Computational Complexity

Luke Mathieson

October 1, 2020

1 What can we decide?

Rice's theorem demonstrates that many of the interesting questions we can ask about Turing Machines are undecidable, and worse, undecidability is not limited to these questions, we can't even tell if a PDA accepts every string[1]. This may lead to the question "why bother with Turing Machines then?". Everything interesting that we could do so far has been possible with some weaker model of computation.

One practical outcome of examining Turing Machines results when we ask slightly more refined questions; "what can we decide with a limited amount of resources?". One central problem with Turing Machines is that they have an infinite tape, thus it is difficult to decide whether they are looping infinitely, or just engaged in some long computation.

The obvious first step is to limit the tape.

2 Linear Bounded Automata

A *linear bounded automaton* (LBA) is a Turing Machine with the additional restriction that only a contiguous section of the tape whose length is linear in the size of the input is used[2]. LBAs recognise the *context sensitive languages*, and their grammars allow rules of the form $\alpha X \beta \to \alpha \gamma \beta$ where $\gamma = (\Sigma \cup V)^*$.

As LBAs have a finite section of tape available, they are perhaps slightly closer to modern computers, but more importantly, we can tell when they're stuck in an infinite loop.

As an LBA can only use cn cells of the tape, where c is some constant and n is the number of symbols in the input, it can only have a finite number of configurations, namely $|Q| \times cn \times |\Gamma|^{cn}$; *i.e.* the combination of the number of states the LBA can be in, times the number of spots the head can be at, times the number of different strings that can be written on the tape. Thus if an LBA arrives at the same configuration twice, it must be stuck in an infinite loop[3] Thus the halting problem for LBAs ($HALT_{LBA}$), along with the acceptance problem (A_{LBA}), is decidable.

Unfortunately, the empty language problem for LBAs (E_{LBA}) and the universal acceptance problem for LBAs (ALL_{LBA}) remain undecidable. It seems that allowing TMs that may not halt is simply too difficult to handle.

3 Moving Toward Things We Can Handle

The apparently inherent undecidability in asking questions about Turing Machines suggests that the things we do day-to-day on computers are not normally questions about Turing Machines[4]. For most things we attempt to do, we know that given enough time, we could solve them, by brute force if nothing else. Thus the problems that we normally face are *decidable* problems, and the computational issues surround how *efficiently* we compute them. To formalise these ideas, we, of course, need to define some concepts.

Definition 3.1 (Input Size)**.** The size of the input to a Turing Machine is the number of symbols in the input. Equivalently, the number of cells of the tape the input uses, or the length of input viewed as a string.

Typically we will denote the size of the input with the symbol n, unless otherwise specified.

For practical purposes however, we will often use *proxies* for the length of the input. That is, we may use other aspects of the input as our measure. This requires some care however, an issue we will return to later.

Definition 3.2 (Computational Step)**.** A computational step is a single transition of a Turing Machine.

[1] This is also essentially why we do not have an equivalent to the Myhill-Nerode Theorem for Context Free Languages.

[2] A computationally equivalent definition uses explict begin and end markers as boundaries.

[3] Note that non-determinism doesn't help here – if there were a path to an accepting configuration from the repeated configuration, a non-deterministic machine would have chosen that path the first time, as there's no reason to repeat exactly the same configuration.

[4] Although some things we'd like to be able to do are – for example checking a piece of code for correctness.

As with input size, it will often be acceptable to take any constant-time operation as a single step, once we define what constant-time means.

3.1 Limiting Computation

With Definitions 3.1 and 3.2 we have enough to establish the basics of *computational commplexity*.

Definition 3.3 (Time Complexity of a Language)**.** A language L has time complexity $f(n)$ for some function f if there exists a Turing Machine that, given an input x, with $|x| = n$, decides $x \in L$ in a number of steps bounded by $f(n)$.

Definition 3.4 (Space Complexity of a Language)**.** A language L has space complexity $f(n)$ for some function f if there exists a Turing Machine that, given an input x, with $|x| = n$, decides $x \in L$ using at most $f(n)$ additional tape cells.

Note that we make no mention of determinism here. Also there is nothing unique about the Turing Machines in these definitions in the sense that a language can be assigned many complexities. Normally of course we are interested the lowest possible complexity.

3.2 Classes of Functions

Although each language has its own complexity, we would like a more robust classification of groups of languages. In one sense, we want to be able to say "yes we can solve this problem" without requiring that we determine the best possible complexity for the language, or that we specify the exact details of the computer we are using.

To do this, we need a more robust way of measuring complexity. To this end, we will use asymptotic notation to create equivalence classes of functional bounds. This turns out to be quite useful as the rate of growth of a function is much more important than the constant multipliers in determining complexity classifications. Then we can group things bounded by some $f(n)$ together in the class $O(f(n))$. For example, the class of languages decided by LBAs can be expressed as $O(n)$-SPACE (also known as LINSPACE).

As a starting point for a definition of *efficient computation*, we will use the Cobham-Edmonds Thesis:

Conjecture 3.5 (Cobham-Edmonds Thesis)**.** Languages can be efficiently computed on some computational device only if they can be decided in $O(n^c)$ for time on a Turing Machine.

Or to put it another way, a language can be efficiently computed when it has polynomial time complexity. This thesis certainly has limitations, but it provides a fairly robust initial basis.

This gives our first complexity class: **P**.

3.3 Polynomial Time

The class **P** (short for polynomial time) is defined as follows:

Definition 3.6 (Polynomial Time)**.** A language L is in class **P** if there exists a constant c and a deterministic Turing Machine that, given input $x \in \Sigma^*$ with $|x| = n$, decides $x \in L$ in at most $O(n^c)$ steps.

P is the basic class of efficiently computable languages. Although phrased here in terms of languages, we can substitute the idea of *decision problems* for an entirely equivalent definition:

Definition 3.7 (Decision Problem)**.** A decision problem is a computational problem that consists of two components:

1. the input, and
2. a yes/no question.

It is easy to see that this is the same as deciding language membership. What may be less obvious is that it bears direct relationships to other forms of problems:

- Search Problems – where we ask for a solution, and
- Optimization Problems – where we are given an optimization criterion and measure and ask for a solution with the best possible value.

Thus this class, although phrased in terms of decision problems, acts as a useful classification for the practical version of problems where we actually want the solution. Many practically useful problems are in **P**:

- Searching
- Sorting

- Shortest Paths in Graphs
- Minimum Spanning Trees
- Linear Programming
- Maximum Flow
- Primality Testing
- Every context free language
- Circuit Value Problem
- Horn-Satisfiability

Most day-to-day algorithms are combinations of these basic problems.

3.4 Nondeterministic Polynomial Time

The definition of **P** explicitly requires a *deterministic* Turing Machine. What happens when we allow a nondeterministic Turing Machine? This gives the class **NP**:

Definition 3.8 (Nondeterministic Polynomial Time)**.** A language L is in class **NP** if there exists a constant c and a *nondeterministic* Turing Machine that, given input $x \in \Sigma^*$ with $|x| = n$, decides $x \in L$ in at most $O(n^c)$ steps.

Interestingly, this turns out to be the same as the class of problems that a *verifiable* in deterministic polynomial time:

Definition 3.9 (**NP** Verifier Definition)**.** A language L is in class **NP** if there exists two constants c and d and a *deterministic* Turing Machine that, given input $x \in \Sigma^*$ with $|x| = n$ and a certificate $y \in \Sigma^*$ with $|y| \in O(n^d)$, decides $x \in L$ in at most $O(n^c)$ steps.

That is, given the input and a polynomial-length proof that the input is a yes instance, the deterministic turing machine can decide the problem in polynomial time.

With several machine models so far, nondeterminism has added no extra power; any NFA can be turned into a DFA, any NTM can be simulated by a DTM. Of course NPDAs are more powerful than DPDAs, and the question remains open for LBAs. What happens with time-restricted TMs?

This is, of course, the most prominent open question in computer science: is **P** a strict subset of **NP**, or are the they same? Or phrased another way, does nondeterminism save us time?

It should be clear that $\mathbf{P} \subseteq \mathbf{NP}$ – we can simply not use any nondeterminism, thus every problem/language in **P** is also in **NP**, but is there anything in **NP** that is not in **P** if they turn out to be different?

3.5 NP-completeness

One major class of problems apparently outside **P** is **NP**-complete. Informally, this is the hardest subset of problems in **NP**.

To establish **NP**-completeness we need a way of relating problems to one another: reductions!

Definition 3.10 (Karp Reduction)**.** A *polynomial-time many-one* reduction[5], denoted $\leq^m_{\mathbf{P}}$, from language/problem A to language/problem B is a function $f : A \to B$ where given an input x, f is computable in time bounded by $|x|^{O(1)}$ and $x \in A \Leftrightarrow f(x) \in B$.

Example 3.11. Consider the following two problems:

> 3-SAT
> *Instance:* A set of boolean variables Var, a set of disjunctive clauses Cl of size 3 over literals[6] of Var.
> *Question:* Is there an assignment $\alpha : Var \to \{\mathsf{True}, \mathsf{False}\}$ such that every $c \in Cl$ evaluates to True?
>
> Vertex Cover
> *Instance:* A graph $G = (V, E)$, an integer k.
> *Question:* Is there a set $V' \subseteq V$ with $|V'| \leq k$ such that for every $uv \in E$ either $u \in V'$ or $v \in V'$ (or both)?

We can show that 3-SAT $\leq^m_{\mathbf{P}}$ Vertex Cover.

Given an instance $(Var, C)l$ of 3-SAT, we construct an instance (G, k) of Vertex Cover as follows:

[5] Or Karp Reduction, after Richard Karp.

[6] A literal of a boolean variable is the negated or unnegated instance that appears in a formula. *e.g.* the literals of variable v are (confusingly) v and $\bar{v}$.

- for each $v \in Var$, we construct a variable gadget where we add two vertices v and $\bar{v}$ to V, and the edge $v\bar{v}$ to E,
- for each $c \in Cl$ with literals l^1, l^2 and l^3, we construct a clause gadget where we add three vertices l^1_c, l^2_c and l^3_c to V, three edges $l^1_c l^2_c$, $l^1_c l^3_c$ and $l^2_c l^3_c$ to E and for each i, we add the edge $l^i_c v$ to E, where v is the corresponding literal from the variable gadget[7], and
- set $k = |Var| + 2 \cdot |Cl|$.

Lemma 3.12. *If (Var, Cl) is in* 3-SAT*, then (G, k) is in* Vertex Cover.

Proof. As (Var, Cl) is in 3-SAT, there is an assignment $\alpha : Var \to \{\mathsf{True}, \mathsf{False}\}$ such that each clause evaluates to True. We select a vertex cover (the set V') as follows:

- if $\alpha(v) = \mathsf{True}$, add the vertex v from the variable gadget to V', otherwise add the vertex $\bar{v}$ from the variable gadget to V',
- for each clause, at least one literal evaluates to True, add the vertices corresponding to the other two literals to the set V', if more than one literal evaluates to True, select one arbitrarily.

Claim 3.13. *V' is a vertex cover for G.*

To be a vertex cover, each edge in G must have at least one endpoint in V'. The edge added by each variable gadget is covered as we select one of the endpoints based on α. For each triangle added by a clause gadget, all three edges are covered as we choose two of the three vertices. It only remains to examine the connections between the variable and clause gadgets. As each clause has at least one true literal, the edge between that literal vertex and its corresponding variable gadget vertex is covered by the variable gadget vertex. The two other edges are covered by the remaining literal vertices in the clause gadget. Therefore all edges are covered, and V' is a vertex cover.

It is also clear that $|V'|$ is $|Var| + 2 \cdot |Cl|$ by construction. Therefore (G, k) is in Vertex Cover. □

Lemma 3.14. *If (G, k) is in* Vertex Cover*, then (Var, Cl) is in* 3-SAT.

Proof. As (G, k) is in Vertex Cover, there is a set $V' \subseteq V$ with $|V'| \leq k = |Var| + 2 \cdot |Cl|$ that is a vertex cover for G.

As V' is of size at most k, we must have exactly one vertex from each variable gadget in V' and two vertices from each clause gadget in V'. As there is one vertex not in V' in each clause gadget, the corresponding vertex in the variable gadget must be in V'. Thus we construct a satisfying assignment α for (Var, Cl) by setting a variable to True the variable gadget vertex corresponding to its unnegated literal is in V', and False if its negated variable is in V'. As each clause has at least one such variable, α satisfies each $c \in C$. □

Lastly, we need to check that the reduction is computable in polynomial time.

Lemma 3.15. *The reduction is computable in time bounded by $O(|\langle (Var, Cl) \rangle|^c)$.*

Proof. For each variable, we add two vertices and an edge, each operation taking a constant number of steps. For each clause we add three vertices and three edges, each taking a constant number of steps, and for each vertex in the clause gadget, we must find the corresponding vertex in the variable gadget and add an edge.

Let $|Var| = n$ and $|Cl| = m$. As we can label each variable and clause in $O(\log(n + m))$ bits, we can construct variable labels using at most $O(\log((n + m)^2)) = O(\log(n + m))$ bits as well. Thus the steps above take at most $O(nm \log(n + m))$ steps. □

In a similar fashion to Turing reductions giving a way to solve one problem if you know how to solve another, Karp reductions give a way to solve one problem in polynomial time if you know how to solve another in polynomial time (and you can construct the reduction). If $A \leq^m_{\mathbf{P}} B$, and $B \in \mathbf{P}$, then we can conclude that $A \in \mathbf{P}$ as well, as we can convert from A to B in polynomial time, solve B in polynomial time, and hence get an answer to A.

Like Turing reductions, this observation will actually be most useful demonstrating that the second problem (probably) doesn't have a polynomial time algorithm.

Definition 3.16 (**NP**-hard)**.** A problem Π is **NP**-hard if, for every problem $\Psi \in \mathbf{NP}$, there is a Karp reduction $\Psi \leq^m_{\mathbf{P}} \Pi$.

That is, Π could be used to solve everything in **NP** in polynomial time, if we knew a polynomial time algorithm for Π. Of course, finding a polynomial time algorithm for any **NP**-hard problem would immediately resolve the question of $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$. Of course there are many problems that are **NP**-hard that obviously don't have polynomial time algorithms, so to say something a touch more interesting, we can refine this notion:

[7] *e.g.* if $l = \bar{x}$, we add the edge between l in the clause gadget and the $\bar{x}$ vertex in the variable gadget for x.

Definition 3.17 (**NP**-complete)**.** A problem Π is **NP**-complete if:

- $\Pi \in$ **NP**, and
- Π is **NP**-hard.

Thus the class **NP**-complete is the set of problems/languages that are at least as hard as anything in **NP**, at least up to some polynomial factor.

To make use of this, we need to establish that at least one problem is **NP**-complete (as it is unlikely that something outside **NP** can be reduced to something in it). Fortunately, the Cook-Levin theorem gives us:

Theorem 3.18 (Cook-Levin Theorem)**.** SAT *is* **NP***-complete.*

Where SAT is like 3-SAT, but we have no bound on the size of the clauses. Luckily it is not hard to also show that SAT $\leq_{\mathbf{P}}^{m}$ 3-SAT, so 3-SAT is also **NP**-complete.

Given this, can show the following:

Theorem 3.19. Vertex Cover *is* **NP***-complete.*

Proof. We have already shown that 3-SAT $\leq_{\mathbf{P}}^{m}$ Vertex Cover, demonstrating that Vertex Cover is **NP**-hard.

We need only show that Vertex Cover $\in$ **NP**.

We give a nondeterministic guess-and-check algorithm for Vertex Cover:

1. Given a graph G and an integer k, guess k vertices from V to form the vertex cover V'.
2. Check that these vertices do form a vertex cover.

If G does have a vertex cover of size at most k, the nondeterminism will guarantee that a suitable set of k vertices is chosen. We can see that this is computable in polynomial time; for each edge in E , we need to check that one of its two endpoints is in V', this takes time $O((|E| \cdot |V|) \log |V|)$. The initial guessing takes time $O(|V| \log |V|)$

We can also use the verifier definition, where the proof string y essentially takes the place of the "guess" component of the algorithm. Given a set of k vertices, we can verify that they are a vertex cover in polynomial time using the above algorithm.

Thus Vertex Cover is in **NP**. □

3.6 NP-complete Problems

Unfortunately, many of the interesting problems in the world turn out to be **NP**-complete (the formal versions given in parentheses):

- Timetabling (Coloring)
- Scheduling in many forms (General Scheduling)
- Route finding (Travelling Salesman, Hamiltonian Path)
- Optimal Resource Allocation (Knapsack, Bin Packing, Partition)
- Detecting features in data mining (Feature Set)
- Place facilities (Dominating Set, Independent Set, Vertex Cover)
- Determining maximum consistent sets of observations (Vertex Cover)
- Virtually anything with constraints (SAT, Integer Linear Programming, Constraint Satisfaction Problem)

The full list of known **NP**-complete problems numbers in the tens of thousands.

Of course we would still like to solve these problems, however **NP**-completeness gives us strong evidence that they have no polynomial time algorithm[8]. Fortunately, **NP**-completeness not actually a complete death sentence for solvability of a problem. Some problems (e.g. Knapsack) have *pseudo-polynomial* time algorithms[9]. Others, like SAT, are often amenable to very clever, but still exponential time algorithms[10].

[8] Sure someone clever would've found one by now...

[9] Once you start including numbers in your input, you get interesting results, as n-bits can store a number up to 2^n, so the meaning of the input is not as tightly linked to the length of the input.

[10] A good exponential can be better than a bad polynomial for reasonable input sizes.