Algorithmen II

Lukas Abelt
lukas.abelt@airbus.com

DHBW Ravensburg
Wirtschaftsinformatik

Ravensburg
19. Mai 2019

Outline

Inhalt

Inhalt

Allgemeines

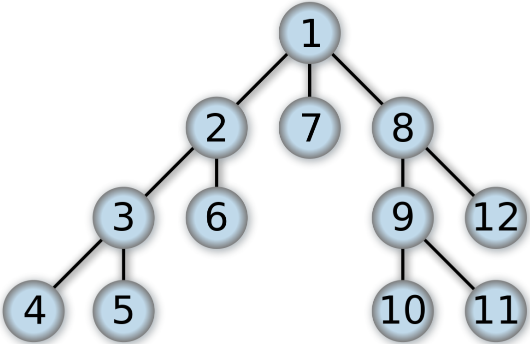
Eigenschaften

- Grundlegende Algorithmenstrategie
- Findet theoretisch:
  - alle Lösungen für ein gegebenes Problem...
  - ...in einer endlichen Zeit
- Nutzt das **trial and error** Prinzip

Methodik

- Sind rekursiv implementiert
- Es werden inkrementell Teillösungen „ausprobiert“
- Wird eine Teillösung als ungeeignet erkannt, wird diese verworfen
- Implikationen daraus:
  - Problem muss ein Kriterium für Nicht-Erfüllbarkeit liefern
  - Oder anders: Die Lösung muss bestimmte Bedingungen erfüllen
  - Man spricht in der Regel von **Constraint-Satisfaction-Problems**
- Visualisieren lässt sich das z.B. als Entscheidungsbaum

Entscheidungsbaum



Quelle: [1]

Vor- und Nachteile

- Es werden alle (sinnvollen) Lösungen ausprobiert
- Umgekehrt kann definitiv ausgesagt werden, dass keine Lösung existiert, wenn Sie nicht über Backtracking gefunden werden kann
- Jedoch sehr ineffizient
  - Komplexität im Worst Case ist $O(z^n)$ (Wobei z der Verzweigungsgrad ist)
  - Somit ergibt sich für alle $z > 1$ eine exponentielle Laufzeit
  - Daher eher für Probleme mit kleinem Lösungsbaum geeignet

Anwendungen I

Von Backtracking (Vgl. [5])

- **Damenproblem**
  - Auf einem $n \times n$ Schachfeld sollen n Damen so platziert werden, dass sie sich nicht gegenseitig schlagen können
- **Springerproblem**
  - Auf einem $M \times N$ Schachfeld soll ein Springer einen Weg finden, durch den jedes Feld **genau einmal** besucht wird.
- **Sudoku**
- **Färbeproblem**
  - Eine Landkarte mit B Ländern soll mit N verschiedenen Farben eingefärbt werden
  - Gesucht wird eine Einfärbung, bei der angrenzende Länder immer verschiedene Farben haben

Anwendungen II

Von Backtracking (Vgl. [5])

- **Wegsuche in Graphen**
  - Hierzu gehört zum Beispiel auch das finden eines Weges in einem Labyrinth
- Viele Backtracking Probleme sind **NP-vollständig**

Inhalt

Motivation

- Das Lösen großer Probleme fällt oft schwer
  - Sowohl für Menschen
  - Als auch Computer
- Meist ist es einfacher, das Problem in Teilprobleme zu unterteilen
- ...und diese separat voneinander zu lösen
- Wir machen dies oft ganz automatisch
- Bei Algorithmen spricht man hierbei vom **Divide and Conquer** Verfahren

Simples Beispiel

Multiplikation

$$a \cdot b = \overbrace{b + b + \cdots + b}^{a \text{ mal}}$$

$$a \cdot b = \overbrace{(b + b + \cdots + b)}^{\frac{a}{2}\text{mal}} + \overbrace{(b + b + \cdots + b)}^{\frac{a}{2}\text{mal}}$$

$$a \cdot b = 2 \cdot \overbrace{(b + b + \cdots + b)}^{\frac{a}{2}\text{mal}}$$

Grundsätze

Des Divide and Conquer Verfahrens

- Ein gegebenes Problem wird aufgeteilt...
- ...in zwei (oder mehr) kleinere Teilprobleme
- Dies geschieht rekursiv solange...
- ...bis sich die Teilprobleme trivial direkt lösen lassen
- Am Ende werden die Teilergebnisse zur Gesamtlösung zusammengefügt
- Es gibt ein ähnliches Vorgehen, bei dem man das Problem lediglich auf *ein* kleineres Teilproblem reduziert
  - Dies nennt man auch **Decrease and Conquer**

Vorteile

Von D&C Algorithmen (Vgl. [6])

- **Starke Lösungsstrategie**
  - Hilft dabei, Lösungen für komplexe Probleme zu finden
  - Solange man einen Weg findet, das Problem in kleinere Subprobleme zu teilen
- **Algorithmeneffizienz**
  - Der D&C Ansatz hat oft effizientere Algorithmen für bekannte Probleme gefunden
  - Zum Beispiel: Quicksort, Mergesort und FFT
  - Wenn sich ein Problem der Größe n immer in p Teilprobleme der Größe $\frac{n}{p}$ teilen lässt, so ist die Komplexität von $O(n \log_p n)$

Vorteile

Von D&C Algorithmen (Vgl. [6])

- **Parallelisierung**
  - Teilprobleme können oft parallel bearbeitet werden
  - Dadurch teils erhebliche Zeitersparnis
- **Speicherzugriff**
  - D&C Algorithmen können den Speicher meist effizienter (=schneller) nutzen
  - Wenn die Teilprobleme klein genug sind können diese ggf. direkt im Prozessorcache berechnet werden
  - Dieser ist im Vergleich zum RAM deutlich schneller durch höhere Taktraten und physische Nähe

Herausforderungen

Bei der Implementierung von D&C Algorithmen (Vgl. [6])

- **Rekursion**
  - Implementierung erfolgt in der Regel über rekursive Aufrufe
  - Dies ist häufig komplexer in der Implementierung und dem Verständnis
- **Aufruftiefe**
  - Je nachdem wie oft der rekursive Aufruf erfolgt führt das zu Problemen
  - Je nach Sprache und Compiler ist ggf. nur eine Rekursionstiefe möglich
  - Dies kommt durch die ggf. beschränkte Größe des *Call Stacks*
  - Bei zu tiefer Rekursion kann es so zum *Stack Overflow* kommen

Herausforderungen

Bei der Implementierung von D&C Algorithmen (Vgl. [6])

- **Auswahl des „trivialen Problems“**
  - Die Auswahl des direkt lösbaren Teilproblems ist nicht immer direkt ersichtlich
  - In einigen Fällen ist ein Teilen bis zum kleinstmöglichen Teilproblem nicht sinnvoll
  - Und effizienter ist es ein größeres Teilproblem direkt zu lösen
  - Beispiel: Determinantenberechnung in Matrizen

Suchalgorithmen

Motivation

- Dienen dazu mit großen Datenmengen zu arbeiten
- Um bestimmte Informationen zu finden
- Beispiele (Digital und analog):
  - Finden einer Übersetzung im Wörterbuch
  - Finden von Websites
  - Suchen von bestimmten Buchabschnitten nach Thema (über Inhaltsverzeichnis oder Index)

Allgemeine Aspekte

Der Suche

- Oft wird nach den *Werten* für bestimmte *Schlüssel* gesucht
- Die Suche in einer beliebigen Sammlung von Daten ist in der Regel nur schwer optimierbar
  - Man muss jedes Element der Sammlung einzeln betrachten um ein bestimmtes Element zu finden
  - Komplexität: $O(N)$
- Aus diesen Gründen werden zum suchen teils spezielle Datenstrukturen verwendet:
  - Symboltabellen
  - Hashtables
  - Suchbäume

Allgemeine Aspekte

Der Suche

- Gemeinsamkeit der Suchstrukturen:
  - Sind meist nach einem bestimmten Kriterium sortiert
- Dadurch lassen sich die Strukturen deutlich einfacher durchsuchen
- Stichwort: **Binärsuche**

Inhalt

Inhalt

Insertion Sort

Grundprinzip (Siehe [3] S. 85ff)

- Gegeben sein eine Liste mit N Elementen
- Jedes Element wird nacheinander betrachtet
- Und an der korrekten Stelle der bereits betrachteten Elemente eingefügt
- Dadurch ergibt sich:
  - Eine bereits sortierte Teilliste
  - Eine Restliste mit den noch einzusortierenden Elementen

Insertion Sort

Praktisches Beispiel

Vor- und Nachteile (Siehe [3] S. 85ff)

- Implementierung ist relativ simpel
- Jedoch viele Vergleiche und ggf. Verschiebungen nötig
- Komplexität beträgt hierfür $O(N^2)$

Inhalt

Bubble Sort

Grundprinzip (Siehe [3] S. 89ff)

- Eine Liste wird Elementweise betrachtet
- Jedes Element wird mit seinem Nachfolger verglichen
- Ist der Nachfolger kleiner, so werden die Elemente getauscht
- Dies wird solange wiederholt, bis die komplette Liste durchlaufen wurde ohne, dass eine Vertauschung durchgeführt wurde
- Der Name „Bubble“ leitet sich davon ab, dass die größten Element sich am oberen Ende der Liste wie eine „Blase“ sammeln

Insertion Sort

Praktisches Beispiel

Vor- und Nachteile

Siehe [3] S. 89ff

- Wohl mit der simpelste Algorithmus
- Jedoch ineffizient $\rightarrow$ Komplexität $O(N^2)$
- Auch wenn z.B. Insertion Sort die gleiche Komplexität hat ist dieser in der Regel deutlich schneller
- Daher nur wenige sinnvolle praktische Anwendungen:
  - Beispielsweise erkennen (und korrigieren) von sehr kleinen Fehlern in „beinahe sortierten“ Arrays (Anwendung in der Computergrafik)

Inhalt

Selection Sort

Grundprinzip (Vgl. [3] S. 82f, [4], S. 272f)

- Gegeben ist eine Liste mit Elementindizes 1 bis N
- Beginnend mit $M = 1$ führe folgende Schritte durch:
  - Suche das Minimum der Liste im Bereich $M \dots N$
  - Tausche das Minimum mit dem Element an der Stelle M
  - Wiederhole diesen Vorgang für die Teilliste von $M + 1 \dots N$ (Solange bis $M = N$)

Inhalt

Heapsort

Grundprinzip (Vgl. [2], S. 12ff)

- Sortiert nicht direkt Listen sondern nur die spezielle Struktur „Heap“
- Das heißt die Daten müssen entweder in dieser Form vorliegen
- ...Oder erst in diese Struktur umgewandelt werden
- Heap kann als Binärbaum interpretiert werden
- Heapsort besteht aus dem wiederholten Entfernen der Wurzel...
- ...Und dem nachfolgenden „versickern“ der restlichen Element

Heap

Definition

Heap (Definition als Liste, Vgl. [2], S. 12ff)

Eine Folge $F = k_1, k_2, \ldots, k_n$ von n Schlüsseln nennen wir dann Heap, wenn

$$k_i \leq k_{\frac{i}{2}}$$

Heap

Heap (Binärbaum)

Ein Heap ist ein vollständiger Binärbaum, in dem der Schlüssel jedes Knotens mindestens so groß ist wie der Schlüssel seiner Söhne

Heapsort

Vorgehen Sortieren

- Gebe den Wurzelknoten des Baumes aus und entferne diesen
- Setze das letzte Element im Baum an die Wurzel
- Versickere die neue Wurzel im Baum
- Wiederhole den Prozess bis der Baum leer ist

Heapsort

Versickern (Vgl. [2], S. 12ff)

Vorgehen Versickern

- Vergleiche den Wurzelknoten mit dem größten Kindknoten
- Ist der Wurzelknoten kleiner als der größte Kindknoten:
  - Vertausche den Kindknoten mit dem Wurzelknoten
  - Wiederhole dies bis beide Kindknoten kleiner als der Wurzelknoten sind (Bzw. keine Kindknoten mehr vorhanden sind)

Inhalt

Quicksort

Grundprinzipien (Vgl. [3] S. 93ff, [4] S. 313-330)

- Basiert auf dem **Divide and Conquer** Prinzip
- Es wird ein Vergleichselement x gewählt
- Und die Liste in eine linke und rechte Teilliste gliedert
  - Linke Teilliste: Alle Elemente sind kleiner(oder gleich) x
  - Rechte Teilliste: Alle Elemente sind größer(oder gleich) x
- Führe wieder Quicksort auf den beiden Teillisten aus

Grundlegendes Vorgehen

Quicksort (Siehe [2] S. 8ff)

- Setze linken Zeiger i auf das erste Element
- Setze rechten Zeiger j auf das letzte Element
- Solange $i < j$
  - Erhöhe i bis $a[i] \geq x$
  - Verringere j bis $a[j] \leq x$
  - Wenn $i < j$, dann vertausche die Elemente

Slowsort

Ein humoristischer Ansatz an Sortierungen

- 1986 von Andrei Broder und Jorge Stolfi entwickelt
- Teil ihres Papers „Pessimal Algorithms and Simplexity Analysis"
- Ziel war ein möglichst ineffizienten Algorithmus zu schaffen
  - Ohne Nutzung von zufälligen Faktoren
  - ...und ohne „überflüssige“ Operationen einzubauen
- Basiert auf dem **Multiply and Surrender** (Parodie auf Divide and Conquer) Prinzip

Slowsort

Ablauf

- Besteht im Grund aus zwei Schritten:
  - 1. Finde das Maximum der Liste und platziere es am Ende
  - 2. Sortiere die verbleibende Teilliste
- Ineffizienz kommt durch die rekursive Umsetzung des ersten Schritts:
  - 1.1 Finde (rekursiv) das Maximum der ersten Listenhälfte
  - 1.2 Finde (rekursiv) das Maximum der zweiten Listenhälfte
  - 1.3 Vergleiche die Maxima und tausche ggf.
- Die untere Grenze der Komplexität lässt sich angeben mit $\Omega(n^{\frac{\log_2(n)}{2+\epsilon}})$
- Damit ist selbst der Best-Case schlechter als der Worst-Case von Bubble Sort

Quellen I

[1] Wikimedia Commons. *File:Depth-first-tree.svg — Wikimedia Commons, the free media repository*. 2014. URL: `https://commons.wikimedia.org/w/index.php?title=File:Depth-first-tree.svg&oldid=143002747` (besucht am 29. 04. 2019).

[2] Prof. E. Fahr. *Theoretische Informatik II - Algorithmen*. 2016.

[3] T. Ottmann und P. Widmayer. *Algorithmen und Datenstrukturen*. Spektrum Akademischer Verlag, 2017. ISBN: 9783662556498.

[4] K. Wayne und R. Sedgewick. *Algorithmen. Algorithmen und Datenstrukturen*. Pearson, 2014. ISBN: 978-3-86894-184-5.

Quellen II

[5] Wikipedia. *Backtracking — Wikipedia, Die freie Enzyklopädie*. 2019. URL: `https://de.wikipedia.org/w/index.php?title=Backtracking&oldid=186371836` (besucht am 29. 04. 2019).

[6] Wikipedia contributors. *Divide-and-conquer algorithm — Wikipedia, The Free Encyclopedia*. 2019. URL: `https://en.wikipedia.org/w/index.php?title=Divide-and-conquer_algorithm&oldid=886471841` (besucht am 30. 04. 2019).

Kontakt

- E-Mail: `lukas.abelt@airbus.com`
- GitHub: `https://www.github.com/LuAbelt`
- GitLab: `https://www.gitlab.com/LuAbelt`
- Telefon(Firma): 07545 - 8 8895
- Telegram: LuAbelt