Universität Bremen

Praktische Informatik 3 WS 16/17

Christoph Lüth
Tobias Brandt
Tristan Bruns
Johannes Ganser
Berthold Hoffmann
Alexander Kurth

3. Übungsblatt

Ausgabe: 08.11.16
Abgabe: 18.11.16, 12:00 Uhr

3.1 *Pizza-Toni liefert wieder.* *20 Punkte*

Mit seinem neuen Lagerhaltungssystem kann "Toni" Corleone jetzt doppelt so viele Pizzen backen wie vorher. Leider kommt die Auslieferung dabei nicht hinterher: Guiseppe mit seiner Vespa fährt leider völlig unstrukturiert durch die Stadt und braucht Stunden für eine Tour.

Deshalb wollen wir auch hier — in Hoffnung auf freie Pizza— mit einer professionalen Lösung aushelfen. Etwas abstrakt stellt sich das Problem wie folgt dar:

Die Karte mit den Lieferung ist ein beliebig verzweigender, über der Knotenmarkierung parametrisierter Graph, dessen Kanten mit einer Kostenfunktion (für den Übergang entlang dieser Kante) gewichtet sind. Ein *Pfad* enthält eine Liste von Knotenmarkierungen, entlang derer der Pfad verläuft, sowie die Summe der Kosten der Übergänge (Kantengewichte). Wir suchen jetzt von den Pfaden, die alle Knoten im Graphen genau einmal enthalten und wieder zum Anfang zurückführen (Hamilton-Zyklen), diejenigen mit den kleinsten Kosten (minimaler Hamilton-Zyklus).

Das Problem ist in der Literatur als das *travelling pizza delivery man problem* bekannt, und ist leider NP-vollständig. Wir lösen es hier durch *brute force*: wir erzeugen alle Zyklen, und filtern davon einen minimalen Hamilton-Zyklen heraus.

Im einzelnen gehen wir wie folgt vor:

1. Der Graph wird durch folgende Datentypen modelliert:

   ```
   type Weight = Integer
   type Graph n = Node n
   data Node n = N n [Edge n]
   data Edge n = E Weight (Node n)
   ```

   Hierbei sollen zwei Knoten gleich sein, wenn ihre Markierung gleich ist.

2. Pfade werden wie folgt modelliert:

   ```
   data Path a = P { cost :: Integer, path :: [a] } deriving Show
   ```

   Implementieren Sie dazu folgende Operationen, welche einen Knoten mit einer Gewichtung zu einem Pfad hinzufügt, bzw. prüft, ob ein Knoten in einem Pfad enthalten ist:

   ```
   add      :: Path a→ Integer→ a→ Path a
   contains :: Eq a ⇒ Path a→ a→ Bool
   ```

3. Implementieren Sie eine Funktion, welche alle von den angegebenen Startknoten ausgehenden Zyklen findet:

   ```
   cycles :: Eq n⇒ Node n→ [Path n]
   ```

   Die nötige Traversionsfunktion führt einen Pfad zu dem momentanen Knoten n mit. Es werden alle von n ausgehenden Kanten traversiert (und zu dem Pfad hinzugefügt), deren Zielknoten nicht in dem Pfad zu n enthalten ist. Ist der Zielknoten wieder der Startknoten, dann haben wir einen Zyklus gefunden.

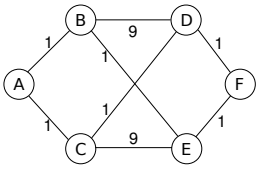

Abbildung 1: Einfacher Beispielgraph

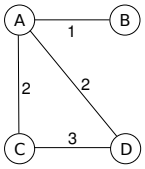

Abbildung 2: Beispielgraph ohne Hamilton-Zyklus

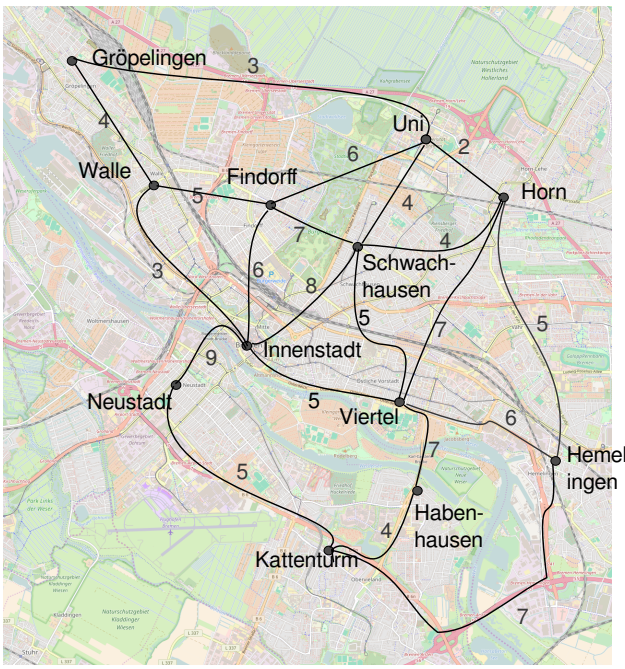

Abbildung 3: Die Pizzaliefertour als Graph. Wie man sieht, sind Toni Corleones Pizzen in der ganzen Stadt beliebt.

4. Wie wir in Abb. 2 sehen, haben nicht alle Graphen einen Hamilton-Zyklus. Wir brauchen deshalb noch eine Funktion

   ```
   nodes :: Eq n⇒ Node n→ [n]
   numNodes :: Eq n⇒ Node n→ Int
   ```

   welche die von einem Startknoten erreichbaren Knoten (auf)zählen. Mit dieser Funktion können wir Hamilton-Zyklen erkennen; es sind Pfade der Länge `numNodes n+1`.

5. Damit können wir das *travelling pizza delivery man problem* lösen, indem wir den kürzesten Hamilton-Zyklus berechnen:

   ```
   tpp :: Eq n⇒ Node n→ Maybe (Path n)
   ```

   - Zuerst berechnen wir alle Zyklen, die vom Startknoten ausgehen;
   - Aus der Ergebnisliste filtern wir alle Pfade, die nicht so lang sind wie die Anzahl der vom Startknoten erreichbaren Knoten;
   - aus dieser Ergebnisliste, wenn sie nicht-leer ist, suchen wir den Pfad mit dem minimalsten Gesamtkosten.

Abb. 3 zeigt die Pizzaliefertour als Graph. Dieser Graph ist auch in dem vorgegebenen Test-Rahmenwerk enthalten. Sie können ihn nutzen, um eigene Tests zu schreiben. Ihre Lösung sollte die kürzeste Pizzatour in diesem Graphen finden.

? *Verständnisfragen*

1. Was ist Polymorphie?
2. Welche zwei Arten der Polymorphie haben wir kennengelernt, und wie unterschieden sie sich?
3. Was ist der Unterschied zwischen Polymorphie in Haskell, und Polymorphie in Java?