Praktische Informatik 3 WS 20/21

Christoph Lüth
Thomas Barkowsky
Tobias Brandt
Alexander Krug
Tarek Soliman
Robert Sachtleben

7. Übungsblatt

Ausgabe: 21.12.2020
Abgabe: 18.01.2021 12:00

Dieses ist das erste Gruppenübungsblatt. Bitte bearbeitet es in Übungsgruppen von idealerweise drei, in Ausnahmefällen auch zwei Studierenden. Das Übungsblatt wird mit insgesamt zwanzig Punkten bewertet, und ist in zwei Wochen (zuzüglich der veranstaltungsfreien Zeit über Weihnachten) zu bearbeiten.

7.1 *Kalter Kaffee 4.0* — *6 Punkte*

Seit der deutschlandweiten Expansion des *Happy Coffeecup* Franchise-Systems im 5. Übungsblatt haben sich die Geschäfte weiterhin sehr positiv entwickelt und konnten europaweit ausgedehnt werden. Damit eröffnen sich völlig neue Probleme z.B. bei der Optimierung der Transportwege zwischen den Filialen, und entsprechend sollen wir unsere Verwaltungssoftware erweitern.

Da sich die neuen Probleme auf Filialen und die Verbindungen zwischen ihnen beziehen, entschließen wir uns zur Nutzung einer Graphdatenbank. Der Chef von *Happy Coffeecup* ist immer noch geizig, daher nehmen wir eine quelloffene Graphdatenbank aus dem Internet als ersten Entwurf — wir müssen schnell fertigwerden, und kosten soll es auch wenig.

Die Graphdatenbank *SimpleGDB* hat folgende Typen und Operationen:

- Typ `G a b` repräsentiert einen Graphen mit Knotendaten vom Typ `a` und Kantendaten vom Typ `b`;
- Typen `Node a`, `Link a b` und `Path a b` repräsentieren Knoten, Kanten und Pfade;
- Funktion `empty` erzeugt einen leeren Graphen;
- Funktionen `add_node` und `add_link` fügen einem Graphen einen Knoten bzw. eine Kante hinzu;
- Funktionen `modify_node` und `modify_link` verändern die Daten eines Knoten bzw. einer Kante des Graphen;
- Funktionen `find_nodes`, `find_links` und `find_paths` ermitteln alle Knoten, Kanten, oder von einem Knoten ausgehende Pfade, die bestimmte Eigenschaften erfüllen;
- Funktion `outgoing` ermittelt alle von einem Knoten ausgehenden Kanten

In `SimpleGrphDB.hs` finden Sie die genauen Typsignaturen und eine naive Implementierung dazu, welche wir nun nutzen wollen, um konkrete Probleme im *Happy Coffeecup* Franchise-System zu lösen. Mit `stack haddock` können Sie sich die Dokumentation des Moduls generieren lassen.

Dazu brauchen wir natürlich zuerst konkrete Datentypen für die Knoten- und Kantendaten des Graphen:

```
data Location = Location { locationName :: String
                         , isOpen :: Bool
                         } deriving (Show,Eq,Ord)

data Road = Road { roadName :: String
                 , cost :: Int
                 , roadLength :: Int
                 } deriving (Show,Eq,Ord)
```

Eine `Location` repräsentiert Standorte und verfügt über einem Namen und einen Flag, ob die korrespondierende Filiale aktuell geöffnet ist, während eine `Road` einen Namen, die Kosten (Benzin, Mautgebühren, Fahrzeugabnutzung in Euro) für einmalige Benutzung der Straße und eine Länge (in km) aufweist. Im Folgenden verwenden wir den Graphen `g1`[1] dargestellt in Abbildung 1. Diesen Graphen finden Sie auch in `Beispielgraphen.hs`.

[1] In `Visualizer.hs` finden Sie bei Interesse Funktionen zum Darstellen eines Graphen im `dot`-Format.

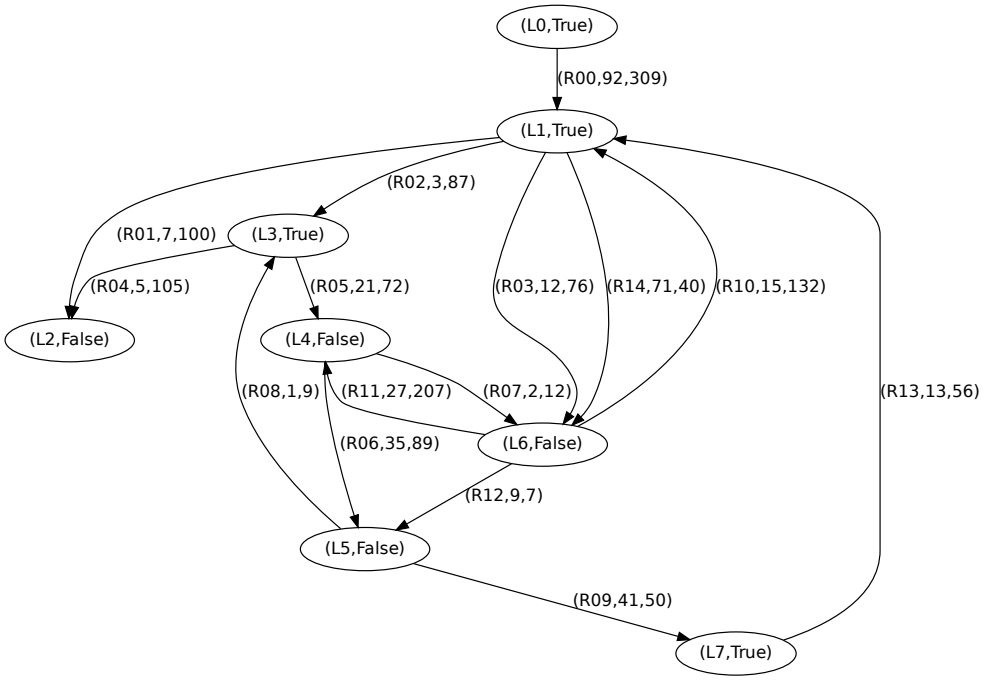

Abbildung 1: Beispielgraph `g1`.

Implementieren Sie nun in `Planner.hs` folgende Funktionen:

```
affordablePaths :: G Location Road → Int → String → Int → String → [([String],Int)]
deadEndLocations :: G Location Road → [Location]
closeLocations :: G Location Road → [Node Location] → G Location Road
increaseCost :: G Location Road → Int → Int → G Location Road
```

- `affordablePaths g maxLen startName maxCost targetName` soll alle Pfade aus maximal `maxLen` Elementen ermitteln, die in `g` von einer Location namens `startName` zu einer Location namens `targetName` führen und deren Summe der Kosten `maxCost` nicht überschreitet. Für jeden solchen Pfad soll im Rückgabewert eine Paar aus der Liste der Straßennamen des Pfades und der Summe der Kosten enthalten sein.

  ```
  affordablePaths g1 10 "L1" 30 "L2"  ⇝
      [(["R01"],7),(["R02","R04"],8),(["R03","R12","R08","R04"],27)]
  ```

- `deadEndLocations g` soll alle Standorte in `g` ermitteln, von denen aus keine Straßen wegführen.

  ```
  deadEndLocations g1  ⇝
      [Location {locationName = "L2", isOpen = False}]
  ```

- `closeLocations g locations` soll die Standorte `locations` in `g` schließen.

- `increaseCost g len amount` soll die Kosten aller Straßen in `g`, die eine Länge von mehr als `len` aufweisen, um `amount` erhöhen.

Hinweise: Für Fälle, in denen Sie Funktionen benötigen, die unabhängig von der Eingabe immer einen bestimmten Wert zurückgeben, kann die vordefinierte Funktion `const` nützlich sein.

7.2 *Machs mit Maps* *8 Punkte*

Für kleine Graphen wie `g1` ist unsere naive Implementierung aus `SimpleGrphDB` noch ausreichend, wenn wir jedoch beispielsweise auf größeren Graphen häufig Knoten oder Kanten modifizieren wollen, oder wir häufig in einem dichten Graphen alle ausgehenden Kanten eines Knoten suchen, dann kommt die Implementierung schnell an ihre Grenzen, da hier immer wieder die gesamte Knoten- bzw. Kantenliste durchlaufen werden muss[2]. Zusätzlich verwendet die naive Implementierung die Label von `Node` (und `Link`), um Knoten (und Kanten) untereinander zu unterscheiden, und kann daher nicht mit unterschiedlichen Knoten (Kanten) mit demselben Label umgehen.

Dabei fällt auf, dass diese Probleme in `SimpleGrphDB` alle in der zugrundeliegenden Repräsentation von Knoten, Kanten und Graphen begründet sind. Hier zahlt sich aus, dass wir den Graphen als abstrakten Datentypen (ADT) implementiert haben: wir können einfach eine weitere Implementierung mit der gleichen Signatur erzeugen und dann in `Planner.hs` die verwendete Implementierung dadurch ändern, dass wir statt `SimpleGrphDB` die neue Implementierung importieren. Genau dies wollen wir jetzt tun:

1. Überlegen Sie sich, wie sie die zugrundeliegende Repräsentation von Graphen so formulieren können, dass Knoten und Kanten schnell gefunden werden können und zudem bei der Modifikation von Knoten keine Kante modifiziert werden muss. Hier bietet sich eine Implementierung an, die Knoten und Kanten intern eindeutige Kennungen (beispielsweise fortlaufende ganze Zahlen) zuweist und dann `Map` und `Set` nutzt, um diese Kennungen (Identifier) zu verwalten. Vervollständigen Sie dazu folgende Implementierung (in `OptimizedGrphDB.hs`):

```
...
import Data.Map (Map)
import qualified Data.Set as Set
import Data.Set (Set)
import qualified Data.Map as Map
...
data G a b = Graph { idToNode :: Map Int (Node a)
                   -- TODO: Verwaltung von Kanten-IDs
                   -- TODO: Verwaltung von noch verfügbaren IDs
                   , outgoingEdges :: Map Int (Set Int)
                   } deriving (Eq,Show)
...
```

2. Implementieren Sie auf Basis dieser neuen Repräsentation die in `OptimizedGrphDB` geforderte Signatur. Die Funktionen sollen hier die gleichen Typsignaturen aufweisen wie in `SimpleGrphDB`.

   Beachten Sie hierbei bei der Implementierung folgende Aspekte:

   - Ihre Implementierung soll damit umgehen können, dass verschiedene Knoten oder Kanten die gleichen Nutzdaten aufweisen, ohne dadurch die Knoten oder Kanten zusammenzulegen.
   - Zwei Knoten (Kanten) sollen gleich sein, wenn ihre Kennungen gleich sind.
   - Die Funktionen `add_node` und `add_link` müssen jeweils eine neue Kennzeichnung (Identifier) erzeugen, die noch nicht in dem Graphen verwendet wird.
   - Die Funktionen `add_link`, `modify_link` und `modify_node` sollen den Graphen nur dann verändern, wenn die Eingabeknoten und -kanten bereits im Graphen existieren.

3. Ersetzen Sie in `Planner.hs` den Import von `SimpleGrphDB` durch einen Import von `OptimizedGrphDB` und überprüfen Sie, ob die Funktionalität unverändert bleibt. Wenn Sie hier den Graphen `g1` verwenden wollen, müssen Sie auch in `Beispielgraphen` den Import anpassen. *Dieser Schritt dient nur der Selbstkontrolle ihrer Implementierung und muss nicht dokumentiert werden.*

Hinweise: Die abstrakten Datentypen `Map` und `Set` bieten viele Operationen an, die hier nützlich sind:

- https://www.stackage.org/haddock/lts-16.26/containers-0.6.2.1/Data-Map-Strict.html#t:Map
- https://www.stackage.org/haddock/lts-16.26/containers-0.6.2.1/Data-Set.html

[2] Vgl. Implementierung von `modify_node`, `modify_link` und `outgoing` in `SimpleGrphDB.hs`

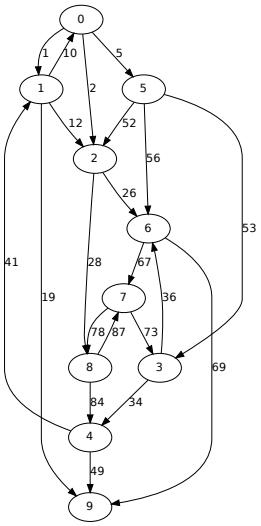

Abbildung 2: Beispielgraph g2

7.3 *Schnell noch Testen* *6 Punkte*

Der letzte Schritt der vorherigen Aufgabe allein ist natürlich noch nicht ausreichend, um zu überprüfen, ob unsere neue Implementierung sich korrekt verhält. Überlegen Sie sich daher drei nicht-triviale Eigenschaften Graphen auf dem Graphen g2 (siehe Abbildung 2) über der implementierten Signatur und implementieren Sie diese als QuickCheck-Properties in `QuickCheckTest.hs` unter Testgruppe `exercise7_3`. Sie finden dort bereits eine Beispiel-Eigenschaft:

```
QC.testProperty "Beispieleigenschaft␣1" $
      λx→ for_random_node $ \n1 → node_elem n1 (modify_node g2 n1 x)
```

Ein QuickCheck-Property im Sinne dieser Aufgabe besteht also aus drei Komponenten: Dem Aufruf von `testProperty`, einer Beschreibung des Properties und der Property selbst. QuickCheck generiert dann zufällige Inputs für diese Funktion und prüft, ob diese zu `True` auswertet. Hier können die folgenden Operatoren hilfreich sein:

- Der Operator ($\Longrightarrow$) dient dabei dazu, QuickCheck bei der Generierung zufälliger Inputs einzuschränken, da es hier oft sinnvoll ist, nur Inputs zu verwenden, die bestimmte Bedingungen erfüllen.
- Der Operator ($===$) ist ähnlich zu ($==$), erzeugt bei Fehlschlagen jedoch eine Fehlermeldung die beschreibt, welcher Wert statt dem erwarteten Wert beobachtet wurde.
- Der Kombinator `for_random_node` erzeugt einen Testfall über einen zufälligen Knoten des Graphen g2, und `for_random_nodes` ähnlich für zwei zufällige Knoten.

Eine Ihrer Eigenschaften sollte die Rückgabe von `find_paths` prüfen, indem Sie prüfen, ob für alle Pfade in der Rückgabe von `find_paths` folgendes gilt:

1. Der Pfad ist tatsächlich ein Pfad, d.h. für alle Kanten in dem Pfad ist das Ziel der vorhergehenden die Quelle der nächsten Kante;
2. Alle Knoten und Kanten, die in dem Pfad vorkommen, sind auch in dem Graphen enthalten;
3. Die Liste der Kanten, und der Zielknoten das Pfades, erfüllen die Prädikate, die `find_paths` übergeben wurden.