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 5

Punkte: **60**

Problem 1 - Universelles Hashing (10 P.)

Sei $\mathcal{U} = \{0, ..., u-1\}$ 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 $a \in \{1, \ldots, p-1\}$, $b \in \{0, \ldots, p-1\}$ zufällig gleichverteilt gewählt wird, *universell* ist.

Problem 2 Vankin's Mile (10 P.) Vankin's Mile ist ein Einzelspieler Brettspiel, das auf einem $b \times b$ Schachbrett gespielt wird. Zunächst legt der Spieler seinen Spielstein auf irgend ein Feld des Spielbretts. Bei jedem Zug darf der Spieler den Spielstein ein Feld nach rechts oder ein Feld nach unten verschieben. Das Spiel endet, wenn der Spielstein aus dem Spielbrett heraus geschoben wird. Jedes Spielfeld hat hierbei einen Wert $w_f \in \mathbb{Z}$. Zu Beginn hat der Spieler einen Spielstand von 0. Jedes mal, wenn der Spielstein auf ein Feld verschoben wird, wird der Wert w_f des Spielfeldes zum aktuellen Spielstand hinzuaddiert. Ziel des Spieles ist es so viele Punkte wie Möglich zu erreichen.

- Geben Sie einen Algorithmus und dessen Laufzeit an, der zu gegebenem Spielfeld einen optimalen Spielpfad berechnet.

Problem 3 Traum-Euros (5 + 5 P.) In einem früheren Leben arbeiteten Sie als Kassierer/in in der untergegangenen Kolonie Nadira. Da Papier eine begrenzte und wertvolle Ressource war, waren Sie als Kassierer von gesetzes Wegen dazu verpflichtet, das Rückgeld Ihrer Kunden mit der kleinsten Anzahl an Geldnoten auszuzahlen. Die Währung von Nadira, Traum-Euros genannt, hatte hierbei folgende Werte: 1, 4, 7, 13, 28, 52, 91 und 365 Euro.

- Geben Sie einen Algorithmus und dessen Laufzeit an, welcher das Problem des Kassierers löst.
- Finden Sie ein Beispiel für das der Greedy-Algorithmus eine höhere Anzahl liefert als optimal wäre

Problem 4 Sommerfeier (15 P.) Sie wurden von Ihrem Arbeitgeber dazu verpflichtet die Sommerfeier Ihrer Firma zu organisieren. Ihre Firma ist in eine strenge Hierarchie unterteilt, ein Baum mit dem Firmenbesitzer als Wurzel. Jedem Mitarbeiter wurde eine Zahl $r_m \in \mathbb{R}$ zugewiesen, die seinen/ihren Spaßfaktor angibt. Damit es zu keinen Streitigkeiten kommt, darf ein Mitarbeiter zur Feier nur dann eingeladen werden, wenn sein direkter Vorgesetzter nicht eingeladen ist. Der Firmenbesitzer muss natürlich zur Feier eingeladen werden.

- Geben Sie einen Algorithmus und dessen Laufzeit an, der zu gegebener Arbeiterhierarchie eine Einladungsliste mit maximalem Spaßgrad erstellt. Der Spaßgrad einer Feier ist hierbei die Summe aller eingeladenen Mitarbeiter.

Problem 5 - Hashing (10 P. + 4 P. + 1 P.)

Implementieren sie eine Hashtabellen Datenstruktur, welche mit offener Adressierung und zwei unabhängigen Hashfunktionen arbeitet. Diese soll Strings hashen können und die Operationen `insert(string S)` und `find(string S)` unterstützen. `find(S)` gibt dabei einen Verweis auf den Eintrag zurück, oder signalisiert das Nicht-Enthaltensein von `S` in der Tabelle (z.B. mit einem Nullpointer oder dem Verweis auf ein "Ende"-Element).

Kollisionen sollen mit offener Adressierung vermieden werden. Aus den beiden Hashfunktionen $h_1, h_2 : \text{String} \mapsto \mathbb{N}$ konstruiert man dazu eine Hashfunktionen $H : \text{String} \times \mathbb{N} \mapsto \mathbb{Z}_M$ mit $M = |T|$ als Größe der Tabelle T und $H(S,i) = (h_1(S) + i \cdot h_2(S)) \mod M$. Um die Position des Elements zu bestimmen prüft man nun für aufsteigende $i >= 0$ ob $T[H(S,i)] \neq S$ bis man einen leeren Platz oder S gefunden hat. Findet man keinen Platz, muss die Hashtabelle vergrößert und neu aufgebaut werden.

Die Eingabeinstanzen sind diesmal Dateien mit jeweils einem Wort pro Zeile. Das Zeilen-Ende wird durch ein ASCII-Line-Feed (\n, 0x0A, 10 in dezimal) markiert. Diese finden Sie in gepackter Form auf der Algorithmik Webseite.

Um die großen Instanzen zu lösen bedarf es einiger Tricks um den Speicherplatzbedarf klein zu halten. So sind die Elemente, die gehasht werden, nicht veränderbar, sprich konstant. Ein String kann so z.B. als Zeiger mit Offset in die Datei der Strings dargestellt werden. Hierzu können Sie entweder die Datei komplett in den Arbeitsspeicher laden oder alternativ einen `MappedByteBuffer` verwenden.

Füllen Sie ihre Hashtabelle mit den Wörtern dieser Datien und suchen Sie danach alle Wörter einmal in der Tabelle. Ihre **Ausgabe** soll aus folgenden Werten bestehen:

- Anzahl W der Wörter
- der Loadfaktor der Tabelle (W/M)
- die durchschnittliche Anzahl der Lookups, die die Tabelle benötigt um alle W Wörter zu finden
- der Median dieser Anzahl von Lookups
- die Anzahl der Rebuilds, die Sie benötigt haben

Sowohl die Wahl von h_1, h_2 als auch die Wahl eines neuen M steht Ihnen frei. Die Hashfunktionen sollten unabhängig und universell sein. Experimentieren Sie mit der Neuwahl der M Werte beim rebuild. Eine Möglichkeit ist z.B. in C/C++ mit der MD5 Implementierung von libopenssl zu arbeiten:

```
#include <openssl/md5.h>

unsigned int md5_hash1(const std::string& S) {
  unsigned int result[MD5_DIGEST_LENGTH/4] = {0};
  MD5((unsigned char*)&S[0], S.length(), (unsigned char*)result);
  return result[0];
}

unsigned int md5_hash2(const std::string& S) {
  unsigned int result[MD5_DIGEST_LENGTH/4] = {0};
  MD5((unsigned char*)&S[0], S.length(), (unsigned char*)result);
  return result[1];
}
```

Alternativ können Sie in einer C++ Implementierung auch libcrypto++ benutzen:

```
#include <crypto++/sha.h>

uint64_t shaBuf[CryptoPP::SHA1::DIGESTSIZE/sizeof(uint64_t)+1];
int salts[2] = { rand(), rand() };


auto h1 = [&shaBuf, &salts](const std::string & str) {
  CryptoPP::SHA1 h;
```

```
  h.Update((const unsigned char*)str.c_str(), str.size());
  h.Update((const unsigned char*)(&salts[0]), sizeof(int));
  h.Final((unsigned char *) (&shaBuf));
  return shaBuf[0];
};

auto h2 = [&shaBuf, &salts](const std::string & str){
  CryptoPP::SHA1 h;
  h.Update((const unsigned char*)str.c_str(), str.size());
  h.Update((const unsigned char*)(&salts[1]), sizeof(int));
  h.Final((unsigned char *) (&shaBuf));
  return shaBuf[1];
};
```

Um ein Gefühl für die Qualität Ihrer Implementierung zu bekommen, können Sie die Hashfunktion ihrer gewählten Programmiersprache verwenden. Für C/C++ bietet sich hier `std::unordered_map` oder alternativ `sparse_hash` von google. Die Musterlösung mit crypot++ benötigt für die sehr große Instanz ca. 2.6 GiB Arbeitsspeicher, 142 Sekunden bei durchschnittlichen $3,13$ Kollisionen und einem finalen Loadfaktor von $0,97$ auf einem Core i7 4700MQ mit 2.4 GHz.