Musterlösung
Übungszettel 2
Algorithmen & Datenstrukturen für Bioinformatiker

Adrian Haß

4. Dezember 2002

1 Korrektheit Aho-Corasick

Beim Aho-Corasick-Algorithmus wird ein Patternbaum gegen einen Text gematcht. Bei Mismatches wird der Baum um die Länge des längsten Suffix des aktuellen Patterns, der Präfix eines anderen oder desselben Patterns ist, verschoben und am Ende dieses Präfix mit den Vergleichen fortgefahren. Dabei benutzen wir sogenannte Fehlerlinks, die den letzten matchenden Knoten mit dem Endknoten des Präfix verbinden. Wir wollen für die Korrektheit vom Aho-Corasick also folgendes zeigen:

> Gegeben ein Alignment von Patternbaum $\mathcal{P}$ am Text T mit Mismatch bei einer Kante ausgehend von Knoten v. Dann wird bei einem Shift um $lp(v)$ und Fortsetzung der Match-Prozedur an n_v kein Vorkommen eines Patterns übersprungen.

Im Beweis führen wir folgende Annahme zum Widerspruch:

> Ein Mismatch taucht im Pattern P_v des Baumes $\mathcal{P}$ bei Knoten v auf. Der Fehlerlink n_v von v zeigt auf Knoten w im Pattern P_w, dessen Präfix längster Suffix von P_v ist. Beim Verschieben des Baumes um $lp(v)$ wird jedoch das Auftreten eines Patterns P_0 übersprungen.

Zum Beweis machen wir uns zur Verdeutlichung eine Zeichnung ($\rightsquigarrow$ Abb.1). Nennen wir den Präfix/Suffix von P_w bzw. P_v zunächst β, dann ist klar, dass dieser im Text vor der Stelle des Mismatch $T[k]$ vollständig enthalten ist. Da also das Vorkommen von P_0 vor P_w liegt, bedeutet das, dass insbesondere β in P_0 enthalten sein muss, sonst würde P_0 ja nicht matchen.

Nun liegt aber auch im Bereich der ersten Stelle von P_0 bis hin zum Beginn von β ein Teilstring α, der auch im Text direkt vor β vorkommt. Da auch P_v hier den Text gematcht hat ist α auch dort vor β enthalten. Somit ist $\alpha\beta$ ein Suffix von P_v, der den Präfix $\alpha\beta$ von P_0 matcht. Da wir angenommen hatten,

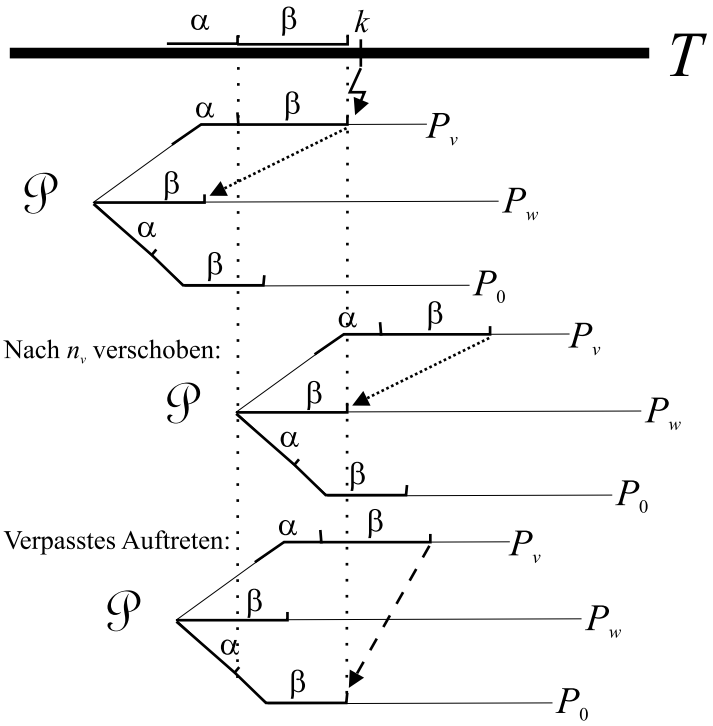

Abbildung 1: Skizze zum Beweis von Aho-Corasic

dass P_0 echt vor P_x beginnt, gilt $|\alpha\beta| > |\beta|$. Somit hätte der Fehlerlink von v auf den Endknoten von $\alpha\beta$ in P_0 zeigen müssen. Dies ist aber ein Widerspruch zur Annahme.

2 Vorbereitung der Implementierung von AC

Ein paar Tips über die man vielleicht nachdenken sollte:

- Man sollte folgende separate Klassen anlegen
  (1) PatternTree
  (2) PTNode
  (3) StringMatcher
- Um zu realisieren, dass Buchstaben in Kanten gespeichert werden, kann eine `Hashtable` verwandt werden. Hier wird unter `Key Character` einfach eine Referenz zum Node gespeichert.
- Man könnte die Baumerzeugung in den Konstruktor schreiben. Dabei sollte man aber ein `String[]` als Patternliste übergeben und die I/O-Prozeduren in die Kernklasse verlegen.

- Man sollte nicht mit Indices arbeiten sondern lieber echte Referenzen halten. Dann braucht man auch keine $lp(v)$-Werte, sondern nur die links n_v. Also muss man den Baum nicht verschieben, sondern die aktuellen Pointer (Zeiger/Referenzen) wandern in Text und Baum.
- Alles an was man von außen ran soll wird `public`, alles auf das andere Klassen zugreifen müssen bekommt `package` (also kein Modifier), sonstige Variablen/Methoden, die nur in der Klasse aufgerufen werden, sind `private`. Natürlich muss dann auch ein extra Package angelegt werden und die Dateien in einem gleichnamigen Verzeichnis gespeichert werden.

3 Endliche Automaten und Reguläre Sprachen

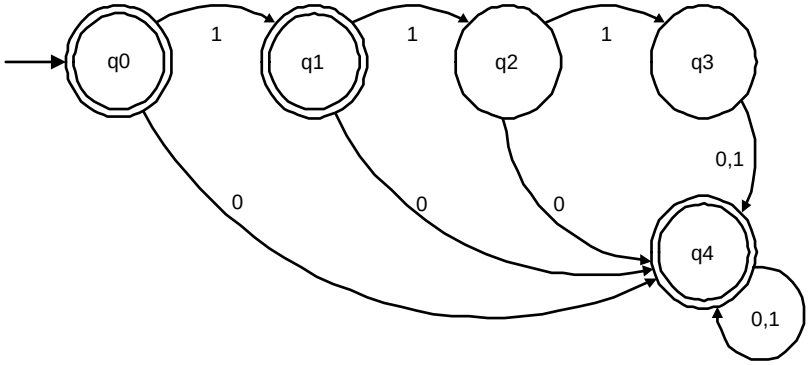

Abbildung 2: Zustandsdiagramm (dfa) Sprache $R_1 = 1^*0(0|1)^* \cup 1111(1)^*$ (verschieden von 11 und 111)

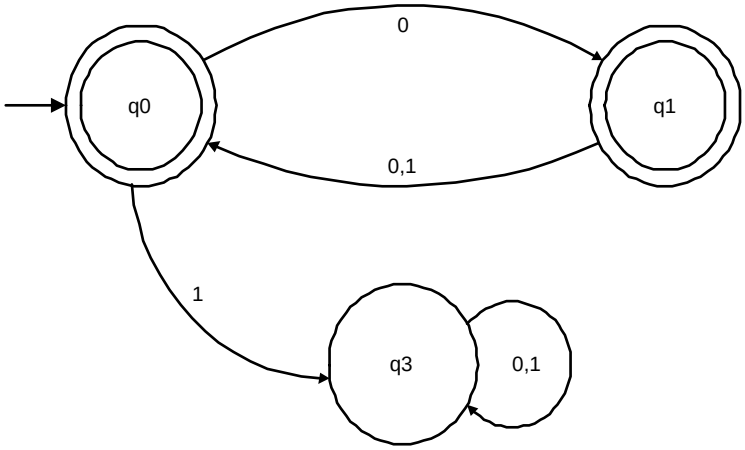

Abbildung 3: Zustandsdiagramm (dfa) Sprache $R_2 = (0(0|1))^*(0|\epsilon) \cup 0$ (ungrade Stellen 0)

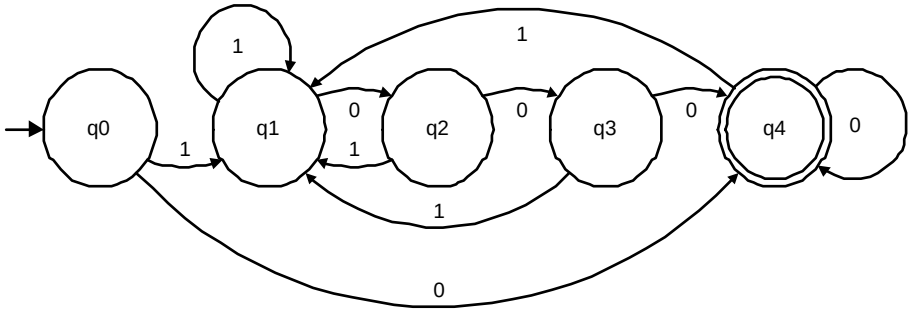

Abbildung 4: Zustandsdiagramm (dfa) Sprache $R_3 = (0|1)^*000 \cup 0 \cup 00$ (durch 8 teilbar)