Team: <2>, <Dmitrij Schaefer, Flah Ahmad>

Aufgabenaufteilung:

1. <Aufgaben, für die Teammitglied 1 verantwortlich ist>, <Dateien, die komplett/zum Teil von Teammitglied 1 implementiert/bearbeitet wurden>

2. <Aufgaben, für die Teammitglied 2 verantwortlich ist>, <Dateien, die komplett/zum Teil von Teammitglied 2 implementiert/bearbeitet wurden>

Quellenangaben: <www.erlang.org; www.stackoverflow.com >

Begründung für Codeübernahme: <Keine Code Übernahme>

Bearbeitungszeitraum: <

25.03.2014 → 8h

26.03.2014 → 5,5 * 2 = 11h

27.03.2014 → 3h

28.03.2014 → 3h

29.03.2014 → 5 * 2 = 10h

30.03.2014 → 2 + 6 = 8h

31.03.2014 → 3h

01.04.2014 → 2 * 3 = 6h

Gesamtzeit = 52h

>,

<Angabe der gemeinsamen Bearbeitungszeiten>

Aktueller Stand: <fertig>

Skizze: <

Unsere Überlegung für den Graphen:

Die Graph Struktur sieht folgender Maßen aus, sie beinhaltet genau 3 Elemente und besteht selbst aus einem Tupel. Entschieden haben wir uns dafür, da für Tupeln sehr schöne Methoden schon von Erlang angeboten bekommen werden, wie die elment(N, Tupel), append, setElement usw.

Wir bleiben einheitilch und verwenden für Edges und Vertex auf weiterhin Tuples.

Struktur des Graphes = { Vertices, EdgesG, EdgesU }

Edge Struketur = {artVonEdge, { V_ID_1, V_ID_2}, {Attri, Value}, {Attr, Value} … }

ArtVonEdge = (edgeU v edgeG) | (Atome);

Vertex Struktur = {vertex, Vertex_ID, Attri, Value}, {Attr, Value} … }

Wir haben uns für ein Modul Graph_ADT entschieden, wo die alle vorgegeben Methoden implementieren.

Aktueller Stand:

Nach der Implementierung der Struktur sind wir auf viele Probleme gestoßen,

mit Tupeln ist es fuer uns viel schwieriger zu implementieren, wir haben uns doch fuer eine andere Innere Struktur im Graph entschieden und zwar Listen. Warum Listen? Weil man mit Listen sehr schön und kurz implementieren kann, Stichwort ist List comprehension.

Damit koennen wir wunderbar durch Listen iterieren und uns die benoetigen Elemente heraus filtern oder was auch immer damit machen.

Syntax Beispiel(Psydocode):

Teilmenge = [X || X ← IrgendEineGraph, AlleUngeradeKnotenIds].

So sieht jetzt die Struktur des Graphen aus:

Graph = { [Vertices], [edgesD], [edgesU] }

Die aeussere Tupel Struktur koennen wir so lassen, da wir genau drei Elemente haben und ohne Probleme mit element(N, Graph) darauf zu greifen konnen.

Implementierung des Parsers:

Grundgedanke von uns, Zeilen einlesen, Woerter aus der Zeile einlesen und sie in unsere Datenstruktur rein packen. Abbruchbedingung waere dann, wenn die eingelesene Datei leer ist.

Genau so sind wir auch vor gegangen, jedoch mit viel mehr Schwierigkeiten als erwartet.

Waren nur in der Lage das in ein Tupel ein zu lesen, war nicht so schlimm, diesen Tupel haben wir in eine Liste konvertiert und den wie gewohnt weiter verarbeiten.

Jedoch haben wir jetzt die gesamte Datei in eine Liste gehabt, die komplett in einem String in der Liste liegt.

Sowas wie einzelne Woerter einlesen konnten wir nicht finden, dies war fuer uns nicht Sinnvoll und entspricht auch nicht unseren Gedanken zum implementieren des Parsers. Somit haben wir Zeilenweise eingelesen, jedoch sah die Struktur von der Zeile sehr sehr unschön aus und zwar mit \n in einem String stehen.

Dieses haben wir mit der Token Technik von Erlang heraus gefiltern und eine drei oder vier Elementige Liste erschaffen, je nach dem in welchen Modus man die Methode aufruft, cost oder maxis.

Abbruchbedingung hat sich als nicht gerade leicht erwiesen, so eine Methode wie gib uns die Anzahl der Zeilen von einer Datei konnten wir nicht finden. Somit haben wir uns selbst so eine Methode gebaut und somit die Abbruchbedingung erfolgreich implementiert.

Wo der Parser für einen Modus implementiert war, war es für die anderen Modies nur ein Kinderspiel, bestand fast nur aus copy/past und etwas anpassen der Struktur oder veraendern der Methoden aufrufe.