Busy Beaver Lecture

Jason Zhang - Code++

Introduction

What are the limits of computation? Are there some problems that computers will never answer?

Turing Machines

Informal Definition

For the purposes of our lecture, a Turing Machine is a computer operating on an infinitely long tape full of 0s. A Turing Machine has states, each of which dictating what to do depending on what it sees.

“Pseudocode” / Example

```
tape = [infinite 0s going both ways, -inf to pos inf]
curr_pos = 0
state = A
while true:
        if state == A:
                tape[curr] = 1 // or = 0
                curr_pos += 1 // or -= 1
                state = B
        elif state == B:
                // …
        elif state == HALT:
                break
```

Important Things to Note

- For this lecture, you do not need to know the specific technical details of a Turing Machine. Just know that it operates on a tape and that it has some number of states (previous example had 2).
- Sometimes they can run forever.
- Turing Machines can do a LOT of stuff. In terms of computational/algorithmic strength, it is just as powerful as Python or C++.
- More examples: https://turingmachine.io/

Theorem.

There are some problems Turing Machines cannot solve.

Example: Turing Machines cannot generally decide if another Turing Machine will stop running.

Busy Beaver Function

Definition

There are many different definitions, but the current most used one is the “shift function”.

S(n) = the largest number of steps a Turing Machine with n states can take given it halts.

Examples

S(1) = 1

S(2) = 6

S(3) = 21

S(4) = 107

S(5) = 47,176,870

S(6) unknown currently. Would be pretty big though.

S(3)

blank: '0'

start state: A

table:

A:

0: {write: 1, R: B}

B:

0: {write: 1, L: B}

1: {write: 0, R: C}

C:

0: {write: 1, L: C}

1: {write: 1, L: A}

S(5)

- Discovered only really recently
- Very difficult to discover
- Took lots of community collaboration

Why?

Theorem. The Busy Beaver function is uncomputable.

Proof. Suppose S(n) could be simulated by a turing machine, let us call it **EvalS**. Given n 1s on a tape, **EvalS** will write S(n) 1s and then halt. Now, let **Double** be a turing machine that doubles the number of 1s. And let **Clean** be a turing machine that gets rid of all of the 1s. Let's create a combination of turing machines that runs like this: **Double** -> **EvalS** -> **Clean**. Suppose it has n_0 states. We create a new turing machine, **Bad**, that first creates n_0 1s (this takes n_0 states), then does **Double** -> **EvalS** -> **Clean** (another n_0 states, for a total of $N = 2 * n_0$ states). This machine will first write N 1s, then run S(N), then clean up all the 1s. This takes longer than S(N), but this machine has N states, meaning it runs longer than itself, contradiction. □

Consequences

- No computer algorithm can exist that computes S(n).
  - It's not that we haven't found one; it's that we will NEVER find one.
- S(n) grows faster than any computable function.
  - Since turing machines are like computers, eventually n will be big enough to simulate your computable function and thus use it to grow faster.

Progress on S(6)

Progress is slow, people often need to check many different cases or just wait and see what happens. This is why it took so long to solve S(5). It is possible we will not see S(6) get resolved in our lifetimes (or maybe ever… more on that in a second.)

Antihydra

The 6-state turing machines contain hard problems. They simulate machines that behave almost randomly, making it difficult to determine what if a given machine will halt or not. One such machine is called the Antihydra. In plain English:

> **Fun Problem :)** Let $a_0 = 8$ and $a_{i+1} = a_i + \lfloor \frac{a_i}{2} \rfloor$. Does there exist a number m such that the number of odd numbers in the subsequence $a_0, \ldots, a_m$ is strictly more than twice the number of even numbers?

If the answer is yes then the machine halts. Otherwise, the machine doesn't. This problem is similar to other hard problems such as Collatz Conjecture,

Deep Result

For some n_0, for all $n \geq n_0$, we will not know the value of S(n).

Theorem (Gödel). Any sufficiently strong recursively enumerable* mathematical system cannot prove its own consistency.

There are turing machines that run forever if the standard system of math we use is consistent, which violates the above theorem. Therefore, if we know $S(n_0)$ we might implicitly be able to determine if math is consistent or not, which we know we cannot (this is a bit of an oversimplification).

This value is somewhere between 6 and 432. We know for a fact that 432 already exhibits this property, but it could be lower. Some experts think it may be lower than 20.

*sufficiently strong is a bit technical as is recursively enumerable. All you need to know is that pretty much all math you will ever see falls under this category.