ASSIGNMENT 5: Solutions CMSC 451 (Spring 2016)

1. *Search vs. decision problems.*

   As discussed in class, NP is a class of decision problems, i.e., the answer is either "yes" or "no." This choice is convenient and often also captures the difficulty of searching for a solution, as you will show in this problem for two examples.

   (a) *3-SAT.* A 3-SAT formula consists of m clauses, each of which is a disjunction of three terms (where a term is one of n variables or its negation). We say a 3-SAT formula φ is *satisfiable* if there is an assignment of the n variables such that φ evaluates to True. The 3-SAT problem asks us to decide whether a given φ is satisfiable.

   Suppose you are given a black box for a function 3SAT that determines whether a given 3-SAT formula is satisfiable. Describe an algorithm that finds a satisfying assignment for φ (assuming it is satisfiable) using polynomially many calls to 3SAT and polynomially many other steps. Prove that your algorithm is correct and analyze its running time. For full credit, your algorithm should use $O(n)$ calls to the black box.

   Solution: Let φ denote the input 3-SAT formula and let $x_1, x_2, \ldots, x_n$ be the n variables in φ. Since x_1 can be set to one of $\{\text{True}, \text{False}\}$ in any satisfying assignment, we first tentatively set $x_1 = \text{True}$ and obtain a new (smaller) 3-SAT formula φ' that does not use variable x_1 such that φ' is satisfiable if and only if there is a satisfying assignment of φ with $x_1 = \text{True}$. We then use the oracle to determine if φ' is satisfiable. If it is satisfiable, we finalize $x_1 = \text{True}$; otherwise, we finalize $x_1 = \text{False}$. We can now repeat this procedure for every variable until a complete satisfying assignment is found.

   The reduced 3-SAT formula φ' is constructed as follows. Initially, we set $\varphi' = \varphi$. We drop all clauses in φ' that contain the term x_1, as such clauses are satisfied once $x_1 = \text{True}$. We also drop the term $\bar{x}_1$ from all clauses that contain it. It is now easy to verify that φ' is satisfiable if and only if φ has a satisfying assignment with $x_1 = \text{True}$. Note that since we drop the term $\bar{x}_1$ from certain clauses, some clauses may now have fewer than three terms. We now add dummy variables to ensure that every clause has exactly three terms (refer to Kleinberg and Tardos, page 472 for details).

   input : Satisfiable 3-SAT formula φ
   output: Satisfying assignment for φ

   let $\varphi' = \varphi$;
   for $i = 1, 2, \ldots, n$ **do**
   - **if** $\mathsf{3SAT}(\varphi'|_{x_i=\text{True}}) = \text{True}$ **then**
     - **let** $\varphi' = \varphi'|_{x_i=\text{True}}$;
   - **else**
     - **let** $\varphi' = \varphi'|_{x_i=\text{False}}$;

   Algorithm 1: Find a satisfying assignment using a satisfiability oracle. In this algorithm, $\varphi|_{x_i=y}$ denotes the reduced formula obtained from φ by setting all instances of x_i to y and all instances of $\bar{x}_i$ to $\bar{y}$.

   Algorithm 1 shows the pseudocode. We make only n calls to the 3SAT oracle (once per variable). In each iteration, we spend $O(m)$ time to obtain the reduced instances. The total runtime is thus $O(m + n\mathsf{3SAT})$.

The correctness of the algorithm follows from the fact that we finalize an assignment only when we are guaranteed that there exists an assignment of the remaining variables.

(b) *3-Coloring.* We say an undirected graph is *3-colorable* if there is an assignment of the colors $\{r, g, b\}$ to the vertices (a *coloring*) such that no two adjacent vertices have the same color. The 3-Coloring problem asks us to decide whether a given graph is 3-colorable.

Suppose you are given a black box for a function 3Color that determines whether a given graph is 3-colorable. Describe an algorithm that finds a coloring of a given graph using polynomially many calls to 3Color and polynomially many other steps. Prove that your algorithm is correct and analyze its running time. For full credit, your algorithm should use $O(n)$ calls to the black box, where n is the number of vertices in the input graph.

Solution: Just as for the 3-SAT problem, the main idea is to "try" a color for each vertex successively and to finalize a color once the oracle guarantees that the remaining instance is still 3-colorable.

How can we force a vertex to take on a particular color? Observe that, by definition, any valid 3-coloring must assign three different colors to the vertices of a triangle. We thus add three new vertices $\{r, g, b\}$ to the graph and add edges $\{\{r, g\}, \{g, b\}, \{b, r\}\}$ to create a triangle. Since any valid coloring must assign different colors to these three vertices, without loss of generality, let vertex r be assigned color "red", vertex g be assigned color "green", and vertex b be assigned color "blue". If, for example, we want to check if there exists a valid 3-coloring in which vertex v is colored blue, we add edges $\{r, v\}$ and $\{g, v\}$ and ask the oracle if the graph is still 3-colorable. Algorithm 2 now shows the complete pseudocode.

```
input  : 3-Colorable graph G
output: Valid 3-coloring of G

Add a triangle on vertices {r, g, b} to G;
for i = 1, 2, ..., n do
    let G' be G with edges {r, v_i} and {g, v_i} added;
    if 3Color(G') = True then
        let col(v_i) = "blue";
        let G = G';
    else
        let G' be G with edges {g, v_i} and {b, v_i} added;
        if 3Color(G') = True then
            let col(v_i) = "red";
            let G = G';
        else
            let col(v_i) = "green";
            let G be G with edges {r, v_i} and {b, v_i} added;
```

Algorithm 2: Find a valid 3-coloring using a 3-colorability oracle.

We make at most 2 calls to the oracle for every vertex v_i in G and thus use $O(n)$ oracle calls in total. Also, we only perform a constant number of operations in each iteration; the total runtime is thus $O(n\text{3Color})$. The correctness of the algorithm follows from the fact that we finalize an assignment of colors only when the remaining graph is 3-colorable.

2. *Nonoverlapping Paths is NP-complete.*

Suppose you are given an undirected graph G and a set $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$, where each P_i is a simple path in G. We say a set of paths $\mathcal{Q} = \{P_{i_1}, P_{i_2}, \ldots, P_{i_t}\} \subseteq \mathcal{P}$ is *nonoverlapping* if no two paths in the set share a vertex. In the Nonoverlapping Paths problem, you are given a graph G, a set of paths $\mathcal{P}$ in G, and a positive integer t, and you are asked whether there is a nonoverlapping set of paths $\mathcal{Q} \subseteq \mathcal{P}$ with $|\mathcal{Q}| \geq t$.

(a) Show that Nonoverlapping Paths is in NP.

Solution: Given G and no more than t paths from $P_1, P_2, \ldots, P_k$ as a certificate, we can check whether these paths are non-overlapping by computing the intersection of the vertex set of each path in polynomial time. (Note also that we can easily check whether each path indeed belongs to $\mathcal{P}$.) Therefore Nonoverlapping Paths is in NP.

(b) Show that Nonoverlapping Paths is NP-hard.

Solution: We give a reduction from Independent Set to Nonoverlapping Paths. Given an instance of Independent Set $(G = (V, E), l)$, we construct an instance of Nonoverlapping Paths $(G' = (V', E'), P_1, P_2, \ldots, P_k, t)$ as follows:

- $V' = E \cup V$, i.e., for each edge $\{v_i, v_j\} \in E$, we create a vertex $v_{\{i,j\}}$ in V', and for each vertex $v_i \in V$, we create a vertex v_i in V'.
- For each $i \in \{1, \ldots, |V|\}$, we connect all vertices of form $v_{\{i,j\}}$ and v_i in an arbitrary order to form a path P_i.
- $t = l$.

From this reduction, we observe that vertices v_i and v_j are adjacent in G if and only if the corresponding paths P_i and P_j are overlapping in G'. Now suppose that G contains an independent set $S = \{v_{i_1}, \ldots, v_{i_l}\}$ of size l. The corresponding paths $P_{i_1}, \ldots, P_{i_l}$ now form a set of $l = t$ nonoverlapping paths in G'. On the other hand, let $P_{i_1}, \ldots, P_{i_t}$ denote a set of t nonoverlapping paths in G'. By construction, vertices $\{v_{i_1}, \ldots, v_{i_t}\}$ do not share any edges in G and thus form an independent set of size $t = l$. Thus, we have shown that G has an independent set of size l if and only if there are t non overlapping paths in the constructed instance.

Since the reduction only adds $|V|+|E|$ vertices and at most $O(\sum_i \deg_G(v_i)) = O(|E|)$ edges, we have a polynomial-time reduction from Independent Set to Nonoverlapping Paths and thus Nonoverlapping Paths in NP-hard.

3. *Restricted Monotone Satisfiability is NP-complete.*

We say an instance of Satisfiability is *monotone* if every term is a non-negated variable (i.e., x_i can appear as a term but $\bar{x}_i$ cannot). Monotone instances are always satisfiable since we can simply set every variable to True, but the problem becomes more difficult if we restrict the number of variables that can be True. In the Restricted Monotone Satisfiability problem, we are given a positive integer k and a monotone satisfiability instance (a set of clauses, each of which is a disjunction of non-negated variables), and we are asked whether there is an assignment of the variables with at most k variables set equal to True such that all clauses evaluate to True.

(a) Show that Restricted Monotone Satisfiability is in NP.

Solution: Given an assignment of truth values as a certificate, we can check in polynomial time that at most k variables are set to True and every clause is satisfied.

(b) Show that Restricted Monotone Satisfiability is NP-hard.

Solution: We give a reduction from Set Cover to Restricted Monotone Satisfiability. Given an instance of Set Cover $I = (\mathcal{S} = \{S_1, \ldots S_m\}, U = \{e_1, \ldots, e_n\}, l)$, we construct an instance of Restricted Monotone Satisfiability as follows:

- For each $i \in \{1, \ldots, m\}$, create a variable x_i.
- For each $j \in \{1, \ldots, n\}$, create a clause $C_j = (\bigvee_{a:\, e_j \in S_a} x_a)$, i.e., for every element in the Set Cover instance, create a clause that is a disjunction of all the sets that contain that element.
- $k = l$

Since the reduction only adds a variable for every set and a clause for every element, the reduction can be performed in polynomial time.

Now suppose that there is a collection of l sets $S_{i_1}, \ldots, S_{i_l}$ that covers all the elements. Then if we set $x_{i_1} = x_{i_2} = \ldots = x_{i_l} =$ True and all other variables to False, we obtain an assignment with $l = k$ True variables. Now consider any clause C_j. By definition, element $e_j \in S_{i_a}$ for some set in the selected set cover. Consequently, the clause C_j is satisfied as we have set $x_{i_a} =$ True.

On the other hand, suppose that there is a satisfying assignment with only k True variables. Selecting the sets corresponding to the True variables leads to a set cover of size $k = l$. This is because for every element e_j, there must be at least one selected set corresponding to the variable that satisfied clause C_j. Thus, we have a polynomial-time reduction from Set Cover to Restricted Monotone Satisfiability and hence Restricted Monotone Satisfiability is NP-hard.

4. *Max Cut.*

A *cut* in an undirected graph is a bipartition of the vertices. The *size* of a cut is the number of edges crossing the cut (i.e., the number of edges whose endpoints are on opposite sides of the cut). We say a cut is *maximum* if its size is as large as possible. The problem of finding a maximum cut is NP-hard. (You are not asked to prove this.)

Design a polynomial-time algorithm that finds a cut whose size is at least half as large as the maximum cut. Prove that your algorithm is correct and analyze its running time.

Solution: Starting from an arbitrary partition of the vertices into two sets A and B, we incrementally try to increase the number of cut edges by moving a vertex from one set to another. For any vertex $v \in V$ and a set $S \subseteq V$, let $\deg_S(v) = |S \cap N(v)|$ denote the number of vertices in set S that are adjacent to v. Observe that for a vertex $v \in A$, if $\deg_A(v) > \deg_B(v)$, then moving v to set B strictly increases the number of edges crossing the cut (A, B). Algorithm 3 details the complete pseudocode.

input : Undirected graph $G = (V, E)$
output: 0.5-approximate MaxCut

```
A = V, B = φ;
change = True;
while change == True do
    change = False;
    for v ∈ A do
        if deg_A(v) > deg_B(v) then
            A = A \ {v};
            B = B ∪ {v};
            change = True;
    for v ∈ B do
        if deg_B(v) > deg_A(v) then
            A = A ∪ {v};
            B = B \ {v};
            change = True;
```

Algorithm 3: Approximation Algorithm for MaxCut

Running Time: In each iteration of the **while** loop, the value of the cut $|\delta(A, B)|$, i.e., the number of edges crossing the cut, increases by at least 1. As a result, the **while** loop can run for at most $O(m)$ iterations. Each iteration takes $O(n)$ time to find a vertex that can be swapped and hence the algorithm runs in time at most $O(mn)$.

Approximation Factor: When the algorithm terminates, for every vertex $v \in A$, we have that $\deg_B(v) \geq \deg_A(v)$, so $\deg_B(v) \geq \frac{1}{2}\deg(v)$. Similarly, for every vertex $v \in B$, $\deg_A(v) \geq \deg_B(v)$ so $\deg_A(v) \geq \frac{1}{2}\deg(v)$ Now the number of edges crossing the cut (A, B) is given by

$$|\delta(A, B)| = \sum_{v \in A} \deg_B(v) = \sum_{v \in B} \deg_A(v).$$

Hence, we have

$$\begin{aligned} 2|\delta(A, B)| &= \sum_{v \in A} \deg_B(v) + \sum_{v \in B} \deg_A(v) \\ &\geq \sum_{v \in A} \frac{1}{2}\deg(v) + \sum_{v \in B} \frac{1}{2}\deg(v) \\ &= \frac{1}{2} \sum_{v \in V} \deg(v) \\ &= \frac{1}{2}(2|E|) \end{aligned}$$

Hence, we have

$$|\delta(A, B)| \geq \frac{|E|}{2} \geq \frac{Opt}{2}$$

The last inequality follows as the the total number of edges in the graph is a trivial upper bound on the maximum number of edges that can cross any cut. Thus we have shown that the size of the cut (A, B) found by our algorithm is at least half the size of the optimal cut.