Übungsblatt 4

Erneut verwenden wie in dieser Übung die GTFS-Daten, welche Sie im Übungs-Repository finden (https://gtfs.org/schedule/reference). Die Daten enthalten Informationen über die S- und U-Bahnlinien Berlins. In der automatischen Abgabeprüfung werden die Regionalbahnen Brandenburgs und Tramlinien Potsdams verwendet.

Für mehr Informationen über die Datenstruktur schauen Sie auch in das Übungsblatt 3, welches ein Diagramm mit den verwendeten Daten und deren Relationen zueinander enthält.

a) Bahnhöfe und Umstiege

In den Daten finden Sie in der Datei `stops.txt` und in Ihrer Network-Klasse in dem Attribut der Haltestellen (`stops`) Informationen zu allen Haltestellen im S- und U-Bahnnetz Berlins.
Wenn Sie die Daten genauer anschauen sehen Sie, dass nicht jeder Bahnhof einfach als ein Eintrag in der Datei `stops.txt` erfasst ist, sondern das einzelne Gleise und Bahnhöfe selbst in der Datei vorkommen.

Bspw. finden Sie folgende Zeilen für den U-Bahnhof Amrumer Straße in den Daten aus dem Repository (Daten gekürzt):

stop_id	stop_name	location_type	parent_station
de:11000:900009101	U Amrumer Str. (Berlin)	1	*leer*
de:11000:900009101::1	U Amrumer Str. (Berlin)	0	de:11000:900009101
de:11000:900009101::2	U Amrumer Str. (Berlin)	0	de:11000:900009101

Ist der `location_type` eines Stops auf den Wert `1` gesetzt, so handelt es sich um einen Bahnhof und der Wert im Feld `parent_station` ist **leer**. Besitzt der `location_type` den Wert `0`, so handelt es sich um ein einzelnes Gleis und der dazugehörige Bahnhof ist mit seiner ID im Feld `parent_station` erfasst (https://gtfs.org/documentation/schedule/reference/#stopstxt).

Um effektiv Routen zwischen Bahnhöfen berechnen zu können benötigen wir eine Methode, welche alle Gleise eines Bahnhofs ermittelt, auf die wir umsteigen können. Die Funktion soll dabei so "klug" sein, dass Sie bei der Angabe einer ID eines Bahnhofs **und** bei der Angabe einer ID eines einzelnen Gleises einen Vektor mit allen zusammengehörigen Elementen aus `stops.txt` zurückgibt.

Bspw. geben die Aufrufe von folgenden Methoden **immer** die drei Stop-Datenstrukturen zurück, die oben in der Tabelle aufgezählt sind:

```
getStopsForTransfer("de:11000:900009101") // Übergabe der ID des Bahnhofs Amrumer Str.
getStopsForTransfer("de:11000:900009101::1") // Übergabe der ID von Gleis 1 Amrumer Str.
getStopsForTransfer("de:11000:900009101::2") // Übergabe der ID von Gleis 2 Amrumer Str.
```

1) Erweitern Sie die Klasse `Network` um eine Methode mit der Sichtbarkeit `public` und folgender Signatur:

   ```
   std::vector<bht::Stop> getStopsForTransfer(const std::string& stopId)
   ```

2) Implementieren Sie die Methode. Als Parameter wird die ID eines Stops übergeben (die ID ist korrekt und muss nicht geprüft werden). Als Ergebnis wird ein Vector zurückgegeben, welche neben dem Stop mit der gegebenen ID auch alle anderen Haltestellen des Bahnhofs enthält.
3) Die Methode funktioniert korrekt, wenn die ID eines Bahnhofs übergeben wird.

4) Die Methode funktioniert korrekt, wenn die ID eines Gleises übergeben wird.

b) Routen berechnen

Die Network-Klasse beinhaltet alle Linien (`routes`), Fahrten (`trips`), Haltestellen (`stops`) und Haltestellen der Fahrten (`stop_times`). Ziel dieser Teilaufgabe ist es, einen **Weg** von einer Haltestelle zu einer anderen zu berechnen. In dieser Übung können Sie die Abfahrt- und Ankunftszeiten noch ignorieren.

Um eine Route berechnen zu können müssen Sie die Haltestellen und Fahrten als gerichteten Graphen interpretieren (https://de.wikipedia.org/wiki/Gerichteter_Graph). Die Haltestellen sind dabei die Knoten und die Fahrten die gerichteten Kanten, welche die Knoten miteinander verbinden. (*Denken Sie daran, dass ein Knoten in Ihrem Graphen immer ein Bahnhof ist, also auch Umstiege zwischen den Gleisen möglich sind wie in Aufgabe a) implementiert.*)

In diesem gerichteten Graphen kann dann mittels Algorithmen wie dem Dijkstra-Algorithmus (https://de.wikipedia.org/wiki/Dijkstra-Algorithmus) oder dem A*-Algorithmus (https://de.wikipedia.org/wiki/A*-Algorithmus) der kürzeste Weg gesucht werden, um von einem Knoten zu einem anderen zu gelangen.

1) Implementieren Sie eine Methode um alle benachbarten Haltestellen zu einer Haltestelle zu suchen. Zwei Haltestellen gelten als benachbart, wenn diese durch einen Fahrt (`trip`) miteinander verbunden werden. Ein Umstieg auf ein anderes Gleis soll dabei über diese Methode möglich gemacht werden, indem nicht nur die nächsten Haltestellen der Linie als Nachbarn zurückgegeben werden sondern auch mögliche andere Gleise zum Umsteigen.

   ```
   std::unordered_set<std::string> getNeighbors(const std::string& stopId)
   ```

2) Erweitern Sie die Klasse Network um eine Methode mit der Sichtbarkeit public und folgender Signatur:

   ```
   std::vector<bht::Stop> getTravelPath(const std::string& fromStopId, const
   std::string& toStopId)
   ```

3) Implementieren Sie die Methode. Als Parameter werden die IDs von zwei Haltestellen übergeben und Ihre Methode berechnet einen kürzesten Weg um von der Start-Haltestelle zur End-Haltestelle zu gelangen unter Berücksichtigung der verfügbaren Fahrten und Linien.
   (Sie können davon ausgehen, dass die übergebenen IDs existieren, jedoch kann es vorkommen, dass kein Weg existiert.)
4) Das Ergebnis der Methode ist ein Vektor mit allen Haltestellen, die auf der Strecke durchlaufen werden.

c) Verwendung von Iteratoren

Im Repository der Übung finden Sie die Dateien `scheduled_trip.h` und `scheduled_trip.cpp`. In dieser Klasse soll eine Fahrt gespeichert werden, deren Haltestellen mit einem **Iterator** durchlaufen werden können.

Dazu ist als innere Klasse in `NetworkScheduledTrip` die Klasse `iterator` definiert, welche die erforderlichen Methoden für einen **bidirektionalen** Iterator bereitstellt.

Ihre Aufgabe in dieser Teilaufgabe ist es die `Network`-Klasse zu erweitern um die Erzeugung von `NetworkScheduledTrip`-Objekten sowie die Implementierung der fehlenden Methoden aus `NetworkScheduledTrip`.

1) Erweitern Sie Ihre `Network`-Klasse um eine public-Methode mit der Signatur

   ```
   NetworkScheduledTrip getScheduledTrip(const std::string& tripId) const;
   ```

2) Die Methode erhält als Parameter die ID einer Fahrt (`trip`) und gibt als Ergebnis ein Objekt vom Typ `NetworkScheduledTrip` zurück.
3) Implementieren Sie die fehlenden Methoden in `NetworkScheduledTrip` und fügen Sie ggf. eigene Attribute, Methoden, Konstruktoren hinzu, wenn Sie diese benötigen.

Mit der `main()`-Methode in der Datei `main.cpp` im Repository können Sie Ihre Implementierung testen. Die Ausgabe des Programms muss wie folgt aussehen:

```
Stop 0: S Erkner Bhf
Stop 1: S Wilhelmshagen (Berlin)
Stop 2: S Rahnsdorf (Berlin)
```

d) Überarbeitung des Navigationsalgorithmus (freiwillig)

Überprüfen Sie Ihren Algorithmus aus a) ob Sie diesen mit der neuen Klasse und der Funktionalität Iteratoren verwenden zu können optimieren können.

e) Abgabe und automatische Auswertung

Die automatische Auswertung prüft die Methoden aus den Aufgaben a) - c). Sie finden im Repository der Übungsaufgabe erneut die Datei `tester.cpp`. Diese Datei enthält wieder die Testfälle für Ihre Anwendung. Diese können Sie sich ansehen und selbst prüfen ob Ihre Anwendung diese Testfälle erfüllt. Die Tests werden mit dem Googletest-Framework realisiert (https://github.com/google/googletest).

Die Datei `tester.cpp` darf von Ihnen nicht verändert werden. Die Integrität der Datei wird bei der automatischen Auswertung per Prüfsumme verifiziert.

Die folgenden Anforderungen werden in der Abgabe geprüft:

1) Die Datei `tester.cpp` ist unverändert.
2) In Ihrem Repository ist die Klasse `Network` in der Header-Datei `network.h` definiert (alles klein geschrieben, genaue Schreibweise beachten, Datei ist direkt im Repository vorhanden und nicht in einer Unterverzeichnis).
3) Ihre Abgabe enthält wieder ein `Makefile` mit dem Target `autotest`, welche Die Datei `tester.cpp` mit den benötigten Quellcode-Dateien übersetzt und die Datei `test_runner` erzeugt. Der Befehl zum Übersetzen Ihrer Anwendung sollte wieder beginnen mit dem unten genannten Kommando gefolgt von Ihren C++-Quelldateien (außer denen mit der `main`-Methode und dem Qt-Fenster)

   ```
   g++ -I. -I/usr/local/include -std=c++17 -o test_runner /usr/local/lib/
   libgtest_main.a /usr/local/lib/libgtest.a tester.cpp <Ihre CPP-Dateien...>
   ```

4) Die Unit-Tests für die Aufgaben a) - c) können ohne Fehler ausgeführt werden.
5) Eine Plagiatsprüfung findet keine Treffer.