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

Moritz Baum, Valentin Buchhold,
Ben Strasser, Tobias Zündorf

Übungsblatt

Praktikum Algorithm Engineering – Routenplanung (WS 16/17)

Ausgabe Mittwoch, den 19. Oktober 2016
Abgabe Donnerstag, den 17. November 2016, 8:00 morgens

Allgemeines

Mit Ihrem Rechneraccount können Sie sich im Poolraum an den Rechnern anmelden. Alternativ ist der Zugriff per SSH möglich. Um von Außen ins Lehrstuhlnetwerk zu kommen verbindet man sich zu `i11ssh.iti.kit.edu`. Auf dieser Maschine soll aber nicht gerechnet werden. Verbinden Sie sich deswegen zu einem der Poolraumrechner weiter. Die Befehle dazu sehen wie folgt aus:

```
ssh username@i11ssh.iti.kit.edu
ssh i11poolX.iti.kit.edu
```

"X" ist dabei die Nummer eines Poolraumrechners. Die Zahlen gehen von 1 bis 12. Falls der SSH-Zugang nicht funktioniert kann es sein, dass der entsprechende Rechner ausgeschaltet wurde. Versuchen sie dann einen weiteren. Ehe Sie Berechnungen durchführen, vergewissern Sie sich bitte per `htop`, dass der Rechner nicht bereits von jemand anderem genutzt wird. Falls er bereits genutzt wird, verwenden Sie bitte einen anderen Poolraumrechner.

Das Erste was Sie tun müssen ist ihr Standardpasswort ändern. Führen Sie dazu `passwd` auf einem der Poolraumrechner aus.

Um Dateien per SSH zu übertragen verwenden Sie `scp` wie folgt:

```
scp von_datei_daheim username@i11ssh.iti.kit.edu:nach_datei_im_poolraum_home
scp username@i11ssh.iti.kit.edu:von_datei_im_poolraum_home nach_datei_daheim
```

Unter `/misc/praktikum/graph` liegen diverse fürs Praktikum relevante Eingabedateien. Bitte verwenden Sie diese nur im Rahmen des Praktikums. Für die Codeverwaltung wird ein GIT-Repository eingesetzt. Die Abgabe des Übungsblatts erfolgt auch über das Repository.

Git

Um auf Ihr GIT-Repository zuzugreifen, führen Sie auf ihrem lokalen Rechner den folgenden Befehl aus.

```
git clone https://username@i11git.iti.kit.edu/git/Praktika/Routenplanung/Teilnehmer/username
```

Dies erstellt ein lokales Verzeichnis `username`. Dieses enthält alle Dateien die im GIT-Repository sind. Die Dateien in diesem Verzeichnis können Sie verändern und auch neue erzeugen. Damit die Veränderungen allerdings auf unserem Server sichtbar werden müssen Sie die veränderten Dateien markieren. Dies tun sie wie folgt:

```
git add file1.cpp
git add file1.h
```

Anschließend bündeln sie die Veränderungen mit einem Commit wie folgt:

```
git commit -m "Created file1"
```

Dieser Commit existiert allerdings anfangs nur bei Ihnen lokal. Um ihn auf den Server zu kopieren führen Sie den folgenden Befehl durch.

```
git push
```

Wenn Sie nun `file1.cpp` ändern wollen dann führen Sie die Modifikationen in ihrer lokalen Kopie der Datei durch. Anschließend müssen sie GIT mitteilen, dass Sie diese Datei verändert haben, einen neuen Commit erstellen und diesen auf den Server kopieren. Dies geht wie folgt:

```
git add file1.cpp
git commit -m "Changed file1.cpp"
git push
```

Weitere Informationen zu GIT finden sie unter folgendem Link:

```
https://rogerdudler.github.io/git-guide/index.de.html
```

Wichtig: Bitte packen Sie keine Dateien mit mehr als 100MB in ein GIT-Repository.

Code

In Ihrem GIT-Repository finden Sie im `code` Unterverzeichnis eine kleine Codebasis die Sie zur Bearbeitung des Übungsblatt verwenden sollen. Der Ihnen zur Verfügung gestellt Code besteht aus mehreren Dateien: `constants.h`, `timer.h`, `id_queue.h`, `vector_io.h`, `decode_vector.cpp`, `encode_vector.cpp`, `compare_vector.cpp` und `compile.sh`. In `constants.h` werden zwei oft verwendete Konstanten definiert: `invalid_id` und `inf_weight`. Erstere gibt eine ungültige ID an und letztere stellt eine unendliche Länge dar. `inf_weight` ist so gewählt, dass die Konstante verdoppelt werden kann, ohne einen Überlauf zu verursachen, d.h., der Ausdruck `inf_weight < inf_weight+inf_weight` ist unproblematisch. In der Datei `timer.h` gibt es eine Funktion die die aktuelle Zeit misst[1]. Diese kann verwendet werden um die Laufzeit ihres Codes zu messen. Die Datei `id_queue.h` enthält eine Prioritätswarteschlange[2]. Die restlichen Dateien dienen dem Einlesen und der Ausgabe von Daten. Jede Datendatei ist das binäre Abbild eines `std::vector`s im Speicher, d.h., ein Vektor von 100 `int`s wird in einer Datei gespeichert die genau 400 Byte lang[3] ist. In `vector_io.h` werden die Funktionen load_vector und save_vector zur Verfügung gestellt. Diese können Sie wie folgt verwenden:

```
vector<unsigned> head = load_vector<unsigned>("arc_head_file_name");
vector<float> lat = load_vector<float>("node_latitude_file_name");
save_vector("my_new_file_name", head);
```

Die restlichen Dateien stellen Hilfsprogramme dar. `encode_vector` und `decode_vector` konvertieren Vektoren von und zu textuellen Darstellungen. Das Programm `compare_vector` vergleicht ob zwei Vektoren identisch sind und wenn sie es nicht sind gibt es eine Übersicht über die Unterschiede. Die Datei `compile.sh` ist ein Shellskript das die drei Programme übersetzt. Erweitern Sie `compile.sh` so, dass es auch die von Ihnen erstellten Programme übersetzt.

[1] Alternativ können Sie die Funktionen aus dem <chrono>-Header verwenden.

[2] std::priority_queue ist problematisch für unseren Anwendungsfall, da sie keine decrease_key Operation besitzt

[3] Wir gehen stets davon aus, dass ein int 32 Bit hat.

Graphen

Knoten und Kanten werden durch numerische IDs identifiziert, die von 0 bis $n-1$ bzw. $m-1$ gehen, wobei n die Anzahl an Knoten und m die Anzahl an gerichteten Kanten ist. Wir speichern gewichtete und gerichtete Graphen in einer Ajdazenzarraydarstellung. Ein gerichteter und gewichteter Graph besteht aus 3 Vektoren. Diese heißen `first_out`, `head` und `weight`. Um über die ausgehenden Kanten eines Knoten zu iterieren können Sie den folgenden Code verwenden:

```
vector<unsigned> first_out = load_vector<unsigned>("first_out_file_name");
vector<unsigned> head = load_vector<unsigned>("head_file_name");
vector<unsigned> weight = load_vector<unsigned>("weight_file_name");

unsigned my_node = 42;
for(unsigned out_arc = first_out[my_node]; out_arc < first_out[my_node+1]; ++out_arc){
    cout<< "There is an arc from " << my_node
        << " to " << head[out_arc]
        << " with weight " << weight[out_arc]
        << endl;
}
```

Hinweis: `head` und `weight` haben so viel Elemente wie es Kanten gibt. `first_out` hat ein Element mehr als es Knoten gibt. Das erste Element von `first_out` ist immer 0 und das letzte ist die Anzahl an Kanten. Für alle Graphen gibt es zwei unterschiedliche Kantengewichte: Reisezeit und Reiselänge. Des Weiteren gibt es für manche Graphen zusätzliche für jeden Knoten die geographische Position. Diese wird als zwei `float` Vektoren abgespeichert die für jeden Knoten den Längen- und Breitengrad angeben.

Im Verzeichnis `/misc/praktikum/graph` liegen die Daten von mehreren Graphen in diesem Format. Manche dienen nur zu Testzwecken während andere zur Aufgabenbewertung verwendet werden. Die Testgraphen entsprechen ganz grob Stupferich, Karlsruhe&Umgebung, Deutschland&Umgebung und (West-)Europa. Die Aufgabengraphen haben die Größe des Deutschlandgraphen.

Achtung: Der Europagraph könnte zu groß sein für den Hauptspeicher von manchen Poolraumrechnern.

Allgemeines zu den Aufgaben

Zu jeder Aufgabe sollen Sie den Quellcode ihrer Programme und die berechneten Lösungen abgeben. Der Quellcode soll durch das Ausführen von `./compile.sh` auf einem der Poolraumrechner übersetzt werden können. Auf den Poolraumrechner ist ein GCC 4.8.5 installiert. Dieser unterstützt C++11 quasi vollständig, allerdings werden einige C++14 Funktionalitäten wie polymorphe Lambdafunktionen **nicht** unterstützt und sind deswegen auch nicht im Praktikum erlaubt. Das verwenden extern Bibliotheken ist nicht erlaubt. Die C++-Standardbibliothek wird nicht als extern angesehen.

Aufgabe 1: Dijkstra's Algorithmus

Implementieren sie Dijkstra's Algorithmus, um die kürzeste Distanz zwischen zwei Knoten in einem Graph zu berechnen.

Um die Korrektheit Ihrer Implementierung zu testen, gibt es in den `test`-Unterverzeichnissen der 4 Testgraphen jeweils Testdaten. Diese bestehen aus jeweils 4 Vektoren: `source`, `target`,

`travel_time_length` und `geo_distance_length`. Sie können mit `load_vector<unsigned>` geladen werden. Die Daten hängen wie folgt zusammen: An der Stelle `travel_time_length[i]` beziehungsweise an der Stelle `geo_distance_length[i]` steht die Reisezeit bzw. Reisedistanz von Knoten `source[i]` zu `target[i]`. Falls es keinen Pfad gibt, ist der Eintrag `inf_weight`. Wenn Ihre Implementierung für jede Testanfrage und jeden Graph denselben Wert wie die vorgegebenen Referenzlösungen berechnet, dann ist ihr Programm wahrscheinlich korrekt.

Für Aufgabe 1 gibt es in `/misc/praktikum/graph/aufgabe1` einen Graph dessen `test`-Unterverzeichnis weder eine `travel_time_length`- noch eine `geo_distance_length`-Datei enthält. Es ist Ihre Aufgabe diese beiden Dateien zu erstellen. Legen Sie die von Ihnen berechneten Dateien in Ihr GIT-Repository unter `/abgabe/aufgabe1/travel_time_length` und `/abgabe/aufgabe1/geo_distance_length` ab.

Wir bewerten die abgegebenen Dateien `/abgabe/aufgabe1/travel_time_length` und `/abgabe/aufgabe1/geo_distance_length` durch einen Vergleich mit unserer Musterlösung. Für jede korrekt beantwortete Anfrage gibt es einen Punkt. Es können bei Aufgabe 1 also bis zu 2000 Punkten erreicht werden, da es 1000 Anfragen und zwei Kantengewichtungen gibt. Weicht eine Antwort vom korrekten Wert auch nur leicht ab, dann gibt es für diese Anfrage keine Punkte.

Wichtig: Erzeugen Sie die Dateien mit der `save_vector`-Funktion. Für selbst ausgedachte Dateiformate gibt es keine Punkte.

Aufgabe 2-4: Contraction Hierarchies

Die weiteren 3 Aufgaben folgen dem selben Schema, wie die erste: Sie sollen die `travel_time_length` und `geo_distance_length` Vektoren für die restlichen Graphen berechnen. Es gibt aber 1000000 Anfragen, weswegen Dijkstras Algorithmus für die Berechnung zu langsam ist. Verwenden Sie deswegen Contraction Hierarchies (CH) um die Berechnungen zu beschleunigen. Die drei verbleibenden Aufgaben unterscheiden sich darin welche CH-Teile Sie implementieren sollen.

- Bei Aufgabe 2 ist die Knotenordnung und der mit Shortcuts augmentierte Graph gegeben. Die Daten liegen in den `travel_time_ch` und `geo_distance_ch` Unterverzeichnissen. Sie müssen in dieser Aufgabe also nur den Anfragealgorithmus implementieren.
- Bei Aufgabe 3 ist der augmentierte Graph nicht mehr gegeben, sondern nur noch die Knotenordnung. Sie müssen also den augmentierten Graph berechnen. Die Implementierung des Anfragealgorithmuses können Sie aus Aufgabe 2 übernehmen.
- Bei Aufgabe 4 gibt es nur noch den Eingabegraphen. Sie müssen also sowohl den augmentierten Graph, als auch die Knotenordnung bestimmen.

Details zum CH-Algorithmus finden Sie in den Einführungsfolien.

Abgabecheckliste

Bevor Sie ihre Antworten abgeben, vergewissern Sie sich bitte, dass

- Die Abgabefrist noch nicht verstrichen ist. Verspätete Abgaben werden nicht gewertet.

- Die abgegebenen Dateien 4000 bzw. 4000000 Byte lang sind und dem vorgeschriebenen Format entsprechen. Wir vergeben keine Punkte, wenn wir nicht feststellen können, welche Antwort zu welcher Anfrage passt.
- Die Abgabedateien in ihrem GIT-Repository unter

  ```
  /abgabe/aufgabe{1,2,3,4}/{travel_time,geo_distance}_length
  ```

  liegen.
- Das GIT-Repository enthält den von Ihnen geschriebenen Quellcode.
- Ein Aufruf von `./compile.sh` übersetzt alle Programme.
- Eurer Code muss sich auf einem Poolraumrechner übersetzten lassen.
- Schreiben sie eine E-Mail an `strasser@kit.edu` mit einer Nachricht was genau Sie wann und wo abgegeben haben.