
Grundlagen der theoretischen Informatik

Kurt Sieber

Fachbereich Mathematik/Theoretische Informatik
Universität Siegen

Vorlesung vom 13.12.2004 (Stand: 19.12.2004)

Berechenbarkeit

Teil II: Berechenbarkeit und Komplexität

Zunächst: **Berechenbarkeit**

Zentrale Fragestellungen:

Welche Probleme lassen sich mit Hilfe von Algorithmen (Computer-Programmen) lösen bzw. nicht lösen?

Welche Funktionen lassen sich mit ihnen berechnen bzw. nicht berechnen?

In der Berechenbarkeitstheorie geht es um die *prinzipielle* Berechenbarkeit, d.h. Zeit- und Speicherplatzbedarf werden ignoriert.

Betrachtungen über Zeitaufwand und Platzbedarf sind Thema der Komplexitätstheorie.

Interessant sind in der Berechenbarkeitstheorie deshalb vor allem die *negativen* Ergebnisse.

Berechenbarkeit

Wir hatten schon **Beispiele:**

1. Das Äquivalenzproblem für DEAs ist algorithmisch lösbar (entscheidbar), d.h. es gibt einen Algorithmus, der als Eingabe zwei DEAs A_1 und A_2 erhält, als Ausgabe die korrekte Antwort auf die Frage

$$L(A_1) = L(A_2)?$$

liefert.

2. Das Äquivalenzproblem für KFGs ist *nicht* algorithmisch lösbar (unentscheidbar), d.h. es gibt *keinen* Algorithmus, der als Eingabe zwei KFGs G_1 und G_2 erhält, als Ausgabe die korrekte Antwort auf die Frage

$$L(G_1) = L(G_2)?$$

liefert.

Berechenbarkeit

Zum Beweis der Entscheidbarkeit bzw. Berechenbarkeit eines Problems braucht man 'nur' einen Algorithmus anzugeben.

Aber wie beweist man, dass ein Problem *nicht* entscheidbar bzw. eine Funktion *nicht* berechenbar ist?

Gibt es überhaupt solche Probleme bzw. Funktionen?
(Wir haben es noch nicht bewiesen.)

Dazu muss man zunächst die Bedeutung der Begriffe *Berechenbarkeit*, *Entscheidbarkeit*, ... genauer festlegen

Berechenbarkeit

Intuition:

Eine Funktion heißt berechenbar, wenn es einen Algorithmus gibt, der sie berechnet.

Folgende Fragen sind zu klären:

1. Über welche Art von Funktionen sprechen wir hier?

Partielle oder totale Funktionen?

Funktionen auf Zahlen, Wörtern, ... ?

(d.h. was ist als Eingabe bzw. Ausgabe für die Algorithmen zugelassen?)

2. Was versteht man unter einem Algorithmus?

3. Was versteht man unter der von einem Algorithmus berechneten Funktion?
-

Berechenbarkeit

Antworten:

1. Wir benutzen hier den üblichen Ansatz, d.h. wir lassen *partielle* Funktionen zu.

Warnung: In einem Teil der Literatur beschränkt man sich auf totale Funktionen. Diesen Ansatz benutzen wir nicht, weil er an manchen Stellen der Theorie zu einer umständlichen Ausdrucksweise führt.

Wir betrachten zunächst Funktionen auf natürlichen Zahlen, d.h. Funktionen $f : \mathbb{N}^k \hookrightarrow \mathbb{N}$. Später werden wir auch Funktionen auf Wörtern betrachten, d.h. Funktionen $f : (\Sigma^*)^k \hookrightarrow \Sigma^*$ über beliebigem Alphabet Σ .

2. Unter einem Algorithmus verstehen wir (zunächst) ein Programm in einer 'hinreichend mächtigen' Programmiersprache.
3. Die von einem Algorithmus berechnete Funktion $f : \mathbb{N}^k \hookrightarrow \mathbb{N}$ wird gleich definiert.

Berechenbarkeit

Bemerkungen:

Es kommt nicht darauf an, ob man Funktionen auf Zahlen oder auf Wörtern benutzt, denn:

Einerseits lassen sich Zahlen als Wörter darstellen (dezimal, binär, unär).

Andererseits lassen sich Wörter als Zahlen codieren:

Die Wörter über einem Alphabet Σ lassen sich 'durchnummerieren', z.B. indem man zuerst die Wörter der Länge 0, dann die der Länge 1, \dots , jeweils in alphabetischer Reihenfolge aufzählt (vgl. Übung 4, Aufgabe 1 a).

So erhält man eine Folge von Wörtern w_0, w_1, \dots , in der *alle* Wörter aus Σ^* enthalten sind. Die Zahl n kann man dann als *Codierung* des Wortes w_n betrachten.

Berechenbarkeit

Wie passen unsere früheren Beispiele in diesen Rahmen (Algorithmen, die als Eingabe DEAs oder KFGs erhalten)?

Ein DEA oder eine KFG besitzt eine *endliche Darstellung*, lässt sich also als Wort über einem geeigneten Alphabet auffassen.

Zum Vergleich:

Algorithmen, die eine (beliebige) Sprache als Eingabe erhalten oder die auf reellen Zahlen arbeiten passen *nicht* in unseren Rahmen, weil Sprachen oder reelle Zahlen keine endliche Darstellung besitzen. (Sie können keine endliche Darstellung besitzen, denn die Menge der Sprachen über einem Alphabet Σ oder die Menge der reellen Zahlen sind *überabzählbar*, vgl. Übung 4, Aufgabe 1 b bzw. Übung 1, Aufgabe 1).

Berechenbarkeit

Definition 3.1 Sei $k \geq 0$, $f : \mathbb{N}^k \hookrightarrow \mathbb{N}$ eine partielle Funktion und π ein Algorithmus, d.h. ein Programm über einer geeigneten Programmiersprache.

1. π **berechnet** die Funktion f , wenn für alle $(n_1, \dots, n_k) \in \mathbb{N}^k$ gilt:
 π hält genau dann für die Eingabe (n_1, \dots, n_k) , wenn $f(n_1, \dots, n_k)$ definiert ist, und liefert in diesem Fall die Ausgabe $f(n_1, \dots, n_k)$.
2. f heißt **berechenbar**, wenn es ein Programm π gibt, das f berechnet.

Die 'geeignete(n) Programmiersprache(n)' definieren wir später. Dann wird (jeweils) genau festgelegt, wie ein Programm π seine Eingabe erhält, und wo es seine Ausgabe abliefert. Manche dieser Programmiersprachen arbeiten auf Wörtern statt auf Zahlen. Dann tritt Σ^* (mit beliebigem Alphabet Σ) an die Stelle von \mathbb{N} .

Berechenbarkeit

Beispiele:

Beispiele für berechenbare Funktionen sind leicht zu finden, etwa

1. Die Funktion

$$\begin{aligned} \text{exp} : \mathbb{N}^2 &\rightarrow \mathbb{N} \\ (m, n) &\mapsto m^n \end{aligned}$$

ist berechenbar (durch irgendein Programm zum Potenzieren).

2. Die *leere* Funktion $f : \mathbb{N}^k \hookrightarrow \mathbb{N}$, die für alle $(n_1, \dots, n_k) \in \mathbb{N}^k$ undefiniert ist, ist berechenbar durch ein Programm, das für *keine* Eingabe (n_1, \dots, n_k) hält.

3. Jede konstante Funktion

$$\begin{aligned} \text{const}_m : \mathbb{N}^k &\rightarrow \mathbb{N} \\ (n_1, \dots, n_k) &\mapsto m \end{aligned}$$

ist berechenbar durch ein Programm, das unabhängig von der Eingabe immer hält und die Zahl m zurückliefert.

Berechenbarkeit

Ein konkretes Gegenbeispiel (mit Beweis) können wir noch nicht angeben, aber wir können schon zeigen, dass es welche gibt.

Satz 3.2 *Es gibt Funktionen (partielle oder totale), die nicht berechenbar sind.*

Beweis:

Programme π sind Wörter über einem geeigneten Alphabet (dem Alphabet der Programmiersprache).

Also ist die Menge aller Programme abzählbar, und damit auch (für jedes $k \geq 0$) die Menge aller berechenbaren Funktionen.

Andererseits gibt es (für jedes $k \geq 0$) *überabzählbar* viele totale Funktionen $f : \mathbb{N}^k \rightarrow \mathbb{N}$ (Beweis durch Diagonalisierung), also sind sogar die “meisten” Funktionen $f : \mathbb{N}^k \rightarrow \mathbb{N}$ (nämlich überabzählbar viele) *nicht* berechenbar. \square

Berechenbarkeit

Satz 3.2 liefert nur die *Existenz* von Funktionen, die nicht berechenbar sind. Unser nächstes Ziel ist es, *konkrete* (interessante) Funktionen dieser Art anzugeben. Dazu benötigen wir einige Vorüberlegungen.

Da die Programme π unserer ‘hinreichend mächtigen’ Programmiersprache Wörter über einem geeigneten Alphabet Σ sind, können wir sie ebenfalls ‘durchnummerieren’, d.h. wir erhalten eine Folge von Programmen $\pi_0, \pi_1, \pi_2, \dots$, in der *alle* Programme enthalten sind.

Die Zahl i nennt man die *Gödelnummer* des Programms π_i . Sie lässt sich als Codierung des Programms auffassen.

Mit Hilfe dieser Codierung lässt sich ein Programm als Eingabe für ein anderes Programm verwenden, indem man einfach die *Gödelnummer* des einen Programms in das andere eingibt.

Berechenbarkeit

Hätten wir Programme betrachtet, die auf Wörtern arbeiten, dann hätten wir es an dieser Stelle einfacher: Dann könnten wir nämlich den *Text* des einen Programms in das andere Programm eingeben. Insofern kann man die Codierung eines Programms durch seine Gödelnummer als einen technischen Trick betrachten, der nötig ist, weil unsere Programme auf Zahlen und nicht auf Wörtern arbeiten.

Auf jeden Fall können wir jetzt Programme betrachten, die mit anderen Programmen 'rechnen' oder die Fragen über andere Programme beantworten. (Beispiele solcher Programme aus der Praxis sind Compiler oder Interpreter.)

Insbesondere können wir jedes Programm 'auf sich selbst' anwenden, d.h. auf seine eigene Codierung.

Diese Möglichkeit werden wir ausnutzen, um eine nicht entscheidbare Menge (und damit eine nicht berechenbare Funktion) zu finden.

Berechenbarkeit

Definition 3.3

1. Eine Menge $B \subseteq \mathbb{N}^k$ heißt **entscheidbar**, wenn die **charakteristische Funktion**

$$c_B : \mathbb{N}^k \rightarrow \mathbb{N}$$
$$c_B(n_1, \dots, n_k) = \begin{cases} 1 & \text{falls } (n_1, \dots, n_k) \in B \\ 0 & \text{sonst} \end{cases}$$

berechenbar ist.

2. Eine Menge $B \subseteq \mathbb{N}^k$ heißt **semi-entscheidbar** oder **akzeptierbar**, wenn die **Akzeptorfunktion**

$$a_B : \mathbb{N}^k \hookrightarrow \mathbb{N}$$
$$a_B(n_1, \dots, n_k) = \begin{cases} 1 & \text{falls } (n_1, \dots, n_k) \in B \\ \text{undefiniert} & \text{sonst} \end{cases}$$

berechenbar ist.

Berechenbarkeit

Mit anderen Worten:

1. B ist entscheidbar, wenn es einen *Entscheidungsalgorithmus* für B gibt, d.h. einen Algorithmus, der für die Elemente aus B die Ausgabe 1 liefert und für alle übrigen Elemente die Ausgabe 0.
2. B ist semi-entscheidbar, wenn es einen *Semi-Entscheidungsalgorithmus* für B gibt, d.h. einen Algorithmus, der für die Elemente aus B die Ausgabe 1 liefert und der für alle übrigen Elemente nicht hält.

Beispiele:

Beispiele entscheidbarer Mengen sind wieder leicht zu finden, etwa:

1. Die Menge $G \subseteq \mathbb{N}$ der geraden Zahlen ist entscheidbar.
 2. Die Menge $P \subseteq \mathbb{N}$ der Primzahlen ist entscheidbar. Ein einfacher (aber ineffizienter) Entscheidungsalgorithmus besteht darin, zu überprüfen, ob n durch eine Zahl m mit $2 \leq m < n$ teilbar ist.
-

Berechenbarkeit

Damit haben wir auch Beispiele für semi-entscheidbare Mengen, denn es gilt

Satz 3.4 *Jede entscheidbare Menge B ist semi-entscheidbar.*

Beweis:

Sei B entscheidbar, d.h. es existiert ein Programm π , das die charakteristische Funktion c_B von B berechnet.

Ein (mögliches) Programm π' , das die Akzeptorfunktion a_B von B berechnet, arbeitet wie folgt:

Zuerst wird π ausgeführt.

Wenn π mit Ausgabe 0 hält, dann geht π' in eine Endlosschleife.

Wenn π mit Ausgabe 1 hält, dann gibt π' ebenfalls die 1 aus. \square