Fakultät IV - Elektrotechnik und Informatik
Algorithmen und Datenstrukturen SoSe 2017
Brock / Mabrouk / Sieverling / Stahl

Aufgabenblatt 8

letzte Aktualisierung: 16. June, 11:06 Uhr

Ausgabe: 16.06.2017
Abgabe: 28.06.2017 19:59

Thema: *Minimale Spannbäume*

Abgabe

Die folgenden Dateien müssen für eine erfolgreiche Abgabe im svn Ordner
`Tutorien/txx/Studierende/deinname/Abgaben/`
eingecheckt sein:

Geforderte Dateien:	
`Blatt08/src/UnionFindSet.java`	Aufgabe 2.1
`Blatt08/src/SimpleGraph.java`	Aufgabe 2.2

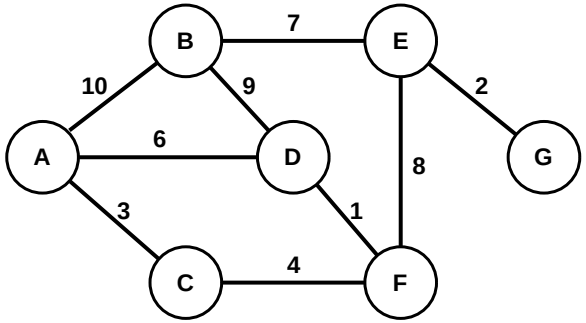
Wichtige Ankündigungen

- Unit tests dürfen geteilt werden. Lösungen dürfen auf keinen Fall geteilt werden! Wir testen auf Plagiate.
- Verändert die Methodenköpfe vorgegebener Klassen nicht. Verändert keine Klassen die nicht Teil der Abgabe sind und fügt keine neuen hinzu.

1. Aufgabe: Minimale Spannbäume

Der Spannbaum T eines Graphen G ist ein Baum (d.h. T ist ein zusammenhängender, ungerichteter und zyklenfreier Graph) der alle Knoten des Graphen G verbindet, und dabei nur Kanten verwendet die auch in G vorhanden sind. Ein Spannbaum ist dann minimal, wenn die Summe seiner Kantengewichte gleich oder kleiner ist als die aller anderen möglichen Spannbäume.

Beispiel: Betrachtet den folgenden ungerichteten Graphen als Ausgangssituation:



Wie sieht der minimale Spannbaum für diesen Graph aus? Wie kann dieser berechnet werden?

1.1. (Tut) Minimalen Spannbaum (MST) mit Prim's Algorithmus berechnen Diskutiert kurz die Funktionsweise des in der Vorlesung vorgestellten Algorithmus von Prim und simuliert diesen auf dem gegebenen Graphen.

1.2. (Tut) Minimalen Spannbaum (MST) mit Kruskal's Algorithmus berechnen Diskutiert kurz die Funktionsweise des in der Vorlesung vorgestellten Algorithmus von Kruskal und simuliert diesen auf dem gegebenen Graphen.

1.3. (Tut) Die Union-Find-Datenstruktur Eine Union-Find bzw. Disjoint-Set Datenstruktur verwaltet disjunkte Mengen (Partitionen).
Besprecht, wie eine Union-Find Struktur funktioniert und wie sie sinnvoll für Kruskals Algorithmus verwendet werden kann.

Besprecht auch die Bedeutung der einzelnen Methoden einer Union-Find Datenstruktur:

- `add` fügt neue Partitionen, die jeweils genau ein Element enthalten, hinzu.
- `getRepresentative` findet die Partition, zu der ein bestimmtes Element gehört, und gibt dessen Repräsentanten zurück. ("find")
- `union` vereint zwei Partitionen miteinander.

2. Aufgabe: Minimale Spannbäume - Implementierung des Kruskal Algorithmus

Der Algorithmus von Kruskal überprüft bei jeder neu betrachteten Kante, ob deren inzidente Knoten bereits verbunden sind. Ein naiver Ansatz wäre, dies mit Hilfe einer Breiten- oder Tiefensuche ausgehend von einem der beiden beteiligten Knoten zu testen.
Mit Hilfe einer *Union-Find-Datenstruktur* kann der Algorithmus von Kruskal jedoch wesentlich effizienter implementiert werden. Implementiert zuerst die union Find Datenstruktur und dann den Algorithmus von Kruskal.

2.1. Union-Find Datenstruktur - Implementierung (40 Punkte) Die vorgegebene Klasse `UnionFindSet.java` stellt die folgenden Methoden zur Verwaltung einer Union-Find Struktur bereit.

- `void add(T element)` Erstellt eine einzelne neue Partition mit dem übergebenem Element.
- `void add(List<T> elements)` Erstellt n Partitionen aus den n Elementen in `elements`.

- `T getRepresentative(T x)` Sucht die Partition, die `x` enthält, und gibt deren Repräsentanten zurück.

- `union(T x, T y)` Die Methode sucht die Partitionen, die `x` und `y` enthalten. Sollten die Partitionen verschieden sein, werden sie vereinigt - d.h. sie bekommen den gleichen Repräsentanten zugeordnet.

Implementiert diese Methoden entsprechend der Beschreibung!

Hinweis: Partitionen werden in unserer Aufgabe stellvertretend durch einen in der Partition enthaltenen Knoten repräsentiert. Für alle Knoten einer Partition gibt `getRepresentative(T x)` den selben "Repräsentanten" zurück. Eine Partition mit nur einem Element hat sich selbst als Repräsentanten.

Es gibt mehrere Methoden zur Implementierung der Union-Find Datenstruktur. Für diese Aufgabe ist eine naive Implementierung erlaubt, die für `union` $O(n)$ Operationen braucht (Ihr dürft aber auch gerne eine effizientere Struktur implementieren). Für die naive Implementierung könnt ihr die vorgegebene HashMap `element2representant` benutzen, die jedem Knoten seinen Repräsentanten zuordnet!

2.2. Kruskal's Algorithmus - Implementierung (60 Punkte) Implementiert den Algorithmus von *Kruskal* in der Methode `toMinSpanTree()` der Klasse `SimpleGraph`. Beachtet, dass `SimpleGraph` ein ungerichteten Grpahen implementiert, in dem jede Kante beim Auruf von `addEdge` zweimal hinzugefügt wird — Einmal als Vorwärts- und einmal als Rückwärtskante.

Die Methode `SimpleGraph toMinSpanTree()` soll den minimalen Spannbaum zurückgeben. Dieser Spannbaum soll ebenfalls ein `SimpleGraph` sein, wird also jede Kante des Baumes zweimal enthalten.

Hinweis: Verwendet in eurer Implementierung eine PriorityQueue für die sortierte Bearbeitung der Kanten, und ein `UnionFindSet` um hinzufügbare Kanten zu erkennen. Die für die PriorityQueue benötigte Methode `compareTo` in `Edge.java` haben wir bereits implementiert.

Behandelt fehlerhafte Eingabegraphen (z.B. nicht zusammenhängende Graphen) mit dem Werfen einer `RuntimeException()`, oder indem ihr `NULL` zurückgebt.