HW 1 (due Monday, Noon, September 7, 2015)

NEW CS 473: Theory II, Fall 2015 Version: **1.1**

Collaboration Policy: For this homework, Problems 1–2 can be worked in groups of up to three students. Submission is online on moodle.

---

1. (50 PTS.) Reduction, deduction, induction, and abduction.

The following question is long, but not very hard, and is intended to make sure you understand the following problems, and the basic concepts needed for proving NP-Completeness.

All graphs in the following have n vertices and m edges.

For each of the following problems, you are given an instance of the problem of size n. Imagine that the answer to this given instance is "yes", and that you need to convince somebody that indeed the answer to the given instance is `yes`. To this end, describe:

(I) An algorithm for solving the given instance (not necessarily efficient). What is the running time of your algorithm?

(II) The format of the certificate that the instance is correct.

(III) A bound on the length of the certificate (its have to be of polynomial length in the input size).

(IV) An efficient algorithm (as fast as possible [it has to be polynomial time]) for verifying, given the instance and the proof, that indeed the given instance is indeed `yes`. What is the running time of your algorithm?

We solve the first such question as an example.

(EXAMPLE)

Shortest Path

Instance: A weighted undirected graph G, vertices s and t and a threshold w.
Question: Is there a path between s and t in G of length at most w?

Solution:

(I) We seen in class the Dijkstra algorithm for solving the shortest path problem in $O(n \log n + m) = O(n^2)$ time. Given the shortest path, we can just compare its price to w, and return yes/no accordingly.

(II) A "proof" in this case would be a path π in G (i.e., a sequence of at most n vertices) connecting s to t, such that its total weight is at most w.

(III) The proof here is a list of $O(n)$ vertices, and can be encoded as a list of $O(n)$ integers. As such, its length is $O(n)$.

(IV) The verification algorithm for the given solution/proof, would verify that all the edges in the path are indeed in the graph, the path starts at s and ends at t, and that the total weight of the edges of the path is at most w. The proof has length $O(n)$ in this case, and the verification algorithm runs in $O(n^2)$ time, if we assume the graph is given to us using adjacency lists representation.

(A) (5 PTS.)

Semi-Independent Set

Instance: A graph G, integer k
Question: Is there a semi-independent set in G of size k? A set $X \subseteq \mathsf{V}(\mathsf{G})$ is semi-independent if no two vertices of X are connected by an edge, or a path of length 2.

(B) (5 PTS.)

3EdgeColorable

Instance: A graph G.
Question: Is there a coloring of the edges of G using three colors, such that no two edges of the same color are adjacent?

Subset Sum

Instance: S: Set of positive integers. t: An integer number (target).
Question: Is there a subset $X \subseteq S$ such that $\sum_{x \in X} x = t$?

(C) (5 PTS.)

3DM

Instance: X, Y, Z sets of n elements, and T a set of triples, such that $(a, b, c) \in T \subseteq X \times Y \times Z$.
Question: Is there a subset $S \subseteq T$ of n disjoint triples, s.t. every element of $X \cup Y \cup Z$ is covered exactly once.?

(See `https://en.wikipedia.org/wiki/3-dimensional_matching#Example` for an example.)

(D) (10 PTS.)

SET COVER

Instance: $(\mathsf{S}, \mathcal{F}, k)$:
S: A set of n elements
$\mathcal{F}$: A family of m subsets of S, s.t. $\bigcup_{X \in \mathcal{F}} X = \mathsf{S}$.
k: A positive integer.
Question: Are there k sets $S_1, \ldots, S_k \in \mathcal{F}$ that cover S. Formally, $\bigcup_i S_i = \mathsf{S}$?

(E) (10 PTS.)

CYCLE HATER.

Instance: An undirected graph $\mathsf{G} = (\mathsf{V}, \mathsf{E})$, and an integer $k > 0$.
Question: Is there a subset $X \subseteq \mathsf{V}$ of at least k vertices, such that no cycle in G contains any vertex of X.

(F) (10 PTS.)

Many Meta-Spiders.

Instance: An undirected graph $\mathsf{G} = (\mathsf{V}, \mathsf{E})$ and an integer k.
Question: Are there k vertex-disjoint meta-spiders that visits all the vertices of G?

A ***meta-spider*** in a graph G is defined by two vertices u, v (i.e., the head and tail of the meta-spider), and a collection Π of simple paths all starting in v and ending at u, that are vertex disjoint (except for u and v). The vertex set of such a spider is all the vertices that the paths of Π visit (including, of course, u and v).

2. (50 PTS.) Beware of algorithms carrying oracles.

Consider the following optimization problems, and for each one of them do the following:

(I) (2 PTS.) State the natural decision problem corresponding to this optimization problem.

(II) (3 PTS.) Either: (A) prove that this decision problem is NP-COMPLETE by showing a reduction from one of the NP-COMPLETE problems seen in class (if you already seen this problem in class state "seen in class" and move on with your life). (B) Alternatively, provide an efficient algorithm to solve this problem.

(III) (5 PTS.) (You have to do this part only if you proved that the given problem is NPC, since otherwise this is not interesting.) Assume that you are given an algorithm that can solve the **decision** problem in polynomial time. Show how to solve the original optimization problem using this algorithm in polynomial time, and output the solution that realizes this optimal solution.

An example for the desired solution and how it should look like is provided in the last page.

(A) (10 PTS.)

NO COVER

> **Instance**: Collection $\mathcal{C}$ of subsets of a finite set S.
> **Target:** Compute the maximum k, and the sets $S_1, \ldots, S_k$ in $\mathcal{C}$, such that $S \not\subseteq \cup_{i=1}^{k} S_i$.

(B) (10 PTS.)

TRIPLE HITTING SET

> **Instance**: A ***ground set*** $U = \{1, \ldots, n\}$, and a set $\mathcal{F} = \{U_1, \ldots, U_m\}$ of subsets of U.
> **Target:** Find the smallest set $S' \subseteq U$, such that S' hits all the sets of $\mathcal{F}$ at least three times. Specifically, $S' \subseteq U$ is a ***triple hitting set*** if for all $U_i \in \mathcal{F}$, we have that S' contains at least three elements of U_i.

(Hint: Think about the NPC problem HITTING SET.)

(C) (15 PTS.)

Max Inner Spanning Tree

> **Instance**: Graph $\mathsf{G} = (\mathsf{V}, \mathsf{E})$.
> **Target:** Compute the spanning tree $\mathcal{T}$ in G where the number of vertices in $\mathcal{T}$ of degree two or more is maximized.

(Hint: Think about the NPC problem Hamiltonian Path.)

(D) (15 PTS.)

Cover by paths (edge disjoint).

> **Instance**: Graph $\mathsf{G} = (\mathsf{V}, \mathsf{E})$.
> **Target:** Compute the minimum number k of paths (not necessarily simple) $\pi_1, \ldots, \pi_k$ that are edge disjoint, and their union cover all the edges in G.

(Hint: Think about the Eulerian path problem.)

Example for a solution for the second problem

You are given the following optimization problem:

Max 3SAT

> **Instance**: A boolean 3CNF formula F with n variables and m clauses.
> **Target:** Compute the assigned to the variables of F, such that the number of clauses of F that are satisfied is maximized.

Solution:

(I) The corresponding decision problem:

SAT Partial

> **Instance**: A boolean CNF formula F with n variables and m clauses, and a parameter k.
> **Question:** Is there an assignment to the variables of F that at least k clauses of F are satisfied (i.e., evaluate to true).

(II) The decision problem SAT Partial is NPC. Indeed, given a positive instance, a certifier would be the desired assignment, and this assignment can be verified in polynomial time, as such the problem is in NP. As for the completeness part, observe that there is an immediate reduction from SAT to this problem.

(III) Let **algSAT∂Sol** be the given solver for SAT Partial. Let F be the given input for Max SAT (i.e., the formula with n variables $x_1, \ldots, x_n$ and m clauses).
Let **algCount**(F') be a function that performs a binary search for the largest number k, such that **algSAT∂Sol**(F', k) returns true. This requires $O(\log m)$ calls to **algSAT∂Sol**, given that F' has m clauses.

The new algorithm **algMaxSAT** (F) is the following:

(1) If F is an empty formula, then return.
(2) $k \leftarrow$ **algCount**(F),
(3) Temporarily set $x_1 = 0$, and compute the resulting formula $F_{x_1=0}$ (all appearances of x_1 disappear, and all clauses that contain $\overline{x_1}$ are satisfied. Let k_0 be the number of clauses satisfied by this.
(4) $r_0 \leftarrow$ **algCount**$(F_{x_1=0})$.
(5) If $k_0 + r_0 = k$ then print "$x_1 = 0$", compute the remaining optimal assignment by calling **algMaxSAT**$(F_{x_1=0})$, Return.
(6) Temporarily set $x_1 = 1$, and compute the resulting formula $F_{x_1=1}$ (all appearances of $\overline{x_1}$ disappear, and all clauses that contain x_1 are satisfied.
(7) print "$x_1 = 1$"
(8) Call recursively **algMaxSAT**$(F_{x_1=0})$, Return.

The correctness of this algorithm is easy to see, so we do not elaborate. As for the running time, observe that **algMaxSAT** calls only a single recursive call (i.e., it is a tail recursion, which implies that the main body of this function is performed n times. Clearly, all other operations in this function, ignoring the time to call **algCount** takes $O(|F|)$ time, where $|F|$ is the length of F. **algCount** calls $O(\log m)$ calls to the oracle. As such, overall, the running time of this algorithm is $O(n|F| + Tn \log m)$, where T is the running time of the oracle.