St. Pölten University of Applied Sciences

/informatik & security

/fh/// st. pölten

Entscheidbarkeit und Berechenbarkeit

Theoretische Informatik

Dipl.-Ing. Hubert Schölnast, BSc
22. April 2023

Fachhochschule St. Pölten, Campus-Platz 1, 3100 St. Pölten, T: +43 (2742) 313 228, F: +43 (2742) 313 228-339, E: csc@fhstp.ac.at, I: www.fhstp.ac.at

Inhaltsverzeichnis

1 Definitionen

1.1 Das Wortproblem

Wir haben eine Grammatik $G = \{V, \Sigma, P, S\}$ und eine Sprache $\mathcal{L}(G)$, die durch diese Grammatik definiert wird.

Σ^* ist die kleenesche Hülle von Σ, was bedeutet, dass Σ^* die Menge aller möglichen Wörter ist, die mit Zeichen aus dem Alphabet Σ geschrieben werden können. Und es ist klar, dass $\mathcal{L}(G)$ eine Teilmenge von Σ^* ist:

$$\mathcal{L}(G) \subseteq \Sigma^*$$

Außerdem haben wir ein Wort w gegeben, d.h. ein Element von Σ^*:

$$w \in \Sigma^*$$

Nun wollen wir wissen, ob dieses bestimmte Wort w zu der durch die Grammatik definierten Sprache gehört oder nicht. Also fragen wir

$$\text{Ist } w \in \mathcal{L}(G)?$$

Diese Frage ist das Wortproblem.

1.2 Charakteristische Funktion

Das Wortproblem kann als Funktion definiert werden, und weil es für jede Sprache eine individuelle Funktion gibt und weil die Funktion einer Sprache für diese Sprache charakteristisch ist, wird sie »charakteristische Funktion« genannt:

$$cf: \Sigma^* \to \{0, 1\}$$

oder

$$cf: \Sigma^* \to \{\text{false}, \text{true}\}$$

- Der Wert von $cf(w)$ ist 1 (oder true), wenn $w \in \mathcal{L}(G)$
- Der Wert von $cf(w)$ ist 0 (oder false), wenn $w \notin \mathcal{L}(G)$

1.3 Problem

Ein Problem im Kontext der theoretischen Informatik, ist nicht etwas das Schwierigkeiten bereitet, sondern eine Aufgabenstellung, oder eine Fragestellung, also etwas, das nach einer Antwort oder Lösung verlangt.

Wenn man versucht festzustellen, ob ein bestimmtes Wort zu einer bestimmten Sprache gehört oder nicht, ist identisch mit dem Versuch festzustellen, ob die charakteristische Funktion dieser Sprache für das vorgegebene Wort den Wert 1 oder 0 hat. Auch die

Umkehrung gilt: Wenn eine Funktion vorliegt, die die Funktionswerte 0 und 1 hat, und wenn man einen konkreten Eingabewert für diese Funktion hat, und herausfinden will, ob der Funktionswert 0 oder 1 ist, dann ist das eine Fragestellung, also ein Problem, und für dieses Problem gibt es eine Sprache. Jeder spezifische Eingabewert für die Funktion ist ein Wort, von dem man wissen will, ob es zur Sprache gehört oder nicht. Somit ist der Begriff „Problem" eng mit dem Begriff „Sprache" verwandt. Jede Sprache definiert ihr eigenes Problem. Und für jedes Problem gibt es eine Sprache, die diesem Problem entspricht.

In der Informatik können wir also die Begriffe „Problem" und „Sprache" synonym verwenden.

Und das bedeutet: Jede Sprache ist ein Problem und jedes Problem ist eine Sprache und jedes Problem ist eine Teilmenge des Kleene-Abschlusses eines Alphabets Σ.

$$P = \mathcal{L}(G)$$
$$P \subseteq \Sigma^*$$

1.4 Berechenbarkeit

- Intuitive Definition:
  Eine Funktion heißt intuitiv berechenbar, wenn es einen Algorithmus gibt, der diese Funktion berechnet.
- Formale Definition:
  Eine Funktion heißt Turing-berechenbar, wenn es eine Turing-Maschine gibt, die die Funktion berechnet, wenn ihr die Parameter der Funktion als Eingabe bereitgestellt werden, das Ergebnis der Funktion auf ihr Band schreibt und anhält.

1.4.1 Church-Turing-These

Jede intuitiv berechenbare Funktion ist Turing-berechenbar und umgekehrt.

Andere Definitionen basieren auf Prädikatenlogik, λ-Kalkül, einfachen Programmiersprachen und anderen, aber sie sind alle äquivalent zu der Definition, die Turing-Maschinen verwendet.

1.5 Entscheidbarkeit

Ein Problem heißt entscheidbar, wenn die charakteristische Funktion dieses Problems Turing-berechenbar ist. Das heißt, wenn es eine Turing-Maschine gibt, die für jedes Wort entscheidet, ob es zur Sprache gehört oder nicht, und die dann anhält. Ein Problem, bei dem das nicht der Fall ist, ist nicht entscheidbar.

1.5.1 entscheidbare Sprachen, entscheidbare Probleme

Eine Sprache ist entscheidbar, wenn ihre charakteristische Funktion berechenbar ist.

Entscheidbare Sprachen werden auch μ-rekursiv berechenbare Sprachen oder einfach nur "rekursive Sprachen" genannt.

Eine Sprache ist entscheidbar, wenn es eine Turingmaschine gibt, die bei jeder Wortform, also bei jedem Element von Σ^* anhält. Je nachdem, ob die Maschine dann in einem akzeptierenden Zustand anhält, oder ob sie in einem nichtakzeptierenden Zustand anhält, wird das Wort akzeptiert (dann gehört es zur Sprache), oder es wird nicht akzeptiert (denn gehört es nicht zur Sprache).

Mit anderen Worten:

Ein Problem ist entscheidbar, wenn es einen Algorithmus gibt, der bei jeder Instanz des Problems anhält. Er antwortet dann entweder mit „ja“ oder „nein“.

1.5.2 rekursiv aufzählbare (semientscheidbare) Sprachen und Probleme

Es gibt Sprachen, bei denen die Turingmaschine nur anhält, wenn das Wort (der Eingabestring) zur Sprache gehört. Wenn die Eingabezeichenfolge nicht zur Sprache gehört, gibt es keine Garantie dafür, dass die Turing-Maschine anhält.

Diese Sprachen werden rekursiv aufzählbar genannt. Andere Bezeichnungen sind: »teilweise entscheidbar«, »halb entscheidbar« und »semientscheidbar«.

Eine Sprache ist rekursiv aufzählbar, wenn es eine Turing-Maschine gibt, die bei allen Wörtern anhält, die zu der Sprache gehören. In diesem Fall wird das Wort akzeptiert. Bei Wörtern, die nicht zur Sprache gehören, hält die Turing-Maschine entweder an und lehnt das Wort ab, oder sie läuft ewig weiter.

Ein Problem ist semientscheidbar, wenn es einen Algorithmus gibt, der bei allen Instanzen des Problems anhält, wenn seine Antwort "Ja" ist. Aber dort, wo die Antwort "nein" sein sollte, antwortet er entweder mit "nein" oder läuft ewig.

1.5.3 unentscheidbare Sprachen und Probleme

Es gibt Sprachen, für die es keine Turingmaschine gibt, die entscheiden kann, ob ein gegebener Eingabestring zu der Sprache gehört oder nicht. Sie werden bald sehen, warum das so sein muss.

Wenn es unentscheidbare Sprachen gibt, muss es auch unentscheidbare Funktionen geben (nämlich die charakteristischen Funktionen dieser Sprachen.)

Eine Sprache ist unentscheidbar, wenn jede Turing-Maschine, die in der Lage ist, die Sprache zu verarbeiten, bei einigen Wörtern, die zur Sprache gehören, ewig läuft, und die das auch bei einigen Wörtern tut, die nicht zur Sprache gehören. (In einigen Fällen kann die Turing-Maschine auch anhalten und das Wort akzeptieren oder ablehnen.)

Ein Problem ist unentscheidbar, wenn jeder Algorithmus, der das Problem verarbeiten kann, bei einigen Instanzen des Problems niemals zu einem Ende kommt, unabhängig davon, ob die Antwort "Ja" oder "Nein" lauten sollte. (In einigen Fällen kann er auch anhalten und dann die richtige Antwort geben.)

2 Wie viele Probleme gibt es?

Σ^* ist die kleenesche Hülle des Alphabets Σ und daher eine unendliche Menge. Seine Kardinalität ist gleich der Kardinalität der Menge der natürlichen Zahlen:

$$|\Sigma^*| = |\mathbb{N}| = \aleph_0$$

Jedes Problem P ist eine Teilmenge von Σ^*, und die Menge aller Teilmengen einer Menge ist die Potenzmenge dieser Menge. Also suchen wir nach der Größe von

$$\mathcal{P}(\Sigma^*)$$

Wenn Sie sich an das Kapitel über Potenzmengen erinnern, werden Sie sich an diese Formel erinnern, die für alle Mengen M gilt:

$$|\mathcal{P}(M)| = 2^{|M|}$$

Daher haben wir:

$$|\mathcal{P}(\Sigma^*)| = 2^{|\Sigma^*|}$$
$$|\mathcal{P}(\Sigma^*)| = 2^{|\mathbb{N}|}$$
$$|\mathcal{P}(\Sigma^*)| = 2^{\aleph_0}$$

Hilft uns das? Nicht wirklich, denn wir wissen bereits, dass das $\aleph_0$ eine unendlich große Zahl ist. Also muss auch $2^{\aleph_0}$ unendlich groß sein. Aber bedeutet das, dass $2^{\aleph_0}$ größer als ist $\aleph_0$? Oder sind sie gleich groß? Unendlich ist unendlich, oder nicht?

3 Cantors Diagonalargument

Machen wir es so einfach wie möglich. Wir verwenden wieder unser Standard-Minimalalphabet

$$\Sigma = \{0, 1\}$$

Und dann listen wir einige Elemente von Σ^* auf:

$$\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, \cdots\}$$

Ein Problem ist eine Sprache und eine Sprache ist eine Teilmenge dieser Menge. Ein Problem könnte also die folgenden Instanzen enthalten, bei denen die Antwort "ja" lauten sollte (das sind die Wörter, die zur Sprache gehören):

$$P_1 = \{\varepsilon, \quad 1, 00, \quad 10, 11, 000, \quad 100, 101, \quad 111, \cdots\}$$

Ein anderes Problem (eine andere Sprache) könnte diese Wörter enthalten:

$$P_2 = \{ \quad 11, 000, 001, 010, \quad 101, 110, \quad \cdots\}$$

Hier ist ein drittes Problem:

$$P_3 = \{ \quad 11, \quad 001, 010, \quad 101, 110, 111, \cdots\}$$

usw.

Schreiben wir dies zusammen mit einigen anderen Aufgaben in eine Tabelle:

Σ^*	ε	0	1	00	01	10	11	000	001	010	011	100	101	110	111	⋯
P_1	ε		1	00		10	11	000				100	101		111	⋯
P_2							11	000	001	010			101	110		⋯
P_3							11		001	010			101	110	111	⋯
P_4	ε	0		00						010	011		101	110		⋯
P_5	ε		1	00	01	10	11	000	001		011	100			111	⋯
P_6		0	1			10		000		010						⋯
P_7		0				10	11		001	010	011		101	110	111	⋯
P_8	ε		1	00				000	001			100			111	⋯
P_9		0	1		01	10	11	000			011	100				⋯
P_{10}	ε	0			01		11		001	010		100		110	111	⋯
P_{11}		0		00	01			000		010			101	110		⋯
P_{12}		0	1	00		10			001		011	100			111	⋯
P_{13}	ε		1			10	11					100	101			⋯
P_{14}	ε				01					010	011		101	110		⋯
P_{15}			1		01	10	11				011	100		110	111	⋯
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

Wie man leicht erkennen kann, ist ganz klar, was in einer Zelle steht, wenn dort etwas steht. Das ist nämlich in der gesamten Spalte, zu der diese Zelle gehört, gleich, und aus der Spaltennummer kann man auch darauf schließen, welche Folgen von Nullen und Einsen das sein muss. Es geht also nur darum, ob in einer Zelle etwas steht oder nicht. Das kann man aber auch einfacher schreiben:

Wir erstellen eine neue Liste und schreiben eine 0 in jede Zelle der neuen Liste, wo die alte Liste leer war, und wir schreiben eine 1 in jede Zelle, wo die alte Liste ein Element von Σ^* enthielt. Wir brauchen auch die Überschrift (die Zeile mit dem blauen Hintergrund) nicht. Wenn wir dies tun, erhalten wir diese Liste:

P_1	1	0	1	1	0	1	1	1	0	0	0	1	1	0	1	⋯
P_2	0	0	0	0	0	0	1	1	1	1	0	0	1	1	0	⋯
P_3	0	0	0	0	0	0	1	0	1	1	0	0	1	1	1	⋯
P_4	1	1	0	1	0	0	0	0	0	1	1	0	1	1	0	⋯
P_5	1	0	1	1	1	1	1	1	1	0	1	1	0	0	1	⋯
P_6	0	1	1	0	0	1	0	1	0	1	0	0	0	0	0	⋯
P_7	0	1	0	0	0	1	1	0	1	1	1	0	1	1	1	⋯
P_8	1	0	1	1	0	0	0	1	1	0	0	1	0	0	1	⋯
P_9	0	1	1	0	1	1	1	1	0	0	1	1	0	0	0	⋯
P_{10}	1	1	0	0	1	0	1	0	1	1	0	1	0	1	1	⋯
P_{11}	0	1	0	1	1	0	0	1	0	1	0	0	1	1	0	⋯
P_{12}	0	1	1	1	0	1	0	0	1	0	1	1	0	0	1	⋯
P_{13}	1	0	1	0	0	1	1	0	0	0	0	1	1	0	0	⋯
P_{14}	1	0	0	0	1	0	0	0	0	1	1	0	1	1	0	⋯
P_{15}	0	0	1	0	1	1	1	0	0	0	1	1	0	1	1	⋯
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

Nun schauen wir uns die Elemente in der Diagonale genauer an:

P_1	1	0	1	1	0	1	1	1	0	0	0	1	1	0	1	⋯
P_2	0	0	0	0	0	0	1	1	1	1	0	0	1	1	0	⋯
P_3	0	0	0	0	0	0	1	0	1	1	0	0	1	1	1	⋯
P_4	1	1	0	1	0	0	0	0	0	1	1	0	1	1	0	⋯
P_5	1	0	1	1	1	1	1	1	1	0	1	1	0	0	1	⋯
P_6	0	1	1	0	0	1	0	1	0	1	0	0	0	0	0	⋯
P_7	0	1	0	0	0	1	1	0	1	1	1	0	1	1	1	⋯
P_8	1	0	1	1	0	0	0	1	1	0	0	1	0	0	1	⋯
P_9	0	1	1	0	1	1	1	1	0	0	1	1	0	0	0	⋯
P_{10}	1	1	0	0	1	0	1	0	1	1	0	1	0	1	1	⋯
P_{11}	0	1	0	1	1	0	0	1	0	1	0	0	1	1	0	⋯
P_{12}	0	1	1	1	0	1	0	0	1	0	1	1	0	0	1	⋯
P_{13}	1	0	1	0	0	1	1	0	0	0	0	1	1	0	0	⋯
P_{14}	1	0	0	0	1	0	0	0	0	1	1	0	1	1	0	⋯
P_{15}	0	0	1	0	1	1	1	0	0	0	1	1	0	1	1	⋯
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

In unserem Beispiel ist dies diese Sequenz: $100111110101111\cdots$. Und in dieser Reihenfolge invertieren wir alle Zeichen. So wird aus jeder 0 eine 1und aus jeder 1 wird eine 0. In unserem Beispiel erhalten wir damit das Problem mit dieser Signatur: $011000001010000\cdots$. Geben wir diesem Problem den Namen P_{iD}. (Dabei steht iD für »invertierte Diagonale«.)

3.1 Hier ist der Trick

Das wollen wir beweisen:

$$|\mathcal{P}(\Sigma^*)| > |\mathbb{N}|$$

Nehmen wir für einen Moment an, dass dies nicht stimmt. Also glauben wir für einen Moment $|\mathcal{P}(\Sigma^*)| \leq |\mathbb{N}|$ und dann werden wir sehen, wohin uns dieser Glaube führen wird.

Wenn $|\mathcal{P}(\Sigma^*)| \leq |\mathbb{N}|$, dann können wir jeder Teilmenge von Σ^* (also jedem Problem) eine bestimmte natürliche Zahl zuordnen. Dann können wir die Probleme nach dieser Nummer sortieren und erhalten eine Liste. Genau das haben wir auf den Seiten oben getan.

Das bedeutet aber auch, dass diese Liste alle Probleme enthalten muss, denn wenn jedes Problem eine Nummer hat, hat es seine eigene Zeile in der Liste: **Die Liste ist vollständig.**

Und wir haben einen Binärcode gefunden, der das Problem beschreibt. Der Code für das Problem an Position 1 der Liste war $101101110001101\cdots$, das Problem an Position 2 hatte den Code $000000111100110\cdots$ usw.

Und wenn jedes Problem in der Liste steht, dann muss auch das Problem mit dem Code $011000001010000\cdots$ da sein. Dies ist der Code, den wir von der umgekehrten Diagonale erhalten.

Mal sehen, wo in dieser Liste wir das Problem P_{iD} mit diesem Code finden können:

P_{iD}	0	1	1	0	0	0	0	0	1	0	1	0	0	0	0	⋯
P_1	1	0	1	1	0	1	1	1	0	0	0	1	1	0	1	⋯
P_2	0	0	0	0	0	0	1	1	1	1	0	0	1	1	0	⋯
P_3	0	0	0	0	0	0	1	0	1	1	0	0	1	1	1	⋯
P_4	1	1	0	1	0	0	0	0	0	1	1	0	1	1	0	⋯
P_5	1	0	1	1	1	1	1	1	1	0	1	1	0	0	1	⋯
P_6	0	1	1	0	0	1	0	1	0	1	0	0	0	0	0	⋯
P_7	0	1	0	0	0	1	1	0	1	1	1	0	1	1	1	⋯
P_8	1	0	1	1	0	0	0	1	1	0	0	1	0	0	1	⋯
P_9	0	1	1	0	1	1	1	1	0	0	1	1	0	0	0	⋯
P_{10}	1	1	0	0	1	0	1	0	1	1	0	1	0	1	1	⋯
P_{11}	0	1	0	1	1	0	0	1	0	1	0	0	1	1	0	⋯
P_{12}	0	1	1	1	0	1	0	0	1	0	1	1	0	0	1	⋯
P_{13}	1	0	1	0	0	1	1	0	0	0	0	1	1	0	0	⋯
P_{14}	1	0	0	0	1	0	0	0	0	1	1	0	1	1	0	⋯
P_{15}	0	0	1	0	1	1	1	0	0	0	1	1	0	1	1	⋯
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

- P_{iD} kann nicht das Problem P_1 sein, weil das 1. Element von P_1 1 ist, während das 1. Element der umgekehrten Diagonale 0 ist. Dass einige der anderen Elemente übereinstimmen und dass es noch mehr Elemente gibt, die unterschiedlich sind, ist unerheblich. Ein einziges nicht passendes Element reicht aus, um zu zeigen: $P_{iD} \neq P_1$.
- P_{iD} kann aber auch nicht das Problem P_2 sein, weil das 2. Element von P_2 0 ist, während das 2. Element der umgekehrten Diagonale 1 ist. Daher gilt: $P_{iD} \neq P_2$.

Sie können dieses Argument für jede einzelne Zeile in der Liste wiederholen. Die Signatur des Problems P_{iD} stimmt mit der Signatur des Problems P_n an der n-ten Position der Signatur nicht überein, daher ist $P_{iD} \neq P_n$ für alle n. Mit anderen Worten: Das Problem P_{iD} steht nicht in der Liste. Anders gesagt: **Die Liste ist nicht vollständig.**

Aus der Annahme, dass $|\mathcal{P}(\Sigma^*)| \leq |\mathbb{N}|$ ist, haben wir aber eine Seite vorher das genaue Gegenteil geschlossen. Weil man aber aus wahren Aussagen nicht zwei einander widersprechende Aussagen schlussfolgern kann, kann das kann nur bedeuten, dass $|\mathcal{P}(\Sigma^*)| \leq |\mathbb{N}|$ nicht wahr, sondern falsch ist. Also muss das Gegenteil wahr sein:

$$|\mathcal{P}(\Sigma^*)| > |\mathbb{N}|$$

Quod erat demonstrandum.
Und das bedeutet, dass die Anzahl aller Probleme überabzählbar unendlich ist:

$$|P| > \aleph_0$$

4 Unlösbare Probleme

In dem Dokument über wir die Anzahl der fleißigen Biber berechnet, die eine bestimmte Anzahl von Zuständen hatten. Fleißige Biber sind eine spezielle Klasse von Turing-Maschinen, aber diese Berechnung funktioniert mit allen Turing-Maschinen. Es ist ein wenig komplizierter, weil "normale" Turing-Maschinen mehr variable Elemente haben als fleißige Biber, aber es ist immer noch möglich, Gruppen von Turing-Maschinen mit einer bestimmten Größe zu definieren, und innerhalb einer solchen Gruppe kann man eine bestimmte Reihenfolge definieren (indem man eine Reihenfolge für die Zeichen in beiden Alphabeten definiert), und dann können Sie jeder Turing-Maschine eine individuelle natürliche Zahl zuweisen, und jede natürliche Zahl ist eine Nummer einer einzigartigen und unverwechselbaren Turing-Maschine.

Die Menge aller Turingmaschinen ist also unendlich groß, aber sie ist nur so groß wie $\mathbb{N}$:

$$|TM| = |\mathbb{N}| = \aleph_0$$

Und das bedeutet wiederum, dass es mehr Probleme als Turing-Maschinen gibt:

$$|P| > |TM|$$

Aber jede Turing-Maschine kann nur genau 1 Sprache akzeptieren, was bedeutet, dass jede Turing-Maschine nur genau 1 Problem lösen kann.

Und das bedeutet, dass es Probleme gibt, für die es keine Turing-Maschine gibt, die es lösen können.

Die Kardinalität der Menge aller Probleme stimmt mit der Kardinalität der Menge der reellen Zahlen überein. In einem gewissen Sinn kann man daher sagen, dass auf jedes Problem, das lösbar ist, unendlich viele unlösbare Probleme kommen. Das entspricht übrigens ganz genau auch der Aussage, dass es für jede reelle Zahl, die das Ergebnis einer Berechnung sein kann (egal, wie ausufernd und komplex diese Berechnung sein mag), unendlich viele reelle Zahlen gibt, die unmöglich das Ergebnis einer Berechnung sein können.

5 Das Halteproblem

Erinnern Sie sich an das Programm will_it_halt.py? Oder die 40 fleißigen Biber mit 5 Zuständen, die seit mehr als 30 Jahren laufen? Wäre es nicht schön, ein Programm zu haben, das den Quellcode eines solchen Programms oder die Beschreibung einer Turing-Maschine lesen könnte und Ihnen sagen würde, ob dieses Programm oder dieser Automat ewig laufen würde oder ob es anhalten würde?

Für viele Programme müssten Sie auch die konkrete Eingabe kennen, da es oft von der Eingabe abhängt, ob ein Programm anhält oder für immer läuft.

Wir brauchen also einen Halt-Tester, der die Beschreibung des Programms oder der Turing-Maschine lesen kann, und dieser Halt-Tester soll auch die Eingaben lesen, die diesem Programm zugeführt werden sollen. Der Halt-Tester würde den Code analysieren und prüfen, was er mit der Eingabe macht. Der Halt-Tester würde irgendwann zu einem Ende kommen, und dann eines der beiden folgenden Ergebnisse anzeigen:

- Das Programm, das getestet wurde, wird ewig laufen, wenn es den angegebenen Input vorgesetzt bekommt,
  oder
- Das Programm, das getestet wurde, wird angehalten, wenn es den angegebenen Input vorgesetzt bekommt.

5.1 Das ist unmöglich

So ein Halt-Tester ist leider unmöglich.

Es gibt ein schönes Video, das erklärt, warum das unmöglich ist: https://youtu.be/92WHN-pAFCs. Es ist 7 Minuten und 51 Sekunden lang. Das hier ist noch kürzer (4:13) und erklärt es auch sehr gut: https://youtu.be/VyHbd6sx5Po

Aber wenn Sie lieber lesen, hier ist der Beweis:

Wir führen wieder einen Beweis durch Widerspruch. Wir nehmen das Gegenteil an, und leiten aus dem Gegenteil dann zwei Aussagen ab, die einander widersprechen. Das ist aber nur möglich, wenn das Gegenteil falsch ist, woraus dann folgt, dass die ursprüngliche Aussage wahr sein muss.

Wir wollen beweisen, dass es keinen universellen Halt-Tester gibt. Das Gegenteil dieser Annahme ist: Es gibt einen universellen Halt-Tester. Das Adjektiv "universell" bedeutet: Der Tester kann für jede Kombination aus Programmbeschreibung und Beschreibung des Inputs exakt feststellen, ob das beschriebene Programm beim Verarbeiten des beschriebenen Inputs zu einem Ende kommt oder nicht. Sobald der Halt-Tester diese Aussage gemacht hat, hält er an.

Für unseren Beweis brauchen wir noch ein einfaches Programm, das wir »Inverter« nennen. Der Inverter akzeptiert nur zwei mögliche Eingaben: Wenn die Eingabe »es hält an« ist, verzweigt der Inverter in eine Endlosschleife und läuft in diesem Fall ewig weiter. Wenn die Eingabe für den Inverter "es läuft ewig" ist, dann hält er sofort an.

Wir bauen nun ein neues größeres Programm, indem wir den Halt-Tester und den Inverter zu einem neuen Programm vereinen, das wir "das absurde Programm" nennen: Das absurde Programm empfängt dieselbe Art von Input wie der Halt-Tester, also ein Programm und dessen Input. Das absurde Programm führt dann intern den Halt-Tester aus, der nach einer gewissen Zeit ganz sicher zu einem Ende kommt, und »es hält an« oder »es läuft ewig« an das absurde Programm zurückmeldet. Diese Meldung wird dann direkt an den Inverter geschickt. Das bedeutet gesamthaft: Wenn das zu testende Programm mit dem gegebenen Input ewig läuft, hält das absurde Programm an. Wenn das zu testende Programm mit dem gegebenen Input irgendwann anhält, läuft das absurde Programm ewig.

Bisher ist noch alles gut. Aber was ist, wenn die absurde Maschine ihren eigenen Code als Eingabe liest. Wenn der Halt-Tester sagt: »Die absurde Maschine hält«, dann läuft der Inverter für immer. Weil er aber ein Teil der absurden Maschine ist, wird auch die absurde Maschine für immer laufen. Der Halt-Tester lag also falsch. Aber wenn der Halt-Tester sagt: »die absurde Maschine läuft ewig«, dann hält der Inverter und damit auch die absurde Maschine sofort an. Also liegt der Halt-Tester auch in diesem Fall falsch. Es gibt also eine Eingabe, bei der der universelle Halt-Tester eine falsche Antwort gibt. Aber wir sind davon ausgegangen, dass der universelle Halt-Tester immer die richtige Antwort gibt. Das ist ein Widerspruch, und das bedeutet, dass es ein Programm wie den universellen Halt-Tester nicht geben kann.

Das Halteproblem ist ein unentscheidbares Problem.

5.2 Andere unentscheidbare Probleme

5.2.1 Posts Korrespondenzproblem

Benannt nach Emil Leon Post (1897-1954), einem in Polen geborenen amerikanischen Mathematiker und Logiker.

Am besten lässt es sich an einem Beispiel beschreiben: Es gibt zwei Wortlisten:

Elementnummer	Liste 1	Liste 2
1	a	aba
2	ab	bb
3	baa	aa

Gibt es eine Folge von Elementnummern, so dass die in dieser Reihenfolge angeordneten und dann verketteten Wörter das gleiche Ergebnis liefern, egal ob Sie diese Folge auf Liste 1 oder Liste 2 anwenden?

Eine Lösung für die gegebenen zwei Listen ist die Reihenfolge, $(1, 3, 2, 3)$ wenn Sie Wörter aus Liste 1 in dieser Reihenfolge schreiben, erhalten Sie:

$$a + baa + ab + baa = abaaabbaa$$

Wenn Sie dasselbe mit Liste 2 tun, erhalten Sie

$$aba + aa + bb + aa = abaaabbaa$$

Für dieses Listen-Paar gibt es also eine gemeinsame Sequenz, die zum gleichen Ergebnis führt. Die Frage, die sich nun stellt, lautet: Wenn man zwei bestimmte Listen von Wörtern gegeben hat: Gibt es dann eine gemeinsame Sequenz, so dass das Ergebnis der verketteten Wörter dann identisch ist? Diese Frage ist das postsche Korrespondenzproblem, und dieses Problem ist nachweislich ebenso unentscheidbar wie das Halteproblem.

5.2.2 Äquivalenzproblem

Das Äquivalenzproblem

Das ist die Frage, ob zwei Turingmaschinen dieselbe Sprache akzeptieren oder nicht.

Dies ist für kontextsensitive Sprachen und sogar für nicht deterministische kontextfreie Sprachen unentscheidbar.