**Aufgabe 3: Eulerkreise**

**Team**: GKAP05\_TeamB(Jonathan Aldag, Michel Brüger)

**Aufgabenbereiche:**

**Beide Teammitglieder haben an folgenden Dateien mitgearbeitet:** EulerCycleAlgorithms.java

**Quellen**:

* **Pseudocode/Algorithmusstruktur:**
  + https://www.geeksforgeeks.org/fleurys-algorithm-for-printing-eulerian-path/
  + https://de.wikipedia.org/wiki/Algorithmus\_von\_Hierholzer
  + https://www.geeksforgeeks.org/bridge-in-a-graph/
  + Vorlesungsfolien zu beiden Algorithmen

**Bearbeitungszeiträume:**

* Jonathan Aldag: 8.6: 4h, 9.6: 6h, 10.6: 2h
* Michel Brüger: ca. 11h
* Gemeinsame Bearbeitungszeit: 10.6: 4h

**Algorithmus von Fleury:**

**Beschreibung des Algorithmus:**

Der Algorithmus von Fleury erwartet einen eulerschen Graph und hat als Ausgabe eine Eulertour.

1. Es wird ein beliebiger Startknoten v0 ausgewählt. Besitzt der Graph zwei Knoten mit ungeradem Grad(hier ist nur ein Eulerpfad möglich), so muss einer dieser als Startknoten ausgewählt werden.
2. Man habe bisher einen Kantenzug v0e1v1…eivi gewählt. Die Kanten sind dabei disjunkt. Die Kante ei+1 ist so zu wählen, dass ei+1 inzident mit vi ist und keine Brücke im Graphen ohne alle Kanten e1,…,en ist. Eine Brückenkante wird nur gewählt, wenn es keine andere Wahl gibt.
3. Wenn alle Kanten durchlaufen sind, ist der Algorithmus beendet. Sonst gehe zu Schritt 2.

**Implementierung des Algorithmus:**

Um den Graphen ohne die bisherigen Kanten darzustellen, haben wir den Inputgraphen kopiert. In der Hauptschleife, die nacheinander Kanten hinzufügt, werden diese aus der Kopie gelöscht. Zum Festhalten der bisher abgelaufenen Kante haben wir eine Liste verwendet.

Die Hauptschleife terminiert, wenn alle Kanten des Graphen in der Liste enthalten sind. Zwei Variablen werden verwendet, um die gerade betrachtete Kante und den letzten Knoten zu speichern: *nextEdge*, *currentNode*.

Die Variable *currentNode* wird vor der Hauptschleife mit einem zufälligen Knoten initialisiert. Im Falle eines Graphen mit zwei Knoten ungeradem Grad wird einer dieser beiden Knoten zufällig als Initialwert ausgewählt.

Innerhalb der Schleife werden nun nacheinander die Kanten von *currentNode* im restlichen Graph betrachtet. Wird eine Kante gefunden, die keine Brücke ist, so wird diese als nächstes in die Ergebnisliste eingefügt. Wird nur eine Brücke gefunden, wird diese eingefügt. Die zugefügte Kante wird aus dem Graph gelöscht und *currentNode* wird auf den gegenüberliegenden Knoten dieser Kante gesetzt. Die Schleife beginnt von Neuem.

Eine Funktion *isBrigde* wird verwendet, um die Brückeneigenschaft einer Kante innerhalb eines bestimmten Graphen festzustellen. Sie lässt eine Tiefensuche über den Graphen laufen, ausgehend von einem der beiden Knoten dieser Kante. Diese Tiefensuche gibt die durchlaufenden Knoten zurück. Nun wird die Kante aus dem Graphen entfernt und es wird die gleiche Tiefensuche gestartet. Ist das Ergebnis beim zweiten Durchlauf kleiner, so handelt es sich bei der betrachteten Kante um eine Brücke.

Ist die Schleife durchlaufen, wird die Ergebnisliste als die gefundene Eulertour zurückgegeben.

**Algorithmus von Hierholzer:**

**Beschreibung des Algorithmus:**

Der Algorithmus von Hierholzer erwartet einen zusammenhängenden Graphen, der nur Knoten mit geradem Grad enthält und somit einen Eulerkreis beinhaltet.

1. Im ersten Schritt wird ein beliebiger Startknoten *v0* ausgewählt. Von diesem ausgehend wird ein Kreis im Graphen gebildet. Es wird abgebrochen, wenn dieser bereits alle Kanten beinhaltet.
2. Alle Kanten dieses Kreises werden nun vernachlässigt. An einem Eckpunkt des bisherigen Kreises *v1* mit positivem Grad lässt man einen weiteren Kreis entstehen.
3. Diesen neu erstellten Kreis fügt man in den bisher existierenden Kreis ein, indem *v1*im bisherigen Kreis durch diesen neuen Kreis ersetzt wird.
4. Gehe zu 2., falls der entstandene Kreis nicht alle Kanten enthält. Andererseits ist der Algorithmus fertig.

**Implementierung des Algorithmus:**

Unsere Implementierung arbeitet primär auf zwei Listen – eine für die bereits abgelaufenen Kanten und eine für die dabei abgelaufenen Knoten. Letztere wird verwendet, um leicht einen Startknoten für einen neuen Kreis innerhalb des Graphen zu identifizieren.

Vor der Hauptschleife, welche nacheinander Unterkreise hinzufügt, sind die Initialwerte für den Startknoten, die Startkante und den Startindex dieser beiden Objekte in ihren jeweiligen Listen definiert. Diese sind immer einen Index voneinander entfernt, deshalb übernimmt dies die Variable *startingIndex* alleinig. Die Startkante ist dabei noch nicht in ihre Liste hinzugefügt, der Knoten wird jedoch in die Liste abgespeichert.

Eine innere Schleife arbeitet nun mit Variablen *currentEdge* und *currentNode*. Sie speichern, an welchem Knoten bzw. welcher Kante der Unterkreis angekommen ist.

Erst wird die Startkante der Liste für abgelaufene Kanten hinzugefügt. Anschließend wird der Nachbarknoten zum jetzig betrachteten Knoten *currentNode* ausgelesen und dieser in die Liste für abgelaufene Knoten eingefügt. Der mitgetragene Index in die beiden Listen wird erhöht. Ist *currentNode* gleich dem anfänglich definierten Startknoten, so wird die innere Schleife abgebrochen und ein neuer Unterkreis ist gefunden.

Anschließend wird eine zufällige noch nicht abgegangene Kante ausgehend von diesem Nachbar gesucht. Kann eine solche nicht gefunden werden, hat der Graph keinen Eulerkreis und es wird eine leere Liste an Kanten zurückgegeben. Die innere Schleife wird nun wiederholt.

Nachdem ein solcher Unterkreis gefunden ist, wird geprüft, ob alle Kanten abgelaufen wurden. Ist dies nicht der Fall, so muss eine neue Startkante und ein neuer Startknoten gefunden werden. Anderweitig bricht die äußere Schleife ab und es wird die Kantenliste als Eulerkreis zurückgegeben. Das Aufwinden des neuen Startknotens geschieht mithilfe der Liste, die alle abgelaufenen Knoten speichert. Diese werden nacheinander durchlaufen und es wird geprüft, ob eine Kante existiert, die noch nicht benutzt wurde. Ist dies der Fall, werden die entsprechenden Variablen *currentNode* und *currentEdge*, sowie *startingIndex*, gesetzt. Wird hier kein passender Knoten gefunden, wird wieder eine leere Liste zurückgegeben, da der Graph keinen Eulerkreis beinhalten kann.

**Konstruktion Eulergraphen:**

**Vorgehensweise für die Konstruktion:**

Aus einer Menge von nicht miteinander verbundenen Knoten werden zufällig zwei ausgewählt und miteinander verbunden. Dann wird immer je ein weiterer Knoten aus der Menge nicht verbundener Knoten zufällig ausgewählt und mit einem ebenfalls zufällig ausgewählten Knoten der Zusammenhangskomponente verbunden. Dies wird so lange fortgeführt bis ein Spannbaum alle Knoten miteinander verbindet.

Anschließend werden je zwei Knoten ungeraden Grades, zwischen denen noch keine Kante besteht, zufällig ausgewählt und miteinander verbunden. Sollten nur noch zwei Knoten ungeraden Grades übrig sein die beide schon per Kante verbunden sind, so wird entweder, falls möglich, ein anderer Knoten der Zusammenhangskomponente gewählt, zu dem von keinem der beiden Knoten eine Kante besteht und dieser wird mit beiden Knoten verbunden, oder es wird ein neuer Knoten zum Graph hinzugefügt welcher dann mit den beiden Knoten ungeraden Grades verbunden wird.

Durch das paarweise verbinden von Knoten ungeraden Grades ergibt sich am Ende ein Graph der ausschließlich aus Knoten graden Grades besteht. Da zusammenhängende Graphen immer nur eine grade Anzahl von Knoten ungeradem grad haben können bleibt auch nie ein Knoten ungeraden Grades übrig. Werden im beschriebenen Sonderfall die beiden letzten Knoten mit einem dritten Knoten graden Grades verbunden, so ändert das auch nichts, da dieser dann 2 neue Kanten hinzugefügt bekommt, der Grad also gerade bleibt.

**Implementierung:**

**Eulerkreise identifizieren:**

Eine gegebene Kantenfolge ist dann ein Eulerkreis, wenn:

1. Die Kantenfolge genauso viele Kanten und Knoten enthält wie der Ursprungsgraph.
2. Kante n immer **einen** Knoten enthält den auch Kante n+1 enthält (keine Schleifen).
3. Keine Kanten doppelt enthalten sind
4. Die letzte Kante den Startknoten enthält.

**Testfälle:**

Die Algorithmen werden mit der Funktion

*public static boolean* isEulerCycle(Graph graph, ArrayList<Edge> edges){

getestet, welche die Kantenfolge relativ zu dem Graph auf die genannten Kriterien (s.o. 1.-4.) zur Eulerkreisidentifikation testet.

**Auswirkungen des Test-Driven Developments:**

Das Testen der ersten paar groben Entwürfe des Algorithmus hat sich nicht maßgeblich von einem normalen Ansatz unterschieden. Im Endeffekt mussten wir uns eine der kleinen Testdateien hernehmen und Schritt für Schritt den Algorithmus mitsamt Fehlerbildung nachvollziehen. Dazu eignet sich in diesem Stadium auch noch die Visualisierung gut. Allerdings war im Anschluss schnell klar, zu welchem Zeitpunkt der Algorithmus von allen kleineren Makeln befreit wurde. Das gibt ein Gefühl von Sicherheit in seine Lösung und ermöglicht ein klares Abhaken einer Teilaufgabe.

Für das Zeitmanagement hat uns diese Entwicklungsart ebenfalls stark weitergeholfen. Fehler, die erst spät in der Testphase entdeckt werden, benötigen teilweise eine komplette Umstrukturierung des Codes. Dies kostet enorm Zeit und wirft evtl. eine persönliche Planung komplett durcheinander oder kann sogar ein rechtzeitiges Fertigstellen gefährden. Während dieser Aufgabe konnten wir den Restaufwand zu jedem Zeitpunkt mit Zuversicht abschätzen.