AN INTRODUCTION TO THE PCP THEOREM

MIN JAE SONG

Abstract. In this paper, we will first define basic concepts in computational complexity. Then, we will introduce the PCP Theorem and its equivalent formulation from the hardness of approximation view. Finally, we will prove a weaker version of the PCP theorem.

Contents

1. Introduction

In this section, we define the basic concepts in computational complexity.

1.1. **The Turing Machine.** When we are given a certain computation problem, we first read the problem from the *problem sheet*, work on our *scratch paper* for intermediate calculations, and then write down the final solution. The Turing machine is a concrete embodiment of this common, yet abstract notion of "computation". We give the Turing machine a computation problem by writing it down on the input tape so that the machine can read it. Then, the machine performs the intermediate calculations on the work tapes according to its built-in rules. Finally,

the machine outputs the final solution on its output tape after performing all the necessary calculations.

The structure of the Turing machine is very simple. It consists of:

(1) *k-tapes*, which are unbounded in length and contain symbols that the machine can read. We have a single *input* tape, which corresponds to the *problem sheet* and $k-1$ *work* tapes, which correspond to the *scratch papers* mentioned above. The last one of the work tapes is designated as the *output* tape.
(2) A *head* for each of the k tapes that can read and write symbols on the tape and move the tape left and right exactly one cell at a time.
(3) A *register* which keeps track of the state the machine is currently in,
(4) A finite table of instructions (called the *transition function*) that, given the machine's current state and its readings on the tapes, performs the next computational step.

The finite set of *states* of the Turing machine is denoted Q. As mentioned above, the machine contains a "register" that can hold a single element of Q; this is the "state" of the machine at that instant. This state determines its action at the next computational step, which consists of the following:

(1) read the symbols in the cells directly under the k heads
(2) for the $k-1$ work tapes, replace each symbol with a new symbol (it has the option of not changing the tape by writing down the old symbol again)
(3) change its register to contain another state from the finite set Q (it has the option not to change its state by choosing the old state again)
(4) move each head one cell to the left or to the right (or to stay in place)

Given the information above, we can formally define the Turing machine:

Definition 1.1 (Turing Machine)**.** a *Turing machine* (TM) M is described by a tuple (Γ, Q, δ) containing:

- A finite set Γ of the symbols that M's tapes can contain. We assume that Γ contains a designated "blank" symbol, denoted $\sqcup$; a designated "start" symbol, denoted $\triangleright$; and the numbers 0 and 1. We call Γ the *alphabet* of M.
- A finite set Q of possible states M's register can be in. We assume that Q contains a designated start state, denoted q_{start}, and a designated halting state, denoted q_{halt}.
- A function $\delta : Q \times \Gamma^k \to Q \times \Gamma^{k-1} \times \{L, S, R\}^k$, where $k \geq 2$, describing the rules M use in performing each step. This function is called the *transition function* of M. L, S, R stands for Left, Stay, and, Right, respectively.

If the machine is in state $q \in Q$ and $(\sigma_1, \sigma_2, ..., \sigma_k)$ are the symbols currently being read in the k tapes, and $\delta(q, (\sigma_1, \sigma_2, ..., \sigma_k)) = (q', (\sigma_2', ...\sigma_k'), z)$ where $z \in \{L, S, R\}^k$, then at the next step the σ symbols in the last $k-1$ tapes will be replaced by the σ' symbols, the machine will be in state q' and the k heads will move as given by z. (If the machine tries to move left from the leftmost position of a tape then it will stay in place.)

All tapes except for the input tape initially have the *start* symbol $\triangleright$ in their first location and in all other locations the *blank* symbol $\sqcup$. The input tape initially has the start symbol $\triangleright$, followed by a finite nonblank string x ("the input"), and the blank symbol $\sqcup$ on the rest of its cells. All heads start at the left ends of the tapes and the machine is in the special starting state q_{start}. This configuration is called

the *start configuration* of M on input x. Each step of the computation is performed by applying the function δ as described previously. The special halting state q_{halt} has the property that once the machine is in q_{halt}, the transition function δ does not allow it to further modify the tape or change states. Obviously, if the machine enters q_{halt}, then it has *halted*. Throughout this paper, we will be only interested in machines that halt for every input in a finite number of steps.

Definition 1.2. Let $f : \{0,1\}^* \to \{0,1\}^*$ and let $T : \mathbb{N} \to \mathbb{N}$ be some functions, and let M be a Turing machine. We say that M *computes* f if for every $x \in \{0,1\}^*$, whenever M is starts with input x, then it halts with $f(x)$ written on its output tape. We say M *computes* f *in* $T(n)$*-time* if its computation on every input x requires at most $T(|x|)$ steps. Moreover, if T can be bounded above by a polynomial, we say M computes f in *polynomial time*.

Nondeterministic Turing Machine. If we let a machine have *two* transition functions δ_0 and δ_1, then the machine becomes *nondeterministic* since it may chose either δ_0 or δ_1 for each computational step. Using this observation, we can define a new type of machine, called the *nondeterministic* Turing machine (NDTM).

An NDTM has two transition functions δ_0 and δ_1, and a special halting state denoted by q_{accept}. When an NDTM M computes a function, we envision that at each computational step M makes a random choice between δ_0 and δ_1 and applies the chosen transition function. For every input x, we say that $M(x) = 1$ if there *exists* some sequence of these choices (which we call the *nondeterministic choices* of M) that would make M reach q_{accept} on input x. Otherwise - if *every* sequence of choices makes M halt without reaching q_{accept} - then we say that $M(x) = 0$. We say that M runs in $T(n)$ time if for every input $x \in \{0,1\}^*$ and every sequence of nondeterministic choices, M halts within $T(|x|)$ steps.

1.2. **Classes P and NP.**

Definition 1.3. A *language* is any subset of $\{0,1\}^*$. We say a Turing machine M *decides* a language $L \subseteq \{0,1\}^*$ if it computes the function $f_L : \{0.1\}^* \to \{0,1\}$, where $f_L(x) = 1 \Leftrightarrow x \in L$.

Definition 1.4 (The class DTIME)**.** Let $T : \mathbb{N} \to \mathbb{N}$ be some function. A language L is in DTIME($T(n)$) iff there is a TM that runs in time $c \cdot T(n)$ for some constant $c > 0$ and decides L.

Definition 1.5 (The class NTIME)**.** For every function $T : \mathbb{N} \to \mathbb{N}$ and $L \subseteq \{0,1\}^*$, we say that $L \in$ NTIME($T(n)$) if there is a constant $c > 0$ and a $c \cdot T(n)$-time NDTM M such that for every $x \in \{0,1\}^*, x \in L \Leftrightarrow M(x) = 1$.

Definition 1.6 (Class **P**)**.** $\mathbf{P} = \cup_{c \in \mathbb{N}}$DTIME($n^c$)

Definition 1.7 (Class **NP**)**.** $\mathbf{NP} = \cup_{c \in \mathbb{N}}$NTIME($n^c$)

P stands for *polynomial time* and NP stands for *nondeterministic polynomial time*. Languages in P have TMs that solve its membership problem in polynomial time and languages in NP have NDTMs that solve its membership problem in polynomial time. The crucial difference between P and NP lies in determinism and nondeterminism. Nondeterminism seems to give more computational power to the Turing Machine, enabling it to efficiently solve problems that avoid being solved by the standard Turing Machine. It is trivial that P $\subseteq$ NP since if we set $\delta_0 = \delta_1$, then the NDTM becomes deterministic.

A somewhat easier way of describing P and NP is that P is the set of problems whose solutions can be *found* in polynomial time, while NP is the set of problems whose solutions can be *checked* in polynomial time. The famous "P vs NP" problem asks, "if a problem's solution can be checked in polynomial time, then can it also be found in polynomial time?". This important problem still remains open. However, it is generally considered that P $\neq$ NP.

Example 1.8 (Sorting Problem)**.** Given a set S of n integers, sort S. This problem is in P since we can compare all the integers in the set with each other, which can be done in $\binom{n}{2}$ steps. Hence, this takes $O(n^2)$ time.

Example 1.9 (SAT)**.** A *boolean formula* over variables $u_1, ..., u_n$ consists of the variables and the logical operators AND ($\wedge$), OR ($\vee$), and NOT ($\neg$). For example, $(u_1 \vee u_2) \wedge u_3$ is a boolean formula. Given a boolean formula φ, determine whether it is satisfiable. This problem is in NP since given a truth assignment, we can check whether φ is satisfied or not in polynomial time.

1.3. **NP and proof verifying system.** Languages in NP can be equivalently defined as languages with efficiently verifiable proof systems.

Definition 1.10 (efficiently verifiable proof systems)**.** A language $L \subseteq \{0,1\}^*$ has an efficiently verifiable proof system if there exists a polynomial $p : \mathbb{N} \to \mathbb{N}$ and a polynomial-time deterministic Turing Machine V such that, given an input x, verifies proofs, denoted π. The following properties hold:

$$x \in L \Rightarrow \exists \pi \in \{0,1\}^{p(|x|)} : V^{\pi}(x) = 1$$
$$x \notin L \Rightarrow \forall \pi \in \{0,1\}^{p(|x|)} : V^{\pi}(x) = 0$$

$V^{\pi}(x)$ has access to an input string x and a proof string π. If $x \in L$, and $\pi \in \{0,1\}^{p(|x|)}$ satisfy $V^{\pi}(x) = 1$, then we call π a *certificate* or a *correct proof*. We denote by NP the class of languages that have efficiently verifiable proof systems.

Equivalence of the two definitions of NP can be understood as the following: Suppose L is decided by a non-deterministic Turing Machine N that runs in polynomial time. For every $x \in L$, there is a sequence of nondeterministic choices that makes N accept the input x. We can use this sequence as a *certificate* or a *correct proof* for x. This certificate has polynomially-bounded length and can be verified in polynomial time by a deterministic machine, which simulates N using these nondeterministic choices and verifies that it would have accepted x after using these nondeterministic choices. Hence, $L \in$ NP according to Definition 1.7.

Conversely, if $L \in$ NP according to Definition 1.7, then we can describe a polynomial time nondeterministic Turing Machine N that decides L. Our machine, N can "guess" the proof in polynomial time by enumerating all possibilities of the certificate π and simulate the polynomial time verifier V in the end. If the verifier accepts, then N accepts. It follows that N accepts input x if and only if a valid certificate exists for x. Hence, the nondeterministic Turing Machine N decides L in polynomial time.

1.4. **NP completeness.** When we encounter problems, we often feel that there are problems that are "harder" than the others. Similarly, we would like to know if there are problems in NP that are harder than the other ones in NP. It turns out that polynomial-time reduction provides a way to see which problems are "harder" than the others!

Definition 1.11 (Reductions, NP-hardness and NP-completeness)**.** A language $L \subseteq \{0,1\}^*$ is *polynomial-time Karp reducible* to a language $L' \subseteq \{0.1\}^*$, denoted by $L \leq_p L'$, if there is a polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that for every $x \in \{0,1\}^*, x \in L$ if and only if $f(x) \in L'$

We say that L' is NP-*hard* if $L \leq_p L'$ for every $L \in$ NP. We say that L' is NP-*complete* if L' is NP-hard and $L' \in$ NP.

It is remarkable that such NP-complete problems do exist.

Definition 1.12. A boolean formula over variables $u_1, ...u_n$ is in *CNF form* (*Conjunctive Normal Form*) if it is an AND of OR's of variables or their negations i.e. it has the form

$$\bigwedge_i (\bigvee_j v_{i_j})$$

where each v_{i_j} is either a variable u_k or its negation $\bar{u}_k$. The terms v_{i_j} are called the *literals* of the formula and the terms $(\vee_j v_{i_j})$ are called its *clauses*. A kCNF is a CNF formula in which all clauses contain at most k literals. We denote by SAT the language of all satisfiable CNF formulae and by 3SAT the language of all satisfiable 3CNF formulae.

Lemma 1.13 (Cook-Levin Reduction)**.** *SAT* $\leq_p$ *3SAT*

Proof. We give a transformation that maps a CNF fomrula φ to a 3CNF formula ψ such that φ is satisfiable if and only if ψ is satisfiable. We first begin by transforming 4CNF formula into a 3CNF formula. Let $C = x_1 \vee x_2 \vee x_3 \vee x_4$ be a clause of φ. We introduce a dummy variable y and split up C into two clauses, $C_1 = x_1 \vee x_2 \vee y$ and $C_2 = x_3 \vee x_4 \vee \bar{y}$. C is satisfiable if and only if C_1 and C_2 are both satisfiable. We can apply this transformation to each of the clauses in φ. This mapping gives us a polytime transformation from 4CNF to 3CNF.

The general case follows from induction. Suppose every $(k-1)$CNF formula can be transformed into an equivalent 3CNF formula in polynomial time. We claim that every kCNF formula can be transformed into an equivalent 3CNF formula in polynomial time as well. Let C be a clause of size k. By applying the above transformation, we introduce a dummy variable y and split C into a pair of clauses C_1 of size $k-1$ and C_2 of size 3 that depend on the k variables of C and an additional dummy variable y. Applying this transformation repeatedly to clauses of size k yields a polynomial-time transformation of a kCNF formula φ into an equivalent $(k-1)$CNF formula φ'. Then, it follows from the induction hypothesis that φ' can be reduced to an equivalent 3CNF formula ψ in polynomial time.

□

Theorem 1.14 (Cook-Levin)**.** *SAT is NP-complete*

For full proof of this theorem, refer to ref[2]. The proof basically uses the locality of Turing machines, which means that each computation step of a Turing machine reads and changes only a few bits of the machine's tapes. As a corollary, we have that 3SAT is NP-complete, since lemma 1.12 shows that SAT is Karp reducible to 3SAT. Other than SAT, there are hundreds of NP-complete problems that are known today. Here are some examples:

- Traveling salesman problem (TSP)
- Maximum cut (MAX-CUT)

- Circuit satisfiability (CKT-SAT)
- Hamiltonian path in a directed graph (dHAMPATH)

We can say that NP-complete problems are the hardest problems in NP since if we find an efficient algorithm for *one* NP-complete problem, say, the traveling salesman problem (TSP), then we can solve all NP problems efficiently by reducing them to TSP and solving it using the algorithm. This can be formally stated as the following theorem:

Theorem 1.15. *If L is NP-complete, then $L \in P$ if and only if $P = NP$*

2. The PCP Theorem

While reading every bit of a proof guarentees that the verifier does not err, one may feel that such a meticulous process is more than necessary. If we allow for a small margin of error, can't we get away with reading only parts of a proof? Surprisingly, the answer to this question is, "Yes". In fact, one view of the PCP Theorem indicates that we can provide a new proof system which is verifiable by reading a constant number of bits of the proof. This proof system provides us with *Probabilistically Checkable Proofs*, hence the acronym "PCP".

2.1. The PCP Theorem.

The class PCP is a generalization of the proof verifying system used to define NP, with the following changes. First, the verifier is *probabilistic* instead of deterministic. Hence, the verifier can have different outputs for the same inputs x and π. Second, the verifier has *random* access to the proof string π. This means each bit in the proof string can be independently *queried* by the verifier via a special *address tape*: If the verifier desires say the ith bit in the proof of the string, it writes i in base-2 on the address tape and then receives the bit $\pi[i]$. Note that since the address size is *logarithmic* in the proof size, this model in principle allows a polynomial-time verifier to check exponentially sized proofs.

Verifiers can be *adaptive* or *nonadaptive*. A nonadpative verifier selects its queries based only on its input and random tape. In other words, no query depends on responses to prior queries. Adaptive queries, on the other hand, depend on bits it has already queried on π to select its next queries. For the purpose of this paper, we will restrict the verifiers to be nonadaptive.

Definition 2.1 (PCP verifier)**.** Let L be a language and $q, r : \mathbb{N} \to \mathbb{N}$. We say that L has an $(r(n), q(n))$-PCP *verifier* if there is a polynomial-time probabilistic algorithm V satisfying:

- *Efficiency*: On input a string $x \in \{0,1\}^n$ and given random access to a string $\pi \in \{0,1\}^*$ of length at most $q(n)2^{r(n)}$ (which we call the *proof*), V uses at most $r(n)$ random coins and makes at most $q(n)$ nonadaptive queries to locations of π. Then it outputs "1" (for "accept") or "0" (for "reject"). We let $V^\pi(x)$ denote the random variable representing V's output on input x and with random access to π.
- *Completeness*: If $x \in L$, then there exists a proof $\pi \in \{0,1\}^*$ such that $\Pr[V^\pi(x) = 1] = 1$. We call this string π the *correct proof* or the *certificate* for x.
- *Soundness*: If $x \notin L$, then for every proof $\pi \in \{0,1\}^*$, $\Pr[V^\pi(x) = 1] \leq 1/2$.

We say that a language L is in $\mathrm{PCP}(r(n), q(n))$ if there are some constants $c, d > 0$ such that L has a $(c \cdot r(n), d \cdot q(n))$-PCP verifier.

Hence, a PCP verifier checks a proof probabilistically by querying $q(n)$ bits of the proof string in locations determined by the $r(n)$ random coin tosses. The constant $1/2$ in the soundness condition is arbitrary, in the sense that changing it to any other positive constant smaller than 1 will not change the class of languages defined. This is because we can execute the verifier multiple times to make the constant as small as we want. For instance, if we run a PCP verifier with soundness $1/2$ that uses r coins and makes q queries c times, it can be seen as a PCP verifier with soundness 2^{-c} that uses cr coins and makes cq queries.

Now that we know what a PCP verifier is, we are in a position to state the PCP Theorem.

Theorem 2.2 (**PCP** Theorem)**.** $\mathbf{NP} = \mathbf{PCP}(\log n, 1)$

Hence the PCP Theorem gives us a new characterization of the class NP. Namely, the class NP is the set of languages which have a $(c \cdot \log n, d)$-PCP verifier. This means that every NP-language has a PCP verifier that verifies proofs of at most $poly(n)$ bits by reading a constant number of bits.

Note that $\mathrm{PCP}(r(n), q(n)) \subseteq \mathrm{NTIME}(2^{O(r(n))}q(n))$ since a nondeterministic machine could "guess" the proof in $2^{O(r(n))}q(n)$ time and verify it deterministically by running the verifier for all $2^{O(r(n))}$ possible outcomes of its random coin tosses. If the verifier accepts for all these possible coin tosses, then NDTM accepts. Hence, as a special case, we have $\mathrm{PCP}(\log n, 1) \subseteq \mathrm{NTIME}(2^{O(\log n)}) = \mathrm{NP}$. This is the trivial direction of the PCP Theorem.

2.2. **Hardness of Approximation View.** The PCP Theorem can be equivalently stated in a seemingly different way. It states that computing approximate solutions to **NP** optimization problems is no easier than computing exact solutions. For concreteness, we focus on MAX-3SAT. MAX-3SAT is a problem of finding, given a 3CNF boolean formula φ as input, an assignment that maximizes the number of satisfied clauses. We first define what an approximation algorithm for MAX-3SAT is.

Definition 2.3 (Approximation of MAX-3SAT)**.** For every 3CNF formula φ, the *value* of φ, denoted by $\mathrm{val}(\varphi)$, is the maximum fraction of clauses that can be satisfied by any assignment to φ's variables. In particular, φ is satisfiable if and only if $\mathrm{val}(\varphi) = 1$.

For every $\rho \leq 1$, an algorithm A is a *ρ-approximation* algorithm for MAX-3SAT if for every 3CNF formula φ with m clauses, $A(\varphi)$ outputs an assignment satisfying at least $\rho \cdot \mathrm{val}(\varphi)m$ of φ's clauses.

Theorem 2.4 (**PCP** Theorem: Hardness of Approximation View)**.** *There exists $\rho < 1$ such that for every $L \in NP$ there is a polynomial-time function f mapping strings to (representations of) 3CNF formulas such that*

$$x \in L \Rightarrow val(f(x)) = 1 \tag{2.5}$$

$$x \notin L \Rightarrow val(f(x)) < \rho \tag{2.6}$$

Theorem 2.3 shows that for every $L \in \mathrm{NP}$, we can covert a ρ-approximation algorithm A for MAX-3SAT into an algorithm deciding L, since (2.4) and (2.5) imply that $x \in L$ iff $A(f(x))$ gives an assignment satisfying at least a ρ fraction of the clauses of $f(x)$. This immediately implies the following corollary:

Corollary 2.7. *There exists some constant $\rho < 1$ such that if there is a polynomial-time ρ-approximation algorithm for MAX-3SAT, then $P = NP$.*

2.3. **Equivalence of the Two Views.** At first glance, the equivalence of the two views is not immediate. One view shows that the PCP theorem gives a way to transform every mathematical proof into a form that is checkable by only looking at constant bits. The other view shows that computing approximate solutions to NP optimization problems is as hard as computing the exact solution. To show equivalence of these views, we first define constraint satisfaction problems.

Definition 2.8 (Constraint Satisfaction Problems (CSP))**.** If q is a natural number, then a qCSP instance φ is a collection of functions $\varphi_1, ..., \varphi_m$ (called *constraints*) from $\{0,1\}^n$ to $\{0,1\}$ such that each function φ_i depends on at most q of its input locations. That is, for every $i \in [m]$ there exist $j_1, ...j_q \in [n]$ and $f : \{0,1\}^q \to \{0,1\}$ such that $\varphi_i(\mathbf{u}) = f(u_{j_1}, ...u_{j_q})$ for every $\mathbf{u} \in \{0,1\}^n$.

We say that an *assignment* $\mathbf{u} \in \{0,1\}^n$ *satisfies* constraint $\varphi_i(\mathbf{u}) = 1$. The fraction of the constraints satisfied by $\mathbf{u}$ is $\frac{\sum_{i=1}^m \varphi_i(\mathbf{u})}{m}$, and we let $\text{val}(\varphi)$ denote the maximum of this value over all $\mathbf{u} \in \{0,1\}^n$. We say that φ is *satisfiable* if $\text{val}(\varphi) = 1$. We call q the *arity* of φ.

We will show that the two views are equivalent by proving that they are both equivalent to the hardness of a certain gap version of qCSP.

Definition 2.9 (Gap CSP)**.** for every $q \in \mathbb{N}, \rho \leq 1$, define ρ-GAPqCSP to be the problem of determining for a given qCSP-instance φ whether

(1) $\text{val}(\varphi) = 1$ (in which case we say φ is a YES instance of ρ-GAPqCSP) or
(2) $\text{val}(\varphi) < \rho$ (in which we say φ is a NO instance of ρ-GAPqCSP).

We say that ρ-GAPqCSP is NP-hard if for every language L in NP there is a polynomial-time function f mapping strings to (representations of) qCSP instances satisfying:

- *Completeness*: $x \in L \Rightarrow \text{val}(f(x)) = 1$
- *Soundness*: $x \notin L \Rightarrow \text{val}(f(x)) < \rho$

To elaborate, suppose the function f mentioned above exists and let Y denote the set of strings that represents the YES instances of ρ-GAPqCSP and N the NO instances. Then the following holds:

- $x \in L \Rightarrow f(x) \in Y$
- $x \notin L \Rightarrow f(x) \in N$

It follows that $x \in L \Leftrightarrow f(x) \in Y$. Since $f(x)$ is polynomial-time and L is *any* language in NP, $L \leq_p Y$ for all $L \in$ NP. Hence, ρ-GAPqCSP is NP-hard.

Theorem 2.10. *There exist constants $q \in \mathbb{N}, \rho \in (0,1)$ such that ρ-GAPqCSP is* **NP***-hard.*

Theorem 2.11. *Theorem 2.2 implies Theorem 2.10*

Proof. Assume NP $\subseteq$ PCP $(\log n, 1)$. Then every language $L \in$ NP has a PCP verifier with $c \log n$ coins and q query bits. Here, the value of q may differ depending on the NP language. However, since every NP language is Karp reducible to SAT, all these numbers can be upper bounded by a universal constant, namely, the number of query bits required by the verifier for SAT. We show that 1/2-GAPqCSP is NP-hard for some q. Let x be the input of the verifier and r an outcome of a random

coin toss. Define $V_{x,r}(\pi)$=1 if the PCP verifier $V^{\pi}(x)$=1 for the coin toss r. Note that given x and r, $V_{x,r}(\pi)$ depends on q bits of the proof π. Hence, $V_{x,r}(\pi)$ can be thought of as a constraint in an instance of qCSP. Then, $\{V_{x,r}\}_{r=c\log n}$ can be seen as an instance of qCSP. Given the completeness and soundness conditions of the PCP verifier, completeness and soundness of the Gap CSP follow easily. □

Theorem 2.12. *Theorem 2.10 implies Theorem 2.2*

Proof. Assume that there exist ρ and q such that ρ-GAPqCSP is NP Hard. Then, any language $L \in$ NP can be represented as an instance of ρ-GAPqCSP. Let ϕ be an instance of qCSP and $\{\phi_i\}_{i=1}^{m}$ the set of its constraints. The verifier will verify the proof π, which is the truth assignment of this instance, by randomly choosing an $i \in [m]$ and checking that ϕ_i is satisfied (by making q queries). If $x \in L$, then the verifier will accept with probability 1 while if $x \notin L$, it will accept with probability at most ρ. The soundness can be boosted at the expense of a constant factor in the randomness and number of queries. □

Theorem 2.13. *Theorem 2.4 is equivalent to Theorem 2.9*

Since 3CNF formulas are special instances of 3CSP, Theorem 2.3 implies Theorem 2.9 so we only need to show the other direction. Before starting, we first prove a useful lemma.

Lemma 2.14. *Every n-variable boolean function can be expressed as a conjunction of OR's and AND's.*

Proof. Let $f(x)$ be the given boolean function. we can construct a clause C_v consisting of OR's such that $C_v(v) = 0$ and $C_v(w) = 1$ when $v \neq w$. Take the AND of all C_v's of v's such that $f(v) = 0$ i.e.

$$\varphi = \bigwedge_{v:f(v)=0} C_v(z_1, z_2, ..., z_n) \tag{2.15}$$

Note that φ has size at most $n2^n$. For every u such that $f(u) = 0$, it holds that $C_u(u) = 0$ and hence $\varphi(u)$ is also equal to 0. On the other hand, if $f(u) = 1$, then $C_v(u) = 1$ for every v such that $f(v) = 0$ and hence $\varphi(u) = 1$. Therefore, $\varphi(u) = f(u)$ for every u. □

Proof of Theorem 2.12. Note that the CNF formula we constructed in the proof consists of at most 2^q clauses. Hence any constraint in an instance of qCSP can be expressed as the AND's of at most 2^q clauses, each clause containing only OR's and depending on at most q variables. Then, we can use the Cook-Levin Reduction presented earlier to reduce these into 3CNF formulas. If a qCSP instance is satisfiable, it immediately follows that its 3CNF representation is satisfiable so we have the completeness condition. if at least ϵ of the constraints are unsatisfiable, then at least $\frac{\epsilon}{q2^q}$ of the clauses in 3CNF is unsatisfiable so we have the soundness condition as well. □

3. A Weak PCP Theorem

The full proof of the PCP Theorem is beyond the scope of this paper. Instead, we will prove a weaker version of the theorem which still captures some of the essential techniques used in the full proof.

Theorem 3.1 (Exponential-sized PCP system for NP)**.** **NP** $\subseteq$ **PCP**(poly(n),1)

The theorem states that every NP statement has an exponentially long proof that can be locally tested by looking at a constant number of bits. This is weaker than the PCP theorem stated in the previous section since the proof it validates may be much larger. In the original theorem, the PCP verifier verifies proofs of polynomial size whereas in this weaker theorem, the verifier verifies proofs of exponential size. Still, it is interesting that exponentially sized proofs can be verified by a constant number of queries.

We will prove this theorem by constructing an appropriate PCP verifier for an NP-complete language. Since all languages in NP can be polynomially reduced to an NP-complete language, constructing a PCP verifier for *one* NP-complete language will suffice.

3.1. **Linear Functions.** We first define linear functions over GF(2) as we will depend on them throughout this study.

Definition 3.2. A function $f : \{0,1\}^n \to \{0,1\}$ is *linear* if, for every $x, y \in \{0,1\}^n$, $f(x+y) = f(x) + f(y)$, and $f(\alpha x) = \alpha f(x)$.

It is also convenient to think about bit strings as vectors and define dot products on them.

Definition 3.3. The *dot product* of any $u, x \in \{0,1\}^n$ is defined as

$$u \odot x = \sum_{i=1}^{n} u_i x_i \pmod 2$$

where $u = u_1 u_2 ... u_n$ and $x = x_1 x_2 ... x_n$.

Definition 3.4. For $u \in \{0,1\}^n$, define $\ell_u : \{0,1\}^n \to \{0,1\}$ such that

$$\ell_u(x) = u \odot x$$

It immediately follows that $\ell_u(x)$ is a linear function.

The next lemma, called the random subsum principle, shows that if two bit strings u, v are different, then for half of the choices of x, $\ell_u(x) \neq \ell_v(x)$. It appears frequently throughout the design of the PCP verifier in the form, $\Pr[\ell_u(x) \neq \ell_v(x)] = \frac{1}{2}$.

Lemma 3.5 (Random Subsum Principle)**.** *Let $u, v \in \{0,1\}^n$. If $u \neq v$, then for half of the choices of x, $\ell_u(x) \neq \ell_v(x)$.*

Proof. Let $u = u_1 u_2 ... u_n \in \{0,1\}^n$ and $v = v_1 v_2 ... v_n \in \{0,1\}^n$ such that $u \neq v$. Then u and v differ in at least one bit. Let k be the least index such that $u_k \neq v_k$. We show that $\ell_u(x) \neq \ell_v(x)$ for half the choices of $x \in \{0,1\}^n$ by a simple counting argument. Let $x = x_1 x_2 ... x_n$ and consider the following.

$$\sum_{i=1, i\neq k}^{n} u_i x_i \tag{3.6}$$

$$\sum_{i=1, i\neq k}^{n} v_i x_i \tag{3.7}$$

By definition,

$$\ell_u(x) = \sum_{i=1, i\neq k}^{n} u_i x_i + u_k x_k \pmod 2 \tag{3.8}$$

$$\ell_v(x) = \sum_{i=1, i\neq k}^{n} v_i x_i + v_k x_k \pmod 2 \tag{3.9}$$

Suppose (3,6) = (3.7). Then, we have to set $x_k = 1$ to ensure $\ell_u(x) \neq \ell_v(x)$. Suppose, on the other hand, (3,6) $\neq$ (3.7). In this case, setting $x_k = 0$ ensures the inequality. Since there are 2^n possible choices of x, but a single bit is fixed for every choice, there are 2^{n-1} possible choices of x where $\ell_u(x) \neq \ell_v(x)$.

□

3.2. **Walsh-Hadamard Code.** We will use the Walsh-Hadamard code (WH code) as our main tool. The WH code is a way of encoding binary strings of length n as binary strings of length 2^n.

Definition 3.10. $WH(u) = \ell_u$

This definition may seem counterintuitive since WH maps binary strings of length n to binary strings of length 2^n while ℓ_u is a function itself. However, we can think of ℓ_u as a binary string of length 2^n. Let $i \in \{0,1\}^n$ and set $\ell_u(i) \in \{0,1\}$ to be the ith bit of the WH code. This way, we can express ℓ_u as a binary string of length 2^n. Quite surprisingly, there is a one-to-one correspondence between linear functions and WH codes.

Theorem 3.11. *A function $f : \{0,1\}^n \to \{0,1\}$ is linear if and only if there exists some $u \in \{0,1\}^n$ such that $f(x) = \ell_u(x)$.*

Proof. If. Suppose $f(x) = \ell_u(x)$. It follows from definition that f is linear.
Only if. Suppose $f : \{0,1\}^n \to \{0,1\}$ is linear. We have to show that there exists some u such that $f(x) = \ell_u(x)$. Consider the following bit vectors:

$$e_1 = 100...0, e_2 = 010...0, ..., e_n = 000...1$$

where $e_i \in \{0,1\}^n$. Using these vectors, we can decompose any vector $x = x_1 x_2 ... x_n$ as,

$$x = x_1 e_1 + x_2 e_2 + ... + x_n e_n$$

Since $f(x)$ is linear,

$$\begin{aligned} f(x) &= f(x_1 e_1) + f(x_2 e_2) + ... + f(x_n e_n) \\ &= x_1 f(e_1) + x_2 f(e_2) + ... + x_n f(e_n) \\ &= \sum_{i=1}^{n} x_i u_i \\ &= \ell_u(x) \end{aligned}$$

where $u_i = f(e_i)$.

□

The Walsh-Hadamard code distributes the original message into a message of much larger length. Hence, we can retrieve the original message with high probability even if the WH code is partially corrupted because the original information is distributed over longer strings and each bit in the codeword depends on a lot of

bits of the source word. This characteristic allows us to build a PCP verifier on the WH code.

3.2.1. **Linearity Testing.** Suppose we are given access to a function $f : \{0,1\}^n \to \{0,1\}$. How do we determine if f is a Walsh-Hadamard codeword? The obvious way would be to check if

$$f(x+y) = f(x) + f(y) \tag{3.12}$$

for every $x, y \in \{0,1\}^n$. Clearly, this test is inefficient since it takes $O(2^n)$ queries to f.

With high confidence, can we test f for linearity by reading only a *constant* number of its values? The natural test would be to chose x, y at random and check if it satisfies equation (3.12). This test accepts a linear function with probability 1. However, it does not guarantee that every function that is not linear is rejected with high probability. For example, if f differs from a linear function at only a small fraction of its inputs, then such a *local* test would reject f with very low probability since the chance of the test encountering the nonlinear part is small. Thus, we will not be able to distinguish f from a linear function very well. This suggests that the test does not work for functions that are "close" to a linear function. However, the test may work for functions that are *far from linear*. So we start by defining the "closeness" of two functions.

Definition 3.13. Let $\rho \in [0,1]$. We say that $f, g : \{0,1\}^n \to \{0,1\}$ are *ρ-close* if $\Pr_{x \in_R \{0,1\}^n}[f(x) = g(x)] \geq \rho$. We say that f is *ρ-close to a linear function* if there exists a linear function g such that f and g are *ρ-close*.

Theorem 3.14 (BLR Linearity Testing)**.** *Let $f : \{0.1\}^n \to \{0,1\}$ be such that*

$$\Pr_{x,y \in_R \{0,1\}^n}[f(x+y) = f(x) + f(y)] \geq \rho$$

for some $\rho > 1/2$. Then f is ρ-close to a linear function.

We defer the proof of Theorem 3.14 to section 5 of this paper as it requires Fourier analysis on Boolean functions. It is often convenient to set $\rho = 1 - \delta$ and say f is $(1-\delta)$-close to a linear function.

Now we define an efficient linearity test $\mathcal{T}(f)$

Definition 3.15. $\mathcal{T}(f)$ repeats the following $K = O(1/\delta)$ times.

(1) choose $x, y \in \{0,1\}^n$ indepedently at random
(2) If $f(x+y) \neq f(x) + f(y)$, then reject and stop. Otherwise, accept.

Lemma 3.16. *For any $\delta \in (0, \frac{1}{2})$, there is a δ-linearity test which randomly reads $K = O(\frac{1}{\delta})$ bits of a 2^n string, and rejects any function that is not $(1-\delta)$-close to a linear function with a probability of at least $\frac{1}{2}$.*

Proof. Setting $K = \frac{2}{\delta}$ for $\mathcal{T}(f)$ gives us the desired test.
Define $\epsilon = \Pr_{x,y \in_R \{0,1\}^n}[f(x+y) \neq f(x) + f(y)]$. By Theorem 3.14, we know that $\epsilon \geq \delta$. Given these conditions, the probability that $\mathcal{T}(f)$ rejects is at least

$$1 - (1-\epsilon)^{2/\delta} > 1 - e^{-2\epsilon/\delta} \geq 1 - e^{-2} > 1/2$$

Since for each iteration, $\mathcal{T}(f)$ queries 3 bits of f, the test randomly reads a total of $6/\delta$ bits. □

To sum up, $\mathcal{T}(f)$ satisfies the following conditions for $\rho > \frac{1}{2}$:

(1) (Completeness) $\Pr[\mathcal{T}(f) = 1] = 1$ when f is linear
(2) (Soundness) $\Pr[\mathcal{T}(f) = 0] \geq \frac{1}{2}$ when f is not ρ-close to linear

Hence, we can see that $\mathcal{T}(f)$ is the desired linearity test that always accepts f if it is linear and rejects with high probability if it is far from linear.

3.2.2. **Local Decoding.** Suppose $\delta < \frac{1}{4}$ and let $f : \{0,1\}^n \to \{0,1\}$ be a function that is $(1-\delta)$-close to a Walsh-Hadamard codeword g. We argue that g is uniquely defined using the random subsum principle.

Lemma 3.17. *Suppose $\delta < \frac{1}{4}$ and the function $f : \{0,1\}^n \to \{0,1\}$ is $(1-\delta)$-close to some linear function g. Then g is unique.*

Proof. Suppose g is not unique, Then there exists two distinct linear functions g and g' such that f is $(1-\delta)$-close to functions both g and g'. By definition,

$$\Pr_{x \in_R \{0,1\}^n}[f(x) = g(x)] \geq 1 - \delta > 3/4 \tag{3.18}$$

$$\Pr_{x \in_R \{0,1\}^n}[f(x) = g'(x)] \geq 1 - \delta > 3/4 \tag{3.19}$$

(3.16) and (3.17) imply,

$$\begin{aligned} &\Pr_{x \in_R \{0,1\}^n}[(f(x) = g(x)) \wedge (f(x) = g'(x))] \\ &= \Pr_{x \in_R \{0,1\}^n}[g(x) = g'(x)] > 9/16 > 1/2 \end{aligned}$$

However by lemma 3.5,

$$\Pr_{x \in_R \{0,1\}^n}[g(x) = g'(x)] = 1/2$$

Hence we have a contradiction.

□

Given $x \in \{0,1\}^n$ and random access to f, can we obtain the value $g(x)$? It seems that by querying f at x, we can obtain $g(x)$ with good probability because $f(x) = g(x)$ for most of the x's. However, the given x could be one of the places where $f(x) \neq g(x)$, so this single-query procedure is insufficient.

Fortunately, there is still a simple way to recover $g(x)$ with high probability. Notice that since g is a Walsh-Hadamard codeword, $g(x+r) = g(x) + g(r)$, for any $r \in \{0,1\}^n$. Then, $g(x+r) - g(r) = g(x)$. Therefore, $f(x+r) - f(r)$ is highly likely to be equal to $g(x)$.

Definition 3.20. Let $\mathcal{D}(f)_x$ be a procedure such that given an input x and oracle access to a function f, decodes $f(x)$. We define $\mathcal{D}(f)_x$ as follows:

(1) choose $r \in \{0,1\}^n$ at random
(2) query $f(x+r), f(r)$ and output $f(x+r) - f(r)$.

Lemma 3.21. *If $f : \{0,1\}^n \to \{0,1\}$ is $(1-\delta)$-close to a Walsh-Hadamard codeword g, then, for all $x \in \{0,1\}^n$,*

$$\Pr[\mathcal{D}(f)_x = g(x)] \geq 1 - 2\delta$$

Proof. Since f is $(1-\delta)$-close to g,

$$\Pr_{r \in_R \{0,1\}^n}[f(r) \neq g(r)] < \delta$$

Similarly,

$$\Pr_{r \in_R \{0,1\}^n}[f(x+r) \neq g(x+r)] < \delta$$

By the union bound,

$$\Pr_{r \in_R \{0,1\}^n}[f(x+r) \neq g(x+r) \vee f(r) \neq g(r)] < 2\delta$$

Hence,

$$\Pr[f(x+r) - f(x) = g(x)] \geq 1 - 2\delta$$

□

This technique is called *local decoding* of the Walsh-Hadamard code since it allows us to recover any bit of the correct codeword (the linear function g) from a corrupted version (the function f) while making only a constant number of queries.

3.3. **A Proof of the Weak PCP Theorem.** We will prove the weak PCP theorem by constructing a PCP verifier on QUADEQ. We first define CKT-SAT and QUADEQ. Then, we will show that QUADEQ is NP-complete by reducing from CKT-SAT.

3.3.1. **QUADEQ is NP-complete.**

Definition 3.22. The language CKT-SAT consists of all circuits, represented as strings, that produce a single bit of output and which have a satisfying assignment. An n-input circuit C is in CKT-SAT iff there exists an assignment $u \in \{0,1\}^n$ such that $C(u) = 1$.

Definition 3.23. The language QUADEQ consists of all sets of quadratic equations over GF(2), represented as strings, which have a satisfying assignment. (a quadratic equation over $u_1, u_2 ...., u_n$ has the form $\sum_{i,j \in [n]} a_{i,j} u_i u_j = b$). A collection A of m quadratic equations is in QUADEQ if there exists an assignment $u \in \{0.1\}^n$ such that u satisfies all the m quadratic equations in the collection.

As mentioned in Section 1, CKT-SAT is NP-complete. Hence, Karp reducing CKT-SAT to QUADEQ will suffice to show that QUADEQ is NP-complete.

Lemma 3.24. *CKT-SAT* $\leq_p$ *QUADEQ*

Proof. Let C be a circuit. We first assign a variable to represent the value of each wire in the circuit. Then we express the AND, NOT, and OR gates using equivalent quadratic polynomials. For instance, if i is an AND gate with input j, k we have the equivalent quadratic polynomial $z_i = z_j z_k$ or $z_i - z_j z_k = 0$. The equivalent quadratic equation for any gate i with input j, k is,

$$z_i = \begin{cases} z_j z_k, & \text{if } i \text{ is an AND gate} \\ z_j + z_k + z_j z_k & \text{if } i \text{ is an OR gate} \\ (1 - z_j), & \text{if } i \text{ is a NOT gate and } j \text{ is its input} \\ z_j & \text{if } i \text{ is an OUTPUT gate and } j \text{ is its input} \end{cases}$$

Clearly, this set of quadratic equations is satisfiable if and only if C is satisfiable. Also, this reduction is also polynomial-time in the input size of the circuit. Hence, CKT-SAT $\leq_p$ QUADEQ.

□

Corollary 3.25. *QUADEQ is NP-complete*

3.3.2. **Constructing a Proof.** QUADEQ gives a set of quadratic equations as input and asks if there is an satisfying assignment for this collection of equations. Since $u_i = (u_i)^2$ in GF(2), we can assume that all terms in the equations have exactly degree two. This allows us to describe m quadratic equations over variables $u_1, u_2, ..., u_n$ using matrices. Let A be a $m \times n^2$ matrix, U an n^2 matrix, and $\mathbf{b}$ an m dimensional vector. QUADEQ asks, given inputs A and $\mathbf{b}$, find U such that (1) $AU = \mathbf{b}$ and (2) U is the *tensor product* $\mathbf{u} \otimes \mathbf{u}$ of some n dimensional vector $\mathbf{u}$.

It would seem natural to think that the proof for QUADEQ should consist of the pair $(U, \mathbf{u}) \in \{0,1\}^{n+n^2}$. Unfortunately, the proof $(U, \mathbf{u})$ is not suitable for a PCP($poly(n), 1$) verifier. Recall that we have $poly(n)$ random coins and are making only a constant number of queries. Since the PCP verifier bases its decision on constant number of bits of the proof, each bit in the proof has to depend on a large number of bits of the natural proof $(U, \mathbf{u})$. In addition, the $poly(n)$ random coins allows the verifier to effectively cover much longer proofs. As the astute reader may have noticed, the Walsh-Hadamard code meets these requirements.

Hence, our PCP verifier will have acess to a long proof $\pi \in \{0,1\}^{2^n+2^{n^2}}$, which we interpret as a pair of functions $f : \{0,1\}^n \to \{0,1\}$ and $g : \{0,1\}^{n^2} \to \{0,1\}$. In a correct proof, f will the be Walsh-Hadamard encoding for $\mathbf{u}$ and the function g will be the Walsh-Hadamard encoding for $\mathbf{u} \otimes \mathbf{u}$.

3.3.3. **Test Procedures for the PCP verifier.** To verify π, our PCP verifier V needs to carry out the following procedures:

Step 1 Check that f, g are linear functions i.e., f, g are Walsh-Hadamard codewords.

Step 2 Check that f, g encode the same source word i.e., g encodes $\mathbf{u} \otimes \mathbf{u}$ and f encodes $\mathbf{u}$.

Step 3 Check that $U = \mathbf{u} \otimes \mathbf{u}$ satisfies $AU = \mathbf{b}$.

All these steps must be carried out by making only a constant number of queries to the proof π. Moreover, tests performed by the verifier have to (1) accept correct proofs with probability 1 and (2) reject proofs with high probability if the instance is not satisfiable.

Step 1. Linearity of f and g can be tested by the efficient linearity testing $\mathcal{T}(f)$ defined in the previous section. The verifier performs a 0.99-linearity test on both f, g, using only a constant number $K = O(\frac{1}{0.01})$ of queries. Step 1 applies this linearity test twice to each of f and g. This gives us the following:

(1) (Completeness) If f and g are linear functions, proof π is accepted with probability 1

(2) (Soundness) If either f or g is not 0.99-close to a linear function, proof π is accepted with probability of at most $(\frac{1}{2})^2 + (\frac{1}{2})^2 = \frac{1}{2}$, i.e. π is rejected with probability of at least $\frac{1}{2}$.

(3) (Efficiency) Step 1 makes $2K$ random queries from f and $2K$ random queries from g. Each query to f and g requires n and n^2 random bits, respectively. Hence, we need $2K(n^2 + n) = O(n^2)$ random bits for Step 1.

Thus, if either f or g is not 0.99-close to a linear function, then Step 1 rejects with high probability. Therefore, for the rest of the procedure we can assume that there exist linear functions $\tilde{f}$, and $\tilde{g}$ such that f is 0.99-close to $\tilde{f}$ and g is 0.99-close to $\tilde{g}$.

We also assume that the verifier can query $\tilde{f}$ and $\tilde{g}$ at any desired point. The reason is that local decoding allows us to retrieve the values of $\tilde{f}$ and $\tilde{g}$ with good probability. Steps 2 and 3 will only use a small queries to $\tilde{f}$ and $\tilde{g}$. Thus with high probability (*say* > 0.9) local decoding will succeed on these queries.

Step 2. If f and g encode the same source word $\mathbf{u}$, where $f = WH(\mathbf{u})$ and $g = WH(\mathbf{u} \otimes \mathbf{u})$, we say that f and g are *consistent*. Consistency of linear functions $\tilde{f}$ and $\tilde{g}$ can be tested by the following test:

Definition 3.26 (Consistency Test)**.** Given $f : \{0,1\}^n \to \{0,1\}$ and $g : \{0,1\}^{n\times n} \to \{0.1\}$, the test procedure $\mathcal{C}(f,g)$ is defined as:

(1) Choose $\mathbf{r}, \mathbf{r}' \in_R \{0,1\}^n$
(2) Output 1 (accept) if $g(\mathbf{r} \otimes \mathbf{r}') = f(\mathbf{r})f(\mathbf{r}')$. Output 0 (reject) if otherwise.

Suppose that $\tilde{g} = WH(\mathbf{w})$ for some $\mathbf{w} \in \{0,1\}^{n\times n}$ and $\tilde{f} = WH(\mathbf{u})$ for some $\mathbf{u} \in \{0,1\}^n$.

Lemma 3.27 (Completeness)**.** *If* $\mathbf{w} = \mathbf{u} \otimes \mathbf{u}$*, then* $\Pr[\mathcal{C}(\tilde{f}, \tilde{g}) = 1] = 1$.

Proof.

$$\begin{aligned}
\tilde{f}(\mathbf{r})\tilde{f}(\mathbf{r}') &= (\mathbf{u} \odot \mathbf{r}) \cdot (\mathbf{u} \odot \mathbf{r}') \\
&= (\sum_{i\in[n]} u_i r_i) \cdot (\sum_{j\in[n]} u_j r'_j) \\
&= (\sum_{i,j\in[n]} u_i u_j r_i r'_j) \\
&= (\mathbf{u} \otimes \mathbf{u}) \odot (\mathbf{r} \otimes \mathbf{r}') \\
&= \tilde{g}(\mathbf{r} \otimes \mathbf{r}')
\end{aligned}$$

□

Lemma 3.28 (Soundness)**.** *If* $\mathbf{w} \neq \mathbf{u} \otimes \mathbf{u}$*, then* $\Pr[\mathcal{C}(\tilde{f}, \tilde{g}) = 0] \geq \frac{1}{4}$.

Proof. Let W be an $n \times n$ matrix with same entries as $\mathbf{w}$, let U be the $n \times n$ matrix such that $U_{i,j} = u_i u_j$, and consider $\mathbf{r}$ as a row vector and $\mathbf{r}'$ as a column vector. In this notation,

$$\tilde{g}(\mathbf{r} \otimes \mathbf{r}') = \mathbf{w} \odot (\mathbf{r} \otimes \mathbf{r}') = \sum_{i,j\in[n]} w_{i,j} r_i r'_j = \mathbf{r}W\mathbf{r}'$$

$$\tilde{f}(\mathbf{r})\tilde{f}(\mathbf{r}') = (\sum_{i\in[n]} u_i r_i) \cdot (\sum_{j\in[n]} u_j r'_j) = \sum_{i,j\in[n]} u_i u_j r_i r'_j = \mathbf{r}U\mathbf{r}'$$

The consistency test outputs 0 (rejects) if $\mathbf{r}W\mathbf{r}' \neq \mathbf{r}U\mathbf{r}'$. The matrices, W and U, can be also seen as an n-tuple of column matrices $(w_1, ..., w_n)$ and $(u_1, ..., u_n)$. $W \neq U$ implies that at least one pair of the column matrices are not equal, i.e., $w_i \neq u_i$ for some $i \in [n]$. We can apply the random subsum principle on this pair of column matrix and conclude that at least $1/2$ of all $\mathbf{r}$ satisfy $\mathbf{r}W \neq \mathbf{r}U$.

Applying the random subsum principle for each $\mathbf{r}$, we see that at least $1/2$ of the $\mathbf{r}'$ satisfy $\mathbf{r}W\mathbf{r}' \neq \mathbf{r}U\mathbf{r}'$. Hence, we conclude that the trial rejects for at least $1/4$ of all pairs of $\mathbf{r}$, $\mathbf{r}'$.

□

Step 2 performs the consistency test 3 times. This gives us the following:

(1) (Completeness) if $f = \tilde{f} = WH(\mathbf{u})$ and $g = \tilde{g} = WH(\mathbf{u} \otimes \mathbf{u})$, then the proof $\pi = (f, g)$ passes Step 2 with probability 1.
(2) (Soundness) if f and g are .99 - close to two linear functions $\tilde{f} = WH(\mathbf{u})$ and $\tilde{g} = WH(\mathbf{w})$ respectively, such that $\mathbf{w} \neq \mathbf{u} \otimes \mathbf{u}$, then:
   - For each iteration, each of $\tilde{f}(\mathbf{r}), \tilde{f}(\mathbf{r}')$, and $\tilde{g}(\mathbf{r} \otimes \mathbf{r}')$ is incorrectly decoded with a probability of at most 0.02. Thus, the probability that all three are decoded correctly is at least $0.98^3 \approx 0.94$.
   - Given that $\tilde{f}(\mathbf{r}), \tilde{f}(\mathbf{r}')$, and $\tilde{g}(\mathbf{r} \otimes \mathbf{r}')$ are correctly decoded in a particular iteration, the probability that π is rejected in this iteration is at least $\frac{1}{4}$ by soundness of the consistency test.
   - Thus, in each iteration the probability that π is rejected is at least $\frac{0.94}{4}$.

   Therefore, the probability that π is rejected in at least one iteration is at least $1 - (1 - \frac{0.94}{4})^3 > \frac{1}{2}$.
(3) (Efficiency) In each iteration, we need n random bits for choosing each $\mathbf{r}$ and $\mathbf{r}'$. For local decoding, we need n, n and n^2 random bits for $\tilde{f}(\mathbf{r}), \tilde{f}(\mathbf{r}')$, and $\tilde{g}(\mathbf{r} \otimes \mathbf{r}')$, respectively, and query 2 bits for each one of them. Hence, Step 2 uses a total of $3(4n+n^2) = O(n^2)$ random bits, and queries $3(2+2+2) = 18$ bits of the proof π.

Step 3. Now we need to check that the linear function $\tilde{g}$ encodes a satisfying assignment. The obvious way to check this would be to compute the kth equation of the input and compare it with b_k. In other words, check if

$$\sum_{i,j \in [n]} A_{k,(i,j)} u_i u_j = b_k \tag{3.29}$$

Denoting by $\mathbf{z}$ the n^2 dimensional vector $(A_{k,(i,j)})$ (where i, j vary over $\{1, ..., n\}$), we see that the above equation can be simply expressed as

$$\tilde{g}(\mathbf{z}) = b_k$$

However, the query bits of this procedure depends on the input size since the more equations we have, the more comparisons with $\mathbf{b}$ we have to make. The PCP verifier queries only a constant number of bits, so the number of queries required have to be independent of m. Again, the random subsum principle provides us with a way out.

Definition 3.30 (Satisfiability Test)**.** Given an $m \times n^2$ matrix A, an m-dimensional vector $\mathbf{b}$, and an n^2-dimensional vector U, the satisfiability test $\mathcal{S}^{A,b}(U)$ consists of the following procedures:

(1) Choose $\mathbf{r} \in_R \{0,1\}^m$
(2) Output 1 (accept) if $(AU) \odot \mathbf{r} = \mathbf{b} \odot \mathbf{r}$. Output 0 (reject) if otherwise.

Lemma 3.31 (Completeness)**.** *If U encodes a satisfying assignment,* $\Pr[\mathcal{S}^{A,b}(U) = 1] = 1$

Proof. If U encodes a satisfying assignment,

$$AU = \mathbf{b}$$

which implies

$$(AU) \odot \mathbf{r} = \mathbf{b} \odot \mathbf{r}$$

□

Lemma 3.32 (Soundness)**.** *If $AU \neq \mathbf{b}$, then* $\Pr[\mathcal{S}^{A,b}(U) = 0] \geq 1/2$

Proof. By the random subsum principle, if $AU \neq \mathbf{b}$, then $(AU) \odot \mathbf{r} \neq \mathbf{b} \odot \mathbf{r}$ for 1/2 of the $\mathbf{r}$'s. □

Step 3 executes the satisfiability test twice. This gives us the following:

(1) (Completeness) If $g = \tilde{g} = WH(\mathbf{u} \otimes \mathbf{u})$ and $\mathbf{u}$ encodes a satisfying assignment, π passes step 3 with probability 1.
(2) (Soundness) If g is 0.99 close to a linear function $\tilde{g} = WH(\mathbf{u} \otimes \mathbf{u})$, but $\mathbf{u}$ does not encode a satisfying assignment, then:
   - In each iteration, the probability that $\tilde{g}(\mathbf{z})$ will be decoded correctly is at least 0.98.
   - Given that $\tilde{g}(\mathbf{z})$ is decoded correctly in one iteration, π is rejected by this iteration with a probability of at least $\frac{1}{2}$.

   Therefore, π will be rejected by step 3 with a probability of at least $1 - (1 - \frac{0.98}{2})^2 > \frac{1}{2}$.
(3) (Efficiency) Each iteration requires m random bits in order to choose $\mathbf{r}$, and n^2 additional random bits for decoding $\tilde{g}(\mathbf{z})$. Two queries from g is needed in order to decode $\tilde{g}(\mathbf{z})$. Therefore, step 3 uses a total of $2(m + n^2) = 2m + 2n^2$ random bits and queries a total of $2(2) = 4$ bits from g.

3.3.4. **Analysis.** Our PCP verifier V is given oracle access to π and performs the 3 steps mentioned above. Here, we show the efficiency, completeness, and soundness of this PCP verifier.

Lemma 3.33 (Efficiency)**.** *V is efficient, that is, $r(|x|) = poly(|x|)$ and $q(|x|) = O(1)$, where $|x|$ is the size of the instance of QUADEQ which depends on m and n.*

Proof. The number of random bits required for each step of the verifier is provided above. Summing them together yields

$$r(x) = (2K + 5)n^2 + (2K + 12)n + 2m$$

which is polynomial in the size of the input.

The total number of queries required is $4K + 22$, which is constant. □

Lemma 3.34 (Completeness)**.** *If an instance x of the QUADEQ problem is satisfiable, then there exists a proof π such that* $\Pr[V^\pi(x) = 1] = 1$.

Proof. Suppose the instance x is satisfiable. Then there must exist a vector $\mathbf{u} \in \{0, 1\}^n$ such that $A(\mathbf{u} \otimes \mathbf{u}) = \mathbf{b}$. If we let $f = WH(\mathbf{u})$ and $g = WH(\mathbf{u} \otimes \mathbf{u})$, the proof $\pi = (f, g)$ passes all three steps of the verifier with probability 1. □

Lemma 3.35 (Soundness)**.** *If an instance x of the QUADEQ problem is not satisfiable, then for all proof π,* $\Pr[V^\pi(x) = 0] \geq \frac{1}{2}$.

Proof. Suppose that the instance of the QUADEQ problem is not satisfiable. Then, for any vector $\mathbf{u} \in \{0, 1\}^n$, we have $A(\mathbf{u} \otimes \mathbf{u}) \neq \mathbf{b}$. Let $\pi = (f, g)$ be any binary string of length $(2^n + 2^{n^2})$. We have:

- If either f or g is not 0.99-close to a linear function, then π will be rejected by Step 1 with probability of at least $\frac{1}{2}$.

- If f and g are both 0.99-close to linear functions $\tilde{f} = WH(\mathbf{u})$ and $\tilde{g} = WH(\mathbf{w})$ respectively with $\mathbf{w} \neq \mathbf{u} \otimes \mathbf{u}$, then π will be rejected by Step 2 with probability of at least $\frac{1}{2}$.
- If f and g are 0.99 close to linear functions $\tilde{f} = WH(\mathbf{u})$ and $\tilde{g} = WH(\mathbf{w})$ respectively with $\mathbf{w} = \mathbf{u} \otimes \mathbf{u}$, then we must have $A(\mathbf{u} \otimes \mathbf{u}) \neq \mathbf{b}$. Hence, π will be rejected by Step 3 with a probability of at least $\frac{1}{2}$.

Therefore, in all cases, there is a probability of at least $\frac{1}{2}$ that π will be rejected by at least one step of the verifier. □

This shows QUADEQ $\in$ PCP($poly(n), 1$), which implies NP $\subseteq$ PCP($poly(n), 1$).

4. Fourier Analysis on Boolean Functions

In this section, we use the set $\{+1, -1\} = \{\pm 1\}$ instead of $\{0, 1\}$ for convenience. We can easily transform $\{0, 1\}$ into $\{\pm 1\}$ via the mapping $b \mapsto (-1)^b$. Note that this maps addition in GF(2) into the multiplication operation over $\mathbb{R}$.

4.1. The Fourier Expansion.

The Fourier expansion of a boolean function $f : \{\pm 1\}^n \to \{\pm 1\}$ is its representation in terms of a certain basis (called the *Fourier basis*). We first define the inner product of two functions.

Definition 4.1. Let f,g be boolean functions defined on $\{\pm 1\}^n$. The inner product of f and g, denoted $\langle f, g \rangle$, is

$$\langle f, g \rangle = \mathbb{E}_{x \in \{\pm 1\}^n}[f(x)g(x)]$$

Definition 4.2. For $\alpha \subseteq [n]$, we define $\chi_\alpha : \{\pm 1\} \to \{\pm 1\}$ by

$$\chi_\alpha(x) = \prod_{i \in \alpha} x_i$$

The Fourier basis is the set of $\chi'_\alpha s$ for all $\alpha \subseteq [n]$

Note that this basis is actually the set of linear functions over GF(2) in disguise. Every linear function of the form $\mathbf{b} \mapsto \mathbf{a} \odot \mathbf{b}$ (with $\mathbf{a}, \mathbf{b} \in \{0, 1\}^n$) is mapped by our transformation to the function taking $\mathbf{x} \in \{\pm 1\}^n$ to $\prod_{i \text{ s.t.} a_i = 1} x_i$. This basis is also orthonormal, as the following theorem shows.

Theorem 4.3. *The 2^n parity functions $\chi_\alpha : \{\pm 1\}^n \to \{\pm 1\}$ form an orthonormal basis for the vector space V of functions $\{\pm 1\}^n \to \mathbb{R}$. That is,*

$$\langle \chi_\alpha, \chi_\beta \rangle = \delta_{\alpha,\beta} \tag{4.4}$$

where $\delta_{\alpha,\beta} = 1$ iff $\alpha = \beta$ and 0 otherwise.

Proof. Suppose $\alpha \neq \beta$. By the random subsum principle, the corresponding linear functions for χ_α and χ_β differ on 1/2 of their inputs. This implies,

$$\langle \chi_\alpha, \chi_\beta \rangle = 1 \cdot \frac{1}{2} + (-1) \cdot \frac{1}{2} = 0$$

Suppose, on the other hand, $\alpha = \beta$. Then, $\chi_\alpha(x)\chi_\beta(x) = 1$ for all $x \in \{\pm 1\}^n$. Hence, $\langle \chi_\alpha, \chi_\beta \rangle = 1$. □

These facts lead us to the *Fourier expansion theorem.*

Theorem 4.5. *Every function $f : \{\pm 1\}^n \to \mathbb{R}$ can be uniquely expressed as a multilinear polynomial,*

$$f(x) = \sum_{\alpha \subseteq [n]} \hat{f}_\alpha \chi_\alpha \tag{4.6}$$

This expression is called the Fourier expansion of f, and the real number $\hat{f}_\alpha$ is called the Fourier coefficient of f on α.

Proof. It suffices to show that the Fourier basis spans the vector space V of boolean functions on $\{\pm 1\}^n$. The dimension of V is 2^n since boolean functions on $\{\pm 1\}^n$ can be expressed as binary strings in $\{\pm 1\}^{2^n}$. The Fourier basis contains 2^n orthonormal functions. Since orthogonality implies linear independence, it follows that the Fourier basis spans V. □

4.2. **Basic Fourier Formulas.** In this section, we introduce some useful formulas.

Lemma 4.7. *For $f : \{\pm 1\}^n \to \mathbb{R}$ and $\alpha \subseteq [n]$, the Fourier coefficient of f on α is given by*

$$\hat{f}_\alpha = \langle f, \chi_\alpha \rangle$$

Proof. We can verifiy this directly using orthonormality of the parity functions.

$$\langle f, \chi_\alpha \rangle = \langle \sum_{\beta \subseteq [n]} \hat{f}_\beta \chi_\beta, \chi_\alpha \rangle = \hat{f}_\alpha \langle \chi_\alpha, \chi_\alpha \rangle = \hat{f}_\alpha$$

□

Theorem 4.8 (Plancherel's Theorem)**.** *For any $f, g : \{\pm 1\}^n \to \mathbb{R}$,*

$$\langle f, g \rangle = \sum_{\alpha \subseteq [n]} \hat{f}_\alpha \hat{g}_\alpha$$

Proof. This can also be verified directly.

$$\langle f, g \rangle = \langle \sum_{\alpha \subseteq [n]} \hat{f}_\alpha \chi_\alpha, \sum_{\beta \subseteq [n]} \hat{g}_\beta \chi_\beta \rangle = \sum_{\alpha, \beta \subseteq [n]} \hat{f}_\alpha \hat{g}_\beta \delta_{\alpha,\beta} = \sum_{\alpha \subseteq [n]} \hat{f}_\alpha \hat{g}_\alpha$$

□

Corollary 4.9 (Parseval's Identity)**.** *For any $f : \{\pm 1\}^n \to \mathbb{R}$,*

$$\langle f, f \rangle = \sum_{\alpha \subseteq [n]} \hat{f}_\alpha^2$$

4.3. **BLR Linearity Testing.** Now we will prove Theorem 3.14, thus completing the proof of the weak PCP theorem. We first rephrase the linearity test using $\{\pm 1\}$ instead of GF(2).

Definition 4.10. For any two vectors $\mathbf{x}, \mathbf{y} \in \{\pm 1\}^n$, we define componentwise multiplication $\mathbf{xy}$ by

$$\mathbf{xy} = (x_1 y_1, ..., x_n y_n)$$

Notice that $\chi_\alpha(\mathbf{xy}) = \chi_\alpha(\mathbf{x})\chi_\alpha(\mathbf{y})$.

Lemma 4.11. *For every $\epsilon \in [0, 1]$, and functions $f, g : \{\pm 1\}^n \to \{\pm 1\}$, f and g agree on $\frac{1}{2} + \frac{\epsilon}{2}$ of its inputs iff $\langle f, g \rangle = \epsilon$.*

Proof. Note that the inner product is equal to the fraction of inputs which they agree on minus the fraction of inputs on which they disagree. Suppose f and g agree on $\frac{1}{2} + \frac{\epsilon}{2}$ of its inputs. Then,

$$\langle f, g\rangle = \mathbb{E}_{x\in\{\pm1\}^n}[f(x)g(x)] = \frac{1}{2} + \frac{\epsilon}{2} - (1 - (\frac{1}{2} + \frac{\epsilon}{2})) = \epsilon$$

This equation also gives us the other direction. □

Recall that the parity function corresponds to a linear function on GF(2). Thus, if f has a large Fourier coefficient, then it has significant agreement with some linear function. This allows us to rephrase the BLR linearity test as the following theorem.

Theorem 4.12. *Suppose that* $f : \{\pm1\}^n \to \{\pm1\}$ *satisfies* $\Pr_{\mathbf{x},\mathbf{y}}[f(\mathbf{xy}) = f(\mathbf{x})f(\mathbf{y})] \geq \frac{1}{2} + \epsilon$. *Then, there is some* $\alpha \subseteq [n]$ *such that* $\hat{f}_\alpha \geq 2\epsilon$.

Proof. We can rephrase the hypothesis as $\mathbb{E}_{\mathbf{x},\mathbf{y}}[f(\mathbf{xy})f(\mathbf{x})f(\mathbf{y})] \geq (\frac{1}{2}+\epsilon)-(\frac{1}{2}-\epsilon) = 2\epsilon$. Expressing f by its Fourier expansion,

$$2\epsilon \leq \mathbb{E}_{\mathbf{x},\mathbf{y}}[f(\mathbf{xy})f(\mathbf{x})f(\mathbf{y})] = \mathbb{E}_{\mathbf{x},\mathbf{y}}[(\sum_\alpha \hat{f}_\alpha\chi_\alpha(\mathbf{xy}))(\sum_\beta \hat{f}_\beta\chi_\beta(\mathbf{x}))(\sum_\gamma \hat{f}_\gamma\chi_\gamma(\mathbf{y}))$$

Since $\chi_\alpha(\mathbf{xy}) = \chi_\alpha(\mathbf{x})\chi_\alpha(\mathbf{y})$, this becomes

$$= \mathbb{E}_{\mathbf{x},\mathbf{y}}[\sum_{\alpha,\beta,\gamma} \hat{f}_\alpha\hat{f}_\beta\hat{f}_\gamma\chi_\alpha(\mathbf{x})\chi_\alpha(\mathbf{y})\chi_\beta(\mathbf{x})\chi_\gamma(\mathbf{y})]$$

Using linearity of expectation and independence of **x, y**

$$\begin{aligned} &= \sum_{\alpha,\beta,\gamma} \hat{f}_\alpha\hat{f}_\beta\hat{f}_\gamma\mathbb{E}_{\mathbf{x},\mathbf{y}}[\chi_\alpha(\mathbf{x})\chi_\alpha(\mathbf{y})\chi_\beta(\mathbf{x})\chi_\gamma(\mathbf{y})] \\ &= \sum_{\alpha,\beta,\gamma} \hat{f}_\alpha\hat{f}_\beta\hat{f}_\gamma\mathbb{E}_{\mathbf{x}}[\chi_\alpha(\mathbf{x})\chi_\beta(\mathbf{x})]\mathbb{E}_{\mathbf{y}}[\chi_\alpha(\mathbf{y})\chi_\gamma(\mathbf{y})] \\ &= \sum_{\alpha,\beta,\gamma} \hat{f}_\alpha\hat{f}_\beta\hat{f}_\gamma\langle\chi_\alpha,\chi_\beta\rangle\langle\chi_\alpha,\chi_\gamma\rangle \end{aligned}$$

Since $\langle\chi_\alpha,\chi_\beta\rangle = \delta_{\alpha,\beta}$, this becomes

$$\begin{aligned} &= \sum_\alpha \hat{f}_\alpha^3 \\ &\leq (\max_\alpha \hat{f}_\alpha) \times (\sum_\alpha \hat{f}_\alpha^2) = \max_\alpha \hat{f}_\alpha \end{aligned}$$

Since $\sum_\alpha \hat{f}_\alpha^2 = \langle f, f\rangle = 1$. Hence $\max_\alpha \hat{f}_\alpha \geq 2\epsilon$. □

BLR linearity testing presented in section 3 states that for all $\rho > 1/2$, if $f(x + y) = f(x) + f(y)$ with probability at least ρ, then f is ρ-close to some linear function. Theorem 4.12 implies this. To see this, recall that $\langle f, g\rangle = 2\epsilon$ iff f and g agree on $1/2+\epsilon$ of their inputs. Theorem 4.12 implies that if $f(x+y) = f(x)+f(y)$ with probability at least $1/2 + \epsilon = \rho$, then there exists a Fourier coefficient greater or equal to 2ϵ.The Fourier coefficient of $\hat{f}_\alpha$ is the inner product of f with some linear function χ_α. Hence, if $\hat{f}_\alpha \geq 2\epsilon$ for some $\alpha \subseteq [n]$, then f agrees with χ_α on $1/2 + \epsilon = \rho$ of its inputs. This implies that f is ρ - close to a linear function.

5. Conclusion

In this paper, we introduced the PCP theorem and its equivalent formulation from the hardness of approximation view. The theorem states that proofs for NP decision problems can be verified by just checking a constant number of bits. Since we are not reading every bit of the proof, it is possible that we may accept a false proof. However, this probability can be made small by running the verifier multiple times. From the hardness of approximation view, the PCP theorem says that approximating a solution arbitrarily is no easier than computing the exact solution. In other words, we cannot approximate a solution arbitrarily unless P=NP.

We also proved a weaker version of the PCP theorem, which states that proofs of exponential size can be verified by randomly checking a constant number of bits. It is weaker in the sense that while the PCP theorem validates proofs of polynomial size, the weaker version validates proofs of exponential size. We proved this result by using properties of the Walsh-Hadamard code and BLR linearity testing.

Acknowledgments. It is a pleasure to thank my mentor, Angela Wu, for her guidance.

References

[1] Harry Lewis, Christos Papadimitrou, Elements of the Theory of Computation, Prentice Hall,1997
[2] Sanjeev Arora, Boaz Barak, Computational Complexity: A Modern Approach, Cambridge University Press, 2009
[3] Oded Goldreich, Computational Complexity, Cambridge Univesity Press, 2008
[4] Jose Falcon, Mitesh Jain, "An introductino to Probabilistically Checkable Proofs and the PCP Theorem", June 2, 2013
[5] Ola Svensson, "Approximation Algorithms and Hardness of Approximation: Lecture 17-18", Retrieved August 10, 2013, from http://theory.epfl.ch/osven/courses/Approx13/Notes/lecture17-18.pdf
[6] Ryan O'Donell, "Analysis of Boolean Functions", Retrieved August 10, 2013, from http://www.contrib.andrew.cmu.edu/ ryanod/?cat=62