Universität Bremen

Praktische Informatik 3 WS 22/23

Christoph Lüth
Raphael Baass
Thomas Barkowsky
Tede von Knorre
Alexander Krug
Tarek Soliman

5. Übungsblatt

Ausgabe: 21.11.22
Abgabe: 27.11.22

Nachdem wir nun schon ein IT-Backbone (2. Übungsblatt) und eine Tabellenkalkulation (4. Übungsblatt) für ihren Bekannten gemacht haben, will dieser nun noch einen Routenplaner für sein neustes Geschäftsmodell: Ein Eis-Lieferdienst.

Google soll davon natürlich nichts mitbekommen, also muss wieder eine selbstgebaute Lösung her.

Um eine Karte der echten Welt zu modellieren ist ihnen als erfahrener Informatiker natürlich gleich ein Graph in den Kopf geschossen.

Schnell finden wir im Internet[1] ein Modul, welches Graphen mit Knoten- und Kantenmarkierungen mit folgenden Funktionen modelliert:

```
data Graph n e
empty :: Graph n e
addEdge, addBiEdge :: n → e → n → Graph n e → Graph n e
nodes :: Eq n ⇒ Graph n e → [n]
edges :: Graph n e → [(n,e,n)]
neighbours :: (Eq n) ⇒ n → Graph n e → [(e,n)]
```

`empty` erzeugt einen leeren Graphen. **`addEdge n1 e n2 g`** fügt eine Kante zum Graphen hinzu. Da die meisten Wege in beide Richtungen benutzbar sind, fügt **`addBiEdge n1 e n2 g`** fügt eine Kante in beide Richtungen hinzu. **`nodes g`** und **`edges g`** geben jeweils alle Knoten bzw. alle Kanten eines Graphen zurück; die Markierungen auf den Kanten sind die *Kosten*. **`neighbours n g`** gibt die Nachbarn eines Knoten zurück.

Ein *Pfad* besteht aus einer Liste von Kanten und den kumulierten Kosten dieses Pfades (d.h. die Summe der Kantenmarkierungen wenn dieser Pfad traversiert wird). Eine Funktion **`comparePath`** vergleicht Pfade nach diesen kumulierten Kosten:

```
data Path a b = Path [a] b

comparePath :: Ord b⇒ Path a b→ Path a b→ Ordering
```

Weil umgekehrte Funktionsanwendung |> wie in der Tabellenkalkulation zum Aufbau eines Testgraphen genutzt wird, ist diese auch vorhanden.

5.1 *Tiefensuche* — *3 Punkte*

Als erster Algorithmus kommt ihnen Tiefensuche in den Sinn. Da hier allerdings anders als in der Vorlesung der schnellste Pfad zum Ziel gefunden werden soll, müssen wir den Algorithmus aus der Vorlesung wie folgt anpassen:

1. Wir müssen wir bei **`trav`** nicht nur den ersten, sondern alle gefundenen Wege berechnen. Der schnellste ist dann einfach der erste der nach Länge sortierten Liste aller Pfade; wird kein Pfad gefunden geben wir **`Nothing`** zurück.
2. Ferner müssen wir bei der Traversion auch die kumulierten Kosten der Pfade berechnen.

Damit hätte **`trav`** die Signatur[2]

```
trav :: [(a, [a], b)]→ [Path a b]
```

wobei in der Liste immer die nächsten Kandidaten, der Pfad zu diesem Kandidaten (in umgekehrter Reihenfolge vom Start), und die kumulierten Kosten stehen.

Für die Suche müssen wir Knoten auf Gleichheit vergleichen können, und Kanten addieren sowie vergleichen können:

[1] Genauer gesagt in der Vorlage für dieses Übungsblatt.
[2] Aus technischen Gründen können wir die Signatur nicht direkt angeben.

```
depthFirstSearch :: (Eq a, Ord b, Num b)⇒ Graph a b → a → a → Maybe (Path a b)
```

Hinweis: Nutzen Sie die Funktionen **sortBy** (aus **Data.List**) und **comparePath**, um die Ergebnispfade nach den Kosten zu sortieren.

5.2 *Breitensuche* *3 Punkte*

Da bei einer Tiefensuche alle Pfade durchgegangen werden, sollten wir auch noch einmal Breitensuche ausprobieren. Die Breitensuche ist genau wie die Tiefensuche, allerdings werden hier die neuen Kandidaten hinten und nicht vorne angehängt.

```
breadthFirstSearch :: (Eq a, Ord b, Num b)⇒ Graph a b → a → a → Maybe (Path a b)
```

5.3 *Dijkstra* *4 Punkte*

Die Performance beider Algorithmen ist nicht zufriedenstellend. Das Eis kommt nur als Vanille- oder Schokosauce an. Um die Firma zu retten will ihr Bekannter die Hälfte der Belegschaft entlassen und den Rest zu achtzig Stunden Arbeit pro Woche verpflichten, weil "so machen das die *big guys*", aber Sie haben eine bessere Idee.

Wir brauchen einen Suchalgorithmus, bei dem der erste gefundene Pfad auch der kürzeste ist. Dazu muss die Funktion **trav** die Invariante haben, dass der Pfad vom aktuellen Kandidaten zum Start immer der bisher kürzeste ist. Dieser Algorithmus ist der Dijkstra-Algorithmus, und die Art und Weise, wie die Kandidaten und Pfade geordnet sind, nennt man eine *Priority Queue*.[3]

Wir implementieren zuerst die Priority Queues in **PQueue.hs**. Diese sind über *drei* Typen parametrisiert, nämlich die Knoten, die kumulierten Kosten und eine beliebige „Nutzlast" (welches später die Pfade sind):

```
data PQueue a b c = PQueue { queue :: [(a, b, c)] }
```

Die Invariante ist, dass die Liste im zweiten Argument nach aufsteigender Größe sortiert ist — das werden später die kumulierten Kosten — und keine zwei Einträge enthält, die im ersten Argument gleich sind — das werden später die Kandidaten. Wir benötigen drei Funktionen darauf:

```
emptyQ :: PQueue a b c -- Leere Priority Queue
deq :: PQueue a b c → Maybe (PQueue a b c, (a, b, c))
enq :: (Eq a, Ord b)⇒ PQueue a b c→ (a, b, c)→ PQueue a b c
```

deq liefert den Kopf und Rest der Queue, wenn diese nicht leer ist (d.h. das *Minimum* der Liste). **enq** fügt ein neues Tripel **(a, b, c)** zu der Liste hinzu, so dass die Invariante erhalten bleibt. Konkret müssen wir also den ersten Platz in der Liste finden, wo **b** kleiner ist als der Nachfolger, und alle anderen Tripel mit dem gleichen **a** an erster Stelle aus der Restliste entfernen; gibt es schon einen Eintrag in der Liste mit dem gleichen **a** an erster Stelle, aber kleinerer zweiten Komponente, wird gar nichts eingefügt.

Betrachten wir die einige Beispiele:

```
q1 = emptyQ |> enq ("B", 7, ()) |> enq ("A", 3, ())
```

- Fügen wir jetzt **("C", 5, ())** zu **q1** hinzu, sollte die Schlange folgende Elemente enthalten:

  ```
  enq ("C", 5, ()) q1 ⇝ [("A", 3, ()), ("C", 5, ()), ("B", 7, ())]
  ```

- Fügen wir zu dieser Schlange **q2** noch **("B", 4, ())** hinzu, entfällt dafür das letzte Element der Schlange:

  ```
  enq ("B", 4, ()) q2 ⇝ [("A", 3, ()), ("B", 4, ()), ("C", 5, ())]
  ```

- Wird zu dieser Schlange **q3** wieder **("B", 6, ())** hinzugefügt, bleibt die Schlange wie sie ist, weil schon ein **"B"** mit niedrigerer Priorität enthalten ist:

  ```
  enq ("B", 6, ()) q3 ⇝ q3,
  ```

[3] Obwohl sie nicht die für eine Schlange typische FIFO implementiert.

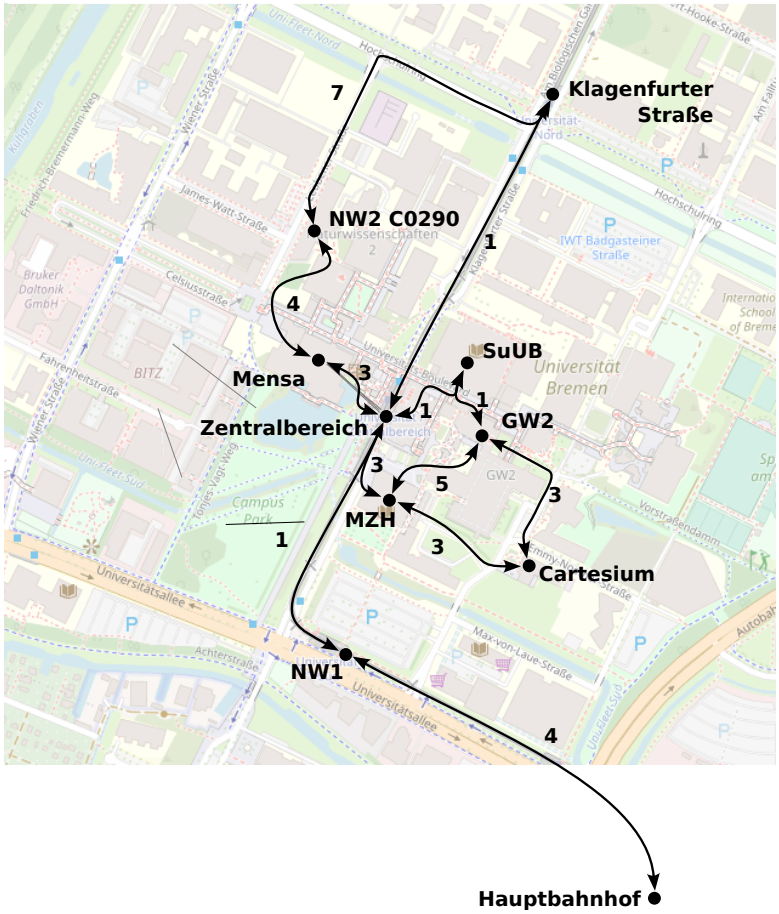

Abbildung 1: Karte des Campus der Universität Bremen. (Die Entfernungen sind kürzer, wenn man die Straßenbahn nutzen kann.)

Die Priority Queue ersetzt die Liste in der Breiten/Tiefensuche. Statt des Kopfes der Listen ist das kleinste Element (Resultat von **`deq`**) der nächste Kandidat, und die neuen Kandidaten werden mit **`enq`** in die Liste eingefügt. Die Traversierungsfunktion hätte die Signatur

```
trav :: PQueue a b [a] → Maybe (Path a b)
```

Damit implementieren wir die Dijkstra-Suche:

```
dijkstra :: (Eq a, Ord b, Num b)⇒ Graph a b → a → a → Maybe (Path a b)
```

Jetzt kommt das Eis rechtzeitig an (bitte ins MZH, Raum 4186).

Als Beispiel betrachten wir den Campus der Universität Bremen, abstrahiert zum Graphen in Abb. 1. Der Graph ist in **`Karten.hs`** als **`karte1`** vordefiniert. Alle drei Suchverfahren sollten hier folgende kürzeste Wege finden (wobei das letzte Beispiel den Umgang mit nicht-existenten Knoten zeigt):

```
search karte1 "MZH" "NW2␣C0290"  ⇝  Just (Path ["MZH", "Zentralbereich", "Mensa", "NW2␣C0290"] 10)
search karte1 "Cartesium" "Mensa"  ⇝  Just (Path ["Cartesium", "GW2", "SuUB", "Zentralbereich", "Mensa"] 8)
search karte1 "Hauptbahnhof" "SuUB"  ⇝  Just (Path ["Hauptbahnhof","NW1","Zentralbereich","SuUB"] 6)
search karte1 "MZH" "Nirgdendwo"  ⇝  Nothing
```