zu Laufzeit- und Speicherverlusten aufgrund der Einschränkungen heute gängiger Programmiersprachen führen.
Am meisten fällt dabei die Bestimmung der kleinen Proben und der Rahmenproben ins Gewicht.
Zur Auffindung der kleinen Proben ist kein Algorithmus bekannt, nur die Existenz ist bewiesen. Um die Rahmenproben zu bestimmen fehlt eine effiziente Methode zur Bestimmung der Perioden einer solchen Probe, da sonst keine Zoom-Sequenz berechnet werden kann. Es ist denkbar hierzu die von Amir, Benson und Farach verwendeten Witness-Tabellen aus dem vorangegangenem Abschnitt in veränderter Form zu verwenden. Crochemore, Gąsienic, Plandowski und Rytter zeigen in Ihrer Arbeit einige Ansätze für eine mögliche approximative Lösung auf, welche jedoch wiederum einige sehr komplexe Datenstrukturen und schwer implementierbare Algorithmen voraussetzen.

Aus diesen Gründen wurde versucht die Ideen für einfachere Algorithmen zu verwenden. Diese können zwar nicht mehr dieselben Laufzeiten mit demselben geringen Speicheraufwand garantieren, wurden dafür aber schon realisiert und sind praktisch anwendbar. Einer dieser Algorithmen wird im Folgenden Kapitel vorgestellt.

4 Algorithmus von Kärkkäinen und Ukkonen

Dieser Algorithmus entstand aus den Ideen des vorangegangenen Kapitels und verfolgt genauso einen pessimistischen Ansatz. Er geht davon aus, dass es nur wenig Fundstellen von P in T gibt und versucht durch kleine Proben aus dem Text eine möglichst große Anzahl an Kandidaten auszuschließen. Dabei werden jedoch keine dynamischen Proben mit einem dynamischen FoF generiert und es wird vermieden, dass sich mehrere Proben gegenseitig beeinflussen. Unter der Annahme, dass dabei nur einige wenige Kandidaten übrig bleiben, genügt es diese durch einen naiven Algorithmus zu verifizieren.
Hierdurch wird der Algorithmus nicht nur wesentlich einfacher zu handhaben als der vorangegangene, er wird zudem auch sehr leicht parallelisierbar.

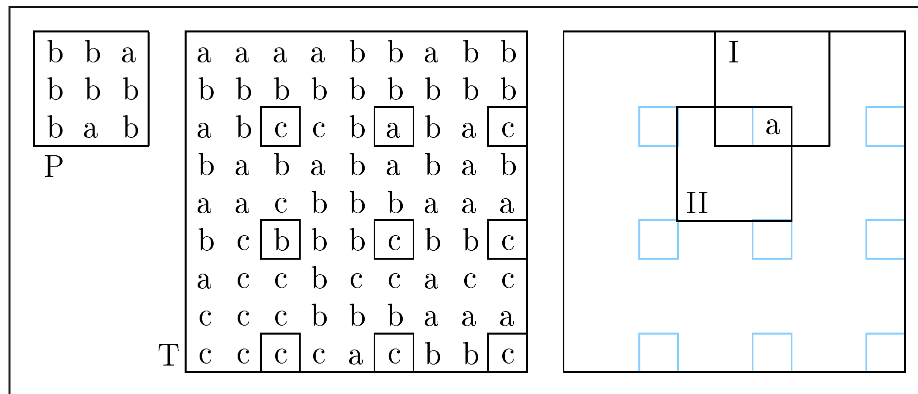


Abbildung 22: Muster, Text und Proben

Zur genaueren Erklärung der grundsätzlichen Idee seien ein Muster P und ein Text T wie in Abbildung 22 gegeben.
Gesucht sind alle Kandidaten (I, J), so dass $T[I + k - 1, J + h - 1] = P[k, h]$ für $0 \leq k, h \leq 2$.
Offensichtlich muss das Muster im Text eine der Zeilen i:=2,5,8 und eine der Spalten j:=2,5,8 überschneiden (wie üblich ab 0 gezählt). Alle Fundstellen müssen also so liegen, dass P einen der Kreuzungspunkte dieser Zeilen und Spalten überdeckt und an dieser Stelle mit dem Text identisch ist.
In einem ersten Schritt genügt es also diese Punkte als Probe zu benutzen und mit dem Muster abzugleichen um eine erste Aussage über mögliche Kandidaten treffen zu können.
Da $c \notin P$ kann es $T[2, 2] = c$ nicht überschneiden. Dies schließt gleichzeitig alle Kandidaten im Bereich $T[0 : 4, 0 : 4]$ aus. Dies ist damit eine sehr gute Probe. Über $T[2, 5] = a$ kann P ohne Konflikte bei $T[0, 4]$ (Position I) und $T[2, 3]$ (Position II) liegen, alle anderen Kandidaten im Bereich $T[0 : 4, 3 : 7]$ kommen nicht als Fundstellen in Betracht.

In einer späteren Phase des Algorithmus würde sich durch einen naiven Vergleich $T[0,4]$ als Fundstelle herausstellen, $T[2,3]$ jedoch nicht.
Das einzige Problem ist $T[5,2] = b$. Diese Position lässt insgesamt 7 Kandidaten als möglich Fundstelle zu und kann nur 2 eliminieren. Somit wäre dies eine sehr schlechte Probe. Nimmt man aber noch $T[5,1] = c$ hinzu, so könnten alle Kandidaten, welche diese beiden Positionen überschneiden, ausgeschlossen werden, da c in P nicht vorkommt. Hier wird deutlich, dass es gut ist, eine möglichst große Probe zu verwenden, da damit viele Kandidaten ausgeschlossen werden können, gleichzeitig erhöht dies jedoch auch die Anzahl der Vergleiche die bei der Suche benötigt werden.
Man muss somit die Probe so groß wählen, dass bei einer möglichst großen Anzahl von Kandidaten, welche ausgeschlossen werden können, die Anzahl der insgesamt benötigten Vergleiche minimiert wird.

Der Algorithmus wird nun gleichmäßig Testpunkte über den Text in der Art verteilen, dass jeder Kandidat mindestens q Positionen im Text überschneidet. Diese Positionen dienen als Proben und können beliebig angeordnet sein, die Anordnung ist jedoch für alle Proben identisch. Die genaue Verteilung der Proben hängt nur von der Geometrie von P und T ab, die Anordnung der Positionen innerhalb der Proben ist im Wesentlichen beliebig.

Im Folgenden wird nun der generelle Algorithmus vorgestellt. Im Anschluss daran wird er mit zwei verschiedenen Arten von Proben spezialisiert.

4.1 Der generelle Algorithmus

Die "Testpunkte" (Proben) bestehen aus q verschiedenen Positionen mit einer beliebigen, aber über den Algorithmus festen relativen Lage zueinander. Dabei ist es wesentlich, dass die Positionen in eine eindeutige Reihenfolge gebracht werden.

Definition 1:
Eine Vorlage (engl. Template) $Q := ((0,0),(i_1,j_1),(i_2,j_2),\ldots)$ ist eine geordnete Folge von q unabhängigen und unterschiedlichen Koordinatenpaaren. Der erste Eintrag ist immer $(0,0)$ und wird als Ursprung (engl. Source) von Q bezeichnet, die weiteren Positionen werden als relative Koordinate zum Ursprung angegeben.

Die Vorlage gibt somit nur die äußere Form aller Proben an. Platziert man sie an eine bestimmte Koordinate (i,j) des Textes oder des Suchmusters, so erhält man eine Probe.

Definition 2:

Eine Probe (engl. Sample) $Q(i,j)$ des Arrays A von der Form Q besteht aus den Positionen $((i,j),(i+i_1,j+j_1),(i+i_2,j+j_2),\ldots)$ relativ zu A.
Liegen diese Positionen alle innerhalb von A, so enthält A die Probe $Q(i,j)$.

Um eine Probe aus dem Text mit einer Probe aus dem Suchmuster vergleichen zu können, nutzt man aus, dass Q hinsichtlich seiner Einträge eine Ordnung aufweist. Legt man Q auf einen Array und hängt die Zeichen unterhalb von Q in der Reihenfolge der Einträge von Q hintereinander, so erhält man eine (eindimensionale) Zeichenfolge, welche sich ohne großen programmiertechnischen Aufwand vergleichen lässt.

Definition 3:

Sei A ein Array, welches aus den Einträgen (a_{ij}) besteht. Mit $s(A,Q(i,j)) := a_{ij}a_{i+i_1 j+j_1}a_{i+i_2 j+j_2}\cdots$ wird die Probenkette (engl. test string) definiert, welche zu einer Vorlage Q gehört, die über der Position (i,j) von A positioniert ist.

Die Idee des Algorithmus war eine gewisse Anzahl von Proben über dem Text zu verteilen und diese Proben mit einer gewissen Anzahl von Proben im Suchmuster zu vergleichen. Für eine Suche fasst man alle diese Proben zu einem Testschema zusammen.

Definition 4:

Ein Testschema (engl. test scheme) $\mathcal{Q}$ für eine Vorlage Q, ein Muster P und einem Text T ist ein Mengenpaar $(\mathcal{Q}_P,\mathcal{Q}_T)$ mit

$$\mathcal{Q}_P \subset \{Q(i,j)|\ P \text{ enthält } Q(i,j)\}$$

und

$$\mathcal{Q}_T \subset \{Q(i,j)|\ T \text{ enthält } Q(i,j)\}$$

$\mathcal{Q}_P$ enthält also eine gewisse Anzahl von Proben aus dem Suchmuster und $\mathcal{Q}_T$ eine gewisse Anzahl von Proben aus dem Text.
Legt man nun P an eine Position (i,j) im Text T, so kann man eine Probe aus P und eine Probe aus T in der Art entnehmen, dass die Proben "übereinander liegen", dass also für die Ursprünge der Proben (i_P,j_P) und (i_T,j_T) gilt:
$i_P = i_T - i$, bzw. $j_P = j_T - j$.
Vergleicht man nun die dazugehörenden Probenketten, so kann man entscheiden, ob hier keine Fundstelle vorliegt oder diese Position näher untersucht werden muss.

Definition 5:

Sei R ein Bereich der Größe $m \times m$ in T an der Stelle (i, j) an der eine Fundstelle von P liegen könnte.
Ein Zeuge für R ist ein Probenpaar $(Q(i_P, j_P), Q(i_T, j_T)) \in \mathcal{Q} \times \mathcal{Q}_T$ in der Art, dass $Q(i_P, j_P)$ in P an derselben Position liegt wie $Q(i_T, j_T)$ in R, also $i_P = i_T - i$, bzw. $j_P = j_T - j$.
Gilt $s(P, Q(i_P, j_P)) = s(T, Q(i_T, j_T))$,
so ist $(Q(i_P, j_P), Q(i_T, j_T))$ ein positiver Zeuge, ansonsten handelt es sich um einen negativen Zeugen.

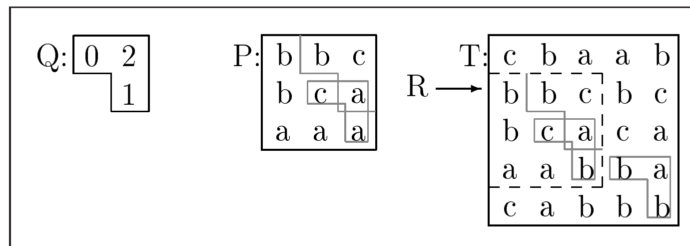


Abbildung 23: Probe, Muster und Text

Beispiel 1:

Sei $Q = ((0,0), (1,1), (0,1))$ und P und T wie in Abbildung 23 gegeben.
Sei $\mathcal{Q}_P = (Q(0,1), Q(1,1))$ und $\mathcal{Q}_T = (Q(1,1), Q(2,1), Q(3,3))$.
Sei R ein Kandidat von P an der Stelle $(1,0)$ im Text. $Q(0,1) \in \mathcal{Q}_P$ ist eine Probe im Muster, $Q(1,1) \in \mathcal{Q}_T$ eine Probe im Text.
Es gilt $s(P, Q(0,1)) = bac = s(T, Q(1,1))$.
$(Q(0,1), Q(1,1))$ ist also ein positiver Zeuge für R.
Betrachtet man stattdessen $Q(1,1) \in \mathcal{Q}_P$ und $Q(2,1) \in \mathcal{Q}_T$,
so gilt $s(P, Q(1,1)) = caa \neq cba = s(T, Q(2,1))$, $(Q(1,1), Q(2,1))$.
$(Q(1,1), Q(2,1))$ ist also ein negativer Zeuge für R. ⌟

Die Idee des Algorithmus ist es nun alle Kandidaten mit negativen Zeugen nicht weiter zu betrachten und nur Kandidaten mit positiven Zeugen näher zu untersuchen. Da es von diesen bei einer guten Wahl des Testschemas sehr viel weniger geben sollte genügt ein naiver Algorithmus um das Muster mit dem Text zu vergleichen ohne die Gesamtlaufzeit des Algorithmus wesentlich zu verschlechtern.
In letzter Konsequenz bedeutet dies, dass es noch nicht einmal mehr nötig ist, das Suchmuster an jeder möglichen Position im Text zu platzieren. Es genügt zu jeder Probenkette im Text die passenden Probenketten im Muster zu suchen um alle potentiellen (und damit näher zu untersuchenden) Fundstellen zu erhalten.

Damit der Algorithmus funktioniert muss es offensichtlich für jeden Kandidaten mindestens einen Zeugen geben. Um unnötige Vergleiche zu vermeiden soll des Weiteren gefordert werden, dass es nur genau einen Zeugen gibt. Bei der Wahl des Testschemas ist hierauf zu achten.

Der Algorithmus besteht aus den folgenden 4 Schritten:

Algorithmus A: `generisch`

1. (Wahl von q, Q und $\mathcal{Q}$)
   Auswahl einer passenden Probengröße q, einer Vorlage Q und eines Testschemas $\mathcal{Q} = (\mathcal{Q}_P, \mathcal{Q}_T)$ für das gegebene Suchuster P und dem Text T über einem Alphabet Σ mit $|\Sigma| =: c$.
   Hierbei ist darauf zu achten, dass das Testschema genau einen Zeugen für jede mögliche Position des Musters im Text enthält.

2. (Vorverarbeitung von P)
   Für jede Probe $Q(i,j) \in \mathcal{Q}_P$ wird die zugehörige Probenkette $s_P := s(P, Q(i_P, j_P))$ bestimmt und in einer Datenstruktur gespeichert, welche es erlaubt anhand von s_P möglichst schnell $Q(i_P, j_P)$ bestimmen zu können.

3. (Test)
   Für jede Probe $Q(i_T, j_T) \in \mathcal{Q}_T$ im Text wird die zugehörige Probenkette $s_T := s(T, Q(i_T, j_T))$ ermittelt. Mit Hilfe der in Schritt 2 erzeugten Datenstruktur werden anhand von s_T eine Liste aller Proben $Q_P(i_P, j_P) \in \mathcal{Q}_P$ bestimmt, für die gilt $s(P, Q(i_P, j_P)) = s(T, Q(i_T, j_T))$. Hierdurch erhält man eine Liste von positiven Zeugen $(Q(i_P, j_P), Q(i_T, j_T))$ mit den dazugehörenden Kandidaten an den Positionen $(i_R, j_R) = (i_T - i_P, j_T - j_P)$ im Text. Diese müssen im nächsten Schritt genau überprüft werden.

4. (Überprüfung)
   Für die im vorhergehenden Schritt bestimmten Positionen muss das Suchmuster $P = (p_{ij})$ mit dem Text $T = (T_{ij})$ verglichen werden. Gilt $t_{i_R+I, j_R+J} = p_{I,J}$ für alle $0 \leq I, J < m$, so ist das Muster bei (i_R, j_R) im Text enthalten.

◎

Der variable Punkt ist die Wahl eines geeigneten Testschemas, also im besonderen die Wahl einer Probengröße und Vorlagenform. Erst hierdurch entsteht ein konkreter Algorithmus mit einer für den zugrunde liegenden Text und das zugrundeliegende Muster guten oder schlechten Laufzeit.

Es wird davon ausgegangen, dass nach verschiedenen Mustern in einem Text gesucht wird. Ist dies nicht der Fall und wird stattdessen ein Muster in vielen Texten gesucht, so kann es sinnvoll sein nicht das Muster, sondern den Text vorzuverarbeiten.

Im Folgenden sollen nun zwei mögliche Testschemata vorgestellt werden welche den generellen Algorithmus konkretisieren.

4.2 Quadratische Vorlagen

Die Zeugen bestehen aus den Verknüpfungen der Proben im Muster mit den Proben im Text, es gibt also $|\mathcal{Q}_P| * |\mathcal{Q}_T|$ Zeugen. Da diese Zahl nach Voraussetzung der Anzahl der möglichen Fundstellen im Text entsprechen muss, kann man offensichtlich die Anzahl der Proben im Text minimieren, wenn man die Anzahl der Proben im Muster maximiert. Durch die Minimierung der Proben im Text wird die Anzahl der zu bestimmenden Probenketten im Text und damit auch die Anzahl der zu suchenden passenden Probenketten im Muster minimiert. Die Maximierung der Proben im Muster ist unter der Annahme, dass das Muster nach einer Vorverarbeitung mehrfach benutzt wird und mit $m \ll n$, akzeptabel. In der nun folgenden Konkretisierung des generischen Algorithmus wird diese Idee mit Hilfe von "quadratischen Vorlagen" umgesetzt.

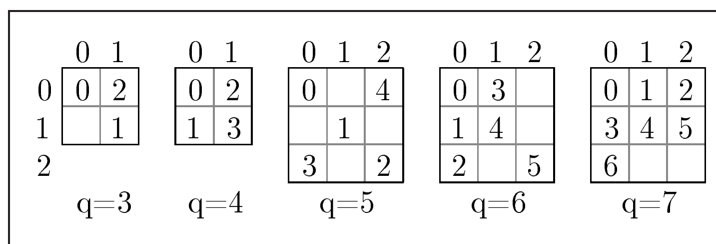


Abbildung 24: Quadratische Vorlagen

Definition 6:
Eine Vorlage $Q = ((0,0), (i_1, j_1), \ldots, (i_{q-1}, j_{q-1}))$ der Größe q heißt <u>quadratische Vorlage</u>, wenn gilt:

$$0 \leq i_k, j_k \leq \lceil \sqrt{q} \rceil - 1 \text{ für } 1 \leq k \leq q - 1.$$

In einer quadratischen Vorlage werden die Koordinaten also in einem möglichst kleinen quadratischen Bereich angeordnet. Die genaue Anordnung innerhalb dieses Bereichs ist bis auf den Ursprung beliebig.
Zur Vereinfachung der Implementierung ist es jedoch oft sinnvoll eine Anordnung wie in Abbildung 24 im Fall $q = 7$ zu wählen. Die einzelnen Testpunkte werden dabei von links nach rechts und oben nach unten in dieser Reihenfolge angeordnet. Alle freien Positionen befinden sich bezüglicher dieser Anordnung am Ende der Vorlage.

Die Größe q der Vorlage wird während der abschließenden Laufzeitanalyse des Algorithmus ermittelt.
Es wird sich dort ein Wert von $q := max\{1, \lceil log_c m^2 \rceil\}$ $(c := |\Sigma|)$ als optimal erweisen. Damit werden die Proben sehr klein. Geht man z.B. von $|\Sigma| = 26$ Zeichen aus, so ist die Probe bei einem gesuchten Muster von 5000×5000 Zeichen immer noch nur 5 Zeichen groß.

Die quadratischen Vorlagen werden an alle Positionen in P in der Art verteilt, dass P jede dieser Vorlagen ganz enthält:

$\mathcal{Q}_P = \{Q(i,j) \mid 0 \leq i,j \leq m - \lceil\sqrt{q}\rceil\}$

Um nur einen Zeugen pro Kandidaten zu erhalten wird die Vorlage im Text ähnlich wie in Abbildung 22 auf Seite 63 genau an den Schnittpunkten der $(m - \lceil\sqrt{q}\rceil + 1)$ten Zeilen und Spalten platziert. Für die Proben im Text ergibt sich somit:

$\mathcal{Q}_T = \left\{ Q\left(i(m - \lceil\sqrt{q}\rceil + 1) - 1, j(m - \lceil\sqrt{q}\rceil + 1) - 1\right) | 1 \leq i,j \leq \left\lfloor \frac{n - \lceil\sqrt{q}\rceil + 1}{m - \lceil\sqrt{q}\rceil + 1} \right\rfloor \right\}$

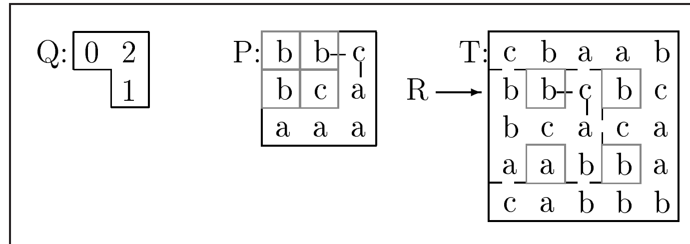


Abbildung 25: Die Lage der Proben

Beispiel 2:
Seien Q, P, T und R wie in Abbildung 23 auf Seite 66 gegeben, die Ursprünge der Proben in P und T sind mit $\square$ gekennzeichnet. Dann ist Q eine quadratische Vorlage (Aus Gründen der Anschaulichkeit wurde $q = 3$ gewählt, optimal wäre $q = \lceil log_c m^2 \rceil = \lceil log_3 3^2 \rceil = 2$).
Als Probenmengen ergeben sich (siehe Abbildung 25)

$\mathcal{Q}_P = \{Q(0,0), Q(0,1), Q(1,0), Q(1,1)\}$

$\mathcal{Q}_T = \{Q(1,1), Q(1,3), Q(3,1), Q(3,3)\}$

Der (positive) Zeuge für R in $\mathcal{Q} = (\mathcal{Q}_P, \mathcal{Q}_T)$ ist $(Q(0,1), Q(1,1))$ mit der Probenkette *bac*. ⌟

Um während der Testphase effektiv zu einer gegebenen Probenkette w eine Liste von Koordinaten zu den passenden Proben in P zu bestimmen, benötigt man eine Funktion $A(w)$ mit $A(w) := \{(i,j) \mid w = s(P, Q(i,j)),\ Q(i,j) \in \mathcal{Q}_P\}$. Diese Funktion benutzt intern eine Datenstruktur $\mathcal{D}$, die während der Vorverarbeitung von P gefüllt wird. $A(w)$ wird im Folgenden auch als *Adressenliste* von w bezeichnet.
Da mit der Maximierung der Proben in P auch die Anzahl der Einträge in $\mathcal{D}$ maximiert wird, kann durch die Wahl von $\mathcal{D}$ die Laufzeit des Algorithmus wesentlich beeinflusst werden.

Zwei mögliche Implementationen von $\mathcal{D}$ sind offensichtlich:
Eine erste Möglichkeit ist die Speicherung der Probenketten w in einem Trie mit $A(w)$ als Wert an den entsprechenden Blättern (siehe Beispiel 3). Es gibt in P maximal m^2 Proben, bei einer Länge von q Zeichen pro Probenkette ergibt sich damit eine Gesamtlänge von $\leq m^2q$ Zeichen. Je nach Implementierung des Tries über direkte Indizes oder über ausbalancierte Suchbäume kann der Trie somit in einer Zeit von $O(m^2qc)$ oder $O(m^2qlog(c))$ aufgebaut werden. Anfragen an A können auf diese Weise in einer Zeit von $O(q)$ oder $O(qlogc)$ beantwortet werden.

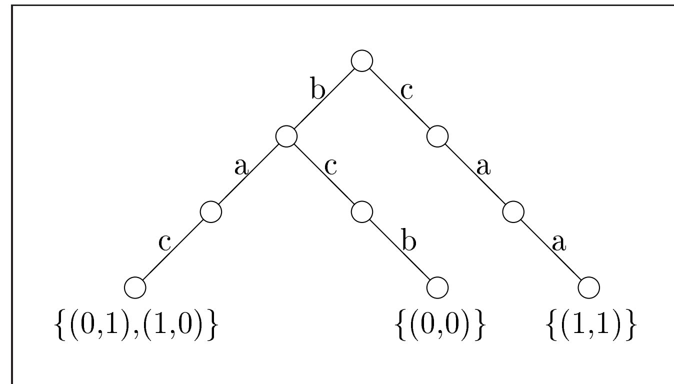


Abbildung 26: Implementierung als Trie

Beispiel 3:
Als Adresslisten ergeben sich
$A(bcb) = \{(0,0)\}$
$A(bac) = \{(0,1),(1,0)\}$
$A(caa) = \{(1,1)\}$
$A(w) = \{\emptyset\}$ für alle anderen w.
Dies führt zum Trie aus Abbildung 26. ⌟

Eine zweite Möglichkeit ist jede Probenkette w als Zahl in einem Zahlensystem zur Basis $c := |\Sigma|$ zu interpretieren. Wandelt man diese dann in ein Dezimalsystem um, so kann man die w als Indizes für ein Array der Größe c^q benutzen. In das Array wird jeweils die zu w gehörenden Adressliste eingetragen (siehe Beispiel 4 auf der nächsten Seite).
Durch die optimierte Wahl von q ist dabei c^q in den meisten Fällen ausreichend klein um die durch leere Arrayeinträge erzeugten Speicherverluste ignorieren zu können. Die Initialisierung des Arrays ist in einer Zeit von $O(c^q)$ möglich. Die gesamte Vorverarbeitung des Suchmusters ist mit einem trivialen Algorithmus und durch eine Basisumwandlung in konstanter Zeit insgesamt in einer Zeit von $O(m^2q)$ implementierbar. Anfragen können in einer Zeit von $O(q)$ beantwortet werden.

Beispiel 4:
Sei $c := |\Sigma| = 26$ und m so gewählt, dass sich $q = 3$ ergibt. Zur Speicherung der Probenketten genügt somit ein Array A mit $c^q = 26^3 = 17576$ Einträgen.. Sei $\Sigma := \{a, b, c, ..., z\}$ und seien die Zeichen als $a = 0, b = 1, c = 2, ...$ codiert. Seinen Q, P und T wie in Abbildung 23 auf Seite 66 gegeben.
Es gilt $s(P, Q(0,1)) = s(P, Q(1,0)) = bac$. Die Umwandlung von bac_{26} ins Dezimalsystem erfolgt nach den üblichen Methoden zur Basisumwandlung:
$bac_{26} = (b * 26^2 + a * 26^1 + c * 26^0)_d = 1 * 676 + 0 * 26 + 2 * 1 = 678$.
Es wird also $A[678] := \{(0,1), (1,0)\}$ gesetzt. ⌟

Anmerkung:

- Man könnte auch auf die Idee kommen die Zeichen einzeln binär zu codieren, zusammenzusetzen und danach ins Dezimalsystem zu wandeln. Dies ist sicherlich möglich, führt jedoch zu Speicherverlusten:
Um 26 Zeichen zu kodieren sind mindestens 5 Bits notwendig ($2^4 = 16$ Zeichen < 26 Zeichen $< 2^5 = 32$ Zeichen). Im Beispiel mit 3 Zeichen benötigt man also $3 * 5 = 15$ Bits. Man braucht also einen Array mit $2^{15} = 32768$ Einträgen. Dies ist beinahe das doppelte der bei der direkten Umwandlung benötigten 17576 Einträge.

- Mit ein wenig Aufwand und der Speicherung von Teilergebnissen ist es möglich die Vorverarbeitung sogar in einer Zeit von $O(m^2)$ anstatt $O(m^2q)$ durchzuführen. Dies setzt jedoch voraus, dass die Positionen innerhalb der Probe von links nach rechts und von oben nach unten angeordnet sind.
Die einzelnen Zeilen der Probe können als $\lceil\sqrt{q}\rceil$ Zahlen zur Basis c angesehen werden. Die ganze Probe kann man als $c^{\lceil\sqrt{q}\rceil}$-wertige Anzahl von $\lceil\sqrt{q}\rceil$ Zahlen (die Zeilen) interpretieren.
Jede Zeile einer Probe ist eine Teilzeichenkette einer Zeile aus P. Mit einem einfachen Durchlauf durch eine Zeile des Suchmusters kann man also die entsprechenden Anteile der Probenzeilen an der jeweiligen Probenkette in $O(m)$ Zeiteinheiten bestimmen. Anschliessend ist es möglich durch einen Durchlauf durch die Spalten von P diese Anteile spaltenweise zusammenzufassen. Die Details sollen hier nicht betrachtet werden, da bei genauerer Betrachtung - nicht zuletzt durch die "leeren" Positionen in den Proben - die Idee auf einen sehr komplexen Algorithmus führt. In praktischen Tests der Autoren wurde jedoch trotz des erheblich gesteigerten Aufwands eine etwas kürzere Laufzeit gemessen.

4.2.1 Laufzeitverhalten

Es soll nun ein Wert für q in der Art bestimmt werden, dass die Test- und Überprüfungsphase in ihrem Zeitbedarf minimiert wird. Dabei wird angenommen, dass der Text T zufällig im Sinne einer Bernoulli-Verteilung ist, d.h. jedes Zeichen kann an jeder Position unabhängig von anderen Zeichen mit einer Wahrscheinlichkeit von $\frac{1}{|\Sigma|}$ vorkommen. Als Erstes soll hierzu die Zeit betrachtet werden, welche man benötigt um einen Kandidaten R zu bearbeiten.

Sei $Q(i_T, j_T) \in \mathcal{Q}_T$ die Probe im Text innerhalb der Überdeckung von R. Die Testphase für R besteht aus der Berechnung von $w := s(T, Q(i_T, j_T))$ und der Bestimmung von $A(w)$, also dem Suchen von w in allen Probenketten von P. Diese Suche ist proportional zur Anzahl der zur Bestimmung von w benötigten Zeichen, also $O(q)$ (eine optimale Datenstruktur zur Speicherung der Probenketten von P vorausgesetzt). $Q(i_T, j_T)$ ist nicht nur in R, sondern in insgesamt $(m - \lceil\sqrt{q}\rceil + 1)^2$ Kandidaten enthalten, in einem Schritt werden also mehrere Kandidaten behandelt. Dies senkt die Laufzeit pro Kandidat, sie ist somit proportional zu $\frac{q}{(m-\lceil\sqrt{q}\rceil+1)^2}$.

Wurde in der Testphase ein positiver Zeuge gefunden, so wird die Überprüfungsphase eingeleitet. Hier wird mit einem naiven Algorithmus jedes Zeichen aus R mit jedem Zeichen aus P verglichen. Kommt es dabei zu einer Nichtübereinstimmung, so wird die Phase beendet, ansonsten ist R eine Fundstelle von P in T. Aufgrund der angenommen zufälligen Verteilung der Zeichen im Text wird die Überprüfungsphase mit einer Wahrscheinlichkeit von $\frac{1}{c^q}$ benötigt. Die stochastisch erwartete Anzahl von Vergleichen innerhalb dieser Phase liegt bei $\alpha := \frac{c}{(c-1)}$. Insgesamt kann also von weniger als $\frac{\alpha}{c^q} = \frac{1}{(c-1)c^{q-1}}$ Vergleichen in dieser letzten Phase ausgegangen werden.

Die Anzahl der Vergleiche soll nun durch ein geeignet gewähltes q minimiert werden. Fasst man die Vergleiche während der Tests und während der Überprüfung zusammen, so ergeben sich

$$\frac{q}{(m-\lceil\sqrt{q}\rceil+1)^2} + \frac{\alpha}{c^q} \qquad (\star)$$

Vergleiche. Durch Rückgriff auf die Methoden der Differentialrechnung erhält man nach Ableitung und elementarer Umformung mit Verzicht auf Rundungen und damit mit Verzicht auf ein ganzzahliges q

$$q = log_c m^2 - log_c \frac{m^2(m+1)}{(m-\sqrt{q}+1)^3} + log_c \frac{c \cdot ln(c)}{c-1}.$$

Dies führt auf die folgende Abschätzung für ein optimales q:

$$log_c m^2 - 1 < q < log_c m^2 + \frac{1}{2}$$

mit $m \geq 2$ und $c \geq 2$.

Eine gute Wahl für ein ganzzahliges q ist somit $q = \lceil log_c m^2 \rceil$, für den Fall $m = 1$ sei $q = 1$.

Setzt man dieses q in die Ausgangsgleichung $(\star)$ ein und berücksichtig, dass es maximal n^2 Kandidaten gibt, so ist es möglich eine Oberschranke für die Anzahl der in der Test- und Überprüfungsphase betrachteten Zeichen anzugeben:

$$n^2\left(\frac{\lceil log_c m^2\rceil}{(m-\left\lceil\sqrt{\lceil log_c m^2\rceil}\right\rceil+1)^2}+\frac{\alpha}{m^2}\right)\leq n^2\frac{4\lceil log_c m^2\rceil+\alpha}{m^2}=O\left(\frac{n^2 log_c m^2}{m^2}\right)=O\left(\frac{|T| log_c |P|}{|P|}\right)$$

(Hierbei wurde ausgenutzt, dass mit $c \geq 2$ und $m \geq 2$ gilt:
$m-\left\lceil\sqrt{\lceil log_c m^2\rceil}\right\rceil+1\geq\frac{m}{2}$)

Theorem 1:
Mit Hilfe von quadratischen Vorlagen der Größe $\lceil log_c m^2\rceil$ ist es möglich ein $m \times m$ großes Muster in einem $n \times n$ großen Text in einer Zeit von $O(\frac{n^2 log_c m^2}{m^2})$ zu finden. Die Vorverarbeitung von P benötigt hierbei $O(m^2)$ Speichereinheiten.

Bei großen Suchmustern ($|P| = \omega(n\sqrt{log_c n})$) kann die Zeit, welche zur Vorverarbeitung des Musters benötigt wird nicht mehr vernachlässigt werden, da sie dann die Gesamtlaufzeit des Algorithmus dominiert. Es ist denkbar in diesem Fall nur einen Teil des Suchmusters in der Vorverarbeitungs- und Testphase zu betrachten und erst in der Überprüfungsphase mit dem gesamten Muster abzugleichen. Nimmt das Suchmuster eine Fläche von $\Theta(n\sqrt{log_c n})$ ein, so steigt die Gesamtlaufzeit auf $O(n\sqrt{log_c n})$.

4.3 Lineare Vorlagen

Verzichtet man auf die Bedingung möglichst weniger Proben im Text und versucht stattdessen die Zeichen, welche im Text zu einer Probe gehören, zu minimieren, so gelangt man zu der Idee, sich überlappende Proben zu verwenden. Schafft man es, diese so zu verarbeiten, dass keines der benötigten Zeichen mehrfach verglichen werden muss, so erreicht man mit ein paar guten Methoden zum Einlesen der Proben einen ähnlich effektiven Algorithmus wie bei der Verwendung von quadratischen Vorlagen im vorigen Abschnitt.
Hierzu werden nun lineare Vorlagen verwendet. Eine lineare Vorlage besteht aus linear angeordneten Proben in einer Reihe des zugrunde liegenden Textes mit einem festen Abstand untereinander.

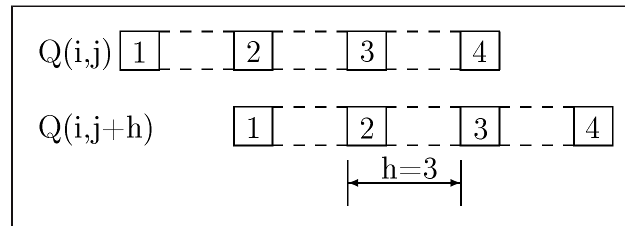


Abbildung 27: lineare Vorlage mit $q = 4$

Definition 7:
Eine lineare Vorlage $Q = ((0,0),(0,h),(0,2h),\ldots,(0,(q-1)h))$ der Größe q besteht aus q Positionen mit derselben vertikalen Position und einem festen horizontalen Abstand h mit $h \geq 1$ (siehe Abbildung 27).

Anmerkung:
$Q(i,j)$ und $Q(i,j+h)$ haben auf diese Weise immer $q-1$ Einträge gemeinsam.

Im Suchmuster werden die linearen Proben in jeder Zeile in die ersten h Spalten gelegt, es gibt hier also keinerlei Überlappung von Positionen, da jeweils eine Position auf ein "Loch" einer anderen Probe fällt. Im Text werden die Proben in jeder m-ten Zeile und dort in jeder h-ten Spalte positioniert. Dabei ist zu beachten, dass in den letzten $m-h$ Spalten des Textes keine Proben mehr liegen müssen, da dort bereits eine vollständige Überlappung des Suchmusters durch die übrigen Proben vorliegt.

Für das Testschema $(\mathcal{Q}_P, \mathcal{Q}_T)$ ergibt sich also:

$$\mathcal{Q}_P = \{Q(i,j) \mid 0 \leq i < m,\ 0 \leq j < h\}$$
$$\mathcal{Q}_T = \left\{Q(im-1, jh-1) \mid 1 \leq i \leq \left\lfloor \frac{n}{m} \right\rfloor,\ 1 \leq j \leq \left\lfloor \frac{n-m}{h} \right\rfloor + 1\right\}$$

Zur vollständigen Definition des Algorithmus bleibt noch die Bestimmung eines geeigneten h und eines geeigneten q.

Die Anzahl der zu $\mathcal{Q}_T$ gehörenden Zeichen im Text ist $\lfloor \frac{n}{m} \rfloor \left(\lfloor \frac{n-m}{h} \rfloor + q \right)$. Um diese Anzahl zu minimieren muss also h maximiert werden. Da jede mögliche Fundstelle im Text mindestens eine Probe enthalten muss, darf der Wert von h aber auch nicht größer als $\left\lfloor \frac{m}{q} \right\rfloor$ sein, der optimale Wert für h ist somit gefunden.
In der abschließenden Analyse wird sich q wieder mit einem Wert von $q = \lceil log_c m^2 \rceil$ als optimal erweisen.

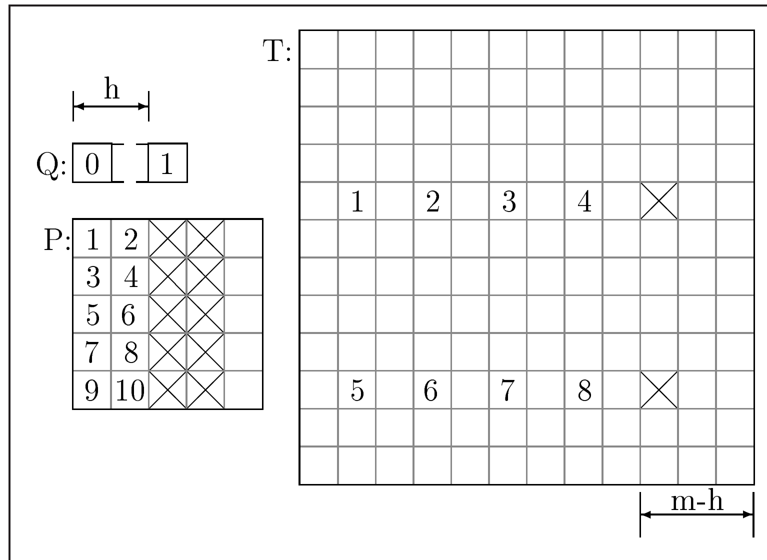


Abbildung 28: Suche mit linearen Vorlagen

Beispiel 5:
Sei P ein 5×5 großes Suchmuster welches in einem 12×12 großen Text T gesucht wird. Sei die Größe des benutzten Alphabets $c = 10$.
Es ergibt sich $q := \lceil log_c m^2 \rceil = \lceil log_{10} 25 \rceil = 2$,
daraus resultiert $h := \left\lfloor \frac{m}{q} \right\rfloor = \lfloor \frac{5}{2} \rfloor = 2$.
Es folgt für das Schema $(\mathcal{Q}_P, \mathcal{Q}_T)$:

$$\mathcal{Q}_P = \{Q(0,0), Q(0,1), Q(1,0), Q(1,1), Q(2,0), Q(2,1), Q(3,0), Q(3,1), Q(4,0), Q(4,1)\}$$

$$\mathcal{Q}_T = \{Q(4,1), Q(4,3), Q(4,5), Q(4,7), Q(9,1), Q(9,3), Q(9,5), Q(9,7)\}$$

Die Zahlen in P und T in Abbildung 28 symbolisieren die Ursprünge der benutzten Proben in der im Schema angegebenen Reihenfolge, die “X” sind Teile der Proben, welche keine Ursprünge darstellen. Dabei ist zu beachten, dass in T, mit Ausnahme der rechten Proben, jeweils die zweite Position einer Probe mit dem Ursprung einer weiteren Probe zusammenfällt. Im Muster ist dies nicht der Fall, hier liegt jeweils ein Teil einer Probe über dem “Loch” einer anderen. ⌟

Durch die Überlappung der Proben können die Probenketten im Text jeweils in konstanter Zeit ermittelt werden. Es genügt hierzu eine Art Fenster von der Größe q in der entsprechenden Zeile über den Text zu schieben und dieses auszulesen (vergleiche Abbildung 27 auf Seite 75 und Abbildung 29). Dabei muss in jedem Schritt nur das jeweils erste Zeichen gelöscht und das nächste hinzugefügt werden.

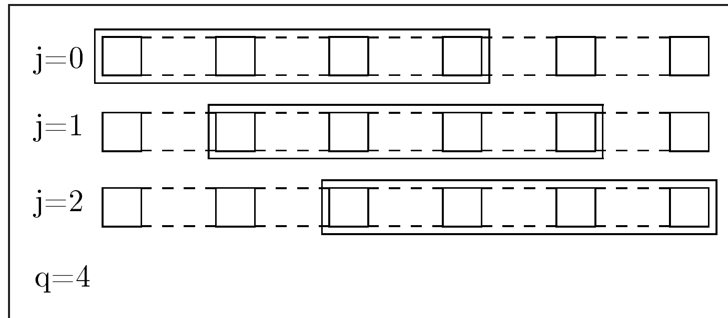


Abbildung 29: Auslesen der Probenketten

Durch die sich überlappenden Proben im Suchmuster ist eine Speicherung der zugehörigen Probenketten in einem Trie nicht mehr möglich. Man kann den Trie jedoch ohne Laufzeitverlust durch Fehlerübergänge zu einem Aho-Corasick "MultiPatternMatching"-Automaten erweitern[1]. Die Adressenlisten werden hierbei an den Zuständen des Automaten vermerkt. Der Zugriff auf eine zu einer Probenkette gehörende Adressliste kann auf diese Weise wieder in einer Zeit von $O(q)$ erfolgen.
Da in der Vorverarbeitung insgesamt $m \left\lfloor \frac{m}{q} \right\rfloor$ Proben mit einer Größe von je q Zeichen in den Automaten eingefügt werden müssen, wird eine Zeit von $O(m^2)$ Einheiten während der Vorverarbeitungsphase benötigt. Unter expliziter Berücksichtigung der an sich konstanten Alphabetgröße innerhalb der O-Notation und abhängig vom genauen Algorithmus zum Aufbau des dem Automaten zugrunde liegenden Tries ergibt sich analog zum vorigen Abschnitt eine Laufzeit von $O(m^2c)$ bzw. $O(m^2log(c))$. Durch die geringere Anzahl von Proben im Suchmuster sinkt die Vorverarbeitung also um einen Faktor von q.

Alternativ ist es auch möglich die Methode der Speicherung in Arrays aus dem vorhergehenden Abschnitt zu erweitern. In diesem Fall werden die Zeichen im "Fenster" jeweils als q einzelne Stellen einer Zahl zur Basis c aufgefasst. Diese Zahl kann direkt an jeder der $O(m^2)$ Positionen im Suchmuster ausgelesen werden, es ergibt sich somit analog zum vorhergehenden Abschnitt eine Zeit von $O(m^2)$ für die Vorverarbeitung des Suchmusters. Auch hier verringert sich also die Laufzeit der Vorverarbeitungsphase um einen Faktor von q.

Zu einer weiteren Verbesserung des Laufzeitverhaltens kann außerdem horizontal der Wert einer Probenkette ω' aus der zum vorhergehenden Fenster gehörenden Probenkette ω bestimmt werden. Hierzu wird die höchstwertige Stelle von ω gestrichen, der verbleibende Wert um eine Potenz erhöht und das neu hinzukommende Zeichen z in codierter Form addiert: $\omega' = (\omega\ mod\ (c^{q-1}))*c+z$. In jeder zu berechnenden Spalte können also bis auf die ganz linke alle Probenketten in konstanter Zeit bestimmt werden.

4.3.1 Laufzeitverhalten

Basis für die Berechnung des Laufzeitverhaltens sind wiederum die stochastischen Überlegungen und einige der mathematischen Überlegungen aus dem vorhergehenden Abschnitt.
Bereits bei der Bestimmung von h wurde die Anzahl der beteiligten Zeichen in der Testphase ermittelt. Da diese Anzahl proportional zur Laufzeit dieser Phase ist, kann hierfür eine Obergrenze abgeschätzt werden:

$$\left\lfloor \frac{n}{m} \right\rfloor \left(\left\lfloor \frac{n-m}{h} \right\rfloor + q \right) = \left\lfloor \frac{n}{m} \right\rfloor \left\lfloor \frac{n-m+qh}{h} \right\rfloor \leq \left\lfloor \frac{n}{m} \right\rfloor \left\lfloor \frac{n}{h} \right\rfloor \leq \frac{n^2}{mh} = \frac{n^2}{m \left\lfloor \frac{m}{q} \right\rfloor}$$

Mit den Überlegungen aus dem letzten Abschnitt ergibt sich hier eine stochastische Zeit von $\frac{n^2 \alpha}{c^q}$ für die anschließende Überprüfungsphase. Für beide Phasen ergibt sich also in der Summe eine Laufzeit von

$$n^2 \left(\frac{1}{m \left\lfloor \frac{m}{q} \right\rfloor} + \frac{\alpha}{c^q} \right).$$

Minimiert man diese Summe in Abhängigkeit von q, so ergibt sich mit den Methoden der Differentialrechnung ein reeller Wert für q:

$$q = log_c m^2 + log_c \frac{c\ ln(c)}{c-1}.$$

Für $c \geq 2$ gilt $0 < log_c \frac{ln(c)}{c-1} < \frac{1}{2}$. Damit ist $q := max\{1, \lceil log_c m^2 \rceil\}$ wiederum eine gute Wahl für q.
Setzt man diesen Wert in die Ausgangsgleichung ein, so ergibt sich nach einigen Umformungen als obere Grenze für die Gesamtlaufzeit der Test- und der Überprüfungsphase:

$$n^2 \left(\frac{1}{m \left\lfloor \frac{m}{q} \right\rfloor} + \frac{\alpha}{c^q} \right) \leq n^2 \left(\frac{2log_c m^2 + \alpha}{m^2} \right) = O\left(\frac{n^2 log_c m^2}{m^2} \right) = O\left(\frac{|T| log_c |P|}{|P|} \right)$$

Es ergibt sich das folgende Theorem:

Theorem 2:

Mit Hilfe von sich überlappenden linearen Vorlagen der Größe $\lceil log_c m^2 \rceil$ können alle Vorkommen eines $m \times m$ großen Musters in einem Text der Größe $n \times n$ in einer Zeit von $O(\frac{n^2 log_c m^2}{m^2})$ gefunden werden. Die Vorverarbeitung des Textmusters benötigt eine Zeit und einen Speicher von $O(m^2)$.

Wie im Fall von quadratischen Vorlagen kann auch bei diesem Algorithmus die für die Vorverarbeitung benötigte Zeit nicht mehr vernachlässigt werden wenn P zu groß wird ($|P| = \omega(n\sqrt{log_c n})$). Benutzt man in der Vorverarbeitung wieder nur einen Bereich der Größe $\theta(n\sqrt{log_c n})$ von P, so kann eine Vorverarbeitungszeit von $O(n\sqrt{log_c n})$ erzielt werden.

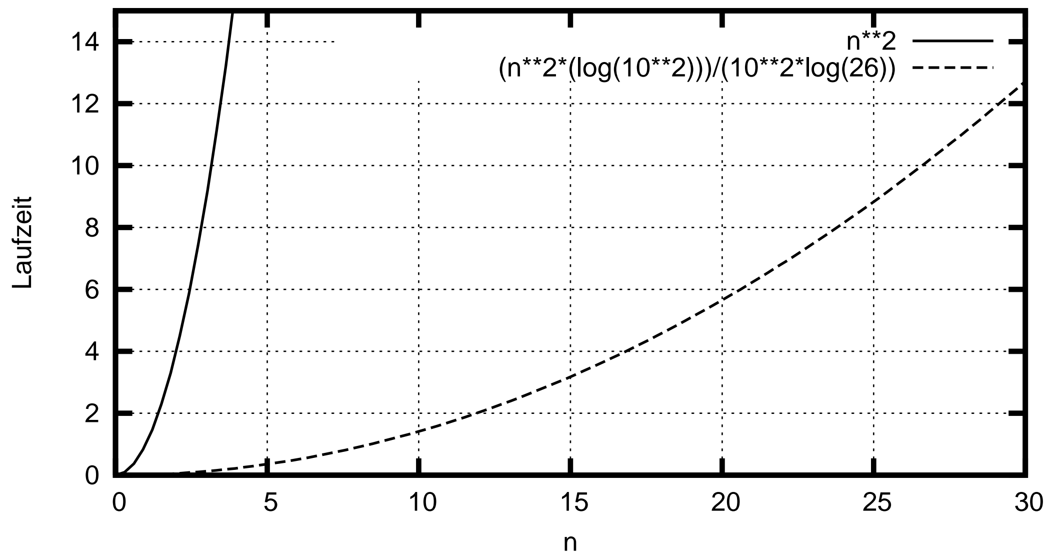
Problematisch wird der Algorithmus, wenn die Probengröße q die Länge der einzelnen Zeilen im Suchmuster überschreitet. In diesem Fall müssen die Proben auf mehrere benachbarte Zeilen erweitert werden.

4.4 Résumé

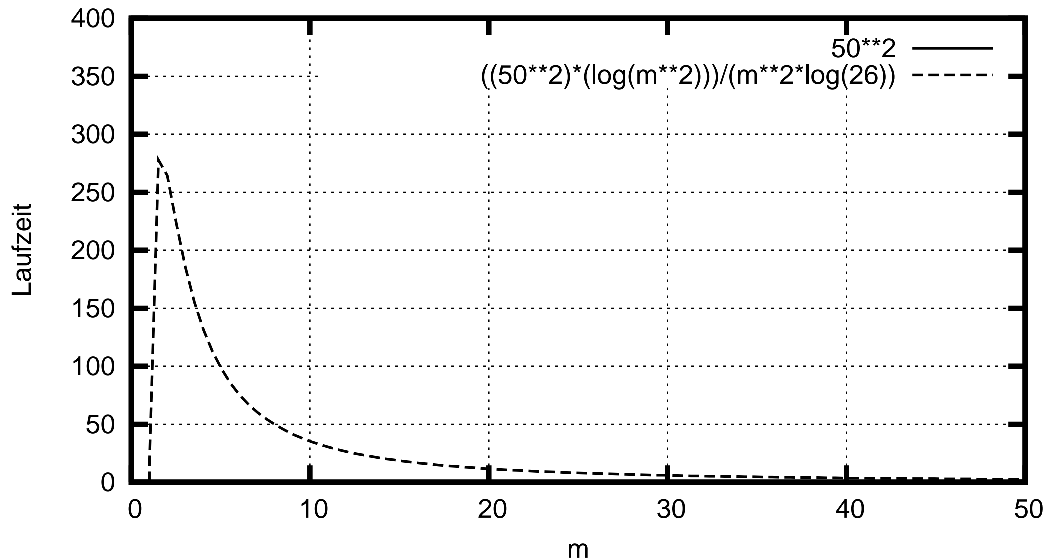
Dieser Algorithmus bietet viele Vorteile. Er ist sehr einfach zu implementieren und erreicht trotzdem eine sehr gute Laufzeit bei angemessenem Speicherverbrauch. Durch die Wahl von geeigneten Vorlagen und Testschemata lässt er sich gut an spezielle Probleme anpassen oder für Verwendungen in Bereichen außerhalb der exakten Mustersuche anpassen.
So beschreibt Park[22] Erweiterungen zur approximativen Zeichenkettensuche, insbesondere für das *k-mismatches*-Problem, also die Suche nach einem Muster welches bis auf k Ausnahmen mit dem Text übereinstimmt. Seine Änderungen bestehen im Wesentlichen darin mehr als einen Zeugen pro Kandidaten zu ermitteln. Aus der Anzahl der positiven und negativen Proben für einen Kandidaten können dann die gewünschten Aussagen gewonnen werden.
Eine weitere interessante Erweiterung ist der Übergang auf höhere Dimensionen. Dieser ist ohne Einschränkung und mit einer entsprechenden Laufzeit von $O(\frac{n^d log_c m^d}{m^d})$ möglich. Im Fall der linearen Vorlagen ist dabei zu beachten, dass die AC-Automaten in jeder Dimension einen Übergang auf eindimensionale Zeichenketten ermöglichen. Hier werden somit wieder die Ideen von Baker[8] und Bird[9] zur Lösung der zweidimensionalen Mustersuche aufgegriffen.

5 Laufzeitvergleich der vorgestellten Algorithmen

Geht man davon aus, dass die Größe des benutzten Alphabets konstant und sehr viel kleiner als m ist, so benötigt der Algorithmus von Amir, Benson und Farach eine Laufzeit von $O(n^2) = O(N)$. Die Suche mit dem Algorithmus von Kärkkäinen und Ukkonen mit quadratischen oder linearen Proben benötigt eine Zeit von $O(\frac{n^2 log_c m^2}{m^2})$.



Versuch 1: Laufzeit bei wachsender Textgröße



Versuch 2: Laufzeit bei wachsender Mustergröße

Für diese Beispiele wurde $|\Sigma| = 26$ angenommen. Im ersten Diagramm gilt $m = 10$. Es ist deutlich zu sehen, dass der Algorithmus von Amir, Benson und Farach bedeutend schlechtere Laufzeiten liefert. Untersucht man die Algorithmen bei wachsender Mustergröße, so liegt die Laufzeit bei einer konstanten Textgröße von $n = 50$ bei 2500 Einheiten (im Diagramm nicht zu sehen), die Laufzeit der anderen Algorithmen sinkt hingegen.
Es soll nun untersucht werden, ob sich diese theoretischen Werte auch in der praktischen Anwendung der Algorithmen niederschlagen.

Für die folgenden Untersuchungen wurden die Algorithmen in der Programmiersprache C implementiert. Als Compiler diente der GCC in der Version 3.3.6, die Untersuchung erfolgte unter Slackware Linux 10.1 mit Kernel 2.6.11 auf einem Intel Celeron Mobile Prozessor mit 1,4 GHz Taktfrequenz und 512 MB physikalischem Speicher. Es wurde darauf geachtet, dass zur Zeit der Untersuchung kein Speicher ausgelagert werden musste, ferner wurden systemfremde Prozesse unterbunden.
Als Gegenprobe wurden alle Versuche mit vergleichbaren Ergebnissen unter Solaris 10 auf einer Sun Ultra 60 mit Sparc Prozessor (360 MHz) und 2 GB RAM wiederholt.
Zur Vereinfachung wurde bei der Implementierung der Vorverarbeitungsphase der Algorithmus von Main und Lorentz durch einen naiven Algorithmus ersetzt, auf den Aufbau eines Suffix-Baums wurde verzichtet. Anhand der ermittelten Laufzeiten ist jedoch deutlich zu ersehen, dass diese Vereinfachung keinen nennenswerten Einfluss auf die Gesamtlaufzeit hat, da die Vorverarbeitungsphase ohnehin keinen großen Anteil dieser Zeit ausmacht.

Für die Versuche wurden Texte und Suchmuster aus verschiedenen Texten generiert. Die Generierung geschah, mit einer Ausnahme, indem die Buchstaben des zugrunde liegenden Textes nacheinander eingelesen und zum Testtext und Testmuster zusammengesetzt wurden. Folgende Eingaben wurden benutzt:

- Bibel: Die deutsche Ausgabe der lutherischen Bibel, Altes und Neues Testament in revidierter Fassung. Zur Vereinfachung wurden nur die Buchstaben benutzt und diese zusätzlich in Großbuchstaben gewandelt. Hieraus resultiert ein Alphabet von 26 Zeichen.

- allesX: Das Alphabet bestand ausnahmslos aus dem Buchstaben “X”. Getestet wurde mit diesem Muster trotzdem mit einem angenommenen Wert von $|\Sigma| = 2$.

- Binär: Eine zufällige Folge von “X” und “-”, also ein rein binäres Alphabet.

- 1zu10: Eine zufällige Folge der Zeichen “X” und “ ”, das Verhältnis dieser Zeichen untereinander betrug 1:10.

- alpha: Eine Folge von zufällig gewählten Zeichen aus der Menge $\{A, B, \ldots, Z\}$.

Zur Erzeugung der Zufälligkeit wurde jeweils die rand-Funktion des C-Compilers benutzt.

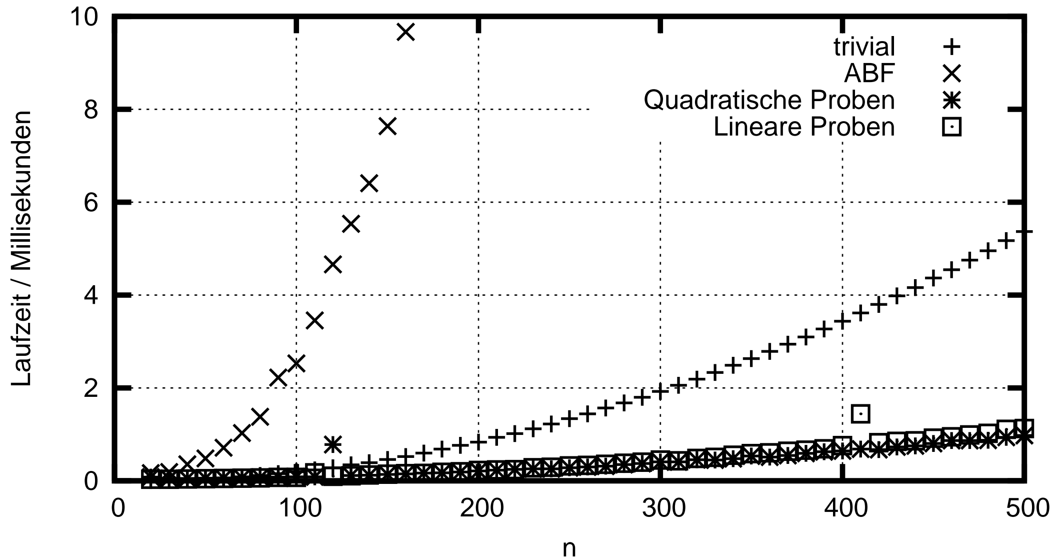
Als Sonderfall wurde außerdem die Suche in Bilddateien behandelt, da hier durch zusammenhängende Flächen oder weiche Übergänge andere Laufzeitverhalten erwartet werden könnten. Um diesen Effekt zu verstärken wurde kein Photo mit vielen verschiedenen Farben und kleinen Einzelelementen, sondern ein Bild aus einer Zeichentrickserie mit nur 64 Farben benutzt. Die einzelnen Farben wurden dabei als ASCII-Zeichen codiert. Um die Besonderheiten eines Bildes nicht zu zerstören wurden die verschieden großen Bilder als Auschschnitte des Gesamtbildes in der oberen linken Ecke erzeugt.

Für spezielle Untersuchung wurden außerdem die folgenden Texte und Suchmuster verwendet:

- pattern: Das Muster aus Abbildung 11 auf Seite 28
- text4: Der Text aus Abbildung 11 auf Seite 28 4 mal aneinander gelegt um insgesamt einen 40×40 Zeichen großen Text zu erhalten.
- Bildausschnitt: Ein Ausschnitt des für die Bildersuche gewandelten Bildes.
- Bild: Das für die Bildersuche gewandelte Bild als Ganzes.

Zusätzlich zu den vorgestellten Algorithmen wurde ein trivialer Algorithmus implementiert. Dieser legt das Suchmuster in der Art an alle Positionen im Text, dass es in diesem ganz enthalten ist. An jeder Position werden Suchmuster und Text zeichenweise verglichen, bei der ersten Nichtübereinstimmung wird mit der nächsten Position fortgefahren. Im besten Fall wird somit eine Laufzeit von $(n-m)^2$ erreicht, im schlimmsten Fall eine Laufzeit von $(n-m)^2 * (m^2)$.

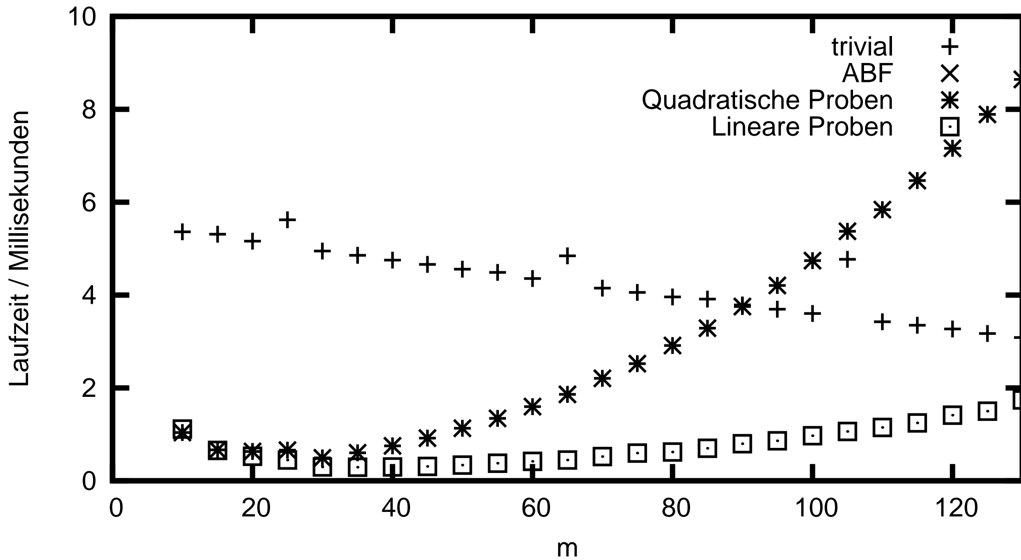
Als Erstes sollen die Laufzeiten hinsichtlich steigender Textgrößen anhand der Bibel verglichen werden.



Versuch 3: Vergleich bei wachsender Textgröße

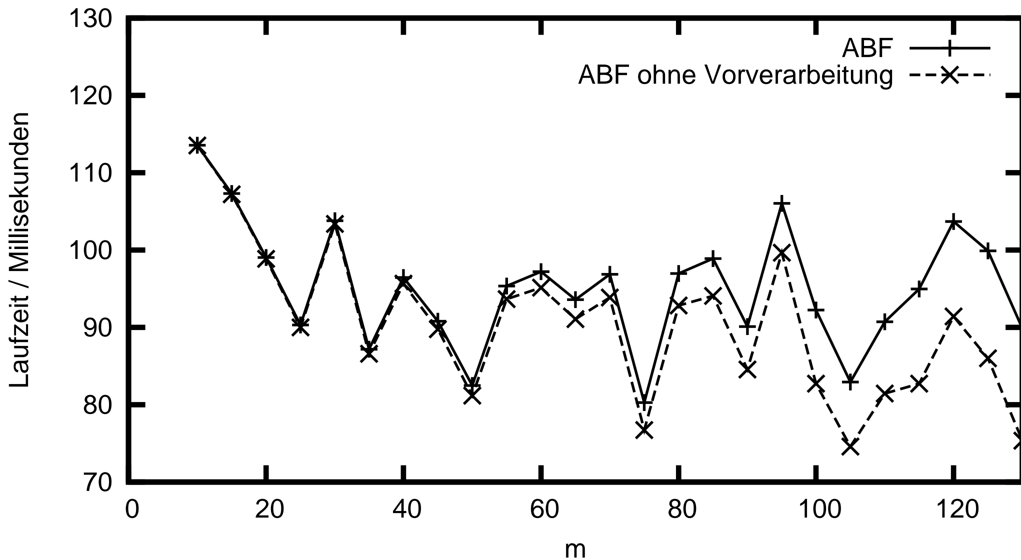
Wie erwartet benötigt der Algorithmus von Amir, Benson und Farach (ABF) eine deutlich höhere Laufzeit als die Suche mit Hilfe von Proben. Die Laufzeit ist sogar höher als bei der trivialen Implementierung. Der Grund liegt darin, dass hier im Gegensatz zum trivialen Verfahren in den einzelnen Schritten der Konsistenzüberprüfung mehrmals der gesamte Text durchlaufen wird.
Der triviale Algorithmus kann als Variante des Algorithmus von Kärkkäinen und Ukkonen mit einer variablen Probengröße angesehen werden. Die Probengröße entspricht hierbei der Anzahl der übereinstimmenden Zeichen im Text und im Muster, welche verglichen werden müssen, bevor der Algorithmus abbricht. In einem Text mit zufällig gewählten Zeichen aus einem großen Alphabet sind dies in der Regel sehr wenige Zeichen, an jeder möglichen Positionen bricht der triviale Algorithmus mit einer Wahrscheinlichkeit von $\frac{1}{|\Sigma|}$ schon nach einem Vergleich ab.
Im Gegensatz zum trivialen Algorithmus können bei der Benutzung von quadratischen und linearen Proben mit jedem Vergleich mehrere Kandidaten als Fundstellen ausgeschlossen werden, dies führt zu einer deutlichen Verringerung der Laufzeit. Da bei den linearen Proben die Anzahl der untersuchten Probenketten im Text nicht maximal minimiert wurde, benötigt dieser Algorithmus erwartungsgemäß ein wenig mehr Zeit als die Suche mit quadratischen Proben.

Untersucht man nun die Laufzeit in Abhängigkeit von der Größe des gesuchten Musters, so ist es zweckmäßig, den Algorithmus von Amir, Benson und Farach aufgrund der wesentlich höheren Laufzeit getrennt darzustellen.



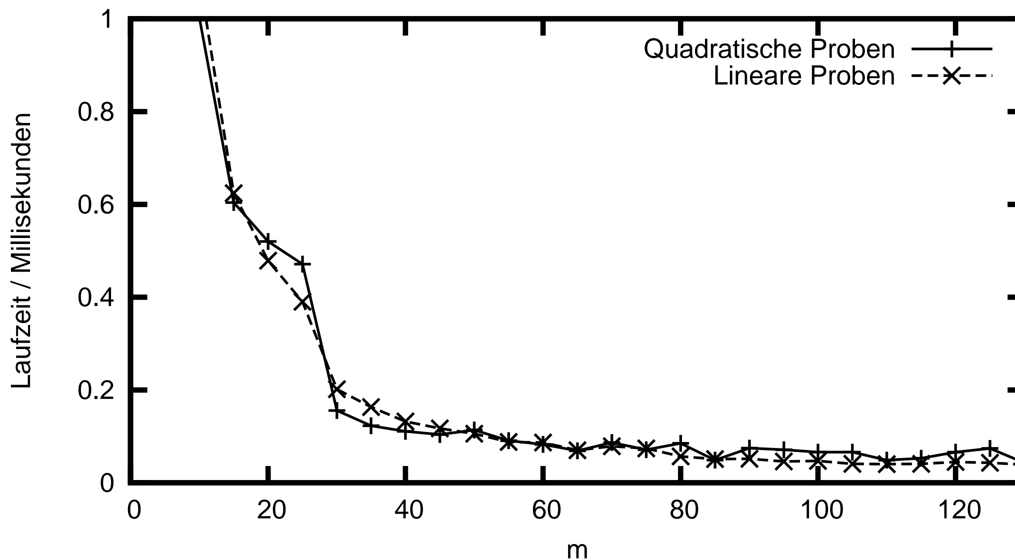
Versuch 4: Vergleich bei wachsender Mustergröße

Die anderen Verfahren unterscheiden sich in der Laufzeit nicht wesentlich, es ist jedoch auffällig, dass nur der triviale Algorithmus mit größeren Suchmustern eine geringere Laufzeit benötigt. Dies liegt daran, dass im Text nur der Bereich $n - m$ untersucht werden muss. Mit steigender Mustergröße m sinkt also die Anzahl der zu untersuchenden Kandidaten.
Auffällig ist, dass die linearen Proben bei größeren Mustern ein besseres Laufzeitverhalten als die quadratischen Proben besitzen. Hinsichtlich der Größe des Eingabetextes waren sie jedoch schlechter. Am Ende dieser Untersuchungen wird sich dieser Unterschied aufklären.



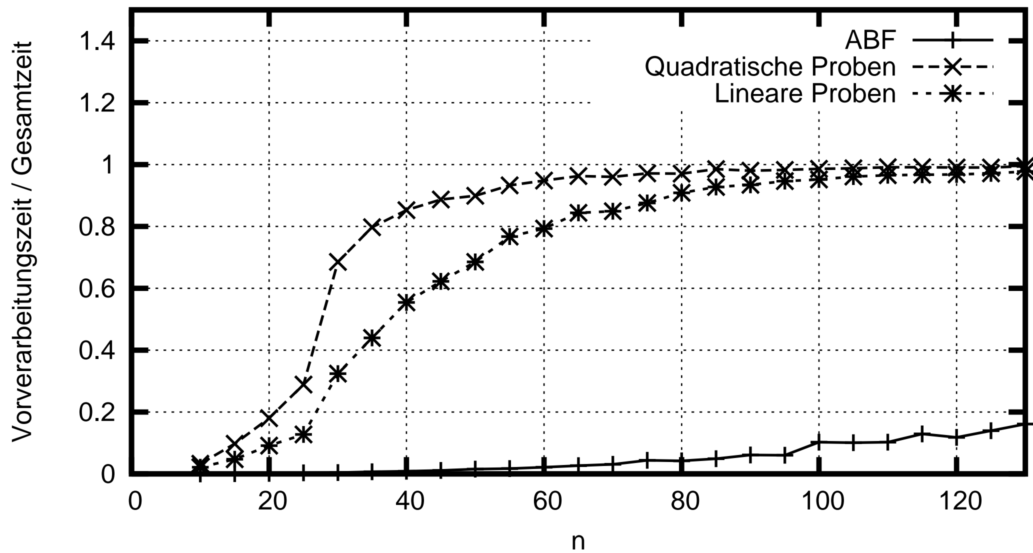
Versuch 5: ABF mit wachsender Mustergröße

Beim Algorithmus von Amir, Benson und Farach ist zu beobachten, dass auch dieser Algorithmus mit wachsender Größe des gesuchten Musters etwas schneller läuft. Gleichzeitig fällt auf, dass selbstverständlich auch die für die Vorverarbeitung benötigte Zeit wächst, diese jedoch die Gesamtzeit nie dominiert.
Betrachtet man auch die anderen Algorithmen ohne Berücksichtigung der Vorverarbeitungszeit, so werden diese ebenfalls mit wachsender Mustergröße beschleunigt. Die Vorverarbeitungszeit darf bei großen Mustern also offensichtlich nicht mehr vernachlässigt werden.



Versuch 6: Wachsende Mustergröße ohne Vorverarbeitung

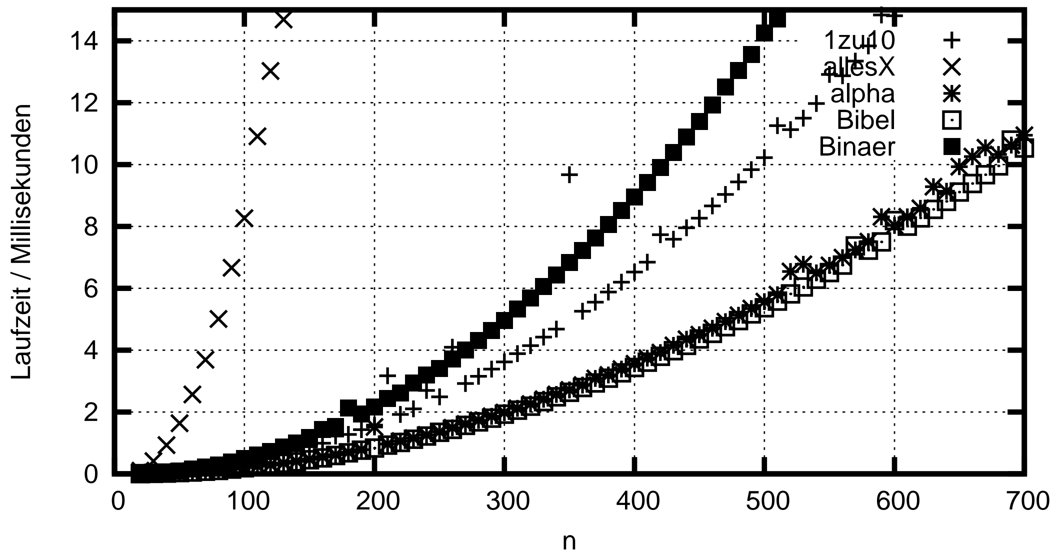
Für die Suche eines (großen) Musters in vielen Texten ist somit den Algorithmen nach dem Verfahren von Kärkkäinen und Ukkonen der Vorzug zu geben, für die Suche in einem einzigen Text sollte man den Algorithmus von Amir, Benson und Farach benutzen.
Noch deutlicher wird das Problem der eigentlich schnelleren Algorithmen beim genauen Vergleich von Vorverarbeitungszeit zur Gesamtlaufzeit.



Versuch 7: Verhältnis der Vorverarbeitungszeit zur Gesamtlaufzeit

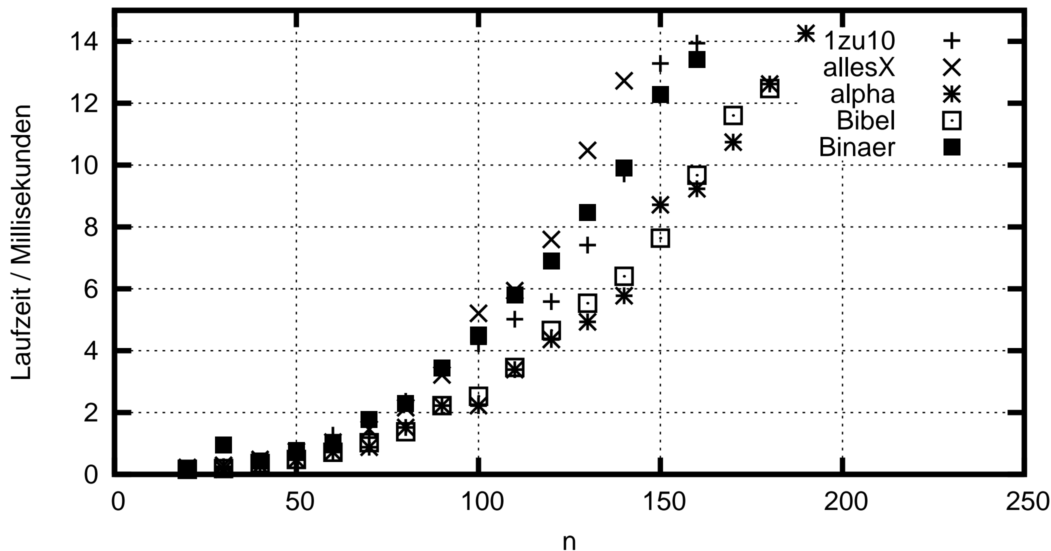
Bei einer Textgröße von $n = 500$ überwiegt in den Algorithmen, welche mit quadratischen bzw. linearen Proben arbeiten, bereits bei einer Mustergröße m von 30 - 40 Zeichen die Vorverarbeitungszeit. Beim Algorithmus von Amir, Benson und Farach hingegen ist selbst bei einer Größe von 130 Zeichen noch nicht einmal ein Anteil der Vorverarbeitungszeit an der Gesamtlaufzeit von 20% erreicht.

Die Güte eines Algorithmus hängt also auch von der jeweiligen Anwendung ab, es gibt in dieser Hinsicht nicht **den** besten Algorithmus, sondern nur denjenigen, welcher für das aktuelle Problem die besten Laufzeiten (und eventuell den geringsten Speicheraufwand) bietet.
Diese Erkenntnisse führen zum nächsten Versuch. Es soll anhand einiger Beispiele die Auswirkung verschiedener Texte auf die Laufzeiten ermittelt werden um herauszufinden, ob eventuell einige Algorithmen für bestimmte Probleme besser oder schlechter geeignet sind.



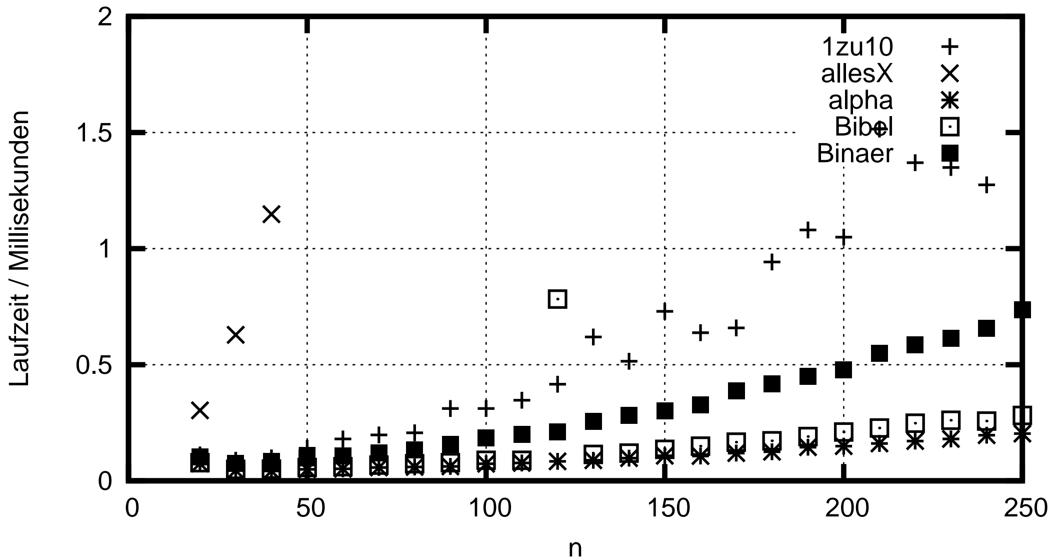
Versuch 8: Verschiedene Eingangsdaten (trivial)

Der triviale Algorithmus ist besonders schlecht für kleine Eingabealphabete. Da mit wachsender Alphabetgröße die Wahrscheinlichkeit einer Nichtübereinstimmung eines Zeichens im Text mit einem Zeichen im Suchmuster ebenfalls steigt, ist dieses Verhalten klar.

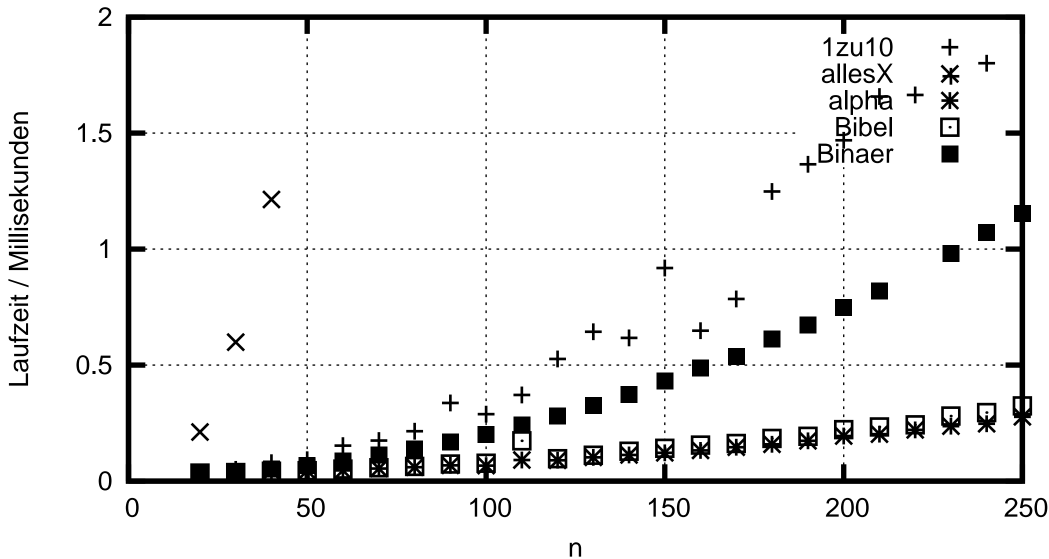


Versuch 9: Verschiedene Eingangsdaten (ABF)

Der Algorithmus von Amir, Benson und Farach verhält sich ähnlich wie der triviale Algorithmus, die Abhängigkeit der Laufzeit zur Alphabetgröße ist jedoch weit weniger ausgeprägt.



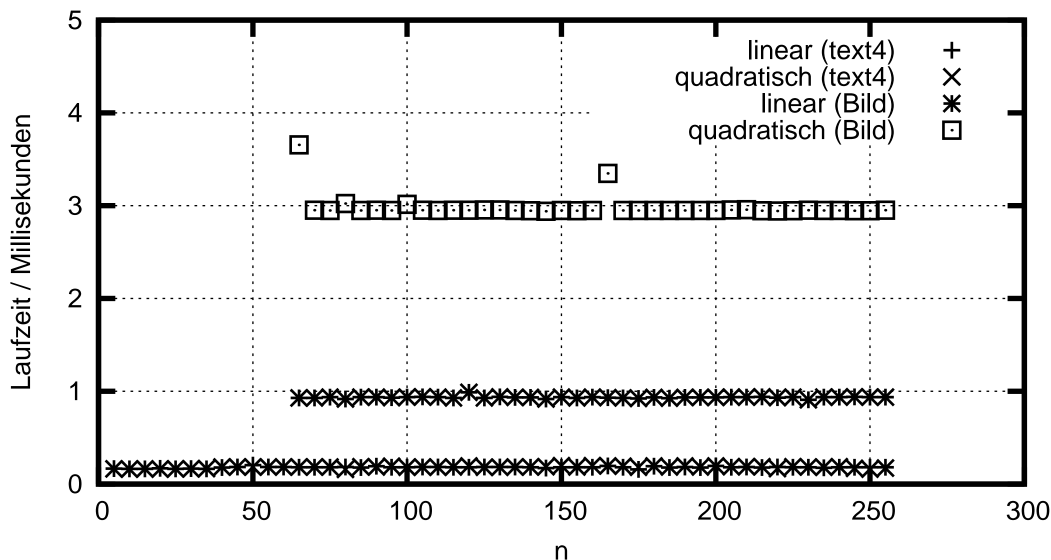
Versuch 10: Verschiedene Eingangsdaten (quadratisch)



Versuch 11: Verschiedene Eingangsdaten (linear)

Sowohl bei der Verwendung von quadratischen, als auch bei der Verwendung von linearen Proben ist deutlich der Ansatz dieser Algorithmen erkennbar. Sie basieren auf der Annahme, dass es nur wenige Fundstellen gibt. Bei jeder erfolgreich geprüften Probe muss das ganze Muster mit dem Text verglichen werden. Im Extremfall mutieren diese Algorithmen damit zum trivialen Verfahren. Sie haben dann sogar eine wesentlich schlechtere Laufzeit, da zusätzlich die zur Vorverarbeitung benötigte Zeit beachtet werden muss.

Beim Versuch mit "allesX" und "1zu10" ist die Wahrscheinlichkeit hoch, dass selbst wenn keine Fundstelle vorliegt die Probenketten von Muster und Text übereinstimmen. Diese Eingabealphabete schneiden daher besonders schlecht ab.
Da außerdem mit wachsender Alphabetgröße die Größe des Templates (q) sinkt, ist auch hier die Suche in den beiden alphabetischen Texten am schnellsten.
Es ist nicht bei jedem Problem möglich die exakte Alphabetgröße (ohne nennenswerte Zeitverluste) vor dem Start des Algorithmus zu bestimmen. Denkt man zum Beispiel an die Suche in Bildern, so kennt man vorher nicht zwingend die Anzahl der benuzten Farben. Die Abhängigkeit von der tatsächlichen Alphabetgröße wurde bereits festgestellt, es bleibt jedoch zu untersuchen, ob eine falsche Wahl der Alphabetgröße bei Berechnung von q gravierende Auswirkungen hat. Hierzu wurde eine einfache Suche mit unterschiedlich gesetzten maximalen Alphabetgrößen durchgeführt.

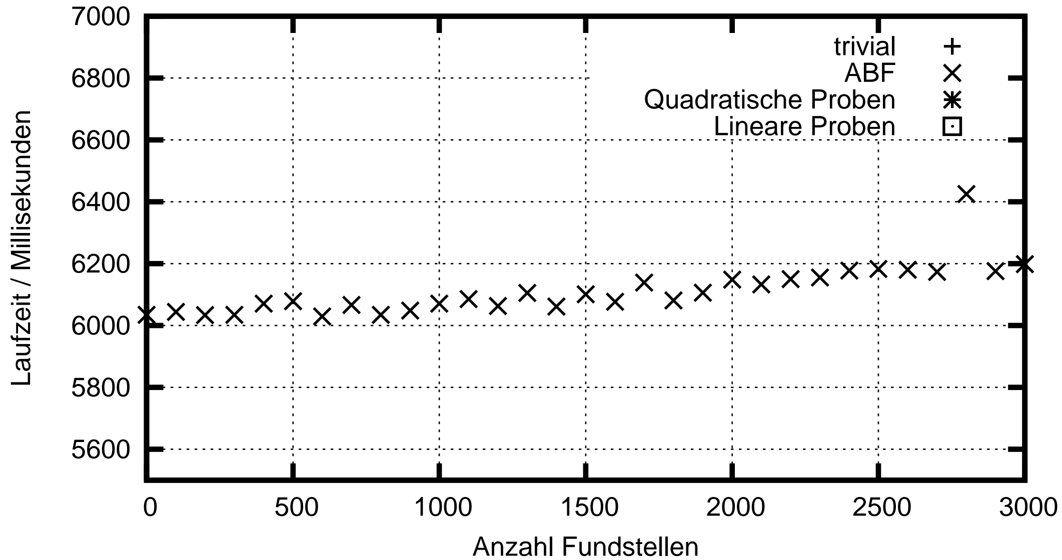


Versuch 12: Vergleich bei wachsender Alphabetgröße

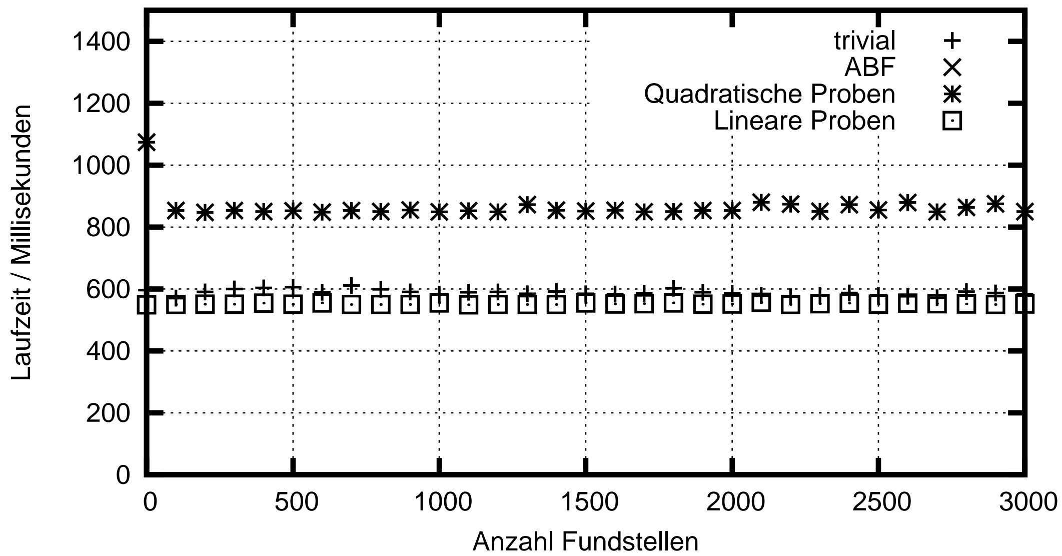
Man sieht keinerlei Auswirkung. Dies liegt vor allen Dingen in der dämpfenden Wirkung des Logarithmus begründet. In der Regel genügt es also eine "Größenordnung" für die Größe des benutzten Alphabets anzugeben.

Es fällt ins Auge, dass bei jedem Algorithmus eine Verarbeitung eines Textes, welcher nur aus Fundstellen besteht, mit sehr langen Laufzeiten verbunden ist. Da es sich hier jedoch um einen sehr extremen Fall handelt, soll nun untersucht werden, ob die Laufzeiten auch in weniger extremen Beispielen mit der wachsenden Anzahl von Fundstellen steigen.
Für diesen Test wurde ein 5×5 großes Suchmuster benutzt, welches in der Diagonalen aus "X" und sonst nur aus "-" besteht. Gesucht wird in einem

4000×4000 großen Text, welcher zu Beginn nur aus "-" besteht. In jedem Testdurchlauf wird nun die Diagonale von oben nach unten mit "X"-Zeichen gefüllt. In jedem Durchlauf existiert also eine Fundstelle mehr.



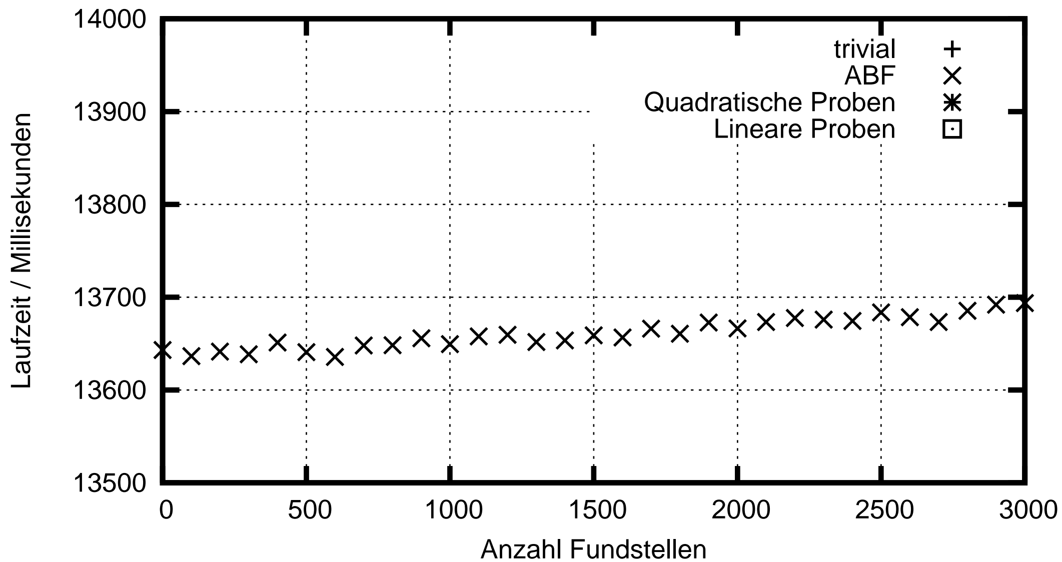
Versuch 13: Vergleich bei wachsender Zahl von Fundstellen



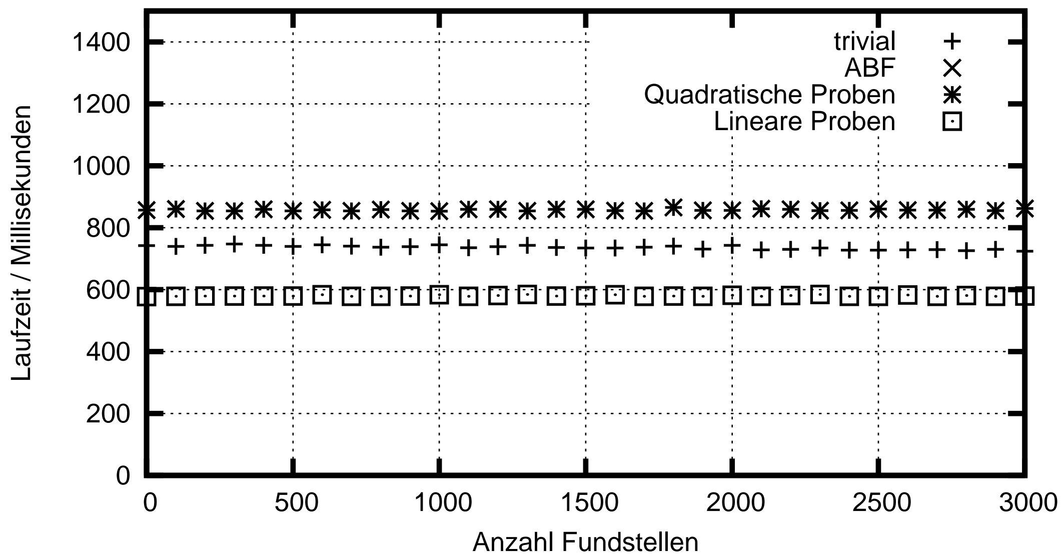
Versuch 14: Vergleich bei wachsender Zahl von Fundstellen

Beim Algorithmus von Amir, Benson und Farach ist ein deutlicher Anstieg der Laufzeit zu erkennen. Dieser ist bedingt durch eine höhere Anzahl konsistenter Kandidaten und der damit verbundenen erhöhten Anzahl von Konsistenztests und Vergleichen von Muster und Text. Die anderen Algorithmen sind in ihren Laufzeiten konstant, da hier mit jeder neuen Fundstelle nur 5^2 Vergleiche hinzukommen. Im Vergleich zu den 4000^2 Positionen ist diese Laufzeiterhöhung

jedoch irrelevant.
Die benutzten Texte enthalten keine Positionen, die “fast” eine Fundstelle darstellen. Deshalb wird der Eingabetext nun um weitere “X”-Werte erweitert. Diese liegen außerhalb eines Korridors von m-Zeichen um die Diagonale in jeder zweiten Zeile und sind zufällig verteilt. Auf diese Weise werden keine neuen Fundstellen erzeugt, die Algorithmen könnten jedoch einige Positionen finden, die in sich konsistent sind, bzw. bei denen die Probenketten übereinstimmen. Damit ist es wahrscheinlicher, dass die Laufzeit tatsächlich steigt.



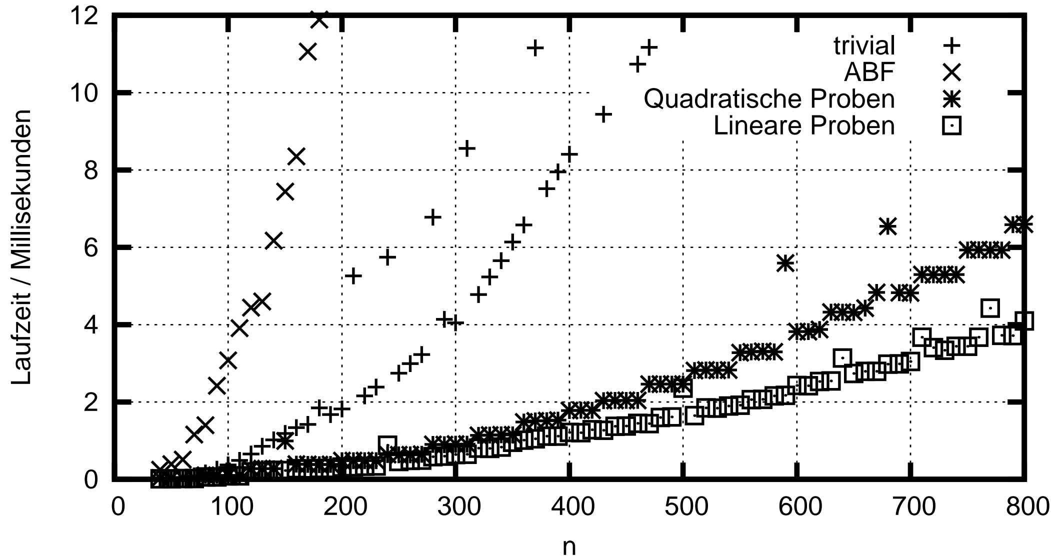
Versuch 15: Vergleich bei wachsender Zahl von Fundstellen



Versuch 16: Vergleich bei wachsender Zahl von Fundstellen

Es ist zu beobachten, dass das triviale Verfahren und der Algorithmus von Amir, Benson und Farach durch die Erhöhung der "Beinahe" Fundstellen eine sehr viel größere Zeit benötigen. Die auf Kärkkäinen und Ukkonen basierenden Verfahren bleiben konsistent. Durch die gewählte Anordnung und Form der Proben entstehen auch bei diesem schwierigerem Text nur selten Situationen, in denen die Probenketten übereinstimmen, die aktuelle Position aber dennoch keine Fundstelle ist.

Hinsichtlich der Anwendungsgebiete ist es interessant die Algorithmen mit Bilddaten zu testen.



Versuch 17: Bildersuche

Auch hier verhalten sich die Algorithmen wie erwartet. Auffällig ist nur die Diskrepanz zwischen der Suche mit linearen und quadratischen Proben. Mit Hinblick auf das Verhalten dieser Algorithmen bei wachsender Größe der Suchmuster ist ein Grund in der gewählten relativ hohen Mustergröße von $m = 40$ zu finden. Dies ist aber sicherlich nicht die einzige Ursache, eine weiterere liegt in der Art der benutzten Daten. Während in der Suche in einem realen Text die Suche mit quadratischen Proben ein klein wenig effektiver war, scheint bei der Suche in Bilddaten die Verwendung von linearen Proben zweckmäßiger. Untersucht man auch die anderen Diagramme hinsichtlich dieser Auffälligkeit, so erkennt man, dass bei allen Versuchen, welche auf strukturierten Daten beruhen, die linearen Proben die bessere Wahl sind, obwohl sie in der Theorie durch die erhöhte Anzahl von Proben eine geringfügig schlechtere Laufzeit besitzen. Besonders in der Untersuchung der Algorithmen bei wachsender Alphabetgröße wird dies noch einmal sehr deutlich. Bei der Untersuchung der Textprobe benötigen beide Varianten annähernd dieselbe Zeit. Bei den Bildproben benötigt der quadratische Algorithmus jedoch ungefähr 2 ms mehr Zeit.

Aus diesen Untersuchungen kann man abschließend das Fazit ziehen, dass der Algorithmus von Kärkkäinen und Ukkonen in den meißten Fällen sehr viel bessere Laufzeiteigenschaften besitzt als der Algorithmus von Amir, Benson und Farach. Die Wahl der Vorlagen hängt jedoch entscheidend von der Art der verarbeiteten Daten ab.
Nur bei der Suche nach einzelnen größeren Suchmustern sollte der Algorithmus von Amir, Benson und Farach benutzt werden, da in diesen Fällen die Vorverarbeitungszeit bei Kärkkäinen und Ukkonen die Gesamtlaufzeit dominiert und den eigentlich schnelleren Algorithmus effektiv langsamer laufen lässt.

6 Résumé und Aussicht

Es wurden in dieser Arbeit einige grundlegende Ideen und Algorithmen für die mehrdimensionale Mustersuche vorgestellt. Diese bilden eine Basis für weitergehende Forschungen und die meisten der bisher bekannten Algorithmen.

Im Vergleich zeigte sich jedoch, dass keiner der Algorithmen generell für jedes Problem gut geeignet ist. Ein Verfahren welches für jedes Problem optimal anwendbar ist und dabei idealer Weise eine einfache Implementierung gewährleistet gibt es bisher nicht.

Der Algorithmus von Amir, Benson und Farach hat sich in den vorangegangenen Untersuchungen als eher schlecht erwiesen. Aufgrund seiner internen Struktur ist er jedoch sehr leicht parallelisierbar. Man kann in diesem Fall einen um nahezu den Faktor $\frac{1}{n}$ schnelleren Algorithmus erwarten. In Verbindung mit großen Suchmustern ist dann auch der nicht unerhebliche Aufwand bei der Implementierung gerechtfertig.
Eine Erweiterung des Alorithmus auf höhere Dimensionen ist denkbar, eine Anwendung für die approximative Mustersuche jedoch nicht.

Die Algorithmen nach dem Verfahren von Kärkkäinen und Ukkonen hingegen haben schon auf einer Einprozessormaschine in der Regel sehr gute Laufzeiteigenschaften und sind sehr einfach zu implementieren. Die Autoren geben in Anlehnung der Berechnungen von Yao[24] einen Beweis dafür, dass die Laufzeit eines idealen d-dimensionalen Suchalgorithmus bei $\Omega\left(\frac{n^d log_c m^d}{m^d}\right)$ liegt. In diesem Sinne ist der von Ihnen angegebene Algorithmus also nicht nur einfach, sondern auch optimal. Hinzu kommt, dass mit leichten Änderungen mit diesen Algorithmen das k-mismatches-Problem gelöst werden kann.
Diese approximative Suche ist zusammen mit der Übetragbarkeit auf höhere Dimensionen eine sehr bedeutende Erweiterung.

Insgesamt jedoch bleibt die Hauptanwendung aller vorgestellten Algorithmen die exakte Suche.
Besonders im Hinblick auf die Anwendung innerhalb von biometrischen Authentifizierungssystemen gewinnt in den letzten Jahren jedoch auch die approximative Suche zunehmend an Bedeutung. Für diese stehen bisher neben den Erweiterungen des Algorithmus von Kärkkäinen und Ukkonen nur wenige Verfahren zur Verfügung. Einige Ansätze für eine abgeschwächte Form der approximativen Suche, einer Suche nach rotierten Mustern im Text, lieferten Amir und Farach[7]. Mit der wirklichen approximativen Suche beschäftigen sich hauptsächlich Navarro und Yates [21][25] direkt, oder indirekt als Initiatoren für weiterführende Algorithmen.

Literatur

[1] A.V. Aho, M.J. Corasick. *Efficient string matching: An aid to bibliographic search*, Comm. ACM, Ausgabe 18, Seite 333-340, 1975.

[2] A.V. Aho, J.E. Hopcroft, J.D. Ullman. *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.

[3] A. Amir, G. Benson. *Two-dimensional periodicity and its application*, Proc. of 3rd Symposium on Discrete Algorithms, Orlando, FL, Jan. 1992.

[4] A. Amir, G. Benson. *Two-Dimensional periodicity in rectangular arrays*, SIAM J. Comp., Ausgabe 27, Seite 90-106, 1996.

[5] A. Amir, G. Benson, M. Farach. *An Alphabet Independent Approach to Two Dimensional Matching.*, SIAM J. Comput., Ausgabe 23, Seite 313-323, 1994.

[6] A. Amir, G. Benson, M. Farach. *The Truth, the Whole Truth, and Nothing but the Truth: Alphabet Independent Two Dimensional Witness Table Construction*, College of Computing, Georgia Institute of Technology, Atlanta, Georgia, 1992.

[7] A. Amir, M. Farach, *Efficient 2-dimensional Approximate Matching of Non-rectangular Figures*, Proceedings of the second annual ACM-SIAM symposium on Discrete algorithms, Seite 212-223, 1991.

[8] T.J. Baker. *A technique for extending rapid exact-match string matching to arrays of more than one dimension.*, SIAM J. Comp, Ausgabe 7, Seite 533-541, 1978.

[9] R.S. Bird. *Two dimensional pattern matching.*, Information Processing Letters, Ausgabe 6, Seite 168-170, 1977.

[10] M. Crochemore, L. Gąsienic, W. Plandowski, W. Rytter. *Two-Dimensional Pattern Matching in Linear Time and Small Space*, STACS, Seite 181-192, 1995.

[11] M. Crochemore, W. Rytter. *Jewels for stringology : text algorithms*, World Scientific Singapore, 2003.

[12] M. Crochemore, W. Rytter. *Text Algorithms*, Oxford University Press, Seite 285-287, 1994.

[13] K. Fredriksson, G. Navarro, E. Ukkonen. *An Index for Two Dimensional String Matching Allowing Rotations*, IFIP TCS 2000, LNCS 1872, Springer-Verlag Berlin, Heidelberg, Seite 59-75, 2000.

[14] L. Gąsienic, W. Plandowski, W. Rytter. *The zooming method: a recursive approach to time-space efficient string-matching*, Theoret. Comput. Sci., Ausgabe 147, Seite 19-30, 1995.

[15] Z. Galil, Y. Rabani. *Truly alphabet-independent two-dimensional matching*, Proc. 33rd Annual IEEE Symposium on the Foundations of Computer Science, Seite 247-256, 1992.

[16] D. Harel, R.E. Tarjan. *Fast algorithms for finding nearest common ancestor*, Computer and System Science, Ausgabe 13, Seite 338-355, 1984.

[17] D.E. Knuth, J.H. Morris, V.R. Pratt. *Fast Pattern matching in strings*, SIAM J. Comp., Ausgabe 6, Seite 323-350, 1977.

[18] J. Kärkkäinen, E. Ukkonen. *Two- and higher-dimensional pattern matching in optimal expected time*, SIAM J. Comput., Ausgabe 29, Seite 571-589, 1999.

[19] G.M. Landau, U. Vishkin. *Efficient string matching in the presence of errors*, Proc. 26th, IEEE FOCS, Seite 126ff, 1985.

[20] M.G. Main, R.J. Lorentz. *An $O(nlogn)$ algorithm for finding all repetitions in a string*, J. of Algorithms, Seite 422-432, 1984.

[21] G. Navarro, R.B. Yates, *Fast Multi-Dimensional Approximate Pattern Matching*, Proceedings of the 10th Annual Symposium on Combinatorial Pattern Matching, Seite 243-257, 1999.

[22] K. Park. *Analysis of two dimensional approximate pattern matching algorithms*, Theoret. Comput. Sci., Ausgabe 201, Seite 263-273, 1998.

[23] P. Weiner. *Linear pattern matching algorithm*, Proc. 14 IEEE Symposium on Switching and Automata Theory, Seite 1-11, 1973.

[24] A.C. Yao. *The complexity of pattern matching for a random string*, SIAM J. of Computing, Ausgabe 8, Seite 368-387, 1979.

[25] R.B. Yates, G. Navarro, *New Models and Algorithms for Multidimensional Approximate Pattern Matching*, Journal of Discrete Algorithms, Ausgabe 1, Seite 21-49, 2000.