Institut für Theoretische Informatik
Lehrstuhl Prof. Dr. D. Wagner

Dipl.-Inform. Julian Dibbelt
Dipl.-Inform. Thomas Pajor

# Übungsblatt 2

Praktikum Algorithm Engineering – Routenplanung (WS 11/12)

**Ausgabe** 08. November 2011
**Abgabe** 22. November 2011

## Problem 1: Dijkstra's Algorithmus

In dieser Aufgabe soll eine erste Version von Dijkstra's Algorithmus implementiert werden. Wir gehen dabei davon aus, dass Sie das Übungsblatt 1 erfolgreich bearbeitet haben, das heißt, dass Sie eine funktionsfähige Entwicklungsumgebung und Zugang zum SVN eingerichtet haben.

Gehen Sie für diese Aufgabe in folgenden Schritten vor.

(a) Führen Sie ein SVN-*Update* auf dem Framework von Übungsblatt 1 aus. Es erscheinen einige neue Dateien. Unter anderem eine Implementierung von einem $k$-ären Heap (`kheap.h`), sowie Code-Stummel von Dijkstra's Algorithmus (`dijkstra.h`) und `blatt2.cpp`.

Laden Sie sich außerdem die neuen Graphen von der Praktikumsseite herunter (diese enthalten nun auch Rückwärtskanten).

(b) Machen Sie sich mit der Funktionsweise von Dijkstra's Algorithmus vertraut. Sie können dazu die Folien der Vorlesung *Algorithmen für Routenplanung* aus dem vergangenen Sommersemester studieren: `http://i11www.iti.uni-karlsruhe.de/_media/teaching/sommer2011/routenplanung/vorlesung2.pdf`.

(c) Öffnen Sie die Datei `algorithms/dijkstra.h`. Diese enthält den Rahmen für Dijkstra's Algorithmus. Implementieren Sie die Methode `run`. Nach Ausführung des Algorithmus soll der Wert `node_labels[t].distance` die korrekte Distanz des Zielknoten $t$ repräsentieren.

*Hinweis:* Sie benötigen dazu die Methoden `extractMin` und `update` der Priority Queue (siehe `data_structures/pqueues/kheap.h`. Achten Sie außerdem darauf, nicht über Rückwärtskanten zu iterieren!

(d) Implementieren Sie Statistiken für die Größe des Suchraumes und die Anzahl relaxierter Kanten in Ihren Algorithmus. Benutzen Sie hierfür die Variablen `num_scanned_nodes` und `num_relaxed_edges` der Klasse `dijkstra`.

(e) Kompilieren Sie `blatt2.cpp` durch Aufruf von `make blatt2`, und lassen Sie Dijkstra's Algorithmus auf den Graphen Florida und Bay Area laufen. Welche Laufzeit beobachten Sie? Wie groß ist der Suchraum und die Anzahl relaxiert Kanten im Schnitt?

(f) Implementieren Sie das *Stoppkriterium* von Dijkstra's Algorithmus. Vergleichen Sie die Statistiken mit denen von Aufgabe (e). Welche Beschleunigung beobachten Sie?

*Hinweis:* Die Suche kann abgebrochen werden, sobald der Zielknoten aus der Priority Queue entfernt wurde.

**Problem 2:** Bidirektionale Suche

In dieser Aufgabe soll die Implementierung von Dijkstra's Algorithmus zu einer bidirektionalen Suche erweitert werden. Bei der bidirektionalen werden gleichzeitig von $s$ und $t$ Dijkstra's Algorithmus jeweils auf dem Vorwärts- bzw. Rückwärtsgraphen ausgeführt, bis sich die Suchräume der beiden Suchen treffen. Weitere Informationen finden Sie unter `http://i11www.iti.uni-karlsruhe.de/_media/teaching/sommer2011/routenplanung/vorlesung2.pdf`.

Gehen Sie in folgenden Schritten vor.

(a) Kopieren Sie Ihre Implementierung von `dijkstra.h` zu `bidijkstra.h`, und benennen Sie die Klasse `dijkstra` in `bidijkstra` um.

(b) Bereiten Sie den Algorithmus so vor, dass Sie zwei Suchen durchführen können.

Erweitern Sie dazu die Struct `node_label` um eine Variable die die Distanz zu $t$ für die Rückwärtssuche speichert, und fügen Sie eine weitere Instanz der Priority Queue hinzu. Fügen Sie außerdem eine Variable $\mu$ hinzu, die die Distanz des bislang kürzesten gefundenen $s$-$t$-Weges speichert.

(c) Implementieren Sie nun die bidirektionale Suche mit der Abwechslungsstrategie *Alternate* (die Suchen wechseln sich nach jeder Iteration ab). Der Algorithmus soll stoppen sobald ein Knoten von beiden Suchen abgearbeitet wurde. Speichern Sie sich die Distanz des kürzesten zusammengesetzten $s$-$t$-Weges.

*Hinweis:* Der kürzeste Weg verläuft nicht notwendigerweise durch den Knoten der für das Stoppen des Algorithmus verantwortlich war.

(d) Überprüfen Sie die Korrektheit Ihres Algorithmus indem Sie die Prüfsumme (Summe der Distanzen aller gefundenen kürzesten Wege) mit der von Dijkstra's Algorithmus aus Aufgabe 1 vergleichen.

(e) Implementieren Sie folgende alternative Abwechslungsstrategie *Minimum Key*: Es kommt stets die Suche zum Zug, dessen Element mit minimalem Schlüssel in der Priority Queue kleiner ist. Benutzen Sie dazu die Methode `minKey` der Priority Queue.

Welche Strategie ist schneller? Welche hat einen kleineren Suchraum?

(f) Implementieren Sie das folgende, schärfere Abbruchkriterium. Der Algorithmus kann abgebrochen werden, sobald $\texttt{minKey}\overrightarrow{Q} + \texttt{minKey}\overleftarrow{Q} \geq \mu$ gilt.

Wie groß ist der Speedup den Sie durch dieses Abbruchkriterium erzielen?