Approximate Algorithms on Checking Proofs with Probabilistic Methods

Yi-Wen Wang

May 26, 2000

Contents

Chapter 1

Introduction

Intuitively, to verify a problem's solution is easier than to solve it. By the definition of NP, let L be a language in NP, then there exists a relation R, for any $x \in L$, there exists a witness y of length $poly(|x|)$ such that $(x, y) \in R$, which can be decided in polynomial time. In fact, a weaker verifier, which runs in logarithmic space, is needed: since 3SAT can be verified in logrithmic space, and every NP-problem can be reduced to 3SAT under a logarithmic-space reduction [13].

A more surprising result is NP=PCP($\log |x|, 1$), which states that for any problem in NP, there exists an relation R such that to verify whether $(x, y) \in R$, the verifier just has to toss $O(\log |x|)$ random bits and to read $O(1)$ bits from y, where the running time of the verifier is a polynomial of $|x|$ [1]. In [3] and [15], it is possible to construct proof systems for any proof in a reasonable formal system, where the verifier runs in $O(|x| + \lg y)$-time. In all these cases, verifier has to perform $\Omega(|x|)$ work.

Recently, Ergün *et al.* suggest an approximate model of PCP, in which it is enough for the verifier to know that the proof is *close* to correct, and the running time of the verifier reduces to sublinear [6]. This idea is also adaptive to interactive proof systems. If we allow the prover and the verifier to communicate after the proof is written, verification can be facilitated. In fact, the efficiency of the algorithms with the approximate PCP system has only a fraction of $O(\log n)$ loss.

What we mean by "*close*" is problem dependent. For example, for a MAX3SAT problem, for a goal of K satisfied clauses, one might consider "close" when there are $(1 - \epsilon)K$ satisfied clauses, for some small ϵ. On the

other hand, for a 3SAT problem with m clauses, this is not acceptible for any $\epsilon > 1/m$ since it might be the case that for each truth assignment, there are unsatisfied clauses of some fraction smaller than ϵ, say, $O(1/m)$ out of the m clauses, such that the expression is overall unsatisfiable. Throughout this thesis, we will concentrate on optimization problems, in which the slightly faulty proof does not ruin the overall task.

Structure of this thesis. In this thesis, we will explore problems related to approximate PCP/IP systems suggeusted in [6] and discuss the relationship between the new method and traditional complexity classes. In Chapter 2, we introduce the models used in later chapters as well as related concepts. In Chapter 3, we start with two simple problems to show how the proof system works and build some basic algorithms to facilitate the proofs of more complicated problems. In Chapter 4, we show how to verify proofs of NP-complete problems with the basic algorithms in Chapter 3 and discuss the relationship to MAXSNP_0 (which is a fragment of NP optimization problems) and coNP. We also apply the proof system to a problem beyond NP and coNP. In Chapter 5, a brief conclusion is given.

Chapter 2

Prelimilaries

2.1 Definitions of IP and PCP

Before we introduce the approximate proof systems, we give a brief review of PCP and IP and important theorems first.

Definition 2.1.1 *(IP [7]) An interactive proof system (IPS) consists of two players, an infinitely powerful prover P and a probabilistic polynomial-time verifier V. P and V share a read-only input tape and a read/write communication tape. P and V have their own private work tape and random-bit tape. The output tape can only be accessed by V. The proof system accepts iff the verifier enters an accepting state.*

An interactive proof system for a language L is a pair $< P, V >$,

1. *If $x \in L$, then $Prob(< P, V > (x)\ accepts) \geq 3/4$.*
2. *If $x \notin L$, then for all P', $Prob(< P', V > (x)\ accepts) \leq 1/4$.*

A round of an interactive protocol is a message from the verifier to the prover followed by a message from the prover to the verifier. We let $IP(f(n))$ represent the languages accepted by interactive proof systems bounded by $f(n)$ rounds. The class $IP = \bigcup IP(poly(n))$.

It is obvious that $NP \subseteq IP(1)$; in polynomial time, V can make sure that whether an instance is in L if P provides a valid proof string without using any random bits. A important relationship between IP and traditional complexity class is proved by Shamir [14].

Theorem 2.1.1 *IP=PSPACE.*

Definition 2.1.2 *(PCP [2]) A language L is in $PCP(f(n), g(n))$ if there is a polynomial-time randomized oracle machine $M^y(r, x)$ which works as follows:*

1. *It takes input x and a (random) string r of length $O(f(n))$, where $n = |x|$.*
2. *It generates a query set $Q(r, x) = \{q_1, \ldots, q_m\}$ of size $m = O(g(n))$.*
3. *It reads the bits $y_{q_1}, \ldots, y_{q_m}$.*
4. *It makes a polynomial-time computation using r, x and $y_{q_1}, \ldots, y_{q_m}$ and ouputs $M^y(r, x) \in \{0, 1\}$.*

Moreover the following acceptance conditions hold for some $\delta > 0$ and all x:

1. *If $x \in L$ then there exists a y such that for every r we have $M^y(r, x) = 1$.*
2. *If $x \notin L$ then for every y we have $Prob_r(M^y(r, x)) = 0) \geq \delta$.*

It is easy to see that PCP(0,$n^{O(1)}$)=NP. In fact, a better result is proved in [1] as follows.

Theorem 2.1.2 $NP = PCP(\log n, 1)$.

2.2 Definitions of Approximate Proof Systems

Before we define the approximate proof system formally, let us give a simple example first.

Among a set of n weighted elements, the verifier V wants to know the i^{th} heaviest one for some reason. The prover P gives V an element e_i and claims that it satisfies V's demand. As a proof, P gives two arrays L and L' with length i and $n - i$ respectively, where L (L') contains all the indices j such that the element e_j in the set is heavier (lighter) than e_i. In some cases, V does not ask for *exactly* the i^{th} heaviest element, but the one *around* i^{th}, say, about $i \pm \epsilon n$. If so, V only has to perform constant work for verification

independent of n as follows. V randomly picks $O(1/\epsilon)$ indices out of L (L'), reads out the corresponding elements from the set, and rejects if one of them is not heavier (lighter) than e_i.

This example gives an approximate proof system to prove that e_i is the i^{th} heaviest one in the set, where the error is bounded by ϵn. To facilitate the discussion of accuracy, we introduce the concept of *distance function* . For any instance x, let y be a candidate of the real solution $f(x)$. A distance function $\triangle(y, f(x))$ denotes how far y is from $f(x)$, by a value ranging from 0 to 1. When $y = f(x)$, $\triangle(y, f(x)) = 0$. The larger the value of $\triangle$, the farther away from $f(x)$ y is. For convenience, in later sections sometimes we refer to the distance as $\triangle$.

In the above example, we check the correctness of P's answer in two phases, the lower bound phase and the upper bound phase. Denote by i' the real order of e_i. In the lower bound phase, the distance $\triangle$ is 0 for $i' \geq i$ and $\triangle = |i - i'|/i$ otherwise. In the upper bound phase, $\triangle$ is 0 if $i \geq i'$ and $\triangle = |i' - i|/(n - i)$ otherwise. It is clear that if the distance $\triangle$ exceeds ϵ, V will find out with high probability.

Now we describe the approximate proof system more formally. In the approximate proof system, P and V share an input tape and a proof tape (write-only for P, read-only for V), all other tapes being private. Both P and V compute in a RAM model with the following characteristics:

1. Each register can store an arbitarily long bit string.

2. Each basic operation between two registers, such as read, write, compare, add, shift, etc., takes constant time.

We also assume that "to randomly pick some element" takes constant time.

Definition 2.2.1 *(Approximate IPS [6]) Let $\triangle(\cdot, \cdot)$ be a distance function. A function f is said to have an $t(\epsilon, n)$-approximate interactive proof system (approximate IPS) with distance function $\triangle$ if there is a randomized verifier V such that for all inputs ϵ and x of size n, the following holds. Let y be the contents of the proof tape, then:*

1. *If $\triangle(y, f(x)) = 0$, there is a prover P, such that V accepts with probability at least 3/4 (over the internal coin tosses of V).*

2. *If $\triangle(y, f(x)) > \epsilon$, for all provers P', V rejects with probability at least 3/4.*

3. *V runs in $O(t(\epsilon, n))$ time.*

Definition 2.2.2 *(Approximate PCPS [6]) Let $\triangle(\cdot, \cdot)$ be a distance function. A function f is said to have an $t(\epsilon, n)$-approximate probabilistically checkable proof system (approximate PCPS) with distance function $\triangle$ if there is a randomized verifier V such that for all inputs ϵ and x of size n, the following holds. On the output tape, there are two strings y and z, where y is a candidate of the solution, z is the proof of $y = f(x)$. Then:*

1. *If $\triangle(y, f(x)) = 0$, there is a proof z, such that V accepts with probability at least 3/4 (over the internal coin tosses of V).*

2. *If $\triangle(y, f(x)) > \epsilon$, for all proof z', V rejects with probability at least 3/4.*

3. *V runs in $O(t(\epsilon, n))$ time.*

Throughout this thesis, we will concentrate on the case that $t(\epsilon, n)$ is sublinear in n.

We modified the definition of the approximate PCPS model given in [6] to a slightly more restricted case: Instead of requiring that the prover is restricted to a function determined before the start of the interaction, we require the prover write down the whole proof before the interaction starts. In the next chapter, we will show that this restriction does not limit the ability of the proof system.

The choice of the distance function $\triangle$ is problem specific and determines the ability to construct a proof system, as well as how interesting the proof system is. The traditional definitions of interactive proof (probabilistically checkable proof) systems for decision problems requre that when $y = f(x)$, an honest prover (an correct proof) can convince the verifier of that fact, and when $y \neq f(x)$, no prover can convince the verifier of that. [EKR] suggest that, in the approximate model, this is achieved by choosing $\triangle(\cdot, \cdot)$ such that $\triangle(y, y') > \epsilon$ whenever $y \neq y'$ and $\triangle(y, y) = 0$. Not every problem has an approximate proof system with the sort of distance function, but such a problem does exist, and we will discuss one in Section 4.1.2. In fact, a more

straightforward distance function is $\triangle(y, y') = \max\{0, 1 - y/y'\}$, where the proof system accepts only when $y > (1 - \epsilon)f(x)$.

An approximate proof system is suitable for optimization problems, which are defined as follows.

Definition 2.2.3 *(Optimization Problem [13]) An optimization problem is defined by a tuple (L, S, v, goal) such that I is the set of instances of the problem, S is a function that maps an instance $x \in L$ to all feasible solutions, v is a function that associates a positive integer to all $s \in S(x)$, and goal is either minimization or maximization.*

The value of an optimal solution to an instance x is called OPT(x) and is the minimum or maximum of $\{v(s)|s \in S(x)\}$ based on goal.

But in most cases, the approximate proof system can only give a lower bound to a maximization problem or an upper bound to a minimization problem. We will discuss this in Chapter 4.

2.3 The Definition of Spot Checkers

A spot checker is like a verifier in the approximate PCPS without either interactions with a prover or a proof string. It *checks* the correctness of the result (of some claim, some program supposed to compute some function, etc.) by examine some"*spots*" in the input and the output.

For example, to check whether a given multiset A of size n contains $O(n)$ distinct elements, one might randomly pick m elements from A and check their distinctness by sorting. It is known that $m = \Omega(\sqrt{n})$ [5]. Another example, to check whether a sequence is well-sorted, is given in Section 3.3.

A formal definition of a spot-checker is as follows.

Definition 2.3.1 *(Spot-Checker [5]) Let $\triangle$ be a distance function. We say that C is an ϵ-spot-checker for f with distance function $\triangle$ if the following holds.*

1. *Given any input x and program P (purporting to compute f), and ϵ, C outputs with probability at least 3/4 (over the internal coin tosses of C) PASS if $\triangle$=0 and FAIL if for all inputs y, $\triangle > \epsilon$.*

2. *The runtime of C is $o(|x| + |f(x)|)$.*

We can observe that any f with an ϵ-spot-checker will trivially have an $O(\log|x|)$-approximate PCPS if the checker runs in time $O(\log|x|)$, or a sublinear time approximate PCPS if $|f(x)| = O(|x|)$.

2.4 Definitions of MAXSNP Class

In order to perform reductions on optimization problems properly, Papadimitriou introduced a new reduction as follows.

Definition 2.4.1 *(L-Reduction [13]) Suppose that A and B are optimization problems (maximization or minimization). An L-reduction from A to B is a pair of functions R and S, both computable in logarithmic space, with the following two additional properties: First, if x is an instance of A with optimum cost OPT(x), then R(x) is an instance of B with optimum cost that satisfies*

$$OPT(R(x)) \leq \alpha \cdot OPT(x),$$

where α is a positive constant. Second, if s is any feasible solution of R(x), then S(s) is a feasible solution of x such that

$$|OPT(x) - c(S(s))| \leq \beta \cdot |OPT(R(x)) - c(s)|,$$

where β is another positive constant particular to the reduction (and we use c to denote the cost in both instances). That is, S is guaranteed to return a feasible solution of x which is not much more suboptimal than the given solution of R(x).

To discuss the NP optimization problems systematically, Papadimitriou also defines a new class named MAXSNP.

Definition 2.4.2 *(SNP [13]) Strict NP, or SNP, is defined to be the class of problems with the properties expressible as*

$$\exists S \forall x_1 \forall x_2 \ldots \forall x_k \phi(S, G, x_1, \ldots, x_k),$$

where ϕ is a quantifier-free First-Order expression involving the variables x_i and the structures G (the input) and S.

Definition 2.4.3 *(MAXSNP$_0$ [13]) MAXSNP$_0$ is defined to be the following class of optimization problems: Problem A in this class is defined in terms of the expression*

$$\max_S |\{(x_1, \ldots, x_k) \in X^k : \phi(G_1, \ldots, G_m, S, x_1, ..., x_k)\}|$$

Definition 2.4.4 *(MAXSNP [13]) MAXSNP is the class of all optimization problems that are L-reducible to a problem in MAXSNP$_0$.*

Notice that, by the definition of MAXSNP, a MAXSNP-complete problem must be in MAXSNP$_0$. To see what a MAXSNP$_0$ problem is like, we give an example below.

Two vertices in a graph are said to be independent if there is no edge between them. Given a graph $G = (\Lambda, E)$ of degree at most k, we want to find the largest independent subset of Λ. This problem is known as k-Degree Independent Set. It is not only an NP-complete problem but also a MAXSNP-complete problem. Now we require the input to be given by a set H of $(k+1)$-tuples $(x, y_1, \ldots, y_k)$ such that for any $x \in \Lambda$, the y_i's are the neighbors of node x (with repetitions when x has fewer than k neighbors). $|H| = |\Lambda|$. The k-Degree Independent Set problem can be written as:

$$\max_{S \subseteq \Lambda} \{|x, y_1, \ldots, y_k) : [(x, y_1, \ldots, y_k) \in H] \wedge [x \in S] \wedge [y_1 \notin S] \wedge \ldots \wedge [y_k \notin S]\}|.$$

S is an independent set [13].

And to point out the close relationship between MAXSNP and PCP, Papadimitriou also defines a variation of PCP called MAXPCP.

Definition 2.4.5 *(MAXPCP [13]) Let $k_1, k_2, k_3 > 0$, and f be a polynomial-time computable function. Assign to each string x and each bit string r with $|r| = k_2 \log |x|$ a set f(x,r) of k_1 numbers in the range $1 \ldots |x|^{k_3}$, and let M be a polynomial-time Turing machine with three inputs. Define now this optimization problem: "Given x, find the string y of length $|x|^{k_3}$ that maximizes the number of strings r with $|r| = k_2 \log |x|$ such that $M(x, r, y|_{f(x,r)})$='yes'."*

It can be proved that MAXSNP=MAXPCP [13].

Chapter 3

Basic Algorithms for the Approximate Proof Systems

3.1 Warming Up

In the **One-Center Problem**, given a set of points, the goal is to find a circle containing all of these points such that the radius of the circle is minimized. For any instance x, once a prover P offers an answer y, how can a verifier V make sure that y is a correct, or at least good enough? V can randomly pick some points and checks that the geometric distance between the point and the center is less than the radius. The question is how many should V picks? With a distance function $\triangle$=(*number of points outside the circle*)/(*number of points*), if $\triangle \geq \epsilon$, a point falls outside of the circle with probability at least ϵ. Thus, within $O(1/\epsilon)$ time, V will reject with probability at least 3/4.

V can further decide whether the given circle is minimized. If it *is*, P should be able to find two points on the circle which form an arc of degree π, or three points which form an arc of degree more than π. In this final step, V requires only constant time [10].

Now consider a similar problem, **Convex Hull Problem**. A convex polygon is a polygon with the property that a line segment connecting any two points inside the polygon must itself lie inside the polygon. The convex hull of a set of planar points is defined as the smallest convex polygon containing all of the points. The problem is to find the convex hull given a set of n points.

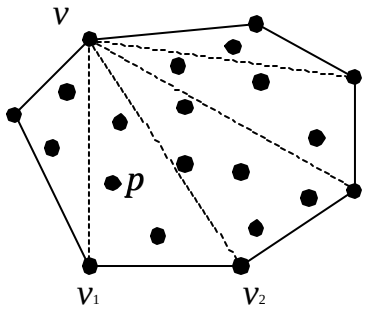

Figure 3.1: Any n-polygon can be devided into $n-2$ triangles.

P gives the answer by an ordered list of a subset of the points. How does V make sure that the answer is correct? First, points in the list with the given ordering must form a convex polygon. Second, all of the points must situate inside the polygon. The later case is easy to check. If V is sure that he got a polygon, since any n-polygon can be devided into $n-2$ triangles, after V picks a point p from the point-set and a vertex v from the answer-list, the prover should be able to find out two consecutive points v_1, v_2 from the list, such that p is in the triangle formed by v, v_1, v_2. If there are at least ϵn points outside the polygon, V will be able to discover one with probability at least $3/4$ within $O(1/\epsilon)$-time.

Now we consider the case to check whether the points in the list form a convex polygon by the given ordering. Let m_l be the length of the list. In most cases m_l is far less than n. V can simply check the points one by one to make sure that they form a convex polygon, if he find out that m_l is less than some value, say, $\log n$. But what if it is not the case?

Since we know that Sorting can be reduced to Convex Hull under the comparison model [10], unless there is a method to check whether a list is sorted in $o(\log n)$ time, checking whether the list forms a convex polygon must take $\Omega(\log n)$ time. We will give an algorithm to this problem later, in Section 3.3.

3.2 Basic Set Operations

In each of the following problems, we first introduce the lower bound algorithm given in [6], then with the same manner we construct an upper bound algorithm. If the claimed answer of P is far from correct, that is, the distance is at least ϵ, V rejects with probability at least $3/4$. If the answer is correct, V accepts with probability 1. For convenience, all these algorithms are presented as an IP protocol followed by a description stating how to modify the algorithm into a PCP version.

3.2.1 Element Distinctness

INSTANCE: An list $X = \{x_1, \ldots, x_n\}$.
QUESTION: Are all the x_i's distinct?

Lower Bound
DISTANCE: $|\{x_i : \exists x_j \text{ s.t. } x_i = x_j \text{ and } i \neq j\}|/|X|$

Algorithm 3.2.1 *(Lower Bound for Distinct Elements)*
Repeat $O(1/\epsilon)$ *times:*

> *V randomly chooses* $i \in [1 \ldots n]$.
> *V sends* x_i *to P.*
> *P returns* j *to V.*
> *V rejects if* $i \neq j$.

If V does not reject, then V accepts.

The PCPS version. P writes a list of ordered pairs containing each input element and its location in the input list (x_i, j) in the order sorted by the value of x_i. V then performs binary search to find (x_i, j) and checks $j = i$. This adds an additional $\log n$ factor to the time.

The above strategy with binary search is suggested in [6]. In fact, the additional $\log n$ factor can be eliminated. Since x_i's can be encoded with a binary string of length at most $O(\log n)$, P can write the answer in an array A of size $O(n^k)$, where the tuple (x_i, j) is represented as $A[x_i] = j$. $A[s] = null$ for any binary encoding s that is not among x_i's. Under a RAM model, V can check the value of $A[x_i]$ in constant time.

Upper Bound
The original question can be midified as follows:
QUESTION: The size of $Y = \{x_i | \exists x_j, x_i = x_j \in X \text{ and } i \neq j\}$.
DISTANCE: 0 if $|Y| > p$ and $(p - |Y|)/p$ otherwise, where p is P's claimed size of Y.

Algorithm 3.2.2 *(Upper Bound for Distinct Elements)*
Performs the set size lower bound algorithm (Algorithm 3.2.3) to make sure that $|Y|$ is large enough.

3.2.2 Size of a Set

INSTANCE: A set S.
QUESTION: The size of S.

Lower Bound
DISTANCE: 0 if $|S| > p$ and $(p - |S|)/p$ otherwise, where p is P's claimed size of S.

Algorithm 3.2.3 *(Lower Bound for Set Size)*
P sends p to V.
P writes the elements of S to an array A of size p.
Perform element distinctness lower bound algorithm on A with parameter $\epsilon/2$. Repeat $O(1/\epsilon)$ times:

> *V randomly chooses an $i \in [1 \ldots n]$ and sends it to P.*
> *P sends V a proof $A[i] \in S$.*

The PCPS version. It's easy to see that P can write down all the answers to all possible queries of V. After V chooses an i, what he has to do is to check out the "answer entry" from the "answer table" according to i.

The lower bound of $|X \cup Y|$ can also be estimated with algorithm 3.2.3. The algorithm used in the One-Center problem is in a similar manner.

Upper Bound
DISTANCE: 0 if $|S| < p$ and $(|S| - p)/|S|$ otherwise, where p is P's claimed size of S.

Algorithm 3.2.4 *(Upper Bound for Set Size)*
P sends p to V.
P writes the elements of S to an array A of size p.
Repeat $O(1/\epsilon)$ *times:*

> *V randomly picks an element* $s \in S$*, and sends it to P.*
> *P sends V an i.*
> *V rejects if* $s \neq A[i]$.

If V does not reject, then V accepts.

Notice that since p is an upper bound, the only thing V should check is whether all elements in S are in A; thus the element distinctness protocol does not need to be performed.

The PCPS version. P writes a list of ordered pairs (x, y), where x is an element of S and y is the evidence that x is in S. All the pairs are sorted by the value of x. V randomly picks an element s from S and performs binary search to find (s, y_s) and checks the validity of y_s. V rejects whenever s is not found in the ordered list or y_s is not a proper piece of evidence. This adds an additional $\log n$ factor to the time requirement. In fact, the $\log n$ factor can be eliminated. See the discussion of Section 3.2.1.

Here are two applications. To estimate the upper bound of $|X \cup Y| = |X| + |Y| - |X \cap Y|$, use algorithm 3.2.4 to estimate $|X| + |Y|$ and algorithm 3.2.6 to estimate $|X \cap Y|$.

3.2.3 Set Equality

INSTANCE: Given two lists $X =< x_1, \ldots, x_n >, Y =< y_1, \ldots, y_n >$.
QUESTION: Is set X equal to set Y?

To make sure that X and Y are two sets of size n, we can perform algorithm 3.2.1 and 3.2.3 first.

Lower Bound
DISTANCE: 0 if X=Y and $|X - Y|/n$ otherwise.

Algorithm 3.2.5 *(Lower Bound for Set Equality)*
P writes an arrays T of length n, where T[i] contains a pointer to the location of x_i *in Y. V randomly picks an i and checks that whether* $x_i = y_{T[i]}$.

The PCPS version. This algorithm is also of approximate PCPS form as well.

Upper Bound
QUESTION: Is $|X \cap Y|$ at most ρn?
DISTANCE: $\max\{(|X \cap Y|/n) - \rho, 0\}$.

Algorithm 3.2.6 [1] *(Upper Bound for Set Union)*
V randomly chooses a set $S \in \{X, Y\}$ and then randomly picks an element from S, and asks P the location of this element. P has to answer which set it is from and its location in this set. V rejects if P's answer is inconsistant with what V knows. Repeat the above steps $O(1/\epsilon\rho)$ times.

The PCPS version. P writes a list T of size $|X \cup Y|$. Each entry of T contains $< a, (b1, b2) >$, where a is an element of $X \cup Y$, $b1 \in \{X, Y\}$ stands for which set a comes from and $b2$ stands for the location of a in set $b1$. All the pairs are sorted by the value of a. After V picks an element, he performs a binary search to find the proper entry and chekcs the correctness of $< b1, b2 >$. This adds an additional $\log n$ factor to the time requirement. The additional $\log n$ factor can in fact be eliminated. See the discussion of Section 3.2.1.

3.3 Checking a Sorted List

3.3.1 The Algorithm

Given a seuqence x of length n, V wants to make sure that x contains a long increasing subsequence. He performs the following algorithm:

Algorithm 3.3.1 *(Check Sorted List)*
Repeat $O((1/\epsilon)\lg 1/\delta)$ times:

> *Randomly choose $i \in$ [1,n].*
> *for $k \leftarrow 0 \ldots \lg i$ do*
>
> > *Repeat $O(\lg 1/\delta)$ times:*
> >
> > > *Randomly choose $j \in [1, 2^k]$.*
> > > *If ($A[i-j] > A[i]$), then reject.*

[1] Modified from the approximate IPS for Two Set Intersection in [6].

for $k \leftarrow 0 \ldots \lg(n-i)$ *do*

Repeat $O(\lg 1/\delta)$ *times:*

Randomly choose $j \in [1, 2^k]$.
If $(A[i] > A[i+j])$, *then reject.*

Accept.

This Procedure runs in $((1/\epsilon) \lg n \lg^2 1/\delta)$ time and satisfies:

- If x is sorted, it accepts.
- If x does not have an increasing subsequence of length at least $(1-\epsilon)n$, then with probability at least 1-δ, it rejects.

That is, once x passes the test, with high probability it has an increasing subsequence of length at least $(1-\epsilon)n$. Here the distance function used is as follows. Let the number of the longest increasing subsequence be n'. Then $\triangle = (n-n')/n$. To see why the algorithm works correctly, we briefly describe the proof in [5]. First, transform the problem into one in graph theory.

Definition 3.3.1 *The graph induced by an array x of intergers having n elements is the directed graph G_x, where $V(G_x) = v_1, \ldots, v_n$ and $E(G_x) = \{(v_i, v_j) : i < j$ and $x[i] < x[j]\}$.*

We can observe that the graph G_x is transitive, i.e., if $(u, v) \in E(G_x)$ and $(v, w) \in E(G_x)$, then $(u, w) \in E(G_x)$. Now we use $\Gamma^+_{(t,t')}(i)$ to denote the set of vertices in the open interval between t and t' that have an incoming edge from v_i. Similarly, $\Gamma^-_{(t,t')}(i)$ denotes the set of vertices between t and t' that have an outgoing edge to v_i.

Definition 3.3.2 *A vertex v_i in the graph G_x is said to be heavy if for all k, $0 \leq k \leq \lg i$, $|\Gamma^-_{(i-2^k,i)}(i)| \geq \eta 2^k$, and for all k, $0 \leq k \leq \lg(n-i)$, $|\Gamma^+_{(i,i+2^k)}(i)| \geq \eta 2^k$, where $\eta = 3/4$.*

With the pigeonhole principle, one can show that if v_i and v_j $(i < j)$ are heavy vertices in the graph G_x, then $(v_i, v_j) \in E(G_x)$. It follows that a graph G_x with $(1-\epsilon)n$ heavy vertices has a path of length at least $(1-\epsilon)n$. Since G_x is induced by x, by definition the length of the increasing subsequence of x is at least $(1-\epsilon)n$.

By this method, one can show that not only there is a long increasing subsequence, but also the points checked belong to this sequence.

3.3.2 Examples

A Lower Bound for the Common Subsequence. Given p strings $I_1, \ldots, I_p$, each of length n, the problem is to find out a common subsequence as long as possible. Now P claims that there is a common subsequence with length at least l. P and V perform the following algorithm.

Algorithm 3.3.2 *(Lower Bound for the Common Subsequence)*

1. *P writes a string s and p arrays $S_1, \ldots, S_p$, each of length l. $S_i[j] = k$ stands for that $s[j]$ appears in the k^{th} element of the i^{th} input string.*

2. *V performs algorithm 3.3.1, and in each of the $O(1/\epsilon)$ repetitions, right after randomly choosing an i, adds the following steps:*

   for $k \leftarrow 1 \ldots p$
   Rejects if $I_k[S_k[i]] \neq s[i]$.

Now we give a proof to the problem left in Section 3.1.

Checking the Convexity of a Polygon. For a convex polygon, we have the following observations (see Fig. 3.2):

1. Define the value of any point v_i in the list to be the direction of angle (v_{i-1}, v_i, v_{i+1}). If the points in the given order do form a convex polygon, then the list is well-ordered, since any concave angle will cause a misordering.

2. If the vertices on a subsequence of the list form a convex polygon, then in this subsequence, for any vertex i, all other vertices will fall in angle (v_{i-1}, v_i, v_{i+1}).

3. If any two angles (v_1, v_2, v_3) and (v'_1, v'_2, v'_3) are well-ordered in the list, v_1, v_2, v_3 all fall in angle (v'_1, v'_2, v'_3) and vise versa. Since the lines spanning out from the two angles either form a convex angle or do not cross to each other, (v_2, v_3, v'_1, v'_2) and (v'_2, v'_3, v_1, v_2) are convex shaped.

Thus, V can check whether all the vertices selected satisfy

1. They are in a long well-ordered subsequence of the list (by Algorithm 3.3.1).

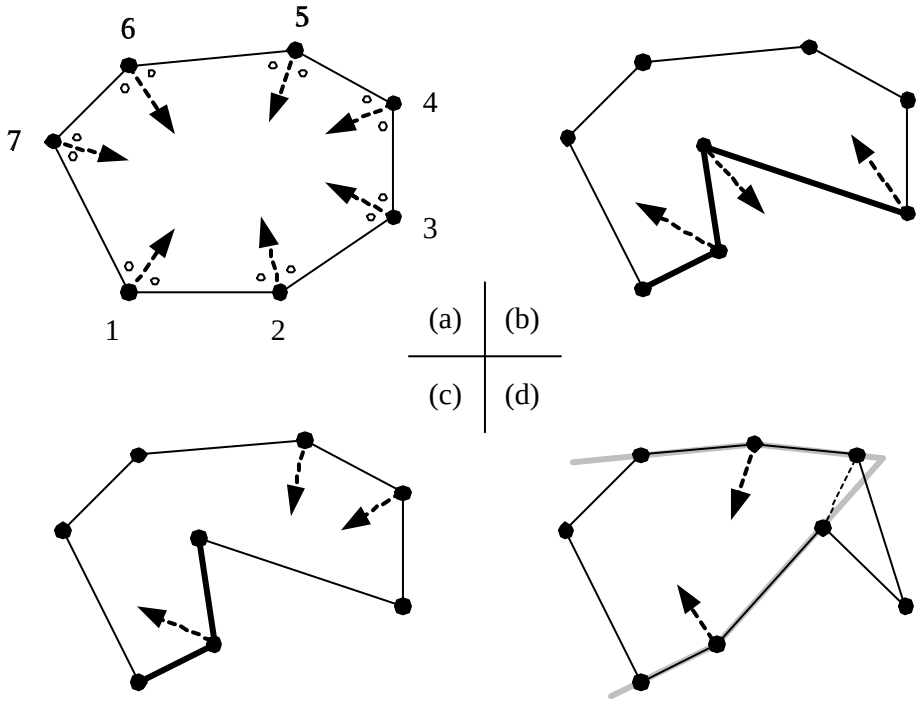

Figure 3.2: (a) Well-ordered vertices form a convex polygon. (b) Any concave angle will cause a misordered subsequence. (c) If a concave angle exists, even the vertices are well-ordered respectively, some points would fall outside a convex angle. (d) All the vertices fall inside the angles can form a convex polygon.

2. For any angle (v_{i-1}, v_i, v_{i+1}), no vertices fall out of it.

Since $O(1/\epsilon)$ vertices are checked, $\max\{O(1/\epsilon^2), O((\log m_l)/\epsilon)\}$ time is needed. If there are at least $O(\epsilon)$ vertices in the list breaking the rule, with high probability V will pick one out.

Chapter 4

Approximate Proof Systems on Various Problems

4.1 On NP-Complete Problems

4.1.1 For Some NP-Complete Problems

MAX3SAT. Given a boolean expression ϕ of m clauses, each of the clauses consists of 3 literals. Is there a truth assignment such that the number of satisfied clauses is at least k?

Algorithm 4.1.1 *(MAX3SAT)*
P writes a truth assignment and an array A of length m, where $A[i]$ indicates the literal that makes clause i satisfied. V randomly picks $O(1/\epsilon)$ clauses, rejects as long as one of them is not satisfied, and accepts otherwise.

If the number of satisfied clauses is less than $(1-\epsilon)m$, V is likely to reject.

k-Degree Independent Set. Given a graph $G = (\Lambda, E)$ with degree at most k. Is there a set $I \subseteq \Lambda$ of size at least K, such that whenever $i, j \in I$ there is no edge between i and j? In this slightly complicated problem, we require a special input structure to facilitate the proof.

Input Structure: An array Λ to represent the list of vertices. For all vertices v_i, $\Lambda[i]$ contains a k-tuple, which represent all its neighbors, duplicated if the degree of some vertex is less than k. If the input is not given in this structure, P should build it up first, and runs the set equivelent algorithm to convince V.

Algorithm 4.1.2 *(k-Degree Independent Set)*

1. *P writes an array A of length K such that $\Lambda[A[i]] \in I$ for all i, and an array Λ', where $\Lambda'[i] = 1$ if $\Lambda[i]$ is in I and 0 otherwise.*

2. *P and V perform the element distinctness with parameter $\epsilon/2$. As for the validity for each A[i], V has to check that each element in A[i] is distinct and $\Lambda'[A[i]] = 1$.*

3. *V randomly picks $O(1/\epsilon)$ elements from A. For any picked element i, let the k-tuple of $\Lambda[A[i]]$ be $(i_1, \ldots, i_k)$. V then checks that for each i_j, $\Lambda'[i_j] = 0$.*

If the size of the clique is less than $(1-\epsilon)k$, V is likely to reject.

One might be interested in expanding the algorithm of k-Degree Independent Set problem to the general Independent Set problem. Before we explore the Independent Set problem, we shall first try a closely related problem, MAX CLIQUE.

MAX CLIQUE. Given a graph $G = (\Lambda, E)$, find a set $\Lambda' \subseteq \Lambda$ such that the G' induced by Λ' from G is a clique and is as large as possible. To give a lower bound k of $|\Lambda'|$, P and V runs the following algorithm.

Algorithm 4.1.3 *(MAX CLIQUE)*

1. *P writes a list Λ' of size k and an array A of length $|\Lambda|$. If $\Lambda[i] \in \Lambda'$, then A[i]=1 and A[i]=0 otherwise.*

2. *P and V perform Algorithm 3.2.3 (to prove the lower bound of size of a set) to make sure that $|\Lambda'|$ is large enough.*

3. *V randomly picks $O(1/\epsilon)$ vertices from Λ', rejects if for any vertex v picked, $\exists v' \in \Lambda'$ such that $(v, v') \notin E$. This can be done easily if G is given by an adjacency matrix, or if P carefully writes down all the proof that $(v_1, v_2) \in E$ for any $v_1, v_2 \in \Lambda'$.*

The above algorithm needs $O(k/\epsilon)$ time, that is, $O(\epsilon|E|^{1/2})$ time, since $k = O(|E|^{1/2})$. But if we apply this method to the Independent Set problem, it fails to be a sublinear time algorithm. We argue as follows. If there is a method in time $o(k)$, let it be $O(k^c)$ for some $c < 1$. Choose $\epsilon =$

$1/(k+1)$. With $O(k^{c+1})$ time, V can verify that G indeed has a clique of size k, with positive and negetive errors both bounded in $1/4$. Notice that, within $o(k^2)$ time, V cannot read in all the structure of the clique. Thus, unless there is some method other than to check "physically", it seems there is no better algorithm to verify the existence of a k-clique. We conclude that there is unlikely to be a straightforward method to give a lower bound for the Independent Set problem with a sublinear-time approximate PCPS.

4.1.2 Another Way To Prove NP=PCP$(\log n, 1)$

The fact that MAX3SAT has an approximate PCPS reveals the close relationship between the approximate proof systems and the formal PCP. Consider the following problem.

ROBE3SAT [9]. Given a boolean expression ϕ in CNF, with 3 literals per clause, ϕ is either satisfiable or at least an ϵ-fraction of the clauses of ϕ are not satisfiable. Is ϕ satisfiable?

It has been proved that there exists a constant ϵ such that this problem is NP-complete [9]. That is, for any instance x in NP, there is a logarithmic space Turing machine that reduces it to ROBE3SAT. Notice that ROBE3SAT is a very special case of MAX3SAT: V either accepts because ϕ is satisfied, or rejects because ϕ is not satisfied (thus more then an ϵ-fraction of clauses are not satisfied), with high probability. Removing the requirement that V runs in a RAM model does not affect the fact that V has to check only $O(\log m)$ bits of the proof. This is consistent with the famous NP=PCP$(\log n, 1)$ theorem.

Notice that this result does not imply that every problem in NP has an sublinear-time approximate PCPS, since the reduction takes polynomial time, exceeding the limitation of sublinear time.

4.2 On MAXSNP$_0$ Problems

In the former section, we saw two MAXSNP-complete problems, MAX3SAT and k-Degree Independent Set problem, with proofs checkable by sublinear-time approximate PCPS. Now we want to discuss the relationship between a sublinear-time approximate PCPS and MAXSNP problems.

In Section 2.4, we know that MAXSNP-complete problems must be in MAXSNP_0. A MAXSNP_0 problem is to find a relation S to maximize $M = |\{(x_1, \ldots, x_k) \in X^k : \phi(G_1, \ldots, G_m, S, x_1, \ldots, x_k)\}|$. If we require a special input structuer of G_i's, then we can construct an approximate PCPS for every MAXSNP_0 problem that gives a lower bound for M as follows.

Input Structure. For any problem with k-ary input relations $G_0(x_1, \ldots, x_k)$, $\ldots$, $G_i(x_1, \ldots, x_k)$, assume that the size of V is n and each $|x_i| = O(\log n)$. Now we require the input relation G_i's not to be given as a logic expresson, but a list of the whole mapping $V^k \to \{0, 1\}$. The list is at most $poly(n)$ long.

The Distance. For any relation S, let M_S be the real value of $|\{(x_1, \ldots, x_k) \in V^k : \phi(G_1, \ldots, G_m, S, x_1, ..., x_k)\}|$ and let the claimed value be m_S. Then $\triangle = \max\{(m_S - M_S)/m_S, 0\}$.

Algorithm 4.2.1 *($MAXSNP_0$)*
P claims the lower bound for M to be m, and writes a feasible relation S, with the structure similar to the input relations. Then P writes all the k-tuple $(x_1, \ldots, x_k)$ satisfying ϕ to an array of length m. V randomly picks $O(1/\epsilon)$ tuples, rejects if any one of them cannot satisfy ϕ, and accepts otherwise.

Notice that for any specific problem, if we assume that each of the relations takes constant length in ϕ, then ϕ is of constant length. And because of our requirement of the special input structure, it is possible for V to check in constant time that each relation in ϕ takes a proper value. Thus the overall value of ϕ can be checked in constant time.

There is no obvious way to prove that any problem with an approximate PCPS has a MAXSNP form. But recall that MAXPCP=MAXSNP. It is interesting to show that all the algorithms used so far can be expressed as a MAXPCP problem.

Recall that the MAXPCP problem is, given x, to find a string y of length $|x|^k$ that maximizes the number of strings r with $|r| = k_2 \log |x|$ such that $M(x, r, y|_{f(x,r)})$=yes. In approximate PCPS, P writes a proof string to V, and V randomly picks some parts out of it to check validity. V runs in time $O(\log |x|)$; thus the random coins he tosses can not exceed $O(\log |x|)$, and he himself of course runs in polynomial time. To convince V, the proof string given by P should make V accept under most of the cases.

Almost all the basic algorithms can be transformed into a MAXPCP version trivially, except one: To check the ordered list. To prove a list contains

a long well-ordered subsequence, Algorithm 3.3.1 reads $O(\log n)$ bits from the proof string. But since we do not require the MAXPCP model to run in sublinear time, the solution is simple. One can check the validity by making sure that all pairs of the input elements are valid. V can randomly choose enough number of pairs of elements from the list L, rejects if there exits a pair $(L[i], L[j])$, where $i < j$ such that $L[i] > L[j]$. There are $\binom{n}{2}$ pairs, and at least ϵn pairs will cause the rejection if there are at least ϵn elements breaking the rule.

Now we can express the problem with a proof checkable by an approximate PCPS, which is constructed with the algorithms we introduced, as follows. For any instance x, find a string y such that $V(x, r, y|_{f(x,r)}) = yes$ for at least some required amount of r's. The required amount is problem dependent. For example, in the problem of checking element distinctness, it is $\binom{n}{2} - \epsilon n$. Notice that, even if for all r's V answers yes, it does not nessesarily mean V gets an exact answer. See the following example.

Approximate lower bounds on sums [6]. Given positive integers $x_1, \ldots, x_n$, P can convince V of a good approximation to a lower bound on $\sum_{i=1}^{n} x_i$. Consider the set $S = \{(i, j) | 1 \leq i \leq n, 1 \leq j \leq x_i\}$ whose cardinality is $\sum_{i=1}^{n} x_i$. Given (i, j), V can determin membership in S in constant time, by verifying that $1 \leq i \leq n$ and $1 \leq j \leq x_i$. Thus, P and V can perform Algorithm 3.2.3 to estimate the lower bound of $|S|$. To bound the error in ϵ, it is enough to work with only the $O(1/\epsilon)$ most significant bits of the weights.

When we transform the above proof system for some fixed ϵ to a MAXPCP one, assume that s is the claimed lower bound of $|S|$. If $|S| \leq (1 - \epsilon)s$, for at least a fraction ϵ of r's, V will reject. On the other hand, if V accepts for all r's, because only $O(1/\epsilon)$ most significant bits are used, it is not enough for V to be sure that he gets a exact answer.

4.3 On coNP Problems

After providing the lower bounds of some NP optimization problems, it is reasonable for us to ask: Is there a simple method for V to make sure that if it is a feasible solution, it is also an optimal one, just as what V can do in the One-Center problem?

If such a method does exist, say, in time $O(\log n)$, consider the following UP problem (we take a MAXSNP problem as an example): "Given a goal

K, is $\max_S |(x_1, ..., x_k) \in X^k : \phi(G_1, \ldots, G_m, S, x_1, \ldots, x_k)|$ less than the goal K?"

If such a method does exist, for a NP Turing machine, it can of course find out that under some relation S, $M = |\{(x_1, ..., x_k) \in X^k : \phi(G_1, \ldots, G_m, S, x_1, \ldots, x_k)\}|$. Then it performs the $O(\log n)$ method to make sure that M is optimized. If it is, compare M with K. If $M < K$, it accepts. For all other cases it rejects. Recall that an SNP problem is of the form $\exists S \forall x_1 \forall x_2 \ldots \forall x_k \phi(S, G, x_1, \ldots, x_k)$. Thus, if any MAXSNP version of some SNP problem has this property, then the corresponding coSNP problem, which states that there is no relation satisfying $\forall x_1 \forall x_2 \ldots \forall x_k \phi$, will also be in UP: just take $K = 2^k$. So we should not respect an NP-complete problem in MAXSNP_0 to have such a property.

Another trial is to give both the upper and the lower bound, just as what we can do when building the basic algorithms in Section 3.2. For example, for the MAX3SAT problem, we know it is easy to check the correctness of the lower bound of satisfied clauses. Now we ask whether there is an approximate IPS which proves that the number of clauses simultaneously satisfied is at most K? For convenience, we will discuss MAXSNP problems only.

Recall that many NP complete problems are in MAXSNP_0 to ask whether there exists a relation such that the number of k-tuples satisfying ϕ achieves a goal K. Now we want to ask that whether the number of k-tuples will not exceed K for any relation. This, again, is a coNP question. In the example of MAX3SAT, if one can prove with some K less than the number of total clauses, he will be able to answer that such a boolean expression is unsatisfiable.

Notice that we don't think there is an approximate PCPS for coNP problems, since this will imply that coNP is in NP. Rather, it is known that coNP is in IP, so what we want to know is whether it is possible to reduce the time V needs in the IP system.

By the definition of approximate IPS, assume that P writes a proof of length $poly(|x|)$ before the conversation with V, and V runs in $O(\log |x|)$ time to finish verification. That is, there exists some string z with length $poly(|x|)$ such that there exists an IP system to prove that x is not in L within time $O(\log |x|)$. In [8] we have the following theorem:

Theorem 4.3.1 *Let $c(\cdot)$ be an integer function and $L \subseteq \{0,1\}^*$. Suppose that L has an interactive proof system in which both the randomness and communication complexities are bounded by $c(\cdot)$, then $L \in Dtime(2^{O(c(\cdot))} poly(\cdot))$.*

Here we have $c(\cdot) = k \log |x|$, thus the time will be in Dtime($poly(|x|)$). But there might be some trouble since what we have is not a standard IP system. This is because when $0 < \triangle < \epsilon$, the verifier's answer is ambiguous. But we will see that it doesn't matter with the polynomial time algorithm constructed by the method in [8].

Definition 4.3.1 *(The Game Tree and Its Value [8]). For some fixed V and x:*

- *The tree T_x: The nodes in the tree, denoted T_x, correspond to possible prefixes of the interaction of V with an arbitrary prover. The root represents the empty interaction and is defined to be at level 0. For every i=0,1,... the edges going out from each $2i_{th}$ level node correspond to the messages V may send given the history so far. The edges going out from each $(2i+1)^{st}$ level node correspond to the messages a prover may send given the history so far. Nodes which correspond to an execution on which V stops have as children one or more leaves, each corresponding to a possible V's random-pad which is consistent with the interaction represented in the father. Thus, leaves correspond to augmented transcripts as defined above.*

- *The value of T_x: The value of the tree is defined bottom-up as follows. The value of a leaf is eighter 0 or 1 depending on whether V accepts in the augmented transcript represented by it or not. The value of an internal node at level 2i is defined as the weighted average of the values of its children, where the weights correspond to the probabilities of the various verifier messages. The value of an internal node at level 2i-1 is defined as the maximum of the values of its children. This corresponds to the prover's strategy of trying to maximize V's accepting probability. The value of the tree is defined as the value of its root.*

To decide if x is in the language accepted by V, it suffices to approximate the value of the tree T_x defined above. Since the communication is only $O(\log(|x|))$ bits, T_x has at most $poly(|x|)$ nodes. Thus, we can construct T_x for any x and compute the value of each of its nodes in time $poly(|x|)$. By the definition of approximate IPS, when $\triangle = 0$, V accepts with probability at least 3/4, thus the value of T_x is at least 3/4, and when $\triangle > \epsilon$, the value of T_x is at most 1/4. But notice that we assume that randomly picking

some content from a memory location takes constant time; in contrast in the IP system, to send a address consumes $O(\log |x|)$ bits. If V needs some interaction like this, it must not take more than constant rounds.

Now we can construct a deterministic polynomial time algorithm A such that

1. If $M = \max_S |(x_1, ..., x_k) \in V^k : \phi(G_1, \ldots, G_m, S, x_1, \ldots, x_k)|$ is less than K, there is some (proof) string z of lengh $|x|^k$ such that $A(x, z) = yes$.

2. if $M \geq (1 + \epsilon)K$, for any string z', $A(x, z') = no$.

Equivalently, there will be an NP algorithm to answer whether x is far from being in L.

Furthermore, if for some coNP-complete problem, we require that, let $p \geq 3/4$ be the possibility that V accepts when $\triangle = 0$, no proof can make V accept with probability more than $p - 1/poly(|x|)$ when $x \notin L$, just as when proving the lower bound of MAX3SAT, V answers a false "PASS" with probability at most $1 - 1/m$. With a counter in logarithmic space, one can tell if the distance is 0 by considering all possible coin-tosses, each round in $O(\log |x|)$, thus logarithmic space. We do not have to require that all the coNP problems have such a good property; that one coNP-complete problem has a proof of such a sort is enough to force the whole coNP class to be in NP.

4.4 Permanent: a #P Problem

The permanent of an $n \times n$ matrix $A = (a_{i,j} : 0 \leq i, j \leq n - 1)$ is defined by

$$perm(A) = \sum_{\pi} \prod_{i=0}^{n-1} a_{i,\pi(i)},$$

where the sum is over all permutations π of $n = \{0, \ldots, n - 1\}$. It is a #P-complete problem. In this section, we only consider 0,1-matrices. To compute the exact value of perm(A), there are no known algorithms within time less than $O(n2^n)$. But there do exist some probabilistic algorithms to approximate the value of perm(A) within time $o(2^n)$. In 1996, Jerrum *et al.* suggest an algorithm within time $O(exp(O(n^{1/2} \log^2 n)))$ [11]. Karmarkar

et al. give an Monte-Carlo algorithm, which runs in $poly(n)2^{n/2}$ time [12]. Cai *et al.* show that to give a probabilistic polynomial time algorithm for perm(A) is rather difficult [4]. Now we shall show that to check the lower bound is much easier, by giving a approximate PCPS for it, though it takes a tremendously long proof.

Algorithm 4.4.1 *(Lower Bound for Permutation)*
Let m be the claimed value of perm(A).

1. *P writes an array of length m, each entry containing an n-tuple $(i_1, \ldots, i_n)$, which is a permutation of $(1, \ldots, n)$. In a feasible n-tuple, for any j^{th} entry i_j, $A_{j,i_j} \neq 0$.*

2. *P and V perform Algorithm 3.2.3 to prove the lower bound for the number of feasible n-tuples.*

By our assumption, to randomly pick a tuple written by P takes constant time. Thus to verify the lower bound takes $O(n/\epsilon)$ time. If we take addressing time in consideration, it takes $O((n \log n)/\epsilon)$ time. It seems uninteresting since we are concerned only with sublinear time algorithms. But recall that an $n \times n$ matrix has n^2 entries; thus the input size is $N = n^2$, and the running time turns to be $O((N^{1/2} \log N)/\epsilon)$.

One might be interested in comparing the accuracy of these algorithms. Let a be the real value of perm(A) and $\hat{a}$ be the estimate one. Once P's proof passes V's verification, V believes that $a \geq \hat{a}(1-\epsilon)$ and $n!-a \geq (n!-\hat{a})(1-\epsilon)$, that is, $\hat{a}(1-\epsilon) \leq a \leq \hat{a}+\epsilon(n!-\hat{a})$. In [11] we have $a/(1+\epsilon_1) \leq \hat{a} \leq a(1+\epsilon_1)$, that is, $\hat{a}/(1+\epsilon_1) \leq a \leq \hat{a}(1+\epsilon_1)$. In [12] we have $(1-\epsilon_2)a \leq \hat{a} \leq (1+\epsilon_2)a$, that is, $\hat{a}/(1+\epsilon_2) \leq a \leq \hat{a}/(1-\epsilon_2)$. The bound in [11] is tighter when $\epsilon_1 = \epsilon_2$. Taking $\epsilon = \epsilon_1/(1+\epsilon_1)$ will cause the same lower bound.

Now consider the upper bound case. A straightforward algorithm modified from the above algorithm is that P writes $(n! - m)$ $(n+1)$-tuples, each tuple $(i_1, \ldots, l_n, k)$ with an additional value k which stands for $A_{k,i_k} = 0$. When $\hat{a} > n!/2$, the upper bound seems to be better than [11]. Unfortunately, it takes too much time for V to verify the value $(n! - m)$, since $n!$ cannot be computed in $o(n)$.

Chapter 5

Conclusions

In this thesis, we explored the approximate proof system in various problems, including polynomial time solvable problems, NP-complete (MAXSNP-complete) problems, and the Permanent problem which is not known to be in NP. We restricted our attention to those problems with proofs that can be checked with sublinear time. It is possible that some NP problems do not have such kind of proofs with proper distance functions. The fact that some NP-complete problems have such a proof does not help to eliminate the possibility. Because of lack of a feasible reduction under the sublinear-time approximate proof system, it is not easy to capture the concrete scope of the languages with such a proof system. Thus we discussed this issue case by case. We also discussed its limitation by two special cases and conclude that coNP problems (beyond NP) do not seem to have such kind of proofs. Once a reasonable reduction is defined, it will be helpful to find the relationship between various kinds of combinatorial problems.

Bibliography

[1] Sanjeev Arora, Carsten Lund, Rajeev Motwan, Madhu Sudan, and Mario Szegedy. Proof verification and hardness of approximation Problems. *33rd Annual Symposium on Foundations of Computer Science*, pages 14–23, 1992.

[2] S. Arora and S. Safra. Probabilistic checking of proofs: a new characterization of NP. *33rd Annual Symposium on Foundations of Computer Science*, pages 2–13, 1992.

[3] L. Babai, L. Fortnow, C. Lund, and M. Szegedy. Checking computations in polylogarithmic time. *Proceedings of the Twenty Third Annual ACM Symposium on Theory of Computing*, pages 21–31, 1991.

[4] Jin-Yi Cai, A. Pavan, and D. Sivakumar. On the hardness of permanent. *16th Annual Symposium on Theoretical Aspects of Computer Science*, Springer LNCS 1563:90–99, 1999.

[5] Funda Ergün, Sampath Kannan, S Ravi Kumar, Ronitt Rubinfeld, and Mahesh Viswanathan. Spot-Checkers. *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing*, pages 259–268, 1998.

[6] Funda Ergün, Ravi Kumar and Ronitt Rubinfeld. Fast approximate PCPs. *Proceedings of the Thirty-First Annual ACM Symposium on the Theory of Computing*, pages 41–50, 1999.

[7] Lance J. Fortnow. *Complexity-theoretic aspects of interactive proof systems*. PhD thesis, Massachusetts Institute of Technology, May 1989. (http://www.neci.nj.nec.com/homepages/fortnow/)

[8] Oded Goldreich and Johan Håstad. On the complexity of interactive proofs with bounded communication. *SIAM Journal on Computing*, 67(4): 205–214, 1998.

[9] Stefan Hougardy. Proof checking and non-approximability. *Lectures on proof verification and approximation algorithms*, Springer LNCS 1367:63–82, 1998.

[10] R. C. T. Lee, R. C. Chang, S. S. Tseng, and Y. T. Tsai. *Introduction to the design and analysis of algorithms.* Unalis corporation, 1999.

[11] M. Jerrum and U. Vazirani. A mildly exponential approximation algorithm for the permanent. *Algorithmica*, 16(4/5):392–401, 1996.

[12] N. Karmarkar, R. Karp, R. Lipton, L. Lovász, and M. Luby. A Monte-Carlo algorithm for estimating the permanent. *SIAM Journal on Computing*, 22(2):284–293, 1993.

[13] Christos M. Papadimitriou. *Computational complexity.* Addison-Wesley Publishing Company, Inc., 1994.

[14] A. Shamir. IP=PSPACE. *31st Annual Symposium on Foundations of Computer Science*, vol. I, pages 11–15, 1990.

[15] D. A. Spielman. Linear-time encodable and decodable error-correcting codes. *Proceedings of the Twenty-Seventh Annual ACM Symposium on the Theory of Computing*, pages 388-397, 1995.