Lehrstuhl für Informatik I
Algorithmen, Komplexität und
wissensbasierte Systeme
Universität Würzburg

Würzburg, den 16. Januar 2020

Prof. Dr. Alexander Wolff
Andre Löffler, M. Sc.

3. Präsenzübungsblatt zur Vorlesung Algorithmen und Datenstrukturen (Winter 2019/20)

Aufgabe 1 – Springer auf Schachfeld

Gesucht ist ein Algorithmus, der ermittelt, wieviele Züge ein Springer benötigt, um auf einem Schachbrett mit $n \times n$ Feldern von Feld (x_1, y_1) zu Feld (x_2, y_2) zu kommen.

a) Wie lässt sich dieses Problem als Kürzeste-Wege-Problem auf einem Graphen modellieren? Was repräsentieren die Knoten und Kanten des Graphen?

b) Ist der Graph zweifärbbar (entsprechend der Definition in Blatt 8 Aufgabe 2)? Begründen Sie Ihre Antwort.

c) Geben Sie für das gewöhnliche Schachbrett mit 8×8 Feldern die genaue Anzahl der Kanten des Graphen an.

d) Geben Sie für ein Schachbrett mit $n \times n$ Feldern eine obere Schranke für die Anzahl der Kanten an. Verwenden Sie dafür die Groß-O-Notation.

e) Geben Sie einen Algorithmus an, der die erforderliche Anzahl an Zügen in $\Theta(n^2)$ Zeit berechnet. Begründen Sie die Laufzeit Ihres Algorithmus.

Aufgabe 2 – Rot-Schwarz-Baum augmentieren

Ein Rot-Schwarz-Baum zur Verwaltung einer dynamischen Menge verschiedener ganzer Zahlen soll so augmentiert werden, dass man zu jeder Zeit bestimmen kann, für welche zwei Zahlen i, j der Menge mit $i < j$ die Differenz $j - i$ am kleinsten ist.

a) Geben Sie die Methode MinGap in Pseudocode an, die das gesuchte Zahlenpaar in konstanter Zeit liefern soll.

   Benennen Sie die Extrainformation, die dafür zu speichern ist, und geben Sie an, wie Sie diese in den Methoden Insert, Delete und Search aufrechterhalten können, ohne deren asymptotische Worst-Case-Laufzeiten zu verschlechtern.

b) Können Sie das Problem auch mit konstantem Speicher für Extrainformation lösen, wenn Sie auf die Methode Delete verzichten (also nur eine *halbdynamische* Menge verwalten)?

Aufgabe 3 – Amortisierte Analyse

In der Vorlesung haben Sie die Implementierung der Funktion Successor für binäre Suchbäume kennengelernt, die auch unten angegeben ist. Die Worst-Case-Laufzeit dieser Funktion ist $\Theta(h)$, wobei h die Höhe des Baumes ist.

```
Node Successor(Node x)
if x.right ≠ nil then
    return Minimum(x.right)
y = x.p
while y ≠ nil and x == y.right do
    x = y
    y = y.p
return y
```

Auf einem binären Suchbaum wird die Funktion Successor nun für jeden der n Knoten genau einmal aufgerufen. Begründen Sie, dass in diesem Fall die amortisierte Laufzeit eines einzelnen solchen Aufrufes von Successor in $O(1)$ ist.

PABS **Aufgabe 4 – Gerichteten Graphen implementieren**

Implementieren Sie einen gerichteten Graphen in Java. Jeder Knoten des Graphen soll einen Verweis auf ein Objekt der Klasse Object speichern. Die gerichteten Kanten des Graphen sollen durch Adjazenzlisten repräsentiert werden. Speichern Sie die Knotenmenge des Graphen als Liste.

Verwenden Sie als Implementierung für die Listen die Klassen die vorgegebenen Klassen List und ListItem. Schreiben Sie die Klassen derart um, dass sie nun Objekte vom Typ DiGraphNode verwalten (anstatt wie bisher vom Typ Double). Verwenden Sie keine Datenstrukturen und Algorithmen der Java API oder zusätzlicher Bibliotheken, die Sie nicht selbst programmiert haben.

Hinweis: Bei dieser Aufgabe sind die PABS-Tests optional. Sie brauchen nicht alle Tests zu erfüllen, um Ihren Code als Lösung abgeben zu können, sofern ihr Code zumindest erfolgreich kompiliert. Entsprechend gibt es Teilpunkte für unvollständige oder fehlerhafte Lösungen.

a) Implementieren Sie im Paket diGraph die Klasse public class DiGraphNode . Diese soll einen Konstruktor

   - `public DiGraphNode(Object key)`

   enthalten, der einen neuen Knoten mit einem Verweis auf das Objekt key der Klasse Object erzeugt. Ferner soll die Klasse DiGraphNode über folgende Methoden verfügen:

   - public List getNeighbors() – gibt die Adjazenzliste des Knotens zurück
   - public Object getKey() – gibt das Objekt zurück, auf das der Knoten verweist

Implementieren Sie außerdem ebenfalls im Paket diGraph die Klasse public class DiGraph. Diese Klasse soll einen Konstruktor

- public DiGraph()

enthalten, welcher einen Graphen mit einer leeren Knotenliste erzeugt. Mit zusätzlich soll die Klasse über folgende Methoden verfügen:

- public DiGraphNode find(Object key) – überprüft, ob der Graph einen Knoten v mit key.equals(v.getKey()) enthält und gibt entweder v oder Null zurück.
- public DiGraphNode addNode(Object key) – erzeugt einen neuen Knoten v mit Verweis auf key, fügt diesen dem Graphen hinzu und gibt dann v zurück
- public void addEdge(DiGraphNode v1, DiGraphNode v2) – erzeugt eine gerichtete Kante vom Knoten v1 zum Knoten v2

b) Implementieren Sie in der Klasse DiGraph den Konstruktor

- public DiGraph(Object[ ] keys, boolean[ ][ ] adjacencyMatrix)

Dieser Konstruktor soll einen Graphen erzeugen, der für jedes Objekt im Feld keys einen Knoten enthält. Vom Knoten fürs Objekt keys[i] zum Knoten fürs Objekt keys[j] soll es genau dann eine gerichtete Kante im Graphen geben, wenn adjacencyMatrix[i][j] den Wert true hat.

Diese Aufgaben werden eventuell gemeinsam in den Übungen am 21. und 22. Januar 2019 gelöst. Sie brauchen Sie nicht vorher zu lösen und auch nicht abzugeben.