Team: 2_1: Sebastian Diedrich, Tim Hagemann

Aufgabenaufteilung:

1. Sebastian Diedrich : Parser und Tests
2. Tim Hagemann : GUI und BFS Suche

Quellenangaben: [1] http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/GraphAlgor/breadthSearch.htm (Zuletzt zugegriffen: 15.10.14)

Bearbeitungszeitraum: 7.10.14 – 23.10.14

Aktueller Stand: Aufgabe vollständig implementiert, bis auf das Speichern von eingelesen Graphen. Außerdem wurde Dijkstra anstatt BFS als Suchalgorithmus implementiert.

Algorithmus für Dijkstra

Algorithmus wurde selbst erstellt mit Hilfe von [1].

```
vars Start, Queue, End, Visited
Start = a node
End = a node
Queue = a queue // Die Queue der noch zu bearbeitenden Knoten
Visited = a map // Bereits besuchte Knoten und deren zusammenhängende Werte
Put Start in Queue

while (Queue has items) // Solange es noch unbearbeitete Knote gibt
    Current = Queue removefirst
    for Node in Current // Für jeden Nachbarn vom aktuell zu bearbeitendem Knoten
        if Node == End
            break
        endif

        if not Node in Visited
            Visited add Node
        endif

        Nodescore = (Current score) + (getedgescore Current to Node) // Wert für
den aktuellen Pfad errechnen
        if Nodescore < (Visited getscore Node) // Wenn neuer Wert kleiner als alter
            Visited setparent Node to Current // Werte neu setzten
            Visited setscore Node to (Current score) + (getedgescore Current to
Node)
        endif
    endfor
endwhile

if not End in Visited
    Error "End not in graph"
else
    Success "Path found, Score: " + Visited getscore End

Endif
```

Datenstrukturen

<siehe letzte Seite>

Implementierung

Parsing

Beim Parsen wird als erstes über ein Reader die gegebenen GKA-Dateien Zeile für Zeile eingelesen und in eine Zeichenkette gespeichert. Dann wird mit Hilfe von RegEx durch diese Kette iteriert um die einzelnen Teile des Graphen zu suchen und auszuwerten, womit dann ein Graph erstellt werden kann. JGraphT stellt hierbei die Datenstruktur für den Graphen bereit.

Suchen

Die Eingabe umfasst den JGraphT-Graphen, den Startknoten und den Endknoten. Der kürzeste Weg wird in einer Liste gespeichert. Alle bis dahin besuchten Knoten werden in einer eigenen Datenstruktur gesichert (Name des Knoten, Anzahl der bisherigen Knoten, Elternknoten – der am kürzesten entfernt ist).

Graphische Darstellung

Mittels JGraphX und Java-Swing erfolgt die Darstellung. Über die GUI ist es möglich die anderen Komponenten (Parsen, Suchen) zu steuern. Es sollte also möglich sein hierdurch Dateien zu laden und zu speichern und dem Suchalgorithmus die benötigten Informationen zu geben.

Testing

Um die Richtigkeit unseres Algorithmus zu überprüfen haben wir mehrere Tests:
- pos. Test: wird ein Weg mittels Algorithmus gefunden
- neg. Test: es soll kein Weg gefunden werden, wenn im Graphen auch keiner existiert
- shortest_way: wird der kürzeste Weg gefunden, wenn es einen gibt

Fragen

1. Was passiert, wenn Knotennamen mehrfach auftreten?

Da der Knotenname eine eindeutige Identifikation ist, werden beide Knoten als ein und der Selbe geachtet. Dies ist auch nötig, da in dem gegebenen Format nur eine Kante pro Zeile und somit auch immer nur eine Kante pro Schritt eingelesen werden kann. Wenn nun zwei Knoten mit gleichem Namen als Unterschiedliche Knoten gehandhabt werden, würde ein falscher Graph, oder eher nicht der Graph, der gewünscht war, entstehen.

2. Wie unterscheidet sich der BFS für gerichtete und ungerichtete Graphen?

Bei gerichteten Graphen muss geprüft werden, ob die adjazenten Knoten auch erreichbar sind.
Sonst werden Wege genommen, die nicht möglich sind (gegen die Einbahnstraße).

3. Wie können sie testen, dass ihre Implementierung auch für sehr große Graphen richtig ist?

Über einen Bulk-Test, wo wir einen sehr großen Graphen (Knotenanzahl > 1000) erzeugen und den implementierten Algorithmus drüber laufen lassen.

Datenstrukturen

UML-Diagramm: