Informatik I: Einführung in die Programmierung

Prof. Dr. Bernhard Nebel
Tim Schulte, Thorsten Engesser
Wintersemester 2017/2018

Universität Freiburg
Institut für Informatik

Übungsblatt 4

Abgabe: Freitag, 17. November 2017, 20:00 Uhr

Wichtige Hinweise: Zur Bearbeitung der Übungsaufgaben legen Sie bitte ein neues Unterverzeichnis `sheet04` im Wurzelverzeichnis Ihrer Arbeitskopie des SVN-Repositories an. Ihre Lösungen werden dann in Dateien in diesem Unterverzeichnis erwartet. Beachten Sie bitte bei allen Aufgaben die *Hinweise zur Bearbeitung der Übungsaufgaben* unter der folgenden URL:

`http://gki.informatik.uni-freiburg.de/teaching/ws1718/info1/guide/hinweise.html`

Bewertet wird bei allen Aufgaben die letzte Version, die zur Deadline des Übungsblattes auf dem SVN-Server eingereicht ist.

Aufgabe 4.1 (Gefangenendilemma; Dateien: (a) `prisoner-description.txt`, `prisoner-diagram.{jpg|pdf}`, (b) `prisoner.py`; Punkte: 2+2+2)

In der Vorlesung wurde das wiederholte Gefangenendilemma sowie einige Beispielstrategien vorgestellt. In dieser Aufgabe geht es darum, eine Strategie Ihrer Wahl (z.B. eine der Beispielstrategien oder eine eigene Strategie) als endlichen Automaten zu implementieren. Die Ausgabefunktion soll jeweils die zu wählende Aktion (`"C"` für cooperate oder `"D"` für defect) ausgeben. Die Übergangsfunktion wählt den nächsten Zustand des Automaten in Abhängigkeit vom eigenen gegenwärtigen Zustand und der gerade gespielten Aktion des Mitspielers (ebenfalls `"C"` oder `"D"`) aus. Gehen Sie davon aus, dass jede neue Runde mit der Wahrscheinlichkeit 1/20 die letzte Runde ist.

(a) Beschreiben Sie kurz die zugrundeliegende Idee Ihrer Strategie in der Datei `prisoner-description.txt`.

Skizzieren Sie den Automaten als Übergangsdiagramm (Datei: `prisoner-diagramm.jpg` oder `prisoner-diagramm.pdf`).

(b) Implementieren Sie den Moore-Automaten Ihrer Strategie. Benutzen Sie dazu das Template `prisoner.py` von der Webseite der Übungen. Sie müssen nur die beiden Funktionen `next_state(state, input)` und `output(state)` implementieren. Beachten Sie dabei, dass der Startzustand des Automaten die `0` ist. Indem Sie das Skript mittels `python3 prisoner.py` in der Konsole starten, können Sie einen vordefinierten Benchmark ausführen.

(c) Implementieren Sie eine weitere Strategie mittels der Funktionen `next_state_opponent(state)` und `output_opponent(state)`, um diese mit der in Aufgabe 4.1(b) definierten Strategie zu vergleichen. Achten Sie darauf, dass die beiden Strategien sich unterscheiden. Wenn Sie diese Teststrategie implementiert haben, wird sie beim Aufruf von `python3 prisoner.py` ebenfalls zum Benchmarken verwendet.

Wir werden die von Ihnen eingereichten Strategie-Implementierungen aus Aufgabenteil 4.1(b) paarweise gegeneinander antreten lassen. Die Implementierung mit der besten Strategie (d.h. den wenigsten gesamt akkumulierten Jahren in Gefangenschaft) erhält einen Preis.

Aufgabe 4.2 (Tower of Hanoi; Datei: `hanoi.py`, `hanoi.txt`; Punkte: 1+3+3)

Das *Tower of Hanoi*-Problem ist ein bekanntes Knobelspiel, welches sich durch einen rekursiven Algorithmus lösen lässt. Dabei sind initial eine Anzahl n von unterschiedlich großen Scheiben der Größe nach (von unten nach oben) kleiner werdend auf einem ersten Stab *Quelle* gestapelt. Zwei weitere, leere Stäbe *Hilf* und *Ziel* befinden sich rechts von diesem ersten. Das Ziel des Spielers ist es nun, die Scheiben von der *Quelle* unter Verwendung des *Hilf*sstabes auf das *Ziel* umzustapeln. Dabei darf in jedem Schritt nur eine Scheibe von einem Stab auf einen anderen bewegt werden und niemals darf eine größere Scheibe auf einer kleineren liegen (siehe auch `http://de.wikipedia.org/wiki/Türme_von_Hanoi` für weitere Informationen).
Das folgende Python Programm löst Türme von Hanoi Instanzen rekursiv. Dabei wird jeder Stab als eine Liste bestehend aus einem Namen für den Stab und einer Liste der Scheiben, die auf ihm liegen, repräsentiert.

```
def move_tower(n, source, helper, target):
    # assumes n <= len(source)
    if n <= 0:
        return None
    # (1) was passiert mit source, target, helper in der nächsten Zeile?
    move_tower(n-1, source, target, helper)
    # (2) was passiert mit source, target, im folgenden Anweisungsblock?
    src_name, src_rod = source
    trg_name, trg_rod = target
    disc = src_rod[-1]
    print("Bewege", disc, "von", src_name, "nach", trg_name)
    del src_rod[-1]
    trg_rod += [disc]
    # (3) was passiert mit source, helper, target in der nächsten Zeile?
    move_tower(n-1, helper, source, target)

move_tower(5, ("Quelle", [5,4,3,2,1]), ("Hilf", []), ("Ziel", []))
```

Im Aufruf in der letzten Zeile werden erstens die Anzahl der Scheiben auf `source` als Integer und zweitens die jeweiligen Inhalte und Bezeichner der drei Stäbe in Form zwei-elementiger Listen als Argumente übergeben. Es wird also beispielhaft eine Türme von Hanoi Instanz mit fünf Scheiben gelöst.
Benutzen sie zunächst den Python-Tutor, um ein Gefühl für die Funktionsweise des Algorithmus zu erlangen. Probieren Sie anstelle des Aufrufs in der letzten Zeile nacheinander:

```
move_tower(0, ["Quelle", [5,4,3,2,1]], ["Hilf", []], ["Ziel", []])
move_tower(1, ["Quelle", [5,4,3,2,1]], ["Hilf", []], ["Ziel", []])
move_tower(2, ["Quelle", [5,4,3,2,1]], ["Hilf", []], ["Ziel", []])
...
```

(a) Beschreiben Sie in einem Satz, was ein Aufruf der Funktion `move_tower` in Abhängigkeit des Arguments `n`, aber für feste Eingaben für `source`, `helper` und `target` macht.

(b) Ersetzen Sie die Kommentare (1), (2) und (3) durch eigene Kommentare, die die jeweiligen Fragen kurz beantworten.

(c) Argumentieren Sie induktiv, warum das Problem vom Programm korrekt gelöst wird. Überlegen Sie sich hierzu zunächst wie das Programm das Problem mit einer einzigen Scheibe löst. Nehmen Sie dann an, das Programm würde auch das n-Scheiben Problem korrekt lösen und begründen Sie, wieso dann auch das $n+1$-Scheiben Problem korrekt gelöst wird.

Aufgabe 4.3 (Operieren mit Tupeln; Datei: `polynomial.py`; Punkte: 3+2)
Im Folgenden repräsentieren wir ein Polynom n-ten Grades

$$p(X) = a_n X^n + \cdots + a_2 X^2 + a_1 X + a_0$$

mit rationalen Koeffizienten ($a_n \neq 0$) durch ein Python-Tupel von floats

`(a_n, ..., a_2, a_1, a_0)`.

Die Tupel-Repräsentation eines Polynoms n-ten Grades ist also immer ein Tupel der Länge $n+1$ und der Eintrag für den höchsten Koeffizient des Polynoms `a_n` ist von 0 verschieden. Das Nullpolynom, bei welchem alle a_i Null sind, wird als leeres Tupel `()` repräsentiert.

(a) Definieren Sie in der Datei `polynomial.py` eine Funktion `first_deriv(t)`, die bei Eingabe eines Tupels `t`, das ein Polynom $p(X)$ repräsentiert, eine Tupel-Repräsentation der 1. Ableitung dieses Polynoms zurückgibt.

(b) Definieren Sie eine Funktion `deriv(t, k)`, die bei Eingabe einer Tupel-Repräsentation `t` eines Polynoms $p(X)$ eine entsprechende Repräsentation der k-ten Ableitung des Polynoms zurückgibt (für negatives k soll die Funktion `None` zurückgeben und für $k = 0$ das eingegebene Tupel selbst).

Aufgabe 4.4 (Erfahrungen; Datei: `erfahrungen.txt`; Punkte: 2)
Legen Sie im Unterverzeichnis `sheet04` eine Textdatei `erfahrungen.txt` an. Notieren Sie in dieser Datei kurz Ihre Erfahrungen beim Bearbeiten der Übungsaufgaben (Probleme, benötigter Zeitaufwand nach Teilaufgabe, Bezug zur Vorlesung, Interessantes, etc.).