BAI3-GKA WiSe23
Graphentheoretische Konzepte und Algorithmen

Praktikumsaufgabe 3 - Hierholzer

Dokumentation

GKA-Gruppe	
Team	
	Iryna Trygub
	Ansgar Deuschel
	Kristoffer Schaaf

Bearbeitete Themen in Stichpunkten:

Iryna Trygub	Graph Generator
Ansgar Deuschel	Hierholzer, Energiemessungen, Dokumentation
Kristoffer Schaaf	Tests, CVSWriter, Dokumentation

Inhaltsverzeichnis

1 Einleitung

Der Hierholzeralgorithmus findet Eulerkreise in einem vollständigen, ungerichteten Graphen, dessen Knotengrade gerade sind. Im folgenden wird seine Implementierung, sowie sein Laufzeit- und Energieverbrauchsverhalten besprochen.

2 Hierholzer Laufzeit

Es wurden drei Messungen durchgeführt. Mit 3.000, 400.000 und 1.000.000 Kanten wurde der Hierholzer-Algorithmus jeweils zehnmal ausgeführt und dabei die Zeit in ms und der Energieverbrauch in Joule mit Hilfe des Commandlinetools PowerLog3.0 von Intel erfasst. Für die einzelnen Messungen ergeben sich folgende grobe Durchschnittswerte:

Kantenzahl	Zeit in ms	Energieverbrauch in J
3000	10	0.5
400000	1300	50
1000000	5500	200

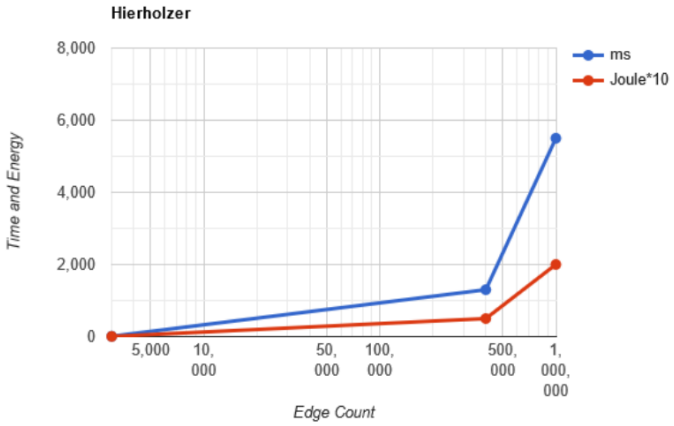

Abbildung 1: Verhalten von Zeit und Energie zu Kantenzahl (log)

Es ist zu erkennen, dass der Energieverbrauch weniger steil mit der Anzahl der Kanten wächst, als der Zeitaufwand. Aber auf Grund der undurchsichtigen Art und Weise, wie PowerLog3.0 den Energieverbrauch erfasst und der hohen Anzahl an Faktoren, die diesen beeinflussen (Speichertakt, RAM-Spannung, Netzteil, CPU-Takt, Core-Spannung, Festplattenbauart, etc.), kann nur vermutet werden, dass diese gemessenen Werte der Realität entsprechen. Anscheinend kann der Prozessor mit wachsender Zahl an Berechnungen gleicher Art, effizienter umgehen, als mit häufiger wechselnden Berechnungsarten. Dies ist aber nur eine Mutmaßung und eine professionelle Beurteilung der Messung ist auf unserem Fachwissen fußend nicht möglich.

3 Hierholzer Implementierung

3.1 Schritte

Zunächst wird eine ArrayList mit den Knoten der Eulerkreise circleNodes deklariert. Die folgenden Schritte werden solange wiederholt, bis im Graph keine Kanten mehr enthalten sind.

- I) Wenn die Liste mit den Knoten der Eulerkreise noch leer ist, wird mit dem ersten Knoten aus dem Graph begonnen, sonst mit irgendeinem aus der in der Liste enthaltenen Knoten, der noch erreichbare Nachbarn hat.

- II) Von diesem Startknoten ausgehend wird nun ein Kreis im Graph gesucht. Dafür werden solange noch unbesuchte Kanten gegangen, bis ein Knoten erreicht ist, der gleich dem Startknoten ist. Um zu überprüfen, ob eine Kante schon besucht wurde, werden solche in einem Hashset visitedEdges, welches zu Beginn einer jeden Iteration geleert wird, zwischengespeichert.

- III) Wenn der Kreis geschlossen ist, werden die gefundenen Knoten zunächst den circleNodes und dann als Graphobject zusammengefügt der Ergebnisliste hinzugefügt, dann die visitedEdges, nach einer für die späteren Verwendung des Graphobjetkts notwendiger Zwischenspeicherung cachedEdges, aus dem Graphen entfernt.

- IV) Nun wird wieder mit Schritt I) begonnen, wenn der Graph noch Kanten enthält.
- V) Wenn der Graph keine Kanten mehr enthält, werden diese mit Hilfe der Zwischenspeicherung cachedEdges wieder dem Graph hinzugefügt. Die Ergebnisliste, bestehend aus den einzelnen Eulerkreisen als Graphobjekte, wird von der Funktion zurückgegeben.

3.2 Datentypen

Zur Ausführung des Hierholzer-Algorithmus müssen verschiedene Objekte wie Kanten und Knoten zwischengespeichert werden. Prüfungen, ob ein Element in einer Datensammlung enthalten ist, hat beim Hashset, anders als bei Listen, eine konstante Komplexität. Deshalb werden die visitedEdges als solches deklariert. Eine Arraylist ist grundsätzlich schneller als eine Linkedlist, es sei denn, es werden oft Elemente hinzugefügt. Dann nämlich sorgt die Implementierung von Java dafür, dass das ganze Array auf einen neuen größeren Speicherbereich kopiert wird, was viel Zeit kostet. Deshalb benutzen wir für alle anderen zwischenspeichernden Datensammlungen die LinkedList.

3.3 Testen der ungeraden Knotengerade

In einem Graphen mit mindestens einem Knoten, welcher einen ungeraden Knotengrad hat, kann der Hierholzer Algorithmus keinen Erfolg haben. Ein Beispiel ist hier3 in den Tests zu finden.

Um sicherzustellen, dass dem Algorithmus ein valider Graph übergeben wird, wurde folgende Methode implementiert:

```
private static boolean graphIsValid(Graph graph) {
    ....
    List<Node> nodesWithFalseDegree = graph.nodes()
                                .filter(node -> node.getDegree() % 2 != 0)
                                .collect(Collectors.toList());
    for(Node node: nodesWithFalseDegree) {
        if(node.getEdgeBetween(node.getId()) == null) {
            return false;
        }
    }
    ...
}
```

Um zu prüfen ob Knoten einen ungeraden Kantengrad haben, werden diese Knoten zuerst alle in einer Liste gespeichert. Das ist notwendig, denn durch die genutzte Graphstream Bibliothek ensteht folgendes Problem: Angenommen es existiert ein Graph mit einem Knoten A und zwei Nachbarknoten B und C. Der Knoten A hat eine Schleife. Der Graph ist vollständig - A,B und C sind also alle miteinander verbunden (Abb. 8).

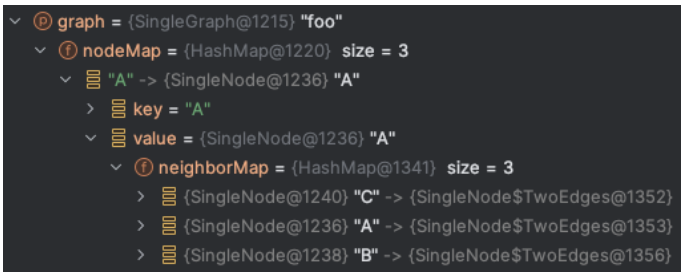

Abbildung 2: Ausschnitt des Graphen mit Loop aus dem Debugger

Wie nun im Debugger zu sehen2 hat der Knoten A nun nur drei Nachbarn. Zusätzlich hat er auch nur drei adjazente Kanten, AA, AB und AC. Das ist soweit alles korrekt. Der daraus resultierende Knotengrad ist allerdings ungerade und würde diesen Graph als invalide auswerten. Bei Betrachtung des Graphen hat dieser aber einen geraden Knotengrad.
Um dieses Problem zu umgehen wird in allen Kanten mit ungeradem Knotengrad also zusätzlich geprüft ob diese Kanten keine Schleifen sind. Erst wenn das der Fall ist, wird der Graph als invalide ausgewertet.

4 Graphgenerator Implementierung

Wir haben einen Euler-Multigraphen basierend auf dem vorgeschlagenen Algorithmus erstellt. Gemäß der Anforderung mussten wir einen Multigraphen mit einer festgelegten Anzahl von Knoten und Kanten erstellen. Wir haben eine Überprüfung hinzugefügt, um sicherzustellen, dass die von den Benutzern angegebene Anzahl von Kanten nicht geringer ist als die Anzahl der Knoten. Dies ist unter anderem notwendig, um den Graphen zusammenhängend zu halten. Wir erstellen eine Liste der Knoten des Graphen und eine Liste der Positionen, deren Länge der Anzahl der Kanten entspricht. Wir weisen jeder Knoten eine der Positionen zu. Wir speichern in einer HashMap namens posHashMap die Position für jeden Knoten. Zuerst wählen wir aus der posHashMap einen Knoten mit der Positionsnummer 0 oder, falls dieser nicht existiert, weisen wir eine zufällige Knoten als Startknoten zu. Danach iterieren wir von null bis zur Anzahl der Kanten.
Wir müssen die actualSource und actualTarget Knoten bestimmen, zwischen ihnen Kanten erstellen und die nächste neue actualTarget-Knoten suchen und die vorherige Zielknoten der Variable actualSource zuweisen. Dies wird in der Funktion getNextNode() implementiert. Eine wichtige Ergänzung zum vorgegebenen Pseudocode wurde eingeführt: Bis wir die Anzahl der Knoten in den Iterationen nicht überschreiten, verwenden wir nicht die Funktion getNextNode(), die Knoten zufällig auswählt, falls sie nicht in posHashMap vorhanden sind. Stattdessen entnehmen wir der Reihe nach Knoten aus der Knotenliste, nachdem wir diese Liste zuvor gemischt haben. Auf diese Weise stellen wir sicher, dass alle Knoten gewählt werden und der Graph zusammenhängend bleibt. Wir verbinden beim Erreichen der vorletzten Kantennummer den aktuellen Knoten mit dem Startknoten, um den Kreis zu schließen.

5 Tests

5.1 Fest definierte Testgraphen

5.1.1 False

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

Für den Hierholzer Algorithmus dürfen keine gerichteten Graphen übergeben werden. Der erste Test prüft dies. Wird ein gerichteter Graph übergeben, beendet sich der Algorithmus und wirft eine Exception.

Der zweite zu testende Graph3 ist das Haus vom Nikolaus. Für diesen Graphen ist ein Eulerweg zu finden, allerdings kein Eulerkreis. Der Hierholzer Algorithmus schlägt hier also fehl. Beim ersten Durchlauf findet dieser den Kreis mit den Knoten A, B und C. Danach B, E, C, D. Anschließend können keine weiteren Kanten markiert werden ohne bisher markierte nochmal zu besuchen.
Der implementierte Algorithmus wirft in diesem Fall eine Exception.

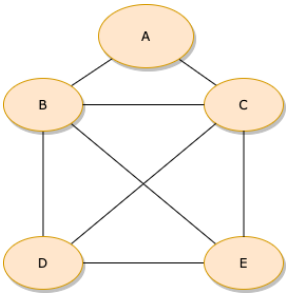

Abbildung 3: Haus vom Nikolaus - ungerade Knotengerade

Der dritte Graph4 besteht aus zwei Komponenten. Diese hängen nicht zusammen. Folglich muss der Algorithmus auch hier abbrechen, da dies nicht erlaubt ist. Wenn nicht alle Komponenten zusammenhängen, können auch nicht alle Kanten vom einem Startpunkt aus markiert werden.

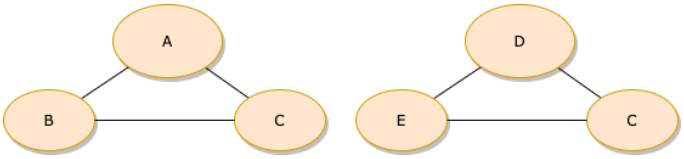

Abbildung 4: Ein Graph mit zwei nicht zusammenhängenden Komponenten

Es folgt ein Graph5 welcher mehr Knoten als Kanten besitzt. Auch in diesem Fall kann kein Eulerkreis zu finden sein. Wie im Beispiel zu sehen ist können hier der Start- und Endknoten niemals verbunden sein.

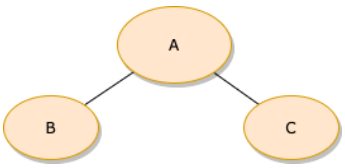

Abbildung 5: Ein Graph mit mehr Knoten als Kanten

5.1.2 True

Der erste Test in welchem der Hierholzer durchlaufen soll testet einen leeren Graphen. Wir geben vor, dass Graphen, welche weder Knoten noch Kanten haben, ein leeres Ergebnis liefern und keine Exception werden.

Der nächste Graph6 besteht aus einem einzigen Knoten. Der Algorithmus findet keine Kante, aber einen Startknoten. Da der Startknoten hierdurch auch zum Endknoten wird, wird auch hier keine Exception geworfen und der Algorithmus läuft durch.

Abbildung 6: Ein Graph mit einem einzigen Knoten

Dieser Graph7 erinnert an eine Sanduhr. Zusätzlich hat er hat mehr Kanten als Knoten. Im Vergleich zum Test bei welchem mehr Knoten als Kanten im Graphen waren, können in diesem Graphen zwei Eulerkreise gefunden werden. Vorausgesetzt als Startknoten werden Knoten A und C genutzt enthält der erste Eulerkreis die Knoten A, B und C - der zweite C, D und E.

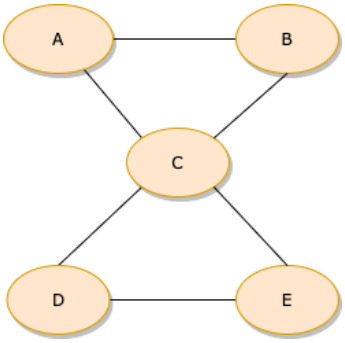

Abbildung 7: Ein Graph mehr Kanten als Knoten

Folgender Graph8 enthält eine Schleife. Auch in diesem Beispiel findet der Hierholzer Algorithmus zwei Eulerkreise. Der Knoten A ist durch eine Schleife mit sich selbst verbunden und bildet somit den ersten Eulerkreis. Der zweite Eulerkreis enthält die Knoten A, B und C.

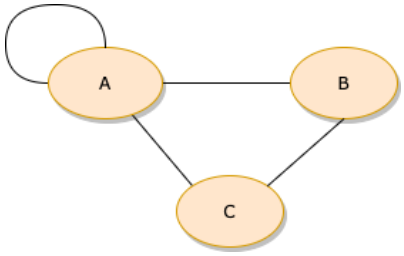

Abbildung 8: Ein Graph mit einer Schleife

Der letzte statische Graph9 ist ein Multigraph. Dieser enthält zwei Knoten, welche durch zwei Multikanten miteinander verbunden sind. Das hier ein Eulerkreis gefunden wird ist nach den letzten Tests offensichtlich. Dieser Test prüft hauptsächlich ob die Multigraphen der Bibliothek Graphstream richtig verarbeitet werden.

Abbildung 9: Ein Graph mehr Kanten als Knoten

5.2 Zufällig generierte Testgraphen

Beim Testen von zufällig generierten Testgraphen entstehen verschiedenste Probleme.
Um aber vorerst eine paar Teilfunktionalitäten des GraphGenerators zu testen wurden Tests implementiert, welche die an den GraphGenerator übergebenen Parameter testet. So wirft dieser zum Beispiel bei weniger übergebenen Kanten als Knoten eine Exception. Außerdem auch, wenn ein alleinstehender Knoten als einzige Komponente des Graphen erstellt werden soll.
In der präsentierten Implementierung bringen sowohl der GraphGenerator als auch der Hierholzer Algorithmus eine gewisse SZufälligkeit"mit sich. Dadurch ist es nicht möglich anhand der übergebenen Kanten- und Knotengröße zu bestimmen, wie viele Eulerkreise der Hierholzer Algorithmus finden wird. Zusätzlich wird die Kantenanzahl im GraphGenerator erhöht, wenn sich mit den übergebenen Parametern keine Eulergraphen bilden lassen. Auch dies erschwert die Vorhersage.
Werden zum Beispiel 10 Knoten und 11 Kanten übergeben erhöht sich die Kantenanzahl auf 12, da es sonst Knoten mit ungradem Knotengrad gäbe.

Um dies genauer zu zeigen folgender Test:

```
@Test
public void graphGenerator() {
    Graph graph = GraphGenerator.createEulerGraph(10,11, "foo");
    System.out.println("Edgecount:" + graph.edges().count());

    List<Graph> eulerianCircles = Hierholzer.findEulerGraphs(graph);

    System.out.println("Euleriancircles:" + eulerianCircles);
}
```

Die erste Ausgabe:
Edgecount: 12
Euleriancircles: [Teilgraph_edge0, Teilgraph_edge1, Teilgraph_edge9]

Die zweite Ausgabe:
Edgecount: 12
Euleriancircles: [Teilgraph_edge3, Teilgraph_edge9]

Die dritte Ausgabe:
Edgecount: 12
Euleriancircles: [Teilgraph_edge9]

Zu sehen ist, dass selbst bei sehr kleinen Graphen schon zu erkennen ist, dass die Ergebnisse nicht an den übergebenen Parametern festgemacht werden können.
Der Vollständigkeit halber folgen drei Versuche mit einem größeren Graphen. Auch hier sind aber keine nutzbaren Werte zu erkennen.

Mit 100 Knoten und 10000 Kanten:

Die erste Ausgabe:	**Die zweite Ausgabe:**	**Die dritte Ausgabe:**
Edgecount: 10001	*Edgecount: 10001*	*Edgecount: 10001*
EulerianCirclesCount: 89	*EulerianCirclesCount: 103*	*EulerianCirclesCount: 112*