CSC363H5 Winter 2016 Assignment 3

Name: Akhil Gupta
SN: 1000357071

Question #	Score
1	
2	
3	
4	
Total	

Acknowledgements:

”I declare that I have not used any outside help (excluding the textbook, the notes on the course website, the teaching assistants, and the instructor) in completing this assignment.”

Name: Akhil Gupta Date: March 15, 2016

Q1. In complexity theory we think of P as the polynomial-time analogue of D, NP as the polynomial-time analogue of SD, and $coNP$ as the polynomial-time analogue of $coSD$. In computability theory, it is not too hard to prove $D = SD \cap coSD$, however, the analogical statement in complexity theory

$$P = NP \cap coNP$$

is an open research problem! Let's study this a bit more closely.

Let PR be the class of *polynomial-time recognizable languages*. That is, a language L is in PR if and only if there is a Turing Machine M such that for all $x \in L$, M halts and accepts x in polynomial time, but for $x \notin L$ the machine M either rejects (with no bound on the running time) or does not halt. A language L is in $coPR$ if and only if its complement $\overline{L} \in PR$.

1. Show that $P = PR \cap coPR$.
   Answer: To show that $P = PR \cap coPR$, we have to show that $P \subseteq PR \cap coPR$ and $PR \cap coPR \subseteq PR$. To show that $P \subseteq PR \cap coPR$, we know that if a language is decided by a Turing Machine, that language is decidable and every decidable language is recognizable. Similary, if a language is decided by a Turing Machine in polynomial time ($\in P$), then that language is recognizable as well ($\in PR$). Therefore, $P \subseteq PR$. Additionally, from the question, we know that a language is $\in PR$ if and only if its complement is $\in PR$ and that $PR \subsetneq coPR$ and $coPR \subsetneq PR$ by definition. If its complement is in PR then that language is recognizable by a Turing Machine in polynomial time as well. Therefore, $P \subseteq coPR$. Hence, $P \subseteq PR \cap coPR$. Now we show that $PR \cap coPR \subseteq P$. Let L be a language such that $L \in PR \cap coPR$. Let M_1 be a Turing Machine that runs in polynomial time such that $L = \mathcal{L}(M_1)$. Let M_2 be a Turing Machine that also runs in polynomial time such that $\overline{L} = \mathcal{L}(M_2)$. We create a TM M which simulates M_1 and M_2. On input $x \in \Sigma^*$, the algorithm simulates M_1 on x and M_2 on x step-by-step using a multi-tape deterministic TM. It halts and outputs whatever the first simulation that halts does. Since the language $L \in PR \cap coPR$, for all $x \in L$, the Turing Machines M_1 and M_2 halt and accept x in polynomial time. So the input will either be accepted by M_1 in polynomial time or M_2 in polynomial time. Therefore, all the inputs will be decided by a determinisitic TM in polynomial time, therefore, the language $L \in P$. Therefore, $PR \cap coPR \subseteq P$. So, $P = PR \cap coPR$.

2. Show $PR \subseteq NP$ and $coPR \subseteq coNP$.
   Answer: To show that $PR \subseteq NP$, we need to show that an arbitrary language in PR is in NP. Hence we need to give a polynomial time verifier for it. We accomplish this by following a similar proof outlined in Theorem 2.10 of Professor Robert's lecture notes. Let L be any language and $L \in PR$ and let M_0 be a Turing Machine such that $\mathcal{L}(M_0) = L$. So we show that $L \in NP$ if and only if there exists a polynomial time verifier for it. The certificate for this problem will be interpreted by the verifier as a enconding of a Turing Machine M_0, an input x and a sequence of encodings of Turing Machine configurations with the last configuration in an accept state. The verifier first checks that the encodings of the Turing Machine, input and configurations are valid, if not returns 0. Then it simulates M_0 on x for one step and checks if the configuration of M_0 on x is encoded at the appropriate place in the certificate. If true, then the verifier returns 1 else it returns 0. Clearly the verifier runs in polynomial time in the length of its input as we assume that we can check encodings of mathematical objects in polynomial time and the TM M_0 runs in polynomial time (definition of a language in PR), and the verifier rejects as soon as the simulation

of M_0 on x exceeds the length of computation encoded by the certificate. Conversely, we know that if there exists a polynomial time verifier, then that language is in NP.

To show that $coPR \subseteq coNP$, we show a similar argument. We know that a language L is in $coPR$ iff its complement $\overline{L} \in PR$ and a language is in $coNP$ iff its complement is in NP. So essentially we need to show that for a language L, $\overline{L} \in PR$ and also in NP. We construct our Turing Machine M_1 such that $\mathcal{L}(M_1) = \overline{L}$. The accept and reject states of M_1 are the reject and accept states of M_0 respectively. So for all $x \notin L$, the TM M_1 halts and accepts x in polynomial time, but for $x \in L$, the machine either rejects or does not halt i.e. it accepts the complement of L. So our certificate to the verifier is the same and the verifier does the exact same procedure as above.

3. Discuss whether or not $NP \subseteq PR$ and $coNP \subseteq coPR$ (proofs are not necessary — just give your intuition!)

   Answer: To show that $NP \subseteq PR$, we have to show that any arbitrary language in NP is in PR. And to show that $coNP \subseteq coPR$, we hsve to show that any arbitrary language in $coNP$ is in $coPR$. For the first case, we showed above that $PR \subseteq NP$, so if $NP \subseteq PR$, then $PR = NP$. And we showed that $coPR \subseteq coNP$, so if $coNP \subseteq coPR$, then $coPR = coNP$. Both these cases are impossible. If $PR = NP$, then by definitions of PR and NP, we would be able to recognize a language and verify a certificate to a language in polynomial time If this were possible, then it would mean that we can both efficiently find and verify an algorithm in polynomial time which would basically break cryptography. Banks multiply 2 huge prime numbers in order to encode our credit-card information and it is extremely hard to factor the number back into primes, but it is easy to check that the 2 numbers are correct. This is the basis of P and NP problems. P is efficiently solvable, and NP is efficiently verifiable.

Q2. Let $G = (V, E)$ be a *directed* graph. A *directed clique* in G is a subset $V' \subseteq V$ such that for all $u, v \in V'$ there are two directed edges $uv \in E$ and $vu \in E$. The *Directed Clique* problem is the following: given a directed graph G and a positive integer k, decide if G contains a directed clique of size at least k.

Show that Clique $\leq_p$ DirectedClique and DirectedClique $\leq_p$ Clique.
Answer: We have to show that if we can solve Clique exactly then we can solve DirectedClique exactly and vice-versa.
We show DirectedClique $\leq_p$ Clique. Let $G = (V, E)$ be a directed graph and let k be a positive integer with $k \leq |G|$, we construct an undirected graph $G' = (V', E')$ such that G has a directed clique C of size at least k if and only if G' has a clique C' of size at least k. We know that for any 2 vertices $(u, v) \in V$, there are 2 directed edges $uv \in E$ and $vu \in E$. We create a undirected graph $G' = (V', E')$ with an edge $u'v' \in E'$ between any two vertices $(u', v') \in V'$ if and only if there are directed edges uv and vu in E for those 2 vertices $(u, v) \in V$. This comes from the fact that a directed edge uv and vu is equivalent to an undirected edge uv or vu. So if G had only one directed edge between any 2 vertices, then it would not constitute as a Clique because one of those 2 vertices will not be reachable and therefore will not be part of the Clique C'. So our undirected graph G' has a Clique C' that has the same size as a directed clique of size at least k. Hence, if a directed graph G has Clique C of size at least k then the undirected graph G' contains an undirected clique of size at least k.
We show Clique $\leq_p$ DirectedClique. Let $G = (V, E)$ be a undirected graph and let k be a positive integer with $k \leq |G|$, we construct a directed graph $G' = (V', E')$ such that G has a clique C of size at least k if and only if G' has a directed clique C' of size at least k. We create a directed graph $G' = (V', E')$ with an edge $u'v'$ and $v'u' \in E'$ between any two vertices $(u', v') \in V'$ if and only if there is an undirected edge uv or vu in E for those 2 vertices $(u, v) \in V$. This comes from the fact that an undirected edge uv or vu is equivalent to a directed edge uv and vu. So our directed graph G' has a Clique C' that has the same size as the undirected clique of size at least k. Hence, if an undirected graph G has Clique C of size at least k then the directed graph G' contains a directed clique of size at least k.

Q3. Consider the following *Clique-Independent Set* problem. Given graph $G = (V, E)$ with n vertices and a positive integer $k \leq n/2$, decide if G contains both a clique of size at least k and an independent set of size at least k.

Show that CliqueIndependentSet is NP-Complete.

Answer: In order to show that CliqueIndependentSet is NP-complete, we have to show that there is a polynomial time verifier for it and we have to reduce a NP-hard language to CliqueIndependentSet. The verifier for the CliqueIndependentSet takes an encoding of a graph $G = (V, E)$, a positive integer $k \leq n/2$ and y (certificate). The verifier then checks that y encodes 2 subsets of vertices S_y and S_z of G with $|S_y| \geq k$, and $|S_x| \geq k$. If not, reject. The verifier then checks that $(uv \in E)$ for each pair of vertices $u, v \in S_y$, and $(uv \notin E)$ for each pair of vertices $u, v \in S_x$ and accepts if all the pairs are connected in S_y (clique of size at least k) and all the pairs are not connected (independent set of size at least k) in S_x. We can see that the verifier runs in polynomial time in the size of G. It takes polynomial time to check that y encodes a subset of vertices, and checking each pair of vertices encoded by y (S_y) is connected requires $O(|V|^2)$ time in the worst-case, and checking each pair of vertices by encoded by y (S_x) is not connected also takes $O(|V|^2)$ time in the worst-case. Morever, if G contains a clique C of size at least k and an independent set of size at least k, then certainly there exists a y that makes the verifier accept. Conversely, if the verifier accepts (G, k) and y, then the set S_y is a clique of size at least k, and the set S_x is an independent set of size at least k. It follows that CliqueIndependentSet $\in NP$.

Now we have to show that CliqueIndependentSet $\in NP$-hard.

In order to show this, we give a reduction from Clique to CliqueIndependentSet.

A graph $G = (V, E)$ has a clique $C \subseteq V$ of size at least k if and only if a graph $G' = (V', E')$ has a clique $C' \subseteq V'$ of size at least k' and an independent set of size at least k'. From Professor Robert's notes, we know that Clique is NP-complete, so by reducing Clique to CliqueIndependentSet, we can show that CliqueIndependentSet is NP-hard. The reduction is similar to Professor Robert's Tutorial 6 notes where he reduces VertexCover to DominatingSet. Our reduction is as follows: Assume that we have have a graph $G = (V, E)$, and $C \subseteq V$ such that C is a clique. Using this, we can reduce G to $G' = (V', E')$ which is in CliqueIndependentSet which has a clique C' of size at least k' and an independent set of size at least k'. In order to create G', all we have to do is add k (where $k \leq n/2$) vertices in V (which is an instance of G) to each vertex in its own clique $C \subseteq V$ (we assume that G has a clique C). The created graph $G' = (V', E')$ will have a clique $C' \subseteq V'$ of size at least k' (obvious from reduction) and an independent set of size at least k'. By adding k vertices to each vertex in C, we get a graph that has k fully connected vertices (clique) and k not connected vertices (independent set). This suggests an obvious reduction, just set $C = C'$ and $k = k'$. This reduction occurs in polynomial time $O(nk)$ as we add k vertices to each vertex in C, and C is a clique in G, and G has n vertices. The converse is true as well. Suppose we are given a graph $G' = (V', E')$ which has a clique $C' \subseteq V'$ of size at least k' and an independent set of size at least k', then the reduction to Clique is trivial as we do not need to modify the graph in any way because G' already has a clique $C' \subseteq V'$ of size at least k'. Therefore, $G = (V, E)$ automatically has a clique $C \subseteq V$ of size at least k by setting $C = C'$ and $k = k'$.

Q4. Let m and n be positive integers and consider the following game – called Pebble Up – played on an $n \times n$ grid. We will denote the square in the ith row of the grid and the jth column of the grid by the pair (i, j). For each $1 \leq i, j \leq n$, the square (i, j) will be given a *colour* (either black or white) and a *number* (some positive integer $k \leq m$). The grid, along with a given colour and integer for each square, will be referred to as a *game board*.

Pebble Up is played on the game board as follows. You have a bag of black and white pebbles, and for each integer k between 1 and m, you must choose to place either a black or a white pebble on all of the squares labelled with the integer k (so, squares with different numbers can have different coloured pebbles, but if two squares have the same number then they must have the same coloured pebble). After choosing a coloured pebble for each integer k, the game moves on to the next phase.

For each square (i, j), with $1 \leq i, j \leq n-1$, examine the set of four squares

$$S_{ij} = \{(i, j), (i+1, j), (i, j+1), (i+1, j+1)\}.$$

This set of four squares is said to be *winning* if at least one of squares has the same colour as the pebble lying on it. You *win* the game if every set S_{ij} is winning.

Associated with Pebble Up is the following decision problem: As input, the algorithm receives two positive integers m, n, and for every (i, j) a colour b_{ij} (either black or white) and a positive integer $k_{ij} \leq m$. The goal is to decide if it is possible to win the Pebble Up game on the given game board (that is, to decide if there exists an assignment of pebbles to each of the numbers in a winning configuration).

Prove that (the decision problem associated with) Pebble Up is NP-Complete.

Answer: In order to show that Pebble-Up is NP-complete, we have to show that it is in NP and that it is NP-hard. To show that it is in NP, we need to give a polynomial time verifier for it. The certificate to the verifier will be an arrangement of pebbles to squares on the board. The verifier will then check all possible sets of 4 squares of the board such that each arrangement of pebbles on the 4 squares leads to a winning configuration. This takes $O(n^2)$ time as there as n rows and n columns on the board.

Now we need to prove that Pebble-Up is NP-hard. In order to do that we need to reduce from $4-SAT$. Let $C_1, C_2, \ldots, C_m$ be a collection of clauses, where each clause contains at most 4 literals. Let the literals be defined as $x_1, x_2, \ldots, x_n$. The first step is to find the clauses which have less than 4 literals and add dummy variables to them. In order to do so, we have to add 6 new boolean variables. $a_1, b_1, b_2, c_1, c_2, c_3$. Since we want each clause to have 4 literals exact, we need to add 1 variable to clauses with 3 literals, 2 variables to clauses with 2 literals, and 3 variables to clauses with 1 variable. Hence, we add a_1 to pad the clauses with 3 literals, b_1 and b_2 to pad clauses with 2 literals, and c_1, c_2 and c_3 to pad clauses with 3 literals. This is similar to what Professor Robert did in his Tutorial 7 notes. This sort of construction creates a $4-SAT$ problem which will have 6 new variables and split the clauses up so that each clause adds up to 4 literals. Therefore, this is polynomial in the length of its input.

We now reduce $4-SAT$ to Pebble-Up. We know that each clause in our $4-SAT$ instance has 4 literals. So we use that fact and reduce each clause to a set of 4 squares on the board. Each set of 4 squares on the game board will correspond to a clause. Suppose a clause $C = y_1 \vee y_2 \vee y_3 \vee y_4$, this will directly match a set of 4 squares on the baord. The set of 4 squares is given as $S_{ij} = \{(i, j), (i+1, j), (i, j+1), (i+1, j+1)\}$

where each square is represented as (i, j) with $1 \leq i, j \leq n - 1$. The clause C will be satisfied if and only if every set of 4 squares is winning, and that occurs when at least one of the squares has the same colour as the pebble lying on it. Now we need a way to label all the squares with numbers (k_{ij}). This can be achieved by looking at the subscript of each literal in the clause. So if $C = y_1 \vee y_2 \vee y_3 \vee y_4$, then the number on the squares are 1, 2, 3 and 4. We also need a way to distinguish between white squares and black squares (b_{ij}). This can be achieved by checking whether the literal in the clause is negated or not. If the literal is negated, then the subscript associated with the literal will correspond to a black square with that subscript as number and white if the literal is not negated. We can now construct the game board. We have to embed m clauses onto an n x n square board. If we have $n = 3$ and $m = 5$ (as illustrated in the question), our game board will be 3 x 3. If we have $n = 3$ and $m = 6$, then our game board will be 5 x 5. In general, our game board will be n = 2 x ceil(sqrt(m)) - 1. This raises another concern when 2 sets of squares overlap each other. In order to solve this, we create partitions between the sets of 4 squares by adding a vertical and horizontal separation. In those partitions, we put the same number but alternate the colour of the squares. This ensures that these partitions will always be winning. Since our game board has n x n squares, the reduction runs in $O(n^2)$ polynomial time.
Now we need to show that this reduction will in fact decide if it is possible to win the Pebble Up game. We reduced $4 - SAT$ to Pebble-Up. Each variable in our $4 - SAT$ instance is mapped to a square on the game board, each clause is mapped to a set of 4 squares and each negated/not negated literal is mapped to the colour (black or white) pebble on the game board. Thus, if there exists a satifsying assignment to our $4 - SAT$ instance, then we can construct our game board as described above. This implies that there is at least one square in every set of 4 squares that has the same colour as the pebble lying on it. On the other hand, if there is a winning configuration of pebbles then we can accordingly create our $4 - SAT$ instance that will be satisfied. Hence, we proved that Pebble Up is in NP and is NP-hard. Therefore, Pebble-Up is NP-complete.