Inside Google - Algorithmen für Suchmaschinen Unscharfe Textsuche

Valeri Schneider
Betreuer: Dirk Nowotka

11. Dezember 2006

Inhaltsverzeichnis

1 Einführung **3**

2 Algorithmen **4**

2.1 Dynamische Programmierung 4
2.2 Bitparallelismus 7
2.3 Automaten 8
2.4 Filteralgorithmen 9

3 Literatur **11**

1 Einführung

Als unscharfe Textsuche (english: „Approximate String Matching“) bezeichnet man die Suche nach einem Muster in einem (langen) Text. Im Gegensatz zu der exakten Suche, bei der nur nach genauen Übereinstimmungen gesucht wird, können hier sowie der Text als auch das gesuchte Muster eine bestimmte Anzahl von Fehlern aufweisen, trotzdem wird die Übereinstimmung als solche erkannt.

Die unscharfe Suche wird seit vielen Jahren erforscht und findet Anwendung in verschiedenen Bereichen wie z.B. Datenbanksuche, Genforschung oder Nachrichtentechnik. Eine der ältesten Anwendungen ist die Korrektur der Rechtschreibfehler in einem geschriebenen Text. Heutzutage ist es sehr wichtig, in Unmengen an der zur Verfügung stehenden Informationen die relevante Information finden zu können. Dies wäre allerdings ohne Algrithmen der unscharfen Suche viel schwieriger, z.B. ein bei der Eingabe eines Suchbegriffes in einer Suchmaschine entstandener Tippfehler würde zu keinem Ergebnis führen, oder eine Information, die in einer Datenbank liegt und Rechtschreibfehler enthält könnte nicht gefunden werden.

Formale Definitionen

Das Problem der unscharfen Suche wird folgendermaßen formal definiert:
Gegeben seien ein Text $T \in \Sigma^*$, ein Muster $M \in \Sigma^*$ (wobei Σ ein beliebieges endliches Alphabet ist), eine maximale Anzahl erlaubter Fehler $k \in R$ und eine Distanzfunktion[1] $d : \Sigma^* \times \Sigma^* \to R$. Gesucht sind alle Textpositionen j zu denen ein i existiert, so dass $d(M, T_{i...j}) \leq k$.

Distanzfunktionen unterscheiden sich darin, welche Operationen bei der Transformation erlaubt sind. Einige besonders wichtige der vielen existierenden Distanzfinktionen werden hier vorgestellt.

Hammingdistanz[1]

Die Hammingdistanz beschreibt die Anzahl der Stellen, an denen sich 2 Strings gleicher Länge unterscheiden. Bei ungleicher Länge ist die Hammingdistanz ∞. Einzige erlaubte Operation ist das Ersetzen von Zeichen (Kosten: 1). Die Hammingdistanz ist besonders wichtig, wenn man fehlererkennendes oder fehlerkorrigierendes Code erstellen möchte.

[1] Eine Distanzfunktion $d(x, y)$ beschreibt die Kosten einer Reihe der Operationen, die notwendig sind um x in y zu verwandeln.

Beispiele: $d(10011, 00010) = 2$, $d(10101, 1001) = \infty$

Episode-Distanz[2]
Die Episode-Distanz erlaubt nur das Einfügen eines Zeichen (Kosten: 1) und ist somit als $|y| - |x|$ definiert, falls die Transformation möglich ist, sonst ∞. Die Episode-Distanz ist nicht symmetrisch, somit kann es vorkommen, dass $d(x, y) \neq d(y, x)$ ist.
Beispiel: $d(orion, operation) = 4$ aber $d(operation, orion) = \infty$

Editdistanz[3]
Die Editdistanz (auch Levenshtein-Distanz genannt) erlaubt das Einfügen, Löschen und Ersetzen von Zeichen und ist symmetrisch. Weiterhin vorgestellte Algorithmen basieren auf der Editdistanz, falls nichts anderes explizit erwähnt ist. Betrachtet man die 3 Operationen als gleich „teuer“ (Kosten: 1), so spricht man von ungewichteter Editdistanz, haben dagegen die Operationen verschiedene Kosten, so spricht man von gewichteter Editdistanz.

Längster gemeinsamer Teilstring[4]
Die LCS-Distanz (english: „longest common substring“) $d(x, y)$ beschreibt wie der Name schon sagt, die Länge des längsten String n, der ein Teilstring von x und y ist. Erlaubt sind nur das Einfügen und das Löschen von Zeichen (Kosten: 1).
Beispiel: $d(aabbc, ccabba) = 3$

2 Algorithmen

2.1 Dynamische Programmierung

Ansatz der dynamischen Programmierung besteht darin, dass die komplexe Lösung eines Problems aus zuvor bereits berechneten optimalen Teillösungen berechnet wird. Die Ergebnisse der Teillösungen werden normaleweise in einer Tabelle abgelegt, somit werden die vielen rekusiven Aufrufe durch das Ablesen der entprechenden Werte aus der Tabelle ersetzt.

Der erste Algorithmus

Der erste Algorithmus [5], der die Editdistanz zweier Zeichenketten berechnet, ist nicht besonders effizient, ist aber einfach zu verstehen und zu imple-

mentieren. Außerdem lässt er sich relativ leicht an andere Distanzfunktionen anpassen [6]. Der Algorithmus berechnet also die Editdistanz zweier Zeichenketten x und y ($edit(x, y)$), indem er die Matrix A mit $|x|+1$ Zeilen $(0 \ldots |x|)$ und $|y| + 1$ Spalten $(0 \ldots |y|)$ mit Werten auffüllt. Dabei bedeutet der Eintrag $A_{i,j}$ die Editdistanz der Zeichenketten $x_{1 \ldots i}$ und $y_{1 \ldots j}$, also die minimale Anzahl der Editoperationen um die Zeichenkette $x_{1 \ldots i}$ in $y_{1 \ldots j}$ zu verwandeln oder umgekehrt. Somit ist der Eintrag $A_{|x|,|y|}$ die gesuchte Editdistanz der Zeichenketten x und y. Die Einträge der Matrix werden folgendermaßen berechnet:

$$A_{i,0} = i$$

$$A_{0,j} = j$$

Diese Regeln repräsentieren die Editdistanz zwischen einer Zeichenkette der Länge i (bzw. j) und einer leeren Zeichenkette. Für zwei nicht leere Zeichenketten $x_{1 \ldots i}$, $y_{1 \ldots j}$ betrachte man die letzten Zeichen x_i und y_j. Sind sie gleich, so sind keine Editoperationen notwendig und die Distanz zwischen $x_{1 \ldots i}$ und $y_{1 \ldots j}$ ist gleich der Distanz zwischen $x_{1 \ldots i-1}$ und $y_{1 \ldots j-1}$. Sind x_i und y_j dagegen verschieden, müssen alle 3 Editoperationen (löschen, einfügen, ersetzen) in Betracht gezogen werden. Die Operation, bei deren Ausführung die Editdistanz minimal ist, wird ausgeführt. Da alle 3 Operationen gleich „teuer“ sind, wird eine 1 zu der bereits berechneten minimalen Distanz addiert.

$$A_{i,j} = \begin{cases} \mathrm{A}_{i-1,j-1}, \text{ falls } x_i = y_j \\ 1 + min(\mathrm{A}_{i-1,j}, A_{i-1,j-1}, A_{i,j-1}), \text{ sonst} \end{cases}$$

Beachte, die Distanzen zwischen allen kürzeren Zeichenketten müssen bereits berechnet sein.

Beispiel

Berechne den Eintrag $A_{3,3}$ in der Matrix, die **edit**(*zelt*,*zeit*) berechnet.

		z	e	i	t
	0	1	2	3	4
z	1	0	1	2	3
e	2	1	0	1	2
l	3	2	1	?	-
t	4	-	-	-	-

Abbildung 1: unvollständige Matrix für **edit**(*zelt*,*zeit*)

$x_3 \neq y_3$, also man hat 3 folgende Möglichkeiten:

1. $\underbrace{\text{ersetze } x_3 \text{ durch } y_3}_{=1}$ (l durch i) + $\underbrace{\textbf{edit}(ze, ze)}_{=A_{2,2} \quad =0} = 1$

2. $\underbrace{\text{lösche } x_3}_{=1}$ (l) + $\underbrace{\textbf{edit}(ze, zei)}_{=A_{2,3} \quad =1} = 2$

3. $\underbrace{\textbf{edit}(zel, ze)}_{=A_{3,2} \quad =1}$ + $\underbrace{\text{füge } y_3 \text{ (i) ein}}_{=1} = 2$

Aus diesen Werten wird das Minimum bestimmt und in die Matrix eingetragen. Die vollständige Matrix A für **edit**(*zelt*,*zeit*):

		z	e	i	t
	0	1	2	3	4
z	1	0	1	2	3
e	2	1	0	1	2
l	3	2	1	1	1
t	4	3	2	2	**1**

Abbildung 2: Vollständig berechnete Matrix für **edit**(*zelt*,*zeit*)

Die Berechnung der Editdistanz mit diesem Algorithmus benötigt die Berechnung aller Einträge der Matrix A, daher ist die Laufzeitkomplexität O(|x|·|y|). Die Platzkomplexität ist O(min(|x|,|y|)), denn es muss nur die zuletzt berechnete Spalte (bzw. Zeile) bekannt sein, damit die nächste Spalte (bzw. Zeile) berechnet werden kann. Möchte man allerdings die genaue Folge von Operationen berechnen, muss die ganze Matrix gespeichert werden.

Anpassung an das Problem der unscharfen Textsuche

1980 wurde der gerade vorgestellte Algorithmus zur Berechnung der Editdistanz in ein Algorithmus [7] konvertiert, der das Problem der Textsuche mit k Fehlern löst. Die einzige Änderung ist die Initialisierung der ersten Zeile der Matrix mit $A_{0,j} = 0$ anstatt mit $A_{0,j} = j$. Dies bedeutet, dass jetzt jede Position des Textes als Anfangsposition des Musters zugelassen ist und der Matrixeintrag $A_{i,j}$ die minimale Editdistanz zwischen dem Musterteil $x_{1...i}$ und einem Suffix des Textes $y_{1...j}$ (also jedem Unterstring des Textes, der an der Position j endet) enthält.

		z	e	i	t
	0	0	0	0	0
z	1	0	1	1	1
e	2	1	0	1	2
l	3	2	1	1	2
t	**4**	**3**	**2**	**2**	**1**

Abbildung 3: Matrix der unscharfen Suche des Musters *zelt* im Text *zeit*

Die letzte Zeile der Matrix enthält somit zu jeder Position im Text die minimale Editdistanz des Musters und so können die Positionen ensprechend der Anzahl von erlaubten Fehlern k aus der letzen Zeile abgelesen werden. In der Abbildung 3 für $k = 1$ ist $j = 4$, für $k = 2$ ist $j = 2$ und $j = 3$.

2.2 Bitparallelismus

Wie bekannt, arbeiten Computer auf Bits innerhalb eines Computerwortes parallel, das heißt die Bitoperationen (z.B. bitweises Und, bitweises Oder, Komplement) werden parallel auf 32 Bits (bei der 32-bit-Architektur) ausgeführt. Die Grundidee dabei ist einen existierenden Algorithmus zu nehmen und ihn so anzupassen, dass seine Laufzeit vom Bitparallelismus profitiert. Gelingt dies, so ist eine Laufzeitverbesserung um einen Faktor w (die Länge des Wortes, also 32 bei der 32-bit-Architektur) möglich.

Parallelisierung der Matrix der dynamischen Programmierung

Der erste Algorithmus, der das Bitparallelismus bei der Berechnung der Matrix der dynamischen Programmierung ausnutzte wurde 1994 von Wright in seiner Arbeit [11] präsentiert. Die Matrix der dynamischen Programmierung kann nicht nur wie bereits erwähnt zeilenweise oder spaltenweise berechnet werden, sondern auch diagonal[2]. Der Algorithmus basiert auf zwei Beobachtungen:

[2]gemeint sind die die sekundären Diagonalen die von rechts oben nach links unten verlaufen

Für jeden Eintrag $A_{i,j}$ gilt:

- $A_{i,j} = A_{i-1,j-1}$ oder $A_{i,j} = A_{i-1,j-1} + 1$
- $A_{i,j}$ unterscheidet sich maximal um 1 von $A_{i-1,j}$ und $A_{i,j-1}$

Daraus kann man die neue Formel für die Berechnung der Matrixeinträge ableiten:

$$A_{i,j} = \begin{cases} A_{i-1,j-1}, & \text{falls} \begin{cases} x_i = y_j \\ \text{oder } A_{i-1,j} = A_{i-1,j-1} - 1 \\ \text{oder } A_{i,j-1} = A_{i-1,j-1} - 1 \end{cases} \\ A_{i-1,j-1} + 1, & \text{sonst} \end{cases}$$

Wie man sieht, lassen sich die sekundären Diagonalen aus zwei Vorgängerdiagonalen berechnen.
Das parallele Rechnen auf Bits nutzt der Algorithmus aus, indem mehrere Zeichen vom Text und vom Muster binär kodiert und in ein Computerwort gepackt und somit mehrere Eintrage der neuen sekundären Diagonale parallel berechnet werden. Der entscheidende Faktor für die Laufzeitverbesserung ist die Größe des verwendeten Alphabets $\sigma = |\Sigma|$, denn je größer σ ist, desto mehr Bits werden benötigt um ein Zeichen des Alphabets zu kodieren und um so weniger Zeichen können in ein Computerwort gepackt werden. Daher ist die Komplexität von diesem Algorithmus $O(|x| \cdot |y| \cdot \log(\sigma)/w)$.

2.3 Automaten

Algorithmen, die auf Automaten basieren, sind seit längerer Zeit bekannt und sind besonders interessant, da mit einem NFA das Problem der unscharfen Suche in liniarer Zeit gelöst werden kann. Allerdings muss man auch sagen, dass diese Interesse meist theoretischer Natur ist, da in der Praxis die Zeit- und Platzkomplexität von der Anzahl der erlaubten Fehlern k und der Länge des gesuchten Musters exponentiell abhängig ist.

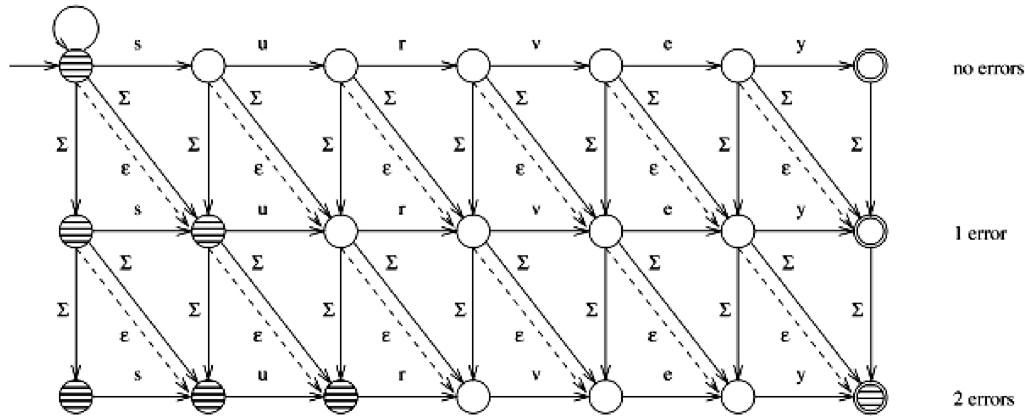

Abbildung 4: Ein NFA für die unscharfe Suche mit 2 Fehlern[3]

Abbildung 4 zeigt einen nicht deterministischen endlichen Automaten mit zwei akzeptierten Fehlern. Jede Reihe zeigt an wie viele Zeichen bereits gelesen wurden, jede Zeile entspricht der Anzahl der Fehler, die bereits zugelassen wurden. Horizontale Pfeile bedeuten die Übereinstimmung der Zeichen, alle anderen Pfeile erhöhen die Editdistanz und entsprechen den Editoperationen. Die senkrechten Pfeile repräsentieren das Einfügen eines Zeichen, die vollen Diagonallinien - das Ersetzen von Zeichen und die diagonalen Strichlinien entsprechen der Operation Löschen.

Befindet sich nun der NFA in einem der rechten Endzustände, so wurde der Muster mit maximal k Fehlern im Text gefunden. Der NFA lässt sich relativ leicht an andere Distanzfunktionen anpassen, indem man z.B. die unerlaubten Operationen (entsprechende Kanten) aus dem Automat entfernt.

2.4 Filteralgorithmen

Die Filteralgorithmen wie der Name schon sagt wenden zuerst einen Filter auf den Text an mit dem Ziel, möglichst viele Textpositionen j als nicht geeignet zu klassifizieren und somit von der Suche auszuschließen. Anschießend muss noch ein Verifizierungsalgorithmus eingesetzt werden, der die gebliebenen Textpositionen überprüft, ob eine unscharfe Übereinstimmung mit dem Muster vorliegt. Die Laufzeit solcher Algorithmen hängt natürlich stark davon ab, viele Textpositionen verworfen werden konnten sowie von der Laufzeit

[3]entnommen aus [6]

des verwendeten Verifizierungsalgorithmus . In der Praxis haben sich die Filteralgorithmen aber als die schnellsten erwiesen [6].

1992 haben wurde von Wu und Manber [9] ein sehr einfacher Filter vorgestellt. Der Algorithmus basiert auf der folgenden Überlegung: Zerlegt man ein Muster in $k + 1$ Teile, so muss sich mindestens ein Teil unverändert im approximierten Vorkommen dieses Musters befinden, denn k Fehler können keine $k + 1$ Teile verändern. So wird das gesuchte Muster zuerst in $k + 1$ Teile geteilt und nach diesen Submustern wird im Text gesucht. Wird eine Übereinstimmung eines Submustern mit dem Text gefunden, so muss jetzt die Umgebung des gefundenen Submusters überprüft werden, also insgesamt ein Textteil der Länge $x + 2k$, wobei x die Länge des Suchmusters ist. Für die Verifikation wurde der Shift-Or-Algorithmus [10] eingesetzt, oft wird aber an dieser Stelle der einfache Algorithmus der dynamischen Programmierung verwendet.

Schlussbemerkung

Das Ziel dieser Ausarbeitung war einen Überblick über die verschiedene Ansätze zum Lösen des Problems der unscharfen Textsuche zu geben. Es existiert allerdings eine Vielzahl von Algorithmen, die in dieser Ausarbeitung nicht behandelt wurden, können aber zum größten Teil in [6] nachgelesen werden.

3 Literatur

[1] Richard W. Hamming: Error-detecting and error-correcting codes. Bell System Technical Journal 29(2): S. 147-160, 1950

[2] Das, G., Fleisher, R., Gasieniek, L., Gunopulos, D. und Kärkäinen, J. 1997. Episode matching. Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching (CPM '97). LNCD, vol. 1264, Springer-Verlag, Berlin, 12-27

[3] Levenshtein, V. 1965. Binary codes capable of correcting spurious insertions and deletions of ones. Prof. Inf. Transmissions 1, 8-17

[4] Needleman, S. und Wunsch, C. 1970. A general method applicable to the search for similarities in the amino acid sequences of two proteins. J. Mol. Biol. 48, 444-453

[5] T. Vintsyuk Speech Discriminationby Dynamic Programming, Cybernetics 4, S. 52-58, 1968

[6] Navarro, G. 2001. A Guided Tour to Approximate String Matching. ACM Computing Surveys 33, 1, 31-88

[7] Sellers, P. 1980. The theory and computation of evolutionary distances: pattern recognition. J. Algor. 1, 359-373

[8] Gene Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. Journal of the ACM, 46(3):395415, 1999.

[9] Wu, S. und Manber, U. 1992b. Fast text searching allowing errors. Commun. ACM 35,10,83-91

[10] Baeza-Yates, R und Gonnet, G. 1992 A new approach to text searching. Commun ACM 35,10, 74-82

[11] Wright, A. 1994. Approximate string matching using within-word parallelism. Software Practice Exper. 24,4,337-362.