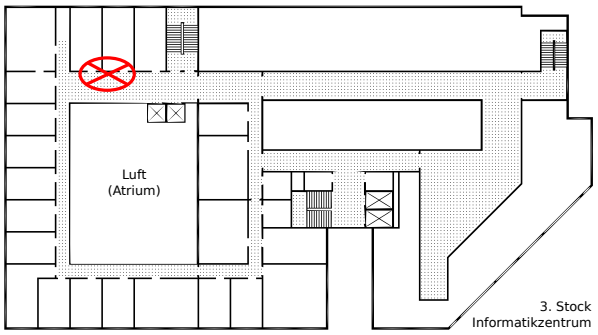
Abteilung Algorithmik **Sommer 2013**
Institut für Betriebssysteme und Rechnerverbund
TU Braunschweig

Prof. Dr. Sándor P. Fekete
Stephan Friedrichs

Algorithmen und Datenstrukturen II
Übung 3 vom 29. 05. 2013

Abgabe der Lösungen bis zum Mittwoch, den 19. 06. 2013 um 14:00 im Hausaufgabenrückgabeschrank.

Bitte die Blätter vorne deutlich mit eigenem Namen sowie Matrikel- und Gruppennummer versehen!



Auf diesem Blatt gibt es 40 Punkte, erreichbar (und abzugeben) sind aber lediglich (Teil-) Aufgaben im Gesamtwert von **maximal 30** Punkten. Mehr wird nicht gewertet!

Aufgabe 1 (Minimum Vertex Cover Approximation): Gegeben sei ein einfacher Graph $G = (V, E)$. Ein *Vertex Cover (VC)* von G ist eine Teilmenge $C \subseteq V$ seiner Knoten, die alle Kanten überdeckt: $\forall e = \{u, v\} \in E : e \cap C \neq \emptyset$. Hat C minimale Kardinalität, ist C ein *Minimum Vertex Cover (MVC)* von G. Wir betrachten Algorithmus 1, MVCApproximation, einen Approximationsalgorithmus für MVC.

```
1:  function MVCApproximation(G = (V, E))
2:      C ← ∅
3:      E' ← E

4:      while E' ≠ ∅ do
5:          {u, v} ← beliebige Kante aus E'
6:          C ← C ∪ {u, v}
7:          E' ← {e ∈ E' | e ∩ {u, v} = ∅}
8:      end while

9:      return C
10: end function
```

Algorithmus 1: Approximation von MVC

a) Wende MVCApproximation auf den Graphen G aus Abbildung 1 an. Gib in jedem Durchlauf der Schleife C und E' an. Wie lautet das am Ende ermittelte VC? Kommen in Zeile 5 mehrere Kanten in Frage, wähle die mit dem kleineren Index.

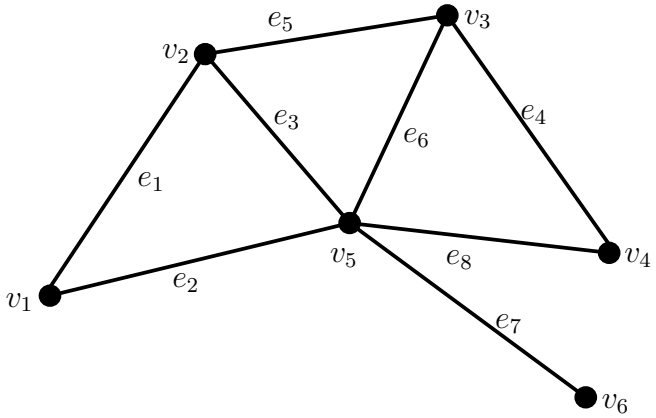


Abbildung 1: Graph G für Aufgabe 1a

b) Zeige: MVCApproximation terminiert und berechnet ein VC auf G.

c) Zeige: MVCApproximation liefert eine 2-Approximation von MVC auf G. (Hinweis: Das ist genau dann der Fall, wenn MVCApproximation für alle G ein VC C zurückgibt, das $|C| \leq 2|V'|$ erfüllt, wobei V' ein MVC von G ist.)

d) Zeige: MVCApproximation liefert keinen besseren Approximationsfaktor als 2.

(2+3+3+2 Punkte)

Aufgabe 2 (Komplexität): Ein großer Onlineshop möchte seinen Dienst mit möglichst wenig Servern betreiben, um Kosten zu sparen. Damit alles läuft, müssen n Jobs mit einer Load von $0 < j_1, \ldots, j_n \leq 1$ dauerhaft ausgeführt werden. Jeder der baugleichen Server kann Jobs mit einer Gesamtload von maximal 1 ausführen und Jobs können nicht auf mehrere Server verteilt werden.

Ideal wäre jetzt ein Algorithmus, der für jede Wahl von $j_1, \ldots, j_n$ bestimmt, wie viele Server gebraucht werden, und wie die Jobs darauf verteilt werden müssen. Leider hat die IT noch keinen effizienten Algorithmus gefunden, sondern lediglich Approximationsalgorithmen.

a) Was genau garantiert ein ρ-Approximationsalgorithmus, $\rho > 1$, für dieses Problem?

b) Es ist bekannt, dass Partition (Problem 1.9 aus dem Vorlesungsskript) ein ,,schweres" Problem ist. Hilf der IT und zeige, dass man mit Hilfe eines ρ-Approximationsalgorithmus' für das Serverproblem auch jede Instanz von Partition lösen kann, wenn $1 \leq \rho < \frac{3}{2}$ ist.

c) Angenommen, es gibt keinen effizienten Algorithmus der Partition löst, was bedeutet dann das Ergebnis von b) für das Serverproblem?

(2+6+2 Punkte)

Aufgabe 3 (TSP-Approximation): Beim Traveling Salesman Problem (TSP) geht es darum, in einem Graphen mit positiven Kantenlängen eine kürzeste Rundreise zu finden, die jeden Knoten genau einmal besucht und zum Ausgangsknoten zurückkehrt

(bekannt aus AuD1 und der großen Übung). Ebenfalls noch aus AuD kennt man die Sache mit den Eulertouren: Eine zusammenhängende Menge von Kanten in einem Graphen kann man genau dann in einem Zug, ohne abzusetzen und ohne Kanten doppelt zu verwenden in einem geschlossenen Weg bereisen, wenn jeder Knoten geraden Grad bezüglich dieser Kanten hat.

TSP ist NP-vollständig, man kann also nicht darauf hoffen, dass man einen Algorithmus finden kann, der für jede Instanz in polynomieller Zeit eine kürzestmögliche Rundreise findet. In dieser Aufgabe geht es darum, polynomielle Algorithmen zu entwickeln, die nicht optimale, aber gute Rundreisen finden.

Dabei ist ein *Minimum Spannung Tree (MST)* die günstigste Kantenauswahl (ein Baum), die alle Knoten des Graphen miteinander verbindet. Wie man in der Vorlesung Netzwerkalgorithmen (oder in der großen Übung) lernen kann, kann man so einen Baum in polynomieller Zeit bestimmen.

Außerdem ist für eine Menge von Knoten ein *Matching* eine Menge von knotendisjunkten Kanten; ein *perfektes Matching* ordnet jedem Knoten genau einen anderen zu. Auch ein kostenminimales Matching kann man in polynomieller Zeit bestimmen; auch das lernt man in Netzwerkalgorithmen.

Wir betrachten nun einen vollständigen Graphen, dessen Kantenlängen $c(e)$ die Dreiecksungleichung erfüllen, d.h. für drei Knoten u, v, w gilt $c(u, v) + c(v, w) \geq c(u, w)$ (wenn man von u nach w möchte, ist der Umweg über v mindestens so lang wie der direkte Weg). Sei $c(\text{MST})$ die Gesamtlänge eines MST, $c(\text{TSP})$ die Gesamtlänge einer kürzesten Rundreise.

a) Zeige: $c(\text{TSP}) \geq c(\text{MST})$. (Hinweis: Betrachte die oben beschriebene Eigenschaft eines MST.)

b) Zeige: Wenn man jede Kante eines MST durch zwei identische Kopien ersetzt (also die Kanten „verdoppelt"), bekommt man einen Eulerschen Graphen.

c) Zeige: Aus einer zugehörigen Eulertour kann man durch „Abkürzen" eine Rundreise machen, die nicht länger ist. (Hinweis: Dreiecksungleichung!)

d) Folgere: Man kann in polynomieller Zeit eine Rundreise berechnen, die nicht länger als $2c(\text{OPT})$ ist, also eine 2-Approximation für das TSP.

e) Jetzt verwandeln wir einen MST noch kostengünstiger in einen Eulerschen Graphen. Dafür sei MAT^* ein kostenminimales perfektes Matching der Knoten, die im MST ungeraden Grad haben (das existiert immer). Zeige: $c(\text{MAT}^*) \leq \frac{1}{2}c(\text{OPT})$. (Hinweis: Betrachte eine optimale Tour, zerlege sie mit Dreicksungleichung in zwei Matchings MAT_1 und MAT_2 der im MST ungeraden Knoten. Schlussfolgere daraus auf das günstigste Matching MAT^*.)

f) Zeige: Die Kanten von MST und MAT^* ergeben zusammen einen Eulerschen Graphen.

g) Folgere: Man kann in polynomieller Zeit eine Rundreise berechnen, die nicht länger als $\frac{3}{2}c(\text{OPT})$ ist, also eine $\frac{3}{2}$-Approximation. (Hinweis: Betrachte eine Eulertour auf $\text{MAT} \cup \text{MST}$ und kürze ab.)

(1+1+2+2+2+1+1 Punkte)

Aufgabe 4 (Implementierung Server Scheduling): Betrachte das Server-Scheduling-Problem aus Aufgabe 2.

a) Implementiere *First Fit (FF)*. Das heißt, jeder neue Job wird dem ersten Server zugewiesen, der ihn noch bewältigen kann. Falls der Job auf keinen Server mehr passt, wird ein neuer angefangen.

b) Implementiere *First Fit Decreasing (FFD)*. Das funktioniert wie FF, nur dass die Tasks vorher absteigend nach Load sortiert werden.

c) Implementiere *Best Fit (BF)*. Dabei wird jeder Job demjenigen Server mit minimalem freien Load zugeordnet, der die Task noch bewältigen kann. Passt der Job auf keinen Server, wird ein neuer eröffnet.

d) Implementiere eine untere Schranke, zum Beispiel $\lceil \sum_{i=1}^{n} j_i \rceil$. Bessere untere Schranken sind natürlich erlaubt, sollten aber kurz dokumentiert werden!

e) Halte in einer Tabelle fest, für welche Testinstanz FF, FFD und BF welche Anzahl von Servern benötigt, sowie die untere Schranke. Welche Heuristik funktioniert am besten?

Einziger Unterschied zwischen der Theorie in Aufgabe 2 und der Implementierung ist, dass die Server nicht die maximale Load 1 haben, sondern einen größeren Integer; auch die einzelnen Jobs haben eine ganzzahlige Load. So umgehen wir Rundungsungenauigkeiten in der Floatingpointarithmetik. Die Testinstanzen stehen auf der Vorlesungshomepage[1] zur Verfügung.

Wir testen deine Software mit

```
javac ServerScheduling1234567.java && java ServerScheduling1234567 \
    < instance_xyz
```

Zur Abgabe: Ersetze `1234567` durch deine Matrikelnummer und gib die Javadatei per Mail an `sfriedr@ibr.cs.tu-bs.de` ab. Nenne in der Mail Name, Matrikel- und Gruppennummer. Es gilt dieselbe Frist wie für die anderen Aufgaben. **(3+1+3+2+1 Punkte)**

[1] `http://www.ibr.cs.tu-bs.de/courses/ss13/aud2/`