Indian Institute of Information Technology
Design and Manufacturing, Kancheepuram
Chennai 600 127, India
An Autonomous Institute under MHRD, Govt of India
http://www.iiitdm.ac.in
Instructor
N.Sadagopan

COM 501 Advanced Data Structures and Algorithms

More on NP and Reductions

1 On Spanning Trees

The class NP (Non-deterministic Polynomial) is the set of decision problems that are solvable using non-deterministic polynomial-time algorithms or verifiable using deterministic polynomial-time algorithms. Consider the classical spanning tree problem; its optimization, decision and verification versions are defined as follows

Optimization version `Input:` A connected weighted graph G
`Question:` Does there exist a spanning tree T of G whose weight is minimum.

Decision version `Input:` A connected weighted graph G, integer k
`Question:` Does there exist a spanning tree T of G whose weight is at most k.

Verification version `Input:` A connected weighted graph G, integer k, Set $E' \subseteq E(G)$
`Question:` Does E' induce a spanning tree of G whose weight is at most k.

We shall now show that using decision version as a black box, one can solve the optimization version. In particular, we make polynomial number of calls to the decision black box, to get the value of 'minimum' and the tree itself.

Obtaining the value of 'minimum' using the decision black box

1. Invoke the decision box with the input $(G, k = l)$, where l equals the sum of the edge weights. If the box returns YES, then
2. Invoke the decision box with the input $(G, l-1)$, If the box returns YES, then call again with the input $(G, l-2)$.
3. By calling the box iteratively, say r times, with each call decrements the value of k by one, we notice that the first $r-1$ calls return YES and the r^{th} call returns NO. This implies that the value of minimum is $l-(r-1)$.
4. To speed up the above process, one use an approach similar to binary search instead of linear search. That is, the first call is (G, l), if it returns YES, then the second call is $(G, \frac{l}{2})$. If it returns NO, then binary search is applied between $\frac{l}{2}$ and l. This ensures, we make polynomial number of calls to the decision box to determine the value of minimum.

How to construct a minimum weight spanning tree using the decision black box

1. The objective is to determine the 'right' subset of edges that form a minimum weight spanning tree. This means that, edges that do not participate in the minimum weight spanning tree can be pruned. Let $m = l-(r-1)$ be the value of minimum obtained from the above process. Let $E(G) = \{e_1, \ldots, e_p\}$.
2. Invoke the decision box with the input $(G-e_1, m)$. If the box returns YES, then there exists a spanning tree of weight m without the edge e_1. Proceed to the next step.

3. Invoke the decision box with the input $(G - e_1 - e_2, m)$. If the box returns NO, e_2 is part of the minimum weight spanning tree and hence, cannot be pruned.

4. Invoke the decision box with the input $(G - e_1 - e_3, m)$. If the box returns YES, then e_3 can be pruned. Similarly, we explore the other edges of $E(G)$. Calls to black box with removal of those edges that participate in the minimum weight spanning tree result in NO and for the other edges it is YES.

5. It is easy to see that, we get NO for $n-1$ edges and YES for $p-(n-1)$ edges, where n is the number of vertices.

6. The $(n-1)$ edges for which the box returned NO form a spanning tree of weight m.

The above exercise shows that a solution to an optimization problem can be obtained from the corresponding decision problem by incurring a polynomial time effort. This is true for any optimization problem and hence, the set NP is defined for the class of decision problems. Moreover, problems such as 3-SAT, Hamiltonian cycle (path) are by definition decision problems and hence it is appropriate to focus on decision problems to define a complexity class.

About verification algorithms: For many combinatorial problems, finding a solution is a challenging task. In particular, run-times of such algorithms are exponential in nature. For such problems, we ask a relatively easier question: if we some how get a certificate (solution set), can we verify the certificate efficiently, using a polynomial-time algorithm. This is precisely the goal of any verification algorithm. For the spanning tree problem, the verification algorithm consists of G, k, E', where E' is the certificate, a subset of edges of G. The goal is to check whether E' induces a spanning tree with weight at most k. Clearly, this check can be done using the standard BFS, and hence the verification can be done in polynomial time.

1.1 Spanning tree variant (Steiner Tree)

In this section, we shall generalize the classical spanning tree problem. We shall focus on unweighted graphs and consider generalizing the simpler version: Input: A connected graph G, Question: Find a spanning tree of G.

Optimization version `Input:` A connected graph G, $R \subseteq V(G)$
`Question:` Does there exist a tree T containing all of R such that $V(T) \setminus R$ is minimum. That is, the number of additional vertices used is minimum.

Decision version `Input:` A connected graph G, $R \subseteq V(G)$, integer k
`Question:` Does there exist a tree T containing all of R such that $|V(T) \setminus R| \leq k$. That is, the number of additional vertices used is at most k.

Verification version `Input:` A connected graph G, $R \subseteq V(G)$, integer k, a set S
`Question:` Is the graph induced on the set $R \cup S$ connected and $|S| \leq k$.

Remarks:
1. In the above definition, if $R = V(G)$, then we get the classical spanning tree problem, and hence this definition clearly generalizes the definition of spanning tree problem.

2. If $R = \{u, v\}$, solution to the above problem is equivalent to finding a shortest path between the vertices u and v in G.

3. For arbitrary R, it is natural to think of greedy algorithms to obtain an optimum solution.

4. Greedy Strategy 1: Since the objective is to connect the elements of R using a minimum number of additional vertices; order the vertices of R as $(u_1, \ldots, u_l)$, further, for each adjacent pair find a shortest path $(u_i, u_{i+1}), 1 \leq i \leq l-1$. Clearly, R together with shortest paths induces a connected subgraph H. Perform BFS/DFS on H to obtain a tree. Intuitively, it is clear that the number of additional vertices used is minimum. However, in practice, it does not give optimum always.

5. Greedy Strategy 2: Construct a bipartite graph $H = (V_1, V_2)$, where $V_1 = R$ and $V_2 = V(G) \setminus R$. Add all edges of G to H except the edges whose end points are contained in V_2. Choose a vertex $u \in V_2$ of maximum degree and include in the solution set S. That is, u is a vertex seeing the maximum number of vertices in V_1. Add u to S and remove u and all the neighbors of u except one. Recompute the degree of each vertex in V_2 and repeat the above process till V_1 is empty. When we remove u and its neighbors, we leave one vertex in the neighborhood to ensure connectedness. Clearly, the graph induced on $R \cup S$ is connected. Here again, this greedy does not yield optimum always.

Steiner Tree is in NP

We shall show that the decision version of Steiner tree is in NP. We now present a non-deterministic polynomial-time algorithm to solve Steiner tree. Similar to vertex cover, independent set problems, we need to guess a subset $S \subseteq V(G) \setminus R$. Let $V(G) \setminus R = \{v_1, \ldots, v_p\}$. The NP machine starts guessing whether v_1 is part of the solution, then v_2 is part of the solution, and so on. After making $O(n)$ guesses in non-deterministic fashion, if the input instance is an YES instance, then the machine stops at the 'right' subset S. Further, the machine checks using a deterministic verification algorithm whether $R \cup S$ is connected and $|S| \leq k$.

Steiner Tree is NP-Hard

We shall present a polynomial-time reduction from EXACT-3-COVER (X3C) to STEINER TREE to establish this result. It is known from the literature that X3C is NP-Hard through a reduction from 3-SAT, which is the first problem known to be NP-Hard. Due to transitivity (composition of polynomials is again a polynomial), every problem in NP polynomially reduces to X3C, and inturn to Steiner tree.

EXACT-3-COVER
Input: A set $X = \{x_1, \ldots, x_{3q}\}$, q is an integer ; $C = \{C_1, \ldots, C_n\}$, $C_i \subseteq X$, $|C_i| = 3$
Question: Does there exist a $C' \subseteq C$ such that C' partitions X.
We now present a reduction: given an instance of X3C, the corresponding instance of Steiner tree is created as follows;
$V(G) = X \cup \{v\} \cup C$
There is vertex in G for each element in X and C, plus the additional vertex v. $E(G) = \{\{C_i, x_j\} \mid x_j \in C_i\} \cup \{\{v, C_i\}, 1 \leq i \leq n\}$.
Note that G is a bipartite graph with bipartition $V_1 = X \cup \{v\}$ and $V_2 = C$. Further, v is universal to C and x_j is adjacent to all C_i's in which it is part of.

Claim:
X3C if and only if (G, R, k), $R = X \cup \{v\}, k = q$.

Necessity: Since X3C is an YES instance, there exists $C' \subseteq C$ such that C' partitions X. Since each C_i in C is a 3-element subset of X, to partition X, we need q elements from C. That is, $|C'| = q$. Let $C' = \{c_1, \ldots, c_q\}$ be the solution to X3C. Clearly, the vertices $\{c_1, \ldots, c_q\}$ in G together with $R = X \cup \{v\}$ induces a connected subgraph. Hence, the necessity.

Sufficiency: Since (G, R, k) is an YES instance, there exists a set of k additional vertices using which R can be made connected. By the construction of G, it must be the case that all k additional vertices are from C. Since $|X| = 3q$, the additional vertices have degree 3 with respect to X and G is an YES instance (every element of X must be included in the solution), it must be the case that the neighborhood of each c_i in C must be distinct. Therefore, c_i's form the partition with respect to X3C. This completes the theorem. Thus, we conclude, Steiner tree is NP-Hard.

Remark:
1. The above reduction shows that Steiner tree is NP-Hard in general graphs, and in particular, NP-Hard in

Bipartite graphs. That is, solving an instance of bipartite graphs under any deterministic model will incur a time exponential in the input size, unless P=NP.

2. The above reduction does not comment about the complexity of Steiner tree in trees, planar graphs, and other special graphs.

3. The other special graphs might have polynomial-time algorithms or they remain NP-Hard. One must identify the 'right candidate' NP-Hard problem to comment about the complexity of Steiner tree in planar and other special graphs.

4. For trees, the Steiner tree problem is polynomial-time solvable using a simple greedy algorithm: given a tree and a set R, to find the solution set, remove iteratively the leaves that are not part of R. At the end of this task, we obtain a tree with all leaves contained in R and the internal nodes of the tree are the additional vertices. Clearly, this is minimum as removal of any internal node will disconnect the tree into forests.

5. Often, choosing the 'right' candidate problem to reduce it to problem of our interest is quite challenging. Further, the reduction, we come up with must be a solution preserving reduction. That is, for the above example, in the necessity part, we constructed the solution to Steiner tree using the solution to X3C, and similarly, the converse.

Solving Steiner tree optimization via decision black box

- Since the value of k cannot exceed n, we make a sequence of calls to the decision box with each call decrements the value of k by one. The box may return a sequence of YES ($r-1$ YES) followed by a NO, the last YES in the sequence precisely determines the value of minimum, which is $k-(r-1)$.

- To identify the right subset of size $k-(r-1)$ from $V(G) \setminus R$, we start pruning vertices that do not participate in an optimum solution. The first invocation: $(G-v_1, R, k-(r-1))$, if the box returns YES, then there is a solution without the vertex v_1. If it returns NO, the solution contains v_1 and hence cannot be pruned. An iterative application of the above rule for other vertices with the modified graph as the input at each iteration, we obtain the desired solution set. That is, retain vertices for which the box returns NO and prune the vertices for which the box returns YES.

1.2 Other Variants of Spanning Tree

- Spanning tree with maximum number of leaves:

  1. `OPT-version:` Input: A connected graph G, Question: Find a spanning tree with the maximum number of leaves

  2. `Decision-version:` Input: A connected graph G, integer k, Question: Find a spanning tree with at least k leaves

  3. `Verification-version:` Input: A connected graph G, integer k, $E' \subseteq E(G)$, Question: Is E' form a spanning tree with the number of leaves at least k.

- Spanning tree with minimum number of leaves:

  1. `OPT-version:` Input: A connected graph G, Question: Find a spanning tree with the minimum number of leaves

2. `Decision-version:` Input: A connected graph G, integer k, Question: Find a spanning tree with at most k leaves

3. `Verification-version:` Input: A connected graph G, integer k, $E' \subseteq E(G)$, Question: Is E' form a spanning tree with the number of leaves at most k.

- Spanning tree with exactly two leaves:

  1. `OPT-version:` No optimization in the question.
  2. `Decision-version:` Input: A connected graph G, integer $k = 2$, Question: Does there exist a spanning tree with exactly two leaves

  3. `Verification-version:` Input: A connected graph G, integer k, $E' \subseteq E(G)$, Question: Is E' form a spanning tree with the number of leaves equals two.

- Spanning tree T such that the maximum degree of T is at most 3.

  1. `OPT-version:` No optimization in the question.
  2. `Decision-version:` Input: A connected graph G, Question: Does there exist a spanning tree T whose maximum degree is at most 3.

  3. `Verification-version:` Input: A connected graph G, integer k, $E' \subseteq E(G)$, Question: Is E' form a spanning tree T whose maximum degree is at most 3.

All of the above variants are in NP. Since these variants ask for a subset of edges, the NP machine guesses the 'right' subset of edges. Towards this end, the machine guesses during the first iteration, whether e_1 is part of the solution; during the second iteration, it guesses whether e_2 is part of the solution, and so on. After m guesses, the machine guesses the 'right' subset; subsequently, it verifies whether E' is the desired set or not. Thus, the non-deterministic algorithm run in time polynomial in the input size.

We shall establish next the NP-Hardness of spanning tree variant using HAMILTONIAN PATH as the candidate NP-Hard problem.

`HAMILTONIAN PATH`
Input: A connected graph G
Question: Find a spanning path; a path visiting all vertices of G.

Note that the following problems are equivalent in terms of time complexity. Since Hamiltonian path is NP-Hard, the other two problems are also NP-Hard. This means, it is unlikely that these problems have deterministic polynomial-time algorithms.

- Finding a Hamiltonian path in a graph.
- Finding a spanning tree with exactly two leaves in a connected graph.
- Given a connected graph, finding a spanning tree whose maximum degree 2.

Determining Hamiltonian path using decision HPATH as a black box:

- By definition, HPATH is a decision problem and there is no parameter k associated with the problem. As before, we prune edges of the graph that will not participate in the Hamiltonian path. The first

call to decision box is with the input $(G - e_1)$, if the box returns YES, then there exists a Hamilton path without the edge e_1. If the box returns NO, then e_1 is part of the Hamilton path. For every other edge of G, similar calls are made to the box to determine whether the edge is part of the Hamilton path. There will be $n - 1$ calls for which the box return NO and the rest YES.

Note: Similar to the above discussion, the other variants such as maximum (minimum) leaf spanning tree, spanning tree with maximum degree 3 can be solved using its decision algorithm as a black box. This shows that if the decision box is solved efficiently (preferably, deterministic polynomial time), then the solution to the corresponding optimization problem can be obtained by spending an additional polynomial effort.

1.3 Complexity of Spanning cycle (path) - Yet Another Variant

In this section, we shall analyze the complexity of Hamiltonian cycle (spanning cycle) and Hamiltonian path (spanning path).
`HAMILTONIAN PATH`
Input: A connected graph G
Question: Find a spanning path; a path visiting all vertices of G.

`HAMILTONIAN CYCLE`
Input: A connected graph G
Question: Find a spanning cycle; a cycle visiting all vertices of G.

Reduction: HCYCLE $\leq^p$ HPATH.
Given an instance G of HCYCLE, we construct the corresponding instance H of HPATH as follows;
$V(H) = V(G_1) \cup V(G_2) \cup \ldots \cup V(G_m)$ where $V(G_i) = V(G) \cup \{x, y\}$
$E(H) = E(G_1) \cup E(G_2) \cup \ldots \cup E(G_m)$ where $E(G_i) = E(G) \cup \{\{x, z\}, \{y, w\}\}$ where $e_i = \{z, w\} \in E(G)$, $E(G) = \{e_1, \ldots, e_m\}$. That is, for each edge $e = \{z, w\}$, we create a graph by adding two additional vertices $\{x, y\}$ with edges $\{\{x, z\}, \{y, w\}\}$ and there are m such graphs.
Claim: HCYCLE if and only if HPATH

- Necessity: Since G has a hamiltonian cycle C, there exists an ordering among vertices, say $C = (v_1, v_2, \ldots, v_n)$. By the construction of H, we have created an instance G_i for each edge of G, and in particular, for the edge $\{v_1, v_n\}$. The cycle C can be extended by including x and y to get a Hamiltonian path $(x, v_1, \ldots, v_n, y)$ in H. Hence, the necessity.

- Sufficiency: Since H has a Hamiltonian path, at least one of G_i in H has a Hamiltonian path. Moreover, any Hamiltonian path P in G_i must start at x and end at y as x and y are vertices of degree one. Let v_i and v_j be the neighbors of x and y, respectively. Clearly, the Hamiltonian path P of G_i is a (v_i, v_j) Hamiltonian path in G, the corresponding instance. Moroever, (v_i, v_j) is adjacent, and hence, we get a Hamiltonian cycle in G. Thus, the claim is established.

One can also, reduce an instance of HPATH to an instance of HCYCLE, which we shall present next.
Reduction: HPATH $\leq^p$ HCYCLE.
Given an instance G of HPATH, we construct the corresponding instance H of HCYCLE as follows;
Similar to the above construction, for each *missing edge* $e \in (\{\{u, v\} \mid u, v \in V(G)\} \setminus E(G))$, we create an instance in H. Let $M = \{e_1, \ldots, e_r\}$ be the set of missing edges.
$V(H) = V(G_1) \cup V(G_2) \cup \ldots \cup V(G_r)$ where $V(G_i) = V(G)$
$E(H) = E(G_1) \cup E(G_2) \cup \ldots \cup E(G_r)$ where $E(G_i) = E(G) \cup e_i$, where $e_i \in M$. That is, for each missing edge (the edge in the underlying complete graph but not in the actual graph), we create an instance in H and there are r such instances.
Claim: HPATH if and only if HCYCLE

- Necessity: Let P be a Hamiltonian path in G. $P = (v_1, \ldots, v_n)$ be the ordering of vertices. If $\{v_1, v_n\} \in E(G)$, then we get a desired Hamiltonian cycle in H. Otherwise, by the construction, there is a G_i in H such that $\{v_1, v_n\} \in E(G_i)$. Thus P together with the missing edge is a Hamiltonian cycle in H.

- Sufficiency: Let $C = (v_1, \ldots, v_n)$ be a Hamiltonian cycle in H. If C has no missing edges, then we trivially obtain a HPATH. Otherwise, the cycle vertices can be re-ordered, so that (v_1, v_n) has the missing edge. By the construction, any C contains at most one missing edge. Thus, $(v_1, \ldots, v_n)$ without the missing edge is the desired Hamiltonian path in G. This establishes the claim.

1.4 Second best spanning tree

Let G be a connected graph and T_m be its minimum weight spanning tree. Let T represents the set of all spanning trees. The second best spanning tree T_s is the minimum weight spanning tree in the set $T \setminus T_m$. A trivial approach to find the second best spanning tree is to list all and pick the second best, however, this algorithm runs in exponential in n as there are at most n^{n-2} labelled spanning trees. We shall next present an algorithm using the minimum weight spanning tree as a black box, and hence, it incurs a polynomial-time effort.

1. For a connected graph G, compute a minimum weight spanning tree using Kruskal's (or Prim's) algorithm and let the tree be T_m. Let $E(T_m) = \{e_1, \ldots, e_{n-1}\}$ and $E_m = E(G) \setminus E(T_m)$, where E_m denotes the set of missing edges - the edges in G but not in T_m. To compute the second best spanning tree T_s, we perform the following;

2. For each edge $e \in E_m$, create the graph $H = T_m + e$. Clearly, the addition of edge e to T_m creates a cycle C and let $\{f_1, \ldots, f_p\}$ are the edges in C. Note that e is f_i for some i. Since T_m is a minimum weight spanning tree, it must be the case that e is an edge of maximum weight. Find, the second maximum weight edge in C, say, f_m. Consider the tree $T_c = H - f_m$. Clearly, T_c is a tree as H has exactly one cycle which is removed on deleting f_m. We perform the above task for each missing edge in E_m and we get m such candidate spanning trees for T_s.

3. The second best spanning tree is the tree of minimum weight in the above candidate set.

4. Similar procedure can be applied to obtain the third best spanning tree. The candidate spanning trees for the third best are (i) all spanning trees that contested for the second best spanning tree except T_s (ii) a new set of candidates obtained using T_s, i.e., with respect to T_s, find the missing edge set and create a new set of spanning trees using the missing edge set.

2 Complexity of Maximum Independent Set

In this section, we present a polynomial time reduction from 3-SAT to MAXIMUM INDEPENDENT SET (MIS). Since 3-SAT is NP-Hard, it follows that MIS is also NP-Hard. Alternately, if there is an efficient algorithm (preferably, a deterministic polynomial-time) to solve MIS, then that algorithm can be used as a black box to solve 3-SAT efficiently.
Reduction: 3-SAT $\leq^p$ MIS
Given an instance of 3-SAT, a boolean formula with n variables and m clauses with each clause contains exactly 3-variables, we shall construct the corresponding instance G of MIS as follows;

$V(G) = \{x_i^a \mid x_i \in C_a\}$ - for each variable appearing in clause C_a, there is a vertex in G, there are $3m$ vertices in G.

$E(G) = \{\{x_i^a, x_j^a\} \mid x_i^a, x_j^a \in C_a\} \cup \{\{x_i^a, x_j^b\} \mid x_i^a \in C_a, x_j^b \in C_b, a \neq b, x_i^a = \neg x_j^b\}$ - In G, we create a triangle for each clause C and further we add edges between a pair of variables across the clauses if one variable is complement of the other.

Claim: 3-SAT iff G has an MIS of size m

- Necessity: Given that the formula is satisfiable, there exists a truth assignment to $(x_1, \ldots, x_n)$ such that on application of $(x_1, \ldots, x_n)$ to the formula, the formula is evaluated to true. In particular, $C_1 \wedge \ldots \wedge C_m = 1$. That is, for each C_a, there exists $x_i^a \in C_a$ such that $x_i^a = 1$ Note that x_i^a could be a true literal (x_2) or a negated literal $(\bar{x_2})$). Consider the set $S = \{x_i^a \mid x_i^a = 1, \forall a\}$. By the construction, we can include exactly one variable from each clause to S and by our construction, a literal and its complement will not be included in S. Thus, S is an independent set of size m in G.

- Sufficiency: Let $S = \{u_1, \ldots, u_m\}$ be an independent set of size m in G. By the construction, it is clear that there does not exist C_a such that $u_i, u_j \in C_a, i \neq j$. This shows that each u_i belongs to exactly one triangle (C_a). Now, set $u_i = 1$ for all u_i in S. Clearly, this gives a truth assignment for the corresponding boolean formula. Hence, the corresponding 3-SAT is satisfiable. This completes the reduction.

Some interesting observations:

1. NP-Hardness theory focuses on the 'hardness'- how difficult the problem is in relation to other NP problems. It does not comment about the solvability of the problem, whereas NP is defined for solvable problems. Establishing NP-Hardness result does not imply the status of NP.

2. Unsolvable problems such as 'Halting problem' and 'post-correspondence problem' are NP-Hard as there is a reduction from 3-SAT. However, they do not belong to NP.

3. All problems in NP have a trivial deterministic algorithm, typically runs in time exponential in the input size. Therefore, every problem in NP is solvable.

4. A problem is NP-complete if it is NP-Hard and belongs to the set NP (the problem is complete for the complexity class NP). This shows that NP-complete problems are the most difficult problems in NP and solving one efficiently will yield efficient solutions to every other problem in NP.

5. Since the set of decision problems are more than the set of optimization problems, the study of NP theory focuses on the set of decision problems. Moreover, the black box technique helps to obtain the solution to optimization problem by incurring a polynomial-time effort.

6. Many interesting problems such as sorting, matrix multiplication, finding the square root of a number are polynomial-time solvable, however, they are not in Class P as these problems do not have the corresponding decision versions.

7. SORTING-VARIANT 1: Input: An array A, Question: Is there a way to sort A; SORTING-VARIANT 2: Input: An array A, Question: Is A sorted. These two questions are not same as the SORTING problem. The first variant is not even a decision question as the answer is always YES (for decision - the answer must be in binary).

8. Decision version exists for problems that have a natural optimization parameter given as part of the question. It is important to note that the following variant is a decision question, however, does not meet the definition of the classical Sorting problem. SORTING-VARIANT 3: Input: An array A, integer k, Question: sort A using at most k comparisons.

9. Since NP machines are hypothetical machines, it is natural to ask how much power can be given to these machines. (i) One can construct a machine that guesses the 'right subset' in $O(1)$ non-deterministic polynomial time (ii) the machine that guess the optimum solution in $O(n)$ time through a series of n guesses (iii) a machine where the non-determinism is only for guessing the 'right' subset and the verification is done using a deterministic polynomial time. Often, we work the NP machine of Type (iii), so that by giving less power to the machine, we can understand the complexity of problem better. It is important to highlight that, although, the verification is deterministic polynomial, the NP machine does this check on the fly (while it guesses a subset).

10. Most of the combinatorial problems that arise in practice fall into either SUBSET problems (vertex cover, clique) or PERMUTATION problems (HPATH). The NP machine as part of guessing explores all subsets or permutations before identifying the right one. It is a belief that this exponential exploration on the given instance is inevitable to find solutions to optimization problems.

11. GRAPH ISOMORPHISM: Input: Graphs G and H, Question: Is G isomorphic to H. This problem is known to be in NP; given a certificate (the mapping between the vertices of G and H), the certificate can be verified in deterministic polynomial time. However, the NP-Hardness status is not known nor a deterministic polynomial-time algorithm that solves the problem. The reason for no progress on NP-Hardness status is due to the challenge in identifying the right candidate NP-Hard problem.

12. Consider the following two reductions: **Reduction 1:** 3-SAT to SPATH, **Reduction 2:** SPATH to 3-SAT. We also know that 3-SAT is NP-Hard and SPATH (shortest path) is in Class P. As a consequence of the first reduction, we conclude that 3-SAT belongs to P and further, every problem in NP belongs to P. Thus P = NP. The second reduction reduces an easier problem to a difficult problem, and is of no use. The second reduction also tells us that if 3-SAT is in P then SPATH in P, which we already know independent of this reduction.