UNIT 1 INTRODUCTION TO COMPLEXITYCLASSES

Structure **Page Nos.**

1.0 INTRODUCTION

Until now we have studied a large number of problems and developed efficient solutions using different problem solving techniques. After developing solutions to simple algorithms (sorting, polynomial evaluation, exponent evaluation, GCD) , we developed solutions to more difficult problems (MCST, single source shortest path algorithms, all pair shortest path algorithms, Knapsack problem, chained matrix multiplications, etc) using greedy, divide and conquer technique and dynamic programming techniques. We also formulated some problems as optimization problems, for example, Knapsack and a single source shortest path algorithm. But so far we have not made any serious effort to classify or quantify those problems which cannot be solved efficiently. In this unit as well as in the subsequent unit we will investigate a class of computationally hard problems. In fact, there is a large number of such problems. In the last unit of this block, we will show how a computationally hard problem can be solved through an approximation algorithm.

The structure of this unit is as follows as: In section 1.2 we provide a background for understanding different classes of problems which are introduced in section 1.3. The section highlights differences between tractable and intractable problems, optimization problems and decision problems and deterministic and non-deterministic algorithms. Section 1.4 introduces the first NP Complete problem which is a basis of all other NP complete problems. Problems from graph theory and combinatorics can be formulated as a language recognition problem, for example, pattern matching problems which

can be solved by building automata. This issue is addressed in 1.5. The key issues discussed in this unit are summarized in the section1.6 and the solutions to CYPs are included in the section 1.7.

1.1 OBJECTIVES

After studying this unit, you should be able to:

- Differentiate between P, NP, NP-Complete, and NP-Hard problems
- Differentiate between tractable and Intractable problems, optimization and decision problems and deterministic and non-deterministic algorithms
- List P, NP and NP – Complete classes of problems
- Define the concept of reduction in NP-Complete problems
- Explain CNF Satisfiability Problem

1.2 SOME PRELIMINARIES TO COMPLEXITY CLASSES

In this section, some preliminaries are presented to help you understand different class complexities.

1.2.1 Tractable Vs. Intractable Problems

The general view is that the problems are **hard** or **intractable** if they can be solved only in exponential time or factorial time. The opposite view is that the problems having polynomial time solutions are **tractable or easy** problems. Although the exponential time function such as 2^n grows more rapidly than any polynomial time algorithms for an input size n, but for small values of n, intractable problems for with exponential time bounded complexity can be more efficient than polynomial time complexity. But in the asymptotic analysis of algorithm complexity, we always assume that the size of n is very large

It is to be kept in mind that intractability is a characteristic of a problem. It is not related to any problem solving technique. To explain this concept, let us take an example of a chained matrix multiplication problem which was examined earlier. The brute force approach to the solution of this problem is intractable but it can be solved in polynomial time through dynamic programming technique

Whereas intractability is concerned, usually there are two types of problems in this class. The problems which generate a non-polynomial amount of outputs such as TSP and Hamiltonian cycle problem. Consider any complete graph with n number of vertices in which every vertex is connected to other vertices that would generate (n-1)! cycles for examining in both the problems, belong

to the first category. Knapsack problem and the sum of subsets problems are also in this category. In the second category, the most well-known problem is Halting problem. The input to this problem is any algorithm which has some input data. Now it is to decide whether the algorithm will ever stop with this input. Alan Turing proved that this type of a problem is **undecidable.**

1.2.2 Optimization Problems Vs. Decision Problems

The Knapsack problem and the single source shortest path problem have been formulated as optimization problems. The former is defined as maximization optimization problem (i.e. maximum profit) and the latter is defined as minimization optimization problem (minimum cost of a path). Depending upon the problem, the optimal value is minimum (single source shortest path) or maximum (Knapsack problem) which is selected among a large number of candidate solutions. Optimization problems can be formally defined as:

An Optimization problem is one in which we are given a set of input values, which are required to be either maximized or minimized w. r. t. some constraints or conditions. Generally an optimization problem has n inputs (call this set as **input domain** or **Candidate set**, C), we are required to obtain a subset of C (call it ***solution set***, S where, $S \subseteq C$) that satisfies the given constraints or conditions. Any subset S⊆C, which satisfies the given constraints, is called a ***feasible*** solution. We need to find a feasible solution that maximizes or minimizes a given objective function. The feasible solution that maximizes or minimizes a given objective function, is called an **optimal solution**. For example, find the shortest path in a graph, find the minimum cost spanning tree, 0/1 knapsack problem and fractional knapsack problem.

Decision Problems:

There is a corresponding decision problem to each optimization problem. Unlike an optimization problem, a decision problem outputs a simple "yes" or "no" answer. Most NP-complete problems that we will discuss will be phrased as decision problems. For example, rather than asking, what is the minimum number of colors needed to color a graph, instead we would phrase this as a decision problem: Given a graph G and an integer k, is it possible to color G with k colors ?

The following examples distinguish between two types of problems.

Example 1: Traveling Salesperson Problem

TSP Optimization Problem- Given a complete graph G = (V,E) and edge weight, the optimization problem find out an optimal path covering all the vertices only once except the starting vertex with the least cost.

TSP Decision Problem – It can be formulated as: Given a complete weighted graph G = (V,E) and an integer K, is there a cycle/path comprising all the vertices only once except the starting vertex and has a total cost ≤ K.

Ex 2: 0-1 Knapsack Problem

The 0-1 Knapsack Optimization Problem- Given the number of items, profit and weight of each item and maximum weight of a Knapsack, the 0-1 Knapsack optimization problem determines the maximal total profit of items that can be placed in the Knapsack

The 0-1 Knapsack Decision Problem-The objective of the decision problem is to determine whether it is possible to earn a profit not less or equal to some amount P while not exceeding the maximum Knapsack capacity.

Example 3: Graph Coloring Problem

Graph Coloring Optimization problem- Given a graph G= (V,E), the graph coloring optimization problem determines minimum numbers of colors needed to color a graph so that no two adjacent vertices of the graph are having the same color.

Graph Coloring Decision Problem- It determines whether a given graph can be colored in at most M colors such that no two adjacent vertices of the graph are having the same color

Example 4: Clique optimization and Decision problem

Given an undirected graph, G= (V, E) clique is a subset of vertices W of V such that all the vertices in W are adjacent to each other consider the following graph.

In this example {V2, V3 and V5} in a clique where as {V2, V3 and V4} is not a clique because V2 is not adjacent to V4. A maximal clique contains maximum number of vertices. For ex. In the given graph a maximal clique is {V1, V2, V3, V5}

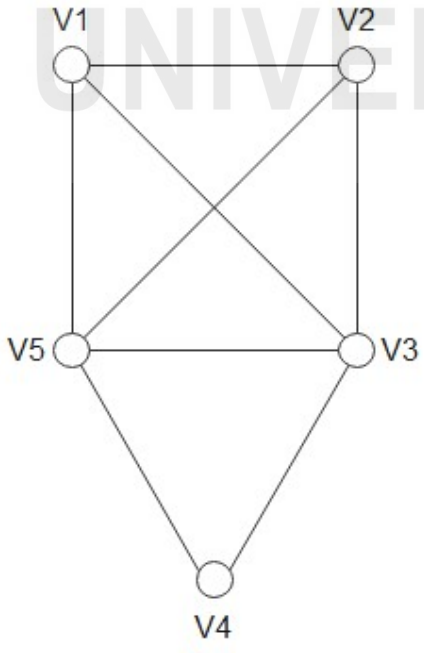


Figure 1: A Problem Instance of a Clique

The clique optimization problem finds out the size of a maximal clique for a given graph. Given a graph and some integer value K, the clique decision problem determines whether there is a clique containing at least K vertices.

We observe that an optimization problem can be formulated as a related decision problem by imposing a bound on the value of optimization. For ex. Graph coloring decision problem determines whether a graph can be colored in at most m colors.

1.2.3 Deterministic Vs. Nondeterministic Algorithms

For a given input, a deterministic algorithm will always produce the same output on an input going through the same sequence of steps, even when run multiple times. All the algorithms that we have seen so far are deterministic.

A non-deterministic algorithm behaves in a completely different way. There may be multiple next steps possible after any given step and the algorithm is allowed to choose any of them in an arbitrary manner. Thus, even for a fixed input, a non-deterministic algorithm may proceed through different sequences of steps on different runs, and may even produce different outputs. It should be understood that non-determinism is a totally abstract theoretical concept and such algorithms do not naturally occur in nature; however, such algorithms have been found to be useful in understanding the behaviour of deterministic algorithms.

1.3 Introduction to P, NP,NP Hard and NP-Complete Problems

The theory of NP-completeness identifies a large class of problems which cannot be solved in polynomial time. Categories the problems into three classes namely P (Polynomial), NP (Non-deterministic Polynomial), and NP complete.

1.3.1 P Class

Almost all problems which we examined so far, belong to polynomial class of problems. For examples, finding out a key in an array, Greatest Common Divisor of two integers, matrix multiplication, sorting algorithms, finding a shortest path in a weighted graph, finding a minimum cost spanning tree, problems implemented through dynamic programming such as, All Pairs Shortest Path Algorithm, Chained Matrix Multiplication and Optimal Binary Search Tree and Binomial Coefficient can be solved in polynomial time.

An algorithm solves a problem in polynomial time if its worst -case time complexity belongs to O(p(n)) where n is a size of a problem and p(n) is polynomial of the problem's input size n. Problems can be classified as tractable and intractable problems. Problems with polynomial time solutions are called **tractable,** problems which do not have polynomial time solutions are called **intractable.**

Problems with the following worst-case time complexities belong to polynomial class of problems:

5n, $5n^3$ +10n, n log^n

Problems with the following worst- case time complexities do not belong to polynomial class of problems:

2^n, n!, $2^{\sqrt{n}}$

Informally P can be defined as a set of problems that has polynomial time solutions. A more formal definition of P is that it includes all decision problems that has yes/no answers. Finally, we can define class P as a class of decision problems that can be solved by deterministic algorithms in polynomial time.

As we have seen in the previous section that many optimization problems can be formulated as decision problems which are easier to implement. For example, consider a graph coloring problem. Instead of asking a minimum number of colors needed to color the vertices of a graph having two adjacent vertices of a graph in different colors, we should ask whether a graph can be colored in no more than m colors where m = 1, 2, …. without coloring the adjacent vertices of a graph in the same color.

1.3.2 NP Class

NP stands for nondeterministic polynomial. It is a class of decision problems that can be solved by nondeterministic polynomial algorithms. Unlike deterministic algorithms which executes the next instruction which is unique because there is no choice for execution of any other instruction, nondeterministic algorithms have a choice among the next instructions.

How do we know that a problem is in NP? The simplest way is to formulate a problem as a decision problem i.e., yes/no question and verify the yes instance of a solution in polynomial time. Most decision problems are in NP. The NP class includes all the problems that have a polynomial time deterministic algorithm. This expression is true. The reason is that if the problem belongs to P class, we can apply deterministic polynomial algorithm to solve / verify it in polynomial time.

An NP algorithm consists of two stages:

(i) Guessing Stage (Nondeterministic stage): Given a problem instance, in this stage a simple string S is produced which can be thought of as a guess (candidate solution) to the problem instance.

(ii) Verification Stage (Deterministic stage). Input to this stage is the problem instance, the distance d and the string S. A deterministic algorithm takes these inputs and outputs yes if S is a correct guess to the problem instance and stops running. In case S is not a correct guess, then the deterministic algorithm outputs no or it may go to an infinite loop and does not halt.

A nondeterministic algorithm is said to be nondeterministic polynomial if the verification stage is completed in polynomial time.

A famous decision problem which is not in NP problem is the **halting** problem, which is defined as follows:

Given an arbitrary program with an arbitrary data, will the program terminate or not? This type of problem is called undecidable problems. Undecidable problems cannot be solved by guessing and checking. Although they are decision problem somehow they cannot be solved by exhaustively checking the solution space

Now Let us write a nondeterministic algorithm for TSP (Traveling Salesperson Problem):

(i) Nondeterministic Stage: The following table displays the results of some input strings S generated at the nondeterministic stage for the problem instance graph given in figure 1with the total distance d value that is claimed to be a tour not greater than d(i.e. 18)and passed to the function *verify* as an input.

String S	*Output*	*Reasons*
[A,B,C,D,A]	False	Total weight of a tour(S string)is greater than 18
[A,D,B,C,A]	False	S is not a correct tour
[A,C,B,D,A]	True	S is a tour with weight not exceeding 18

The third string is the correct guess for the tour. Therefore, the nondeterministic stage of the algorithm is satisfied. In general, function *verify*at the verification stage return True if the guess for a particular instance is correct ,otherwise the *verify* function does not return true.

(ii) Deterministic Stage : In this stage we write a*verify* function that verifies whether the string S produced is a tour with weight no greater than 18.

Pseudocode

```
Boolean verify(weighted graph G, d, S)
{
 if ( S is a tour and the total weight of the edges in S ≤ d)
  return True;
else
return False;
}
```

The algorithm first checks to see whether S is indeed a tour. If the sum of weights is no greater than d, it returns "True".

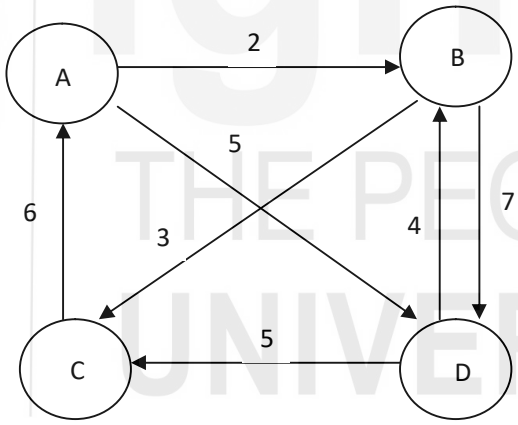


Figure 2: A Problem Instance for TSP

What other decision problem are in NP? There are many such problems. Just to add few examples here: Knapsack, Graph coloring problem, clique, etc.

- Finally there are a large number of problems that are trivially in NP. It means that every problem in P is also in NP as shown in the following diagram(figure3):

T.

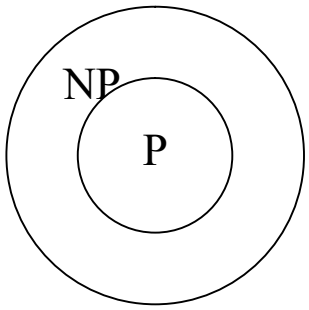


Figure 3: P as a Subset of NP

In this figure NP contains P as a proper subset. However, this is not a case all the time. So far, no proof has been provided that there is a problem in NP that is not in P. Therefore (NP-P) may be completely empty.

Figure 2: Set of all decision problem.

The important issue in theoretical computer science is whether P = NP or P≠NP . If P=NP, we would have polynomial time solution from any known decision problems. For the later case, when P≠NP, we would have to find a problem in NP that is not in P. But to show P=NP, we have to find a polynomial time algorithm for each problem is NP, or find a polynomial-time algorithm for any NP-complete problem.

Check Your Progress – 1

Q1: List the problems belonging to polynomial class

Q2: Formulate Graph Coloring Problem as an optimization and decision problems

Q3: What is Halting Problem?

1.3.3 NP Complete Problems

NP Complete Problems – NP complete problems are the most difficult problem, also called hardest problems in the subset of NP-class. It has been shown that there are large class of problems in NP which are NP-Complete problems as shown in figure 3

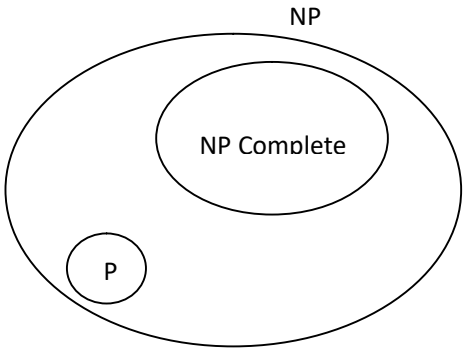


Figure 4: NP Problems containing both P and NP-Complete Problems

The set of NP-Complete problems keeps on growing all the time. It includes problems like Traveling Salesperson Problem, Hamiltonian Circuit Problem, Satisfiability Problem, etc. The list is becoming larger. The common feature is that there is no polynomial time solution for any of these problems exists in the worst cases. In other words, NP-Complete problems usually take super polynomial or exponential time or space in the worst cases.

It is very unlikely that all the NP-Complete problems can be solved in polynomial time but at the same time no one has claimed that NP-Complete problems cannot be solved in polynomial time.

The following is true from the definition of NP-Completeness:

All NP problems can be solved in polynomial time, if any NP-Complete problem is solved in polynomial time.

An important feature of NP-Complete problem definition is that any problem in NP can be **reduced** or mapped to it in polynomial time. To reduce one problem X to another problem Y, we need to do mapping such that a problem instance of X can be transformed to a problem instance of Y, solve Y and then do the mapping of the result back to the original problem. Let us try to understand the process of reduction through one example. You want to add two decimal numbers using our calculator. First the two numbers are entered into a calculator. Then the numbers will be mapped (converted) to binary. The addition operation will be done in binary. Finally, the result will be mapped(reconverted) to decimal for display. All these steps will be done in polynomial time.

The main take away from the above discussion is that if we have two problems: X and Y, X is a NP-Complete problem and Y is known to be in NP. Further, X is polynomially reducible to Y so that we can solve X using Y. Since X is NP-Complete, every problem in NP can be reduced to it in polynomial time

1.4 CNF – Satisfiability Problem-A First NP-Complete Problem

Let us recall that for a new problem to be NP-Complete problem , it must be in NP-class and then a known NP- Complete problem must be polynomially reduced to it. In this unit we will not show any reduction example, because we are interested in the general idea. Some examples of reductions will be covered in the next unit. But one might be wondering how the first NP-Complete problem was proven to be NP-Complete and how the reduction was done. The satisfiability problem was the first NP-Complete problem ever found. The problem can be formulated as follows: Given a Boolean expression, find out whether the expression is satisfiable or not. Whether an assignment to the logical variables that gives a value 1(True)
A logical variable, also called a Boolean variable is a variable having only one of the two values: 1 (True) or False (0). A **literal** is a logical variable or negation of a logical variable. A **clause** combines literals through logical **or**(V) operator(s). ***A conjunctive normal form (CNF)*** combines several clauses through logical **and (∧)** operator(s)

The following is an example of logical expression in CNF having three clauses combined by **and (∧)** operators and logical variables X_1, X_2, X_3, X_4, and complement of X_1, X_2, X_3 *and* $X_4 (\overline{X_1}, \overline{X_2}, \overline{X_3}$ *and* $\overline{X_4})$:

$$\left(X_2 \bigvee \overline{X_3}\right) \bigwedge (\overline{X_1} \bigvee \overline{X_2} \bigvee X_4) \wedge (X_1 \vee X_3 \vee X_4)$$

Circuit Satisfability Decision Problem asks for a given logical expression in CNF, Whether some combination of True and False values to the logical variables makes the output of the expression as True.

For instance, if we assign X_1 = True ,X_2=False and X_3 = True, then The following logical expression in CNF satisfiability is Yes because the assignments of True and False values to the logical variables make the Boolean expression True.

$(X_1 \vee X_2 \vee X_3) \wedge \left(\mathrm{X1} \vee \overline{X_2}\right) \wedge X_3$

But the assignments of X_1= True, X_2 – True and X_3 – True make the following Boolean expression False.

$(X_1 \vee X_2) \wedge \overline{X_2} \wedge X_3$
Therefore the answer to CNF Satisfiability is False.
Since it is not difficult to write a polynomial time algorithm to evaluate a Boolean expression and check whether the result is true, CNF Satisfiability problem is in NP. But to show that CNF satisfiability is NP-Complete, Cook applied the fact that every problem in NP is solvable in polynomial time by a nondeterministic Turing machine. Cook demonstrated that the actions of Turing machine can be simulated by a lengthy and complicated Boolean expression but still in polynomial time. The Boolean

expression would be true if and only if the program being run by a nondeterministic Turing machine produced a Yes answer for its input.
Several new problems such as Hamiltonian Circuit problem, TSP problem, Graph Coloring problem, etc. were proven to be NP-Complete after the satisfiability problem was shown to be NP-Complete.

Check Your Progress-2

Q1: What is P Vs NP problem?

Q2 What is Circuit Satisfiability Decision Problem?

1.5 Summary

In this unit the following topics were discussed:

- Three classes of problems in terms of its time complexities in the worst cases: P, NP, NP-Complete.
- Relationship between P, NP, NP-Complete
- Differences between tractable and intractable problems, optimization and decision problems and deterministic and nondeterministic algorithms
- CNF Satisfiability Problem

1.6 Solution to Check Your Progress

Check Your Progress – 1

Q1: List the problems belonging to polynomial class

Almost all problems which we examined so far, belong to polynomial class of problems. For examples, finding out a key in an array, Greatest Common Divisor of two integers, matrix multiplication, sorting algorithms, finding a shortest path in a weighted graph, finding a minimum cost spanning tree, problems implemented through dynamic programming such as, All Pairs Shortest Path Algorithm, Chained Matrix Multiplication and Optimal Binary Search Tree and Binomial Coefficient can be solved in polynomial time.

Q2: Formulate Graph Coloring Problem as an optimization as well as decision problems

Ans. Graph Coloring Optimization problem- Given a graph G= (V,E), the graph coloring optimization problem determines minimum numbers of colors needed to color a graph so that no two adjacent vertices of the graph are having the same color.

Graph Coloring Decision Problem- It determines whether a given graph can be colored in at most M colors such that no two adjacent vertices of the graph are having the same color

Q3 What is Halting Problem?

Ans. A famous decision problem which is not in NP problem is the **Halting** problem, which is defined as follows:

Given an arbitrary program with an arbitrary data, will the program terminate or not? This type of problem is called undecidable problems. Undecidable problems cannot be solved by guessing and checking. Although they are decision problem somehow they cannot be solved by exhaustively checking the solution space

Check Your Progress-2

Q1: What is P Vs NP problem?

If P=NP, we would have polynomial time solution for many known decision problems. When P≠NP. we would have to find a problem in NP that is not in P. But to show P=NP, we have to find a polynomial time algorithm for each problem is NP.

Q2 What is Circuit Satisfiability Decision Problem?

Ans. Circuit Satisfiability Decision Problem asks for a given logical expression in CNF, Whether some combination of True and False values to the logical variables makes the output of the expression as True

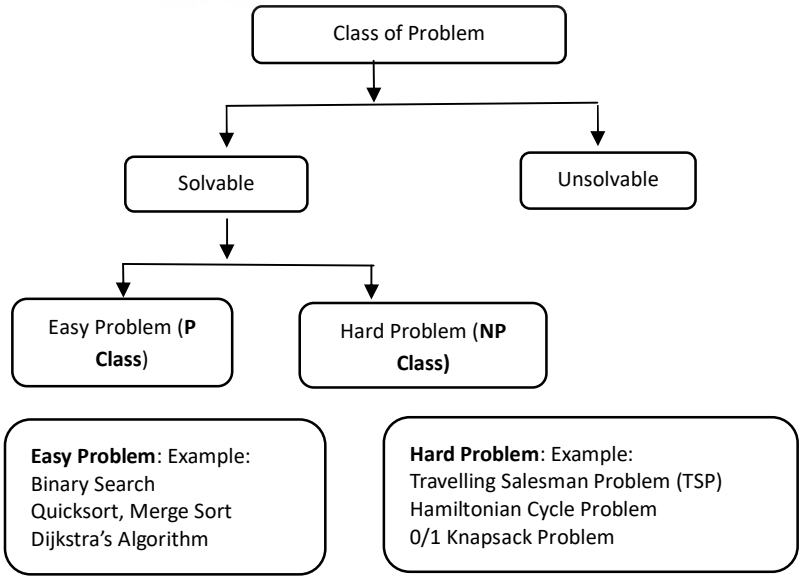
.

UNIT 2 NP-COMPLETENESS AND NP-HARD PROBLEMS

Structure **Page No**

2.0 INTRODUCTION

First, let us review the topics discussed in the previous unit. In general, a class of problems can be divided into two parts: Solvable and unsolvable problems. Solvable problems are those for which an algorithm exist, and Unsolvable problems are those for which algorithm does not exist such as Halting problem of Turing Machine etc. Solvable problem can be further divided into two parts: **Easy problems and Hard Problems**.



A Problem for which we know the algorithm and can be solved in polynomial time is called a P-class (or Polynomial -Class) problem, such as Linear search, Binary Search, Merge sort, Matrix multiplication etc.

There are some problems for which polynomial time algorithm(s) are known. Then there are some problems for which neither we know any polynomial-time algorithm nor do scientists believe them to exist. However, exponential time algorithms can be easily designed for such problems. The latter class of problems is called NP (non-deterministic polynomial). Some P and NP problems are listed in table-1.

Table-1: Examples of P-Class and NP-Class of Problems

P problems	NP problems
Linear Search	0/1 Knapsack Prob.
Binary Search	TSP
Selection Sort	Sum of Subsets
Merge Sort	Graph Colouring
Matrix Multiplication	Hamiltonian Cycle

There is a huge scope of Research related to this topic:

We Know that Linear search takes O(n) times and Binary Search takes $O(logn)$time, Researchers are trying to search an algorithm which takes lesser time than $O(logn)$,may be $O(1)$. We know that the lower bound of any comparison based sorting algorithm is $O(nlogn)$,we are searching a faster algorithm than $O(nlogn)$, may be $O(n)$.

We mentioned above it is easy to design an exponential time algorithm for any NP problem. However, for any such problem, we ultimately want some polynomial time algorithm. Any exponential time taking algorithm is very time-consuming algorithm as compared to any polynomial time taking algorithm, for example:

$2^n, 3^n, 5^n, \ldots$ takes more time than any polynomial algorithms $n^3, n^4, \ldots, n^{10}$.Even if, for example, $n^{10} \vee n^{20}$ is smaller than 2^n for some large value of n. Researchers from Mathematics or Computer science are trying to write (or find) the polynomial time algorithm for those problems for which till now no polynomial time algorithm exist. This is a basic framework for NP-Hard and NP-Complete problems. The hierarchy of classes of problem can be illustrated by following figure1.

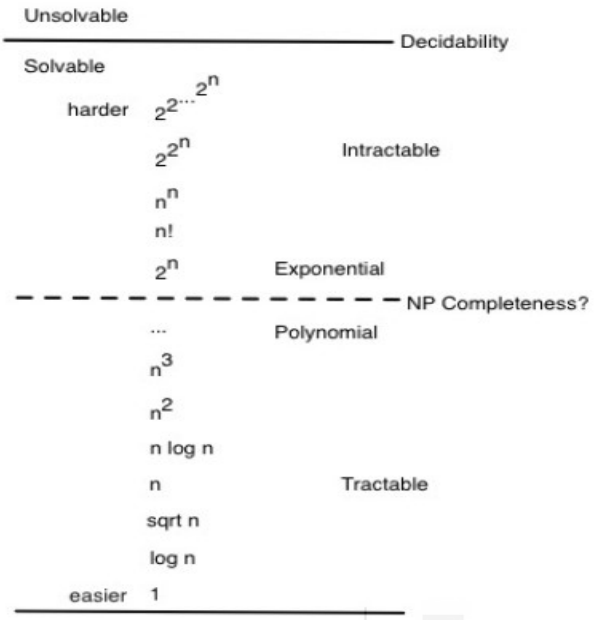


Figure-2: Hierarchy of classes of problems

Problems that can be solvable in a reasonable (polynomial) time is called ***tractable problem*** and the ***intractable problems*** means, as they grow large, we are unable to solve them in reasonable time. Reasonable time means we want polynomial time algorithm for those problems. That is, on an input of size n the worst-case running time is $O(n^k)$ for some constant k.

The obvious question is "whether all problems are solvable?". The answer is "No". For example, the "Halting Problem" of Turing machine is not solvable by any computer, no matter how much time is given. Such problems are called undecidable or unsolvable problems.

2.1 OBJECTIVES

After studying this unit, you should be able to:

- Define P, NP, NP-Complete Classes of Problems through language theoretic framework
- Explain NP-Hard problem such as CLIQUE, Vertex-Cover Problem (VCP), 0/1 Knapsack Problem etc.
- Understanding polynomial time reduction of NP-Complete problems
- Proving NP completeness
- Explore P, NP and NP – Complete classes of problems

2.2 Definition of P and NP as Classes of Languages

Many problems from graph theory, combinatorics can be defined as language recognition problems which require Yes/No answer for each instance of a problem. The solution to the problem formulated as a language recognition problem can be solved by a finite

automata or any advanced theoretical machine like Turing machine (refer to MCS-212/Block2/Unit 2)

Using a formal language theory we say that the language representing a decision problem is accepted by an algorithm A is the set of strings L = { x ∈ (0,1) ; A(x) =1}. If an input string x is accepted by the algorithm, then A(x) = 1, and if x is rejected by the algorithm, then A(x) = 0.

Let us introduce now important classes of languages :P, NP,NP- Complete class of languages:

P-Class (Polynomial class):

If there exist a polynomial time algorithm for L then problem L is said to be in class P (Polynomial Class), that is in worst case L can be solved in (n^k) , where n is the input size of the problem and k is positive constant.

In other word,

$$P = \left\{ L \middle| \begin{matrix} L\ can\ be\ decided (accepted) by\ deterministic\ Turing\ machine \\ (algorithm) \in polynomial\ time \end{matrix} \right\}$$

Note: Deterministic algorithm means the next step to execute after any step is unambiguously specified. All algorithms that we encounter in real-life are of this form in which there is a sequence of steps which are followed one after another.

Example: Binary Search, Merge sort, Matrix multiplication problem, Shortest path in a graph (Bellman/Dijkstra's algorithm) etc.

NP-Class (Nondeterministic Polynomial time solvable):

NP is the set of decision problems (with 'yes' or 'no' answer) that can be solved by a **N**on-deterministic algorithm (or Turing Machine) in **P**olynomial time.

$$NP = \left\{ L \middle| \begin{matrix} L\ can\ be\ decided\ (accepted) by\ Nondeterministic\ Turing\ machine \\ (algorithm) \in polynomial\ time \end{matrix} \right\}$$

Let us elaborate Nondeterministic algorithm (NDA). In NDA, certain steps have deliberate ambiguity; essentially, there is a choice that is made during the runtime and the next few steps depend on that choice. These steps are called the non-deterministic steps. It should be noted that a non-deterministic algorithm is an abstract concept and no computer exists that can run such algorithms.

Here is an example of an NDA:

Algorithm: (Nondeterministic **Linear search)**
/* **Input:** A linear list A with n elements and a searching element x.
Output: Finds the location LOC of $x$in the array A (by returning an index)
or return LOC=0 to indicate $x$is not present in A. */

$NDLSearch(A, n, x)$

{

1. [Initialize]: Set j=1 and LOC=0.
2. $j = Choice(\ )$; // **Nondeterministic Statement**

```
3.  if(x = A[j])
4.    {
5.      LOC=j
6.  printf
7.    }
8.  if(LOC = 0)
9.  printf
}
```

Here we assume that Line No 2 of $NDLSearch(A, n, x)$ takes 1 unit of time to find the location of searched element x (i.e. index j). So the overall time complexity of this algorithm is $O(1)$. Here, line no. 2 of the algorithm is nondeterministic (magical). In j is chosen correctly, the algorithm will execute correctly, but the exact behavior can only be known when the algorithm is running.

An DA algorithm is said to be correct if for every input value its output value is correct. Since the actual behaviour of an NDA algorithm is only known when it is running, we define the correctness of an NDA algorithm differently than above. An NDA algorithm is said to be correct if for every input value **there are some correct choices during the non-deterministic steps** for which the output value is correct. In the above example, there is always some correct value of j in line no. 2 for which the algorithm is correct. Note that j can depend on the input. For example, if A[1]=x, then j=1 is a correct choice, and if A does not contain x, then any j is a correct choice. Hence, the above algorithm is a correct NDA for the search problem.

So, we can define NP class as follows:

"A nondeterministic algorithm (NDA) but takes polynomial time". It should not difficult to view a deterministic algorithm as also a non-deterministic algorithm that **does not** make any non-deterministic choice. Thus, if there exists a DA for a problem, then the same algorithm can also be thought of as an NDA; what this means is that if a DA exists for a problem, then an NDA also exists for a problem. Answer to to the converse question is not always known. Further, if the DA algorithm is polynomial-time then the NDA algorithm is also polynomial-time. So, we can say the following relationship hold between P and NP class of problem.

$$P \subseteq NP \quad \text{..... (1)}$$

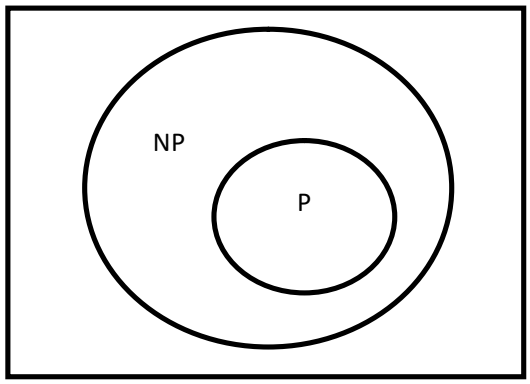


Fig.2: Relationship between P and NP class of problem

A central question in algorithm design is the P=NP question. We discussed above that if a problem has a polynomial time DA then it has a (trivial) polynomial-time NDA. The P=NP question asks the converse direction. Currently we know many problems for which there are polynomial time NDA. The question asks whether a polynomial-time NDA can be always converted to a polynomial-time DA. This is effectively asking whether the non-deterministic choices made by an NDA algorithm can be completely eliminated, maybe at the cost of a slight polynomial increase in the running time. The scientist community believes that the answer is no, i.e., P is not equal to NP, in other words, there are NP problems which cannot be solved in polynomial time. However, the exact answer is not known even after several decades of research.

2.3 Polynomial time Reduction

In general, any new problem requires a new algorithm. But often we can solve a problem X using a known algorithm for a related problem Y. That is, we can reduce X to Y in the following informal sense: A given instance x of X is translated to a suitable instance y of Y, so that we can use the available algorithm for Y. Eventually the result of this computation on y is translated back, so that we get the desired result for x. Let us consider an example to understand the concept of a reduction step.

Suppose we want an algorithm for multiplying two integers, and there is already an efficient algorithm available that can compute the square of an integer. It needs S(n) time for an integer of n digits. Can we somehow use it to multiply arbitrary integers a, b efficiently, without developing a multiplication algorithm? Certainly, squaring and multiplication are related problems. More precisely, we can use the identity $ab = \frac{((a+b)^2-(a-b)^2)}{4}$. We only have to add and subtract the factors in O(n) time, apply our squaring algorithm in S(n) time, and divide the result by 4, which can be easily done in O(n) time, since the divisor is a constant. Thus, we have reduced multiplication (problem X) to squaring (problem Y) as follows. We have taken an instance of X (factors a, b), transformed it quickly into some instances of Y (namely $a + b$ and $a - b$), solved these instances of Y by the given squaring algorithm, and finally applied another fast manipulation on the results (addition, division by 4) to get the solution ab to the instance of problem X. It is essential that not only a fast algorithm for Y is available, but the transformations are fast as well.

There are two different purposes of reductions:

1. Solving a problem X with help of an already existing algorithm for a different problem Y.

2. Showing that a problem Y is at least as difficult as another problem X.

Note that (1) is of immediate practical value, and even usual business: Ready-to-use implementations of standard algorithms exist in software packages and algorithm libraries. One can just apply them as black boxes, using their interfaces only, without caring about their internal details. This is nothing but a reduction!

Point (2) gives us a way to classify problems by their algorithmic complexity. We can compare the difficulty of two problems without knowing their "absolute" time complexity. If Y is at least as difficult as X, then research on improved algorithms should first concentrate on problem X.

Reductions-Formal definition:

Reduction is a general technique for showing similarity between problems. To show the similarity between problems we need one base problem. A procedure which is used to show the relationship (or similarity) between problems is called ***Reduction step,*** and symbolically can be written as

$$A \leqslant_p B$$

The meaning of the above statement is "problem A is polynomial time reducible to problem B" and if there exist a polynomial time algorithm for problem A then problem B can also have polynomial time algorithm. Here problem A is taken as a base problem

We define a reduction for decision problems X, Y as follows: We say that X is reducible to Y in t(n) time, if we can compute in t(n) time, for any given x with |x| = n, an instance y = f(x) of Y such that the answer to x is Yes if and only if the answer to y is Yes. If the time t(n) needed for the reduction is bounded by a polynomial in n, we say that X is polynomial-time reducible to Y.

Let us understand the concept of reduction using mathematical description. Suppose that there are two problems, A and B. You know (or you strongly believe at least) that it is impossible to solve problem A in polynomial time. You want to prove that B cannot be solved in polynomial time. How would you do this?
We want to show that
$(A \notin P) \Rightarrow (B \notin P)$ ----- (2)
To prove (2), we could prove the contrapositive
$(B \in P) \Rightarrow (A \in P)$ [Note: $(Q \rightarrow P$is the contrapositive of $P \rightarrow Q]$

In other words, to show that B is not solvable in polynomial time, we will suppose that there is an algorithm that solves B in polynomial time, and then derive a contradiction by showing that A can be solved in polynomial time.
How do we do this? Suppose that we have a subroutine that can solve any instance of problem B in polynomial time. Then all we need to do is to show that we can use this subroutine to solve problem A in polynomial time. Thus we have "reduced" problem A to problem B.
It is important to note here that this supposed subroutine is really a *fantasy*. We know (or strongly believe) that A cannot be solved in polynomial time, thus we are essentially proving that the subroutine cannot exist, implying that B cannot be solved in polynomial time.

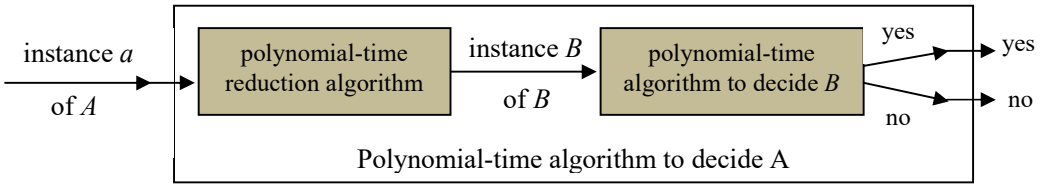


Figure 7 Reduction step of problem A to problem B

A problem A can be polynomially reduced to a problem B if there exists a polynomial-time computable function f that converts an instance α of A into an instance β of B.

Reduction is a general technique for showing similarity between problems. To show the similarity between problems we need one base problem.
We take SAT (Satisfiability) problem as a base problem.

SAT problem: Given a set of clauses $C1, C2, \ldots, Cm$in CNF form, where Ci contains literals from $x1, x2, \ldots, xn$. The Problem is to check if all clauses are simultaneously satisfiable.

Note: Cook-Levin theorem shows that **SAT** is NP-Complete, which will be discussed later.

To understand the reduction step, consider a **CNF-SAT (or simply SAT)** problem and see how to reduce it to the Independent Set (IS) problem.

Consider a Boolean formula

$$f(x_1, x_2, x_3) = (\overline{x_1} \vee \overline{x_2} \vee x_3) \wedge (x_1 \vee x_2 \vee \overline{x_3}) \ldots\ldots\ldots\ldots (1)$$

We know that for n variables, we have total 2^n possible values. Since there are 3 variables so 2^3 =8 possible values (as shown in figure 3). The question asked by the SAT problem is if there is some of$x_1, x_2, x_3$for which the formula f is satisfiable, That is, out of 8 possible values of x_1, x_2, x_3does any assignment make the formula f TRUE?

x_1	x_2	x_3
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Fig. 3 8 possible values for 3 variables

It can be easily verified that solution to the given formula in (1) will be either $(x_1, x_2, x_3) = (0,0,0)$ or $(x_1, x_2, x_3) = (0,1,0)$, which evaluate the formula f to 1 (TRUE). So the formula is satisfiable. We do not know of any polynomial time algorithm to find this out for arbitrary formulas, and all known algorithms run in exponential time.

In the IS problem, a graph G and an integer k is given and the question is to find out if there is a subset of vertices with at least k vertices such that there is no edge between any two vertices in that subset. Such a subset is known as an independent set.

A reduction from SAT to IS is an algorithm that converts any SAT instance, which is a Boolean formula (denoted f), to an IS instance, i.e., a graph G and an integer k. The reduction must run in polynomial time and ensure the following: If the formula is satisfiable then G must have an independent set with at least k vertices. And if the formula is not satisfiable then G must not have any independent set with k or more vertices. The catch is that the reduction must do the above **without** finding out if f is satisfiable (the requirement should be obvious since there is no known algorithm to determine satisfiability that runs in polynomial time).

Even though it may appear surprising how to perform the conversion, it is indeed possible as is shown in Fig. 3.9. It is easy to check that the formula on the left is satisfiable (by setting x1=1, x2=0,x3=1,x4=0) and the graph on the right has an independent set with 3 vertices (x1,x2,x3).

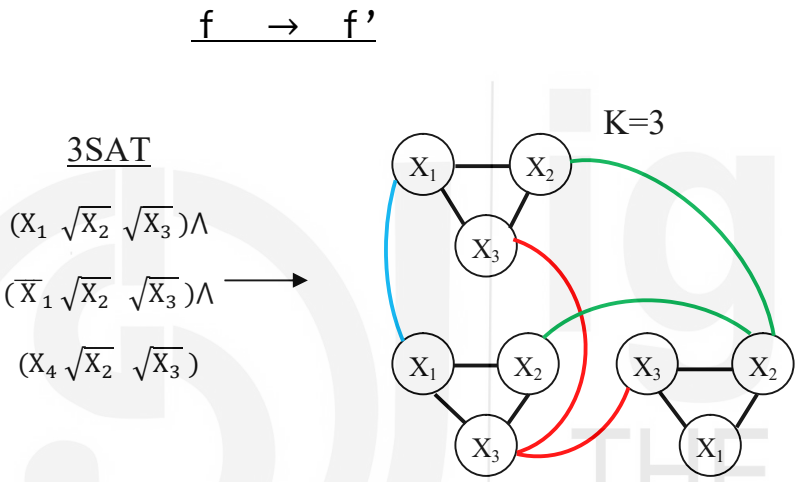


Fig 3.8 The reduction algorithm from SAT to IS converts the formula on the left to the pair of integer K and the graph shown on the right. It can be

A procedure which is used to perform this conversion between problems is called ***Reduction step,*** and symbolically can be written as

$SAT \leqslant_p 0/1\, knapsack$ SAT <=p IS

The meaning of the above statement is "SAT problem is polynomial time reducible to Independent Set problem". The implication is that if there exist a polynomial time algorithm for IS problem then SAT problem can also have polynomial time algorithm. And furthermore, if there is no polynomial time algorithm for SAT then there cannot be a polynomial time algorithm for IS. Here SAT problem is taken as a base problem. There are similar reductions known between thousands of problems, like CLIQUE, Sum-of subset, Vertex cover problem (VCP), Travelling salesman problem (TSP), 0/1-Knapsack, etc.

2.4 NP-Hard and NP-Complete problems

In some books, NP-Hard or NP-Complete problems are stated in terms of *language-recognition problems.* This is because the theory of NP-Completeness grew out of

automata and formal language theory. Here we will not be using this approach to define these problems, but you should be aware that if you look in the book, it will often describe NP-Complete problems as languages.

In theoretical computer science, the classification and complexity of common problem definitions have two major sets; **P** which is "Polynomial" time and **NP** which "Non-deterministic Polynomial" time. There are also **NP-Hard** and **NP-Complete** sets, which we use to express more sophisticated problems. In the case of rating from easy to hard, we might label these as "easy", "medium", "hard", and finally "hardest":

- $Easy \rightarrow P$
- $Medium \rightarrow NP$
- $Hard \rightarrow NP - Complete$
- $Hardest \rightarrow NP - Hard$

The following figure 6 shows a relationship between P, NP, NP-C and NP-Hard problems:

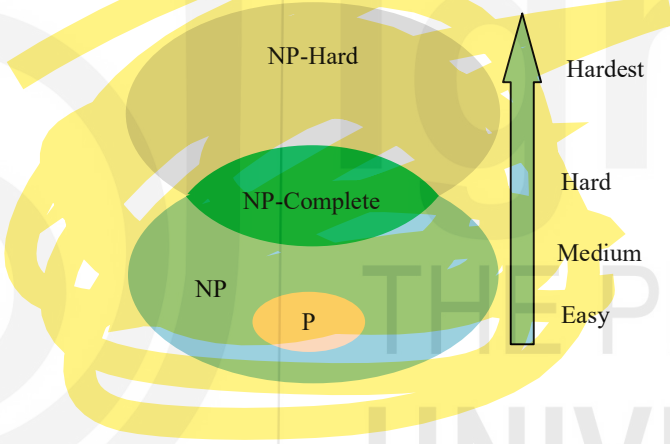


Figure 6: Relationship between P, NP and NP-Complete and NP-Hard

Using the diagram, we assume that P and NP are not the same set, or, in other words, we assume that $P \neq NP$. Another interesting result from the diagram is that there is an overlap between NP and NP-Hard problem. We call this **NP-Complete** problem (problem belongs to both NP and NP-Hard sets).

NP-Complete problems are the "hardest" problems to solve among all NP problems, in that if one of them can be solved in polynomial time then every problem in NP can be solved in polynomial time. This is where the concept of reduction comes in. There may be even harder problems to solve that are not in the class of NP, called NP-Hard problems.

The study of NP Completeness is important: the most cited reference in all of Computer Science is Garey & Johnson's (1979) book Computers and Intractability: A Guide to the Theory of NP-Completeness. (A text book is the second most cited reference in Computer Science!).

In 1979 Garey & Johnson wrote, "The question of whether or not the NP-complete problems are intractable is now considered to be one of the foremost open questions of contemporary mathematics and computer science."

Over 30 years later, in spite of a million-dollar prize offer and intensive study by many of the best minds in computer science, this is still true: No one has been able to either

- Prove that there are problems in NP that cannot be solved in polynomial time (which would mean P ≠NP), or
- Find a polynomial time solution for a single NP Complete Problem (which would mean P=NP).

Although either alternative is possible, most computer scientists believe that P ≠ NP

Now, to understand the concept of NP-Hard problem, let us consider by the following example.

CNF-SAT problem is well known NP-Hard problem (a base problem to prove other problems are NP-Hard) and let us call all the other exponential time taking problems (0/1 knapsack, TSP, VCP, Hamiltonian cycle etc.) as a hard problem. We can say that if all these hard problems are related to CNF-SAT problem and if CNF-SAT is solved (in polynomial time) then all these hard problems can also be solved in polynomial time.

2.4.2 NP-Hard Problem

Definition: A problem L is said to be NP-Hard problem if there is a polynomial time reduction from already known NP-Hard problem L' to the given problem L.

Symbolically $L' \leqslant_P L for all L' \in NP$

Note that L need not be in NP class.

Let us consider an example to understand the concept of NP-Hard. We have already seen a CNF-SAT problem is polynomial time reducible to 0/1 knapsack problem, and denoted as **CNF-SAT**$\leqslant_p$ $0/1\, knapsack$ **---(1)**

Here CNF-SAT is already known NP-Hard problem and this known NP-hard problem is polynomial time reduces to given problem L (i.e., 0/1 knapsack problem). By writing **CNF-SAT**$\leqslant_p$ $0/1\, knapsack$ means that "if 0/1 knapsack problem is solvable in polynomial time, then so is 3CNF-SAT problem, which also means that, if 3CNF-SAT is not solvable in polynomial time, then the 0/1 knapsack problem can't be solved in polynomial time either."

In other word, here we show that any instance (say I_1) or formula of CNF-SAT problem is converted into a 0/1 knapsack problem (say instance I_2) and we can say that if 0/1 knapsack problem is solved in polynomial time by using an algorithm A then the same algorithm A can also be used to solve CNF-SAT problem in polynomial time. Note that reduction (or conversion) step takes polynomial time.

2.4.3 NP-Complete Problem

NP-Complete problems are the "hardest" problems to solve among all NP problems. The set of NP-complete problems are all problems in the complexity class NP, for which it is known that if anyone is solvable in polynomial time, then they all are, and conversely, if

anyone is not solvable in polynomial time, then none are. In other word we can say that if any problem in NP-Complete is polynomial-time solvable, then P=NP.

Definition: A decision problem is said to be NP-Complete if the following two conditions are satisfied:

1. $L \in NP$
2. L is NP Hard (that is every problem (say L') in NP is polynomial-time reducible to L).

An equivalent definition of NP-completeness is that (i) $L \in NP$ and (ii) $L' \leqslant_P L$, that is, there is a polynomial time reduction from some known NP-complete problem to L.

Consider an example to understand the concept of NP-Complete. Suppose a well-known NP-Hard problem CNFSAT is reduces to some problem L, that is **CNF-SAT**$\leqslant_p L$ then L is also NP-Hard. To prove a problem is NP-complete, all we need to do is to show that our problem is in NP (that is there exist a nondeterministic polynomial time algorithm for our problem). You may wonder how the first NP-complete problem was proved to be so, since according to the equivalent definition above, another NP-complete problem would be required.

Cook proved that there exists a Nondeterministic polynomial time algorithm for the CIRCUIT-SAT problem (similar to 3CNF-SAT), and also showed that any other problem for which a similar algorithm exists can be reduced in polynomial-time to the CIRCUIT-SAT problem, thereby discovering the first NP-complete problem. Very quickly many other problems, like 3CNF-SAT, were discovered to be NP-complete (see Figure 10).

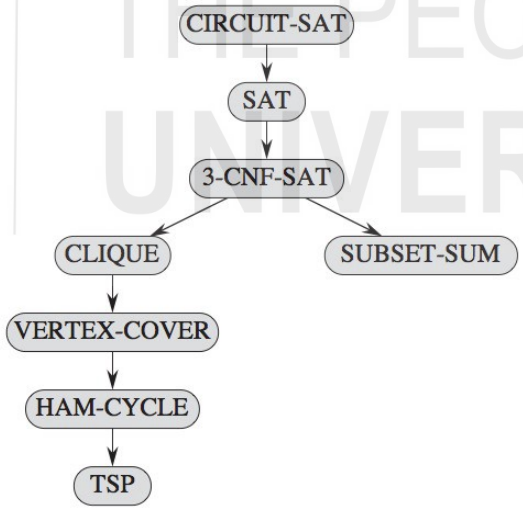


Figure 10 Few early NP-complete problems. An array from A -> B indicates that first A was proved to be NP-complete, and then B was proved as NP-complete using a polynomial-time reduction from A.

2.4.4 Relation between P, NP, NP-Complete and NP-Hard Problems:

The following figure 9 shows the relation between P, NP, NP-Complete and NP-Hard class of problems.

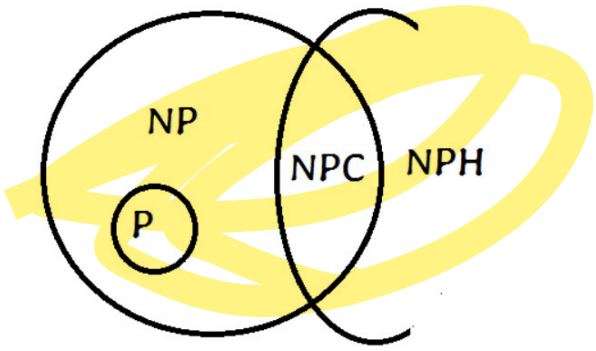


Figure 9: Relation between P, NP, NP-Complete and NP-Hard

From figure-9, it is clear that intersection of NP-Hard and NP-class (there exist a nondeterministic polynomial time algorithm for problem) is NP-Complete problem.

P- class means polynomial time (deterministic) algorithm exist for the problem. NP-class means nondeterministic polynomial time algorithm exist for those problems. whether P=NP or not, it's an open question to all computer scientist. So today we can say $P \subseteq NP$.

Check your progress 1

Q1. Differentiate between P, NP, NP-C and NP-Hard Problems with a suitable diagram.

Q2. Which of the following option is true, if we assume $P \neq NP$?

A. NP-complete = NP
B. NP-complete ∩ P = ∅
C. NP-hard = NP
D. P = NP-complete

Q3Let L be an NP-complete problem and Q and R be two other problems not known to be in NP. Q is polynomial time reducible to L and L is polynomial-time reducible to R. Which one of the following statements is true?

A. Q is NP-Complete
B. Q is NP-Hard
C. R is NP-Complete
D. R is NP-Hard

Q4 Consider two decision problems A1, A2 such that A1 reduces in polynomial time to 3-SAT and 3-SAT reduces in polynomial time to A2. Then which one of the following is consistent with the above statement?

A. A1 is in NP, A2 is NP hard
B. A2 is in NP, A1 is NP hard

C. Both A1 and A2 are in NP

D. Both A1 and A2 are in NP hard

Q5: Consider the following statements about NP-Complete and NP-Hard problems. Write **TRUE/FALSE** against each of the following statements

1. The first problem that was proved as NP-complete was the circuit satisfiability problem.

2. NP-complete is a subset of NP Hard problems, i.e. NP-Complete⊆NP-Hard.

3. SAT problem is well-known NP-Hard as well as NP-Complete problem.

Q6 Which of the following statements are **TRUE**?

1. The problem of determining whether there exists a cycle in an undirected graph is in P.

2. The problem of determining whether there exists a cycle in an undirected graph is in NP.

3. If a problem A is NP-Complete, there exists a non-deterministic polynomial time algorithm to solve A.

2.5 Definitions of some well-known NP-Complete problems

We have seen many problems that can be solved in polynomial time such as Binary search, Merge sort, matrix multiplication etc. The class of all these problems are so solvable are in P-Class. The following are some well-known problems that are NP-complete when expressed as decision problem (with 'YES' or 'NO' answer). In the previous unit we have defined

1. Boolean satisfiability problem (SAT)
2. 3-CNF SAT
3. 0/1 Knapsack problem.
4. Travelling salesman problem (decision version)
5. Subset sum problem.
6. Clique problem.
7. Vertex cover problem.
8. Hamiltonian cycle problem.
9. Graph coloring problem

1. **SAT problem** is defined as follows:

Input: Given a CNF formula f having m clauses $C_1, C_2, \ldots\ldots, C_m$ in n variables $x_1, x_2, \ldots\ldots, x_n$. There is no restriction on number of variables in each clause.

The problem is to find an assignment (value) to a set of variables $x_1, x_2, \ldots\ldots, x_n$ which satisfy all the m clauses simultaneously.

In other word, "is there an assignment of values (0 or 1) to the variables $x_1, x_2, \ldots\ldots, x_n$ which makes the formula f to TRUE (or 1)"?. Note that there are 2^npossible assignments (values) to the n variable so exhaustive search will take time $O(2^n)$.

$SAT = \{f: f$ is a given Boolean formula in CNF, Is this formula f satisfiable?$\}$

SAT:
Instance: A Boolean formula $\emptyset$consisting of variables, parentheses, and and/or/not operators.
Question: Is there an assignment of True/False values to the variables that makes the formula evaluate to True?

2. **3-CNF-SAT Problem:** 3-CNF SAT problem is defined as follows:

3-SAT:
Instance: A CNF formula with 3 literals in each clause.
Question: Is there an assignment of True/False values to the variables that makes the formula evaluate to True?

3. **0/1Knapsack problem:**

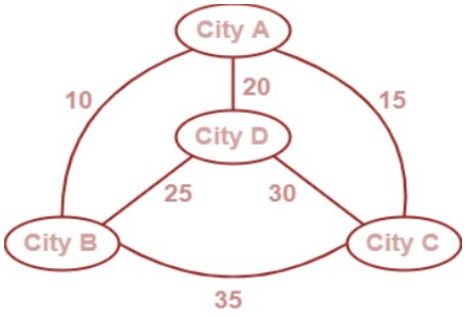
A decision version of 0/1 knapsack problem is NP-complete.

Given a list $\{i_1, i_2, \ldots, i_n\}$, a knapsack with capacity M and a desired value V . Each object i_i has a weight w_i and value v_i. Can a subset of items $X \subseteq \{i_1, i_2, \ldots, i_n\}$ be picked whose total weight is at most M and at total value is at least V? that satisfy the following constraints:

Maximize (the value) $\sum_{i=1}^{n} v_i x_i$ such that $\sum_{i=1}^{n} w_i x_i \leq M$ and $x_i \in \{0,1\}$

4. **Traveling Salesman Problem (TSP):**

Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point. There is a integer cost $C(i,j)$ to travel from city i to city j and the salesman wishes to make the tour whose total cost is minimum, where the total cost is the sum of individual costs along the edges of the tour. For example, consider the following graph:



A TSP tour in the graph is A-B-D-C-A. The cost of the tour is 10+25+30+15=80

In formal language, TSP problem is defined as:

$$TSP = \left\{ \langle G, C, k \rangle \middle| \begin{matrix} G = (V, E) is\ a\ complete\ graph, C \\ is\ the\ function \\ \\ V \times V \to Z, k \in Z \wedge Ghasa \end{matrix} \middle| \begin{matrix} Travelling \\ saleaman\ tour \\ with\ cost\ at\ most\ k \end{matrix} \right\}$$

TSP is a famous NP-Hard problem. There is no polynomial time know solution for this problem. There are few different approaches to solve TSP. if we use Naïve method (brute force method), it takes $\theta(n!)$ time to solve TSP. Dynamic programming approach takes $O(n^2 2^n)$ time to solve TSP.

TSP as a decision problem is defined as follow:

Input: Given a undirected connected weighted graph $G(V, E)$, and an integer k.

Question: Does a graph G has tour of cost at most k? If P ≠ NP, then we can not find the minimum-cost tour in polynomial time.

5. **Sum-of-Subset problem:**

Given a set of positive integer S = $\{x_1, x_2, ..., x_n\}$ and a target sum K, the decision problem asks for a subset $S'$of S($i.e. S' \subseteq S$) having a sum equal to K.

SUBSET_SUM = {⟨S, k⟩: ∃ subset $S' \subseteq S$ such that sum of elements of S' equal to given sum k}.

$$SUBSET\text{-}SUM = \{\langle S, t\rangle |\ S = \{x_1, \ldots, x_k\}, \text{ and for some } \{y_1, \ldots, y_l\} \subseteq \{x_1, \ldots, x_k\}, \text{ we have } \Sigma y_i = t\}.$$

Example: Suppose an input Set S = {2, 3, 5, 8, 10, 4, 6}, and Sum k = 9
Output: True
There is a subset (4, 5) and (3,6) with sum 9.

Input set S = {3, 34, 4, 12, 5, 2}, sum = 30
Output: False
There is no subset that add up to 30

6. CLIQUE Problem:

CLIQUE is a complete subgraph problem.

A **clique** in an undirected graph G = (V, E) is a subset $V' \subseteq V$, each pair of which is connected by an edge in E (a complete subgraph of G). The **clique problem** is the problem of finding a clique of maximum size in G. This can be converted to a decision problem by asking whether a clique of a given size k exists in the graph:

CLIQUE = {⟨G, k⟩: G is a graph containing a clique of size k}

To understand a CLIQUE problem, let us first define a Complete Graph. A complete graph is one in which every vertex of G is connected with every other vertices of G, and it is denoted by K_n. Some examples of complete graph for n = 1,2,3,4 and 5 are shown in figure-3

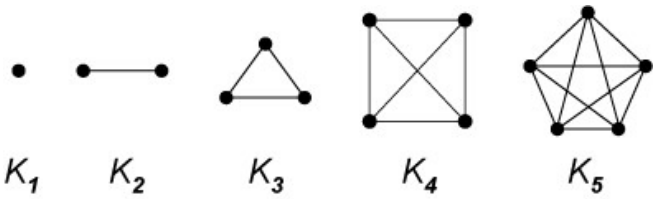


Figure3 Complete graph K_n for $n = 1,2,3,4 \wedge 5$

In any complete graph, if number of vertices $|V| = n$ then the total number of edges in K_n is: $|E| = \frac{n(n-1)}{2}$.

Consider the following three examples:

Ex1:-

In a graph of EX1, a CLIQUE of size k=4 is possible

Ex:- 2

K = 4 □

K = 3 ✓

In this example a CLIQUE of size k=5 or

k=4 is not possible, but of size k=3 is possible.

Ex:- 3

K = 4 ✓

K = 3 ✓

K = 2

So,
Decision problem : A graph is having a CLIQUE of size k or not.
For example, consider a graph of Exampe2, question is, Is there a clique of size k=4, answer is NO. Next Is there a clique of size k=3, the answer is Yes.
Optimization problem: Find what is the max. CLIQUE size of a graph.

7. Vertex Cover Problem (VCP):

A vertex cover of an undirected graph G = (V, E) is a subset of the vertices $V' \subseteq V$ such that for any edge $(U, V) \in E$, then either $u \in V'$ or $v \in V'$or both.

In other word, a vertex cover is a subset of all of the vertices of a graph such that every edge in the graph is "covered" or incident to at least one vertex in the vertex cover subset. The size of the vertex cover, then, is simply the number of vertices in the vertex cover subset. For Example, the vertex cover set for the following graph, as shown in figure-5 is $\{a, c, d, e\} \vee \{a, b, d, e\}$.

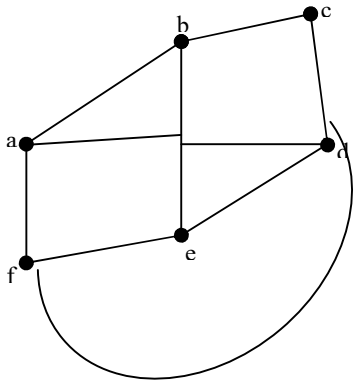


Figure-5: Vertex Cover for G is $\{a, c, d, e\} \vee \{a, b, d, e\}$

Vertex Cover as a decision problem

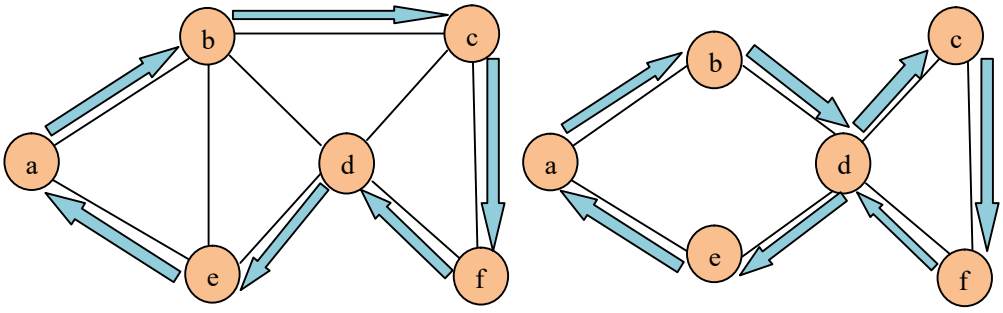
The problem VERTEX COVER, stated as a decision problem, is to determine whether a given graph $G(V, E)$ with |V|=n, has a vertex cover of size k, where $k \leq n$.

VERTEX-COVER = {$\langle G, k \rangle$: graph G has a vertex cover of size k}

VERTEX COVER:
Instance: A graph G and an integer K.
Question: Is there a set of K vertices in G that touches each edge at least once?
So, the input to VERTEX COVER is a graph G and an integer k. The algorithm returns either a certificate/witness, or "no such vertex cover exists".

8. **Hamiltonian Cycle problem:**

A Hamiltonian Cycle of an undirected graph $G(V, E)$ is a simple cycle that passes through all the vertices of the graph G exactly once. For example, the following graph as shown in figure, contains a cycle $a - b - c - f - d - e - a$ that visits each vertex exactly once. Therefore, the cycle is a Hamiltonian cycle. On the other hand, in the second graph, there exists a cycle in the graph that visits all vertices, $a - b - d - c - f - d - e - a$ but vertex d appears twice in the cycle. Thus, it is not a Hamiltonian cycle



Hamiltonian Graph

Non Hamiltonian

Figure7 Example of Hamiltonian and Non-Hamiltonian Graph

We can write this HAMILTONIAN CYCLE problem as a decision problem as follow:

HAM-CYCLE=$\{G(V,E\}|DoesagraphGhaveahamiltoniancycle?\}$
Or
Instance: An undirected graph G = (V, E).
Question: Does G contain a cycle that visits every vertex exactly once?

9. Graph colouring Problem:

The natural graph coloring optimization problem is to color a graph with the fewest number of colors. We can phrase it as a decision problem by saying that the input is a pair (G, k) and then

- The search problem is to find a k-coloring of the graph G if one exists.
- The decision problem is to determine whether or not G has a k coloring.
- Clearly, solving the optimization problem solves the search problem which in turn solves the decision problem.
- Conversely, if we can efficiently solve the decision problem then there is a clever algorithm that we can use to efficiently solve the decision and optimization problems.

Formally it is the graph coloring decision problem which is NP-complete. More precisely, the decision problem for any fixed $k \geq 3$ is NP-complete. However, 2-Colorability is in P.

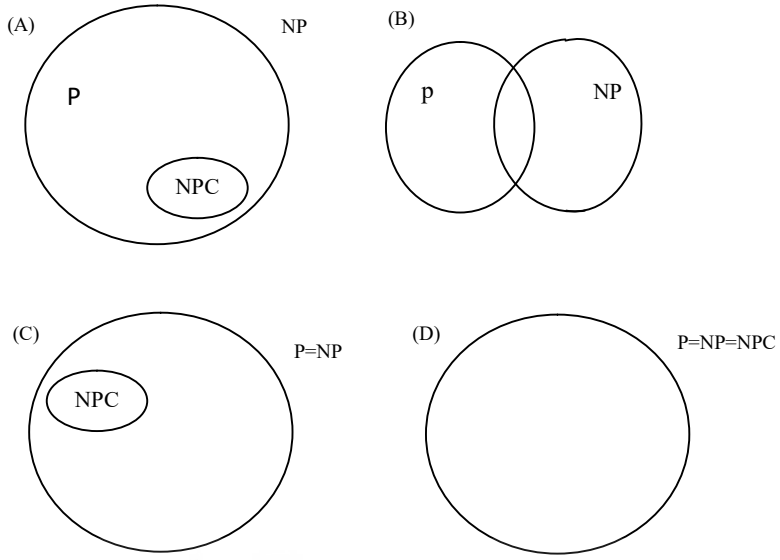
Check your Progress-2

Q1. Which of the following is an NP-complete problem?

A. Given a graph G, is there a cycle that visits every vertex exactly once?
B. Given a graph G, is there a cycle that visits every vertex at least once?
C. Given a graph G, is there a cycle that visits every vertex at most once?

Q2. Which of the following are NP-complete?
D. Given a graph G, can G be coloured using 2 colours?
E. Given a graph G, can G be coloured using 3 colours?
F. Given a graph G, can G be coloured using 4 colours?

Q.3: Suppose a polynomial time algorithm is discovered that correctly computes the largest clique in a given graph. In this scenario, which one of the following represents the correct Venn diagram of the complexity classes P, NP and NP Complete (NPC)?



2.8 Summary

- There are many problems which have decision and optimization versions, for example Traveling salesman problem (TSP). Optimization: find Hamiltonian cycle of minimum weight.

  Decision: is there a Hamiltonian cycle of weight $\leq k$.

- P = set of problems that can be solved in polynomial time. For example: Binary search, Merge sort, Quick sort, matrix multiplication, Dijkstra's algorithm etc.

- NP = set of problems for which there is a non-deterministic polynomial time algorithm. Examples: 0/1 Knapsack, TSP, CNFSAT, 3-CNF SAT, CLIQUE, VCP etc.

- ***Important results***: Clearly $P \subseteq NP$ (means any problem that can be solved by deterministic machine in polynomial time can also be solved by non-deterministic machine in polynomial time) but P≠NP

- Open question to all scientist: Does P = NP?

- A lot of times you can solve a problem by reducing it to a different problem. We can reduce Problem B to Problem A if, given a solution to Problem A, we can easily ("easily" means polynomial time) construct a solution to Problem B. That is reduction (or translation) of one problem into another in such a way that a fast solution to one problem would automatically give us a fast solution to the other.

- A problem L is said to be **NP-Hard problem** if there is a polynomial time reduction from already known NP-Hard problem L' to the given problem L. Symbolically $L' \preccurlyeq_P L for all L' \in NP$.Note that L need not be in NP class.

- A problem is ***NP-complete*** if the problem is both NP-hard and NP.

- **Reduction:** A problem R can be reduced to another problem Q if any instance of R can be rephrased to an instance of Q, the solution to which provides a solution to the instance of R. This rephrasing is called a transformation or reduction. The intuition behind this is that, If R reduces in polynomial time to Q, R is "no harder to solve" than Q.
- If A is polynomial-time reducible to B, we denote this $A \leqslant_P B$
- Definition of NP-Hard and NP-Complete: If all problems $R \in NP$are polynomial-time reducible to Q, then Q is NP-Hard.
- We say Q is NP-Complete if Q is NP-Hard and $Q \in NP$
- If $R \leqslant_P Q$ and R is NP-Hard, Q is also NP-Hard.
- Circuit Satisfiability (SAT) problem is the first NP-Complete problem.
- (Properties of polynomial time reductions):

✓ if Problem L1 is polynomial time reducible to L2 and if L2 is solvable in polynomial time then L1 can also be solvable in Polynomial time, that is, If $L_1 \leqslant_P L_2$ and $L_2 \in P$ then $L_1 \in P$.

✓ 2. If $L_1 \leqslant_P L_2$ and $L_2 \in NP$ then $L_1 \in NP$

✓ 3. If Problem L1 is polynomial time reducible to L2 and if L1 is not likely to be solvable in polynomial time then L2 is also not likely to be solvable in polynomial time, that is If $L_1 \leqslant_P L_2$ and $L_1 P$ then $L_2 P$.

- Definitions of some well-known decision problems:
  - $SAT = \{f : f$ is a given Boolean formula in CNF with n variables and m clauses, Is this formula f satisfiable?$\}$
  - $3CNF - SAT = \{f : f$ is a given Boolean formula in CNF with n variables and m clauses and at most 3 literals per clause, Is this formula f satisfiable?$\}$
  - **CLIQUE** = $\{\langle G, k\rangle$: G is a graph containing a clique of size $k\}$
  - **VERTEX-COVER** = $\{\langle G, k\rangle$: graph G has a vertex cover of size k$\}$
  - **HAM-CYCLE**=$\{G(V, E\}|$ Does a Graph G has a Hamiltonian Cycle?$\}$
  - $$TSP = \left\{ \langle G, C, k\rangle \middle| \begin{matrix} G = (V, E)\ is\ a\ complete \\ graph, C\ is\ the\ function \\ \\ V \times V \to Z, k \in Z \wedge Ghasa \end{matrix} \middle| \begin{matrix} Travelling \\ saleaman\ tour \\ with\ cost\ at\ most\ k \end{matrix} \right\}$$

2.9 Solutions/Answers

Answer1:

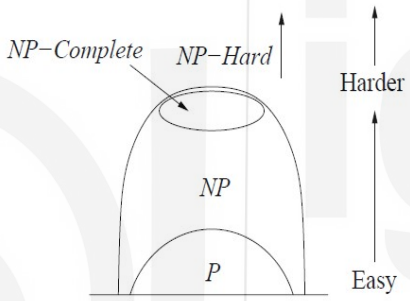
P = set of problems that can be solved in polynomial time. For example: Binary search, Merge sort, Quick sort, matrix multiplication, Dijkstra's algorithm etc.

NP = set of problems for which a solution can be verified in polynomial time. Examples: 0/1 Knapsack, TSP, 3-CNF SAT, CLIQUE, VCP etc.

A problem L is said to be ***NP-Hard problem*** if there is a polynomial time reduction from already known NP-Hard problem L' to the given problem L. Symbolically $L' \leqslant_P L for all L' \in NP$.Note that L need not be in NP class.

A problem is ***NP-complete*** if the problem is both NP-hard and in NP class.

A relationship between P, NP, NP-Complete and NP-Hard is shown as:



(Objective Questions)

Answers 2-B

Answer 3-D

Answer 4-A

Answer 5: 1-True, 2-True, 3-True

Answer 6 : 1-True, 2-True, 3-True

Check your progress 2

Answer 1 : Option A
It is trivial to visit every vertex in the graph and return to the starting vertex if we are allowed to visit a vertex any number of times.

Answer 2: Option E, F
Any graph cycle detection algorithm can be used to identify if a graph has any cycle; such algorithms run in polynomial time.

Answer 3: Option D

2.11 FURTHER READINGS

1. Introduction to Algorithms. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein (PHI)
2. Algorithm Design, Jon Kleinberg and Eva Tardos, Pearson
3. *Fundamentals of Algorithmics,* G. Brassard & P. Brately: (Prentice-Hall International, 1996).
4. *Fundamental Algorithms* (*Second Edition*), D.E. Knuth: (Narosa Publishing House).
5. *Fundamentals of Computer Algorithms*, E. Horowitz & S. Sahni: (Galgotia Publications).
6. *The Design and Analysis of Algorithms,* Anany Levitin: (Pearson Education, 2003).

****

UNIT 3 HANDLING INTRACTABILITY

Structure

3.0 INTRODUCTION

It has been stated in the previous units that the large class of optimization problems belong to NP -hard class of problems.It is widely accepted that NP-hard problems are intractable problems, i.e., although not yet proven, it appears that such problems do not have polynomial time solutions.Their time complexities are exponential in the worst cases. Some examples of intractable problems are traveling salesman problem, vertex cover problems and graph coloring problems. Besides these combinatorial problems, there are several thousand computational problems in different domains like, biology, data science, finance and operation research fall into NP-hard category.

An exhaustive search can be applied to solve these combinatorial problems but it works when problem instances are quite small. An optimization technique such as dynamic programming may be applied to find solutions for combinatorial problems some such problems but it also has the similar limitation as in exhaustive search that the problem instance should be small. Another limitation of this technique is that the problems must follow the principle of optimality.

Problem solving techniques such as **backtracking and branch and bound techniques** perform better in comparison to exhaustive search. But unlike exhaustive search, these techniques construct the solutions step by step (considering only one element at a time) and performs evaluation of the partial solution. In case the results do not represent a better solution, further exploration of the remaining elements are not considered. Backtracking and branch and bound techniques apply some kinds of intelligence on the exhaustive search. It provides efficient solution if it has good design. But in the worst case it takes exponential time. One real advantage of branch and bound technique is that it can handle a large problem instance.

We can also apply Tabu search or Simulated Annealing which are heuristic local search methods or Genetic Algorithms and Particle Swarm Optimization which are meta heuristic techniques to find out solutions to the optimization problems.

But all these methods in spite of good performance do not ensure rigorous guarantees for achieving the quality of solution i.e., how far the proposed solution is far away from the optimal solution. Sometimes we should ask questions whether there is a possibility of finding near optimal solutions (approximate solution) to combinatorial optimization problems efficiently? Approximation algorithms have been found to be efficient algorithms with good quality solutions which are measured in terms of the maximum distance between the proposed approximate solution and the optimal

solution over all the problem instances. What does it mean? It means that the approximation algorithms always produce solution quite close to the optimal solution.

The focus of the unit will be to discuss few techniques such as backtracking and branch and bound techniques and approximation algorithms to handle intractable problems.

3.1 OBJECTIVES

The main objectives of the unit are to:

- Describe and apply backtracking and branch and bound techniques to combinatorial problems
- Define the concepts of state space tree and approximation ratio
- Formulate Hamiltonian Cycle problem, Subset Sum problem and Knapsack problem
- List approximation ratios of few combinatorial problems

3.2 BACKTRACKING AND BRANCH AND BOUND TECHNIQUES

Design of both techniques is based on state space tree, similar to a binary tree. Each node of the tree represents a partial solution. A node is terminated as soon as it is evident that no solution can be provided by the node's descendants.

How is backtracking different from branch and bound technique?

They differ in terms of types of problems they can solve and how nodes in the state space tree are generated?

Backtracking generally does not apply to optimization problems whereas branch and bound technique can be used to solve optimization problems because it computes a bound and proceeds further with a best bound. Depth first search is used to generate a tree in backtracking whereas there is no such restriction applicable to generate a state space tree using branch and bound.

3.2.1 Backtracking

Backtracking is a general technique for design of algorithms. It is applied to solve problems in which components are selected in sequence from the specified set so that it satisfies some criteria or objective. The backtracking procedure includes depth first search of a state space tree, verifying whether a node is leading to any solution(called promising node)or not but dead ends (called non promising node), does backtracking to the parent of the node if a node is not promising and continuing with the search process on the next child.

In this section we will use backtracking technique to solve two problems: (i) Hamiltonian Circuit problem and (ii) Subset Sum problem.

(i) Hamiltonian circuit problem

Suppose G = (V, E) is a connected graph with n vertices, Hamiltonian circuit problem determines a cycle that visits every vertex of a graph only once except the starting

vertex.$V_1$is a starting vertex of a cycle where$V_1 \in$ G and V_1, V_2……..$V_{n+1}$are distinct vertices in the cycle exceptV_1 and V_{n+1}vertices which are equal.

The following is a problem instance of a graph:

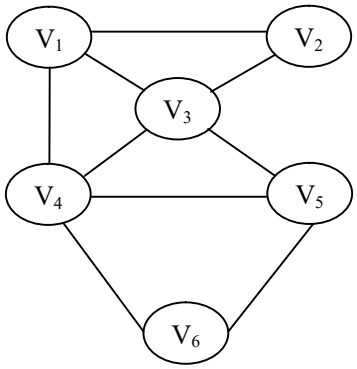


Figure 3.1: A graph for finding Hamiltonian cycle

The following figure represents the state space tree of the above graph(figure 3.1)

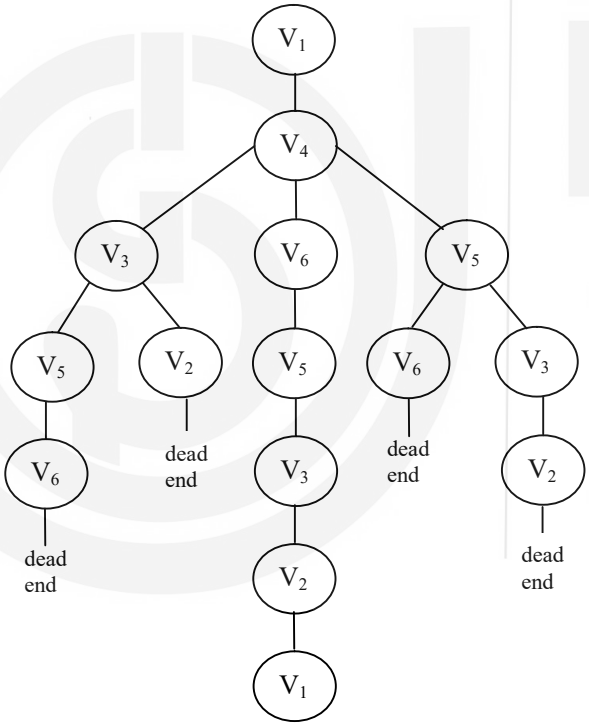


Figure 3.2 : State Space Tree representing Hamiltonian circuit of a graph

Design of a state space tree- Let us pick up V_1 as the starting vertex in a state space tree of a graph given at figure 3.2. Among the three options available to select a node from V_1, we pick upV_4. From V_4 it can go to V_3or V_6 or V_5. If we follow the order of V_6, V_5,V_3, V_2 and V_1, we get a correct Hamiltonian cycle. In other two cases there are dead ends. So the algorithm backtracks. For example in the left most branch of state space tree the algorithm backtracks from V_6 to V_5 to V_3 and then toV_2. This branch also does not provide any result. The algorithm backtracks again to $V_3$and then toV$_4$ and then follows the next path in the tree.

In order to explore another Hamiltonian cycle, the process can re-start from the leaf node of the tree through backtracking .

(ii) Subsetsum Problem

Given a positive integer W and a set S of n positive integer values i.e., S = { s_1, s_2,.. s_n}, the main objective of the Subset Sum problem is to search for all combinations of subsets of integers whose sum is equal to W. As an example let us take S = { 1,4,6,9} and W = 10, there are two solution subsets : {1,9} and {4,6}. In some cases, problem instances may not have any solution subset.

We will assume that elements in the set are in the sorted order, i.e., $s_1 \leq s_2 \leq \cdots \leq s_n$.

Design of state space tree for subset sum problem:

S = {4,6,7,8} and W = 18

The state space tree is designed as a binary tree. The root of the tree does not represent any decision about a node. It is simply a starting point of the tree. Its left and right sub trees include and exclude the element of the set represented by 1 and 0 respectively at every level.

At level 1 in the left branch of the tree, the second element $s_2$is included while in the right subtree it is excluded. A subset of a given set is represented by a path from the root node to the leaf node in the tree. A path from the root node of the tree to a node at the i_{th} level represent the inclusion of the first i numbers in the subset.

Let s' be the sum of numbers of all the nodes included at the i_{th} level. If s' is equal to W then the problem has a final solution. If we want to find out all the subsets, we need to do backtracking to the parent of the node or stop if no further subsets are required.

If the sum s' is not equal to W then the following two inequalities hold and the node can be declared as non-promising node.

(i) $s' + s_{i+1} > W$ (the large value of the sum)
(ii) $s' + \sum_{j=i+1}^{n} s_j < W$ (the small value of the sum)

The complete solution to the problem S = { 4, 6, 7, 8} and W =18 is given in figure 3.3 in form of the state space tree

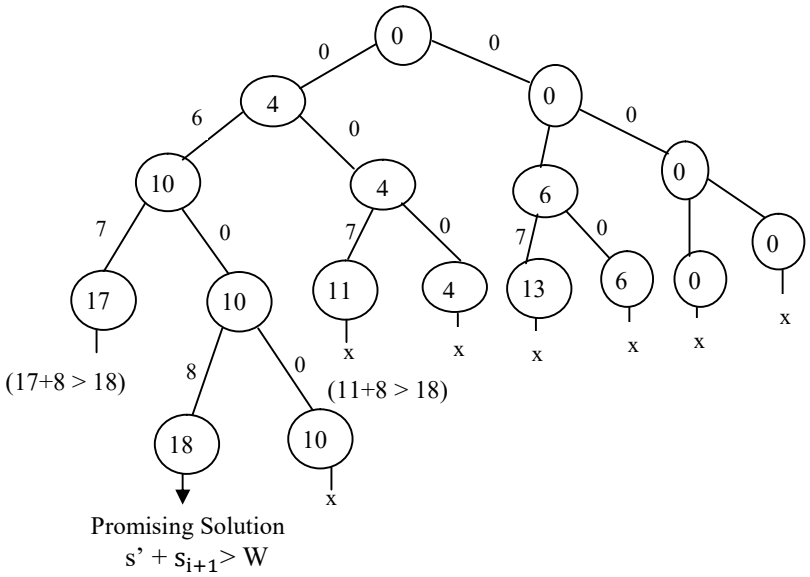


Figure 3.3 State space tree to the problem instance S = { 4, 6, 7, 8} and W =18 through backtracking algorithm

3.2.2 Branch and Bound

The similarity between backtracking and branch and bound techniques is design of state space tree. There are three distinct features of branch and bound technique: (i) unlike backtracking which traverse the tree in DFS, branch and bound does not restrict traversing in a particular way (ii) branch and bound technique computes a bound (value) of a node to decide whether the node is promising or not promising (iii) generally used for optimization problems

In branch and bound technique, search path in state space tree at a particular node is stopped if any of the following reasons occurs:

- Constraint violation
- The value of a bound of a node is inferior to the value of the best solution achieved so far

Let us now apply this technique to solve the Knapsack optimization problem. The knapsack problem is defined as: given a set of objects, its profit p_i, weight w_i , i= 1,2 ….n and W which is a knapsack capacity of Knapsack , obtain subsets of the objects which gives maximum profit and the total weight of the subsets do not exceed knapsack capacity W.

Before designing a state space tree of knapsack problem using branch and bound technique, sorting of objects in descending order by their profit to weight ratio(p_i/w_i) isdone i.e., $p_1/w_1 \geq p_2/w_2 \geq \ldots p_n/w_n$. This sequence will return the first object with the best profit and the last object with the least profit in terms per weight unit respectively. In case ties occur, it should be resolved arbitrarily.

State space tree- It is constructed in form of a binary tree. The root node of the tree is the initial point where no item has been selected (p = 0, w = 0). As in backtracking, the left branch of the tree includes the next object while the right branch excludes it. A node is represented by the three values:

w – total weight of items selected at a node

p – total profit

bound – total profits of any subset of items

One of the simplest way to calculate the bound is given below:

bound = $p + (W-w)(p_{i+1}/w_{i+1})$

The bound at any node is addition of profit p, the total profits of already selected items with the product of the left over capacity of knapsack (W-w) and the best profit per weight unit which is :

p_{i+1}/w_{i+1}.

Example: Given a knapsack problem instance, apply the branch and bound to find a subset which gives the maximum profit. The following is a problem instance:

Object	Profit(Rs.) (p)	Weight(w)	p/w
1	40	4	10
2	42	7	6
3	25	5	5
4	12	3	4

Knapsack capacity = 10

The following is the state space tree (figure 4)of representing the given instance of knapsack problem:

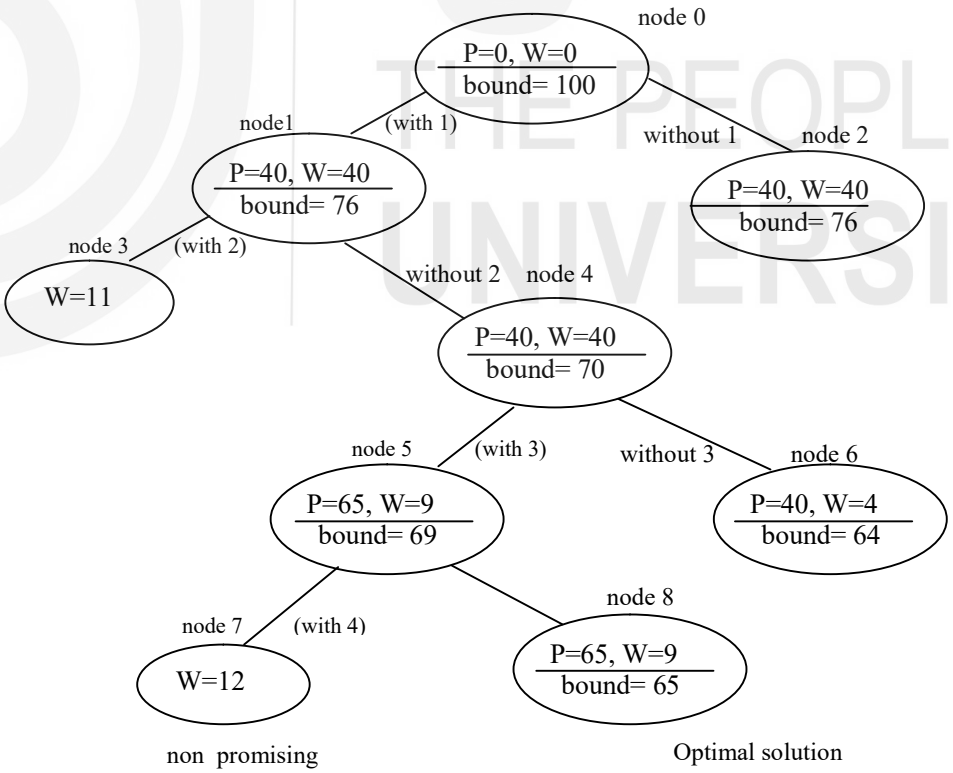


Figure 3.4: State space tree for knapsack problem using branch and bound technique

The root node indicates that no items have been selected as yet. Therefore p=0 and w= 0 and the bound is computed as per equation (i) which is Rs.100(0+ (10-0) (40/4)). The left branch of the tree includes the item1 and which is the only item in the subset while the right branch excludes the item 1. The total profit p and weight w is Rs 40 and 4 respectively. The bound value is 76 (40 + (10 -4)(42/7)). All three values are shown at node 1. Node 2 at the right branch excludes item 1.Therefore p = 0 and w = 0, because no item has been selected in the subset at node 2. Bound at this node is60(0 + (10-0) (42/7)). Compared to node 2, node 1 is more promising for optimization because it has a larger bound. Node 3 and node 4 are child nodes of node 1 have a subset with item 1 and item 2or without item 2 respectively. Let us calculate the total profit(p), total weight(w) and bound at node 3 first. Since w(total weight) of a subset represented by a node 3 exceeds 10 which is a capacity of the knapsack , this node is considered non promising. Since item 2 is not included at node 4 values of p and w are the same as its parent node 2 i.e., p= 40 , w= 4 . The bound= Rs.70(40 + (10-4)(25/5)). Compared to node 2, node 4 is selected for expanding the state space tree because its bound is larger than node 2. Now we move to node 5 and 6 which represent subsets including and excluding item 3 respectively. The total profit(p) , total weight (w) and bound at node 5 are:

p(item1 + item3) = Rs. 40 +Rs 25 = Rs.65

w(item 1 +item3) = 4 + 5 = 9

bound= 65+ (10 -9)(12/3)= 69

We will repeat the computation of p, w and bound at node 6:

P(item 1) = 40

w(item 1) = 4

bound = 40 + (10 – 4)(12/3) = 64

We will continue with node 5 because of its larger bound. Node 6 and node 7 represent a subset with and without item 4 respectively.

Node 7 is a non promising node because its total weight is 12(9 +3) exceeding the knapsack capacity.

p and w at node 8(without item 4) are 65 and 9 respectively, same as its parent

bound = 65 + (10-9)(0) =65

There is a single subset {1,3) represented by node 8. The other two nodes 2 and 6 have lower bounds than node 8. Hence node 8 is the final and optimal solution to the knapsack problem.

Check Your Progress -1

Q1. Define intractable problems in computer science.

Q2 What are the three distinct features of branch and bound technique?

Q3 How is backtracking different from branch and bound technique?

3.3 APPROXIMATION ALGORITHMS BASICS

As discussed in the previous unit, the decision version of hard combinatorial problems belongs to NP- complete complexity class whereas the optimization version of combinatorial problems is in **NP-hard** complexity class. Problems in this category are as difficult as NP-complete complexity class. There is no solution to these problems in polynomial time. Although, they are of practical significance.

The branch and bound technique which was discussed in the previous section is a major breakthrough because it is applicable to hard combinatorial problems with large problem instances. But the main issue is that such good performance cannot be guaranteed.

In this section we present a radically approach to deal with hard combinatorial problems called approximation algorithms which provide a solution to the hard combinatorial problem, reasonably close to optimal solution but in polynomial time. In real life situations, we usually work with inaccurate data. In such cases, getting near optimal solution is accepted and good enough. We require to find a bound that provides a measure how close a proposed solution is to optimal solution. For example, consider a TSP(Traveling Salesperson Optimization Problem) problem which is NP-Hard .

Further suppose that the nodes correspond to points in an Euclidean space, e.g., a 3D room, and the distance between any two points is their Euclidean distance.
There is an algorithm whose solution has the following guarantee: approxvalue is less than twice optimal value (i.e.<2xoptvalue) where, optvalue is the optimal solution for the problem and approxvalue is the approximate solution that the algorithm outputs.

3.3.1 Approximation Ratio

To precisely understand the notion of approximation algorithm, we define the term approximation ratio which can be defined as a ratio between the results obtained by the approximation algorithm and the optimal results of the algorithm.
Consider a minimization optimization problem P in which the main task is to minimize the given objective function. For example, vertex cover problem, TSP problem, graph coloring problem, etc. are combinatorial minimization optimization problem. Assume C is the value returned by a proposed polynomial time approximation algorithm for any such optimization problem on some problem instances whose size is n. Let C^* be the optimal solution for the given problem. Then the approximation ratio$\rho(n)$ is defined as :
$\rho(n)=\frac{C}{C^*}$
For a minimization problem, $C^*$is less and equal to ($\leq$)C and the approximation ratio $\frac{C}{C^*}$ defines the factor by which the cost of the polynomial approximate solution is larger than the cost of the optimal solution. Similarly for maximization problem, C is less and equal ($\leq$)C^* and the approximation ratio $\frac{C^*}{C}$ defines the factor by which the cost of an optimal solution$C^*$is larger than the proposed solution C.

It should be noted that the ratio, as defined above, is always at least 1. A ratio equal to 1 indicates the value returned by the algorithm is always equal to the optimal value, in other words, the algorithm solves the problem exactly. Since we want approximation algorithms that run in polynomial time, under the belief that P is different from NP, it will be not be possible to design an approximation algorithm in polynomial time for any NP-complete problem with 1 approximation ratio. The objective, therefore, becomes to design algorithms with a ratio that is very close to 1.

Check Your Progress-2

Q1 Write two examples of combinatorial optimization problems where approximation ratio can be used for finding near optimal solution.

Q2Define the approximation ratio of a minimization optimization problem. Explain the implication of an algorithm with a ratio that is very close to 1.

3.4 SUMMARY

Backtracking and branch and bound techniques apply some kinds of intelligence on the exhaustive search. It takes exponential time in the worst case for the solution of difficult combinatorial problems but it provides efficient solution if it has good design. But unlike exhaustive search, these techniques construct the solutions step by step, (one element at a time) and perform evaluation of the partial solution. If no solution is achieved by a given result, the tree is not further expanded.

State space tree is a principal mechanism employed by both backtracking and branch and bound techniques which is a binary tree where nodes represent partial solutions. Expansion of the tree is stopped as soon it is ensured that no solutions can be achieved by considering choices that correspond to the node's descendants.

Approximation algorithms are often applied to find near optimal solution to NP-hard optimization problems. Approximation ratio is the main metric to measure the accuracy of the solution to combinatorial optimization problems.

3.4 SOLUTION TO CHECK YOUR PROGRESS

Check Your Progress -1

Q1. Define intractable problems in computer science.

Ans. A problem in computer science is intractable if it cannot be solved it in polynomial time. These problems are hard problems which usually take exponential time in the worst case.

Q2 What are the three distinct features of branch and bound technique?

Ans. There are three distinct features of branch and bound technique: (i) unlike backtracking which traverse the tree in DFS, branch and bound does not restrict traversing in a particular way (ii) branch and bound technique computes a bound (value) of a node to decide whether the node is promising or not promising (iii) generally used for optimization problems

Q3 How is backtracking different from branch and bound technique?

Ans. They differ in terms of types of problems they can solve and how nodes in the state space tree are generated?

(i) Backtracking generally does not apply to optimization problems whereas branch and bound technique can be used to solve optimization problems because it computes a bound on the possible values of the objective function.

(ii) Depth first search is used to generate a tree in backtracking whereas there is no such restriction applied to branch and bound to generate a state space tree.

Check Your Progress-2

Q1 Write two examples of combinatorial optimization problems where approximation ratio can be used for finding near optimal solution.

Ans. 1 (i) Traveling Salesperson Problem- Given a weighted directed graph, the traveling salesperson optimization problem determines a path that starts and ends at the same vertex and visits all the vertices of the graph only once with minimal total weight.

(iii) Vertex Cover optimization problem- Given an undirected graph, the vertex cover optimization problem selects the minimum subset of vertices such that every edge in the input graph contains at least one selected vertex

Suppose an input graph G = (V,E)

The vertex cover optimization problem finds a subset S □ V such that for every edge (x, y) ∈ E ,either x ∈ S or y ∈ S. S should have minimum number of vertices.

Q2 Define the approximation ratio of a minimization optimization problem. Explain the implication of an approximation algorithm with a ratio that is very close to 1.

Ans. a minimization problem ,$0 < C^* \leq$ C and the approximation ratio = $\frac{C}{C^*}$defines the factor by which the cost of the polynomial approximate solution is larger than the cost of the optimal solution. When the approximation ratio is close to 1 indicates the value returned by the algorithm is always equal to the optimal value, in other words, the algorithm solves the problem exactly.