Aufgabe 2: Optimale Wege

In dieser Aufgabe werden zwei Algorithmen zum Finden optimaler Wege implementiert und verglichen.

Aufgabenstellung

Implementieren Sie nun unter Verwendung der ADT Graph folgende zwei Algorithmen, **so, wie sie in der Vorlesung vorgestellt wurden**:

1. Den **Dijkstra**-Algorithmus, so wie er in der Vorlesung vorgestellt wurde. Die Startecke in der Schnittstelle dient lediglich den Tests und beschränkt den Algorithmus nicht in seiner Funktionalität!
   Schnittstellen: `dijkstra:dijkstra(<Filename>,<StartVertex>,[d|ud])`;
   Rückgabewert: `[{11,0,11}, {33,5,11}, {<Vertex>,<Entf>,<Vorg>},…]`; bzw. für zB die Laufzeitmessung: `dijkstra:dijkstra(<Graph>,<StartVertex>,[d|ud])`;
2. Der Algorithmus von **Bellmann-Ford**, so wie er in der Vorlesung vorgestellt wurde. Die Startecke in der Schnittstelle dient lediglich den Tests und beschränkt den Algorithmus nicht in seiner Funktionalität! Achtung: wenn der Algorithmus einen Kreis negativer Länge findet, ist kein Erlang-error zu werfen, sondern lediglich eine IO-Ausgabe zu machen und das errechnete Ergebnis zurück zu geben!
   Schnittstelle: `bellmannford:bellmannford(<Filename>,<StartVertex>,[d|ud])`;
   Rückgabewert: `[{11,0,11}, {33,5,11}, {<Vertex>,<Entf>,<Vorg>},…]`; bzw. für zB die Laufzeitmessung: `bellmannford:bellmannford(<Graph>,<StartVertex>,[d|ud])`
3. Führen Sie mit den unter 1. und 2. implementierten Versionen aussagenkräftige Messungen durch (**Einheit: ms!**), mit der Sie die Laufzeitkomplexität der einzelnen Algorithmen nachweisen können. **Im Entwurf** sind für die Messungen bereits die Parameter der einzelnen Messungen anzugeben: welches Ziel (z.B. Laufzeitkomplexität) soll geprüft werden und welche Parameter (z.B. Anzahl Ecken, Grad der Ecken, Gewichte an den Kanten etc) werden in welchem Versuch mit welchen Erwartungen verändert. **Erstellen Sie aus den Messungen ein pdf,** indem die Messungen dokumetiert werden (Versuchsaufbau, Resultate, Interpretation, geforderte Nachweise etc.). Sofern Ihre Messungen die erwartete Komplexität **nicht bestätigen**, ist die Implementierung zu verbessern oder ausführlich herzuleiten, warum Ihre Implementierung eine andere Laufzeitkomplexität hat.

Auf der Hauptseite zur Vorlesung sind Graphen vorgegeben. Die Attributwerte stellen die Kosten der Kante dar. Die Graphen 01 (18119), 03 (11), 04 (11) und 05 (71) sind zum Test der Korrektheit zu verwenden (in der Klammer steht die Startecke). Führen Sie mit Ihren Algorithmen eine **Zeitmessung** durch (aufg2test.beam, Aufruf `aufg2test:zeitmessung()`) und **geben die damit erzeugte Datei mit ab**. Für diesen Test werden die folgenden Graphen (im gleichen Ordner wie die beam Dateien) benötigt: testaufg2.graph und graph_de.graph. Darüber hinaus sind für die Zeitmessung (**Einheit: ms!**) Graphen mittels dem Tool gengraph zu generieren und zu verwenden. (sowohl gerichtet, wie auch ungerichtet!)

Die zugehörigen Dateien heißen `dijkstra.erl` und `bellmannford.erl`. Weitere Dateien neben `adtgraph.erl` und `util.erl` sind nicht zulässig.

Abnahme

Bis Donnerstag Abend 20:00 Uhr vor Ihrem Praktikumstermin ist ein **Entwurf** für die Aufgabe **als *.pdf** Dokument (Dokumentationskopf nicht vergessen!) mir per E-Mail (mit cc an den/die

Teampratner_in) zuzusenden. Der Entwurf muss so gestaltet sein, dass er als einziges Dokument für eine Implementierung (auch für nicht Teammitglieder_innen) ausreichend ist.

Am Tag vor dem Praktikumstermin bis **20:00 Uhr**: bitte finaler Stand (als *.zip) zusenden, der in der Vorführung des Praktikums eingesetzt wird und alle Vorgaben erfüllen muss.

Am Tag des Praktikums findet eine Besprechung mit Teams statt. Die **Besprechung muss erfolgreich absolviert werden**, um weiter am Praktikum teilnehmen zu können. Bei der Besprechung handelt es sich nicht um die Abnahme.

Abgabe: Unmittelbar am Ende des Praktikums ist von **allen Teams** für die **Abgabe** der erstellte und sinnvoll dokumentierte Code abzugeben. Zu dem Code gehören die Sourcedateien und eine **Readme.txt** Datei, in der beschrieben wird, wie das System zu starten ist! Des weiteren ist der aktuelle Dokumentationskopf abzugeben. **In den Sourcedateien ist auf den Entwurf zu verweisen**, um die Umsetzung der Vorgaben zu dokumentieren. Alle Dateien sind als **ein *.zip Ordner** (mit cc an den/die Teampratner_in) per E-Mail abzugeben. Die Abgabe gehört zu den PVL-Bedingungen und ist einzuhalten, terminlich wie auch inhaltlich!

Beachten Sie die Regularien zum Praktikum!

Graphentheoretische Konzepte und Algorithmen

Aufgabe 2

Algorithmus von Dijkstra

Vorbereitung *l_{ij}: Länge der Kante v_iv_j. l_{ij}:=∞, falls es eine solche Kante nicht gibt. Für jede Ecke $v_i \in V$ werden drei Variable angelegt:*

1. **$Entf_i$**: *die bisher kürzeste Entfernung von v_1 nach v_i an. Startwert 0 für i=1 und ∞ sonst.*
2. **$Vorg_i$** *Vorgänger von v_i auf dem bisher kürzesten Weg von v_1 nach v_i an. Startwert v_1 für i=1 und undefiniert sonst.*
3. **OK_i** *= true, falls die kürzeste Entfernung von v_1 nach v_i bekannt ist. Startwert false.*

Iteration *Wiederhole (i,j seien dabei die Laufvariablen, h ein fester Wert)*

Suche unter den Ecken v_i mit OK_i = false eine Ecke v_h mit dem kleinsten Wert von $Entf_i$.
Setze OK_h := true.
Für alle Ecken v_j mit OK_j=false, für die die Kante v_hv_j existiert:
Falls gilt $Entf_j > Entf_h + l_{hj}$ dann
Setze $Entf_j := Entf_h + l_{hj}$
Setze $Vorg_j := h$

solange es noch Ecken v_i mit OK_i=false gibt.

Schnittstellen

```
dijkstra:dijkstra(<Filename>,<StartVertex>,[d|ud])
```

Rückgabewert:

```
[{11,0,11}, {33,5,11},{<Vertex>,<Entf>,<Vorg>},…]
```

Für zB Laufzeitmessung:

```
dijkstra:dijkstra(<Graph>,<StartVertex>,[d|ud])
```

Benötigte Dateien: *dijkstra.erl* und *adtgraph.erl* sowie *util.erl*

Algorithmus von Bellman-Ford

Vorbereitung *l_{ij}: Länge der Kante v_iv_j. l_{ij}:=∞, falls es eine solche Kante nicht gibt. Für jede Ecke $v_i \in V$ werden zwei Variable angelegt:*

1. **$Entf_i$**: *die bisher kürzeste Entfernung von v_1 nach v_i an. Startwert 0 für i=1 und ∞ sonst.*
2. **$Vorg_i$** *Vorgänger von v_i auf dem bisher kürzesten Weg von v_1 nach v_i an. Startwert v_1 für i=1 und undefiniert sonst.*

Iteration *Wiederhole |V|-1 mal (i,j seien dabei die Laufvariablen)*

Für alle Kanten (v_iv_j) aus E

Falls gilt $Entf_j > Entf_i + l_{ij}$ dann

Setze $Entf_j := Entf_i + l_{ij}$

Setze $Vorg_j := i$

Für alle Kanten (v_iv_j) aus E

Falls gilt $Entf_j > Entf_i + l_{ij}$ dann

STOP mit Ausgabe "Zyklus negativer Länge gefunden“

Schnittstellen

```
bellmanford:bellmanford(<Filename>,
                        <StartVertex>,[d|ud])
```

Rückgabewert:

```
[{11,0,11}, {33,5,11},{<Vertex>,<Entf>,<Vorg>},…]
```

Für zB Laufzeitmessung:

```
bellmanford:bellmanford(<Graph>,
                        <StartVertex>,[d|ud])
```

Benötigte Dateien: *bellmanford.erl* und *adtgraph.erl* sowie *util.erl*

Durchzuführende Tests

```
2> aufg2test:zeitmessung().
Graph mit 765 Ecken und 3385 Kanten aus 'testaufg2.graph' importiert.
        dijkstra d
.......
        bellmannford d
.......
        >>--<<
Graph mit 765 Ecken und 3385 Kanten aus 'testaufg2.graph'
        dijkstra ud
.......
        bellmannford ud
.......
        >>--<<
Graph mit 23 Ecken und 44 Kanten aus 'graph_de.graph
        dijkstra d
        bellmannford d
        >>--<<
Graph mit 23 Ecken und 44 Kanten aus 'graph_de.graph
        dijkstra ud
        bellmannford ud
        >>--<<
ok
3>
```

Benötigt: testaufg2.graph
graph_de.graph

results441000.log 23.02.2018 15:16 Textdokument 2 KB

Ergebnisse

- Fehler beim Verständnis fachlicher Begriffe.
  Beispiel: Mehrfachkante. Bei Unsicherheit ggf. nachfragen und Klären. Unsicherheiten führen zu unsicherem Code.

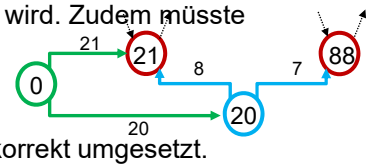
- Fehler im Entwurf.
  Beispiel: Fehler im beschriebenen Algorithmus.
  Hier wird nicht die Ecke mit *OK=false* und *Entf=min* gefunden, sondern es wird die erste Ecke genommen, die *for* aus *getVertices* auswählt. Die Vorgabe wurde falsch umgesetzt. Wichtig ist daher ein detaillierter Abgleich der Vorgabe mit dem Entwurf.

```
for Vertex in getVertices(Graph)
    NearestVertex <- nil
    NearestValue <- infinity
    if Vertex.ok = false and Vertex.entf < NearestValue
          NearestValue <- Vertex.entf
          NearestVertex <- Vertex
    NearestVertex.ok <- true
    for TargetVertex in getTarget(Graph,Vertex)
          if TargetVertex.ok = false
```

32

Ergebnisse

- Fehler bei der technischen Umsetzung.
  Beispiel: Annahme: nächste Ecke mit *OK=false* und *Entf=min* ergibt sich aus letzter Ecke mit diesen Eigenschaften und Kante mit minimalem Gewicht. Solche Überlegungen müssten im Entwurf gemacht werden, da dort über die inhaltliche Aufgabe nachgedacht wird. Zudem müsste geprüft werden, ob die Veränderung den vorgegebenen Algorithmus korrekt umsetzt.



Beispiel: Algorithmus aus dem Entwurf nicht korrekt umgesetzt.
Hier wurde mit *getEdges* alle Kanten im ungerichteten Fall dem Graphen entnommen, jedoch wurde nicht die entgegengesetzte Richtung betrachtet.
Wichtig ist daher ein detaillierter Abgleich der Vorgabe mit dem Entwurf.

Wiederhole : Für alle Kanten:

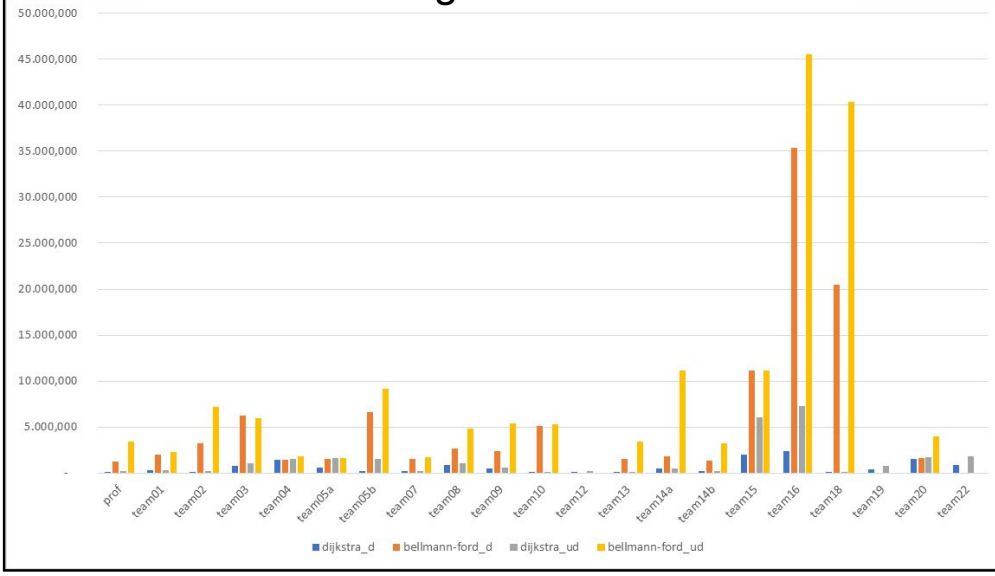
- Wenn $Entf_j$ größer ist als $Entf_i$ plus l_{ij} dann:

```
{Distancetable, Vertices
Edges = getEdges(Graph),
```

```
forEach1([Vi, Vj|En], Graph, Distancetable)
    {Entfi, _Vorgi} = getEntfVorgFrc
    {Entfj, _Vorgj} = getEntfVorgFrc
    Iij = getValE(Graph, Vi, Vj, wei
```

33

Ergebnisse


50.000,000
45.000,000
40.000,000
35.000,000
30.000,000
25.000,000
20.000,000
15.000,000
10.000,000
5.000,000
-
prof
team01
team02
team03
team04
team05a
team05b
team07
team08
team09
team10
team12
team13
team14a
team14b
team15
team16
team18
team19
team20
team22
dijkstra_d
bellmann-ford_d
dijkstra_ud
bellmann-ford_ud