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 4

Punkte: **50**

Problem 1 - Happy Hour (10 P.)

Sie und Ihre $(n-1)$ Freunde wollen sich zur einstündigen Happy Hour in einer (Milch-)Bar treffen. Sie können sich allerdings nicht auf eine Uhrzeit festlegen und einigen sich daher darauf, dass jeder zu einer zufälligen Uhrzeit (innerhalb dieser Stunde) kommt. Sie haben dabei alle die gleiche Strategie: Wenn Sie die Bar betreten, warten Sie maximal 20 Minuten darauf, dass alle Ihre $(n-1)$ anderen Freunde auch eingetroffen sind. Ist dies nicht der Fall, so bestellen Sie einfach beleidigt selber etwas.

Wie hoch ist die Wahrscheinlichkeit im Fall $n = 2$ dafür, dass Sie sich beide rechtzeitig treffen?
Wie hoch ist die Wahrscheinlichkeit in Abhängigkeit von n wenn gilt $n > 2$?

Problem 2 - Dijkstras Algorithmus (15 P.)

Sei $G = (V, E, \gamma)$ ein zusammenhängender, gerichteter, kantengewichteter Graph ($E \subseteq V \times V$), wobei $V = \{1, \ldots, n\}$ und $\gamma : E \to \mathbb{N} \setminus \{0\}$. Ein Vektor $D \in \mathbb{N}^n$ heißt *Distanzvektor*, falls D_i für alle i die Distanz eines kürzesten Weges von 1 nach i ist.

Zeigen Sie, dass ein Vektor $D \in \mathbb{N}^n$ genau dann ein Distanzvektor ist, wenn folgende 3 Bedingungen gelten:

1. $D_1 = 0$
2. für alle $(i, j) \in E$ gilt $D_j \leq D_i + \gamma(i, j)$,
3. für alle $j \neq 1$ existiert $(i, j) \in E$ mit $D_j = D_i + \gamma(i, j)$.

Dieses Kriterium liefert einen $\mathcal{O}(|E|)$-zeitbeschränkten Algorithmus um zu testen, ob z. B. eine Implementierung des Dijkstra-Algorithmus ein korrektes Ergebnis liefert.

Problem 3 - Teilungsknoten (10 P.)

Sei ein ungerichteter, zusammenhängender Graph $G = (V, E)$ gegeben. Ein Knoten $v \in V$ sei ein Teilungsknoten von G, wenn der Graph durch das Entfernen des Knotens v und aller inzidenten Kanten in mindestens 2 Zusammenhangskomponenten zerfällt.
Beschreiben Sie einen Algorithmus, der in $\mathcal{O}(|V| + |E|)$ Schritten alle Teilungsknoten in einem Graphen findet.
Zeigen Sie, dass ihr Algorithmus nur die geforderte Anzahl an Schritten nicht überschreitet.

Problem 4 - Minimum Spanning Tree (3 + 6 + 6)

Beschreiben (3P.) Sie kurz (aber präzise) Prims Algorithmus [1], der für einen gegebenen gewichteten, ungerichteten und zusammenhängenden Graphen einen Spannbaum berechnet, der ein minimales Gewicht hat (minimum spanning tree - MST).
Beschreiben Sie, welche Schritte notwendig sind um mit dem selben Algorithmus einen minimalen Spannwald für gewichtete, ungerichtete aber **nicht** zusammenhängende Graphen zu berechnen (minimum spanning forest - MSF).

Implementierung
Sie können alle Implementierungsaufgaben grundsätzlich in der Sprache Ihrer Wahl lösen, allerdings muss Ihr Tutor die Implementierung im Zweifelsfall verstehen. Außerdem muss es einen freien Compiler für Ubuntu geben, der die Sprache kompiliert. Völlig problemlos sind also C, C++, Java oder Ada. Das Testsystem läuft mit Ubuntu 12.04 oder neueren.

Implementieren (6P.) Sie Prims Algorithmus ([1]), zur Berechnung eines minimalen Spannbaums (MST) für einen gewichteten, ungerichteten und zusammenhängenden Graphen $G = (V, E, \gamma)$, $\gamma : e \in E \mapsto \mathbb{N}$.

Implementieren (6P.) Sie ihre Erweiterung für Prims Algorithmus, sodass ihr Algorithmus der auch auf nicht-zusammenhängende Graphen funktioniert und dort den minimalen Spannwald (MSF) berechnet.

Geben Sie als Ergebnis jeweils das Gewicht des MST sowie die IDs der Kanten aus, die den MST bilden.

Auf der Algorithmik Webseite finden Sie eine Datei (`instances.tgz`), die einen zusammenhängenden Graphen `connected.txt` sowie einen nicht- zusammenhängenden Graphen **disconnected.txt** enthält.

Die Dateien enthalten in einer Zeile eine Kante, definiert durch Tabulator- getrennte Werte (V_1, V_2, G). V_i sind eindeutige IDs der Knoten, G ist das Gewicht der Kante.

Die Instanz `connected.txt` enthält $|V| = 20, |E| = 90$ und `disconnected.txt` enthält $|V| = 1000, |E| = 24485$ Werte.

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.

Eine Anmerkung:
Im Optimalfall ist Ihre abgegebene Ausgabe korrekt. In diesem Fall werden wir Ihren Quellcode nur grob auf Plausibilität überprüfen. Ihr Programm sollte auf dem Computer des Tutors in akzeptabler Zeit terminieren und keine Libraries verwenden, die schon die geforderte Funktionalität mitbringen. Sollte Ihre Ausgabe nicht korrekt sein, hängt es von der Qualität Ihres Codes ab, inwieweit wir diesen inspizieren werden.

References

[1] `https://en.wikipedia.org/wiki/Prim's_algorithm`

Für die folgenden Übungsblätter
Leider gab es ein Missverständnis bezüglich der Themen, die Sie in der Vorlesung bereits behandelt wurden. Die vorherige Version des Übungsblatts enthielt deshalb Übungen zu Themen, die noch nicht behandelt worden waren.
Falls Sie die folgenden Aufgaben bereits bearbeitet haben, behalten Sie bitte ihre Aufschriebe. Die Aufgaben werden auf den folgenden Übungsblättern erneut gestellt werden!

Problem 2 - Universelles Hashing (15 P.)

Sei $\mathcal{U}$ mit $|\mathcal{U}| = u$ eine Menge zu hashender Elemente und $p \geq u$ eine Primzahl. Die Funktionenklasse $h_{a,b} : \mathcal{U} \to \mathbb{Z}_m$ hasht diese Elemente nach $\{0, \ldots, m-1\}$. Beweisen Sie, dass die Klasse von Hashfunktionen $h_{a,b}(x) = (((ax + b) \mod p) \mod m)$ unter der Vorraussetzung, dass $a \in \{1, \ldots, p-1\}$, $b \in \mathbb{N}$ zufällig gleichverteilt gewählt wird, *universell* ist.

Problem 3 - Vertex Cover

Das Vertex Cover Problem sei wie folgt definiert:

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

Zeigen Sie mittels Polynomzeitreduktion, dass die folgenden drei Vertex Cover Varianten gleich schwer sind (NP hart). Gegeben sei ein Graph $G = (V, E)$:

Entscheidungsproblem:	Gibt es ein VC der Größe k?
Optimierungsproblem:	Berechne die Größe k^* eines optimalen VC.
Berechnungsproblem:	Gebe ein optimales VC: $C \subseteq V$ an.

Problem 4 - Vertex Cover Greedy

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

function GreedyVC($G_0 = (V_0, E_0)$)
  $C \leftarrow \emptyset$
  $i \leftarrow 0$
  while $|E_i| > 0$ **do**
    $v_i \leftarrow$ Knoten in G_i mit maximalem Grad
    $C \leftarrow C \cup v_i$

$G_{i+1} \leftarrow G_i \setminus v_i$
$i \leftarrow i + 1$
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 5 - Erwartete Turmhöhe in Skiplisten
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 6 - Skiplisten
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.