Algorithmik

Funke/Bahrdt/Krumpe/Mendel/Seybold SS 2015

http://www.fmi.informatik.uni-stuttgart.de/alg

Institut für Formale
Methoden der Informatik
Universität Stuttgart

Übungsblatt 6

Punkte: **60 + 10 Bonuspunkte**

Problem 1 - Vertex Cover (12 P.)

Das Vertex Cover Problem sei wie folgt definiert:

Gegeben sei ein ungerichteter Graph $G = (V, E)$.
Gesucht ist $C \subseteq V$ mit $|C| = k$ sodass $\forall e = \{u, v\} \in E : e \cap C \neq \emptyset$.
Ein Vertex Cover sei optimal (oder minimal) falls $|C|$ minimal ist.

Zeigen Sie mittels Polynomzeitreduktion, dass die folgenden drei Vertex Cover Varianten gleich schwer sind (NP hart):

Entscheidungsproblem:	Gibt es in G ein VC der Größe k?
Optimierungsproblem:	Berechne die Größe k^* eines optimalen VC in G.
Berechnungsproblem:	Gebe ein optimales VC von G an.

Problem 2 - Vertex Cover Greedy (10 P.)

Der folgende Greedy Algorithmus berechnet für einen Graphen $G = (V, E)$ ein Vertex Cover C:

function GREEDYVC($G = (V, E)$)
  $C \leftarrow \emptyset$
  while $|E| > 0$ **do**
    $v_i \leftarrow$ Knoten in G mit maximalem Grad
    $C \leftarrow C \cup \{v_i\}$
    $G \leftarrow G \setminus \{v_i\}$
  end while
  return C
end function

Beweisen Sie, dass `GreedyVC(G)` nicht besser als $\Omega(\log(n))$ approximiert.
Beweisen Sie, dass `GreedyVC(G)` nicht schlechter als $\mathcal{O}(\log(n))$ approximiert und damit eine $\Theta(\log(n))$ APX ist.

Problem 3 Aproximierung für RUCKSACK (15 P.)

Betrachten Sie das folgende RUCKSACK-Problem.
Gegeben sei:

- eine Menge von n Objekten $\mathcal{U} = \{u_1, u_2, \ldots u_n\}$
- eine Wertfunktion $w : \mathcal{U} \mapsto \mathbb{R}$
- eine Gewichtsfunktion $g : \mathcal{U} \mapsto \mathbb{R}$
- ein Maximalgewicht des Rucksacks $M \in \mathbb{R}$

Gesucht ist eine Teilmenge $K \subseteq \mathcal{U}$ welche folgende zwei Bedingungen erfüllt:

$$\sum_{u \in K} g(u) \leq M \tag{1}$$

$$\sum_{u \in K} w(u) \; ist \; maximal \tag{2}$$

Zeigen Sie, dass folgender Algorithmus eine 2-APX von RUCKSACK berechnet:

1. Sortiere Gegenstände u_i nach Ihrem Wert-Gewicht-Verhältnis $\frac{w(u_i)}{g(u_i)}$

2. Füge iterativ den Gegenstand mit jeweils höchstem Wert-Gewicht-Verhältnis zu dem Rucksack K hinzu, bis dieser voll ist.

3. Sei u_k der erste zu große Gegenstand. Gebe $\{u_k\}$ als Rucksack zurück, falls $w(u_k) > \sum_{u_i \in K} w(u_i)$, sonst K

Problem 4 - Erwartete Turmhöhe in Skiplisten (10 P.)
Wir betrachten den diskreten Wahrscheinlichkeitsraum $(\mathbb{N}_0, \Pr)$ mit $\Pr : \mathbb{N}_0 \to [0,1]$ wobei $i \mapsto 1/2^{i+1}$. Sei $H : \mathbb{N}_0 \to \mathbb{R}$ die Zufallsvariable der Elementarereignisse $(H : i \mapsto i)$. Zeigen Sie, dass

- Die Abbildung Pr macht $(\mathbb{N}_0, \Pr)$ tatsächlich zu einem Wahrscheinlichkeitsraum $(1 = \sum_{x \in \mathbb{N}_0} \Pr(x))$.
- Der Erwartungswert der Zufallsvariable H ist 1.

Problem 5 - Skiplisten (3 + 4 + 6 P.)
Beschreiben sie in Pseudocode wie man die Zeiger in einer Skipliste bei den Operationen `insert(key)` und `erase(key)` modifiziert, sodass die Skipliste weiterhin korrekt für `find(key)` verwendbar bleibt.

Implementieren Sie eine Skipliste, welche Integer Werte als Keys verwalten kann und die Operationen `insert (key)`, `find (key)` und `erase (key)` unterstützt. Alle diese Operationen sollen erwartet in $\mathcal{O}(\log n)$ Zeit realisiert werden. Die Skipliste soll randomisiert arbeiten und die Element Höhen wie unten definiert geometrisch verteilt bestimmen. Die Skipliste soll doppelte Keys verwerfen, d.h. ein bestimmter Key kann maximal einmal in der Liste vorkommen.

Um das Ergebnis der Aufgabe deterministisch zu machen, besteht die Eingabe sowohl aus den Keys selber als auch den Zufallszahlen für die Bestimmung der Elementhöhen. Die Eingabeinstanz ist eine Textdatei aus $2m$ Integer Werten mit einem Wert pro Zeile, wobei sich sich Zeile $2k$ der k'te einzufügende Wert V_k befindet. In Zeile $2k+1$ befindet sich eine zufällig, gleich verteilte Zahl N_k aus dem Intervall $[0, 2^{30})$, aus der Sie wie folgt die Höhe des einzufügenden Elementes bestimmen:

$$\text{ElementHeight}(N_k) = \max\left(1, \min_p : \sum_{i=1}^{p} \frac{1}{2^p} \geq \frac{N_k}{2^{30}}\right).$$

D.h. die Höhe von Element k ist 1 wenn $N_k \in [0, 2^{29}]$, 2 wenn $N_k \in (2^{29}, 2^{29} + 2^{28}]$ usw.
Eine C Implementierung könnte wie folgt aussehen:

```
unsigned int getGeomHeight(const unsigned int N) {
  double DN = (double)N/(1<<30);
  double C = .5;
  unsigned int height = 0;
  while (DN>0) {
    DN -= C;
    C /= 2;
    height++;
  }
  return height;
}
```

Fügen Sie alle m Werte der Eingabe in Ihre Skipliste ein. Löschen Sie danach die Werte V_i mit $i \in [\frac{5}{10}m, \frac{6}{10}m)$.
Geben Sie an wie viel Speicher Ihre Datenstruktur insgesamt verbraucht und wie viel Byte Sie durchschnittlich pro Element brauchen. Geben Sie die Gesamtlaufzeit für die Einfüge- und Löschoperationen an. (Es reicht wenn Sie das gesamte Programm extern einmessen.)
Optional: Geben Sie die Datenstruktur für die `small.txt` Instanz am Ende auf geeignete Weise aus.

Auf der Algorithmik Webseite finden Sie zwei Dateien, von denen Sie **eine** benötigen:

(a) `instances.tgz`

(b) `generator.tgz`

`instances.tgz` ist groß (4.5MB) und enthält direkt `small.txt` und `large.txt`.
`generator.tgz` ist sehr klein und enthält ein C++ Programm sowie ein Makefile, welches diese beiden Instanzen direkt generiert.

Die Instanz `small.txt` enthält $m = 10$ und `large.txt` enthält $m = 500000$ Werte (und jeweils doppelt so viele Zeilen).

Zu dem Quellcode gehört ein Shellscript/Makefile, welches Ihr Programm compiliert und auf den beiden Instanzen ausführt. Geben Sie die Instanzen selber **nicht** mit ab. Gehen Sie stattdessen davon aus, dass diese im gleichen Ordner wie Ihre Programm liegen.

Problem Bonusaufgabe Travelling Salesmann Problem (4+6 P.) Implementieren Sie ein exaktes Lösungsverfahren für euklidisches TSP mit Hilfe dynamischen Programmierens. Die Eingabeinstanz ist eine Datei mit $2m$ Integer- Werten mit einem Wert pro Zeile, wobei sich in Zeile $2k$ und $2k+1$ die x und y Position der Stadt k befindet. Der Abstand zweier Städte a, b mit den Positionen (x_a, y_a) und $(x_b, y_b$ sei als $d(a,b) = \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2}$ definiert. Alle x und y Werte sind aus dem Intervall $(0, 2^{10}]$.

Geben Sie eine optimale Rundreise, welche in Stadt 0 beginnt (und endet), sowie deren Länge an. Die Instanz `small.txt` enthält $m = 10$ und `large.txt` enthält $m = 24$ Städte. Geben Sie die Ausgabe Ihres Programms sowie den Quellcode ab. Zu dem Quellcode gehört ein Makefile/Shellscript, welches Ihr Programm kompiliert und auf den Instanzen ausführt.

Implementierungshinweise: Verwenden Sie zur Darstellung der Teilmengen S einen Bitvektor. Hierfür kann ein Integer verwendet werden. Mit diesem und einem weiteren Integer für eine Stadt können Sie dann die Tabelle der Teillösungen adressieren.