"Strong" NP-Completeness Results: Motivation, Examples, and Implications

M. R. GAREY AND D. S. JOHNSON

Bell Laboratories, Murray Hill, New Jersey

ABSTRACT The NP-completeness of a computational problem is frequently taken to imply its "intractability" However, there are certain NP-complete problems involving numbers, such as PARTITION and KNAPSACK, which are considered by many practitioners to be tractable The reason for this is that, although no algorithms for solving them in time bounded by a polynomial in the input length are known, algorithms are known which solve them in time bounded by a polynomial in the input length *and* the magnitude of the largest number in the given problem instance. For other NP-complete problems involving numbers it can be shown that no such "pseudo-polynomial time" algorithm can exist unless P = NP. In this paper we provide a standard framework for stating and proving "strong" NP-completeness results of this sort, survey some of the strong NP-completeness results proved to date, and indicate some implications of these results for both optimization and approximation algorithms

KEY WORDS AND PHRASES NP-completeness, pseudo-polynomial time, approximation schemes

CR CATEGORIES 5 25

1. *Introduction*

The class of NP-complete problems [1, 13, 14] has been the focus of considerable theoretical and practical interest in recent years. This class contains many well-known and much-studied problems, e.g. the traveling salesman problem, the knapsack problem, and the graph coloring problem, and is characterized by two important properties:

(1) No NP-complete problem is known to be solvable by a polynomial time algorithm.

(2) If we had a polynomial time algorithm for *one* of the NP-complete problems, we could obtain polynomial time algorithms for *all* the NP-complete problems.

In the light of these two properties it is widely conjectured that no NP-complete problem can be solved by a polynomial time algorithm. For this reason the NP-complete problems are frequently considered to be computationally "intractable."

The issues we shall be discussing arise from a subtle point involved in the definition of "polynomial time algorithm." It is best illustrated by an example. Consider the PARTITION problem, in which we are given a finite set A of positive integers having overall sum $2b$ and are asked whether there exists a subset of A that sums exactly to b. Although this problem is known to be NP-complete [13], it can be solved by a straightforward dynamic programming algorithm, based on the following formulation.

Let $A = \{a_1, a_2, \ldots, a_n\}$ and let $T(i, j)$, $1 \leq i \leq n$, $0 \leq j \leq b$, denote the truth value of the statement "there exists a subset of $\{a_1, a_2, \ldots, a_i\}$ whose sum is exactly j." To start with, we know that $T(1, j)$ is false for all j except $j = 0$ and $j = a_1$. Inductively, we have that

Authors' address Bell Laboratories, 600 Mountain Ave , Murray Hill, NJ 07974

$T(i+1, j)$, $1 \leq i < n$, $0 \leq j \leq b$, is true if and only if either $T(i, j)$ is true or $a_{i+1} \leq j$ and $T(i, j - a_{i+1})$ is true. The desired subset for PARTITION exists if and only if $T(n, b)$ is true.

The obvious iterative algorithm based on this formulation can solve the PARTITION problem in time $\mathcal{O}(nb)$. Although this may appear at first glance to be a polynomial time algorithm, in fact it is not, and herein lies the subtle point to which we referred. For the purposes of the theory of NP-completeness, as well as for most formal complexity theory, the time complexity of an algorithm is expressed in terms of a single "instance size" parameter which reflects the number of symbols that would be required to describe the instance in a "reasonable" and "concise" manner. The parameter nb in our example does not do this. If an integer N is to be described "concisely," it must be represented using only $\mathcal{O}(\log N)$ symbols, as would be the case using its binary or decimal representations. Thus an instance of PARTITION could be described with only $\mathcal{O}(n \log b)$ symbols, and nb is not bounded by any polynomial function of this.

For this reason our $\mathcal{O}(nb)$ algorithm for PARTITION is not a polynomial time algorithm (in the sequel we shall refer to it as a "pseudo-polynomial time" algorithm). However, the existence of such an algorithm does indicate that the NP-completeness of PARTITION depends strongly on the fact that arbitrarily large values of the a_i are allowed. If any upper bound were imposed on these numbers in advance, even a bound which is a polynomial function of n, this algorithm *would* be a polynomial time algorithm for the restricted problem. In practice there might be many situations in which some such bound is imposed naturally on the input numbers.

For example, consider the following scheduling problem which is equivalent to PARTITION: Given n tasks $T_1, T_2, \ldots, T_n$ with lengths $a_1, a_2, \ldots, a_n$ summing to $2b$, is there a way of scheduling these tasks on two processors so that all tasks are completed by time b? Here a *schedule* is merely an assignment to each task T_i of a processor $P(i) \in \{1, 2\}$ and a starting time $S(i) \geq 0$, so that whenever $P(i) = P(j)$, the two execution intervals $(S(i), S(i) + a_i)$ and $(S(j), S(j) + a_j)$ are disjoint. The time at which all tasks are completed is given by $\text{Max}\{S(i) + a_i\}$. It is easy to see that the desired schedule exists if and only if there is a subset of $\{a_1, a_2, \ldots, a_n\}$ which sums exactly to b.

In this application of PARTITION to scheduling we observe that the proposed schedule length b would not be likely to be extremely large. After all, the schedule itself must take an acceptable amount of time or there would be no point in constructing it. And if b is an acceptable amount of time, then a running time of $\mathcal{O}(nb)$ for constructing that schedule might also be acceptable.

Thus, one must take care when interpreting an NP-completeness result for a problem involving numbers not to infer intractability in a broader sense than is justified by the theory. The ordinary NP-completeness results for such problems leave open the possibility of algorithms which are, for certain applications, suitably efficient to be used in practice.

We shall see, however, that not all NP-complete problems involving numbers are like PARTITION. For some the theory of NP-completeness can be used to essentially foreclose even the possibility of algorithms of the sort we have been discussing. In Section 2 we shall introduce some concepts which allow us to focus on the issues involved, including the notions of *number problem, pseudo-polynomial time algorithm,* and *strong NP-completeness.* Although these terms will be applied only at the level of abstract problems, we will show how they are interpreted at the language-theoretic level where NP-completeness is formally defined. In Section 3 we survey some of the strong NP-completeness results that have been proved to date, and in Section 4 we discuss some implications of these results for work on "approximation" algorithms. Finally, in Section 5, we briefly discuss some alternative terminology that has been proposed.

2. *Strong NP-Completeness and Pseudo-Polynomial Time*

We begin this section by reminding the reader of the formal definition of NP-completeness

in terms of languages. If Σ is a finite set of symbols, we let Σ^* denote the set of all finite strings of symbols from Σ. A *language* over Σ is any subset L of Σ^*. The three key concepts upon which the definition of NP-completeness is built are the two classes of languages, P and NP, and the notion of a *polynomial transformation* from one language to another.

The class P consists of all those languages which are recognizable by a deterministic Turing machine (a *DTM*) that runs in polynomial bounded time. The class NP consists of all those languages which can be recognized by a polynomial time-bounded "nondeterministic" Turing machine (an *NDTM*), and contains P as a subclass. (For an explanation of DTM's, NDTM's, and how they recognize languages, see [1].) If $L_1 \subseteq \Sigma^*$ and $L_2 \subseteq \Gamma^*$ are languages, a *polynomial transformation* from L_1 to L_2 is a function $f: \Sigma^* \to \Gamma^*$ which can be computed by a polynomial time-bounded DTM and which has the property that, for all $x \in \Sigma^*$, $x \in L_1$ if and only if $f(x) \in L_2$.

A language L is *NP-hard* if for each $L' \in$ NP there exists a polynomial transformation from L' to L. It is *NP-complete* if in addition $L \in$ NP. As a corollary we have that an NP-complete language L will be in P if and only if P = NP, so that the NP-complete languages are in a sense the "hardest" languages in NP.

The issues raised in Section 1, dealing as they do with the "numbers" in problem instances, are clearly meaningless at the formal language level where there *are* no numbers. (In addition there are no graphs, sets, functions, or any of the other combinatorial objects out of which abstract problems are built.) Thus the terminology we shall introduce to handle these issues must be defined at the abstract problem level and then *interpreted* in terms of the formal theory, as indeed must all discussions that are carried on at the problem level.

In preparation let us examine the usual correspondence between the two levels. At the problem level we restrict our attention to decision problems (problems where the solution is either the answer "yes" or the answer "no"). Such a problem Π can be viewed as consisting of a set D_Π of finite combinatorial objects called *instances* and a subset $Y_\Pi \subseteq D_\Pi$ of instances for which the answer is "yes." It usually is specified in a standard, two-part format, the first part describing a *generic instance* of the problem in terms of various components which are sets, graphs, functions, numbers, etc., and the second part stating a *question* asked in terms of the generic instance. For each instance $I \in D_\Pi$, this question, when particularized to I, will have answer "yes" if and only if $I \in Y_\Pi$.

The correspondence between such an abstract problem and a language is determined by an "encoding scheme" which is associated with the problem. The associated encoding scheme sets up a correspondence between instances of the problem and the strings over some alphabet Σ which "encode" them. The problem itself corresponds to the language over Σ^* consisting of all those strings which encode instances that have "yes" answers (that is, instances belonging to Y_Π). If this language is in P, we say that the problem itself is in P and can be solved by a polynomial time algorithm (algorithms correspond to DTM's). If the language is NP-complete, we say that the problem is NP-complete. This is, of course, an abuse of terminology, but it is rarely misleading since one observes that, if two different encoding schemes for the same abstract problem are both "reasonable" and "concise," then the resulting languages will either both be NP-complete or both not.

The terms "reasonable" and "concise" are unfortunately not rigorously definable and must be left to the common sense of the reader, as guided by the way in which the generic instance for the problem is specified. In essence, however, anyone claiming to apply the theory to a particular abstract problem implicitly asserts that he is prepared, if requested, to present a specific "reasonable and concise" encoding scheme for his problem and to show that the claimed result holds for the induced language. The demonstration of the result which is described at the abstract problem level provides an indication of how this would be done.

The terminology we shall introduce is based on two functions on problem instances which, as with encoding schemes, do not usually have to be specified explicitly, although such a specification is necessary for formal correctness. The first function we call

Length: $D_\Pi \to Z^+$ (the positive integers) and use to help specify what we mean by a "reasonable and concise" encoding scheme. Any such scheme will be required to have the following property: There exist polynomials p and p' such that if x is the string encoding instance $I \in D_\Pi$ and n is the number of symbols in x, then $n \leq p$ (Length[I]) and Length[I] $\leq p'(n)$.

For example in the PARTITION problem we might take Length[I] $= n \cdot \lceil \log_2 b \rceil$, Length[$I$] $= n + \lceil \log_2 b \rceil$, or a variety of other functions. If two observers have different notions of what Length[I] should be, it will normally not make a difference so long as both embody the same notions of reasonability and concision, in which case the two should be polynomially related, each bounded above by a polynomial function of the other. Generic instances of abstract problems are usually described in ways which suggest what Length functions the writer would find acceptable. If two observers fix on notions of Length which are *not* polynomially related, however, then we would say that they have actually fixed on two *different* abstract problems, since the corresponding "reasonable and concise" encoding schemes might well yield languages having different properties with respect to polynomial time.

Our second function on problem instances is Max: $D_\Pi \to Z^+$ and is intended to capture the notion of the magnitude of the largest number occurring in the instance. For example, in a PARTITION instance, Max[I] could be $\max\{a_i : 1 \leq i \leq n\}$. Once again, the function usually will not have to be specified explicitly, since generic problem instances usually are described in such a way as to make clear which of their components are numbers and the "numbers" that occur are usually restricted to integers. Moreover, in view of the fact that instances must be describable by finite strings, any more complicated "number" that might occur can be described symbolically in terms of integers, as with "13/9," "$\sqrt{5}$," or "$\cos(\pi/7)$."

As before two observers are allowed differing views as to what Max[I] should be without having an effect on our intended distinctions. In this case the requirement is that there be two-variable polynomials q and q' such that for all $I \in D_\Pi$,

$$\text{Max}[I] \leq q'(\text{Max}'[I], \text{Length}'[I]) \quad \text{and} \quad \text{Max}'[I] \leq q(\text{Max}[I], \text{Length}[I]),$$

where primes are used to differentiate the functions for the first observer from those of the second. Thus for PARTITION we could just as well take Max to be $\sum_{i=1}^{n} a_i$ as $\max\{a_i : 1 \leq i \leq n\}$. If, however, our two observers have notions of Max and Length which are not so related, we shall again consider them as viewing different problems.

There is one additional property of the functions Max and Length that we will require, which relates them more closely to "reasonable" encoding schemes. Implicit in the ways we "prove" NP-completeness results at the abstract problem level is the assumption that, in a "reasonable" encoding scheme, the encoded strings are "decodable." For instance, there must exist polynomial time DTM's which, given a string x over the alphabet of the encoding scheme, can determine whether x encodes an instance of the problem and, if so, can produce individual descriptions of each of the components that make up that instance. We shall require that Max and Length be such that, given any "reasonable" encoding scheme, there exist polynomial time DTM's which accept as input any string x encoding an instance I under that scheme and output the values of Length[I] and Max[I], written in binary notation.

In the definitions that follow we assume that every problem has associated with it a Length function, a Max function, and a "reasonable and concise" encoding scheme satisfying the properties described above. Thus, although the definitions are stated at the abstract problem level, their interpretation at the language level will be immediate.

Definition 1. A problem Π is a *number problem* if there exists *no* polynomial p such that for all instances I of Π, Max[I] $\leq p$(Length[I]).

Observe that not all problems involving numbers are number problems. For example, in the CLIQUE problem an instance I consists of a graph $G = (V, E)$, where V is the vertex set and E the edge set, and an integer $K \leq |V|$, and we are asked whether G contains

a complete subgraph on K vertices. The single number occurring in an instance, K, is bounded by $|V|$ and hence by Length[I]. Similarly, numerical subscripts which serve only to provide distinct names for objects must by "concision" have values bounded by a polynomial in Length[I] and hence can have no effect on whether a particular problem is a number problem.

Definition 2. A *pseudo-polynomial time algorithm* is an algorithm that runs in time bounded by a polynomial in the two variables Length[I] and Max[I].

Note that, by definition, a polynomial time algorithm is also a pseudo-polynomial time algorithm. We have, however, already seen an example of a pseudo-polynomial time algorithm which is *not* a polynomial time algorithm, the $\mathcal{O}(nb)$ algorithm for the PARTITION problem described in Section 1. Other examples can be found in [3, 10, 16, 17, 22]. All these algorithms work on dynamic programming principles similar to those used in our example.

If Π is a decision problem and p is a polynomial (over the integers), let Π_p denote the problem obtained by restricting Π to only those instances I for which Max[I] does not exceed p(Length [I]).

Definition 3. A problem Π is *NP-hard in the strong sense* if there exists a polynomial p (over the integers) such that Π_p is NP-hard. Π is *NP-complete in the strong sense* if in addition Π belongs to NP.

A few elementary observations will help point out the significance of these definitions.

OBSERVATION 1. *If Π is solvable by a pseudo-polynomial time algorithm, then for every polynomial p (over the integers), Π_p is solvable by a polynomial time algorithm.*

OBSERVATION 2. *If Π is NP-hard in the strong sense, then Π cannot be solved by a pseudo-polynomial time algorithm unless $P = NP$.*

OBSERVATION 3. *If Π is* not *a number problem, then an algorithm for Π is a pseudo-polynomial time algorithm if and only if it is a polynomial time algorithm.*

OBSERVATION 4. *If Π is* not *a number problem, then Π is NP-complete (NP-hard) if and only if it is NP-complete (NP-hard) in the strong sense.*

These observations make clear that our second and third definitions introduce nothing new if Π is not a number problem, but if Π is a number problem they introduce a nontrivial distinction (unless P = NP). Those problems which can be proved NP-complete in the strong sense have a more forceful claim to intractability than those which are merely NP-complete; they cannot even be solved by a *pseudo*-polynomial time algorithm unless P = NP. Thus for anyone interested in the computational complexity of a number problem, it is not enough simply to prove ordinary NP-completeness. One must go on to ask the questions: Is the problem also NP-complete in the strong sense? Does it have a pseudo-polynomial time algorithm? (Observe, however, that there could exist an NP-complete problem which is neither NP-complete in the strong sense nor solvable by a pseudo-polynomial time algorithm, just as there might exist problems in NP which are neither NP-complete nor polynomially solvable [15].)

How are strong NP-completeness results proved? One way to prove that Π is NP-complete in the strong sense is simply to exhibit a specific polynomial p and prove that Π_p is NP-complete in the usual way. A second, more powerful, technique uses the following definition and observation.

Definition 4. Let Π and Π' denote arbitrary decision problems with instance sets D_Π and $D_{\Pi'}$, "yes" sets Y_Π and $Y_{\Pi'}$, and specified functions Max, Length, Max′, and Length′, respectively. A *pseudo-polynomial transformation* from Π to Π' is a function $f: D_\Pi \to D_{\Pi'}$ such that

(a) For all $I \in D_\Pi$, $I \in Y_\Pi$ if and only if $f(I) \in Y_{\Pi'}$.

(b) f can be computed in time bounded by a polynomial in the two variables Max[I] and Length[I].

(c) There exists a polynomial p such that for all $I \in D_\Pi$, $p(\text{Length}'[f(I)]) \geq \text{Length}[I]$.

(d) There exists a polynomial q such that for all $I \in D_\Pi$,

$$\text{Max}'[f(I)] \leq q(\text{Max}[I], \text{Length}[I]).$$

OBSERVATION 5. *If* Π *is NP-hard in the strong sense and there exists a pseudo-polynomial transformation from* Π *to* Π', *then* Π' *is NP-hard in the strong sense.*

We shall see applications of pseudo-polynomial transformations in Section 3.

3. *Some Strongly NP-Complete Problems*

In this section we illustrate our definitions by presenting some examples. The simplest strong NP-completeness results are those in which we can bound Max[I] by a constant, and still have the restricted problem be NP-complete. For instance, restricting the MAX CUT problem of [13] to instances in which only edge weights of 0 and 1 are allowed yields the SIMPLE MAX CUT problem, shown NP-complete in [8]. The general TRAVELING SALESMAN problem, when similarly restricted, is of course the NP-complete HAMILTONIAN CIRCUIT problem [13]. A third example of this type is that of scheduling tasks on two processors subject to precedence constraints [23]. This problem can be solved in polynomial time if all the task lengths are equal, but is NP-complete if each task length may be either 1 or 2.

However, the full power of the definitions does not come into play until we consider problems Π which *can* be solved in polynomial time whenever there is a constant bound on Max[I], but which are nonetheless NP-complete in the strong sense (there exists a *non*constant polynomial p such that Π_p is NP-complete).

To our knowledge the first such problem discovered is 3-PARTITION, a close relative of the PARTITION problem discussed in Section 1. In 3-PARTITION, one is given a sequence of $3n$ nonnegative integers $A = \langle a_1, a_2, \ldots, a_{3n} \rangle$ whose sum is nb. The question asked is whether there exists a partition of the given integers into n disjoint groups of three such that each group sums exactly to b.

The proof that 3-PARTITION is NP-complete in the strong sense is too long and involved to repeat here. It can be found in [4], where it makes up the major portion of the arguments leading to the proof of Theorem 3.1, with 3-PARTITION being referred to there as "P[3, 1]." By a series of complicated transformations, the known NP-complete problem 3-DIMENSIONAL MATCHING is reduced to P[3, 1]. A careful examination of the transformations involved reveals in each step the existence of a polynomial which bounds Max[I] in terms of Length[I], so that the proof does indeed show that, in our terminology, 3-PARTITION is NP-complete in the strong sense.

The particularly simple structure of 3-PARTITION makes it quite useful for proving other strong NP-completeness results using pseudo-polynomial transformations. In fact 3-PARTITION can be substituted for PARTITION in many already existing NP-completeness proofs, thus converting ordinary NP-completeness results to strong NP-completeness results with little additional effort. One of the simplest examples of this concerns the BIN PACKING problem of [12]. Here we are given a set $\{x_1, x_2, \ldots, x_n\}$ of *items,* each item x_i having *size* s_i, and k bins of capacity B. We ask whether we can assign all the items to bins so that no bin receives items whose total size exceeds B. If $k = 2$, this problem is equivalent to the PARTITION problem, and this fact was the basis for the original NP-completeness proof. However, if $k = n/3$ it is not difficult to see how to embed 3-PARTITION into the problem in an analogous, pseudo-polynomial fashion, thus yielding a proof of strong NP-completeness for BIN PACKING.

So far, most of the other applications of 3-PARTITION for proving strong NP-completeness have been in the realm of scheduling theory. For those familiar with scheduling terminology we list a few of these below: (1) scheduling jobs in a 3-machine flowshop nonpreemptively (preemptively) to meet a given deadline [7] ([9]); (2) scheduling jobs in a 2-machine flowshop to meet a given bound on average finishing time [7]; (3) scheduling independent tasks on a single processor, subject to start times and deadlines [5]; (4) scheduling independent tasks on a single processor to meet a given bound on total weighted tardiness [18].

Finally we mention two examples of problems proved NP-complete in the strong sense using transformations from problems other than 3-PARTITION. In [6] the rectilinear

STEINER TREE problem is proved NP-complete by a transformation from NODE COVER [13] in which the numbers (point coordinates) are bounded by a polynomial in Length[I]. In [20] the Euclidean TRAVELING SALESMAN problem is proved NP-complete using a similarly bounded transformation from EXACT COVER [13].

4. *Implications for Approximation Algorithms*

Although NP-completeness results technically refer only to decision problems (and we are abusing our terminology even when we say this), they often are taken to apply to the corresponding optimization problems as well. This further abuse of terminology is not too misleading, as it is normally the case that the optimization problem can be solved in polynomial time if and only if the corresponding feasibility problem can be so solved. This "equivalence" continues to hold when "pseudo-polynomial time" is substituted for "polynomial time," so that strong NP-completeness results can also tell us about the possibility of finding pseudo-polynomial time optimization algorithms. In addition strong NP-completeness results have important implications concerning algorithms for finding "near-optimal" solutions to optimization problems, a topic with which this section is concerned.

Recent work by Ibarra, Kim, Sahni, and Horowitz [10, 11, 22] has provided numerous examples of a method for converting pseudo-polynomial time optimization algorithms (which *always* find optimal solutions) into polynomial time "approximation algorithms" (which only guarantee finding solutions that are "near" optimal). Actually, what is created might more appropriately be called an "approximation scheme."

An *approximation scheme* for an optimization problem Π is an algorithm A that takes two inputs: one an input I to the problem, the other a number ϵ, $0 < \epsilon \leq 1$, which prescribes the degree of accuracy desired. The algorithm then produces a solution for I with value $A(I)$, such that if $I^* \neq 0$ is the value of an optimal solution,

$$|A(I) - I^*|/|I^*| \leq \epsilon.$$

If for every fixed ϵ the algorithm A operates in time bounded by a polynomial in Length[I], we shall call A a *polynomial time approximation scheme.* If algorithm A operates in time bounded by a polynomial in the two variables Length[I] and $1/\epsilon$, we shall say that A is *fully polynomial.*

In the cited references, fully polynomial time approximation schemes are designed for the KNAPSACK problem and a number of scheduling problems. The basic technique involves rounding the input numbers in such a way that the given pseudo-polynomial optimization algorithm will work rapidly on them without introducing too large a cumulative error.

This technique does not necessarily work for *all* optimization problems that can be solved in pseudo-polynomial time, and in applications to date the resulting approximation schemes do not appear to be practical for small values of ϵ (say, less than 0.01). Nevertheless, the possibility that a fully polynomial time approximation scheme might exist is certainly worth considering when confronting a number problem known to be NP-complete.

It is here that strong NP-completeness results enter the picture. It can be shown that most problems cannot have a fully polynomial time aproximation scheme *unless* they also can be solved by a pseudo-polynomial time optimization algorithm.

THEOREM 1. *Suppose Π is an optimization problem such that, for all problem instances I, the only possible solution values are positive integers and the optimal solution value I^* is strictly bounded above by a polynomial q in the two variables Length[I] and Max[I]. Then if Π has a fully polynomial time approximation scheme A, it also has a pseudo-polynomial time optimization algorithm A'.*

PROOF. Suppose such an approximation scheme A exists. Our optimization algorithm A' proceeds as follows. Given I, set $\epsilon = (q(\text{Length}[I], \text{Max}[I]))^{-1}$. Then apply A to I and ϵ, constructing a solution in time polynomial in Length[I] and q(Length[I], Max[I]), and

hence in pseudo-polynomial time. By definition of approximation scheme, the value $A'(I)$ for this solution obeys

$$|A'(I) - I^*| \leq \epsilon I^* < I^*/q(\text{Length}[I], \text{Max}[I]) < 1,$$

since I^* is strictly bounded above by $q(\text{Length}[I], \text{Max}[I])$. But since all solution values are integers, this means that $A'(I) = I^*$. Thus A' is a pseudo-polynomial time optimization algorithm. □

The restrictions on Π in the statement of Theorem 1 may seem narrow at first glance. However, just as most combinatorial problems involving numbers can be converted into ones involving integers by polynomially bounded scaling, so too can most optimization problems be converted to the form desired here, without affecting the ratios between various solution values. This is indeed the case for the optimization versions of all the problems discussed in this paper. (Our results, and the whole concept of approximation scheme, of course do not apply when 0 is a possible optimal solution value.)

As a consequence of Theorem 1, if we can prove that a given optimization problem is NP-complete in the strong sense, we will have foreclosed the possibility not only of a pseudo-polynomial time optimization algorithm, but also of a fully polynomial time approximation scheme (unless P = NP).

It should be pointed out that Theorem 1 also has something to say about nonnumber optimization problems involving integers, such as CLIQUE, CHROMATIC NUMBER [13], SIMPLE MAX CUT [8], etc. In light of Observation 3 of Section 2, such a problem can have a fully polynomial time approximation scheme only if it has a polynomial time algorithm. Hence if the problem is NP-complete, it cannot have such a scheme unless P = NP. Similar conclusions can be derived for number problems like BIN PACKING [12], where I^* is bounded by a polynomial in Length[I] alone, independent of the value of Max[I].

One final caveat is in order, however. Even assuming P ≠ NP, none of these results implies the existence of a fixed error ratio $\epsilon > 0$ which cannot be guaranteed by any polynomial time algorithm. They do imply, though, that as ϵ approaches 0 such algorithms must become either very slow as a function of $1/\epsilon$ or else very hard to find.

5. *Alternative Formulations*

In preparing this paper we benefited greatly from the suggestions of a number of people (see Acknowledgments). Several alternative formulations were suggested which are worthy of mention, along with our reasons for not using them.

Perhaps the most obvious is the idea of *unary NP-completeness*. Here we drop the restriction that all numbers must be encoded with a number of symbols proportional to the logarithm of their magnitudes and require that all numbers be encoded in unary notation (representing the integer N by a string of N 1's). Except for this, the encoding scheme is still required to be "reasonable and concise." We call a problem unary NP-complete if it is NP-complete under such a unary encoding. It is not hard to see that this corresponds to our notion of strong NP-completeness.

Our main objection to this formulation is philosophical. One reason that NP-completeness is such an appealing concept is that, at the level of abstract problems, it is essentially encoding independent. Given any two "reasonable and concise" encoding schemes for a problem, the resulting two languages will either both be NP-complete or both not. The introduction of unary NP-completeness would make the theory encoding-dependent, and in fact would require us to use encoding schemes which certainly are *not* "reasonable and concise."

A related formulation, mentioned in [7], is to leave the standard restrictions on encodings untouched, but to allow input measures other than just the length of the encoded string. It is not difficult to redefine all the standard terminology so that each term is qualified by

"with respect to input measure $M[I]$." Ordinary and strong NP-completeness would then correspond to $M[I]$ = Length$[I]$ and $M[I]$ = Length$[I]$ + Max$[I]$, respectively, and would merely be two special cases of the general definition. Alternative input measures more complicated than these might well be useful. In particular, by allowing different problem parameters to contribute to the input measure in different ways, we might be able to address more finely detailed complexity questions.

However, this added power has disadvantages at the informal level, where we would regularly have to spell out the particular input measure we are using, instead of just assuming a natural one. In the approach we have taken strong NP-completeness can be applied to problems with the same precision as ordinary NP-completeness, even though we never explicitly spell out Length and Max functions, but trust to the reader's intuitive notion of "reasonableness."

Furthermore, although different input measures may provide a convenient way of asking detailed complexity questions, they give no particular insight into the way such questions are answered. Such questions are answered by placing restrictions on the problem instances allowed and then determining whether the restricted problem remains NP-complete (see for example [8]). In fact this is exactly the way in which strong NP-completeness is proved, the restriction being the polynomial bound on Max$[I]$ in terms of Length$[I]$.

Consequently, there is no need to change the basic definitions of NP-completeness by introducing new encodings or input measures. It seems more appropriate to build on the already accepted definitions, defining strong NP-completeness in terms of the restrictions that are used for proving it, and this is what we have done.

ACKNOWLEDGMENTS. A large number of people made useful comments based on a preliminary draft of this paper. Among these, we would especially like to thank A.V. Aho, E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, R.M. Karp, and R.E. Tarjan.

REFERENCES

(Note References [2, 18, 19, 21] are not cited in the text.)

1 AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974

2 COFFMAN, E. G., AND GRAHAM, R. L. Optimal scheduling for two-processor systems. *Acta Informatica 1* (1972), 200–213

3 DANTZIG, G. B. Discrete variable extremum problems. *Oper. Res. 5* (1957), 266–277

4 GAREY, M. R., AND JOHNSON, D. S. Complexity results for multiprocessor scheduling under resource constraints. *SIAM J. Comptng. 4* (1975), 397–411

5 GAREY, M. R., AND JOHNSON, D. S. Two processor scheduling with start times and deadlines. *SIAM J. Comptng. 6* (1977), 416–426

6 GAREY, M. R., AND JOHNSON, D. S. The rectilinear Steiner tree problem is *NP*-complete. *SIAM J. Appl. Math. 32* (1977), 826–834

7 GAREY, M. R., JOHNSON, D. S., AND SETHI, R. The complexity of flowshop and jobshop scheduling. *Math. Oper. Res. 1* (1976), 117–129

8 GAREY, M. R., JOHNSON, D. S., AND STOCKMEYER, L. Some simplified *NP*-complete graph problems. *Theoret. Comptr. Sci. 1* (1976), 237–267

9 GONZALEZ, T., AND SAHNI, S. Flow shop and job shop schedules. Tech. Rep. No. 75-14, Dept. of Comptr. Sci., U. of Minnesota, Minneapolis, Minn., 1975

10 HOROWITZ, E., AND SAHNI, S. Exact and approximate algorithms for scheduling nonidentical processors. *J. ACM 23*, 2 (April 1976), 317–327

11 IBARRA, O. H., AND KIM, C. E. Fast approximation algorithms for the knapsack and sum of subset problems. *J. ACM 22*, 4 (Oct. 1975), 463–468.

12 JOHNSON, D. S., DEMERS, A., ULLMAN, J. D., GAREY, M. R., AND GRAHAM, R. L. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM J. Comptng. 3* (1974), 299–325.

13 KARP, R. M. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, R.E. Miller and J. W. Thatcher, Eds., Plenum Press, New York, 1972, pp. 85–104

14 KARP, R. M. On the complexity of combinatorial problems. *Networks 5* (1975), 45–68

15 LADNER, R. E. On the structure of polynomial time reducibility. *J. ACM 22*, 1 (Jan. 1975), 155–171.

16 LAWLER, E. L. A "quasi-polynomial" algorithm for sequencing jobs to minimize total tardiness. Memo No. ERL-M558, Electron. Res. Lab., U. of California, Berkeley, Calif., 1975

17. LAWLER, E.L., AND MOORE, J.M. A functional equation and its application to resource allocation and sequencing problems *Manage. Sci. 16* (1969), 77–84.
18. LENSTRA, J.K., RINNOOY KAN, A H G, AND BRUCKER, P Complexity of machine scheduling problems *Annals Discrete Math 1* (1977), 343–362
19. NEMHAUSER, G L, AND YU, P.L A problem in bulk service scheduling *Oper Res 20* (1972), 813–819
20. PAPADIMITRIOU, C H, AND STEIGLITZ, K Some complexity results for the traveling salesman problem Proc. 8th Annual ACM Symp on Theory of Comptng, 1976, pp 1–9
21 SAHNI, S. Computationally related problems *SIAM J Comptng. 3* (1974), 262–279
22 SAHNI, S Algorithms for scheduling independent tasks *J ACM 23,* 1 (Jan 1976), 116–127
23 ULLMAN, J D *NP*-complete scheduling problems. *J Comptr Syst Sci 10* (1975), 384–393

RECEIVED OCTOBER 1976, REVISED DECEMBER 1977