Theorie der Informatik

G. Röger
Frühjahrssemester 2020

Universität Basel
Fachbereich Informatik

Übungsblatt 8 — Lösungen

Aufgabe 8.1 (Automaten; 2 Punkte)

Schreiben Sie eine Zusammenfassung über die verschiedenen Typen von Automaten, die in Kapitel C behandelt wurden. Sie müssen nicht erklären, was DFA/NFA/PDA/DTM/NTM sind, oder wie sie definiert sind. Ihre Zusammenfassung sollte den Fokus stattdessen auf die Unterschiede dieser Automaten legen und darauf, wie diese Unterschiede die Komplexität der Sprachen beeinflussen, die mit ihnen akzeptiert werden können. Auf welche Art können die Automaten beispielsweise informationen speichern während sie die Eingabe lesen, und wie viel Information ist nötig um verschiedene Sprachen zu akzeptieren? *Eine gute Antwort kann in ca. einer halben Seite in LaTeXgeschrieben werden.*

Lösung:

Die folgende Antwort ist eine mögliche Antwort aber da die Frage offen ist, sind auch andere Antworten möglich.

DFAs, NFAs und PDAs können das Eingabewort nur einmal von links nach rechts lesen. Das heisst, dass sich der Automat Informationen über das Wort „merken“ muss. Zum Beispiel muss sich ein Automat für die Sprache $\{\mathtt{a}^n\mathtt{b}^n \mid n \geq 0\}$ die Anzahl an `a`s merken um sie später mit der Anzahl von `b`s zu vergleichen. DFAs und NFAs können Informationen nur in ihrem aktuellen Zustand speichern. Da die Anzahl der Zustände endlich ist, bedeutet das, dass sie nur eine endliche Menge an Daten speichern können. Reguläre Sprachen, wie zum Beispiel die Sprache aller Wörter, die mit `b` aufhören, können von einem DFA erkannt werden, weil für sie nur eine endliche Menge an Informationen gespeichert werden muss (in diesem Fall, ob das letzte gelesene Symbol ein `b` war). Das Pumping-Lemma baut auf dieser Beobachtung auf: wenn eine Sprache regulär ist und von einem DFA/NFA akzeptiert werden kann, dann muss es wiederholbare Teile geben, da der DFA/NFA nur eine endliche Menge von Daten speichern kann.
PDAs können zusätzliche Informationen auf einem Stack speichern. Verglichen mit DFAs/NFAs macht sie das mächtiger, weil der Stack keine Obergrenze für die Anzahl von Elementen hat. Kontextfreie Sprachen, wie $\{\mathtt{a}^n\mathtt{b}^n \mid n \geq 0\}$ können von PDAs erkannt werden: während die `a`s gelesen werden, kann der PDA sie „zählen“, indem er Elemente auf den Stack hinzufügt. Wenn die `b`s gelesen werden, kann er dann ihre Zahl mit der Anzahl der Elemente auf dem Stack vergleichen, indem er für jedes gelesene `b` ein Symbol vom Stack entfernt. Allerdings können PDAs auf ihren Stack nur von oben zugreifen und sie konsumieren die Einträge, die sie davon lesen. Zum Beispiel ist es nicht möglich unten auf den Stack zu schauen ohne vorher alle anderen Elemente zu entfernen. Das bedeutet, dass nicht kontextfreie Sprachen, wie $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$ nicht von PDAs erkannt werden können.
Turing-Maschinen sind noch mächtiger als PDAs, weil sie beliebig viel Informationen auf ihrem Band speichern können und jeden Teil dieser Informationen zu jeder Zeit lesen können, ohne ihn zu konsumieren. Für jede Sprache, die von einer Grammatik erzeugt werden kann, gibt es eine Turing-Maschine, die sie akzeptiert, und es ist die Möglichkeit beliebig viel Informationen zu speichern und jederzeit zu lesen, die das möglich macht. Später im Kurs werden wir sehen, dass Turing-Maschinen trotz dieser Fähigkeiten ihre Grenzen haben.

Aufgabe 8.2 (Deterministische Turing-Maschinen; 2+2+2+2 Punkte)

Im Rahmen dieser Aufgabe soll ein Simulator für deterministische Turing-Maschinen implementiert werden.
Ein Gerüst dafür finden Sie auf der Vorlesungsseite. Implementieren Sie die vorgegebenen Methoden und wählen Sie geeignete private Felder für die Klassen. Implementieren Sie nur die durch

`// TODO` gekennzeichneten Stellen; fügen Sie keine weiteren Methoden, Klassen oder öffentliche Felder hinzu.

(a) Implementieren Sie die folgenden Methoden der Klasse `Tape`, die das beidseitig unendliche Band einer Turing-Maschine (inklusiv der Position des Schreib-/Lesekopfes) repräsentieren soll:

Der Konstruktor `Tape(word, blank)` bekommt die initiale Bandbelegung und das zu verwendende Blank-Symbol übergeben. Der Schreib-/Lesekopf soll sich initial auf dem ersten Zeichen der Eingabe befinden.

Die Methode `read()` soll das Zeichen unter dem Kopf zurückgeben.

Die Methode `write(symbol)` soll das Zeichen `symbol` an der Position des Kopfes auf das Band schreiben.

Die Methoden `moveLeft()` und `moveRight()` sollen den Kopf um eine Position nach rechts oder nach links verschieben.

Die Methode `dumpAlpha()` soll den Teil des bisher verwendeten Bands ausgeben, der links der aktuellen Kopfposition ist (exklusive der aktuellen Position).

Die Methode `dumpBeta()` soll den Teil des bisher verwendeten Bands ausgeben, der rechts der aktuellen Kopfposition ist (inklusive der aktuellen Position).

Die Methode `usedSpace()` soll die Grösse (die Anzahl der Bandfelder) des bisher verwendeten Bandes zurückgeben.

(b) Implementieren Sie den Konstruktor `TuringMachine(Q, Sigma, Gamma, delta, q0, blank, E)`, wobei die einzelnen Parameter den normalen Elementen einer Turing-Maschine entsprechen. Bei der Erstellung der Turing-Maschine soll sichergestellt werden, dass sie korrekt spezifiziert ist: Ist `delta` eine totale Funktion? Ist `Sigma` Teilmenge von `Gamma`? Ist das `blank` in `Gamma`, aber nicht in `Sigma`? Ist `E` eine Teilmenge von `Q`? Werfen Sie eine `InvalidSpecificationException`, wenn die Spezifikation nicht in Ordnung ist.

(c) Implementieren Sie die restlichen Methoden der Klasse `TuringMachine`:

`initialize(word)` soll die Turing-Maschine initialisieren (d.h. die initiale Konfiguration herstellen). Stellen Sie bitte sicher, dass nur Symbole aus dem Eingabealphabet in `word` vorkommen (sonst werfen Sie eine `InvalidSpecificationException`).

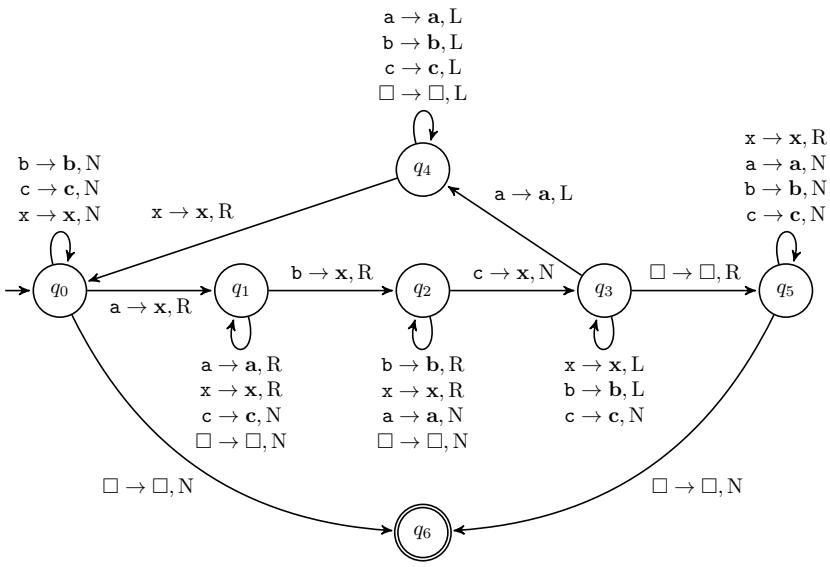
`step()` soll einen Schritt der Turing-Maschine ausführen. Werfen Sie eine `RuntimeException`, wenn die Maschine nicht initialisiert wurde oder in einem Endzustand ist.

Die Methode `dumpConfiguration()` soll die aktuelle Konfiguration der Turing-Maschine ausgeben.

Die Methode `dumpStatistics()` soll ausgeben, wieviele Schritte bereits gemacht wurden und wieviel Platz auf dem Band verwendet wurde.

`run(maxSteps)` soll die Turing-Maschine laufen lassen. Die Maschine soll nur anhalten wenn sie in einem Endzustand ist oder wenn die Anzahl Schritte `maxSteps` erreicht hat. Dies ist nützlich, um eine Maschine auf einem Wort testen zu können, auf dem sie nicht hält. Die Methode `run()` ruft `run(Integer.MAX_VALUE)` auf, um ein unbeschränktes Laufen der Turingmaschine zu simulieren. Rufen Sie `dumpConfiguration()` am Anfang der Simulation und nach jedem Schritt auf, und rufen Sie `dumpStatistics()` am Ende der Simulation auf.

(d) Betrachten Sie die folgende Turing-Maschine.



Implementieren Sie die `main`-Methode der Klasse `SimulateTM`, die ein Objekt der Klasse `TuringMachine` erstellen soll, welches die oben dargestellte Turing Maschine repräsentieren soll. Testen Sie verschiedene Eingaben für die Turing Maschine. Darunter mindestens:

- das leere Wort, und
- zwei Wörter, die in der Sprache liegen welche die Turing Maschine akzeptiert (aber nicht das leere Wort oder *aabbcc*), und
- zwei Wörter, die nicht in der Sprache liegen welche die Turing Maschine akzeptiert.

Dabei soll bei jedem Schritt die Konfiguration ausgegeben werden. Für Wörter, die nicht in der Sprache liegen, verwenden Sie bitte einen kleinen Wert für `maxSteps`, bei dem man schon sehen kann, dass die TM nicht mehr halten wird. Für Wörter die in der Sprache liegen, rufen Sie `run()` ohne Parameter auf.

Zu Ihrer Kontrolle: Auf der Eingabe *aabbcc* benötigt die Turing-Maschine 29 Schritte und verwendet 8 Bandzellen.