GKA - Aufgabe 3 - Entwurf

Inhaltsverzeichnis

GKA - Aufgabe 3 - Entwurf

Allgemeines

Team: 02 Eugen Deutsch, Phillip Schackier

Aufgabenaufteilung:
Die Aufgabe wurde zusammen bearbeitet.
Quellenangaben: GKA Skript
Bearbeitungszeitraum:
11.11.15 - 3 Stunde: Beide
12.11.15 - 1 Stunden: Eugen

Aktueller Stand: Das Skript, sowie einige Testfälle wurden erstellt.
Änderungen in der Skizze: -

Package-/Ordner-Hierarchie und Sichtbarkeiten

OptimizedFlow
----- fordf <public class>
----- edmondsk<public class>
----- maxflow<public class>
----- benchmark <public class>
----- tests
---------- FlowTest<public class>

Der Aufbau wurde mit Team 07 abgesprochen, um die Austauschbarkeit zu gewährleisten.

Klassen/interface Hierarchien und Sichtbarkeiten

Es gibt keinerlei Vererbungen untereinander. Die Klassen sind nicht instanziierbar und dienen nur als Container für die Methoden.

Implementationen und Spezifikationen

Benchmark

- **public static void main(String** args**[])**: Hier wird ermittelt, wie viele Zugriffe die Algorithmen benötigen und wie viel Zeit die Operationen in Anspruch nehmen. Dazu verwenden wir den folgenden Versuchsaufbau:

Wir nehmen alle vorgegebenen Graphen und führen auf diesen die beiden

GKA - Aufgabe 3 - Entwurf

Algorithmen aus, wobei wir als source und target jeweils alle Möglichkeiten durchspielen und dann die Zeiten und Zugriffe für einen Graphen aufsummieren, sodass wir jeweils 14 Zugriffszahlen und 14 Zeiten erhalten, bei 14 Graphen. Die Zeiten werden in ms ermittelt.

Ausgegeben wird das Ergebnis in eine *.csv Datei.

fordf

- **public static int fordfulkerson(Graph** graph, **Vertex** source**, Vertex** target**)**:

Gibt den Wert des größten Flusses von source nach target zurück und verwendet dabei den folgenden Algorithmus:
1) Merke dir für alle Kanten den Flusswert 0. Die Quelle wird markiert mit undefiniert als Vorgänger und unendlich als Flusswert.

2) Prüfe ob alle markierten Ecken inspiziert wurden und falls ja, fahre mit 4 fort.
Für alle markierten, nicht inspizierten Ecken (v1):
Suche alle Kanten, die die derzeitige Ecke als Teil von sich haben und zu einer unmarkierten Ecke führen (v1 -> v2), sofern . Markiere v2 mit v1 als Vorgänger und dem Flusswert min(Kapazität von (v1 -> v2), Flusswert von v1).

Markiere v2 mit v1 als Nachfolger und dem Fl

Step 1, init:
for all vertices v: v.setFlow(0)
source.mark(null, inifinity)

Step 2, inspect, mark:
if (vertices.getMarked().all { v -> v.inspected? }) goto 4

for all vertices.getMarked() v:
if (v.inspected?) continue
else v.inspect()

inspect()
for all edges e(this -> v2):
if (not(v2.marked? or v2.flow.full?))
newFlow = min(e.capacity - e.flow, this.flow)

```
            v2.mark(+this, newFlow)
        end
        if (target.marked?) goto 3
    end

    for all edges e(v2 -> this):
        if (not(v2.marked? or v2.flow.empty?))
            newFlow = min(e.flow, this.flow)
            v2.mark(-this, newFlow)
        end
        if (target.marked?) goto 3

    end


Step 3, enlarge:
    actualNode = null
    pre = target.pre
    while (pre != null)
        edge(pre -> actualNode).flow += target.flow
        edge(actualNode -> pre).flow -= target.flow
    for all vertices.getMarked() v:
        if (v != source) v.unmark()

Step 4, end:
    Edges.cut() { e -> e.v1.inspected || e.v2.inspected } = d
```

Bedingungen: source != null, target != null, graph != null, source != target, graph.has(source, target), source.hasPathTo(target)

- **public static long fordfulkersonRtm(Graph** graph, **Vertex** source**,** **Vertex** target**)**:

Wie **fordfulkerson**, nur das die Laufzeit gemessen wird.

- **public static longfordfulkersonAcc(Graph** graph, **Vertex** source**, Vertex** target**)**:

Wie **fordfulkerson**, nur das die Anzahl der Zugriffe ermittelt wird.

edmondsk

- **public static int edmondskarp(Graph** graph**, Vertex** source**, Vertex**

target**)**:
Gibt den Wert des größten Flusses von source nach target zurück und verwendet dabei den **fordfulkerson** Algorithmus mit dem Unterschied, dass nicht irgendeine Ecke bei Schritt 2 gewählt wird, sondern diejenige, die die zum kürzesten, vergrößernden Weg führt.

Bedingungen: source != null, target != null, graph != null, source != target, graph.has(source, target), source.hasPathTo(target)

- **public static long edmondskarpRtm(Graph** graph**, Vertex** source**,** **Vertex** target**)**:
Wie **edmondskarp**, nur das die Laufzeit gemessen wird.

- **public static long edmondskarpAcc(Graph** graph**, Vertex** source**,** **Vertex** target**)**:
Wie **edmondskarp**, nur das die Anzahl der Zugriffe ermittelt wird.

maxflow

- **public int findMaxFlow(Graph** graph**, Vertex** source**, Vertex** target**,** **Integer** variant**)**:
Benutzt den **edmondskarp** oder den **fordfulkerson** Algorithmus, je nachdem welche Variante gewählt wurde. Liefert dann den Wert des größten Flusses von source nach target zurück.

variant == 1: **fordfulkerson**
variant == 2: **edmondskarp**

Bedingungen: graph != null, source != null, target != null, variant.within(1, 2), source != target, graph.has(source, target), source.hasPathTo(target)

Fehlerbehandlung

Wenn eine der Bedingungen nicht erfüllt wird, so wird der Algorithmus nicht ausgeführt und als Ergebnis wird 0 zurückgegeben. Es werden keinerlei Fehler geworfen.

GKA - Aufgabe 3 - Entwurf

Tests

Getestet werden die beiden Algorithmen vor allem in folgenden Situationen:

- Alle unerfüllten Bedingungen
- Graph mit zwei Elementen
- normaler Graph mit einigen Elementen
- größerer Graph
- Vorgegebene Graphen mit unterschiedlichem Start und Ziel

Export

Alles zusammen: flow.jar
Tests: flowTests.jar
Klassen: edmondsk.jar, fordf.jar, maxflow.jar