Praktikumsaufgaben

Graphentheoretische Konzepte und Algorithmen

SoSe 24

P. Stelldinger

Aufgaben

Praktikum 1 3
Praktikum 2 4
Praktikum 3 6

Allgemeines zu allen Aufgaben

Die Aufgabenstellung ist aus folgenden Gründen nicht ganz genau spezifiziert:

- Sie sollen einen Entwurfsspielraum haben. Das heißt, Sie können relativ frei entscheiden, wie Sie die Aufgabe lösen, allerdings sollten Sie Ihre Entscheidungen bewusst fällen und begründen können. Insofern gibt es auch keine Musterlösung.
- Durch die unterschiedlichen Lösungen können Sie von Ihren Kommilitonen noch lernen und das *Structured Walk-Through* bleibt spannend.

Darüber hinaus dürfen Sie gerne unterschiedliche Quellen nutzen, aber nur wenn Sie diese auch angeben. Sie sollen auch nicht unbedingt alles selber programmieren (den in der jeweiligen Aufgabe geforderten Kernalgorithmus aber schon), nutzen Sie den vorgegeben Datentyp oder gerne auch andere Libaries.
Für die Bearbeitung der Praktikumsaufgaben erhalten Sie im Teams-Kanal der LV

- Beispielgraphen für das Praktikum (absichtlich auch mit Fehlern)
- Links zu verschiedenen Libaries

Das Ergebnis der Bearbeitung einer Aufgabe wird von jeder Gruppe im Praktikum vorgestellt. Das heißt, die Aufgaben muss *vor Praktikumstermin fertig bearbeitet* sein. Über die Vorstellung hinaus wird für jede Aufgabe erwartet,

1. Implementierung der gestellten Aufgabe, also
   - eine korrekte und möglichst effiziente Implementierung in Java, die der vorgegeben Beschreibung entspricht,
   - die Kommentierung der zentralen Eigenschaften/Ereignisse etc. im Code und
   - hinreichende Testfälle in JUnit und ihre Kommentierung.

   Hinweis: Wenn Sie in der Implementierung englische Fachbegriffe nutzen wollen, können Sie diese im Index des Buches von Diestel nachschlagen.

2. Schriftliche Erläuterung Ihrer Lösung (Lösungsdokumentation) *etwa 4-7 Seiten*
    - Geben Sie Ihre Lösungsdokumentation am Vorabend des Praktikumstermins per E-Mail an peer.stelldinger@haw-hamburg.de ab! In dem Betreff-Feld tragen Sie bitte ein "GKA Praktikum Gruppe [1-3] Team [1-10]".
    - Bringen Sie Ihre Lösungsdokumentation zum Praktikumstermin mit!
    - Bitte geben Sie folgende Daten im Kopf Ihrer Lösungsdokumentation an:
        - **Team:** Namen der Teammitglieder
        - **Aufgabenaufteilung:**
            a) Aufgaben, für die Teammitglied 1 verantwortlich ist;
            Dateien, die komplett/zum Teil von Teammitglied 1 implementiert/bearbeitet wurden
            b) Aufgaben, für die Teammitglied 2 verantwortlich ist;
            Dateien, die komplett/zum Teil von Teammitglied 2 implementiert/bearbeitet wurden
        - **Quellenangaben:** Angabe von wesentlichen Quellen, z.B. Web-Seiten/Bücher, von denen Quellcode/Algorithmen übernommen wurden, Namentliche Nennung von Studierenden der HAW, von denen Quellcode übernommen wurde
        - **Bearbeitungszeitraum:** Datum und Dauer der Bearbeitung an der Aufgabe von allen Teammitgliedern und Angabe der gemeinsamen Bearbeitungszeiten
    - kurze Beschreibung der Algorithmen und
    - Datenstrukturen
    - wesentliche Entwurfsentscheidungen ihrer Implementierung
    - Umsetzung der Aspekte der Implementierung
    - umfassende Dokumentation der Testfälle, wobei die Abdeckung der Testfälle diskutiert werden soll
    - Beantwortung der in der Aufgabe gestellten Fragen

Es gibt drei Praktikumsaufgaben, weitere Details werden in der jeweiligen Aufgabenstellung angegeben.

Eine Praktikumsaufgabe gilt als erledigt, wenn

1. Sie die Lösungsdokumentation am Praktikumsanfang abgegeben haben,
2. Sie im Praktikum Ihre Implementierung vorgestellt haben und diese von mir (mindestens) als ausreichend anerkannt wurde.

Ein Bonuspunkt für die Praktikumsaufgabe wird vergeben, wenn die Lösung und Präsentation sehr gut gelungen sind.

Beachten Sie: Die drei Aufgaben bauen teilweise aufeinander auf. Sie können (und sollten!) also dieselbe Codebasis nutzen und weiterentwickeln.

Aufgabe 1: Laden, Speichern, Visualisieren und Traversieren von Graphen

Die Graphen, mit denen Sie arbeiten, sollen in diesem Format (siehe auch VL-Folien) gespeichert und gelesen werden:
gerichtet

```
<name node1>[ -> <name node2>[(<edge name>)][: <edgeweight>]];
```

ungerichtet

```
<name node1>[ -- <name node2> [(<edge name>)][: <edgeweight>]];
```

Die Aufgabe umfasst:

- die Einarbeitung in die gewählte library
- das Einlesen/ Speichern von ungerichteten sowie gerichteten Graphen und die Visualisierung der Graphen,

  (Hinweis: reguläre Aussdrücke in Java nutzen)
- die Implementierung des Breadth-First Search (BFS) Algorithmus zur Traversierung eines Graphen,
- dabei soll als Ergebnis der kürzeste Weg (in Form einer geeigneten Datenstruktur) und die Anzahl der benötigten Kanten angegeben werden.
- einfache JUnit-Tests, die Ihre Methoden überprüfen und
- JUnit-Tests, die das Lesen, das speichern sowie den BFS-Algorithmus überprüfen unter Benutzung der gegeben *.gka*-Dateien
- die Beantwortung der folgenden Fragen:
  1. Was passiert, wenn Knotennamen mehrfach auftreten?
  2. Wie unterscheidet sich der BFS für gerichtete und ungerichtete Graphen?
  3. Wie können Sie testen, dass Ihre Implementierung auch für sehr große Graphen funktioniert?

Aufgabe 2: Berechnung des Spannbaums

In der Vorlesung wurde der Algorithmus von Kruskal zur Berechnung des Spannbaums vorgestellt.
Der Algorithmus von Prim dient ebenfalls der Berechnung eines minimalen Gerüstes in einem zusammenhängenden, ungerichteten, kantengewichteten Graphen.
Der Algorithmus beginnt mit einem trivialen Graphen T, der aus einem beliebigen Knoten des gegebenen Graphen besteht. In jedem Schritt wird nun eine Kante mit minimalem Gewicht gesucht, die einen weiteren Knoten mit T verbindet. Diese Kante und der entsprechende Knoten werden zu T hinzugefügt. Das Ganze wird solange wiederholt, bis alle Knoten in T vorhanden sind; dann ist T ein minimales Gerüst.

Algorithmus

Wähle einen beliebigen Knoten als Startgraph T.
Solange T noch nicht alle Knoten enthält:

- Wähle eine Kante e minimalen Gewichts aus, die einen noch nicht in T enthaltenen Knoten v mit T verbindet.
- Füge e und v dem Graphen T hinzu.

Für eine effiziente Implementierung des Algorithmus von Prim muss man möglichst einfach eine Kante finden, die man dem entstehenden Baum T hinzufügen kann. Dafür **soll** eine einfache Priority-Queue **oder** ein Fibonacci-Heap [1] benutzt werden.
In der Priority Queue sind alle Knoten gespeichert, die noch nicht zu T gehören. Alle Knoten haben einen Wert, der dem der leichtesten Kante entspricht, durch die der Knoten mit T verbunden werden kann. Existiert keine solche Kante, wird dem Knoten der Wert ω zugewiesen. Die Warteschlange liefert nun immer einen Knoten mit dem kleinsten Wert zurück. Die Effizienz des Algorithmus hängt infolgedessen von der Implementierung der Warteschlange ab.
Bei Verwendung eines Fibonacci-Heaps ergibt sich eine optimale Laufzeit.

Es sollen Algorithmen für minimale Spannbäume so implementiert werden, dass der minimale Spannbaum visualisiert und die Kantengewichtssumme des minimalen Spannbaums berechnet werden kann.

[1] aus einer geeigneten Bibliothek, zB java oder graphstream

Die Aufgabe umfasst folgende Teile:

1. Implementierung der beiden Algorithmen (Kruskal & Prim) mit einer möglichst kurzen Laufzeit. Geben Sie die Laufzeit der Algorithmen an.

   - Implementieren und testen Sie den Algorithmus von Kruskal (die Beschreibung gab's in der VL).
   - Implementieren und testen Sie den Prim-Algorithmus mit einer einfachen (aber weniger effizienten) Prioritätswarteschlange
   - **oder** einer effizienten Prioritäswarteschlange basierend auf Fibonacci-Heaps (wobei Sie selbst rausfinden, was Fibonacci-Heaps sind; aber eher nicht selbst implementieren).

   so dass Sie folgendes liefern:

   - eine kurze Erläuterung der Prioritätswarteschlange ohne oder mit dem Fibonacci-Heap,
   - als Ergebnis den minimalen Spannbaum und sein Gesamtgewicht,
   - JUnit-Tests, die die beiden Algorithmen unter Benutzung der zu konstruierenden randomisierten Graphen (s.u) umfassend überprüfen und
   - JUnit-Tests, die die beiden Algorithmen unter Benutzung der zu konstruierenden randomisierten Graphen (s.u) umfassend gegeneinander prüfen.

2. Überlegen Sie sich eine allgemeine Konstruktion eines ungerichteten Graphen für eine vorgegebene Anzahl von Knoten und eine vorgegebene Anzahl von Kanten mit beliebigen, aber unterschiedlichen, nicht-negativen Kantengewichten.

3. Erzeugen Sie mindestens 3 randomisierte, ungerichtete, gewichtete Graphen mit beliebigen, aber unterschiedlichen Kantenbewertungen. Die Graphen sollen möglichst groß sein, aber bei dem Kruskal-Algorithmus jeweils eine Laufzeit von unter einer, unter fünf und unter zehn Minuten haben. Es kann passieren, dass Sie keine dieser Laufzeiten erreichen bevor die Graphen nicht mehr in den Arbeitsspeicher passen. Dann haben Sie ggf. sehr effizient implementiert.

4. Wenden Sie die beiden Algorithmen auf die erzeugten Graphen an. Stellen Sie sicher, dass die Kantengewichtssumme der eventuell unterschiedlichen, minimalen Spannbäume gleich sind.

5. Was stellen Sie hinsichtlich der Zugriffe auf den Graphen und der Laufzeit fest?

Aufgabe 3: Eulerkreise

Der Algorithmus von Hierholzer ist auch ein Algorithmus, mit dem man in einem ungerichteten Graphen Eulerkreise bestimmt. Er geht auf Ideen von Carl Hierholzer zurück.

Algorithmus

Voraussetzung:
Sei $G = (V, E)$ ein zusammenhängender Graph, der nur Knoten mit geradem Grad aufweist.

i. Wähle einen beliebigen Knoten v_0 des Graphen und konstruiere von v_0 ausgehend einen Unterkreis K in G, der alle Eigenschaften eines Eulerkreises besitzt.

ii. Vernachlässige nun alle Kanten dieses Unterkreises.

iii. Am ersten Eckpunkt des ersten Unterkreises, dessen Grad größer 0 ist, lässt man nun einen weiteren Unterkreis entstehen, der wiederum ein Eulerkreis ist.

iv. Erstelle so viele Unterkreise, bis alle Kanten von einem Unterkreis durchlaufen wurden.

v. Nun erhält man den Eulerkreis, indem man mit dem ersten Unterkreis beginnt und bei jedem Schnittpunkt mit einem anderen Unterkreis, den letzteren einfügt, und danach den ersten Unterkreis wieder bis zu einem weiteren Schnittpunkt oder dem Endpunkt fortsetzt.

Den Algorithmus von Fleury kennen Sie aus der Vorlesung. Nun sollen beide implementiert und getestet werden.

Die Aufgabe umfasst folgende Teile:

1. Implementieren Sie *zuerst !!* Ihre Tests, die die Algorithmen zur Eulerkreissuche überprüfen:
   a) Entwerfen Sie bitte Tests erst für kleine, gespeicherte Graphen (sowohl Eulergraphen als auch andere). Wann ist eine gegebene Kantenfolge ein Eulerkreis?
   b) Erzeugen Sie dann randomisierte, ungerichtete Eulergraphen. Bitte entwerfen Sie weitere Tests damit. Beschreiben Sie bitte die Konstruktion von und begründen Sie, warum der Knotengrad immer gerade ist.
   c) Entwerfen Sie einen Test für große, ungerichtete Eulergraphen, der wiederholt (20-100 Mal) mit immer unterschiedlich vielen Kanten und Knoten durchläuft.
2. Bitte implementieren und testen Sie die beiden Algorithmen: Fleury und Hierholzer
3. Erläutern Sie, inwiefern sich das Entwerfen der Tests vor der eigentlichen Implementierung ausgewirkt hat. Welche Schwierigkeiten gab es? Welche Vorteile haben Sie dabei bemerkt?