BAI3-GKA WiSe20
Graphentheoretische Konzepte und Algorithmen

Praktikumsaufgabe 3

Deckblatt

Geben Sie bitte Ihre Namen, Ihr Team und die Gruppe an:

GKA-Gruppe	3
Team	C
Name	Hani Alshikh
Name	Jannik Stuckstätte

Geschätzte Arbeitszeiten in Stunden

Name	Jannik Stuckstätte: 25 Std.
Name	Hani Alshikh: 20 Std.
Name	
Name	

1 Dokumentation der Implementierung

1.1 Algorithmen und Datenstrukturen

- FlowAlgorithmNodeMark

  für beide Implementierungen wird eine eigene Datenstruktur „FlowAlgorithmNodeMark" verwendet um die aus- eingehenden Knoten bei der inspektion zu markieren. Dies erlaubt uns die relevanten Daten wie der Vorgänger oder das Delta bei jedem Knoten zu speichern und zu zugriffen

- Ford und Fulkerson

  Wir haben uns für die Speicherung von der Kapazität, dem Fluss und der Markierung eines Knotens für den Einsatz mehreren Arrays entschieden. Dies ermöglicht uns anhand des Indexes eines Knotens einen schnellen Zugriff auf den benötigten Daten und spart uns, im Vergleich zu Graphstream Attribute, die mehrfache Abfragen nach den Attributen, das Prüfen, ob das Attribut vorhanden ist, und das Casten auf den richtigen Datentyp.

  Für die DFS-Suche haben wir uns für die Rückgabe eines Optional entschieden. Hierdurch werden Prüfungen auf nicht vorhandene Erweiterung des Pfades deutlich lesbarer und die Gefahr eines unerwarteten NullPointers deutlich reduziert.

- Edmonds und Karp

  Es wird hier auch aufgrund der oben genannten Gründe Arrays für die Speicherung der relevanten Daten benutzt.

  Für die BFS-Suche wird die LinkedList-Klasse verwendet, die funktionalität eines Queues anbietet und eine Laufzeit von O(1) Rechenschritte für das Addieren und Entfernen von Elementen, was für unseren Fall mehr als genug ist.

1.2 Entwurfsentscheidungen

Die Implementierungen von Ford und Fulkerson sowie Edmonds und Karp sind auf einen schwachen zusammenhängenden schlichten Digraph beschränkt.

- FlowAlgorithmGKA

  Wir haben uns für die abstrakte Klasse FlowAlgorithmGKA entschieden, da die Implementierung von Edmonds und Karp auf die Implementierung von Ford und Fulkerson basierend ist. Somit entstehen mehrerer Gemeinsamkeiten, die laut des DRY Prinzips abstrahiert werden sollten.

  Die Abstraktion bietet seiner Kinder-Klassen alle relevanten Daten über das übergebene Netzwerk sowie die relevanten Daten Strukturen für die Berechnung des maximalen Fluss.

  Für die Vergrößerung der Flussstärke wird hier auch eine Methode angeboten, die beginnend mit der Sinke durch die markierten Knoten bis zur Quelle alle Kanten entsprechend um delta erhöht bzw. vermindert.

- FlowAlgorithmNodeMark

  Da wir für beide Implementierungen die Knoten mit (±vorgänger, delta) markieren sollten, haben wir uns für eine eigene Datenstruktur FlowAlgorithmNodeMark entschieden. Anhand der eigenen Implementierung können wir gezielt die gewünschten Daten speichern und zugreifen. Dies spart uns die mehrfache Initialisierung und verwaltung verschiedenen Datenstrukturen bei jeder Inspektion.

- Ford und Fulkerson

  Zur Implementierung des Ford und Fulkerson-Algorithmus haben wir uns so strikt wie möglich an den in der Vorlesung vorgestellten Algorithmus orientiert. Die 4 Schritte des Algorithmus sind in unserer Implementierung sehr deutlich zu erkennen. Die DFS-Suche wird solange laufen bis alle Knoten inspiziert sind. Die ausgehenden sowie die eingehenden Kanten werden schrittweise bearbeitet. Es werden entsprechend irrelevanten Kanten übersprungen und die relevanten in einem Durchgang je inspizierter Knoten bearbeitet. Bei jedem erfolgreichen Traversierung bis zur Sinke wird den Pfad entsprechend vergrößert.

  Außerdem haben wir uns bei der Implementierung für eine normale Klasse entschieden, da sich den maximalen Fluss eines Netzwerks nicht ändert und wir dann prüfen können, ob für die Instance bei mehrfachen Aufrufe der Compute-Methode die Berechnung schon getätigt wurde.

- Edmonds und Karp

  Auch bei der Implementierung des Edmonds und Karp-Algorithmus haben wir uns so strikt wie möglich an den in der Vorlesung vorgestellten Algorithmus orientiert. Dies war uns möglich, da die Implementierung des Edmonds und Karp-Algorithmus auf die Implementierung des Ford und Fulkerson-Algorithmus basiert ist.

  Der einzige Unterschied ist die verwendete Art der Breiten-Suche. In diesem Fall wird die BFS-Breitensuche anhand der Queue-Implementierung LinkList verwendet.

  Wir haben uns hier auch aufgrund der oben genannten Gründe für eine normale Klasse entschieden.

1.3 Testfälle

- "Reguläre"Testfälle

  Wir haben zunächst einige vordefinierte Testfälle geschrieben, um unsere Implementierung auf unkomplizierten und einfach nachvollziehbaren Pfaden zu testen. Dies führt gerade zu Beginn der Entwicklung dazu, dass bei Fehlern die Analyse dieser deutlich vereinfacht wird. Hierfür haben wir, wie bei unseren Tests der vorangegangenen Aufgabe, unter anderem alle möglichen Flüsse, auf einem gegebenen Netzwerk, gegen die Ergebnisse einer Referenz-Implementierung der GraphStream-Bibliothek getestet.

- Randomisierte Testfälle unter Anwendung eines Netzwerk-Generators

  Nachdem diese einfachen Testfälle von unseren Implementierungen bestanden worden sind, haben wir, wie in der Aufgabenstellung vorgesehen, einen Netzwerk-Generator implementiert, welcher für eine gegebene Anzahl an Knoten und Kanten ein zufälliges Netzwerk erstellt.

  Die Verwendung von randomisierten Testfällen ist äußerst sinnvoll, da sie verhindert, dass unbewusste Annahmen über die Beschaffenheit des Netzwerks mit in die Erstellung der Implementierung selbst und der Testfälle einfließt und Fehler hierdurch nicht erkannt werden.

1.4 Benchmarking

Wir haben uns für zwei Arten der Laufzeitmessung entschieden, um die Geschwindigkeit unserer Implementierung prüfen zu können.

- ungenaue aber für die Demonstration genügende Laufzeitmessung

  Wir haben uns in diesem Fall für einen einfachen Laufzeitmessungsmuster entschieden, der uns ermöglicht, während der zeitlich beschränkten Demonstration die Laufzeit unsere Implementierungen vorzustellen.

  Es wird hierbei die verschiedenen Faktoren der Java Virtual Machine (JVM) nicht beachtet. Nur die Zeitdifferenz vor dem Ausführen und nach dem Ausführen wird berechnet und vorgestellt.

- Java measuring harness (JMH)

  Um kräftige Aussagen treffen zu können und möglichst presiese berechnung der Laufzeit zu erzielen, haben wir uns für JMH entschieden, da dies teil der OpenJDK ist und vom Core-Entwickler verwendet wird.

2 Beantwortung der Fragen

1. **Welcher Algorithmus/welche Implementierung ist schneller? Wie schnell für die Netzwerke Ihrer Praktikumsgruppe?**

   Der Algorithmus von Ford und Fulkerson findet einen maximalen Fluss in $O(Ef)$ Rechenschritte
   Der Algorithmus von Edmonds und karp findet einen maximalen Fluss in $O(VE^2)$ Rechenschritte
   wobei E die Anzahl der Kanten und V die Anzahl der Knoten des Netzwerkes und f den Wert des maximalen Flusses bezeichnen.

   Also, In der Praxis ist der Algorithmus von Edmond und Karp schneller, aber theoretisch kann Ford Fulkerson im Fall $O(Ef) < O(VE^2)$ schneller sein.

	Ford Fulkerson	**Fehlerspielraum**	**Edmond Karp**	**Fehlerspielraum**
50 V 800 E 10 UB	1.669	± 0.100	1.222	± 0.036
50 V 800 E 100 UB	1.873	± 0.557	1.223	± 0.084
50 V 800 E 1000 UB	2.014	± 0.589	1.229	± 0.071
800 V 300.000 E 10 UB	25433.159	± 10374.111	22485.959	± 3405.667
800 V 300.000 E 100 UB	27456.315	± 10736.751	23578.658	± 1711.557
800 V 300.000 E 1000 UB	30506.608	± 10505.654	23715.502	± 3517.037
2.500 V 2.000.000 E 10 UB	1068262.889	± 414730.093	816698.716	± 49747.122

*Bei allen Fällen wurden 5 duchläufe zum erwärmen der JVM ausgeführt.
*Bei allen Fällen wurden 10 Iterationen auf 10 verschiedenen Netzwerken durchgeführt.
*UP UpperBound: die maximale Kapazität einer Kante.
*bei dem letzten Fall besteht die Ausnahme, dass wir die eine Minute Grenze überschritten haben und deswegen aufgehört haben.

2. **Was haben Sie unternommen, um eine bessere Laufzeit zu erreichen?**

   - Eigene Datenstruktur-Implementierung FlowAlgorithmNodeMark.
   - Die Verwendung von Arrays, die konstante Zeit für den Zugriff anbieten.
   - Die Verwendung einer LinkedList, die konstante Zeit für den Zugriff auf das erste und letzte Element anbietet (add und poll), da LinkedList einen Pointer auf das erste und letzte Element speichert.
   - Die Verwendung von Streams, die eine verbesserte Laufzeit der Bearbeitung anbieten.

3. **Lässt sich die Laufzeit Ihrer Implementierung durch andere Datenstrukturen verbessern?**

   Nicht zu unserem Wissen. Wir haben überall versucht, Datenstrukturen so zu wählen, dass wir die konstante Laufzeit von O(1) bei jedem Rechenschritt zu erreichen.

4. **Was passiert, wenn Sie nicht-ganzzahlige Kantengewichte wählen?**

   Da wir double benutzen, kümmert sich Java, um die addition und subtraction der nicht-ganzzahligen Kantengewichte. Allerdings kommt das mit dem Nachteil, dass der resultierende maximale Fluss eine korrektheit von 0.n Ziffern zuweist, die nach unseren Tests und im Vergleich zu Graphstream implementierung plausibel ist.

   Die richtige empfohlene Vorgehensweise, ist das Berechnen des Hauptnenners aller Kantengewichte, so erhält man durch Multiplikation mit dem Hauptnenner ein ganzzahliges Netzwerk. Der berechnete ganzzahlige maximale Fluss muss dann durch den Hauptnenner geteilt werden, um den maximalen Fluss des originalen Netzwerks zu bekommen.

5. **Was passiert bei negativen Kantengewichten?**

   Wir haben uns da Gedanken gemacht und unsere Theorie wäre, dass eine Negative Kante in eine Richtung zu positiver gemacht werden kann, indem man die Richtung umtauscht. Somit hat man nach der Bearbeitung ein neues Netzwerk, das nur aus positiven Kanten besteht und der Fluss kann dann ganz normal berechnet werden.

   Aus zeitlichen Gründen konnten wir leider unsere Behauptung nicht prüfen und/oder in unsere Implementierung mitnehmen.