# Terminieren, abstürzen, sterben, … vom Enden und Verenden von Algorithmen

GEORG TROGEMANN

Abstract: When an algorithm ends, this may have internal or external reasons. External causes include hardware failures, power breakdowns, or the termination by the user. One could speak of a natural inner end when the algorithm reaches its final state under its own control. However, it could be just as natural that it never reaches an end, for example when a non-rational number is calculated. Theoretical computer science teaches us that there can be no algorithm that decides for all other algorithms whether they end. In addition to hardware-breakdowns and software failures yet another type of algorithmic ending comes into play when in the field of natural computing the algorithms are realized by living organisms. Typical software life cycles provide further forms of continuation and survival of algorithms within semiotic ecosystems. The article concludes with a thought experiment on artificial intelligence.

## terminieren – Das Halteproblem

Eine Forderung an Algorithmen sei, so kann man häufig lesen, dass sie terminieren, also nach einer endlichen Zahl von Schritten zu einem Ergebnis kommen und enden. Schon ein nur oberflächlicher Blick in die Originalliteratur zeigt, dass dem nicht so ist. Algorithmen stehen im Zentrum der Berechenbarkeitstheorie, die verbunden ist mit Namen wie Kurt Gödel, Emil Post, Alonso Church und Alan Mathison Turing. Ihre Grundlegung erfolgte vor allem in den Jahren zwischen 1930 und 1940 im Zuge der Suche nach den Grenzen formaler Systeme. Wir greifen hier die bahnbrechende Arbeit von Alan Turing heraus, dessen Maschinenmodell heute als erstes genannt wird, wenn man nach einer Präzisierung des Algorithmenbegriffs fragt. Seine Untersuchungen zum maschinellen Rechnen werden 1936 unter dem etwas eigentümlichen Titel »On computable numbers, with an application to the Entscheidungsproblem« publiziert. Gleich auf der ersten Seite am Ende des

ersten Absatzes findet sich eine Annäherung an den Begriff: »According to my definition, a number is computable if its decimal can be written down by a machine«.[1] Berechenbar (algorithmisierbar) ist also, was von einer Maschine angeschrieben werden kann. Als explizite Beispiele nennt er die transzendenten Zahlen p und e sowie die Realteile algebraischer Zahlen. So ist die irrationale Zahl $\sqrt{2}$ als Lösung der Gleichung $x^2 - 2 = 0$ eine algebraische Zahl, die nach Turing berechenbar ist. Wir sehen also, dass Algorithmen enden, war nie eine Forderung von Turing. Es reicht, wenn sie in endlicher Zeit die nächste Ziffer liefern, zum Beispiel $\pi = 3.14159265 \ldots$ danach irgendwann eine 3, dann eine 5, die 8 und so weiter. Ein Ende wird diese Berechnung nicht finden. Dennoch ist die Terminierung von Algorithmen eine zentrale Frage für die gesamte Theorie der Berechenbarkeit. Das von Turing im gleichen Papier formulierte Halteproblem und seine Erweiterung durch Rice zählen zu den zentralen Ergebnissen der theoretischen Informatik. Dazu müssen wir allerdings etwas ausholen und zeigen, in welchem Kontext die Arbeit von Turing stand und was damit erreicht wurde.

Der Mathematiker David Hilbert war Anfang des 20. Jahrhunderts überzeugt, dass eine ›vollständige‹, ›widerspruchsfreie‹ und ›entscheidbare‹ Axiomatisierung der Mathematik möglich sei. Sein Ziel war, den durch die Grundlagenkrise in Frage gestellten Ruf der Mathematik, unanfechtbare Wahrheiten zu produzieren, auf der Basis rein syntaktischer Methoden wiederherzustellen. Es geht dabei nicht mehr um einzelne mathematische Sätze und deren Beweise, sondern um Metamathematik und unwiderlegbare Aussagen über die grundsätzliche Leistungsfähigkeit der Mathematik und ihrer Kalküle. Ein formales System heißt in diesem Zusammenhang ›vollständig‹, wenn alle wahren Aussagen innerhalb des Systems beweisbar sind, es heißt ›widerspruchsfrei‹, wenn sich für keine Aussage gleichzeitig deren Negation beweisen lässt und es heißt ›entscheidbar‹, wenn für jede Aussage entschieden werden kann, ob sie innerhalb des Formalismus beweisbar ist.[2] Nachdem Kurt Gödel 1930 zunächst gezeigt hatte, dass der Beweiskalkül für die Logik erster Stufe korrekt und vollständig ist und damit stark genug, um das hilbertsche Programm zu stützen, erfolgte 1931 der Schock. Gödel zeigte in seinem ersten Unvollstän-

1 Turing, Alan Mathison,1936, S. 230.

2 Hoffmann, Dirk W., 2018, S. 44–45.

digkeitssatz, dass jedes formale System, das stark genug ist, die elementare Arithmetik zu formalisieren, entweder widersprüchlich oder unvollständig ist. Im zweiten Unvollständigkeitssatz zeigte er dann, dass ein solches formales System seine eigene Widerspruchsfreiheit nicht beweisen kann.[3] Die Frage der Entscheidbarkeit war aber zu diesem Zeitpunkt noch offengeblieben. In einem Lehrbuch aus dem Jahre 1928 hatte Hilbert zusammen mit Wilhelm Ackermann die Entscheidbarkeit, bezogen auf die Prädikatenlogik erster Stufe, wie folgt präzisiert: »Das Entscheidungsproblem ist gelöst, wenn man ein Verfahren kennt, das bei einem vorgelegten logischen Ausdruck durch endlich viele Operationen die Entscheidung über die Allgemeingültigkeit bzw. Erfüllbarkeit erlaubt.«[4] Turing zeigte nun in seiner Arbeit, dass es ein Verfahren zur Lösung des Entscheidungsproblems nicht geben kann. Damit war das hilbertsche Programm endgültig gescheitert. Für den Beweis benutzt er jenes Maschinenmodell, das heute unter dem Begriff ›Turing-Maschine‹ bekannt ist und das den Berechenbarkeitsbegriff formal präzise fasst. Dabei appelliert Turing aber durchaus an die Intuition. Man soll sich vorstellen, wie ein Schulkind mit Papier und Bleistift eine Rechnung auf einem in Karos unterteiltem Papier ausführt. Das Band der Turingmaschine erhält man also, indem man die Karos eines Rechenheftes nicht mehr zweidimensional anordnet, sondern eindimensional hintereinander hängt. Jedes Karo kann genau eine Ziffer aufnehmen und für jede auszuführende Rechnung sollen immer genügend Kästchen zur Verfügung stehen. Durch einen Schreib-Lesekopf, der sich jeweils um ein Kästchen nach links oder rechts bewegen kann, können die aktuellen Ziffern (0 oder 1 oder blank) gelesen und auch überschrieben werden. Wer die Rechnung ausführt (egal ob Schulkind oder Maschine), soll immer nur ein Kästchen nach dem anderen betrachten und verändern können. Es gibt auch nur eine begrenzte Anzahl von bekannten Regeln (Geisteszuständen), die zur Anwendung kommen können, um die Zeichen auf dem Papier zu manipulieren. Durch die Analogie zwischen Maschine und menschlichem Rechner versucht Turing zu begründen, dass alle im intuitiven Sinne berechenbaren Funktionen auch von Maschinen ausgeführt werden können.

3 Neunhäuserer, Jörg, 2019, S. 91.

4 Hilbert, D.; Ackermann, W., 1928, S. 73–74. Die Bedeutung des Entscheidungsproblems für die Mathematik zeigt sich auch daran, dass der deutsche Begriff auch unter englischsprachigen Wissenschaftlern verwendet wurde.

Ein zentraler Absatz in Turings Artikel ist die Einführung der ›universellen Maschine‹.

> It is possible to invent a single machine which can be used to compute any computable sequence. If this machine U is supplied with a tape on the beginning of which is written the S.D [standard description] of some computing machine M, then U will compute the same sequence as M.[5]

Gibt man der universellem Maschine U die Beschreibung einer Maschine M, dann wird sie sich wie diese Maschine M verhalten. Man braucht also nicht unendlich viele Maschinen, eine einzige, die alle anderen simulieren kann, reicht aus. Turing gibt eine präzise Konstruktion dieser universellen Maschine an, mit der er, unter Verwendung der cantorschen Diagonalmethode, zeigt, dass es nicht-berechenbare Funktionen (Zahlen) gibt. Im Weiteren zeigt er dann die Unentscheidbarkeit des Halteproblems. »There can be no machine E which, when supplied with the S.D of an arbitrary machine M, will determine whether M ever prints a given symbol (0 say).«[6] Die universelle Maschine U bekommt also eine Maschine M (in Standardbeschreibung, d.h. als binäre Zahl codiert) vorgelegt und soll entscheiden, ob die Maschine (das Programm, der Algorithmus) hält oder nicht hält (beispielsweise eine 0 schreibt, wenn die Maschine M nicht hält). Es ist leicht einzusehen, dass das Halteproblem partiell-entscheidbar ist. Dazu kann man die Maschine M einfach auf einer universellen Maschine laufen lassen und wenn sie terminiert beispielsweise eine 1 ausgeben. Also muss die Komplementärmenge, d.h. die Menge der Programme, die nicht halten, unentscheidbar sein.

Die Besonderheit der Algorithmik besteht darin, dass eine statische Folge von Zeichen (das Programm und seine Ein- und Ausgaben) einen maschinellen Prozess codiert. Welche Zeichenfolgen korrekte Programme darstellen, ist durch die Syntax der verwendeten Programmiersprache festgelegt. Ein syntaktisch korrektes Programm kann aber vollkommen unsinnige Berechnungen durchführen. Woran wir wirklich interessiert sind, ist der maschinelle Prozess, den die Zeichen codieren, also ihre Semantik. In der Informatik sagt die Semantik nichts über die Beziehung eines Algorithmus zu seiner Außenwelt oder seine

5 Turing, Alan Mathison, 1936, S. 241–242, wie Anm. 1.

6 Ebd., S. 248.

Bedeutung für den Programmierer aus, sondern lediglich etwas darüber, welche inneren Zustände die Maschine durchläuft, wenn sie den Algorithmus ausführt. Die Semantik gibt also Auskunft darüber, welche mathematische Funktion berechnet wird und welche Eigenschaften die berechnete Funktion hat. Das Halteproblem ist also ein semantisches Problem der Algorithmik, das durch Analyse der Syntax nicht gelöst werden kann. Durch die Verallgemeinerung des Halteproblems konnte Henry Gordon Rice 1953 zeigen, dass keine nicht-triviale Eigenschaft von Algorithmen algorithmisch entschieden werden kann. Wobei nicht-trivial in diesem Zusammenhang heißt, dass es mindestens einen Algorithmus geben muss, der die gesuchte Eigenschaft hat, und mindestens einen, der sie nicht hat. Der Satz von Rice lautet dann wie folgt:[7]

> Sei E eine nicht-triviale funktionale Eigenschaft von Turing-Maschinen. Dann ist das folgende Problem unentscheidbar:
> Gegeben: Turing-Maschine M
> Gefragt: Besitzt M die Eigenschaft E?

Salopp formuliert: Algorithmen können keine allgemeinen Aussagen darüber machen, was andere Algorithmen tun, ob sie beispielsweise terminieren oder nicht.

Wir wollen die bisherigen Ergebnisse kurz zusammenfassen: 1) Turings Berechenbarkeitsbegriff hat nicht vorausgesetzt, dass Algorithmen halten. Wir kennen irrationale Zahlen (z. B. $\pi$, e, $\sqrt{2}$), die berechenbar sind, deren Berechnung aber nicht terminiert. 2) Halteproblem: Es kann kein Programm geben, das von allen anderen Programmen entscheidet, ob sie terminieren. 3) Partielle Entscheidbarkeit des Halteproblems: Wenn allerdings ein Programm mit einer bestimmten Eingabe hält, dann lässt sich das auch entscheiden. Man lässt das Programm dafür einfach auf einer universellen Turing-Maschine laufen und schaut, ob es hält. 4) Aus dem Satz von Rice folgt, dass man algorithmisch nichts über die Semantik (die berechnete Funktion und ihre Eigenschaften) beliebiger anderer Algorithmen sagen kann. Die Betonung liegt allerdings, wie schon beim Halteproblem, auf ›beliebig‹. Für Subklassen von Algorithmen lassen sich durchaus oft Lösungen finden.

7 Hoffmann, Dirk W., 2018, S. 300, wie Anm. 2.

Die berühmte ›Goldbachsche Vermutung‹, eine vom Mathematiker Christian Goldbach im 18. Jahrhundert aufgestellte Aussage, gehört zu den bekanntesten ungelösten Problemen der Zahlentheorie. Sie lautet: Jede gerade Zahl, die größer ist als 2, ist die Summe zweier Primzahlen. So ist 4 = 2 + 2, wobei 2 eine Primzahl ist.[8] In Abbildung 1 ist links ein Algorithmus zur Überprüfung der goldbachschen Vermutung skizziert, rechts sind die ersten zehn Goldbachzahlen aufgelistet. Der Algorithmus beginnt mit der Zahl 4 und untersucht von da ausgehend alle geraden Zahlen (6, 8, 10, 12, 14 …). Für jede einzelne gerade Zahl werden systematisch alle Kombinationen ungerader Zahlen durchprobiert und überprüft, ob beide Primzahlen sind. Hinter dem Ausdruck ›`i prim`‹ verbirgt sich also ein Unterprogramm, das eine Zahl i darauf testet, ob es eine Primzahl ist. Der Algorithmus bricht nur ab, wenn er eine gerade Zahl findet, die nicht die Summe zweier Primzahlen ist. Das Beispiel zeigt, dass die Lösbarkeit des Halteproblems große Vorteile für die Mathematik hätte. Hartnäckige mathematisch ungelöste Probleme, wie die goldbachsche Vermutung lassen sich nämlich wie der einfache Algorithmus zeigt, auf das Halteproblem reduzieren. Man könnte der Halteproblemlösung den Algorithmus aus Abbildung 1 vorlegen und hätte damit die goldbachsche Vermutung widerlegt oder bewiesen, je nachdem, ob der Algorithmus hält oder nicht. Das Beispiel macht auch intuitiv klar, warum sich semantische Aussagen über Funktionen nicht auf syntaktische Betrachtungen reduzieren lassen. Um die goldbachsche Vermutung zu beweisen, muss man Zahlentheorie betreiben, davon steht aber überhaupt nichts in dem kurzen Code, über den entschieden wird. Ein Verfahren, welches das Halteproblem löst, müsste also das hier codierte Problem verstehen und anschließend Zahlentheorie betreiben. Umgekehrt kann man bereits an diesem kleinen Beispiel sehen, wie schwierig es für Programmierer oft ist, zuverlässige Aussagen über das Programmverhalten zu machen. Niemand auf der Welt kann im Moment sagen, ob der kurze Code aus Abbildung 1 terminieren wird oder nicht.

Turings Arbeit ist in der Metamathematik verortet. Obwohl er auch konkrete Berechnungen vorführt, geht es ihm weniger darum zu zeigen, wie bestimmte Zahlen (Funktionen) berechnet werden können, sondern, was prinzipiell

8 Die 2 ist die einzige gerade Zahl, die Primzahl ist. Als Summanden für Zahlen größer 4 kommen also nur ungerade Zahlen in Frage.

```
n = 4;
conjecture = true;
while (conjecture == true) {
   n = n + 2;
   conjecture = false;
   for (i = 3, to n-2, n+2) {
        j = n - i;
        if (i prim && j prim){
            conjecture = true;
            break;
        }
   }
}
print("Goldbachs Vermutung ist falsch!");
```

```
(4 = 2 + 2)
6 = 3 + 3
8 = 3 + 5
10 = 3 + 7 = 5 + 5
12 = 5 + 7
14 = 3 + 11 = 7 + 7
16 = 3 + 13 = 5 + 11
18 = 5 + 13 = 7 + 11
20 = 3 + 17 = 7 + 13
22 = 3 + 19 = 5 + 17 = 11 + 11
...
```

Abb. 1: Goldbachsche Vermutung als Pseudocode (links). Die ersten 10 Goldbachzahlen (rechts).

unberechenbar ist. Die Grenzen der Mathematik selbst stehen zur Diskussion und sollen im hilbertschen Programm auf ein tragfähiges Fundament gestellt werden. Turing-Maschinen sind Teil dieser Mathematik und basieren auf Formalismen und regelbasierten Zeichenmanipulationen. Hier gibt es noch nicht die heutige Trennung zwischen einer mathematischen Maschine und ihrer Anwendung; die Mathematik selbst ist die Anwendung. Die Terminierung der Algorithmen ist dabei eine rein logische Frage, keine Frage der Hardware oder sonstiger äußerer Umstände. 1936 gab es noch keine programmierbaren, elektronischen Computer im heutigen Sinne, Turings Maschinen sind noch reine Papiermaschinen. In Turings zeichenbasierter Welt des Formalen gibt es zwar konzeptionell eine minimale Hardware, nämlich das Papierband und den Schreib-Lesekopf. Um prinzipielle mathematische Aussagen machen zu können, muss diese aber idealisiert werden. Das Band braucht potentiell unendliche viele Kästchen, um die berechenbaren Probleme auch tatsächlich berechnen zu können. Auch Zeit, zum Beispiel für die Bewegung des Kopfes, für das Lesen, Prozessieren und Schreiben, kommt nicht explizit vor, sondern nur implizit als Anzahl von Schreib-Lese-Schritten, die notwendig sind, um ein bestimmtes Problem zu lösen. Um die im Zentrum stehenden formalen Fragen zu bearbeiten, ist es notwendig, sich abstrakte Prozesse vorzustellen, die keinen Alterungsprozessen, Materialermüdungen und sonstigen Defekten unterliegen. Kaputtgehen ist keine Option für mathematische Operationen.

Einzig die Logik entscheidet, ob die Berechnung endet oder nicht. In dieser Denkweise potentiell unendlicher Ressourcen, Laufzeiten und Programmgrößen gibt es auch kein Außen. Es geht um die inneren Grenzen der Formalismen, die auch von innen her ausgelotet werden. Selbst philosophische Grundfragen der Mathematik, etwa wie mathematisches Wissen zustande kommt, was mathematische Erkenntnis ist oder in welchem Sinne mathematisches Wissen als sicher gelten kann, werden durch die Berechenbarkeitstheorie nicht wirklich berührt. Natürlich können die Berechnungen immer auch eine praktische Anwendung haben, doch wie sich die Algorithmen zur Wirklichkeit verhalten, wird dort nicht geklärt.

## abstürzen – Computer Bugs

Am 9. September 1947 gerät eine Motte zwischen die Kontakte des gerade schaltenden Relays #70 des elektromechanisches Rechners Harvard Mark II im Naval Weapons Laboratory, Dahlgren, Virginia. Die Begegnung bedeutet für beide das Ende, die Motte findet den Tod, die Berechnung ihren vorzeitigen Abbruch. Die Motte wird von den Operatoren der Maschine entfernt und der Fehler akkurat protokolliert. Heute gilt das Ereignis als ›first actual case of bug being found‹. Das komplette Logbuch, mit aufgeklebter Motte und entsprechender Notiz, ist heute Teil der Sammlung des Smithsonian National Museum of American History (vgl. Abb. 2).[9] In der Informatik meint der Begriff ›debugging‹ das Finden und Beheben von Softwarefehlern, wobei als Ursprung irreführender Weise meist der Fehler bei der Mark II angegeben wird. Tatsächlich war der Begriff ›bug‹ bei den Ingenieuren aber schon im 19. Jahrhundert im Gebrauch und meinte immer schon »a defect or fault in a machine, plan, or the like«.[10] Das Finden eines echten Bugs war also ein eher amüsantes Ereignis, worauf die Protokollnotiz ja auch hinweist. So genannte ›debugger‹ helfen Programmierern heute dabei, die Probleme in Programmen aufzuspüren. Der in der Mark II gefundene Computer Bug war dagegen kein

9 https://en.wikipedia.org/wiki/Software_bug#/media/File:H96566k.jpg (zuletzt abgerufen: 19. Mai 2020)

10 Shapiro, Fred, R., 1994.

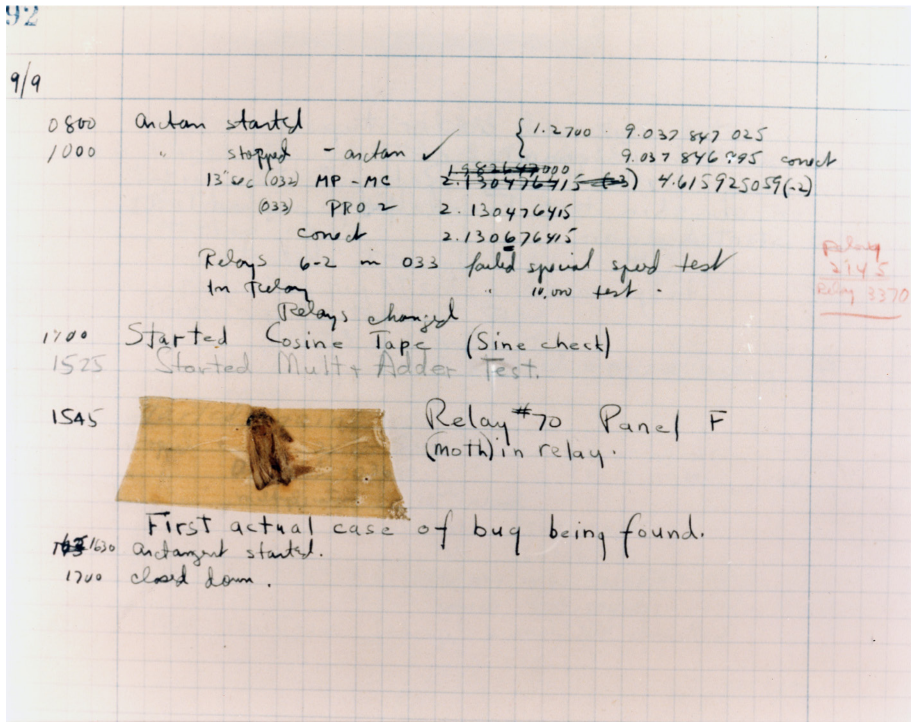


Abb. 2: Der erste ›Computer Bug‹, Bild mit freundlicher Genehmigung des U.S. Naval Historical Center Online Library Photograph.

Softwareproblem, sondern ein Hardwareproblem. In der Praxis des Rechnens sind es diese beiden Fehlertypen – Hardwarefehler bzw. Programmierfehler – die zu katastrophalen Systemabstürzen führen können.

Rechenprozesse in heutigen Computern sind ein komplexes, hierarchisches und heterarchisches Zusammenspiel unterschiedlicher semiotischer Systeme (Betriebssysteme, Maschinensprachen, Compiler und Interpreter, Entwicklungsumgebungen, höhere Programmiersprachen, Netzwerkprotokolle, etc.) mit Schnittstellen zur Außenwelt. Im Gegensatz zu den in sich abgeschlossenen mathematischen Operationen bei Turing gibt es hier eine enge Wechselwirkung zwischen den Algorithmen und ihrer Umwelt einerseits und zwischen der Hard- und den Softwareebenen andererseits. Roboter, Betriebssysteme, Internet-Dienste usw. sind auf kein Ende hin programmiert, sie sind in ihrem Kern immer schon Endlosschleifen und die Terminierung als programmierte

Option ist der Ausnahmefall. In heutigen stark miteinander vernetzten Systemen zwingt schon der Verdacht auf versteckte Fehler zu vorbeugenden Maßnahmen. Der Y2K Bug zum Jahrtausendwechsel hatte letztlich keine Folgen. Die Industrie musste aber viele Milliarden Euro investieren, da Kettenreaktionen befürchtet wurden, deren Folgen nicht absehbar waren. Die Schwierigkeit, fehlerfreie Programme zu entwickeln, besteht darin, dass für alle Eventualitäten, in die ein Programm während der Ausführung durch die Interaktion mit seiner Umgebung kommen kann, bereits während der Entwicklung Lösungen vorhergesehen werden müssen. Software unterscheidet sich dabei nicht so sehr von anderen komplexen Technologien. Wie die noch immer aktuelle Analyse von Charles Perrow aus dem Jahr 1988 gezeigt hat, kommen bei Katastrophen Ereignisse zusammen, die einzeln beherrschbar und auch vorhersehbar sind.[11] Erst das unwahrscheinliche, gleichzeitige Auftreten und ihr Zusammenspiel macht diese Ereignisse für komplexe und vor allem eng gekoppelte Systeme bedrohlich. Der Ausdruck ›Software‹ wurde erstmals 1958 von John W. Tukey verwendet, als Komplement zum älteren Begriff ›Hardware‹, der die physischen Bestandteile des Computers bezeichnet. Die 60 Jahre, in denen man von Softwareentwicklung sprechen kann, drücken sich nicht zuletzt in verschiedenen Listen berühmter Programmierfehler aus. Im Internet zu findende Verzeichnisse mit detaillierten Ursachenerklärungen tragen Titel wie: »11 most costly software errors in history«, »20 famous software desasters«, »5 most embarrasing software errors in history«. Software-Engineering ist der Versuch der Informatik, das Risiko solcher Fehler zu minimieren.

## sterben – Organismen als Algorithmen

Informatik ist die neue Hilfswissenschaft der Naturwissenschaften. Computersimulationen werden in vielen Bereichen der Naturwissenschaft mit großem Erfolg eingesetzt. Umgekehrt versucht aber auch die Algorithmik von den Naturwissenschaften zu lernen und sich deren Wissen zunutze zu machen. Einmal, indem sie sich von ihr inspirieren lässt und natürliche Prozesse als Vorbild für algorithmische Prinzipien nimmt. Auf diese Weise sind sehr leistungsfähige

11 Perrow, Charles, 1988.

heuristische Algorithmen entstanden, die aus der Informatik nicht mehr wegzudenken sind. Künstliche Neuronale Netze und die hebbsche Lernregel, Genetische Algorithmen, Membrane Computing und Schwarmintelligenz zählen zu den vermutlich bekanntesten Beispielen. Zum anderen können lebendige Organismen aber auch direkt genutzt werden, um damit zu rechnen, ohne den Umweg über numerische Simulationen zu nehmen. Dazu braucht es dann allerdings ein anderes Programmierparadigma. Bei der Programmierung von Turing-Maschinen muss folgende Frage beantwortet werden: Wie kann ich eine Folge elementarer Operationen so hintereinanderschalten, dass ein bestimmter Maschinenzustand erreicht wird? In der Molekular-Programmierung wird die Aufgabe nach Rozenberg etwas anders formuliert: »How to design a set of initial molecules so that a certain type of molecular complexes will be formed.«[12] Es geht also nicht mehr darum, jeden einzelnen Befehl und die Reihenfolge festzulegen, sondern eine Anfangskonfiguration zu finden, so dass sich aufgrund der Eigendynamik des Systems von selbst ein bestimmter molekularer Komplex formt. Dieser Komplex aus Molekülen repräsentiert dann das gesuchte Ergebnis. Allgemeiner ausgedrückt kann jeder natürliche, selbstablaufende Prozess als Rechenprozess interpretiert werden, wenn es gelingt folgende Frage zu beantworten:

> Wie kann ich einen Anfangszustand des Systems so konfigurieren, dass sich messbare Zustände einstellen, die ich als gesuchtes Ergebnis interpretieren kann?

Wir suchen hier nicht mehr nach Universalrechnern, sondern sind auch mit einer Speziallösung zufrieden, wenn die Aufgabe ausreichend interessant oder wichtig ist.

Die Grundidee für solche Rechenprinzipien finden sich bereits in der frühen Kybernetik. Die beiden englischen Kybernetiker Stafford Beer und Gordon Pask untersuchten in den 1950er Jahren verschiedene Substrate, um selbstorganisierende Maschinen zu bauen. Sie sahen fixe Hardware-Strukturen als Behinderung und suchten nach neuen Möglichkeiten, Systeme herzustellen (auszubrüten), deren ›Intelligenz‹ sich von selbst verstärkt. Natürliche, sich selbsterhaltende Lebensformen sollten hierfür mit kybernetischen Funktionen ›überlagert‹ werden. Statt neue Systeme ›from scratch‹ zu entwickeln, werden

12 Rozenberg, Grzegorz, 2008, S. 378.

hoch-dynamische Prozesse mit Kontrollprozessen gekoppelt. So experimentierte Stafford Beer beispielsweise mit Wasserflöhen.

> Beer then investigated groups of Daphnia, a freshwater crustacean. He added iron filings to the tank, which were eaten by the animals. Electromagnets were used to couple the tank with the environment (the experimenter). Beer could change the properties of magnetic fields, which in turn effected changes in the electrical characteristics of the colony. Initially this approach seemed to have potential, as the colony »retains stochastic freedom within the pattern generally imposed—a necessary condition in this kind of evolving machine; it is also self-perpetuating, and selfrepairing, as a good fabric should be« (Beer). However, not all of the iron filings were ingested by the crustaceans and eventually the behavior of the colony was disrupted by an excess of magnets in the water.[13]

Algorithmische Experimente mit Organismen zählen heute zum Bereich des ›Natural Computing‹, der sehr heterogene Ansätze versammelt. Baumgardner et al. nutzen beispielsweise Bakterien, um graphentheoretische Probleme zu lösen.[14] Ein weiterer Organismus, der nicht zuletzt ebenfalls aufgrund seiner algorithmischen Problemlösungsfähigkeiten zunehmendes Forschungsinteresse erfährt, sind Schleimpilze. Atsushi Tero et al. verwenden den Schleimpilz ›Physarum Polycephalum‹, um von diesem einzelligen Organismus das Tokioter Eisenbahnsystem entwerfen zu lassen.[15] Abbildung 3 (linke Seite, A - F) zeigt die Netzwerkbildung in Physarum Polycephalum innerhalb eines Zeitraums von 26 Stunden. Die horizontale Breite jedes Feldes beträgt dabei 17 cm. Ein kleines Plasmodium von Physarum wird im Modell, einer durch die Pazifikküste begrenzten Versuchsarena, am Standort Tokio platziert und durch zusätzliche Nahrungsquellen in jeder der größeren Städte der Region (weiße Punkte) ergänzt. Das Plasmodium wächst unter diesen Randbedingungen aus der ursprünglichen Nahrungsquelle heraus und besiedelt nach und nach auch die anderen Nahrungsquellen. Das sich ausbreitende Myzel bildet dabei ein Netzwerk von Röhren aus, die die Nahrungsquellen miteinander verbinden. Die deutlich sichtbaren Röhren sind die gesuchte Lösung des Entwurfsproblems. Eine robuste Netzwerkleistung von Transportsystemen

13 Bird, Jon & Di Paolo, Ezequiel, 2008, S. 200.

14 Baumgardner, Jordan et al., 2009.

15 Tero, Atsushi, et al., 2010.

wie dem Tokioter Eisenbahnsystem erfordert einen komplexen Kompromiss zwischen Kosten, Wegeeffizienz und Fehlertoleranz. Der Versuch für das Tokioter Netz belegt, dass der einzellige Physarum Polycephalum ähnlich gute Bewertungen erreicht wie das reale, von Ingenieuren entworfene Netzwerk. In Abbildung 3 (mitte) ist dargestellt, wie durch Beleuchtung Randbedingungen verändert werden können. (A) Ohne Beleuchtung entsteht ein Physarum-Netzwerk, das den verfügbaren Raum gleichmäßig ausnutzt. (B) Dem sich entwickelnden Physarum-Netz werden mit Hilfe einer Beleuchtungsmaske geographische Beschränkungen auferlegt, die das Wachstum auf schattigere Gebiete beschränken, die niedrig gelegenen Bezirken in der Region Tokio entsprechen. Der Ozean und die Binnenseen wurden ebenfalls stark beleuchtet, um Wachstum an diesen Stellen zu verhindern. Abbildung 3 (rechts) zeigt den Vergleich der Physarum-Netze mit dem realen Eisenbahnnetz von Tokio. Das resultierende Netzwerk (C) wurde mit dem realen Schienennetz im Raum Tokio verglichen (D). Der ›minimale Spannbaum‹ (E) verbindet die gleiche Menge von Stadtknoten und (F) zeigt ein Modellnetzwerk, das durch das Hinzufügen zusätzlicher Verbindungen zum minimalen Spannbaum entsteht. Wir sehen hier die Wechselwirkung zwischen Algorithmik und natürlichen Prinzipien. Der Schleimpilz wird nicht nur beobachtet, sondern seine Kernmechanismen werden in biologisch inspirierten mathematischen Modellen erfasst und so die Übertragung der Ergebnisse auf andere Anwendungsbereiche ermöglicht. Die Erkenntnisse darüber, wie Schleimpilze das Optimierungsproblem lösen, führen auf diese Weise zu neuen mathematischen Optimierungsverfahren in der Simulation. Am Ende der Rechenprozesse steht der Tod der Schleimpilze.

## Software-Lebenszyklen

Der Lebenszyklus einer Software beginnt mit der Entwicklungsidee, erreicht mit der lautstarken Marktankündigung einen Höhepunkt und endet in der Regel leise durch die stetige Abnahme der Installationszahlen. Dazwischen liegen je nach Vorgehensmodell unterschiedliche Entwicklungsstufen und meist eine Reihe von Update- und Upgrade-Releases. Je nachdem, wie stark Software in ihre Umwelt eingebunden ist, entsteht unterschiedlicher Druck zur permanenten Anpassung und Weiterentwicklung. Die Komplexität der

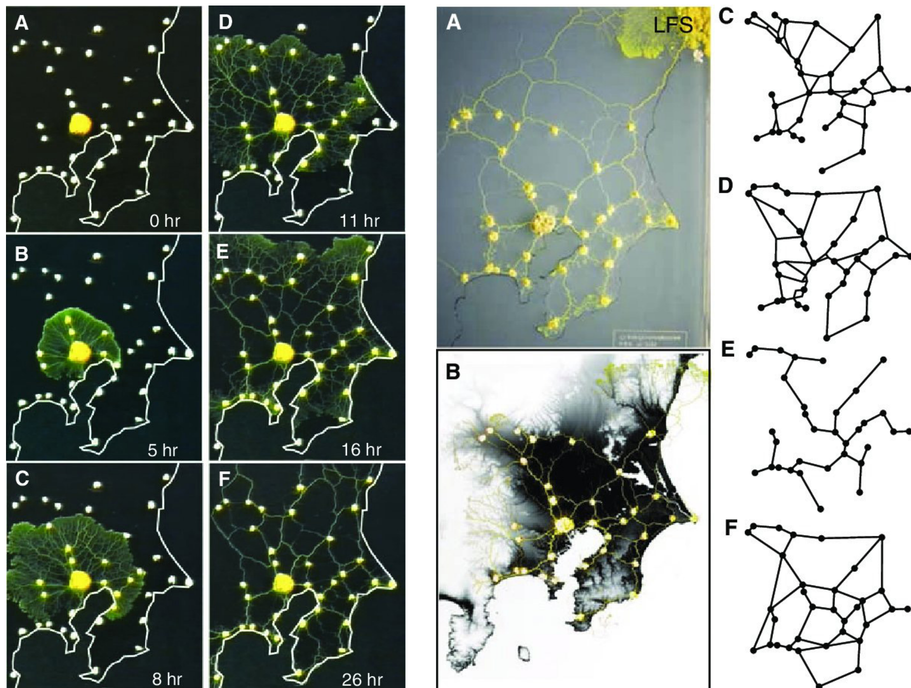


Abb. 3: Netzwerkformierung in ›Physarum Polycephalum‹, Toshiyuki Nakaga et al., Research Institute for Electronic Science, Hokkaido University, Sapporo, Japan.

Modellierung, die dabei durch Software geleistet wird, drückt sich in der Klassifikation bei Lehman wie folgt aus: »[…] any Program is a model of a model within a theory of a model of an abstraction of some portion of the world or of some universe of some discourse.«[16] Im Weiteren unterscheidet Lehman dann drei Klassen von Programmen (S, P und E), die sich durch die Art der Einbettung in den Anwendungsbereich unterscheiden. S-Programme (specification) sind Programme, die formal definiert sind und vollständig aus der Spezifikation des Problems abgeleitet werden können. Als Beispiel nennt er das ›Travelling Salesman Problem‹, das ›kleinste gemeinsame Vielfache‹ einer Zahl und das so genannte ›Damenproblem‹ beim Schach. Diese Probleme beziehen sich, wie auch die mathematischen Betrachtungen bei Turing, auf ihr eigenes, formalisierbares Diskursuniversum. Obwohl die Probleme aus

16 Lehman, 1980, S. 1061.

praktischen Situationen abgeleitet sein können, werden sie künstlich separiert und stehen, wenn sie ausgeführt werden, in keiner kausalen Beziehung zur Welt mehr. Die Korrektheit einer Zinseszinsberechnung beispielsweise lässt sich anhand der Spezifikation prüfen, man muss nichts über Schuldner und deren private Situation wissen.

Die Klasse der P-Programme (real world problem solution) zeichnet sich dadurch aus, dass es für die Akzeptanz der Lösung nicht mehr ausreicht, wenn die Software eine formale Spezifikation erfüllt, sondern die Lösung muss sich im alltäglichen Betrieb bewähren. Als Beispiel nennt er das Travelling Salesman Problem, wie es sich in der Praxis darstellt. Die praxistaugliche Bestimmung von Reiserouten für Verkäufer kann zwar immer noch auf Methoden des klassischen, mathematischen Problems beruhen, muss aber Werturteile, Terminkalender, Arbeitsabläufe in unterschiedlichen Betrieben bis hin zu Idiosynkrasien der Verkäufer und Kunden berücksichtigen. Ob die angebotene Softwarelösung akzeptabel ist, kann nicht mehr durch Vergleich mit einer formalen Spezifikation entschieden werden, sondern durch die subjektive Bewertung und die Zufriedenheit der Nutzer. In Lehmans S-P-E-Betrachtungsweise ist die Wettervorhersagesoftware ein typisches P-programm. Obwohl die hydrodynamischen Gleichungen bekannt sind, können algorithmisch nur Approximationen simuliert werden. Die Gemeinschaft der Meteorologen entwickelt ständig neue Simulationsmodelle und auch die Erfahrungen mit der Performance aktueller Modelle und sonstige technische Weiterentwicklungen und Erkenntnisse führen zu immer wieder neuen algorithmischen Vorhersagemethoden. Das Wissen über das Wetter ist also durchaus dynamisch, trotzdem verändert die Wettervorhersage nicht das Wetter selbst. In anderen Anwendungsfällen bildet Software nicht einfach Realität ab, sondern erzeugt sie. Programme werden Teil interagierender Netze und sind eingebunden in realweltliche Prozesse. Dabei verändern sie die Wirklichkeit, für die sie ursprünglich nur Modelle sein sollten.

Lehmans E-Programme (embedded) zeigen in diese Richtung, obwohl Lehmans obige Softwaredefinition zeigt, dass er insgesamt noch in einer Modelltheorie gefangen ist, die von der semiotischen Beschreibbarkeit des zu modellierenden Weltausschnitts ausgeht. Tatsächlich muss man aber von der Umkehrung ausgehen: Software stellt Realität her! Verkehrskontrollsysteme, Social-Media-Anwendungen oder algorithmischer Aktienhandel modellieren nicht nur ihre Anwendungsbereiche, sie verändern sie. Der algorithmische

Hochfrequenzhandel hat Rückwirkungen auf das gesamte Börsenspiel und erhöht beispielsweise das Risiko unvorhersehbarer ›Flash Crashes‹ beträchtlich. Das heißt, die Existenz dieser Systeme wirkt direkt auf die realen Gegebenheiten zurück und erzeugt permanenten Druck auf die Anpassung der Sichtweisen, die ihren Funktionen und Leistungsangeboten zugrunde liegen. Robotik, lernfähige KI-Systeme, Big-Data-Algorithmen, selbstfahrende Autos und viele andere Software-Systeme müssen deshalb vor allem in ihrem Potential zur Realitätserzeugung untersucht werden, will man ihre tatsächliche gesellschaftliche Bedeutung verstehen.

## Paradoxien des Endes künstlicher Intelligenzen

Die Kognitionswissenschaft identifiziert Denken und Intelligenz als Informationsverarbeitung. Aus der multiplen Realisierbarkeit von Information, zum Beispiel als digitale Schaltkreise auf Silizium-Chips, als Qubits in Quantencomputern oder als DNA in Molekularcomputern wird die Substratunabhängigkeit kognitiver Prozesse abgeleitet.[17] Der abstrakte Informationsbegriff ermöglicht die ontologische Gleichsetzung von Maschine und Organismus. Interessant ist in diesem Zusammenhang die Frage, inwieweit die diskutierten Beschränkungen der Berechenbarkeitstheorie (Gödel, Turing, Church et al.) auch einer symbolbasierten KI bereits grundsätzliche Grenzen auferlegen. Vor allem die fehlende Fähigkeit formaler Kalküle, aus sich selbst herauszuspringen, dürfte sich als Hindernis für die Realisierung höherer Formen der Intelligenz erweisen. Es gibt aber auch andere schwerwiegende Gründe, heutigen KIs das Denken abzusprechen, insbesondere mit Hinweis auf ›Situiertheit‹ und ›Embodiment‹ des menschlichen Denkens und seine biologische Entwicklungsgeschichte und soziale Gebundenheit. Dennoch können zweifellos schon heute komplexe kognitive Fähigkeiten sehr beeindruckend simuliert werden.

Zum Abschluss wollen wir ein kleines Gedankenexperiment anstellen: Was würde folgen, wenn sich die Prämisse der Kognitionswissenschaft überraschend

17 Siehe dazu auch den Begriff ›Funktionalismus‹ in der Philosophie des Geistes. Zum Beispiel: Stanford Encyclopedia of Philosophy, https://plato.stanford.edu/entries/functionalism/ (zuletzt abgerufen: 4. Juni 2020).

als richtig erweist, dass Intelligenz und vielleicht auch Bewusstsein keine materiellen Phänomene sind, sondern organisatorische? Was, wenn sich Intelligenz doch digital realisieren lässt? Was für Schlussfolgerungen ergäben sich daraus für die Eigenschaften digitaler Intelligenzen? Sie erbten tatsächlich Merkmale aus beiden Welten, der digitalen genauso wie der menschlichen. Um als ›starke KI‹ zu gelten, müssten diese Systeme die gleichen intellektuellen Fähigkeiten haben wie Menschen. Sie müssten also wie Menschen logisch denken können, lernfähig sein, Entscheidungen treffen, in menschlicher Sprache kommunizieren und sich selbst und ihr Tun reflektieren können. Wie stark diese Fähigkeiten mit anderen Veranlagungen verbunden sind, ob beispielsweise, um von Denken sprechen zu können, immer schon Bewusstsein und Empfindungen nötig sind, ist noch immer strittig. Sollte silikonbasierte digitale Intelligenz aber möglich sein, würden diese Intelligenzen aufgrund ihrer Digitalität immer schon über die Möglichkeiten menschlicher Intelligenz hinausgehen.

Das zentrale Merkmal digitaler Systeme sind diskrete Zustände. Diese lassen sich nicht nur kopieren und durch Netzwerke übertragen, sie erlauben auch den Zugriff auf jedes einzelne Zeichen in allen semiotischen Schichten. Denken auf digitaler Basis ist damit nicht nur vollkommen transparent, sondern auch bis ins einzelne Bit veränderbar. Auch subsymbolische neuronale Netze erlauben im Prinzip diese detaillierte Manipulation, hier fehlt es im Gegensatz zur symbolischen KI allerdings (noch) am Verständnis, um etwa gezielte Eingriffe an einzelnen Neuronengewichten vorzunehmen. Ohne großen Aufwand könnte jederzeit eine identische Kopie einer digitalen Intelligenz – zum Beispiel als Backup – hergestellt werden, mit allen ›Erfahrungen‹, die die KI bis zu diesem Zeitpunkt gemacht hat. Digitale Intelligenzen könnten aber auch problemlos über Netzwerke versendet und ausgetauscht werden. Ihre Erfahrungen ließen sich anders als menschliche Erfahrungen problemlos vergesellschaften. Die biologisch motivierte Unterscheidung zwischen Ontogenese und Phylogenese würde sich generell erübrigen, bzw. sie würde ergänzt durch viele weitere neue Möglichkeiten, erworbene Fähigkeiten an andere Intelligenzen weiterzugeben. Die für Lebewesen charakteristischen binären Zustände lebendig-tot lösten sich auf in eine Vielzahl möglicher Enden mit genauso vielen möglichen Fortsetzungen in anderer Form. So, wie es für digitale Software-Module und Daten schon heute üblich ist, könnten Denkprozesse beliebig angehalten und zu einem späteren Zeitpunkt an einem anderen Ort (auf einer anderen Ma-

schine) fortgesetzt werden. Intelligenz würde zur beliebig konfigurierbaren, rekombinierbaren und damit frei skalierbaren Größe. Module für bestimmte kognitive Leistungen könnten mit anderen kognitiven Modulen zu leistungsfähigeren Systemen aufgabenbezogen oder dauerhaft zusammengeschlossen werden. Durch Client-Server-Architekturen könnten Denkleistungen ausgelagert und nur bei Bedarf hinzugebucht werden usw. Gedankenexperimente wie das bekannte ›Chinese Room Experiment‹ von Searle liefern überzeugende intuitive Argumente gegen starke KI als regelbasierte Zeichenmanipulation, obwohl sie kein Beweis sind. Auch die hier angestellten Überlegungen zu digitalen Intelligenzen stützen eher die Intuition, dass ›starke KI‹ nicht auf digitalen Silicon-Chips realisierbar sein wird.

## Bibliographie


Baumgardner, Jordan et al.: »Solving a Hamiltonian Path Problem with a bacterial computer«, in: *Journal of Biological Engineering* 2009, vol. 3:11.

Bird, Jon & Di Paolo, Ezequiel: »Gordon Pask and His Maverick Machines«, in: Philip Husbands/Owen Holland/Michael Wheeler (Hgg.), *The Mechanical Mind History*, MIT Press, 2008, S. 185–211.

Hilbert, D.; Ackermann, W.: *Grundzüge der theoretischen Logik*, 1. Auflage. Berlin, Heidelberg, Springer-Verlag, 1928.

Hoffmann, Dirk W.: *Grenzen der Mathematik. Eine Reise durch die Kerngebiete der mathematischen Logik*, 3. Auflage, Springer-Verlag, 2018.

Lehman, Meir M.: »Programs, Life Cycles, and Laws of Software Evolution«, in: *Proceedings of the IEEE*, Vol. 68, No. 9, September 1980, S. 1060–1076.

Mancuso, Stefano: *Brilliant Green: The Surprising History and Science of Plant Intelligence*, Apple Books 2015, italienisches Original: *Verde brillante: Sensibilità e intelligenza del mondo vegetale*, Giunti Editore S.p.A. Firenze-Milano, 2013.

Miller, George A.: »The cognitive revolution: a historical perspective«, in: *Trends in Cognitive Sciences*, Vol. 7, No.3, March 2003, S. 141–144.

Neunhäuserer, Jörg: *Einführung in die Philosophie der Mathematik*, Berlin, Springer Spektrum, 2019.

Perrow, Charles: *Normale Katastrophen: Die unvermeidbaren Katastrophen der Großtechnik*, Campus Verlag, Frankfurt/New York, 1988.

Rice, Henry Gordon: »Classes of recursively enumerable sets and their decision problems«, in: *Transactions of the American Mathematical Society*, Band 74, 1953, S. 358–366.

Rozenberg, Grzegorz: »Computer Science, Informatics, and Natural Computing – Personal Reflections«, in: S. Barry Cooper/Benedikt Löwe/Andrea Sorbi (Hgg.): *New Computational Paradigms–Changing Conceptions of What is Computable*, Springer Science + Business Media, 2008, S. 373–379.

Shapiro, Fred R.: »The first Bug«, in: *BYTE*, April 1994, S. 308.

Tero, Atsushi, et al.: »Rules for Biologically Inspired Adaptive Network Design«, in: *Science*, vol. 327, 2010, S. 439–442.

Turing, Alan Mathison: »On computable numbers, with an application to the Entscheidungsproblem«, in: *Proc. Lond. Math. Soc.*, Series 2, Vol. 42, 1936, S. 230–265.