Texts in Computational Complexity: Four Advanced Topics Related to NP and NPC

Oded Goldreich
Department of Computer Science and Applied Mathematics
Weizmann Institute of Science, Rehovot, ISRAEL.

November 21, 2005

1 Main Text

We discuss four relatively advanced topics. These topics are typically not mentioned in a basic course on complexity. Still, pending on time constraints, we suggest discussing them at least at a high level.

Preliminaries. We denote by $\mathcal{PC}$ (standing for "Polynomial-time Check") the class of search problems that correspond to polynomially-bounded binary relations that have efficiently checkable solutions. That is, $R \in \mathcal{PC}$ if the following two conditions hold:

1. For some polynomial p, if $(x, y) \in R$ then $|y| \leq p(|x|)$.
2. There exists a polynomial-time algorithm that given (x, y) determines whether or not $(x, y) \in R$.

1.1 NP sets that are neither in P nor NP-complete

Thousands of problems have been shown to be NP-complete (cf., [2, Apdx.], which contains a list of more than three hundreds main entries). Things reached a situation in which people seem to expect any NP-set to be either NP-complete or in $\mathcal{P}$. This naive view is wrong: *Assuming $\mathcal{NP} \neq \mathcal{P}$, there exist sets in $\mathcal{NP}$ that are neither NP-complete nor in $\mathcal{P}$, where here NP-hardness allows also Cook-reductions.*

Theorem 1 *Assuming $\mathcal{NP} \neq \mathcal{P}$, there exist sets in $\mathcal{NP} \setminus \mathcal{P}$ such that some set in $\mathcal{NP}$ is not Cook-reducible to them.*

We mention that some natural problems (e.g., factoring) are conjectured to be neither solvable in polynomial-time nor NP-hard. One candidate class of such problems is $\mathcal{NP} \cap \mathrm{co}\mathcal{NP}$, where $\mathrm{co}\mathcal{NP} = \{\{0,1\}^* \setminus S : S \in \mathcal{NP}\}$ (see Section 1.4). Specifically, if $\Delta \stackrel{\rm def}{=} \mathcal{NP} \cap \mathrm{co}\mathcal{NP} \neq \mathcal{P}$ and $\mathcal{NP} \neq \mathrm{co}\mathcal{NP}$ then Δ is a class of sets that satisfy the conclusion of Theorem 1.[1] Below, the existence of sets satisfying this conclusion is proved based on the weaker assumption that $\mathcal{NP} \neq \mathcal{P}$ (which is actually a necessary condition for this conclusion). (We recommend to either state Theorem 1 without a proof or merely provide the proof idea.)

[1] This implication is based on Theorem 7.

Proof Sketch: The basic idea is modifying an arbitrary set in $\mathcal{NP} \setminus \mathcal{P}$ so as to fail all possible reductions (from $\mathcal{NP}$ to the modified set) as well as all possible polynomial-time decision procedures (for the modified set). Specifically, starting with $S \in \mathcal{NP} \setminus \mathcal{P}$, we derive $S' \subset S$ such that on one hand there is no polynomial-time reduction of S to S' while on the other hand $S' \in \mathcal{NP} \setminus \mathcal{P}$. The process of modifying S into S' proceeds in iterations, alternatively failing a potential reduction (by dropping sufficiently many strings from the rest of S) and failing a potential decision procedure (by including sufficiently many strings from the rest of S). Specifically, each potential reduction of S to S' can be failed by dropping finitely many elements from the current S', whereas each potential decision procedure can be failed by keeping finitely many elements of the current S'. These two assertions are based on the following two corresponding facts:

1. Any polynomial-time reduction (of any set not in $\mathcal{P}$) to a finite set (i.e., a finite subset of S) must fail, because only sets in $\mathcal{P}$ are Cook-reducible to a finite set. Thus, for any potential reduction (i.e., a polynomial-time oracle machine), there exists an input x on which this reduction fails. It follows that this failure is due to a finite set of queries (i.e., the set of all queries made by the reduction when invoked on an input that is smaller or equal to x). Thus, for every finite set $F \subset S$, any reduction of S to S' can be failed by dropping a finite number of elements from S' and without dropping elements of F.

2. For every finite set F, any polynomial-time decision procedure for $S \setminus F$ must fail, because S is (trivially) Cook-reducible to $S \setminus F$. Thus, for any potential decision procedure (i.e., a polynomial-time algorithm), there exists an input x on which this procedure fails. It follows that this failure is due to a finite prefix of S (i.e., the set $\{z \in S : z \leq x\}$). Thus, for every finite set F, any polynomial-time decision procedure for $S \setminus F$ can be failed by keeping a finite subset of $S \setminus F$.

As stated, the process of modifying S into S' proceeds in iterations, alternatively failing a potential reduction (by dropping finitely many strings from the rest of S) and failing a potential decision procedure (by including finitely many strings from the rest of S). This can be done efficiently because *it is inessential to determine the first possible points of alternation* (in which sufficiently many strings were dropped (resp., included) to fail the next potential reduction (resp., decision procedure)). It suffices to guarantee that adequate points of alternation (albeit highly non-optimal ones) can be efficiently determined. Thus, S' is the intersection of S and some set in $\mathcal{P}$, which implies that $S' \in \mathcal{NP} \setminus \mathcal{P}$. Following are some comments regarding the implementation of the foregoing idea.

The foregoing plan calls for an ("effective") enumeration of all polynomial-time oracle machines (resp., polynomial-time algorithms). However, none of these sets can be enumerated (by an algorithm). Instead, we enumerate all corresponding machines along with all possible polynomials, and for each pair (M, p) we consider executions of machine M with time bound specified by the polynomial p. That is, we use the machine M_p obtained from the pair (M, p) by suspending the execution of M on input x after $p(|x|)$ steps. We stress that we do not know whether machine M runs in polynomial-time, but the computations of any polynomial-time machine is "covered" by some pair (M, p).

Let us clarify the process in which reductions and decision procedures are ruled out. We present a construction of a "filter" set F in $\mathcal{P}$ such that the final set S' will equal $S \cap F$. Recall that we need to select F such that each polynomial-time reduction of S to $S \cap F$ fails, and each polynomial-time procedure for deciding $S \cap F$ fails. The key observation is that for every finite F each polynomial-time reduction of S to $S \cap F$ fails, whereas for every co-finite F (i.e., finite $\{0,1\}^* \setminus F$) each polynomial-time procedure for deciding $S \cap F$ fails. Furthermore, each of these failures occur on some input, and this input is determined by finite portions of S and F. Thus, we alternate between failing possible reductions and decision procedures, while not trying to determine the "optimal" points of alternation but rather determining points of alternation in a way that allows for efficiently deciding membership in F. Specifically, we let $F = \{x : f(|x|) \equiv 0 \bmod 2\}$, where $f : \mathbb{N} \to \{0\} \cup \mathbb{N}$ is defined next such that $f(n)$ can be computed in time poly(n).

The value of $f(n)$ is defined by the the following experiment that consists of exactly n^3 computation steps (where cubic time is selected to allow for some non-trivial manipulations of data as conducted next). For $i = 0, 1, ...$, we scan all inputs in lexicographic order trying to find an input on which the $i + 1^{\text{st}}$ (modified) machine fails (where this machine is an oracle machine if i is even and a standard machine otherwise). In order to determine whether or not a failure occurs on a particular input, we may need to know the value of $f(n')$ for some n', which we just compute recursively (while counting the recursive steps in our total number of steps).[2] The point is that, when considering an input x, we may need the values of f only on $\{1, ..., p_{i+1}(|x|)\}$, where p_{i+1} is the polynomial bounding the running-time of the $i + 1^{\text{st}}$ (modified) machine, and obtaining such a value takes at most $p_{i+1}(|x|)^3$ steps Also note that we may need to decide membership in $S \in \mathcal{NP}$, which we do by running the straightforward exponential-time algorithm (which tries all possible NP-witnesses). If we detect a failure of the $i + 1^{\text{st}}$ machine, we increase i and proceed to the next iteration. When we reach the allowed number of steps (i.e., n^3 steps), we halt outputting the current value of i (i.e., the current i is output as the value of $f(n)$). Indeed, it is most likely that we will reach n^3 steps before examining inputs of length $3 \log_2 n$, but this does not matter. What matters is that f is monotonically non-decreasing (because more steps allow to fail at least as many machines) as well as unbounded (see Exercise 10). Furthermore, by construction, $f(n)$ is computed in poly(n) time. $\square$

Comment: The proof of Theorem 1 actually establishes that *for every $S \notin \mathcal{P}$ there exists $S' \notin \mathcal{P}$ such that S' is Karp-reducible to S but S is not Cook-reducible to S'*.[3] Thus, if $\mathcal{P} \neq \mathcal{NP}$ then there exists an infinite sequence of sets $S_1, S_2, ...$ in $\mathcal{NP} \setminus \mathcal{P}$ such that S_{i+1} is Karp-reducible to S_i but S_i is not Cook-reducible to S_{i+1}. That is, there exists an infinite hierarchy of problems (albeit unnatural ones), all in $\mathcal{NP}$, such that each problem is "easier" than the previous ones (in the sense that it can be reduced to the previous problems while these problems cannot be reduced to it).

1.2 Promise Problems

Promise problems are a natural generalization of (search and decision) problems, where one explicitly considers a set of legitimate instances (rather than considering any string as a legitimate instance). As noted before, this provides a more adequate formulation of natural computational problems (and indeed this formulation is used in all informal discussions). In fact, standard presentation of such problems (see, e.g., [2]) use phrases like "given a graph and an integer..." (or "given a collection of sets..."). In other words, we assumed that the input instance has a certain format (or rather were "promised" that this is the case). Indeed, we claimed that in these cases the assumption can be removed without affecting the complexity of the problem, but we avoided providing a formal treatment of this issue, which is done next.

[2] We do not bother to present an efficient implementation of this process. That is, we may afford to recompute $f(n')$ every time we need it (rather than store it for later use).

[3] The said Karp-reduction (of S' to S) maps x to itself if $x \in F$ and otherwise maps x to a fixed no-instance of S.

We note that the notion of promise problems was originally introduced in the context of decision problems, and is typically used only in that context. However, we believe that it is as natural in the context of search problems, and present things accordingly.

1.2.1 Definitions

In the context of search problems, a promise problem is a relaxation in which one is only required to find solutions to instances in a predetermined set, called the promise. The requirement of efficient checking of solutions is adapted in an analogous manner.

Definition 2 (search problems with a promise): *A* search problem with a promise *consists of a binary relation $R \subseteq \{0,1\}^* \times \{0,1\}^*$ and a* promise *set P. Such a problem is also referred to as the* search problem R with promise P.

- *The search problem R with promise P is* solved by algorithm A *if for every $x \in P$ it holds that $(x, A(x)) \in R$ if $x \in S_R = \{x : R(x) \neq \emptyset\}$ and $A(x) = \perp$ otherwise, where $R(x) = \{y : (x,y) \in R\}$.*

  The time complexity of A on inputs in P *is defined as $T_{A|P}(n) \stackrel{\text{def}}{=} \max_{x \in P \cap \{0,1\}^n}\{t_A(x)\}$, where $t_A(x)$ is the running time of $A(x)$ and $T_{A|P}(n) = 0$ if $P \cap \{0,1\}^n = \emptyset$.*

- *The search problem R with promise P is in the promise problem extension of $\mathcal{PF}$ if there exists a polynomial-time algorithm that solves this problem.*[4]

- *The search problem R with promise P is in the promise problem extension of $\mathcal{PC}$ if there exists a polynomial T and an algorithm A such that, for every $x \in P$ and $y \in \{0,1\}^*$, algorithm A makes at most $T(|x|)$ steps and it holds that $A(x,y) = 1$ if and only if $(x,y) \in R$.*

We stress that nothing is required of the solver in the case that the input violates the promise (i.e., $x \notin P$); in particular, in such a case the algorithm may halt with a wrong output. (Indeed, the standard formulation of search problems is obtained by considering the trivial promise $P = \{0,1\}^*$.) In addition to the foregoing motivation for promise problems, we mention one natural class of search problems with a promise. These are search problem in which the promise is that the instance has a solution (i.e., in terms of the foregoing notation $P = S_R$). We refer to such search problems by the name candid search problems.

Definition 3 (candid search problems): *An algorithm A solves the* candid search problem of the binary relation R *if for every $x \in S_R \stackrel{\text{def}}{=} \{x : \exists y \text{ s.t. } (x,y) \in R\}$ it holds that $(x, A(x)) \in R$. The time complexity of such an algorithm is defined as $T_{A|S_R}(n) \stackrel{\text{def}}{=} \max_{x \in P \cap \{0,1\}^n}\{t_A(x)\}$, where $t_A(x)$ is the running time of $A(x)$ and $T_{A|S_R}(n) = 0$ if $P \cap \{0,1\}^n = \emptyset$.*

Note that nothing is required when $x \notin S_R$: In particular, A may output a wrong solution (although no solutions exist) or run for more than $T_{A|S_R}(|x|)$ steps. Note that for $R \in \mathcal{PC}$, *if we "know" the time complexity of algorithm A* (e.g., if we can compute $T_{A|S_R}(n)$ in poly(n)-time), then we may modify A into an algorithm A' that solves the (general) search problem of R (i.e., halts on each

[4]In this case it does not matter whether the time complexity of A is defined on inputs in P or on all possible strings. Suppose that A has polynomial time complexity T on inputs in P, then we can modify A to halt on any input x after at most $T(|x|)$ steps. This modification may only effects the output of A on inputs not in P (which is OK by us). The modification can be implemented in polynomial-time by computing $t = T(|x|)$ and emulating the execution of $A(x)$ for t steps. A similar comment applies to the definition of $\mathcal{PC}$, $\mathcal{P}$ and $\mathcal{NP}$.

input) in time $T_{A'}(n) = T_{A|S_R}(n) + \text{poly}(n)$. However, as we shall see in Section 1.3, the naive assumption by which we always know the running-time of an algorithm that we design is not necessarily valid.

Decision problems with a promise. In the context of decision problems, a promise problem is a relaxation in which one is only required to determine the status of instances that belong to a predetermined set, called the promise. The requirement of efficient verification is adapted in an analogous manner. In view of the standard usage of the term, we refer to *decision problems with a promise* by the name *promise problems*. Formally, promise problems refer to a three-way partition of the set of all strings into yes-instances, no-instances and instances that violate the promise. Standard decision problems are obtained as a special case by insisting that all inputs are allowed (i.e., the promise is trivial).

Definition 4 (promise problems): *A* promise problem *consists of a pair of non-intersecting sets of strings, denoted* $(S_{\text{yes}}, S_{\text{no}})$*, and* $S_{\text{yes}} \cup S_{\text{no}}$ *is called the* promise.

- *The promise problem* $(S_{\text{yes}}, S_{\text{no}})$ *is* solved by algorithm A *if for every* $x \in S_{\text{yes}}$ *it holds that* $A(x) = 1$ *and for every* $x \in S_{\text{no}}$ *it holds that* $A(x) = 0$*. The promise problem is in the promise problem extension of* $\mathcal{P}$ *if there exists a polynomial-time algorithm that solves it.*
- *The promise problem* $(S_{\text{yes}}, S_{\text{no}})$ *is in the promise problem extension of* $\mathcal{NP}$ *if there exists a polynomial* p *and a polynomial-time algorithm* V *such that the following two conditions hold:*
  1. Completeness: *For every* $x \in S_{\text{yes}}$*, there exists* y *of length at most* $p(|x|)$ *such that* $V(x, y) = 1$*.*
  2. Soundness: *For every* $x \in S_{\text{no}}$ *and every* y*, it holds that* $V(x, y) = 0$*.*

We stress that for algorithms of polynomial-time complexity, it does not matter whether the time complexity is defined only on inputs that satisfy the promise or on all strings (see Footnote 4). Thus, the extended classes $\mathcal{P}$ and $\mathcal{NP}$ (like $\mathcal{PF}$ and $\mathcal{PC}$) are invariant under this choice.

Reducibility among promise problems. The notion of a Cook-reduction extend naturally to promise problems, when postulating that a query that violates the promise (of the problem at the target of the reduction) may be answered arbitrarily.[5] The latter convention is consistent with the conceptual meaning of reductions and promise problems. Recall that reductions captures procedures that make subroutine calls to an arbitrary procedure that solves the reduced problem. But in case of promise problems, such a solver may behave arbitrarily on instances that violate the promise. Nevertheless, the main property of a reduction is preserved: if the promise problem Π is Cook-reducible to a promise problem that is solvable in polynomial-time, then Π is solvable in polynomial-time.

We warn that the extension of a complexity class to promise problems does not necessarily inherit the "structural" properties of the standard class. For example, in contrast to Theorem 7, there exists promise problems in $\mathcal{NP} \cap \text{co}\mathcal{NP}$ such that every set in $\mathcal{NP}$ can be Cook-reduced to them: see Exercise 11. Needless to say, $\mathcal{NP} = \text{co}\mathcal{NP}$ does not seem to follow from Exercise 11.

[5] It follows that Karp-reductions among promise problems are not allowed to make queries that violate the promise. Specifically, we say that the promise problem $\Pi = (\Pi_{\text{yes}}, \Pi_{\text{no}})$ is Karp-reducible to the promise problem $\Pi' = (\Pi'_{\text{yes}}, \Pi'_{\text{no}})$ if there exists a polynomial-time mapping f such that for every $x \in \Pi_{\text{yes}}$ (resp., $x \in \Pi_{\text{no}}$) it holds that $f(x) \in \Pi'_{\text{yes}}$ (resp., $f(x) \in \Pi'_{\text{no}}$).

1.2.2 Discussion

The following discussion refers both to the decision and search versions of promise problems. Recall that promise problems offer the most direct way to capture natural computational problems.

Restricting a computational problem. In addition to the foregoing motivation to promise problems, we mention their use in formulating the notion of a restriction of a computational problem to a subset of the instances. Specifically, such a restriction means that the promise set of the restricted problem is a subset of the promise problem of the unrestricted problem. For example, when we say that 3SAT is a restriction of SAT, we refer to the fact that the set of allowed instances is now restricted to 3CNF formulae (rather than to arbitrary CNF formulae). In both cases, the natural computational problem is to determine satisfiability (or to find a satisfying assignment), but the set of instances (i.e., the promise set) is further restricted in the case of 3SAT. The fact that a restricted problem is never harder than the original problem is captured by the fact that the restricted problem is reducible to the original one (via the identity mapping).

The standard convention of avoiding promise problems. Recall that, although promise problems provide a good framework for presenting natural computational problems, one usually manages to avoid this formulation. This is done by relying on the fact that for all the (natural) problems considered in the previous sections, it is easy to decide whether or not a given instance satisfies the promise. For example, given a formula it is easy to decide whether or not it is in CNF (or 3CNF). Actually, the issue arises already when talking about formulae: What we are actually given is a string that is supposed to encode a formula (under some predetermined encoding scheme), and so the promise (which is easy to decide for natural encodings) is that the input string is a valid encoding of a formula. In any case, if the promise is efficiently recognizable (i.e., membership in it can be decided in polynomial-time) then we may avoid mentioning the promise by using one of the following two "nasty" conventions:

1. *Extending the set of instances to the set of all possible strings* (and allowing trivial solutions for the corresponding dummy instances). For example, in the case of a search problem, we may either define all instance that violate the promise to have no solution or define them to have a trivial solution (e.g., be a solution for themselves); that is, for a search problem R with promise P, we may consider the (standard) search problem of R where R is modified such that $R(x) = \emptyset$ for every $x \notin P$ (or, say, $R(x) = \{x\}$ for every $x \notin P$). In the case of a promise (decision) problem $(S_{\rm yes}, S_{\rm no})$, we may consider the problem of deciding membership in $S_{\rm yes}$, which means that instances that violate the promise are considered as no-instances.

2. *Considering every string as a valid encoding of an object that satisfies the promise.* That is, fixing any string x_0 that satisfies the promise, we consider every string that violates the promise as if it were x_0. In the case of a search problem R with promise P, this means considering the (standard) search problem of R where R is modified such that $R(x) = R(x_0)$ for every $x \notin P$. Similarly, in the case of a promise (decision) problem $(S_{\rm yes}, S_{\rm no})$, we consider the problem of deciding membership in $S_{\rm yes}$.

We stress that in the case that the promise is efficiently recognizable the aforementioned conventions (or modifications) do not effect the complexity of the relevant (search or decision) problem. That is, rather that considering the original promise problem we consider a (search or decision) problem (without a promise) that is computational equivalent to the original one. Thus, in some sense we loss nothing by studying the latter problem rather than the original one. On the other hand, even in case the two problems are computationally equivalent, it is useful to have a formulation that allows to distinguish between them (as we do distinguish between the different NP-complete problems although they are all computationally equivalent). This conceptual concern becomes of crucial importance in the case (to be discussed next) that the promise is not efficiently recognizable.

The foregoing transformations of promise problems into computationally equivalent standard (decision and search) problems does not necessarily preserve the complexity of the problem in the case that the promise is not efficiently recognizable. In this case, the terminology of promise problems is unavoidable. Consider, for example, the problem of deciding whether a Hamiltonian graph is 3-colorable. On the face of it, such a problem may have fundamentally different complexity than the problem of deciding whether a given graph is both Hamiltonian and 3-colorable.

The notion of a promise problem provides an adequate formulation for a variety of computational complexity notions and results. Examples include the notion of "unique solutions" and the formulation of "gap problems" as capturing various approximation tasks.

1.2.3 The common convention

In spite of the foregoing opinions, we adopt the common convention of focusing on standard decision and search problems. That is, by default, all complexity classes refer to standard decision and search problems, and the exceptions in which we refer to promise problems are stated explicitly as such. Indeed, an exception appears in Section 1.3.

1.3 Optimal search algorithms for NP-relations

We refer to the candid search problem of any relation in $\mathcal{PC}$. Recall that $\mathcal{PC}$ is the class of search problems that allow for efficient checking of the correctness of candidate solutions (see preliminaries), and that the candid search problem is a search problem in which the solver is promised that the given instance has a solution (see Definition 3).

We claim the existence of an *optimal algorithm for solving any candid search problem of any relation in* $\mathcal{PC}$. Furthermore, we will explicitly present such an algorithm, and prove that it is optimal in a very strong sense: for any algorithm solving the candid search problem of $R \in \mathcal{PC}$, our algorithm solves the same problem in time that is at most a constant factor slower (ignoring a fixed additive polynomial term, which may be disregarded in case the problem is not solvable in polynomial-time). Needless to say, we do not know the time complexity of the aforementioned optimal algorithm (indeed if we knew it then we would have resolved the P-vs-NP Question). However, viewed differently, we "reduce" the P-vs-NP Question to determining the time complexity of a single explicitly presented algorithm (i.e., the optimal algorithm claimed in Theorem 5).

Theorem 5 *For every binary relation $R \in \mathcal{PC}$ there exists an algorithm A that satisfies the following:*

1. *A solves the candid search problem of R.*

2. *There exists a polynomial p such that for every algorithm A' that solves the candid search problem of R and for every $x \in S_R \stackrel{\text{def}}{=} \{x : R(x) \neq \emptyset\}$ it holds that $t_A(x) = O(t_{A'}(x) + p(|x|))$, where t_A* (resp., *$t_{A'}$*) *denotes the number of steps taken by A* (resp., *A'*) *on input x.*

Interestingly, we establish the optimality of A without knowing what its (optimal) running-time is. Furthermore, the optimality claim is "point-wise" (i.e., it refers to any input) rather than "global" (i.e., referring to the (worst case) time complexity as a function of the input length).

We stress that the hidden constant in the O-notation depends only on A', but in the following proof the dependence is exponential in the length of the description of algorithm A' (and it is not known whether a better dependence can be achieved). Indeed, this dependence as well as the idea underlying it constitute one negative aspect of this otherwise amazing result. Another negative aspect is that the optimality of algorithm A refers only to inputs that have a solution (i.e., inputs in S_R). Finally, we note that the theorem as stated refers only to models of computation that have machines that can emulate a given number of steps of other machines with a constant overhead. We mention that in most natural models the overhead of such emulation is at most poly-logarithmic in the number of steps, in which case it holds that $t_A(x) = \widetilde{O}(t_{A'}(x) + p(|x|))$.

Proof Sketch: Fixing R, we let M be a polynomial-time algorithm that decides membership in R, and let p be a polynomial bounding the running-time of M. We present the following algorithm A that merely emulates all possible search algorithms "in parallel" and checks the result provided by each of them (using M), halting whenever it obtains a correct solution.

Since there are infinitely many possible algorithms, it may not be clear what we mean by the expression "emulating all possible algorithms in parallel." What we mean is emulating them at different "rates" such that the infinite sum of these rates converges to 1 (or to any other constant). Specifically, we will emulate the i^{th} possible algorithm at rate $1/(i+1)^2$, which means emulating a single step of this algorithm per $(i+1)^2$ emulation steps (performed for all algorithms). Note that a straightforward implementation of this idea may create a significant overhead, involved in switching frequently from the emulation of one machine to the emulation of another. Instead, we present an alternative implementation that proceeds in iterations. In the j^{th} iteration, for $i = 1, ..., 2^{j/2} - 1$, algorithm A emulates $2^j/(i+1)^2$ steps of the i^{th} machine. Each of these emulations is conducted in one chunk, and thus the overhead of switching between the various emulations is insignificant (in comparison to the total number of steps being emulated). In the case that some of these emulations halts with output y, algorithm A invokes M on input (x, y) and output y if and only if $M(x, y) = 1$. Furthermore, the verification of a solution provided by a candidate algorithm is also emulated at the expense of its step-count. (Put in other words, we augment each algorithm with a canonical procedure (i.e., M) that checks the validity of the solution offered by the algorithm.)

Clearly, whenever $A(x)$ outputs y (i.e., $y \neq \perp$) it must hold that $(x, y) \in R$. To show the optimality of A, we consider an arbitrary algorithm A' that solves the candid search problem of R. Our aim is to show that A is not much slower than A'. Intuitively, this is the case because the overhead of A results from emulating other algorithms (in addition to A'), but the total number of emulation steps wasted (due to these algorithms) is inversely proportional to the rate of algorithm A', which in turn is exponentially related to the length of the description of A'. The punch-line is that since A' is fixed, the length of its description is a constant. Details follow.

For every x, let us denote by $t'(x)$ the number of steps taken by A' on input x, where $t'(x)$ also accounts for the running time of $M(x, \cdot)$; that is, $t'(x) = t_{A'}(x) + p(|x|)$, where $t_{A'}(x)$ is the number of steps taken by $A'(x)$. Then, the emulation of $t'(x)$ steps of A' on input x is "covered" by the j^{th} iteration of A, provided that $2^j/(2^{|A'|+1})^2 \geq t'(x)$ where $|A'|$ denotes the length of the description of A'. (Indeed, we assume that the algorithms are emulated in lexicographic order, and note that there are at most $2^{|A'|+1} - 2$ algorithms that precede A' in lexicographic order.) Thus, on input x, algorithm A halts after at most $j_{A'}(x)$ iterations, where $j_{A'}(x) = 2(|A'| + 1) + \log_2(t_{A'}(x) + p(|x|))$, after emulating a total number of steps that is at most

$$t(x) \stackrel{\text{def}}{=} \sum_{j=1}^{j_{A'}(x)} \sum_{i=1}^{2^{j/2}-1} \frac{2^j}{(i+1)^2} \;<\; 2^{j_{A'}(x)+1} \;=\; O\left(2^{2|A'|} \cdot (t_{A'}(x) + p(|x|))\right).$$

The question of how much time is required for emulating these many steps depends on the specific model of computation. In many models of computation, the emulation of t steps of one machine by another machine requires $\widetilde{O}(t)$ steps of the emulating machines, and in some models this emulation can even be performed with constant overhead. The theorem follows. $\square$

Comment: By construction, the foregoing algorithm A does not halt on input $x \notin S_R$. This can be easily rectified by letting A emulate a straightforward exhaustive search for a solution, and halt with output $\perp$ if this this exhaustive search indicates that there is no solution to the current input. This extra emulation will be done in parallel to all other emulations at a rate of one step for the extra emulation per each step of everything else.

1.4 The class coNP and its intersection with NP

By prepending the name of a complexity class (of decision problems) with the prefix "co" we mean the class of complement sets; that is,

$$\mathrm{co}\mathcal{C} \stackrel{\mathrm{def}}{=} \{\{0,1\}^* \setminus S : S \in \mathcal{C}\}$$

Specifically, $\mathrm{co}\mathcal{NP} = \{\{0,1\}^* \setminus S : S \in \mathcal{NP}\}$ is the class of sets that are complements of sets in $\mathcal{NP}$. Recalling that sets in $\mathcal{NP}$ are characterized by their witness relations such that $x \in S$ if and only if there exists an adequate NP-witness, it follows that their complement sets consists of all instances for which there are no NP-witness (i.e., $x \in \{0,1\}^* \setminus S$ if there is no NP-witness for x being in S).

Another perspective on $\mathrm{co}\mathcal{NP}$ is obtained by considering the search problems in $\mathcal{PC}$. Recall that for such $R \in \mathcal{PC}$, the set of instances having a solution (i.e., $S_R = \{x : \exists y \text{ s.t. } (x,y) \in R\}$) is in $\mathcal{NP}$. It follows that the set of instances having no solution (i.e., $\{0,1\}^* \setminus S_R = \{x : \forall y\ (x,y) \notin R\}$) is in $\mathrm{co}\mathcal{NP}$.

It is widely believed that $\mathcal{NP} \neq \mathrm{co}\mathcal{NP}$ (which means that $\mathcal{NP}$ is not closed under complementation). Indeed, this conjecture implies $\mathcal{P} \neq \mathcal{NP}$ (because $\mathcal{P}$ is closed under complementation). The conjecture $\mathcal{NP} \neq \mathrm{co}\mathcal{NP}$ means that some sets in $\mathrm{co}\mathcal{NP}$ do not have NP-proof systems (because $\mathcal{NP}$ is the class of sets having NP-proof systems). As we will show next, under this conjecture, the complements of NP-complete sets do not have NP-proof systems; for example, there exists no NP-proof system for proving that a given formula is not satisfiable. We first establish this fact for NP-completeness in the standard sense (i.e., under Karp-reductions).

Proposition 6 *Suppose that $\mathcal{NP} \neq \mathrm{co}\mathcal{NP}$ and let $S \in \mathcal{NP}$ such that every set in $\mathcal{NP}$ is Karp-reducible to S. Then $\overline{S} \stackrel{\mathrm{def}}{=} \{0,1\}^* \setminus S$ is not in $\mathcal{NP}$.*

Proof: We first observe that the fact that every set in $\mathcal{NP}$ is Karp-reducible to S implies that every set in $\mathrm{co}\mathcal{NP}$ is Karp-reducible to $\overline{S}$. We next claim that *if S' is in $\mathcal{NP}$ then every set that is Karp-reducible to S' is also in $\mathcal{NP}$*. Applying the claim to $S' = \overline{S}$, we conclude that $\overline{S} \in \mathcal{NP}$ implies $\mathrm{co}\mathcal{NP} \subseteq \mathcal{NP}$, which in turn implies $\mathcal{NP} = \mathrm{co}\mathcal{NP}$ in contradiction to the main hypothesis.

We now turn to prove the foregoing claim; that is, we prove that if S' has an NP-proof system and S'' is Karp-reducible to S' then S'' has an NP-proof system. Let V' be the verification procedure associated with S', and let f be a Karp-reduction of S'' to S'. Then, we define the verification procedure V'' (for membership in S'') by $V''(x,y) = V'(f(x),y)$. That is, any NP-witness that $f(x) \in S'$ serves as an NP-witness for $x \in S''$ (and these are the only NP-witnesses for $x \in S''$).

This may not be a "natural" proof system (for S''), but it is definitely an NP-proof system for S''. ■

Assuming that $\mathcal{NP} \neq \text{co}\mathcal{NP}$, Proposition 6 implies that sets in $\mathcal{NP} \cap \text{co}\mathcal{NP}$ cannot be NP-complete with respect to Karp-reductions. In light of other limitations of Karp-reductions (see, e.g., Exercise 9), one may wonder whether or not the exclusion of NP-complete sets from the class $\mathcal{NP} \cap \text{co}\mathcal{NP}$ is due to the use of a restricted notion of reductions (i.e., Karp-reductions). The following theorem asserts that this is not the case: *some sets in $\mathcal{NP}$ cannot be reduced to sets in the intersection $\mathcal{NP} \cap \text{co}\mathcal{NP}$ even under general reductions* (i.e., Cook-reductions).

Theorem 7 *If every set in $\mathcal{NP}$ can be Cook-reduced to some set in $\mathcal{NP} \cap \text{co}\mathcal{NP}$ then $\mathcal{NP} = \text{co}\mathcal{NP}$.*

In particular, assuming $\mathcal{NP} \neq \text{co}\mathcal{NP}$, no set in $\mathcal{NP} \cap \text{co}\mathcal{NP}$ can be NP-complete, even when NP-completeness is defined with respect to Cook-reductions. Since $\mathcal{NP} \cap \text{co}\mathcal{NP}$ is conjectured to be a proper superset of $\mathcal{P}$, it follows (assuming $\mathcal{NP} \neq \text{co}\mathcal{NP}$) that there are decision problems in $\mathcal{NP}$ that are neither in $\mathcal{P}$ nor NP-hard (i.e., specifically, the decision problems in $(\mathcal{NP} \cap \text{co}\mathcal{NP}) \setminus \mathcal{P}$). We stress that Theorem 7 refers to standard decision problems and not to promise problems (see Section 1.2 and Exercise 11).

Proof: Using the theorem's hypothesis, we will show that for every $S \in \text{co}\mathcal{NP}$ it holds that $S \in \mathcal{NP}$. Fixing any $S \in \text{co}\mathcal{NP}$, let $S' \in \mathcal{NP} \cap \text{co}\mathcal{NP}$ be a set such that S is Cook-reducible to S'. Such a reduction exists because $\overline{S} \stackrel{\text{def}}{=} \{0,1\}^* \setminus S$ is in $\mathcal{NP}$, and thus $\overline{S}$ is Cook-reducible to some set in $\mathcal{NP} \cap \text{co}\mathcal{NP}$ (whereas S is Cook-reducible to $\overline{S}$). Let us denote by M the oracle machine reducing S to S'. That is, on input x, machine M makes queries and decides whether or not to accept x, and its decision is correct provided all queries are answered according to S'.

To show that $S \in \mathcal{NP}$, we will present an NP-proof system for S. This proof system (or rather its verification procedure), denoted V, accepts a pair of the form $(x, ((z_1, \sigma_1, w_1), ..., (z_t, \sigma_t, w_t))$ if the following two conditions hold:

1. On input x, machine M accepts after making the queries $z_1, ..., z_t$, and obtaining the corresponding answers $\sigma_1, ..., \sigma_t$.

   That is, V check that, on input x, after obtaining the answers $\sigma_1, ..., \sigma_{i-1}$ to the first $i-1$ queries, the i^{th} query made by M equals z_i. In addition, V checks that M outputs 1 (indicating acceptance).

2. For every i, it holds that if $\sigma_i = 1$ then w_i is an NP-witness for $z_i \in S'$, whereas if $\sigma_i = 0$ then w_i is an NP-witness for $z_i \in \{0,1\}^* \setminus S'$.

   Thus, if this condition holds then it is the case that each σ_i indicates the correct status of z_i with respect to S' (i.e., $\sigma_i = 1$ if and only if $z_i \in S'$).

We stress that we use the fact that both S' and $\overline{S}' \stackrel{\text{def}}{=} \{0,1\}^* \setminus S$ have NP-proof systems, and refer to the corresponding NP-witnesses.

Note that V is indeed an NP-proof system for S. Firstly, the length of the corresponding witnesses is bounded by the running-time of the reduction (and the length of the NP-witnesses supplied for the various queries). Next note that V runs in polynomial time (i.e., verifying the first condition requires an emulation of the polynomial-time execution of M on input x when using the σ_i's to emulate the oracle, whereas verifying the second condition is done by invoking the relevant NP-proof systems). Finally, observe that $x \in S$ if and only if there exists a sequence

$y \stackrel{\text{def}}{=} ((z_1, \sigma_1, w_1), ..., (z_t, \sigma_t, w_t))$ such that $V(x,y) = 1$. In particular, $V(x,y) = 1$ holds only if y contains a valid sequence of queries and answers made by $M(x)$ to the oracle S' and M accepts based on that sequence. ■

The world view – a digest. Recall that on top of the $\mathcal{P} \neq \mathcal{NP}$ conjecture, we mentioned two other conjectures (which clearly imply $\mathcal{P} \neq \mathcal{NP}$):

1. The conjecture that $\mathcal{NP} \neq \text{co}\mathcal{NP}$ (equivalently, $\mathcal{NP} \cap \text{co}\mathcal{NP} \neq \mathcal{NP}$).

   This conjecture is equivalent to the conjecture that CNF formula have no short proofs of unsatisfiability (i.e., the set $\{0,1\}^* \setminus \texttt{SAT}$ has no NP-proof system).

2. The conjecture that $\mathcal{NP} \cap \text{co}\mathcal{NP} \neq \mathcal{P}$.

   Notable candidates for the class $\mathcal{NP} \cap \text{co}\mathcal{NP} \neq \mathcal{P}$ include decision problems that are computationally equivalent to the integer factorization problem (i.e., the search problem (in $\mathcal{PC}$) in which, given a composite number, the task is to find its prime factors).

Combining these conjectures, we get the world view depicted in Figure 1, which also shows the class of co$\mathcal{NP}$-complete sets (defined next).

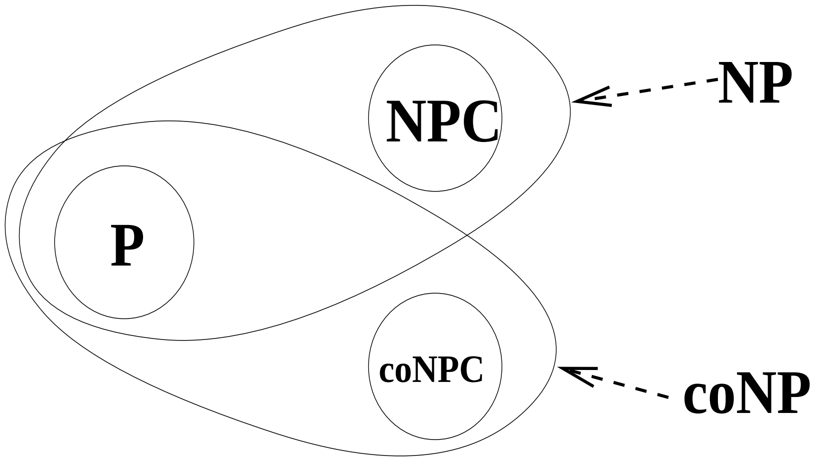

Figure 1: The world view under $\mathcal{P} \neq \text{co}\mathcal{NP} \cap \mathcal{NP} \neq \mathcal{NP}$.

Definition 8 *A set S is called* co$\mathcal{NP}$-hard *if every set in* co$\mathcal{NP}$ *is Karp-reducible to S. A set is called* co$\mathcal{NP}$-complete *if it is both in* co$\mathcal{NP}$ *and* co$\mathcal{NP}$*-hard.*

Indeed, insisting on Karp-reductions is essential for a distinction between $\mathcal{NP}$-hardness and co$\mathcal{NP}$-hardness.

2 Notes

The existence of NP-sets that are neither in P nor NP-complete (i.e., Theorem 1) was proven by Ladner [4], Theorem 7 was proven by Selman [6], and the existence of optimal search algorithms for NP-relations (i.e., Theorem 5) was proven by Levin [5]. (Interestingly, the latter result was proved in the same paper in which Levin presented the discovery of NP-completeness, independently of

Cook and Karp.) Promise problems were explicitly introduced by Even, Selman and Yacobi [1]; see [3] for a survey of their numerous applications.

Exercise 9 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 to show that any decision problem in $\mathcal{P}$ is Karp-reducible to any *non-trivial* decision problem, where the decision problem regarding a set S is called non-trivial if $S \neq \emptyset$ and $S \neq \{0,1\}^*$. It follows that every non-trivial set in $\mathcal{P}$ is Karp-reducible to its complement.

Exercise 10 Referring to the proof of Theorem 1, prove that the function f is unbounded (i.e., for every i there exists an n such that n^3 steps of the process defined in the proof allow for failing the $i+1^{\text{st}}$ machine).

Guideline: Consider n' such that $f(n') = i$. Assuming, towards the contradiction that $f(n) = i$ for every $n > n'$, it follows that $F = F'' \cup F''$, where $F' = \{x : |x| \leq n' \wedge f(|x|) \equiv 0 \pmod 2\}$ is a finite set and $F'' = \{x : |x| > n'\}$ if i is odd and $F'' = \emptyset$ otherwise. In case i is odd, the $i+1^{\text{st}}$ machine tries to decide $S \cap F$ (which differs from S on finitely many strings), and must fail on some x. Derive a contradiction by showing that the number of steps that the process makes till reaching and considering the input x is at most $\exp(\text{poly}(|x|))$. A similar argument applies to the case that i is even, where we use the fact that F is finite and so the relevant reduction of S to $S \cap F$ must fail on some input x.

Exercise 11 (NP-complete promise problems in coNP (following [1])) Consider the promise problem `xSAT`, having instances that are pairs of CNF formulae. The yes-instances consists of pairs (ϕ_1, ϕ_2) such that ϕ_1 is satisfiable and ϕ_2 is unsatisfiable, whereas the no-instances consists of pairs such that ϕ_1 is unsatisfiable and ϕ_2 is satisfiable.

1. Show that `xSAT` is in the intersection of (the promise problem classes that are analogous to) $\mathcal{NP}$ and co$\mathcal{NP}$.

2. Prove that any promise problem in $\mathcal{NP}$ is Cook-reducible to `xSAT`. In designing the reduction, recall that queries that violate the promise may be answered arbitrarily.

   Guideline: Show a reduction of `SAT` to `xSAT`. Specifically, show that the search problem associated with `SAT` is Cook-reducible to `xSAT`, by following the ideas used in the reduction of $R_{\texttt{SAT}}$ to `SAT` (i.e., the "self-reducibility" of SAT). Actually, we need a more careful implementation of the search process. Suppose that we know (or assume) that τ is a prefix of a satisfying assignment to ϕ, and we wish to extend τ by one bit. Then, for each $\sigma \in \{0,1\}$, we construct a formula, denoted ϕ'_σ, by setting the first $|\tau|+1$ variables of ϕ according to the values $\tau\sigma$. We query the oracle about the pair (ϕ'_1, ϕ'_0), and extend τ accordingly (i.e., we extend τ by the value 1 if and only if the answer is positive). Note that if both ϕ'_1 and ϕ'_0 are satisfiable then it does not matter which bit we use in the extension, whereas if exactly one formula is satisfiable then the oracle answer is reliable.

3. Pinpoint the source of failure of the proof of Theorem 7 when applied to the reduction provided in the previous item.

References

[1] S. Even, A.L. Selman, and Y. Yacobi. The Complexity of Promise Problems with Applications to Public-Key Cryptography. *Information and Control*, Vol. 61, pages 159–173, 1984.

[2] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, New York, 1979.

[3] O. Goldreich. On Promise Problems (a survey in memory of Shimon Even [1935-2004]). *ECCC*, TR05-018, 2005.

[4] R.E. Ladner. On the Structure of Polynomial Time Reducibility. *Journal of the ACM*, Vol. 22, 1975, pages 155–171.

[5] L.A. Levin. Universal Search Problems. *Problemy Peredaci Informacii 9*, pages 115–116, 1973. Translated in *problems of Information Transmission 9*, pages 265–266.

[6] A. Selman. On the structure of NP. *Notices Amer. Math. Soc.*, Vol. 21 (6), page 310, 1974.