Entscheidungsbäume

Frage: Gibt es einen Sortieralgorithmus mit $o(n \log n)$ Vergleichen?

Definition Entscheidungsbaum

Sei T ein Binärbaum und $A = \{a_1, \ldots, a_n\}$ eine zu sortierenden Menge. T ist ein *Entscheidungsbaum* für A, falls

1. innere Knoten einen Vergleich von $a_i, a_j \in A$ enthalten. Falls $a_i < a_j$ wird nach links verzweigt, sonst nach rechts.
2. Blätter eine Permutation $a_{\pi(1)}, \ldots, a_{\pi(n)}$ enthalten, die direkt aus den Vergleichen folgt.

Beispiel: Entscheidungsbaum T für a_1, a_2, a_3

- 3 paarweise Vergleiche genügen, um 3 Elemente zu sortieren.
- Damit besitzt T Tiefe 3.
- Ferner besitzt T als Blätter die $3! = 6$ Permutationen von A.

Untere Schranke für die Anzahl Vergleiche

Satz Untere Schranke für die Anzahl Vergleiche

Jeder vergleichsbasierte Sortieralgorithmus benötigt zum Sortieren von n Elementen $\Omega(n \log n)$ Vergleiche im worst-case.

Beweis:

- Sei T ein Entscheidungsbaum für A, $|A| = n$ der Höhe h.
- Damit muss T als Blätter alle Permutationen von A enthalten.
- D.h. T besitzt mindestens $n!$ Blätter. Jeder Binärbaum der Höhe h besitzt andererseits höchstens 2^h Blätter.
- Daher gilt $2^h \geq n!$ bzw. $h \geq \log_2(n!)$.
- Aus der Stirling-Formel folgt $n! > \left(\frac{n}{e}\right)^n$ und damit

$$h \geq n \log_2 \left(\frac{n}{e}\right) = n \log n - n \log e = \Omega(n \log n).$$

- D.h. es gibt einen Pfad in T mit $\Omega(n \log n)$ Vergleichen.

Berechnung von Binomialkoeffizienten

Problem: Berechne aus $n, k \in \mathbb{N}_0$ effizient $\binom{n}{k}$.

- Wir kennen bereits die Rekursion $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$.
- Berechnen Binomialkoeffizienten mittels Divide and Conquer.

Algorithmus DIV-BINOM

EINGABE: $n, k \in N_0$

1. If $(k = 0)$ return 1;
2. If $(n < k)$ return 0;
3. return DIV-BINOM$(n-1, k-1)$ + DIV-BINOM$(n-1, k)$;

AUSGABE: $\binom{n}{k}$

- **Korrektheit:** Folgt aus obiger Rekursionsformel.

Laufzeit von DIV-BINOM

Satz Laufzeit von DIV-BINOM

DIV-BINOM benötigt zur Berechnung von $\binom{2n}{n}$ mindestens $\frac{4^n}{2n+1}$ Aufrufe.

Beweis:

- DIV-BINOM bricht die Rekursion mit Werten 0 und 1 ab.
- $\binom{n}{k}$ wird als Summe der Rückgabewerte 0 und 1 berechnet.
- Damit benötigt man mindestens $\binom{n}{k}$ viele Aufrufe.
- Es gilt $\binom{2n}{n} \geq \binom{2n}{i}$ für $i = 0, \ldots, 2n$.
- Daraus folgt

$$(2n+1)\binom{2n}{n} \geq \sum_{i=0}^{2n} \binom{2n}{i} = 2^{2n} = 4^n.$$

- D.h. DIV-BINOM benötigt mindestens $\frac{4^n}{2n+1}$ Aufrufe.

Dynamische Programmierung

Nachteile des Divide and Conquer Ansatzes

- Subprobleme sind nicht unabhängig voneinander.
- Zwischenergebnisse werden nicht gespeichert.
- Damit werden die Subprobleme oft mehrfach gelöst.

Definition Paradigma Dynamische Programmierung

Dynamische Programmierung verwendet die Strategien

1. **Bottom-up** Ansatz: Man beginnt bei trivialen Subproblemen.
2. **Kombination** gespeicherter Lösungen von Subproblemen: Setze Lösungen zu Lösungen größerer Probleme zusammen.

Beispiel aus der Vorlesung

- Warschall-Algorithmus berechnet Erreichbarkeit in (V, E).
  - **Bottom-up**: $W_0[u, v] = 1$ falls $(u, v) \in E$. Sonst $W_0[u, v] = 0$.
  - **Kombination**: $W_k[u, v] = \max\{W_{k-1}[u, v], W_{k-1}[u, k] \cdot W_{k-1}[k, v]\}$.

Binomialkoeffizienten revisited

Algorithmus DYN-BINOM

EINGABE: $n, k \in \mathbb{N}_0$

1. If $(k > n)$ return 0;
2. For $i \leftarrow 0$ to n
   1. $a[i, 0] = 1$; $a[i, i] = 1$;
3. For $i \leftarrow 1$ to n
   1. For $j \leftarrow 1$ to $i - 1$
      1. $a[i, j] \leftarrow a[i - 1, j - 1] + a[i - 1, j]$;
4. return $a[n, k]$;

AUSGABE: $\binom{n}{k}$

- **Korrektheit:** Zeilenweise Berechnung des Pascal'schen Dreiecks.
- Schritt 1 initialisiert die Grenzen des Pascal'schen Dreiecks.
- Schritt 2 berechnet die i-te Zeile des Pascal'schen Dreiecks.
- **Laufzeit:** $\mathcal{O}(nk)$ für $k \leq n$, d.h. $\mathcal{O}(n^2)$.

Motivation von Optimierungsproblemen

Definition Vollständig geklammertes Matrixprodukt

Sei $M = A_1 \cdot \ldots \cdot A_n$ ein Produkt von Matrizen. M heißt *vollständig geklammertes Matrixprodukt*, falls $n = 1$ oder M ein geklammertes Produkt von zwei vollständig geklammerten Matrixprodukten ist.

Beispiel: $n = 4$

- Es existieren die 5 vollständig geklammerten Matrixprodukte $(((A_1A_2)A_3)A_4), ((A_1(A_2A_3))A_4), (A_1((A_2A_3)A_4)),$ $(A_1(A_2(A_3A_4))), ((A_1A_2)(A_3A_4))$.
- Man kann zeigen, dass die Anzahl der vollständig geklammerten Matrixprodukte exponentiell in n ist.

Kosten einer Lösung

Definition Kosten einer Matrixmultiplikation

Sei $A_1 = (a_{i,j})$ eine (n_1, n_2)-Matrix und A_2 eine $(n_2 \times n_3)$-Matrix. Dann definieren wir die Kosten der Berechnung von $A_1 \cdot A_2$ als $n_1 n_2 n_3$.

Die Kosten eines vollständigen geklammerten Matrixprodukts sind definiert als die Summe der Kosten aller Matrixprodukte.

Begründung:

- Seien $a^{(1)}_{i,j}$, $a^{(2)}_{i,j}$ die Einträge von A_1 bzw. A_2
- Sei $m_{i,j} = \sum_{k=1}^{n_2} a^{(1)}_{i,k} \cdot a^{(2)}_{k,j}$ für $i \in [n_1], j \in [n_3]$.
- Die Berechnung aller $m_{i,j}$ erfordert $n_1 \cdot n_2 \cdot n_3$ Multiplikationen.

Beispiel: $A_1 \in \mathbb{Z}^{10\times 5}, A_2 \in \mathbb{Z}^{5\times 10}$ und $A_3 \in \mathbb{Z}^{10\times 10}$

- $((A_1 A_2)A_3)$ liefert Kosten von $500 + 1000 = 1500$.
- $(A_1(A_2 A_3))$ liefert dagegen nur Kosten von $500 + 500 = 1000$.

Optimierungsprobleme

Definition Optimierungsproblem

Sei Π ein Problem mit Lösungsraum L und $c : L \to \mathbb{R}$ eine Kostenfunktion. Wir bezeichnen Π als *Optimierungsproblem*, falls eine Lösung $\ell \in L$ mit optimalen Kosten $c(\ell)$ gesucht wird.

1. Falls $c(\ell)$ maximiert wird, nennen wir Π ein *Maximierungsproblem*.
2. Falls $c(\ell)$ minimiert wird, nennen wir Π ein *Minimierungsproblem*.

Problem Minimierungsproblem Matrizenklammerung Π_M

Gegeben: $(p_i \times p_{i+1})$-Matrizen für $i \in [n]$.
Gesucht: Vollständige Klammerung mit minimalen Kosten.

Aufsplitten einer optimalen Lösung

- Sei ℓ eine optimale vollständige Klammerung für Π_M.
- Wir verwenden die Notation $A_{1\ldots n}$ für $A_1 \cdot \ldots \cdot A_n$.
- In ℓ werden $A_{1\ldots k}$ und $A_{k+1\ldots n}$ für ein $k \in [n-1]$ berechnet.

Lemma Optimalität der Subprobleme

Sei Π_M ein Matrizenklammerungs-Problem mit $M = A_1 \ldots A_n$ und ℓ eine optimale Lösung, die $A_{1\ldots k}$, $A_{k+1\ldots n}$ berechnet. Dann liefert ℓ für diese Subprobleme optimale Lösungen.

Beweis:

- **Annahme:** Sei ℓ' eine bessere Lösung für $A_{1\ldots k}$.
- Seien $c(\ell(A_{i\ldots j}))$ die Kosten der Lösung ℓ für $A_{i\ldots j}$.
- Berechne in ℓ das Produkt $A_{1\ldots k}$ gemäß ℓ' mit $c(\ell') < c(\ell(A_{1\ldots k}))$.
- Dann gilt $c(\ell') + c(\ell(A_{k+1\ldots n})) < c(\ell(A_{1\ldots k})) + c(\ell(A_{k+1\ldots n})) = c(\ell)$. (Widerspruch zur Minimalität der Lösung ℓ für Π_M)
- Beweis liefert analog die Optimalität von ℓ für $A_{k+1\ldots n}$.

Dynamische Programmierung für Π_M

Lösung mittels Dynamischer Programmierung

- Seien $m[i,j]$ die minimalen Kosten zur Berechnung von $A_{i\ldots j}$.
- **Bottom-up:** $m[i,i] = 0$ für $i = 1,\ldots,n$.
- **Kombination:** Berechnung von $A_{i\ldots j}$ für alle $i < j$.
- Bestimme k mit $i \leq k < j$, das folgende Kosten minimiert:
  - Berechne $A_{i\ldots k}$ und $A_{k+1\ldots j}$ mit Kosten $m[i,k] + m[k+1,j]$.
  - Berechne $A_{i\ldots k} \cdot A_{k+1\ldots j}$ mit Kosten $p_i p_{k+1} p_{j+1}$
- D.h. wähle $m[i,j] = \min_{1\leq k<j}\{m[i,k] + m[k+1,j] + p_i p_{k+1} p_{j+1}\}$.
- Die Gesamtkosten für $M = A_1 \cdot \ldots \cdot A_n$ sind $m[1,n]$.

Algorithmus DYN-MATRIXKLAMMERUNG

Algorithmus DYN-MATRIXKLAMMERUNG

EINGABE: $p_1, \ldots, p_{n+1} \in \mathbb{N}$

1. For $i \leftarrow 1$ to n
   1. $m[i,i] \leftarrow 0$;
2. For $\ell \leftarrow 2$ to n //ℓ ist die Länge des Intervalls $[i,j]$
   1. For $i \leftarrow 1$ to $n-(\ell-1)$
      1. $j \leftarrow i+\ell-1$;
      2. $m[i,j] \leftarrow \min_{i \le k < j}\{m[i,k]+m[k+1,j]+p_i p_{k+1} p_{j+1}\}$.
      3. $s[i,j] \leftarrow k$, an dem das Minimum angenommen wird.

AUSGABE: Arrays m und s

- **Korrektheit:** Die Intervalllängen ℓ wachsen stetig.
- Dadurch werden in Schritt 2.1.2 nur bekannte Werte verwendet.
- Bei Terminierung stehen in $m[1,n]$ die minimalen Kosten.
- Übung: s liefert die optimale Klammerung.
- **Laufzeit** für die Schleifen über ℓ, i, k : $\mathcal{O}(n^3)$.

Beispiel für ein Maximierungsproblem

Problem Maximierungsproblem Rucksack Π_R

Gegeben: n Gegenstände mit Gewichten $w_i \in \mathbb{N}$ und Profiten $p_i \in \mathbb{N}$ für $i \in [n]$. Kapazitätsschranke B

Gesucht: $\max_{J \subseteq [n]} \{\sum_{j \in J} p_j \mid \sum_{j \in J} w_j \leq B\}$, bzw. dasjenige J, an dem das Maximum angenommen wird.

Anmerkungen:

- Lösung von Rucksack mit Dynamischer Programmierung möglich.
- Liefert einen Algorithmus mit Laufzeit $\mathcal{O}(n \sum_{i \in [n]} p_i)$.
- Man beachte: Eingabelänge der p_i ist nur $\Theta(\sum_{i \in [n]} \log p_i)$.
- D.h. der Algorithmus ist exponentiell in der Eingabelänge.
- Π_R gehört zu den algorithmisch schweren Problemen (s. DiMa II).