Probabilistically Checkable Proofs and Hardness of Approximation

Rohan Bodke, Andrew Tung, Ekam Kaur

December 11, 2022

Abstract

We discuss probabilistically checkable proofs, along with the PCP theorem and its use in approximation algorithms.

1 Introduction

The discovery of NP-complete problems in the 1970s deemed fast algorithms to many problems essentially nonexistent. Because of this, many began looking for approximation algorithms, ones that are efficient and run in polynomial time, but only output approximate answers, rather than exact ones.

However, these approximation algorithms had varying degrees of success for different problems. In some cases arbitrarily tight approximation algorithms were found. However in others no constant factor approximation algorithms could be found! But until the 1990s, the problem of proving the lack of these approximation algorithms seemed out of reach. But suddenly, the idea of probabilistically checkable proofs and the PCP theorem led to the concept of "hardness of approximation", showing that even approximating problems cannot be done in polynomial time for many NP-complete problems.

In Section 2, we define probabilistically checkable proofs, give an illustrating example, and state the PCP theorem. Later, in Section 3, we prove the applicability of the PCP theorem to hardness of approximation. Finally, in Section 4, we highlight how the PCP theorem can be used to prove the impossibility of an approximation algorithm for a specific optimization problem.

2 Definitions and the PCP Theorem

A probabilistic Turing machine (PTM) is a Turing machine that on a step, can behave like a normal Turing machine, but also has the option to flip a (fair) coin to decide which of two options it does.

Definition 1. A language L is $(r(n), q(n))$-*probabilistically checkable* if there exists a PTM M that satisfies

- Completeness: if a string x is in L, then there exists a string π, known as *the proof*, such that M will, with certainty, accept x in polynomial time using $O(r(n))$ coin flips and $O(q(n))$ (nonadaptive) random access queries of π.
- Soundness: if a string x is not in L, no matter what the proof π is, M can use $O(r(n))$ coin flips and $O(q(n))$ random access queries and reject x in polynomial time with probability at least 1/2.

The PTM M can only access the proof π using these *random access queries*. A random access query works as follows: M does some computation, outputs a positive integer smaller than the length of π, and queries for the ith character of π. These queries are nonadaptive, meaning that they are all done without the information of any previous queries.

Note that the probability 1/2 in the soundness criterion can be improved to any probability less than 1 by repeating the process a large constant number of times. Also note that the length of π is at most exponential, otherwise it is impossible for the PTM to query for some of the bits of π in polynomial time.

Definition 2. The class of $(r(n), q(n))$-probabilistically checkable languages is denoted by $\mathsf{PCP}(r(n),\, q(n))$.

Example 1. Let A be an $n \times n$ square matrix. The *permanent* of A, denoted by $\operatorname{perm}(A)$, is defined as

$$\operatorname{perm}(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} a_{i,\sigma(i)},$$

where S_n is the symmetric group over $\{1, 2, \ldots, n\}$.

Consider the language $\textsc{Perm}$, which contains all tuples $\langle A, k\rangle$, where A is a 0/1 matrix, k is an integer, and $\operatorname{perm}(A) = k$. We prove that $\textsc{Perm} \in \mathsf{PCP}(\mathsf{poly}, \mathsf{poly})$. Let A_i be the $n-1 \times n-1$ matrix consisting of A but with the first row and ith column removed. Our algorithm uses the following easy-to-prove fact about permanents:

$$\operatorname{perm}(A) = \sum_{i=1}^{n} a_{1,i} \operatorname{perm}(A_i).$$

This means that we can calculate the permanent of an $n \times n$ matrix using the permanents of n $n-1 \times n-1$ matrices.

Let $n^{100} < p < n^{1000}$ be a fixed prime. For every a, b there is a polynomial of degree $n-1$ $P_{a,b}$ such that $P_{a,b}(i)$ is the a,bth element of A_i. Note that the values of P (and its coefficients) are taken modulo p to ensure a nearly constant computation time of a value of $P_{a,b}$. We can extend the values of A_i to $i > n$ by letting the a,bth element of A_i be $P_{a,b}(i)$. Now the permanent of A_i is an nth degree (multivariate) polynomial in its elements, and is thus an $n(n-1)$ degree univariate polynomial in i due to the definition of A_i for $i > n$.

The "proof" in our algorithm contains this $n(n-1)$ degree polynomial P. It can be represented in a polynomial amount of bits because each of its coefficients is less than p. To find the permanent of the original matrix A, we use the aforementioned summation and use the polynomial P to find each value. It appears that we are done, but this only satisfies the completeness criterion of PCP.

Now we suppose that the permanent of A isn't k. When this happens, the polynomial P received must be incorrect, because otherwise we would instantly reject after noticing that the sum isn't k. We have to prove that P is incorrect, but we can't directly calculate P in polynomial time! To do this, we flip $\log p = O(\log n)$ random coins and choose a random number k between 1 and p. Then we compute $P(k)$, and check recursively whether this is the permanent of A_k. We can do this since A_k is a smaller matrix than A. There is a small probability that P is wrong but $P(k)$ is still equal to $\text{perm}(A_k)$. This probability is essentially 0, because there is a unique polynomial of degree $n(n-1)$ passing through $n(n-1)+1 \ll p$ points. Even accounting for the recursive probabilities is not enough to add up to $1/2$, the necessary probability to falsify soundness.

Note that our proof must thus contain the polynomial P for not only A, but also every matrix that can be recursed in this fashion. The proof contains an exponential number of bits, but on any input, we only query for a polynomial number of them. Furthermore, obviously only a polynomial number of coin flips are used. This proves that $\textsc{Perm} \in \mathsf{PCP}(\mathsf{poly}, \mathsf{poly})$.

Now we go over a few fundamental equivalences, demonstrating the generality of PCP:

- $\mathsf{PCP}(\mathsf{log}, 0) = \mathsf{PCP}(0, \mathsf{log}) = \mathsf{P}$. The first is because a Turing machine can simulate a logarithmic number of coin flips in polynomial time. The second falls because without randomness we can just try out all possible values of the proof. There are essentially only a polynomial amount of such values, since the same (logarithmic number) of indices are queried no matter the proof. This statement really demonstrates the power of combining randomness with access queries!

- $\mathsf{PCP}(\mathsf{log}, \mathsf{poly}) = \mathsf{NP}$. Obviously every NP problem is in $\mathsf{PCP}(\mathsf{log}, \mathsf{poly})$. We show the converse holds as well. Given an input to the PCP verifier, there's only one proof given (since the proof is deterministic). This proof can be exponentially long, but only a polynomial number of bits have a nonzero probability of being queried, since there are only a logarithmic number of random coins. All of these bits will be part of our NP verifier. The NP verifier can now just do what the PCP one did, but go through all possible values of the random coin flips in polynomial time.

Above we proved that $\mathsf{NP} = \mathsf{PCP}(\mathsf{log}, \mathsf{poly})$, and one may think that that is the only representation of NP. However, there is in fact a much stronger representation, proved in the 1990s, known as the PCP theorem.

Theorem 1 (PCP theorem)**.** *$\mathsf{PCP}(\mathsf{log}, 1) = \mathsf{NP}$.*

The PCP theorem seems unbelievable. As previously mentioned, $\mathsf{PCP}(\mathsf{log}, 0) = \mathsf{P}$. By just increasing the query count to a constant, the class of problems drastically widen to NP. One might believe that a Turing machine can simulate these queries in polynomial time, but the random coin flips make the amount of queries essentially polynomial, rendering this impossible.

The PCP theorem is very difficult to prove, so we will not do so here.

3 Hardness of Approximation

Many optimization problems are known to be unsolvable in polynomial time, unless $\mathsf{P} = \mathsf{NP}$, for example because the corresponding decision problem is NP-complete. For these problems, the only hope is that there is an approximation algorithm that can provide very close answers, but not the precise ones.

Definition 3. Consider a maximization problem with answer $f(x)$ given input x. For $\rho < 1$, a *ρ-approximation algorithm* is an polynomial-time algorithm that always outputs a number between $\rho f(x)$ and $f(x)$. For a minimization problem, the algorithm always outputs a number between $f(x)$ and $f(x)/\rho$.

Example 2. Consider the problem Max-3Sat, where given an instance of 3Sat, the goal is to find the maximum fraction of satisfied clauses under any assignment. We provide a 1/2-approximation greedy algorithm for Max-3Sat. The first variable x is in a clause either as x or $\overline{x}$. If more clauses contain x then $\overline{x}$, let x be true, otherwise let x be false. Now delete all satisfied clauses and move to the next variable, and do the same thing. Clearly, at least half of all clauses are satisfied with this algorithm, which is obviously at least half of the optimum.

In fact, a 7/8-approximation algorithm also exists, but it is complicated and unrelated to the topic here.

The PCP theorem seems unrelated to approximation algorithms, but in fact it is one of the most useful theorems in this field! We use the PCP theorem to prove the following:

Theorem 2. *Consider a $\mathsf{NP} = \mathsf{PCP}(\log, 1)$ problem L. There exists a $\rho < 1$ for which there is a polynomial-time computable function f mapping binary strings to instances of* 3Sat *such that if a string is in L, then the corresponding formula is satisfiable, and if a string is not in L, then less than ρ of the clauses of the formula are satisfied under the same arrangement.*

Before we prove the theorem, let's dissect it. Suppose we had a ρ-approximation algorithm to Max-3Sat, where ρ is the same as in the theorem. We can use the algorithm to solve L in polynomial time: convert the input into a 3Sat formula, and use the approximation algorithm to get a number between 0 and 1. If the input is in L, then the 3Sat formula is satisfiable and the approximation algorithm outputs a number at least ρ. Otherwise the approximation algorithm outputs a number less than ρ. We can thus use the number to solve L.

But this is impossible unless $\mathsf{P} = \mathsf{NP}$! Therefore there cannot be a ρ-approximation algorithm to Max-3Sat for some $\rho < 1$ (unless $\mathsf{P} = \mathsf{NP}$).

Now we prove the theorem.

Proof. There must exist an algorithm for L that uses a logarithmic number of random coins and a constant number q of queries. All of the following is done assuming a fixed input x. Our job is to convert x to a 3Sat formula that satisfies the above conditions.

Given an input x, our algorithm flips some random coins. There are a polynomial amount of possible options. Let's say r is one such set of flip values. Now given x and r, the algorithm queries for q indices $i_1, i_2, \ldots, i_q$, and receives q bits $a_1, a_2, \ldots, a_q$. These q bits are what determine whether the algorithm accepts or rejects.

Therefore there exists a function from $\{0,1\}^q \to \{0,1\}$ where the domain is the bits $a_1, a_2, \ldots, a_q$ and the output is whether our algorithm accepts or rejects. We can convert this function into an instance of 3Sat. The number of clauses is only in terms of q, so it's constant. We combine all of these formulas from all values of r into one polynomial-sized formula.

If x is in L, then there exists a proof (i.e. the values of $a_1, a_2, \ldots, a_q$) such that the algorithm accepts, no matter the value of r. Therefore the whole formula is satisfiable, since every part is satisfiable. One may think that it is possible for the satisfying assignment to be different in each part, but this is impossible because the proof is constant no matter the value of r.

if x is not in L, then no matter the proof, at least half the values of r (by soundness) yield an unsatisfiable formula, where at least one of the clauses is not satisfiable. Therefore at most $rc - r/2$ out of the rc clauses are satisfied, where c is the length of the formula for a fixed value of r, which is only in terms of the constant q. Therefore if we let $\rho = 1 - 1/2c + \varepsilon < 1$, we get the desired. □

4 Approximating Max-IndSet

Now we look at the problem Max-IndSet, where given a graph, we have to find the largest independent set in the graph. After IndSet was proven to be NP-complete, many tried to find approximation algorithms to Max-IndSet for years. However none were found. It wasn't until after the PCP theorem was proved that the impossibility of such algorithms was concluded.

Theorem 3. *For* any $\rho < 1$*, there is no ρ-approximation algorithm for* Max-IndSet *(unless* $\mathsf{P} = \mathsf{NP}$*).*

Proof. We first prove that there exists such a ρ. We reduce Max-3Sat to Max-IndSet, which is sufficient. Given a 3Sat formula, create a graph where each literal is a vertex (so each clause contains 3 vertices). If any two literals are in the same clause, draw an edge between those two vertices. Furthermore, draw an edge between any two opposite literals, like x and $\overline{x}$. The maximum independent set of this graph is equal to the maximum number of clauses satisfied. Consider an independent set in the graph. Each vertex (or literal) in the independent set is set to true. This is possible because there are no opposite literals, as each such pair has an edge. Furthermore, there is only literal per clause because any two literals in the same clause have an edge. The converse also easily holds.

Now we use this fact to prove that there is no such ρ. Given a graph G and a positive integer k, consider the graph G^k, that has $\binom{|G|}{k}$ vertices, representing every subset of k vertices of G. Two vertices of G^k share an edge if and only if the union of the two subsets in G do *not* form an independent set. Let c be the size of the largest independent set in G. The largest independent set in G^k is obviously the $\binom{c}{k}$ vertices consisting of all subsets of k vertices that are only in the largest independent set in G. Now if there was a ρ^k-approximation algorithm for Max-IndSet, running it on G^k would find output at least $\rho^k\binom{c}{k}$. We reverse engineer this number to approximate c. Indeed, we get a number greater than ρc! But this means that there must be a ρ-approximation algorithm for G. The same idea works for any graph, so there is a ρ-approximation algorithm for MAX-INDSET, contradiciton. Therefore, there can't be a ρ^k-approximation algorithm for any k. Letting k be arbitrarily large we get the desired. $\square$

References

[1] Sanjeev Arora and Boaz Barak. *Compuational Complexity: A Modern Approach*. 2009. Chap. 11. ISBN: 978-0521424264.

[2] Sanjeev Arora and Shmuel Safra. *Probabilistic Checking of Proofs: A New Characterization of NP*. 1998. DOI: `https://doi.org/10.1145/273865.273901`.

[3] Sanjeev Arora et al. *Proof Verification and Hardness of Approximation Problems*. May 1998. DOI: `https://doi.org/10.1145/278298.278306`.

[4] Dorit S. Hochbaum. *Approximation Algorithms for NP-Hard Problems*. July 1996. Chap. 10.

[5] Lucas Trevisan. *Inapproximability of Combinatorial Optimzation Problems*. July 24, 2004. DOI: `https://doi.org/10.48550/arXiv.cs/0409043`.