PROJEKTAUFGABE AE

Remove Duplicates - Spotify playlist cleaner

RAPHAEL DRECHSLER

INHALTSVERZEICHNIS

1 PROBLEMSTELLUNG "REMOVE DUPLICATES"

Problem
Gegeben ist eine Sequenz. Diese Sequenz enthält ggf. Duplikate.
Ziel des umzusetzenden Algorithmus ist das Entfernen der Duplikate aus dieser Sequenz.

Für die Umsetzung eines entsprechenden Algorithmus sollen die zwei folgenden Ansätze betrachtet werden:

- Naiver Ansatz: Nutzung eines einfachen Arrays
- Ansatz im Fokus: Nutzung einer Hash-Table

Naiver Ansatz
In der Implementierung nach dem naiven Ansatz würden alle Daten einer Sequenz in einem Array gespeichert werden, insofern sie nicht bereits im Array enthalten sind. Beim betrachten eines Elementes aus der Sequenz muss also in der naiven Umsetzung das komplette Array nach einem identischen Element durchsucht werden. Im in Hinblick auf die Laufzeit schlimmsten Fall, muss daher jedes Element im Array mit dem aktuell untersuchten Element des Sequenz verglichen werden. Bei einer Sequenz-Liste der Größe n müssen also im wors-case

$$\sum_{k=1}^{n-1} k \tag{1}$$

Vergleichsoperationen durchgeführt werden. Betrachtet man die folgende Umformung entsprechend der Gaußschen Summenformel

$$\sum_{k=1}^{n-1} k = \frac{(n-1)^2 + (n-1)}{2} = \frac{1}{2}(n-1)^2 \tag{2}$$

ergibt sich für eine Umsetzung dieses Ansatzes eine theoretische obere Komplexitätsgrenze von

$$O(n^2) \tag{3}$$

Ansatz Hash-Table
Der Ansatz "Hash-Table" setzt an dem Punkt der Komplexitätsbetrachtung an Würde man für die Einordnung der Elemente in das Array eine Direktadressierung verwenden, so würde man ein Element selbst als Schlüssel interpretieren, mit dem ein Feld im Array adressiert wird.

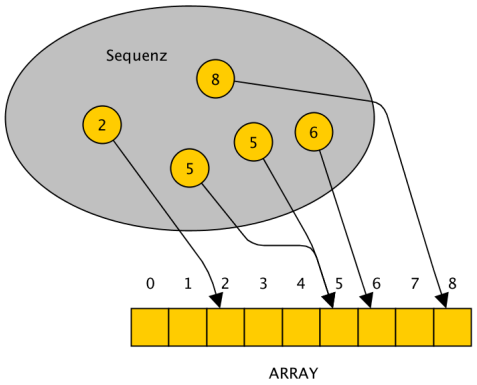

Abbildung 1: Skizze: Prinzip der direkten Adressierung *nach* [1]

Der Zeitaufwand für die Prüfung, ob ein Element bereits im Array gespeichert ist, wäre dabei $O(1)$.
Ist das Universum $U = 0, 1, ..., m$, in denen sich die Schlüssel der Elemente befinden klein, lässt sich somit schnell auf ein Array der Größe $A[0..m-1]$ zugreifen. Ab einer bestimmten Größe des Universums kann eine Umsetzung der Direkt-Adressierung Aufgrund der erforderlichen Größe des Ziel-Arrays nicht mehr sinnvoll bzw. möglich sein.
Die Hash-Table löst dieses Problem, indem sie das große Universum U einem kleineren Array $A[0..m-1]$ gegenüberstellt. Die Adressierung der Array-Felder pro Element erfolgt weiterhin auf Grundlage des Element-Wertes. Um nur existierende Schlüssel zu erhalten wird zur Ermittlung des Schlüssels eine Funktion h (sogenannte Hash-Funktion) eingesetzt, welche die Werte der Elemente im Universum U auf existierende Schlüssel abbildet. *vgl.*[1]

$$h : U -> 0, ..., m-1 \tag{4}$$

Die Skizze gestaltet sich wie folgt:

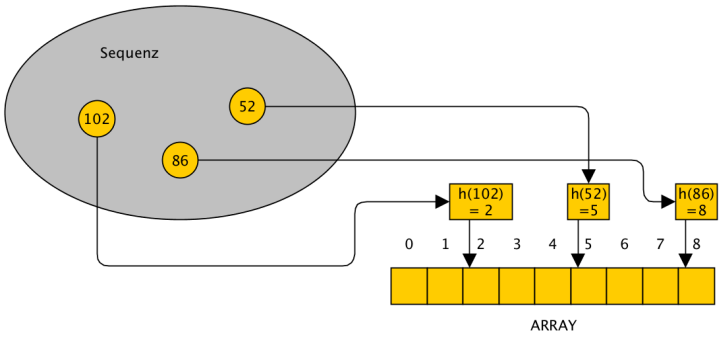

Abbildung 2: Skizze: Prinzip der Adressierung in einer Hash-Table*nach* [1]

Dadurch, dass gilt $|U| > m$ wird über Anwendung des Schubfachprinzips ersichtlich, dass die Hash-Funktion h nicht injektiv ist.
Fälle in denen gilt

$$h(a) = h(b) | a, b \in U, a \neq b \tag{5}$$

werden als Kollision bezeichnet. *vgl.*[1]
Bei der Umsetzung des Algorithmus muss also eine entsprechende Strategie zur Auflösung solcher Kollisionen mit betrachtet werden.

Kollisions-Auflösung im python dict-Objekt
Als Kollisions-Auflösungs-Strategie soll im Rahmen der Umsetzung das Verfahren implementiert werden, welches im python-Standard beim Zugreifen auf *dict*-Objekte genutzt wird. Durch die ggf. notwendige Behandlung von Kollisionen ergibt sich für einen Zugriff auf ein Feld des Arrays bei einer entsprechenden Implementierung der folgende Zeitbedarf.

Operation	average-case	worst-case
Element einfügen	$O(1)$	$O(n)$
Auf Element zugreifen	$O(1)$	$O(n)$

Tabelle 1: Zugriffs-Komplexität im python dict [2]

2 ANWENDUNGSSZENARIO SPOTIFY PLAYLISTS

Als Anwendungsfall soll das disjunkte Vereinigen von Titeln mehrerer Spotify-Playlists betrachtet werden.
In Spotify lassen sich für eine geöffnete Playlist per Tastatur-Kurzbefehl "Strg + A" und "Strg + C" alle Titel der Playlist in die Zwischenablage kopieren. Die Titel werden dabei als URI repräsentiert.

```
https://open.spotify.com/track/757530vPBymdi31CtXstxP
https://open.spotify.com/track/1Qi256uJuMihknGuuFcQoC
https://open.spotify.com/track/0JFBf2PloRfMkPg5DjXhDx
https://open.spotify.com/track/7iXF2W9vKmDoGAhlHdpyIa
```

Abbildung 3: Beispiel: Spotify-Titel URIs

Die URIs werden durch den Anwender in einem separatem *.txt*-File gesammelt. Somit werden mehrere Playlists in diesem *.txt*-File vereinigt.
Die Bereinigung der Duplikate soll nun der zu implementierende Algorithmus übernehmen.
Nach Abschluss der Bereinigung lassen sich alle URIs des *.txt*-Files in die Zwischenablage kopieren und per Tastatur-Kurzbefehl "Strg + V" in Spotify in eine (sinnvollerweise neue) Playlist einfügen.

3 FUNKTIONSWEISEN DER ALGORITHMEN

Im Rahmen der Lösungs-Umsetzung des "Remove Duplicates"-Problems sollen drei verschiedene Implementierungen in der Sprache python betrachtet werden.
Dabei soll das Hauptaugenmerk auf der ersten Implementierung liegen. Diese soll die Funktionalität der *dict*-Objekte in python nachempfinden, indem Sie die entsprechende hash-Funktion und die entsprechende Kollisions-Auflösungs-Strategie implementiert.
Zum Vergleich sollen zwei weitere Implementierungen betrachtet werden:

- der in Kapitel 1 beschriebene naive Ansatz
- eine Lösung, welche die tatsächlichen python *dict*-Objekte nutzt

Im Folgenden werden die Implementierungen mittels Pseudo-Code beschrieben.

3.1 Funktionsweise Haupt-Algorithmus: dict-like

Zunächst wird eine hash-table in Form eines Arrays initialisiert.

Haupt-Algorithmus: Hash-Tabelle initialisieren

Ergebnis : table, tableSize
1 tableSize ← 8;
2 table ← Array der Größe tableSize;

Anschließend werden die Elemente (URIs) aus der Sequenz-Datei des Nutzers in die angelegte hash-table geschrieben. Tritt dabei ein Element doppelt auf, wird dieses als Duplikat erkannt und ignoriert.
Als hash-Funktion wird dabei die python-Standardfunktion *hash()* eingesetzt. Die Strategie zur Kollisionsauflösung entspricht derjenigen, die bei python *dict*-Objekten Anwendung findet und lässt sich dem folgenden Pseudocode entnehmen.

Haupt-Algorithmus: Elemente in Array einfügen

```
   Daten : Sequenz − Datei
   Ergebnis : table
 1 für Jede URI in Sequenz − Datei tue
 2 |   index ← hash(URI);
 3 |   uriHandled ← flase;
 4 |   pertub ← None;
 5 |   tue
 6 |   |   wenn bereits Element in table[index] enthalten dann
 7 |   |   |   wenn table[index] == URI dann
 8 |   |   |   |   //Duplikat liegt vor - URI wird ignoriert
 9 |   |   |   |   uriHandled ← true;
10 |   |   |   sonst
11 |   |   |   |   wenn pertub besitzt Wert dann
12 |   |   |   |   |   pertub = pertub >> 5;
13 |   |   |   |   |   index ← ((5 ∗ slotindex) + 1 + pertub)%tableSize;
14 |   |   |   |   sonst
15 |   |   |   |   |   pertub = index;
16 |   |   |   |   Ende
17 |   |   |   Ende
18 |   |   sonst
19 |   |   |   table[index] ← URI;
20 |   |   |   uriHandled ← true;
21 |   |   Ende
22 |   solange !uriHandled;
23 |   wenn URIs in table > 2/3 ∗ tableSize dann
24 |   |   tue
25 |   |   |   tableSize ← tableSize ∗ 2
26 |   |   solange 4 ∗ |URIs in table| >= tableSize;
27 |   |   newTable ← table;
28 |   |   Kopiere alle URIs in table unter Errechnung neues Schlüssels in newTable;
29 |   |   table ← newTable;
30 Ende
```

Anschließend werden die in der hash-table enthaltenen URIs in eine Output-*.txt*-Datei geschrieben. Diese enthält damit die Duplikat-freie Liste der URIs und kann vom Anwender genutzt werden.

3.2 Funktionsweise Vergleichs-Algorithmus 1: Naiver Ansatz

Der in Kapitel 1 skizzierte naive Ansatz gestaltet sich grundlegend wie folgt.

Algorithmus Naiver Ansatz

```
   Daten : Sequenz − Datei
   Ergebnis : table
 1 table ← [];
 2 für Jede URI in Sequenz − Datei tue
 3 |   uriInTable ← false;
 4 |   für Jede processedUri in table tue
 5 |   |   wenn processedUri == URI dann
 6 |   |   |   uriInTable ← true;
 7 |   Ende
 8 |   wenn uriInTable dann
 9 |   |   //Duplikat erkannt, ignoriere es
10 |   sonst
11 |   |   table.append(URI);
12 |   Ende
13 Ende
```

Anschließend erfolgt die Ausgabe der Duplikat-freien Liste per Erzeugen einer Output-Datei.

3.3 Funktionsweise Vergleichs-Algorithmus 2: python dict

Die Funktionsweise des zweiten Vergleichs-Algorithmus ist diejenige, deren Verhalten in der Umsetzung des Haupt-Algorithmus angestrebt wird.
Folglich zeichnet sich der Quellcode der Implementierung durch die Nutzung von python-Standard-Operatoren aus. Der wesentliche Teil der Implementierung umfasst die folgenden Zeilen Code:

```
uriDict = {}
for uri in sequenzFile:
   if uri not in uriDict:
      uriDict[uri] = 1
   else:
      #ignore duplicate
```

Analog zu den anderen zwei Implementierung erfolgt im Anschluss eine Ausgabe aller Elemente des dicts per Output-File.

4 MESSUNGEN – VORGEHEN

4.1 Was wird gemessen?

DRT
Gemessen werden nur die Zeiten bis Array fertig ist. Sequenz-File öffnen und Outpuf-File schreiben ist nich.

Gemessen werden pro Algo verscheidene Instanzen: - Größe variiert - Ausgangs-File variiert

4.2 Wie wird gemessen?

Gemessen wird mit Python zeug simple Algebra weglassen output-file ausgabe ins terminal Nutzung mehrerer Ausgabe-Files Nutzung sh script

Achtung: Profiling anders -> Beschreiben in ...

4.3 Was zählt?

?wird das erst später beschrieben? -> Boxplot -> Aus

5 MESSUNGEN – DATEN

5.1 Test-Daten

5.2 Echt-Daten

5.3 Tatsächliche Messungen

Reihenfolge: - Profiling & Tailoring - VGL Test Real - Vgl mit theoretischer Laufzeit

6 PROFILING UND OPTIMIERUNG

Nur in 2 und 1 möglich. Profiling mit VizSnake

6.1 Profiling und Optimierung Haupt-Algorithmus

Nutzung von hash-Std. Funktion 3 x Verbesserung möglich
Dann noch Tailoring: Nur URI nutzen. -> Verbesserung?

6.2 Profiling und Optimierung Vergleichs-Algorithmus 1

Analog zu Erkenntnis davor. Dann noch Tailoring: Nur URI nutzen. -> Verbesserung?

7 VERGLEICH ECHT VS. TEST-DATEN

7.1 Haupt-Algorithmus

DRT
Was gibt's? Nach dem Vergleich: Auf Echt.

7.2 Vergleichs-Algorithmus 1

DRT
Was gibt's? Nach dem Vergleich: Auf Echt.

7.3 Vergleichs-Algorithmus 2

DRT
Was gibt's? Nach dem Vergleich: Auf Echt.

8 VERGLEICH MIT THEORETISCHER LAUFZEIT

8.1 Haupt-Algorithmus

8.2 Vergleichs-Algorithmus 1

8.3 Vergleichs-Algorithmus 2

9 VERGLEICH DER ALGORITHMEN

9.1 Grafischer Vergleich

Alle drei in einen Graphen. Wird bestimmt lustig.

9.2 Statistischer Hypothesen-Test

Was eig gegen wen?

LITERATUR

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[2] Jonathan Hartley. Time complexity - python wiki. https://wiki.python.org/moin/TimeComplexity. Zuletzt bearbeitet: 05.06.2017, Zugriff: 01.02.2018.