CPSC 320 Notes: What's in a Reduction?

March 22, 2018

To reduce a problem A to another problem B, we typically proceed as follows: give one algorithm that takes a (legal) instance a of A and converts it into a legal instance b of B and a second algorithm that takes the corresponding solution s_b to b and transforms it into a solution s_a to a. (The second algorithm can use whatever bookkeeping information it needs from the first.)

We've used reductions to solve new problems based on problems we could already solve. For example, reducing hospital/intern matching to stable marriage.

But... there's another way to use reductions. A more **sinister** way.[1]

1 Boolean Satisfiability

Boolean satisfiability (SAT) is—as far as Computer Scientists know—a hard problem. In the version of SAT we discuss here, you're given a propositional logic expression like: $(x_1 \vee \overline{x_2} \vee x_3 \vee x_4) \wedge (x_5) \wedge (\overline{x_1}) \wedge (x_2 \vee \overline{x_3} \vee \overline{x_5}) \wedge (\overline{x_2} \vee x_3)$ and must determine whether any assignment of truth values to variables (the x_i's) makes the expression true, which we call *satisfying* the expression.

Formally, we'll say that an instance of SAT has n variables $x_1, x_2, \ldots, x_n$ and a statement that is a conjunction (an "and" connected by $\wedge$) of c clauses. Each clause is a disjunction (an "or" connected by $\vee$) of literals, and each literal is either a variable x_i or its negation $\overline{x_i}$. (For convenience, we'll insist on using the variables $x_1, x_2, \ldots x_n$ for some n, without skipping any.) A solution to SAT is simply `YES` (there is an assignment that makes the expression true) or `NO` (there isn't).

1. Is the example SAT instance above (in the first paragraph of this section) satisfiable? If not, explain why not. If so, prove it by giving an assignment that makes the statement true.

[1] Well, OK. Just **another** way.

2. Give the trivial instance(s) of SAT.[2]

3. Build a small but non-trivial instance of SAT and check whether it's satisfiable.

4. If I gave an assignment of truth values to the variables and said it satisfies the expression, how long would it take (in terms of the length of the input, i.e., the total number of literals in all the clauses) to test whether my statement is true?

5. A brute force algorithm could make a list of the variables $x_1, \ldots, x_n$ in the problem, try every assignment of truth values to these variables, and return `YES` if any satisfies the expression or `NO` otherwise. Asymptotically, how many truth assignments might this algorithm try (in terms of n)?

[2]The conjunction ("and") of zero conjuncts is **true**. The disjunction ("or") of zero disjuncts is **false**. Why? These are the identity elements for "and" and "or".

2 3-SAT and SAT

The 3-SAT problem is just like SAT, except **every** clause must be **exactly** of length 3. Let's build a reduction from SAT to 3-SAT. (So, we're solving SAT in terms of 3-SAT.)

1. If you really wanted a clause like (x_5) in 3-SAT, how could you represent it? *Hint:* one variable can appear multiple times in a clause. *Challenge:* How can you do it if one variable is **not** allowed to appear multiple times?

2. If you really wanted a clause like $(x_1 \vee \overline{x_2} \vee x_3 \vee x_4)$ in 3-SAT, how could you represent it? Note that the 3-SAT instance you create must be satisfiable if and only if the original SAT instance was. *Hint:* Create a brand new variable, y_1.[3] y_1 must be true or false, and we don't care which, since it's not actually part of the original problem. Could you force some of the literals in the original clause to be true when y_1 is true and some to be true when y_1 is false?

3. Extend your 4-literal clause plan above to a 5-literal clause like $(x_1 \vee \overline{x_2} \vee x_3 \vee x_4 \vee \overline{x_5})$.

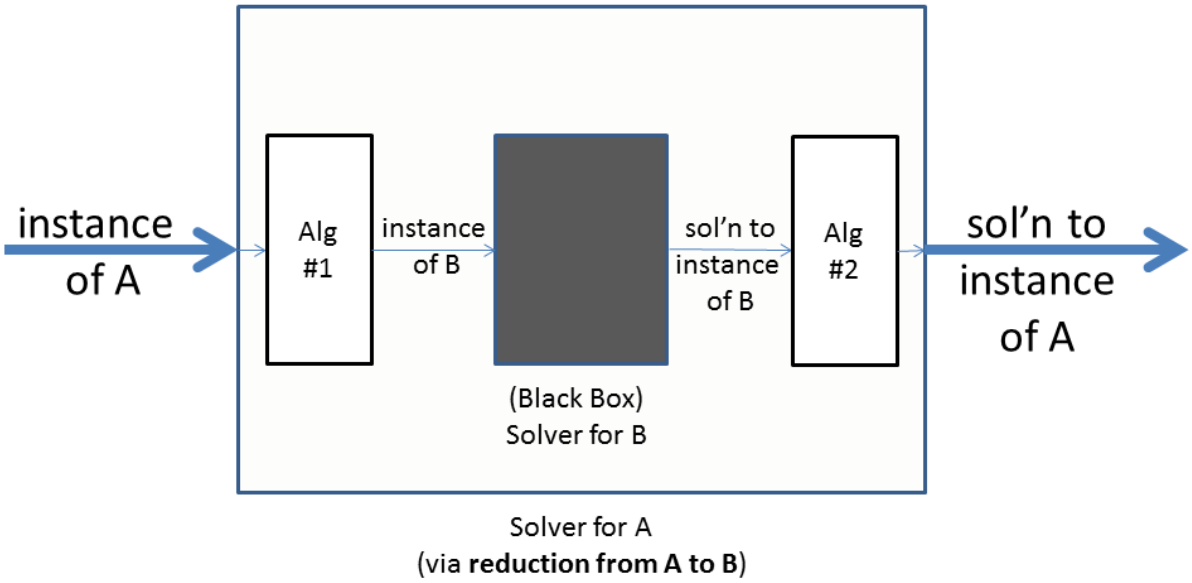
4. Extend your 5-literal clause plan to a 100-literal clause. How could you represent such a clause in 3-SAT?

5. Give a reduction from SAT to 3-SAT.

[3] y_1 must really be x_{n+1} since all the variables look like x_i, but it reads better with a distinctive name.

3 What does a reduction tell us?

Considering a reduction to be two algorithms that "connect" one problem to another, as in this diagram:



1. **SCENARIO #1 (what we've done up till now):** 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?

2. **SCENARIO #2 (what we usually think of NP-completeness as meaning):** 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?

3. **SCENARIO #3 (what NP-completeness technically means):** 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?

4 What does NP-completeness tell us?

Most Computer Scientists think "P $\neq$ NP". If that's true, then there is no correct, deterministic algorithm for any NP-complete problem that runs in polynomial time.[4] Specifically: if a problem is NP-complete, it's hopeless to write an algorithm that scales to arbitrarily large problem sizes and definitely, precisely solves every possible instance of those sizes correctly for exactly that problem.

Figure 1: By Béria L. Rodríguez, CC BY-SA 3.0

1. Imagine you visit the largest, seated, outdoor, bronze Buddha in the world. It's pretty impressive... but presumably there's a larger **standing** (reclining?), outdoor, bronze Buddha; a larger seated, **indoor**, bronze Buddha; and a larger, seated, outdoor Buddha **in some other material**.

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

2. Go solve a big NP-complete problem in your browser, on your phone, and laugh in the face of NP-completeness: `http://www.msoos.org/2013/09/minisat-in-your-browser/`.

   Note, however: There really are an enormous number of NP-complete problems that are **HARD** and **important** to solve, including many interesting instances of SAT. There are also many interesting problems that are either definitely or probably (if P $\neq$ NP or similar conditions) **harder** than NP, for example the problem of "AI Planning".

3. A **huge** number of problems are solved using SAT solvers because "SAT is an easy target for reductions".

   Explain that quote.

[4] Technical note: that doesn't mean the algorithm has to run in exponential time. There are options in between, like $2^{\sqrt{n}}$.

5 Challenge

1. Why wouldn't our "trick" for reducing SAT to 3-SAT work in "2-SAT"?
2. Give a polynomial-time algorithm to solve 2-SAT.
3. Find a good bound on the length of the 3-SAT instance created by our SAT to 3-SAT reduction in terms of the length of the initial SAT instance. (We take "length" to be $1 + \sum_{i=1}^{c}(1 + k_i)$.)

Fun Communications of the ACM reference with discussion of industrial and research applications of SAT: `http://goo.gl/KQoKFd`.

Solving NP-complete problems is not just for industry and research, it's for art as well (Travelling Salesperson Problem): `http://www.cgl.uwaterloo.ca/csk/projects/tsp/`.