12.6 Non-polynomial Time Complexity

Non-deterministic polynomial time is abbreviated as NP time. For these types of problems if we are able to certify a solution anyhow then it becomes easy to prove its correctness using the polynomial time algorithm. For this reason, NP class problems are called polynomial time verifiable. In NP problem, it is allowed to make a guess about the answer, and then to verify the correctness of the answer done in polynomial time. Let us consider the case of the vertex cover problem. A vertex cover of a graph G is the set of vertices C such that each edge of G is incident to at least one vertex of C. If a set of vertices is chosen, it can be easily checked in polynomial time whether the set of vertices is a vertex cover of G or not. To find the minimum, such vertex cover is called minimum vertex cover problem. This problem is in class NP.

NP is the set of all decision problems that can be solved in polynomial time on a non-deterministic Turing machine. Already, we have discussed about deterministic and non-deterministic Turing machines. The difference between a deterministic and non-deterministic Turing machine is that the deterministic one operates like a conventional computer, performing each instruction in sequence, forming a computational path, whereas a non-deterministic Turing machine can form a 'branch off' where each branch can execute a different statement in parallel, forming a computational tree. The time required to solve the NP problem using any currently known algorithm increases very quickly as the size of the problem grows.

Definition: NP is defined as the set of all decision problems for which an algorithm exists which can be carried out by a non-deterministic Turing machine in polynomial time.

Consider some examples.

Example 12.13 **Prime Factorization:** A number N can be de-

composed into prime numbers where $N \geq 2$ such that $N = p_1 \times p_2 \times .....p_k$, where pi are k prime numbers. But there are some numbers which have only two prime factors. 'Prime factorization' is finding which prime numbers multiply together to make the original number.

Consider a number N, which has to be broken into two factors. The process is

i) Guess two numbers A and B.

ii) Multiply B times A, and check whether the result matches with N.

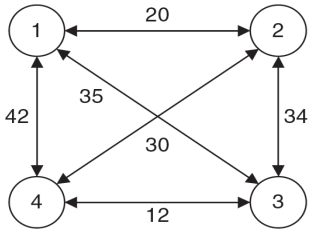
The checking can be performed in polynomial time.

The algorithm seems to be easy, but not really. In 2009, several researchers factored a 232-digit number, utilizing several hundreds of machines. It took more than 2 years. These types of large numbers left for prime factorization are called RSA number. There are huge amounts of cash prizes for finding the factors of these large numbers! Prime factorization of such large numbers is used for RSA public key cryptography.

Example 12.14 **Traveling Salesman Problem:** The problem is to find the minimum distance covered by a salesman who wishes to travel around a given set of cities, travelling each city exactly once and returning to the beginning.

Let us consider that an area of travelling consists of 4 cities as given in the following. The cost (distance) to travel from one city to another is given in the following matrix. We are assuming that the cost of ith node to jth node is the same as the cost of jth to ith. [They may be different (if the road is one way).]

1 picture



	1	2	3	4
1	0	20	35	42
2	20	0	34	30
3	30	34	0	12
4	41	30	12	0

Consider a tour starts and ends on node 1. From node 1, the person has to follow the path going to any of 2, 3, or 4 and returning to 1. In general, we can say that the tour consists of an edge $\langle 1, k\rangle, k \in V-\{1\}$ (V is the set of all vertices) and a path from k to 1. According to the condition of the problem, the salesman can travel each city exactly once.

Thus, the path from k to 1 must go through all the vertices in $V - \{1, k\}$, and this path must be the shortest. Let the length of the shortest path starting from node i and covering all the nodes exactly once and ending on node I be $l(i, S)$. If $i = 1$, then the function becomes

$$l(1, V - \{1\}) = \underset{2 \le k \le n}{min}\{c_{lk} + 1(k, V1, k)\}$$

c_{1k} is the cost of node 1 to node k.
In general, it can be written as

$$l(i, S) = min\{\underset{j \in s}{c_{ij}} + 1(j, S - \{j\})$$

where $i \neq S$.
If $n = 4, i = 1$, then $S = \{2, 3, 4\}$.
So, $l(1, \{2, 3, 4\}) = min\{[c_{12}+l(2, \{3, 4\})], [c_{\{}13\}+l(3, \{2, 4\})], [c_{14}+$

$l(4, \{2, 3\})]\}$

$$l(2, \{3, 4\}) = min\{[c_2 3 + l(3, \{4\})], [c_2 4 + l(4, \{3\})]\}$$
$$l(3, \{2, 4\}) = min\{[c_3 2 + l(2, \{4\})], [c_3 4 + l(4, \{2\})]\}$$
$$l(4, \{2, 3\}) = min\{[c_4 2 + l(2, \{3\})], [c_4 3 + l(3, \{2\})]\}$$

$l(3, \{4\}) = min\{c_3 4 + l(4, \emptyset)\} = 12 + 41 = 53[l(4, \emptyset) = c_4 1]$

$l(4, \{3\}) = min\{c_4 3 + l(3, \emptyset)\} = 12 + 30 = 42[l(3, \emptyset) = c_3 1]$
$l(2, \{4\}) = min\{c_2 4 + l(4, \emptyset)\} = 30 + 41 = 71[l(4, \emptyset) = c_4 1]$
$l(4, \{2\}) = min\{c_4 2 + l(2, \emptyset)\} = 30 + 20 = 50[l(2, \emptyset) = c_2 1]$
$l(2, \{3\}) = min\{c_2 3 + l(3, \emptyset)\} = 34 + 30 = 64[l(3, \emptyset) = c_3 1]$
$l(3, \{2\}) = min\{c_3 2 + l(2, \emptyset)\} = 34 + 20 = 54[l(2, \emptyset) = c_2 1]$

$$l(2, \{3, 4\}) = min\{[34 + 53], [30 + 42]\} = min\{87, 72\} = 72$$
$$l(3, \{2, 4\}) = min\{[34 + 71], [12 + 50]\} = min\{105, 62\} = 62$$
$$l(4, \{2, 3\}) = min\{[30 + 64], [12 + 54]\} = min\{94, 66\} = 66$$

$l(1, \{2, 3, 4\}) = min\{[20+72], [35+62], [42+66]\} = min\{92, 97, 108\} = 92$

If the element of the set S is 3, there are 3 distinct choices for i. If $|S|$ is 2, there are 2 distinct choices for i. In general, for each value of $|S|$ there are $n - 1$ distinct choices for i.

Not including 1 and I, the number of distinct sets of size k that can be constructed is ${}^{n-2}C_k$. The number of steps required for TSP of node n is

2 picture

$$\sum_{k=0}^{n-2}(n-1)({}^{n-2}C_k) = (n-1)\sum_{k=0}^{n-2}{}^{n-2}C_k = (n-1)\left[{}^{n-2}C_0 + {}^{n-2}C_1 + \cdots\cdots + {}^{n-2}C_{n-2}\right]$$

We know that $(1+x)^n = {}^nC_0x^0 + {}^nC_1x^1 + {}^nC_2x^2 + \ldots\ldots + {}^nC_nx^n$ .If we put $x = 1$, it becomes 2^n.
Therefore, the simplification of the previous formula is $(n-1)2^{n-1}$.

The complexity of the problem is $O(n2^n)$, and it is an NP problem. The complexity of the problem is O(n2n), and it is an NP problem.

Example 12.15 **Vertex Cover Problem:** It is one type of optimization problem in computer science.

Given a graph G, the aim is to find the minimum vertex cover $C \subseteq G$.

Given a graph $G = \{V, E\}$ and an integer $k \leq V$, the problem is to find V', subset of at most k vertices, such that each edge of G is incident to at least one vertex in V'.

The algorithm can be constructed by assuming a set of k vertices C for a given graph G. We have to check whether there exists a vertex cover of $\leq k$ vertices.

The checking can be done by the following steps.

i) Check whether C contains at most k number of vertices.
ii) Check whether C is a vertex cover of G.

$$\begin{aligned}&InputG = \{V, E\}, C = \{V'\}, where|V'| \leq k\\ &for i = 1 to |E|\\ &\quad checktheedgesendsonV'i\\ &\quad removethemfrom\{E\}.\end{aligned}$$

End

Thus, this checking operation is performed in $O(|V'|+|E|)$ i.e. in polynomial time.

A graph with n number of vertices can have 2^n number of such vertex combination. For each of the combinations, the same checking algorithm is performed. Among these, the set of vertices with minimum number is the minimum vertex cover of G. So, a minimum vertex cover algorithm is carried out by a non-deterministic Turing machine in polynomial time.

12.7 Polynomial Time Reducibility

In a deterministic Turing machine for a single state and single input, there is only one move. Thus, a problem fed to a deterministic Turing machine is either solved or not. $Polynomial - timereduction$ is a reduction which is computable by a deterministic Turing machine in polynomial time. This is called polynomial time many-to-one reduction. A reduction which converts instances of one decision problem into instances of a second decision problem is called many to one reduction proposed by Emil Post (1944). Already we have learnt about Turing reduction where a problem A is reduced to a problem B, to solve A, assuming B is already known. Many-to-one reductions are stronger than the Turing reduction.

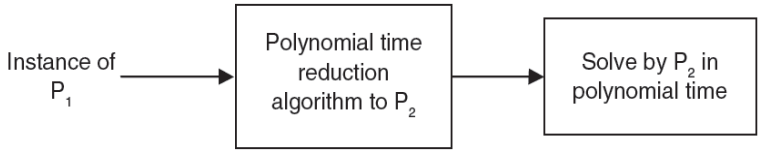
The polynomial time reducibility concept is used in the standard definitions of NP complete.

If P_1 and P_2 are two decision problems, then a problem P_1 is called polynomial time reducible to P_2 if there is a polynomial time algorithm which transforms an instance of P_1 to an instance of P_2.

*Theorem*1**:** If there is a polynomial time reduction from problem P_1 to problem P_2 and if problem P_2 is in P, then P_1 also is in P.

Proof : A Problem P_1 is polynomial time reducible to P_2 if there is a polynomial time algorithm which converts an instance of P_1 to an instance of P_2. Let the size of the input of P_1 be n. According to the condition of polynomial time reducibility, the instance of P_2 from the instance of P_1 can be obtained in polynomial time. Let the time complexity of this conversion be $O(n^i)$. According to the big oh notation, f(n) is said to be O(g(n)) if there exists a positive constant C and a value k such that $f(n) \leq Cg(n)$ for all $n \geq k$. Thus, the size of the converted instance of P_2 is less than or equal to Cn^i, where C is a constant.
The problem P_2 is in P. Let the complexity of solving a problem in P_2 be $O(m^k)$ where m is the size of the problem instance of P_2. Thus, the instance of P_2 of size Cn^i can be solved in complexity $O((Cn^i)^k)$. The total conversion time from the input in P_1 to the solution by P_2 (in other words the solution of problem P_1) takes $O(n^i + (Cn^i)^k) = O(n^i + C^k n^{ik}) = O(n^{ik})$. It is in polynomial time. It is proved that P_1 is in P.

3 picture



12.8 Deterministic and Non-deterministic Algorithm

An algorithm is called a deterministic algorithm if, for a function, the output generated for a given input is the same for all trials. The behaviour of this type of algorithm is predictable as the underlying machine follows the same sequence of

states for a given input to produce output. A mathematical function is deterministic. Hence, the state is known at every step of the algorithm.

An algorithm is called a non-deterministic algorithm if, for a function, the output generated for a given input is different for different trails. An algorithm that solves a problem in non-deterministic polynomial time can run in polynomial time or exponential time depending on the choices it makes during execution. An algorithm is non-deterministic if the underlying machine follows a different sequence of states for a given input. Some examples of non-deterministic algorithm are spanning tree construction algorithm, searching algorithm, merge sort algorithm, vertex set covering problems, etc. To know more about these problems, follow some books on algorithm.

12.8.1 Tractable and Intractable Problem

A problem is called a tractable problem if there is a polynomial time algorithm to solve it for all instances. There exists a deterministic Turing machine to solve these problems. In other words, it can be said that a problem is called a tractable problem, when there exists a deterministic Turing machine M such that

- M runs in polynomial time on all input
- for $x \in L$, M gives output 1
- for $x \neq L$, M gives output 0.

A tractable problem may be of constant [$O(1)$], logarithmic [O(log n) or $O(n \log n)$], linear [$O(n)$], or polynomial [$O(n^k)$, where k is a finite number]. The upper bound of a tractable problem is O(polynomial).