Universität Hamburg

DER FORSCHUNG | DER LEHRE | DER BILDUNG

Universität Hamburg
Fakultät für Mathematik, Informatik und Naturwissenschaften

Abschlussbericht

64-174 Seminar Effiziente Programmierung in C
Algorithmen
Wintersemester 13/14

Verfasser:

Marcel Hellwig

Betreuer:

Nathanael Hübbe

Vorgelegt am: 15. März 2014

Inhaltsverzeichnis

1 Einleitung **3**

2 Dynamische Programmierung **4**
2.1 Einleitung 4
2.2 Beispiel 4
2.3 Fazit 5

3 Divide and Conquer **6**
3.1 Einleitung 6
3.2 Max-Subarray-Problem 6
3.3 Fazit 7

4 Greedy **8**
4.1 Einleitung 8
4.2 Huffman-Coding 8
4.3 Fazit 9

5 Zusammenfassung **11**

1 Einleitung

Algorithmen sind eine abstrakte Beschreibung für eine Folge von nacheinander folgenden Handlungen. Dies kann eine Beschreibung für das Binden von Schnürsenkeln sein, aber auch wie man einen Kuchen backt oder einen Schrank aufbaut.

Im informatischem Sinne versteht man unter einem Algorithmus eine Nacheinanderausführung von Befehlen die ein Computer oder eine Abstrakte Maschine ausführen soll.Dargstellt kann so ein Algorithmus entweder mit Hilfe von Flow-Charts (Abbildung 1.1), oder auch mit Hilfe von Pseudocode (Abbildung 1.2) dargestellt werden. Die eigentliche Absicht dahinter ist, dass man den Algorithmus bei genauerem betrachten nachvollziehen und verstehen kann.

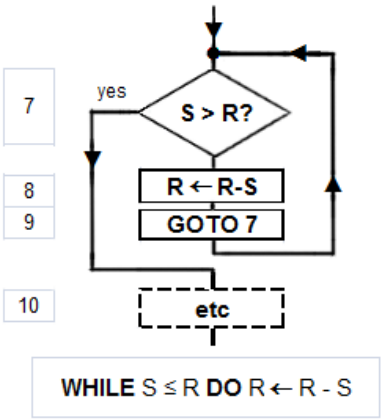

Abbildung 1.1: Flow-Chart

```
procedure bubbleSort( A : list of sortable items )
   repeat
     swapped = false
     for i = 1 to length(A) - 1 inclusive do:
       //if this pair is out of order
       if A[i-1] > A[i] then
         //swap them and remember something changed
         swap( A[i-1], A[i] )
         swapped = true
       end if
     end for
   until not swapped
end procedure
```

Abbildung 1.2: Bubble Sort mit Pseudocode

Im folgenden werden drei verschiede Klassen von Algorithmen vorgestellt und ihre Anwendung anhand eines Beispiels besprochen. Auch werden ihre Stärken und Schwächen aufgezeigt.

2 Dynamische Programmierung

2.1 Einleitung

Unter dynamischer Programmierung versteht man das Lernen eines Programms anhand seiner bisherigen Ausführung. Ein sehr einfaches Beispiel wird hier besprochen. Weitaus komplexere Beispiele werden in dem Gebiet der neuronalen Netze behandelt. Die Dynamische Programmierung ist hierbei nicht explizit Teil des Gebiets, bildet aber die Grundlage des Ganzen.

2.2 Beispiel

Das hier gezeigte Beispiel bezieht sich auf die Fibonacci Folge. Sie ist eine unendliche Folge, die sich rekursiv aus der Addition des Vor- und Vorvorgänger berechnet. Sie ist eine der klassischen Beispiele für die dynamische Programmierung. Hier eine Berechnungsfunktion in C.

$$f(n) = \begin{cases} n & \text{, if } n < 2 \\ f(n-1) + f(n-2) & \text{, else} \end{cases}$$

Abbildung 2.1: Definition der Fibonacci Folge

```c
uint64_t fibonacci(unsigned int n)
{
    if (n < 2) return n;

    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Abbildung 2.2: Fibonacci in C

Diese Funktion berechnet (baum-)rekursiv den Fibonacciwert zu einem gegeben n. Die Laufzeit ist dabei $\Theta(n^3)$. Dies ist ein großes Problem, denn für 10, beträgt die theoretische Laufzeit schon 1000, bei 50 125.000. Abhilfe schafft es die bereits berechneten Werte in einer Liste zu speichern. Ein dazu entsprechender Code ist in Abbildung 2.3 zu sehen.

```c
static uint64_t *results;

static uint64_t fib_help(unsigned int n)
{
    if (n < 2) return n;
    if (results[n] == 0)
        results[n] = fib_help(n - 1) + fib_help(n - 2);
    return results[n];
}

```

```
11 uint64_t fibonacci(unsigned int n)
12 {
13     results = calloc(n+1, sizeof(*results));
14     uint64_t res = fib_help(n);
15     free(results);
16     return res;
17 }
```

Abbildung 2.3: Fibonacci mit dyn. Programmierung

Die nötige Tabelle wird in Zeile 13 initialisiert und in Zeile 15 freigegeben. Die eigentliche Berechnung findet in der Funktion *fib_help* statt. In Zeile 6 wird überprüft, ob der geforderte Wert bereits berechnet wurde, wenn nicht, wird er berechnet, in die Tabelle eingefügt und anschließend zurückgegeben.
Das Problem an dieser Variante ist, dass $2 \cdot n$ Speicher verbraucht wird. Mit einiger Überlegung kommt man drauf, dass man lediglich die lezten zwei Elemente der Berechnung speichern muss. Dies resultiert in folgenden, iterativen Code.

```
1  uint64_t fibonacci(unsigned int n)
2  {
3      uint64_t last = 0;
4      uint64_t cur = 1;
5
6      for(unsigned int i = 0; i < n - 1; i++)
7      {
8          uint64_t tmp = last + cur;
9          last = cur;
10         cur = tmp;
11     }
12
13     return cur;
14 }
```

Abbildung 2.4: Iterative Fibonacci Variante

2.3 Fazit

Dynamische Programmierung ist immer dann hilfreich, wenn Ergebnisse häufiger gebraucht werden, oder sich ein aktuelles Ergebnis auf frühere Ergebnisse zurückführen lässt. Wie viele oder welche Elemente muss von Fall zu Fall neu bedacht werden. Auch muss ein Kompromiss zwischen verfügbaren Speicher und Rechenzeit gefunden werden. Auf eingebetteten System steht meist nicht viel Speicherplatz zur Verfügung, demnach muss mehr Rechenzeit erbracht werden. Auf modernen PC-System hingegen spielt Speicher selten eine Rolle.

3 Divide and Conquer

3.1 Einleitung

Unter Divide and Conquer versteht man ein Paradigma, bei dem es darum geht ein komplexes Problem zu lösen, indem man es in viele, gleichartige Probleme zerteilt und diese dann löst. Ein großer Vorteil von D&D ist die Parallelisierbarkeit der Teilprobleme, da diese in den meisten Fällen disjunkt sind. Auf Maschinen mit vielen Recheneinheiten (CPU oder GPU-Computing) können solche Arten von Probleme effizient gelöst werden. Da sich Lösungen allerdings auch zwischen den Grenzen befinden können, müssen diese gesondert betrachtet werden. Ein Zwischenproblem kann aber erst dann gelöst werden, wenn die zwei Teilprobleme die die Grenzen bilden gelöst wurden. Auch muss am Ende über alle Teilprobleme der beste Wert (o.ä.) gefunden werden. Dies führt zu einer allgemeinen Zeitkomplexität von $O(n \cdot log(n))$.

3.2 Max-Subarray-Problem

Das Max-Subarray-Problem beschäftigt sich mit der Findung einer Summe in einem gegeben Array, wobei die Summe Maximal in einem Teilinterval sein soll.

$$12\ -13\ \underbrace{78\ -3\ 23\ -46\ 92\ 23}_{\sum = 167}\ -13\ -35$$

Abbildung 3.1: Ein (gelöstes) Max-Subarray-Problem

Das Array wird mithilfe eines C-Arrays mit einer gegebenen Länge repräsentiert. Das Aufteilen funktioniert rekursiv bis zu einem Rekursionsabbruch, in diesem Fall bis zur Länge 1. Falls die Länge größer eins ist, wird das Array in ein linkes und ein rechts Teilarray zerlegt und daraus rekursiv das Maximum berechnet.

```
1  MaxSubResult sub_dc(int arr[], uint l, uint u)
2  {
3      if (l == u)
4          return (MaxSubResult){
5              .sum = arr[l] > 0 ? arr[l] : 0, .start = l, .end = l
6          };
7
8      int m = (u+l) / 2;
9      MaxSubResult a = sub_dc(arr, l, m);
10     MaxSubResult b = sub_dc(arr, m+1, u);
11     MaxSubResult c = inBetween(arr, l, m, u);
12
13     MaxSubResult max = a.sum >= b.sum ? a : b;
14     max = max.sum >= c.sum ? max : c;
```

```
15      return max;
16  }
```

Abbildung 3.2: Max-Subarray: Divide and Conquer Ansatz

Anschließend wird die vorhin erwähnte Zwischenlösung berechnet. Der Code dafür ist in Abbildung 3.3 zu sehen. Hier wird zuerst das links Teilarray betrachtet und von der Mitte ausgehend ein Maximum berechnet. Danach wird das rechte Teilarray betrachtet und ebenfalls das Maximum errechnet. Sinn hierbei ist es, das Maximum über die Grenze m zu berechnen, da diese in den rekursiven Abschnitten zuvor nicht berücksichtigt wurde.

```
1   MaxSubResult inBetween(int arr[], uint l, uint m, uint u)
2   {
3       MaxSubResult result = { .sum = 0, .start = m, .end = m };
4
5       int64_t i;
6       int sum = 0;
7       for (i = m; i >= l; i--) {
8           sum += arr[i];
9           if (sum > result.sum) setSub(&result, sum, i, m);
10      }
11
12      sum = result.sum;
13      for (i = m+1; i <= u; i++) {
14          sum += arr[i];
15          if (sum > result.sum) setSub(&result, sum, result.start, i);
16      }
17
18      return result;
19  }
```

Abbildung 3.3: Max-Subarray: Zwischenlösung

3.3 Fazit

Divide and Conquer ist genau dann ein gutes Mittel, wenn sich ein Problem in mehrere gleichartige, disjunkte Teilprobleme zerlegen lässt. Diese Teilprobleme können u.U. parallel berechnet werden, was die Gesamtlaufzeit stark reduzieren kann. Dennoch müssen die Subprobleme danach noch linear zusammengefügt werden um das Problem in seiner Gesamtheit zu lösen. Diese Methode resultiert in einer allgemeinen Komplexität von $O(n \cdot log(n))$, wobei der $log(n)$ Teil dem rekursiven Abstieg entspricht und n dem Zusammenfügen der Teilergebnisse.

4 Greedy

4.1 Einleitung

Greedy Algorithmen sind eine sehr spezielle Art von Algorithmen. Sie unterscheiden sich einerseits dadurch, dass sie nicht stets das globale Maximum, sondern lediglich ein lokales suchen (welches durchaus das Globale sein kann) und es auch mehrere Lösungen geben kann. Ein Beispiel ist in Abbildung 4.1 zu sehen.

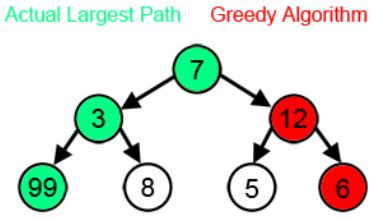

Abbildung 4.1: Greedy Beispiel

Der Algorithmus geht hier so vor, dass er für den aktuellen Knoten den besten Wert auswählt und über die Kante den Knoten besucht. Wie zu erkennen, ist die Summe der Knotenwerte nicht das globale Maximum (109), aber ein lokales (25). Dies kann in Verbindung mit Heuristiken nützlich sein, da sich die Zeitkomplexität erheblich senkt, denn es wird nicht der komplette Baum Untersucht, sondern der aktuelle Knoten und die dazugehörigen Kanten.
Diese Eigenschaft macht Greedy zu einer sehr speziellen, gleichzeitig mächtigen Klasse von Algorithmen. Beispielhaft wird hier der Huffman Code gezeigt, mit dem man eine Präfixfreie Codewörter erzeugen kann.

4.2 Huffman-Coding

In Abbildung 4.2 ist ein vollständiger Huffman Baum zu sehen. Der Weg dahin führt über mehrere Zwischenschritte, bei denen einige nicht deterministisch sind, was zu unterschiedlichen Lösungen führen kann.
Der erste Schritt ist es die Eingabesymbole der Wahrscheinlichkeit (oder der absoluten Häufigkeit) aufsteigend zu sortieren.

Buchstabe	b	p	’	m	j	o	d	a	i	r	u	l	s	e	’ ’
Abs. Häufigkeit	1	1	2	2	3	3	3	4	4	5	5	6	6	8	12

Tabelle 4.1: Häufigkeitsverteilung der Buchstaben

Anschließend nimmt man die zwei Zeichen mit der geringsten Wahrscheinlichkeit, fügt sie zu einem Knoten zusammen und addiert ihre Wahrscheinlichkeit und fügt sie geordnet in die Liste ein. Danach werden wieder die zwei Zeichen mit der geringsten Wahrscheinlichkeit genommen,

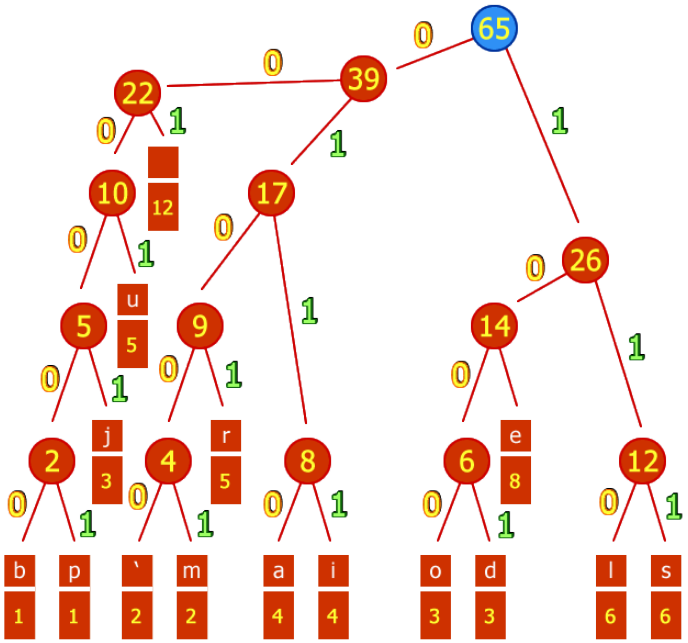

Abbildung 4.2: Huffman Baum

ein Knoten gebildet und ihre Wahrscheinlichkeit addiert. Dies wiederholt man so lange, bis die Liste leer ist.

Um nun das Codewort zu erhalten beschriftet man alle Kanten die links von einem Knoten abgehen mit einer 0 und alle die rechts abgehen mit einer 1 (man kann dies auch umdrehen, muss dies aber global eindeutig machen). Anschließend kann man das Codewort von der Wurzel zum Blatt ablesen, indem man die 0 und 1 konkateniert.

Buchstabe	b	p	’	m	j	o	d
Codewort	000000	000001	01000	01001	00001	1000	1001
Codewort (alt.)	000000	000001	10000	10001	00001	0100	0101

Buchstabe	a	i	r	u	l	s	e	’ ’
Codewort	0110	0111	0101	0001	110	111	101	001
Codewort (alt.)	1001	0001	1100	1101	011	111	101	001

Tabelle 4.2: Codewörter für das gegebene Alphabet

Das alternative Codewort entsteht dadurch, dass man sich bei drei Symbolen mit gleicher Wahrscheinlichkeit für eine andere Zusammensetzung entscheidet. Somit ist die Lösung keineswegs eindeutig, aber dennoch korrekt, minimal und präfixfrei.

4.3 Fazit

Greedy Algorithmen sind immer dann eine gute Lösung, wenn man aus einer Vielzahl an Lösungen nur eine interessant ist. Häufig ist dies bei Graphen und Pfadfindungsalgorithmen

der Fall. Hier wird meistns nicht *der* Pfad gesucht, sondern ein beliebiger Pfad, der bestimmte Kriterien erfüllt, wie z.B. Minimaler Spannbaum.
Weitere Vertreter dieser Familie sind z.B. Prims Algorithmus oder Kruskal.

5 Zusammenfassung

Es gibt eine Vielzahl an unterschiedlichen Algorithmen und es ist gut sie zu kennen. Sowohl ihre stärken als ihre schwächen um sie in der richten Situation sinnvoll anwenden zu können. Es gibt noch weitere Klassen, wie z.B. die Scanline, Backtracking oder Pruning.
Die hier vorgestellten Beispiele dienen der Anschauung und Verständnis des jeweiligen Themas. Es gibt weitere Verteter ihrer Klasse und unzählige weitere Beispiele. Man kann auch Algorithmen miteinander kombinieren. Dies hängt immer von der Problemstellung und der jeweiligen Platform ab, auf der man entwickelt.
Es ist prinziepiell eine gute Idee viele verschiedene Algorithmen zu kennen oder auch Problemstellungen, wie z.B. das „Travelling salesman problem“ und sein Problem auf bereits wohl untersuchte Probleme zurückzuführen um dann eine bereits erarbeitete Lösung nutzen zu können.

Lizenz und Quellen

1.1 – https://upload.wikimedia.org/wikipedia/commons/d/d3/Euclid_flowchart_1.png
1.2 – https://en.wikipedia.org/wiki/Bubble_sort#Pseudocode_implementation
4.1 - http://en.wikipedia.org/wiki/File:Greedy-search-path-example.gif
4.2 – http://en.wikipedia.org/w/index.php?title=Huffman_coding&oldid=599247528