NP-Completeness and the Satisfiability Problem

Fern Warwick 2261449

April 3, 2025

1 Introduction

In 2000, the Clay Mathematics Institute published the Millenium Prize Problems - a set of seven open problems which each carry a $1,000,000 prize for their proof or disproof (Jaffe 2005). One of these problems is the $\mathcal{P}$ versus $\mathcal{NP}$ problem, one of the most famous problems in theoretical computer science which was introduced in 1971 by Stephen Cook (Cook 1971b). This problem was a major development in the area of **Complexity Theory**, which explores how fast computers are able to compute a program. In this report we explore the development of Complexity Theory, in particular focusing on the satisfiability problem, which was the first problem shown to be NP-Complete (Garey and Johnson 1979).

2 Turing Machines

2.1 Deterministic Turing Machines

The first computational models created by computer scientists come from Automata Theory, which is the study of abstract automata or 'machines' (Hopcroft, Motwani, and Ullman 2014). In the 1930s Alan Turing and Alonzo Church studied a particular type of automata known as **finite-state automata** and built on this with the development of the **Turing Machine**. We utilise this model due to its flexibility and versatility. There are many variations on the model, but we begin with the one-tape **Deterministic Turing Machine** (DTM).

The DTM contains a **finite state control**, a **read-write head**, and an infinite sequence of **tape squares**, labelled $\ldots, -2, -1, 0, 1, 2, \ldots$. By convention we always set the start of our input at position 1, which we refer to as the **initialisation** of our Turing Machine.

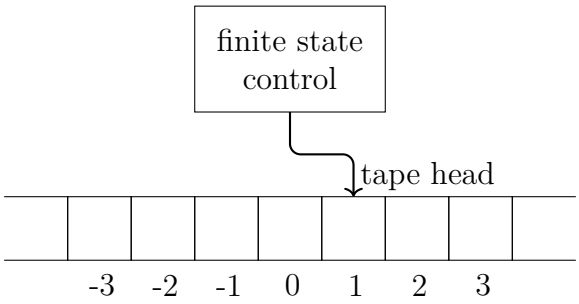

Figure 1: A one-tape DTM that has been initialised.

There is also a set number of states that the DTM can be in, which influence the action the finite state control takes. When the tape head reads a cell, based on the value of the cell and the state it will do three things:

- Rewrite the item on the tape head;
- Choose to move the tape forwards or backwards one cell, or stay on the same cell;
- Optionally change the state of the DTM.

We now provide a formal definition.

Definition 2.1. *(Garey and Johnson 1979) A **program** in a one-tape Deterministic Turing Machine (DTM) is a tuple (Γ, Q, δ) such that:*

- *Γ is a finite set of tape symbols, including a subset $\Sigma \subset \Gamma$ **input symbols** and a **blank symbol** $b \in \Gamma \setminus \Sigma$;*
- *Q is a finite set of states, including a unique start-state q_0 and two unique halt-states q_Y and q_N;*
- *δ represents a transition function $\delta : Q \setminus \{q_Y, q_N\} \times \Gamma \to Q \times \Gamma \times \{-1, 0 + 1\}$.*

*In our set of tape symbols, the subset Σ is referred to as an **alphabet**.*

For the sake of brevity, in this report we refer to a program in a one-tape DTM as a DTM. Now that we have defined a program, what is left is to consider the possible set of inputs into a program:

Definition 2.2. *Consider an alphabet $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$. We denote Σ^* to be the set of all finite strings of symbols from Σ. A **language** is a subset $L \subseteq \Sigma^*$ and an **instance** or **input** is an element $x \in \Sigma^*$.*

Remark 2.3. *By convention the string of length 0 is given by ϵ.*

As an example, for the binary alphabet $\{0, 1\}$ we have

$$\{0, 1\}^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, \ldots\}.$$

2.1.1 Encodings

The physical manifestation of a Turing Machine tends to take the form of either an electronic computer or a physical tape passing through a tape head. In either case, the tape head can only read up to two pieces of information. In the case of an electronic TM, either there is electricity passing through the tape head or no electricity. In the case of a paper tape, there is either a punched hole or no punched hole. Therefore, for our Turing Machine model to be faithful to a real-world model, we implement it using the alphabet $\{0, 1\}$.

There are many different items that we will want to represent in our DTM, which we need to represent through this alphabet; we use a process called **encoding** to represent them.

To represent a number, we note that we can use a representation known as **binary**. Typically we use a system called **base 10**, which means that a number like 354 can be also written as $3 \cdot 10^2 + 5 \cdot 10^1 + 4 \cdot 10^0$. We can represent a number in base 2, or **binary**, in a similar way:

$$\begin{aligned} 9 \text{ in base } 10 &= 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\ &= 1001 \text{ in base } 2. \end{aligned}$$

We can also represent graphs in binary; for an undirected graph, we can label the vertices with some indexing set and create a matrix such that

$$a_{i,j} = \begin{cases} 1 \text{ if there is an edge between } v_i \text{ and } v_j \\ 0 \text{ otherwise} \end{cases}$$

Such a matrix is called an **adjacency matrix**. We then input it row by row into the TM, which acts as an encoding for our graph, as shown in Figure 2.

$$\begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix}$$

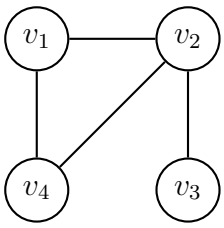

Figure 2: An adjacency matrix and its corresponding graph.

2.1.2 Decision problems

One use case of the Turing Machine is to solve problems.

Definition 2.4. *A decision problem, Π, consists of a set D_Π of **instances** and a subset $Y_\Pi \subseteq D_\Pi$ of **yes-instances.***

A common way to give decision problems (and how they will be referred to in this report) is in the **instance-problem** form. Here is an example decision problem:

DIVISIBILITY BY FOUR
INSTANCE: A natural number x.
PROBLEM: Is x divisible by 4?

Remark 2.5. *Any decision problem describes a language, as the set of yes-instances (given by instances for which the answer to the problem is 'yes') is a subset of all possible instances in Π.*

Definition 2.6. *Let Σ be the alphabet for a program M in a Deterministic Turing Machine, and consider an instance $x \in \Sigma^*$. We say that M **halts, computes** x or **decides** x if there exists $T \in \mathbb{N}$ such that for all $t \geq T$, the state, location and contents of all cells of M after t steps is equal to that after $t+1$ steps. Furthermore, we say that*

- *M **accepts** x if M halts on state q_Y;*
- *M **rejects** x if M halts on state q_N;*
- *The language L_M **recognised** by M is given by the set*

$$L_M = \{x \in \Sigma^* : M \text{ accepts } x\}$$

Example 2.7. *We can compute the decision problem DIVISIBILITY BY FOUR using a Turing Machine.*

Set $\Sigma = \{0, 1\}$ and let $Q = \{q_0, q_1, q_2, q_Y, q_N\}$. Lastly, set our transition function as below:

	0	1	b
q_0	$(q_0, 0, +1)$	$(q_0, b, +1)$	$(q_1, 1, -1)$
q_1	$(q_2, 0, -1)$	$(q_N, 1, +1)$	$(q_N, b, +1)$
q_2	$(q_Y, 0, +1)$	$(q_N, 1, +1)$	$(q_N, b, +1)$

Table 1: A table of the transition function for M. An output from the transition function is taken by taking the input symbol along the top row and the current state along the left hand column and reading the corresponding cell.

Consider the number 28. We know that 28 is equal to 7 times 4, so it should return a successful result. Indeed, it does:

Current state				*Tape*					*Transition function*
q_0	b	**1**	1	1	0	0	b		$(q_0, 1, +1)$
q_0	b	1	**1**	1	0	0	b		$(q_0, 1, +1)$
q_0	b	1	1	**1**	0	0	b		$(q_0, 1, +1)$
q_0	b	1	1	1	**0**	0	b		$(q_0, 0, +1)$
q_0	b	1	1	1	0	**0**	b		$(q_0, 0, +1)$
q_0	b	1	1	1	0	0	**b**		$(q_1, b, -1)$
q_1	b	1	1	1	0	**0**	b		$(q_2, 0, -1)$
q_2	b	1	1	1	**0**	0	b		$(q_Y, 0, +1)$

SUCCESS

By inspection of the transition function, it can be seen that the program recognises the language

$$L_M = \{x \in \{0,1\}^* : \textit{The two rightmost symbols of } x \textit{ are both } 0\}.$$

2.2 The multitape DTM

From our initial computational model, we would like for there to be a variety of uses for the Turing Machine. While implementing this, we run into a rather striking problem; the time taken for computations for larger languages grows considerably.

Increasing the size of the alphabet is not feasible, so we instead introduce a DTM on multiple tapes. These have separate read and write heads, though there is still only one finite state control.

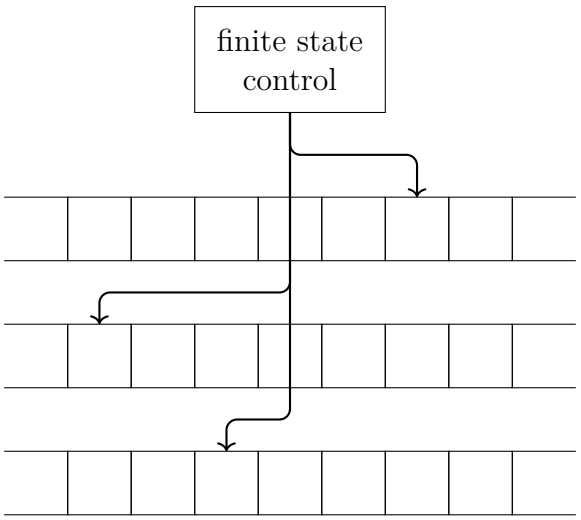

Figure 3: A 3-tape DTM. The tape heads all move independently, but the DTM only works in one state at any time.

Definition 2.8. *A program in a k-tape DTM is a tuple (Γ, Q, δ) such that:*

- *Γ is a finite set of tape symbols, including a subset $\Sigma \subset \Gamma$ of input symbols and a unique blank symbol $b \in \Gamma \setminus \Sigma$;*
- *Q is a finite set of states, including a unique start-state q_0 and two unique halt-states q_Y and q_N;*
- *δ represents a transition function $\delta : Q \setminus \{q_Y, q_N\} \times \Gamma^k \to Q \times \Gamma^k \times \{-1, 0, 1\}^k$.*

We note that Definition 2.6 also holds for a k-tape DTM, and therefore we do not need to redefine computation on such machines. We motivate the importance of existence of k-tape DTMs with the following example, which computes on 2 tapes.

Example 2.9. *(Palindrome checker) Consider the following decision problem:*

Palindrome
INSTANCE: *A string $x \in \{0, 1\}^*$.*
PROBLEM: *Is x a palindrome?*

Here we say x is a palindrome if it reads the same forwards as it does backwards. So 100111001 *is a palindrome, but* 10010 *is not.*

*We use a 2-tape DTM to compute this. The first tape is an **input tape** which takes in an input, and the second is a **work tape** which does the majority of calculations. A sketch of a transition function is given below based on the function given in (Arora and Barak 2009):*

1. *Input x on the input tape, and set the work tape to only consist of blank symbols. Initialise the tape at the first symbol of our input.*
2. *Copy the number from the first tape onto the second tape.*
3. *Move the tape head to the very left on the input tape, and to the very right of the work tape.*
4. *Begin comparing the symbols in both tapes. If they are not the same, reject the input. Otherwise, move the input head to the right and the work head to the left and repeat as before. Continue until state (b, b) is reached, at which point the input must have been read the same way forwards as backwards, so accept the input.*

We end this section on DTMs with an important lemma:

Lemma 2.10. *(Sipser 2013) A k-tape DTM can compute a language L if and only if a one-tape DTM can also compute L.*

2.3 The Church-Turing thesis

Lemma 2.10 shows that any one-tape DTM and any k-tape DTM can compute the same set of languages. A natural question to ask is if this is true for all computers. This was hypothesised to be true by Turing and Church:

The Church-Turing thesis. *(Turing 1937) A language can be computed by some Turing machine if and only if it can be computed by some machine of any other 'mechanical' model of computation.*

This thesis is left deliberately unclear; instead of rigour Church and Turing are aiming to capture a sense of what it means for something to be a computer. We use the term **Turing complete** to refer to something that can complete actions that a traditional tape can as well. In particular, this now means we do not need to always resort to going to a fundamental computation at tape level, so long as we ensure that we maintain a sufficient level of rigour in our calculations.

2.4 Nondeterministic Turing Machines

We now turn to nondeterministic machines. A nondeterministic Turing Machine has a similar structure to that of a DTM with one key difference; instead of a transition function which uniquely determines the next state, position and adjusts the read symbol, there is a **transition relation**, which gives a choice of state, new tape symbol, and change of direction to go to.

Definition 2.11. *A program in a **nondeterministic Turing Machine** (NDTM) is a tuple* (Γ, Q, δ) *such that:*

- Γ *is a finite set of tape symbols, including a subset* $\Sigma \subset \Gamma$ *of input symbols and a unique blank symbol* $b \in \Gamma \setminus \Sigma$*;*
- Q *is a finite set of states, including a unique start state* q_0 *and two unique halt states* q_Y *and* q_N*;*
- δ *represents a **transition relation**, where, given an input state* q *and tape symbol* X*,* $\delta(q, X)$ *is the set of tuples*

$$\{(q_{k_1}, Y_1, D_1), (q_{k_2}, y_2, D_2), \ldots, (q_{k_n}, y_n, D_n)\}$$

where q_{k_i} *represents a new state,* Y_i *represents a number to rewrite on the current tape position, and* D_i *represents a direction (given by* $-1, 0$ *or* $+1$*) to shift the tape head.*

The NDTM makes a random choice of transition function. We refer to a sequence of these choices (that begins with the first transition in the computation) as a **branch** of the computation. We now provide a formal definition of accepting and rejecting in a nondeterministic TM.

Definition 2.12. *Let* Σ *be the alphabet for a program* M *in a nondeterministic Turing Machine, and consider an instance* $x \in \Sigma^*$*. We say that* M ***halts, computes*** x *or **decides*** x *if there exists* $T \in \mathbb{N}$ *such that for all* $t \geq T$*, the state, location and contents of all cells of* M *at step* t *is equal to that at step* $t + 1$ *in all branches of the computation. Furthermore, we say that:*

- M ***accepts*** x *if there exists some branch of* M *that halts on* q_Y*;*
- M ***rejects*** x *if all branches of* M *that halt on* q_N*;*
- *The language* L_M ***recognized*** *by* M *is given by the set*

$$L_M = \{x \in \Sigma^* : M \text{ accepts } x.\}$$

Definition 2.13. *We say that an NDTM program* M *is a **decider** if* M *halts for any instance* x *in* M*.*

We note that the key distinction between a nondeterministic and deterministic Turing Machine is the presence of a choice of transition. This yields an important question of whether or not NDTMs are inherently more powerful than DTMs. To show that this is not the case, we can visualise the set of all possible computations on an input x as a tree, as given in Figure 4.

An accepting computation on x is a branch of the NDTM with finite height. To find an accepting computation on x, we use a **breadth-first search**. This involves starting at the root node, checking its children, and then checking its grandchildren. This process continues until we find a point at which state q_Y is reached. The formal proof for this is given below.

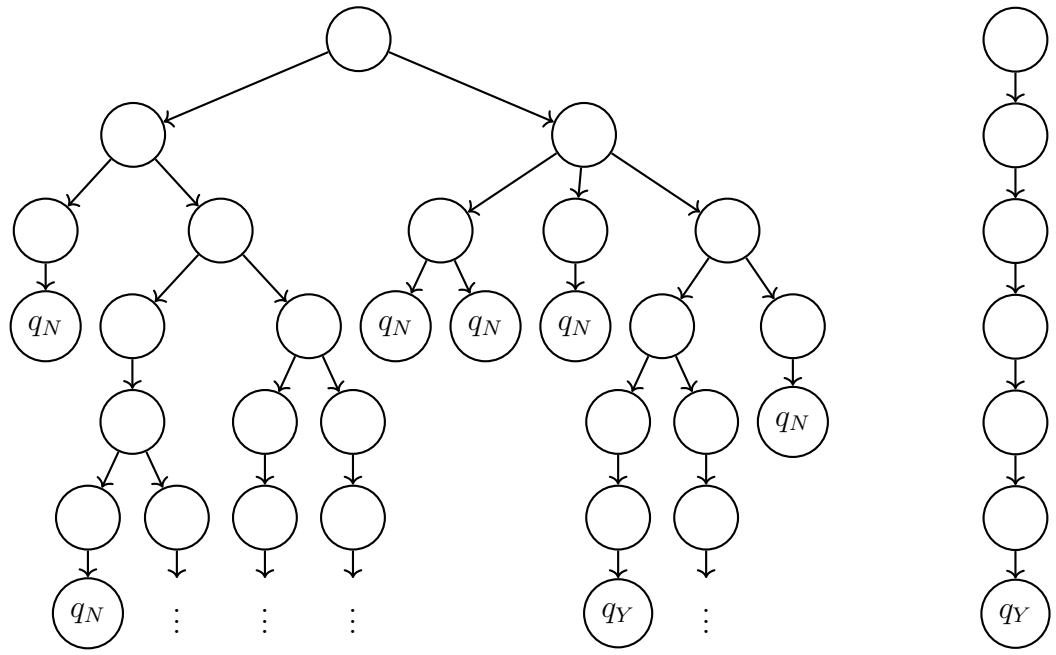

Figure 4: A graph of a nondeterministic computation alongside a deterministic computation. A branch of the computation that does not halt is indicated by ⋮. For all branches, only the halting state is shown.

Theorem 2.14. *Any language can be recognised by a program in a deterministic TM if and only if it can be recognised by a program in a nondeterministic TM.*

Proof. ($\implies$) We can express the transition function of a DTM as the transition relation of an NDTM which contains a single branch.

($\impliedby$) We will design a multitape DTM that computes this. Without loss of generality construct an arbitrary ordering in each set of tuples within the transition relation. In the DTM set the first tape to be an input tape, the second to be a tape of ID's, and the third to be a scratch tape. In the beginning, there will be 1 ID (the 'root' node of the tree of nondeterministic computation), and as always we begin at the first symbol of the input.

The transition function is given in the following way:

1. Examine the state and scanned symbol of the current ID. If it halts on state q_Y, we are done.

2. Otherwise, consider the transition relation given for the respective state and tape symbol, for which there will be k possible options. Append k IDs at the end of the ID list.

3. Bring the tape head back to the current ID, erase it and go to the next tape head.

The above acts as a breadth-first search through all possible values of the TM, so if the NDTM program accepts an input, so must the constructed DTM program. □

3 Time complexity

We have now formalised our concept of computation. Before we continue, let us motivate the rest of this report with a proposition:

Proposition 3.1. *For any $a, b > 1$, there exists an integer N such that for all $n > N$,*

$$a^n > n^b.$$

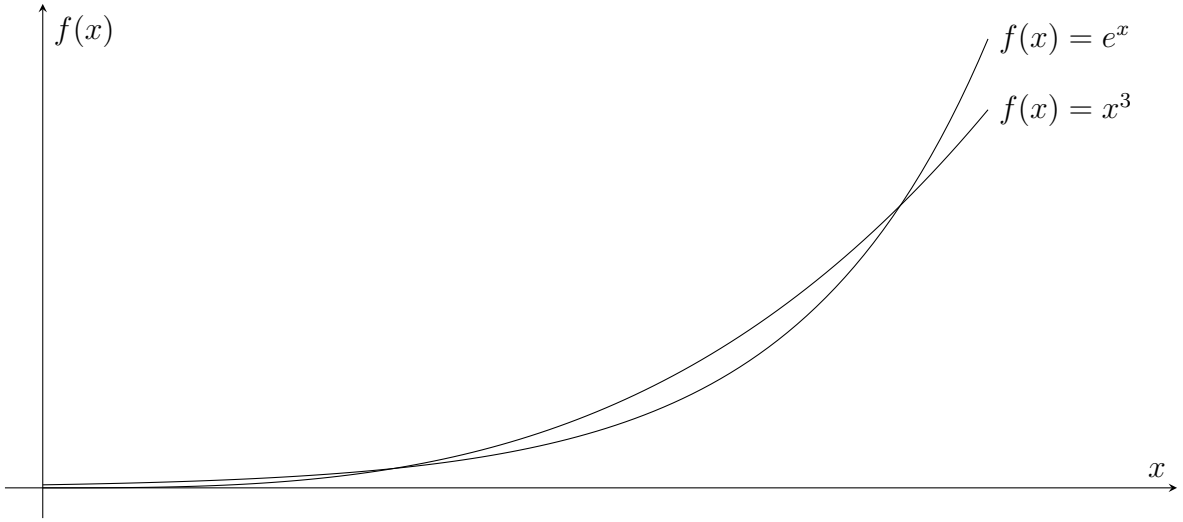

Figure 5: A graph of $f(x) = e^x$ against $f(x) = x^3$. Even though e^x is smaller than x^3 at first, it eventually becomes much larger than x^3.

It is important for the programs in Turing Machines we design to be as fast as possible. The assertion arising from Proposition 3.1 means that if a program in a DTM were to take a polynomial amount of time, it will eventually be much faster than a program that takes an exponential amount of time. The area of study that explores how fast a language can be computed is referred to as **Complexity Theory**, which we spend the rest of this report formalising.

Definition 3.2. *Let M be a program in a DTM, and let x be an instance that halts in M. Denote the **time used** to compute an instance x as $t_M(x)$, defined to be the number of steps M takes to compute x. If all instances $x \in \Sigma^*$ in M halt, the **time complexity** of M for input length n is defined by*

$$T_M(n) = \max\{t_M(x) : x \in \Sigma^* \text{ and } |x| = n\}.$$

Definition 3.3. (**DTIME**) *A language L is in* **DTIME**$(T(n))$ *if there exists a DTM M that runs with time $c \cdot T_M(n)$ for some constant $c > 0$ and computes all instances $x \in L$.*

Remark 3.4. *Remark 2.5 notes that a decision problem can be used to describe a language. We refer to a decision problem Π as being in a complexity class if the language it describes is in that complexity class.*

Intuitively, **DTIME** exists with the aim to classify a complexity class. We motivate this with some examples:

Example 3.5. *Here are some examples of time complexities and their respective* **DTIME** *classes. For all of these, L is a language, with all instances $x \in L$ having size $|n|$.*

- *If $T_M(n) = n^2 + \log n + 1$, $L \in$* **DTIME**(n^2).
- *If $T_M(n) = 2^n + n^2$, $L \in$* **DTIME**(2^n).
- **DTIME**$(n^i) \subseteq$ **DTIME**(n^{i+j}) *for all $i, j > 0$.*

Remark 3.6. *In Definition 3.3 it should be noted that there is no requirement that the most efficient computation acts in the time specified - only that there exists a DTM that acts in that time. We can therefore have any language in one* **DTIME** *class also be in a slower* **DTIME** *class by adding a large number of dummy steps to the end of a computation on an instance.*

It is important to note at this point the assertion that **DTIME** acts as a classifier for all DTMs independent of the number of tapes in the DTM. The following theorem shows that this must be the case.

Theorem 3.7. *(Papadimitriou 1994) (Linear speedup theorem) If the time complexity for a k-tape DTM that computes a language L is given by $T(n)$, then the time complexity for a one-tape DTM that also computes L is at most*

$$\frac{T(n)}{c} + 2n + 3$$

where c is a constant.

Now that we know all computations on DTMs have a well defined time complexity, we can begin to classify languages by generalised time complexity.

Definition 3.8. *(The class $\mathcal{P}$)* $\mathcal{P} = \bigcup_{c \geq 0}$ **DTIME**(n^c).

This stands for **Polynomial deterministic time** and includes the set of any inputs that take a polynomial amount of time to run on a program in a DTM. We say that a decision problem Π is **tractable** or **efficient** if all instances in Π are also in $\mathcal{P}$ and **intractable** if not.

Example 3.9. *Consider the following decision problem:*

Complete graph
INSTANCE: *A graph G*
PROBLEM: *Is G complete?*

Recall from Section 2.1.1 that we can encode a graph by its adjacency matrix. We design a program in a DTM that checks, given a graph $G = (V, E)$, if for any $i \neq j$, $a_{i,j} = 1$. This takes $|V|^2 - |V|$ steps (since there are that many entries to check), and so Complete graph is in $\mathcal{P}$.

3.1 The class $\mathcal{NP}$

We have thus far only looked at determinism. We now turn to nondeterministic TMs and look at polynomially bound computations on them.

Definition 3.10. *Let an NDTM program M with alphabet Γ be a decider, and let $x \in \Gamma^*$ be an instance. Denote the time used to compute an instance as $t_M(x)$, defined to be the maximum number of steps that M uses on any branch to compute x. The time complexity of M for input length n is defined by*

$$T_M(n) = \max\{t_M(x) : x \in \Gamma^* \text{ and } |x| = n\}.$$

Definition 3.11. *(The class* **NTIME***) A language L is in* $\mathbf{NTIME}(T(n))$ *if there exists an NDTM that runs with time $c \cdot T(n)$ for some constant $c > 0$ and computes L.*

Definition 3.12. *(The class $\mathcal{NP}$)* $\mathcal{NP} = \bigcup_{c \geq 0} \mathbf{NTIME}(T(n))$.

Recalling Theorem 2.14, if an NDTM recognises a language then so must a DTM; our method for this was by finding an arbitrary branch that accepts the instance. This is incredibly inefficient (Sipser 2013) and a language in $\mathcal{NP}$ cannot be recognised efficiently with a DTM through this method. Instead of recognising a language, if we instead require a program in a DTM to accept an instance in polynomial time, this is possible simply by following the branch that accepts an instance. The DTM that follows this branch is known as a **decider**.

Proposition 3.13. *A language L is in $\mathcal{NP}$ if and only if for some input $x \in L$ (with $|x| = n$) there exists some polynomial p and some DTM M that accepts x with $t_M(x) < p(n)$.*

Proof. ($\implies$) Consider some language $l \in \mathcal{NP}$. A verifier for $x \in l$ can be given by following the branch of the computation that accepts it; this takes a polynomial number of steps and we are done.

($\impliedby$) Consider a language l such that for all $x \in l$, x can be computed in polynomial time by some program in a DTM. Label these inputs and programs x_i and M_i respectively. Set a transition relation for an NDTM to encompass these transition functions, which accepts all $x \in l$ in a polynomial amount of time, and hence $l \in \mathcal{NP}$. □

What is left for us is to consider what languages are in $\mathcal{P}$ and what languages are in $\mathcal{NP}$. There is one such relationship that we know of:

Corollary 3.14. $\mathcal{P} \subseteq \mathcal{NP}$.

Proof. If a language $L \in \mathcal{P}$, then it must have a polynomial time DTM that computes all inputs in L. Since this runs in polynomial time, we implement it as our verifier which will correctly show the same result. □

4 NP-completeness

The consensus among computer scientists is that $\mathcal{P} \neq \mathcal{NP}$ (Arora and Barak 2009). This seems to imply that there exist a set of languages that are fundamentally harder to compute than others. These languages are known as **NP-complete languages** and have a property that if any of them are in $\mathcal{P}$, then so is every other language in $\mathcal{NP}$.

4.1 Polynomial transformations

We will construct the set of NP-complete languages through the use of **polynomial transformations**.

Definition 4.1. *A* ***polynomial transformation*** *from a language $L_1 \subseteq \Sigma_1^*$ to a language $L_2 \subseteq \Sigma_2^*$ is a function $f : \Sigma_1^* \to \Sigma_2^*$ that satisfies these conditions:*

1. *There is a polynomial time DTM program that computes f;*
2. *For all $x \in \Sigma_1^*$, we have $x \in L_1$ if and only if $f(x) \in L_2$.*

We write $L_1 \propto L_2$ to indicate that there exists a polynomial transformation from L_1 to L_2.

Remark 4.2. *We say that for two decision problems Π_1, Π_2, $\Pi_1 \propto \Pi_2$ if there exists a polynomial transformation from Π_1 to Π_2.*

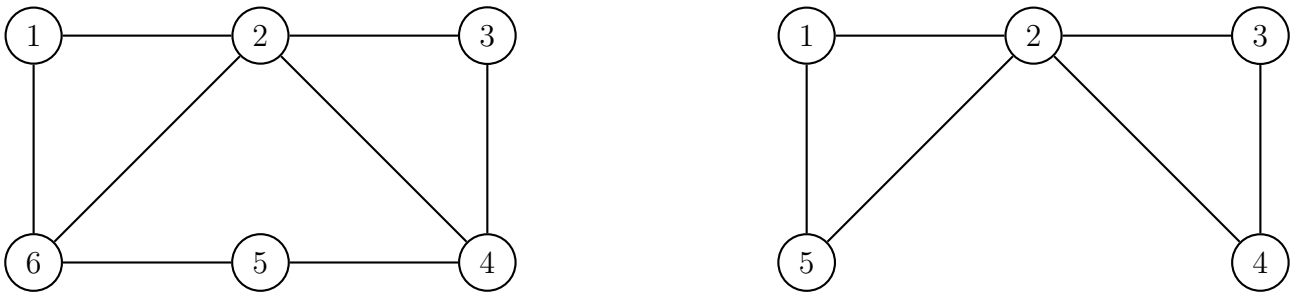

Figure 6: Two graphs. The first contains a Hamiltonian circuit (via the path $(1, 2, 3, 4, 5, 6, 1)$) while the second does not.

Definition 4.3. *Two languages L_1 and L_2 are **polynomally equivalent** if both $L_1 \propto L_2$ and $L_2 \propto L_1$.*

Let us demonstrate this with an example. Here are two decision problems based on graphs (one unweighted, one weighted):

Hamiltonian Circuit (HC)
INSTANCE: A graph $G = (V, E)$.
PROBLEM: Does there exist a Hamiltonian Circuit in G? (A Hamiltonian Circuit is a path in G that visits every node exactly once before returning to the original node - see Figure 6 for an example)

Traveling Salesman (TS)
INSTANCE: A tuple (V, E, m) where (V, E) describes a weighted graph G.
PROBLEM: Does there exist a cycle of G shorter than length m?

Example 4.4. *We will show that $HC \propto TS$.*

Let $G = (V, E)$ be the graph for consideration in HC.

We define f to map G to the weighted graph $f(G) = (V, E')$ where the weightings of E' are given by the formula

$$\text{weight on edge } (v_i, v_j) \;= \begin{cases} 1 & \text{if } (v_i, v_j) \in E \\ 2 & \text{if } (v_i, v_j) \notin E \end{cases}$$

We set the requirement of cycle length to be at maximum $|V|$.

Since a complete graph with n vertices has $\frac{n(n-1)}{2}$ edges, it takes this number of comparisons to determine the weight of each edge in G'. This takes a polynomial amount of time, and so the first requirement is satisfied.

We now need to satisfy the second requirement. Consider the path for a tour in G given by $(v_1, v_2, \ldots, v_m, v_1)$. This is also a tour in $f(G)$; moreover, since each edge in the tour is in E, the weights of each edge in the tour must be equal to 1. Therefore, the total length of the tour in $f(G)$ is given by $m = |V|$ and thus $f(G) \in TS$.

Now suppose that there is a tour in $f(G)$ given by $(v_1, v_2, \ldots, v_m, v_1)$ with total length $m = |V|$. By the construction of f, We know that each edge given by the tour must also be an edge in G. Hence this is also a valid tour in G, and so $G \in HC$.

4.2 Constructing the class of NP-complete languages

We now turn to some important lemmas regarding polynomial transformations, which we use to define the class of NP-complete languages.

Lemma 4.5. *If $L_1 \propto L_2$, then $L_2 \in \mathcal{P}$ implies $L_1 \in \mathcal{P}$ (and equivalently, $L_1 \notin \mathcal{P}$ implies $L_2 \notin \mathcal{P}$).*

Proof. Let Σ_1^* and Σ_2^* be the alphabets of L_1 and L_2 respectively. Let $f : \Sigma_1^* \to \Sigma_2^*$ be a polynomial transformation. Finally, let M_f denote a DTM program that recognises f and M_2 be a polynomial time program that recognises L_2.

We need to construct a DTM program that recognises L_1, which can be given by the composition $M_2 \circ M_f$. Since $x \in L_1 \iff f(x) \in L_2$, this program recognises L_1.

Now, M_f and M_2 are both polynomial DTMs (by the definition of a polynomial transformation and L_2 being contained in $\mathcal{P}$ respectively), so their composition must also be a polynomial DTM. We have therefore found a polynomial DTM that recognises L_1, so $L_1 \in \mathcal{P}$. □

Lemma 4.6. *(Garey and Johnson 1979) If $L_1 \propto L_2$ and $L_2 \propto L_3$, then $L_1 \propto L_3$.*

Definition 4.7. *A language L is **NP-complete** if $L \in \mathcal{NP}$ and for all other languages $L' \in \mathcal{NP}$, $L' \propto L$.*

Remark 4.8. *We say that a decision problem Π is **NP-complete** if $\Pi \in \mathcal{NP}$ and for all other decision problems Π', $\Pi' \propto \Pi$.*

We now state the most crucial theorem of this chapter.

Theorem 4.9. *If L is NP-complete and $L \in \mathcal{P}$, then $\mathcal{P} = \mathcal{NP}$.*

Proof. Since $\mathcal{P} \subseteq \mathcal{NP}$, it is sufficient to show that if the conditions of the theorem are satisfied then $\mathcal{NP} \subseteq \mathcal{P}$. Consider some arbitrary language $L' \in \mathcal{NP}$. Since L is NP-complete, $L' \propto L$. By Lemma 4.5, since $L \in \mathcal{P}$, $L' \in \mathcal{P}$ and $\mathcal{NP} \subseteq \mathcal{P}$ as required. □

5 Cook's Theorem

Let us now go about finding NP-complete problems. The first problem shown to be NP-complete is a problem in Boolean algebra known as the satisfiability problem, proven by Stephen Cook in 1971. Before we explore the proof, we give some terminology.

5.1 Boolean algebra

Definition 5.1. *A variable u is a **Boolean variable** if $u \in \{T, F\}$*[1]*. The **complement** of u, denoted $\overline{u}$, is defined by*

$$\overline{u} = \begin{cases} F \text{ if } u = T \\ T \text{ if } u = F \end{cases}$$

Definition 5.2. *Let $U = \{u_1, u_2, \ldots, u_m\}$ be a set of boolean variables. Then,*

- *A **clause** over U is a set created by choosing at most one element from the sets $\{u_1, \overline{u_1}\}, \{u_2, \overline{u_2}\}, \ldots \{u_m, \overline{u_m}\}$,*
- *A **truth assignment** is a function $t : U \to \{T, F\}$,*

[1] Here T = True, F = False

- *A clause is **satisfied** by a truth assignment if, under the truth assignment given, at least one member of the clause is True. A collection C of clauses over U (in Conjunctive Normal Form) is **satisfiable** if there exists some truth assignment such that each element of C is satisfied under that truth assignment.*

Example 5.3. *Consider the set of boolean variables $\{u_1, u_2\}$. The clause $\{u_1, \overline{u_2}\}$ can be satisfied by any truth assignment such that either of $t(u_1) = T$, $t(u_2) = F$ hold.*

Now consider the set of clauses $\{\{u_1, \overline{u_2}\}, \{\overline{u_1}, u_2\}\}$. This is satisfiable and can be satisfied by $t(u_1) = t(u_2) = T$. On the other hand, the collection $\{\{u_1, \overline{u_2}\}, \{u_1, u_2\}, \{\overline{u_2}\}\}$ cannot be satisfied by any truth assignment and so is unsatisfiable.

5.2 The satisfiability problem

We now turn to the satisfiablity decision problem, and Cook's seminal proof.

Satisfiability (Sat)
INSTANCE: A set U of boolean variables and a collection C of clauses over U
PROBLEM: Is there a satisfying truth assignment for C?

The theorem is now given.

Theorem 5.4. *(Cook 1971a) Satisfiability is NP-complete.*

Proof. We give a sketch proof based on (Garey and Johnson 1979). First we acknowledge that Sat $\in \mathcal{NP}$, as given a truth assignment that it satisfies C in polynomial time. Thus our first requirement is met.

For the second requirement, we consider the set of languages L_{Sat} which are represented by Sat. We need to show that for all languages $L \in \mathcal{NP}$, $L \propto L_{\text{Sat}}$. As there are an infinite number of languages in $\mathcal{NP}$, we do not create a transition function between the languages. Instead we inspect a polynomial time program in a NDTM that recognises them, as these will have certain distinguishing features. This gives a proof that acts simultaneously for all languages $L \in \mathcal{NP}$.

Consider a polynomial time NDTM program $M = (\Gamma, Q, \delta)$ with alphabet Σ which recognises an arbitrary language L in polynomial time. That is, for any computation of $x \in L$ with $|x| = n$, there exists some polynomial p such that $T_M(n) < p(n)$. Denote the polynomial transformation we construct by f_L, and instead of creating a polynomial transformation from L to L_{Sat} we create a mapping from Γ to instances of Sat. This has the property that for all instances $x \in \Sigma^*$, $x \in L$ if and only if $f_L(x)$ has a satisfying truth assignment.

If an instance $x \in \Sigma^*$ is accepted by M, then there must exist an accepting computation such that the number of steps in the checking stage are bounded by $p(n)$. In particular, we must only utilise the tape squares $-p(n)$ to $(p(n)$. Hence, we can describe the NDTM at time $0 \leq t \leq p(n)$ by:

- The contents of the tape squares;
- The state of M;
- The position of the read-write head.

Since there are a finite number of these, we can assign them to be boolean variables, for which we call the variable set U. Label $Q = \{q_0, q_1 = q_Y, q_2 = q_N, q_3, \cdots, q_r\}$ where $r = |Q| - 1$, and label $\Gamma = \{s_0 = b, s_1, \cdots, s_v\}$ where $v = |\Gamma| - 1$. Below is a table of the description of the variables.

Variable	Range	Meaning
$Q[i,k]$	$0 \leq t \leq p(n)$ $0 \leq k \leq r$	At time t, M is in stage q_k.
$H[i,j]$	$0 \leq t \leq p(n)$ -p(n) $\leq$j $\leq$p(n)+1	At time t, the read-write head is scanning tape square j.
$S[i,j,k]$	$0 \leq t \leq p(n)$ $-p(n) \leq j \leq p(n)+1$ $0 \leq k \leq v$	At time t, the contents of tape square j is symbol s_k.

Table 2: A table of the variables and their meanings.

When M computes x, it induces a truth assignment on the variables. The construction of f_L is such that an accepting computation on x in polynomial time will return a truth assignment that satisfies U. Given this construction, we then have

$$\begin{aligned} x \in L &\iff \text{there is an accepting computation of } M \text{ on } x \\ &\iff \text{there is an accepting computation of } M \text{ on } x \text{ that takes } p(n) \text{ or fewer steps} \\ &\iff \text{there is a satisfying truth assignment for the collection of clauses in } f_L(x). \end{aligned}$$

This means that f_L satisfies one of the two requirements required of a polynomial transformation. We now complete our description of f_L.

Clause group	Restriction imposed
G_1	At time t, M is in exactly 1 state
G_2	At time t, the read-write head is scanning exactly one tape square
G_3	At time t, each tape square contains exactly one symbol from Γ
G_4	At time 0, the computation is in the initial configuration of its checking stage for input x
G_5	By time $p(n)$, M has entered state q_Y and hence has accepted x
G_6	For each time $0 \leq t \leq p(n)$, the configuration of M at time $t+1$ follows a single application of the transition function of M at time t.

Table 3: A table of clause groups and their meanings.

The table of clause groups clearly specifies a truth assignment such that if $x \in L$, then this truth assignment is satisfied. However, if $x \notin L$, then either it must be rejecting or does not halt (in which case clauses G_5 is violated) which does not satisfy the truth assignment. Hence $x \in L$ if and only if there is some satisfying truth assignment for the clause group given above.

What is now left to show is the for a fixed language L, $f_L(x)$ can be constructed in time bounded by some polynomial function of $n = |x|$. But the choice of our clauses was arbitrary, so we can adjust the clause group for our satisfying computation.

Our final task is to show that the time taken to encode SAT will be bounded above by some polynomial. This is easily done; the collection of clauses is polynomial on n, as is the number of possible sets of clauses, so the size of encoding is also polynomial on n as desired.

Hence, for any language $L \in \mathcal{NP}$, f_L acts as a polynomial transformation from L to L_{SAT}. It follows that SAT is NP-complete. □

6 Conclusion and further developments

In this report we have introduced and formalised the Turing Machine and demonstrated its computational effectiveness compared to other mechanical methods of operation. Through this, the Turing Machine has been shown to compute languages with different complexity classes, of which we have layed out a foundation to prove whether or not a language is in $\mathcal{P}$ or $\mathcal{NP}$.

Following the proof of Cook's Theorem in 1971, proving the existence of NP-complete problems became far easier, as if one can prove there exists a polynomial transformation from SAT to some decision problem Π, then Π is NP-complete by the transitivity of polynomial transformations. Richard Karp implemented this technique to show that there exist far more problems that are NP-complete (Karp 1972). These include the vertex cover problem (determining the minimum number of vertices that are at the endpoint of every edge) and the knapsack problem (determining, given a set of items with weights and value, the maximum value possible while keeping the total weight under a certain amount).

While a large focus of complexity theory is on finding methods to show that $\mathcal{P} = \mathcal{NP}$, there have also been methods discovered that could be used towards a proof of inequality. The only known technique that can be used for this is a technique known as **diagonalisation** (Arora and Barak 2009). While a proof or disproof of the famed conjecture is yet to be found, this technique was used to prove that if $\mathcal{P} \neq \mathcal{NP}$, there exist languages that are neither in $\mathcal{P}$ nor NP-complete (Ladner 1975).

References

Arora, Sanjeev and Boaz Barak (2009). *Computational complexity : a modern approach*. eng. ISBN: 9780511804090.

Cook, Stephen A. (1971a). "Characterizations of Pushdown Machines in Terms of Time-Bounded Computers". eng. In: *Journal of the ACM* 18.1, pp. 4–18. ISSN: 0004-5411.

— (1971b). "The complexity of theorem-proving procedures". eng. In: *Proceedings of the Annual ACM Symposium on Theory of Computing*. New York, NY, USA: ACM, pp. 151–158. ISBN: 9781450374644.

Garey, Michael R and David S Johnson (1979). *Computers and intractability : a guide to the theory of NP-completeness*. eng. A series of books in the mathematical sciences. New York: W. H. Freeman. ISBN: 0716710447.

Hopcroft, John E., Rajeev Motwani, and Jeffrey D. Ullman (2014). *Introduction to automata theory, languages, and computation*. eng. Third, Pearson new international edition. Always learning. ISBN: 9781292056166.

Jaffe, Arthur (2005). "The Millenium Grand Challenge in Mathematics". In: *Notices of the American Mathematical Society*.

Karp, Richard M. (1972). "Reducibility Among Combinatorial Problems". In: *Proceedings of a Symposium on the Complexity of Computer Computations*.

Ladner, Richard E. (1975). "On the Structure of Polynomial Time Reducibility". eng. In: *Journal of the ACM* 22.1, pp. 155–171. ISSN: 0004-5411.

Papadimitriou, Christos H (1994). *Computational complexity*. eng. Reading, Mass.: Addison-Wesley. ISBN: 0201530821.

Sipser, Michael (2013). *Introduction to the theory of computation*. eng. Third edition. ISBN: 9781473775558.

Turing, A. M. (1937). "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* s2-42.1, pp. 230–265. DOI: https://doi.org/10.1112/plms/s2-42.1.230.