Decision, search, and optimization problems; self reducibility

Vassos Hadzilacos

Recall that, according to the certificate-verifier characterization, **NP** consists of ***decision*** problems of the form "the xs such that $\exists y\, P(x, y)$" where $|y|$ is polynomial in $|x|$ and $P(x, y)$ is a polynomial-time decidable predicate. A y that confirms that x is a yes-instance of the problem is what we called a certificate; and the predicate $P(x, y)$ verifies this fact. Restricting our attention to decision problems has the advantage of focusing on yes-or-no problems, and makes the theory simpler. In practice, however, we are not just interested in that one-bit of information; rather, in the case of yes-instances, we are interested in the certificate itself. We call the problem of ***finding*** a certificate (when it exists) a ***search*** problem.

Consider, for example, some of the **NP**-complete problems we have encountered:

- **Satisfiability**
  Instance: $\langle F \rangle$, where F is a propositional formula
  Decision problem SAT: Does F have a satisfying truth assignment?
  Search problem SEARCH-SAT: Output a satisfying truth assignment of F, if one exists; otherwise, output "none".

- **Exact Cover**
  Instance: $\langle U, \mathcal{C} \rangle$, where U is a finite set and $\mathcal{C}$ is a collection of subsets of U
  Decision problem XCOV: Is there a subset of $\mathcal{C}$ consisting of disjoint sets whose union is U?
  Search problem SEARCH-XCOV: Output a subset of $\mathcal{C}$ consisting of disjoint sets whose union is U, if one exists; otherwise, output "none".

- **Independent Set**
  Instance: $\langle G, k \rangle$, where G is an undirected graph and $k \in \mathbb{Z}^+$.
  Decision problem INDSET: Does G have an independent set of at least k nodes?
  Search problem SEARCH-INDSET: Output an independent set of at least k nodes, if one exists; otherwise, output "none".

- **Traveling Salesman**
  Instance: $\langle n, C, b \rangle$, where $n \in \mathbb{Z}^+$, C is an $n \times n$ "cost" matrix of non-negative integers, and $b \in \mathbb{Z}^+$.
  Decision problem TSP: Is there a tour of cities $1, \ldots, n$ whose total cost is at most b, where $C[i, j]$ is the cost of traveling from city i to city j?
  Search problem SEARCH-TSP: Output a tour whose total cost is at most b, if one exists; otherwise, output "none".

Some **NP** problems are optimization problems disguised as decision problems. For example, the Independent Set problem is really a maximization problem: We are not interested in an independent set of some particular size k, but in the largest possible independent set. Similarly, the Traveling Salesman problem is really a minimization problem: We want to find the cheapest possible tour. For problems like these, in addition to the decision and search versions of the problem, there is also an optimization version.

- **Independent Set**
  Instance: $\langle G \rangle$, where G is an undirected graph. (Comparing to the instance of the decision and search versions, note that there is no k parameter.)
  Optimization problem Opt-IndSet: Output an independent set of maximum size.

- **Traveling Salesman**
  Instance: $\langle n, C \rangle$, where $n \in \mathbb{Z}^+$ and C is an $n \times n$ "cost" matrix of non-negative integers. (Note that there is no b parameter.)
  Optimization problem Opt-TSP: Output a tour of minimum cost.

Other **NP** problems, such as Satisfiability and Exact Set Cover, are decision versions of pure ***feasibility*** problems. In these cases the problem asks to find something (a satisfying truth assignment or an exact cover), not a "best" something: there is no obvious way in which one satisfying truth assignment or exact cover is "better" than another.

It is clear that the search and, when it exists, the optimization version of a problem is at least as hard as the decision version, in the sense that if we can solve efficiently the search (or optimization) version, we can also solve efficiently the decision problem. For example, if we have an algorithm Search-XCov-Solver that solves the search version of Exact Cover, we can use it to solve the decision version as follows: Given an instance $\langle U, \mathcal{C} \rangle$, we run Search-XCov-Solver on $\langle U, \mathcal{C} \rangle$, and we output 1 (yes-instance) if the algorithm returns an exact cover and we output 0 (no-instance) if the algorithm returns "none". Similarly, if we are given an algorithm Opt-TSP that solves the optimization version of TSP, we can use it to solve the decision version of TSP: Given an instance $\langle n, C, b \rangle$, we use Opt-TSP on input $\langle n, C \rangle$; we then compute the cost of the tour output by this algorithm; if that cost is at most b we output 1; otherwise we output 0.

The above arguments show that

$$\text{XCov} \leq_T^p \text{Search-XCov} \quad \text{and} \quad \text{TSP} \leq_T^p \text{Opt-TSP}.$$

Note that these are ***Cook*** reductions (i.e., polytime Turing reductions), not ***Karp*** reductions (i.e., polytime mapping reductions). This is because Karp reductions can relate only decision problems (with output only 1 or 0), while Cook reductions can relate problems with arbitrary output, which is what we need here.

Reasoning in the same way one can argue that for any **NP** problem A, $A \leq_T^p$ Search-A and $A \leq_T^p$ Opt-A (if A has an optimization counterpart). This simple fact is useful; it justifies our focus on the simpler, decision versions, of these problems: If, as we believe, **NP**-complete problems are not solvable in polynomial time then their search and optimization counterparts are also not solvable in polynomial time.

But what about the converse: Are the search and optimization counterparts of **NP**-complete problems polytime-reducible to the (seemingly simpler) decision versions? For example, is it the case that Search-Sat $\leq_T^p$ Sat and that Opt-TSP $\leq_T^p$ TSP? This would also be interesting to know: If, contrary to our expectation, it turns out that $\mathbf{P} = \mathbf{NP}$, and so all **NP**-complete problems are solvable in polytime, such reductions would show that their search and optimization counterparts, which are more relevant in practice, are also solvable in polytime!

When a search or optimization version of a problem A Cook-reduces to the decision version (i.e., Search-$A \leq_T^p A$ or Opt-$A \leq_T^p A$) we say that A is ***self-reducible***. It turns out that

Fact *Every **NP**-complete problem is self-reducible.*

The proof of this fact is beyond the scope of this course, but we will demonstrate some specific examples of self-reducibility. The ideas underlying these examples can be applied to many other cases.

Theorem 10.5 Sat *is self-reducible; i.e.,* Search-Sat $\leq_T^p$ Sat.

PROOF. Let SAT-ORACLE be an oracle for the SAT decision problem. We will describe a reduction algorithm SEARCH-SAT-SOLVER that uses this oracle to solve SEARCH-SAT in polynomial time. As is the convention in Cook reductions, we charge one time unit for each use of the oracle. (Thus, if there is a polytime algorithm for the decision problem SAT, we obtain a polytime algorithm for SEARCH-SAT via this reduction.)

The idea behind the reduction is as follows; to find a satisfying assignment for a given propositional formula F with variables $x_1, x_2, \ldots, x_n$.

- We use SAT-ORACLE to find out if F is satisfiable; if not, we return "none".
- If F is satisfiable, we substitute x_1 by the Boolean constant 0 in F, obtaining a new formula, denoted $F[x_1{=}0]$ with $n-1$ variables $x_2, \ldots, x_n$, and use SAT-ORACLE to find out if $F[x_1{=}0]$ is satisfiable; if so, we know we can satisfy F by setting $x_1 = 0$ and it remains to find a satisfying assignment for $F[x_1{=}0]$; if not, we know we can satisfy F by setting $x_1 = 1$ and it remains to find a satisfying assignment for $F[x_1{=}1]$. Let F_1 be the formula $F[x_1{=}0]$ or $F[x_1{=},1]$, whichever of the two was found to be satisfiable.
- We now proceed recursively with F_1: We substitute x_2 by the Boolean constant 0 in F_1 and obtain a new formula with $n-2$ variables $x_3, \ldots, x_n$; by using SAT-ORACLE again we find a truth value for x_2 that satisfies F_1 and therefore F.
- We continue in the same way until we have found truth values for all n variables that satisfy the original formula F.

This reduction algorithm is shown in pseudocode below. If F is a propositional formula whose variables include $x_1, \ldots, x_k$ then $\phi[x_1{=}b_1, x_2{=}b_2, \ldots, x_k{=}b_k]$, where $b_i \in \{0,1\}$ for each $i \in [1..k]$, denotes the propositional formula obtained from ϕ by replacing every instance of variable x_i by the Boolean constant b_i.

```
    SEARCH-SAT-SOLVER(F)
1   let x_1, ..., x_n be the variables of F
2   if SAT-ORACLE(F) = 1 then return "none"
3   else
4       F_0 := F
5       for i := 1 to n do
6           if SAT-ORACLE(F_{i-1}[x_i=0]) = 1 then v_i := 0; F_i := F_{i-1}[x_i=0]
7           else v_i := 1; F_i := F_{i-1}[x_i=1]
8       return (v_1, ..., v_n)
```

Each iteration of the for loop takes time $O(|F|)$, the time needed to substitute v_i for x_i in F_{i-1}, so the running time of the reduction algorithm is $O(n|F|) = O(|F|^2)$, which is polynomial in the size of the input F. (Recall that the cost of call to the oracle is one unit of time per call.)

The correctness of the reduction algorithm follows from the fact that the for-loop satisfies the following invariant: "F_i is a satisfiable formula and $F_i = F[x_1{=}v_1, x_2{=}v_2, \ldots, x_i{=}v_i]$". (The fact that this is an invariant can be proved by a straightforward induction.) Thus, when the loop terminates, $F[x_1{=}v_1, x_2{=}v_2, \ldots, x_i{=}v_i]$ is a satisfiable formula, which means that $\tau(x_i) = v_i$, for each $i \in [1..n]$, is a satisfying truth assignment for F. □

In the self-reduction of SAT shown in the proof of Theorem 10.4, the satisfying truth assignment, i.e., the certificate that the SEARCH-SAT-SOLVER is seeking, is constructed piece-by-piece — in this case, literally bit-by-bit! The same general idea applies to many other cases, including in self-reductions of optimization problems. Consider, for example, the reduction OPT-INDSET $\leq_T^p$ INDSET. Given an oracle for the Independent Set decision problem, we want to find a maximum independent set of a graph. In outline, the overall strategy is as follows:

(1) Using the oracle multiple times, search for the size of the maximum independent set: Is there an independent set of size at least 1? If so, of size at least 2? If so, of size at least 3? And so on, up to at most the number of nodes in the graph. Let k be the size of the maximum independent set.

(2) Use the oracle multiple times to determine if each node u is needed to obtain a maximum independent set of size k. That is, construct the maximum independent set one node at a time, using the oracle to guide the decision whether to keep a node.

We leave the details of this reduction algorithm as an exercise.

An important point related to Step (1) of the previous reduction is now in order. To find the size of the optimal independent set, we performed a ***sequential*** search, from 1 up to (at most) the number of nodes in the graph. This was fine in the case of the Independent Set problem because the number of nodes in the graph is polynomial in the size of the graph. In some cases, however, a sequential search does not lead to a polytime reduction. Consider, for example, the optimization version of TSP. Here we want to find the minimum cost of a tour. The above strategy would be to start from cost 0 and search sequentially until we find the minimum b such that there is a tour of cost at most b. Note, however, that b is exponentially larger than the number of bits required to store the cost matrix, so this is not a polytime reduction in the size of the input. The solution is to use binary search, instead of sequential search, to find the minimum cost of a tour. As an exercise you should try to fill out the details of the self reduction OPT-TSP $\leq_T^p$ TSP, including an explanation of why your reduction is, in fact, a polytime reduction.

Earlier we mentioned that it can be shown that all **NP**-complete problems are self-reducible. Is this true about ***all*** **NP** problems? It is not known whether this is the case. An interesting example is the problem of factoring integers:

- **Factoring**
  Instance: $\langle n \rangle$, where $n \in \mathbb{Z}+$
  Decision problem FACTOR: Are there integers $p, q \neq 1$ such that $n = p \cdot q$?
  Search problem SEARCH-FACTOR: Output integers $p, q \neq 1$ such that $n = p \cdot q$, if such integers exist; otherwise, output "none".

The decision version FACTOR is known to be in **P**, via the Agrawal-Kayal-Saxena primality testing algorithm. On the other hand, we do not know whether SEARCH-FACTOR is in **P**. In fact, we hope that it is not, since so much of cryptography and secure communication is based on the ***assumption*** that finding the factors of very large numbers is computationally intractable.