NP-Completeness

Jessica Guo and Audrey Wei

May 22, 2022

Abstract

We study the fundamentals of time complexity in computational complexity theory. We discuss polynomial time and nondeterministic polynomial time. We consider NP-complete problems, the hardest problems in NP, which hold importance in the work on the P vs. NP problem.

1 Introduction

In a world where technology is growing by leaps and bounds, the theory of computation is increasingly critical to our understanding of the limits of computers. This paper serves as an introduction to computational complexity theory, specifically within the realms of time complexity. This paper will focus on NP-completeness, which holds both theoretical and practical significance—NP-completeness plays a role in the study of the infamous P versus NP problem, which essentially asks if any problem whose solution can be verified in polynomial time can also be decided in polynomial time, as well as in the search for algorithms that can solve real-life problems in a reasonable amount of time.

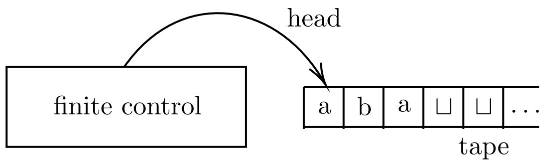
First, we will introduce the topic by discussing time complexity, especially in regards to Turing machines. Next, we will define polynomial time, giving simple examples along the way, before moving on to introducing non-deterministic polynomial time. Finally, we will discuss the heart of the paper, NP-completeness, and close with a proof of the Cook-Levin Theorem.

2 Time Complexity

Even when a problem can theoretically be solved, considerations of the time and space required to solve a problem can determine if it is solvable in practice. Time complexity serves to measure the time it takes to solve a computational problem.

We can measure time complexity by analyzing Turing machines, models of computation. The *Church-Turing thesis* tells us that Turing machines are equivalent to our intuitive notion of algorithms. We will first define a Turing machine and its important components.

Definition 2.1. A *Turing machine* (TM) is an idealized computer used to simulate an algorithm. A TM has a tape, a tape head, and a finite control as shown below.



The *tape*, which is infinite and has unlimited memory, initially contains the input string which can later be modified. The tape can also contain the blank symbol ⊔ to represent an empty cell.

The *head* is used to read and write symbols and is able to move left and right.

The *finite control* tells the tape head where to go, performs calculations using a finite set of states, and outputs "accept" or "reject" when the designated accepting and rejecting states are entered.

Definition 2.2. The *configuration* of a TM is the setting of the current state, head location, and contents of the tape.

Definition 2.3. A TM is *deterministic* when the current configuration determines the next configuration.

Definition 2.4. A *nondeterministic* TM (NTM) has the additional ability of "guessing," such that there are multiple possibilities for the next configuration on different "branches" of the machine's computation. A NTM accepts if and only if at least one of its branches accepts.

Definition 2.5. A TM *recognizes* a language if the TM accepts when the input string is in the language and rejects or loops otherwise.

When a machine loops, it can be confused with a machine that is just taking a long time to accept, as it never halts - it never enters a q_{accept} or q_{reject} state. For efficiency purposes, TMs that halt on all inputs are preferable as they will never loop.

Definition 2.6. A TM *decides* a language if it recognizes the language and either accepts or rejects (never loops) on each input.

Example 2.1. We describe a TM M that recognizes $A = \{0^k 1^k | k \geq 0\}$ as follows:

M = "On input string w:

1. Scan across the tape and reject if a 0 is found to the right of a 1.
2. Repeat step 3 if both 0s and 1s remain on the tape.
    3. Scan across the tape, crossing off a single 0 and a single 1.

4. If 0s still remain after all the 1s have been crossed off or if 1s still remain after all of the 0s are crossed off, reject. Otherwise, if no 0s or 1s remain on the tape, accept."

Step 1 checks whether or not the input string is in the format where the 0s are in front of 1s. If a 0 is between two 1s or if a 1 is between two 0s, this obviously does not satisfy the format of language A, so M rejects.

Looking at language A, we can also observe that the number of 0s is equal to the number of 1s. Repeating Step 3 allows us to see whether or not there is an equal number of 0s and 1s. Step 4 checks this new modified tape. When there is not an equal number of 0s and 1s, then the resulting tape will either have some remaining 0s or some remaining 1s and the TM will reject. When there is an equal number of 0s and 1s, the resulting tape will have all of its 0s and 1s crossed out and the TM will accept.

Definition 2.7. Suppose that a halting deterministic TM uses at most $f(n)$ steps on any input of length n. We call $f(n)$ the *running time* or *time complexity* of our TM.

Since the exact computation of running time is complex, time complexity is often estimated using the order of the expression for running time. One way to express this estimation is through big-O notation, which estimates the time complexity expression by taking the highest order term and ignoring its coefficient.

Definition 2.8. Let f and g be functions that take in natural numbers and output nonegative real numbers. Then $f(n)$ is $O(g(n))$ if for positive constants c and n_0 where integer $n \geq n_0$,

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

Example 2.2. Let $f_1(n) = 8n^2 + 3n + 9$ be the time complexity of M_2. The highest order term is $8n^2$, and we ignore its coefficient of 8, so $f_1(n) = O(n^2)$.

Definition 2.9. Many machines have time complexities in the form of $O(n^c)$ where c is greater than or equal to 0. This time complexity is referred to as *polynomial.*

Definition 2.10. Many other machines have time complexities in the form of $2^{O(n)}$. This time complexity is referred to as *exponential.*

Example 2.3. Let $f_2(n) = 2^n$ be the time complexity of M_3. Since this is an exponential function, the time complexity will be exponential, not polynomial.

Example 2.4. For TM M that recognizes the language $A = \{0^k1^k | k \geq 0\}$, we can find its running time by analyzing the time for each step of M.

We call n the length of our input. On step 1, M scans the whole tape which uses n steps, then the head moves back to the left end, which takes another n steps, taking a total of $2n$ or $O(n)$ steps. On step 2 and 3, M will use $O(n^2)$

as it will scan the tape (taking n steps) a maximum of $n/2$ times (crossing off 2 symbols each time). On step 4, M performs another scan, which takes $O(n)$. Therefore, the total running time is $O(n) + O(n^2) + O(n) = O(n^2)$.

Definition 2.11. Let $t(n)$ be a function which takes the natural numbers and outputs nonnegative real numbers. The *time complexity class* $TIME(t(n))$ is the collection of languages that are decidable by an $O(t(n))$ TM.

Example 2.5. $A = \{0^k 1^k | k \geq 0\}$ is in $TIME(n^2)$ since M decides A in $O(n^2)$.

3 Polynomial Time

In the section above, we learned about time complexities that are polynomial and exponential. Polynomial time (P) computer algorithms are considered faster than exponential time algorithms.

Definition 3.1. P is the class of languages that are decidable in polynomial time on a deterministic single-tape TM. The mathematical notation of P is

$$P = \bigcup_k TIME(n^k).$$

Polynomial time such as n^2 grows significantly slower than exponential time such as 2^n. When $n = 10000$, n^2 will be equal to 100 million, a large but manageable number, while 2^n will be a number much greater than the total number of atoms in the universe.

Broadly speaking, because polynomial time algorithms are more reasonable to compute than exponential time algorithms, P algorithms are more useful computational models for practical use. But only using polynomial time algorithms will not provide a solution to every problem. This will be further explored in the following section.

An example of a language that is in P is $PATH$. The $PATH$ problem determines whether or not a directed graph G that contains two nodes s and t has a directed path that connects s to t.

$PATH = \{\langle G, s, t\rangle \,|\,$ G is a directed graph that has a directed path from s to t$\}$.

Note that $\langle G, s, t\rangle$ denotes an encoding of the information G, s, t into some string.

Theorem 3.1. *$PATH \in P$.*

Proof. Let's first write the algorithm M for $PATH$.

M = "On input $\langle G, s, t\rangle$ where G is a directed graph with nodes s and t

1. Place a mark on node s.
2. Repeat the next step until no additional nodes are marked:

3. Scan all of the edges of G. If there is an edge (x, y) where x is a marked node and y is an unmarked node, mark node y.

4. If t is marked, accept. Otherwise reject."

We can check that this algorithm works correctly because if there is a path from s to t, M is guaranteed to accept because when following the steps, node t will always be marked at the end. If there is no path from s to t, M is guaranteed to reject because t will never be marked.

Analyzing this algorithm, we can also show that $PATH$ runs in polynomial time. The first and fourth steps are executed once. When the maximum length of a potential path in G is set to n, then step 3 is executed at most n times. Thus, adding the steps together, we get $1 + 1 + n$ which means M runs in polynomial time. Therefore, M is a polynomial time algorithm for $PATH$. $\square$

4 Non-Deterministic Polynomial Time

As mentioned in the previous section, polynomial time algorithms are able to provide solutions to problems in a reasonable time complexity. However, for many useful problems, we have not been able to find a polynomial time algorithm. To solve these problems, we introduce the time complexity class non-deterministic polynomial time (NP). Interestingly, researchers have been unable to prove the existence of any NP languages that are in P. This question of whether P equals NP still is one of the most well-known unsolved questions in mathematics and computer science.

Some problems where a polynomial time solution is unknown have a feature called polynomial verifiability that is important for understanding their complexity. We do not know of a fast/polynomial time solution, but, if given a proposed possible solution, we can quickly verify that solution.

Definition 4.1. A verifier for a language A is an algorithm N, where

$$A = \{w | N \text{ accepts } \langle w, c \rangle \text{ for some string } c\}.$$

We measure the time of a verifier in terms of length w. This means that a polynomial time verifier runs in polynomial time. A language A is a *polynomially verifiable* if it has a polynomial time verifier. Here, c represents proof or a certificate of membership in A.

Another way to view a verifier is in terms of a NTM. A NTM's ability to guess can be thought of as the ability to generate a candidate certificate of membership.

We can define the nondeterministic time complexity class NTIME($t(n)$) based on the deterministic time complexity class TIME($t(n)$).

Definition 4.2. NTIME($t(n)$) = $\{L | L$ is a language decided by an $O(t(n))$ time nondeterministic TM$\}$.

Definition 4.3. *NP* is the class of languages that have polynomial time verifiers. The mathematical notation of NP is

$$NP = \bigcup_k NTIME(n^k).$$

We now prove that the NTM definition of NP is equivalent to the earlier definition of polynomial verifiablity.

Theorem 4.1. A language is in NP if and only if it is polynomially verifiable.

Proof. We prove this theorem in two parts. First, we have to convert a polynomial time verifier to an equivalent polynomial time NTM. Then we have to convert a polynomial time NTM to an equivalent polynomial time verifier. The NTM simulates the verifier by guessing the certificate while the verifier simulates the NTM by using the accepting branch as the certificate.

For the first step, let's assume that there exists V, a TM that runs in time n^k and verifies A. We can construct a polynomial time NTM N that decides A as follows.

$N =$ "On input w of length n:

1. Nondeterministically select string c of length at most n^k.
2. Run V on input $\langle w, c \rangle$.
3. If V accepts, accept. Otherwise reject.

To prove the other direction of the theorem, we assume that A is decided by a polynomial time NTM N. We can construct a polynomial time verifier V as follows.

$V =$ "On input $\langle w, c \rangle$ where w and c are strings:

1. Simulate N on input w, treating each symbol of c as a description of the nondeterministic choice to make at each step.
2. If this branch of N's computation accepts, accept. Otherwise reject.

□

Using Theorem 4.1, we can approach proving problems to be in NP in two different ways. The first way is finding the certificate of the problem. The second way is finding the nondeterministic polynomial time steps in the Turing machine.

An example of a NP problem is $HAMPATH$. The $HAMPATH$ problem determines whether in a directed graph G, there is a directed path from node s to node t that goes through each node exactly once.

$HAMPATH = \{\langle G, s, t \rangle \mid \text{G is a directed graph with a Hamiltonian path from s to t}\}$.

Example 4.1. $HAMPATH \in NP$.

We can write a NTM M that decides $HAMPATH$ as follows:

M = "On input $\langle G, s, t\rangle$ where G is a directed graph with nodes s and t:

1. Nondeterministically guess a list of n nodes, $a_1, ..., a_n$ where n is the number of nodes in G.
2. Check for repetitions in the list. If any repetitions are found, reject.
3. Check whether $s = a_1$ and $t = a_n$. If either fail, reject.
4. For each i between 1 and $n - 1$, check whether (a_i, a_{i+1}) is an edge of G. If any are not, reject. Otherwise, accept.

We can see that when there is a Hamiltonian path from s to t, then M is guaranteed to accept because no nodes are gone through more than once. If there is not a Hamiltonian path from s to t, then M is guaranteed to reject because there will be overlapping nodes.

Let's also check that this algorithm runs in nondeterministic polynomial time. Step 1 is a nondeterministic selection which obviously runs in polynomial time. Step 2 and 3 are both quick checks which that can be run in polynomial time. Step 4 also clearly runs in polynomial time since only $n - 1$ checks are performed. Therefore this algorithm runs in NP time.

5 NP-Completeness

NP-complete problems are problems in NP whose complexity is linked to the complexity of all problems in NP. So, finding any algorithm in P that solves an NP-complete problem would mean that all problems in NP can be solved in P, proving $P = NP$.

Definition 5.1. Language A is *polynomial time reducible* to language L , denoted $A \leq_p L$, if there exists a polynomial time computable function f such that for any input w,

$$w \in A \Leftrightarrow f(w) \in L.$$

Reducibility allows us to determine whether or not a language is in P by checking if this language is polynomial time reducible to a language already known to be in P. The theorem below shows this idea.

Theorem 5.1. If $A \leq_p B$ and $B \in$ P, then $A \in$ P.

Proof. Let M be the polynomial time algorithm that decides B and let f be the polynomial time reduction from A to B. We construct a polynomial time algorithm N deciding A:

N = "On input w:

1. Compute $f(w)$
2. Run M on input $f(w)$ and output whatever M outputs."

Since $A \leq_p B$ and f is a reduction from A and B, we have $w \in A$ whenever $f(w) \in B$. This also means that M accepts $f(w)$ whenever $w \in A$. In addition, N runs in polynomial time because the two steps both run in polynomial time. □

Definition 5.2. Language L is *NP-complete* if

1. $L \in$ NP
2. For every $A \in$ NP, $A \leq_p L$.

A problem is *NP-hard* when it satisfies the second condition of the definition. To set up our first NP-complete problem, we will start with a few definitions.

Definition 5.3. A *Boolean formula* ϕ is an expression of *Boolean variables*, True (1) or False (0) values, and *Boolean operations,* AND($\wedge$), OR($\vee$), and NOT($\neg$), which manipulate Boolean variables.

Definition 5.4. ϕ is *satisfiable* if ϕ is True for some assignment of 0s and 1s to its variables.

Example 5.1. Let $\phi = x \wedge (\overline{y} \vee z)$. The assignment $x = 1, y = 1, z = 1$ makes ϕ evaluate to 1, so ϕ is satisfiable.

Example 5.2. The *satisfiability problem* tests if a Boolean formula ϕ is satisfiable. We will abbreviate this problem to SAT.

$$SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable Boolean formula}\}.$$

To prove some language L is NP-complete, we show $L \in NP$ and $A \leq_p L$ where A is known to be NP-complete. By proving that SAT is NP-complete, we can use SAT as our language A to solve other NP-complete problems.

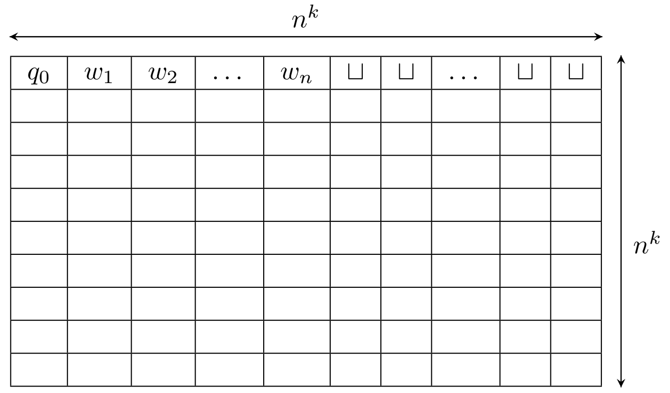
Theorem 5.2 (Cook-Levin Theorem)**.** SAT is NP-complete.

Proof. In order to prove that SAT is NP-complete, we have to prove that SAT satisfies 2 conditions:

1. $SAT \in$ NP
2. For every $A \in$ NP, $A \leq_p SAT$.

Since a nondeterministic TM can guess an assignment to ϕ and accept if it satisfies ϕ, SAT is in NP.
Next, we show that every language A in NP is polynomial time reducible to SAT. Let N be the NTM that decides A in n^k time. We need to construct ϕ that is satisfiable iff N accepts w.

We proceed by using a tableau, as shown in the figure below. A tableau for NTM N on input w lists the configurations of a branch of computation. Each row of our n^k by n^k tableau represents a configuration of N on w, where the first row is the first configuration, the second row is the second configuration, and so on, where the last row is the n^kth configuration. Our tableau accepts if any row is an accepting configuration. N accepting w is equivalent to our tableau accepting. We show that there is an accepting tableau for N on w iff ϕ is satisfiable.



Each cell of our tableau is called $cell[i, j]$, where i is the row and j is the column. We say that $x_{i,j,s}$ is True if and only if $cell[i, j]$ holds symbol s.
We know that $1 \leq i, j \leq n^k$ and s, t are in the set made up of N's tape alphabet and state set. Note that $\bigwedge$ and $\bigvee$ represent iterated AND and OR expressions respectively.

In order to make our tableau correspond to ϕ, we say that

$$\phi = \phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{move}} \wedge \phi_{\text{accept}}.$$

Since at least 1 variable s for each cell is True, and $x_{i,j,s}$ and $x_{i,j,t}$ cannot both be true, we say that

$$\phi_{\text{cell}} = \bigwedge_{i,j} \left[\left(\bigvee_{s} x_{i,j,s} \right) \wedge \left(\bigwedge_{\substack{s,t \\ s \neq t}} \left(\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}} \right) \right) \right].$$

Then, we express the locations and symbols for the start configuration in the first row:

$$\phi_{\text{start}} = x_{1,1,q_0} \wedge x_{1,2,w_1} \wedge x_{1,3,w_2} \wedge x_{1,4,w_3} \wedge \dots$$

We also require that the accept state q_{accept} is on at least 1 of the cells, so

$$\phi_{\text{accept}} = \bigvee_{i,j} x_{i,j,q_{\text{accept}}}.$$

To express ϕ_{move}, we first need need to ensure that configurations following past configurations are legal moves. A 2 by 3 *window* of cells is *legal* if it follows the transition functions of N. The (i,j)-window means the rows of the window are $i, i+1$ and the columns of the window are $j-1, j, j+1$. So, $cell[i,j]$ is in the upper middle position (holding b).

a	b	c
d	e	f

To construct ϕ_{move}, all windows in the tableau have to be legal, with each of the 6 cells yielding a legal window. So, we can say that

$$\phi_{\text{move}} = \bigwedge_{i,j} \Big[\bigvee_{\substack{a,b,c,d,e,f \\ \text{is a legal window}}} \big(x_{i,j-1,a} \wedge x_{i,j,b} \wedge x_{i,j+1,c} \wedge \cdots \wedge x_{i+1,j+1,f} \big) \Big].$$

Now we have our ϕ, we have to show that its size is polynomial. We note that our n^k by n^k tableau has n^{2k} cells, and each cell has some number of variables depending on the number of possible symbols. Since the number of symbols does not depend on the input n, the total number of variables is $O(n^{2k})$.

$\phi_{\text{cell}}, \phi_{\text{move}}, \phi_{\text{accept}}$ involve each cell of the tableau, so they each have size $O(n^{2k})$. ϕ_{start} only involves the cells in first row of the tableau, so it has size $O(n^k)$. So, ϕ has size $O(n^{2k})$, meaning it is polynomial.

Therefore, our tableau can be reduced in polynomial time to produce ϕ, so A is polynomial time reducible to SAT and thus SAT is NP-complete. □

From the Cook-Levin Theorem and Theorem 5.1, we can reach the following corollary.

Corollary 5.1. $SAT \in$ P if and only if P = NP.

6 Acknowledgements

We would like to thank our mentor, Kerri Lu, and the PRIMES Circle program coordinators, Marisa Gaetz and Mary Stelow, for their time, support, and guidance.

References

[1] Sisper, M. (2013). *Introduction to the Theory of Computation* (3rd ed.).