3

Polynomial-time Reductions

Overview: Reductions are procedures that use "functionally specified" subroutines. That is, the functionality of the subroutine is specified, but its operation remains unspecified and its running time is counted at unit cost. Thus, a reduction solves one computational problem by using oracle (or subroutine) calls to another computational problem. Analogously to our focus on efficient (i.e., polynomial-time) algorithms, here we focus on efficient (i.e., polynomial-time) reductions.

We present a general notion of (polynomial-time) reductions among computational problems, and view the notion of a "Karp-reduction" (also known as "many-to-one reduction") as an important special case that suffices (and is more convenient) in many cases. Reductions play a key role in the theory of NP-completeness, which is the topic of Chapter 4.

In the current chapter, we stress the fundamental nature of the notion of a reduction per se and highlight two specific applications: reducing search problems and optimization problems to decision problems. Furthermore, in these applications, it will be important to use the general notion of a reduction (i.e., "Cook-reduction" rather than "Karp-reduction"). We comment that the aforementioned reductions of search and optimization problems to decision problems further justify the common focus on the study of the decision problems.

Organization. We start by presenting the general notion of a polynomial-time reduction and important special cases of it (see Section 3.1). In Section 3.2, we present the notion of optimization problems and reduce such problems to corresponding search problems. In Section 3.3, we discuss the reduction of search problems to corresponding decision problems, while emphasizing the special case in which the search problem is

reduced to the decision problem that is implicit in it. (In such a case, we say that the search problem is self-reducible.)

Teaching Notes

We assume that many students have heard of reductions, but we fear that most have obtained a conceptually distorted view of their fundamental nature. In particular, we fear that reductions are identified with the theory of NP-completeness, whereas reductions have numerous other important applications that have little to do with NP-completeness (or completeness with respect to any other class). In particular, we believe that it is important to show that (natural) search and optimization problems can be reduced to (natural) decision problems.

On Our Terminology. We prefer the terms *Cook-reductions* and *Karp-reductions* over the terms "general (polynomial-time) reductions" and "many-to-one (polynomial-time) reductions." Also, we use the term *self-reducibility* in a non-traditional way; that is, we say that the search problem of R is self-reducible if it can be reduced to the decision problem of $S_R = \{x : \exists y \text{ s.t. } (x, y) \in R\}$, whereas traditionally, *self-reducibility* refers to decision problems and is closely related to our notion of *downward self-reducible* (presented in Exercise 3.16).

A Minor Warning. In Section 3.3.2, which is an advanced section, we assume that the students have heard of NP-completeness. Actually, we only need the students to know the definition of NP-completeness. Yet the teacher may prefer postponing the presentation of this material to Section 4.1 (or even to a later stage).

3.1 The General Notion of a Reduction

Reductions are procedures that use "functionally specified" subroutines. That is, the functionality of the subroutine is specified, but its operation remains unspecified and its running time is counted at unit cost. Analogously to algorithms, which are modeled by Turing machines, reductions can be modeled as *oracle* (Turing) machines. A reduction solves one computational problem

(which may be either a search problem or a decision problem) by using oracle (or subroutine) calls to another computational problem (which again may be either a search or a decision problem). Thus, such a reduction yields a (simple) transformation of algorithms that solve the latter problem into algorithms that solve the former problem.

3.1.1 The Actual Formulation

The notion of a general algorithmic reduction was discussed in Section 1.3.3 and formally defined in Section 1.3.6. These reductions, called Turing-reductions and modeled by oracle machines (cf. Section 1.3.6), made no reference to the time complexity of the main algorithm (i.e., the oracle machine). Here, we focus on efficient (i.e., polynomial-time) reductions, which are often called *Cook-reductions*. That is, we consider oracle machines (as in Definition 1.11) that run in time that is polynomial in the length of their input. We stress that the running time of an oracle machine is the number of steps made during its (own) computation, and that the oracle's reply on each query is obtained in a single step.

The key property of efficient reductions is that they allow for the transformation of efficient implementations of the subroutine (or the oracle) into efficient implementations of the task reduced to it. That is, as we shall see, if one problem is Cook-reducible to another problem and the latter is polynomial-time solvable, then so is the former.

The most popular case is that of reducing decision problems to decision problems, but we will also explicitly consider reducing search problems to search problems and reducing search problems to decision problems. Note that when reducing to a decision problem, the oracle is determined as the unique valid solver of the decision problem (since the function $f : \{0,1\}^* \to \{0,1\}$ solves the decision problem of membership in S if, for every x, it holds that $f(x) = 1$ if $x \in S$ and $f(x) = 0$ otherwise). In contrast, when reducing to a search problem, the oracle is not uniquely determined because there may be many different valid solvers (since the function $f : \{0,1\}^* \to \{0,1\}^* \cup \{\perp\}$ solves the search problem of R if, for every x, it holds that $f(x) \in R(x) \stackrel{\text{def}}{=} \{y : (x,y) \in R\}$ if $R(x) \neq \emptyset$ and $f(x) = \perp$ otherwise).[1] We capture both cases in the following definition.

Definition 3.1 (Cook-reduction): *A problem Π is* Cook-reducible *to a problem Π' if there exists a polynomial-time oracle machine M such that for every function f that solves Π' it holds that M^f solves Π, where $M^f(x)$ denotes the output of M on input x when given oracle access to f.*

[1] Indeed, the solver is unique only if for every x it holds that $|R(x)| \leq 1$.

Note that Π (resp., Π') may be either a search problem or a decision problem (or even a yet-undefined type of a problem). At this point, the reader should verify that *if Π is Cook-reducible to Π' and Π' is solvable in polynomial time, then so is Π*; see Exercise 3.2 (which also asserts other properties of Cook-reductions).

We highlight the fact that a Cook-reduction of Π to Π' yields a simple transformation of efficient algorithms that solve the problem Π' into efficient algorithms that solve the problem Π. The transformation consists of combining the code (or description) of any algorithm that solves Π' with the code of reduction, yielding a code of an algorithm that solves Π.

An Important Example. Observe that the second part of the proof of Theorem 2.6 is actually a Cook-reduction of the search problem of any R in $\mathcal{PC}$ to a decision problem regarding a related set $S'_R = \{\langle x, y' \rangle : \exists y'' \text{ s.t. } (x, y'y'') \in R\}$, which is in $\mathcal{NP}$. Thus, that proof establishes the following result.

Theorem 3.2: *Every search problem in $\mathcal{PC}$ is Cook-reducible to some decision problem in $\mathcal{NP}$.*

We shall see a tighter relation between search and decision problems in Section 3.3; that is, in some cases, R will be reduced to $S_R = \{x : \exists y \text{ s.t. } (x,y) \in R\}$ rather than to S'_R.

3.1.2 Special Cases

We shall consider two restricted types of Cook-reductions, where the first type applies only to decision problems and the second type applies only to search problems. In both cases, the reductions are restricted to making a single query.

Restricted Reductions Among Decision Problems. A Karp-reduction is a restricted type of a reduction (from one decision problem to another decision problem) that makes a single query, and furthermore replies with the very answer that it has received. Specifically, for decision problems S and S', we say that S is Karp-reducible to S' if there is a Cook-reduction of S to S' *that operates as follows*: On input x (an instance for S), the reduction computes x', makes query x' to the oracle S' (i.e., invokes the subroutine for S' on input x'), and answers whatever the latter returns. This reduction is often represented by the polynomial-time computable mapping of x to x'; that is, the standard definition of a Karp-reduction is actually as follows.

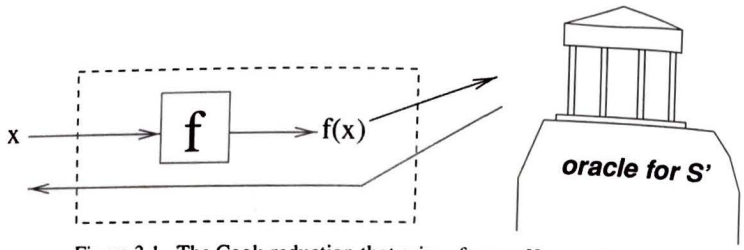

Figure 3.1. The Cook-reduction that arises from a Karp-reduction.

Definition 3.3 (Karp-reduction): *A polynomial-time computable function f is called a* Karp-reduction of S to S' *if, for every x, it holds that $x \in S$ if and only if $f(x) \in S'$.*

Thus, syntactically speaking, a Karp-reduction is not a Cook-reduction, but it trivially gives rise to one (i.e., on input x, the oracle machine makes query $f(x)$, and returns the oracle answer; see Figure 3.1). Being slightly inaccurate but essentially correct, we shall say that Karp-reductions are special cases of Cook-reductions.

Needless to say, Karp-reductions constitute a very restricted case of Cook-reductions. Specifically, Karp-reductions refer only to reductions among decision problems, and are restricted to a single query (and to the way in which the answer is used). Still, Karp-reductions suffice for many applications (most importantly, for the theory of NP-completeness (when developed for decision problems)). On the other hand, due to purely technical (or syntactic) reasons, Karp-reductions are not adequate for reducing search problems to decision problems. Furthermore, Cook-reductions that make a single query are inadequate for reducing (hard) search problems to any decision problem (see Exercise 3.12).[2] We note that *even within the domain of reductions among decision problems, Karp-reductions are less powerful than Cook-reductions.* Specifically, whereas each decision problem is Cook-reducible to its complement, some decision problems are *not* Karp-reducible to their complement (see Exercises 3.4 and 5.10).

Augmentation for Reductions Among Search Problems. Karp-reductions may (and should) be augmented in order to handle reductions among search problems. The augmentation should provide a way of obtaining a solution for

[2] Cook-reductions that make a single query overcome the technical reason that makes Karp-reductions inadequate for reducing search problems to decision problems. (Recall that Karp-reductions are a special case of Cook-reductions that make a single query; cf. Exercise 3.11.)

the original instance from any solution for the reduced instance. Indeed, such a reduction of the search problem of R to the search problem of R' operates as follows: On input x (an instance for R), the reduction computes x', makes query x' to the oracle R' (i.e., invokes the subroutine for searching R' on input x') obtaining y' such that $(x', y') \in R'$, and uses y' to compute a solution y to x (i.e., $y \in R(x)$). Thus, such a reduction can be represented by two polynomial-time computable mappings, f and g, such that $(x, g(x, y')) \in R$ for any y' that is a solution of $f(x)$ (i.e., for y' that satisfies $(f(x), y') \in R'$). Indeed, f is a Karp-reduction (of $S_R = \{x : R(x) \neq \emptyset\}$ to $S_{R'} = \{x' : R'(x') \neq \emptyset\}$), but (unlike in the case of decision problems) the function g may be non-trivial (i.e., we may not always have $g(x, y') = y'$). This type of reduction is called a Levin-reduction and, analogously to the case of a Karp-reduction, it is often identified with the two aforementioned mappings themselves (i.e., the (polynomial-time computable) mappings f of x to x', and the (polynomial-time computable) mappings g of (x, y') to y).

Definition 3.4 (Levin reduction): *A pair of polynomial-time computable functions, f and g, is called a* Levin-reduction of R to R' *if f is a Karp-reduction of $S_R = \{x : \exists y \text{ s.t. } (x, y) \in R\}$ to $S_{R'} = \{x' : \exists y' \text{ s.t. } (x', y') \in R'\}$ and for every $x \in S_R$ and $y' \in R'(f(x))$ it holds that $(x, g(x, y')) \in R$, where $R'(x') = \{y' : (x', y') \in R'\}$.*

Indeed, the (first) function f preserves the existence of solutions; that is, for any x, it holds that $R(x) \neq \emptyset$ if and only if $R'(f(x)) \neq \emptyset$, since f is a Karp-reduction of S_R to $S_{R'}$. As for the second function (i.e., g), it maps any solution y' for the reduced instance $f(x)$ to a solution for the original instance x (where this mapping may also depend on x). We mention that it is natural also to consider a third function that maps solutions for R to solutions for R' (see Exercise 4.20).

Again, syntactically speaking, a Levin-reduction is not a Cook-reduction, but it trivially gives rise to one (i.e., on input x, the oracle machine makes query $f(x)$, and returns $g(x, y')$ if the oracle answers with $y' \neq \perp$ (and returns $\perp$ otherwise); see Figure 3.2).

3.1.3 Terminology and a Brief Discussion

Cook-reductions are often called general (polynomial-time) reductions, whereas Karp-reductions are often called many-to-one (polynomial-time) reductions. Indeed, throughout the current chapter, whenever we neglect to mention the type of a reduction, we actually mean a Cook-reduction.

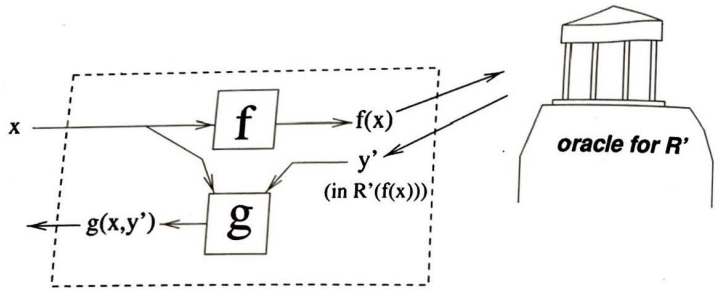

Figure 3.2. The Cook-reduction that arises from a Levin-reduction.

Two Compound Notions. The following terms, which refer to the existence of several reductions, are often used in advanced studies.

1. We say that two problems are computationally equivalent if they are reducible to each other. This means that the two problems are essentially as hard (or as easy). Note that computationally equivalent problems need not reside in the same complexity class.
   For example, as we shall see in Section 3.3, for many natural relations $R \in \mathcal{PC}$, the search problem of R and the decision problem of $S_R = \{x : \exists y \text{ s.t. } (x, y) \in R\}$ are computationally equivalent, although (even syntactically) the two problems do not belong to the same class (i.e., $R \in \mathcal{PC}$ whereas $S_R \in \mathcal{NP}$). Also, each decision problem is computationally equivalent to its complement, although the two problems may not belong to the same class (see, e.g., Section 5.3).
2. We say that a *class of problems, $\mathcal{C}$, is reducible to a problem Π' if every problem in $\mathcal{C}$ is reducible to* Π'. We say that the class $\mathcal{C}$ is reducible to the class $\mathcal{C}'$ if for every $\Pi \in \mathcal{C}$ there exists $\Pi' \in \mathcal{C}'$ such that Π is reducible to Π'.
   For example, Theorem 3.2 asserts that *$\mathcal{PC}$ is reducible to $\mathcal{NP}$*. Also note that $\mathcal{NP}$ is reducible to $\mathcal{PC}$ (see Exercise 3.9).

On the Greater Flexibility of Cook-reductions. The fact that we allow Cook-reductions (rather than confining ourselves to Karp-reductions) is essential to various important connections between decision problems and other computational problems. For example, as will be shown in Section 3.2, a natural class of optimization problems is reducible to $\mathcal{NP}$. Also recall that $\mathcal{PC}$ is reducible to $\mathcal{NP}$ (cf. Theorem 3.2). Furthermore, as will be shown in Section 3.3, many natural search problems in $\mathcal{PC}$ are reducible to a corresponding *natural* decision problem in $\mathcal{NP}$ (rather than merely to some problem in $\mathcal{NP}$). In all of these results, the reductions in use are (and must be) Cook-reductions.

Recall that we motivated the definition of Cook-reductions by referring to their natural ("positive") application, which offers a transformation of efficient implementations of the oracle into efficient algorithms for the reduced problem. Note, however, that once defined, reductions have a life of their own. In fact, the actual definition of a reduction does not refer to the aforementioned natural application, and reductions may be (and are) also used toward other applications. For further discussion, see Section 3.4.

3.2 Reducing Optimization Problems to Search Problems

Many search problems refer to a set of potential solutions, associated with each problem instance, such that different solutions are naturally assigned different "values" (resp., "costs"). For example, in the context of finding a clique in a given graph, the size of the clique may be considered the value of the solution. Likewise, in the context of finding a 2-partition of a given graph, the number of edges with both end points in the same side of the partition may be considered the cost of the solution. In such cases, one may be interested in finding a solution that has value exceeding some threshold (resp., cost below some threshold). Alternatively, one may seek a solution of maximum value (resp., minimum cost).

For simplicity, let us focus on the case of a value that we wish to maximize. Still, the two different aforementioned objectives (i.e., exceeding a threshold and optimization) give rise to two different (auxiliary) search problems related to the same relation R. Specifically, for a binary relation R and a *value function* $f : \{0,1\}^* \times \{0,1\}^* \to \mathbb{R}$, we consider two search problems.

1. *Exceeding a threshold*: Given a pair (x, v), the task is to find $y \in R(x)$ such that $f(x, y) \geq v$, where $R(x) = \{y : (x, y) \in R\}$. That is, we are actually referring to the search problem of the relation
$$R_f \stackrel{\text{def}}{=} \{(\langle x, v\rangle, y) : (x, y) \in R \wedge f(x, y) \geq v\}, \tag{3.1}$$
   where $\langle x, v\rangle$ denotes a string that encodes the pair (x, v).
2. *Maximization*: Given x, the task is to find $y \in R(x)$ such that $f(x, y) = v_x$, where v_x is the maximum value of $f(x, y')$ over all $y' \in R(x)$. That is, we are actually referring to the search problem of the relation
$$R'_f \stackrel{\text{def}}{=} \{(x, y) \in R : f(x, y) = \max_{y' \in R(x)} \{f(x, y')\}\}. \tag{3.2}$$
   (If $R(x) = \emptyset$, then we define $R'_f(x) = \emptyset$.)

Examples of value functions include the size of a clique in a graph, the amount of flow in a network (with link capacities), and so on. The task may be to find a clique of size exceeding a given threshold in a given graph or to find a maximum-size clique in a given graph. Note that in these examples, the "base" search problem (i.e., the relation R) is quite easy to solve, and the difficulty arises from the auxiliary condition on the value of a solution (presented in R_f and R'_f). Indeed, one may trivialize R (i.e., let $R(x) = \{0,1\}^{\mathrm{poly}(|x|)}$ for every x), and impose all necessary structure by the function f (see Exercise 3.6).

We confine ourselves to the case that f is (rational-valued and) polynomial-time computable, which in particular means that $f(x,y)$ can be represented by a rational number of length polynomial in $|x| + |y|$. We will show next that in this case, the two aforementioned search problems (i.e., of R_f and R'_f) are computationally equivalent.

Theorem 3.5: *For any polynomial-time computable $f : \{0,1\}^* \times \{0,1\}^* \to \mathbb{Q}$ and a polynomially bounded binary relation R, let R_f and R'_f be as in Eq. (3.1) and Eq. (3.2), respectively. Then, the search problems of R_f and R'_f are computationally equivalent.*

Note that *for $R \in \mathcal{PC}$ and polynomial-time computable f, it holds that $R_f \in \mathcal{PC}$.* Combining Theorems 3.2 and 3.5, it follows that *in this case both R_f and R'_f are reducible to $\mathcal{NP}$.* We note, however, that even in this case it does not necessarily hold that $R'_f \in \mathcal{PC}$ (unless, of course, $\mathcal{P} = \mathcal{NP}$). See further discussion following the proof.

Proof: The search problem of R_f is reduced to the search problem of R'_f by finding an optimal solution (for the given instance) and comparing its value to the given threshold value. That is, we construct an oracle machine that solves R_f by making a single query to R'_f. Specifically, on input (x, v), the machine issues the query x (to a solver for R'_f), obtaining the optimal solution y (or an indication $\perp$ that $R(x) = \emptyset$), computes $f(x,y)$, and returns y if $f(x,y) \geq v$. Otherwise (i.e., either $y = \perp$ or $f(x,y) < v$), the machine returns an indication that $R_f(\langle x, v\rangle) = \emptyset$.

Turning to the opposite direction, we reduce the search problem of R'_f to the search problem of R_f by first finding the optimal value $v_x = \max_{y \in R(x)}\{f(x,y)\}$ (by binary search on its possible values), and next finding a solution of value v_x. In both steps, we use oracle calls to R_f. For simplicity, we assume that f assigns *positive* integer values (see Exercise 3.7), and let $\ell = \mathrm{poly}(|x|)$ be such that $f(x,y) \leq 2^\ell - 1$ for every $y \in R(x)$. Then, on input x, we first find $v_x = \max\{f(x,y) : y \in R(x)\}$, by making oracle calls of the form $\langle x, v\rangle$. The point is that $v_x < v$ if and only if $R_f(\langle x, v\rangle) = \emptyset$, which in turn is indicated by the oracle answer $\perp$ (to the query $\langle x, v\rangle$). Making ℓ queries, we determine v_x (see Exercise 3.8). Note that in case $R(x) = \emptyset$, all the answers will indicate that $R_f(\langle x, v\rangle) = \emptyset$, and we halt indicating that $R'_f(x) = \emptyset$ (which is indeed due to $R(x) = \emptyset$). Thus, we continue only if $v_x > 0$, which indicates that $R'_f(x) \neq \emptyset$. At this point, we make the query (x, v_x), and halt returning the oracle's answer, which is a string $y \in R(x)$ such that $f(x,y) = v_x$. ■

Comments Regarding the Proof of Theorem 3.5. The first direction of the proof uses the hypothesis that f is polynomial-time computable, whereas the opposite direction only uses the fact that the optimal value lies in a finite space of exponential size that can be "efficiently searched." While the first direction is proved using a Levin-reduction, this seems impossible for the opposite direction (i.e., finding an optimal solution does not seem to be Levin-reducible to finding a solution that exceeds a threshold).

On the Complexity of R_f and R'_f. Here, we focus on the natural case in which $R \in \mathcal{PC}$ and f is polynomial-time computable. In this case, Theorem 3.5 asserts that R_f and R'_f are computationally equivalent. A closer look reveals, however, that $R_f \in \mathcal{PC}$ always holds, whereas $R'_f \in \mathcal{PC}$ does *not* necessarily hold. That is, the problem of finding a solution (for a given instance) that exceeds a given threshold is in the class $\mathcal{PC}$, whereas the problem of finding an optimal solution is not necessarily in the class $\mathcal{PC}$. For example, the problem of finding a clique of a given size K in a given graph G is in $\mathcal{PC}$, whereas the problem of finding a maximum-size clique in a given graph G is not known (and is quite unlikely)[3] to be in $\mathcal{PC}$ (although it is Cook-reducible to $\mathcal{PC}$).

The foregoing discussion suggests that the class of problems that are reducible to $\mathcal{PC}$, which seems different from $\mathcal{PC}$ itself, is a natural and interesting class. Indeed, for every $R \in \mathcal{PC}$ and polynomial-time computable f, the former class contains R'_f.

3.3 Self-Reducibility of Search Problems

The results to be presented in this section further justify the focus on decision problems. Loosely speaking, these results show that for many natural relations

[3] See Exercise 5.14.

R, the question of whether or not the search problem of R is efficiently solvable (i.e., is in $\mathcal{PF}$) is equivalent to the question of whether or not the "decision problem implicit in R" (i.e., $S_R = \{x : \exists y \text{ s.t. } (x,y) \in R\}$) is efficiently solvable (i.e., is in $\mathcal{P}$). In fact, we will show that these two computational problems (i.e., R and S_R) are computationally equivalent. Note that the decision problem of S_R is easily reducible to the search problem of R, and so our focus is on the other direction. That is, we are interested in relations R for which the search problem of R is reducible to the decision problem of S_R. In such a case, we say that R is self-reducible.[4]

Definition 3.6 (the decision implicit in a search and self-reducibility): *The* decision problem implicit in the search problem of R *is deciding membership in the set* $S_R = \{x : R(x) \neq \emptyset\}$, *where* $R(x) = \{y : (x,y) \in R\}$. *The search problem of* R *is called* self-reducible *if it can be reduced to the decision problem of* S_R.

Note that the search problem of R and the problem of deciding membership in S_R refer to the same instances: The search problem requires finding an adequate solution (i.e., given x find $y \in R(x)$), whereas the decision problem refers to the question of whether such solutions exist (i.e., given x determine whether or not $R(x)$ is non-empty). Thus, S_R corresponds to the intuitive notion of a "decision problem implicit in R," because S_R is a decision problem that one implicitly solves when solving the search problem of R. Indeed, for any R, *the decision problem of* S_R *is easily reducible to the search problem for* R (see Exercise 3.10). It follows that *if a search problem* R *is self-reducible, then it is computationally equivalent to the decision problem* S_R.

Note that the general notion of a reduction (i.e., Cook-reduction) seems inherent to the notion of self-reducibility. This is the case not only due to syntactic considerations, but is also the case for the following inherent reason. An oracle to any decision problem returns a single bit per invocation, while the intractability of a search problem in $\mathcal{PC}$ must be due to the lack of more than a "single bit of information" (see Exercise 3.12).

We shall see that self-reducibility is a property of many natural search problems (including all NP-complete search problems). This justifies the relevance of decision problems to search problems in a stronger sense than established

[4] Our usage of the term *self-reducibility* differs from the traditional one. Traditionally, a decision problem is called (downward) self-reducible if it is Cook-reducible to itself via a reduction that on input x only makes queries that are smaller than x (according to some appropriate measure on the size of instances). Under some natural restrictions (i.e., the reduction takes the disjunction of the oracle answers), such reductions yield reductions of search to decision (as discussed in the main text). For further details, see Exercise 3.16.

in Section 2.4: Recall that in Section 2.4, we showed that the fate of the search problem class $\mathcal{PC}$ (wrt $\mathcal{PF}$) is determined by the fate of the decision problem class $\mathcal{NP}$ (wrt $\mathcal{P}$). Here, we show that for many natural search problems in $\mathcal{PC}$ (i.e., self-reducible ones), the fate of such an individual problem R (wrt $\mathcal{PF}$) is determined by the fate of the individual decision problem S_R (wrt $\mathcal{P}$), where S_R is the decision problem implicit in R. (Recall that $R \in \mathcal{PC}$ implies $S_R \in \mathcal{NP}$.) Thus, here we have "fate reductions" at the level of individual problems, rather than only at the level of classes of problems (as established in Section 2.4).

3.3.1 Examples

We now present a few search problems that are self-reducible. We start with SAT (see Appendix A.2), the set of satisfiable Boolean formulae (in CNF), and consider the search problem in which given a formula one should find a truth assignment that satisfies it. The corresponding relation is denoted R_{SAT}; that is, $(\phi, \tau) \in R_{\text{SAT}}$ if τ is a satisfying assignment to the formula ϕ. Indeed, the decision problem implicit in R_{SAT} is SAT. Note that R_{SAT} is in $\mathcal{PC}$ (i.e., it is polynomially bounded, and membership of (ϕ, τ) in R_{SAT} is easy to decide (by evaluating a Boolean expression)).

Proposition 3.7 (R_{SAT} is self-reducible): *The search problem of* R_{SAT} *is reducible to* SAT.

Thus, the search problem of R_{SAT} is computationally equivalent to deciding membership in SAT. Hence, in studying the complexity of SAT, we also address the complexity of the search problem of R_{SAT}.

Proof: We present an oracle machine that solves the search problem of R_{SAT} by making oracle calls to SAT. Given a formula ϕ, we find a satisfying assignment to ϕ (in case such an assignment exists) as follows. First, we query SAT on ϕ itself, and return an indication that there is no solution if the oracle answer is 0 (indicating $\phi \notin \text{SAT}$). Otherwise, we let τ, initiated to the empty string, denote a prefix of a satisfying assignment of ϕ. We proceed in iterations, where in each iteration we extend τ by one bit (as long as τ does not set all variables of ϕ). This is done as follows: First we derive a formula, denoted ϕ', by setting the first $|\tau| + 1$ variables of ϕ according to the values $\tau 0$. We then query SAT on ϕ' (which means that we ask whether or not $\tau 0$ is a prefix of a satisfying assignment of ϕ). If the answer is positive, then we set $\tau \leftarrow \tau 0$ else we set $\tau \leftarrow \tau 1$. This procedure relies on the fact that if τ is a prefix of a satisfying assignment of ϕ and $\tau 0$ is not a prefix of a satisfying assignment of ϕ, then $\tau 1$ must be a prefix of a satisfying assignment of ϕ.

R, the question of whether or not the search problem of R is efficiently solvable (i.e., is in $\mathcal{PF}$) is equivalent to the question of whether or not the "decision problem implicit in R" (i.e., $S_R = \{x : \exists y \text{ s.t. } (x, y) \in R\}$) is efficiently solvable (i.e., is in $\mathcal{P}$). In fact, we will show that these two computational problems (i.e., R and S_R) are computationally equivalent. Note that the decision problem of S_R is easily reducible to the search problem of R, and so our focus is on the other direction. That is, we are interested in relations R for which the search problem of R is reducible to the decision problem of S_R. In such a case, we say that R is self-reducible.[4]

Definition 3.6 (the decision implicit in a search and self-reducibility): *The* decision problem implicit in the search problem of R *is deciding membership in the set* $S_R = \{x : R(x) \neq \emptyset\}$, *where* $R(x) = \{y : (x, y) \in R\}$. *The search problem of* R *is called* self-reducible *if it can be reduced to the decision problem of* S_R.

Note that the search problem of R and the problem of deciding membership in S_R refer to the same instances: The search problem requires finding an adequate solution (i.e., given x find $y \in R(x)$), whereas the decision problem refers to the question of whether such solutions exist (i.e., given x determine whether or not $R(x)$ is non-empty). Thus, S_R corresponds to the intuitive notion of a "decision problem implicit in R," because S_R is a decision problem that one implicitly solves when solving the search problem of R. Indeed, for any R, *the decision problem of* S_R *is easily reducible to the search problem for* R (see Exercise 3.10). It follows that *if a search problem* R *is self-reducible, then it is computationally equivalent to the decision problem* S_R.

Note that the general notion of a reduction (i.e., Cook-reduction) seems inherent to the notion of self-reducibility. This is the case not only due to syntactic considerations, but is also the case for the following inherent reason. An oracle to any decision problem returns a single bit per invocation, while the intractability of a search problem in $\mathcal{PC}$ must be due to the lack of more than a "single bit of information" (see Exercise 3.12).

We shall see that self-reducibility is a property of many natural search problems (including all NP-complete search problems). This justifies the relevance of decision problems to search problems in a stronger sense than established

[4] Our usage of the term *self-reducibility* differs from the traditional one. Traditionally, a decision problem is called (downward) self-reducible if it is Cook-reducible to itself via a reduction that on input x only makes queries that are smaller than x (according to some appropriate measure on the size of instances). Under some natural restrictions (i.e., the reduction takes the disjunction of the oracle answers), such reductions yield reductions of search to decision (as discussed in the main text). For further details, see Exercise 3.16.

in Section 2.4: Recall that in Section 2.4, we showed that the fate of the search problem class $\mathcal{PC}$ (wrt $\mathcal{PF}$) is determined by the fate of the decision problem class $\mathcal{NP}$ (wrt $\mathcal{P}$). Here, we show that for many natural search problems in $\mathcal{PC}$ (i.e., self-reducible ones), the fate of such an individual problem R (wrt $\mathcal{PF}$) is determined by the fate of the individual decision problem S_R (wrt $\mathcal{P}$), where S_R is the decision problem implicit in R. (Recall that $R \in \mathcal{PC}$ implies $S_R \in \mathcal{NP}$.) Thus, here we have "fate reductions" at the level of individual problems, rather than only at the level of classes of problems (as established in Section 2.4).

3.3.1 Examples

We now present a few search problems that are self-reducible. We start with SAT (see Appendix A.2), the set of satisfiable Boolean formulae (in CNF), and consider the search problem in which given a formula one should find a truth assignment that satisfies it. The corresponding relation is denoted R_{SAT}; that is, $(\phi, \tau) \in R_{\text{SAT}}$ if τ is a satisfying assignment to the formula ϕ. Indeed, the decision problem implicit in R_{SAT} is SAT. Note that R_{SAT} is in $\mathcal{PC}$ (i.e., it is polynomially bounded, and membership of (ϕ, τ) in R_{SAT} is easy to decide (by evaluating a Boolean expression)).

Proposition 3.7 (R_{SAT} is self-reducible): *The search problem of* R_{SAT} *is reducible to* SAT.

Thus, the search problem of R_{SAT} is computationally equivalent to deciding membership in SAT. Hence, in studying the complexity of SAT, we also address the complexity of the search problem of R_{SAT}.

Proof: We present an oracle machine that solves the search problem of R_{SAT} by making oracle calls to SAT. Given a formula ϕ, we find a satisfying assignment to ϕ (in case such an assignment exists) as follows. First, we query SAT on ϕ itself, and return an indication that there is no solution if the oracle answer is 0 (indicating $\phi \notin \text{SAT}$). Otherwise, we let τ, initiated to the empty string, denote a prefix of a satisfying assignment of ϕ. We proceed in iterations, where in each iteration we extend τ by one bit (as long as τ does not set all variables of ϕ). This is done as follows: First we derive a formula, denoted ϕ', by setting the first $|\tau| + 1$ variables of ϕ according to the values $\tau 0$. We then query SAT on ϕ' (which means that we ask whether or not $\tau 0$ is a prefix of a satisfying assignment of ϕ). If the answer is positive, then we set $\tau \leftarrow \tau 0$ else we set $\tau \leftarrow \tau 1$. This procedure relies on the fact that if τ is a prefix of a satisfying assignment of ϕ and $\tau 0$ is not a prefix of a satisfying assignment of ϕ, then $\tau 1$ must be a prefix of a satisfying assignment of ϕ.

We wish to highlight a key point that has been blurred in the foregoing description. Recall that the formula ϕ' is obtained by replacing some variables by constants, which means that ϕ' per se contains Boolean variables as well as Boolean constants. However, the standard definition of SAT disallows Boolean constants in its instances.[5] Nevertheless, ϕ' can be simplified such that the resulting formula contains no Boolean constants. This simplification is performed according to the straightforward Boolean rules: That is, the constant `false` can be omitted from any clause, but if a clause contains only occurrences of the constant `false`, then the entire formula simplifies to `false`. Likewise, if the constant `true` appears in a clause, then the entire clause can be omitted, and if all clauses are omitted, then the entire formula simplifies to `true`. Needless to say, if the simplification process yields a Boolean constant, then we may skip the query, and otherwise we just use the simplified form of ϕ' as our query. ■

Other Examples. Reductions analogous to the one used in the proof of Proposition 3.7 can also be presented for other search problems (and not only for NP-complete ones). Two such examples are searching for a 3-coloring of a given graph and searching for an isomorphism between a given pair of graphs (where the first problem is known to be NP-complete and the second problem is believed not to be NP-complete). In both cases, the reduction of the search problem to the corresponding decision problem consists of iteratively extending a prefix of a valid solution, by making suitable queries in order to decide which extension to use. Note, however, that in these two cases, the process of getting rid of constants (representing partial solutions) is more involved. Specifically, in the case of Graph 3-Colorability (resp., Graph Isomorphism), we need to enforce a partial coloring of a given graph (resp., a partial isomorphism between a given pair of graphs); see Exercises 3.13 and 3.14, respectively.

Reflection. The proof of Proposition 3.7 (as well as the proofs of similar results) consists of two observations.

1. For every relation R in $\mathcal{PC}$, it holds that the search problem of R is reducible to the decision problem of $S'_R = \{\langle x, y'\rangle : \exists y'' \text{ s.t. } (x, y'y'') \in R\}$. Such a reduction is explicit in the proof of Theorem 2.6 and is implicit in the proof of Proposition 3.7.

[5] While the problem seems rather technical in the current setting (since it merely amounts to whether or not the definition of SAT allows Boolean constants in its instances), the analogous problem is far from being so technical in other cases (see Exercises 3.13 and 3.14).

2. For specific $R \in \mathcal{PC}$ (e.g., S_{SAT}), deciding membership in S'_R is reducible to deciding membership in $S_R = \{x : \exists y \text{ s.t. } (x, y) \in R\}$. This is where the specific structure of SAT was used, allowing for a direct and natural transformation of instances of S'_R to instances of S_R.
   We comment that if S_R is NP-complete, then S'_R, which is always in $\mathcal{NP}$, is reducible to S_R by the mere hypothesis that S_R is NP-complete; this comment is elaborated in the following Section 3.3.2.

For an arbitrary $R \in \mathcal{PC}$, deciding membership in S'_R is not necessarily reducible to deciding membership in S_R. Furthermore, deciding membership in S'_R is not necessarily reducible to the search problem of R. (See Exercises 3.18, 3.19, and 3.20.)

In general, self-reducibility is a property of the search problem and not of the decision problem implicit in it. Furthermore, under plausible assumptions (e.g., the intractability of factoring), there exist relations $R_1, R_2 \in \mathcal{PC}$ having the same implicit-decision problem (i.e., $\{x : R_1(x) \neq \emptyset\} = \{x : R_2(x) \neq \emptyset\}$) such that R_1 is self-reducible but R_2 is not (see Exercise 3.21). However, for many natural decision problems, this phenomenon does not arise; that is, *for many natural NP-decision problems S, any NP-witness relation associated with S* (i.e., $R \in \mathcal{PC}$ such that $\{x : R(x) \neq \emptyset\} = S$) *is self-reducible*. For details, see the following Section 3.3.2.

3.3.2 Self-Reducibility of NP-Complete Problems

In this section, we assume that the reader has heard of NP-completeness. Actually, we only need the reader to know the definition of NP-completeness (i.e., a set S is $\mathcal{NP}$-complete if $S \in \mathcal{NP}$ and every set in $\mathcal{NP}$ is reducible to S). Indeed, the reader may prefer to skip this section and return to it after reading Section 4.1 (or even later).

Recall that, in general, self-reducibility is a property of the search problem R and not of the decision problem implicit in it (i.e., $S_R = \{x : R(x) \neq \emptyset\}$). In contrast, in the special case of NP-complete problems, self-reducibility holds for any witness relation associated with the (NP-complete) decision problem. That is, *all search problems that refer to finding NP-witnesses for any NP-complete decision problem are self-reducible.*

Theorem 3.8: *For every R in $\mathcal{PC}$ such that S_R is $\mathcal{NP}$-complete, the search problem of R is reducible to deciding membership in S_R.*

In many cases, as in the proof of Proposition 3.7, the reduction of the search problem to the corresponding decision problem is quite natural. The

following proof presents a generic reduction (which may be "unnatural" in some cases).

Proof: In order to reduce the search problem of R to deciding S_R, we compose the following two reductions:

1. A reduction of the search problem of R to deciding membership in $S'_R = \{\langle x, y' \rangle : \exists y'' \text{ s.t. } (x, y'y'') \in R\}$.
   As stated in Section 3.3.1 (in the paragraph titled "Reflection"), such a reduction is implicit in the proof of Proposition 3.7 (as well as being explicit in the proof of Theorem 2.6).
2. A reduction of S'_R to S_R.
   This reduction exists by the hypothesis that S_R is $\mathcal{NP}$-complete and the fact that $S'_R \in \mathcal{NP}$. (Note that we need not assume that this reduction is a Karp-reduction, and furthermore it may be an "unnatural" reduction).

The theorem follows. ■

3.4 Digest and General Perspective

Recall that we presented (polynomial-time) reductions as (efficient) algorithms that use functionally specified subroutines. That is, an efficient reduction of problem Π to problem Π' is an efficient algorithm that solves Π while making subroutine calls to any procedure that solves Π'. This presentation fits the "natural" ("positive") application of such a reduction; that is, *combining such a reduction with an efficient implementation of the subroutine* (that solves Π'), *we obtain an efficient algorithm for solving* Π.

We note that the existence of a polynomial-time reduction of Π to Π' actually means more than the latter implication. For example, a moderately inefficient algorithm for solving Π' also yields something for Π; that is, if Π' is solvable in time t' then Π is solvable in time t such that $t(n) = \text{poly}(n) \cdot t'(\text{poly}(n))$; for example, if $t'(n) = n^{\log_2 n}$ then $t(n) = \text{poly}(n)^{1+\log_2 \text{poly}(n)} = n^{O(\log n)}$. Thus, *the existence of a polynomial-time reduction of* Π *to* Π' *yields a general upper bound on the time complexity of* Π *in terms of the time complexity of* Π'.

We note that tighter relations between the complexity of Π and Π' can be established whenever the reduction satisfies additional properties. For example, suppose that Π is polynomial-time reducible to Π' by a reduction that makes queries of linear length (i.e., on input x each query has length $O(|x|)$). Then, if Π' is solvable in time t' then Π is solvable in time t such that $t(n) = \text{poly}(n) \cdot t'(O(n))$; for example, if $t'(n) = 2^{\sqrt{n}}$ then $t(n) = 2^{O(\log n)+\sqrt{O(n)}} = 2^{O(\sqrt{n})}$. We further note that bounding other complexity measures of the reduction (e.g., its space complexity) allows for relating the corresponding complexities of the problems.

In contrast to the foregoing "positive" applications of polynomial-time reductions, the theory of NP-completeness (presented in Chapter 4) is famous for its "negative" application of such reductions. Let us elaborate. The fact that Π is polynomial-time reducible to Π' means that *if solving* Π' *is feasible, then solving* Π *is feasible*. The direct "positive" application starts with the hypothesis that Π' is feasibly solvable and infers that so is Π. In contrast, the "negative" application uses the counter-positive: It starts with the hypothesis that solving Π is infeasible and infers that the same holds for Π'.

Exercises

Exercise 3.1 (a quiz)

1. What are Cook-reductions?
2. What are Karp-reductions and Levin-reductions?
3. What is the motivation for defining all of these types of reductions?
4. Can any problem in $\mathcal{PC}$ be reduced to some problem in $\mathcal{NP}$?
5. What is self-reducibility and how does it relate to the previous question?
6. List five search problems that are self-reducible. (See Exercise 3.15.)

Exercise 3.2 Verify the following properties of Cook-reductions:

1. Cook-reductions preserve efficient solvability: If Π is Cook-reducible to Π' and Π' is solvable in polynomial time, then so is Π.
2. Cook-reductions are transitive: If Π is Cook-reducible to Π' and Π' is Cook-reducible to Π'', then Π is Cook-reducible to Π''.
3. Cook-reductions generalize efficient decision procedures: If Π is solvable in polynomial time, then it is Cook-reducible to any problem Π'.

In continuation of the last item, show that a problem Π is solvable in polynomial time if and only if it is Cook-reducible to a trivial problem (e.g., deciding membership in the empty set).

Exercise 3.3 Show that Karp-reductions (and Levin-reductions) are transitive.

Exercise 3.4 Show that some decision problems are not Karp-reducible to their complement (e.g., the empty set is not Karp-reducible to $\{0,1\}^*$).

A popular exercise of dubious nature is showing that any decision problem in $\mathcal{P}$ is Karp-reducible to any *non-trivial* decision problem, where the decision