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

Moritz Baum, Ben Strasser,
Tobias Zündorf

Übungsblatt

Praktikum Algorithm Engineering – Routenplanung (WS 15/16)

Ausgabe 22. Oktober 2015
Abgabe 16. November 2015

Allgemeines

Mit Ihrem Rechneraccount können Sie sich im Poolraum an den Rechnern oder per ssh von daheim aus anmelden. Per ssh melden Sie sich wie folgt an:

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

wobei X eine Zahl zwischen 1 und 12 ist. 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` ob der Rechner bereits von jemand anderem genutzt wird.

Das erste was Sie tun müssen ist ihr Standardpasswort ändern. Führen Sie dazu `passwd` aus.

Die fürs Praktikum relevanten Daten sind in `/misc/praktikum`. In `/misc/praktikum/code.git` liegt ein GIT-Repository das Code enthält, der Ihnen zum Lösen der Aufgaben zur Verfügung gestellt wird. In `/misc/praktikum/graph` liegen die Eingabedaten für die Aufgaben. Ferner gibt es für jeden Teilnehmer ein eigenes Verzeichnis `/misc/praktikum/username` auf das nur er Zugriff hat. Dieses Benutzerverzeichnis ist zum Hinterlegen der Aufgabenlösungen da.

Git

Verwenden Sie zur Verwaltung Ihres Codes ein GIT-Repository. Um das Repository zu erstellen führen sie folgenden Code aus:

```
cd /misc/praktikum/username
git clone ../code.git
```

Dies erzeugt ein Repository im Verzeichnis `/misc/praktikum/username/code.git` mit dem Sie ihren Code verwalten sollen. Falls Sie im Poolraum arbeiten, dann können Sie die Dateien in diesen Verzeichnis direkt verändern. Um die Veränderungen ins GIT-Repository zu übernehmen führen Sie die folgenden Befehle aus:

```
git add file1.cpp
git add file1.h
git commit -m "Created file1"
```

Wenn Sie `file1.cpp` ändern und die Änderungen speichern wollen müssen Sie, im Gegensatz zu svn, die Datei erneut angeben. Dies sieht dann wie folgt aus:

```
git add file1.cpp
git commit -m "Fixed bug in file1.cpp"
```

Sie können auch per SSH von einem anderen Rechner aus arbeiten (z.B. von daheim). Dazu muss das Repository auf der Server Seite ein "bare"-Repository sein. Dies erzeugen Sie wie folgt:

```
cd /misc/praktikum/username
git clone --bare ../code.git
```

Um das GIT-Repository auf den anderen Rechner zu übertragen führen Sie den folgenden Befehl aus:

```
git clone ssh://username@i11poolX.iti.kit.edu:/misc/praktikum/username/code.git
```

Dies erzeugt ein Verzeichnis mit dem Namen `code`. In diesem können Sie wie gewohnt Veränderungen vornehmen. Um die Veränderungen wieder auf den Poolraumrechner zurück zu übertragen, führen Sie folgenden Befehl aus:

```
git push origin master
```

Hinweis: Das `origin master` ist nur beim ersten `git push` notwendig. Falls Sie den Code von mehreren Rechnern aus verändern wollen benötigen sie noch `git pull`. Dieser Befehl kopiert die Commits vom Server in das lokale Repository.

Code

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. Diese kann verwendet werden um die Laufzeit ihres Codes zu messen. Die Datei `id_queue.h` enthält eine Prioritätswarteschlange[1]. 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[2] 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

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

[2] Sofern wir eine Maschine mit 32 Bit Integer haben.

Vektoren identisch sind. 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.

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` is 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. Die benannten Graphen dienen nur zu Testzwecken. Für jede Aufgabe gibt es einen eigenen Graph. 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.

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 jedes der 4 Testgraphen fertig berechnete Längen. Wenn Ihre Implementierung alle Längen gleich berechnet, dann können sie annehmen, dass ihr Code fehlerfrei ist. Genauer gibt es jeweils 4 Vektoren: `source`, `target`, `travel_time_length` und `geo_distance_length`. Diese 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, so ist der Eintrag `inf_weight`. Für den Graphen in `/misc/praktikum/graph/aufgabe1` gibt es nur die `source` und `target` Vektoren. Es ist Ihre Aufgabe die dazugehörige Reisezeiten und die Reisedistanzen zu berechnen. Legen Sie diese unter

`/misc/praktikum/username/aufgabe1/{travel_time_length,geo_distance_length}` ab. Laden Sie Ihren Code in Ihr GIT-Repository.

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. Die Anzahl an Anfragen ist aber sehr viel größer, weswegen Dijkstra's Algorithmus zu langsam ist um alle Anfragen rechtzeitig lösen zu können. 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.

Geben Sie bei den Aufgaben 3 und 4 ferner die berechneten CHs und Knotenordnungen ab. Legen Sie diese in `/misc/praktikum/username/aufgabe{3,4}/{travel_time,geo_distance}_ch` ab. Bitte schreiben Sie zudem eine E-Mail an strasser@kit.edu mit einer Nachricht was genau Sie wann und wo abgegeben haben.