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

Moritz Baum, Julian Dibbelt,
Ben Strasser

Übungsblatt 2

Praktikum Algorithm Engineering – Routenplanung (WS 13/14)

Ausgabe 30. Oktober 2013
Abgabe 6. November 2013

Aufgabe 1: Code Update

Im Git-Repository liegt neuer Code. Erstellen Sie einen neuen Ordner und laden Sie die neue Version des Codes herunter, um zu verhindern, dass ihre Veränderungen am Code des letzten Blatts zu Konflikten führen.

Aufgabe 2: Dijkstras Algorithmus

Die Datei `dijkstra.h` enthält eine unvollständig Implementierung von Dijkstras Algorithmus. Vervollständigen Sie diese. Füllen sie dazu alle Funktionen aus, die mit `// Implement me` markiert sind. Um die Korrektheit zu testen, benutzen wir die zufälligen Testanfragen in den `*.q`-Dateien. Sie können ihre Implementierung testen indem, Sie folgenden Befehl ausführen:

```
./check_dijkstra_against_distance_test_queries karlsruhe.gr karlsruhe.100.q
```

Die 100 gibt an wie viele Anfragen in der Datei enthalten sind. Je mehr Anfrage, je höher die Zuversicht, dass die Implementierung korrekt ist. Das Problem ist aber, dass mehr Anfrage mehr Rechenzeit benötigen. Der Befehl zeigt Ihnen auch an, wie schnell die Anfragen im Durchschnitt sind. Für eine aussagekräftige Messung benötigt man mehr als 100 Anfragen. Auf Europa werden wir dies aber aus Zeitgründen nicht tun.

Die Anfragen haben zufällige Start- und Zielknoten, die nach einer Gleichverteilung gezogen werden. Sind die resultierende Pfade im Mittel eher kurz oder lang? Erklären Sie, warum diese Anfragen in einem Navigationssystem eher selten vorkommen. Erklären Sie, warum die gemessen Laufzeiten dennoch Aussagekraft haben.

Im Durchschnitt braucht eine Anfrage 46s auf meinem Rechner auf Europa. Für ein Webserver mit vielen Anfragen sind mehrere Sekunden pro Anfrage zu langsam. Der Rest dieses und des nächsten Aufgabenblatts werden sich damit beschäftigen diese Anfragen zu beschleunigen. Auf dem nächsten Blatt werden wir ein Verfahren verwenden, dass Anfragen unter einer Millisekunde beantwortet.

Aufgabe 3: Automatische Optimierungen

Öffnen Sie die Datei `Makefile`. Diese enthält die folgende Zeile:

```
CFLAGS = -Wall -std=c++0x -D_GLIBCXX_DEBUG
```

`CFLAGS` enthält die Einstellungen mit denen der Code übersetzt wird. Eine Auswahl von möglichen Optionen sind:

Option	Beschreibung
`-Wall`	Warne wenn der Code komisch aussieht.
`-std=c++0x`	Benutze C++11.
`-D_GLIBCXX_DEBUG`	Teste ob die STL richtig verwendet wurde. Ein Beispiel wäre der Indextest im Zugriffsoperator von `std::vector`. Diese Tests benötigen viel Rechenzeit.
`-DNDEBUG`	Schalte `assert` aus.
`-O3`	Führe automatisch Mikrooptimierungen während des Übersetzens durch.

Verändern Sie `CFLAGS` und finden Sie heraus, welche Kombination das schnellste Programm ergibt.

Mit den richtigen Optionen braucht eine Anfrage 1.9s auf meinem Rechner auf Europa.

Warnung: Es kann sein, dass Sie die `.o`-Dateien löschen müssen um eine neue Übersetzung zu erzwingen. Alternativ können sie auch `make -B` aufrufen.

Aufgabe 4: Zeitstempelflaggen

Ein häufiger Beschleunigungsansatz besteht daraus, die Anzahl an besuchten Knoten dramatisch zu reduzieren. Dies erlaubt Laufzeiten die sublinear in der Anzahl an Knoten sind. Dies bringt aber nur etwas, wenn die Initialisierung von Dijkstras Algorithmus nicht selber linear ist. Die aktuell verwendete Implementierung der `visited_flags` verwendet die `BoolFlags` aus `bool_flags.h`. Dieser elementare Ansatz benötigt $\Theta(n)$ Rechenzeit, um alle Werte auf `false` zu setzen, wobei n die Anzahl an Knoten ist. Er ist demnach linear. Leider ist er aber in der Initialisierung von Dijkstras Algorithmus nötig. Allerdings sind die sich hinterm Θ verbergende Konstanten sehr klein, was den Ansatz für "kleine" n kompetitiv macht.

Um eine sublineare Initialisierung zu ermöglichen wurden Zeitstempelflaggen entworten. Die Idee ist, dass jede Anfrage ihren eigenen Zeitstempel bekommt. Für jeden Knoten speichert man, wann er zuletzt besucht wurde. Ein Knoten wurde genau dann besucht, wenn sein Zeitstempel mit dem der Anfrage übereinstimmt. Dies erlaubt es, in konstanter Zeit alle Werte auf `false` zu setzen.

In der Datei `timestamp_flags.h` befindet sich eine unvollständige Implementierung mit der selben Schnittstelle als `bool_flags.h`. Vervollständigen Sie die Implementierung. Passen Sie `check_dijkstra_against_distance_test_queries.cpp` so an, dass die Zeitstempelflaggen verwendet werden. Ist der Code schneller geworden? Wie sieht es aus, wenn Start- und Zielknoten geographisch nahe beisammen sind?

Auf meinem Rechner braucht eine Anfrage 2.1s auf Europa.

Randfall: Funktioniert ihre Implementierung auch nach 2^{16} Anfragen noch?

Aufgabe 5: Bidirektionale Variante von Dijkstras Algorithmus

Eine weitere Optimierung besteht darin, simultan vom Start- und vom Zielknoten aus Suchen zu starten. Man kann abbrechen, sobald sich die Suchräume treffen. Eine unvollständige Implementierung finden Sie in `bidirectional_dijkstra.h`. Vervollständigen Sie diese. Verwenden Sie `check_bidirectional_dijkstra_against_distance_test_queries.cpp` zum testen. Was für eine Beschleunigung beobachten Sie? Braucht die bidirektionale Variante mehr Speicher als die unidirektionale?

Ein bidirektionaler Dijkstra braucht auf meinem Rechner 1.2s auf Europa mit `BoolFlags` und 1.3s mit `TimestampFlags`.

Aufgabe 6: Pfade Anzeigen

Überprüfen Sie, dass ihre Implementierung nicht nur korrekte Distanzen berechnet, sondern auch die Pfade selbst korrekt berechnet. Passen sie dazu `show_path.cpp` so an, dass er ihre Implementierung der bidirektionalen Variante verwendet. Schauen Sie sich einige Pfade in Google Earth an, um offensichtliche Fehler zu erkennen.