

**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](#_Toc27849889)

[List of Figure v](#_Toc27849890)

[LIST OF ABBREVIATIONS vi](#_Toc27849891)

[1. INTRODUCTION 1](#_Toc27849892)

[1.1 Overview 1](#_Toc27849893)

[1.2 Asymptotic Complexity 1](#_Toc27849894)

[1.3 Models of computation 3](#_Toc27849895)

[1.3.1 Deterministic Models: 4](#_Toc27849896)

[1.3.2 Non Deterministic Models: 5](#_Toc27849897)

[2. COMPUTATIONAL COMPLEXITY CLASSES 8](#_Toc27849898)

[2.1 Introduction 8](#_Toc27849899)

[2.2 P class: 8](#_Toc27849900)

[2.3 NP Class: 10](#_Toc27849901)

[2.4 Overview of showing problems to be NP-complete 11](#_Toc27849902)

[2.4.1 Decision Problems vs. Optimization Problems 11](#_Toc27849903)

[2.4.2 Reductions 12](#_Toc27849904)

[2.4.3 A formal-language framework: 13](#_Toc27849905)

[2.5 Polynomial-time verification: 14](#_Toc27849906)

[2.5.1 Hamiltonian Cycles 14](#_Toc27849907)

[2.5.2 Verification Algorithms 15](#_Toc27849908)

[2.6 NP-completeness and reducibility 16](#_Toc27849909)

[2.6.1 Reducibility 16](#_Toc27849910)

[2.6.2 NP-completeness 18](#_Toc27849911)

[2.6.3 Circuit Satisfiability 18](#_Toc27849912)

[2.6.4 NP-completeness proofs: 25](#_Toc27849913)

[2.6.5 Formula Satisfiability: 26](#_Toc27849914)

[3. COMPLEXITY OF THEOREM PROVING PROCEDURES 28](#_Toc27849915)

[*Theorem 1* 28](#_Toc27849916)

[*Theorem 2* 30](#_Toc27849917)

[REFERENCES 34](#_Toc27849918)

# 

# List of Figure

[Figure 1.1 : Asymptotic complexity 2](#_Toc27849953)

[Figure 1.2: Models of computation 3](#_Toc27849954)

[Figure 2.1:Commonly believed relationship between P and NP 10](file:///C:\Users\himal\Desktop\Seminar%20I.docx#_Toc27849955)

[Figure 2.2: Reduction process 12](#_Toc27849956)

[Figure 2.3: One possible Hamiltonian cycle through every vertex of a dodecahedron 15](#_Toc27849957)

[Figure 2.4: Polynomial Time Reduction 17](#_Toc27849958)

[Figure 2.5: Reduction Algorithm 17](#_Toc27849959)

[Figure 2.6: Relationship between P, NP, and NPC 18](#_Toc27849960)

[Figure 2.7: (a) The NOT gate. (b) The AND gate. (c) The OR gate. 19](#_Toc27849961)

[Figure 2.8: Instances of Circuit Satisfiability Problem 20](#_Toc27849962)

[Figure 2.9: The sequence of configurations produced by an algorithm 23](#_Toc27849963)

# 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 unction 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 we focus 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, we cannot precisely define the complexity of an algorithm 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.

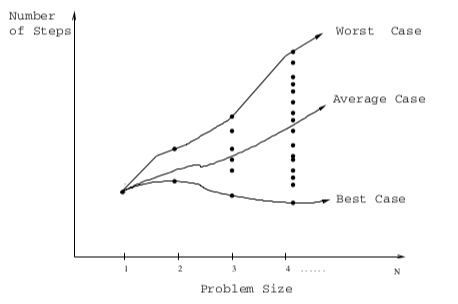


Figure 1.1 : Asymptotic complexity [3]

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 refers 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.

An algorithm that performs steps always finishes in steps and always returns the same result is known as deterministic algorithm whereas an algorithm that has levels might not return the same result on different runs is known as non-deterministic algorithm. A non-deterministic algorithm may never finish due to the potentially infinite size of the fixed height tree. In non-deterministic algorithm, even for the same input, can exhibit different behaviors on different runs, as opposed to a [deterministic algorithm](https://en.wikipedia.org/wiki/Deterministic_algorithm).. A [concurrent algorithm](https://en.wikipedia.org/wiki/Concurrent_algorithm) can perform differently on different runs due to a [race condition](https://en.wikipedia.org/wiki/Race_condition). A [probabilistic algorithm](https://en.wikipedia.org/wiki/Probabilistic_algorithm)'s behaviors depends on a [random number generator](https://en.wikipedia.org/wiki/Random_number_generator). An algorithm that solves a problem in [nondeterministic polynomial time](https://en.wikipedia.org/wiki/Nondeterministic_polynomial_time) can run in polynomial time or exponential time depending on the choices it makes during execution. The nondeterministic 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].

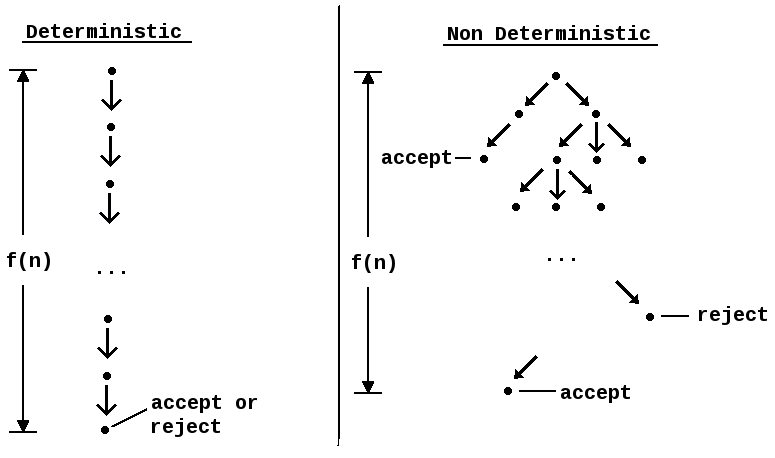


Figure 1.2: Determinism and Non-determinism in algorithms[6]

### Deterministic Models:

A deterministic [model of computation](https://en.wikipedia.org/wiki/Model_of_computation), for example a [deterministic Turing machine](https://en.wikipedia.org/wiki/Deterministic_Turing_machine), is 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.

A [deterministic algorithm](https://en.wikipedia.org/wiki/Deterministic_algorithm) is an algorithm which, given a particular input, will always produce the same output, with the underlying machine always passing through the same sequence of states. There may be non-deterministic algorithms that run on a deterministic machine, for example, an algorithm that relies on random choices. Generally, for such random choices, one uses a [pseudorandom number generator](https://en.wikipedia.org/wiki/Pseudorandom_number_generator), but one may also use some external physical process, such as the last digits of the time given by the computer clock.

A pseudorandom number generator is a deterministic algorithm, that is designed to produce sequences of numbers that behave as random sequences. A [hardware random number generator](https://en.wikipedia.org/wiki/Hardware_random_number_generator), however, may be non-deterministic.

In essence, a Turing machine is imagined to be a simple computer that reads and writes symbols one at a time on an endless tape by strictly following a set of rules. It determines what action it should perform next according to 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 see an 'A', change it to 'B', move left, and change to state 3."

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.

A deterministic Turing machine has a transition function that, for a given state and symbol under the tape head, specifies three things:

* the symbol to be written to the tape,
* the direction (left, right or neither) in which the head should move, and
* the subsequent state of the finite control.

For example, an X on the tape in state 3 might make the DTM write a Y on the tape, move the head one position to the right, and switch to state 5.

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, one considers 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**), the set of rules may prescribe more than one action to be performed for any given situation. 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. One is to say that the machine is the "luckiest possible guesser"; 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, we say that the NTM accepts the input.

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[3].

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 relation 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. (If the machine is deterministic, the possible computations are all prefixes of a single, possibly infinite, path.)

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. An NTM accepts an input string if and only if atleastone of the possible computational paths starting from that string puts the machine into an accepting state. When simulating the many branching paths of an NTM on a deterministic machine, we can stop the entire simulation as soon as any branch reaches an accepting state.

# COMPUTATIONAL COMPLEXITY CLASSES

## Introduction

Computational complexity theory focuses on classifying computational problems according to their inherent difficulty, and relating these classes to each other. A computational problem is a task solved by a computer. A computation problem is solvable by mechanical application of mathematical steps, such as an [algorithm](https://en.wikipedia.org/wiki/Algorithm).

A problem is regarded as inherently difficult if its solution requires significant resources, whatever the algorithm used. The 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.

Closely related fields in theoretical computer science are [analysis of algorithms](https://en.wikipedia.org/wiki/Analysis_of_algorithms) and [computability theory](https://en.wikipedia.org/wiki/Computability_theory). A key distinction between analysis of algorithms and computational complexity theory is that the former is devoted to analyzing the amount of resources needed by a particular algorithm to solve a problem, whereas the latter asks a more general question about all possible algorithms that could be used to solve the same problem. More precisely, computational complexity theory tries to classify problems that can or cannot be solved with appropriately restricted resources. In turn, imposing restrictions on the available resources is what distinguishes computational complexity from computability theory: the latter theory asks what kind of problems can, in principle, 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).

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

P can also be viewed as a uniform family of [boolean circuits](https://en.wikipedia.org/wiki/Boolean_circuit" \o "Boolean circuit). A language *L* is in P if and only if there exists a [polynomial-time uniform](https://en.wikipedia.org/wiki/Circuit_complexity#Polynomial-time_uniform) family of boolean circuits , such that

* For all, {\displaystyle C\_{n}} takes  bits as input and outputs bit
* For all  in , {\displaystyle C\_{|x|}(x)=1}
* For all  not in ,  {\displaystyle C\_{|x|}(x)=1} {\displaystyle C\_{|x|}(x)=0}

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.

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, we conclude that . What we don’t know 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 called [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 NP

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}

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

Informally, a problem is in the class NP-complete – if it is in NP and is as “hard” as any problem in NP. Most theoretical computer scientists believe that the NP-complete problems are intractable, since given the wide range of NP-complete problems that have been studied too date without anyone having discovered a polynomial time solution to any of them to any of them, 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 we cannot rule out the possibility that the NP-complete problems are in fact solvable in polynomial time.

### Decision Problems vs. Optimization Problems

Many problems of interest are optimization problems, in which each feasible (i.e. “legal”) solution has an associated value, and we wish to find the feasible solution with the best value. For example, in a problem that we call SHORTEST-PATH, we are given an undirected graph and vertices and , and we wish to find the path from to that uses the fewest edges. NP-completeness applies directly not to optimization problems, however, but to decision problems, in which the answer is simply “yes” or “no” (or, more formally, “1” or “0”.

The given optimization problem can be casted as a related decision problem by imposing a bound to the value to be optimized. For SHORTEST-PATH, for example, a related decision problem, which we call PATH, is whether, given a directed graph , vertices and , and an integer , a path exists from to consisting of at most K-edges.

The relationship between an optimization and its related decision problem works in our favor when we try to show that the optimization problem is “hard”. That is decision problem in a sense “easier” or at least “no harder”. As a specific example, we can solve PATH by solving SHORTEST-PATH and then computing the number of edges in the shortest path found to the value of the decision problem parameter . In the other words, if an optimization problem is easy, its related decision problem is easy as well. Stated in way that has more relevance to NP-completeness, if we can provide evidence that a decision problem is hard, we also provide evidence that its related optimization problem is hard. Thus, even though it restricts attention to decision problems, the theory of NP-completeness often has implications for optimization problem as well.

### Reductions

The notation of showing that one problem is no harder or no easier than another applies when the both problems are decision problems. The advantage of this idea is taken in almost every NP-completeness proof, as follows. Let us consider a decision problem, say A, which we would like to solve in polynomial time. We call the input to a particular problem an instance of that problem; or example, in PATH, an instance would be a particular graph G, particular vertices u and v of G, and a particular integer k. Now suppose that there is a different decision problem, say B, that we already know how to solve in polynomial time. Finally, suppose that we have 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”.

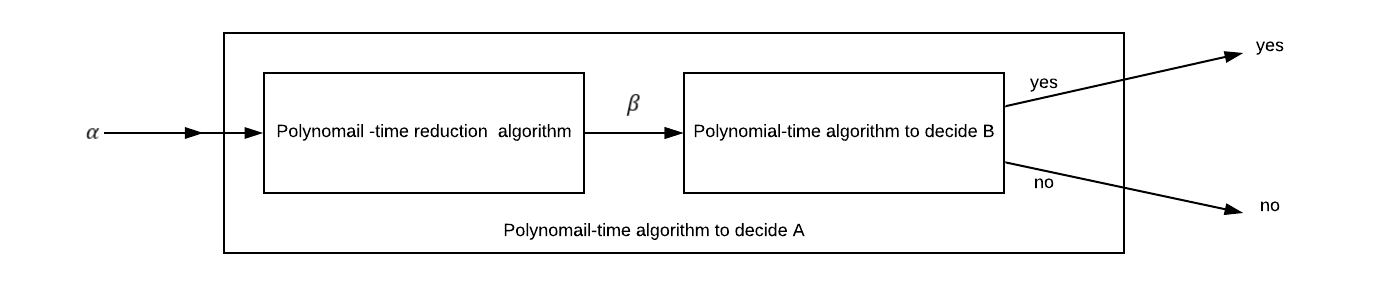


Figure 2.2: Reduction process

We call such a procedure 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 we have a way to decide on in polynomial time. In other words, by “reducing” solving problem B, we use “easiness” of B to prove the “easiness” of A.

### 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. We denote the empty string 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. We say that 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

## 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, we are also given a path p from u to v . We can easily check whether the length of the p is at most k, and if so, we can view 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 we know, given a certificate verification is easy.

### Hamiltonian Cycles

The problem of finding a Hamiltonian cycle in an undorectd graph has been studied for over hundred years. Formally Hamiltonian cycle of an undirected graph G=(V,E) 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.[0]Bondy an murty.

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

HAM\_CYCLE ={(G) : G is a Hamiltonian graph}

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 we use reasonable encoding of graph 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 .’

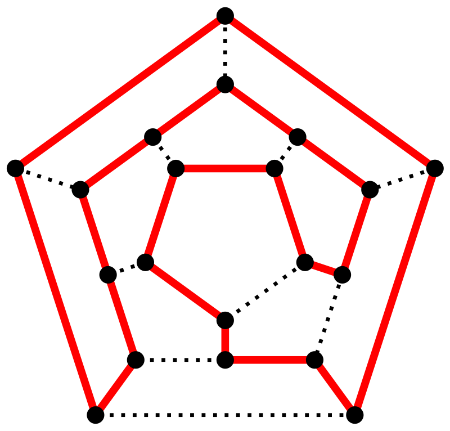
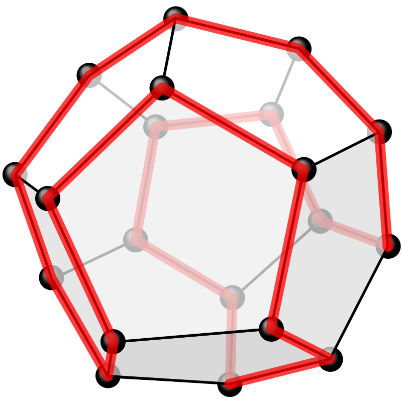


Figure 2.3: One possible Hamiltonian cycle through every vertex of a dodecahedron

### 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.

From the above discussion on Hamiltonian cycle problem, it follows 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 we could decide HAM-CYCLE in polynomial time, then we could solve every problem in NP in polynomial time. In fact, if NP-P should turn out to be nonempty, we could say with certainty that HAM-CYCLE ∈ NP-P.

### 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 , we transform it 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, we say 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 .

We call the function the reduction function, and the polynomial-time algorithm that computes is called a reduction algorithm.

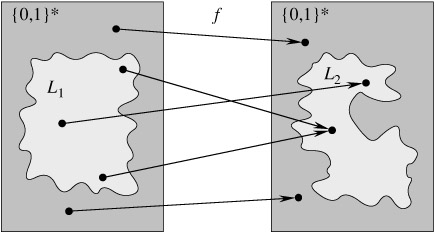


Figure 2.4: Polynomial Time Reduction

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.

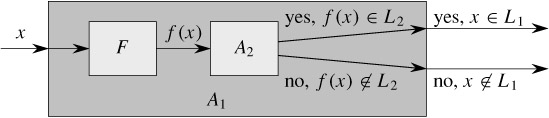


Figure 2.5: Reduction Algorithm

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.

### 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. We can now define the set of NP-complete languages, which are the hardest problems in NP.

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, we say that *L* is *NP-hard*. We also define NPC to be the class of NP-complete languages.

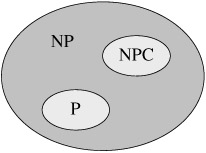


Figure 2.6: Relationship between P, NP, and NPC

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

### Circuit Satisfiability

We have defined the notion of an NP-complete problem, but up to this point, we have not actually proved that any problem is NP-complete. Once we prove that at least one problem is NP-complete, we can use polynomial-time reducibility as a tool to prove the NP-completeness of other problems. Thus, we now focus on demonstrating the existence of an NP-complete problem: the circuit-satisfiability problem.

Unfortunately, the formal proof that the circuit-satisfiability problem is NP-complete requires technical detail beyond the scope of this text. Instead, we shall informally describe a proof that relies on a basic understanding of boolean combinational circuits.

Boolean combinational circuits are built from boolean combinational elements that are interconnected by wires. A boolean combinational element is any circuit element that has a constant number of boolean inputs and outputs and that performs a well-defined function. Boolean values are drawn from the set {0, 1}, where 0 represents FALSE and 1 represents TRUE.

The boolean combinational elements that we use in the circuit-satisfiability problem compute a simple boolean function, and they are known as logic gates. Figure shows the three basic logic gates that we use in the circuit-satisfiability problem: the NOT gate (or inverter), the AND gate, and the OR gate. The NOT gate takes a single binary input *x*, whose value is either 0 or 1, and produces a binary output *z* whose value is opposite that of the input value. Each of the other two gates takes two binary inputs *x* and *y* and produces a single binary output *z*.

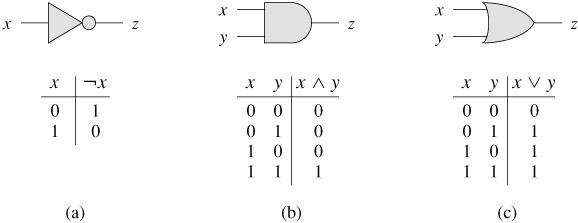


Figure 2.7: (a) The NOT gate. (b) The AND gate. (c) The OR gate.

The operation of each gate, and of any boolean combinational element, can be described by a truth table, shown under each gate in Figure. A truth table gives the outputs of the combinational element for each possible setting of the inputs. For example, the truth table for the OR gate tells us that when the inputs are *x* = 0 and *y* = 1, the output value is *z* = 1. We use the symbols ¬ to denote the NOT function,  to denote the AND function, and  to denote the OR function. Thus, for example, 0 1  = 1.

We can generalize AND and OR gates to take more than two inputs. An AND gate's output is 1 if all of its inputs are 1, and its output is 0 otherwise. An OR gate's output is 1 if any of its inputs are 1, and its output is 0 otherwise.

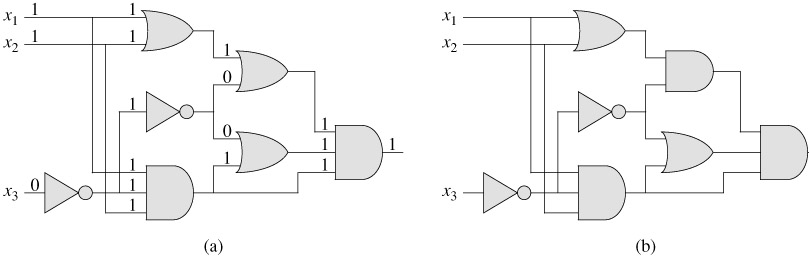


Figure 2.8: Instances of Circuit Satisfiability Problem

(a) The assignment 〈*x*1 = 1, *x*2 = 1, *x*3 = 0〉 to the inputs of this circuit causes the output of the circuit to be 1. The circuit is therefore satisfiable.

(b) No assignment to the inputs of this circuit can cause the output of the circuit to be 1. The circuit is therefore unsatisfiable.

The circuit-satisfiability problem is, "Given a boolean combinational circuit composed of AND, OR, and NOT gates, is it satisfiable?" In order to pose this question formally, however, we must agree on a standard encoding for circuits. The size of a boolean combinational circuit is the number of boolean combinational elements plus the number of wires in the circuit. One can devise a graphlike encoding that maps any given circuit *C* into a binary string 〈*C*〉 whose length is polynomial in the size of the circuit itself. As a formal language, we can therefore define

CIRCUIT-SAT = {〈*C*〉 : *C* is a satisfiable boolean combinational circuit}.

The circuit-satisfiability problem arises in the area of computer-aided hardware optimization. If a subcircuit always produces 0, that subcircuit can be replaced by a simpler subcircuit that omits all logic gates and provides the constant 0 value as its output. It would be helpful to have a polynomial-time algorithm for this problem.

Given a circuit *C*, we might attempt to determine whether it is satisfiable by simply checking all possible assignments to the inputs. Unfortunately, if there are *k* inputs, there are 2*k* possible assignments. When the size of *C* is polynomial in *k*, checking each one takes Ω(2*k*) time, which is superpolynomial in the size of the circuit.  In fact, as has been claimed, there is strong evidence that no polynomial-time algorithm exists that solves the circuit-satisfiability problem because circuit satisfiability is NP-complete.

***Lemma 1: The circuit-satisfiability problem belongs to the class NP*** [3]***.***

***Proof***  We shall provide a two-input, polynomial-time algorithm *A* that can verify CIRCUIT-SAT. One of the inputs to *A* is (a standard encoding of) a boolean combinational circuit *C*. The other input is a certificate corresponding to an assignment of boolean values to the wires in *C*.

The algorithm *A* is constructed as follows. For each logic gate in the circuit, it checks that the value provided by the certificate on the output wire is correctly computed as a function of the values on the input wires. Then, if the output of the entire circuit is 1, the algorithm outputs 1, since the values assigned to the inputs of *C* provide a satisfying assignment. Otherwise, *A* outputs 0.

Whenever a satisfiable circuit *C* is input to algorithm *A*, there is a certificate whose length is polynomial in the size of *C* and that causes *A* to output a 1. When-ever an unsatisfiable circuit is input, no certificate can fool *A* into believing that the circuit is satisfiable. Algorithm *A* runs in polynomial time: with a good implementation, linear time suffices. Thus, CIRCUIT-SAT can be verified in polynomial time, and CIRCUIT-SAT ∈ NP.

The second part of proving that CIRCUIT-SAT is NP-complete is to show that the language is NP-hard. That is, we must show that every language in NP is polynomial-time reducible to CIRCUIT-SAT. The actual proof of this fact is full of technical intricacies, and so we shall settle for a sketch of the proof based on some understanding of the workings of computer hardware.

A computer program is stored in the computer memory as a sequence of instructions. A typical instruction encodes an operation to be performed, addresses of operands in memory, and an address where the result is to be stored. A special memory location, called the *program counter*, keeps track of which instruction is to be executed next. The program counter is automatically incremented whenever an instruction is fetched, thereby causing the computer to execute instructions sequentially. The execution of an instruction can cause a value to be written to the program counter, however, and then the normal sequential execution can be altered, allowing the computer to loop and perform conditional branches.

At any point during the execution of a program, the entire state of the computation is represented in the computer's memory. (We take the memory to include the program itself, the program counter, working storage, and any of the various bits of state that a computer maintains for bookkeeping.) We call any particular state of computer memory a configuration. The execution of an instruction can be viewed as mapping one configuration to another. Importantly, the computer hardware that accomplishes this mapping can be implemented as a boolean combinational circuit, which we denote by *M* in the proof of the following lemma.

***Lemma 2: The circuit-satisfiability problem is NP-hard*** [4]***.***

***Proof***  Let *L* be any language in NP. We shall describe a polynomial-time algorithm *F* computing a reduction function *f* that maps every binary string *x* to a circuit *C* = *f* (*x*) such that *x* ∈ *L* if and only if *C* ∈ CIRCUIT-SAT. Since *L* NP, there must exist an algorithm *A* that verifies *L* in polynomial time. The algorithm *F* that we shall construct will use the two-input algorithm *A* to compute the reduction function *f* . Let *T*(*n*) denote the worst-case running time of algorithm *A* on length-*n* input strings, and let *k* ≥ 1 be a constant such that *T* (*n*) = *O*(*nk*) and the length of the certificate is *O*(*nk*). (The running time of *A* is actually a polynomial in the total input size, which includes both an input string and a certificate, but since the length of the certificate is polynomial in the length *n* of the input string, the running time is polynomial in *n*.)

The basic idea of the proof is to represent the computation of *A* as a sequence of configurations. As shown in Figure, each configuration can be broken into parts consisting of the program for *A*, the program counter and auxiliary machine state, the input *x*, the certificate *y*, and working storage. Starting with an initial configuration *c*0, each configuration *ci* is mapped to a subsequent configuration *ci*+1 by the combinational circuit *M* implementing the computer hardware. The output of the algorithm *A*-0 or 1-is written to some designated location in the working storage when *A* finishes executing, and if we assume that thereafter *A* halts, the value never changes. Thus, if the algorithm runs for at most *T*(*n*) steps, the output appears as one of the bits in *cT*(*n*).

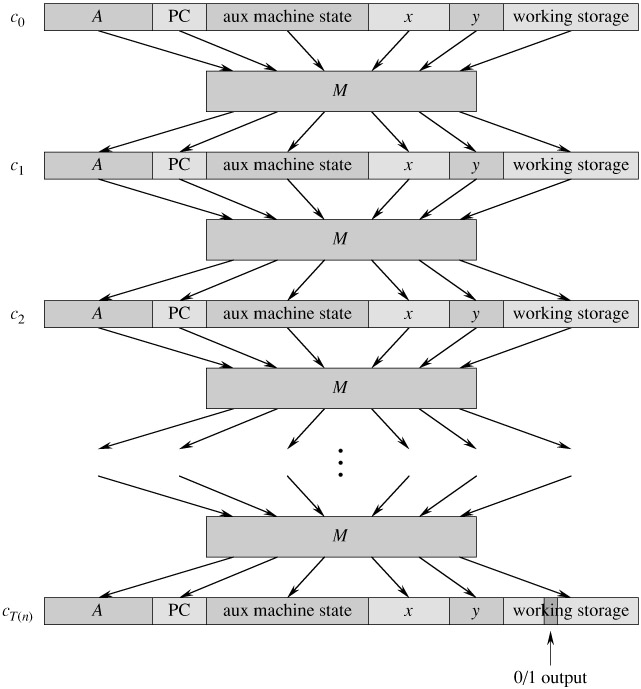


Figure 2.9: The sequence of configurations produced by an algorithm

A running on an input *x* and certificate *y*. Each configuration represents the state of the computer for one step of the computation and, besides *A, x*, and *y*, includes the program counter (PC), auxiliary machine state, and working storage. Except for the certificate *y*, the initial configuration *c*0 is constant. Each configuration is mapped to the next configuration by a boolean combinational circuit *M*. The output is a distinguished bit in the working storage.

The reduction algorithm *F* constructs a single combinational circuit that computes all configurations produced by a given initial configuration. The idea is to paste together *T*(*n*) copies of the circuit *M*. The output of the *i*th circuit, which produces configuration *ci*, is fed directly into the input of the (*i* +1)st circuit. Thus, the configurations, rather than ending up in a state register, simply reside as values on the wires connecting copies of *M*.

Recall what the polynomial-time reduction algorithm *F* must do. Given an input *x*, it must compute a circuit *C* = *f*(*x*) that is satisfiable if and only if there exists a certificate *y* such that *A*(*x*, *y*) = 1. When *F* obtains an input *x*, it first computes *n* = |*x*| and constructs a combinational circuit *C*′ consisting of *T*(*n*) copies of *M*. The input to *C*′ is an initial configuration corresponding to a computation on *A*(*x*, *y*), and the output is the configuration *cT*(*n*).

The circuit *C* = *f*(*x*) that *F* computes is obtained by modifying *C*′ slightly. First, the inputs to *C*′ corresponding to the program for *A*, the initial program counter, the input *x*, and the initial state of memory are wired directly to these known values. Thus, the only remaining inputs to the circuit correspond to the certificate *y*. Second, all outputs to the circuit are ignored, except the one bit of *cT*(*n*) corresponding to the output of *A*. This circuit *C*, so constructed, computes *C*(*y*) = *A*(*x*, *y*) for any input *y* of length *O*(*nk*). The reduction algorithm *F*, when provided an input string *x*, computes such a circuit *C* and outputs it.

Two properties remain to be proved. First, we must show that *F* correctly computes a reduction function *f* . That is, we must show that *C* is satisfiable if and only if there exists a certificate *y* such that *A*(*x*, *y*) = 1. Second, we must show that *F* runs in polynomial time.

To show that *F* correctly computes a reduction function, let us suppose that there exists a certificate *y* of length *O*(*nk*) such that *A*(*x*, *y*) = 1. Then, if we apply the bits of *y* to the inputs of *C*, the output of *C* is *C*(*y*) = *A*(*x*, *y*) = 1. Thus, if a certificate exists, then *C* is satisfiable. For the other direction, suppose that *C* is satisfiable. Hence, there exists an input *y* to *C* such that *C*(*y*) = 1, from which we conclude that *A*(*x*, *y*) = 1. Thus, *F* correctly computes a reduction function.

To complete the proof sketch, we need only show that *F* runs in time polynomial in *n* = |*x*|. The first observation we make is that the number of bits required to represent a configuration is polynomial in *n*. The program for *A* itself has constant size, independent of the length of its input *x*. The length of the input *x* is *n*, and the length of the certificate *y* is *O*(*nk*). Since the algorithm runs for at most *O*(*nk*) steps, the amount of working storage required by *A* is polynomial in *n* as well.

The combinational circuit *M* implementing the computer hardware has size polynomial in the length of a configuration, which is polynomial in *O*(*nk*) and hence is polynomial in *n*. (Most of this circuitry implements the logic of the memory system.) The circuit *C* consists of at most *t* = *O*(*nk*) copies of *M*, and hence it has size polynomial in *n*. The construction of *C* from *x* can be accomplished in polynomial time by the reduction algorithm *F*, since each step of the construction takes polynomial time.

### 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, we shall show how to prove that languages are NP-complete without directly reducing *every* language in NP to the given language. We shall illustrate this methodology by proving that various formula-satisfiability problems are NP-complete.

***Lemma 3* *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, we have *L*′′≤P *L*′. By supposition, *L*′ ≤P *L*, and thus by transitivity, we have *L*′′ ≤P *L*, which shows that *L* is NP-hard. If *L* ∈ NP, we also have *L* ∈ NPC.

In other words, by reducing a known NP-complete language *L*′ to *L*, we implicitly reduce every language in NP to *L*. Thus, [Lemma 3](http://www.euroinformatica.ro/documentation/programming/!!!Algorithms_CORMEN!!!/DDU0231.html#ch34ex33) 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 we develop a catalog of known NP-complete problems, we will have more and more choices for languages from which to reduce.

### Formula Satisfiability:

When we are unable to solve the exponential time consuming problems in polynomial time, we can show 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, we have to show association between them 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 we need some problem as base problem called Formula Satisfiability also called Boolean Satisfiability Problem.

We formulate 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, we assume 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.

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*〉.

|  |
| --- |
|  |

# 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*** [6]***.***

**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 we have to use some boolean variables:

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)

We have 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)

We have 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)

We have 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.

*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 we have 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 , we modify and as follows. We select graphs 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.

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

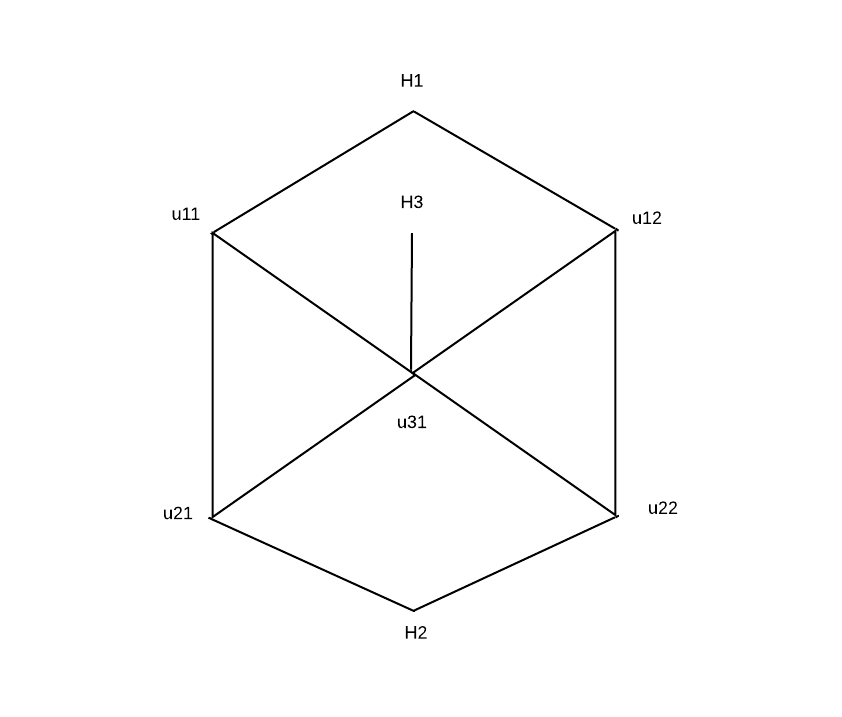
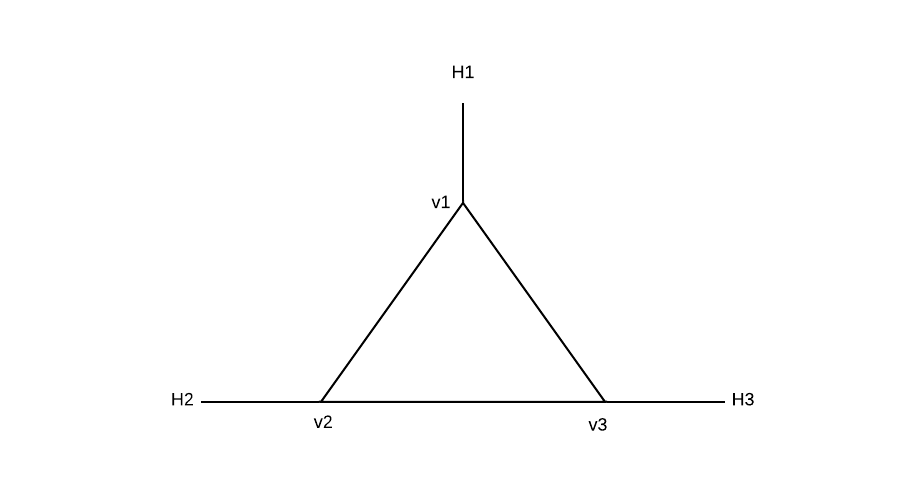


Figure 3.2: G2'

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