8. Das Halteproblem

Definition 122

Unter dem speziellen Halteproblem H_s versteht man die folgende Sprache:

$$H_s = \{w \in \{0,1\}^*;\ M_w \text{ angesetzt auf } w \text{ hält}\}$$

Hierbei ist $(M_\epsilon, M_0, M_1, \ldots)$ eine berechenbare Auflistung der Turing-Maschinen.

Wir definieren weiter

Definition 123

$$L_d = \{w \in \Sigma^*;\ M_w \text{ akzeptiert } w \text{ nicht}\}$$

Satz 124

L_d ist nicht rekursiv aufzählbar.

Beweis:

Wäre L_d r.a., dann gäbe es ein w, so dass $L_d = L(M_w)$.
Dann gilt:

$$\begin{aligned} M_w \text{ akzeptiert } w \text{ nicht} \quad &\Leftrightarrow \quad w \in L_d \\ &\Leftrightarrow \quad w \in L(M_w) \\ &\Leftrightarrow \quad M_w \text{ akzeptiert } w \end{aligned}$$

$\Longrightarrow$ Widerspruch!

Korollar 125

L_d ist nicht entscheidbar.

Satz 126

H_s ist nicht entscheidbar.

Beweis:

Angenommen, es gäbe eine Turing-Maschine M, die H_s entscheidet. Indem man i.W. die Antworten von M umdreht, erhält man eine TM, die L_d entscheidet. Widerspruch!

9. Unentscheidbarkeit

Definition 127

Unter dem (allgemeinen) Halteproblem H versteht man die Sprache

$$H = \{\langle x, w\rangle \in \{0,1\}^*;\ M_x \text{ angesetzt auf } w \text{ hält}\}$$

Satz 128

Das Halteproblem H ist nicht entscheidbar.

Beweis:

Eine TM, die H entscheidet, könnten wir benutzen, um eine TM zu konstruieren, die H_s entscheidet.

Bemerkung: H und H_s sind beide rekursiv aufzählbar!

Definition 129

Seien A, $B \subseteq \Sigma^*$. Dann heißt A (effektiv) reduzierbar auf B gdw $\exists f : \Sigma^* \to \Sigma^*$, f total und berechenbar mit

$$(\forall w \in \Sigma^*)[w \in A \Leftrightarrow f(w) \in B]\,.$$

Wir schreiben auch

$$A \hookrightarrow_f B \text{ bzw. } A \hookrightarrow B\,.$$

bzw. manchmal

$$A \leq B \text{ oder auch } A \preceq_f B\,.$$

Ist A mittels f auf B reduzierbar, so gilt insbesondere

$$f(A) \subseteq B \text{ und } f(\bar{A}) \subseteq \bar{B}\,.$$

Satz 130

Sei $A \hookrightarrow_f B$.

(i) B *rekursiv* $\Rightarrow$ A *rekursiv.*

(ii) B *rekursiv aufzählbar* $\Rightarrow$ A *rekursiv aufzählbar.*

Beweis:

(i) $\chi_A = \chi_B \circ f$.

(ii) $\chi'_A = \chi'_B \circ f$.

Definition 131

Das Halteproblem auf leerem Band H_0 ist

$$H_0 = \{w \in \{0,1\}^*;\ M_w \text{ hält auf leerem Band}\}\,.$$

Satz 132

H_0 ist unentscheidbar (nicht rekursiv).

Beweis:

Betrachte die Abbildung f, die definiert ist durch:

$$\{0,1\}^* \ni w \mapsto f(w),$$

$f(w)$ ist die Gödelnummer einer TM, die, auf leerem Band angesetzt, zunächst $c_2(w)$ auf das Band schreibt und sich dann wie $M_{c_1(w)}$ (angesetzt auf $c_2(w)$) verhält. Falls das Band nicht leer ist, ist es unerheblich, wie sich $M_{f(w)}$ verhält.

f ist total und berechenbar.

Es gilt: $w \in H \Leftrightarrow M_{c_1(w)}$ angesetzt auf $c_2(w)$ hält
$\Leftrightarrow M_{f(w)}$ hält auf leerem Band
$\Leftrightarrow f(w) \in H_0$

also $H \hookrightarrow_f H_0$ und damit H_0 unentscheidbar.

Bemerkung

Es gibt also keine allgemeine algorithmische Methode, um zu entscheiden, ob ein Programm anhält.

Kapitel V Algorithmen und Datenstrukturen

1. Analyse von Algorithmen

Wir wollen die Ressourcen bestimmen, die, in Abhängigkeit von der Eingabe, ein Algorithmus benötigt, z.B.

1. Laufzeit
2. Speicherplatz
3. Anzahl Prozessoren
4. Programmlänge
5. Tinte
6. ...

Beispiel 133

Prozedur für Fakultätsfunktion

func $fak(n)$
  $m := 1$
  for $i := 2$ **to** n **do**
    $m := m * i$
  do
  return (m)

Diese Prozedur benötigt $O(n)$ Schritte bzw. arithmetische Operationen.

Jedoch: die Länge der Ausgabe ist etwa

$$\lceil \log_2 n! \rceil = \Omega(n \log n) \text{ Bits.}$$

Bemerkung:

Um die Zahl $n \in \mathbb{N}_0$ in Binärdarstellung hinzuschreiben, benötigt man

$$\ell(n) := \begin{cases} 1 & \text{für } n = 0 \\ 1 + \lfloor \log_2(n) \rfloor & \text{sonst} \end{cases}$$
$$= \begin{cases} 1 & \text{für } n = 0 \\ \lceil \log_2(n+1) \rceil & \text{sonst} \end{cases}$$

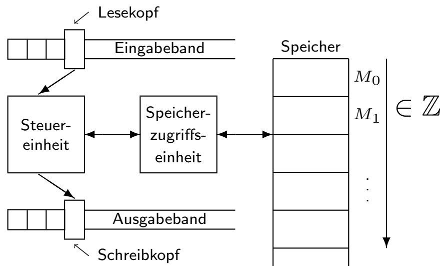
Um die Notation zu vereinfachen, vereinbaren wir im Zusammenhang mit Komplexitätsabschätzungen

$$\log(0) := 0\,.$$

1.1 Referenzmaschine

Wir wählen als Referenzmaschine die Registermaschine (engl. random access machine, RAM) oder auch WHILE-Maschine, also eine Maschine, die WHILE-Programme verarbeiten kann, erweitert durch

- IF ... THEN ... ELSE ... FI
- Multiplikation und Division
- indirekte Adressierung
- arithmetische Operationen wie $\sqrt{n}$, $\sin n$,...



Registermaschine

1.2 Wachstumsverhalten von Funktionen

f, g seien Funktionen von $\mathbb{N}_0$ nach $\mathbb{R}_+$.

- $g = \mathcal{O}(f)$ [auch: $g(n) = \mathcal{O}(f(n))$ oder $g \in \mathcal{O}(f)$] gdw.

$$(\exists c > 0 \exists n_0 \in \mathbb{N}_0 \forall n \geq n_0)[g(n) \leq c \cdot f(n)]$$

- $g = \Omega(f)$ [auch: $g(n) = \Omega(f(n)$ oder $g \in \Omega(f)$] gdw.

$$(\exists c > 0 \exists n_0 \in \mathbb{N}_0 \forall n \geq n_0)[g(n) \geq c \cdot f(n)]$$

- $g = \Theta(f)$ gdw. $g = \mathcal{O}(f)$ und $g = \Omega(f)$

f, g seien Funktionen von $\mathbb{N}_0$ nach $\mathbb{R}_+$.

- $g = o(f)$ gdw.

$$(\forall c > 0 \exists n_0 \in \mathbb{N}_0 \forall n \geq n_0)[g(n) \leq c \cdot f(n)]$$

- $g = \omega(f)$ gdw.

$$(\forall c > 0 \exists n_0 \in \mathbb{N}_0 \forall n \geq n_0)[g(n) \geq c \cdot f(n)]$$

- $g = \Omega_\infty(f)$ gdw.

$$(\exists c > 0)[g(n) \geq c \cdot f(n) \text{ für unendlich viele } n]$$

- $g = \omega_\infty(f)$ gdw.

$$(\forall c > 0)[g(n) \geq c \cdot f(n) \text{ für unendlich viele } n]$$

Beispiel 134

- n^3 ist <u>nicht</u> $\mathcal{O}\left(\frac{n^3}{\log n}\right)$.
- $n^3 + n^2$ ist <u>nicht</u> $\omega(n^3)$.
- $100n^3$ ist <u>nicht</u> $\omega(n^3)$.

Bemerkung:

Die Groß-O-Notation wurde von D. E. Knuth in der Algorithmenanalyse eingeführt, siehe z.B.

Donald E. Knuth:
Big omicron and big omega and big theta.
SIGACT News **8**(2), pp. 18–24, ACM SIGACT, 1976

Sie wurde ursprünglich von Paul Bachmann (1837–1920) entwickelt und von Edmund Landau (1877–1938) in seinen Arbeiten verbreitet.

1.2 Zeit- und Platzkomplexität

Beim Zeitbedarf zählt das uniforme Kostenmodell die Anzahl der von der Registermaschine durchgeführten Elementarschritte, beim Platzbedarf die Anzahl der benutzten Speicherzellen.

Das logarithmische Kostenmodell zählt für den Zeitbedarf eines jeden Elementarschrittes

$$\ell(\text{größter beteiligter Operand}),$$

beim Platzbedarf

$$\sum_x \ell(\text{größter in } x \text{ gespeicherter Wert})\,,$$

also die maximale Anzahl der von allen Variablen x benötigten Speicherbits.

Beispiel 135

Wir betrachten die Prozedur

func $dbexp(n)$
  $m := 2$
  for $i := 1$ **to** n **do**
    $m := m^2$
  do
  return (m)

Die Komplexität von $dbexp$, die $n \mapsto 2^{2^n}$ berechnet, ergibt sich bei Eingabe n zu

	Zeit	Platz
uniform	$\Theta(n)$	$\Theta(1)$
logarithmisch	$\Theta(2^n)$	$\Theta(2^n)$

Bemerkung:
Das (einfachere) uniforme Kostenmodell sollte also nur verwendet werden, wenn alle vom Algorithmus berechneten Werte gegenüber den Werten in der Eingabe nicht zu sehr wachsen, also z.B. nur polynomiell.

1.3 Worst Case-Analyse

Sei A ein Algorithmus. Dann sei

$$T_A(x) := \text{Laufzeit von } A \text{ bei Eingabe } x\,.$$

Diese Funktion ist i.A. zu aufwändig und zu detailliert. Stattdessen:

$$T_A(n) := \max_{|x|=n} T_A(x) \quad (= \text{maximale Laufzeit bei Eingabelänge } n)$$

1.4 Average Case-Analyse

Oft erscheint die Worst Case-Analyse als zu pessimistisch. Dann:

$$T_A^{\mathsf{avg}}(n) = \frac{\sum_{x;\ |x|=n} T_A(x)}{|\{x;\ |x| = n\}|}$$

oder allgemeiner

$$\begin{aligned} T_A^{\mathsf{avg}}(n) &= \sum T_A(x) \cdot \mathsf{Pr}\left\{x \mid |x| = n\right\} \\ &= \mathbf{E}_{|x|=n}\left[T_A(x)\right] , \end{aligned}$$

wobei eine (im Allgemeinen beliebige) Wahrscheinlichkeitsverteilung zugrunde liegt.

Bemerkung:

Wir werden Laufzeiten $T_A(n)$ meist nur bis auf einen multiplikativen Faktor genau berechnen, d.h. das genaue Referenzmodell, Fragen der Implementierung, usw. spielen dabei eine eher untergeordnete Rolle.

2. Sortierverfahren

Unter einem Sortierverfahren versteht man ein algorithmisches Verfahren, das als Eingabe eine Folge $a_1, \ldots, a_n$ von n Schlüsseln $\in \Sigma^*$ erhält und als Ausgabe eine auf- oder absteigend sortierte Folge dieser Elemente liefert. Im Folgenden werden wir im Normalfall davon ausgehen, dass die Elemente aufsteigend sortiert werden sollen. Zur Vereinfachung nehmen wir im Normalfall auch an, dass alle Schlüssel paarweise verschieden sind.

Für die betrachteten Sortierverfahren ist natürlich die Anzahl der Schlüsselvergleiche eine untere Schranke für die Laufzeit, und oft ist letztere von der gleichen Größenordnung, d.h.

$$\text{Laufzeit} = O(\text{Anzahl der Schlüsselvergleiche}) .$$

2.1 Selection-Sort

Sei eine Folge $a_1, \ldots, a_n$ von Schlüsseln gegeben, indem a_i im Element i eines Feldes $A[1..n]$ abgespeichert ist.

Algorithmus SelectionSort

```
for i := n downto 2 do
    m := Index des maximalen Schlüssels in A[1..i]
    vertausche A[i] und A[m]
od
```

Nach Beendigung von SelectionSort ist das Feld A aufsteigend sortiert.

Satz 136

SELECTIONSORT *benötigt zum Sortieren von* n *Elementen genau* $\binom{n}{2}$ *Vergleiche.*

Beweis:

Die Anzahl der Vergleiche (zwischen Schlüsseln bzw. Elementen des Feldes A) zur Bestimmung des maximalen Schlüssels in $A[1..i]$ ist $i-1$.

Damit ergibt sich die Laufzeit von SELECTIONSORT zu

$$T_{\text{SELECTIONSORT}} = \sum_{i=2}^{n}(i-1) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \binom{n}{2}.$$

2.2 Insertion-Sort

Sei eine Folge $a_1, \ldots, a_n$ von Schlüsseln gegeben, indem a_i im Element i eines Feldes $A[1..n]$ abgespeichert ist.

Algorithmus InsertionSort

```
for i := 2 to n do
    m := Rang (∈ [1..i]) von A[i] in {A[1], ..., A[i − 1]}
    a := A[i]
    schiebe A[m..i − 1] um eine Position nach rechts
    A[m] := a
od
```

Nach Beendigung von InsertionSort ist das Feld A aufsteigend sortiert.

Der Rang von $A[i]$ in $\{A[1], \ldots, A[i-1]\}$ kann trivial mit $i-1$ Vergleichen bestimmt werden. Damit ergibt sich

Satz 137

InsertionSort benötigt zum Sortieren von n Elementen maximal $\binom{n}{2}$ Vergleiche.

Beweis:

Übungsaufgabe!

Die Rangbestimmung kann durch binäre Suche verbessert werden. Dabei benötigen wir, um den Rang eines Elementes in einer k-elementigen Menge zu bestimmen, höchstens

$$\lceil \log_2(k+1) \rceil$$

Vergleiche, wie man durch Induktion leicht sieht.

Satz 138

InsertionSort *mit binärer Suche für das Einsortieren benötigt zum Sortieren von* n *Elementen maximal*

$$n \lceil \operatorname{ld} n \rceil$$

Vergleiche.

Beweis:

Die Abschätzung ergibt sich durch einfaches Einsetzen.

Achtung: Die Laufzeit von InsertionSort ist dennoch auch bei Verwendung von binärer Suche beim Einsortieren im schlechtesten Fall $\Omega(n^2)$, wegen der notwendigen Verschiebung der Feldelemente.

Verwendet man statt des Feldes eine doppelt verkettete Liste, so wird zwar das Einsortieren vereinfacht, es kann jedoch die binäre Suche nicht mehr effizient implementiert werden.