CITS2211 Discrete Structures
Turing Machines

Unit coordinator: Débora Corrêa
Lecturer: Matt Ryan

Highlights

We have seen that FSMs and PDAs are surprisingly powerful

But there are some languages they can not recognise

We will study a new type of automaton: **Turing machines**. And the Languages they describe: Turing Decidable and Turing Recognisable

TMs capture our intuitive notion of an **algorithm**

Motivation 1

It's actually quite useful to know if you've been asked to solve an impossible (or just infeasible) problem.

For instance, if someone ever asks you to write a timetabling program, which finds a timetable for students, rooms, and lecturers, such that there are no clashes – it's handy to know this is not *practically* possible.

Any real timetabling program will solve a more limited problem, and may end up producing some clashes.

Motivation 2

And once we've looked at the proof of the halting problem, we'll see that the following questions are ones which it is impossible to write a program to answer precisely (even in theory):

- Does this program enter an infinite loop?
- Does this program contain malware?
- Does this program try to add an integer to a string?

We will see that for each of these questions, we can write a detector-program that *does* detect any program which tries to do these things – but our detector-program will also catch many perfectly harmless programs, as well. It is impossible to write a detector which catches *only and all* those programs which meet the condition.

Reading

Introduction to the Theory of Computation by Michael Sipser

Chapter 3: The Church-Turing Thesis
Section 3.1 Turing Machines
Section 3.3 Definition of Algorithm

Chapter 4: Decidability
Section 4.1 Decidability
Section 4.2 Undecidability

Lecture Outline

1. Turing Machines Definition
2. TMs as recognisers
3. TMs as computers
4. TM powers

5. Computable functions
6. The Church Turing Thesis
7. The Halting Problem

Turing Machine Definitions

What are the limits of FSMs?

The main limitation of finite state machines is the lack of auxiliary memory; this prevents us from recognising relatively simple languages such as

$$L = \{0^n 1^n \mid n \in \mathbb{N}\}$$

The *Turing machine* was invented by the British mathematician Alan Turing in a paper published in 1937 as an attempt to precisely specify the nature of "computation". You can find a copy of this classic paper on the cits2211 Resources web page.

The Turing machine appears to be precisely the correct abstract model for a digital computer. The Church-Turing thesis (later in these notes) expresses this idea formally.

Components of a Turing machine

A Turing machine has two main components:

1. A finite state machine that acts as a “controller"
2. A two-way infinite tape, divided into cells, from which the machine can **read** input symbols, **move** left or right, and onto which the machine can **write** output symbols

Components of a Turing machine (cont)

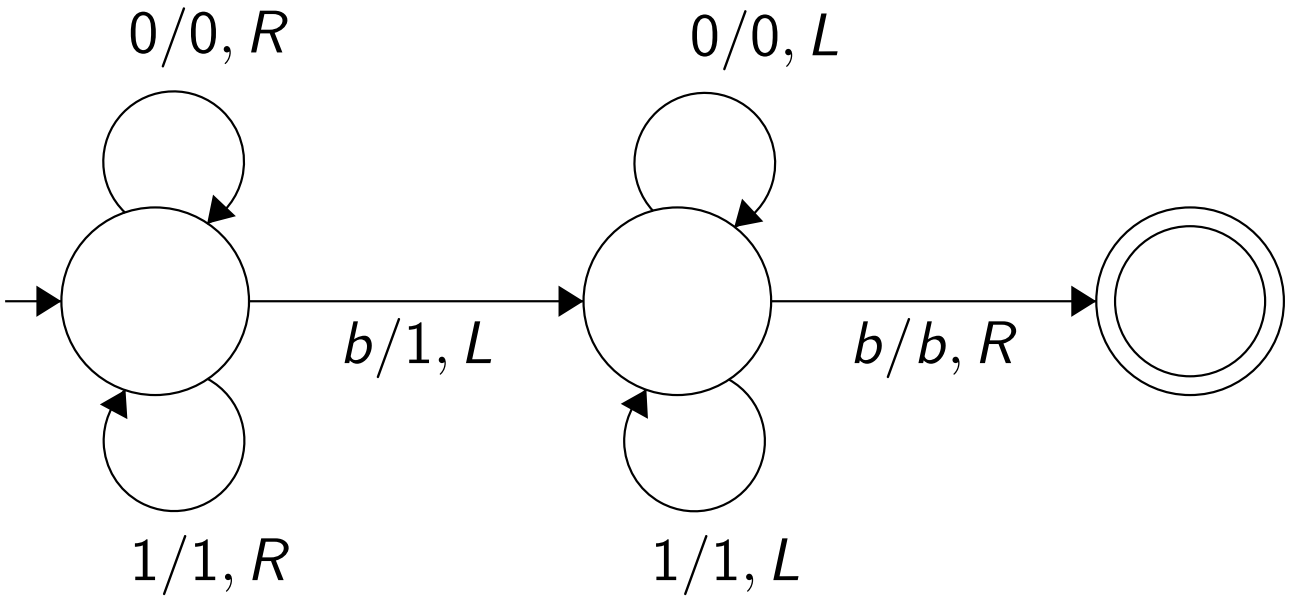
The **tape** can be used to write down intermediate results, and therefore it acts as **input, output** and **memory** device.

Demo

Chapter one Turing Machine Example by Mark Meysenburg

http://www.youtube.com/watch?v=IkYhfk4X47c

Turing Machine Diagrams



Input (read tape) and output (write tape) written as i/o

Move direction written as R **or** L**.**

The machine **halts** when it reaches a state and input symbol for which there is no outgoing edge.

Turing machine computation

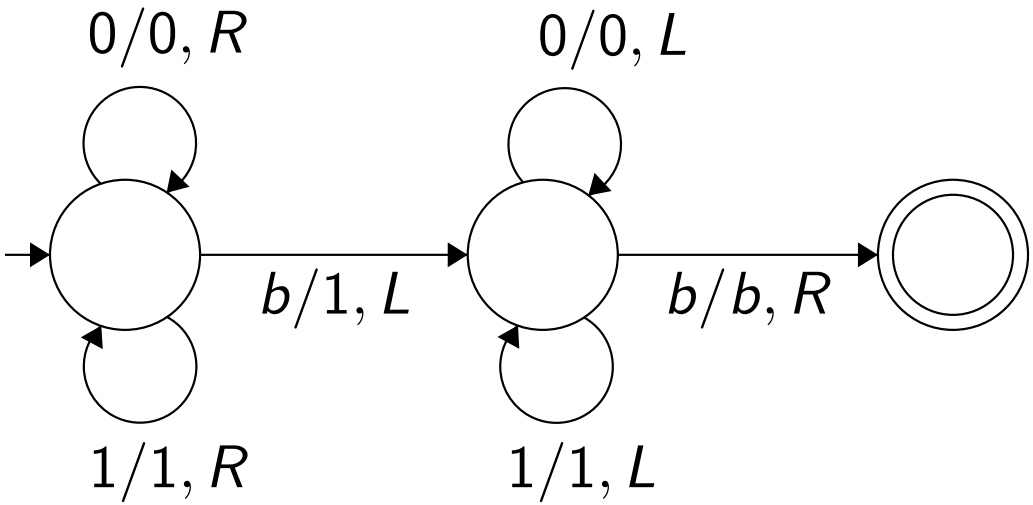
The Turing machine is assumed to start with its reader/writer positioned over the leftmost non-blank tape cell, and with the finite state controller in state s_0.

Then, at each computational step, the Turing machine either **halts** or performs the following actions:

1. The symbol on the current tape cell is read
2. A symbol (possibly the same one) is written on the tape cell (erasing the old symbol)
3. The finite state machine moves to a new state (possibly the same one)
4. The read/write head is moved one cell left or right

TM Example

How does this Turing Machine process the input tape 1001 ? Try some other examples: can you see a pattern ?



Conventions: TMs starts at the leftmost non-blank symbol; b represents the blank symbol; all blanks around the given tape symbols; TM ***halts*** *if there is no transition for the current state and input.*

Formal Definition of a Turing Machine

Definition: A **Turing Machine (TM)** is defined to be a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ where

- Q is a finite set of *states*
- Σ is a finite alphabet of *input symbols* (not including the *blank symbol* b)
- Γ is the finite *tape alphabet* including $b \in \Gamma$ and $\Sigma \subset \Gamma$
- $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ is the *transition function*
- $q_0 \in Q$ is the start state
- $q_{accept} \in Q$ is the accept state
- $q_{reject} \in Q$ is the reject state

Convention: Diagrams often do not show the reject state and its transitions (as for FSMs and PDAs)

TMs as Recognisers

Turing machines recognise languages

Definition: As soon as the machine enters a accept state q_{accept} it halts. The accepting state can be shown by a state with no outgoing transitions.

Definition: A Turing machine M **accepts** a string w if it reaches an accept state when it is run with w as the initial string on the tape, starting from (as usual) the read/write head positioned on the first symbol of w.

Definition: If there is no matching transition when the machine is not in an accepting state then the machine halts and rejects a word. Looping forever also means rejection.

Turing machines recognise languages

Definition: The **language recognised** by a Turing machine is

$$L(M) = \{w \mid w \text{ is accepted by } M\}$$

Our first indication that Turing machines are more powerful than FSMs is that Turing machines can recognise some non-regular languages.

A Turing machine for $\{0^n1^n\}$

Recall that the language $L = \{0^n1^n : n \in Z\}$ is not regular.

A TM strategy for recognising this language is:

1. check the first symbol is 0 and erase it
2. move to the right-hand symbol of the string
3. check the final symbol is 1 and erase it
4. move back to the left most symbol and repeat from step 1
5. if there are no 0s or 1s left on the tape then accept
6. otherwise reject the string

This type of high-level description is called an ***implementation level description***

A Turing machine for $\{0^n1^n\}$

When starting at the left-hand symbol of the string, the machine will successfully terminate with an empty string if and only if the string contains a number of 0s, followed by an equal number of 1s.

We will try and get away with five states:

0 – at the left end of the string

1 – at the right end of the string

2 – moving right

3 – moving left

4 – final accepting state

At the ends of the string

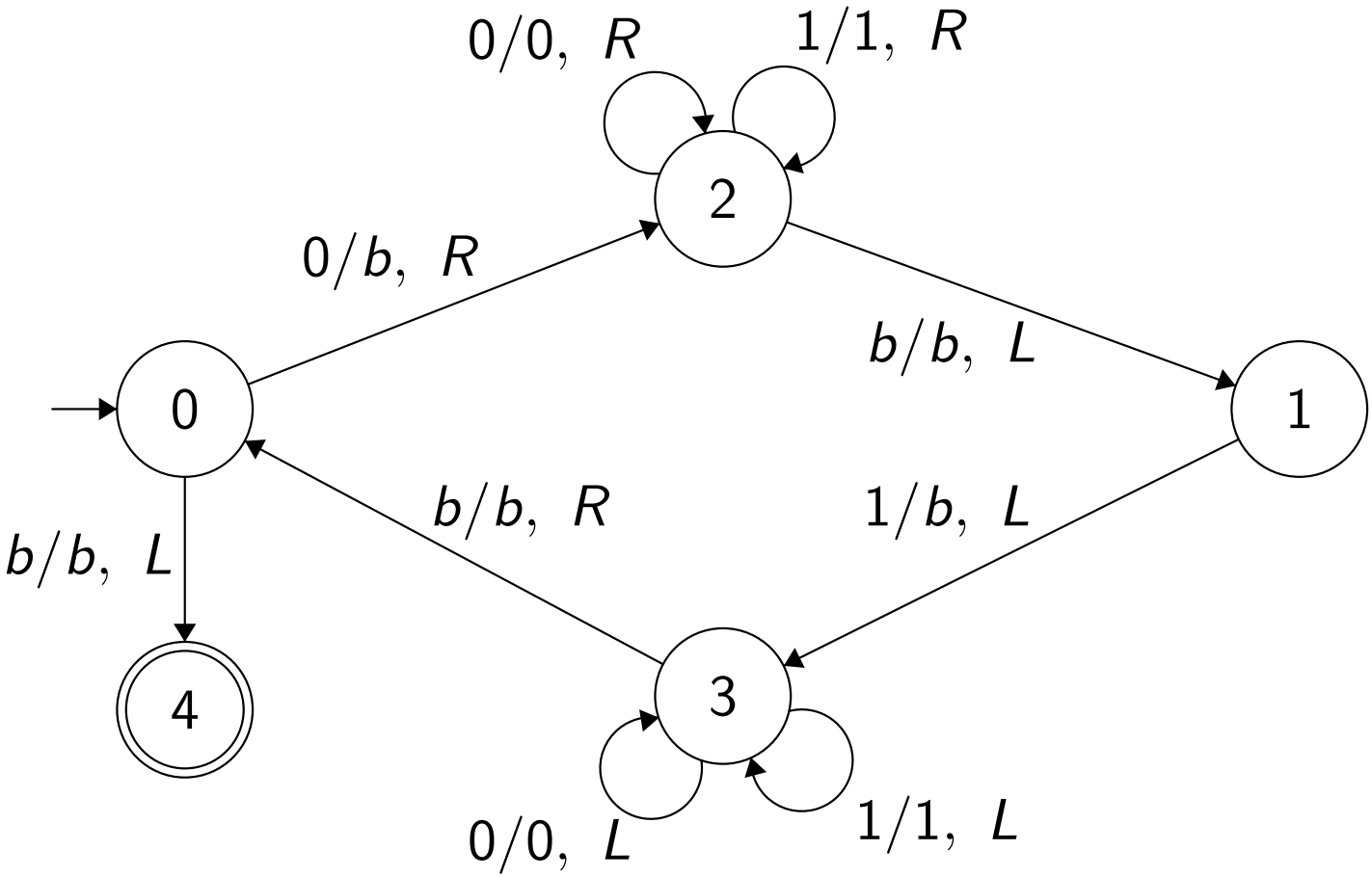
At the left-most end of the string we want to

- *Accept if the symbol is blank*
- *Reject if the symbol is 1 (nothing)*
- *Erase and start moving right if the symbol is 0*

At the right most end of the string we want to:

- *Reject if the symbol is b or 0 (nothing)*
- *Erase and start moving left if the symbol is 1*

Our Turing machine for $\{0^n1^n\}$



TMs as Computers

Computing Functions

As well as recognising languages, we can view a Turing machine as **computing a function**.

We need to decide on a way to represent numbers in the Turing machine, but the particular mechanism is not important.

A simple mechanism to represent numbers is to use **unary** notation. That is, the numbers are just represented by strings of 1s of differing lengths, *starting with one 1 to represent 0*.

For example,

1111 represents 3

111111 represents 5

One variable functions

Using this representation, we can consider 1-variable functions and convince ourselves that Turing machines really *can* do proper computation.

For example, how could we write a Turing machine that would act as a "doubler"

Input: A string of 1s of length n
Output: A string of 1s of length $2n$

Even though this is a relatively straightforward computation, it may take some thought to produce a Turing machine that performs in this fashion. (See tutorial questions)

TM Powers

How powerful are Turing machines?

We have found a non-regular language 0^n1^n that can be recognised by a Turing machine, and hence the set of languages recognisable by Turing machines is strictly larger than the set of regular languages.

Are there any languages that **cannot** be recognised by Turing machines or are they all-powerful?

How powerful are Turing machines?

We have found a non-regular language 0^n1^n that can be recognised by a Turing machine, and hence the set of languages recognisable by Turing machines is strictly larger than the set of regular languages.

Are there any languages that **cannot** be recognised by Turing machines or are they all-powerful?

In fact, with a few preliminaries we can produce an identical argument to that which showed that there are languages that cannot be recognised by finite state machines.

The key observation is that the set of all TMs is **countable**; this means that in principle we can list all the possible TMs.

Countability of Turing machines[1]

We'll begin by showing that the number of possible Turing machines is the smallest infinity, the infinity of natural numbers.

We can define a Turing machine as a set of states and a set of transitions from each state to another state (where the transitions are based on the symbol being read). A crucial aspect of this definition is that both sets are finite.

Because of this, the number of Turing machines is countable. That is, we can "flatten" each machine into one finite-length string that describes it, and we can place these strings into a one-to-one association with natural numbers, just as we can with (for instance) integers or rational numbers.

[1] Notes for this section are taken from MIT open courseware 6.045J / 18.400J Automata, Computability, and Complexity Spring 2011

Number of Machines

It is relatively easy to see that we can specify any Turing machine by listing its transitions which we can do using a finite alphabet, for example $\{b, 0, 1, /, L, R\}$. So if we just concatenate the list of transitions describing a machine into one long string, then each machine is described by a single string.

However, we have already seen that the set of all strings over any finite alphabet is countable — we merely list them in lexicographic order.

In particular, in principle we could simply write down every possible string in order over the alphabet give above, and then just throw out the strings that do not describe Turing machines. Every Turing machine will appear many times over on this list.

Number of Problems

The number of problems, on the other hand, is a greater infinity: namely, the infinity of real numbers This is because we can define a problem as a function that maps every input string $w \in \{0,1\}^*$ to an output (0 or 1).

But since there are infinitely many inputs, to specify such a function requires an infinite number of bits. So as in Cantor's proof, we can show that the infinity of **problems** is greater than the infinity of **TMs**.

Number of Problems vs Number of Machines

The upshot is that there are far more problems than there are Turing machines to solve them. From this perspective, the set of computable problems is just a tiny island in a huge sea of unsolvability.

Admittedly, most of the unsolvable problems are not things that human beings will ever care about, or even be able to define. On the other hand, Turing's proof of the unsolvability of the halting problem shows that at least some problems we care about are unsolvable.

Computable functions

Computable Functions and The Halting Problem

Turing machines compute functions

It is often convenient to view a Turing machine as computing a function $f : \mathbb{N} \to \mathbb{N}$

We will represent a natural number n in unary as a sequence of $n + 1$ ones. (We need $n + 1$ to distinguish 0 from a blank tape)

If we start the machine on a tape containing this string, then it will compute and do one of three things

- Halt with the tape containing the valid representation of another number m
- Halt with the tape containing some other string
- Fail to halt

Turing Computable Functions

This Turing machine then computes the partial function

$$f(n) = \begin{cases} m & \text{if the machine stops in a valid state} \\ \textit{undefined} & \text{otherwise} \end{cases}$$

Definition: A function is said to be *Turing-computable* if there is some Turing machine that computes that function.

How does a Turing machine capture our intuitive notion of what it means for there to be an algorithm to compute some function?

It is more or less clear that if we can find a Turing machine to calculate some function, then we could program a digital computer to calculate the same function (after all, we can program the computer just to simulate the Turing machine).

What about the other direction? Are there things that we can compute (say on a digital computer) that we *cannot* write Turing machine programs for?

The Church-Turing thesis

All attempts to formalize the nature of computation yield precisely the same collection of computable functions.

Turing was not the only mathematician attempting to formally model what is means to be "computable". However no matter what system, and in what language (the language of sets, or logic, or other types of machine) they are expressed, any system that is as powerful as a Turing machine has been shown to be precisely equivalent to a Turing machine.

This leads us to the working hypothesis that Turing machines actually capture the "correct" notion of computability for digital computers.

Universal Turing Machines

Definition: 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 the machine to be simulated as well as its input from its own tape.

With this encoding of a Turing Machine as a 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.

The Halting problem

Kings College Cambridge 1937

Preliminaries

We have now covered the four main ideas needed to understand the halting problem:

Turing Machine Model A TM comprises a finite state machine plus an input tape of symbols which can be read and written and moved over one square at a time.

Halting and Looping A TM halts if it reaches a state and input value for which no rule exists to make a further move. A TM loops, by running forever, if it never reaches a halting state.

Specifying a Turing Machine can be done by encoding the states and rules of the TM as a string of symbols, which can then be written on an input tape.

Universal Turing Machine is a machine that can simulate the behaviour of an arbitrary Turing Machine.

The Halting problem $HALT_{TM}$

Determine whether a Turing Machine halts on a given input (by accepting or rejecting that input)

$HALT_{TM} = \{(M, w) \mid M \text{ is a TM and } M \text{ halts on input } w\}$

$HALT_{TM}$ is undecidable

The Halting problem $HALT_{TM}$ is **undecidable.**

That is, there can be *no algorithm (TM)* that can decide whether M, w halts or not.

Proof sketch

Suppose the Halting problem is decideable.

Then we can find a **Turing machine** X such that when X is presented with the input string (s_T, α) it will produce either 1 (for halt) or 0 (for not halt) depending on whether TM machine T (specified by string s_T) halts on input string α or not.

Now define a **machine** Y by adding extra tuples to X so that Y first runs X, and then loops indefinitely if X produces a 1, or halts if X produces a 0.

Proof sketch (cont)

Finally, define a third **machine** Z that takes a single string β and does the following

- Copies the input to form (β, β)
- Runs Y on that input

The contradiction

What does Z do when presented with the string s_Z?

It writes down (s_Z, s_Z) and then runs Y on that input.

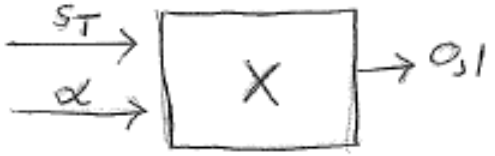
Suppose Z halts on input s_Z. Then X with input (s_Z, s_Z) will halt with output 1. So Y with input (s_Z, s_Z) will loop indefinitely. So Z with input s_Z will loop indefinitely. Contradiction!

But now suppose Z loops forever on input s_Z. Then X with input (s_Z, s_Z) will halt with output 0. So Y with input (s_Z, s_Z) will halt. So Z with input s_Z will halt. Contradiction!

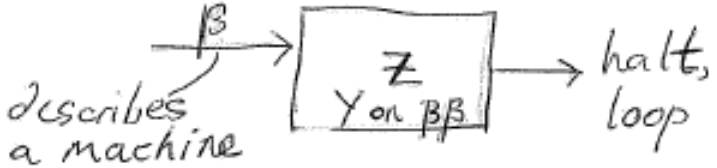
Therefore Z halts on the string s_Z if and only if Z does not halt on the string s_Z

This is a contradiction, and so there cannot be a Turing machine X that can solve the Halting problem.

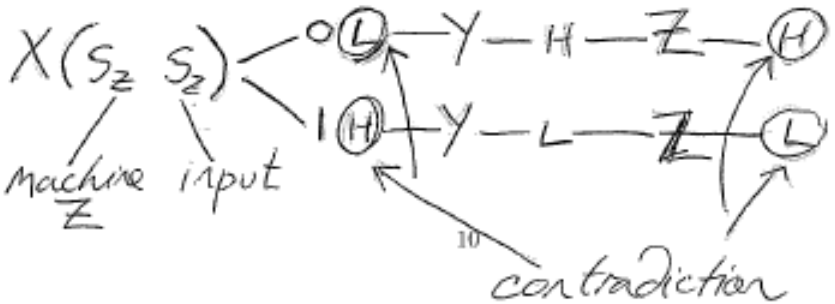
QED


S_T
α
X
0,1


X
$S_T\alpha$
Y
halt, loop


β
describes a machine
Z
Y on $\beta\beta$
halt, loop


S_Z
Z
?


$X(S_Z\ S_Z)$
0
1
Y
H
Z
L
machine Z
input
contradiction

Corollaries

We have demonstrated a particular problem which Turing machines cannot solve, namely: can we determine (algorithmically), given a description of machine M and some input w, whether M halts on w?

The inputs (our ws) already are strings, and we can express turing machines (our Ms) as strings, so we can also express pairs (M, w) as strings.

And this means we can define *languages* which are subsets of the set of all possible (M, w) pairs.

e.g. If we let our alphabet Σ be $\{0, 1\}$, then we could define a language over $(0 + 1)^*$ as follows:

$L = \{w | w \in (0 + 1)^* \wedge w$ represents a pair $(M, w') \wedge M$ has 5 states$\}$

Corollaries – unrecognizable languages

But we can also define languages such as the following:

$L = \{w | w \in (0+1)^* \wedge w \text{ represents a pair } (M, w') \wedge M \text{ halts on } w'\}$

which are languages that *cannot be recognized* by a Turing machine.

In fact, we can construct many similar languages:

$$L = \{w | w \in (0+1)^* \wedge w \text{ represents a turing machine } M \wedge M \text{ halts when given the string “0”}\}$$

A "natural" unsolvable problem: Rado's busy beavers

Fix the alphabet to be $\{b, 1\}$ (just two symbols), and fix a number s of states.

Find the s-state Turing machine that given a completely blank tape, writes down as many marks (1s) as possible, and then halts. The machine is allowed a single "Halt" state.

This problem is known as the "Busy Beaver" problem, and is notoriously difficult even to think about.

In fact, Rado demonstrated that the function itself, called $\Sigma(s)$, is uncomputable.

See the cits2211 Resources web page for details

Summary

1. Turing Machines comprise an FSM controller with a read/write tape used as memory
2. TMs can recognise languages; TMs can compute functions
3. There are more problems than there are TMs to solve them
4. The Church Turing Thesis: All attempts to formalize the nature of computation yield precisely the same collection of computable functions
5. The Halting Problem: Determine whether a Turing Machine halts on a given input (by accepting or rejecting that input)
6. The Halting Problem is undecidable: that means there can be no algorithm (TM) that can decide whether M,w halts or not.

Automata summary

- The simplest automata we have seen are FSMs; they can recognize *regular* languages. But they can't recognize non-regular languages (an example of such a language is 0^n1^n over the alphabet $\{0, 1\}$.
- Push-down automata (PDAs) are an FSM with a *stack* that can be used for storing data; they can recognize the *context-free* languages. But they can't recognise non–context-free languages. (An example of such a language is $a^n b^n c^n$ over the alphabet $\{a, b, c\}$.
- Turing machines (TMs) use an FSM as a "controller", and an infinite tape of cells for input, output, and storage. They can recognize the class of languages called "Turing-recognizable" (or, alternatively, "recursively enumerable"). But there are languages even TMs cannot recognize – we saw examples in the slides.