GKAP Praktikum 04 15.06.2015

– Team: Teamnummer 2_5 - Dimitri Meier, Saeed Shanidar

– Aufgabenaufteilung:

(a) Aufgaben, für die Dimitri Meier verantwortlich ist:
   a. Fleury Algorithmus
   b. Hierholzer Algorithmus
   c. Eulergraph-Generator
   d. Tests

(b) Aufgaben, für die Saeed Shanidar verantwortlich ist:
   a. Dokumentation
   b. Theorie Aufgaben
   c. Eulergraph-Generator

(c) Aufgaben, für die Dimitri Meier und Saeed Shanidar verantwortlich sind:
   a. Fleury Algorithmus
   b. Hierholzer Algorithmus
   c. Eulergraph-Generator
   d. Tests

– Quellenangaben:

Breitensuche (BFS)-Algorithmus http://de.wikipedia.org/wiki/Breitensuche

Dijkstra-Algorithmus: http://de.wikipedia.org/wiki/Dijkstra-Algorithmus

A*-Algorithmus: http://de.wikipedia.org/wiki/A*-Algorithmus

Prim-Algorithmus: http://de.wikipedia.org/wiki/Algorithmus_von_Prim

Kruskal-Algorithmus: http://de.wikipedia.org/wiki/Algorithmus_von_Kruskal

http://www-m9.ma.tum.de/Allgemeines

Fleury-Algorithmus:
http://de.wikipedia.org/wiki/Eulerkreisproblem#Algorithmus_von_Fleury

Hierholzer-Algorithmus: http://de.wikipedia.org/wiki/Algorithmus_von_Hierholzer

Zusätzlich Die Vorlesungsfolien:

- https://pub.informatik.haw-hamburg.de/home/pub/prof/padberg_julia/Home_GKA_SoSe15/Folien/vl04.pdf

- https://pub.informatik.haw-hamburg.de/home/pub/prof/padberg_julia/Home_GKA_SoSe15/Folien/vl05.pdf

- Verwendete Library:

Java Universal Network/Graph Framework: JUNG http://jung.sourceforge.net/

Hipster heuristic search for Java: http://www.hipster4j.org/

JGraphT: http://jgrapht.org/

– Aktueller Stand:

- GUI: wurde erweitert
- Algorithmus:
  - BFS fertig Implementiert.
  - Dijkstra fertig Implementiert
  - A* fertig Implementiert
  - Kruskal fertig Implementiert
  - Prim fertig Implementiert
  - Fibonacci-Heap fertig Implementiert
  - Fleury fertig Implementiert
  - Hierholzer fertig Implementiert
  -
- Graph-Generator
  - Ungerichtet & Gewichtet-Generator fertig Implementiert
  - Ungerichtet & Gewichtet & Attribuiert-Generator fertig Implementiert
  - Zusammenhängende Ungerichtet & Gewichtet-Generator fertig Implementiert
  - Eulergraph-Generator fertig Implementiert

- Tests:
  - GraphBuilder und den Algorithmus ausführlich getestet.
  - Test auf randomisierte unterschiedlich große Graphen mit Kruskal und Prim, wobei die Gesamtweglänge der Ergebnis Spannbaume gleich sein müssen.

- Dokumentation:
    - Praktikums 01 : Fertig
    - Praktikums 02 : Fertig
    - Praktikums 03 : Fertig
    - Praktikums 04 : Fertig

Algorithmus von Kruskal

Der Algorithmus wird mit einem Wald, welcher aus Bäumen mit einzelnen Knoten besteht, initialisiert. Die Kanten werden ihrem Gewicht nach in einer Warteschlange sortiert. In jeder Iteration wird eine Kante aus der Warteschlange entfernt. Falls die Endpunkte der Kante zu verschiedenen Bäumen gehören, werden diese über die Kante vereinigt. Dies wird so lange wiederholt, bis alle Knoten zum selben Baum gehören oder keine Kanten mehr in der Warteschlange sind.

Die effizienteste Implementierung des Algorithmus funktioniert mit einer "Union-Find" Datenstruktur. Diese bietet effiziente Mengenoperationen wie beispielsweise die Vereinigung zweier Mengen oder die Überprüfung, ob etwas in einer Menge enthalten ist. Wenn eine Kante aus der Warteschlange entfernt wird, ermöglicht diese Datenstruktur die Überprüfung, ob die zwei Endpunkte der Kante zum selben Baum gehören. Sollte das der Fall sein, so werden die Bäume (welche als Mengen repräsentiert werden) einfach vereinigt.

Der Algorithmus wird zuerst initialisiert:

- Alle Kanten werden entsprechend ihres Gewichts sortiert und in eine Warteschlange eingefügt.
- Jeder Knoten des Graphen wird zur "Union-Find" Datenstruktur hinzugefügt. Die Datenstruktur weißt jedem Knoten eine ID zu (hier dargestellt über die Farbe). Diese ID repräsentiert den Baum, zu welchem dieser Knoten gehört. Während der Ausführung des Algorithmus werden diese IDs über die Farben der Knoten dargestellt.

Anschließend arbeitet der Algorithmus die Warteschlange mit den Kanten ab. In jeder Iteration wird die Kante am Anfang der Warteschlange entnommen. Dann werden die zwei Endpunkte der Kante verglichen. Falls ihre IDs nicht gleich sind (das heißt falls sie zu unterschiedlichen Bäumen gehören), dann werden die zu den Endpunkten gehörigen Bäume vereinigt. Die "Union-Find" Datenstruktur vereinigt die beiden Bäume, indem allen Knoten in beiden Bäumen eine neue ID zugewiesen wird. In diesem Fall wird die Kante zum Spannbaum hinzugefügt. Ansonsten sind die Endpunkt im gleichen Baum; wenn die Kante hinzugefügt würde, wäre im Baum anschließend ein Kreis enthalten.

Die beschriebene Schleife wird so lange wiederholt, bis keine Kanten mehr in der Warteschlange übrig sind oder alle Knoten im selben Baum enthalten sind (das heißt ihre IDs gleich sind).

Algorithmus von Prim

Der Algorithmus beginnt den Baum mit einem beliebigen einzelnen Knoten, und fügt dann in jeder Iteration eine Kante hinzu. Dies hinzugefügte Kante hat unter allen Kanten, welche den Baum mit Knoten außerhalb des Baums verbinden, minimales Gewicht. Dieser Vorgang wird so lange wiederholt, bis alle Knoten im Baum sind.

Die effizienteste Implementierung des Algorithmus erfolgt über eine Warteschlange. Diese Warteschlange speichert alle Knoten, welche schon besucht wurden, aber noch nicht in den Baum eingefügt sind. Die Reihenfolge der Knoten in der Warteschlange wird über die Distanz jedes Knoten zum Baum bestimmt. In jeder Iteration wird also der Knoten, welcher über die billigste Kante mit dem Baum verbunden ist, aus der Warteschlange entfernt.

Die schrittweise Beschreibung des Algorithmus hat folgende Form. Zuerst werden alle Werte initialisiert:

- Ein beliebiger Knoten wird als Wurzel des Baums ausgewählt.
- Eine Distanztabelle speichert die Distanzen aller Knoten zum Baum. Die Wurzel hat Distanz Null, und da alle anderen Knoten keine Verbindung zum Baum haben, wird ihre Distanz auf Unendlich gesetzt.
- Die Wurzel wird zur Warteschlange hinzugefügt.

Als nächstes behandelt der Algorithmus alle Elemente der Warteschlange. In jeder Iteration wird der erste Knoten der Warteschlange (mit der minimalen Distanz) aus der Schlange entfernt. Zu Beginn ist das die Wurzel, später andere Knoten. Danach werden alle Nachbarn des entfernten Knoten mit ihren sie zum entfernten Knoten verbindenden Kanten betrachtet. Für jeden dieser Knoten wird überprüft, ob das Gewicht dieser verbindenden Kante geringer ist als der aktuelle Distanzwert. Falls das der Fall ist, wird der Distanzwert dieses Nachbarn auf den neuen, geringeren Wert gesetzt. Ansonsten wird der Knoten noch in die Warteschlange eingefügt, falls er das noch nicht ist. Somit werden die ihn verlassenden Kanten später noch betrachtet werden.

Diese Schritte werden so lange wiederholt, bis keine Knoten mehr in der Warteschlange enthalten sind. Dann ist der Algorithmus fertig und gibt als Ergebnis alle Kanten des minimalen Spannbaums zurück.

Algorithmus von Fleury

Im Algorithmus von Fleury spielen Brückenkanten eine wichtige Rolle. Das sind Kanten, ohne die der Graph in zwei Zusammenhangskomponenten zerfallen würde.

Der Algorithmus fügt einer anfangs leeren Kantenfolge alle Kanten eines Graphen hinzu, sodass ein Eulerkreis entsteht.

1. Wähle einen beliebigen Knoten als aktuellen Knoten.
2. Wähle unter den unmarkierten, mit dem aktuellen Knoten inzidenten Kanten eine beliebige Kante aus. Dabei sind zuerst Kanten zu wählen, die im unmarkierten Graphen keine Brückenkanten sind.
3. Markiere die gewählte Kante und füge sie der Kantenfolge hinzu.
4. Wähle den anderen Knoten der gewählten Kante als neuen aktuellen Knoten.
5. Wenn noch unmarkierte Kanten existieren, dann gehe zu Schritt 2.

Ob eine Kante eine Brückenkante ist, kann mittels Bereitensuche(BFS)-Algorithmus in Laufzeit O(|E|) überprüft werden.

Da pro Schritt eine Kante markiert und Imaginär entfernt wird, benötigen wir |E| Iterationen. Die Anzahl der pro Iteration geprüften Kanten entspricht dem Grad des aktuellen Knotens. Insgesamt kann man die gesamte Anzahl überprüfter Kanten durch O(|E|) beschränken. Die gesamte Laufzeit ist damit von der Größenordnung $O(|E|)^2$.

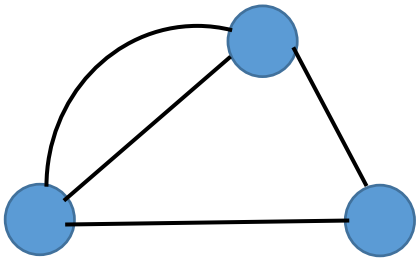
Algorithmus von Hierholzer

Mit dem Algorithmus von Hierholzer lässt sich eine Euler-Tour in einem eulerschen Graphen G(V; E) bestimmen. Der Algorithmus benutzt die Tatsache, dass sich eulersche Graphen in schrittweise Kantendisjunkte Zyklen zerlegen lassen.

Theorieteil 4

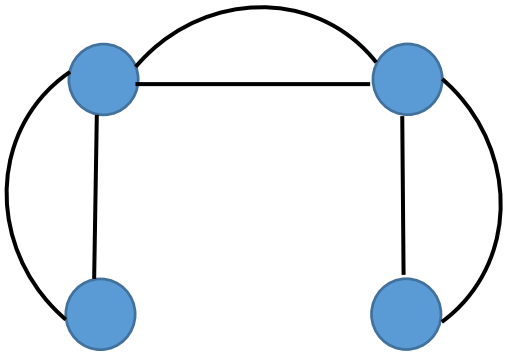
Aufgabe X:

1. Geben Sie bitte mit Begründung einen zusammenhängenden Graphen an, der einen Hamiltonkreis, aber keinen Eulerkreis enthält.



Bei **Hamiltonkreis** ist ein geschlossener Pfad in einem Graphen, der jeden Knoten genau einmal enthält. Im Gegensatz zum **Eulerkreis**, bei dem alle Kanten genau einmal durchlaufen werden

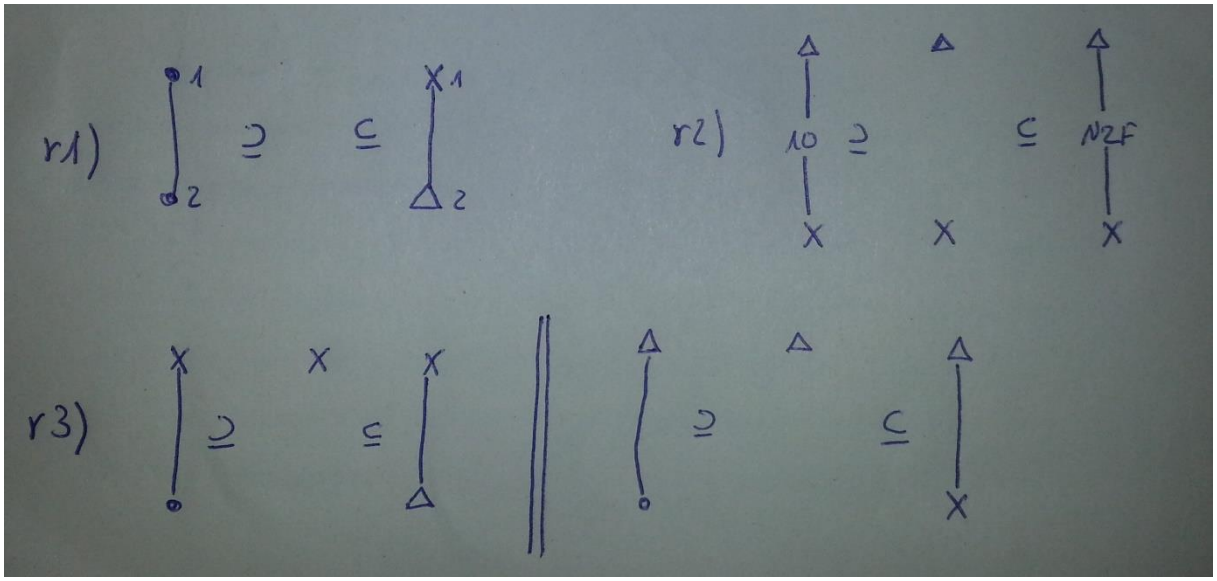
2. Geben Sie bitte mit Begründung einen zusammenhängenden Graphen an, der einen Eulerkreis, aber keinen Hamiltonkreis enthält.



Bei **Hamiltonkreis** ist ein geschlossener Pfad in einem Graphen, der jeden Knoten genau einmal enthält. Im Gegensatz zum **Eulerkreis**, bei dem alle Kanten genau einmal durchlaufen werden

Aufgabe XI:

1. Geben Sie eine Graphgrammatik 2F an, die für einen gegebenen Startgraphen G eine 2-Färbung erzeugt, wobei die Regeln solange wie möglich angewendet werden. Wenn es keine 2-Färbung gibt, dann soll ein Knoten mit dem Label N 2 F erzeugt und abgebrochen werden.



Aufgabe XII:

Die Brauerei braut Bier und stellt die Fässer in ihr kleines Lager. Das Lager der Brauerei fasst jedoch nur 40 Fässer. Die Fässer werden mit einem der drei Pferdewagen zum Gasthof transportiert. Ein Pferdewagen transportiert genau 10 Fässer Bier. In der Gaststätte lassen sich aus einem Fass 50 Gläser Bier zapfen. Die Kellnerin kann maximal 6 Gläser tragen, geht aber nur los, wenn mindestens 3 Gläser gefüllt auf dem Tresen stehen.
Modellieren Sie dieses Szenario bitte mit Hilfe eines Stellen/ Transitionsnetzes