Computer Science 1000: Part #8

Models of Computation

MODELS OF COMPUTATION: AN OVERVIEW

TURING MACHINES

PROVING UNSOLVABILITY

Models of Computation: An Overview

Alan Turing
(1912-1954)

Turing A. M.. "On computable numbers, with an application to the Entscheidungsproblem*."* Proceedings of the London Mathematical Society*, 2 s. vol. 42 (1936–1937), pp. 230–265.*

Why is the above crucial to modern computation?

Models of Computation: An Overview (Cont'd)

IS EVERY PROBLEM SOLVABLE?

- To investigate this question, need basic model of computation (cf. computational time complexity as basic model for investigating algorithm runtime).
- Typical properties of a model compared to the real thing being modeled:
    1. Captures important properties of real thing.
    2. Probably differs in scale from real thing.
    3. Omits some details of real thing.
    4. Lacks full functionality of real thing.

Models of Computation: An Overview (Cont'd)

- Every model based on assumptions, and information derived with a model is only as good as those assumptions.
- Necessary properties of a model of a computing agent:
  1. Accepts input.
  2. Can store and retrieve information wrt memory.
  3. Acts on stored algorithm instructions based on the current state of and the data item currently being processed by the agent.
  4. Produces output.
- Many models of computation proposed in early 20th century in response to Hilbert's Program.

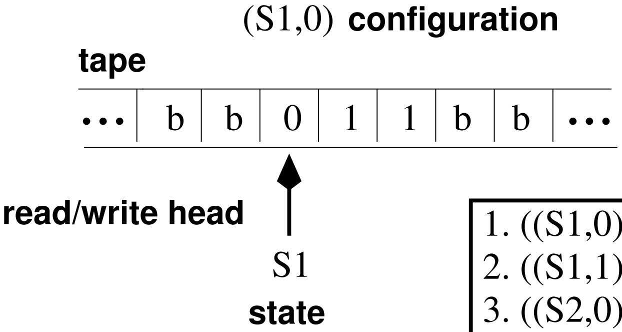
Models of Computation: An Overview (Cont'd)

David Hilbert
(1862–1943)

Kurt Gödel
(1906–1978)

- Hilbert (1920): Formalize mathematical proof to eliminate ambiguities and allow automation of proof.
- Gödel (1931): Every reasonable arithmetic system has true statements that are unprovable in that system.

Turing Machines: Overview



1. ((S1,0),(1,S2,R))
2. ((S1,1),(1,S2,R))
3. ((S2,0),(1,S2,R))
4. ((S2,1),(0,S2,R))
5. ((S2,b),(b,S3,L))

instructions

Turing Machines: Overview (Cont'd)

- A **Turing Machine** consists of (1) a two-way infinite tape, (2) a tape-square alphabet, (3) a read/write head that can be positioned on any tape square, (4) a set of states $\{$ **S1, S2, . . . , Sn** $\}$, and (5) a set of instructions.
- The tape functions as input, memory, and output at TM start, execution, and termination.
- The alphabet can be any number of symbols plus a special blank (**b**) symbol; focus here on the alphabet $\{$ **b, 0, 1** $\}$.
- At any given time, a TM is in a particular state **Si** and the read/write head is reading symbol x; this pair (**Si,** x) is called the TM's **configuration**.

Turing Machines: Overview (Cont'd)

- A TM instruction specifies what the TM does next when it is in a specified configuration.
- There are several ways of writing TM instructions, e.g.

```
if state is S1 and symbol is 0 then
    write 1 in current tape square
    set state to S2
    move r/w head one square right
```

$$\Updownarrow$$

((S1,0),(1,S2,R))

$$\Updownarrow$$

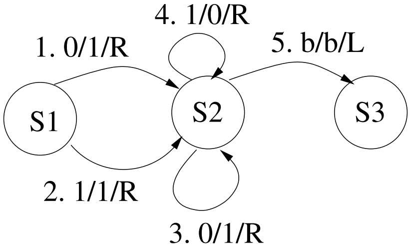
(1,0,1,2,R)

Turing Machines: Overview (Cont'd)

There are several ways of writing TM instruction-sets, e.g.,

1. ((S1,0),(1,S2,R))
2. ((S1,1),(1,S2,R))
3. ((S2,0),(1,S2,R))
4. ((S2,1),(0,S2,R))
5. ((S2,b),(b,S3,L))

instructions

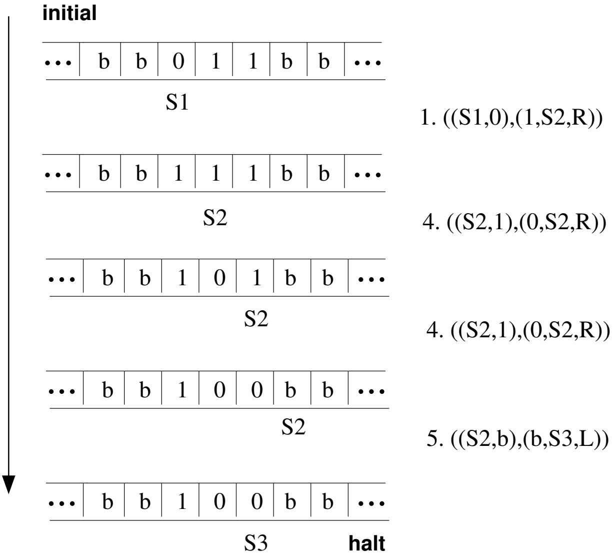


state diagram

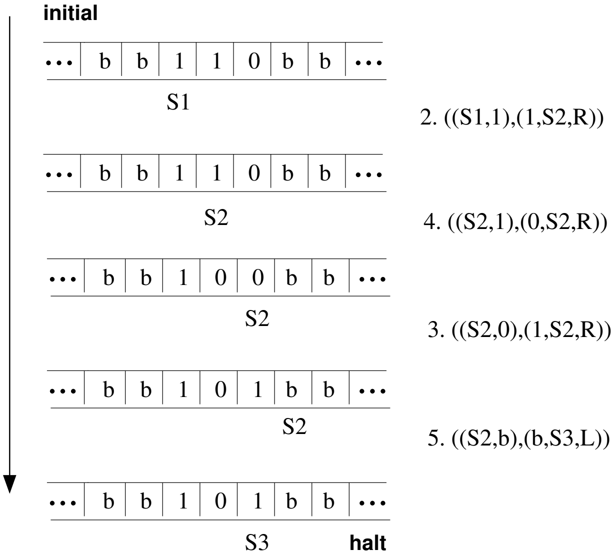
Turing Machines: Overview (Cont'd)

- Starting from an initial configuration, a TM executes instructions until it halts.
- TM operation conventions:
  1. The input is placed on the TM tape.
  2. The initial position of the TM read/write head is the leftmost non-blank tape square, i.e., the leftmost square of the input.
  3. The initial TM state is S1.
  4. At each point, there is at most one instruction that matches the current TM configuration, i.e., the TM is **deterministic**.
  5. The TM halts when there is no instruction that matches the current TM configuration.
  6. On halting, the output is the contents of the TM tape.
- Note that TM instructions execute in TM-configuration order, not instruction-order, cf. Python programs.

Turing Machines: Overview (Cont'd)



Turing Machines: Overview (Cont'd)



Turing Machines: Overview (Cont'd)

- A TM is an adequate model of computing agent:
  1. **Accepts input**: TM encodes input on and reads symbols from tape.
  2. **Can store and retrieve information wrt memory**: During execution, TM writes symbols on and later can read these symbols from tape.
  3. **Acts on stored algorithm instructions based on the current state of and the data item currently being processed by the agent**: TM configuration dictates executed instruction.
  4. **Produces output**: If TM halts, tape is output.
- TM are more capable from real computers because TM tape (memory) is unlimited; hence, a task that is TM-solvable might not be real-computer-solvable.

Turing Machines: Overview (Cont'd)

- A TM instruction-set is an algorithm:
  1. **Is well-ordered**: As our TM are deterministic, at most one instruction executable for any TM configuration.
  2. **Consists of unambiguous and effectively computable operations**: TM instructions are unambiguous to TMs.
  3. **Halts in finite time**: Relative to TM-appropriate inputs, a TM always halts (appropriate inputs also key to algorithms halting).
  4. **Produces output**: Output is tape contents after execution and halting on TM-appropriate input.
- When we write a TM for a task, we write a set of TM instructions to do that task.

Turing Machines: Example Tasks

1. Invert the bits in a given binary string, e.g., $1101 \rightarrow 0010$.
2. Add a parity bit to the end of a binary string such that the total number of 1-bits in the resulting string is odd, e.g., $101 \rightarrow 1011$, $001 \rightarrow 0010$.
3. Increment a unary number by 1, e.g., $111 \rightarrow 1111$, where

   0 is represented as 1 in unary
   1 is represented as 11 in unary
   2 is represented as 111 in unary
   3 is represented as 1111 in unary

   and so on.
4. Add two unary numbers of value > 0 separated by a blank symbol, e.g., $11b11 \rightarrow 111$.

Turing Machines: A Bit Inverter

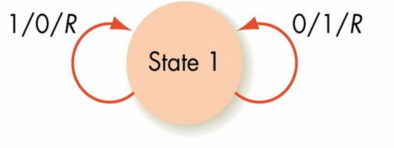


Figure 11.4
State Diagram for the Bit Inverter Machine

Turing Machines: A Bit Inverter (Cont'd)

initial

...	b	b	0	1	1	1	b	...

S1

((S1,0),(1,S1,R))

...	b	b	1	1	1	1	b	...

S1

((S1,1),(0,S1,R))

...	b	b	1	0	1	1	b	...

S1

((S1,1),(0,S1,R))

...	b	b	1	0	0	1	b	...

S1

((S1,1),(0,S1,R))

...	b	b	1	0	0	0	b	...

S1 **halt**

Turing Machines: A Bit Inverter (Cont'd)

initial

...	b	b	1	1	0	1	b	...
			S1					

((S1,1),(0,S1,R))

...	b	b	0	1	0	1	b	...
				S1				

((S1,1),(0,S1,R))

...	b	b	0	0	0	1	b	...
					S1			

((S1,0),(1,S1,R))

...	b	b	0	0	1	1	b	...
						S1		

((S1,1),(0,S1,R))

...	b	b	0	0	1	0	b	...
							S1	**halt**

Turing Machines: A Parity Bit Machine

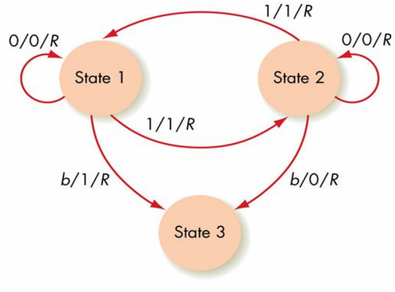


Figure 11.5
State Diagram for the Parity Bit Machine

Turing Machines:
A Parity Bit Machine (Cont'd)

initial

•••	b	b	1	0	1	b	b	•••

S1

((S1,1),(1,S2,R))

•••	b	b	1	0	1	b	b	•••

S2

((S2,0),(0,S2,R))

•••	b	b	1	0	1	b	b	•••

S2

((S2,1),(1,S1,R))

•••	b	b	1	0	1	b	b	•••

S1

((S1,b),(1,S3,R))

•••	b	b	1	0	1	1	b	•••

S3 **halt**

Turing Machines:
A Parity Bit Machine (Cont'd)

initial

…	b	b	0	0	1	b	b	…

S1

((S1,0),(0,S1,R))

…	b	b	0	0	1	b	b	…

S1

((S1,0),(0,S1,R))

…	b	b	0	0	1	b	b	…

S1

((S1,1),(1,S2,R))

…	b	b	0	0	1	b	b	…

S2

((S2,b),(0,S3,R))

…	b	b	0	0	1	0	b	…

S3 **halt**

Turing Machines: A Unary Incrementer (Take I)

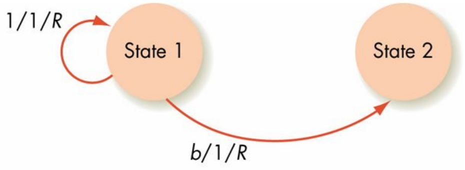
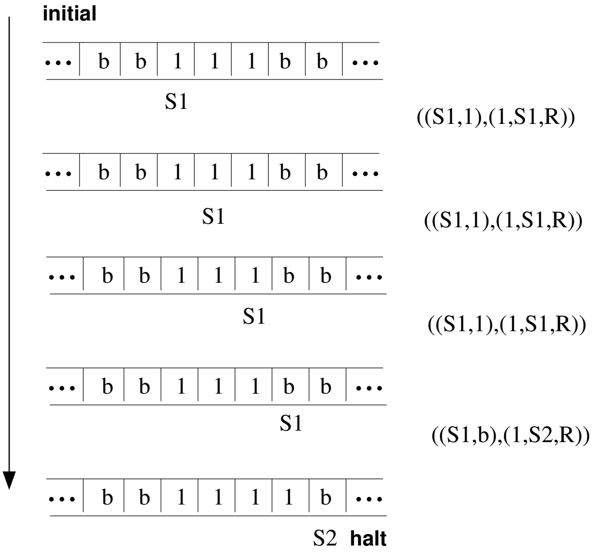
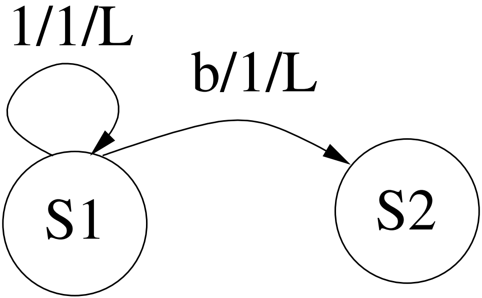


Figure 11.6
State Diagram for Incrementer

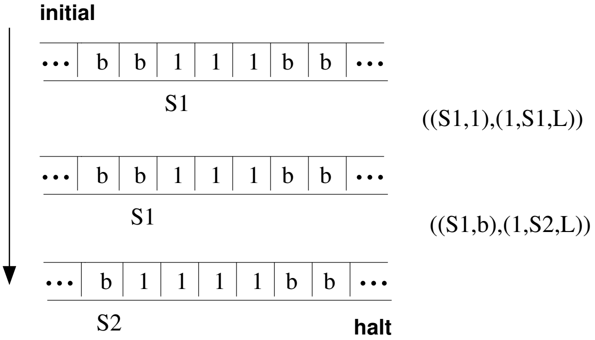
Turing Machines:
A Unary Incrementer (Take I) (Cont'd)



Turing Machines: A Unary Incrementer (Take II)



Turing Machines:
A Unary Incrementer (Take II) (Cont'd)



Turing Machines: A Unary Adder

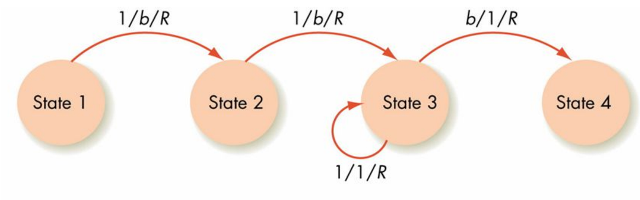
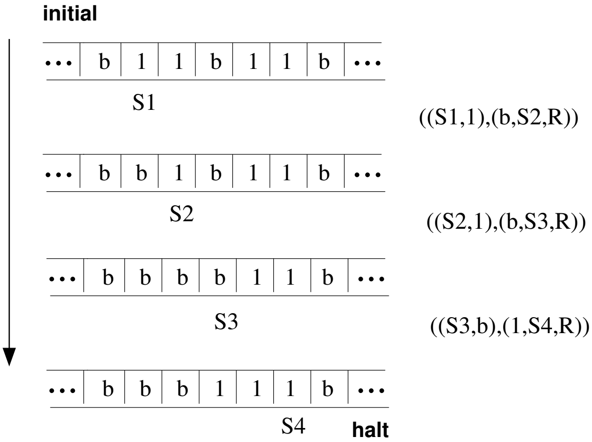
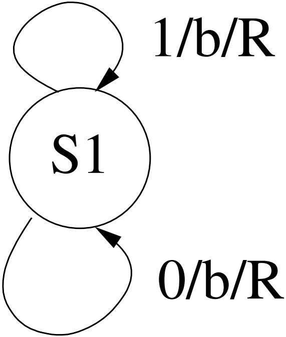


Figure 11.8
State Diagram for the Addition Machine

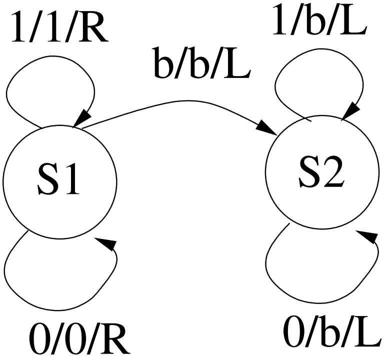
Turing Machines:
A Unary Adder (Cont'd)



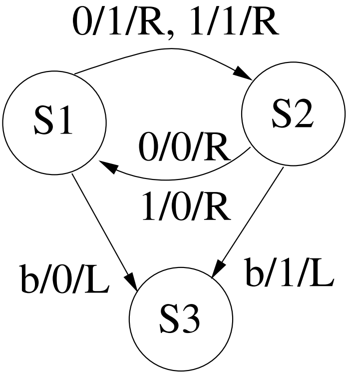
Turing Machines: Mystery Machine #1



Turing Machines:
Mystery Machine #2



Turing Machines: Mystery Machine #3



Proving Unsolvability: The Church-Turing Thesis

- We know that every TM is an algorithm — does every algorithm have a corresponding TM?

The Church-Turing Thesis: For every symbol-manipulation algorithm there is a TM.

- Not provable, but two lines of evidence:
  1. Every proposed s-m algorithm has a TM.
  2. TM can simulate and is thus equivalent to other proposed models of computation.
- C-T Thesis $\Rightarrow$ TM defines limits of solvability!

Proving Unsolvability: The Church-Turing Thesis (Cont'd)

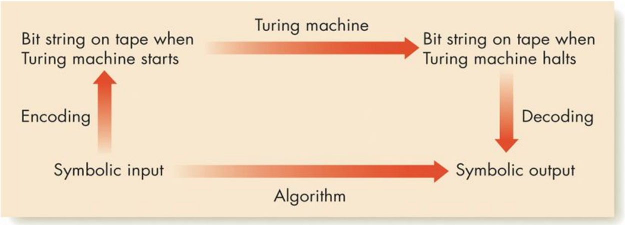


Figure 11.9
Emulating an Algorithm by a Turing Machine

Proving Unsolvability: The Halting Problem

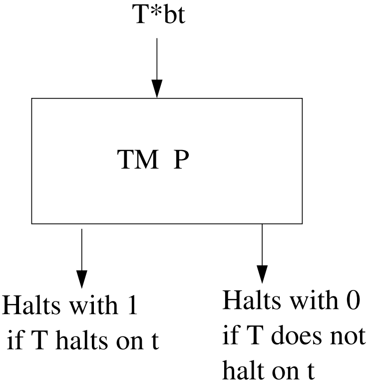
- Easy to prove if a given TM halts on a given configuration; what about if a given TM halts on a given input (**Halting Problem**), e.g., does the TM with instruction-set

  ((S1,b),(b,S1,R))
  ((S1,0),(0,S1,R))
  ((S1,1),(1,S1,R))

  halt on input tape ...b000b ...?
- Prove that HP is unsolvable by contradiction — that is, start by assuming that HP is solvable and then derive something that is impossible, which is a contradiction and would hence imply that HP is not solvable.

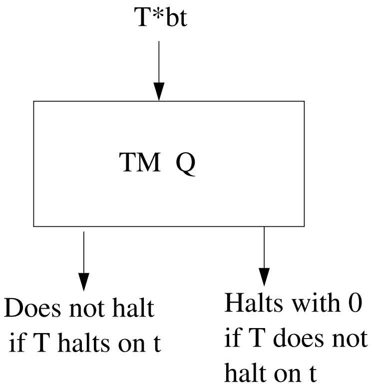
Proving Unsolvability: The Halting Problem (Cont'd)

Suppose you have a TM P that solves HP:



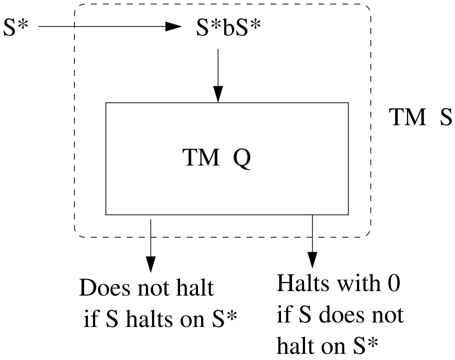
Proving Unsolvability: The Halting Problem (Cont'd)

Modify TM P to create TM Q:



Proving Unsolvability: The Halting Problem (Cont'd)

Modify TM Q to create TM S:

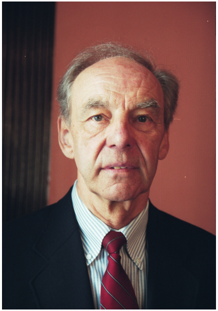


...which is impossible — hence, HP is not solvable!

Proving Unsolvability: The Halting Problem (Cont'd)

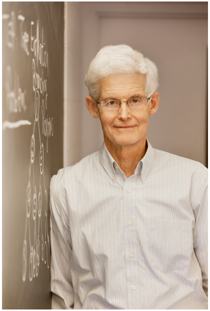
- The unsolvability of HP has practical consequences:
  - No program can decide if a given program halts on all possible inputs.
  - No program can decide if two given programs produce the same output for all possible inputs.
  - No program can decide if a given program run on a given input will produce a given output.
- **The Fine Print**: All of this unsolvability holds in general, i.e., relative to **all** possible programs and inputs — there may yet be programs that work relative to specific classes of given programs, e.g., programs that halt in $\leq 10^9$ steps.

Proving Unsolvability:
The Next Generation



Juris Hartmanis
(1928–)

Jack Edmonds
(1934–)



Stephen Cook
(1939–)

Developed theory of *polynomial-time* unsolvability . . .

. . . but that is a story for another day . . .

. . . And If You Liked This . . .

- MUN Computer Science courses on this area:
  - COMP 4741: Formal Languages and Computability
  - COMP 4742: Computational Complexity
- MUN Computer Science professors teaching courses / doing research in in this area:
  - Miklos Bartha
  - Antonina Kolokolova
  - Manrique Mata-Montero
  - Todd Wareham