GKAP Praktikum 01 13.04.2015

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

–**Aufgabenaufteilung**:

(a) Aufgaben, für die Dimitri Meier verantwortlich ist:
Implementierung / Bearbeitung der GUI, Algorithmus (BFS) und Dokumentation.

(b) Aufgaben, für die Saeed Shanidar verantwortlich ist:
Implementierung / Bearbeitung von Algorithmus (BFS) und Dokumentation.

– Quellenangaben:

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

- Verwendete Library:

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

– Bearbeitungszeitraum:

- Dimitri Meier: Dauer der Bearbeitung und kennenlernen der
  - GUI ca. 2 Stunden
  - ADT ca. 8 Stunden
  - Umstieg (JGraphT/ JGraphX) -> JUNG und Einarbeitung in die Libraries-> ca. 10 stunden

- Saeed Shanidar: Dauer der Bearbeitung der
  - Dokumentation ca. 3 Stunden
  - Theorie Aufgaben ca. 3 Stunden

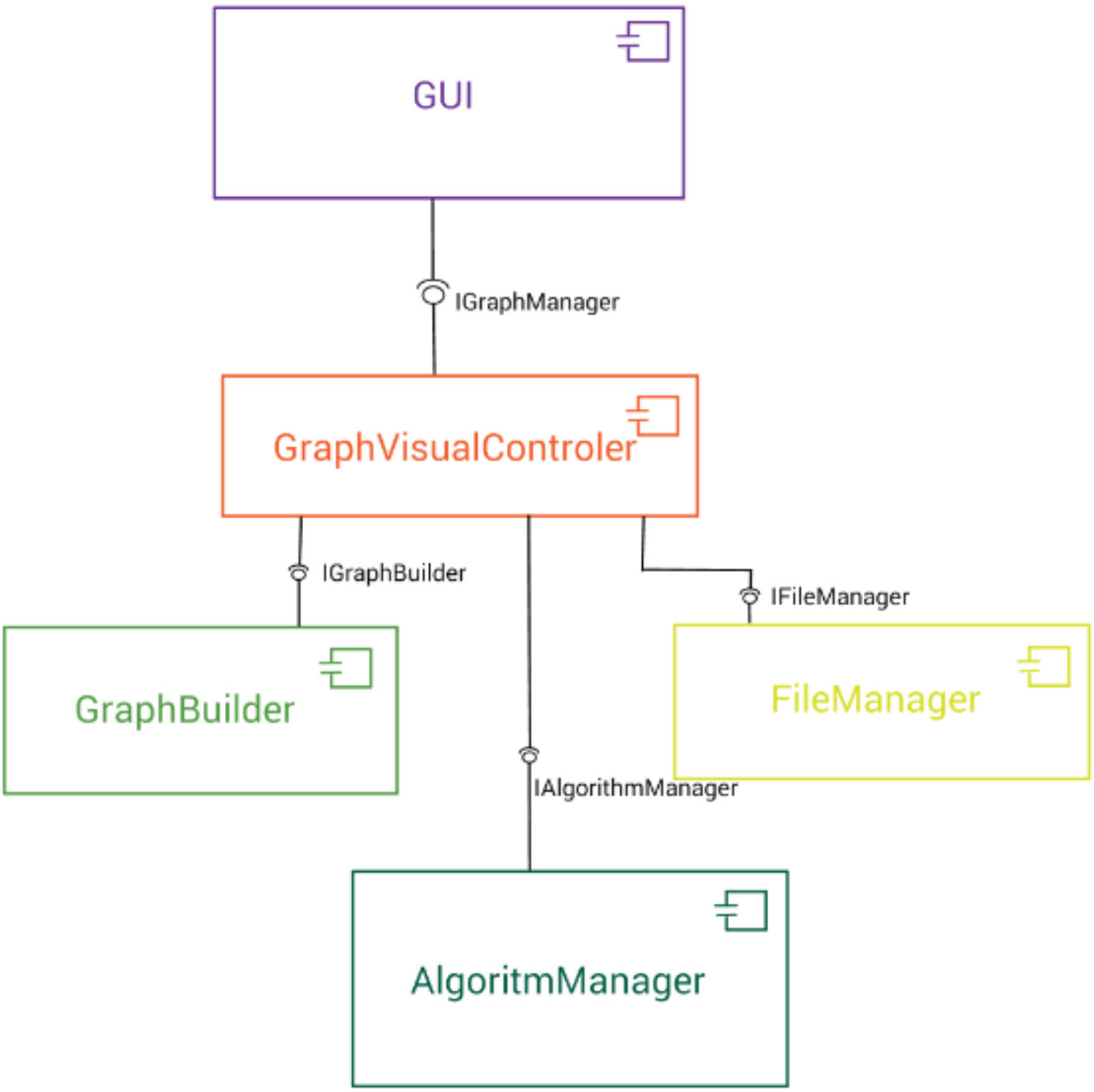
- Gemeinsame Bearbeitungszeiten: Dauer der Bearbeitung der
  - Algorithmus ca. 8 Stunden
  - Dokumentation ca. 1,5 Stunden

– Aktueller Stand:

- GUI: fertig Implementiert.
- Algorithmus: BFS fertig Implementiert.
- Tests: GraphBuilder und den Algorithmus ausführlich getestet.
- Dokumentation: fertig.

Dokumentation:

UML Komponenten Diagramm



Algorithmus:

Breadth First Search:
Der Algorithmus besteht aus zwei Funktionalitäten.

- 1: Ziel suche
- 2: kürzesten Pfad ermitteln.

Bei der Ziel Suche wird zunächst eine Stack aufgebaut, welcher die Knoten darstellt. In diesem landen alle Knoten die bei der Suche angefasst werden.
Doch vorher wird überprüft:

- Ob der Knoten schon angefasst wurde
- Ob der Knoten schon im Stack bereits enthalten ist
  Und die aktuelle Pfadtiefe dem Knoten gesetzt

Wenn der Zielknoten gefunden wurde, kommt die zweite Funktionalität ins Spiel.

Jetzt wird der Rückweg anhand des bereits aufgebauten Stacks ermittelt, indem wir den Stack Stück für Stück in einer Schleife abarbeiten.

Dabei wird pro abgebauten Knoten auf die Pfadtiefe geguckt.
Ist die Tiefe genau um eins kleiner als der letzte Knoten, der zum Pfad gehört, dann prüfen wir ob die beiden Knoten eine Verbindung beinhalten.

Wenn eine Verbindung besteht, dann muss dieser Knoten ein Teil des Pfades sein. Dies wird für alle übrigen Knoten im Stack durchgeführt.

Auf diesem Wege lässt sich der kürzeste Pfad ermitteln, Source ↔ Target.

Datenstrukturen:

JUNG Lirary, Eigene Vertex und Edge Klasse.

Implementierung:

Wir haben uns an der MVC Architektur orientiert um unser Projekt zu erstellen. Die Hauptaspekte der grundlegenden Implementierung, vom File Laden und Speichern, sowie das Erstellen eines Graphen, wurden in zwei separaten Komponenten behandelt. FileManagerKomponente kümmert sich nur um das File laden und die GraphManagerKomponente um das Erstellen eines Graphen.

Die FileManagerKomponente gibt beim Laden der Datei eine List<String> zurück. Diese enthält die Informationen des Graphen.

Die GraphManagerKomponente nimmt nun diese Liste<String> und parst die Informationen in den ADT- Graphen.

Der fertige Graph wird über die GraphManagerKomponente an die GUI übergegeben, und aufbereitet.

Genau andersrum funktioniert das Prinzip beim Speichern.
Die GraphManagerKomponente liest die Informationen aus dem ADT-Graphen aus und gibt der FileManagerKomponente die Informationen als List<String>, damit diese in eine Datei gespeichert werden können.

Tests:

- GraphBuilderTest:
  Hier haben wir rudimentäre Tests durchgeführt um das korrekte Laden der verschiedenen Graphen sicher zu stellen, sowie das Erstellen der richtigen Graphen überprüft. Dabei werden alle verschiedenen Graphen Kombinationen (directed, undirected, attributed, weighted) getestet.

- AlgorithmManagerTest:
  - Breadth First Search (BFS): Hier haben wir die Breitensuche auf ihre korrekte Funktionalität geprüft.
    Unabhängig ob der Graph gerichtet- oder ungerichtet ist, soll der Algorithmus korrekt arbeiten.

Wesentliche Entwurfsentscheidungen der Implementierung

Wir haben uns für JUNG Library entschieden, nachdem wir die Problematiken und Fehler der Library JGraphT und JGraphX festgestellt haben. Diese sind nicht nur schlecht dokumentiert und beispielslos, sondern sind auch ungünstig zum Visualisieren aufgebaut. Da uns der Aufwand der Implementierung dieser Library ins ungewisse hoch erschien, haben wir uns an die JUNG Library gewagt. Laut der bisherigen Erfahrung, scheint uns diese sehr sympathisch und äußerst gut dokumentiert zu sein. Außerdem lässt sich dank der vielen und guten Beispiele das Datenkonstrukt äußerst leicht handeln.

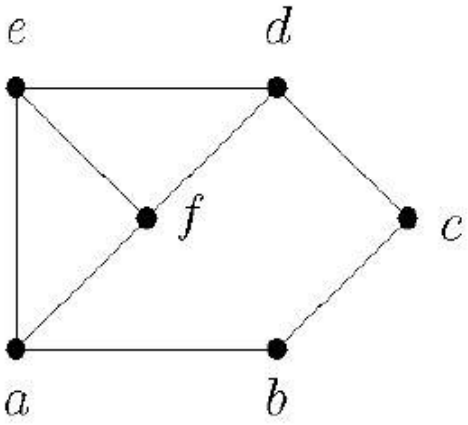
Beantwortung der Fragen:

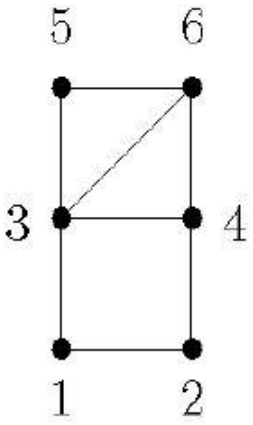
GKA Praktikum 08.04.2015

Theorieteil 1

Aufgabe I :

Sind die beiden folgenden Graphen isomorph? Geben Sie entweder einen Isomorphismus an, oder begründen Sie, warum keiner existiert.





Die folgenden beiden Graphen sind nicht isomorph, da es sich nicht um bijektive Abbildungen handelt. Es muss im linken Graph ein Knoten existieren, der genauso viele Kanten hat wie Knoten „3“ (4 Kanten) im zweiten Graph.

Aufgabe II :

Existiert ein schlichter Graph mit fünf Knoten und den folgenden Knotengraden?
Wenn ja, wie groß ist die Anzahl der Kanten?
Falls möglich, zeichnen Sie einen Graphen mit den gegebenen Eigenschaften.

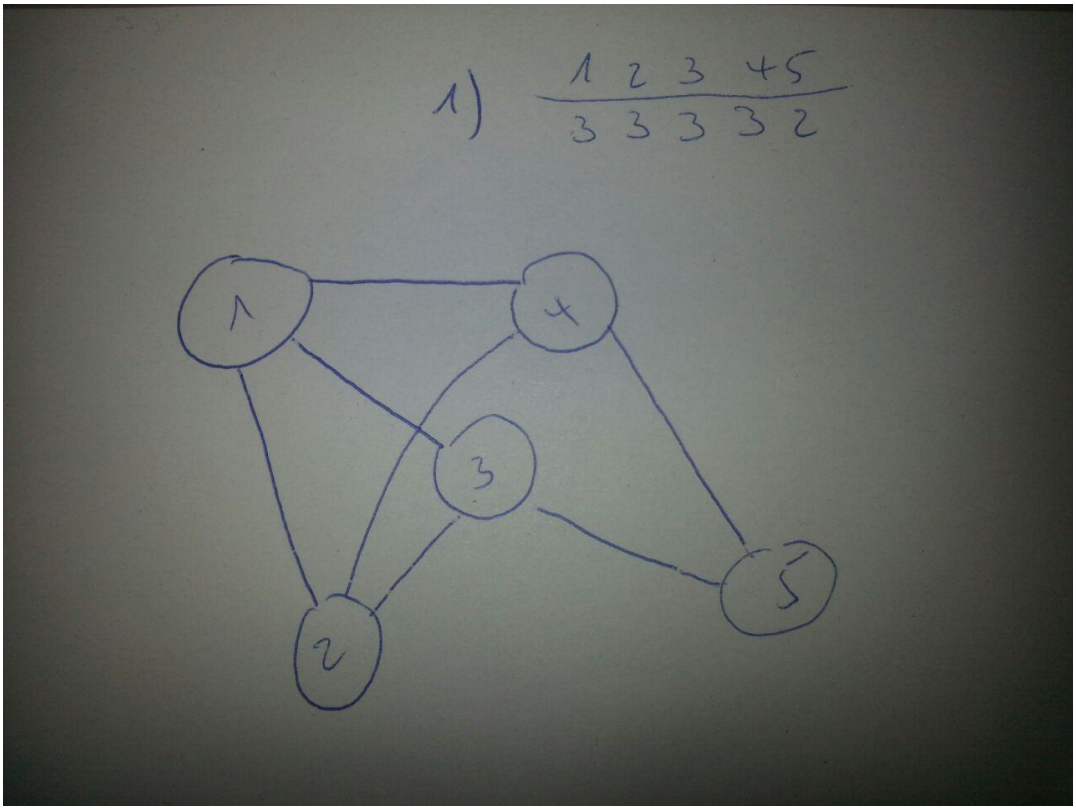
Definition

Sei $v \in V$ ein Knoten eines Graphen $G = (V, E)$.

1. Falls G ungerichtet ist, ist die Zahl $d(v)$ definiert als $d(v) = |\{e \in E | v \in s\_t(e)\}| + |\{e \in E | v \in s\_t(e) \wedge |s\_t(e)| = 1\}|$, d.h. die Anzahl der Kanten, deren Endknoten v ist (dabei werden Schlingen doppelt gezählt, was durch den zweiten Summanden zum Ausdruck kommt!). $d(v)$ heißt **Grad des Knotens** v.
2. Falls G gerichtet ist, ist die Zahl $d_-(v)$ [bzw. $d_+(v)$] definiert als $d_-(v) = |\{e \in E | s(e) = v\}|$ bzw. $d_+(v) = |\{e \in E | t(e) = v\}|$, d.h. die Anzahl der Kanten, deren Ausgangsknoten [bzw. Endknoten] v ist. $d_-(v)$ [bzw. $d_+(v)$] heißt **Ausgangsgrad** [bzw. **Eingangsgrad**] des Knotens v.

1. 3; 3; 3; 3; 2
2. 1; 2; 3; 4; 4
3. 0; 1; 2; 2; 3
4. 1; 2; 3; 4; 5

Anzahl der kanten von 1. Graph = 7

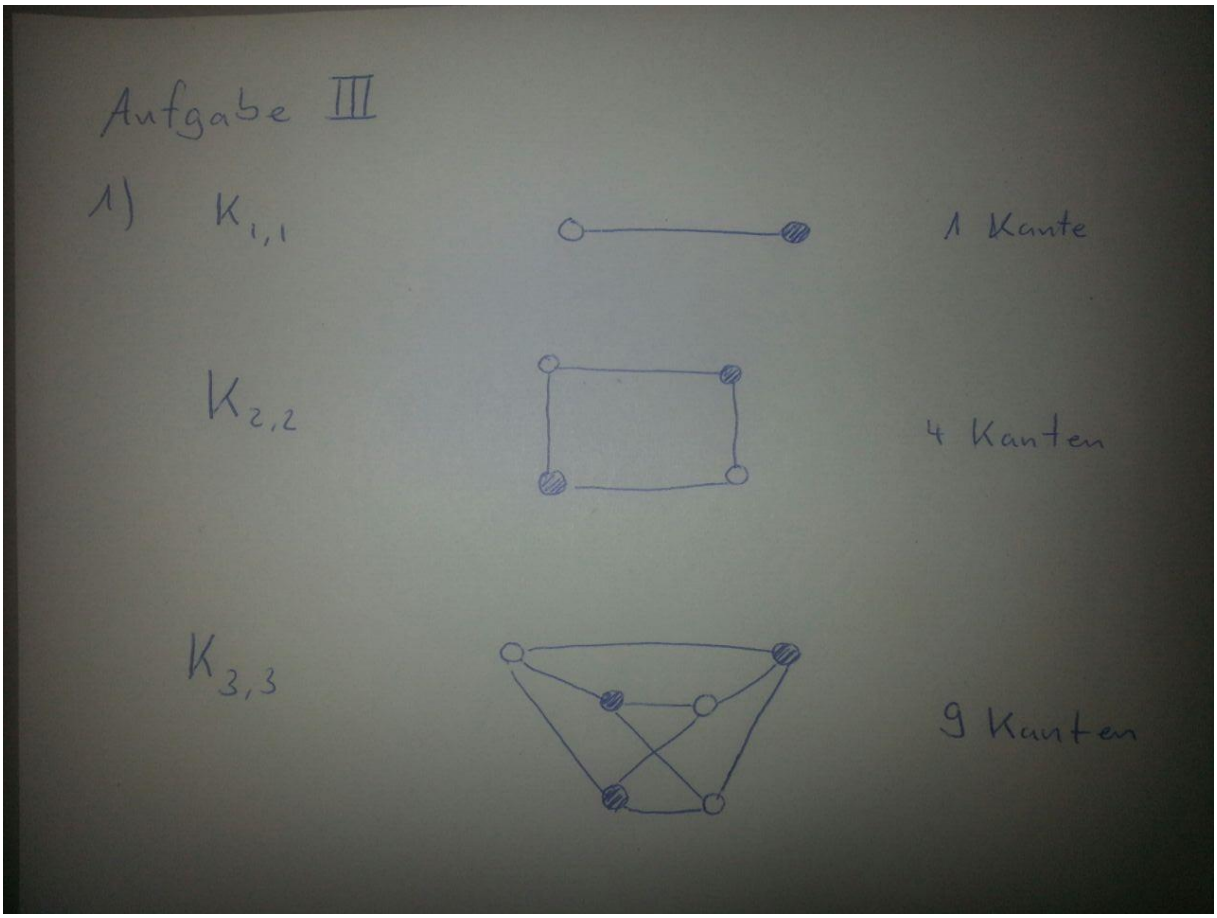


Aufgabe III :

Ein vollständiger, bipartiter Graph Kn,m hat eine Partitionierung X und Y mit
|X| = n und |Y| = m.

Definition vollständiger bipartites Graph kn,m:
Graph G heißt vollständig bipartit, falls eine Bipartien (Aufteilung in 2 Mengen, wobei keine Kante zwischen zwei Knoten eine Menge existieren darf) existieret, sodass jeder Knoten von A mit jeder Knoten von B verbunden ist. Kn,m symbolisiert die Anzahl der jeweiliger Knotenmengen.

1. Geben Sie K1,1, K2,2 und K3,3 und die Anzahl der Kanten an.



2. Bitte bestimmen Sie die Anzahl von Kanten in vollständigen, bipartiten Graphen Kn;n.

Die Anzahl der Kanten beträgt n * m

3. Beweisen Sie bitte diesen Zusammenhang.
Sei Kn,m vollständig bipartit, mit G=(V, E) und der Kante {v, w} element E, gilt entweder (v elemnt A UND w element B) ODER (v elemnt B UND w element A) , wobei A, B zwei disjunktive Teilmenge von V sind.
Dann muss jeder Knoten von A mit jeden Knoten von B verbunden sein.
Dann existieren |B| Kanten je Knoten von A, also |A| * |B|.