Table of contents:

I am covering NP-completeness and linear programming sections. But remember all chapters before Exam-II will be covered, in both True/False problems and long algorithm problems.

NP-completeness

- The **Hamiltonian Path** problem is: Given a graph G = (V, E), is there a path P in G that visits every vertex exactly once? The **Hamiltonian Cycle**

problem is: Given a graph G = (V, E), is there a cycle C in G that visits every vertex exactly once? Both problems are NP-complete.

- The **Longest Path** is the problem of deciding whether a graph G = (V, E) has a **simple path** of length greater or equal to a given number k.
- The **Minimum Leaf Spanning Tree** is the problem of deciding whether a graph G = (V, E) has a spanning tree T that contains at most k leaves? Prove it is NP-complete problem.

Questions:

a) Prove **Longest Path** and **Minimum Leaf Spanning Tree** are NP
b) Prove that **Hamiltonian Path** is reducible to **Longest Path**
c) Prove that **Hamiltonian Path** is reducible to **Minimum Leaf Spanning Tree**
d) Prove that **Hamiltonian Cycle** is reducible to **Longest Path** directly

Longest Path: Given ordered list of nodes on a path of length >= k, polynomial time Certifier:

- check that the length of the path is at least k in O(n)
- check that nodes do not repeat in O(n log n) or O(n)
- check that there are edges between every two adjacent nodes on the path in O(n)

Minimum Leaf Spanning Tree: Given a subgraph G' = (V, E'), polynomial time Certifier:

- check if <=k nodes in G' connect to only one edge.
- check if all nodes are connected, if G' is a tree (no loop) through one run of DFS, in O(|V|+|E|).

Proof of reducibility $Y \leq_p X$(in page 473, General Strategy for Proving New Problems NP-Complete)

1. Choose a problem Y that's known to be NP-complete
2. Consider an arbitrary instance s_Y of problem Y, and show how to **construct**, in **polynomial** time, an instance s_X of problem X that satisfies the following properties:
    a. If s_Y is "yes" instance of Y,

then s_X is "yes" instance of X

    b. If s_X is "yes" instance of X,

then s_Y is "yes" instance of Y

$s_Y$is an arbitrary instance of Y
$s_X$is a special instance of X, constructed given s_Y

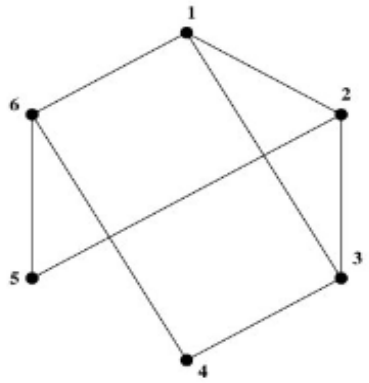
a) Hamiltonian Path is reducible to Longest Path

Given an instance of Hamiltonian Path on a graph G = (V, E) . We create an instance of the longest path problem G', k as follows: we use exactly the same graph G' = G and we set k = |V| - 1.

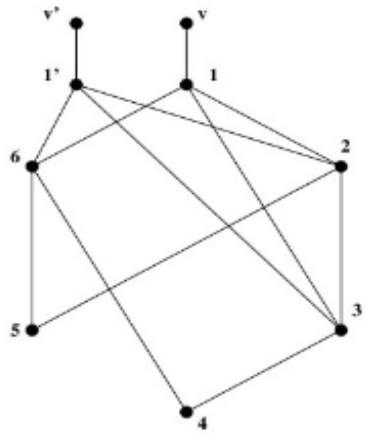
Then there exists a simple path of length k in G' if and only if G contains a Hamiltonian path.

b) Hamiltonian Cycle is reducible to Longest Path

Given a graph G= (V,E) we construct a graph G' such that G contains a Ham cycle iff G' contains a simple path of length at least N+2. This is done by choosing an arbitrary vertex u in G and adding a copy, u', of it together with all its edges. Then add vertices v and v' to the graph and connect v with u and v' to u'. We give a cost of 1 to all edges in G'. See figure below:



G



G'

A) If there is a HC in G we can find a simple path of length n+2 in G': This path starts at v' goes to U' and follows the HC to u and then to v. The length is n+2

B) If there is a simple path of length n+2 in G', there is a HC in G: This path must include nodes v' and v because there are only n nodes in G and a simple path of length n+2 must include v and v'. Moreover, v and v' must be the two ends of this path, otherwise, the path will not be a simple path since there is only one way to get to v and v'. So, to find the HC in G, just follow the path from u' to u.

c) Hamiltonian Path is reducible to Minimum Leaf Spanning Tree:

Given an instance of Hamiltonian Path on a graph G = (V, E). We create an instance of the minimum leaf spanning tree problem G', k as follows: We use exactly same graph G' = G, and set k = 2. A tree with 2 leaves is a path, so a spanning tree with two leaves is a Hamiltonian Path in the graph. So there exists a Hamiltonian path if and only if there exist a spanning tree with 2 leaves.

Linear Programming

A company makes three products and has 4 available manufacturing plants. The production time (in minutes) per unit produced varies from plant to plant as shown below:

		Manufacturing Plant			
		1	2	3	4
Product	1	5	7	4	10
	2	6	12	8	15
	3	13	14	9	17

Similarly the profit ($) contribution per unit varies from plant to plant as below:

		Manufacturing Plant			
		1	2	3	4
Product	1	10	8	6	9
	2	18	20	15	17
	3	15	16	13	17

If,

- one week, there are 35 working hours available at each manufacturing plant
- we need at least 100 units of product 1, 150 units of product 2 and 100 units of product 3.

how much of each product should be produced to gain most profit? Formulate this problem as a linear program. You do not have to solve the resulting LP. *In this problem assume the No. of product could be float.*

Variables:

let xij= amount of product i (i=1,2,3) made at plant j (j=1,2,3,4) per week.

Objective: Maximize total profit.

Maximize

10x11 + 8x12 + 6x13 + 9x14 + 18x21 + 20x22 + 15x23 + 17x24 + 15x31 + 16x32 + 13x33 + 17x34

Constraints:

We first formulate each constraint in words and then in a mathematical way.

- Limit on the number of minutes available each week for each workstation

5x11 + 6x21 + 13x31 <= 35(60)
7x12 + 12x22 + 14x32 <= 35(60)
4x13 + 8x23 + 9x33 <= 35(60)
10x14 + 15x24 + 17x34 <= 35(60)

- Lower limit on the total amount of each product produced

x11 + x12 + x13 + x14 >= 100
x21 + x22 + x23 + x24 >= 150
x31 + x32 + x33 + x34 >= 100

- Do not forget that No. of products cannot be negative

x11>=0
…
x34>=0

NP-complete

Consider an instance of the Satisfiability Problem, specified by clauses C1, . . .Cm, over a set of Boolean variables x1, . . . xn . We say that the instance is **monotone** if each term in each clause consists of **nonnegated variables**; that is, each term is equal tox_i, for some i, rather than $\acute{x_i}$. For example, suppose we have the three clauses: $(x_1 \vee x_2), (x_1 \vee x_3), (x_2 \vee x_3)$. This is monotone, and obviously the assignment that sets all three variables to 1 satisfies all the clauses. But we can observe that this is not the only satisfying assignment; we could also have set x_1

and x_2 to 1, and x_3 to 0. Indeed, for any monotone instance, it is natural to ask how few variables we need to set to 1 in order to satisfy it.

Given a monotone instance of Satisfiability, together with a number k, the problem of Monotone Satisfiability with Few True Variables (MSFTV) asks: Provided m clauses, is there a satisfying assignment for the instance in which at most k variables are set to 1?
(1) Prove that MSFTV problem is NP.

(2) Prove MSFTV is NP-complete by proving that **Vertex Cover Problem** $\leq_p$ **MSFTV Problem**. (Hint: construct a graph with n nodes and m edges, could all edges be covered with at most k vertex?)

Solution:
(1) This problem is NP. Given a candidate assignment of Boolean variables' values, we just check each clause to see if every clause is True.

(2) We create a graph G=(V,E), where every v_i correponds to one variable x_i, and one edge e_l = (x_i, x_j) corresponds to a clause (x_i v x_j), and if any of {x_i, x_j} is value 1, then this edge is considered to be covered. Finding if there is a vertex cover with at most k (VCK) nodes (i.e. setting at most k of x_1, ... x_n value 1) is NP-complete (the "yes" or "no" solution is obvious after solving the standard vertex cover problem). We can claim that answer to MSFTV instance is "yes" if and only if answer to VCK instance is "yes".

1> If MSFTV instance is "yes": since each clause is True, every edge connect at least one node whose corresponding value is 1, which means the edge is covered. Since at most k variables are 1, at most k nodes are selected.

2> If VCK instance is "yes": there is a vertex cover S in G of size at most k, and assign the corresponding variables in m clauses to 1. Since each ege is covered by a member of S, each clause contains at least one variable set to 1, and so all clauses are satisfied.

Thus, Vertex Cover Problem $\leq_p$ Path Selection Problem. Since Vertex Cover problem is NP-complete, and Path Selection problem is NP, Path Selection is NP-complete.

NP-complete

You are given two decision problems L1 and L2 in NP. You are given that L1 ≤p L2. For each of the following statements, state whether it is true, false or an open question. Justify your answers.

(a) If L1 ∈ P, then L2 ∈ P.

(b) If L1 ∈ NP-complete, then L2 ∈ NP-complete.

(c) If L2 ∈ P, then L1 ∈ P.

(d) Suppose L2 is solvable in O(n). Then, L1 is also solvable in O(n).

Solution:

(a) **Open question**. If P = NP, then L2 ∈ P. Otherwise, L2 can be P, NP or NP-complete.

(b) **True**. L2 is NP-complete, since we can convert any problem in NP to problem regarding L1 in polynomial time. Therefore, if we can reduce problems of L1 to problems in L2 in polynomial time, we can reduce all problems in NP to problems in L2. This is the technique we use to prove NP-completeness – by reducing an unknown problem to a known NP-complete problem.

(c) **True**. We can solve problems in L1 by first reducing it to problems in L2 in polynomial time, and then use the polynomial time algorithm for L2 solve the reduce problem. This gives a polynomial time algorithm for L1.

(d) **False**. The reduction L1 ≤p L2 may take more than linear time, therefore, we are not guaranteed a linear time algorithm for L1. We are guaranteed that L1 ∈ P can be solved in polynomial time – time to reduce L1 to L2 plus the time to solve problem in L2.

NP-complete concepts: (a post from stack exchange)

First of all, let's remember a preliminary needed concept to understand those definitions.

- **Decision problem**: A problem with a **yes** or **no** answer.

Now, let us define those *complexity classes*.

P

P is a complexity class that represents the set of all decision problems that can be solved in polynomial time. That is, given an instance of the problem, the answer yes or no can be decided in polynomial time.

NP

NP is a complexity class that represents the set of all decision problems for which the instances where the answer is "yes" have proofs that can be verified in polynomial time.

This means that if someone gives us an instance of the problem and a certificate (sometimes called a witness) to the answer being yes, we can check that it is correct in polynomial time.

NP-Complete

NP-Complete is a complexity class which represents the set of all problems X in NP for which it is possible to reduce any other NP problem Y to X in polynomial time.

Intuitively this means that we can solve `Y` quickly if we know how to solve `X` quickly. Precisely, `Y` is reducible to `X`, if there is a polynomial time algorithm `f` to transform instances `y` of `Y` to instances `x = f(y)` of `X` in polynomial time, with the property that the answer to `y` is yes, if and only if the answer to `f(y)` is yes.

It can be shown that *every NP problem can be reduced to 3-SAT*. The proof of this is technical and requires use of the technical definition of NP (*based on non-deterministic Turing machines*). This is known as *Cook's theorem*.

What makes NP-complete problems important is that if a deterministic polynomial time algorithm can be found to solve one of them, every NP problem is solvable in polynomial time (one problem to rule them all).

NP-hard

Intuitively, these are the problems that are *at least as hard as the NP-complete problems*. Note that NP-hard problems *do not have to be in NP*, and *they do not have to be decision problems*.

The precise definition here is that *a problem X is NP-hard, if there is an NP-complete problem Y, such that Y is reducible to X in polynomial time*.

But since any NP-complete problem can be reduced to any other NP-complete problem in polynomial time, all NP-complete problems can be reduced to any NP-hard problem in polynomial time. Then, if there is a solution to one NP-hard problem in polynomial time, there is a solution to all NP problems in polynomial time.