Algorithms and Complexity, weeks 1-5 key points

Sam Barrett

May 19, 2021

Turing machine basics I

- Turing machines are **precise models of computation**
- First tape of a Turing machine is **always** a read-only **input tape**.
- The k^{th} tape is always the output tape
- The output tape is also considered a work tape
- The alphabet of a TM is denoted as Γ, or sometimes Σ, it is **finite**.
- Each tape must always start with the *left-of-tape* symbol, $\triangleright$
- $\{0,1\}^*$ is the set of bitstrings, with ε denoting the empty string.

Turing machine basics II

In a single stage of computation a TM may:

- read the character at any/all of the tape heads
- writes a character at any/all of the work tape heads
- may move any/all tape heads to the left or to the right. (tapes are not recursive)

what does it mean to say that M **computes** f?

It means that for every bitstring $x \in \{0,1\}^*$, if we start in state $q_{\texttt{start}}$ with the initial configuration showing x (meaning x appears on the input tape and and the work tapes are blank), when we run M, we eventually reach $q_{\texttt{halt}}$ with the output tape showing $\triangleright$ on the leftmost cell and then the bitstring $f(x)$ followed by all blanks.

Computable functions

Basics

Definition

(Computable functions) We say a function $f : \{0,1\}^* \to \{0,1\}^*$ is **computable** if there exists some Turing machine that computes it.

No variation of Turing machine affects this fact.

Thesis (Church's Thesis (informal))

any algorithm that computes a function from bitstrings to bitstrings can be converted into a Turing machine that computes the same function

Boolean functions, languages and Decidability I

Definition (Language)

A *language* can be defined as any set of words

Definition (Boolean Function)

A **boolean function** is a function of the form: $f : \{0,1\}^* \to \{0,1\}$. Noting that the output is a single bit rather than a bitstring.

There is a one-to-one correspondence between languages and boolean functions.

- For a given boolean function f the corresponding language is the set of bitstrings $\mathcal{X}$ s.t. $\forall x \in \mathcal{X}, f(x) = 1$
- For a language L, the corresponding boolean function sends x to 1 if $x \in L$ and to 0 otherwise.

Boolean functions, languages and Decidability II

This allows us to treat boolean functions, languages and decision problems as essentially the same thing.
A decision problem is said to be **decidable** when the corresponding boolean function is **computable**. I.e. given a language L, for L to be decidable there must exist some Turing machine that will start with a bitstring x and will run continuously until it halts and upon halting there will be a 1 on the output tape if $x \in L$ or 0 if it is not in the language.

Data Representation

- We can encode many real-life data types as bitstrings, but not all. (e.g. Real numbers cannot)
- We can encode multiple inputs as a single bitstring.

Code as Data

- We can not only encode many data types as bitstrings, we can encode program code or even other Turing machines as bitstring inputs to a TM as our TMs are essentially 6-tuples (see full notes for formal definition).
- We can therefore say that for **any** bitstring α we can construct a corresponding TM: M_α

The Universal Turing Machine, $\mathcal{U}$

$\mathcal{U}$ is a Turing machine interpreter written as a Turing machine. It takes 2 inputs (encoded as a single input): α and x, where α is the bitstring describing the machine to be interpreted and x is the bitstring input.
We define $\mathcal{U}$ as having 4 tapes and the basic alphabet of $\{\triangleright, \square, 0, 1\}$. Intuitively, $\mathcal{U}$ works by simulating the M_α by *providing* it with the 3 non-input tapes to M_α as its input, work and output tape respectively.

Diagonalisation & the Halting problem I

Problem

(The halting problem) the set of pairs $\langle \alpha, x \rangle$ (encoded as a single bitstring) such that the machine M_α executed on input x halts.

Diagonalisation & the Halting problem II

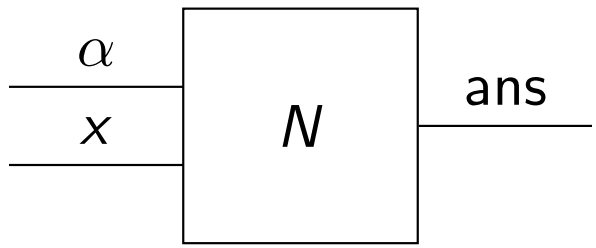
Turing's proof is as follows:

Proof.

Suppose that N is a machine that solves the halting problem.
We can convert it into a machine N' that, given x, runs forever if $\langle x, x \rangle \in \texttt{HALT}$ (i.e. the machine M_x executed on x halts), and halts otherwise.
We know $N' = M_\alpha$ for some α, i.e. there exists some bitstring α that represents our new machine, as we know every machine can be represented as a bitstring.
Running N' on α halts if it runs forever and runs forever if it halts.
We have derived a contradiction.

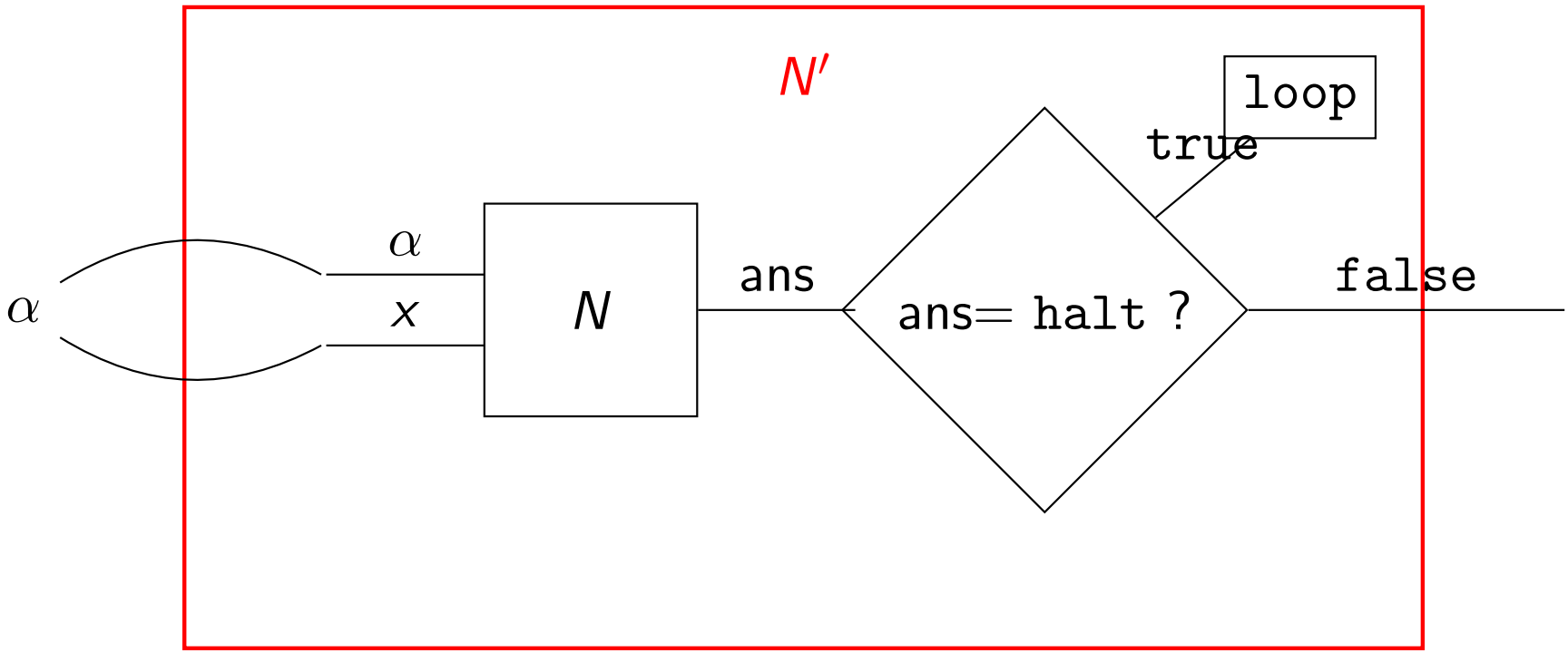
Diagonalisation & the Halting problem III

More intuitively, we can consider the described machine N as:



Where `ans` is whether N halts.
We can then construct the wrapper N' as:

Diagonalisation & the Halting problem IV



Where the outermost α is the bitstring of N'

You can clearly see that if N were to halt then N' would hang. We can therefore derive a contradiction as if we pass N' into N' it would have to half if it hangs and vice versa!

Upper bound notation

- If we have two function $f : \mathbb{N} \to \mathbb{N}$ and $g : \mathbb{N} \to \mathbb{N}$
- We say that $f(n)$ is $O(g(n))$ is f is **no bigger** than g up to **a** constant factor, i.e. given a constant c and a value n_0 s.t. $\forall n, n \geq n_0$ we have:

$$f(n) \leq c \cdot g(n)$$

- We say that $f(n)$ is $o(g(n))$ if f is **not as big as** g, even up to any constant factor. Or, if, **for any** $\varepsilon > 0$, there is a n_0 s.t. $\forall n, n \geq n_0$ we have:

$$f(n) \leq \varepsilon \cdot g(n)$$

Whenever $f(n)$ is $o(g(n))$ is it also $O(g(n))$, this is the case if you take c to be 1

Lower bound notation

- If we have two function $f : \mathbb{N} \to \mathbb{N}$ and $g : \mathbb{N} \to \mathbb{N}$
- We say that $f(n)$ is $\Omega(g(n))$ when $g(n)$ is $O(f(n))$
- We say that $f(n)$ is $\omega(g(n))$ when $g(n)$ is $o(f(n))$
- We say that $f(n)$ is $\Theta(g(n))$ when it is **both** $O(g(n))$ and $\Omega(g(n))$
  This informally means $f(n)$ *and* $g(n)$ *are the same, up to a constant factor*

Time complexity

Definition (Worst case running time:)

The running time of a machine M is the time taken from the input state, where x sits on the input tape and the other tapes are blank, to reach the halt state ($q_{\texttt{halt}}$).
For any number n, we define $\texttt{WT}_M(n)$ to be the **worst case** running time for an input of length n.

Definition (DTIME classes)

DTIME is a **set** of complexity classes.
A complexity class can be thought of as a set of decision problems (or languages or boolean functions...)
DTIME(n^2) is an example of a complexity class. A function $f : \{0,1\}^* \to \{0,1\}$ is in DTIME(n^2) when there is some machine that decides it and has worst case running time in $O(n^2)$

This definition is robust.

Polynomial Time

We can define the complexity (super) class of **polynomial time decision problems** as:

$$\mathrm{P} \stackrel{\text{def}}{=} \bigcup_{k \geq 1} \mathrm{DTIME}(n^k)$$

Exponential Time

We can define the complexity class of **exponential time decision problems** as:

$$\mathrm{EXP} \stackrel{\mathrm{def}}{=} \bigcup_{k \geq 1} \mathrm{DTIME}(2^{n^k})$$

It is clear that $\mathrm{P} \subseteq \mathrm{EXP}$

Space Complexity

We can extend this notion to space usage of our models.
The space usage of a Turing Machine for an input x is the number of cells on the **work tapes** that are non-blank at **some point during execution**. Blank cells are ignored as there are infinitely many of these.

The SPACE Complexity class

The spacial analogue for DTIME we say a problem is in SPACE(n^2) if there exists a TM which decides it and has worst case space usage in $O(n^2)$

L & PSPACE

We can define logarithmic space, L which defines the set of things that can be computed with a machine using a logarithmic number of cells (relying on the fact we do not count the input tape cells).

Definition (Logspace)

$$\mathsf{L} \stackrel{\text{def}}{=} \mathsf{SPACE}(\log n)$$

We can also define polynomial space, PSPACE:

Definition (polynomial space)

$$\mathsf{PSPACE} \stackrel{\text{def}}{=} \bigcup_{k \geq 1} \mathsf{SPACE}(n^k)$$

It is clear that $\mathsf{L} \subseteq \mathsf{PSPACE}$

Space vs. Time

We can show that, in **all cases**, space complexity is less than or equal to time complexity, meaning also that:

$$P \subseteq PSPACE$$

See notes for a proof

Equally we can show that, in all cases, time is less than or equal to exponentiated space, i.e.

$$L \subseteq P$$

See notes for a proof

Nondeterministic Time Complexity

We can informally define NP as:

Definition (NP - informal)

Problems for which **checking** a solution is easy

We have two methods for formally defining NP:

1. Using certificates
2. Using nondeterministic Turing machines

For both, we will use the example of n-Sudoku. Where n describes the dimension of the grids.
Let SUD be the set of **solvable** n-Sudoku puzzles.
Given a puzzle x, a solution **certifies** that $x \in$ SUD The size of a solution is polynomial in $|x|$. The time taken to check a candidate solution is also polynomial in $|x|$

Using Certificates

Definition (NP - certificates)

A language L is in NP if there is a polytime machine for checking polynomially-sized certificates of L, precisely:
If there is a polynomial, p, which gives the size of a candidate certificate, and a polynimal time machine M s.t. $\forall x \in \{0,1\}^*$, where x is a bitstring representing an n-Sudoku puzzle, the following are equivalent:

- $x \in L$
- There is some bitstring u representing a solution to x, of length $p|x|$ s.t. $M\langle x, u\rangle = 1$

In these cases we say that u certifies that $x \in L$

Using Nondeterministic Turing Machines I

Nondeterministic TMs are TMs which have two transition functions δ_0, δ_1 and an additional accepting state q_{accept}. It starts at $q_{\texttt{start}}$ and follows transitions from either transition function until it reaches either $q_{\texttt{halt}}$ or q_{accept}, at which point no more transitions take place.

Using Nondeterministic Turing Machines II

Definition (NP - Nondeterministic TMs)

A language L is in NP when there is a polytime nondeterministic machine M, s.t. $\forall x \in \{0,1\}^*$ the following are equivalent:

- $x \in L$
- When M is executed on input x, there is some sequence of transitions that lead to q_{accept}

Nondeterministic Space complexity

What does it mean for a nondeterministic Turing machine (NDTM) to be in nondeterministic polynomial space complexity? Let M be a NDTM, it is in polyspace if $\mathtt{WS}_M$ is $O(n^k)$ for some $k \geq 1$.
The worse case space complexity is polynomial, therefore NPSPACE is the class of languages that can be decided by a polyspace NDTM. We can therefore say:

$$\text{PSPACE} \subseteq \text{NPSPACE}$$

in the same way that we say

$$\text{P} \subseteq \text{NP}$$

Savitch's Theorem: PSPACE = NPSPACE

We can show this equivalence using a special case of Savitch's theorem.
Suppose we have a NDTM, M and for an input size n the space usage is polynomial in n. The length of a configuration of M is also polynomial in n.
Let us say for $n \geq 1000$ a configuration has a length at most $7n^{18}$.
Consider the digraph representing all $\leq 2^{7n^{18}}$ configurations of M and the transitions linking them.
With this graph, we want to, using a space-efficient algorithm, work out if there is a path from the start configuration to any accepting configuration. If such a path exists, we know that the input is accepted.

Finding a path space-efficiently I

We find this required path by generalising our problem.
Given nodes s and t and a number k, how much space do we use when deciding whether there is path of length $\leq 2^k$ from $s \to t$?
We will assume we have some function which calculates this, D, where $D(k)$ returns this result.
We will now argue, by induction, that $D(k) \leq k \cdot 7n^{18}$
For each configuration t, check if t is accepting and whether there is a path from the start configuration to t.
Storing t requires at most $7n^{18}$ bits, $D(7n^{18}$ bits to check for the path, $\therefore$ at most $7n^{18} + D(7n^{18})$ bits total.
This is polynomial, as is required by Savitch's theorem.

Finding a path space-efficiently II

Proof.

Base Case If $k = 0$, just check if $s = t$

Inductive step Find whether there is a path from s to t of length $\leq 2^{k+1}$:

1. For each node z, test whether there is a path from s to z of length $\leq 2^k$ and a path from z to t of length $\leq 2^k$
2. By our inductive hypothesis, this takes $\leq k \cdot 7n^{18}$ cells, plus a further $7n^{18}$ cells to store z, totalling $\leq (k+1) \cdot 7n^{18}$
   $\therefore D(k+1) \leq (k+1) \cdot 7n^{18}$
   $\therefore \forall k D(k) \leq k \cdot 7n^{18}$

□