Christoph Lüth
Bernd Gersdorf
Christian Maeder
René Wagner
Dennis Walter
Diedrich Wolter
Simon Ulbricht

6. Übungsblatt

Ausgabe: 09.12.10
Abgabe: 10.01.11

6.1 *Das Liniennetz* *10 Punkte*

In dieser Übung widmen Sie sich der Berechnung von Verbindungen zwischen Straßenbahnlinien und Bussen der BSAG[1]. Konkret implementieren Sie ein Programm zur Berechnung der (zeit-)optimalen Fahrt von einer Starthaltestelle zu einer Zielhaltestelle. Die Musterdatei `Vorlage-Blatt06.hs` definiert vereinfachte Fassungen der BSAG-Linienpläne (unter Auslassung einiger Stationen und Linien). Ein *Linienplan* gibt den Liniennamen und die Liste der bedienten Stationen inklusive der Fahrtdauer zur nächsten Station an und ist vom Typ

```
type Line = (LineName, [(String, Int)])
```

Ein *Liniennetz* ist ein Graph, der alle benachbarten Haltestellen (in beiden Fahrtrichtungen) aller vorgegebenen Linien verbindet, sowie Umstiege zwischen verschiedenen Linien an derselben Station ermöglicht. Abb. 1 veranschaulicht die Situation. Die Kanten des Graphen sind mit *Übergangskosten* versehen, welche die Fahrt- bzw. Wartedauer in Minuten angeben. Beachten Sie, dass es für Haltestellen mit Umstiegsmöglichkeiten mehrere Knoten im Graphen gibt, die vollständig miteinander zu den Kosten *switch* verbunden sind. Wir nehmen also vereinfachend eine konstante Wartezeit an allen Stationen und zwischen allen Linienpaaren an.

Wir repräsentieren Graphen $G = (V, E \subseteq V \times \mathbb{N} \times V)$ (V ist die Knotenmenge, gerichtete Kanten $(n, c, n') \in E$ verbinden zwei Knoten n, n' zu den Kosten c) als eine endliche Abbildung (`Map`) von (Quell-)Knoten zu Paaren aus je einem (Ziel-)Knoten und den dazugehörigen Übergangskosten.

```
data Node = N LineName String
type Edge = (Node, Int, Node)
type Network = Map Node [(Int, Node)]
```

1. Implementieren Sie eine Funktion zum Hinzufügen einer Kante in den Liniennetzgraphen.

   ```
   addEdge :: Edge → Network → Network
   ```

2. Implementieren Sie eine Funktion zum Hinzufügen der Einzelabschnitte eines Linienverlaufs zu einem gegebenen Liniennetz:

   ```
   addLineNetwork :: Line → Network → Network
   ```

   Für die triviale Linie `(LN "AB", [("A", 3), ("B", 0)])`, die nur von A nach B und zurück fährt, sollen also die Kanten `("A", 3, "B")` und `("B", 3, "A")` hinzugefügt werden. (Die abschließende `0` in jedem Linienplan hat keine Bedeutung und existiert nur aus strukturellen Gründen.)

3. Zudem müssen alle Umstiegsmöglichkeiten zwischen zwei Bahnen zu einem Liniennetz hinzugefügt werden.

   ```
   addSwitches :: Int → Line → Line → Network → Network
   ```

   Hierbei gibt das erste Argument die konstanten Umstiegskosten (d.h. die Wartedauer beim Umstieg) an. Beachten Sie, dass die von dieser Funktion hinzugefügten Kanten die Form `(N ln1 stop, switch, N ln2 stop)` haben, d.h. ihre Knoten haben die Haltestelle gemeinsam.

4. Schließlich fügen Sie die Puzzleteile zur *exportierten* Funktion

   ```
   makeNetwork :: [Line] → Int → Network
   ```

   zusammen, die aus Linienplänen und Umstiegskosten ein Liniennetz formt.

[1] `http://www.bsag.de`

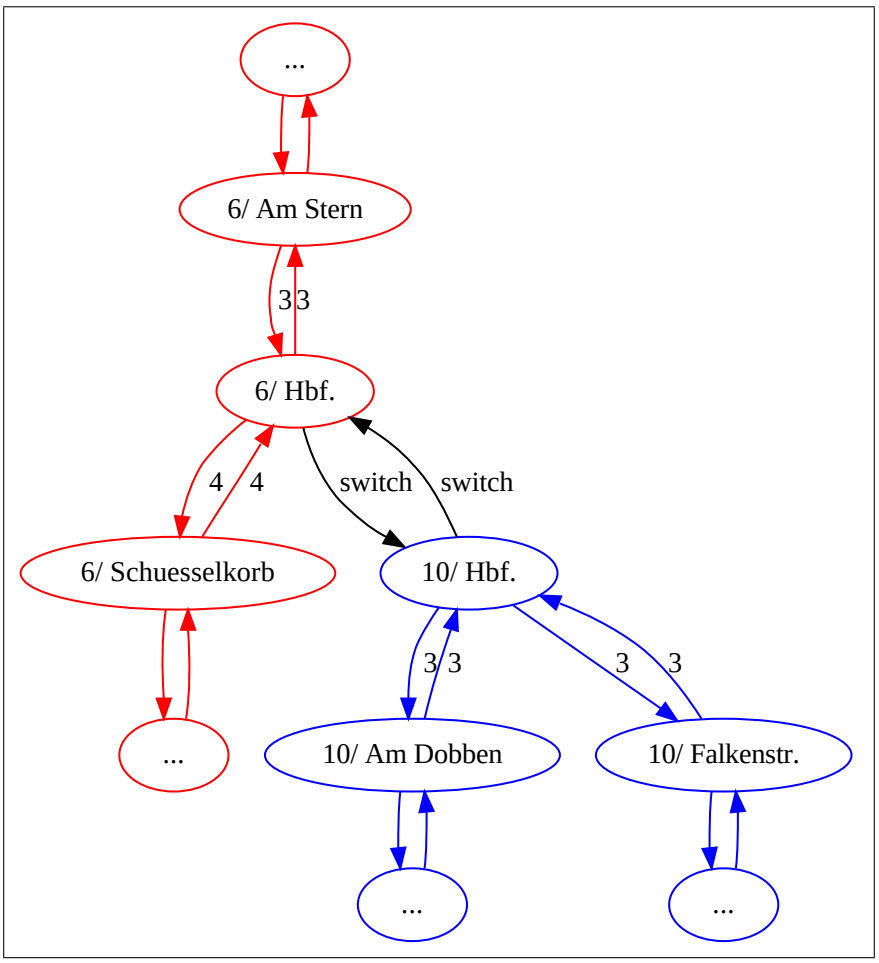

Abbildung 1: Graphrepräsentation der Liniennetzes, mit Umsteigemöglichkeit am Hauptbahnhof

Nützliche Funktionen zur Lösung dieser Aufgabe sind u.a. `foldr`, `Data.Map.insertWith`, `(.)` oder auch `Data.Set.intersection` und `Data.Set.fromList`.

6.2 *Auf kürzestem Wege* *10+2 Punkte*

Implementieren Sie nun eine funktionale Variante von Dijkstras Algorithmus zur Bestimmung des kürzesten Pfades von einem gegebenen Startknoten zu einem Zielknoten. In unserer Anwendung wollen wir die zeitschnellste Bus-/Bahnreise von einer Haltestelle zu einer anderen finden.

Der Algorithmus zur Ermittlung des kürzesten Pfades von A nach B arbeitet iterativ über zwei Mengen. Die Menge S enthält Knoten, für die der kürzeste Pfad von A bereits berechnet wurde. Die Menge $Q \subseteq \mathbb{N} \times Node \times [Node]$ beinhaltet insbesondere alle Knoten außerhalb S, die von irgend einem $n \in S$ durch Traversion *einer* Kante erreicht werden können (2. Komponente). Für diese Knoten ist zusätzlich der bisher ermittelte Pfad (3. K.) sowie dessen Länge (1. K.) enthalten. Dieser Pfad ist ggf. noch nicht der kürzestmögliche.

Initial setzen wir $S_0 = \{\}$ und $Q_0 = \{(0, A, [])\}$, d.h. der einzige bekannte Pfad von A nach A ist leer und hat die Länge 0. In jeder Iteration $i \geq 0$ wird nun ein Tripel $(l, n, p) \in Q_i$ ausgewählt, so dass l minimal ist, d.h. für alle $(l', n', p') \in Q_i$ ist $l \leq l'$. Ist $n \in S_i$, so fahren wir mit $S_{i+1} = S_i$ und $Q_{i+1} = Q_i \setminus \{(l, n, p)\}$ fort.[2] Ist $Q_i = \{\}$, so gibt es keinen Pfad von A nach B. Ist $n = B$ terminiert der Algorithmus und liefert $(l, reverse\ (n : p))$ als kürzesten Pfad inklusive seiner Länge. Andernfalls betrachten wir die Menge der Nachfolger N_i von n im Graphen. Die nächste Iteration wird dann durchgeführt mit

$$S_{i+1} = S_i \cup \{n\} \quad \text{und} \quad Q_{i+1} = (Q_i \setminus \{(l, n, p)\}) \cup \{(l + cost_G(n, n'), n', n : p) \mid n' \in N_i \setminus S\}. \qquad (1)$$

Hierbei liefert $cost_G(n, n')$ die Kosten für die (bei der obigen Anwendung immer existierende) Kante

[2] $X \setminus Y = \{x \in X \mid x \notin Y\}$, d.h. alle Elemente aus X, die nicht in Y enthalten sind.

zwischen n und n'. Anschaulich fügen wir also neue Kandidaten für die kürzesten Pfade von A zu den Knoten in $N_i \backslash S_i$ hinzu, und zwar diejenigen mit n als vorletztem Knoten im Pfad. Für n selbst ist der kürzeste Pfad von A gefunden als $(l, \mathit{reverse}\ (n : p))$.[3]

Beobachtung: Die Operationen auf Q_i (Einfügen und minimales Element entnehmen) entsprechen gerade denen, die Sie von Prioritätswarteschlangen (kurz: *Heaps*) kennen. Dies legt die Verwendung des folgenden Heap-Datentyps zur Realisierung der Q_i nahe, bei dem die Priorität gerade der Pfadlänge entspricht und die verwalteten Werte Paare aus Knoten und Pfaden sind:

```
type Queue = PriorityQueue Int (Node, [Node])
```

1. Implementieren Sie eine exportierte Funktion

   ```
   findRoute :: Network → Node → Node → (Int, [Node])
   ```

   die gegeben ein Liniennetz sowie eine Start- und eine Zielhaltestelle (der Einfachheit halber inklusive der Linienangabe, d.h. als `Node`) die kürzeste Fahrt berechnet. Die Rückgabe besteht hierbei aus der errechneten Pfadlänge und der Sequenz der befahrenen Stationen und erforderlichen Umstiege.

2. Definieren Sie (mittels **let** oder **where**) innerhalb von `findRoute` die Unterfunktion

   ```
   iterfind :: Queue → Set Node → (Int, [Node])
   ```

   welche die eigentliche Arbeit übernimmt, indem Sie den oben beschriebenen Algorithmus unter Verwendung eines Heap für die Darstellung der Kandidatenpfade in den Q_i, sowie einer Menge von `Node`s zur Verwaltung der Knoten mit bekannten kürzesten Pfaden realisiert.

3. Sie erhalten *zwei Bonuspunkte*, wenn Sie zusätzlich eine Funktion zur Verkürzung der Rückgabe von `findRoute` implementieren:

   ```
   abbrevRoute :: [Node] → [(Node, Node)]
   ```

   Diese Funktion soll einen Pfad auf Paare von Stationen reduzieren, an denen aus einer Linie ein- und ausgestiegen werden muss.

Testfälle

Für beide Aufgaben sind sinnvolle Testfälle zu entwerfen. Beispiele sind, dass (i) die hinzugefügten Stationen einer Linie sowie Umstiege auch anschließend zu den entsprechenden Kosten im Liniennetz vorhanden sind; (ii) dass für die vorgegebenen Linienpläne durch `makeNetwork` ein *zusammenhängender* Graph erzeugt wird; (iii) dass an den von `findRoute` gelieferten Übergangsstationen auch wirklich ein Umstieg möglich ist; oder (iv) dass der von `iterfind` berechnete kürzeste Pfad zumindest mal ein echter Pfad ist, d.h. dass über seine Knotensequenz Start und Ziel verbunden werden.

Gutes Gelingen!

[3] Gäbe es einen kürzeren Pfad p', so hätte dieser einen Knoten m als vorletzte Station, für den der kürzeste Pfad mit Länge l_m bereits ermittelt wurde, d.h. $m \in S_i$. Dann wäre aber in einer vorangehenden Iteration j ein Tripel $(l_m + \mathit{cost}_G(m,n), n, p')$ mit $l_m + \mathit{cost}_G(m,n) < l$ in Q_{j+1} eingefügt worden, so dass entweder l nicht minimal in Q_i wäre, oder $n \in S_i$ gelten müsste. Beides führt zum Widerspruch.