Probabilistically Checkable Proofs and Approximating Solutions to Hard Problems

Kyle Littlefield

June 8, 2005

Contents

1 Introduction **2**

2 A Brief Introduction to Theory of Computation **3**

2.1 Some Notation 3

2.2 Modelling Computation - Turing Machines 3

2.3 **NP** Problems 5

2.4 Sample **NP** Problems 7

2.5 Optimizations for **NP** Problems 9

3 Defining NP in terms of probability **11**

3.1 Definition of PCP 11

3.2 Extending **NP** to **PCP**: An obvious connection 13

3.3 The PCP Theorem: **NP** $=$ **PCP** $(\log n, 1)$ 14

4 Applications of the PCP Theorem **14**

4.1 Establishing the Non-approximability of MAX-SNP problems . . 15

4.2 Clique Approximability 17

5 Conclusion **20**

1 Introduction

In theoretical computer science, one is often interested in the worst case behavior for an algorithm. It is not a surprise that there are some problems for which there is no known algorithm that does not, in its worse case, degenerate to a brute force search over all possibilities. Since the number of possibilities for many of these problems is exponential in the size of the input, such brute-force algorithms quickly become impractical as input size grows. The set of problems that can be solved by a brute force search is known as **NP**. The set of problems that are in **NP** are normally defined through a polynomial time, deterministic check of a "proof" applied to the input.

Of great interest is whether **NP** $=$ **P**, where **P** is the set of all problems for which an algorithm exists whose worst case running time is bounded by a polynomial $p(n)$ for all inputs of size n. Problems having such a solution are in a sense easy, in that a doubling of computational power multiplies the size of problems that can be solved by a constant factor, regardless of the current size of input that can be solved. For problems for which the best solution grows exponentially in input size, a doubling of computational power only adds a constant term to the size of problems that can be solved. Since computational power has (for quite a while) grown exponentially, problems solvable in polynomial time are those for which progress is being made "by leaps and bounds"; exponential growth problems can be described as "if you can't solve it today, you won't be able to anytime soon" or "you won't be able to solve much larger problems anytime soon".

Empirical evidence (i.e. lots of people trying to find efficient solutions to these problems) indicates that **NP** $\neq$ **P**; that there are some problems in **NP** that can not be solved in polynomial time by any algorithm. However, there is no proof **P** $\neq$ **NP** — indeed the problem is easily the most important and well-known unsolved problem in the field.

Probabilistically checkable proofs (**PCP**s) are a recent development that provide an alternate definition of **NP** that is based not on deterministic proof checking, but on randomized proof checking. In particular, recent results have shown that **NP** $=$ **PCP** $(\log n, 1)$. Showing that **NP**, a set of problems normally defined in deterministic terms, is equivalent to a set of problems whose solutions can be recognized in a probabilistic manner, provides new ways for tackling **NP** problems and showing interesting and useful results about them. Recent expositions [3] make this subject highly accessible. The purpose of this paper is to describe how **PCP**s work and to apply them to prove some performance limits for certain **NP** problems.

2 A Brief Introduction to Theory of Computation

Before one can consider probabilistically checkable proofs, it is necessary to start with some definitions of the set of problems under consideration, and the model of computation that is used to work with these problems in a theoretical setting.

2.1 Some Notation

Definition 2.1. *A function $f(n)$ is $O(g(n))$ if there are constants c and n_0 such that for all $n > n_0$, $f(n) < cg(n)$. This is just a formalization of the idea that the relative growth rate of one function $g(n)$ is at least as great as that of another function $f(n)$. We assume that both f and g are increasing for large n - this will be true of all functions considered in this paper.*

Corollary 2.2. *One immediate consequence of this definition is that a polynomial in x of degree k is $O\left(x^k\right)$.*

Definition 2.3. *An algorithm for solving a problem is $poly\,(n)$ if it is $O\left(n^k\right)$ for some k. Here n is the size of the input to the algorithm.*

2.2 Modelling Computation - Turing Machines

The standard model of computation is the Turing machine. Basically, a Turing machine is a model of a person sitting at a desk, reading from and writing to a tape of paper on the desk. At any point in time, the person can take his current state and read the single symbol on the tape directly in front of him. He can then erase this symbol and write a new one, choose to move the tape either to the left or the right, and transition to a new state. Formalizing this notion gives a concise definition of a Turing machine.

Definition 2.4. *A Turing machine consists of*

i. A finite set of states Q, one of which is the start state, one is the accept state, and one is the reject state.

ii. An input alphabet Σ.

iii. A tape alphabet Γ, including a unique end-of-input symbol and satisfying $\Sigma \subset \Gamma$

iv. *A transition function of the form* $Q \times \Gamma \rightarrow Q \times \Gamma \times \{$*Left or Right*$\}$

The tape alphabet is read from/written to a single, infinite length tape. There is one read head which is held in a fixed position while the tape moves. The transition function specifies for a given state of the machine and the symbol just read from the tape,

i. the new state to transition to

ii. the symbol to write onto the tape at the current position

iii. whether to move the read/write head to the left or the right

Figure 1 graphically shows a Turing machine (without showing the tape alphabet or transition function):

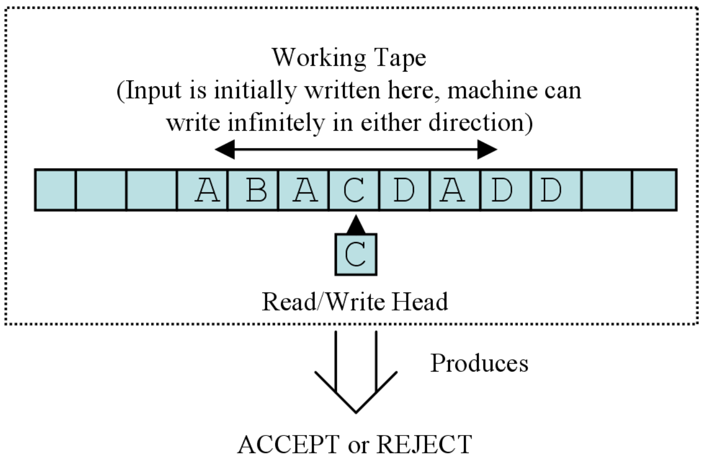
Working Tape
(Input is initially written here, machine can write infinitely in either direction)
A B A C D A D D
C
Read/Write Head
Produces
ACCEPT or REJECT

Figure 1: Graphical representation of a Turing machine

A Turing machine is executed by writing the input on the work tape, putting the machine in the start state and repeatedly executing the transition function for the current state and symbol. The machine terminates when it transitions into either the accept or reject state. Alternately, the machine may never halt. By appending a few state transitions to the accept/reject state, and redefining these states, we can further connect a Turing machine with the real world by having it print the answer on its tape before halting. If a machine M terminates in the accept state for an input x, then we say that M ACCEPTS x. Likewise, if it terminates in the reject state, we say that M REJECTS x. While this may seem like a limited model of computation, Turing machines can perform any computation done by the actual computers that exist in practice. Thus Turing machines provide a very simple, complete model of computation. [1]

As a further simplification, the alphabet Γ of the Turing machine is not really very important. Any Turing machine M has an equivalent Turing machine N that has a tape alphabet consisting of only $\{0, 1\}$. Equivalent in this case means that the same result (ACCEPT, REJECT, or run forever) is achieved by both machines for all inputs. We must slightly relax the requirements of the above description for this to work, by removing the requirement that Γ have a unique end-of-input symbol. Instead we require only that N be able to recognize the end of the input. The transformation works as follows. Let a be the size of the tape alphabet Γ, and $k = \lceil \log_2 a \rceil$. Then each symbol $s \in \Gamma$ can be given a unique k-digit binary representation. Inputs for M will be given as inputs to N after each symbol has been replaced by its k-digit encoding. The idea is then to decode a symbol by reading k tape positions (always transitioning to the right), then write the encoding of the symbol M would write (by transitioning k symbols to the left), then transition left or right by k tape positions to simulate the move M makes. This notion can be formalized - we will not do so here. The decoding of the symbol causes a 2^k blowup in the number of states of the machine, because each state has to have its own decoding routine. When running N, each transition of M corresponds to a $3k$ path of transitions in N, consisting of reading the encoded symbol of Γ in k transitions, traversing back over the encoded symbol to write the encoding of the symbol M would write in this situation, and finally traversing k tape spots to the left or right to simulate the single left/right movement of the transition of M. The increase in running time of this transformation is therefore bounded for all inputs by $3k$. For a given machine, this bound is a constant. Therefore, for any function $g(n)$ such that the running time of M is $O(g(n))$, the running time of N is also $O(g(n))$. The downside of this transformation is that it makes it harder to represent individual Turing machines such that a person can understand what is happening, but when considering Turing machines as a group of objects, eliminating having to deal with alphabets is a great simplification.

2.3 NP Problems

Moving on with defining things, we reach the class of problems known as **NP**. All problems in **NP** are decision problems, the answer is to either ACCEPT or

[1]Not extending to such realms as quantum computing.

REJECT the input, and we want to accept all inputs with a certain feature. The class **NP** is typically defined as the set of problems that have an efficient certifier M, which is a Turing machine such that:

i. For all $x \in L$ there exists a proof π_x such that M ACCEPTS (x, π_x) and the length of $\pi_x \leq dn^c$ (alternately, $poly(n)$), where n is the size of x and c and d are some positive constants. (The notation (x, y) as input for M means that x and y are given as input to M on its tape, separated by a special input symbol corresponding to the comma.)

ii. For all $x \notin L$, for all π, M REJECTS (x, π)

iii. M terminates in a time that is polynomial in n for all x and π

The meaning of $x \in L$ and $x \notin L$ will be made clear shortly. For now, just think of them as meaning, respectively, for all inputs having the desired feature, and for all inputs not having that feature. The proof for a verifier is not the same as a mathematical proof, which gives a series of justified steps to reach a conclusion. Here a proof is merely some suggestion to the verifier about how to determine (in polynomial time) that the input either has the feature or it doesn't. A proof for the verifier is like a hint for solving a math problem. It allows the verifier to go through a sequence of rigorous deductions to conclude that $x \in L$, but it is not necessarily this proof itself (although it could be). Note that for a given input, multiple such proofs likely exist — a verifier does not have to accept every proof that shows $x \in L$, it need only accept one. This normally corresponds to the verifier expecting the input to be formatted in a certain way.

How do we translate problems involving real objects and complex representations into something that can be put into a Turing machine? This is done by simply picking an encoding and using it. For example, if we want to encode a graph, this may be done by encoding first the number of vertices in binary and then the adjacency matrix representation of the graph. The idea here is to just flatten the representation of the problem to one that can be written on the single working tape of a Turing machine.

Additionally, when we think about **NP** problems, we assume that there are a finite number of possible "proofs" that must be considered to determine whether to accept the input. However, there is no need to explicitly add this as a requirement to the above; it is already contained in the definition. This is simple to see, since we can just check every possible proof up to length dn^c, running the verifier on each. If any accepts, then the input is accepted, otherwise it is rejected. The number of such possible proofs is $\sum_{k=1}^{dn^c} 2^k = 2^{dn^c+1} - 1$. Clearly this number is finite but exponential in terms of input size. Thus any problem in **NP** can be solved by a search over a number of possibilities that is exponential in the size of the input.

Next we move on to **NP** as language recognition, which will clarify the meaning of $x \in L$ and $x \notin L$ above. To start with, the definition of a language is:

Definition 2.5. *A language is a set (either finite or infinite) of strings over an alphabet of symbols.*

For example, one language is the set of strings that can be written using the English alphabet that contain only A's and B's, and are one or more symbols long. This language is $\{$A, B, AA, AB, BA, BB, AAA, $\ldots\}$.

Take the set of all problem inputs such that the solver should accept. Then the encodings of all of these inputs relative to a given encoding scheme constitutes a language. Thus every **NP** problem is one of deciding whether a given string belongs in a language. I.e. **NP** problems are language recognition problems.

This definition in terms of an efficient verifier formalizes the notion that problems in **NP** can be solved by brute-force search over a finite number of possibilities, and that it is easy to write a solver for any **NP** problem. (Given the verifier M, the constants c, and d, and the encoding scheme.) Since **P** $\subseteq$ **NP**, there are clearly some problems in **NP** that can be solved efficiently, i.e. in polynomial time. This is clear, since some languages (such as the example above) have an obvious polynomial time recognition algorithm. For the example, a single linear pass of the input suffices. The question is whether a particular **NP** problem has such a polynomial solution algorithm or whether no algorithm does significantly better than the brute-force search strategy given above. There is no general technique for determining the answer to this question.

2.4 Sample NP Problems

The staple problem in **NP** is 3SAT, the purpose of which is to determine if a boolean formula has an assignment of values to its variables such that the statement is true. The form of the boolean formula is tightly constrained — it is a conjunction of disjunctions, each of which has three base variables, which may or may not be negated. While this form seems rather restrictive, any boolean formula which uses AND, OR, and NOT can be reduced to the given form through a sequence of simple manipulations.

An example 3SAT problem would be to find assignments of true or false for A, B, C, and D such that the following is true:

$$(A \vee B \vee C) \wedge (\neg A \vee \neg B \vee D) \wedge (\neg B \vee \neg C \vee \neg D)$$

In the example given, assigning true to A, false to B, true to C, and true to D is one satisfying assignment (there are others). Clearly, there is a verifier that runs in $O(n)$ time for 3SAT. The proofs for this verifier consist simply of an assignment of either true or false to each variable. The verifier then simply plugs these values into the formula, taking $O(n)$ time to evaluate the formula. In addition, the proof can be no longer than the number of variables, which is certainly less than the entire input. So proofs are size bounded by $O(n)$. Thus 3SAT is in **NP** by the definition of an efficient certifier.

What is not obvious about the 3SAT problem is that it is at least as hard as any other problem in **NP**. This means that any other instance of a problem in **NP** can be encoded in a 3SAT instance through a polynomial (in time and size) process. This process is called a reduction. (The proof for this can be found in any introductory theory of computation book.) The reduction being polynomial is crucial, since if this is the case, a polynomial 3SAT algorithm could be used to solve any other problem in **NP** in polynomial time. If the reduction itself does not take polynomial time, then nothing can be determined. Thus a polynomial time algorithm for solving 3SAT would allow all **NP** problems to be solved in polynomial time. Such a problem (and there are many) for which a polynomial time solution implies every **NP** problem can be done in polynomial time is called **NP**-complete.

Another problem in NP is Clique, the goal of which is to determine for an input graph G and an integer k, if there is a set of k vertices in G, each pair of which have an edge between them. Figure 2 gives an example in which the largest clique is of size 4, consisting of the set $\{C, D, F, G\}$. Thus the result would be ACCEPT for any $k \leq 4$ and REJECT for any $k > 4$.

Another simple problem on graphs is the Independent Set problem, which is exactly like Clique, except that the set of vertices being sought should have no edge between any pair of vertices in the set. The connection between the two is that solving Independent Set is as simple as just flipping all the edges (adding any edge that doesn't exist and removing any that do) and then solving Clique on this graph. Like 3SAT, both of these problems are **NP**-complete.

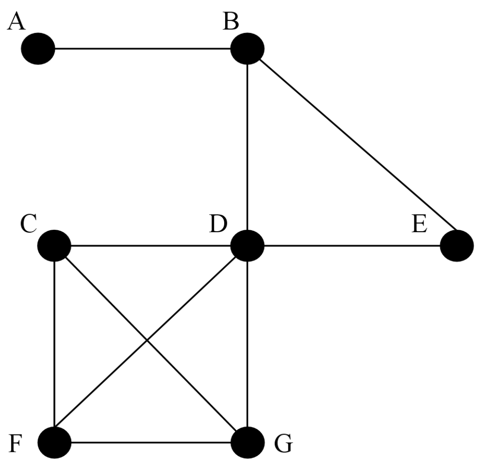

Figure 2: A sample Clique problem

2.5 Optimizations for NP Problems

The above **NP**-complete problems (and many others like them) seem to be very difficult to solve, since no polynomial time algorithm has been found for any of them. Notice though that all problems in **NP** are decision problems, the only answers that are available are ACCEPT and REJECT. Most problems in **NP** have a related optimization problem. We are interested in these optimizations because they may have polynomial time solutions that guarantee a certain level of performance. For 3SAT, the obvious optimization problem is to satisfy the largest number of clauses (call this problem MAX-3SAT). For Clique it is to find the largest clique in a graph (call this problem MAX-CLIQUE). The performance ratio of a maximization optimization problem is defined to be the maximum value of $\frac{OPT(x)}{c(x)}$ over all possible inputs x, where $c(x)$ is the cost function of the solution found by an approximation algorithm and $OPT(x)$ is the cost function of the optimal solution for the input. Naming this object the cost function makes it sound complicated, but it is normally something quite simple. For example, in MAX-3SAT it is the number of clauses satisfied. For MAX-CLIQUE it is the size of the clique found. Clearly, no solution can do better than optimal, so this definition makes all performance ratios ≥ 1. Some **NP** problems have an optimization that is a minimization. The performance ratio for a minimization optimization problem is defined as the maximum value of $\frac{c(x)}{OPT(x)}$, and is also ≥ 1.

Unlike the **NP** decision problems, where there is a whole set of equivalent **NP**-complete problems upon which research can be focused, optimization problems have a much wider range of behavior and little correlation between one another. For the most part, each optimization problem must be treated individually; conclusions about one do not apply to others.

Current research into these optimization problems indicates that they fall into four main categories

i. Problems for which a $poly\left(n\right)$ solution is known that achieves a ratio of 1. Obviously these correspond to the class **P** and are of no further interest to the discussion at hand.

ii. Problems for which a $poly\left(n\right)$ solution can be found which achieves a performance ratio of $1+\epsilon$ for any $\epsilon>0$. Note that the value of ϵ affects the complexity of the algorithm necessary to find the approximation. In general, the smaller the value of ϵ, the higher the degree and/or coefficients of the polynomial approximation algorithm that achieves the $1+\epsilon$ performance ratio. A problem for which this is the case is said to have a polynomial time approximation scheme (PTAS).

iii. Problems for which a $poly\left(n\right)$ algorithm can be found that achieves a performance ratio of $1+\epsilon$ for any ϵ greater than some constant $\delta, \delta>0$. But for $\epsilon<\delta$ no polynomial time algorithm exists.

iv. Problems for which no $poly\left(n\right)$ algorithm achieves a performance ratio of $1+\epsilon$ for any $\epsilon\geq 0$. This category can be further subdivided based on looking at the performance measure as a function of input size. For example, problems whose best algorithm can achieve a performance ratio of $O\left(\log n\right)$, $O\left(\log\log n\right)$, and $O\left(\sqrt{n}\right)$ all fall in this category, but are very different.

In terms of interesting proofs, categories one and two are fairly boring. All that has to be done is to find an algorithm (for case 1), or an algorithm-creation algorithm, which takes ϵ and returns a polynomial algorithm with the desired performance. In each case, these can be proved constructively. The last two categories are much more interesting, since one must argue not just about the performance of those algorithms that have been created, but also about all imaginable algorithms. Showing which category a problem belongs to (and possibly finding δ in case 3) is the problem of proving a lower bound for an approximation problem. The current state of research in optimization problems is that for many problems we do not have any lower bound, or we have only a lower bound that differs greatly from the performance of the best known algorithm for the problem. Clearly, the best situation is to have the two bounds (algorithmic and theoretical) agree, since at this point we can stop looking for better algorithms. In order to do this, we must first establish which category a problem falls in, then establish a lower bound. As will be shown later in this paper, **PCP**s have in some instances been used to do this.

3 Defining NP in terms of probability

In this section we will define classes of verifiers which can verify language membership proofs probabilistically, and then state that one of these classes is equivalent to **NP**. The specific classes of verifiers under discussion are probabilistically checkable proofs (**PCP**s).

3.1 Definition of PCP

PCPs are exactly what one might guess from their name, proof checking schemes with randomness. As in the classic definition of **NP**, **PCP** relies on verifiers which are Turing machines. The machines that will be used have a few additions from those described in the previous section.

A language L is in **PCP** $(q(n), r(n))$ if it has a $(q(n), r(n))$ verifier. A verifier V for this purpose is a polynomial time Turing machine with a few additions. First, it has access (via another tape) to an infinite stream τ of random bits. Secondly, it has an oracle which takes as input a position in the proof and returns the value of the bit of the proof at that position. Access to the oracle is controlled by a third tape. The oracle here is not any sort of pseudo-magical question answering device. All it does is access any location in the proof in constant time; doing this is outside the normal model of a Turing machine. So from the point-of-view of the Turing machine, the oracle is magic, but from our perspective what it is doing is reasonable and straightforward. While adding two tapes may make a Turing machine seem much more powerful, it is an elementary result that multi-tape Turing machines (such as this) are no more powerful than single tape Turing machines. Figure 3 graphically displays a PCP verifier.

A verifier is $(q(n), r(n))$ if for all inputs of length n it reads at most $q(n)$ random bits, and queries at most $r(n)$ bits of the proof. A verifier for a language L must satisfy

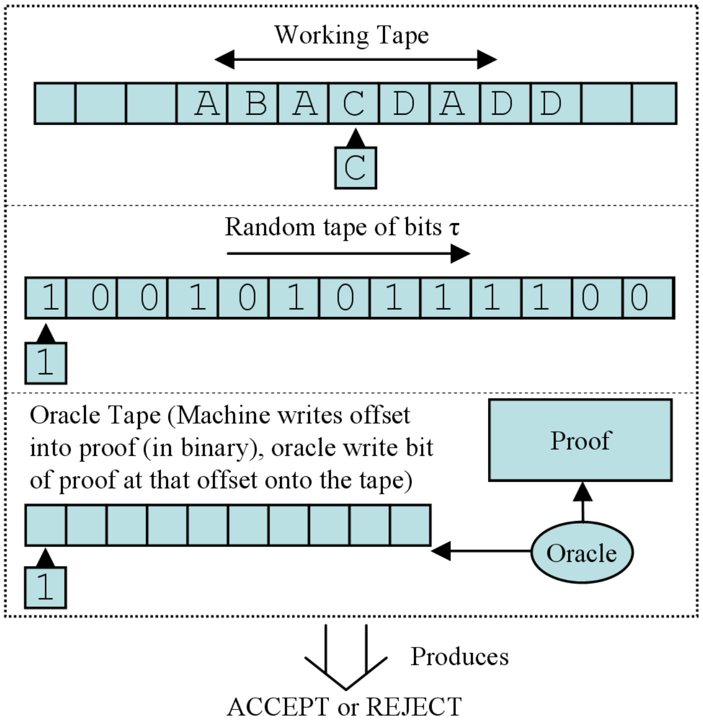
Working Tape
A B A C D A D D
C
Random tape of bits τ
1 0 0 1 0 1 0 1 1 1 1 0 0
1
Oracle Tape (Machine writes offset into proof (in binary), oracle write bit of proof at that offset onto the tape)
Proof
Oracle
1
Produces
ACCEPT or REJECT

Figure 3: A PCP Verifier

i. For all $x \in L$, there is a proof π_x such that

   $Prob_{q(n)}\left[V \text{ ACCEPTS } (x, \tau, \pi_x)\right] = 1$,

ii. For all $x \notin L$, for all proofs π

   $Prob_{q(n)}\left[V \text{ ACCEPTS } (x, \tau, \pi)\right] < \frac{1}{2}$

The probability is taken over the uniform distribution of all possible binary strings τ of length $q(n)$. Furthermore, the bits of the proof π that V queries depend only on x and τ, not on any bits of the proof previously read. Note the similarity between this and the classic deterministic verifiers for **NP** problems. The behavior of the verifier works as follows. For all inputs in the language, there exists at least one proof for which the verifier always accepts the input. This is exactly the same as the classic deterministic verifiers. For inputs not in the language, or for proofs not in the form the verifier expects, the verifier will reject the input at least half the time.

Initially, it might seem that the oracle is unnecessary to read the proof. But in fact it is critical if $r(n)$ is to be sub-linear, since if the proof π is just put on a tape of the Turing machine, access to the last bit requires looking at (traversing over) all bits before it and hence at the entire proof.

The idea behind **PCP**s is that for any $x \in L$, assuming that we are using the right proof format, the verifier will never reject the input. On the other hand, for any $\epsilon > 0$, a fixed number of runs of the verifier will reduce the probability of an erroneous acceptance of any input $x \notin L$ below ϵ. This number is given by $\left\lceil \log_{\frac{1}{2}} \epsilon \right\rceil$, but its value is not as important as the fact that it is fixed, regardless of the input. In the same vein, the constant $\frac{1}{2}$ is not important, any constant < 1 will do. The definition can be tweaked in other ways and remain equivalent, Arora [2] presents several ways.

3.2 Extending NP to PCP: An obvious connection

Having defined **PCP**s and verifiers, there is an immediate connection that can be made between the classic deterministic definition of **NP** and expressing it using **PCP**s, namely the following:

Theorem 3.1. ***NP*** $\subseteq$ ***PCP*** $(1, poly\,(n))$

Proof. Let L be any fixed language in **NP** and M its verifier as defined in the deterministic view of **NP**. Then there is a proof π_x for every $x \in L$ that is recognized by M. Additionally, every proof for $x \notin L$ is rejected by M. By definition, the total number of symbols in the proof is $O\left(n^c\right)$ for some constant c. Thus we can construct a **PCP** verifier V from M by having it read zero random bits, copy the proof onto its work tape, reposition back to the start position, and finally execute M. This verifier then accepts π_x for x with probability 1 and accepts any proof for $x \notin L$ with probability 0, clearly meeting the requirements for a **PCP** verifier. Since V can read no more than the total number of bits in the proof, V reads at most $O\left(n^c\right)$ bits. Thus L can be recognized by a $\left(O\left(1\right), O\left(n^c\right)\right)$ verifier. Since L is an arbitrary language in **NP**, **NP** $\subseteq$ **PCP** $(1, poly\,(n))$. □

This definition is not particularly useful, since it establishes the relation in only one direction. Additionally, it does not make any use of the probabilistic nature of **PCP**s, or any of the parts that were added to the Turing machine. Since the power of this new construct is not being used, it would be odd if there were results that come out of this characterization of **NP** that had not been found earlier using the classic definition. And there are no such results. Fortunately, there is another way to connect **PCP**s and **NP**, which does take into account the probabilistic backing of **PCP**s.

3.3 The PCP Theorem: NP $=$ PCP $(\log n, 1)$

A more useful way to connect **NP** and **PCP** is by proving the following theorem.

Theorem 3.2. ***NP*** $=$ ***PCP*** $(\log n, 1)$

This is a surprising result, since it says that any **NP** language has a verifier that can look at a constant number of bits in the proof, and still reject invalid proofs with a high probability, even for very large inputs. It seems very counterintuitive that this can be done. Naively, one would expect that for some inputs, the required proof must be quite large, and that by looking at only a constant number of bits, you would still have no clue rather to accept or reject with any degree of certainty.

The proof of this result is not simple, but it is also not horribly complicated. As stated by Arora [1], the proof does not involve anything significantly more complicated than understanding that

> *A non-zero univariate polynomial of degree d has at most d roots in a field.*

This is not a hard concept to understand, but its application to the proof is not simple. This explains why the main part of Arora's PhD thesis [1] spends 30 pages proving this theorem. Connecting together all of the details is no trivial task. For the purposes of this paper we will take **NP** $=$ **PCP** $(\log n, 1)$ as given and use this to show results about the approximation guarantees of any algorithm for some problems in **NP**.

4 Applications of the PCP Theorem

Having defined the classic definition of **NP** and the recent development of **PCP**, we are ready to move on to actually applying these concepts for the purpose of establishing some results about the approximability of certain **NP** problems. In this section, we will use the PCP theorem to show which of the empirical categories (as listed in 2.5) certain **NP** optimization problems fall within.

4.1 Establishing the Non-approximability of MAX-SNP problems

As a first application of the **PCP** theorem, we will show that a small group of problems in **NP** do not have polynomial time approximation schemes (PTASs). As mentioned previously, the optimization versions of most **NP** problems are unrelated. The class **MAX-SNP** is a small set of problems for which the optimization versions are related to a limited degree. In particular, if a given problem A in **MAX-SNP** is reducible to B in **MAX-SNP** and B has no polynomial time approximation algorithm with a performance ratio less than $1 + \epsilon$, then the A has no polynomial time approximation algorithm with a performance ratio less than $1+\phi(\epsilon)$. However, nothing is asserted about how ϕ maps values. The set of **MAX-SNP** problems is similar to **NP** in that there are **MAX-SNP**-complete problems to which all others **MAX-SNP** problems can be reduced. Because of these reductions, showing that there is one **MAX-SNP**-complete problem that does not have a PTAS shows that no **MAX-SNP**-complete problem can have a PTAS. In particular, MAX-3SAT as described earlier is a **MAX-SNP**-complete problem. We will show that MAX-3SAT does not have a PTAS and hence that neither does any **MAX-SNP**-complete problem. Hence all **MAX-SNP**-complete problems must fall into category iii of the outline given in section 2.5.

Theorem 4.1. *No **MAX-SNP**-complete problem has a PTAS, unless **P** $=$ **NP**.* [2]

Proof. The proof of this is a simple argument by contradiction. We assume initially that **P** $\neq$ **NP** and that there is a PTAS for MAX-3SAT. Then we argue that the existence of a PTAS implies that **P** $=$ **NP**. As mentioned in the introduction, **P** $\neq$ **NP** is still an open question, so this (as in much theory of computation) remains an assumption.

Take any arbitrary language L in **NP**. By the PCP theorem (theorem 3.2), there is a $(\log n, 1)$ verifier for L. Call this verifier V. Let Σ be the alphabet of symbols for L. Using standard notation, $\Sigma^\star$ is the set of all strings that contain only zero or more elements of Σ (the set of all strings that can be written with Σ).

For any string $x \in \Sigma^\star$, we will construct a 3SAT instance S_x with two properties.

i. For any $x \in L$, S_x will be satisfiable.

[2]This result is proved in Arora [1] Theorem 6.3, the presentation used here follows Hougardy [3] Theorem 3.10

ii. For any $x \notin L$, only a constant fraction δ of the clauses of S_x will be satisfiable.

Now if a PTAS existed for MAX-3SAT, there would be a polynomial time algorithm with a performance ratio $\zeta < \frac{1}{\delta}$. Let the number of clauses in S_x be k. For any $x \in L$, S_x is satisfiable, thus this algorithm must satisfy $OPT(x)/c_{sat}(x) \leq \zeta$, or equivalently, $c_{sat}(x) \geq \frac{k}{\zeta}$. Combining with the definition of ζ gives $c_{sat}(x) > k\delta$. On the other hand, for $x \notin L$, the algorithm can return no more than optimal – it satisfies $c_{unsat}(x) \leq OPT(x) = k\delta$. Since the ranges of c_{sat} and c_{unsat} are disjoint, L could be recognized in polynomial time. This contradicts the assumption that **P** $\neq$ **NP**, and thus there must not be a PTAS for MAX-3SAT.

To complete this proof, we need only fill in the details about how to create S_x. First, for any possible random string τ provided to the verifier, let S_τ be the boolean formula that expresses which proofs π are accepted by V on input x, encoding the bits by $1 = TRUE$ and $0 = FALSE$. Because V queries only a constant number of bits of the proof, each of the formulas S_τ is bounded above in size by a constant, regardless of the input x. Let t be the total number of formulas S_τ, of which there are no more than $2^{O(\log n)} = O(n^c)$ for some c (following earlier notation, n is the size of x).

Next, for each S_τ, let $\bar{S_\tau}$ be S_τ written as a 3SAT formula. As stated earlier, this can always be done. Let s be the maximum number of clauses that appear in any $\bar{S_\tau}$. The conversion from any boolean formula to 3SAT causes only a finite increase in the size of the formula. Combined with the fact that the size of each S_τ is bounded by a constant, s is also bounded by a constant regardless of input.

Finally, define S_x to be the conjunction of all $\bar{S_\tau}$. The size of S_x is bounded by $t \cdot s$, which combining the above, is $poly(n)$. Thus this encoding into a 3SAT formula is polynomial in time and space, as required for a reduction.

We now need to show that S_x has the requisite properties. For $x \in L$ this is easy. By the definition of the verifier V, there is a proof π_x such that $VACCEPTS(x, \pi_x)$ for any random string τ. This means that π_x is a satisfying assignment for each S_τ. Thus it is a satisfying assignment for S_x and hence S_x is satisfiable.

For $x \notin L$ the reasoning is harder, but not much. Since V accepts x at most half the time for any proof π, at most $\frac{1}{2}$ of the formulas $\bar{S_\tau}$ are simultaneously satisfiable. At least one clause in each unsatisfiable $\bar{S_\tau}$ must be unsatisfiable for this to be the case. Since there are at most $t \cdot s$ total clauses, this gives at most $t \cdot s - \frac{1}{2}t$ simultaneously satisfiable clauses in S_x. The ratio between these is $1 - \frac{1}{2}\frac{1}{s}$, which since s is a constant, is a constant fraction < 1, as desired. This completes the proof.

$\square$

A few comments on this result are in order. For MAX-3SAT, the best approximation algorithm (due to Karloff and Zwick [4]) achieves a performance ratio of $\frac{8}{7}$, and it is also known that this is the best achievable. As mentioned earlier, the reductions that allow other **MAX-SNP**-complete problems to be reduced to MAX-3SAT do not preserve this performance ratio in any nice way. To find the optimum performance ratio for a given problem in **MAX-SNP**, two techniques are available, neither of which is a mechanical process. One can either argue about the problem using some other method, or carefully study the reduction from the problem to a **MAX-SNP** problem with a known optimum performance ratio.

The use of the PCP theorem provides a clean, short proof of this theorem, but it leaves a lot of questions unanswered about the practical level to which each **MAX-SNP** problem can be approximated.

4.2 Clique Approximability

For a second application of the PCP Theorem, we will show that the MAX-CLIQUE problem has no polynomial time algorithm which achieves a constant performance ratio. Recall that this is the definition for category iv of the type outlines given in section 2.5.

We begin with a less stringent theorem, which by itself only places MAX-CLIQUE in category iii.

Theorem 4.2. *MAX-CLIQUE cannot be approximated in polynomial time with a performance ratio < 2, unless **P** $=$ **NP**.* [3]

Proof. As in the previous proof, we will show this by contradiction, first assuming that **P** $\neq$ **NP**, then show that a MAX-CLIQUE approximation algorithm with a performance ratio less than 2 implies that **P** $=$ **NP**.

Let L be an arbitrary language in **NP**. Let the notation $\omega(G)$ be the size of the maximum clique in a graph G. Then for any input x of length n, we will show how to construct a graph G_x such that for some function $f(n)$,

i. If $x \in L$, then $\omega(G_x) = f(n)$, and

ii. If $x \notin L$, then $\omega(G_x) < \frac{1}{2}f(n)$

[3]This proof follows Hougardy [3] Theorem 3.1

We begin by describing how to construct the graph G_x. Let V be the $(\log n, 1)$ PCP verifier for L. Let $q(n) = O\,(logn)$ be the maximum number of random bits used by V on inputs of size n. Let c (a constant) be the maximum number of bits of a proof queried by V. The vertex set of G_x consists of all accepting runs of V for the input x. Each of these can be described by a tuple of the random string queried by V and the set of values it then reads from the proof. Thus each vertex can be represented by a tuple $\langle \tau, \alpha_1, \alpha_2, \ldots, \alpha_c \rangle$, where τ is the random string used by V and α_i is the value of the ith bit the verifier queried in the proof. Since the length of the whole tuple is $2^{O(logn)} = O\,(n^c)$ for some c, the size of the graph G_x is polynomial in the size of x. Checking whether a given tuple is a member of the vertex set of G_x is very easy. Just run V with input x and random string τ, and answers to the queries of α_i. By the definition of how a verifier works, we do not even need the other bits of the proof, since the positions queried are fixed based on x and τ.

Define two vertices $\langle \tau, \alpha_1, \alpha_2, \ldots, \alpha_c \rangle$ and $\langle \bar{\tau}, \bar{\alpha_1}, \bar{\alpha_2}, \ldots, \bar{\alpha_c} \rangle$ to have an edge between them if there is some proof π that is consistent with both tuples. Consistent here means that the proof is accepted by the verifier for both random strings. Thus if any of the α_i and $\bar{\alpha_i}$ come from the same position in the proof, they must have the same value. Since the verifier can be run in polynomial time for each vertex pair, and the number of vertexes in the graph is polynomial in input size, the total task of constructing the graph is polynomial in input size. This means we have a polynomial time reduction from the input x to the graph G_x, as required for a reduction in **NP**.

Next, observe that G_x is a $2^{q(n)}$-partite graph. (A k-partite graph is one whose vertex set can be split into k sets, such that within each set no two vertices share an edge.) This is the case because no vertices whose tuple has the same τ can share an edge. Since there are $2^{q(n)}$ random strings τ, the graph must be $2^{q(n)}$-partite.

Now, for any fixed proof π, any two vertices that are consistent with π are by definition adjacent. So for all proofs π, we can conclude that

$$\omega(G_x) \geq \text{ number of random strings } \tau \text{ for which } V \text{ ACCEPTS } (x, \tau, \pi)$$

By definition, this is equal to $2^{q(n)} \cdot Prob_{q(n)}\,[V \text{ ACCEPTS } (x, \tau, \pi)]$.

From the other side of the issue, consider a clique C in G_x. All vertices in C which query a position p of a proof must get the same answer α. Thus there must be one proof π_0 that is consistent with all vertices of C. So we conclude that

$$\omega(G_x) \leq \text{ number of random strings } \tau \text{ for which } V \text{ ACCEPTS } (x, \tau, \pi)$$

Combining the two inequalities just derived, we get

$$\omega(G_x) = 2^{q(n)} \cdot \max_{\pi} Prob_{q(n)} [V \text{ ACCEPTS } (x, \tau, \pi)]$$

where the max is taken over all proofs π. Now for $x \in L$,

$$\max_{\pi} Prob_{q(n)} [V \text{ ACCEPTS } (x, \tau, \pi)] = 1$$

so $\omega(G_x) = 2^{q(n)}$. For $x \notin L$,

$$\max_{\pi} Prob_{q(n)} [V \text{ ACCEPTS } (x, \tau, \pi)] < \frac{1}{2}$$

so $\omega(G_x) = \frac{2^{q(n)}}{2}$. (The function $f(n)$ mentioned at the beginning of the proof is obviously $2^{q(n)}$). Thus if MAX-CLIQUE could be approximated to within a factor of 2, we could use that algorithm to recognize L in polynomial time. Since L can be any **NP**-complete language, this brings about a contradiction. As in the previous proof, we are forced to conclude that MAX-CLIQUE does not have a polynomial time approximation with performance ratio less than 2 (assuming as always that **P** $\neq$ **NP**). □

Corollary 4.3. *MAX-CLIQUE cannot be approximated in polynomial time for any constant performance ratio, unless* ***P*** $=$ ***NP****.*

Proof. The step from the last proof to this one is simple. In the definition of the class PCP given earlier, it was noted that the constant $\frac{1}{2}$ for $x \notin L$ was unimportant, any constant is just as valid. If instead of choosing $\frac{1}{2}$ we choose any ϵ, then in the above proof we can replace the constant factor that MAX-CLIQUE can not be approximated to by $\frac{1}{\epsilon}$. Since we can choose ϵ arbitrarily small, MAX-CLIQUE can not be approximated in polynomial time for any constant performance ratio. □

The difference between this problem and the one for theorem 4.1 is that here we have control over the magnitude of the factor that appears in front of $f(n)$ for $x \notin L$. While in theorem 4.1, we could only prove the existence of such a gap and not control its magnitude. This explains why the proofs (which in a general sense are the same) have such a different outcome.

5 Conclusion

The development of **PCP**s and the PCP Theorem has led to a substantial improvements in placing bounds on approximation algorithms, and even, although it was not shown in this paper, for inventing algorithms. However, the field of approximation algorithms is still rife with open questions, and for the invention of techniques that can be applied more generally. **PCP**s take a step in this direction, but they will not be the last word in this area.

References

[1] ARORA, S. *Probabilistic checking of proofs and hardness of approximation problems*. PhD thesis, UC Berkeley, 1994.

[2] ARORA, S. The approximability of NP-hard problems. In *STOC* (1998), pp. 337–348.

[3] HOUGARDY, S., PRMEL, H. J., AND STEGER, A. Probabilistically checkable proofs and their consequences for approximation algorithms. *Discrete Mathematics 136* (1994), 173–223.

[4] KARLOFF, H., AND ZWICK, U. A 7/8-approximation algorithm for MAX 3SAT? In *Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science, Miami Beach, FL, USA* (1997), IEEE Press.