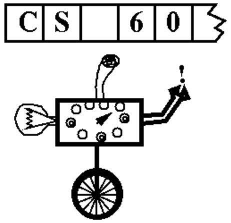
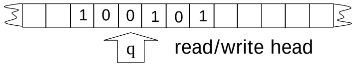


Turing Machines and Undecidability

October 29, 2001

Turing Machine Depiction



infinitely-extendable tape (defaults to blank cells)

(finite alphabet)

read/write head

control state (one of a finite set)

Control is determined by a finite set of **rules** (each a 5-tuple):
For a given state and read symbol, determines the
symbol to write, the next state, and the direction to move.

---

add1.tm

Current	Read	Write	Move	Next State
start	_	_	left	add1
add1	0	1	right	end
add1	_	1	right	end
add1	1	0	left	add1
end	0	0	right	end
end	1	1	right	end

Assumes that head starts to the right of the non-blank symbols on the tape.

Execution Trace

```
 1 0 1 1 1 1 1 1[_]
start
 1 0 1 1 1 1 1[1]_
add1
 1 0 1 1 1 1[1]0 _
add1
 1 0 1 1 1[1]0 0 _
add1
 1 0 1 1[1]0 0 0 _
add1
 1 0 1[1]0 0 0 0 _
add1
 1 0[1]0 0 0 0 0 _
add1
 1[0]0 0 0 0 0 0 _
add1
```

```
 1 1[0]0 0 0 0 0 _
end
 1 1 0[0]0 0 0 0 _
end
 1 1 0 0[0]0 0 0 _
end
 1 1 0 0 0[0]0 0 _
end
 1 1 0 0 0 0[0]0 _
end
 1 1 0 0 0 0 0[0]_
end
 1 1 0 0 0 0 0 0[_]
end
```

Other Examples

- See `/cs/cs60/tm/*.tm`
  - E.g., complete binary adder
  - See `add.tm` and `add.doc`
- Can go on to implement more complex operations
  - Multiplication, comparisons
  - Composition of operations
  - Looping, counters...
  - In fact, can implement any function that C, Java, or rex can compute!
    - Turing machines are a lot more painful to program, though.

Universal Turing Machines

- Any specific Turing machine has a finite description (why?).
  - So, we can put this description on the tape of another Turing machine.
- There exists a Universal Turing Machine
  - Starting with the tape containing a description of any machine T and an input, can simulate the computation of T on that input.
  - i.e., an interpreter for Turing machines.
  - By Church's Thesis, this machine can compute any intuitively computable function.

Church’s Thesis

- a.k.a Turing's thesis, Church-Turing thesis, ...
- Every *intuitively computable* function is computable by some Turing machine
- Generally accepted, but cannot be proved.

Implications of Church's Thesis

- There are non-computable functions
  - e.g., functions mapping natural numbers to the natural numbers which no algorithm can compute
  - Why?
- Remaining question: are there *interesting* problems that are unsolvable?

Non-Proof of Church’s Thesis

- Idea:
  - Formalize the notion of computability
  - Show that a Turing machine can compute every computable function.
- Why doesn't this approach fly?

Disproving Church’s Thesis

- Conceptually this is possible:
  - Find a function that everyone agrees is computable.
  - Prove that no TM can compute it.
- However, *every* attempt so far to characterize computability is equivalent to the TM.
  - Lambda calculus, partial recursive functions, phrase-structure grammars, register machines, ...

Extensions of Church's Thesis

- For a while, some further conjectured that every realistic model of computation could be "efficiently" simulated by a Turing Machine.
- Some evidence this is *not* true, though no proof
  - Randomized algorithms
  - Quantum algorithms

An Unsolvable Problem

- Consider any reasonably rich programming language (such as rex or Java).
- Any computable function can be computed in such a language (given sufficient memory).
- Each program in the language is a finite string of symbols.
- Thus, we can enumerate the set of syntactically-correct programs.

An Unsolvable Problem

- Enumerate the programs: P_0, P_1, P_2, ...
- The possible inputs to those programs can be described as finite strings of bits. They can be enumerated too: I_0, I_1, I_2, ...
- A reasonable question to ask is:
  Does a program P halt on input I, or not?
- This question is **undecidable**
  - No algorithm exists for computing the answer!

Undecidability

- "2-input" question: Does $P_i(I_j)$ halt?
- Let's simplify this to a 1-input question:
  Does $P_i(P_i)$ halt?
  where the argument means the source code of P_i.
  - Seems weird, but programs act on program code all the time (interpreters, compilers, editors, ...)
  - At least as easy to solve as the 2-input question
  - Equivalent to answering the question:
    Does $P_i(P_i)$ diverge?
    (where diverge = not halt)

Does $P_i(P_i)$ diverge?

- Suppose this were decidable.
  - Then there would be a program P_k which computes the answer for *any* input P_i.
- Given this program P_k, we can construct a variant P_m
  - Given an input P_i, return "yes" if $P_i(P_i)$ diverges and intentionally diverge if $P_i(P_i)$ halts.
  - This is computable if halting/divergence were computable

What does $P_m(P_m)$ do?

Esoteric Problems?

- The halting problem, despite its practical attractiveness, may seem far removed from your computing experiences.
- However, many other “every day” problems can be shown to be non-computable.
- This is done by showing that within those problems lies the power to simulate a Turing machine.
- With such power comes inherent limitations.

Post’s Correspondence Problem

- Is there an algorithm that will accept any finite collection of pairs of strings $[x_1, y_1]$ $[x_2, y_2]$ … $[x_n, y_n]$ and determine whether there is a “correspondence”
  $x_{i1}\ x_{i2}\ x_{i3} \ldots x_{in} == y_{i1}\ y_{i2}\ y_{i3} \ldots y_{in}$
  (pairs are allowed to be used multiple times in achieving a correspondence)?
- Example: [a, aaa], [abaaa, ab], [ab, b]
  Correspondence: abaaa a a ab == ab aaa aaa b

An Abstract Generalization

- **Rice’s Theorem:**
  Any non-trivial property of computable functions is undecidable on the set of representations of those functions.
  - Non-trivial means not uniformly true or false for all functions.
- This theorem arises in the study of “Recursion Theory” or “Theory of Computation”.

Other Undecidable Problems

- There are any number of "natural" programming languages for which *typechecking* is undecidable.
- Also, all sorts of questions that a compiler cares about are undecidable:
  - Will this line of code ever be executed?
  - Will these two variables ever refer to the same object in memory?
  - Is this the smallest possible compiler output?