

**Tribhuvan University**

**Institute of Science and Technology**

**Seminar Report**

**On**

**The Complexity of Theorem Proving procedures**

**Submitted to**

**Central Department of Computer Science & Information Technology**

**Tribhuvan University, Kirtipur**

**Kathmandu, Nepal**

**Submitted by**

**Himal Raj Gentil**

Roll No.: 6 / 075

**In partial fulfillment of the requirement for Master's Degree in Computer Science and Information Technology (M.Sc. CSIT), 1st Semester**



**Tribhuvan University**

**Institute of Science and Technology**

**Student’s Declaration**

I hereby declare that I am the only author of this work and that no sources other than the listed here have been used in this work.

**Himal Raj Gentil**

**Supervisor’s Recommendation**

I hereby recommend that this Seminar report is prepared under my supervision by **Mr. Himal Raj Gentil** entitled “**The Complexity of Theorem Proving Procedures**” be accepted as fulfilment in partial requirement for the degree of Master's of Science in Computer Science and Information Technology. In my best knowledge, this is an original work in computer science.

... ... ... ... ... ... ... ... ... … … …

**Asst. Prof. Ram Krishna Dahal**

Central Department of Computer Science

and Information Technology



**Tribhuvan University**

**Institute of Science and Technology**

**LETTER OF APPROVAL**

This is certify that the seminar report prepared by Mr. Madan Nath entitled “**The complexity of Theorem Proving Procedures**” in partial fulfillment of the requirements for the degree of Master's of Science in Computer Science and Information technology has been well studied. In our opinion, it is satisfactory in the scope and quality as a project for the required degree.

**Evaluation Committee**

…………………………… …………………………….

**Asst. Prof. Nawaraj Paudel** **Asst. Prof. Ram Krishna Dahal**

Head of Central Department of Computer (Supervisor)

Science & Information Technology Central Department of Computer Science

& Information Technology

..…………………………………

(External)

Date:

# ABSTRACT

Computational Complexity Theory plays important role in Computer Science. It deals about the time and storage needed to compute computational algorithms. Computational time are classified into different computational complexity classes. Theses classes classify the computational problem into P, NP, NPC and NP-HARD problems, although there are many other complexity classes too. P vs. NP is one of the seven Millennium Prize problems which is dedicated to the field of computational complexity and is one of the biggest unsolved mystry of Theoretic Computer Science. Theoretic computer scientists and mathematicians are trying to solve this problem since last hundred years but still it remains unsolved till today. In this report, first two theorems of research paper named as “The Complexity of Theorem Proving Procedures” published by Stephen A. Cook , University of Toronto are discussed in detail. The terminologies in order to understand the research paper are discussed in detail and elavorated version of theorems are presented.

Keywords: Computational Complexity, P, NP, NPC, NP-HARD, Millennium Prize.

Table of Contents

[ABSTRACT iii](#_Toc28294150)

[List of Figure v](#_Toc28294151)

[LIST OF ABBREVIATIONS vi](#_Toc28294152)

[1. INTRODUCTION 1](#_Toc28294153)

[1.1 Overview 1](#_Toc28294154)

[1.2 Asymptotic Complexity 1](#_Toc28294155)

[1.3 Determinism and non-determinism in Algorithms 3](#_Toc28294156)

[1.3.1 Deterministic Models: 4](#_Toc28294157)

[1.3.2 Non Deterministic Models: 5](#_Toc28294158)

[2. COMPUTATIONAL COMPLEXITY CLASSES 7](#_Toc28294159)

[2.1 Introduction 7](#_Toc28294160)

[2.2 P class: 7](#_Toc28294161)

[2.3 NP Class: 8](#_Toc28294162)

[2.4 Overview of showing problems to be NP-complete 10](#_Toc28294163)

[2.4.1 Reductions 10](#_Toc28294164)

[2.4.2 A formal-language framework: 11](#_Toc28294165)

[2.5 Polynomial-time verification: 12](#_Toc28294166)

[2.5.1 Hamiltonian Cycles 12](#_Toc28294167)

[2.5.2 Verification Algorithms 13](#_Toc28294168)

[2.6 NP-completeness and reducibility 14](#_Toc28294169)

[2.6.1 Reducibility 14](#_Toc28294170)

[2.6.2 NP-completeness 16](#_Toc28294171)

[2.6.3 NP-completeness proofs: 16](#_Toc28294172)

[2.6.4 Formula Satisfiability: 17](#_Toc28294173)

[3. COMPLEXITY OF THEOREM PROVING PROCEDURES 20](#_Toc28294174)

[*Theorem 1* 20](#_Toc28294175)

[*Theorem 2* 22](#_Toc28294176)

[REFERENCES 26](#_Toc28294177)

# 

# List of Figure

[Figure 1.1 : Asymptotic complexity [3] 2](#_Toc28294178)

[Figure 1.2: Determinism and Non-determinism in algorithms[6] 3](#_Toc28294179)

[Figure 2.1:Commonly believed relationship between P and N [12] 9](file:///C:\Users\himal\Desktop\Seminar%20I.docx#_Toc28294180)

[Figure 2.2: Reduction process[13] 11](#_Toc28294181)

[Figure 2.3: One possible Hamiltonian cycle through every vertex of a dodecahedron[13] 13](#_Toc28294182)

[Figure 2.4: Polynomial Time Reduction[13] 15](#_Toc28294183)

[Figure 2.5: Reduction Algorithm[13] 15](#_Toc28294184)

[Figure 2.6: Relationship between P, NP, and NPC [13] 16](#_Toc28294185)

# LIST OF ABBREVIATIONS

NTM : Non-deterministic Turing Machine

P : Polynomial Time

NP : Non-deterministic Polynomail Time

NTIME : Non-polynomail Time

HAM-CYCLE : Hamiltonian Cycle

NPC : Non-deterministic Polynomail Time Complete

SAT : Satisiability Problem

# INTRODUCTION

## Overview

In computer science, computational complexity means how hard is a problem to be solved by distinguished model of computation. Basically, hardness of problem is defined by two factors time taken by computation machine to solve a particular problem and storage it needs during computation. Here, the distinguished model of computation is architecture of computer such as deterministic, non-deterministic, petri-nets, etc.

An algorithm has its computational complexity as the minimum of all possible algorithm to solve a problem. Complexity of an algorithm is determined as a function where, is the sie of the input and is the computational complexity which may be either best, worst or average case. These cases of complexities are bounded by the amount of resources i.e. time and storage used to compute the algorithm for certain problem [1].

The time taken by the algorithm defines the time complexity and the memory taken by the algorithm defines the space complexity. Throughout, this seminar report it is focused on the time complexity rather than space complexity although the space complexity also has great impact on defining the overall computational complexity of an algorithm.

## Asymptotic Complexity

In general, the complexity of an algorithm cannot be defined precisely rather than in best case. Best case complexity is assumed by the minimum number of steps taken on any instance of size which is represented by curve passing through the lowest point of each column. It might be more difficult to precisely define average case and worst case complexity. The worst case is defined by the maximum number of steps taken on any instance of size which is represented by the curve passing through the highest point of each column and the average case is the average number of steps taken on any instance of size . The best, average and worst time complexity are numerical function representing time versus problem size such as where represents the problem size and the final output of the function is time [2].

Due to the above mentioned reasons, it is generally focused on the behavior of the complexity for large , where tends to infinity i.e. asymptotic behavior of the complexity for large , when tends to infinity i.e. asymptotic behavior of the complexity which is expressed by big oh (O) notation.

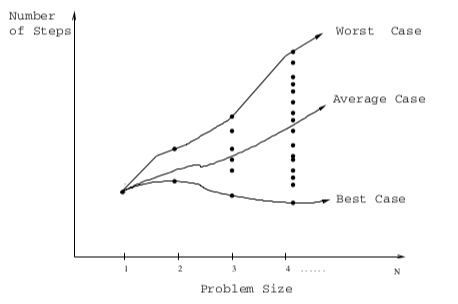


Figure 1.1 : Asymptotic complexity [3]

For example, let us take a function f(n) such as and . Here, determines the asymptotic behavior when n becomes very large. The slower growth rate of the asymptotic function determines the better algorithm. The above mentioned linear asymptotic function (best case) is always better than quadratic asymptotic function (worst case).

Average case analysis is an alternative to worst case analysis. In average case, it is not bounded by worst case but the time spent on randomly chosen input is calculated. Such kind of analysis is harder since probabilistic arguments are involved and often assumption about the distribution of input are required which might be difficult to justify [3,4].

## Determinism and non-determinism in Algorithms

In computer science, determinism in algorithms are referred to the property of an algorithm in which, algorithm always behaves the same way for some input whereas in non-determinism, the algorithm behavior cannot be predicted. Its behavior may change rom run to run for same input.

Deterministic algorithms are such algorithms that performs steps always finishes in steps and always returns the same result. Non-deterministic algorithm has quite different behavior. They contain levels and might not return the same result on different runs. A non-deterministic algorithm may never terminate due to the potentially infinite size of the fixed height tree. A non-deterministic algorithm can show different behavior on different runs, which is just opposite of a deterministic algorithm. A non-deterministic algorithm may behave differently on different runs due to several issues. For example, a probabilistic algorithm’s behaviors depend on the random number generator, a concurrent algorithm can perform differently on different runs due to a race condition. An algorithm that solves the problem in non-deterministic polynomial time can run in polynomial time or exponential time depending on the choices it makes during execution. The non-deterministic algorithms are often used to find an approximation to a solution, when the exact solution would be too costly to obtain using a deterministic one. [5]

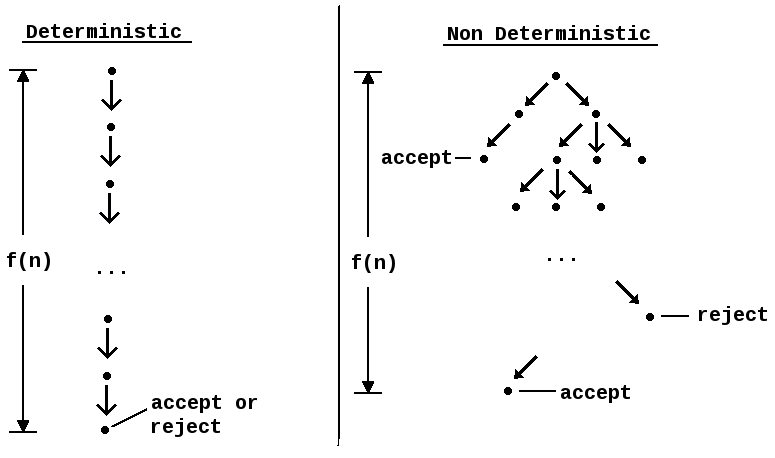


Figure 1.2: Determinism and Non-determinism in algorithms[6]

### Deterministic Models:

A model of computation such that the successive states of the machine and the operations to be performed are completely determined by the preceding state is known as deterministic model of computation. For example: Deterministic Turing Machine.

Turing machine is known to be a simple computer that reads and write symbols one at a time on endless tape by strictly following the set of rules. What action a Turing Machine should perform next is determined by its internal state and what symbol it currently sees. An example of one of a Turing Machine’s rules might thus be: “If you are in state 2 and you can see an ‘A’, change it to ‘B’, move left and change to state 3”.[7]

In a [deterministic Turing machine](https://en.wikipedia.org/wiki/Deterministic_Turing_machine) (DTM), the set of rules prescribes at most one action to be performed for any given situation.

Turing machine can be formally defined as a 7-[tuple](https://en.wikipedia.org/wiki/Tuple)  where

* {\displaystyle Q} is a finite, non-empty set of *states*;
* {\displaystyle \Gamma } is a finite, non-empty set of *tape alphabet symbols*;
* {\displaystyle b\in \Gamma }  is the *blank symbol* (the only symbol allowed to occur on the tape infinitely often at any step during the computation);
* {\displaystyle \Sigma \subseteq \Gamma \setminus \{b\}} is the set of *input symbols*, that is, the set of symbols allowed to appear in the initial tape contents;
* {\displaystyle q\_{0}\in Q}is the *initial state*;
* {\displaystyle F\subseteq Q} is the set of *final states* or *accepting states*. The initial tape contents is said to be *accepted* by {\displaystyle M}if it eventually halts in a state from {\displaystyle F}.
* {\displaystyle \delta :(Q\setminus F)\times \Gamma \not \to Q\times \Gamma \times \{L,R\}} is a [partial function](https://en.wikipedia.org/wiki/Partial_function) called the [*transition function*](https://en.wikipedia.org/wiki/State_transition_system), where is left shift, is right shift. If {\displaystyle \delta } is not defined on the current state and the current tape symbol, then the machine halts;[2]

Anything that operates according to these specifications is a Turing machine. A relatively uncommon variant allows "no shift", say N, as a third element of the set of directions {\displaystyle \{L,R\}}.

The 7-tuple for the 3-state [busy beaver](https://en.wikipedia.org/wiki/Busy_beaver) looks like this :

* {\displaystyle Q=\{{\mbox{A}},{\mbox{B}},{\mbox{C}},{\mbox{HALT}}\}} (states);
* {\displaystyle \Gamma =\{0,1\}}  (tape alphabet symbols);
* {\displaystyle b=0} (blank symbol);
* {\displaystyle \Sigma =\{1\}} (input symbols);
* (initial state);
* {\displaystyle F=\{{\mbox{HALT}}\}} (final states);
* {\displaystyle \delta =} (transition function).

Initially all tape cells are marked with .

### Non Deterministic Models:

In a [non-deterministic model of computation](https://en.wikipedia.org/wiki/Non-deterministic_algorithm), such as [non-deterministic Turing machines](https://en.wikipedia.org/wiki/Non-deterministic_Turing_machine), some choices may be done at some steps of the computation. In complexity theory, one considers all possible choices simultaneously, and the non-deterministic time complexity is the time needed, when the best choices are always done. In other words, it is considered that the computation is done simultaneously on as many (identical) processors as needed, and the non-deterministic computation time is the time spent by the first processor that finishes the computation.

By contrast, in a **nondeterministic Turing machine** (**NTM**),more than one action can be prescribed by set of rules. For example, an X on the tape in state 3 might allow the NTM to:

* Write a Y, move right, and switch to state 5

**or**

* Write an X, move left, and stay in state 3.

How does the NTM "know" which of these actions it should take? There are two ways of looking at it. It cab be assumed that the machine is the “luckiest possible guesser” which means it always picks a transition that eventually leads to an accepting state, if there is such a transition. The other is to imagine that the machine "[branches](https://en.wikipedia.org/wiki/Many-worlds_theory)" into many copies, each of which follows one of the possible transitions. Whereas a DTM has a single "computation path" that it follows, an NTM has a "computation tree". If at least one branch of the tree halts with an "accept" condition, it is said that the NTM accepts the input.[7]

A nondeterministic Turing machine can be formally defined as a 6-tuple  {\displaystyle M=(Q,\Sigma ,\iota ,\sqcup ,A,\delta )}, where

* {\displaystyle Q}is a finite set of states
* {\displaystyle \Sigma } is a finite set of symbols (the tape alphabet)
* {\displaystyle \iota \in Q} is the initial state
* {\displaystyle \sqcup \in \Sigma } is the blank symbol
* {\displaystyle A\subseteq Q} is the set of accepting (final) states
* {\displaystyle \delta \subseteq \left(Q\backslash A\times \Sigma \right)\times \left(Q\times \Sigma \times \{L,S,R\}\right)} is a relation on states and symbols called the transitionrelation. {\displaystyle L} is the movement to the left, {\displaystyle S} is no movement, and {\displaystyle R} is the movement to the right.[8]

The difference with a standard (deterministic) [Turing machine](https://en.wikipedia.org/wiki/Turing_machine) is that for those, the transition relation is a function (the transition function). Configurations and the yields on configurations, which describes the possible actions of the Turing machine given any possible contents of the tape, are as for standard Turing Machines, except that the yields relation is no longer single-valued The input for an NTM is provided in the same manner as for a deterministic Turing machine: the machine is started in the configuration in which the tape head is on the first character of the string (if any), and the tape is all blank otherwise. If and only if atleastone of the possible computational paths starting from that string puts the machine into an accepting state, at such condtion an input is accepted by NTM. An NTM’s operations are stopped as soon as any branch reaches an accepting state.

# COMPUTATIONAL COMPLEXITY CLASSES

## Introduction

The process of classifying computational problem into different classes according to their inherent difficulty, and relations between these classes among each other is said to be computational complexity theory. A problem is known to be inherently difficult if its solution requires significant resources, whatever the algorithm used. The computational complexity theory formalizes this intuition, by introducing mathematical [models of computation](https://en.wikipedia.org/wiki/Models_of_computation) to study these problems and quantifying their [computational complexity](https://en.wikipedia.org/wiki/Computational_complexity), i.e., the amount of resources needed to solve them, such as time and storage. Other measures of complexity are also used, such as the amount of communication (used in [communication complexity](https://en.wikipedia.org/wiki/Communication_complexity)), the number of [gates](https://en.wikipedia.org/wiki/Logic_gate) in a circuit (used in [circuit complexity](https://en.wikipedia.org/wiki/Circuit_complexity)) and the number of processors (used in [parallel computing](https://en.wikipedia.org/wiki/Parallel_computing)). One of the roles of computational complexity theory is to determine the practical limits on what computers can and cannot do. The [P versus NP problem](https://en.wikipedia.org/wiki/P_versus_NP_problem), one of the seven [Millennium Prize Problems](https://en.wikipedia.org/wiki/Millennium_Prize_Problems), is dedicated to the field of computational complexity.[9]

Analysis of algorithm and computability theory are known as closely related fields in theoretical computer science. Analysis of algorithm is devoted to analyzing the amount of resources needed by an particular algorithm to solve problem whereas complexity theory deals with all possible algorithms that could be used to solve same problem. The actual work of computational theory is try to classify problem that can or cannot be solved with appropriately restricted resources. Imposing restrictions on the available resources is what distinguishes computational complexity from computability theory. The computability theory focuses on what kind of problem can be solved algorithmically.

## P class:

An algorithm A is of polynomial complexity if there exists a polynomial such that computing time of A is or every input size . P is the set of all decision problems solvable by deterministic algorithm in polynomial time. An algorithm is said to be solvable in polynomial time if the number of steps required to complete the algorithm for a given input is or some non-negative integer k , where is the size of the input. It contains all [decision problems](https://en.wikipedia.org/wiki/Decision_problem) that can be solved by a [deterministic Turing machine](https://en.wikipedia.org/wiki/Deterministic_Turing_machine) using a [polynomial](https://en.wikipedia.org/wiki/Polynomial) amount of [computation time](https://en.wikipedia.org/wiki/Computation_time), or [polynomial time](https://en.wikipedia.org/wiki/Polynomial_time).

Formally, P is the class of languages that are decidable in polynomial time on a deterministic single tape Turing Machine. In other words

[10]

Polynomial-time algorithms are closed under composition. Intuitively, this says that if one writes a function that is polynomial-time assuming that function calls are constant-time, and if those called functions themselves require polynomial time, then the entire algorithm takes polynomial time. One consequence of this is that P is [low](https://en.wikipedia.org/wiki/Low_(complexity)) for itself. This is also one of the main reasons that P is considered to be a machine-independent class; any machine "feature", such as [random access](https://en.wikipedia.org/wiki/Random_access), that can be simulated in polynomial time can simply be composed with the main polynomial-time algorithm to reduce it to a polynomial-time algorithm on a more basic machine.

A [language](https://en.wikipedia.org/wiki/Formal_language) *L* is in P if and only if there exists a deterministic Turing machine *M*, such that

* runs for polynomial time on all inputs
* For all *x* in ,  outputs 1
* For all  not in outputs

The circuit definition can be weakened to use only a [logspace uniform](https://en.wikipedia.org/wiki/Circuit_complexity" \l "Logspace_uniform" \o "Circuit complexity) family without changing the complexity class.[11]

Sorting algorithm usually require or time. Bubble sort takes linear time in the best case, but in average and worst case. Heap sort takes in all cases. Quick sort takes on average and in worst case.

## NP Class:

NP class is the set of all decision problems solvable by non deterministic algorithms in polynomial time. Since, deterministic algorithms are just the special case of non-deterministic ones, it is concluded that . It is unknown and what has become perhaps the most unsolved problem in computer science is whether or . Decision problems are assigned complexity classes (such as NP) based on the fastest known algorithms. Therefore, decision problems may change classes if faster algorithms are discovered. It is easy to see that the complexity class [P](https://en.wikipedia.org/wiki/P_(complexity)) (all problems solvable, deterministically, in polynomial time) is contained in NP (problems where solutions can be verified in polynomial time), because if a problem is solvable in polynomial time then a solution is also verifiable in polynomial time by simply solving the problem. But NP contains many more problems, the hardest of which are known as[NP-complete](https://en.wikipedia.org/wiki/NP-complete) problems. An algorithm solving such a problem in polynomial time is also able to solve any other NP problem in polynomial time.

Figure 2.1:Commonly believed relationship between P and N [12]

The complexity class NP can be defined in terms of [NTIME](https://en.wikipedia.org/wiki/NTIME) as follows:

{\displaystyle {\mathsf {NP}}=\bigcup \_{k\in \mathbb {N} }{\mathsf {NTIME}}(n^{k}).}

where {\displaystyle {\mathsf {NTIME}}(n^{k})} is the set of decision problems that can be solved by a [non-deterministic Turing machine](https://en.wikipedia.org/wiki/Non-deterministic_Turing_machine) in  {\displaystyle O(n^{k})} time.

Alternatively, NP can be defined using deterministic Turing machines as verifiers. A [language](https://en.wikipedia.org/wiki/Formal_language)  is in NP if and only if there exist polynomials  and , and a deterministic Turing machine , such that

* For all  and , the machine  runs in time  on input {\displaystyle (x,y)}
* For all  in , there exists a string  of length  such that {\displaystyle M(x,y)=1}
* For all  not in  and all strings  of length , {\displaystyle M(x,y)=0} [10]

Boolean satisfiability problem, Knapsack problem, Vertex cover problem, Subgraph isomorphism problem are some of the NP-class problems which are solved in .

## Overview of showing problems to be NP-complete

Basically, if a problem is in NP and is “hard” as any problem in NP is said to be NP-complete problem. Till today, polynomial time solution for NP complete problem are not found although a wide range of NP-complete problem are not found although a wide range of NP-complete problem have been studied since many years. It would be truly astounding if all of them could be solved in polynomial time. Yet, given the effort devoted thus far to proving that NP-complete problems are intractable, without the conclusive outcome it cannot ruled out the possibility that the NP-complete problems are in fact solvable in polynomial time.

### Reductions

Reduction is known as the process of transforming a problem into another one so that it can be showed that one problem is no harder or no easier than other. The advantage of this idea is taken in almost every NP-completeness proofs. For example: Let it be considered a decision problem A called SHORTEST-PATH, it is given that an undirected graph and vertices and , and it is wished to find the path rom to that uses the fewest edges. Again, let it be considered there is a different decision problem, say B, that it is already known how to solve in polynomial time. Finally, suppose there is a procedure that transforms any instance of A into some instance of B with the following characteristics:

1. The transformation takes polynomial time.
2. The answers are the same. That is, the answer for is “yes” if and only if the answer for is also “yes”.

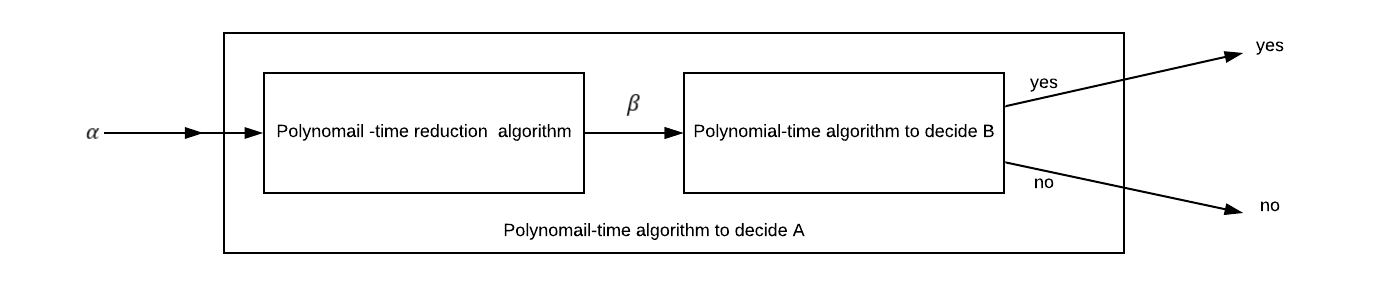


Figure 2.2: Reduction process[13]

Such procedure is called as a polynomial time reduction algorithm and it provides us a way to solve problem A in polynomial time.

1. Given an instance of problem A, use a polynomial-time reduction algorithm to transform it to an instance of problem B.
2. Run the polynomial-time decision algorithm for B on the instance
3. Use the answer for as the answer for .

As long as each of these steps takes polynomial time, all three together do also and so there is a way to decide on in polynomial time. In other words, by “reducing” solving problem B, “easiness” of B is used to prove the “easiness” of A. [13]

### A formal-language framework:

An alphabet is a finite set of symbols. A language L over is any set of strings made up of symbols from . For example, if = {0,1}, the set L={10,11,101,111,1011,1101,10001,…} is the language of binary representation of prime numbers. Empty string is denoted by , and the empty language by . The language of all strings over is denoted by . For example , if , then ={ ,0,1,00,01,10,11,000,….} is the set of all binary strings. Every language L over is a subset of .

The formal language framework allows us to express the relation between decision problem and algorithms that solve them concisely. If an algorithm A accepts a string if, given input x, the algorithm’s output A(x) is 1. The language accepted by an algorithm A is the set of strings i.e. the set of strings that the algorithm accepts. An algorithm A rejects a string if . A language L is decided by an algorithm A if every binary strings in L is accepted by A and every binary string not in L is rejected by A. A language L is accepted in polynomial time by an algorithm A if it is accepted by A and if in addition there is a constant k such that or any length-n string , algorithm A accepts x in . A language L is decided in polynomial time by an algorithm A if there is a constant k such that for any length-n string , the algorithm correctly decides whether in time . Thus, to accept a language, an algorithm need only worry about strings in , but to decide a language, it must correctly accept or reject every string in [13]

## Polynomial-time verification:

Algorithms can verify membership in languages. For example, suppose that for a given instance (G,u,v,k) of the decision problem PATH, it is also given a path p from u to v . It can be easily checked whether the length of the p is at most k, and if so, it can viewed p as a “certificate” that the instance needed belongs to PATH. For the decision problem PATH, this certificate does not seem to buy as much. After all, PATH belongs to P –in fact, PATH can be solved in linear time – and so verifying membership from a given certificate takes as long as solving the problem from scratch. For the no polynomial-time decision algorithm which it is known, given a certificate verification is easy.

### Hamiltonian Cycles

The problem of finding a Hamiltonian cycle in an undirected graph has been studied for over hundred years. Formally Hamiltonian cycle of an undirected graph is a simple cycle that contains each vertex in V. A graph that contains a Hamiltonian cycle is said to be Hamiltonian ; otherwise, it is non Hamiltonian.

Hamiltonian-cycle problem can be defined in formal language as:

Given a problem instance (G), one possible decision algorithm list all permutations of the vertices of G and them checks each permutation to see if it is a Hamiltonian path. If reasonable encoding of graph is used as its adjacency matrix, the number m of vertices in the graph is , where is the length of the encoding of G. There are m! possible permutations of the vertices, and therefore the running time is , which is not for any constant .’ [13]

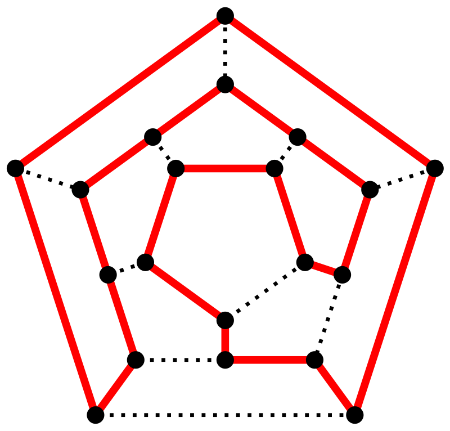
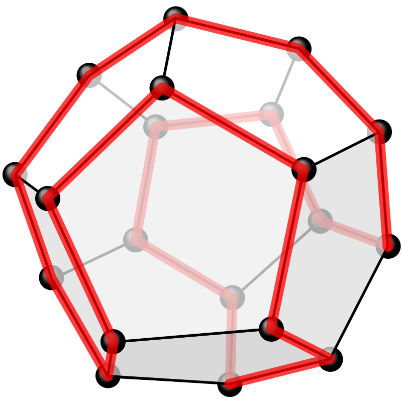


Figure 2.3: One possible Hamiltonian cycle through every vertex of a dodecahedron[13]

### Verification Algorithms

A verification algorithm is a two-argument algorithm A, where one argument is an ordinary input string x and the other is a binary string y called a certificate. A two-argument algorithm verifies an input string if there exists a certificate y such that . The language verified by a verification algorithm is .

Intuitively, an algorithm A verifies a language L if for any string . Moreover, for any string , there must be no certificate proving that . For example, in the Hamiltonian cycle problem, the certificate is the list of vertices in Hamiltonian cycle itself offers enough information to verify the fact. Conversely, if the graph is not Hamiltonian, there is no list of vertices that can fool the verification algorithm believing that the graph is Hamiltonian, since the verification algorithm carefully checks the proposed “cycle” to be sure. It can be verified that the provided cycle is Hamiltonian by checking whether it is the permutation o vertices and whether each of the consecutive edges along the cycle exists in the graph. This verification algorithm can be certainly implemented to run in time, where n is the length of encoding of .

The complexity class NP is the class of languages that can be verified by a polynomial-time algorithm. More precisely, a language belongs to NP if and only if there exist a two input polynomial-time algorithm the constant such that

which means that algorithm verifies language in polynomial time. [13]

From the above discussion on Hamiltonian cycle problem, it can be concluded that HAM-CYCLE ∈ NP.

## NP-completeness and reducibility

Perhaps the most compelling reason why theoretical computer scientists believe that is the existence of NP-complete problems. This class has surprising property that if any NP-complete problem can be solved in polynomial time, then every problem in NP has a polynomial-time solution, that is P = NP. Despite years of study, though, no polynomial-time algorithm has ever been discovered for any NP-complete problem.

The language HAM-CYCLE is one NP-complete problem. If it could be decided HAM-CYCLE in polynomial time, then it could be solved every problem in NP in polynomial time. In fact, if NP-P should turn out to be non-empty, it could be said with certainty that HAM-CYCLE ∈ NP-P.[13]

### Reducibility

A problem Q can be reduced to another problem Q’ if any instance of Q can be easily rephrased as an instance of Q’, the solution to which provides a solution to the instance of Q. For example, the problem of solving linear equation in an indeterminate x reduces to the problem of solving the quadratic equations. Given an instance , it could be transformed to , whose solution provides a solution to . Thus, I a problem Q reduces to another problem Q’, then Q is, in a sense, “no harder to solve” than Q’.

In formal language framework for decision problems, it could be said that a language L1 is polynomially reducible to a language L2, written if there exists a polynomial-time computable function such that or all , if and only if .

Function can be called as the reduction function, and the polynomial-time algorithm that computes is called a reduction algorithm.

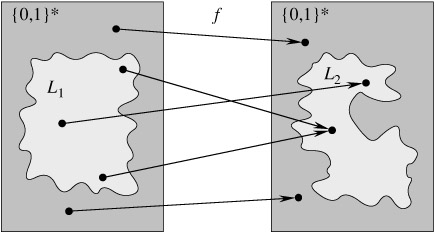


Figure 2.4: Polynomial Time Reduction[13]

An illustration of a polynomial-time reduction from a language *L*1 to a language *L*2 via a reduction function *f*. For any input *x* ∈ {0, 1}\*, the question of whether *x* ∈ *L*1 has the same answer as the question of whether *f*(*x*) ∈ *L*2. Each language is a subset of {0, 1}\*. The reduction function *f* provides a polynomial-time mapping such that if *x* ∈ *L*1, then *f*(*x*) ∈ *L*2. Moreover, if *x* ∉ *L*1, then *f* (*x*) ∉ *L*2. Thus, the reduction function maps any instance *x* of the decision problem represented by the language *L*1 to an instance *f* (*x*) of the problem represented by *L*2. Providing an answer to whether *f*(*x*) ∈ *L*2 directly provides the answer to whether *x* ∈ *L*1.

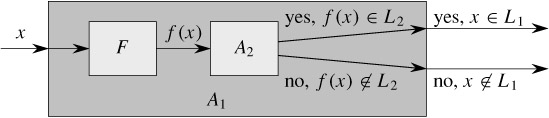


Figure 2.5: Reduction Algorithm[13]

The algorithm *F* is a reduction algorithm that computes the reduction function *f* from *L*1 to *L*2 in polynomial time, and *A*2 is a polynomial-time algorithm that decides *L*2. Illustrated is an algorithm *A*1 that decides whether *x* ∈ *L*1 by using F to transform any input *x* into *f* (*x*) and then using *A*2 to decide whether *f*(*x*) ∈ *L*2. [13]

### NP-completeness

Polynomial-time reductions provide a formal means for showing that one problem is at least as hard as another, to within a polynomial-time factor. That is, if *L*1 ≤P *L*2, then *L*1 is not more than a polynomial factor harder than *L*2, which is why the "less than or equal to" notation for reduction is mnemonic. The set of NP-complete languages, which are the hardest problems in NP can be defined as

A language *L* ⊆ {0, 1}\* is *NP-complete* if

1. *L* ∈ NP, and
2. *L*′ ≤P *L* for every *L*′∈ NP.

If a language *L* satisfies property 2, but not necessarily property 1, it is said that *L* is *NP-hard*. It is also defined NPC to be the class of NP-complete languages. [13]

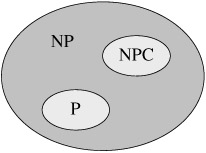


Figure 2.6: Relationship between P, NP, and NPC [13]

Both P and NPC are wholly contained within NP, and P ∩ NPC = Ø.

### NP-completeness proofs:

The NP-completeness of the circuit-satisfiability problem relies on a direct proof that *L* ≤P CIRCUIT-SAT for every language *L* ∈ NP. In this section, it is shown how to prove that languages are NP-complete without directly reducing *every* language in NP to the given language. This methodology should be illustrated by proving that various formula-satisfiability problems are NP-complete.

***Lemma 1:* *If L is a language such that L′ ≤P L for some L′ ∈ NPC, then L is NP-hard. Moreover, if L ∈ NP, then L ∈ NPC*** [5]***.***

***Proof*** : Since *L*′ is NP-complete, for all *L*′′∈ NP, then *L*′′≤P *L*′. By supposition, *L*′ ≤P *L*, and thus by transitivity,  *L*′′ ≤P *L*, which shows that *L* is NP-hard. If *L* ∈ NP, then *L* ∈ NPC.

In other words, by reducing a known NP-complete language *L*′ to *L*, it is implicitly reduced every language in NP to *L*. Thus, [Lemma](http://www.euroinformatica.ro/documentation/programming/!!!Algorithms_CORMEN!!!/DDU0231.html#ch34ex33) 1 gives us a method for proving that a language *L* is NP-complete:

1. Prove *L* ∈ NP.
2. Select a known NP-complete language *L*′.
3. Describe an algorithm that computes a function *f* mapping every instance *x* ∈ {0, 1}\* of *L*′ to an instance *f*(*x*) of *L*.
4. Prove that the function *f* satisfies *x* ∈ *L*′ if and only if *f* (*x*) ∈ *L* for all *x* ∈ {0, 1}\*.
5. Prove that the algorithm computing *f* runs in polynomial time.

(Steps 2-5 show that *L* is NP-hard.) This methodology of reducing from a single known NP-complete language is far simpler than the more complicated process of showing directly how to reduce from every language in NP. Proving CIRCUIT-SAT ∈ NPC has given us a "foot in the door." Knowing that the circuit-satisfiability problem is NP-complete now allows us to prove much more easily that other problems are NP-complete. Moreover, as it is developed a catalog of known NP-complete problems, there will be more and more choices for languages from which to reduce. [13]

### Formula Satisfiability:

When the exponential time consuming problems cannot be solved in in polynomial time, it could be shown the similarities between those algorithms so that if one problem is solved in polynomial time, then exponential time consuming problems can also be solved in polynomial time. For gaining similarities between exponential time consuming problems, association between them should be shown in-order to show that the properties they are having are similar such that if one is solved then another can also be solved. In-order to relate them it is needed some problem as base problem called Formula Satisfiability also called Boolean Satisfiability Problem.

It can be formulated the **(formula) satisfiability** problem in terms of the language SAT as follows. An instance of SAT is a boolean formula *φ* composed of

1. *n* boolean variables: *x*1, *x*2, ..., *xn*;
2. *m* boolean connectives: any boolean function with one or two inputs and one output, such as  (AND),  (OR), ¬ (NOT) and
3. parentheses. (Without loss of generality, it is assumed that there are no redundant parentheses, i.e., there is at most one pair of parentheses per boolean connective.)

A Boolean formula is satisfiable if there exists some assignment of the values 0 and 1 to its variables that causes it to evaluate true i.e. 1. [11]

Boolean satisfiability problem i.e.SAT can be represented as Disjunctive Normal Form (DNF) and Conjunctive Normal Form (CNF).

DNF is formed by Oring of clauses of AND’s.

For example:

3-DNF is formed by clauses having only three distinct variable.

For example:

CNF is formed by Anding of clauses of OR’s.

For example:

3-CNF is formed by clauses having only three distinct variable.

For example:

The satisfiability problem asks whether a given boolean formula is satisfiable; in formal-language terms,

SAT = {〈*φ* : *φ* is a satisfiable boolean formula}.

As an example, the formula

*φ* =

has the satisfying assignment 〈*x*1 = 1, *x*2 = 0, *x*3 = 0〉

*φ* =

=

=(1)

=1

and thus and thus this formula *φ* belongs to SAT.

The naive algorithm to determine whether an arbitrary boolean formula is satisfiable does not run in polynomial time. There are 2*n* possible assignments in a formula *φ* with *n* variables. If the length of 〈*φ*〉 is polynomial in *n*, then checking every assignment requires Ω(2*n*) time, which is superpolynomial in the length of 〈*φ*〉. [13]

|  |
| --- |
|  |

# COMPLEXITY OF THEOREM PROVING PROCEDURES

*Theorem 1****.*** ***If a set L’ of string is accepted by some non-deterministic Turing Machine within polynomial time, then L’ is P-reduicible to L {DNF Tautologies} i.e. L’ in NP L .***

**Proof**:

Consider a non- deterministic Turing Machine working in polynomial time.

Given, where =Input string and =(DNF Boolean Expression i.e. Output)

i.e. and write such that is satisfiable if and only if accepts .

can be constructed in polynomial time from M and w.

M has {q1,q2,q3,…….qs} as state set.

1,x2,x3,……….xm} as the input symbols where x1=blank symbol

Id’s of the turing machine =

After some states at will accept or reject.

q ≤ P(n)

In-order to find expression some boolean variables must be used:

1. C< i, j, t > = 1 if the ith cell in the tape contains jth  symbol at time t, Otherwise 0.

Range for:

i => 1 ≤ i ≤ P(n)

j => 1 ≤ j ≤ m

t => 1 ≤ t ≤ P(n)

There are O(P2(n)) variables of this form.

1. H< i, t, > = 1 if the head is scanning ith cell at time t, Otherwise 0.

i => 1 ≤ i ≤ P(n)

t => 1 ≤ t ≤ P(n)

There are O(P2(n)) variables of this form.

1. S< k, t > = 1 if the state is qk at time t.

k => 1 ≤ k ≤ s

t => 1 ≤ t ≤ P(n)

There are O(P(n)) variables of this form.

is satisfied if and only if and only if it represents a sequence of valid sequence of moves.

*Conditions to Consider:*

1. The head is scanning only one cell at any instance t.

2. Each cell contains only one symbol at any instance t.

3. Sate is unique for particular t.

4. The contents of the cell pointed by head alone changes in the next instant.

5. The change is specified by a move of turing machine.

6. Initial Id

7. Final Id

Each variable is represented by a symbol.

Notation: ⋃(x1,x2,………xr)=(x1 ∨ x2 ∨ x3……∨ xr => 1 iff exactly one of x1……..xr = 1 others 0

* + - * Length of the expression r2

*For condition 1:*

At = ⋃ (H< 1, t >,H< 2, t >,…………….H< P(n), t>)

A=A0 . A1. A2 . ………… . AP(n)

Length of A = O(P3(n))

*For condition 2:*

B=

B i,t = ⋃( C< i, 1, t >, C< i, 2, t >,…………..C<i, m, t> )

B=B0 B1……………BP(n)

Length of B = O(p2(n))

*For condition 3:*

St = ⋃(S< 1 , t >, S< 2, t >, ……………….S< 3, t >)

C=S0 S1……………SP(n)

Length of St = O(P(n))

*For condition 4:*

D=C0 C1……………CP(n)

Length of D = O(p2(n))

*For condition 5:*

E = E0 E1……………EP(n)

Length of E = O(p2(n))

*For condition 6:*

F = H< 1, 0 > S< 1, 0 > C< 1, ↓, 0 > C<1 , ↓, 0)……………… C< n, ↓, 0 > C< n+1, 1, 0) C<n+2, 1, 0>………. C< P(n), 1, 0) >

*For condition 7*:

G = S<2, P(n)>

So, the total complexity of wo = ABCDEF = O(p3(n)) which is polynomial time.

This completes the proof of theorem 1.

If L’ ∈ NP, L is accepted by NTM M and L’ L (Boolean Satisfiability Problem)

This also shows that Boolean Satisfiability Problem in NP-Complete. [14]

*Theorem 2****: DNF tautologies is P-reducible to D3 and D3 is P-reduicible to Subgraph Pairs*** [7]***.***

***Proof***

To show DNF tautologies is P-reduicible to D3, let A be a proposition formula in disjunctive normal form. Say , where, each Ri , is an atom or negation of atom, and . Then A is a tautology I and only if is a tautology where

Where is a new atom. Since it has reduced the number of conjuncts in , this process may be repeated until eventually a formula is found with at most three conjuncts per disjunct. Clearly the entire process is bounded in time by a polynomial in the length of .

It remains to show that D3 is P-reducible to subgraph pairs. Suppose is formula in disjunctive normal form with three conjuncts per disjunct.Thus , where , and each is an atom or negation of an atom. Now let be the complete graph with vertices {v1, v2, …….vk}, and let be the graph with vertices 1i , such that is connected by an edge to if and only if , and the two literals (,) do not form an opposite pair (that is they are neither of the form nor of the form ()). Thus there is a alsiying truth assignment to the formula iff there is a graph homomorphism such that for each , for some . (The homomorphism tells for each which of should be falsified, and the selective lack of edges in guarantees that the resulting truth assignment in consistently specified.)

In order to guarantee that a one-one homomorphism has the property that for each , for some , and are modiied as follows. Graphs are selected which are sufficiently distinct from each other that if is formed from by attaching to , , and is formed from by attaching to each of , then every one-one homomorphism has the property just stated. It is not hard to see such a construction csn be carried out in polynomial time. Then can be embedded in if and only if A D3. This completes the proof of theorem 2. [14]

***Illustration****:* ***D3 reducible to subgraph pairs.***

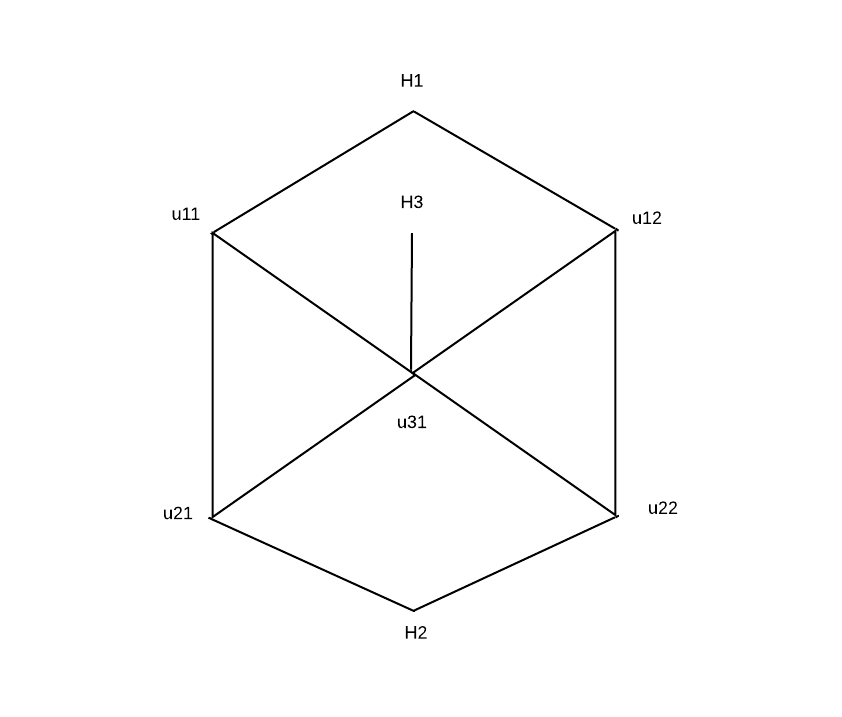
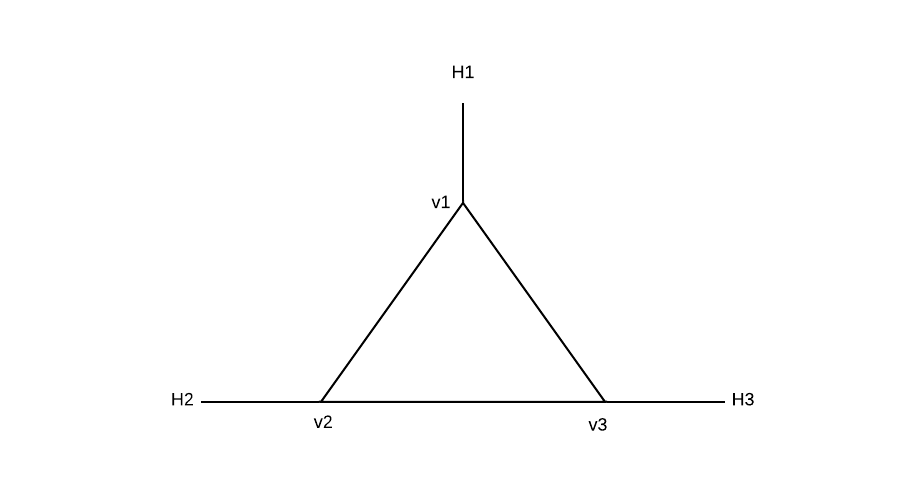


Figure 3.2: G2'

Figure 3.1: G1'

Adjacent Matrix for G2:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | u11 | u12 | u21 | u22 | u31 | H1 | H2 | H3 |
| u11 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 |
| u12 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| u21 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| u22 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 |
| u31 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| H1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| H2 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| H3 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |

such that for each , for some :

=

=

=

=

=

Here, ,,,, all satisfies the falsifying truth assignment.

And A=

Also, A=

Here, there are two possibilities of embedding G1 to G2 i.e. v1, v2, v3 can be embedded on , and , .

So, G1 is subgraph pair of G2

{\displaystyle c\_{l}.}

# REFERENCES

[1] Computational complexity. (2019, November 25). Retrieved from <https://en.wikipedia.org/wiki/Computational_complexity>

[2] Oemrawsingh, H., & Ollongren, A. (1979). On the proof of a theorem by chomsky—hopcroft—ullman. International Journal of Computer Mathematics, 7(1), 37–41. doi: 10.1080/00207167908803154

[3,4,5] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (n.d.). Patience-Hall of India (2nd ed.). ISBN-81-203-2141-3

[6,7] Cook, S. A. (1971). The complexity of theorem-proving procedures. Proceedings of the Third Annual ACM Symposium on Theory of Computing - STOC 71. doi: 10.1145/800157.805047