Klausur „Algorithmen und Datenstrukturen (AuD) II“ (Sommer 2010)

21. September 2010, 11:00 – 13:00

Aufgabe 1 [Gesamtpunktzahl: 9]

a) Aus Vorlesung und Übung kennen Sie den auf dynamischem Programmieren beruhenden *Algorithmus für den sog. k-approximativen Match*. Bei dieser fehlertoleranten Variante des Abgleichs zwischen einem Muster und einem Text sind bis zu k-mal eine der folgenden Editieroperationen anwendbar:

- Ersetzung des Textzeichens durch das Musterzeichen,
- Löschen des Textzeichens und
- Einfügen des Musterzeichens in den Text

a1) Erstellen Sie die im Algorithmus für k-approximativen Match verwendete **Differenzentabelle** für den **Text** *Preis* und das **Muster** *Prise*.
Für welches kleinste `k` ergeben sich k-approximative Übereinstimmungen zwischen dem (gesamten) Muster und dem Text?

a2) Lesen Sie für **alle** k-approximativen Übereinstimmungen mit dem in a1) bestimmten kleinsten `k` die jeweils erforderlichen Editieroperationen aus der Differenzentabelle aus und geben sie diese in nachvollziehbarer Weise an.

a3) Welcher k-approximative Match mit minimalem k ergibt sich mit welchen Editieroperationen für den **Text** *Prise* und das **Muster** *Preis*?

b) Aus Vorlesung und Übung wissen Sie, wie mit *Suffixtries* überprüft werden kann, ob (und wenn ja, wo) in einem Text gesuchte Strings vorkommen.

b1) Erstellen Sie den Suffixtrie für den **Text** *abrakadabra* in kompakter Form.

b2) Erläutern Sie, wie mit dem in b1) erstellten Suffixtrie gezeigt werden kann, dass die Suchstrings *ada* und *raka* im **Text** *abrakadabra* vorkommen, der Suchstrings *adaba* aber nicht.

Aufgabe 2 [Gesamtpunktzahl: 11]

Ein ungerichteter Graph heißt *bipartit*, wenn sich seine Knoten so in zwei disjunkte Mengen `R` und `B` aufteilen lassen, dass die Knoten in den jeweiligen Mengen *keine* verbindenden Kanten besitzen.

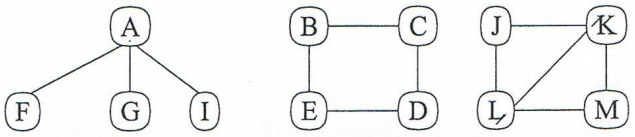

Abbildung 1: *Ungerichtete* Graphen: bipartit, bipartit, nicht bipartit (v.l.n.r.)

Beispiele: In Abb. 1 sind von links nach rechts zu sehen:

- bipartiter Graph mit den Knotenmengen `R = {A}` und `B = {F,G,I}`
- bipartiter Graph mit den Knotenmengen `R = {B,D}` und `B = {C,E}`
- kein bipartiter Graph (da sie verbunden sind, müssten die Knoten K und L unterschiedlichen Mengen angehören und der Knoten J kann weder in der Menge mit K noch in der Menge mit L liegen).

a) Welche der folgenden, durch ihre Adjazenzmatrizen angegebenen Graphen sind bipartit, welche nicht? Begründen Sie Ihre Antworten kurz, aber nachvollziehbar.

```
      0 1 0 0 1          0 1 1 1 0          0 1 0 0 0 1
      1 0 1 0 0          1 0 0 0 1          1 0 1 0 0 0
G1 =  0 1 0 1 0     G2 = 1 0 0 0 1     G3 = 0 1 0 1 0 0
      0 0 1 0 1          1 0 0 0 1          0 0 1 0 1 0
      1 0 0 1 0          0 1 1 1 0          0 0 0 1 0 1
                                            1 0 0 0 1 0
```

b) Konzipieren Sie einen Algorithmus, um für einen beliebigen ungerichteten Graphen festzustellen, ob er bipartit ist oder nicht. Beschreiben Sie den Algorithmus zunächst in Worten und wenden ihn auf einen bipartiten und einen nicht bipartiten Graphen an.

c) Setzen Sie Ihren Algorithmus aus b) dann in Haskell als Prädikat `isbp` um. Dabei stehen ihnen die Funktionen aus dem ADT `Graph` sowie alle Ihnen bekannten Funktionen aus den Modulen `Prelude.hs` und `List.hs` zur Verfügung (s.u.). Wenn Sie weitere eigene Hilfsfunktionen verwenden, sind diese zu definieren.

Moduldefinition und Signaturen des ADT Graph:

```
module Graph(Graph,mkGraph,adjacent,nodes,edgesU,edgesD,edgeIn,
                  weight) where
import Array

mkGraph :: (Ix n,Num w)
  => Bool -> (n,n) -> [(n,n,w)] -> (Graph n w)

adjacent :: (Ix n,Num w) => (Graph n w) -> n -> [n]

nodes :: (Ix n,Num w) => (Graph n w) -> [n]

edgesU :: (Ix n,Num w) => (Graph n w) -> [(n,n,w)]
edgesD :: (Ix n,Num w) => (Graph n w) -> [(n,n,w)]

edgeIn :: (Ix n,Num w) => (Graph n w) -> (n,n) -> Bool

weight :: (Ix n,Num w) => n -> n -> (Graph n w) -> w
```

Aufgabe 3 [Gesamtpunktzahl: 10]

Sei ein Muster `P` ein String der Länge `m` aus Buchstaben und null oder mehr Vorkommen eines Fragezeichens `'?'` als sog. Metazeichen. Dieses Metazeichen bedeutet, dass das vorausgehende Zeichen *optional* wird, d.h. für einen erfolgreichen Abgleich ('Match') zwischen Muster `P` und Text `T` in `T` vorkommen *kann*, aber *nicht muss*.

Beispiele:

- Das Muster `P = "baye?risch"` passt sowohl auf `T = "bayerisch"`, als auch auf `T = "bayrisch"`.
- Für das Muster `P = aab?cd?e"`
- ergeben sich u.a. erfolgreiche Abgleiche mit den folgenden Texten:
  - `"aace"`
  - `"aabce"`
  - `"aacde"`
  - `"aabcde"`

a) Konzipieren Sie einen Algorithmus, um festzustellen, ob für ein Muster `P` (der Länge `m`) mit null oder mehr Vorkommen eines Fragezeichens `'?'` ein Match mit einem Präfix eines Texts `T` (der Länge `n`) existiert und beschreiben Sie diesen Algorithmus zunächst in Worten.

b) Setzen Sie Ihren Algorithmus **in Haskell** als Funktion **mwo** (für match with optionals) um.

Der Rückgabewert von **mwo** soll ein Bool-Wert (für: *Liegt (mindestens ein) Match vor oder nicht?*) sein.

c) Konstruieren Sie sich ein Muster aus mindestens 5 Elementen, das drei aufeinanderfolgende optionale Elemente enthält (z.B. `a?b?a?`). Geben Sie dann Beispiele von Texten an, bei denen ein Match zwischen Muster und Text

- alle optionalen Elemente,
- kein optionales Element bzw.
- eine von Ihnen gewählte beliebige Teilmenge der optionalen Elemente

benötigt.

Was können Sie daraus ableiten für den Einfluss der Zahl `q` der optionalen Elemente im Muster auf den Aufwand Ihres Algorithmus ($\mathcal{O}$-Notation)?

Aufgabe 4 [Gesamtpunktzahl: 10]

Betrachten Sie folgendes Problem: Gegeben sind drei Becher Becher1 mit drei, Becher2 mit fünf und Becher3 mit acht Litern Fassungsvermögen, allerdings ohne Markierungen für Teilvolumen. Anfänglich ist der Becher mit acht Litern Fassungsvermögen voll. *Wie lassen sich vier Liter lediglich durch Umfüllen zwischen Bechern erreichen?* (Ausgießen ist also nicht erlaubt, lediglich Umgießen zwischen zwei Bechern Quelle und Senke bis entweder die Quelle leer und/oder die Senke gefüllt.)

Hinweis: eine Lösung ergibt sich dadurch, dass man durch geeignetes Umgießen für zwei Liter im Becher mit drei Litern Fassungsvermögen sorgt (wie?) und dann aus dem vollen Becher mit fünf Litern das Gefäß mit drei Litern Fassungsvermögen auffüllt. Dieses Problem soll mit Hilfe einer *Suche mit Backtracking* gelöst werden.

a) Entwickeln und skizzieren Sie zunächst *von Hand* den zugehörigen Suchraum bis zu einer Tiefe von mindestens zwei durchgeführten Zügen.

b) Konzipieren Sie dann Datenstrukturen in Haskell, um die *Situationen* (d.h. die Aufteilungen der Flüssigkeit auf die drei Becher) und die Zustände (Knoten) des Suchraums darzustellen. Wie sieht der Startzustand, wie sehen mögliche Zielzustände aus?

c) Für die Verwendung in der Haskell-Funktion `searchDfs` (s.u.) ist eine Nachfolgerfunktion `succ` erforderlich, die einem *Knoten* im Suchraum die Liste der Folgeknoten zuordnet.

Um `succ` zu definieren, soll eine Hilfsfunktion `possibleMoves` verwendet werden, die einer *Situation* (s.o.) die Liste der möglichen Folgesituationen zuordnet. Diese sei definiert als

```
possibleMoves sit = fromMug1 sit ++ fromMug2 sit ++ fromMug3 sit
```

Konzipieren Sie im folgenden die Funktion `fromMug1`, die für eine Situation – wie in b) definiert – bestimmt, welche Folgesituationen sich durch einmaliges Umgießen *aus Becher1* gewinnen lassen. (Die Funktionen `fromMug2` und `fromMug3` arbeiten analog und brauchen in der Klausur nicht definiert zu werden).

c1) Beschreiben Sie Ihren Ansatz zunächst detailliert mit Worten.

c2) Definieren Sie dann die Funktion `fromMug1` (und gegebenfalls erforderliche Hilfsfunktionen) in Haskell.

Zur Erinnerung:

```
searchDfs:: (Eq node) => (node -> [node]) -> (node -> Bool)
                                          -> node -> [node]

searchDfs succ goal x = search' (push x emptyStack)
 where
   search' s
    | stackEmpty s = []
    | goal (top s) = top s : search' (pop s)
    | otherwise    = let x = top s
                     in search' (foldr push (pop s) (succ x))
```