Suchalgorithmen

Theorie (L)

Inhaltsverzeichnis

Version vom 17. April 2024

1 Einleitung

Neben dem Sortieren ist das Suchen eine der häufigsten Tätigkeiten, die ein Computer ausführt. Miller und Ranum beschreiben dies in ihrem Buch wie folgt:

> Searching is the algorithmic process of finding a particular item in a collection of items.

Das Resultat einer Suche kann verschieden ausfallen:

- Wenn man nur wissen möchte, *ob* das gesuchte Objekt in der Menge vorhanden ist, genügt als Rückgabewert `True` oder `False`.
- Falls das gesuchte Objekt modifiziert werden soll, möchte man wissen, *wo* das Element innerhalb der Datenstruktur zu finden ist, sofern es überhaupt darin liegt. Möglicherweise tritt ein Wert auch mehrfach auf, so dass mehrere Positionen ermittelt werden müssen.

2 Sequentielle Suche

Voraussetzung

Die Daten sind in einer Datenstruktur abgelegt, in der – mit Ausnahme des ersten Elements – jedes Element Nachfolger von genau einem anderen Element ist.

Diese Art von Datenorganisation wird in Python durch Listen oder Tupel realisiert. Durch einen Index greift man auf die einzelnen Werte zu.

Der Algorithmus

Beginnend mit der ersten Position prüft man der Reihe nach jedes Element, bis man entweder gefunden hat, wonach man sucht oder bis man das Arrayende erreicht hat, was bedeutet, dass das Element nicht vorhanden ist.

Beispiel 2.1

Suche 8 in `A = [26, 37, 54, 8, 93, 70, 65, 82, 49]`:

26	37	54	8	93	70	65	82	49	Vergleiche
8									1
	8								1
		8							1
			8						1

Implementierung in Python

```
def linear_search(L, item):
    matches = []
    for i in range(0, len(L)):
        if L[i] == item:
            matches.append(i)
    return matches
```

Laufzeitanalyse der sequentiellen Suche

	Best Case	Average Case	Worst Case
Element $\in$ Liste	$O(1)$	$O(n/2) = O(n)$	$O(n)$
Element $\notin$ Liste	$O(n)$	$O(n)$	$O(n)$

3 Binäre Suche

Voraussetzungen

Die Werte sind ...

- in einer sequentiellen Datenstruktur abgelegt,
- in aufsteigender Reihenfolge sortiert.

Der Algorithmus (binäre Suche)

1. Setze $a = 0$ und $b = n$.
2. Wiederhole, so lange wie $a \leq b$:
   - 2.1 Bestimme $m = \lfloor (a + b)/2 \rfloor$
   - 2.2 Ist $e < A[m]$?
     ja: Setze $b = m - 1$
   - 2.3 Ist $e > A[m]$?
     ja: Setze $a = m + 1$
   - 2.4 Ist $e = A[m]$:
     ja: gibt m als Wert zurück
3. gib -1 als Wert zurück („not found“)

Beispiel 3.1

Suche 54 in `A = [8, 26, 37, 49, 54, 65, 70]`:

0	1	2	3	4	5	6		
8	26	37	49	54	65	70	Vergl.	$a = 0, b = 6, m = 3$
			54				1	$a = 4, b = 6, m = 5$
					54		1	$a = 4, b = 4, m = 4$
				54			1	gefunden

Beispiel 3.2

Suche 29 in `A = [8, 26, 37, 49, 54, 65, 70]`:

0	1	2	3	4	5	6		
8	26	37	49	54	65	70	Vergl.	$a = 0, b = 6, m = 3$
			29				1	$a = 0, b = 2, m = 1$
	29						1	$a = 2, b = 2, m = 2$
		29					1	$a = 2, b = 1$
								nicht gefunden

Iterative Implementierung in Python

```
1  def search_binary(L, item):
2      lower = 0
3      upper = len(L)-1
4      while (lower <= upper):
5          mid = (lower + upper)//2
6          if item < L[mid]:
7              upper = mid - 1
8          elif item > L[mid]:
9              lower = mid + 1
10         else:
11             return mid
12     return -1 # (semantisch) sinnloser Index
```

Laufzeitanalyse der binären Suche

Bei jedem Schritt wird die Menge der zu durchsuchenden Elemente etwa halbiert. Im schlimmsten Fall müssen wir das Verfahren so lange durchführen, bis wir eine Liste mit nur noch einem Element haben, die das gesuchte Objekt enthält oder nicht.

Anzahl Schritte	ungefähre Anzahl Elemente
1	$n/2^1$
2	$n/2^2$
...	...
k	$n/2^k$

$n/2^k = 1 \Rightarrow n = 2^k \Rightarrow k = \log_2(n) \Rightarrow T(n) \in O(\log n)$

Bemerkung

Damit die binäre Suche angewendet werden kann, müssen die zu durchsuchenden Daten in geordneter Form vorliegen.

Ist dies nicht der Fall, müssen sie zuvor mit einem Sortierverfahren in die richtige Reihenfolge gebracht werden. Die Kosten dafür betragen beim vergleichbasiertem Sortieren mindestens $O(n \log n)$.

Da die Laufzeitkomplexität fürs Sortieren bereits grösser als die der sequentiellen Suche, lohnt es sich nicht, die Daten extra zu sortieren, nur um die schnellere binäre Suche anwenden zu können.

4 String-Matching

Eine weitere zentrale Suchaufgabe besteht darin, ein Textmuster (*pattern*) p in einer Zeichenkette (*string*) t zu finden.

Dabei sollen hier Algorithmen betrachtet werden, die nach *exakten* Übereinstimmungen (*matches*) suchen.

Zeichenketten und Muster werden als Listen repräsentiert, deren Elemente die einzelnen Zeichen sind.

Anwendungen

- Textverarbeitungsprogramme
- Untersuchung von DNA- und Proteinsequenzen in der Bioinformatik
- Erkennung von Plagiaten
- Virenscanner

4.1 Naive Methode (Brute Force)

Beispiel 4.1

Suche das Textmuster `ABBA` in der Zeichenkette `ABABBCABBACB`

A	B	A	B	B	C	A	B	B	A	C	B	Vergl.
A	B	B										3
	A											1
		A	B	B	A							4
			A									1
				A								1
					A							1
						A	B	B	A			4

Analyse (Worst Case)

Text: `aaaaaaa` ($n = 7$ Zeichen)

Pattern: `aab` ($m = 3$ Zeichen, $m \leq n$)

a	a	a	a	a	a	a	Vergleiche
a	a	b					3
	a	a	b				3
		a	a	b			3
			a	a	b		3
				a	a	b	3
							(7-3+1)*3=15

Allgemein:

$O((n - m + 1)m) = O(mn - m^2 + m) = O(mn)$

Implementierung in Python

```
1  def matcher_naive(pat, text):
2      matches = []
3      n = len(text)
4      m = len(pat)
5      for i in range(0, n-m+1):
6          j = 0
7          while j < m and text[i+j] == pat[j]:
8              j += 1
9          if j==m:
10             matches.append(i)
11     return matches
```

4.2 Der Boyer-Moore-Horspool-Algorithmus

Schnelle Verschiebungen

Stimmt das Muster an irgend einer Stelle nicht mit dem entsprechenden Teilstring des Textes überein, soll es so weit wie möglich nach rechts verschoben werden, ohne einen Treffer zu verpassen.

```
B B A A A B C B B B A A B B A B A A B B B A
A B B A
→ → → A B B A
        → → → → A B B A
                → → → A B B A
                      → A B B A
```

Bei Nichtübereinstimmung (rot), kann ich das Muster so weit nach rechts verschieben, bis das letzte Zeichen im Text (blau) mit dem nächsten Zeichen im Muster übereinstimmt. Kommt das Zeichen im Muster nicht vor, kann man sogar um die Länge des Musters verschieben.

Beispiel 4.2

Suche das Muster `ABBA` (Pattern p) in der Zeichenkette `ABABBCABBACB` (Text t).

Alphabet: $\Sigma = \{\texttt{A}, \texttt{B}, \texttt{C}\}$ (jedes Symbol in $t \cup p$)

Bad Character Table (BCT): Um wie viele Positionen darf man das Muster nach rechts verschieben, wenn über seinem rechten Ende im Text das Symbol $\sigma \in \Sigma$ steht und man keinen Treffer verpassen möchte?

```
? ? ? A
A B B A    ⇒   3 Zeichen
-------------------------
? ? ? B
A B B A    ⇒   1 Zeichen
-------------------------
? ? ? C
A B B A    ⇒   4 Zeichen
```

Implementierung der BCT in Python

```
1 def bad_character_table(pat, alph):
2     m = len(pat)
3     D = dict()
4     for i in range(0, len(alph)):
5         D[alph[i]] = m
6     for i in range(0, m-1):
7         D[pat[i]] = m-i-1
8     return D
```

Zeilen 4–5: Jedem Symbol wird provisorisch die Länge des Musters $m = |p|$ zugeordnet.

Zeile 6–7: Jedem Zeichen im Muster (ausser dem Letzten) wird sein kürzester Abstand vom rechten Ende zugewiesen.

Aufwand: $O(|\Sigma| + m)$

Beispiel 4.2 (Fortsetzung)

Bad Chraracter Table:

A	B	C
3	1	4

A	B	A	B	B	C	A	B	B	A	C	B	Vergl.
A	B	B	A									1
	A	B	B	A								1
		A	B	B	A							1
						A	B	B	A			4

Implementierung in Python

```
1  def matcher_bmh(pat, text, alph):
2      matches = []
3      bct = bad_character_table(pat, alph)
4      n = len(text)
5      m = len(pat)
6      i = 0 # Position im Text
7      while (i < n-m+1):
8          j = m-1 # letzte Position im Muster
9          while (j>-1 and pat[j] == text[i+j]):
10             j = j-1
11         if j == -1: # alle Zeichen matchen
12             matches.append(i)
13         i = i + bct[text[i+m-1]] # shift aufgrund BCT
14     return matches
```

Worst Case-Analyse

Text: `aaaaaa` ($n = 6$ Zeichen)

Pattern: `baa` ($m = 3$ Zeichen, $m \leq n$)

a	a	a	a	a	a	Vergleiche
b	a	a				3
	b	a	a			3
		b	a	a		3
			b	a	a	3
						(6-3+1)*3=12

Allgemein: $O((n - m + 1)m) = O(mn - m^2 + m) = O(mn)$

Solche Text-Muster-Strukturen sind jedoch eher die Ausnahme!

Best Case-Analyse

Text: `aaaaaaa` ($n = 6$ Zeichen)

Pattern: `bbb` ($m = 3$ Zeichen, $m \leq n$)

a	a	a	a	a	a	Vergleiche
b	b	b				1
			b	b	b	1
						(6//3)*1=2

Allgemein: $O(n/m)$

Auch solche Text-Muster-Strukturen sind eher die Ausnahme.

Average Case-Analyse

Ricardo Baeza-Yates und Mireill Régnier haben in ihrem Artikel in der Fachzeitschrift *Theoretical Computer Science* 1992 gezeigt, dass die Komplexität des Boyer-Moore-Horspool-Algorithmus im Mittel $O(n)$ ist.

Bemerkung

Der hier vorgestellten Boyer-Moore-Horspool-Algorithmus (BMH) ist eine Vereinfachung des Boyer-Moore-Algorithmus' (BM), der zusätzlich zur Bad Character Table allfällige Übereinstimmungen am Ende des Musters einbezieht, um es beim ersten Mismatch eventuell noch weiter nach rechts zu verschieben.

Wenn man den Fachartikeln im Internet Glauben schenkt, so ist für natürliche Sprachen der BMH-Algorithmus dem BM-Algorithmus überlegen. Dies liegt offenbar daran, dass der BM-Algorithmus mehr Aufwand zur Verschiebung des Suchmusters betreibt, was sich in einer grösseren Anzahl von Anweisungen niederschlägt. Siehe z. B.:

`https://www.ncbi.nlm.nih.gov/pmc/articles/PMC61442/`