CS 5114
Solutions to Homework Assignment 6

Meghendra Singh

March 31, 2017

[15] **1. CLRS Exercise 34.3-6.** A language L is ***complete*** for a language class C with respect to polynomial-time reductions if $L \in C$ and $L' \leq_{\text{p}} L$ for all $L' \in C$. Show that $\emptyset$ and $\{0,1\}^*$ are the only languages in P that are not complete for P with respect to polynomial-time reductions.

In order to show that $\emptyset$ (the set containing no elements) and $\{0,1\}^*$ (the set of all finite strings over the binary alphabet {0,1}) are the only languages in class P that are not complete for P with respect to polynomial-time reductions, we need to prove three cases:

1. $\emptyset$ is not complete for P.
2. $\{0,1\}^*$ is not complete for P.
3. Any language other than $\emptyset$ and $\{0,1\}^*$ in P is complete for P.

Case 1. $\emptyset$ is not complete for P: Assume $L = \emptyset$ be complete for class P. Therefore, $\emptyset \in$ P and $L' \leq_{\text{p}} L$ for all $L' \in$ P. Let $L_1 \in$ P be an arbitrary language in P such that $L_1 \neq L$. Since $\emptyset$ is complete for P, there exists a polynomial time computable function f such that, for all $x \in L_1$, $f(x) \in L$. Because L is the empty-set ($\emptyset$) there cannot be any element present in L. Given $L_1 \neq L, f(x) \notin L$, and such an f doesn't exist. This is a contradiction to our initial assumption that $L = \emptyset$ is complete for P. Hence, $\emptyset$ is not complete for P.

Case 2. $\{0,1\}^*$ is not complete for P: Assume $L = \{0,1\}^*$ be complete for class P. Therefore, $\{0,1\}^* \in$ P and $L' \leq_{\text{p}} L$ for all $L' \in$ P. Let $L_1 \in$ P be an arbitrary language in P such that $L_1 \neq L$. Since $\{0,1\}^*$ is complete for P, there exists a polynomial time computable function g such that, for all $x \in L_1$, $g(x) \in L$. Because L is the set of all possible finite strings over the binary alphabet {0,1}, $\overline{L} = \emptyset$ (complement of L is the empty-set). Also since $L_1 \neq L$ and $L_1 \subset L$, there should be at-least one string in L which is not present in L_1. For any string $s \notin L_1$, $g(s) \in \overline{L}$, but this is a contradiction because $\overline{L} = \emptyset$ and therefore such a g doesn't exist. Hence, $\{0,1\}^*$ is not complete for P.

Case 3. Any language other than $\emptyset$ and $\{0,1\}^* \in$ P is complete for P: Assume $L \in$ P be an arbitrary language in P such that $L \neq \emptyset$ and $L \neq \{0,1\}^*$. Assume two arbitrary strings x and y, such that $x \in L$ and $y \notin L$. This is a valid assumption, because L cannot be the empty-set (i.e. $\emptyset$), which implies there will be at-least one string $x \in L$. Also since L is not the Kleene closure over the binary alphabet (i.e. $\{0,1\}^*$) there will always be some string $y \in \{0,1\}^*$ which is not present in L (i.e. $y \notin L$). Now, let $L' \in$ P be any other arbitrary language in P such that $L' \neq L$. Since, $L' \in P$ there exists a polynomial time algorithm D, which can **decide** if an arbitrary string $z \in L'$. Therefore, $D(z) = 1$ if $z \in L'$

and $D(z) = 0$ if $z \notin L'$. We can easily define another polynomial time algorithm M such that:

$$\begin{aligned} &\forall z \in L' \\ &\qquad \text{if } [D(z) = 1] \text{ then } M(z) = x \\ &\text{and, if } [D(z) = 0] \text{ then } M(z) = y \end{aligned}$$

Here, we are basically mapping all elements present in L' to one element (x) which we know is present in L, also, we are mapping all elements not present in L' to one element (y) which we know is not present in L. M is thus the reduction function which reduces the language L' to L in polynomial time. Since, $L' \leq_{\mathrm{p}} L$, and $L \in$ P, we have proved that L is ***complete*** for class P. Moreover, L was an arbitrarily chosen language in class P, such that $L \neq \emptyset$ and $L \neq \{0,1\}^*$ we can say that all languages in P except for $\emptyset$ and $\{0,1\}^*$ are ***complete*** for P.

---

[15] 2. CLRS Exercise 34.3-8. The reduction algorithm F in the proof of Lemma 34.6 constructs the circuit $C = f(x)$ based on knowledge of x, A, and k. Professor Sartre observes that the string x is input to F , but only the existence of A, k, and the constant factor implicit in the $O(n^k)$ running time is known to F (since the language L belongs to NP), not their actual values. Thus, the professor concludes that F cant possibly construct the circuit C and that the language CIRCUIT-SAT is not necessarily NP-hard. Explain the flaw in the professor's reasoning.

We know that for a language L' to be ***NP-hard***, the following property needs to be satisfied:

$$L \leq_{\mathrm{p}} L' \text{ for every } L \in \text{NP}$$

For proving that CIRCUIT-SAT is NP-hard, we need to show that any language in class NP can be reduced to CIRCUIT-SAT in polynomial time. For this we can select an arbitrary language L in NP and prove that it can be reduced to CIRCUIT-SAT in polynomial time. The proof of Lemma 34.6 does exactly this, by presenting a reduction algorithm F which constructs the single combinational circuit $C = f(x)$ based on knowledge of $x \in L$, A, and k, effectively reducing L to CIRCUIT-SAT in polynomial time. Here, A is the polynomial time verifier of an arbitrary language $L \in$ NP, x is an input to A and k is the constant which represents the exponent in the worst case running time $O(n^k)$ of A with respect to an input size n. The basic idea of the proof is to represent the computation of A as a sequence of configurations c_i and subsequently use a reduction algorithm F to construct a single combinational circuit to represent all of these configurations (in a way unfold the configurations). Therefore, F constructs a circuit C which is satisfiable and F runs in polynomial time, we can be sure that any $L \in$ NP is reducible to CIRCUIT-SAT. The knowledge that there exists a reduction algorithm F which runs in polynomial time with respect to the input size of the arbitrary language $L \in$ NP, is sufficient to prove that for every language $L \in$ NP, $L \leq_{\mathrm{p}}$ CIRCUIT-SAT. Hence, CIRCUIT-SAT is ***NP-hard***.

The flaw in the professor's reasoning is the conclusion that CIRCUIT-SAT is not necessarily NP-hard, because F doesn't know A and hence can't construct C. This is incorrect because in order to prove that CIRCUIT-SAT is NP-hard, we only need to show the existence of an algorithm F which correctly computes the reduction function and runs in polynomial time. We don't actually need to construct F, hence whether F knows A or not, is of no consequence. What we do know is that the algorithm A whose running time is dependent on some constant k (i.e. polynomial time with respect to input size n or $O(n^k)$) should definitely exist for L to be in NP. If A exists, we can be sure about the existence of F which also runs in polynomial time as shown in the proof of Lemma 34.6. This evidence (existence of A and F) is sufficient to prove that CIRCUIT-SAT is NP-hard.

---

[15] **3. CLRS Exercise 34.4-5.** Show that the problem of determining the satisfiability of boolean formulas in disjunctive normal form is polynomial-time solvable.

A boolean formula is in disjunctive normal form (DNF) if it is a disjunction of conjunctive clauses, can also be described as an OR of ANDs. Unlike conjunctive normal form (CNF), literals in a DNF clause are connected by the AND operators, whereas the clauses themselves are connected by OR operators. Similar to CNF, there is no restriction on the number of literals that can be present in a clause as long as each literal is a valid variable or its negation in DNF. Also, there is no restriction on the number of clauses that can be present in DNF. An example formula in DNF is as follows:

$$\phi(X_1, X_2, X_3, X_4, X_5, X_6) = (X_2 \wedge \neg X_2 \wedge X_6) \vee (\neg X_1 \wedge \neg X_3 \wedge X_3) \vee (X_1 \wedge X_4 \wedge X_5)$$

Here, there are a total of 3 clauses each with 3 literals. In order to determine if a boolean formula in DNF is satisfiable, we need to check if there is at-least one clause in the entire formula, which can be satisfied. This is because all the clauses are connected by OR operators, and we know that even if one of the boolean operands connected by ORs takes a TRUE value, the evaluation of the OR operations would always result in a TRUE value. Also, the boolean value of a clause in DNF would always be FALSE if it contains two literals l_1 and l_2, such that, $l_1 = \neg l_2$ or $\neg l_1 = l_2$. This is because such a pair of literals would produce a FALSE value and by the definition of boolean AND operation, if there is at-least one FALSE value in a series of boolean values connected by AND operators, the evaluation of AND operations would result in a FALSE value. Hence, if there is at-least one clause in the DNF formula which does not have a variable and its negation, we can be sure that there is at-least one truth assignment for which the formula would evaluate to TRUE and hence is satisfiable. In the DNF formula $\phi(X_1, X_2, X_3, X_4, X_5, X_6)$ above, we can see that the first two clauses have pairs of literals which are negations of each other (X_2 and $\neg X_2$ in the first clause and $\neg X_3$ and X_3 in the second clause). However, the last clause doesn't have any such pair, making the formula $\phi(X_1, X_2, X_3, X_4, X_5, X_6)$ satisfiable. The algorithm DNF-SAT presented below determines the satisfiability of a DNF boolean formula in polynomial time:

```
DNF-SAT($\phi$)
 1  for each clause $C_i \in \phi$
 2      $FLAG = TRUE$
 3      for each literal $l_j \in C_i$
 4          // Check if negation of $l_j$ is present in $C_i$
 5          if $\neg l_j \in C_i$
 6              $FLAG = FALSE$
 7      // if no literal and its negation pair is found, $\phi$ is satisfiable
 8      if $FLAG == TRUE$
 9          return 1
10  // if no satisfiable clause found, $\phi$ is not satisfiable
11  return 0
```

The above algorithm checks each clause of the input formula ϕ and returns 1 as soon as it finds the first clause which does not contain a literal and its negation. If such a clause is not found, DNF-SAT returns 0. Therefore, if DNF-SAT$(\phi) = 1 \implies \phi$ is satisfiable i.e. there is at-least one satisfying assignment for ϕ, and if DNF-SAT$(\phi) = 0 \implies \phi$ is not satisfiable. Assuming that the boolean formula in DNF (i.e. ϕ) contains a total of n literals in m clauses, we can see that lines 1 through 9 of DNF-SAT would execute m times (once for each clause). We can also see that lines 3 through 6 can execute $m * n$ times (once for each literal) in the worst case, and lines 5 and 6 would execute $m * n * n = m * n^2$ times in the worst case. Therefore we can write the time complexity of DNF-SAT as $O(mn^2)$, where m is the total number of clauses and n is the total number of literals in the DNF formula ϕ. Clearly, DNF-SAT executes in polynomial time in terms of the input (number of literals and clauses).

---

[15] 4. CLRS Exercise 34.4-6. Suppose that someone gives you a polynomial-time algorithm to decide formula satisfiability. Describe how to use this algorithm to find satisfying assignments in polynomial time.

Here, we need to find a satisfying assignment for a formula ϕ, given that we have access to a polynomial-time algorithm F-SAT which decides boolean formula satisfiability. F-SAT can be defined as follows:

$$\text{F-SAT}(\phi) = 1 \text{ if } \phi \text{ is satisfiable}$$
$$\text{F-SAT}(\phi) = 0 \text{ if } \phi \text{ is not satisfiable}$$

We can write an algorithm GET-SAT-ASSIGNMENT that uses F-SAT to compute one satisfying assignment for an input boolean formula ϕ. GET-SAT-ASSIGNMENT would return a boolean array X of size n = number of variables in ϕ, that would hold the values of the satisfying boolean assignments for each of the n variables. However if the formula ϕ does not have a satisfying assignment, GET-SAT-ASSIGNMENT would return 0. We can start by calling F-SAT(ϕ) to check if ϕ is satisfiable. If F-SAT(ϕ) returns 0, it means that no satisfying assignment exists for ϕ, hence GET-SAT-ASSIGNMENT should exit and return 0.

Otherwise, we know that there is at-least one combination of TRUE/FALSE values of the n variables in ϕ for which ϕ is evaluated to TRUE. Since there are only two possible values (TRUE/FALSE) for any variable $x_i \in \phi$, we can call F-SAT on the formula ϕ' created by replacing x_i with either TRUE or FALSE. If F-SAT returns 1 on x_i = TRUE, then this should be the satisfying assignment for the variable x_i. On the other hand if F-SAT returns 0 on x_i = TRUE, then x_i = FALSE should be the the satisfying assignment for x_i, because we know that the original formula ϕ is satisfiable. We can iteratively repeat this process on all the variables to obtain one satisfying assignment for the original formula ϕ in polynomial time. The algorithm GET-SAT-ASSIGNMENT is as follows:

```
GET-SAT-ASSIGNMENT($\phi$)
 1  if F-SAT($\phi$) == 0
 2  // $\phi$ is not satisfiable, therefore no satisfying assignment exists
 3      return 0
 4  $n$ = number of variables in $\phi$
 5  Let $X$ be a boolean array of size $n$
 6  $i = 1$
 7  while $i \leq n$
 8      // Make new formula $\phi'$ by setting $x_i$ =TRUE in $\phi$
 9      $\phi' = \phi \leftarrow x_i$ = TRUE
10      if F-SAT == 1
11          $X[i]$ = TRUE
12      else
13          $X[i]$ = FALSE
14          $\phi' = \phi \leftarrow x_i$ = FALSE
15      $\phi = \phi'$
16      $i++$
17  return $X$
```

The above algorithm would return a satisfying assignment for the input boolean formula ϕ. Since F-SAT executes in polynomial time, we can assume that its time-complexity can be given by $T(n) \in O(n^k)$ for some constant k. In GET-SAT-ASSIGNMENT We make two calls to F-SAT, once at line 1, and subsequently inside the while loop at line 10. Since the while loop executes once for every variable in ϕ, so in the worst case lines 7 through 16 would execute a total of n times. Therefore the time complexity of GET-SAT-ASSIGNMENT can be given by $T(n) \in O(n * n^k) \implies T(n) \in O(n^{k+1})$ for some constant k. Hence, the algorithm GET-SAT-ASSIGNMENT finds one satisfying assignment for a boolean formula in polynomial time.