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

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 to certify that the seminar report prepared by **Mr. Himal Raj Gentil** 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
Head of Central Department of Computer Science & Information Technology

……………………………..

Asst. Prof. Ram Krishna Dahal
(Supervisor)
Central Department of Computer Science & Information Technology

..…………………………………

(Internal)

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. These 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 mystery 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 elaborated version of theorems are presented. Polynomial time reduction and verification of NP-complete problems are discussed in detail too.

Keywords: Computational Complexity, Computational Classes, Millennium Prize Problem, Boolean Satisfiability.

Table of Contents

List of Figure

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

1. INTRODUCTION

1.1 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 $n \rightarrow f(n)$ where, n is the size of the input and $f(n)$ 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 problem3. [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. This seminar report gives the general idea of reducing the time complexity of an algorithm using different methodologies which are further discussed in detail.

1.2 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 n 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 n 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 n. The best, average and worst time complexity are numerical function representing time versus problem size such as $f(x) = x^2 + 3x - 1$ where x 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 n, where n tends to infinity i.e. asymptotic behavior of the complexity for large n, when n tends to infinity i.e. asymptotic behavior of the complexity which is expressed by big oh (O) notation.

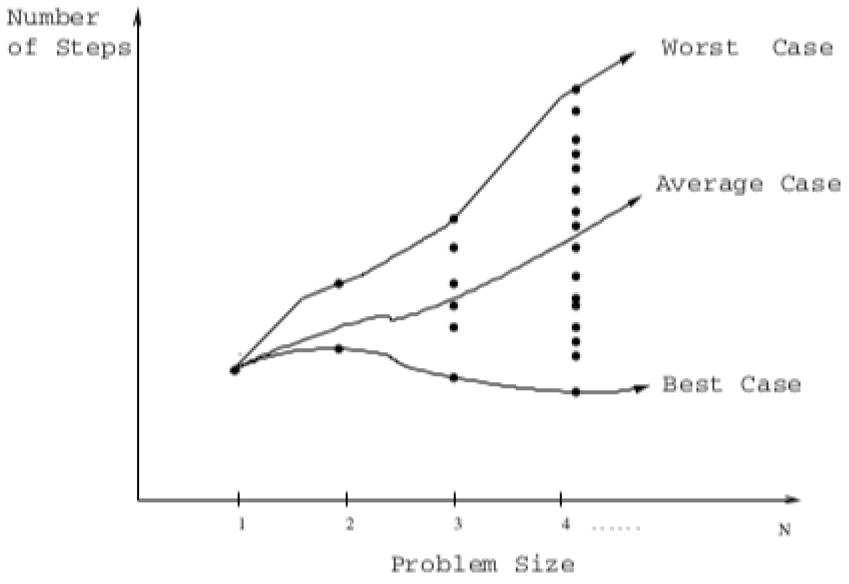


Figure 1.1 : Asymptotic complexity [3]

For example, let us take a function f(n) such as $f(n) = c * n$ and $f(n) = c * n^2 + k$. Here, $f(n)$ 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 $f(n) = c * n$ (best case) is always better than quadratic asymptotic function $f(n) = c * n^2 + k$ (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].

1.3 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 $f(n)$ steps always finishes in $f(n)$ steps and always returns the same result. Non-deterministic algorithm has quite different behavior. They contain $f(n)$ 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]

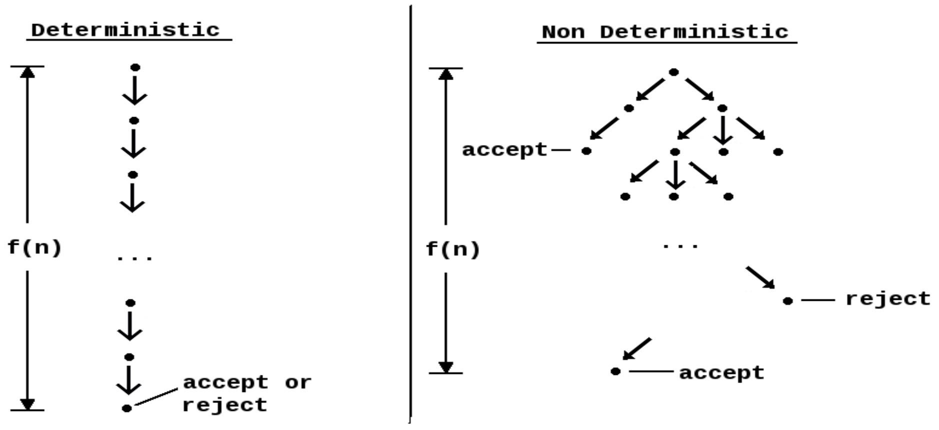


Figure 2.1: Determinism and Non-determinism in algorithms[6]

1.3.1 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". [1]

In a 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 $M = (Q, \Gamma, b, \Sigma, d, q_0, F)$ where

- Q is a finite, non-empty set of *states*;
- Γ is a finite, non-empty set of *tape alphabet symbols*;
- $b \in \Gamma$ is the *blank symbol* (the only symbol allowed to occur on the tape infinitely often at any step during the computation);
- $\Sigma \subseteq \Gamma \setminus \{b\}$ is the set of *input symbols*, that is, the set of symbols allowed to appear in the initial tape contents;
- $q_0 \in Q$is the *initial state*;
- $F \subseteq Q$ is the set of *final states* or *accepting states*. The initial tape contents is said to be *accepted* by Mif it eventually halts in a state from F.
- $\delta: \left(\frac{Q}{F}\right) * \Gamma \rightarrow Q * \Gamma * \{L, R\}$ is a partial function called the *transition function*, where L is left shift, R is right shift. If δ 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 .

The 7-tuple for the 3-state busy beaver looks like this :

- $Q = \{A, B, C, HALT)$ (states);
- $\Gamma = \{0,1\}$ (tape alphabet symbols);
- $b = 0$ (blank symbol);
- $\Sigma = \{0,1\}$ (input symbols);
- $q_0 = a$ (initial state);
- $F = \{HALT\}$ (final states);
- $\delta = \mathrm{Q} * \Gamma * \{\mathrm{L}, \mathrm{R}\}$ (transition function).

Initially all tape cells are marked with 0.

1.3.2 Non Deterministic Models

In a non-deterministic model of computation, such as non-deterministic Turing machines, 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 can 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" 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. [1]

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

- $Q$is a finite set of states
- Σ is a finite set of symbols (the tape alphabet)
- $\iota \in Q$ is the initial state
- $\sqcup \in \Sigma$ is the blank symbol
- $A \in Q$ is the set of accepting (final) states
- $\delta \in (Q \backslash A * \Sigma) * (Q * \Sigma * \{L, S, R\}$ is a relation on states and symbols called the transition relation. L is the movement to the left, S is no movement, and R is the movement to the right.[7]

The difference with a standard (deterministic) 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 at least one of the possible computational paths starting from that string puts the machine into an accepting state, at such condition an input is accepted by NTM. An NTM's operations are stopped as soon as any branch reaches an accepting state.

1.4 Problem Definition and Scope

Before publishing the research paper named "The Theorem of Complexity Proving Procedures" by Stephen A. Cook in 1971, whole theoretic computer scientists and mathematicians were unaware of computational complexity class called NP-completeness of a problem. The computational complexity field was only known to P, NP and NP-Hard problems. Theoretic computer scientists and mathematicians and were trying to find a polynomial time solution for each individual NP and NP-Hard problems so that non-polynomial time consuming problems could be solved in polynomial time. The dilemma of whether P=NP or not was present but there were no any clue or possible procedure of determining it. Although P=NP or not remains unsolved today, but the possible procedure of way of proving it is known today.

After publishing "The Theorem of Complexity Proving Procedures" where it is shown that Boolean Satisfiability Problem is NP-complete class problem led to the invention of NP-complete computational complexity classes. Due to the invention NP-complete class whole computational complexity theory revolutionized and due to the properties of NP-complete class, possible ideas about marching towards proving whether P=NP or not came. It also can be said that possibilities of way of proving P=NP was developed due to the existence of NP-complete class.

1.5 Objective and Limitation

The objective of this seminar report is to show how a NP problem can be reduced to a NP-complete problem in polynomial time and also talk about the possibilities and consequences of converting NP-complete problem to P-class problem. It helps in building the overall general idea about the computational complexity classes and the special characteristics possessed NP-complete problem which might be useful in proving whether P=NP or not.

Although, the process reduction of NP class problem to NP-complete problem is discussed in detail, with the help theoretic illustration of non-deterministic computational machine but this seminar report lacks the real implementation of reduction process. Possibilities of reduction of NP-complete problems to P class problems are discussed only but there is not any standard procedure and implementation illustrated in this seminar report.

2. LITERATURE REVIEW

In 1936, Alan Turing developed his theoretical computational model which was based on how he perceived mathematicians think. As digital computers were developed in the 40's and 50's, the Turing Machine proved itself the right theoretical model for computer. But quickly, it was discovered that the Turing Machine model fails to account for the amount of time and memory needed by the computer which is considered critical issue today and was even more in those early days of computing. The key idea to measure time and space as a function of length of the input came in early 1960's by Hartmanis and Stearns. And thus, the computational complexity was born.

In those early days of complexity, researchers just tried understanding these new measures and how they related with each other. P is the first notation of efficient computation by the time polynomial in the input size. This lead to complexities more important concept, NP-completeness and the most fundamental question, whether P=NP.The existence of NP-complete problems was proved, independently by Stephen Cook in the United States and Leonid Levin in the Soviet Union. Cook prove that the Satisfiability problem is NP-complete. Levin, a student of Kolmorov in Moscow State University proved that the variant of the tiling problem is NP-complete.

The work of Cook and Karp in the early 70's showed that a large number of combinatorial and logical problems were NP-complete i.e. as hard as any problem computable in non-deterministic polynomial time. The P=NP question is equivalent to an efficient solution of any of these problems. In the thirty year hence, this problem has become one of the outstanding open questions in computer science and indeed all mathematics. In the 70's the complexity classes were grown as researcher try to encompass different models of computation, such as probabilistic complexity classes were developed.

In the 80's, the rise of finite models like circuit that capture computation in an inherently different way. A new approach to problem like P=NP arose from those circuits and though they have had limited success in separating complexity class, this approach brought combinatorial techniques into the area and led to much better understanding of the limit of these devices.

In the 90's study of new model of computation like quantum computer and propositional proof system started. Tools from the past have greatly helped an understanding of these new areas and till now research in computational complexity is going on. [12]

3. COMPUTATIONAL COMPLEXITY CLASSES

3.1 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 to study these problems and quantifying their 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), the number of gates in a circuit (used in circuit complexity) and the number of processors (used in 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, one of the seven Millennium Prize Problems, is dedicated to the field of computational complexity.[8]

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.

3.2 P class

An algorithm A is of polynomial complexity if there exists a polynomial $P(\ )$ such that computing time of A is $O(P(n))$ or every input size n. 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 $O(n^k)$ or some non-negative integer k , where n is the size of the input. It contains all decision problems that can be solved by a deterministic Turing machine using a polynomial amount of computation time, or polynomial time. [9]

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

$$P = \cup_k \ TIME(n^k) \text{ [1]}$$

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 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, 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 L is in P if and only if there exists a deterministic Turing machine M, such that

- M runs for polynomial time on all inputs
- For all x in L, M outputs 1
- For all x not in L, M outputs 0 [1]

Sorting algorithm usually require $O(n\ log\ n)$ or $O(n\text{^}2)$ time. Bubble sort takes linear time in the best case, but $O(n\text{^}2)$ in average and worst case. Heap sort takes $O(n\ log\ n)$ in all cases. Quick sort takes $O(n\ log\ n)$ on average and $O(n\text{^}2)$ in worst case.

3.3 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 $P \subseteq NP$. It is unknown and what has become perhaps the most unsolved problem in computer science is whether $P = NP$ or $P \neq NP$. 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 (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 problems. An algorithm solving such a problem in polynomial time is also able to solve any other NP problem in polynomial time. [9]

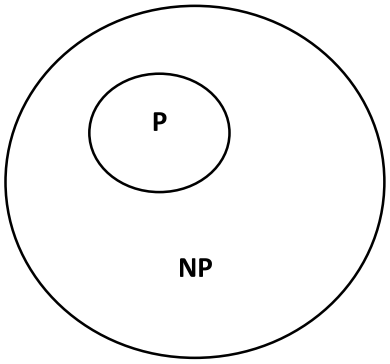


Figure 3.3:Commonly believed relationship between P and N [9]

The complexity class NP can be defined in terms of NTIME as follows:

$$NP = \bigcup_{k \in N} NTIME(n^k)$$

where $NTIME(n^k)$ is the set of decision problems that can be solved by a non-deterministic Turing machine in $O(n^k)$ time.

Alternatively, NP can be defined using deterministic Turing machines as verifiers. A language L is in NP if and only if there exist polynomials p and q, and a deterministic Turing machine M, such that

- For all x and y, the machine M runs in time $p(|x|)$ on input (x, y)
- For all x in L, there exists a string y of length $q(|x|)$ such that $M(x, y) = 1$
- For all x not in L and all strings y of length $q(|x|)$, $M(x, y) = 0$ [1]

Boolean satisfiability problem, Knapsack problem, Vertex cover problem, Subgraph isomorphism problem are some of the NP-class problems which are solved in $NTIME(n^k)$.

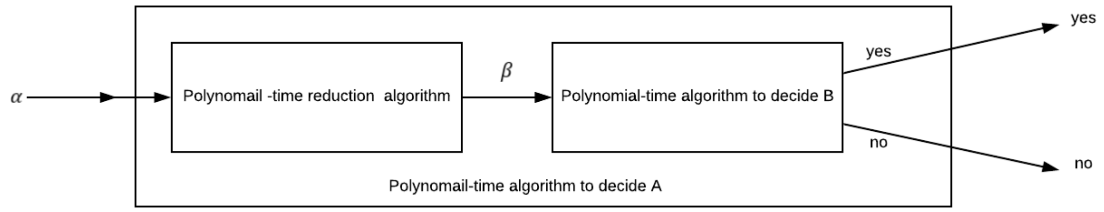
3.4 Overview of showing problems to be NP-complete

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.

3.4.1 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 G and vertices u and v, and it is wished to find the path rom u to v 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 α of A into some instance β of B with the following characteristics:

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



α
Polynomail -time reduction algorithm
β
Polynomial-time algorithm to decide B
yes
no
yes
no
Polynomail-time algorithm to decide A

Figure 3.4: Reduction process[10]

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

As long as each of these steps takes polynomial time, all three together do also and so there is a way to decide on α in polynomial time. In other words, by "reducing" solving problem B, "easiness" of B is used to prove the "easiness" of A. [10]

3.4.2 A formal-language framework

An alphabet Σ is a finite set of symbols. A language L over Σ is any set of strings made up of symbols from Σ. For example, if Σ = {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 ε, and the empty language by ϕ. The language of all strings over Σ is denoted by Σ^*. For example , if $\Sigma = \{0,1\}$, then Σ^*={ ∈,0,1,00,01,10,11,000,….} is the set of all binary strings. Every language L over Σ is a subset of Σ^*.

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 $x \in \{0,1\}^*$ if, given input x, the algorithm's output A(x) is 1. The language accepted by an algorithm A is the set of strings $L = \{x \in \{0,1\}^*: A(x) = 1\}$ i.e. the set of strings that the algorithm accepts. An algorithm A rejects a string x if $A(x) = 0$. 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 $x \in L$, algorithm A accepts x in $O(n^k)$. 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 $x \in \{0,1\}^*$, the algorithm correctly decides whether $x \in L$ in time $O(n^k)$. Thus, to accept a language, an algorithm need only worry about strings in L, but to decide a language, it must correctly accept or reject every string in $\{0,1\}^*$. [10]

3.5 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.

3.5.1 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 $G = (V, E)$ 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.

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 reasonable encoding of graph is used as its adjacency matrix, the number m of vertices in the graph is $\Omega(\sqrt{n})$, where $n = |(G)|$ is the length of the encoding of G. There are m! possible permutations of the vertices, and therefore the running time is $\Omega(\mathrm{m!}) = \Omega(\sqrt{\mathrm{n}}!) = \Omega(2^{\sqrt{n}})$, which is not $O(n^k)$ for any constant k.' [10]

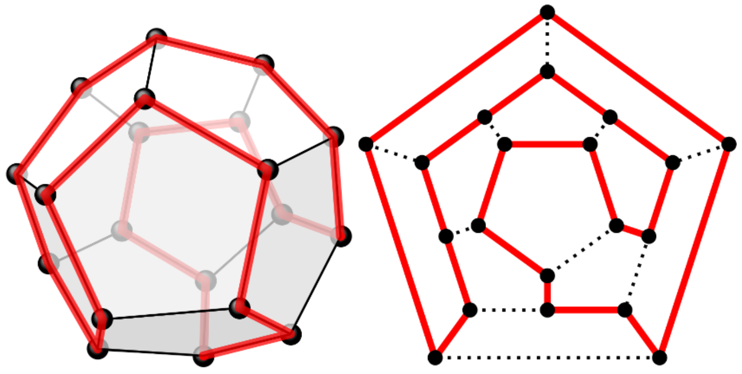


Figure 3.5: One possible Hamiltonian cycle through every vertex of a dodecahedron[10]

3.5.2 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 A verifies an input string x if there exists a certificate y such that $A(x, y) = 1$. The language verified by a verification algorithm A is $L = \{x \in \{0,1\}^* : \textit{there exists } y \in \{0,1\}^* \textit{ such that } A(x, y) = 1\}$.

Intuitively, an algorithm A verifies a language L if for any string x $\in L$. Moreover, for any string $x \notin L$, there must be no certificate proving that $x \in L$. 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 of vertices V and whether each of the consecutive edges along the cycle exists in the graph. This verification algorithm can be certainly implemented to run in $O(n^2)$ time, where n is the length of encoding of G.

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

$$L = \{x \in \{0,1\}^* : \textit{there exists a certificate } y \textit{ with } |y| = O(|x|^c) \textit{ such that } A(x, y) = 1\}$$

which means that algorithm A verifies language L in polynomial time. [10]

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

3.6 NP-completeness and reducibility

Perhaps the most compelling reason why theoretical computer scientists believe that $P \neq NP$ 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. [10]

3.6.1 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 $ax + b = 0$, it could be transformed to $0x^2 + ax + b = 0$, whose solution provides a solution to $ax + b = 0$. Thus, 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 L_1 is polynomially reducible to a language L_2, written $L_1 \leq_p L_2$ if there exists a polynomial-time computable function $f: \{0,1\}^* \rightarrow \{0,1\}^*$ such that for all $x \in \{0,1\}^*$, $x \in L_1$ if and only if $f(x) \in L_2$.

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

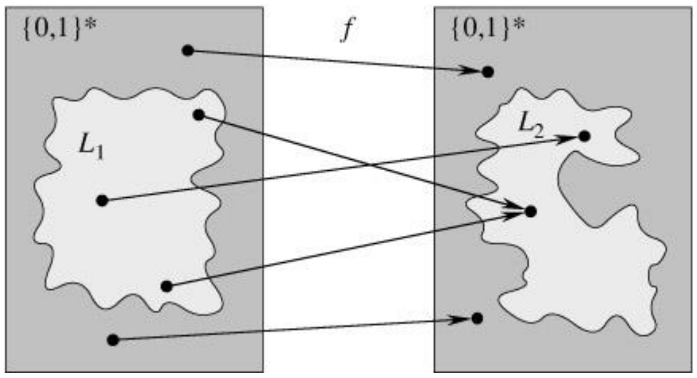


Figure 3.6: Polynomial Time Reduction[10]

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 $x \in \{0, 1\}^*$, the question of whether $x \in L_1$ has the same answer as the question of whether $f(x) \in L_2$. Each language is a subset of $\{0, 1\}^*$. The reduction function *f* provides a polynomial-time mapping such that if $x \in L_1$, then $f(x) \in L_2$. Moreover, if $x \notin L_1$, then $f(x) \notin L_2$. 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 $f(x) \in L_2$ directly provides the answer to whether $x \in L_1$.

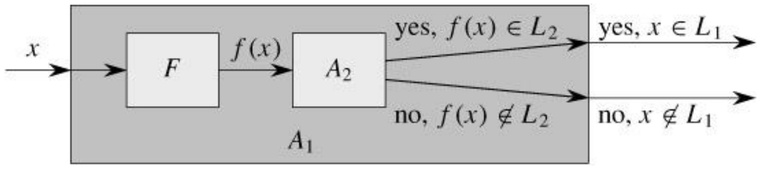


Figure 3.7: Reduction Algorithm[10]

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 $x \in L_1$ by using F to transform any input *x* into $f(x)$ and then using A_2 to decide whether $f(x) \in L_2$. [10]

3.6.2 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 $L_1 \leq_P L_2$, 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 $L \subseteq \{0, 1\}^*$ is *NP-complete* if

1. $L \in$ NP, and
2. $L' \leq_P L$ for every $L' \in$ 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. [10]

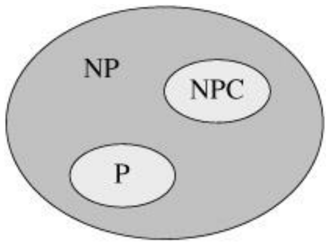


Figure 3.8: Relationship between P, NP, and NPC [10]

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

3.6.3 NP-completeness proofs

The NP-completeness of the formula satisfiability problem relies on a direct proof that $L \leq_P$ SAT for every language $L \in$ 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 $L' \leq_P L$ for some $L' \in$ NPC, then L is NP-hard. Moreover, if $L \in$ NP, then $L \in$ NPC. [5]

Proof : Since L' is NP-complete, for all $L'' \in$ NP, then $L'' \leq_P L'$. By supposition, $L' \leq_P L$, and thus by transitivity, $L'' \leq_P L$, which shows that L is NP-hard. If $L \in$ NP, then $L \in$ 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 1 gives us a method for proving that a language L is NP-complete:

1. Prove $L \in$ NP.
2. Select a known NP-complete language L'.
3. Describe an algorithm that computes a function f mapping every instance $x \in \{0, 1\}^*$ of L' to an instance $f(x)$ of L.
4. Prove that the function f satisfies $x \in L'$ if and only if $f(x) \in L$ for all $x \in \{0, 1\}^*$.
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 SAT $\in$ NPC has given us a "foot in the door." Knowing that the 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. [10]

3.6.4 Formula Satisfiability

When the exponential time consuming problems cannot be solved 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: $x_1, x_2, ..., x_n$;
2. *m* boolean connectives: any boolean function with one or two inputs and one output, such as $\wedge$ (AND), $\vee$ (OR), $\neg$ (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. [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: $(x1 \wedge \neg \text{x2} \wedge x3) \vee (\neg \text{x1} \wedge x2 \wedge \neg x3) \vee (\neg \text{x4})$

3-DNF is formed by clauses having only three distinct variable.
For example: $(x1 \wedge \neg \text{x2} \wedge x3) \vee (\neg \text{x1} \wedge x2 \wedge \neg x3)$

CNF is formed by Anding of clauses of OR's.
For example: $(x1 \vee \neg \text{x2} \vee x3) \wedge (\neg \text{x1} \vee x2 \vee \neg x3) \wedge (\neg \text{x4})$

3-CNF is formed by clauses having only three distinct variable.
For example: $(x1 \vee \neg \text{x2} \vee x3) \wedge (\neg \text{x1} \vee x2 \vee \neg x3)$

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

SAT = { $\langle \varphi : \varphi$ is a satisfiable boolean formula}.

As an example, the formula
$\varphi = (x1 \vee \neg \text{x2} \vee x3) \wedge (\neg \text{x1} \vee x2 \vee \neg x3)$
has the satisfying assignment $\langle x_1 = 1, x_2 = 0, x_3 = 0 \rangle$
$\varphi = (1 \vee \neg 0 \vee 0) \wedge (\neg 1 \vee 0 \vee \neg 0)$

$=(1 \vee 1 \vee 0) \wedge (0 \vee 0 \vee 1)$

$=(1) \wedge (1)$

$=1$

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 $\langle\varphi\rangle$ is polynomial in n, then checking every assignment requires $\Omega(2^n)$ time, which is superpolynomial in the length of $\langle\varphi\rangle$. [9]

4. 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 $\propto$ L . [11]

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

$$M -> P(n)$$

Given, $M, w -> w_0$ where w=Input string and w_0=(DNF Boolean Expression i.e. Output)

i.e. M and w write w_0 such that w_0 is satisfiable if and only if M accepts w.

w_0 can be constructed in polynomial time from M and w.

M has $\{q_1,q_2,q_3,\ldots\ldots.q_s\}$ as state set.

$\Gamma = \{x_1,x_2,x_3,\ldots\ldots\ldots.x_m\}$ as the input symbols where x_1=blank symbol

Id's of the turing machine $M = Q_0 \vdash Q_1 \vdash Q_2 \vdash \cdots \ldots \ldots .. \vdash Q_q$

After some states at Q_q, M will accept or reject.

$$Q_0 \vdash Q_1 \vdash Q_2 \vdash \cdots \ldots \ldots .. \vdash Q_q \vdash Q_{q+1} \ldots \ldots \vdash Q_{P(n)}$$
$$q \le P(n)$$

In-order to find expression w_0 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:
   $i => 1 \le i \le P(n)$

   $j => 1 \le j \le m$

   $t => 1 \le t \le P(n)$

   There are $O(P^2(n))$ variables of this form.

2. H< i, t, > = 1 if the head is scanning ith cell at time t, Otherwise 0.
   $i => 1 \le i \le P(n)$
   $t => 1 \le t \le P(n)$

   There are $O(P^2(n))$ variables of this form.

3. S< k, t > = 1 if the state is q_k at time t.
   $k => 1 \le k \le s$
   $t => 1 \le t \le P(n)$

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

w_0 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: $\cup(x_1,x_2,\ldots\ldots\ldots x_r) = (x_1 + x_2 + x_3\ldots\ldots + x_r) \prod_{i\neq j} (\neg x_i + \neg x_j)$ => 1 iff exactly one of x1……..xr = 1 others 0

⇨ Length of the expression r^2

For condition 1:

$A_t = \cup\,(H< 1, t >, H< 2, t >, \ldots\ldots\ldots\ldots H< P(n), t>)$

$A=A_0 \,.\, A_1 .\, A_2 \,.\, \ldots\ldots\ldots\ldots \,.\, A_{P(n)}$

Length of A = $O(P^3(n))$

For condition 2:

$B=\prod_{i,j} B\ i,t$

$B_{i,t} = \cup(\ C< i, 1, t >, C< i, 2, t >, \ldots\ldots\ldots\ldots C<i, m, t>\ )$

$B=B_0\ B_1\ldots\ldots\ldots\ldots B_{P(n)}$

Length of B = $O(p^2(n))$

For condition 3:

$S_t = \cup(S< 1 , t >, S< 2, t >, \ldots\ldots\ldots\ldots\ldots S< 3, t >)$

$C=S_0\ S_1\ldots\ldots\ldots\ldots S_{P(n)}$

Length of $S_t = O(P(n))$

For condition 4:

$\prod_{i,j,t}(C < i, j, t > \equiv C < i, j, t+1 > +H < i, t >)$

$D = C_0\ C_1 \ldots\ldots\ldots\ldots C_{P(n)}$

Length of D = $O(p^2(n))$

For condition 5:

$E_{i,j,k,t} = C(< i, j_l, t+1)H < i_l, t+1 >$

$E = E_0\ E_1 \ldots\ldots\ldots\ldots E_{P(n)}$

Length of E = $O(p^2(n))$

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 w_o = ABCDEF = $O(p^3(n))$ which is polynomial time.

This completes the proof of theorem 1.

If L' $\in$ NP i.e. L' is accepted by NTM M then L' $\propto$ L (Boolean Satisfiability Problem)

This also shows that Boolean Satisfiability Problem is NP-Complete. [11]

Illustration:

Suppose, there is DNF formula satisfiability problem:

$DNF = (x1 \wedge \neg x2 \wedge x3)$ such that $x_i = \{x1, x2, x3\}$

The problem is to guess values for x_i which leads the 3-CNF formula to an accepting state. There are 8 i.e. 2^3 possible combinations that may satisfy 3-CNF.

Similarly, for n variables $=2^n$ possible combinations which takes exponential time.

Again let us consider another problem called $0/1$ $knapsack$ $problem$ as:

$n = 3$

$P = \{10,8,12\}$

$W = \{5,4,3\}$

$x_i = \{\frac{0}{1}, \frac{0}{1}, \frac{0}{1}\}$

The possible solutions for both problems can be represented by same state space tree as:

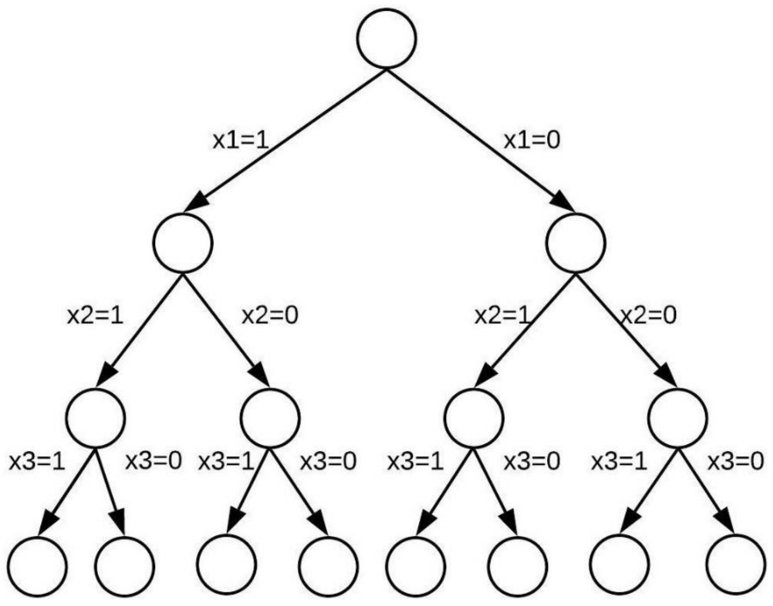


Figure 4.1 State Space Tree

At such conditions, when same solution strategy works for both the problems then reduction could be done. Here, knapsack problem can be reduced to DNF as ($x1 \wedge \neg x2 \wedge x3$).

This also proves that DNF is NP-Complete problem, since DNF is in NP and every problem for instance (Knapsack) is reduced to DNF.

Theorem 2: DNF tautologies is P-reducible to D_3 and D_3 is P-reduicible to Subgraph Pairs. [11]

Proof

To show DNF tautologies is P-reduicible to D_3, let A be a proposition formula in disjunctive normal form. Say $A = B_1 \vee B_2 \vee \ldots \vee B_k$, where$B_1 = R_1 \wedge \ldots\ldots \wedge R_s$, each R_i , is an atom or negation of atom, and $s > 3$. Then A is a tautology I and only if A' is a tautology where

$$A' = P \wedge R_3 \wedge \ldots \wedge R_s \vee \neg P \wedge R_1 \wedge R_2 \vee B_2 \vee \ldots \vee B_k,$$

Where P is a new atom. Since it has reduced the number of conjuncts in B_1, 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 A.

It remains to show that D_3 is P-reducible to subgraph pairs. Suppose A is formula in disjunctive normal form with three conjuncts per disjunct. Thus $A = C_1 \vee \ldots \vee C_k$, where $C_i = R_{i1} \wedge R_{i2} \wedge R_{i3}$, and each R_{ij} is an atom or negation of an atom. Now let G_1 be the complete graph with vertices $\{v_1, v_2, \ldots\ldots v_k\}$, and let G_2 be the graph with vertices $\{u_{ij}\}$, $1 \leq i \leq k$, $1 \leq j \leq 3$, such that u_{ij} is connected by an edge to u_{rs} if and only if $i \neq r$, and the two literals (R_{ij}, R_{rs}) do not form an opposite pair (that is they are neither of the form $(P, \neg P)$ nor of the form $(\neg P, P)$). Thus there is a alsiying truth assignment to the formula A iff there is a graph homomorphism $\emptyset: G_1 \rightarrow G_2$ such that for each i, $\emptyset(i) = u_{ij}$ for some j. (The homomorphism tells for each i which of R_{i1}, R_{i2}, R_{i3} should be falsified, and the selective lack of edges in G_2 guarantees that the resulting truth assignment in consistently specified.)

In order to guarantee that a one-one homomorphism $\emptyset: G_1 \rightarrow G_2$ has the property that for each i, $\emptyset(i) = u_{ij}$ for some j, $G_1$and G_2 are modiied as follows. Graphs $H_1, H_2 \ldots H_k$ are selected which are sufficiently distinct from each other that if G'_1 is formed from G_1 by attaching H_i to v_i, $1 \leq i \leq k$, and G'_2 is formed from G_2 by attaching H_i to each of u_{i1} *and* u_{i2} *and* u_{i3}, $1 \leq i \leq k$, then every one-one homomorphism $\emptyset: G'_1 \rightarrow G'_2$ has the property just stated. It is not hard to see such a construction csn be carried out in polynomial time. Then G'_1 can be embedded in G'_2 if and only if A $\notin D_3$. This completes the proof of theorem 2. [11]

Illustration*: D_3 *reducible to subgraph pairs.

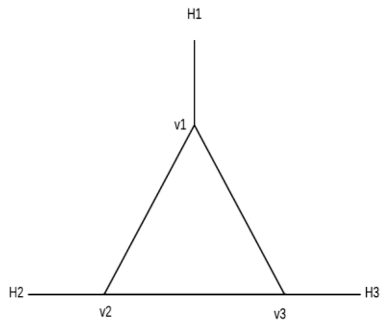


Figure 4.2: G1'

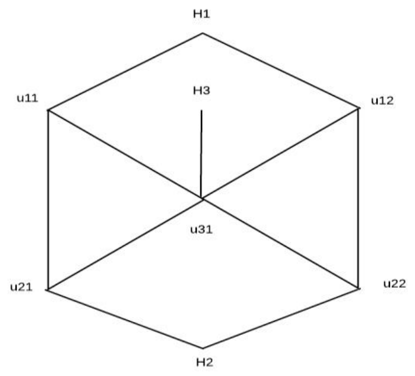


Figure 4.3: G2'

Adjacent Matrix for G_2':

	u_{11}	u_{12}	u_{21}	u_{22}	u_{31}	H_1	H_2	H_3
u_{11}	0	0	1	0	1	1	0	0
u_{12}	0	0	0	1	1	1	0	0
u_{21}	1	0	0	0	1	0	1	0
u_{22}	0	1	0	0	1	0	1	0
u_{31}	1	1	1	1	0	0	0	1
H_1	1	1	0	0	0	0	0	0
H_2	0	0	1	1	0	0	0	0
H_3	0	0	0	0	1	0	0	0

$\emptyset: G_1 \rightarrow G_2$ such that for each i, $\emptyset(i) = u_{ij}$ for some j :

$u_{11} = \neg u_{11} \wedge \neg u_{12} \wedge u_{21} \wedge \neg u_{22} \wedge u_{31} \wedge H_1 \wedge \neg H_2 \wedge \neg H_3$

$u_{12} = \neg u_{11} \wedge \neg u_{12} \wedge \neg u_{21} \wedge u_{22} \wedge u_{31} \wedge H_1 \wedge \neg H_2 \wedge \neg H_3$

$u_{21} = u_{11} \wedge \neg u_{12} \wedge \neg u_{21} \wedge \neg u_{22} \wedge u_{31} \wedge \neg H_1 \wedge H_2 \wedge \neg H_3$

$u_{22} = \neg u_{11} \wedge u_{12} \wedge \neg u_{21} \wedge \neg u_{22} \wedge u_{31} \wedge \neg H_1 \wedge H_2 \wedge \neg H_3$

$u_{31} = u_{11} \wedge u_{12} \wedge u_{21} \wedge u_{22} \wedge \neg u_{31} \wedge \neg H_1 \wedge \neg H_2 \wedge \neg H_3$

Here, $u_{11}, u_{12}, u_{21}, u_{22}, u_{31}$ all satisfies the falsifying truth assignment.

And A=$u_{11} \wedge u_{21} \wedge u_{31} \notin D_3$

Also, A=$u_{12} \wedge u_{22} \wedge u_{31} \notin D_3$

Here, there are two possibilities of embedding G_1 to G_2 i.e. v_1, v_2, v_3 can be embedded on u_{11}, u_{21}, u_{31} and u_{12}, u_{22}, u_{31} .

So, G_1 is subgraph pair of G_2.

5. CONCLUSION

Although, till today it is unknown whether P=NP or not. But, the sophisticated methodologies for conversion to classify, reduce and verify the computational problem have been already developed by theoretic computer science researchers and mathematicians. In this seminar report, only the reduction of Np class problems to NP-complete problems are discussed but not about NP-hard class problems to NP-complete.

A problem L is NP-Hard if and only if Boolean satisfiability reduces to L (SAT $\leq_P$ L). A problem is NP-Hard if and only if L is in NP-Hard and $L \in NP$. Since, $\leq_P$ is a transitive relation , it follows that if satisfiability $\leq_P L_1$ and $L_1 \leq_P L_2$, then satisfiability $\leq_P L_2$. To show that an NP-Hard decision problem is NP-complete a polynomial time non-deterministic algorithm should be exhibited for it. Since, optimization problem might be NP-Hard, only the decision decision problem can be NP-complete. It can be shown that knapsack decision problem can be reduces to kanapsack optimization problem and it can also be shown that clique decision problem can be reduces to clique decision problem. In face, it can be also shown that these optimization problem can reduce to their decision problem. Yet, the optimization problems cannot shown to be NP-complete but decision problem can be shown.

In order to solve P vs. NP problem it is needed to develop a polynomial time algorithm that solves one of the NP-complete problems in polynomial time. If only one of the NP-complete problems is solved in polynomial time, then whole NP-class problems can be solved in polynomial time since, to be NP-complete problem every problem in NP should be polynomially reduced to that particular NP-complete problem. If one can do so then it can be proved $P = NP$. Conversely it can be also proved that NP-complete problems are intractable i.e. can't be solved in polynoamial time to prove $P \neq$ NP.

REFERENCES

[1] “Best, Worst, and Average-Case Complexity,” Department of Computing Science, Umeå University, 02-Jun-1997. [Online]. Available: https://www8.cs.umu.se/kurser/TDBA77/VT06/algorithms/BOOK/BOOK/NODE13.HTM. [Accessed: 08-Nov-2019].

[2] M. Sipser, “Section 7.1 : Measuring Complexity ,” in *Introduction to the Theory of Computation*, 2nd ed., Thomson Course Technology Inc, 2006, pp. 248–248.

[3] “Big O Notation and Complexity,” Data Structure and Algorithms, 15-Nov-2013. [Online]. Available: http://datastructurenalgorithms.blogspot.com/2013/11/topic-1-big-o-notation-and-complexity.html. [Accessed: 15-Nov-2019].

[4] “Review of Asymptotic Complexity,” Cornell CIS Computer Science. [Online]. Available: http://www.cs.cornell.edu/courses/cs3110/2012sp/lectures/lec19-asymp/review.html. [Accessed: 20-Nov-2019].

[5] R. W. Floyd, “ Nondeterministic Algorithms,” Journal of the ACM, vol. 14, no. 4, Oct. 1967, pp. 636–644, 1967.

[6] “Nondeterministic algorithm,” Wekipedia The Free Encyclopedia. [Online]. Available: https://en.wikipedia.org/wiki/Nondeterministic_algorithm#/media/File:Difference_between_deterministic_and_Nondeterministic.svg. [Accessed: 23-Nov-2019].

[7] J. Erikson, “Nondeterministic Turing Machines ,” Jeff Erickson. [Online]. Available: http://jeffe.cs.illinois.edu/teaching/algorithms/models/09-nondeterminism.pdf. [Accessed: 03-Dec-2019].

[8] S. A. Cook, “The P versus NP Problem,” University of Toronto. [Online]. Available: https://www.cs.toronto.edu/~toni/Courses/Complexity2015/handouts/cook-clay.pdf [Accessed: 07-Dec-2019].

[9] E. Horowitz, S. Sahni, and S. Rajasekaran, “NP-Hard and NP-Completeness Problems ,” in Computer Algorithms/C++ , 2nd ed., University Press, pp. 515–573.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, “NP-Completeness,” in Introduction to Algorithms, 2nd ed., New Delhi: Prentice-Hall of India, pp. 966–1003.

[11] S. A. Cook, “The complexity of theorem-proving procedures,” Proceedings of the third annual ACM symposium on Theory of computing, pp. 151–158, May 1971.

[12] L. Fortnow and S. Homer, “A Short History of Computational Complexities,” Boston University Open BU, Oct. 2003.[Online].Available: http://open.bu.edu [Accessed: 09-Dec-2019].