

Programmierübungen

Wintersemester 2006/2007

2. Übungsblatt

2. Mai 2007

Abgabe bis Montag, 14. Mai.

In den Vortragsfolien der Programmierübungen oder im Skript zur Einführung in die Informatik abgedruckte Quelltexte können verwendet werden, müssen aber der Programmierrichtlinie entsprechend formatiert und kommentiert werden.

Bitte bearbeiten Sie die Übungsaufgaben in Kleingruppen mit bis zu 3 Teilnehmern. Alle Teilnehmer einer Kleingruppe müssen in der selben Übungsgruppe eingetragen sein. Ihre Abgabe muss eine Datei `contributions.txt` enthalten, in der aufgelistet ist welche Anteile der Bearbeitung von jedem einzelnen Teilnehmer erstellt wurden. Jeder Teilnehmer muss die Funktionsweise aller Teile der Bearbeitung erläutern können. Beachten Sie die Programmierrichtlinie und kommentieren Sie Ihren Quelltext. Dokumentieren Sie unbedingt Ihre Lösungsidee in den Quelltext-Kommentaren.

<http://www.iste.uni-stuttgart.de/ps/Lehre/SS2007/inf-prokurs>

Aufgabe 2.1: Text laden

(4 Punkte)

Auf diesem Aufgabenblatt sollen mehrere Ansätze zur Lösung des folgenden Problems verglichen werden:

In einem längeren Text T wird nach allen Positionen gesucht, an denen ein kürzerer Suchbegriff S in dem Text vorkommt.

Diese Aufgabe soll Grundfunktionalität für die nachfolgenden Aufgaben bereit stellen.

Erstellen Sie ein Paket, mit dessen Hilfe der Text aus einer Datei komplett in den Hauptspeicher geladen werden kann. Der gesamte Text soll in einem String-Objekt bereitgestellt werden. Verwenden Sie einen Zeiger-Typ `type String_Access is access String;`, um das String-Objekt auf der Halde zu allozieren.

Falls Ihr Programm die Textdatei zeilenweise liest (z. B. mit `Ada.Text_IO.Get_Line`), so achten Sie darauf, dass die einzelnen Zeilen in dem String-Objekt durch ein besonderes Zeichen (`Ada.Characters.Latin_1.CR`) getrennt werden.

Achten Sie auf einen effizienten Ladevorgang. Die häufige Verwendung der Konkatenation & kann die Laufzeit nachteilig beeinflussen. Gehen Sie davon aus, dass beliebig lange Texte geladen werden sollen (jedoch nicht mehr als Positive'Last Zeichen inkl. Zeilenumbrüche).

Aufgabe 2.2: Naive Teilstringsuche

(2 Punkte)

Mit Hilfe des Pakets aus Aufgabe 1 soll ein Text geladen werden. In diesem Text soll nun nach einem bestimmten Muster gesucht werden. Als Muster zählt jeder String, der höchstens so lang ist wie der geladene Text. Gesucht werden nun alle Index-Positionen im Text, an denen das Muster vorkommt. Implementieren Sie den naiven Algorithmus,

der jede mögliche Indexposition von T der Reihe nach überprüft. Geben Sie ein kurzes Testprogramm ab, mit dem die korrekte Funktionsweise Ihrer Implementierung getestet werden kann.

Index:	123456789	123456789	123456789	123456789
Text:	abcdbcbcd	abcdbcbcd	abcdbcbcd	abcdbcbcd
Muster:	bcd	bcd	bcd	bcd
Treffer:	{}	{2}	{2}	{2}
Index:	123456789	123456789	123456789	
Text:	abcdbcbcd	abcdbcbcd	abcdbcbcd	
Muster:	bcd	bcd	bcd	
Treffer:	{2}	{2}	{2, 7}	

Aufgabe 2.3: Endlicher Automat

(5 Punkte)

Es fällt schnell auf, dass das naive Verfahren aus Aufgabe 2 etliche unnötige Vergleiche zwischen Muster und Text durchführt. Eine Verbesserung des Verfahrens ist der Algorithmus von Knuth, Morris und Pratt (1977). Die Funktionsweise dieses Verfahrens lässt sich durch Generieren und Simulieren eines endlichen Automaten beschreiben:

- Aus einem Muster mit m Zeichen Länge wird ein endlicher Automat generiert mit den Zuständen Z_1 bis Z_{m+1} .
- Jeder Zustand Z_i repräsentiert den Vergleich des Zeichens aus dem Muster mit Index i . Z_1 ist der Start- und Z_{m+1} der einzige Endzustand.
- Es wird der Text T zeichenweise gelesen und als Eingabe für die Simulation des endlichen Automaten verwendet. Immer wenn der Endzustand erreicht wurde, stimmte das Muster mit dem entsprechenden Teilstring des Texts überein.
- Die Transitionen des Automaten werden so gewählt, dass der Vergleich von Muster und Text simuliert wird. Ist der Automat im Zustand Z_i mit $1 \leq i \leq m$, so bedeutet dies, dass $i - 1$ gelesene Zeichen des Texts mit dem Muster übereingestimmt haben. Wird als nächstes das i -te Zeichen des Musters gelesen, so wird in den Zustand Z_{i+1} gewechselt. Wird ein anderes Zeichen gelesen, so wird in den Zustand Z_k übergegangen, mit dem größten k , $1 \leq k \leq i$, so dass die k ersten Zeichen des Musters mit den letzten Eingabezeichen übereinstimmen. Siehe ein Beispiel für das Muster „abac“ in Abbildung 1.

Implementieren Sie die Mustersuche mit endlichen Automaten. Geben Sie ein kurzes Testprogramm ab, mit dem die korrekte Funktionsweise Ihrer Implementierung getestet werden kann.

Aufgabe 2.4: Suffix-Baum

(6 Punkte)

Soll in dem selben Text wiederholt nach verschiedenen Mustern gesucht werden, so kann es sich lohnen für den Text einen Suffix-Baum zu erstellen. Voraussetzung für die Existenz eines Suffix-Baums ist, dass das letzte Zeichen des Texts an keiner anderen Stelle im Text vorkommt (in der Praxis kann einfach ein besonderes Zeichen an das

Ende des Texts angehängt werden). In der Literatur bezeichnet man dieses besondere Zeichen meist mit \$; für die Praxis eignet sich jedoch `Ada.Characters.Latin_1.EOT` deutlich besser, da \$ auch in gewöhnlichen Texten vorkommt.

Ein Suffix-Baum für einen Text $T = t_1 t_2 \dots t_n$ (der Länge n) ist ein Baum, der genau n Blätter besitzt. Diese Blätter sind mit den Zahlen $1 - n$ beschriftet. Eine Kante des Baums ist mit je einem Buchstaben t_i des Texts beschriftet und von keinem Knoten des Baums gehen zwei Kanten aus, die mit dem gleichen Buchstaben beschriftet sind. Die inneren Knoten und die Kanten des Suffix-Baums sind so verteilt, dass zu jedem Blatt i (genau) ein Pfad existiert, dessen Kantenbeschriftungen aneinandergehängt das Suffix $t_i t_{i+1} \dots t_n$ ergeben. Ein Beispiel für den Text „abac“ ist in Abbildung 2 gegeben.

Ein Suffix-Baum kann sehr einfach erstellt werden, indem mit nur dem Wurzelknoten begonnen wird und der Reihe nach alle Suffixe des Texts in den Baum eingefügt und dabei Kanten und Knoten nach Bedarf erstellt werden.

Zur Mustersuche im Suffix-Baum wird auf dem Wurzelknoten gestartet. Dann wird das Muster Zeichen für Zeichen gelesen und jeweils die mit dem aktuellen Zeichen beschriftete Kante überquert. Nachdem das Muster vollständig gelesen wurde ist ein bestimmter (innerer) Knoten erreicht. Jedes Blatt, das von diesem Knoten aus erreichbar ist, ist mit der Index-Position eines Vorkommens des Musters beschriftet.

Implementieren Sie die Mustersuche mit Suffix-Bäumen. Geben Sie ein kurzes Testprogramm ab, mit dem die korrekte Funktionsweise Ihrer Implementierung getestet werden kann.

Aufgabe 2.5: Zeitmessung

(3 Punkte)

Messen Sie, wieviel Zeit die einzelnen Verfahren zum Auffinden von Teilwörtern benötigen. Verwenden Sie mehrere ausreichend große Texte und Suchmuster. Ermitteln Sie folgende Daten:

- Ermitteln Sie die benötigte Rechenzeit zum Laden des Texts getrennt von der Zeit zur Suche nach Mustern.
- Ermitteln Sie die Zeit zum Aufbau eines Suffix-Baum getrennt von der Suche nach Mustern (ein Suffix-Baum kann für viele Suchen in dem selben Text verwendet werden).
- Erzeugen Sie (möglicherweise sinnlose) Texte und Muster, die zur Worst-Case Rechenzeit in den einzelnen Verfahren führen.
- Testen Sie die Verfahren auf realen Texten.

Die Zeitmessung kann z. B. mit dem Paket `Ada.Calendar` erfolgen. Beachten Sie, dass „gleichzeitig“ ablaufende Programme Ihre Zeitmessung beeinflussen können. Interpretieren Sie Ihre Messergebnisse. Haben Sie diese Zahlen erwartet? Geben Sie eine Textdatei mit den gemessenen Zahlen, den Eingabedaten und einer kurzen Interpretation ab.

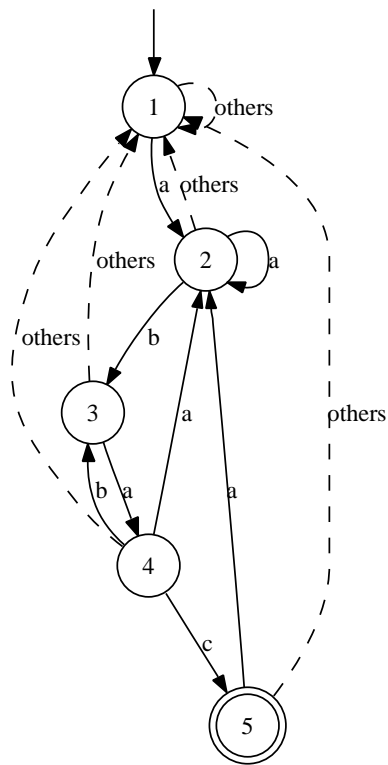


Abbildung 1: Endlicher Automat für Muster „abac“

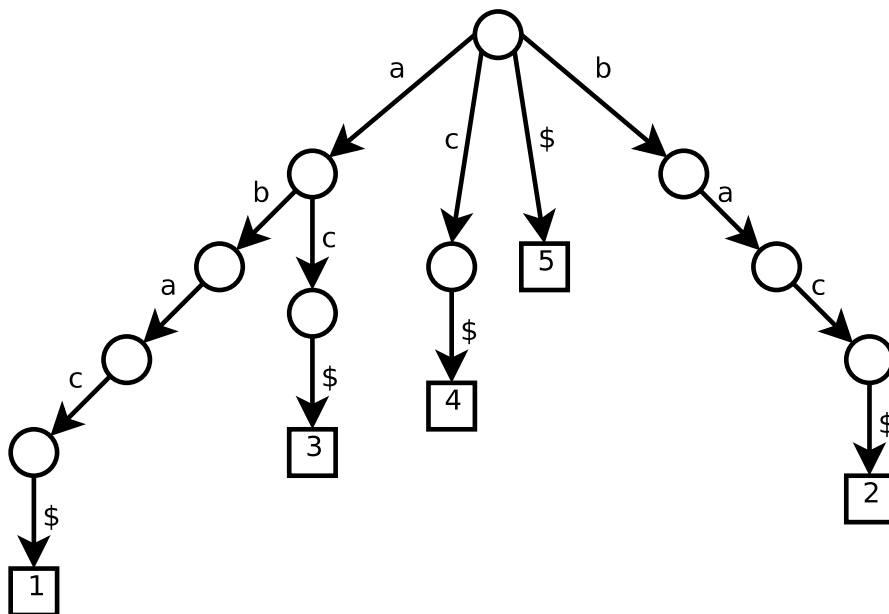


Abbildung 2: Suffix-Baum für Text „abac“