CPSC 320 Notes: What's in a Reduction? Solution-ish stuff

March 22, 2018

1 Boolean Satisfiability

1. Yes, this is satisfiable. Solving it by hand: we need $x_5 = T$ and $x_1 = F$. Once x_5 is true, then the last two clauses indicate that x_2 and x_3 have to have the same truth values. Assigning both true or both false will also handle the first clause. So, for example: $x_1 = F, x_2 = T, x_3 = T, x_4 = T, x_5 = T$ is a truth assignment that satisfies the clause.

2. We'll leave this to you, but the footnote takes you a long way towards an answer. :)

3. Again, we leave this to you.

4. Given an assignment of truth values to the variables in an instance, we could use the algorithm below. The first step takes time proportional to the number of variables in the problem instance, and the number of variables is at most linear in the length of the instance (since each variable has to be mentioned in the input!). The following loop takes constant time per element of the input (at worst) and so linear time in the length of the input. Overall, then, the algorithm takes time polynomial (indeed, linear) in the length of the input, even in the worst case.

   (a) Build an array $T[1 \ldots n]$ such that $T[i]$ is true if and only if variable x_i is true.

   (b) For each clause:

      i. For each literal in the clause:

         A. If the literal is of the form $-i$ and $T[i]$ is false, skip to the next clause.

         B. If the literal is of the form i and $T[i]$ is true, skip to the next clause.

      ii. If we reach this point (i.e., none of the literals in this clause are true), terminate immediately indicating that the assignment does **not** satisfy the expression.

   (c) If we reach this point (i.e., each clause has at least one true literal), terminate indicating the assignment **does** satisfy the expression.

5. Brute force would need to try 2^n different truth assignments in the worst case. (Note that trying one truth assignment may take more than constant time; so, the brute force algorithm may actually take $\omega(2^n)$ time to run!)

2 3-SAT and SAT

1. You could transform (x_5) into $(x_5 \vee x_5 \vee x_5)$. If you couldn't repeat variables, you could introduce two new variables and make four clauses like $(x_5 \vee a \vee b)$, putting the two new variables in all four combinations of TT, TF, FT, FF and therefore forcing x_5 to be true. (Note that if you had many one-variable clauses to transform, you could reuse a and b to the same effect many times rather than needing new variables each time. Also note that we could use this to force a variable—say t—to necessarily be true and similarly force another variable—say f—to necessarily be false. That's handy

because once we've done that, we can "or" t onto any short clauses to pad them out, all for the cost of a constant number of extra variables and clauses to set t up.)

2. You can transform $(x_1 \vee \overline{x_2} \vee x_3 \vee x_4)$ into $(x_1 \vee \overline{x_2} \vee x_{n+1}) \wedge (\overline{x_{n+1}} \vee x_3 \vee x_4)$. Notice that because the new variable x_{n+1} **must** be either true or false, then either at least one of the first two terms in the original clause must be true or at least one of the last two literals in the original clause must be true. Overall then, at least one of the literals in the original clause must be true, as we wanted!

   One way to think of this is to transform the two clauses into conditionals headed by x_{n+1}. Then, one of the clauses is "if x_{n+1} is true, then..." and the other is "if x_{n+1} is false, then...". Between them, they insist that one or the other of the split pieces of the clause must still be true.

3. You can use a function like this to transform any clause (list of literals $l_1, l_2, \ldots, l_k$) of length $k \geq 3$ into a list of clauses of length 3:

```
// Precondition: k >= 3
FIX(l1, l2, ..., lk):
  If k = 3:
    Output the clause l1, l2, l3
  Else (k > 3):
    Let x be the index of a so far unused variable
    Output the clause l1, l2, x
    FIX(-x, l3, l4, ..., lk)
```

   First, note that we could preprocess the input in linear time to find the first unused index of a variable and then in constant time get a new unused index if we wanted. So, that operation takes polynomial time.

   Second, notice that the "else" part of this function ensures the same guarantee as above. That is, if `FIX(l1, l2, ..., lk)` for some `k` ensures that at least one of `l1, l2, ..., lk` must be true any satisfying assignment, then `FIX(l1, l2, ..., lk, lk+1)` ensures one of `l1, l2, ..., lk, lk+1` must be true in any satisfying assignment (by the same argument as above).

   Overall, this function takes linear time in `k`, since it reduces `k` by one on each recursive call and stops when `k = 3`.

4. Our whole reduction is the following. (Note that I'd write the easier algorithm 2 before trying algorithm 1. Note also that I ignore the slightly tricky case where a clause has no literals.)

```
Algorithm 1: Given an instance of SAT, generate an instance of 3-SAT
as follows:

For each clause in the SAT problem:
  If the clause has 1 literal l1: Output (l1, l1, l1)
  Else if the clause has 2 literals l1, l2: Output (l1, l2, l2)
  Else the clause has literals l1, l2, ..., lk:
    Call FIX(l1, l2, ..., lk)

Algorithm 2: Given a solution to the 3-SAT instance (YES or NO),
output the same answer as the solution to the SAT instance.
```

   Since there is a polynomial number of clauses (indeed, linear in the length of the input) and each clause takes polynomial time (as described above), the reduction takes polynomial time.

We can also prove the reduction correct. (Indeed, we've already made arguments about the pieces of the reduction above.) We do this in two steps.

First, assume the answer to the 3-SAT instance is `YES`. In that case, there is an assignment of truth values to the variables in the 3-SAT problem such that each clause has at least one true literal. We need to show that the answer to the SAT instance is `YES`.

Let each variable in the SAT instance take on its truth value in the solution to the 3-SAT instance. Any 3-SAT clause generated from a SAT clause with 3 or fewer variables must have at least one literal **that appeared in the original SAT clause** that is true. So, all the SAT clauses with 3 or fewer variables have a true literal under this assignment.

What about the SAT clauses with 4 or more variables? Let's prove by contradiction that every such clause has at least one true literal. Imagine that one such had no true literal. Then, the only literals that can be true in the corresponding 3-SAT clauses are the "extra" variables introduced by `FIX`. In that case, the first of those variables needs to be true, since it appears positively with two "original" literals in the first recursive call to `FIX`. That means it appears negated in the next recursive call, which forces the next such variable to be true. That argument continues inductively, forcing all introduced variables to be true, but in the base case, the only introduced variable appears negatively, which leaves one clause with no true literals. But, the 3-SAT solution made at least one literal true in every 3-SAT clause; so, this is a contradiction. Thus, every SAT clause with 4 or more literals has at least one true literal.

That concludes our argument: If 3-SAT's answer is `YES`, then SAT's answer is `YES`.

Going the other way: Assume that there is a solution to the SAT instance, then there's an assignment of truth values to the SAT variables such that every clause has at least one true literal. Let the variables in the 3-SAT instance that are also in the SAT instance take on their values from the SAT solution. That means every 3-SAT clause generated from a SAT clause of length at most 3 has at least one true literal. For 3-SAT clauses generated from longer SAT clauses (via `FIX`), at least one of the generated clauses has a true literal. If that's not the last clause generated in the recursion in `FIX`, then it includes a new, introduced variable. We don't need to set that to true; so, we can set it to false and make the **next** generated clause true, which allows us inductively to make all subsequent clauses generated in the recursion true. Conversely, if this is not the first clause generated in the recursion, it starts with the negation of an introduced variable. We can make that variable true, since we don't need that literal to be true, which makes the clause **before** this one true, and allows us to continue making clauses true inductively back to the start of the recursion.

Thus, if the SAT instance's answer is `YES`, so is the 3-SAT instance's answer.

That proves our reduction correct. Since `SAT` is NP-hard, then `3-SAT` is as well. Since `3-SAT` is also in NP, then `3-SAT` is NP-complete.

3 What does a reduction tell us?

1. **SCENARIO #1:** Say our reduction's two algorithms take $O(f(n))$ time and we have a solution to the underlying problem problem that also takes $O(f(n))$ time. What do we know about the original problem?

   In this scenario, the reduction and the solution to the underlying problem together yield an $O(f(n))$ solution to the original problem. (This is how we'd normally use reductions in practice!)

2. **SCENARIO #2:** Say our reduction's two algorithms take $O(g(n))$ time and we know that there is **no solution** to the original problem that runs in $O(g(n))$ time. What do we know about the underlying problem? Why?

In this situation, there canot be an $O(g(n))$ solution to the underlying problem. Why not? Well, say there were, then as in the previous part, we could use that to generate an $O(g(n))$ solution to the original problem, but we said no such solution exists!

If we assume P $\neq$ NP, this is how we use reductions in an NP-completeness proof. (Although our use is technically more similar to what we do in the next part!)

3. **SCENARIO #3:** Say that we know (which we do) that if SAT can be solved in polynomial time, then **any** problem in the large set called "NP" can also be solved in polynomial time. What does our redution from SAT to 3-SAT tell us? Why?

   It tells us that if 3-SAT can be solved in polynomial time, then so too can **any** problem in NP be solved in polynomial time.

4 What does NP-completeness tell us?

1. List as many ways as you can think of to "get around" an NP-complete problem.

   We'll likely discuss things like: solving only "small" cases being sometimes good enough, solving (scalable) special cases quickly, approximating the optimal solution (for non-decision problems), and altering the problem to something similar that isn't NP-complete.

2. Left as an exercise :)

3. A **huge** number of problems are solved using SAT solvers because "SAT is an easy target for reductions". That's because SAT is a very expressive "language". It turns out to be quite natural to encode other problems as logic problems, and we can solve logic problems using SAT!

5 Challenge

Left to you!