Fakultät IV - Elektrotechnik und Informatik
Algorithmen und Datenstrukturen SoSe 2017
Brock / Mabrouk / Sieverling / Stahl

Aufgabenblatt 9

letzte Aktualisierung: 23. Juni, 18:29 Uhr

Ausgabe: 23.06.2017
Abgabe: 5.07.2017 23:59

Thema: *Maximum Flow - Minimum Cut, Greedy Algorithmen, Strategien für Algorithmendesign*

Abgabe

Die folgenden Dateien müssen für eine erfolgreiche Abgabe im svn Ordner `Tutorien/txx/Studierende/deinname@TU-BERLIN.DE/Abgaben/` eingecheckt sein:

Geforderte Dateien:	
`Blatt09/src/Network.java`	Aufgabe 1.2, 1.3
`Blatt09/src/Atm.java`	Aufgabe 2.4

Wichtige Ankündigungen

- Unit tests dürfen geteilt werden. Lösungen dürfen auf keinen Fall geteilt werden! Wir testen auf Plagiate.
- Verändert die Methodenköpfe vorgegebener Klassen nicht. Verändert keine Klassen die nicht Teil der Abgabe sind und fügt keine neuen hinzu.

1. Aufgabe: Flussgraphen

In dieser Aufgabe kommen wir nochmal zu Algorithmen in Flussgraphen zurück. Ein oft auftretendes Problem ist die Unterteilung eines Graphen in mehrere Teilgraphen, wobei die Aufteilung so gewählt werden soll, dass möglichst wenige und möglichst unbedeutende Kanten (Kanten mit geringer Kapazität) entfernt werden. Dies ist das Minimum-Cut Problem.

1.1. (Tut) Min-Cut Implementierung Besprecht den möglichen Lösungsweg noch mal basierend auf der Beschreibung aus der Vorlesung!

Besprecht, welche Teilaufgaben bereits in Blatt 7 implementiert wurden, und deren Wiederverwendung die Implementierung vereinfacht!

1.2. Knotenmenge S berechnen (30 Punkte) Implementiert die Methode `findReachableNodes(startNode)`, die eine Liste aller Knoten zurückgibt, die im Restflußgraphen mit der startNode über Kanten mit positiver Restkapazität verbunden sind. Das Resultat entspricht der Menge S im Pseudocode der Vorlesung.

Tip: Testet diese Methode gut, bevor ihr sie in der zweiten Unteraufgabe weiterverwendet!

Verwendet beim Testen die Methode `fordFulkerson()` zur Erzeugung eines Restflußgraphen. Den Algorithmus habt ihr schon im Blatt 7 kennengelernt und implementiert. Eine korrekte Implementierung ist bereits in der Vorgabe enthalten. Die Methode speichert den finalen Restflußgraphen im Attribut `residualGraph`.

1.3. Cut-set berechnen (40 Punkte) Implementiert die Methode `List<Edge> findMinCut()`, die den minimalen Schnitt in einem Graphen berechnet. Die Methode gibt eine Liste von zu schneidenden Kanten (cut-set) zurück. Verwendet bei der Implementierung `fordFulkerson()` und `findReachableNodes()`!

2. Aufgabe: Gestückelte Geldausgabe

Um der Bank Transportkosten zu sparen, soll ein Geldautomat bei der Geldausgabe möglichst wenige Banknoten ausgeben. Entwerft einen Greedy-Algorithmus, der für jeden Betrag die kleinstmögliche Stückelung berechnet. Ihr könnt davon ausgehen, dass der Automat genügend Banknoten zur Verfügung hat.

Hinweis: Es gibt folgende Geldscheine: 5 Euro, 10 Euro, 20 Euro, 50 Euro, 100 Euro, 200 Euro.

2.1. (Tut) Greedy-Algorithmus Wie kann das Problem mit einem greedy (gierigen) Algorithmus gelöst werden?

2.2. (Tut) Ist der Greedy-Algorithmus vollständig? Ist der Greedy-Algorithmus vollständig? D.h. finden wir zu jeder beliebigen Eingabe (Geldsumme) immer eine Lösung (Stückelung)?

2.3. (Tut) Ist der Greedy-Algorithmus optimal? Angenommen, es gäbe zusätzlich 40-Euro-Banknote. Beantwortet dazu folgende Fragen:

1. Was wäre die optimale Lösung für einen Betrag von 85 Euro?
2. Welches Ergebnis würde euer Algorithmus liefern?
3. Welche Eigenschaft muss ein Banknotensystem haben, damit der Greedy-Algorithmus stets die beste Lösung findet?

2.4. Geldausgabestückelung - Implementierung (30 Punkte) Implementiert die Methode `getDivision(int total)` in der Klasse `atm.java`. Diese soll den übergebenen Betrag in möglichst wenige Banknoten stückeln. Verwendet dabei den Greedy-approach!

3. Aufgabe: Strategien für Algorithmen-Design

Für das Design von optimalen oder fast-optimalen Algorithmen gibt es eine Reihe von Strategien:

3.1. (Tut) Greediness Greedy bedeutet, dass man immer den lokal vielversprechensten Schritt Richtung Lösung macht. Lokal bedeutet in dem Kontext, dass z.b. nur die direkt angrenzenden Kanten inspiziert werden.

Ein greedy approach ignoriert die Folgen seiner Entscheidungen für spätere Schritte, und kann deswegen schneller berechnet werden.

Wo habt ihr bereits Greediness angewandt?

3.2. (Tut) Backtracking Backtracking bedeutet, dass man nur einen Teil des gerade untersuchten potentiellen Lösungsweges verwirft, wenn man erkennt dass dieser keine Lösung darstellt. D.h. man geht nur ein Stück zurück (Backtrack) und wiederverwendet den Rest des Weges für den nächsten potentiellen Lösungsweg.

Wo habt ihr bereits Backtracking angewandt?

3.3. (Tut) Divide-and-Conquer Divide-and-Conquer bedeutet, dass man das Problem in zwei oder mehrere ungefähr gleich große Unterprobleme aufteilt (divide), diese löst (conquer), und dann die Lösungen effizient kombiniert.

Vorraussetzungen zur Anwendung:

1. Das Problem muss in Teilprobleme zerlegbar sein.
2. Die Lösung muss aus Teillösungen zusammengesetzt werden können
3. Das Lösen der Teilprobleme und das Zusammensetzen muss eine niedrigere Komplexität haben als das ursprüngliche Problem

Oft wird divide-and-conquer rekursiv angewandt!

Wo habt ihr Divide-and-Conquer bereits angewandt?

Hinweis: Zwei weitere wichtige Strategien zum Algorithmendesign sind: Dynamic Programming und Branch-and-Bound. Diese werden in späteren Blättern und Vorlesungen behandelt.