TU Berlin, Fakultät IV
Brock/Pfisterer/Koenig
MAR 5-1

Aufgabenblatt 7

Minimale Spannbäume

Abgabe (bis 30.06.2025 23:59 Uhr)

Die folgenden Dateien werden für die Bepunktung berücksichtigt:

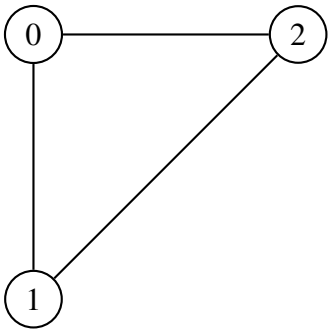
`Blatt07/src/EdgeWeightedGraph.java`	Aufgabe 3.1
`Blatt07/src/Clustering.java`	Aufgabe 3.2 bis 3.5

Als Abgabe wird jeweils nur die letzte Version im main branch in git gewertet.

Aufgabe 1: Union Find (Tutorium)

1.1 Was ist eine Äquivalenzrelation? Was sind Äquivalenzklassen und wie hängen diese mit Zusammenhangskomponenten zusammen?

1.2 Wozu wird Union Find benutzt? In welchem Algorithmus spielt er eine Rolle? Wie funktioniert Union Find? Gehen Sie den Algorithmus schrittweise an folgendem Beispiel durch:

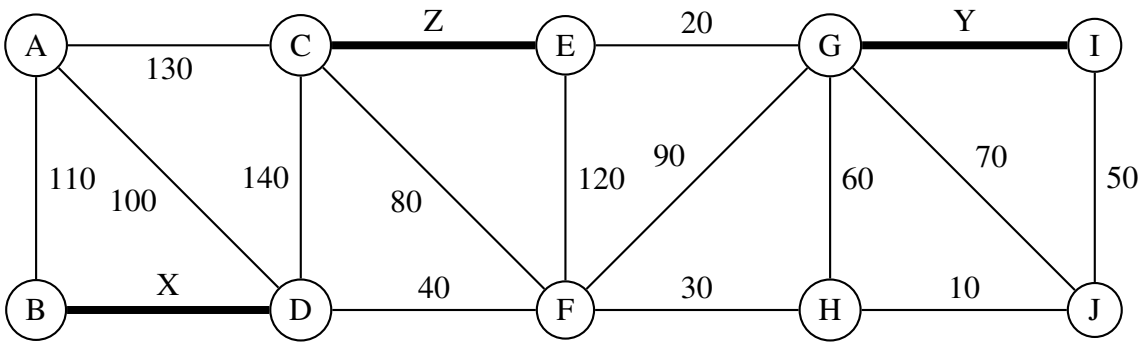


1.3 Funktioniert der folgende Code-Abschnitt für Union Find? Warum wäre das der Fall?

```
public void union(int p, int q) {
   if (connected(p, q)) return;
   for (int i = 0; i < id.length; i++)
      if (id[i] == id[p]) id[i] = id[q];
   count--;
}
```

Aufgabe 2: Minimum Spanning Tree (Tutorium)

Nehmen Sie an, der Minimum Spanning Tree (MST) dieses Graphen enthalte die Kanten X, Y und Z.



2.1 Notieren Sie die Gewichte der restlichen Kanten, d.h. alle außer X, Y und Z, die zum MST dieses Graphen gehören müssen unter der obigen Annahme.

2.2 Geben Sie jeweils die größte obere Schranke für die Gewichte der Kanten X, Y und Z an, mit der garantiert ist, das alle drei Kanten tatsächlich Teil des MST sind.
Hinweis: Geben Sie individuell für jedes Gewicht x, y, z eine Schranke an, wobei x, y und z die entsprechenden Gewichte zu den Kanten X,Y und Z sind.

2.3 Setzen Sie für X, Y und Z folgende Gewichte ein und führen Sie Kruskal und Prim ab A aus.

$$x = 120$$
$$y = 50$$
$$z = 80$$

Aufgabe 3: Clustering (Hausaufgabe)

In dieser Aufgabe werden Sie mithilfe des *Prim*-MST-Algorithmus ein Clustering-Problem lösen. Ein Cluster ist eine Gruppe von Objekten, die über ähnliche Eigenschaften verfügen. Diese Ähnlichkeit bzw. Unähnlichkeit lässt sich zum Beispiel durch eine Norm ausdrücken. In dieser Aufgabe betrachten wir Punkte im $\mathbb{R}^n$ mit der euklidischen Norm.
Clusteranalyse wird in vielen verschiedenen Bereichen angewendet, wie z.B. in der Bildverarbeitung oder bei der Klassifizierung von Gendaten. Sie werden sich als Anwendungsbeispiel, neben den von uns erstellten n-dimensionalen Graphen, auch den IRIS Datensatz aus dem *UCI Machine Learning Repository* anschauen(https://archive.ics.uci.edu/ml/index.php).
Der Algorithmus auf dem diese Aufgabe beruht, folgt diesen Schritten:

1. Das Problem in einen gewichteten Graphen umwandeln. Dabei ist für die Kantengewichte z.B. eine geeignete Norm zu wählen.
2. Einen MST auf dem Graphen berechnen.
3. Der Annahme folgend, dass die Knoten, die zu einem Cluster gehören, mit Kanten verbunden sind, die ein kleines Gewicht haben, werden in dem MST nun eine zu bestimmende Anzahl an Kanten mit zu großen Gewichten entfernt.
4. Durch das Entfernen von Kanten aus dem MST, ist dieser nun nicht mehr zusammenhängend. Die einzelnen Zusammenhangskomponeten sind die gesuchten Cluster.

Für Interessierte haben wir im Git-Repository einen Artikel abgelegt, der ebenfalls mit einem MST-Algorithmus Cluster bestimmt.

3.1 Graphen einlesen (5 Punkte)

Schauen Sie sich den Konstruktor

```
public EdgeWeightedGraph(In in)
```

der Klasse `EdgeWeightedGraph` genau an, um die Datenstruktur zu verstehen. Jeder Knoten in dem Graphen hat eine Position im $\mathbb{R}^n$, die in der Objektvariablen `coord[][]` gespeichert wird. In diesem Konstruktor wird eine Textdatei eingelesen, in der sowohl der Graph durch seine Kanten definiert wird, als auch die Positionen der Knoten im n-dimensionalen Raum. Die Datei hat das folgende Format:

Input schematisch

```
[number of nodes]
[number of edges]
[dimensions]
[1. edge node1] [1. edge node2] [1. coordinate node1]... [last coordinate node1] [1. coordinate node2]... [last coordinate node2]
[2. edge node1] [2. edge node2] [1. coordinate node1]... [last coordinate node1] [1. coordinate node2]... [last coordinate node2]
.
.
.
[last edge node1] [last edge node2] [1. coordinate node1]... [last coordinate node1] [1. coordinate node2]... [last coordinate node2]
```

Zwei Beispiele finden Sie unter src/graph_bigger.txt und src/graph_small.txt.
Im Konstruktor wird der Graph aufgebaut, die Koordinaten gespeichert und die Gewichte der Kanten über die euklidische Norm definiert: die Kante zwischen x_i und x_j hat das Gewicht $\omega_{ij} = \|x_i - x_j\|_2$

Implementieren Sie die Methoden:

```
public double[][] getCoordinates()
```

und

```
public void setCoordinates(double[][] coord)
```

Dies sind die get- und die set-Methoden für die Objektvariable `coord`.

Hinweise:

- Gehen Sie die ganze Klasse `EdgeWeightedGraph` durch, sodass Sie sie danach verstanden haben.
- Das Dateiformat ist sehr ineffizient, da die Koordinaten jedes Knotens für jede seiner Kanten redundant gespeichert ist. Machen Sie sich darüber keine Gedanken.

3.2 Clustering Konstruktoren (0 Punkte)

Schauen Sie sich die beiden Konstruktoren der Klasse `Clustering` an:

```
public Clustering(EdgeWeightedGraph G)
public Clustering(In in)
```

Für den ersten Konstruktor ist der Graph bereits gegeben, aber es gibt keine Label. Das Clustering können Sie dann über die `plotClusters()`-Methode überprüfen.
In dem zweiten Konstruktor wird der Graph erst erstellt. Das Inputfile, welches hier eingelesen wird, hat folgende Form:

Input mit Label schematisch

```
[number of nodes]
[dimensions]
[1. coordinate node 1]... [last coordinate node 1] [LABEL]
[1. coordinate node 2]... [last coordinate node 2] [LABEL]
.
.
.
[1. coordinate node n]... [last coordinate node n] [LABEL]
```

Die Datei definiert also die Positionen von Datenpunkten (z.B. mehrdimensionalen Messwerten), genauer die Koordinaten der Punkte im $\mathbb{R}^n$. Außerdem ist für jeden Datenpunkt ein Label (als `String`) angegeben, welches ihn einem spezifischen Cluster zuordnet. Diese Cluster werden in der Objektvariable `labeled` gespeichert, damit Sie später vergleichen können, ob der Algorithmus die Cluster richtig zugeordnet hat. In dem Konstruktor wird ein vollständiger Graph erzeugt, in dem jeder der Knoten mit allen anderen verbunden ist.

Hinweis:

- Schauen Sie sich an, wie die Label gespeichert werden, damit Sie diese Information in Aufgabe 3.5 verwenden können.
- Die Datei iris.txt hat die vorgegebene gelabelte Struktur.

3.3 Clusteranalyse 1 (30 Punkte)

Implementieren Sie den in der Einleitung erläuterten Algorithmus in einer ersten Variante. Hier wird die Anzahl der zu erwartenden Cluster mitgegeben:

```
public void findClusters(int numberOfClusters)
```

Dafür müssen Sie sich überlegen, welche und wie viele Kanten aus dem MST entfernt werden müssen. Anschließend müssen die Zusammenhangskomponenten gefunden werden. Dazu ist es sinnvoll z.B. eine Hilfsmethode `connectedComponents` zu schreiben, welche die Zusammenhangskomponenten findet. Sie können hier die Klasse `UF` nutzen, welche den Union Find Algorithmus implementiert.
Die Cluster werden in der Klassenvariable `clusters` gespeichert und sollten innerhalb der Cluster nach Knoten (Indizes der Datenpunkte) sortiert sein.

Hinweise:

- Wenn Sie die Zusammenhangskomponenten mithilfe einer Hilfsmethode finden, können Sie diese Funktion in der nächsten Teilaufgabe wiederverwenden.
- Testen Sie Ihre Methode auch mit der Methode `plotClusters()`. Diese visualisiert die Cluster im 2-dimensionalen Raum. Nutzen Sie dazu die Ihnen gegeben Beispieldateien: graph_bigger.txt, graph_small.txt und iris.txt

3.4 Clusteranalyse 2 (35+10 Punkte)

Implentieren Sie eine zweite Variante des Algorithmus, in der die Anzahl der Cluster nicht mitgegeben ist:

```
public void findClusters(double threshold)
```

In dieser Variante untersucht der Algorithmus die Ähnlichkeit der Kanten des MST mit dem Variationskoeffizienten, dessen Berechnung Sie in dieser Methode implementieren sollen:

```
public double coefficientOfVariation(List <Edge> part)
```

Der Variationskoeffizient ist folgendermaßen definiert

$$CV(X) = \frac{\sigma}{\mu} = \frac{\sqrt{\frac{1}{n}\sum_{i=0}^{n} x_i^2 - (\frac{1}{n}\sum_{i=0}^{n} x_i)^2}}{\frac{1}{n}\sum_{i=0}^{n} x_i} \tag{1}$$

wobei X Ihre Liste der Kanten und x_i jeweils das Kantengewicht der Kante mit Index i. Die Methode gibt Ihnen also einen Variationskoeffizienten für eine Liste zurück. Sie sollen nun in `findClusters` entscheiden, ob einzelne Kanten entfernt werden oder nicht. Dazu müssen Sie zunächst die Kanten, die in Ihrem MST sind, nach ihrem Kantengewicht sortieren. Angefangen mit der Kante mit dem geringsten Gewicht fügen Sie Kanten zur Lösungsmenge hinzu und berechnen sodann immer den Variationskoeffizienten. Wenn der zurückgegebene Koeffizient dann den `threshold` überschreitet, kann diese nicht Teil der Lösungsmenge sein. Die Knoten, du durch Hinzunahme dieser Kante verbunden werden würde, sind den anderen Knoten im jeweiligen Cluster nicht ähnlich genug. Wenn Sie die Lösungsmenge der Kanten erzeugt haben, gehen Sie vor wie in 3.3, um die Klassenvariable `clusters` zu füllen.

3.5 Validierung (20 Punkte)

In dieser Methode implementieren Sie einen einfachen Weg, um zu überprüfen, wie gut der Algorithmus einen gegebenen Datensatz clustert.

```
public int[] validation()
```

Dabei gehen Sie davon aus, dass das Clustering über die Anzahl der Cluster berechnet wurde (siehe 3.3) und diese mit der Anzahl der Label übereinstimmt. Die Methode gibt ein Array zurück, das als Dimension die Anzahl an Clustern hat. Für jedes dieser Cluster werden die Datenpunkte gezählt, die in der Analyse richtig zugeordnet wurden und die Anzahl der richtig zugeordneten Punkte wird dann in das Array geschrieben.

Hinweise:

- Testen Sie diese Methode an dem IRIS Datensatz, den wir Ihnen mitgegeben haben.
- Sie werden feststellen, dass der Algorithmus für die ersten beiden Klassen sehr gut funktioniert, für die dritte dagegen nicht.
- In allen Datensätzen, an denen Ihr Code getestet wird, sind die gelabelten Cluster nach aufsteigenden Knoten (Datenpunktindex) sortiert. Wenn Sie Ihre berechneten Cluster also in der Liste `clusters` so sortieren, dass jeweils für den ersten Eintrag in jedem Cluster die Reihenfolge aufsteigend ist, müssen Sie kein kompliziertes Matching vornehmen, sondern können das erste Cluster mit dem ersten gelabelten Cluster vergleichen. (Beispiel dieser Reihenfolge: `[0,3,5], [1,2,6], [4,7,8]`)

Was Sie nach diesem Blatt wissen sollten:

- Was Union Find ist und wie es funktioniert
- Was ein MST ist und welche Eigenschaften er hat
- Wofür man einen MST nutzen kann
- Was ein Schnitt ist und wie die Schnitteigenschaft definiert ist
- Was kreuzende Kanten sind
- Was das allgemeine Vorgehen zum Bestimmen eines MST ist
- Wie der *Prim*- und der *Kruskal*-Algorithmus funktionieren
- Was die beiden Algorithmen unterscheidet