BAI3-GKA WiSe23
Graphentheoretische Konzepte und Algorithmen

Praktikumsaufgabe 2

Dokumentation

GKA-Gruppe	
Team	
	Iryna Trygub
	Ansgar Deuschel
	Kristoffer Schaaf

Bearbeitete Themen in Stichpunkten:

Iryna Trygub	Graph Generator, Kruskal
Ansgar Deuschel	Tests, Refactoring
Kristoffer Schaaf	Tests, Dokumentation, Ops

Inhaltsverzeichnis

1 Einleitung

Der Kruskal-Algorithmus ist ein Greedy-Algorithmus, der verwendet wird, um den minimalen Spannbaum eines Graphen zu finden. Der Algorithmus funktioniert in mehreren Schritten:

1. Zuerst werden alle Kanten des Graphen nach ihrem Gewicht sortiert. Die Kante mit dem kleinsten Gewicht wird zuerst betrachtet.
2. Für jeden Knoten wird eine neue Teilmenge erstellt
3. Für jede Kante (v1, v2) aus E, beginnend mit der kleinsten Gewichtskante:
   (a) Überprüfe, ob v1 und v2 in verschiedenen Teilmengen liegen. Wenn ja, füge die Kante dem minimalen Spannbaum hinzu und vereinige die beiden Teilmengen.
   (b) Wenn v1 und v2 bereits in derselben Teilmenge sind, gehe zur nächsten Kante.
4. Der Prozess wird fortgesetzt, bis alle Knoten im neuen Graphen verbunden sind oder alle Kanten betrachtet wurden. Wenn alle Knoten verbunden sind, stoppt der Algorithmus. Wenn noch Kanten übrig sind, werden sie ignoriert, da sie einen Kreis im Graphen verursachen würden.

Der Kruskal-Algorithmus ist besonders nützlich in der Operations Research, insbesondere wenn es darum geht, die kostengünstigste Verbindung zwischen verschiedenen Stationen zu finden, beispielsweise beim Ausbau eines Schienennetzes[1].

2 Laufzeit

Die Laufzeit des Algorithmus hängt stark von Schritt 1 und 2 ab. Im Allgemeinen kann jeder Sortieralgorithmus verwendet werden, um die Kanten des Graphen zu sortieren. Unter Verwendung der Prioritätswarteschlange als Datenstruktur in Schritt 1 und 2 ist das Finden und Entfernen der kleinsten Kante in $O(log(n))$ Schritten möglich[2].

3 Dokumentation der Implementierung

3.1 Disjoint Set

Die Bibliothek Graphstream bietet eine eigene Implementierung des DisjointSet. Aufgabe hiervon ist es, die verschiedenen Teilmengen so zu verwalten, dass diese disjunkt zueinander sind. Wie in Anhang B2 beschrieben[KN12] werden drei verschiedene Operation für die Verwaltung der Teilmengen benötigt. Aufgrund einer fehlenden Operation in der Graphstream Implementierung und der fehlenden Erweiterbarkeit bedingt durch die Scopes der implementierten Operationen, haben wir uns dazu entschieden unser eigenes Disjoint Set zu implementieren. Dieses hat nach wie vor die drei benötigten Operationen, ist aber zustandslos und bietet somit nur statische Methoden.

- *makeSet(*$HashSet\langle Node\rangle$*)* oder *makeSet(*$HashSet\langle MultiGraph\rangle$*)*
  Hierbei wird ein neues Set erstellt. Dieses hat den Typen HashSet und enthält weitere Teilmengen von Knoten. Auch diese werden in HashSets gespeichert. Final hat das DisjointSet also den Typen $HashSet\langle HashSet\langle Node\rangle\rangle$.
  Dieser Operation ein Set von Nodes übergeben werden, welches nach Operationsaufruf das einzige Element des Disjoint Sets wäre. Das entspräche der Vorgabe. Um aber direkt bei Initialisierung des DisjointSets die Teilmengen für jeden Knoten zu erstellen, gibt es eine Überschreibung der ersten Methodensignatur. In dieser wird ein Graph übergeben und ein initialisertes DisjointSet mit einer Teilmenge pro Knoten wird zurückgegeben.

[1] https://www.bwl-lexikon.de/wiki/kruskal-algorithmus/
[2] https://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html

- *union(HashSet⟨Node⟩, HashSet⟨Node⟩, HashSet⟨HashSet⟨Node⟩⟩)*
  In dieser Funktion werden zwei Teilmengen konkateniert und dem DisjointSet hinzugefügt. Die ursprünglichen Teilmengen werden anschließend aus dem DisjointSet entfernt. Da das DisjointSet zustandslos ist, muss die Operation zusätzlich das bisherige DisjointSet mit als Parameter übergeben bekommen. In diesem Punkt unterscheidet sich unsere Implementierung von der Vorgabe.

- *find-set(HashSet⟨HashSet⟨Node⟩⟩, Node)*
  Diese Operation sucht die Teilmenge in dem DisjointSet, welche den übergebenen Knoten enthält. Auch hier muss aufgrund der Zustandslosigkeit das DisjointSet mit als Parameter übergeben werden.

3.1.1 Laufzeit DisjointSet

Die Methoden add(), remove() und contains() des HashSets haben eine Komplexität von $O(1)$[3]. Somit hat make-set eine Komplexität von $O(1)$, union $O(1)$ und find-set $O(n)$.

3.2 Rückgabewert Kruskal

Der Rückgabewert der Kruskal Methode createMinimalSpanningForrest(Multigraph graph) ist ein neu erzeugtes KruskalResult Objekt. In diesem ist der Spannwald als Graph und als Set aus Kanten enthalten. Zusätzlich ist die gesamte Größe der Kantengewichte gespeichert. Zur Erzeugung des Spannwaldgraphen wird die Hilfsfunktion createSpanningForrestGraph(Multigraph graph, *HashSet⟨Edge⟩* minimalSpanningTree) genutzt. In dieser wird über alle Kanten des minimalen Spannwaldes iteriert und ein entsprechender Graph zusammengebaut.

3.2.1 Alleinstehende Knoten im Spannwald

Anhand Punkt 1 in der Einleitung1 wird ersichtlich, dass alleinstehende Knoten - also Knoten ohne Kanten - durch die fehlenden Kanten nicht weiter beachtet werden. Hierfür wurde eine neue Funktion eingeführt, welche alle betroffenen Knoten zu Beginn in einem Set speichert und diese dann dem Graph hinzufügt, welcher den Spannwald final darstellt.

3.3 Die Main Methode

In der Main Methode wird der Kruskal Algorithmus auf einen zufällig generierten Graphen angewendet. Die Größe des Graphen muss hierfür hart gecoded werden. Damit nach Ausführung der Main Methode der zufällig generierte Graph betrachtet werden kann, wird dieser über den GraphFileWriter des letzten Praktikums in eine Datei im Pfad *src/main/resources/graphs* abgelegt. Ein Ausschnitt aus einer solchen Datei könnte so aussehen:

```
#undirected:Graph;
node196--node122(edge63)::1;
node63--node52(edge1075)::1;
```

Die Summe aller Kantengewichte aus dem minimalen Spannwald ist in dem KruskalResult Objekt enthalten. Dieses wird in der Konsole ausgegeben, bevor der finale minimale Spannwald visualisiert wird. Zusätzlich steht in der Konsole, wie viele Knoten und Kanten der erzeugte Graph enthalten hat.

[3] https://www.baeldung.com/java-collections-complexity

4 Tests

4.1 Fest definierte Testgraphen

Getestet wird zum einen mit verschiedenen fest definierten Testgraphen. Diese entsprechen bestimmten Vorgaben und sollten alle relevanten Edge Cases abdecken.

Der erste zu testende Graph ist der Testgraph aus der letzten Praktikumsaufgabe1. Soweit gibt es in diesem Graphen keine Edge Cases. Der Test dient lediglich einmal den Happy Path zu testen.

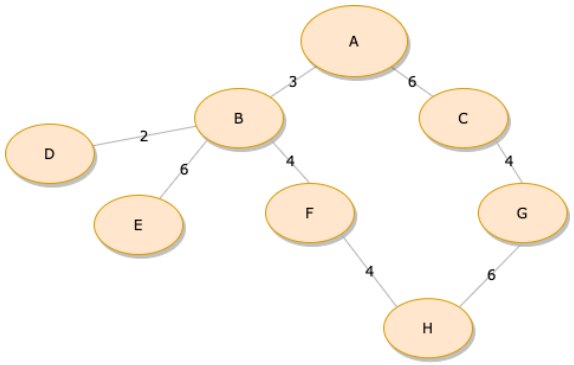

Abbildung 1: Ein zusammenhängender Graph

Der zweite Graph besteht aus zwei nicht zusammenhängenden Teilgraphen2. Da in dieser Aufgabe nicht nur ein Spannbaum sondern ein Spannwald erstellt werden soll. Muss der Algorithmus entsprechend auch auf einem nicht zusammenhängendem Teilgraphen getestet werden. Dass die beiden Graphen die gleiche Anzahl an Knoten und an Kanten mit den gleichen Gewichten hat, ist nicht weiter relevant.

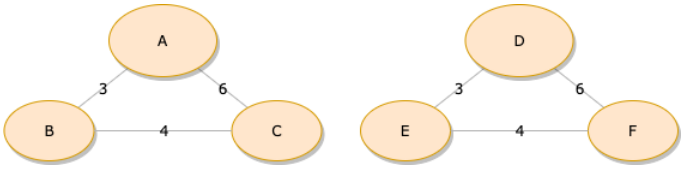

Abbildung 2: Zwei nicht zusammenhängende Graphen

Ein spannender Fall ist das Betrachtet eines einzelnen Knotens als Graphen3, wenn dieser keine Kanten hat. Durch die Implementation werden in den meisten Fällen nur Knoten beachtet, welche auch Kanten haben, da diese ansonsten nicht in der Prioritätswarteschlange referenziert werden. Die Definition eines minimalen Spannbaums lautet allerdings: Ein minimaler Spannbaum ist ein Baum, der alle Knoten eines Graphen verbindet und dabei die minimale Summe der Kantengewichte hat[4]. In diesem speziellen Fall ist der Graph selbst ein Baum und da es keine Kanten gibt, ist die Summe der Kantengewichte gleich Null. Daher ist der Graphen selbst der minimale Spannbaum.

Abbildung 3: Graph mit einem einzelnen Knoten

[4]https://en.wikipedia.org/wiki/Minimum_spanning_tree

Der Vollständigkeit halber testen wir auch einen Graphen welcher aus zwei nicht zusammenhängenden Teilgraphen besteht4. Der eine besteht aus einem Graphen ohne Edge Cases, der zweite ist ein einzelner Knoten ohne Kanten.

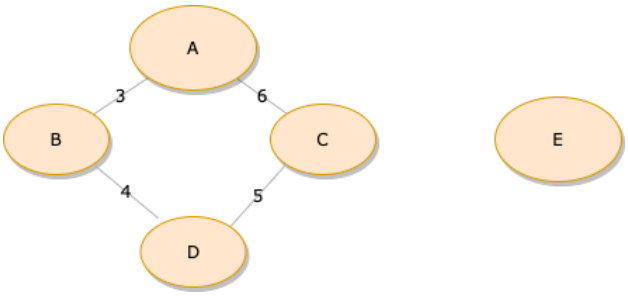

Abbildung 4: Zwei nicht zusammenhängende Graphen, einer mit einem einzelnen Knoten

4.2 Zufällig generierte Graphen

Als Feedback vom letzten Praktikum haben wir mitgenommen, dass wir auch mit zufällig generierten Graphen testen sollen. Ein Problem hierbei ist offensichtlich wie das Ergebnis generisch getestet werden soll. Hierfür bietet sich der Kruskal Algorithmus der Graphstream Bibliothek an. Wie oben beschrieben gibt auch dieser einen Graphen zurück und kann somit verglichen werden. Generiert werden unsere Graphen über den GraphGenerator. Dieser erwartet als Parameter die Anzahl an Knoten und Kanten, das maximale Gewicht der Kanten und ob diese gerichtet sein sollen. Der zurückgegebene Graph ist entweder vollständig ungerichtet oder vollständig gerichtet.

4.2.1 Assertions

Das Überschreiben der Equals Methode des Graphs von Graphstream erschien in unserem Fall zu aufwändig. Wir hätten hierfür bereits bestehenden Code stark refactoren müssen. Außerdem sind sowohl im Typen Kruskal als auch Prim nur die Felder *graph* und *treeEdges* gespeichert. *graph* ist hierbei der unveränderte Originalgraph und *treeEdges* ist eine Liste mit allen Kanten des minimalen Spannwaldes. Bei größeren Graphen besteht die Möglichkeit, dass unser Kruskal Algorithmus einen anderen minimalen Spannbaum findet als die GraphStream Algorithmen Kruskal oder Prim. Als mögliche Assertions für die Tests mit zufällig erzeugten Graphen kommt also nur das Vergleichen der aufsummierten Kantengewichte in Frage.
Damit die Tests aussagekräftig sind, haben wir die Kantengewichte in einem möglichst großen Bereich (1-100) zufällig generiert.

4.2.2 Ergebnisse und Laufzeit der Tests

Nr	**Name**	**Knoten**	**Kanten**	**Dauer (min)**
1	Kruskal FewerNodesThanEdges	50000	450000	4,88
2	Prim FewerNodesThanEdges	50000	450000	0,0065
3	Kruskal SameNodesThanEdges	75000	75000	4,84
4	Prim SameNodesThanEdges	75000	75000	0,0046
5	Kruskal MoreNodesThanEdges	100000	50000	4,99
6	Prim MoreNodesThanEdges	100000	50000	0,0043

Tabelle 1: Open List

In der obigen Tabelle 1 ist die Laufzeit unseres Kruskalalgorithmus im Vergleich zu dem Primalgorithmus von graphstream dargestellt.

Auffällig ist, dass eine Steigerung der Kantenanzahl zu einem geringeren Anstieg der Laufzeit führt, als bei einem gleichzahligen Anwachsen der Knotenanzahl. Da der Algorithmus über die Kanten iteriert, können wir uns dieses Phänomen nicht erklären. Zum Finden der kantenlosen Knoten iterieren wir auch über die Knoten, dies jedoch mit einer Zeitkomplexität von O(n) und über Streams.
Außerdem ist zu beachten, dass der Primalgorithmus deutlich schneller läuft.

Literatur

[KN12] Sven Oliver Krumke and Hartmut Noltemeier. *Graphentheoretische Konzepte und Algorithmen, 3. Auflage.* Leitfäden der Informatik. Teubner, 2012.