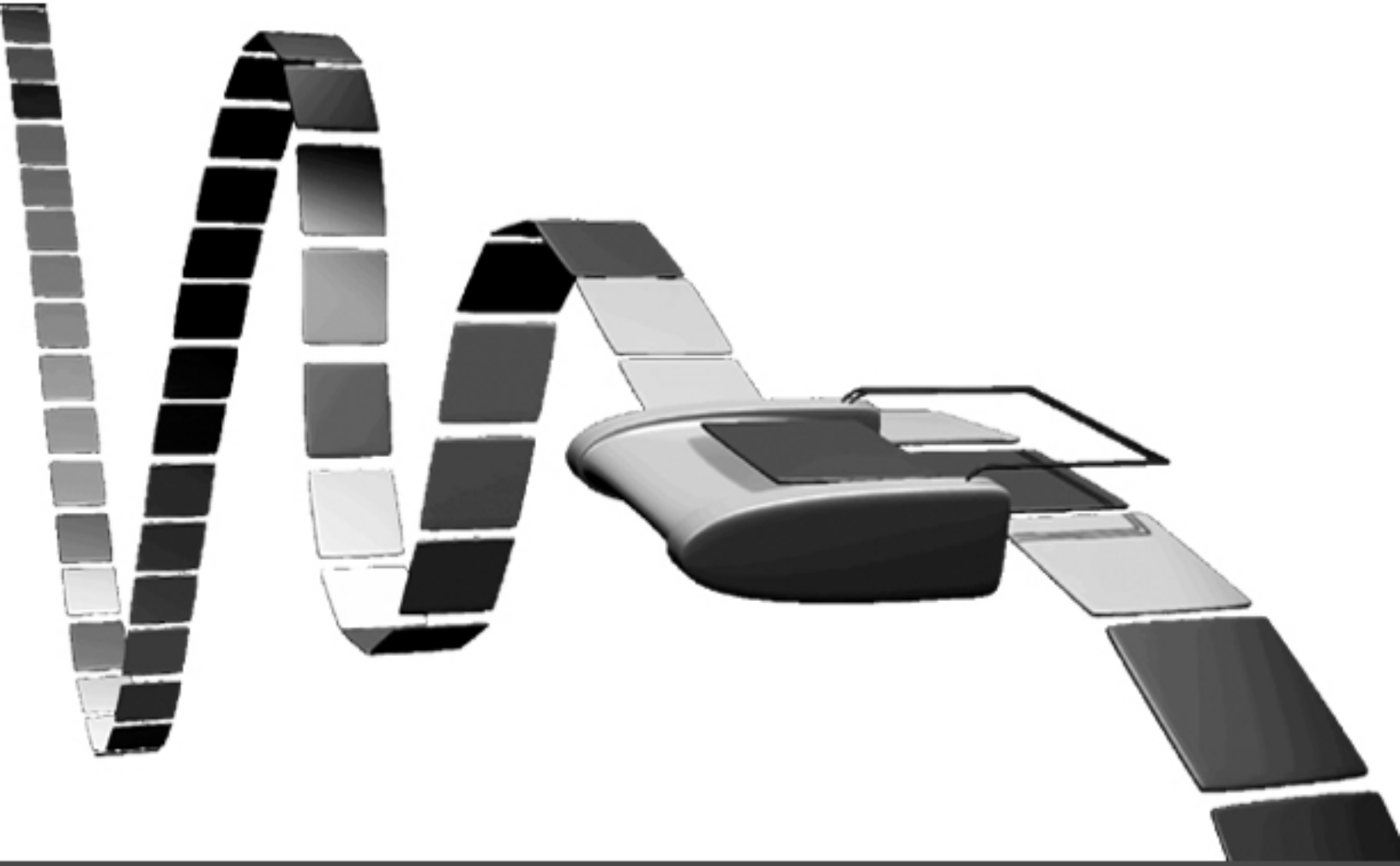
A Theoretical Introduction to Turing Machine



Merlin Forbes

First Edition, 2012

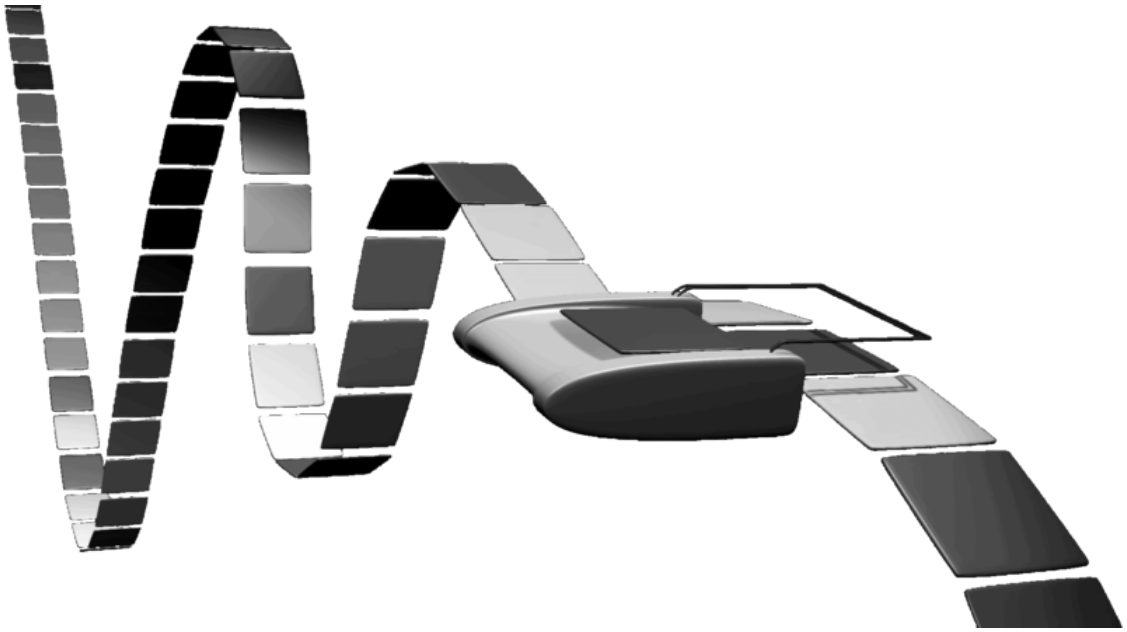
ISBN 978-81-323-1690-9

Published by:
Learning Press
4735/22 Prakashdeep Bldg,
Ansari Road, Darya Ganj,
Delhi - 110002
Email: info@wtbooks.com

Table of Contents

Chapter 1

Introduction to Turing Machine



An artistic representation of a Turing machine (Rules table not represented)

A **Turing machine** is a theoretical device that manipulates symbols on a strip of tape according to a table of rules. Despite its simplicity, a Turing machine can be adapted to simulate the logic of any computer algorithm, and is particularly useful in explaining the functions of a CPU inside a computer.

The "Turing" machine was described by Alan Turing in 1937, who called it an "*a*(utomatic)-machine". Turing machines are not intended as a practical computing technology, but rather as a thought experiment representing a computing machine. They help computer scientists understand the limits of mechanical computation.

Turing gave a succinct definition of the experiment in his 1948 essay, "Intelligent Machinery". Referring to his 1936 publication, Turing wrote that the Turing machine, here called a Logical Computing Machine, consisted of:

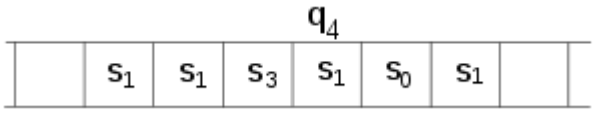
...an infinite memory capacity obtained in the form of an infinite tape marked out into squares, on each of which a symbol could be printed. At any moment there is one symbol in the machine; it is called the scanned symbol. The machine can alter the scanned symbol and its behavior is in part determined by that symbol, but the symbols on the tape elsewhere do not affect the behavior of the machine. However, the tape can be moved back and forth through the machine, this being one of the elementary operations of the machine. Any symbol on the tape may therefore eventually have an innings. (Turing 1948, p. 61)

A Turing machine that is able to simulate any other Turing machine is called a universal Turing machine (**UTM**, or simply a **universal machine**). A more mathematically-oriented definition with a similar "universal" nature was introduced by Alonzo Church, whose work on lambda calculus intertwined with Turing's in a formal theory of computation known as the Church–Turing thesis. The thesis states that Turing machines indeed capture the informal notion of effective method in logic and mathematics, and provide a precise definition of an algorithm or 'mechanical procedure'.

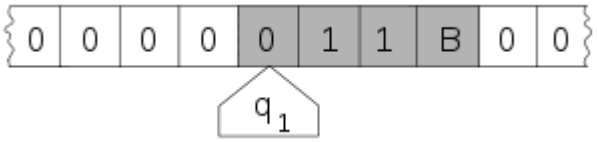
Studying their abstract properties yields many insights into computer science and complexity theory.

Informal description

The Turing machine mathematically models a machine that mechanically operates on a tape on which symbols are written which it can read and write one at a time using a tape head. Operation is fully determined by a finite set of elementary instructions such as "in state 42, if the symbol seen is 0, write a 1; if the symbol seen is 1, shift to the right, and change into state 17; in state 17, if the symbol seen is 0, write a 1 and change to state 6;" etc. In the original article ("On computable numbers, with an application to the Entscheidungsproblem"), Turing imagines not a mechanism, but a person whom he calls the "computer", who executes these deterministic mechanical rules slavishly (or as Turing puts it, "in a desultory manner").



The head is always over a particular square of the tape; only a finite stretch of squares is given. The instruction to be performed (q_4) is shown over the scanned square. (Drawing after Kleene (1952) p.375.)



Here, the internal state (q_1) is shown inside the head, and the illustration describes the tape as being infinite and pre-filled with "0", the symbol serving as blank. The system's full state (its *configuration*) consists of the internal state, the contents of the shaded squares including the blank scanned by the head ("11B"), and the position of the head. (Drawing after Minsky (1967) p. 121).

More precisely, a Turing machine consists of:

1. **TAPE** which is divided into cells, one next to the other. Each cell contains a symbol from some finite alphabet. The alphabet contains a special *blank* symbol (here written as 'B') and one or more other symbols. The tape is assumed to be arbitrarily extendable to the left and to the right, i.e., the Turing machine is always supplied with as much tape as it needs for its computation. Cells that have not been written to before are assumed to be filled with the blank symbol. In some models the tape has a left end marked with a special symbol; the tape extends or is indefinitely extensible to the right.
2. A **HEAD** that can read and write symbols on the tape and move the tape left and right one (and only one) cell at a time. In some models the head moves and the tape is stationary.
3. A finite **TABLE** ("action table", or *transition function*) of instructions (usually quintuples [5-tuples] : $q_i a_j \rightarrow q_{i1} a_{j1} d_k$, but sometimes 4-tuples) that, given the *state*(q_i) the machine is currently in *and* the *symbol*(a_j) it is reading on the tape (symbol currently under HEAD) tells the machine to do the following in sequence (for the 5-tuple models):
    - Either erase or write a symbol (instead of a_j written a_{j1}), *and then*
    - Move the head (which is described by d_k and can have values: 'L' for one step left *or* 'R' for one step right *or* 'N' for staying in the same place), *and then*
    - Assume the same or a *new state* as prescribed (go to state q_{i1}).

    In the 4-tuple models, erase or write a symbol (a_{j1}) and move the head left or right (d_k) are specified as separate instructions. Specifically, the TABLE tells the machine to (ia) erase or write a symbol *or* (ib) move the head left or right, *and then* (ii) assume the same or a new state as prescribed, but not both actions (ia) and (ib) in the same instruction. In some models, if there is no entry in the table for the current combination of symbol and state then the machine will halt; other models require all entries to be filled.

4. A **STATE REGISTER** that stores the state of the Turing table, one of finitely many. There is one special *start state* with which the state register is initialized. These states, writes Turing, replace the "state of mind" a person performing computations would ordinarily be in.

Note that every part of the machine—its state and symbol-collections—and its actions—printing, erasing and tape motion—is *finite*, *discrete* and *distinguishable*; it is the potentially unlimited amount of tape that gives it an unbounded amount of storage space.

Formal definition

Hopcroft and Ullman (1979, p. 148) formally define a (one-tape) Turing machine as a 7-tuple $M = \langle Q, \Gamma, b, \Sigma, \delta, q_0, F \rangle$ where

- Q is a finite, non-empty set of *states*
- Γ is a finite, non-empty set of the *tape alphabet/symbols*
- $b \in \Gamma$ is the *blank symbol* (the only symbol allowed to occur on the tape infinitely often at any step during the computation)
- $\Sigma \subseteq \Gamma \setminus \{b\}$ is the set of *input symbols*
- $q_0 \in Q$ is the *initial state*
- $F \subseteq Q$ is the set of *final* or *accepting states*.
- $\delta : Q \setminus F \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is a partial function called the *transition function*, where L is left shift, R is right shift. (A relatively uncommon variant allows "no shift", say N, as a third element of the latter set.)

Anything that operates according to these specifications is a Turing machine.

The 7-tuple for the 3-state busy beaver looks like this :

Q = { **A**, **B**, **C**, **HALT** }
Γ = { 0, 1 }
b = 0 = "blank"
Σ = { 1 }
δ = see state-table below
q_0 = **A** = initial state
F = the one element set of final states {**HALT**}

Initially all tape cells are marked with 0.

State table for 3 state, 2 symbol busy beaver

Tape symbol	Current state A			Current state B			Current state C		
	Write symbol	Move tape	Next state	Write symbol	Move tape	Next state	Write symbol	Move tape	Next state
0	1	R	**B**	1	L	**A**	1	L	**B**
1	1	L	**C**	1	R	**B**	1	R	**HALT**

Additional details required to visualize or implement Turing machines

In the words of van Emde Boas (1990), p. 6: "The set-theoretical object [his formal seven-tuple description similar to the above] provides only partial information on how the machine will behave and what its computations will look like."

For instance,

- There will need to be some decision on what the symbols actually look like, and a failproof way of reading and writing symbols indefinitely.
- The shift left and shift right operations may shift the tape head across the tape, but when actually building a Turing machine it is more practical to make the tape slide back and forth under the head instead.
- The tape can be finite, and automatically extended with blanks as needed (which is closest to the mathematical definition), but it is more common to think of it as stretching infinitely at both ends and being pre-filled with blanks except on the explicitly given finite fragment the tape head is on. (This is, of course, not implementable in practice.) The tape *cannot* be fixed in length, since that would not correspond to the given definition and would seriously limit the range of computations the machine can perform to those of a linear bounded automaton.

Alternative definitions

Definitions in literature sometimes differ slightly, to make arguments or proofs easier or clearer, but this is always done in such a way that the resulting machine has the same computational power. For example, changing the set $\{L,R\}$ to $\{L,R,N\}$, where N ("None" or "No-operation") would allow the machine to stay on the same tape cell instead of moving left or right, does not increase the machine's computational power.

The most common convention represents each "Turing instruction" in a "Turing table" by one of nine 5-tuples, per the convention of Turing/Davis (Turing (1936) in *Undecidable*, p. 126-127 and Davis (2000) p. 152):

> (definition 1): $\mathbf{(q_i, S_j, S_k/E/N, L/R/N, q_m)}$
> **(** current state $\mathbf{q_i}$ **,** symbol scanned $\mathbf{S_j}$ **,** print symbol $\mathbf{S_k}$/erase **E**/none **N ,** move_tape_one_square left **L**/right **R**/none **N ,** new state $\mathbf{q_m}$ **)**

Other authors (Minsky (1967) p. 119, Hopcroft and Ullman (1979) p. 158, Stone (1972) p. 9) adopt a different convention, with new state $\mathbf{q_m}$ listed immediately after the scanned symbol S_j:

> (definition 2): $\mathbf{(q_i, S_j, q_m, S_k/E/N, L/R/N)}$
> **(** current state $\mathbf{q_i}$ **,** symbol scanned $\mathbf{S_j}$ **,** new state $\mathbf{q_m}$ **,** print symbol $\mathbf{S_k}$/erase **E**/none **N ,** move_tape_one_square left **L**/right **R**/none **N)**

Example: state table for the 3-state 2-symbol busy beaver reduced to 5-tuples

Current state	Scanned symbol	Print symbol	Move tape	Final (i.e. next) state	5-tuples
A	0	1	R	**B**	(**A**, 0, 1, R, **B**)
A	1	1	L	**C**	(**A**, 1, 1, L, **C**)
B	0	1	L	**A**	(**B**, 0, 1, L, **A**)
B	1	1	R	**B**	(**B**, 1, 1, R, **B**)
C	0	1	L	**B**	(**C**, 0, 1, L, **B**)
C	1	1	N	**H**	(**C**, 1, 1, N, **H**)

In the following table, Turing's original model allowed only the first three lines that he called N1, N2, N3 (cf Turing in *Undecidable*, p. 126). He allowed for erasure of the "scanned square" by naming a 0th symbol S_0 = "erase" or "blank", etc. However, he did not allow for non-printing, so every instruction-line includes "print symbol S_k" or "erase" (cf footnote 12 in Post (1947), *Undecidable* p. 300). The abbreviations are Turing's (*Undecidable* p. 119). Subsequent to Turing's original paper in 1936–1937, machine-models have allowed all nine possible types of five-tuples:

	Current m-configuration (Turing state)	Tape symbol	Print-operation	Tape-motion	Final m-configuration (Turing state)	5-tuple	5-tuple comments	4-tuple
N1	q_i	S_j	Print(S_k)	Left L	q_m	(q_i, S_j, S_k, L, q_m)	"blank" = S_0, 1=S_1, etc.	
N2	q_i	S_j	Print(S_k)	Right R	q_m	(q_i, S_j, S_k, R, q_m)	"blank" = S_0, 1=S_1, etc.	
N3	q_i	S_j	Print(S_k)	None N	q_m	(q_i, S_j, S_k, N, q_m)	"blank" = S_0, 1=S_1, etc.	(q_i, S_j, S_k, q_m)
4	q_i	S_j	None N	Left L	q_m	(q_i, S_j, N, L, q_m)		(q_i, S_j, L, q_m)
5	q_i	S_j	None N	Right R	q_m	(q_i, S_j,		(q_i, S_j,

						N, R, q_m)		R, q_m)
6	q_i	S_j	None N	None N	q_m	(q_i, S_j, N, N, q_m)	Direct "jump"	(q_i, S_j, N, q_m)
7	q_i	S_j	Erase	Left L	q_m	(q_i, S_j, E, L, q_m)		
8	q_i	S_j	Erase	Right R	q_m	(q_i, S_j, E, R, q_m)		
9	q_i	S_j	Erase	None N	q_m	(q_i, S_j, E, N, q_m)		(q_i, S_j, E, q_m)

Any Turing table (list of instructions) can be constructed from the above nine 5-tuples. For technical reasons, the three non-printing or "N" instructions (4, 5, 6) can usually be dispensed with.

Less frequently the use of 4-tuples are encountered: these represent a further atomization of the Turing instructions (cf Post (1947), Boolos & Jeffrey (1974, 1999), Davis-Sigal-Weyuker (1994)).

The "state"

The word "state" used in context of Turing machines can be a source of confusion, as it can mean two things. Most commentators after Turing have used "state" to mean the name/designator of the current instruction to be performed—i.e. the contents of the state register. But Turing (1936) made a strong distinction between a record of what he called the machine's "m-configuration", (its internal state) and the machine's (or person's) "state of progress" through the computation - the current state of the total system. What Turing called "the state formula" includes both the current instruction and *all* the symbols on the tape:

Thus the state of progress of the computation at any stage is completely determined by the note of instructions and the symbols on the tape. That is, the **state of the system** may be described by a single expression (sequence of symbols) consisting of the symbols on the tape followed by Δ (which we suppose not to appear elsewhere) and then by the note of instructions. This expression is called the 'state formula'.
—*Undecidable*, p.139–140, emphasis added

Earlier in his paper Turing carried this even further: he gives an example where he places a symbol of the current "m-configuration"—the instruction's label—beneath the scanned square, together with all the symbols on the tape (*Undecidable*, p. 121); this he calls "the

complete configuration" (*Undecidable*, p. 118). To print the "complete configuration" on one line he places the state-label/m-configuration to the *left* of the scanned symbol.

A variant of this is seen in Kleene (1952) where Kleene shows how to write the Gödel number of a machine's "situation": he places the "m-configuration" symbol q_4 over the scanned square in roughly the center of the 6 non-blank squares on the tape and puts it to the *right* of the scanned square. But Kleene refers to "q_4" itself as "the machine state" (Kleene, p. 374-375). Hopcroft and Ullman call this composite the "instantaneous description" and follow the Turing convention of putting the "current state" (instruction-label, m-configuration) to the *left* of the scanned symbol (p. 149).

Example: total state of 3-state 2-symbol busy beaver after 3 "moves" (taken from example "run" in the figure below):

1**A**1

This means: after three moves the tape has ... 000110000 ... on it, the head is scanning the right-most 1, and the state is **A**. Blanks (in this case represented by "0"s) can be part of the total state as shown here: **B**01 ; the tape has a single 1 on it, but the head is scanning the 0 ("blank") to its left and the state is **B**.

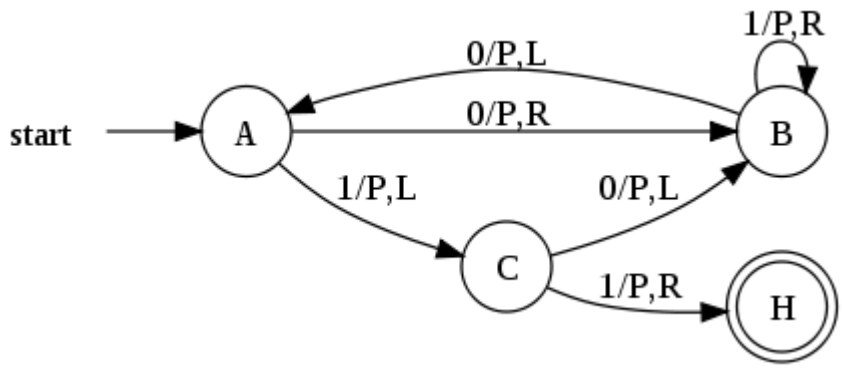
"State" in the context of Turing machines should be clarified as to which is being described: (*i*) the current instruction, or (*ii*) the list of symbols on the tape together with the current instruction, or (*iii*) the list of symbols on the tape together with the current instruction placed to the left of the scanned symbol or to the right of the scanned symbol.

Turing's biographer Andrew Hodges (1983: 107) has noted and discussed this confusion.

Turing machine "state" diagrams

The table for the 3-state busy beaver ("P" = print/write a "1")

Tape symbol	**Current state A**			**Current state B**			**Current state C**		
	Write symbol	Move tape	Next state	Write symbol	Move tape	Next state	Write symbol	Move tape	Next state
0	P	R	**B**	P	L	**A**	P	L	**B**
1	P	L	**C**	P	R	**B**	P	R	**HALT**

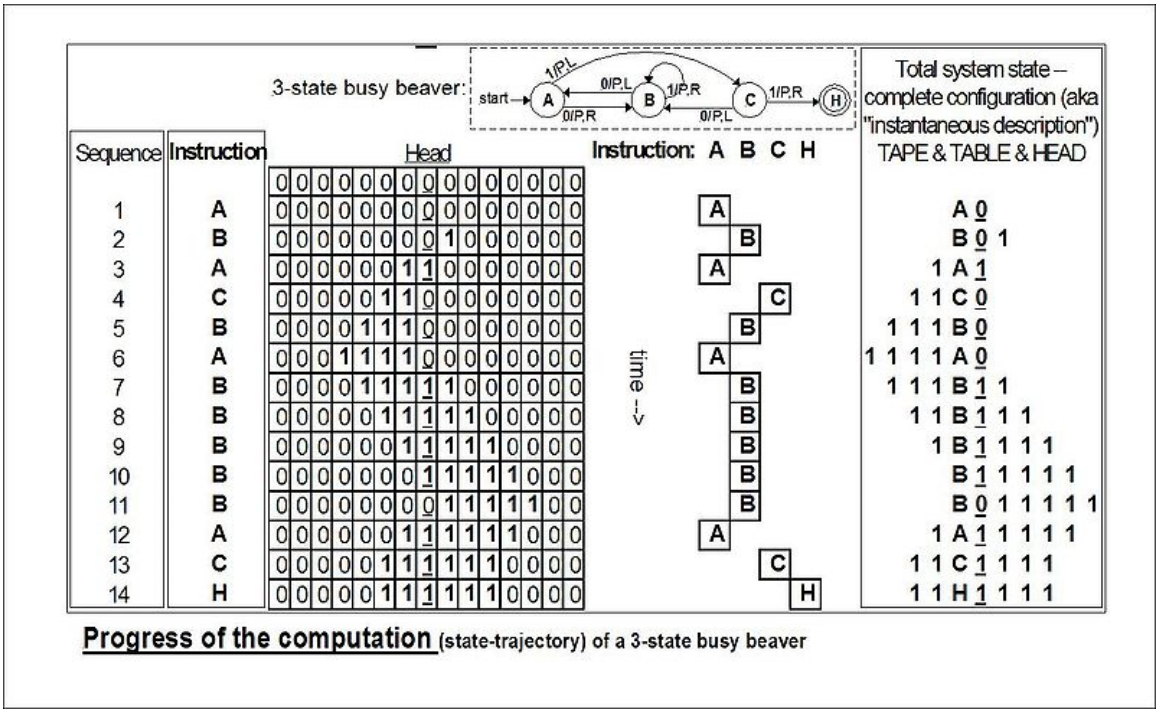


The "3-state busy beaver" Turing Machine in a finite state representation. Each circle represents a "state" of the TABLE—an "m-configuration" or "instruction". "Direction" of a state *transition* is shown by an arrow. The label (e.g.. **0/P,R**) near the outgoing state (at the "tail" of the arrow) specifies the scanned symbol that causes a particular transition (e.g. **0**) followed by a slash /, followed by the subsequent "behaviors" of the machine, e.g. "**P P**rint" then move tape "**R R**ight". No general accepted format exists. The convention shown is after McClusky (1965), Booth (1965), Hill and Peterson (1974).

To the right: the above TABLE as expressed as a "state transition" diagram.

Usually large TABLES are better left as tables (Booth, p. 74). They are more readily simulated by computer in tabular form (Booth, p. 74). However, certain concepts—e.g. machines with "reset" states and machines with repeating patterns (cf Hill and Peterson p. 244ff)—can be more readily seen when viewed as a drawing.

Whether a drawing represents an improvement on its TABLE must be decided by the reader for the particular context.



Progress of the computation (state-trajectory) of a 3-state busy beaver

The evolution of the busy-beaver's computation starts at the top and proceeds to the bottom.

The reader should again be cautioned that such diagrams represent a snapshot of their TABLE frozen in time, *not* the course ("trajectory") of a computation *through* time and/or space. While every time the busy beaver machine "runs" it will always follow the same state-trajectory, this is not true for the "copy" machine that can be provided with variable input "parameters".

The diagram "Progress of the computation" shows the 3-state busy beaver's "state" (instruction) progress through its computation from start to finish. On the far right is the Turing "complete configuration" (Kleene "situation", Hopcroft-Ullman "instantaneous description") at each step. If the machine were to be stopped and cleared to blank both the "state register" and entire tape, these "configurations" could be used to rekindle a computation anywhere in its progress (cf Turing (1936) *Undecidable* pp. 139–140).

Models equivalent to the Turing machine model

Many machines that might be thought to have more computational capability than a simple universal Turing machine can be shown to have no more power (Hopcroft and Ullman p. 159, cf Minsky (1967)). They might compute faster, perhaps, or use less memory, or their instruction set might be smaller, but they cannot compute more powerfully (i.e. more mathematical functions). (Recall that the Church–Turing thesis *hypothesizes* this to be true for any kind of machine: that anything that can be "computed" can be computed by some Turing machine.)

A Turing machine is equivalent to a pushdown automaton that has been made more flexible and concise by relaxing the last-in-first-out requirement of its stack. (Interestingly, this seemingly minor relaxation enables the Turing machine to perform such a wide variety of computations that it can serve as a clearer model for the computational capabilities of all modern computer software.)

At the other extreme, some very simple models turn out to be Turing-equivalent, i.e. to have the same computational power as the Turing machine model.

Common equivalent models are the multi-tape Turing machine, multi-track Turing machine, machines with input and output, and the *non-deterministic* Turing machine (NDTM) as opposed to the *deterministic* Turing machine (DTM) for which the action table has at most one entry for each combination of symbol and state.

Read-only, right-moving Turing Machines are equivalent to NDFA's (as well as DFA's by conversion using the NDFA to DFA conversion algorithm).

For practical and didactical intentions the equivalent register machine can be used as a usual assembly programming language.

Choice c-machines, Oracle o-machines

Early in his paper (1936) Turing makes a distinction between an "automatic machine"—its "motion ... completely determined by the configuration" and a "choice machine":

...whose motion is only partially determined by the configuration ... When such a machine reaches one of these ambiguous configurations, it cannot go on until some arbitrary choice has been made by an external operator. This would be the case if we were using machines to deal with axiomatic systems.
—*Undecidable*, p. 118

Turing (1936) does not elaborate further except in a footnote in which he describes how to use an a-machine to "find all the provable formulae of the [Hilbert] calculus" rather than use a choice machine. He "suppose[s] that the choices are always between two possibilities 0 and 1. Each proof will then be determined by a sequence of choices i_1, i_2, ..., i_n ($i_1 = 0$ or 1, $i_2 = 0$ or 1, ..., $i_n = 0$ or 1), and hence the number $2^n + i_1 2^{n-1} + i_2 2^{n-2} + ... +i_n$ completely determines the proof. The automatic machine carries out successively proof 1, proof 2, proof 3, ..." (Footnote ‡, *Undecidable*, p. 138)

This is indeed the technique by which a deterministic (i.e. a-) Turing machine can be used to mimic the action of a nondeterministic Turing machine; Turing solved the matter in a footnote and appears to dismiss it from further consideration.

An oracle machine or o-machine is a Turing a-machine that pauses its computation at state "**o**" while, to complete its calculation, it "awaits the decision" of "the oracle"—an

unspecified entity "apart from saying that it cannot be a machine" (Turing (1939), Undecidable p. 166–168). The concept is now actively used by mathematicians.

Comparison with real machines

It is often said that Turing machines, unlike simpler automata, are as powerful as real machines, and are able to execute any operation that a real program can. What is missed in this statement is that, because a real machine can only be in finitely many *configurations*, in fact this "real machine" is nothing but a linear bounded automaton. On the other hand, Turing machines are equivalent to machines that have an unlimited amount of storage space for their computations. In fact, Turing machines are not intended to model computers, but rather they are intended to model computation itself; historically, computers, which compute only on their (fixed) internal storage, were developed only later.

There are a number of ways to explain why Turing machines are useful models of real computers:

1. Anything a real computer can compute, a Turing machine can also compute. For example: "A Turing machine can simulate any type of subroutine found in programming languages, including recursive procedures and any of the known parameter-passing mechanisms" (Hopcroft and Ullman p. 157). A large enough FSA can also model any real computer, disregarding IO. Thus, a statement about the limitations of Turing machines will also apply to real computers.
2. The difference lies only with the ability of a Turing machine to manipulate an unbounded amount of data. However, given a finite amount of time, a Turing machine (like a real machine) can only manipulate a finite amount of data.
3. Like a Turing machine, a real machine can have its storage space enlarged as needed, by acquiring more disks or other storage media. If the supply of these runs short, the Turing machine may become less useful as a model. But the fact is that neither Turing machines nor real machines need astronomical amounts of storage space in order to perform useful computation. The processing time required is usually much more of a problem.
4. Descriptions of real machine programs using simpler abstract models are often much more complex than descriptions using Turing machines. For example, a Turing machine describing an algorithm may have a few hundred states, while the equivalent deterministic finite automaton on a given real machine has quadrillions. This makes the DFA representation infeasible to analyze.
5. Turing machines describe algorithms independent of how much memory they use. There is a limit to the memory possessed by any current machine, but this limit can rise arbitrarily in time. Turing machines allow us to make statements about algorithms which will (theoretically) hold forever, regardless of advances in *conventional* computing machine architecture.
6. Turing machines simplify the statement of algorithms. Algorithms running on Turing-equivalent abstract machines are usually more general than their counterparts running on real machines, because they have arbitrary-precision data

types available and never have to deal with unexpected conditions (including, but not limited to, running out of memory).

One way in which Turing machines are a poor model for programs is that many real programs, such as operating systems and word processors, are written to receive unbounded input over time, and therefore do not halt. Turing machines do not model such ongoing computation well (but can still model portions of it, such as individual procedures).

Limitations of Turing machines

Computational Complexity Theory

A limitation of Turing Machines is that they do not model the strengths of a particular arrangement well. For instance, modern stored-program computers are actually instances of a more specific form of abstract machine known as the random access stored program machine or RASP machine model. Like the Universal Turing machine the RASP stores its "program" in "memory" external to its finite-state machine's "instructions". Unlike the Universal Turing Machine, the RASP has an infinite number of distinguishable, numbered but unbounded "registers"—memory "cells" that can contain any integer (cf. Elgot and Robinson (1964), Hartmanis (1971), and in particular Cook-Rechow (1973); references at random access machine). The RASP's finite-state machine is equipped with the capability for indirect addressing (e.g. the contents of one register can be used as an address to specify another register); thus the RASP's "program" can address any register in the register-sequence. The upshot of this distinction is that there are computational optimizations that can be performed based on the memory indices, which are not possible in a general Turing Machine; thus when Turing Machines are used as the basis for bounding running times, a 'false lower bound' can be proven on certain algorithms' running times (due to the false simplifying assumption of a Turing Machine). An example of this is binary search, an algorithm that can be shown to perform more quickly when using the RASP model of computation rather than the Turing machine model.

Concurrency

Another limitation of Turing machines is that they do not model concurrency well. For example, there is a bound on the size of integer that can be computed by an always-halting nondeterministic Turing Machine starting on a blank tape. By contrast, there are always-halting concurrent systems with no inputs that can compute an integer of unbounded size. (A process can be created with local storage that is initialized with a count of 0 that concurrently sends itself both a stop and a go message. When it receives a go message, it increments its count by 1 and sends itself a go message. When it receives a stop message, it stops with an unbounded number in its local storage.)

History

They were described in 1936 by Alan Turing.

Historical background: computational machinery

Robin Gandy (1919–1995)—a student of Alan Turing (1912–1954) and his life-long friend—traces the lineage of the notion of "calculating machine" back to Babbage (circa 1834) and actually proposes "Babbage's Thesis":

That the whole of development and operations of analysis are now capable of being executed by machinery.
—(italics in Babbage as cited by Gandy, p. 54)

Gandy's analysis of Babbage's Analytical Engine describes the following five operations (cf p. 52–53):

1. The arithmetic functions +, −, × where − indicates "proper" subtraction $x - y = 0$ if $y \geq x$
2. Any sequence of operations is an operation
3. Iteration of an operation (repeating n times an operation P)
4. Conditional iteration (repeating n times an operation P conditional on the "success" of test T)
5. Conditional transfer (i.e. conditional "goto")

Gandy states that "the functions which can be calculated by (1), (2), and (4) are precisely those which are Turing computable." (p. 53). He cites other proposals for "universal calculating machines" included those of Percy Ludgate (1909), Leonardo Torres y Quevedo (1914), M. d'Ocagne (1922), Louis Couffignal (1933), Vannevar Bush (1936), Howard Aiken (1937). However:

... the emphasis is on programming a fixed iterable sequence of arithmetical operations. The fundamental importance of conditional iteration and conditional transfer for a general theory of calculating machines is not recognized ...
—Gandy p. 55

The Entscheidungsproblem (the "decision problem"): Hilbert's tenth question of 1900

With regards to Hilbert's problems posed by the famous mathematician David Hilbert in 1900, an aspect of problem #10 had been floating about for almost 30 years before it was framed precisely. Hilbert's original expression for #10 is as follows:

10. Determination of the solvability of a Diophantine equation. Given a Diophantine equation with any number of unknown quantities and with rational integral coefficients: To devise a process according to which it can be determined in a finite number of operations whether the equation is solvable in rational integers.

The Entscheidungsproblem [decision problem for first-order logic] is solved when we know a procedure that allows for any given logical expression to decide by finitely many

operations its validity or satisfiability ... The Entscheidungsproblem must be considered the main problem of mathematical logic.

—quoted, with this translation and the original German, in Dershowitz and Gurevich, 2008

By 1922, this notion of "Entscheidungsproblem" had developed a bit, and H. Behmann stated that

... most general form of the Entscheidungsproblem [is] as follows:

> A quite definite generally applicable prescription is required which will allow one to decide in a finite number of steps the truth or falsity of a given purely logical assertion ...

—Gandy p. 57, quoting Behmann

Behmann remarks that ... the general problem is equivalent to the problem of deciding which mathematical propositions are true.

—*ibid.*

If one were able to solve the Entscheidungsproblem then one would have a "procedure for solving many (or even all) mathematical problems".

—*ibid.*, p. 92

By the 1928 international congress of mathematicians Hilbert "made his questions quite precise. First, was mathematics *complete* ... Second, was mathematics *consistent* ... And thirdly, was mathematics *decidable*?" (Hodges p. 91, Hawking p. 1121). The first two questions were answered in 1930 by Kurt Gödel at the very same meeting where Hilbert delivered his retirement speech (much to the chagrin of Hilbert); the third—the Entscheidungsproblem—had to wait until the mid-1930s.

The problem was that an answer first required a precise definition of "*definite general applicable prescription*", which Alonzo Church would come to call "effective calculability", and in 1928 no such definition existed. But over the next 6–7 years Emil Post developed his definition of a worker moving from room to room writing and erasing marks per a list of instructions (Post 1936), as did Princeton professor Church and his two students Stephen Kleene and J. B. Rosser by use of Church's lambda-calculus and Gödel's recursion theory (1934). Church's paper (published 15 April 1936) showed that the Entscheidungsproblem was indeed "undecidable" and beat Turing to the punch by almost a year (Turing's paper submitted 28 May 1936, published January 1937). In the meantime, Emil Post submitted a brief paper in the fall of 1936, so Turing at least had priority over Post. While Church refereed Turing's paper, Turing had time to study Church's paper and add an Appendix where he sketched a proof that Church's lambda-calculus and his machines would compute the same functions.

But what Church had done was something rather different, and in a certain sense weaker. ... the Turing construction was more direct, and provided an argument from first principles, closing the gap in Church's demonstration.

—Hodges p. 112

And Post had only proposed a definition of calculability and criticized Church's "definition", but had proved nothing.

Alan Turing's a- (automatic-)machine

In the spring of 1935 Turing as a young Master's student at King's College Cambridge, UK, took on the challenge; he had been stimulated by the lectures of the logician M. H. A. Newman "and learned from them of Gödel's work and the Entscheidungsproblem ... Newman used the word 'mechanical' ... In his obituary of Turing 1955 Newman writes:

To the question 'what is a "mechanical" process?' Turing returned the characteristic answer 'Something that can be done by a machine' and he embarked on the highly congenial task of analysing the general notion of a computing machine.
—Gandy, p. 74

Gandy states that:

I suppose, but do not know, that Turing, right from the start of his work, had as his goal a proof of the undecidability of the Entscheidungsproblem. He told me that the 'main idea' of the paper came to him when he was lying in Grantchester meadows in the summer of 1935. The 'main idea' might have either been his analysis of computation or his realization that there was a universal machine, and so a diagonal argument to prove unsolvability.
—*ibid.*, p. 76

While Gandy believed that Newman's statement above is "misleading", this opinion is not shared by all. Turing had a life-long interest in machines: "Alan had dreamt of inventing typewriters as a boy; [his mother] Mrs. Turing had a typewriter; and he could well have begun by asking himself what was meant by calling a typewriter 'mechanical'" (Hodges p. 96). While at Princeton pursuing his PhD, Turing built a Boolean-logic multiplier (see below). His PhD thesis, titled "Systems of Logic Based on Ordinals", contains the following definition of "a computable function":

It was stated above that 'a function is effectively calculable if its values can be found by some purely mechanical process'. We may take this statement literally, understanding by a purely mechanical process one which could be carried out by a machine. It is possible to give a mathematical description, in a certain normal form, of the structures of these machines. The development of these ideas leads to the author's definition of a computable function, and to an identification of computability with effective calculability. It is not difficult, though somewhat laborious, to prove that these three definitions [the 3rd is the λ-calculus] are equivalent.
—Turing (1939) in *The Undecidable*, p. 160

When Turing returned to the UK he ultimately became jointly responsible for breaking the German secret codes created by encryption machines called "The Enigma"; he also became involved in the design of the ACE (Automatic Computing Engine), "[Turing's]

ACE proposal was effectively self-contained, and its roots lay not in the EDVAC [the USA's initiative], but in his own universal machine" (Hodges p. 318). Arguments still continue concerning the origin and nature of what has been named by Kleene (1952) Turing's Thesis. But what Turing *did prove* with his computational-machine model appears in his paper *On Computable Numbers, With an Application to the Entscheidungsproblem* (1937):

[that] the Hilbert Entscheidungsproblem can have no solution ... I propose, therefore to show that there can be no general process for determining whether a given formula U of the functional calculus K is provable, i.e. that there can be no machine which, supplied with any one U of these formulae, will eventually say whether U is provable.
—from Turing's paper as reprinted in *The Undecidable*, p. 145

Turing's example (his second proof): If one is to ask for a general procedure to tell us: "Does this machine ever print 0", the question is "undecidable".

1937–1970: The "digital computer", the birth of "computer science"

In 1937, while at Princeton working on his PhD thesis, Turing built a digital (Boolean-logic) multiplier from scratch, making his own electromechanical relays (Hodges p. 138). "Alan's task was to embody the logical design of a Turing machine in a network of relay-operated switches ..." (Hodges p. 138). While Turing might have been just curious and experimenting, quite-earnest work in the same direction was going in Germany (Konrad Zuse (1938)), and in the United States (Howard Aiken) and George Stibitz (1937); the fruits of their labors were used by the Axis and Allied military in World War II (cf Hodges p. 298–299). In the early to mid-1950s Hao Wang and Marvin Minsky reduced the Turing machine to a simpler form (a precursor to the Post-Turing machine of Martin Davis); simultaneously European researchers were reducing the new-fangled electronic computer to a computer-like theoretical object equivalent to what was now being called a "Turing machine". In the late 1950s and early 1960s, the coincidentally-parallel developments of Melzak and Lambek (1961), Minsky (1961), and Shepherdson and Sturgis (1961) carried the European work further and reduced the Turing machine to a more friendly, computer-like abstract model called the counter machine; Elgot and Robinson (1964), Hartmanis (1971), Cook and Reckhow (1973) carried this work even further with the register machine and random access machine models—but basically all are just multi-tape Turing machines with an arithmetic-like instruction set.

1970–present: the Turing machine as a model of computation

Today the counter, register and random-access machines and their sire the Turing machine continue to be the models of choice for theorists investigating questions in the theory of computation. In particular, computational complexity theory makes use of the Turing machine:

Depending on the objects one likes to manipulate in the computations (numbers like nonnegative integers or alphanumeric strings), two models have obtained a dominant position in machine-based complexity theory:
the off-line multitape Turing machine..., which represents the standard model for string-oriented computation, and
the *random access machine (RAM)* as introduced by Cook and Reckhow ..., which models the idealized Von Neumann style computer.
—van Emde Boas 1990:4
Only in the related area of analysis of algorithms this role is taken over by the RAM model.
—van Emde Boas 1990:16

Kantorovitz (2005), was the first to show the most simple obvious representation of Turing Machines published academically which unifies Turing Machines with mathematical analysis and analog computers.

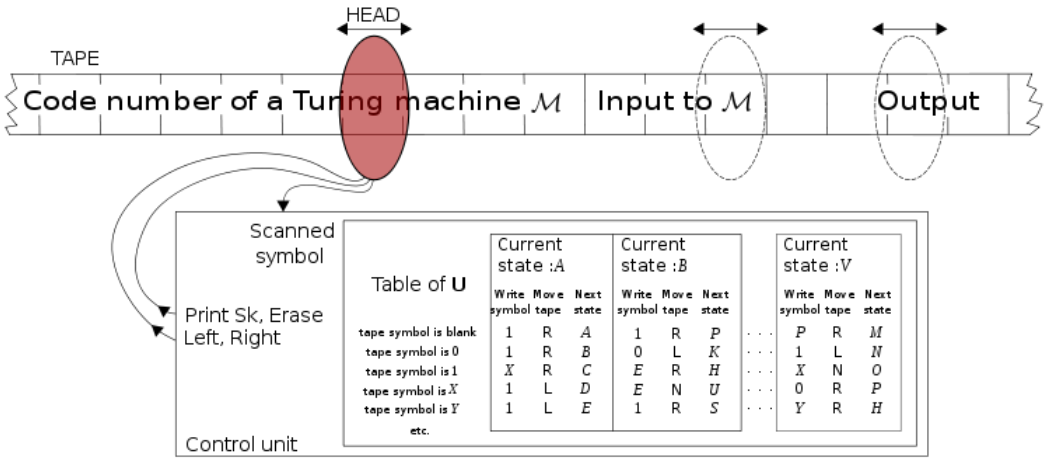
Chapter 2

Universal Turing Machine

In computer science, a **universal Turing machine** is a Turing machine that can simulate an arbitrary Turing machine on arbitrary input. The universal machine essentially achieves this by reading both the description of machine to be simulated as well as the input thereof from its own tape. Alan Turing introduced this machine in 1936–1937. This model is considered by some (for example, Martin Davis (2000)) to be the origin of the stored program computer—used by John von Neumann (1946) for the "Electronic Computing Instrument" that now bears von Neumann's name: the von Neumann architecture. It is also known as **universal computing machine**, **universal machine**, **machine U**, **U**.

In terms of computational complexity, a multi-tape universal Turing machine need only be slower by logarithmic factor compared to the machines it simulates.

Introduction



Every Turing machine computes a certain fixed partial computable function from the input strings over its alphabet. In that sense it behaves like a computer with a fixed program. However, we can encode the action table of any Turing machine in a string. Thus we can construct a Turing machine that expects on its tape a string describing an action table followed by a string describing the input tape, and computes the tape that the

encoded Turing machine would have computed. Turing described such a construction in complete detail in his 1936 paper:

"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 an action table] of some computing machine **M**, then **U** will compute the same sequence as **M**."

Stored-program computer

Davis makes a persuasive argument that Turing's conception of what is now known as "the stored-program computer", of placing the "action table" -- the instructions for the machine—in the same "memory" as the input data, strongly influenced John von Neumann's conception of the first discrete-symbol (as opposed to analog) computer—the EDVAC. Davis quotes *Time* magazine to this effect, that "everyone who taps at a keyboard... is working on an incarnation of a Turing machine," and that "John von Neumann [built] on the work of Alan Turing" (Davis 2000:193 quoting *Time* magazine of 29 March 1999).

Davis makes a case that Turing's Automatic Computing Engine (ACE) computer "anticipated" the notions of microprogramming (microcode) and RISC processors (Davis 2000:188). Knuth cites Turing's work on the ACE computer as designing "hardware to facilitate subroutine linkeage" (Knuth 1973:225); Davis also references this work as Turing's use of a hardware "stack" (Davis 2000:237 footnote 18).

As the Turing Machine was encouraging the construction of computers, the UTM was encouraging the development of the fledgling computer sciences. An early, if not the very first, assembler was proposed "by a young hot-shot programmer" for the EDVAC (Davis 2000:192). Von Neumann's "first serious program ... [was] to simply sort data efficiently" (Davis 2000:184). Knuth observes that the subroutine return embedded in the program itself rather than in special registers is attributable to von Neumann and Goldstine. Knuth furthermore states that

"The first interpretive routine may be said to be the "Universal Turing Machine" ... Interpretive routines in the conventional sense were mentioned by John Mauchly in his lectures at the Moore School in 1946 ... Turing took part in this development also; interpretive systems for the Pilot ACE computer were written under his direction" (Knuth 1973:226).

Davis briefly mentions operating systems and compilers as outcomes of the notion of program-as-data (Davis 2000:185).

Some, however, might raise issues with this assessment. At the time (mid-1940s to mid-1950s) a relatively small cadre of researchers were intimately involved with the

architecture of the new "digital computers". Hao Wang (1954), a young researcher at this time, made the following observation:

Turing's theory of computable functions antedated but has not much influenced the extensive actual construction of digital computers. These two aspects of theory and practice have been developed almost entirely independently of each other. The main reason is undoubtedly that logicians are interested in questions radically different from those with which the applied mathematicians and electrical engineers are primarily concerned. It cannot, however, fail to strike one as rather strange that often the same concepts are expressed by very different terms in the two developments." (Wang 1954, 1957:63)

Wang hoped that his paper would "connect the two approaches." Indeed, Minsky confirms this: "that the first formulation of Turing-machine theory in computer-like models appears in Wang (1957)" (Minsky 1967:200). Minsky goes on to demonstrate Turing equivalence of a counter machine.

With respect to the reduction of computers to simple Turing equivalent models (and vice versa), Minsky's designation of Wang as having made "the first formulation" is open to debate. While both Minsky's paper of 1961 and Wang's paper of 1957 are cited by Shepherdson and Sturgis (1963), they also cite and summarize in some detail the work of European mathematicians Kaphenst (1959), Ershov (1959), and Péter (1958). The names of mathematicians Hermes (1954, 1955, 1961) and Kaphenst (1959) appear in the bibliographies of both Sheperdson-Sturgis (1963) and Elgot-Robinson (1961). Two other names of importance are Canadian researchers Melzak (1961) and Lambek (1961).

Mathematical theory

With this encoding of action tables as strings it becomes possible in principle for Turing machines to answer questions about the behaviour of other Turing machines. Most of these questions, however, are undecidable, meaning that the function in question cannot be calculated mechanically. For instance, the problem of determining whether any particular Turing machine will halt on a particular input, or on all inputs, known as the Halting problem, was shown to be, in general, undecidable in Turing's original paper. Rice's theorem shows that any non-trivial question about the behaviour or output of a Turing machine is undecidable.

A universal Turing machine can calculate any recursive function, decide any recursive language, and accept any recursively enumerable language. According to the Church-Turing thesis, the problems solvable by a universal Turing machine are exactly those problems solvable by an *algorithm* or an *effective method of computation*, for any reasonable definition of those terms. For these reasons, a universal Turing machine serves as a standard against which to compare computational systems, and a system that can simulate a universal Turing machine is called Turing complete.

An abstract version of the universal Turing machine is the universal function, a computable function which can be used to calculate any other computable function. The utm theorem proves the existence of such a function.

When Alan Turing came up with the idea of a universal machine he had in mind the simplest computing model powerful enough to calculate all possible functions which can be calculated. Claude Shannon first explicitly posed the question of finding the smallest possible universal Turing machine when in 1956 he showed that two symbols were sufficient, so long as enough states were used. Shannon himself proved that it was always possible to exchange states by symbols.

Efficiency

Without loss of generality, the input of Turing machine can be assumed to be in the alphabet {0, 1}; any other finite alphabet can be encoded over {0, 1}. The behavior of a Turing machine M is determined by its transition function. This function can be easily encoded as a string over the alphabet {0, 1} as well. The size of the alphabet of M, the number of tapes it has, and the size of the state space can be deduced from the transition function's table. The distinguished states and symbols can be identified by their position, e.g. the first two states can by convention be the start and stop states. Consequently, every Turing machine can be encoded as a string over the alphabet {0, 1}. Additionally, we convene that every invalid encoding maps to a trivial Turing machine that immediately halts, and that every Turing machine can have an infinite number of encodings by padding the encoding with an arbitrary number of (say) 1's at the end, just like comments work in a programming language. It should be no surprise that we can achieve this encoding given the existence of a Gödel number and computational equivalence between Turing machines and μ-recursive functions. Similarly, our construction associates to every binary string α, a Turing machine M_α.

Starting from the above encoding, in 1966 F. C. Hennie and R. E. Stearns showed that given a Turing machine M_α that halts on input x within N steps, then there exists a multi-tape universal Turing machine that halts on inputs α, x (given on different tapes) in $CN \log N$, where C is a machine-specific constant that does not depend on the length of the input x, but does depend on M's alphabet size, number of tapes, and number of states. Effectively this is a $O(N \log N)$ simulation.

Smallest machines

After some time, the smallest known universal Turing machine was due to Marvin Minsky who in 1962 discovered a 7-state 4-symbol universal Turing machine using 2-tag systems. Applying Shannon's result to Minsky's UTM upon conversion to a 2-symbol machine Minsky machine would require 43 states.

Other smaller universal Turing machines have since been found. If we denote by (m,n) the class of UTMs with m states and n symbols the following tuples were found by Yurii Rogozhin in 1996: (24, 2), (10, 3), (7, 4), (5, 5), (4, 6), (3, 10), and (2, 18). In 1985,

Stephen Wolfram conjectured a 2-state 5-symbol universal Turing machine. This conjecture was proved by Matthew Cook working as a research assistant to Stephen Wolfram. The proof was based on emulating the Rule 110 Elementary Cellular Automaton. The model had, at the time, the smallest product (2,5)=10 of any known universal Turing machine. According to Wolfram other smaller UTMs should exist and he proposed a 2-state 3-symbol Turing Machine as a candidate. On 24 October 2007, Wolfram announced the Turing equivalence of the system had been proven by Alex Smith -- an undergraduate studying electronic and computer engineering at the University of Birmingham -- responding to a contest established by Wolfram. However, on 29 October 2007 Vaughan Pratt of Stanford University claimed that he discovered a flaw in the proof. Wolfram Research and Smith himself disputed Pratt's interpretation . Pratt's main point was that the same argument that would make Wolfram's 2,3 Turing machine universal would make a Linear Bounded Automaton (LBA) universal. Smith explained that the LBA would need to be restarted at running time to perform a computation, while the 2, 3 Turing machine restarts automatically, therefore the proof does not make an LBA universal as Pratt first thought. Other small (weak/semi-weak) universal Turing machines were found by Watanabe, Rogozhin, Margenstern and more recently Neary and Woods.

Example of universal-machine coding

The following example is taken from Turing (1936).

Turing used seven symbols { A, C, D, R, L, N, ; } to encode each 5-tuple; as described in Turing machine, his 5-tuples are only of types N1, N2, and N3. The number of each "m-configuration" (instruction, state) is represented by "D" followed by a unary string of A's, i.e. "q3" = DAAA. In a similar manner he encodes the symbols blank as "D", the symbol "0" is "DC", the symbol "1" as DCC, etc. The symbols "R", "L", and "N" remain as is.

After encoding each 5-tuple is then "assembled" into a string in order as shown in the following table:

Finally, the codes for all four 5-tuples are strung together into a code started by ";" and separated by ";" i.e.:

;DADDCRDAA**;**DAADDRDAAA**;**DAAADDCCRDAAAA**;**DAAAADDRDA

This code he placed on alternate squares—the "F-squares" -- leaving the "E-squares" (those liable to erasure) empty. The final assembly of the code on the tape for the U-machine consists of placing two special symbols ("e") one after the other, then the code separated out on alternate squares, and lastly the double-colon symbol "**::**" (blanks shown here with "." for clarity):

ee.**;**.D.A.D.D.C.R.D.A.A.**;**.D.A.A.D.D.R.D.A.A.A.**;**.D.A.A.A.D.D.C.C.R.D.A.A.A.A.**;**.D.A.A.A.A.D.D.R.D.A.**::**.....

The U-machine's action-table (state-transition table) is responsible for decoding the symbols. Turing's action table keeps track of its place with markers "u", "v", "x", "y", "z" by placing them in "E-squares" to the right of "the marked symbol" -- for example, to mark the current instruction **z** is placed to the right of ";" **x** is keeping the place with respect to the current "m-configuration" DAA. The U-machine's action table will shuttle these symbols around (erasing them and placing them in different locations) as the computation progresses:

> ee.**;** .D.A.D.D.C.R.D.A.A. **;**
> **z**D.A.A**x**D.D.R.D.A.A.A.**;**.D.A.A.A.D.D.C.C.R.D.A.A.A.A.**;**.D.A.A.A.A.D.D.R.D.A.**::**.....

Turing's action-table for his U-machine is very involved.

A number of other commentators (notably Penrose 1989) provide examples of ways to encode instructions for the Universal machine. As does Penrose, most commentators use only binary symbols i.e. only symbols { 0, 1 }, or { blank, mark | }. Penrose goes further and writes out his entire U-machine code (Penrose 1989:71–73). He asserts that it truly is a U-machine code, an enormous number that spans almost 2 full pages of 1's and 0's. For readers interested in simpler encodings for the Post-Turing machine the discussion of Davis in Steen (Steen 1980:251ff) may be useful.

Chapter 3

Turing Machine Equivalents and Turing Machine Examples

Turing machine equivalents

Machines equivalent to the Turing machine model

Turing equivalence:

Many machines that might be thought to have more computational capability than a simple universal Turing machine can be shown to have no more power (Hopcroft and Ullman p. 159, cf Minsky). They might compute faster, perhaps, or use less memory, or their instruction set might be smaller, but they cannot compute more powerfully (i.e. more mathematical functions). (Recall that the Church-Turing thesis *hypothesizes* this to be true: that anything that can be "computed" can be computed by some Turing machine.)

While none of the following models have been shown to have more power than the single-tape, one-way infinite, multi-symbol Turing-machine model, their authors defined and used them to investigate questions and solve problems more easily than they could have if they had stayed with Turing's *a*-machine model.

The sequential-machine models:

All of the following are called "sequential machine models" to distinguish them from "parallel machine models" (van Emde Boas (1990) p. 18).

Tape-based Turing machines

Turing's *a*-machine model: Turing's (1936) a-machine (his name) was left-ended, right-end-infinite. He provided symbols **əə** to mark the left end. Any of finite number of tape symbols were permitted. The instructions (if a universal machine), and the "input" and "out" were written on only on "F-squares", and markers were to appear on "E-squares". In essence he divided his machine into two tapes that always moved together. The

instructions appeared in a tabular form called "5-tuples" and were not executed sequentially.

Single-tape machines with restricted symbols and/or restricted instructions

The following models are single tape Turing machines but restricted with (i) restricted tape symbols { mark, blank }, and/or (ii) sequential, computer-like instructions, and/or (iii) machine-actions fully-atomized.

Post's "Formulation 1" model of computation

Emil Post (1936) in an independent description of a computational process, reduced the symbols allowed to the equivalent binary set of marks on the tape { "mark", "blank"=not_mark }. He changed the notion of "tape" from 1-way infinite to the right to an infinite set of rooms each with a sheet of paper in both directions. He atomized the Turing 5-tuples into 4-tuples—motion instructions separate from print/erase instructions. Although his (1936) model is ambiguous about this, Post's (1947) model did not require sequential instruction execution.

His extremely simple model can emulate any Turing machine, and although his 1936 *Formulation 1* does not use the word "program" or "machine", it is effectively a formulation of a very primitive programmable computer and associated programming language, with the boxes acting as an unbounded bitstring memory, and the set of instructions constituting a program.

Wang machines

In an influential paper, Hao Wang (1954, 1957) reduced Post's "formulation 1" to machines that still use a two-way infinite binary tape, but whose instructions are simpler — being the "atomic" components of Post's instructions — and are by default executed sequentially (like a "computer program"). His stated principal purpose was to offer, as an alternative to Turing's theory, one that "is more economical in the basic operations". His results were "program formulations" of a variety of such machines, including the 5-instruction Wang **W-machine** with the instruction-set

> { SHIFT-LEFT, SHIFT-RIGHT, MARK-SQUARE, ERASE-SQUARE, JUMP-IF-SQUARE-MARKED-to xxx }

and his most-severely reduced 4-instruction Wang B-machine ("B" for "basic") with the instruction-set

> { SHIFT-LEFT, SHIFT-RIGHT, MARK-SQUARE, JUMP-IF-SQUARE-MARKED-to xxx }

which has not even an ERASE-SQUARE instruction.

Many authors later introduced variants of the machines discussed by Wang:

Minsky (1961) evolved Wang's notion with his version of the (multi-tape) "counter machine" model that allowed SHIFT-LEFT and SHIFT-RIGHT motion of the separate heads but no printing at all. In this case the tapes would be left-ended, each end marked with a single "mark" to indicate the end. He was able to reduce this to a single tape, but at the expense of introducing multi-tape-square motion equivalent to multiplication and division rather than the much simpler { SHIFT-LEFT = DECREMENT, SHIFT-RIGHT = INCREMENT }.

Davis, adding an explicit HALT instruction to one of the machines discussed by Wang, used a model with the instruction-set

> { SHIFT-LEFT, SHIFT-RIGHT, ERASE, MARK, JUMP-IF-SQUARE-MARKED-to xxx, JUMP-to xxx, HALT }

and also considered versions with tape-alphabets of size larger than 2.

Böhm's theoretical machine language P"

In keeping with Wang's project to seek a Turing-equivalent theory "economical in the basic operations", and wishing to avoid unconditional jumps, a notable theoretical language is the 4-instruction language P" introduced by Corrado Böhm in 1964 — the first "GOTO-less" imperative "structured programming" language to be proved Turing-complete.

Multi-tape Turing machines

In practical analysis, various types of **multi-tape Turing machines** are often used. Multi-tape machines are similar to single-tape machines, but there is some constant *k* number of independent tapes.

The TABLE has full independent control over all the heads, any of all of which move and print/erase their own tapes (cf Aho-Hopcroft-Ullman 1974 p. 26). Most models have tapes with left ends, right ends unbounded.

This model intuitively seems much more powerful than the single-tape model, but any multi-tape machine, no matter how large the *k*, can be simulated by a single-tape machine using only quadratically more computation time (Papadimitriou 1994, Thrm 2.1). Thus, multi-tape machines cannot calculate any more functions than single-tape machines, and none of the robust complexity classes (such as polynomial time) are affected by a change between single-tape and multi-tape machines.

Two-stack Turing machine

Two-stack Turing machines have a read-only input and two storage tapes. If a head moves left on either tape a blank is printed on that tape, but one symbol from a "library" can be printed.

Formal definition: multi-tape Turing machine

A k-tape Turing machine can be described as a 6-tuple $M = \langle Q, \Gamma, s, b, F, \delta \rangle$ where

- Q is a finite set of states
- Γ is a finite set of the tape alphabet
- $s \in Q$ is the initial state
- $b \in \Gamma$ is the blank symbol
- $F \subseteq Q$ is the set of final or accepting states
- $\delta : Q \times \Gamma^k \to Q \times (\Gamma \times \{L, R, S\})^k$ is a partial function called the transition function, where L is left shift, R is right shift, S is no shift.

Deterministic and non-deterministic Turing machines

If the action table has at most one entry for each combination of symbol and state then the machine is a "deterministic Turing machine" (DTM). If the action table contains multiple entries for a combination of symbol and state then the machine is a "non-deterministic Turing machine" (NDTM). The two are computationally equivalent, that is, it is possible to turn any NDTM into a DTM (and *vice versa*).

Oblivious Turing machines

An **oblivious** Turing machine is a Turing machine where movement of the various heads are fixed functions of time, independent of the input. In other words, there is a predetermined sequence in which the various tapes are scanned, advanced, and written to. Pippenger and Fischer (1979) showed that any computation that can be performed by a multi-tape Turing machine in *n* steps can be performed by an oblivious two-tape Turing machine in *O(n log n)* steps.

Register machine models

van Emde Boas (1990) includes all machines of this type in one category (group, class, collection) -- "the register machine". However, historically the literature has also called the most primitive member of this group i.e. "the counter machine" -- "the register machine". And the most primitive embodiment of a "counter machine" is sometimes called the "Minsky machine".

The "counter machine", also called a "register machine" model

The primitive model register machine is, in effect, a multitape 2-symbol Post-Turing machine with its behavior restricted so its tapes act like simple "counters".

By the time of Melzak, Lambek, and Minsky (all 1961) the notion of a "computer program" produced a different type of simple machine with many left-ended tapes cut from a Post-Turing tape. In all cases the models permit only two tape symbols { mark, blank }.

Some versions represent the positive integers as only a strings/stack of marks allowed in a "register" (i.e. left-ended tape), and a blank tape represented by the count "0". Minsky (1961) eliminated the PRINT instruction at the expense of providing his model with a mandatory single mark at the left-end of each tape.

In this model the single-ended tapes-as-registers are thought of as "counters", their instructions restricted to only two (or three if the TEST/DECREMENT instruction is atomized). Two common instruction sets are the following:

(1): { INC (r), DEC (r), JZ (r,z) }, i.e.
{ INCrement contents of register #r; DECrement contents of register #r; IF contents of #r=Zero THEN Jump-to Instruction #z}
(2): { CLR (r); INC (r); JE (r_i, r_j, z) }, i.e.
{ CLeaR contents of register r; INCrement contents of r; compare contents of r_i to r_j and if Equal then Jump to instruction z}

Although his model is more complicated than this simple description, the Melzak "pebble" model extended this notion of "counter" to permit multi- pebble adds and subtracts.

The Random Access Machine (RAM) model

Melzak (1961) recognized a couple serious defects in his register/counter-machine model: (i) Without a form of indirect addressing he would be not be able to "easily" show the model is Turing equivalent, (ii) The program and registers were in different "spaces", so self-modifying programs would not be easy. When Melzak added indirect addressing to his model he created a random access machine model.

(However, with Gödel numbering of the instructions Minsky (1961) offered a proof that with such numbering the general recursive functions were indeed possible; in Minsky (1967) he offers proof that μ recursion is indeed possible).

Unlike the RASP model, the RAM model does not allow the machine's actions to modify its instructions. Sometimes the model works only register-to-register with no accumulator, but most models seem to include an accumulator.

van Emde Boas (1990) divides the various RAM models into a number of sub-types:

- SRAM, the "successor RAM" with only one arithmetic instruction, the successor (INCREMENT h). The others include "CLEAR h", and an IF equality-between-register THEN jump-to xxx.
- RAM: the standard model with addition and subtraction
- MRAM: the RAM agumented with multiplication and division
- BRAM, MBRAM: Bitwise Boolean versions of the RAM and MRAM
- N****: Non-deterministic versions of any of the above with an N before the name

The Random Access Stored Program (RASP) machine model

The RASP is a RAM with the instructions stored together with their data in the same 'space' -- i.e. sequence of registers. The notion of a RASP was described at least as early as Kiphengst (1959). His model had a "mill" -- an accumulator, but now the instructions were in the registers with the data—the so-called von Neumann architecture. When the RASP has alternating even and odd registers—the even holding the "operation code" (instruction) and the odd holding its "operand" (parameter), then indirect addressing is achieved by simply modifying an instruction's operand (cf Cook and Reckhow 1973).

The original RASP model of Elgot and Robinson (1964) had only three instructions in the fashion of the register-machine model, but they placed them in the register space together with their data. (Here COPY takes the place of CLEAR when one register e.g. "z" or "0" starts with and always contains 0. This trick is not unusual. The unit 1 in register "unit" or "1" is also useful.)

$$\{ \text{INC} (r), \text{COPY} (r_i, r_j), \text{JE} (r_i, r_i, z) \}$$

The RASP models allow indirect as well as direct-addressing; some allow "immediate" instructions too, e.g. "Load accumulator with the constant 3". The instructions may be of a highly-restricted set such as the following 16 instructions of Hartmanis (1971). This model uses an accumulator A. The mnemonics are those that the authors used (their CLA is "load accumulator" with constant or from register; STO is "store accumulator"). Their syntax is the following, excepting the jumps: "n, <n>, <<n>>" for "immediate", "direct" and "indirect"). Jumps are via two "Transfer instructions" TRA—unconditional jump by directly "n" or indirectly "< n >" jamming contents of register n into the instruction counter, TRZ (conditional jump if Accumulator is zero in the same manner as TRA):

{ ADD n , ADD < n >, ADD << n >>, SUB n, SUB < n >, SUB << n >>, CLA n, CLA < n >, CLA << n >>, STO < n >, STO << n >>, TRA n, TRA < n >, TRZ n, TRA < n >, HALT }

The Pointer machine model

A relative late-comer is Schönhage's Storage Modification Machine (1970) or pointer machine. Another version is the Kolmogorov-Uspensii machine, and the Knuth "linking

automaton" proposal. Like a state-machine diagram, a node emits at least two labelled "edges" (arrows) that point to another node or nodes which in turn point to other nodes, etc. The outside world points at the center node.

Machines with input and output

Any of the above tape-based machines can be equipped with input and output tapes; any of the above register-based machines can be equipped with dedicated input and output registers. For example, the Schönhage pointer-machine model has two instructions called "*input* λ_0,λ_1" and "*output* β" (Schönhage 1990 p. 493)

It is difficult to study sublinear space complexity on multi-tape machines with the traditional model, because an input of size n already takes up space n. Thus, to study small DSPACE classes, we must use a different model. In some sense, if we never "write to" the input tape, we don't want to charge ourself for this space. And if we never "read from" our output tape, we don't want to charge ourself for this space.

We solve this problem by introducing a ***k*-string Turing machine with input and output**. This is the same as an ordinary k-string Turing machine, except that the transition function δ is restricted so that the input tape can never be changed, and so that the output head can never move left. This model allows us to define deterministic space classes smaller than linear. Turing machines with input-and-output also have the same time complexity as other Turing machines; in the words of Papaditriou 1994 Prop 2.2:

For any k-string Turing machine M operating within time bound $f(n)$ there is a $(k+2)$-string Turing machine M' with input and output, which operates within time bound $O(f(n))$.

k-string Turing machines with input and output are used in the formal definition of the complexity resource DSPACE in, for example, Papadimitriou 1994 (Def. 2.6).

Other equivalent machines and methods

- Multidimensional Turing machine: For example, a model by Schönhage (1990) uses the four head-movement commands { **N**orth, **S**outh, **E**ast, **W**est }.

- Single-tape, multi-head Turing machine: In an undecidability proof of the "problem of tag", Minsky 1961 and Shepherdson and Sturgis (1963) described machines with a single tape that could write along the tape with one head and read further along the tape with another.

- Markov's (1954) Normal Algorithm is another remarkably simple computational model equivalent to the Turing machines.

- Lambda calculus

- Queue machine

Turing machine examples

Turing's very first example

The following table is Turing's very first example (Turing 1937):

"1. A machine can be constructed to compute the sequence 0 1 0 1 0 1..." (0 <blank> 1 <blank> 0...) (*Undecidable* p. 119)

Configuration		Behavior	
m-configuration (state)	**Tape symbol**	**Tape operations**	**Final m-configuration (state)**
b	blank	P0, R	c
c	blank	R	e
e	blank	P1, R	f
f	blank	R	b

With regard to what actions the machine actually does, Turing (1936) (*Undecidable* p. 121) states the following:

"This [example] table (and all succeeding tables of the same kind) is to be understood to mean that for a configuration described in the first two columns the operations in the third column are carried out successively, and the machine then goes over into the m-configuration in the final column." (Undecidable p. 121)

He makes this very clear when he reduces the above table to a single instruction called "b" (*Undecidable* p. 120), but his instruction consists of 3 lines. Instruction "b" has three different symbol possibilities {None, 0, 1}. Each possibility is followed by a sequence of actions until we arrive at the rightmost column, where the final m-configuration is "b":

Current m-configuration (instruction)	**Tape symbol**	**Operations on the tape**	**Final m-configuration (instruction)**
b	None	P0	b
b	0	R, R, P1	b
b	1	R, R, P0	b

As observed by a number of commentators including Turing (1937) himself, (e.g., Post (1936), Post (1947), Kleene (1952), Wang (1954)) the Turing instructions are not atomic

— further simplifications of the model can be made without reducing its computational power.

As stated in Turing machine, Turing proposed that his table be further atomized by allowing only a single print/erase followed by a single tape movement L/R/N. He gives us this example of the first little table converted (*Undecidable*, p. 127):

Current m-configuration (Turing state)	Tape symbol	Print-operation	Tape-motion	Final m-configuration (Turing state)
q_1	blank	P0	R	q_2
q_2	blank	P blank, i.e. E	R	q_3
q_3	blank	P1	R	q_4
q_4	blank	P blank, i.e. E	R	q_1

Turing's statement still implies five atomic operations. At a given instruction (m-configuration) the machine:

1. observes the tape-symbol underneath the head
2. based on the observed symbol goes to the appropriate instruction-sequence to use
3. prints symbol S_j or erases or does nothing
4. moves tape left, right or not at all
5. goes to the final m-configuration for that symbol

Because a Turing machine's actions are not atomic, a simulation of the machine must atomize each 5-tuple into a sequence of simpler actions. One possibility — used in the following examples of "behaviors" of his machine — is as follows:

(q_i) Test tape-symbol under head: If the symbol is S_0 go to $q_i.01$, if symbol S_1 go to $q_i.11$, if symbol S_2 go to $q_i.21$, etc.
($q_i.01$) print symbol S_j0 or erase or do nothing then go to $q_i.02$
($q_i.02$) move tape left or right nor not at all then go to qm0
($q_i.11$) print symbol S_j1 or erase or do nothing then go to $q_i.12$
($q_i.12$) move tape left or right nor not at all then go to qm1
($q_i.21$) print symbol S_j2 or erase or do nothing then go to $q_i.22$
($q_i.22$) move tape left or right nor not at all then go to qm2
(etc — all symbols must be accounted for)

So-called "canonical" finite state machines do the symbol tests "in parallel".

In the following example of what the machine does, we will note some peculiarities of Turing's models:

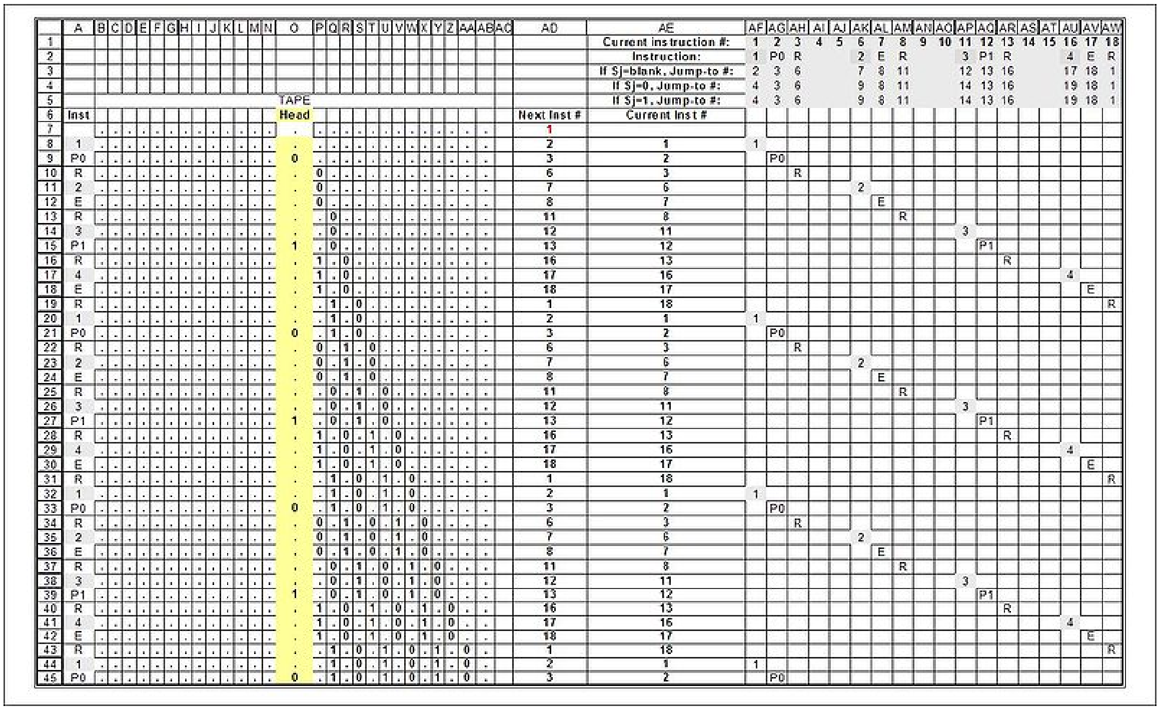
"The convention of writing the figures only on alternate squares is very useful: I shall always make use of it." (Undecidable p. 121)

Thus when printing he skips every other square. The printed-on squares are called F-squares; the blank squares in between may be used for "markers" and are called "E-squares" as in "liable to erasure." The F-squares in turn are his "Figure squares" and will only bear the symbols 1 or 0 — symbols he called "figures" (as in "binary numbers").

In this example the tape starts out "blank", and the "figures" are then printed on it. For brevity only the TABLE-states are shown here:

Sequence	**Instruction identifier**	**Head**
		. . . .
1	**1**	. . . .
2	**2**	. . . . . **0** . . . . .
3	**3**	. . . . . . **0** . . . . .
4	**4**	. . . . . **1** . **0** . . . . .
5	**1**	. . . . . . **1** . **0** . . . . .
6	**2**	. . . . . **0** . **1** . **0** . . . . .
7	**3**	. . . . . . **0** . **1** . **0** . . . . .
8	**4**	. . . . . **1** . **0** . **1** . **0** . . . .
9	**1**	. . . . . . **1** . **0** . **1** . **0** . . . .
10	**2**	. . . . . **0** . **1** . **0** . **1** . **0** . . .
11	**3**	. . . . . . **0** . **1** . **0** . **1** . **0** . .
12	**4**	. . . . . **1** . **0** . **1** . **0** . **1** . **0** .
13	**1**	. . . . . . **1** . **0** . **1** . **0** . **1** . **0**
14	**2**	. . . . . **0** . **1** . **0** . **1** . **0** . **1** . **0**

The same "run" with all the intermediate tape-printing and movements is shown here:



	A	B–N	O	P–AC	AD	AE	AF	AG	AH	AI	AJ	AK	AL	AM	AN	AO	AP	AQ	AR	AS	AT	AU	AV	AW
1						Current instruction #:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
2						Instruction:	1	P0	R			2	E	R			3	P1	R			4	E	R
3						If Sj=blank, Jump-to #:	2	3	6			7	8	11			12	13	16			17	18	1
4						If Sj=0, Jump-to #:	4	3	6			9	8	11			14	13	16			19	18	1
5			TAPE			If Sj=1, Jump-to #:	4	3	6			9	8	11			14	13	16			19	18	1
6	Inst		Head		Next Inst #	Current Inst #																		
7		. . . .	.	. . . .	1																			
8	1	. . . .	.	. . . .	2	1	1																	
9	P0	. . . .	0	. . . .	3	2		P0																
10	R	. . . .	.	0	6	3			R															
11	2	. . . .	.	0	7	6						2												
12	E	. . . .	.	0	8	7							E											
13	R	. . . .	.	. 0	11	8								R										
14	3	. . . .	.	. 0	12	11											3							
15	P1	. . . .	1	. 0	13	12												P1						
16	R	. . . .	.	1 . 0	16	13													R					
17	4	. . . .	.	1 . 0	17	16																4		
18	E	. . . .	.	1 . 0	18	17																	E	
19	R	. . . .	.	. 1 . 0	1	18																		R
20	1	. . . .	.	. 1 . 0	2	1	1																	
21	P0	. . . .	0	. 1 . 0	3	2		P0																
22	R	. . . .	.	0 . 1 . 0	6	3			R															
23	2	. . . .	.	0 . 1 . 0	7	6						2												
24	E	. . . .	.	0 . 1 . 0	8	7							E											
25	R	. . . .	.	. 0 . 1 . 0	11	8								R										
26	3	. . . .	.	. 0 . 1 . 0	12	11											3							
27	P1	. . . .	1	. 0 . 1 . 0	13	12												P1						
28	R	. . . .	.	1 . 0 . 1 . 0	16	13													R					
29	4	. . . .	.	1 . 0 . 1 . 0	17	16																4		
30	E	. . . .	.	1 . 0 . 1 . 0	18	17																	E	
31	R	. . . .	.	. 1 . 0 . 1 . 0	1	18																		R
32	1	. . . .	.	. 1 . 0 . 1 . 0	2	1	1																	
33	P0	. . . .	0	. 1 . 0 . 1 . 0	3	2		P0																
34	R	. . . .	.	0 . 1 . 0 . 1 . 0	6	3			R															
35	2	. . . .	.	0 . 1 . 0 . 1 . 0	7	6						2												
36	E	. . . .	.	0 . 1 . 0 . 1 . 0	8	7							E											
37	R	. . . .	.	. 0 . 1 . 0 . 1 . 0 . . .	11	8								R										
38	3	. . . .	.	. 0 . 1 . 0 . 1 . 0 . . .	12	11											3							
39	P1	. . . .	1	. 0 . 1 . 0 . 1 . 0 . . .	13	12												P1						
40	R	. . . .	.	1 . 0 . 1 . 0 . 1 . 0 . .	16	13													R					
41	4	. . . .	.	1 . 0 . 1 . 0 . 1 . 0 . .	17	16																4		
42	E	. . . .	.	1 . 0 . 1 . 0 . 1 . 0 . .	18	17																	E	
43	R	. . . .	.	. 1 . 0 . 1 . 0 . 1 . 0 .	1	18																		R
44	1	. . . .	.	. 1 . 0 . 1 . 0 . 1 . 0 .	2	1	1																	
45	P0	. . . .	0	. 1 . 0 . 1 . 0 . 1 . 0 .	3	2		P0																

A close look at the table reveals certain problems with Turing's own example—not all the symbols are accounted for.

For example, suppose his tape was not initially blank. What would happen? The Turing machine would read different values than the intended values.

A copy subroutine

This is a very important subroutine used in the "multiply" routine.

The example Turing machine handles a string of 0s and 1s, with 0 represented by the blank symbol. Its task is to double any series of 1s encountered on the tape by writing a 0 between them. For example, when the head reads "111", it will write a 0, then "111". The output will be "1110111".

In order to accomplish its task, this Turing machine will need only 5 states of operation, which are called $\{s_1, s_2, s_3, s_4, s_5\}$. Each state does 4 actions:

1. Read the symbol under the head
2. Write the output symbol decided by the state
3. Move the tape to the left or to the right decided by the state
4. Switch to the following state decided by the current state

Initial m-configuration (current instruction)	Tape symbol	Print operation	Tape motion	Final m-configuration (next instruction)
s_1	0	N	N	H
s_1	1	E	R	s_2
s_2	0	E	R	s_3
s_2	1	P1	R	s_2
s_3	0	P1	L	s_4
s_3	1	P1	R	s_3
s_4	0	E	L	s_5
s_4	1	P1	L	s_4
s_5	0	P1	R	s_1
s_5	1	P1	L	s_5
H	-	-	-	

A "run" of the machine sequences through 16 machine-configurations (aka Turing states):

Sequence	Instruction identifier		Head	
1	s_1	0 0 0 0 **1**	**1**	0 0 0 0 0
2	s_2	0 0 0 0 0	**1**	0 0 0 0 0
3	s_2	0 0 0 0 0	0	**1** 0 0 0 0
4	s_3	0 0 0 0 0	0	0 **1** 0 0 0
5	s_4	0 0 0 0 **1**	0	**1** 0 0 0 0
6	s_5	0 0 0 **1** 0	**1**	0 0 0 0 0
7	s_5	0 0 **1** 0 **1**	0	0 0 0 0 0
8	s_1	0 0 0 **1** 0	**1**	**1** 0 0 0 0
9	s_2	0 0 0 0 **1**	0	0 **1** 0 0 0
10	s_3	0 0 0 0 0	**1**	0 0 **1** 0 0
11	s_3	0 0 0 0 0	0	**1** 0 0 **1** 0
12	s_4	0 0 0 0 **1**	**1**	0 0 **1** 0 0
13	s_4	0 0 0 **1 1**	0	0 **1** 0 0 0
14	s_5	0 0 **1 1** 0	0	**1** 0 0 0 0
15	s_1	0 0 0 **1 1**	0	**1 1** 0 0 0
16	H	0 0 0 **1 1**	0	**1 1** 0 0 0

The behavior of this machine can be described as a loop: it starts out in s_1, replaces the first 1 with a 0, then uses s_2 to move to the right, skipping over 1s and the first 0 encountered. s_3 then skips over the next sequence of 1s (initially there are none) and replaces the first 0 it finds with a 1. s_4 moves back to the left, skipping over 1s until it

finds a 0 and switches to s_5. s_5 then moves to the left, skipping over 1s until it finds the 0 that was originally written by s_1.

It replaces that 0 with a 1, moves one position to the right and enters s_1 again for another round of the loop.

This continues until s_1 finds a 0 (this is the 0 in the middle of the two strings of 1s) at which time the machine halts.

Alternative description

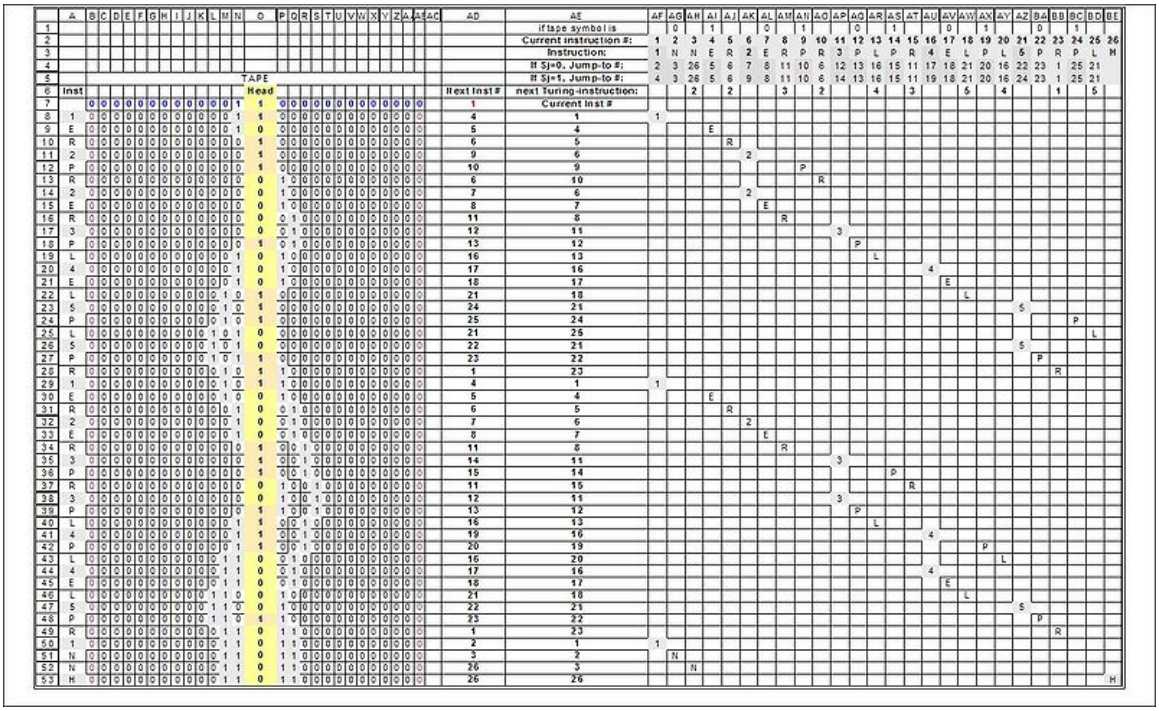
Another description sees the problem as how to keep track of how many "1"s there are. We can't use one state for each possible number (a state for each of 0,1,2,3,4,5,6 etc), because then we'd need infinite states to represent all the natural numbers, and the state machine is *finite* - we'll have to track this using the tape in some way.

The basic way it works is by copying each "1" to the other side, by moving back and forth - it is intelligent enough to remember which part of the trip it is on. In more detail, it carries each "1" across to the other side, by recognizing the separating "0" in the middle, and recognizing the "0" on the other side to know it's reached the end. It comes back using the same method, detecting the middle "0", and then the "0" on the original side. This "0" on the original side is the key to the puzzle of how it keeps track of the number of 1's.

The trick is that before carrying the "1", it marks that digit as "taken" by replacing it with an "0". When it returns, it fills that "0" back in with a "1", *then moves on to the next one*, marking it with an "0" and repeating the cycle, carrying that "1" across and so on. *With each trip across and back, the marker "0" moves one step closer to the centre*. This is how it keeps track of how many "1"'s it has taken across.

Interestingly, when it returns, the marker "0" looks like the end of the collection of "1"s to it - any "1"s that have already been taken across are invisible to it (on the other side of the marker "0") and so it is as if it is working on an (N-1) number of "1"s - similar to a proof by Mathematical induction.

A full "run" showing the results of the intermediate "motions".

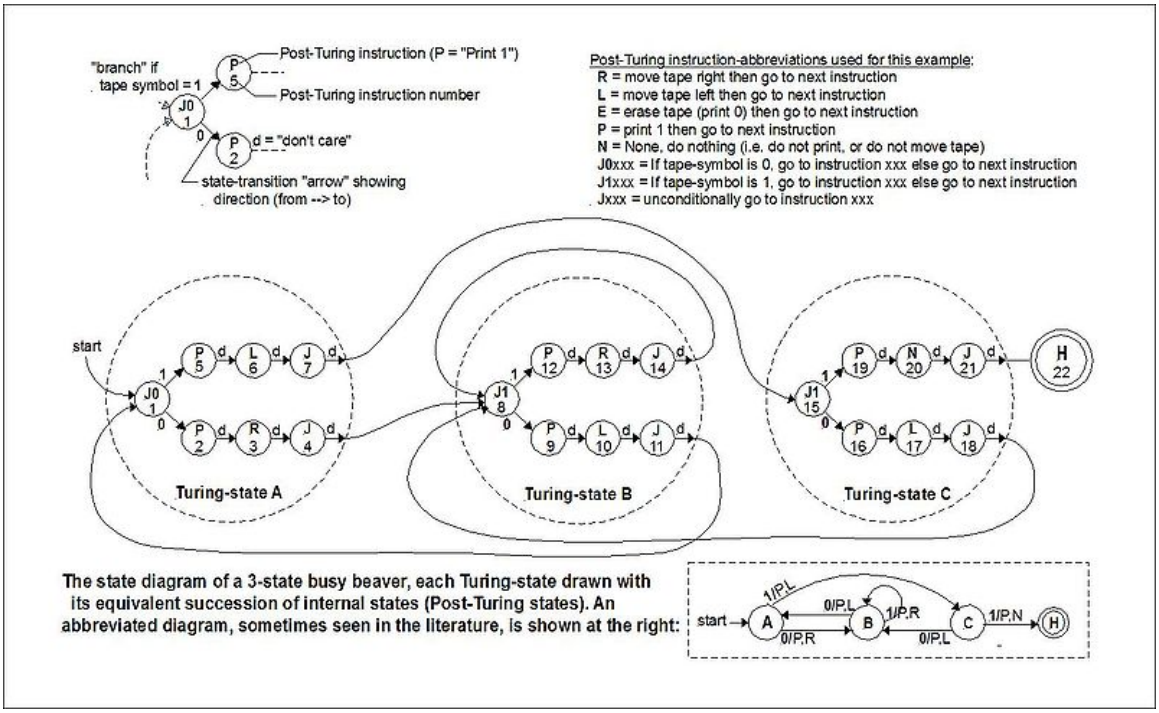


3-state Busy Beaver

The following Turing table of instructions was derived from Peterson (1988) page 198, Figure 7.15. Peterson moves the head; in the following model the tape moves.

Tape symbol	**Current state A**			**Current state B**			**Current state C**		
	Write symbol	Move tape	Next state	Write symbol	Move tape	Next state	Write symbol	Move tape	Next state
0	1	R	**B**	1	L	**A**	1	L	**B**
1	1	L	**C**	1	R	**B**	1	N	**HALT**

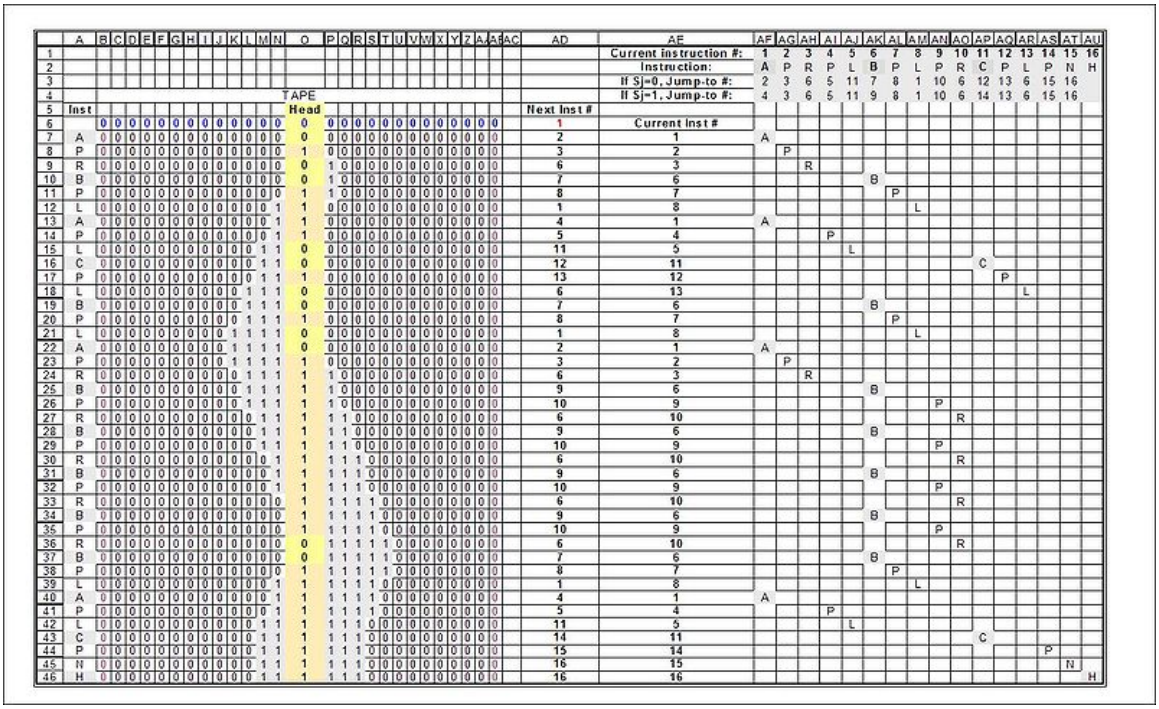
The "state" drawing of the 3-state busy beaver shows the internal sequences of events required to actually perform "the state". As noted above Turing (1937) makes it perfectly clear that this is the proper interpretation of the 5-tuples that describe the instruction (*Undecidable*, p. 119).



The following table shows the "compressed" run — just the Turing states:

Sequence	Instruction identifier		Head	
1	**A**	0 0 0 0 0 0	**0**	0 0 0 0 0 0 0
2	**B**	0 0 0 0 0 0	**0**	1 0 0 0 0 0 0
3	**A**	0 0 0 0 0 **1**	**1**	0 0 0 0 0 0 0
4	**C**	0 0 0 0 **1 1**	**0**	0 0 0 0 0 0 0
5	**B**	0 0 0 **1 1 1**	**0**	0 0 0 0 0 0 0
6	**A**	0 0 **1 1 1 1**	**0**	0 0 0 0 0 0 0
7	**B**	0 0 0 **1 1 1**	**1**	1 0 0 0 0 0 0
8	**B**	0 0 0 0 **1 1**	**1**	**1 1** 0 0 0 0 0
9	**B**	0 0 0 0 0 **1**	**1**	**1 1 1** 0 0 0 0
10	**B**	0 0 0 0 0 0	**1**	**1 1 1 1** 0 0 0
11	**B**	0 0 0 0 0 0	**0**	**1 1 1 1 1** 0 0
12	A	0 0 0 0 0 **1**	**1**	**1 1 1 1** 0 0 0
13	**C**	0 0 0 0 **1 1**	**1**	**1 1 1** 0 0 0 0
14	**H**	0 0 0 0 **1 1**	**1**	**1 1 1** 0 0 0 0

The full "run" of the 3-state busy beaver. The resulting Turing-states (what Turing called the "m-configurations" — "machine-configurations") are shown highlighted in grey in column A, and also under the machine's instructions (columns AF-AU)):



Wolfram Turing machine examples

In 1985, Stephen Wolfram conjectured that a one-dimensional two-state cellular automaton he later called Rule 110 was Turing complete (universal). Matthew Cook verified that conjecture in 2000. Wolfram's book *A New Kind of Science* gives useful examples of how Turing machines can be visually represented, and further conjectured that an even simpler 2-state 3-symbol Turing machine might also be universal. In October 2007, Wolfram Research awarded Alex Smith a $25,000 prize for verifying the second conjecture, though Vaughan Pratt disputed the validity of the proof, which later Wolfram Research and Smith himself clarified. Pratt's main argument is that the proof argument would make a Linear Bounded Automata (LBA) universal. Smith later explained however that the LBA would need to be restarted at running time to perform a computation, while the 2, 3 Turing machine would restart automatically, therefore the proof does not make an LBA universal as Pratt first thought.

Chapter 4

Post–Turing Machine and Langton's Ant

Post–Turing machine

A **Post–Turing machine** is a "program formulation" of an especially simple type of Turing machine, comprising a variant of Emil Post's Turing-equivalent model of computation described below. (Post's model and Turing's model, though very similar to one another, were developed independently. Turing's paper was received for publication in May of 1936, followed by Post's in October.) A Post–Turing machine uses a binary alphabet, an infinite sequence of binary storage locations, and a primitive programming language with instructions for bi-directional movement among the storage locations and alteration of their contents one at a time. The names "Post–Turing program" and "Post–Turing machine" were used by Martin Davis in 1973–1974 (Davis 1973, p.69ff). Later in 1980, Davis used the name "Turing–Post program" (Davis, in Steen p. 241).

1936: Post model

In his 1936 paper "Finite combinatory processes—formulation 1" (which can be found on page 289 of *The Undecidable*), Emil Post described a model of extreme simplicity which he conjectured is "logically equivalent to recursiveness", and which was later proved to be so. The quotes in the following are from this paper.

Post's model of a computation differs from the Turing-machine model in a further "atomization" of the acts a human "computer" would perform during a computation.

Post's model employs a "symbol space" consisting of a "two-way infinite sequence of spaces or boxes", each box capable of being in either of two possible conditions, namely "marked" (as by a single vertical stroke) and "unmarked" (empty). Initially, finitely-many of the boxes are marked, the rest being unmarked. A "worker" is then to move among the boxes, being in and operating in only one box at a time, according to a fixed finite "set of directions" (instructions), which are numbered in order (1,2,3,...,n). Beginning at a box "singled out as the starting point", the worker is to follow the set of instructions one at a time, beginning with instruction 1.

The instructions may require the worker to perform the following "basic acts" or "operations":

(a) *Marking the box he is in (assumed empty),*
(b) *Erasing the mark in the box he is in (assumed marked),*
(c) *Moving to the box on his right,*
(d) *Moving to the box on his left,*
(e) *Determining whether the box he is in, is or is not marked.*

Specifically, the i^{th} "direction" (instruction) given to the worker is to be one of the following forms:

(A) *Perform operation* O_i [O_i = (a), (b), (c) *or* (d)] *and then follow direction* j_i,
(B) *Perform operation* (e) *and according as the answer is yes or no correspondingly follow direction* j_i' *or* j_i'' ,
(C) *Stop*.

(The above indented text and italics are as in the original.) Post remarks that this formulation is "in its initial stages" of development, and mentions several possibilities for "greater flexibility" in its final "definitive form", including

(1) replacing the infinity of boxes by a finite extensible symbol space, "extending the primitive operations to allow for the necessary extension of the given finite symbol space as the process proceeds",
(2) using an alphabet of more than two symbols, "having more than one way to mark a box",
(3) introducing finitely-many "physical objects to serve as pointers, which the worker can identify and move from box to box".

1947: Post's formal reduction of the Turing 5-tuples to 4-tuples

As briefly mentioned in Turing machine, Post, in his paper of 1947 (*Recursive Unsolvability of a Problem of Thue*) atomized the Turing 5-tuples to 4-tuples:

"Our quadruplets are quintuplets in the Turing development. That is, where our standard instruction orders either a printing (overprinting) **or** motion, left or right, Turing's standard instruction always order a printing **and** a motion, right, left, or none"(footnote 12, Undecidable p. 300)

Like Turing he defined erasure as printing a symbol "S0". And so his model admitted quadruplets of only three types (cf p. 294 *Undecidable*):

$q_i\ S_j\ L\ q_l$,
$q_i\ S_j\ R\ q_l$,
$q_i\ S_j\ S_k\ q_l$

At this time he was still retaining the Turing state-machine convention – he had not formalized the notion of an assumed *sequential* execution of steps until a specific test of a symbol "branched" the execution elsewhere.

1954, 1957: Wang model

For an even further reduction – to only four instructions – of the Wang model presented here see Wang B-machine.

Wang (1957, but presented to the ACM in 1954) is often cited (cf Minsky (1967) p. 200) as the source of the "program formulation" of binary-tape Turing machines using numbered instructions from the set

write 0
write 1
move left
move right
if scanning 0 then goto instruction *i*
if scanning 1 then goto instruction *j*

where *sequential execution* is assumed, and Post's single "if ... then ... else" has been "atomised" into two "if ... then ..." statements. (Here '1' and '0' are used where Wang used "marked" and "unmarked", respectively, and the initial tape is assumed to contain only '0's except for finitely-many '1's.)

Wang noted the following:

- "Since there is no separate instruction for halt (stop), it is understood that the machine will stop when it has arrived at a stage that the program contains no instruction telling the machine what to do next." (p.65)
- "In contrast with Turing who uses a one-way infinite tape that has a beginning, we are following Post in the use of a 2-way infinite tape." (p. 65)
- Unconditional gotos are easily derived from the above instructions, so "we can freely use them too". (p.84)

Any binary-tape Turing machine is readily converted to an equivalent "Wang program" using the above instructions.

1974: first Davis model

Martin Davis was a undergraduate student of Emil Post's. Along with Stephen Kleene he completed his PhD under Alonzo Church (Davis (2000) 1st and 2nd footnotes p. 188).

The following model he presented in a series of lectures to the Courant Institute at NYU in 1973–1974. This is the model to which Davis formally applied the name "Post–Turing machine" with its "Post–Turing language". The instructions are assumed to be executed sequentially (Davis 1974, p. 71):

"Write 1
"Write B
"To A if read 1
"To A if read B
"RIGHT
"LEFT

Note that there is no "halt" or "stop".

1978 second Davis model

The following model appears as an essay *What is a computation?* in Steen pages 241–267. For some reason Davis has renamed his model a "Turing–Post machine" (with one back-sliding on page 256.)

In the following model Davis assigns the numbers "1" to Post's "mark/slash" and "0" to the blank square. To quote Davis: "We are now ready to introduce the Turing–Post Programming Language. In this language there are seven kinds of instructions:

"PRINT 1
"PRINT 0
"GO RIGHT
"GO LEFT
"GO TO STEP i IF **1** IS SCANNED
"GO TO STEP i IF **0** IS SCANNED
"STOP

"A Turing–Post program is then a list of instructions, each of which is of one of these seven kinds. Of course in an actual program the letter *i* in a step of either the fifth or sixth kind must replaced with a definite (positive whole) number." (Davis in Steen, p. 247).

- Confusion arises if one does not realize that a "blank" tape is actually printed with all zeroes — there is no "blank".
- Splits Post's "GO TO" ("branch" or "jump") instruction into two, thus creating a larger (but easier-to-use) instruction set of seven rather than Post's six instructions.
- Does not mention that instructions PRINT 1, PRINT 0, GO RIGHT and GO LEFT imply that, after execution, the "computer" must go to the next step in numerical sequence.

1994 (2nd Edition) Davis–Sigal–Weyuker's Post–Turing program model

"Although the formulation of Turing we have presented is closer in spirit to that originally given by Emil Post, it was Turing's analysis of the computation that has made

this formulation seem so appropriate. This language has played a fundamental role in theoretical computer science." (Davis et al. (1994) p. 129)

This model allows for the printing of multiple symbols. The model allows for B (blank) instead of S_0. The tape is infinite in both directions. Either the head or the tape moves, but their definitions of RIGHT and LEFT always specify the same outcome in either case (Turing used the same convention).

PRINT σ ;Replace scanned symbol with σ
IF σ GOTO L ;IF scanned symbol is σ THEN goto "the first" instruction labelled L
RIGHT ;Scan square immediately right of the square currently scanned
LEFT ;Scan square immediately left of the square currently scanned

Note that only one type of "jump" – a conditional GOTO – is specified; for an unconditional jump a string of GOTO's must test each symbol.

This model reduces to the binary { 0, 1 } versions presented above, as shown here:

PRINT 0 = ERASE ;Replace scanned symbol with 0 = B = BLANK
PRINT 1 ;Replace scanned symbol with 1
IF 0 GOTO L ;IF scanned symbol is 0 THEN goto "the first" instruction labelled L
IF 1 GOTO L ;IF scanned symbol is 1 THEN goto "the first" instruction labelled L
RIGHT ;Scan square immediately right of the square currently scanned
LEFT ;Scan square immediately left of the square currently scanned

Examples of the Post–Turing machine

Atomizing Turing quintuples into a sequence of Post–Turing instructions

The following "reduction" (decomposition, atomizing) method – from 2-symbol Turing 5-tuples to a sequence of 2-symbol Post–Turing instructions – can be found in Minsky (1961). He states that this reduction to "a *program* ... a sequence of *Instructions*" is in the spirit of Hao Wang's B-machine (italics in original, cf Minsky (1961) p. 439).

(Minsky's reduction to what he calls "a sub-routine" results in 5 rather than 7 Post–Turing instructions. He did not atomize Wi0: "Write symbol Si0; go to new state Mi0", and Wi1: "Write symbol Si1; go to new state Mi1". The following method further atomizes Wi0 and Wi1; in all other respects the methods are identical.)

This reduction of Turing 5-tuples to Post–Turing instructions may not result in an "efficient" Post–Turing program, but it will be faithful to the original Turing-program.

In the following example, each Turing 5-tuple of the 2-state busy beaver converts into

(i) an initial conditional "jump" (goto, branch), followed by

(ii) 2 tape-action instructions for the "0" case – Print or Erase or None, followed by Left or Right or None, followed by
(iii) an unconditional "jump" for the "0" case to its next instruction
(iv) 2 tape-action instructions for the "1" case – Print or Erase or None, followed by Left or Right or None, followed by
(v) an unconditional "jump" for the "1" case to its next instruction

for a total of 1 + 2 + 1 + 2 + 1 = 7 instructions per Turing-state.

For example, the 2-state busy beaver's "A" Turing-state, written as two lines of 5-tuples, is:

Initial m-configuration (Turing state)	Tape symbol	Print operation	Tape motion	Final m-configuration (Turing state)
A	0	P	R	**B**
A	1	P	L	**B**

The table represents just a single Turing "instruction", but we see that it consists of two lines of 5-tuples, one for the case "tape symbol under head = 1", the other for the case "tape symbol under head = 0". Turing observed (Undecidable, p. 119) that the left-two columns – "m-configuration" and "symbol" – represent the machine's current "configuration" – its state including both Tape and Table at that instant – and the last three columns are its subsequent "behavior". As the machine cannot be in two "states" at once, the machine must "branch" to either one configuration or the other:

Initial m-configuration and symbol S	Print operation	Tape motion	Final m-configuration
S=0 -->	P -->	R -->	**B**
--> **A** <			
S=1 -->	P -->	L -->	**B**

After the "configuration branch" (J1 xxx) or (J0 xxx) the machine follows one of the two subsequent "behaviors". We list these two behaviors on one line, and number (or label) them sequentially (uniquely). Beneath each jump (branch, go to) we place its jump-to "number" (address, location):

	Initial m-configuration & symbol S	Print operation	Tape motion	Final m-configuration case S=0	Print operation	Tape motion	Final m-configuration case S=1
	If S=0 then:	P	R	**B**			
	---> **A** <						
	If S=1 then:				P	L	**B**
instruction #	1	2	3	4	5	6	7
Post–Turing instruction	**J1**	**P**	**R**	**J**	**P**	**L**	**J**
jump-to instruction #	**5**			**B**			**B**

Per the Post–Turing machine conventions each of the Print, Erase, Left, and Right instructions consist of two actions:

(i) Tape action: { P, E, L, R}, then
(ii) Table action: go to next instruction in sequence

And per the Post–Turing machine conventions the conditional "jumps" J0xxx, J1xxx consist of two actions:

(i) Tape action: look at symbol on tape under the head
(ii) Table action: If symbol is 0 (1) and J0 (J1) then go to xxx else go to next instruction in sequence

And per the Post–Turing machine conventions the unconditional "jump" Jxxx consists of a single action, or if we want to regularize the 2-action sequence:

(i) Tape action: look at symbol on tape under the head
(ii) Table action: If symbol is 0 then go to xxx else if symbol is 1 then go to xxx.

Which, and how many, jumps are necessary? The unconditional jump **J**xxx is simply **J0** followed immediately by **J1** (or vice versa). Wang (1957) also demonstrates that only one conditional jump is required, i.e. either **J0**xxx or **J1**xxx. However, with this restriction the machine becomes difficult to write instructions for. Often only two are used, i.e.

(i) { **J0**xxx, **J1**xxx }
(ii) { **J1**xxx, **J**xxx }
(iii) { **J0**xxx, **J**xxx },

but the use of all three { **J0**xxx, **J1**xxx, **J**xxx } does eliminate extra instructions. In the 2-state Busy Beaver example that we use only { **J1**xxx, **J**xxx }.

2-state Busy Beaver

The mission of the busy beaver is to print as many ones as possible before halting. The "Print" instruction writes a 1, the "Erase" instruction (not used in this example) writes a 0 (i.e. it is the same as P0). The tape moves "Left" or "Right" (i.e. the "head" is stationary).

State table for a 2-state Turing-machine busy beaver:

	Current state **A**:			Current state **B**:		
	Write symbol:	Move tape:	Next state:	Write symbol:	Move tape:	Next state:
tape symbol is 0:	1	R	**B**	1	L	**A**
tape symbol is 1:	1	L	**B**	1	N	**H**

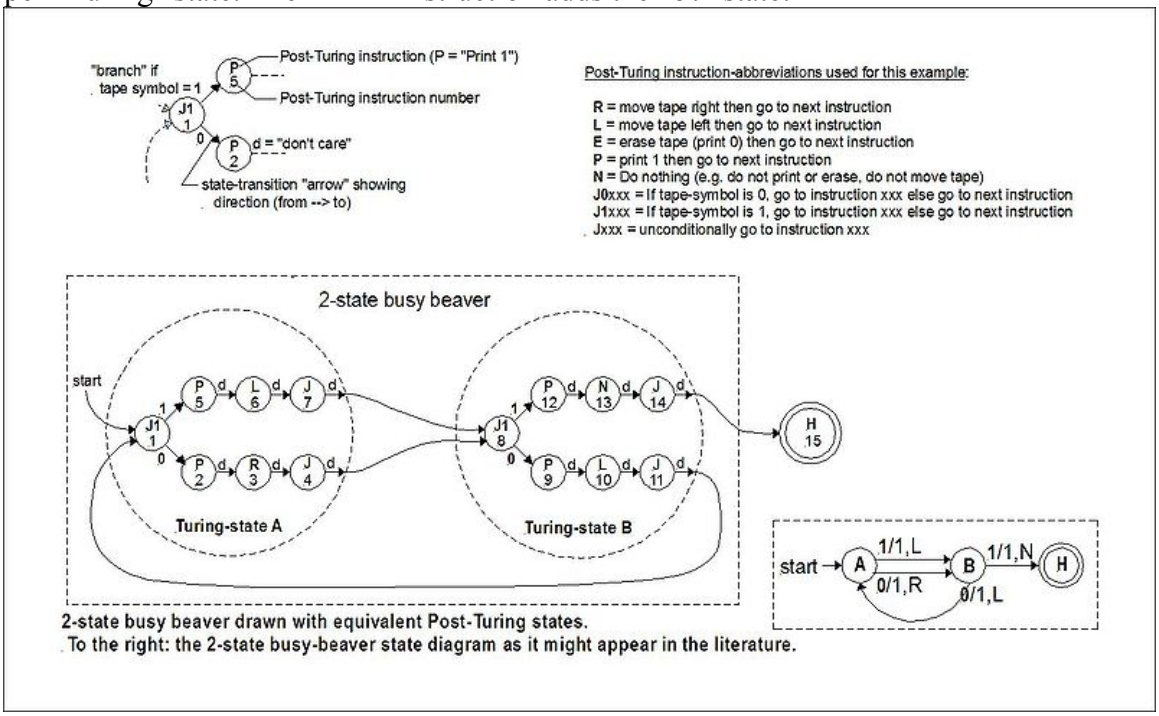
Instructions for the Post–Turing version of a 2-state busy beaver: observe that all the instructions are on the same line and in sequence. This is a significant departure from the "Turing" version and is in the same format as what is called a "computer program":

Instruction #:	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Instruction:	J1	P	R	J	P	L	J	J1	P	L	J	P	N	J	H
Jump-to #:	5			8			8	12			1			15	
Turing-state label:	**A**							**B**							**H**

Alternately, we might write the table as a string. The use of "parameter separators" ":" and instruction-separators "," are entirely our choice and do not appear in the model. There are no conventions, for some useful ideas of how to combine state diagram conventions with the instructions – i.e. to use arrows to indicate the destination of the jumps). In the example immediately below, the instructions are *sequential* starting from "1", and the parameters/"operands" are considered part of their instructions/"opcodes":

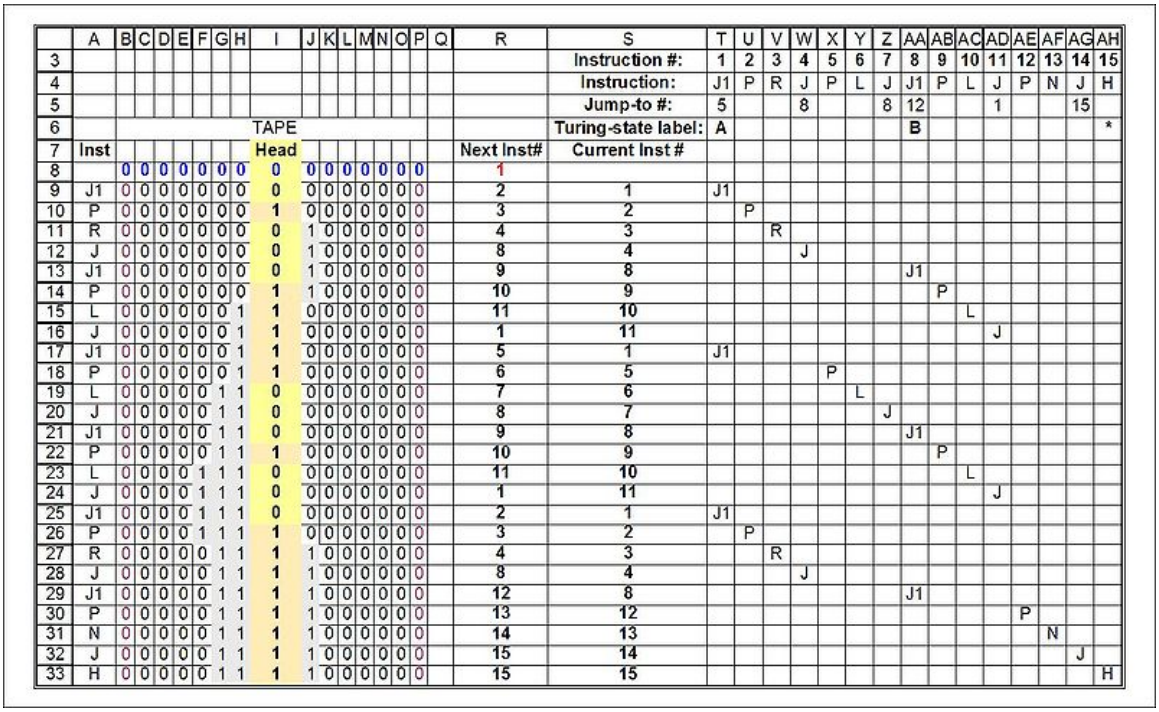
J1:5, P, R, J:8, P, L, J:8, J1:12, P, L, J1:1, P, N, J:15, H

The state diagram of a two-state busy beaver (little drawing, right-hand corner) converts to the equivalent Post–Turing machine with the substitution of 7 Post–Turing instructions per "Turing" state. The HALT instruction adds the 15th state:



2-state busy beaver drawn with equivalent Post-Turing states.
To the right: the 2-state busy-beaver state diagram as it might appear in the literature.

A "run" of the 2-state busy beaver with all the intermediate steps of the Post–Turing machine shown:



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	AA	AB	AC	AD	AE	AF	AG	AH
3																			Instruction #:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
4																			Instruction:	J1	P	R	J	P	L	J	J1	P	L	J	P	N	J	H
5																			Jump-to #:	5			8			8	12			1			15	
6						TAPE													Turing-state label:	A							B							*
7	Inst								Head									Next Inst#	Current Inst #															
8		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		1																
9	J1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		2	1	J1														
10	P	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0		3	2		P													
11	R	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0		4	3			R												
12	J	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0		8	4				J											
13	J1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0		9	8								J1							
14	P	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0		10	9									P						
15	L	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0		11	10										L					
16	J	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0		1	11											J				
17	J1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0		5	1	J1														
18	P	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0		6	5					P										
19	L	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0		7	6						L									
20	J	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0		8	7							J								
21	J1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0		9	8								J1							
22	P	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0		10	9									P						
23	L	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0		11	10										L					
24	J	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0		1	11											J				
25	J1	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0		2	1	J1														
26	P	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0		3	2		P													
27	R	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0		4	3			R												
28	J	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0		8	4				J											
29	J1	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0		12	8								J1							
30	P	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0		13	12												P			
31	N	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0		14	13													N		
32	J	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0		15	14														J	
33	H	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0		15	15															H

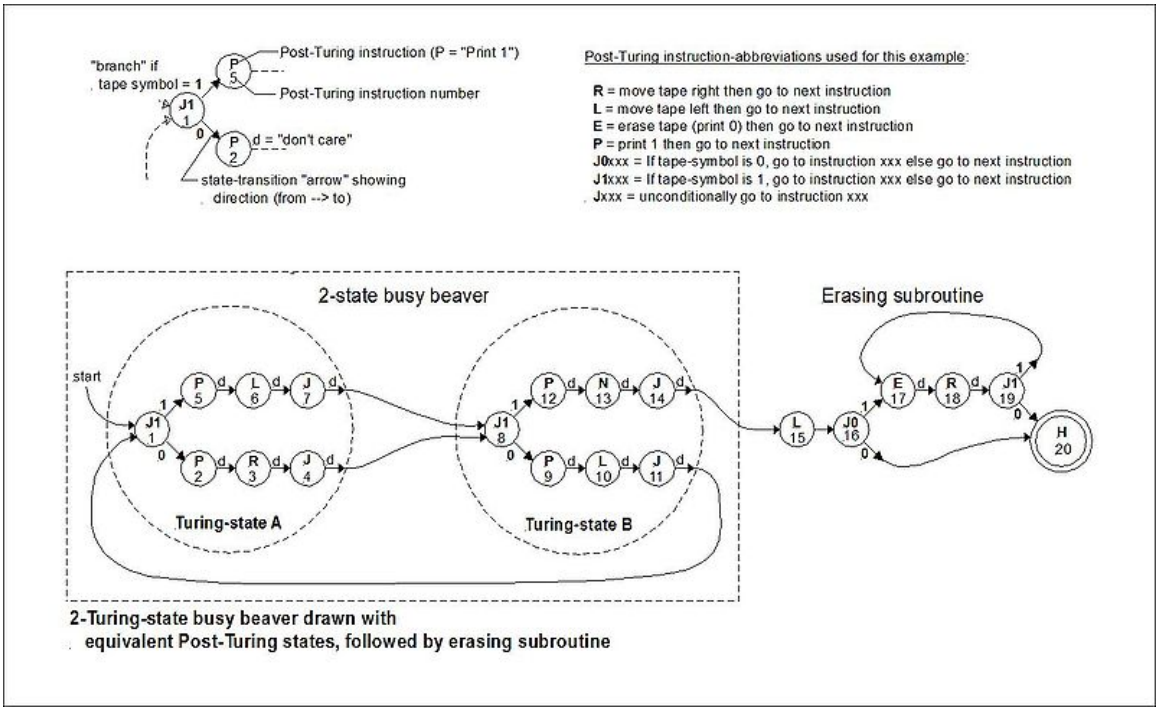
Two state busy beaver followed by "tape cleanup"

The following is a two-state Turing busy beaver with additional instructions 15–20 to demonstrate the use of "Erase", J0, etc. These will erase the 1's written by the busy beaver:

Instruction #:	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
Instruction:	J1	P	R	J	P	L	J	J1	P	L	J	P	N	J	L	J0	E	R	J1	H
Jump-to #:	5			8			8	12			1			15		20			17	
Turing-state label:	**A**							**B**							*					

Additional Post–Turing instructions 15 through 20 erase the symbols created by the busy beaver. These "atomic" instructions are more "efficient" than their Turing-state equivalents (of 7 Post–Turing instructions). To accomplish the same task a Post–Turing machine will (usually) require fewer Post–Turing states than a Turing-machine, because (i) a jump (go-to) can occur to any Post–Turing instruction (e.g. P, E, L, R) within the Turing-state, (ii) a grouping of move-instructions such as L, L, L, P are possible, etc.:

Instruction #:	**16**	**17**	**18**	**19**	**20**
Instruction:	J0	E	R	J1	H
Jump-to #:	20			17	



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	AA	AB	AC	AD	AE	AF	AG	AH	AI	AJ	AK	AL	AM
3																			Instruction #:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
4																			Instruction:	J1	P	R	J	P	L	J	J1	P	L	J	P	N	J	L	J0	E	R	J1	H
5																			Jump-to #:	5			8			8	12			1			15		20			17	
6									TAPE										Turing-state label:	A							B												H
7	Inst								Head									Next Inst#	Current Inst #																				
8		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		1																					
9	J1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		2	1	J1																			
10	P	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0		3	2		P																		
11	R	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0		4	3			R																	
12	J	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0		8	4				J																
13	J1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0		9	8								J1												
14	P	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0		10	9									P											
15	L	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0		11	10										L										
16	J	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0		1	11											J									
17	J1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0		5	1	J1																			
18	P	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0		6	5					P															
19	L	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0		7	6						L														
20	J	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0		8	7							J													
21	J1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0		9	8								J1												
22	P	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0		10	9									P											
23	L	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0		11	10										L										
24	J	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0		1	11											J									
25	J1	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0		2	1	J1																			
26	P	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0		3	2		P																		
27	R	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0		4	3			R																	
28	J	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0		8	4				J																
29	J1	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0		12	8								J1												
30	P	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0		13	12													P							
31	N	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0		14	13														N						
32	J	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0		15	14															J					
33	L	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0		16	15																L				
34	J0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0		17	16																	J0			
35	E	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0		18	17																		E		
36	R	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0		19	18																			R	
37	J1	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0		17	19																				J1
38	E	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0		18	17																		E		
39	R	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0		19	18																			R	
40	J1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0		17	19																				J1
41	E	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0		18	17																		E		
42	R	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0		19	18																			R	
43	J1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0		17	19																				J1
44	E	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		18	17																		E		
45	R	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		19	18																			R	
46	J1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		20	19																				J1
47	H	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		20	20																				H

Example: Multiply 3 × 4 with a Post–Turing machine

An example of "multiply" $a \times b = c$ on a Post–Turing machine. At the start, the tape (shown on the left) has two numbers on it – **a'** = 3' (4 marks), **b'** = 4' (5 marks). (A single mark would represent "0".) At the end the tape will have the product **c'** = 12' (13 marks) to the right of b. Note "top" and "bottom" are there just to clarify what the P–T machine is doing.

This example is a reference to show how a "multiply" computation would proceed on a single-tape, 2-symbol { blank, 1 } Post–Turing machine model.

This particular "multiply" algorithm is recursive through two loops. The head moves. It starts to the far left (the top) of the string of unary marks representing **a'** :

- Move head far right. Establish (i.e. "clear") register **c** by placing a single blank and then a mark to the right of **b**
- **a_loop**: Move head right once, test for the bottom of **a'** (a blank). If blank then done else erase mark;
- Move head right to **b'** . Move head right once past the top mark of **b'** ;
- **b_loop**: If head is at the bottom of **b'** (a blank) then move head to far left of **a'** , else:

- Erase a mark to locate counter (a blank) in **b'** .
- Increment **c'** : Move head right to top of **c'** and increment **c'** .
- Move head left to the counter inside **b'** ,
- Repair counter: print a mark in the blank counter.
- Decrement **b'** −count: Move head right once.
- Return to b_loop.

Multiply **a** × **b** = **c**, for example: 3 × 4 = 12. The scanned square is indicated by brackets around the mark i.e. [|]. An extra mark serves to indicate the symbol "0":
At the start of the computation **a'** is 4 unary marks, then a separator blank, **b'** is 5 unary marks, then a separator mark. An unbounded number of empty spaces must be available for **c** to the right:

....a'.b'.... = : ....**[|]** | | | . | | | | | ....

During the computation the head shuttles back and forth from **a'** to **b'** to **c'** back to **b'** then to **c'** , then back to **b'** , then to **c'** ad nauseam while the machine counts through **b'** and increments **c'** . Multiplicand **a'** is slowly counted down (its marks erased – shown for reference with x's below). A "counter" inside **b'** moves to the right through **b** (an erased mark shown being read by the head as **[.]**) but is reconstructed after each pass when the head returns from incrementing **c'** :

....**a**.**b**.... = : ....xxx | . | | **[.]** | | . | | | | | | | ...

At end of computation: **c'** is 13 marks = "successor of 12" appearing to the right of **b'** . **a'** has vanished in process of the computation

....**b**.**c** = ... | | | | | . | | | | | | | | | | | | | ...

Footnotes

^ a: Difference between Turing- and Post–Turing machine models

In his chapter XIII *Computable Functions*, Kleene adopts the Post model; Kleene's model uses a blank and one symbol "tally mark ¤" (Kleene p. 358), a "treatment closer in some respects to Post 1936. Post 1936 considered computation with a 2-way infinite tape and only 1 symbol" (Kleene p. 361). Kleene observes that Post's treatment provided a further

reduction to "atomic acts" (Kleene p. 357) of "the Turing act" (Kleene p. 379). As described by Kleene "The Turing act" is the combined 3 (time-sequential) actions specified on a line in a Turing table: (i) print-symbol/erase/do-nothing followed by (ii) move-tape-left/move-tape-right/do-nothing followed by (iii) test-tape-go-to-next-instruction: e.g. "s1Rq1" means "Print symbol "¤", then move tape right, then if tape symbol is "¤" then go to state q1".

Kleene observes that Post atomized these 3-actions further into two types of 2-actions. The first type is a "print/erase" action, the second is a "move tape left/right action": (1.i) print-symbol/erase/do-nothing followed by (1.ii) test-tape-go-to-next-instruction, OR (2.ii) move-tape-left/move-tape-right/do-nothing followed by (2.ii) test-tape-go-to-next-instruction.

But Kleene observes that while

"Indeed it could be argued that the Turing machine act is already compound, and consists psychologically in a printing and change in state of mind, followed by a motion and another state of mind [, and] Post 1947 does thus separate the Turing act into two; we have not here, primarily because it saves space in the machine tables not to do so."(Kleene p. 379)

In fact Post's treatment (1936) is ambiguous; both (1.1) and (2.1) could be followed by "(.ii) go to next instruction in numerical sequence". This represents a further atomization into three types of instructions: (1) print-symbol/erase/do-nothing then go-to-next-instruction-in-numerical-sequence, (2) move-tape-left/move-tape-right/do-nothing then go-to-next-instruction-in-numerical-sequence (3)test-tape then go-to-instruction-xxx-else-go-to-next-instruction-in-numerical-sequence.

Langton's ant

Langton's ant after 11000 steps. A red pixel shows the ant's location.

Langton's ant is a two-dimensional Turing machine with a very simple set of rules but complicated emergent behavior. It was invented by Chris Langton in 1986 and runs on a square lattice of black and white cells. The universality of Langton's ant was proven in 2000. The idea has been generalized in several different ways, such as turmites which add more colors and more states.

Rules

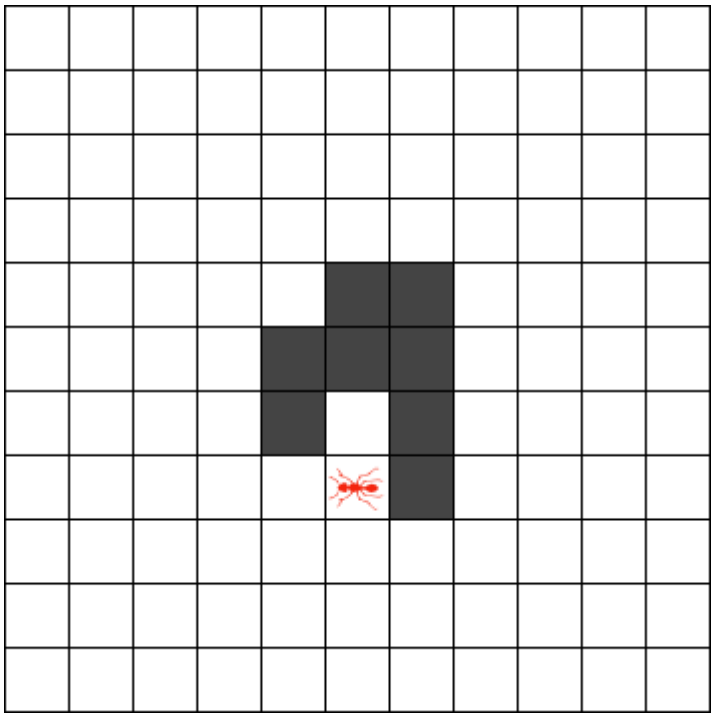


Image of first 200 steps of Langton's ant

Squares on a plane are colored variously either black or white. We arbitrarily identify one square as the "ant". The ant can travel in any of the four cardinal directions at each step it takes. The ant moves according to the rules below:

- At a white square, turn 90° right, flip the color of the square, move forward one unit
- At a black square, turn 90° left, flip the color of the square, move forward one unit

These simple rules lead to surprisingly complex behavior: after an initial period of apparently chaotic behavior, that lasts for about 10,000 steps (in the simplest case), the ant starts building a recurrent "highway" pattern of 104 steps that repeat indefinitely. All finite initial configurations tested eventually converge to the same repetitive pattern suggesting that the "highway" is an attractor of Langton's ant, but no one has been able to prove that this is true for all such initial configurations. It is only known that the ant's trajectory is always unbounded regardless of the initial configuration - this is known as the **Cohen-Kung theorem**.

Langton's ant can also be described as a cellular automaton, where most of the grid is colored black or white, and the "ant" square has one of eight different colors assigned to encode the combination of black/white state and the current direction of motion of the ant.

Universality

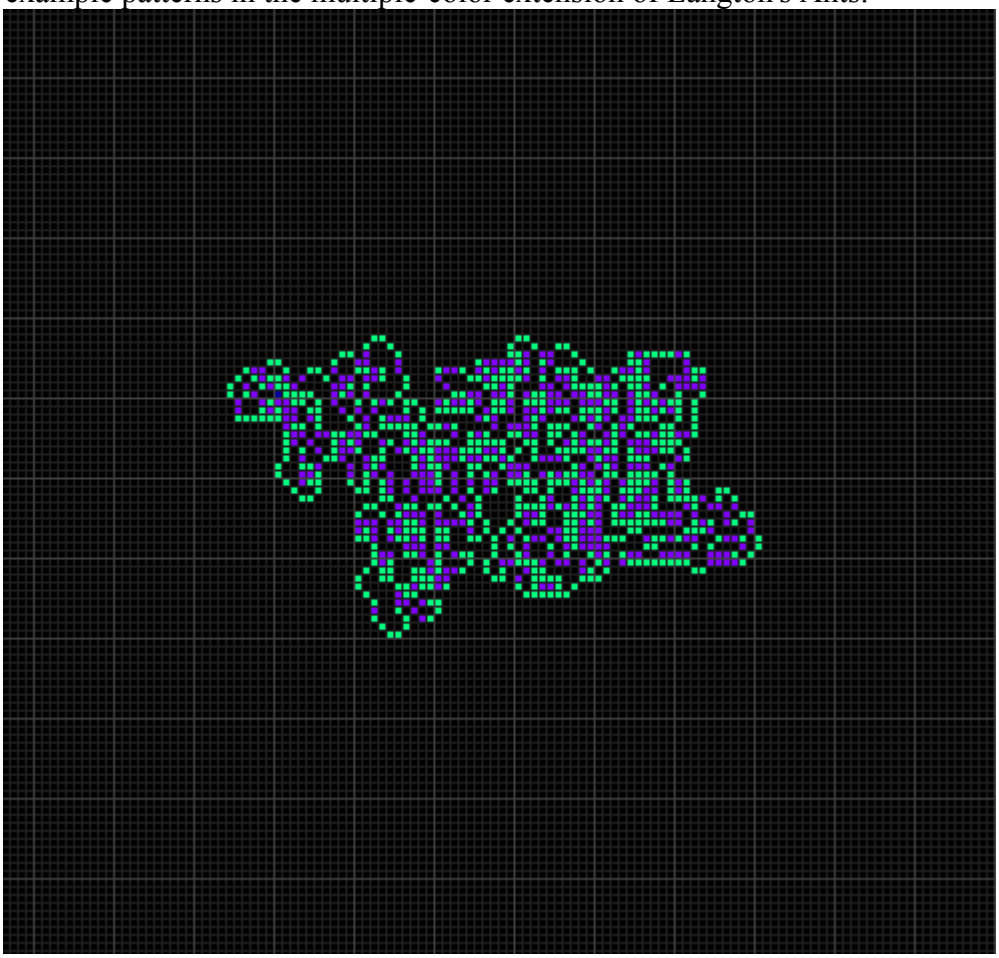
In 2000, Gajardo et al. showed a construction that calculates any boolean circuit using the trajectory of a single instance of Langton's ant. Thus, it would be possible to simulate a Turing machine using the ant's trajectory for computation. This means that the ant is capable of universal computation.

Extension to multiple colors

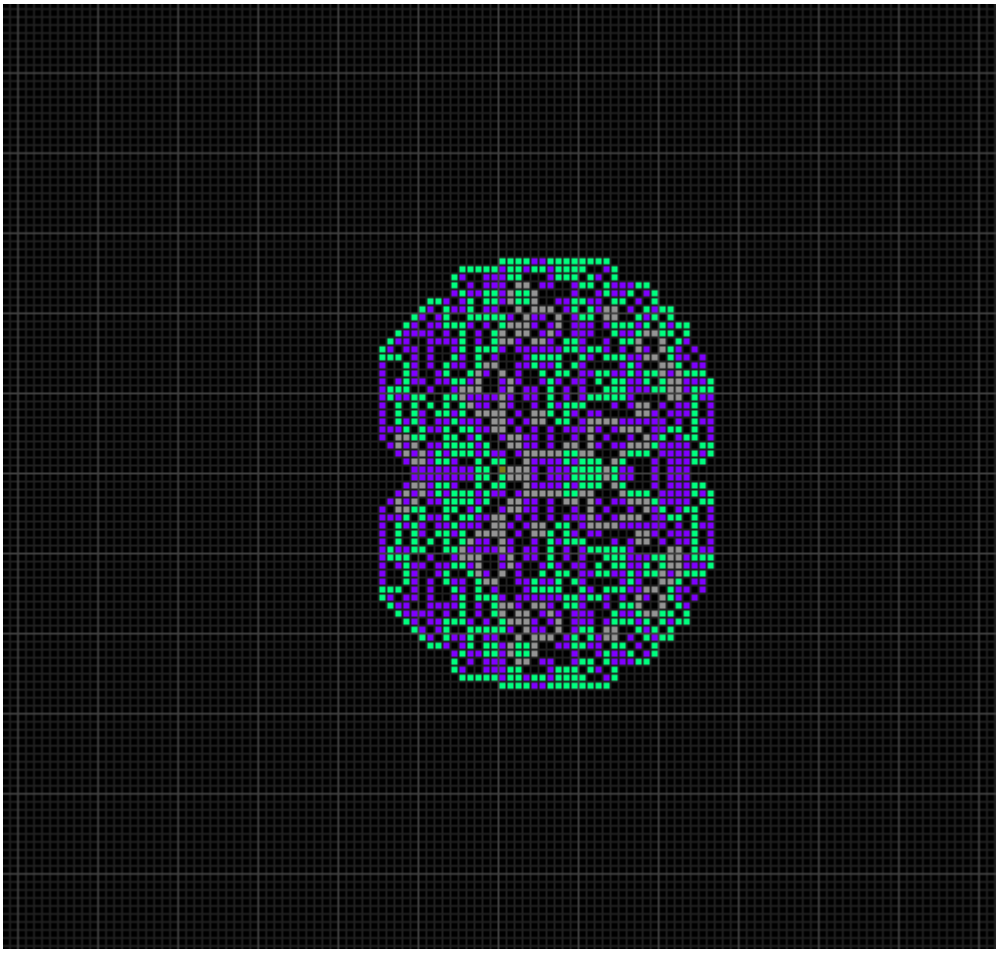
Greg Turk and Jim Propp considered a simple extension to Langton's ant where instead of just two colors, more colors are used. The colors are modified in a cyclic fashion. A simple naming scheme is used: for each of the successive colors, a letter 'L' or 'R' is used to indicate whether a left or right turn should be taken. Langton's ant has the name 'RL' in this naming scheme.

Some of these extended Langton's ants produce patterns that become symmetric over and over again. One of the simplest examples is the ant 'RLLR'. One sufficient condition for this to happen is that the ant's name, seen as a cyclic list, consists of consecutive pairs of identical letters 'LL' or 'RR' (the term "cyclic list" indicates that the last letter may pair with the first one.) The proof involves Truchet tiles.

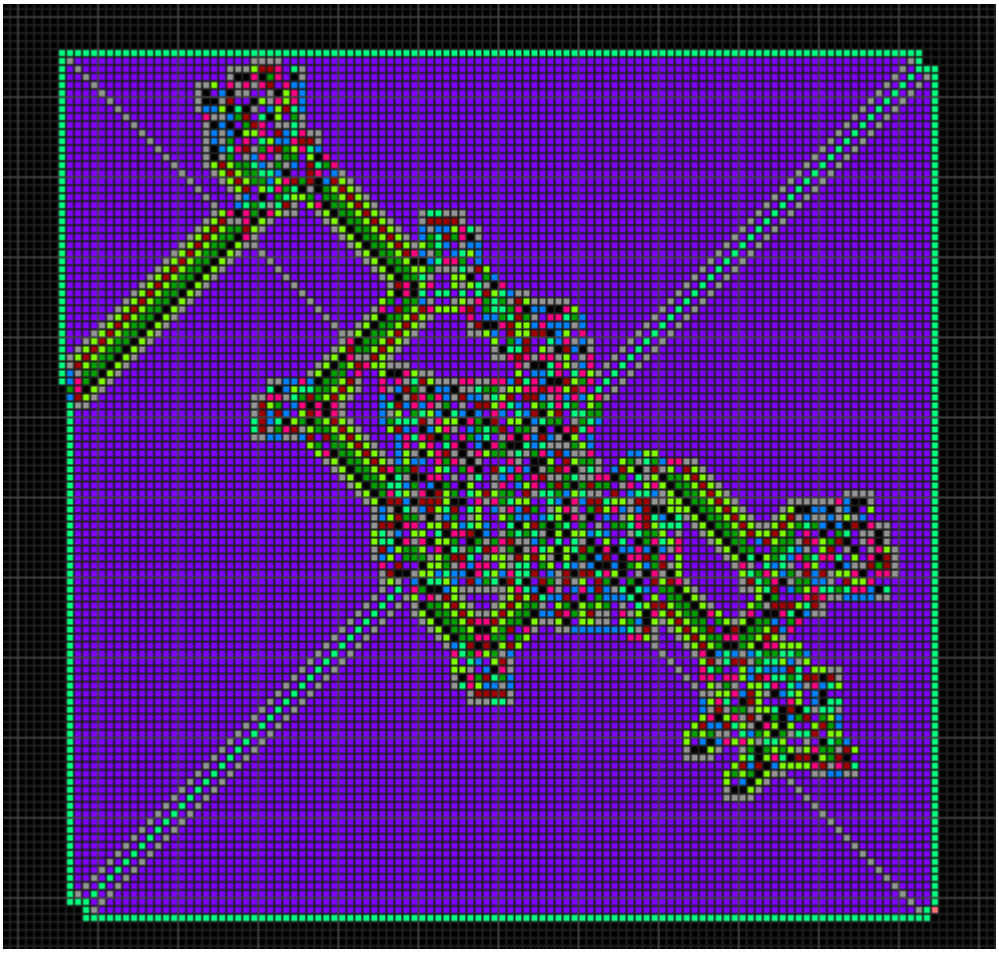
Some example patterns in the multiple-color extension of Langton's Ants:



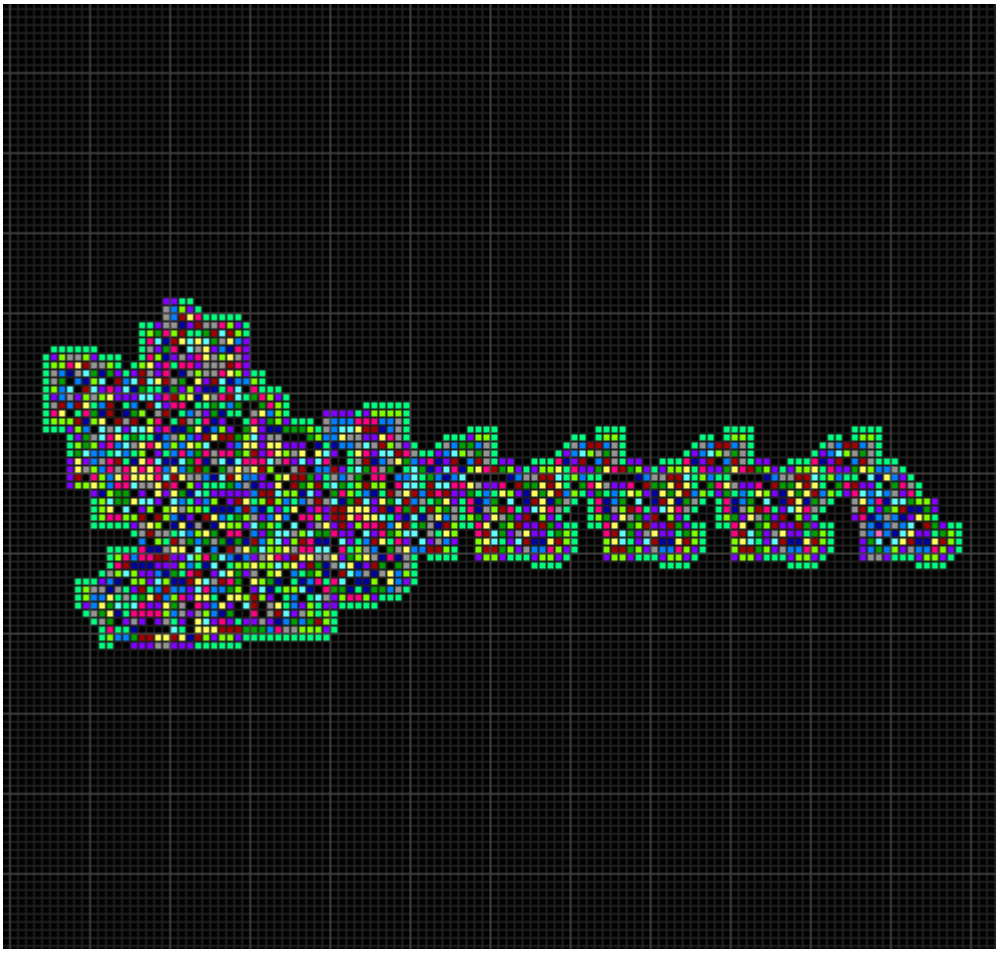
RLR: grows chaotically. It is not known if this ant ever produces a highway.



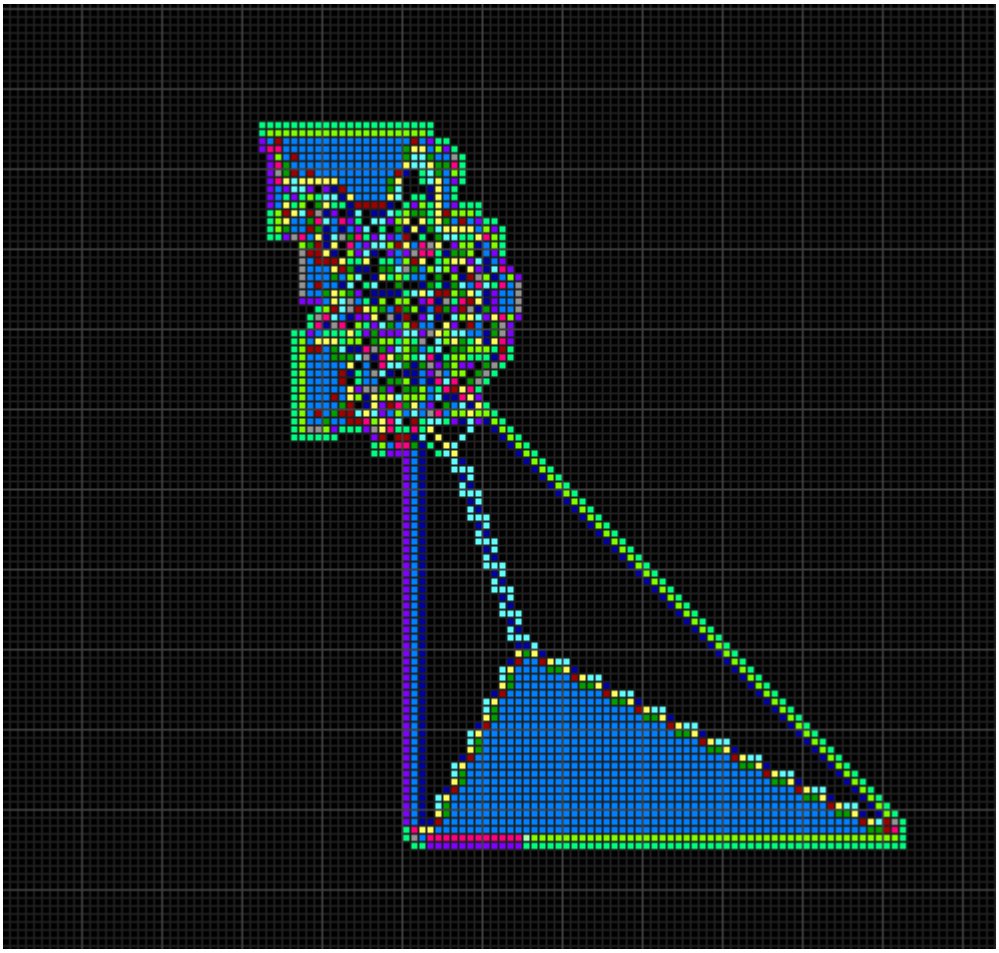
LLRR: grows symmetrically.



LRRRRRLLR: fills space in a square around itself.



LLRRRLRLRLLR: creates a convoluted highway.

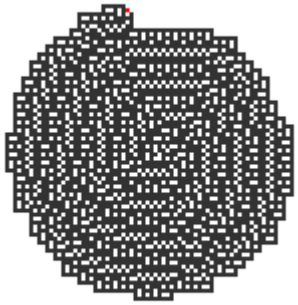


RRLLLRLLLRRR: creates a filled triangle shape that grows and moves.

Extension to multiple states

A further extension of Langton's Ants is to consider multiple states of the Turing machine - as if the ant itself has a color that can change. These ants are called turmites, a contraction of "Turing machine termites". Common behaviours include the production of highways, chaotic growth and spiral growth.

Some example turmites:



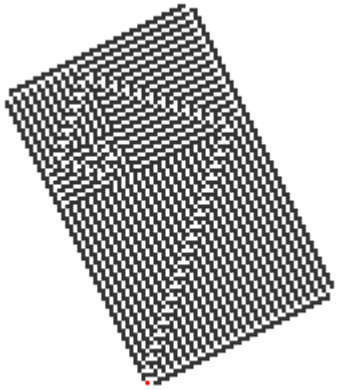
Spiral growth.

Semi-chaotic growth.

Production of a highway after a period of chaotic growth.

Chaotic growth with a distinctive texture.

Growth with a distinctive texture inside an expanding frame.



Constructing a Fibonacci spiral.

Extension to multiple ants

Multiple Langton's ants can co-exist on the 2D plane, as long as there is a rule for what happens when they meet. Ed Pegg, Jr. considered turmites that can turn for example *both* left and right, splitting in two and annihilating each other when they meet.

Chapter 5

Non-Deterministic Turing Machine and Turmite

Non-deterministic Turing machine

In theoretical computer science, a Turing machine is a theoretical machine that is used in thought experiments to examine the abilities and limitations of computers.

In essence, a Turing machine is imagined to be a simple computer that reads and writes symbols one at a time on an endless tape by strictly following a set of rules. It determines what action it should perform next according to its internal "state" and what number it currently sees. An example of one of a Turing Machine's rules might thus be: "If you are in state 2 and you see an 'A', change it to a 'B' and move left."

In a **deterministic Turing machine**, the set of rules prescribes at most one action to be performed for any given situation. A **non-deterministic Turing machine** (**NTM**), by contrast, may have a set of rules that prescribes more than one action for a given situation. For example, a non-deterministic Turing machine may have both "If you are in state 2 and you see an 'A', change it to a 'B' and move left" and "If you are in state 2 and you see an 'A', change it to a 'C' and move right" in its rule set.

An ordinary (deterministic) Turing machine (DTM) has a transition function that, for a given state and symbol under the tape head, specifies three things: the symbol to be written to the tape, the direction (left or right) in which the head should move, and the subsequent state of the finite control. For example, an X on the tape in state 3 might make the DTM write a Y on the tape, move the head one position to the right, and switch to state 5.

A non-deterministic Turing machine (NTM) differs in that the state and tape symbol no longer *uniquely* specify these things; rather, many different actions may apply for the same combination of state and symbol. For example, an X on the tape in state 3 might now allow the NTM to write a Y, move right, and switch to state 5 *or* to write an X, move left, and stay in state 3.

Definition

A nondeterministic Turing machine can be formally defined as a 6-tuple $M = (Q, \Sigma, \iota, \sqcup, A, \delta)$, where

- Q is a finite set of states
- Σ is a finite set of symbols (the tape alphabet)
- $\iota \in Q$ is the initial state
- $\sqcup \in \Sigma$ is the blank symbol
- $A \subseteq Q$ is the set of accepting states
- $\delta \subseteq (Q \backslash A \times \Sigma) \times (Q \times \Sigma \times \{L, R\})$ is a relation on states and symbols called the *transition relation*..

The difference with a standard (deterministic) Turing machine is that for those, the transition relation is a function (the transition function).

Configurations and the *yields* relation on configurations, which describes the possible actions of the Turing machine given any possible contents of the tape, are as for standard Turing machines, except that the yields relation is no longer single-valued. The notion of string acceptance is unchanged: a non-deterministic Turing machine accepts a string if, when the machine is started on the configuration in which the tape head is on the first character of the string (if any), and the tape is all blank otherwise, at least one of the machine's possible computations from that configuration puts the machine into a state in *A*. (If the machine is deterministic, the possible computations are the prefixes of a single, possibly infinite, path.)

Resolution of multiple rules

How does the NTM "know" which of these actions it should take? There are two ways of looking at it. One is to say that the machine is the "luckiest possible guesser"; it always picks the transition which eventually leads to an accepting state, if there is such a transition. The other is to imagine that the machine "branches" into many copies, each of which follows one of the possible transitions. Whereas a DTM has a single "computation path" that it follows, an NTM has a "computation tree". If any branch of the tree halts with an "accept" condition, we say that the NTM accepts the input.

Variations

Equivalence with DTMs

In particular, nondeterministic Turing machines are equivalent with deterministic Turing machines. This equivalency refers to what can be computed, as opposed to how quickly.

NTMs effectively include DTMs as special cases, so it is immediately clear that DTMs are not more powerful. It might seem that NTMs are more powerful than DTMs, since

they can allow trees of possible computations arising from the same initial configuration, accepting a string if any one branch in the tree accepts it.

But it is possible to simulate NTMs with DTMs.

One approach is to use a DTM of which the configurations represent multiple configurations of the NTM, and the DTM's operation consists of visiting each of them in turn, executing a single step at each visit, and spawning new configurations whenever the transition relation defines multiple continuations; this is effectively how a multitasking operating system implements the execution of multiple concurrent processes with a single processor and a single memory array.

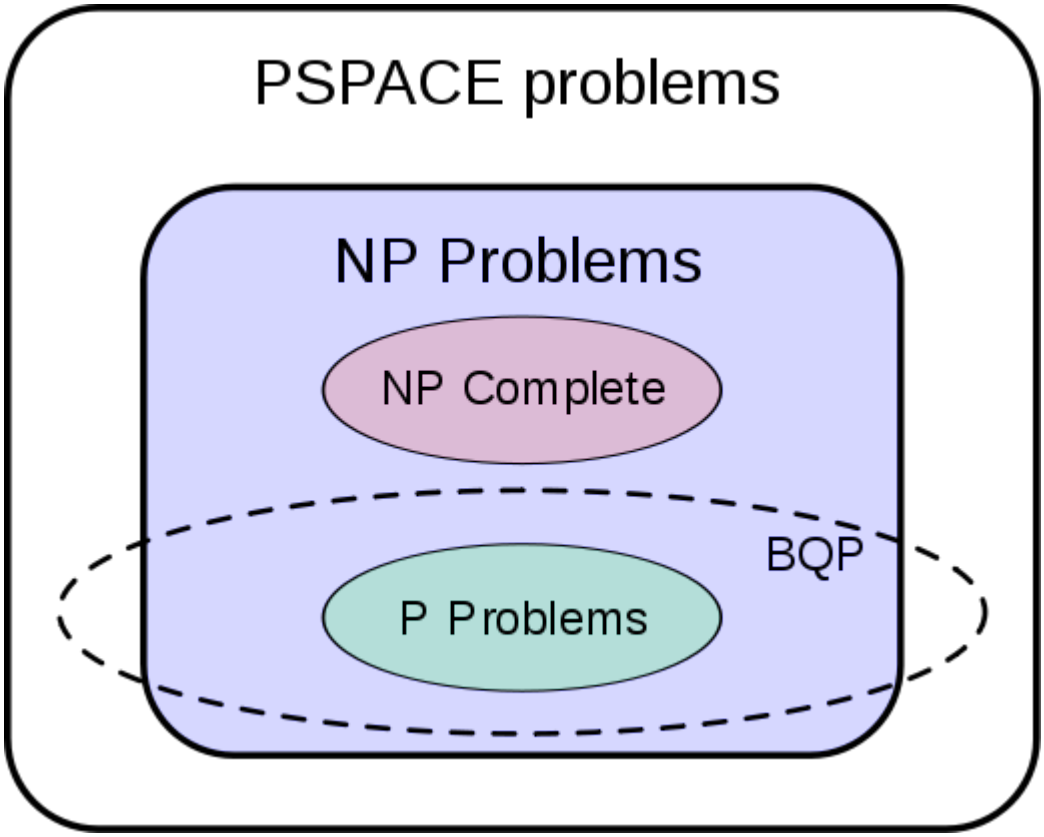
Another construction simulates NTMs with 3-tape DTMs, of which the first tape always holds the original input string, the second is used to simulate a particular computation of the NTM, and the third encodes a path in the NTM's computation tree. The 3-tape DTMs are easily simulated with a normal single-tape DTM.

In this construction, the resulting DTM effectively performs a breadth-first search of the NTM's computation tree, visiting all possible computations of the NTM in order of increasing length until it finds an accepting one. Therefore, the length of an accepting computation of the DTM is, in general, exponential in the length of the shortest accepting computation of the NTM. This is considered to be a general property of simulations of NTMs by DTMs; the most famous unresolved question in computer science, the P = NP problem, is related to this issue.

Bounded non-determinism

An NTM has the property of bounded non-determinism, *i.e.*, if an NTM always halts on a given input tape *T* then it halts in a bounded number of steps, and therefore can only have a bounded number of possible configurations.

Comparison with quantum computers



The suspected shape of the range of problems solvable by quantum computers in polynomial time. Note that the figure suggests $\mathbf{P} \neq \mathbf{NP}$ and $\mathbf{N}P \neq \mathbf{PSPACE}$, if this is not true then the figure should look otherwise.

It is a common misconception that quantum computers are NTMs. It is believed but has not been proven that the power of quantum computers is incomparable to that of NTMs. That is, problems likely exist that an NTM could efficiently solve but that a quantum computer cannot. A likely example of problems solvable by NTMs but not by quantum computers in polynomial time are NP-complete problems.

Turmite

A 2-state 2-color turmite on a square grid. Starting from an empty grid, after 8342 steps the turmite (a red pixel) has exhibited both chaotic and regular movement phases.

In computer science, a **turmite** is a Turing machine which has an orientation as well as a current state and a "tape" that consists of an infinite two-dimensional grid of cells. The terms **ant** and **vant** are also used. Langton's ant is a well-known type of turmite defined on the cells of a square grid. Paterson's worms are a type of turmite defined on the edges of an isometric grid.

It has been shown that turmites in general are exactly equivalent in power to one-dimensional Turing machines with an infinite tape, as either can simulate the other.

History

Langton's ants were invented in 1986 and declared "equivalent to Turing machines". Independently, in 1988, Allen H. Brady considered the idea of two-dimensional Turing machines with an orientation and called them "TurNing machines".

Apparently independently of both of these, Greg Turk investigated the same kind of system and wrote to A. K. Dewdney about them. A. K. Dewdney named them "tur-mites" in his "Computer Recreations" column in Scientific American in 1989. Rudy Rucker relates the story as follows:

Dewdney reports that, casting about for a name for Turk's creatures, he thought, "Well, they're Turing machines studied by Turk, so they should be tur-something. And they're like little insects, or mites, so I'll call them tur-mites! And that sounds like termites!" With the kind permission of Turk and Dewdney, I'm going to leave out the hyphen, and call them turmites.
—Rudy Rucker, *Artificial Life Lab*

Relative vs. Absolute Turmites

Turmites can be categorised as being either *relative* or *absolute*. Relative turmites, alternatively known as 'Turning machines', have an internal orientation. Langton's Ant is such an example. Relative turmites are, by definition, isotropic; rotating the turmite does not affect its outcome. Relative turmites are so named because the directions are encoded *relative* to the current orientation, equivalent to using the words 'left' or 'backwards'. Absolute turmites, by comparison, encode their directions in absolute terms: a particular instruction may direct the turmite to move 'North'. Absolute turmites are two-dimensional analogues of conventional Turing machines, so are occasionally referred to as simply "Two-dimensional Turing machines".

Specification

The following specification is specific to turmites on a two-dimensional square grid, the most studied type of turmite. Turmites on other grids can be specified in a similar fashion.

As with Langton's ant, turmites perform the following operations each timestep:

1. turn on the spot (by some multiple of 90°)
2. change the color of the square
3. move forward one square

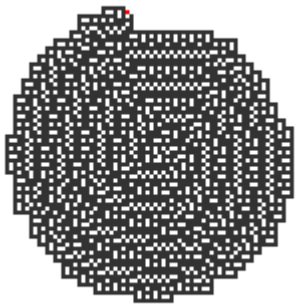
As with Turing machines, the actions are specified by a state transition table listing the current internal state of the turmite and the color of the cell it is currently standing on. For example, the turmite shown in the image at the top of this page is specified by the following table:

		Current color					
		0			**1**		
		Write color	**Turn**	**Next state**	**Write color**	**Turn**	**Next state**
Current state	**0**	1	R	0	1	R	1
	1	0	N	0	0	N	1

The direction to turn is one of **L** (90° left), **R** (90° right), **N** (no turn) and **U** (180° U-turn).

Examples

Some example 2-state 2-color turmites on a square grid, all starting from an empty configuration:

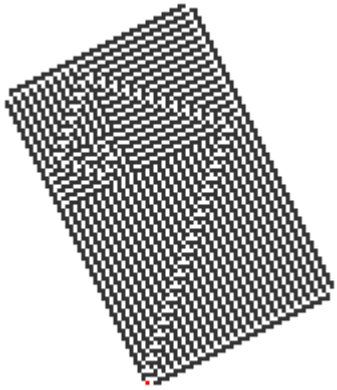


Spiral growth.

Production of a highway after a period of chaotic growth.

Chaotic growth with a distinctive texture.

Growth with a distinctive texture inside an expanding frame.



Constructing a Fibonacci spiral.

Starting from an empty grid or other configurations, the most commonly observed behaviours are chaotic growth, spiral growth and 'highway' construction. Rare examples become periodic after a certain number of steps.

Turmites and the Busy Beaver game

Allen H. Brady searched for terminating turmites (the equivalent of busy beavers) and found a 2-state 2-color machine that printed 37 1's before halting, and another that took 121 steps before halting. He also considered turmites that move on a triangular grid, finding several busy beavers here too.

Ed Pegg, Jr. considered another approach to the busy beaver game. He suggested turmites that can turn for example *both* left and right, splitting in two. Turmites that later meet annihilate each other. In this system, a Busy Beaver is one that from a starting pattern of a single turmite lasts the longest before all the turmites annihilate each other.

Other grids

Following Allen H. Brady's initial work of turmites on a triangular grid, hexagonal tilings have also been explored. Much of this work is due to Tim Hutton, and his results are on the Rule Table Repository. He has also considered Turmites in three dimensions, and collected some preliminary results. Allen H. Brady and Tim Hutton have also investigated one-dimensional relative turmites on the integer lattice, which Brady termed *flippers*. (One-dimensional *absolute* turmites are of course simply known as Turing machines.)

Chapter 6

Busy Beaver

In computability theory, a **busy beaver** (from the colloquial expression for "industrious person") is a Turing machine that attains the maximum "operational busyness" (such as measured by the number of steps performed, or the number of nonblank symbols finally on the tape) among all the Turing machines in a certain class. The Turing machines in this class must meet certain design specifications and are required to eventually halt after being started with a blank tape.

A **busy beaver function** quantifies these upper limits on a given type of "operational busyness", and is a noncomputable function. In fact, a busy beaver function can be shown to grow faster asymptotically than does any computable function. The concept was first introduced by Tibor Radó as the "busy beaver game" in his 1962 paper, "On Non-Computable Functions".

The busy beaver game

The ***n*-state busy beaver game** (or **BB-*n* game**), introduced in Tibor Radó's 1962 paper, involves a class of Turing machines, each of which is required to meet the following design specifications:

- The machine has *n* "operational" states plus a Halt state, where *n* is a positive integer, and one of the *n* states is distinguished as the starting state. (Typically, the states are labelled by 1, 2, ..., *n*, with state 1 as the starting state, or by *A*, *B*, *C*, ..., with state *A* as the starting state.)
- The machine uses a single two-way infinite (or unbounded) tape.
- The tape alphabet is {0, 1}, with 0 serving as the blank symbol.
- The machine's *transition function* takes two inputs:

    1. the current non-Halt state,
    2. the symbol in the current tape cell,

    and produces three outputs:

1. a symbol to write over the one in the current tape cell (it may be the same symbol as the one overwritten),
2. a direction to move (left or right; that is, shift to the tape cell one place to the left or right of the current cell), and
3. a state to transition into (which may be the Halt state).

The transition function may be seen as a finite table of 5-tuples, each of the form (current state, current symbol, symbol to write, direction of shift, next state).

"Running" the machine consists of starting in the starting state, with the current tape cell being any cell of a blank (all-0) tape, and then iterating the transition function until the Halt state is entered (if ever). If, and only if, the machine eventually halts, then the number of 1s finally remaining on the tape is called the machine's *score*.

The *n*-state busy beaver (BB-*n*) game is a contest to find such an *n*-state Turing machine having the largest possible score — the largest number of 1s on its tape after halting. A machine that attains the largest possible score among all *n*-state Turing machines is called an *n*-state **busy beaver**, and a machine whose score is merely the highest so far attained (perhaps not the largest possible) is called a *champion n*-state machine.

Radó required that each machine entered in the contest be accompanied by a statement of the exact number of steps it takes to reach the Halt state, thus allowing the score of each entry to be verified (in principle) by running the machine for the stated number of steps. (If entries were to consist only of machine descriptions, then the problem of verifying every potential entry is undecidable, because it is equivalent to the well-known halting problem — there would be no effective way to decide whether an arbitrary machine eventually halts.)

The busy beaver function Σ

The busy beaver function, $\Sigma: N \rightarrow N$, is defined such that $\Sigma(n)$ is the maximum attainable score (the maximum number of 1s finally on the tape) among all halting 2-symbol *n*-state Turing machines of the above-described type, when started on a blank tape.

Radó showed that Σ is a well-defined function; that is, for every *n*, the *n*-state busy beaver game does indeed have a maximum attainable score. To do this, he used the basic principle that a nonempty finite set of nonnegative integers must have a largest element:

There are finitely many Turing machines with *n* states and 2 symbols (specifically there are $[4(n+1)]^{2n}$ of them). In addition it is trivial that some of these are halting machines; i.e., there exists at least one *n*-state, 2-symbol Turing Machine that will halt, for every n. Now define:

- E_n is the finite, non-empty set of halting *n*-state 2-symbol Turing machines of the type described in the preceding section (two-way infinite tape, transition function defined by 5-tuples, etc.).

- σ(*M*), for each Turing machine *M* in E_n, is the *score* of machine *M* — the number of 1s finally on the tape after machine *M* is run to completion with an initially blank tape.
- $\Sigma(n) = \max\{\sigma(M)|M \in E_n\}$ (i.e., the maximum score among all n-state 2-symbol halting Turing machines started with a blank tape).

Since σ(*M*) is a non-negative integer for any *M* in E_n, and since E_n is a non-empty finite set, Σ(*n*) is a well-defined non-negative integer for every nonnegative integer *n*.

This infinite sequence **Σ** is the **busy beaver function**, and any *n*-state 2-symbol Turing machine *M* for which σ(*M*) = Σ(*n*) (i.e., which attains the maximum score) is called a **busy beaver**. Note that for each *n*, there exist at least two *n*-state busy beavers (because, given any *n*-state busy beaver, another is obtained by merely changing the shift direction in a halting transition).

Non-computability of Σ

Radó's 1962 paper proved that if *f*: N → N is any computable function, then Σ(*n*) > *f(n)* for all sufficiently large *n*, and hence that Σ is not a computable function.

Moreover, this implies that it is undecidable by a general algorithm whether an arbitrary Turing machine is a busy beaver. (Such an algorithm cannot exist, because its existence would allow Σ to be computed, which is a proven impossibility. In particular, such an algorithm could be used to construct another algorithm that would compute Σ as follows: for any given *n*, each of the finitely many *n*-state 2-symbol Turing machines would be tested until an *n*-state busy beaver is found; this busy beaver machine would then be simulated to determine its score, which is by definition Σ(*n*).)

A noteworthy fact is that, theoretically, every *finite* sequence of Σ values, such as Σ(0), Σ(1), Σ(2), ..., Σ(*n*) for any given *n*, *is* (trivially) computable, even though the *infinite* sequence Σ is not computable. Furthermore, for sufficiently small *n*, it is also practical to compute Σ(*n*). Thus, it is not hard to show that Σ(0) = 0, Σ(1) = 1, Σ(2) = 4, and with progressively more difficulty it can be shown that Σ(3) = 6 and Σ(4) = 13 (sequence A028444 in OEIS). Σ(*n*) has not yet been determined for any instance of *n* > 4, although lower bounds have been established.

Σ, complexity and unprovability

A variant of Kolmogorov complexity is defined as follows [cf. Boolos, Burgess & Jeffrey, 2007]: The *complexity* of a number *n* is the smallest number of states needed for a BB-class Turing machine that halts with a single block of *n* consecutive 1s on an initially blank tape. The corresponding variant of Chaitin's incompleteness theorem states that, in the context of a given axiomatic system for the natural numbers, there exists a number *k* such that no specific number can be proved to have complexity greater than *k*, and hence that no specific upper bound can proven for Σ(*k*). (The latter is because "the complexity of *n* is greater than *k*" would be proved if "*n* > Σ(*k*)" were proved.) As

mentioned in the cited reference, for any axiomatic system of "ordinary mathematics" the least value *k* for which this is true is far less than 10↑↑10 (using Knuth up-arrow notation); consequently, *in the context of ordinary mathematics, neither the value nor any upper-bound of Σ(10↑↑10) can be proven*. (Gödel's first incompleteness theorem is illustrated by this result: in an axiomatic system of ordinary mathematics, there is a true-but-unprovable sentence of the form "Σ(10↑↑10) = *n*", and there are infinitely many true-but-unprovable sentences of the form "Σ(10↑↑10) < *n*".)

Max shifts function

In addition to the function Σ, Radó [1962] introduced another extreme function for the BB-class of Turing machines, the **maximum shifts function**, *S*, defined as follows:

- $s(M) =$ the number of shifts *M* makes before halting, for any *M* in E_n,
- $S(n) = \max\{s(M)|M \in E_n\} =$ the largest number of shifts made by any halting *n*-state 2-symbol Turing machine.

Because these Turing machines are required to have a shift in each and every transition or "step" (including any transition to a Halt state), the max-shifts function is at the same time a max-steps function.

Radó showed that *S* is noncomputable for the same reason that Σ is noncomputable — it grows faster than any computable function. He proved this simply by noting that for each *n*, $S(n) \geq \Sigma(n)$, because a shift is required to write a 1 on the tape; consequently, *S* grows at least as fast as Σ, which had already been proved to grow faster than any computable function.

The following connection between Σ and *S* was used by Lin & Radó [*Computer Studies of Turing Machine Problems*, 1965] to prove that Σ(3) = 6: For a given *n*, if *S*(*n*) is known then all *n*-state Turing machines can (in principle) be run for up to *S*(*n*) steps, at which point any machine that hasn't yet halted will never halt. At that point, by observing which machines have halted with the most 1s on the tape (i.e., the busy beavers), one obtains from their tapes the value of Σ(*n*). The approach used by Lin & Radó for the case of *n* = 3 was to conjecture that *S*(3) = 21, then to simulate all the essentially different 3-state machines for up to 21 steps. By analyzing the behavior of the machines that had not halted within 21 steps, they succeeded in showing that none of those machines would ever halt, thus proving the conjecture that *S*(3) = 21, and determining that Σ(3) = 6 by the procedure just described.

Inequalities relating Σ and *S* include the following (from [Ben-Amram, et al., 1996]), which are valid for all $n \geq 1$:

$$S(n) \geq \Sigma(n)$$
$$S(n) \leq (2n-1)\Sigma(3n+3)$$
$$S(n) < \Sigma(3n+6);$$

and an asymptotically improved bound (from [Ben-Amram, Petersen, 2002]): there exists a constant c, such that for all $n \geq 2$,

$$S(n) \leq \Sigma(n + \lceil 8n/\log_2 n \rceil + c).$$

Known values

The function values for $\Sigma(n)$ and $S(n)$ are only known exactly for $n < 5$. The current 5-state busy beaver champion produces 4,098 1s, using 47,176,870 steps (discovered by Heiner Marxen and Jürgen Buntrock in 1989), but there remain about 40 machines with non-regular behavior which are believed to never halt, but which have not yet been proven to run infinitely. At the moment the record 6-state busy beaver produces over 10^{18267} 1s, using over 10^{36534} steps (found by Pavel Kropitz in 2010). As noted above, these busy beavers are 2-symbol Turing machines.

Milton Green, in his 1964 paper "A Lower Bound on Rado's Sigma Function for Binary Turing Machines", constructed a set of Turing machines demonstrating that

$$\Sigma(2k) > 3 \uparrow^{k-2} 3 > A(k-2, k-2) \quad (k \geq 2),$$

where $\uparrow$ is Knuth up-arrow notation and A is Ackermann's function.

Thus

$$\Sigma(10) > 3 \uparrow\uparrow\uparrow 3 = 3 \uparrow\uparrow 3^{3^3} = 3^{3^{3^{\cdot^{\cdot^{\cdot^3}}}}}$$

(with 3^{27} = 7,625,597,484,987 terms in the exponential tower), and

$$\Sigma(12) > 3 \uparrow\uparrow\uparrow\uparrow 3 = g_1,$$

where the number g_1 is the enormous starting value in the sequence that defines Graham's number.

In contrast, the current bound on $\Sigma(6)$ is 10^{18267}, which is less than 3^{333} (tiny in comparison).

Generalizations

For any model of computation there exist simple analogs of the busy beaver. For example, the generalization to Turing machines with n states and m symbols defines the following **generalized busy beaver functions**:

1. $\Sigma(n, m)$: the largest number of non-zeros printable by an n-state, m-symbol machine started on an initially blank tape before halting, and

2. $S(n, m)$: the largest number of steps taken by an n-state, m-symbol machine started on an initially blank tape before halting.

For example the longest running 3-state 3-symbol machine found so far runs 119,112,334,170,342,540 steps before halting. The longest running 6-state, 2-symbol machine which has the additional property of reversing the tape value at each step produces 6,147 1s after 47,339,970 steps. So $S_{RTM}(6) \geq 47{,}339{,}970$ and $\Sigma_{RTM}(6) \geq 6{,}147$.

Likewise we could define an analog to the Σ function for register machines as the largest number which can be present in any register on halting, for a given number of instructions.

Applications

In addition to posing a rather challenging mathematical game, the busy beaver functions have a profound application. Many open problems in mathematics could be solved in a systematic way given the value of $S(n)$ for a sufficiently large n.

Consider any conjecture that could be disproven via a counterexample among a countable number of cases (e.g. Goldbach's conjecture). Write a computer program that sequentially tests this conjecture for increasing values (in the case of Goldbach's conjecture, we would consider every even number ≥ 4 sequentially and test whether or not it is the sum of two prime numbers). We will consider this program to be simulated by an n-state Turing machine (although we could alternatively define the busy beaver function for any well-defined programming language). If it finds a counterexample (an even number ≥ 4 that is not the sum of 2 primes in our example), it halts and notifies us. However, if the conjecture is true, then our program will never halt. (This program halts *only* if it finds a counterexample.)

Now, this program is simulated by an n-state Turing machine, so if we know $S(n)$ we can decide (in a finite amount of time) whether or not it will ever halt by simply running the machine that many steps. And if, after $S(n)$ steps, the machine does not halt, we know that it never will and thus that there are no counterexamples to the given conjecture (i.e., no even numbers that are not the sum of two primes). This would prove the conjecture to be true.

Thus specific values (or upper bounds) for $S(n)$ could be used to systematically solve many open problems in mathematics (in theory). However, current results on the busy beaver problem suggest that this will not be practical for two reasons:

- It is extremely hard to prove values for the busy beaver function (and the max shift function). It has only been proven for extremely small machines with fewer than 5 states, while one would presumably need at least 20-50 states to make a useful machine. Furthermore, every known exact value of $S(n)$ was proven by enumerating every n-state Turing machine and proving whether or not each halts.

One would have to calculate $S(n)$ by some less direct method for it to actually be useful.

- But even if one did find a better way to calculate $S(n)$, the values of the busy beaver function (and max shift function) get very large, very fast. $S(6) > 10^{36534}$ already requires special pattern-based acceleration to be able to simulate to completion. Likewise, we know that $S(10) > \Sigma(10) > 3 \uparrow\uparrow\uparrow 3$ is a gigantic number. Thus, even if we knew, say, $S(30)$, it is completely unreasonable to run any machine that number of steps. There is not enough computational capacity in the known universe to have performed even $S(6)$ operations.

Proof for uncomputability of S(n) and Σ(n)

Suppose that $S(n)$ is a computable function and let *EvalS* denote a TM, evaluating $S(n)$. Given a tape with n 1s it will produce $S(n)$ 1s on the tape and then halt. Let *Clean* denote a Turing machine cleaning the sequence of 1s initially written on the tape. Let *Double* denote a Turing machine evaluating function $n + n$. Given a tape with n 1s it will produce $2n$ 1s on the tape and then halt. Let us create the composition *Double* | *EvalS* | *Clean* and let n_0 be the number of states of this machine. Let *Create_n₀* denote a Turing machine creating n_0 1s on an initially blank tape. This machine may be constructed in a trivial manner to have n_0 states (the state i writes 1, moves the head right and switches to state $i + 1$, except the state n_0, which halts). Let N denote the sum $n_0 + n_0$.

Let *BadS* denote the composition *Create_n₀* | *Double* | *EvalS* | *Clean*. Notice that this machine has N states. Starting with an initially blank tape it first creates a sequence of n_0 1s and then doubles it, producing a sequence of N 1s. Then *BadS* will produce $S(N)$ 1s on tape, and at last it will clear all 1s and then halt. But the phase of cleaning will continue at least $S(N)$ steps, so the time of working of *BadS* is strictly greater than $S(N)$, which contradicts to the definition of the function $S(n)$.

The uncomputability of $\Sigma(n)$ may be proved in a similar way. In the above proof, one must exchange the machine *EvalS* with *EvalΣ* and *Clean* with *Increment* - a simple TM, searching for a first 0 on the tape and replacing it with 1.

The uncomputability of $S(n)$ can also be trivially established by reference to the halting problem. As $S(n)$ is the maximum number of steps that can be performed by a halting Turing machine, any machine which runs for more steps must be non-halting. One could then determine whether a given Turing machine with n states halts by running it for $S(n)$ steps; if it has still not halted, it never will. As being able to compute $S(n)$ would provide a solution to the provably uncomputable halting problem, it follows that $S(n)$ must likewise be uncomputable.

Examples of busy beaver Turing machines

For an example of a 3-state busy beaver's state table and its "run" see Turing machine examples.

These are tables of rules for the Turing machines that generate Σ(1) and *S*(1), Σ(2) and *S*(2), Σ(3) (but not *S*(3)), Σ(4) and *S*(4), and the best known lower bound for Σ(5) and *S*(5), and Σ(6) and *S*(6).

In the tables, columns represent the current state and rows represent the current symbol read from the tape. Each table entry is a string of three characters, indicating the symbol to write onto the tape, the direction to move, and the new state (in that order). The Halt state is shown as **H**.

Each machine begins in state A with an infinite tape that contains all 0s. Thus, the initial symbol read from the tape is a 0.

Result Key: (starts at the position underlined, halts at the position **in bold**)

1-state, 2-symbol busy beaver

	A
0	1R**H**
1	(not used)

Result: 0 0 1 **0** 0 (1 step, one "1" total)

2-state, 2-symbol busy beaver

	A	**B**
0	1R**B**	1L**A**
1	1L**B**	1R**H**

Result: 0 0 1 1 **1** 1 0 0 (6 steps, four "1"s total)

3-state, 2-symbol busy beaver

	A	**B**	**C**
0	1R**B**	0R**C**	1L**C**
1	1L**C**	1R**B**	1R**H**

Result: 0 0 1 1 1 **1** 1 1 0 0 (14 steps, six "1"s total).

Note that unlike the previous machines, this one is a busy beaver only for Σ, but not for S. (S(3) = 21.)

4-state, 2-symbol busy beaver

	A	**B**	**C**	**D**
0	1R**B**	1L**A**	1R**H**	1R**D**
1	1L**B**	0L**C**	1L**D**	0R**A**

Result: 0 0 1 **0** 1 1 1 1 1 1 1 1 1 1 1 1 0 0 (107 steps, thirteen "1"s total)

current 5-state, 2-symbol best contender (possible busy beaver)

	A	B	C	D	E
0	1R**B**	1R**C**	1R**D**	1L**A**	1R**H**
1	1L**C**	1R**B**	0L**E**	1L**D**	0L**A**

Result: 4098 "1"s with 8191 "0"s interspersed in 47,176,870 steps.

current 6-state, 2-symbol best contender

	A	B	C	D	E	F
0	1R**B**	1R**C**	1L**D**	1R**E**	1L**A**	1L**H**
1	1L**E**	1R**F**	0R**B**	0L**C**	0R**D**	1R**C**

Result: $\approx 3.515 \times 10^{18267}$ "1"s in $\approx 7.412 \times 10^{36534}$ steps.

Exact values and lower bounds for some S(n, m) *and Σ*(n, m)

The following table lists the exact values and some known lower bounds for $S(n, m)$ and $\Sigma(n, m)$ for the generalized busy beaver problems. Known exact values are shown as plain integers and known lower bounds are preceded by a greater than or equal to ($\geq$) symbol. Note: entries listed as "???" are bounded from below by the maximum of all entries to left and above. These machines either haven't been investigated or were subsequently surpassed by a machine preceding them.

The Turing machines that achieve these values are available on both Heiner Marxen's and Pascal Michel's webpages. Each of these websites also contains some analysis of the Turing machines and references to the proofs of the exact values.

Values of S(*n*,*m*):

	2-state	3-state	4-state	5-state	6-state
2-symbol	6	21	107	$\geq$ 47,176,870	$\geq 7.4 \times 10^{36534}$
3-symbol	38	$\geq$ 119,112,334,170,342,540	$\geq 1.0 \times 10^{14072}$	???	???
4-symbol	$\geq$ 3,932,964	$\geq 5.2 \times 10^{13036}$	???	???	???
5-symbol	$\geq 1.9 \times 10^{704}$	???	???	???	???
6-symbol	$\geq 2.4 \times 10^{9866}$	???	???	???	???

Values of $\Sigma(n,m)$:

	2-state	**3-state**	**4-state**	**5-state**	**6-state**
2-symbol	4	6	13	$\geq 4{,}098$	$\geq 3.5 \times 10^{18267}$
3-symbol	9	$\geq 374{,}676{,}383$	$\geq 1.3 \times 10^{7036}$	???	???
4-symbol	$\geq 2{,}050$	$\geq 3.7 \times 10^{6518}$	???	???	???
5-symbol	$\geq 1.7 \times 10^{352}$	???	???	???	???
6-symbol	$\geq 1.9 \times 10^{4933}$	???	???	???	???

Chapter 7

Halting Problem

In computability theory, the **halting problem** is a decision problem which can be stated as follows: Given a description of a program, decide whether the program finishes running or continues to run, and will thereby run forever. This is equivalent to the problem of deciding, given a program and an input, whether the program will eventually halt when run with that input, or will run forever.

Alan Turing proved in 1936 that a general algorithm to solve the halting problem for *all* possible program-input pairs cannot exist. We say that the halting problem is *undecidable* over Turing machines.

B. Jack Copeland (2004) attributes the actual term *halting problem* to Martin Davis.

Formal statement

The halting problem is a decision problem about properties of computer programs on a fixed Turing-complete model of computation. The problem is to determine, given a program and an input to the program, whether the program will eventually halt when run with that input. In this abstract framework, there are no resource limitations of memory or time on the program's execution; it can take arbitrarily long, and use arbitrarily much storage space, before halting. The question is simply whether the given program will ever halt on a particular input.

For example, in pseudocode, the program

```
while True: continue
```

does not halt; rather, it goes on forever in an infinite loop. On the other hand, the program

```
print "Hello World!"
```

halts very soon.

The halting problem is famous because it was one of the first problems proven algorithmically undecidable. This means there is no algorithm which can be applied to

any arbitrary program and input to decide whether the program stops when run with that input.

Representing the halting problem as a set

The conventional representation of decision problems is the set of objects possessing the property in question. The **halting set**

$$K := \{ (i, x) \mid \text{program } i \text{ will eventually halt if run with input } x\}$$

represents the halting problem.

This set is recursively enumerable, which means there is a computable function that lists all of the pairs (i, x) it contains. However, the complement of this set is not recursively enumerable.

There are many equivalent formulations of the halting problem; any set whose Turing degree equals that of the halting problem is such a formulation. Examples of such sets include:

- $\{ i \mid$ program i eventually halts when run with input 0 $\}$
- $\{ i \mid$ there is an input x such that program i eventually halts when run with input x $\}$.

Importance and consequences

The historical importance of the halting problem lies in the fact that it was one of the first problems to be proved undecidable. (Turing's proof went to press in May 1936, whereas Alonzo Church's proof of the undecidability of a problem in the lambda calculus had already been published in April 1936.) Subsequently, many other such problems have been described; the typical method of proving a problem to be undecidable is with the technique of *reduction*. To do this, the computer scientist shows that if a solution to the new problem were found, it could be used to decide an undecidable problem (by transforming instances of the undecidable problem into instances of the new problem). Since we already know that *no* method can decide the old problem, no method can decide the new problem either.

One such consequence of the halting problem's undecidability is that there cannot be a general algorithm that decides whether a given statement about natural numbers is true or not. The reason for this is that the proposition stating that a certain algorithm will halt given a certain input can be converted into an equivalent statement about natural numbers. If we had an algorithm that could solve every statement about natural numbers, it could certainly solve this one; but that would determine whether the original program halts, which is impossible, since the halting problem is undecidable.

Yet another consequence of the undecidability of the halting problem is Rice's theorem which states that the truth of *any* non-trivial statement about the function that is defined by an algorithm is undecidable. So, for example, the decision problem "will this algorithm halt for the input 0" is already undecidable. Note that this theorem holds for the *function defined by the algorithm* and not the algorithm itself. It is, for example, quite possible to decide if an algorithm will halt within 100 steps, but this is not a statement about the function that is defined by the algorithm.

Gregory Chaitin has defined a halting probability, represented by the symbol Ω, a type of real number that informally is said to represent the probability that a randomly produced program halts. These numbers have the same Turing degree as the halting problem. It is a normal and transcendental number which can be defined but cannot be completely computed. This means one can prove that there is no algorithm which produces the digits of Ω, although its first few digits can be calculated in simple cases.

While Turing's proof shows that there can be no general method or algorithm to determine whether algorithms halt, individual instances of that problem may very well be susceptible to attack. Given a specific algorithm, one can often show that it must halt for any input, and in fact computer scientists often do just that as part of a correctness proof. But each proof has to be developed specifically for the algorithm at hand; there is no *mechanical, general way* to determine whether algorithms on a Turing machine halt. However, there are some heuristics that can be used in an automated fashion to attempt to construct a proof, which succeed frequently on typical programs. This field of research is known as automated termination analysis.

Since the negative answer to the halting problem shows that there are problems that cannot be solved by a Turing machine, the Church–Turing thesis limits what can be accomplished by any machine that implements effective methods. However, not all theoretically possible machines are subject to the Church–Turing thesis (e.g. oracle machines are not). It is an open empirical question whether there are actual deterministic physical processes that, in the long run, elude simulation by a Turing machine. It's also an open question whether any such process could usefully be harnessed in the form of a calculating machine (a hypercomputer) that could solve the halting problem for a Turing machine amongst other things. It is also an open empirical question whether any such physical processes are involved in the working of the human brain, thus whether humans can solve the halting problem.

Sketch of proof

The proof shows there is no total computable function that decides whether an arbitrary program *i* halts on arbitrary input *x*; that is, the following function *h* is not computable:

$$h(i,x) = \begin{cases} 1 & \text{if program } i \text{ halts on input } x, \\ 0 & \text{otherwise.} \end{cases}$$

Here *program i* refers to the *i* th program in an enumeration of all the programs of a fixed Turing-complete model of computation.

f(*i*,*j*)		i 1	i 2	i 3	i 4	i 5	i 6
j	1	1	0	0	1	0	1
j	2	0	0	0	1	0	0
j	3	0	1	0	1	0	1
j	4	1	0	0	1	0	0
j	5	0	0	0	1	1	1
j	6	1	1	0	0	1	0

f(*i*,*i*)		1	0	0	1	1	0
g(*i*)		U	0	0	U	U	0

Possible values for a total computable function *f* arranged in a 2D array. The orange cells are the diagonal. The values of *f*(*i*,*i*) and *g*(*i*) are shown at the bottom; *U* indicates that the function *g* is undefined for a particular input value.

The proof proceeds by directly establishing that every total computable function with two arguments differs from the required function *h*. To this end, given any total computable binary function *f*, the following partial function *g* is also computable by some program *e*:

$$g(i) = \begin{cases} 0 & \text{if } f(i,i) = 0, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The verification that *g* is computable relies on the following constructs (or their equivalents):

- computable subprograms (the program that computes *f* is a subprogram in program *e*),
- duplication of values (program *e* computes the inputs *i*,*i* for *f* from the input *i* for *g*),

- conditional branching (program *e* selects between two results depending on the value it computes for *f*(*i*,*i*)),
- not producing a defined result (for example, by looping forever),
- returning a value of 0.

The following pseudocode illustrates a straightforward way to compute *g*:

```
procedure compute_g(i):
    if f(i,i) == 0 then
        return 0
     else
        loop forever
```

Because *g* is partial computable, there must be a program *e* that computes *g*, by the assumption that the model of computation is Turing-complete. This program is one of all the programs on which the halting function *h* is defined. The next step of the proof shows that *h*(*e*,*e*) will not have the same value as *f*(*e*,*e*).

It follows from the definition of *g* that exactly one of the following two cases must hold:

- $g(e) = f(e,e) = 0$. In this case $h(e,e) = 1$, because program *e* halts on input *e*.
- $g(e)$ is undefined and $f(e,e) \neq 0$. In this case $h(e,e) = 0$, because program *e* does not halt on input *e*.

In either case, *f* cannot be the same function as *h*. Because *f* was an *arbitrary* total computable function with two arguments, all such functions must differ from *h*.

This proof is analogous to Cantor's diagonal argument. One may visualize a two-dimensional array with one column and one row for each natural number, as indicated in the table above. The value of *f*(*i*,*j*) is placed at column *i*, row *j*. Because *f* is assumed to be a total computable function, any element of the array can be calculated using *f*. The construction of the function *g* can be visualized using the main diagonal of this array. If the array has a 0 at position (*i*,*i*), then *g*(*i*) is 0. Otherwise, *g*(*i*) is undefined. The contradiction comes from the fact that there is some column *e* of the array corresponding to *g* itself. If *f* were the halting function *h*, there would be a 1 at position (*e*,*e*) if and only if *g*(*e*) is defined. But *g* is constructed so that *g*(*e*) is defined if and only if there is a 0 in position (*e*,*e*).

Common pitfalls

The difficulty in the halting problem lies in the requirement that the decision procedure must work for all programs and inputs. A particular program either halts on a given input or does not halt. Consider one algorithm that always answers "halts" and another that always answers "doesn't halt." For any specific program and input, one of these two algorithms answers correctly, even though nobody may know which one.

There are programs (interpreters) that simulate the execution of whatever source code they are given. Such programs can demonstrate that a program does halt if this is the case: the interpreter itself will eventually halt its simulation, which shows that the original program halted. However, an interpreter will not halt if its input program does not halt, so this approach cannot solve the halting problem as stated. It does not successfully answer "doesn't halt" for programs that do not halt.

The halting problem is theoretically decidable for linear bounded automata (LBAs) or deterministic machines with finite memory. A machine with finite memory has a finite number of states, and thus any deterministic program on it must eventually either halt or repeat a previous state:

"*...any finite-state machine, if left completely to itself, will fall eventually into a perfectly periodic repetitive pattern*. The duration of this repeating pattern cannot exceed the number of internal states of the machine..."(italics in original, Minsky 1967, p. 24)

Minsky warns us, however, that machines such as computers with e.g. a million small parts, each with two states, will have on the order of $2^{1,000,000}$ possible states:

"This is a 1 followed by about three hundred thousand zeroes ... Even if such a machine were to operate at the frequencies of cosmic rays, the aeons of galactic evolution would be as nothing compared to the time of a journey through such a cycle" (Minsky p. 25)

Minsky exhorts the reader to be suspicious—although a machine may be finite, and finite automata "have a number of theoretical limitations":

"...the magnitudes involved should lead one to suspect that theorems and arguments based chiefly on the mere finiteness [of] the state diagram may not carry a great deal of significance" (ibid).

It can also be decided automatically whether a nondeterministic machine with finite memory halts on none of, some of, or all of the possible sequences of nondeterministic decisions, by enumerating states after each possible decision.

Formalization of the halting problem

In his original proof Turing formalized the concept of *algorithm* by introducing Turing machines. However, the result is in no way specific to them; it applies equally to any other model of computation that is equivalent in its computational power to Turing machines, such as Markov algorithms, Lambda calculus, Post systems, register machines, or tag systems.

What is important is that the formalization allows a straightforward mapping of algorithms to some data type that the algorithm can operate upon. For example, if the formalism lets algorithms define functions over strings (such as Turing machines) then there should be a mapping of these algorithms to strings, and if the formalism lets

algorithms define functions over natural numbers (such as computable functions) then there should be a mapping of algorithms to natural numbers. The mapping to strings is usually the most straightforward, but strings over an alphabet with n characters can also be mapped to numbers by interpreting them as numbers in an n-ary numeral system.

Relationship with Gödel's incompleteness theorem

The concepts raised by Gödel's incompleteness theorems are very similar to those raised by the halting problem, and the proofs are quite similar. In fact, a weaker form of the First Incompleteness Theorem is an easy consequence of the undecidability of the halting problem. This weaker form differs from the standard statement of the incompleteness theorem by asserting that a complete, consistent and sound axiomatization of all statements about natural numbers is unachievable. The "sound" part is the weakening: it means that we require the axiomatic system in question to prove only *true* statements about natural numbers (it's very important to observe that the statement of the standard form of Gödel's First Incompleteness Theorem is completely unconcerned with the question of truth, but only concerns the issue of whether it can be proven).

The weaker form of the theorem can be proven from the undecidability of the halting problem as follows. Assume that we have a consistent and complete axiomatization of all true first-order logic statements about natural numbers. Then we can build an algorithm that enumerates all these statements. This means that there is an algorithm $N(n)$ that, given a natural number n, computes a true first-order logic statement about natural numbers such that, for all the true statements, there is at least one n such that $N(n)$ yields that statement. Now suppose we want to decide if the algorithm with representation a halts on input i. By using Kleene's T predicate, we can express the statement "a halts on input i" as a statement $H(a, i)$ in the language of arithmetic. Since the axiomatization is complete it follows that either there is an n such that $N(n) = H(a, i)$ or there is an n' such that $N(n') = \neg H(a, i)$. So if we iterate over all n until we either find $H(a, i)$ or its negation, we will always halt. This means that this gives us an algorithm to decide the halting problem. Since we know that there cannot be such an algorithm, it follows that the assumption that there is a consistent and complete axiomatization of all true first-order logic statements about natural numbers must be false.

Recognizing partial solutions

There are many programs that either return a correct answer to the halting problem or do not return an answer at all. If it were possible to decide whether any given program gives only correct answers, one might hope to collect a large number of such programs and run them in parallel, in the hope of being able to determine whether any programs halt. Curiously, recognizing such partial halting solvers (PHS) is just as hard as the halting problem itself.

Suppose someone claims that program PHSR is a partial halting solver recognizer. Construct a program H:

```
input a program P
X := "input Q. if Q = P output 'halts' else loop forever"
run PHSR with X as input
```

If PHSR recognizes the constructed program X as a partial halting solver, that means that P, the only input for which X produces a result, halts. If PHSR fails to recognize X, then it must be because P does not halt. Therefore H can decide whether an arbitrary program P halts; it solves the halting problem. Since this is impossible, the program PHSR could not have been a partial halting solver recognizer as claimed. Therefore, no program can be a complete partial halting solver recognizer.

Another example, H_T, of a Turing machine which gives correct answers only for *some* instances of the halting problem can be described by the requirements that, if H_T is started scanning a field which carries the first of a finite string of *a* consecutive "1"s, followed by one field with symbol "0" (i. e. a blank field), and followed in turn by a finite string of *i* consecutive "1"s, on an otherwise blank tape, then

- H_T halts for any such starting state, i. e. for any input of finite positive integers *a* and *i*;
- H_T halts on a completely *blank* tape if and only if the Turing machine represented by *a* does not halt when given the starting state and input represented by *i*; and
- H_T halts on a *nonblank* tape, scanning an appropriate field (which however does not necessarily carry the symbol "1") if and only if the Turing machine represented by *a* does halt when given the starting state and input represented by *i*. In this case, the final state in which H_T halted (contents of the tape, and field being scanned) shall be equal to some particular intermediate state which the Turing machine represented by *a* attains when given the starting state and input represented by *i*; or, if all those intermediate states (including the starting state represented by *i*) leave the tape blank, then the final state in which H_T halted shall be scanning a "1" on an otherwise blank tape.

While its existence has not been refuted (essentially: because there's no Turing machine which would halt *only* if started on a blank tape), such a Turing machine H_T would solve the halting problem only *partially* either (because it doesn't necessarily scan the symbol "1" in the final state, if the Turing machine represented by *a* does halt when given the starting state and input represented by *i*, as explicit statements of the halting problem for Turing machines may require).

History of the halting problem

- 1900 – David Hilbert poses his "23 questions" cf Hilbert problems at the Second International Congress of Mathematicians in Paris, "Of these, the second was that of proving the consistency of the 'Peano axioms' on which, as he had shown, the rigour of mathematics depended" (Hodges p. 83, Davis' commentary in Davis, 1965, p. 108).
- 1920–1921 – Emil Post explores the halting problem for tag systems, regarding it as a candidate for unsolvability. (Source: *Absolutely unsolvable problems and*

relatively undecidable propositions – account of an anticipation, in Davis, 1965, pp. 340–433.) Its unsolvability was not established until much later, by Marvin Minsky [1961].

- 1928—Hilbert recasts his 'Second Problem' at the Bologna International Congress (cf Reid pp. 188–189). Hodges claims he posed three questions: i.e. #1: Was mathematics *complete*? #2: Was mathematics *consistent*? #3: Was mathematics *decidable*? (Hodges p. 91). The third question is known as the *Entscheidungsproblem* (Decision Problem) (Hodges p. 91, Penrose p. 34)
- 1930 – Kurt Gödel announces a proof as an answer to the first two of Hilbert's 1928 questions [cf Reid p. 198]. "At first he [Hilbert] was only angry and frustrated, but then he began to try to deal constructively with the problem... Gödel himself felt – and expressed the thought in his paper – that his work did not contradict Hilbert's formalistic point of view" (Reid p. 199).
- 1931—Gödel publishes "On Formally Undecidable Propositions of Principia Mathematica and Related Systems I", (reprinted in Davis, 1965, p. 5ff)
- 19 April 1935 – Alonzo Church publishes "An Unsolvable Problem of Elementary Number Theory", wherein he identifies what it means for a function to be *effectively calculable*. Such a function will have an algorithm, and "...the fact that the algorithm has *terminated* becomes effectively known ..." (italics added, Davis, 1965, p. 100).
- 1936—Church publishes the first proof that the *Entscheidungsproblem* is unsolvable [*A Note on the Entscheidungsproblem*, reprinted in Davis, 1965, p. 110].
- 7 October 1936 – Emil Post's paper "Finite Combinatory Processes. Formulation I" is received. Post adds to his "process" an instruction "(C) Stop". He called such a process "type 1 ... if the process it determines terminates for each specific problem." (Davis, 1965, p. 289ff)
- 1937 – Alan Turing's paper *On Computable Numbers With an Application to the Entscheidungsproblem* reaches print in January 1937 (reprinted in Davis, 1965, p. 115). Turing's proof departs from calculation by recursive functions and introduces the notion of computation by machine. Stephen Kleene (1952) refers to this as one of the "first examples of decision problems proved unsolvable".
- 1939 – J. Barkley Rosser observes the essential equivalence of "effective method" defined by Gödel, Church, and Turing (Rosser in Davis, 1965, p. 273, "Informal Exposition of Proofs of Gödel's Theorem and Church's Theorem").
- 1943—In a paper, Stephen Kleene states that "In setting up a complete algorithmic theory, what we do is describe a procedure ... which procedure necessarily terminates and in such manner that from the outcome we can read a definite answer, 'Yes' or 'No,' to the question, 'Is the predicate value true?'."
- 1952—Kleene (1952) Chapter XIII ("Computable Functions") includes a discussion of the unsolvability of the halting problem for Turing machines and reformulates it in terms of machines that "eventually stop", i.e. halt: "... there is no algorithm for deciding whether any given machine, when started from any given situation, *eventually* ***stops***." (Kleene (1952) p.382)
- 1952 – "Davis [Martin Davis] thinks it likely that he first used the term 'halting problem' in a series of lectures that he gave at the Control Systems Laboratory at

the University of Illinois in 1952 (letter from Davis to Copeland, 12 Dec. 2001.)"
(Footnote 61 in Copeland (2004) pp.40ff)