Average Case Complexity Theory

Sriram Venkatesh

February 24, 2024

Abstract

Average case complexity is a branch of computational complexity theory that is focused on analyzing the average resource usage of computational problems over the distribution of inputs. In this research paper, we focus on Leonid Levin's theory of average case complexity. This paper will investigate average case problems as distributional problems and attempt to produce an analogue of results in worst case complexity theory in the average case.

This paper is rooted in the author's exploration and exposition of the ideas presented in [9], [3], and [5].

1 Introduction

1.1 History

The advancement of human civilization can be attributed to our ability to solve problems efficiently. The dawn of the new age has come with newer kinds of problems, and we have increasingly turned to machines to help us solve these. Computational complexity theory is focused on deciding which problems are easy for a computer, and which problems are difficult. Algorithms for solving computational problems come in varying degrees of complexity, and the Turing machine, created by Alan Turing [11], gives us an abstract way of generically representing algorithms that solve these problems. It was from the creation of this model of computation that computational complexity theory was born.

The first significant study of computational complexity theory came from Juris Hartmanis and Richard Stearns in their famous 1965 paper [4], where they introduced the definitions of time and space complexity. Jack Edmonds later explained in his paper [2] what it means to have an efficient algorithm, which is one that performs a number of operations polynomial in the size of the input.

Complexity theory began to gain much more traction with the independent contributions of Stephen Cook [1] and Leonid Levin [8]. They introduced the classes P and NP, and established the notions of polynomial time reductions. They proved that the boolean satisfiability problem(SAT) is NP complete, meaning that every other NP problem has a polynomial time reduction to SAT. They also introduced the P vs NP problem for the first time. One of the many interpretations of this problem is whether every computational problem that can

be verified in polynomial time (class NP) can also be solved in polynomial time (the class P). Following this, Richard Karp published his paper [7] which outlined 21 different NP complete problems. Solving, or proving it is impossible to solve any one of these problems in polynomial time would finally lay the P vs NP problem to rest. There are many practical implications to this problem. If $P = NP$, then every NP problem has some hidden trick that allows it to be solved efficiently. This could mean that we can come up with efficient algorithms to crack cryptographic ciphers. If $P \neq NP$, then we would know that computers are fundamentally limited in their problem solving ability.

1.2 Average Case Complexity

The traditional notion of computational complexity theory deals with the worst case analysis, which is the running time of an algorithm over all inputs of the same size. In most real life scenarios, a theory of average case complexity may be more useful. Consider the algorithm quicksort, which performs in worst case $O(n^2)$ but average case $O(n \log n)$. Many applications continue to use quicksort as the default sorting algorithm because it is slightly more efficient than other sorting algorithms on the average. We are also concerned with average case complexity in cryptographic situations where in order for a scheme to be feasible the algorithm that solves it cannot be efficient on average.

The definitions for worst case complexity do not easily extend to that of the average case, so a new set of definitions was created. Leonid Levin's paper [9] on the theory of average case complexity was only 2 pages which built up the theory from the definitions. However, Levin was very reluctant to provide any motivation behind his definitions and would simply state that his definitions were the natural ones. This paper attempts to present this theory in an understandable manner for the reader. The paper will prove the proposition that the Bounded Halting Problem is DistNP complete and will cover the following topics.

1. Standard definitions of worst case complexity theory
2. Average case problems as distributional problems
3. Reductions in the average case
4. DistNP complete problems
5. Current State of Research

2 Preliminaries

In this section we present some basic definitions of the theory of computation before discussing average case complexity theory.

The theory of computation is a field of study which attempts to answer the following question: "What are the fundamental capabilities and limitations of computers?" The theory of computation is divided into three branches: automata theory and formal languages, computability theory, and computational complexity theory.

The goal of computational complexity theory is to classify computational problems according to their resource usage into problem classes. Resource usage is categorized by both time and space complexity. A computational problem is a collection of infinite inputs along with a set of solutions for each input. For example, consider the computational problem of finding the largest cycle in a graph. The input would be a set of nodes and edges, and the solution would be the size of the largest cycle.

An *alphabet* is a collection of symbols from which an input string for a computational problem is constructed. An example of an alphabet for a problem would be the set $\{0, 1\}$, and the input string would be a binary number. For the purposes of this paper, all the inputs to computational problems will be encoded as binary numbers.

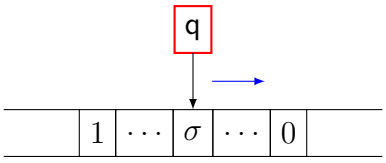
Decision problems are a set of computational problems involving "yes" or "no" answers. These problems can be viewed as a *language*, which is the set of inputs strings whose answer is "yes". The goal of a decision problem is to verify if a string is a member of a language. A decision version of the largest cycle problem would be to determine if a graph contains a cycle. The language associated with this decision problem is the set of all graphs that contain a cycle. The boolean satisfiable problem is another classic decision problem that we will formally define below.

Definition 2.1. A *boolean formula* is built from boolean variables, operators AND ($\wedge$), OR ($\vee$), NOT (negation, $\neg$), and parentheses. A formula is said to be satisfiable if it can be made TRUE by assigning boolean values to its variables. The Boolean satisfiability problem (SAT) is, given a formula, to check whether it is satisfiable.

2.1 Turing machines

A Turing machine is a model of computation that is able to simulate any computer algorithm, such as the algorithm to decide if a boolean formula is satisfiable. A Turing machine describes an abstract model of computation that manipulates symbols on an infinite tape. By providing a mathematical description of a very simple device capable of arbitrary computations, we are able to prove properties of computation in general. Despite how versatile the Turing machine is, even it cannot solve certain problems. Problems that are *undecidable* do not admit a Turing machine that computes the correct answer on all inputs.

Initially, the input to the problem is written on the tape of the Turing machine, and the tape is empty everywhere else. The machine will move the tape head across the tape to read the input. The machine performs the instructions given in the algorithm until it either reaches an accepting or rejecting state.



A Turing machine can do the following

1. A Turing machine can both read and write on the infinite tape.

2. The current location is marked by the tape head, which can move left or right.

3. The current state of the Turing machine is taken from a finite set of states, and depicts the "state of mind" of a person performing the computation.

4. The Turing machine will immediately halt when the accept or reject locations are encountered.

We will now formally define a Turing machine.

Definition 2.2. A Turing machine M is a 7 element tuple $\{Q, \Sigma, \Gamma, \delta, Q_s, Q_a, Q_r\}$, where

1. Q is the finite set of states,

2. Σ is the input alphabet not containing the blank symbol $\sqcup$,

3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,

4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function,

5. $Q_s \in Q$ is the starting state,

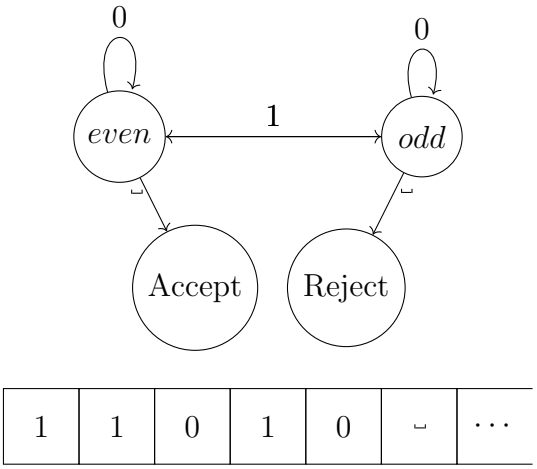
6. $Q_a \in Q$ is the accepting state,

7. $Q_r \in Q$ is the rejecting state, where $Q_r \neq Q_a$.

The most important part of the Turing machine is the transition function δ. The function δ takes the form: $Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$. If the machine was in state q and the tape head was reading the value a, the operation $\delta(q, a) = (s, b, L)$, would move the state to s, replace the value of a with b and move the tape head one location to the left. If the last value was R, then the machine would move the tape head to the right.

The Turing machine $M = \{Q, \Gamma, \Sigma, \delta, Q_s, Q_a, Q_r\}$ computes as follows. M will receive its input of size n consisting of symbols from the input alphabet. The first n tape locations of M will be filled with the input. Note that the input alphabet does not contain the blank symbol but the tape alphabet does, so the first blank symbol appearing on the tape denotes the end of the input string. The tape head starts on the left-most location. The transition function defines how the tape head moves through the machine M. Q_s denotes the state when we start, Q_a denotes the accept state which indicates a "yes" instance, and Q_r denotes the reject state which indicates a "no" instance.

We will see how a Turing machine can solve the decision problem of determining if the number 1 bits in a binary string is even. The Turing machine that solves this problem on input 11010 is displayed below.



The set of finite states q are $\{\text{even}, \text{odd}, \text{Accept}, \text{Reject}\}$. The input alphabet is $\{0, 1\}$ and the tape alphabet is $\{0, 1, \text{␣}\}$. The starting state is even, the accept state is yes, and the reject state is no. The transition function is specified as follows.

$$\begin{aligned}
\delta(\text{even}, 0) &= (\text{even}, 0, R) \\
\delta(\text{even}, 1) &= (\text{odd}, 1, R) \\
\delta(\text{odd}, 0) &= (\text{odd}, 0, R) \\
\delta(\text{odd}, 1) &= (\text{even}, 1, R) \\
\delta(\text{even}, \text{␣}) &= (\text{Accept}, \text{␣}, R) \\
\delta(\text{odd}, \text{␣}) &= (\text{Reject}, \text{␣}, R)
\end{aligned}$$

The machine will start off in the even state, and place the tape head at the beginning of the tape. The state then moves to odd, since the tape head read a 1. Then, the tape head moves back to even since another 1 was read. This continues until the tape head reads the "␣" token while in the odd state. This will move the machine to the reject state, and the machine will halt. We can confirm that the input string has 3 ones, which is in fact odd.

We don't usually describe algorithms using the formal definition of a Turing machine which would be too cumbersome. We can describe algorithms in three ways. We can give a formal description which spells out the exact workings of a Turing machine, including the states, the transitions function, etc. The second option is to describe simply how the tape head moves and how data is stored in the tapes, ignoring the states and transition functions. The final option is to just describe the algorithm using plain English, without regard to the implementation of a Turing machine. For instance, we would describe an algorithm to solve the above problem with the following steps:

1. Set count=0
2. Iterate the variable i through the values of the input string
3. When $i == 1$, increment count by 1.
4. If count is even, output "yes"

5. If count is odd, output "no"

If necessary, we can describe any algorithm we want using the formal definition of a Turing machine.

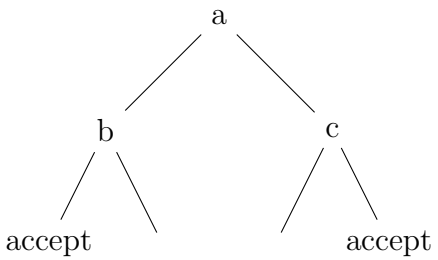
The kind of Turing machines we have seen so far are known as *deterministic Turing machines*(DTM). Given a configuration (a pair of the state and current tape symbol), the transition function of a DTM can only have one possible configuration as output. In the above problem, if the machine was in the even state and it read a 1, the only possible output for the transition function is to move to the odd state. There exists another form of Turing machines known as *nondeterministic Turing machines*(NTM). The following differences exist between deterministic and nondeterministic Turing machines.

1. The transition function outputs a set of 3 tuples rather than a single 3 tuple, where a tuple (a, b, c) says that the machine should move to state a, write b in the location, and move in direction c. This set can contain either $0, 1$, or more than 1 element.

2. An NTM is able to perfectly guess which of the outputs of the transition function it should take in order to reach an accept state.

3. There are no longer any reject states for an NTM. The machine rejects the input only if the machine has finished computation and has not reached an accept state. If any accept state is reached, the machine accepts the input.

NTMs can be thought of as trees with possibly many paths that could lead to an accept state.



The starting configuration a, has two possible successor configurations. The configuration b has one accept state, and the configuration c has another accept state. The machine will accept the input string because there exists at least one accept state. In contrast, a DTM can be thought of as having only one path from the starting configuration to either the accept or reject state.

It may seem as if NTMs are much more powerful than DTMs, but it is possible to simulate any NTM using a DTM simply by considering more states and a more complex transition function [11].

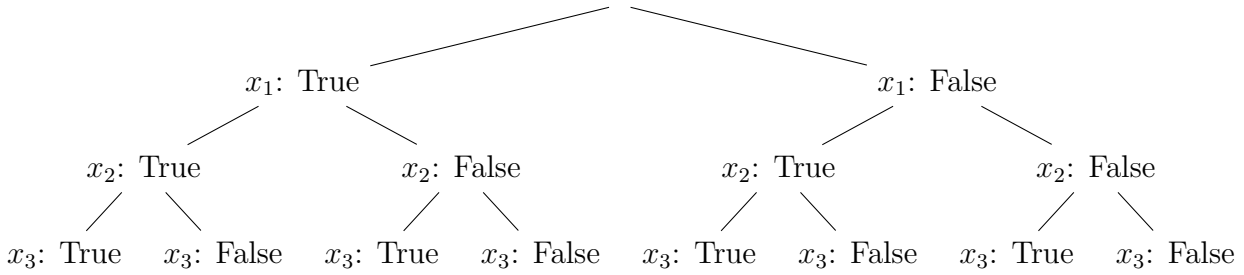
2.2 Complexity Classes

Computational problems, or languages, are part of different complexity classes. We will present definitions for two such classes.

Definition 2.3. The complexity class P consists of all languages that can be solved by a DTM in polynomial running time with respect to the size of the input. The complexity class NP consists of all languages that can be solved by a NTM in polynomial time. The class NP can also be defined as the set of languages whose solutions can be verified by a DTM in polynomial time.

The problem which asked to decide whether the number of 1 bits in a string are even was solved by a DTM. The number of operations this machine performed was polynomial in the size of the input string, since the tape head only made one pass through the input. Therefore, this problem is in the class P.

The running time of an NTM is represented as the maximum length of a path from the starting configuration, or root, of the NTM tree to one of the halting configurations, or leaves. The SAT problem, stated in definition 2.1 is solvable by a nondeterministic Turing machine. To see this, consider the following NTM tree that solves SAT on input $(x_1 \vee x_2) \wedge (\neg x_1 \wedge x_3)$.



The machine nondeterministically decides which boolean value to assign to each variable in the formula. Each leaf represents a viable assignment of boolean values to each variable. For example, the left-most leaf represents an assignment of true to every variable, and the right-most leaf represents an assignment of false to every variable. If any of these paths represent an assignment that evalues to true, the machine will accept the input. If none of the paths are satisfiable, then we will reject the input.

The running time of this NTM is polynomial in the size of the input, because the length of every path from root to leaf is the number of boolean variables in our boolean formula. Therefore, this problem is in NP. We can also see why NP problems have deterministic polynomial time verifiers. A verifier can check whether an assignment is satisfiable, but can't actually generate a satisfiable instance. In order to verify that the assignment at any leaf is indeed satisfiable, we can simply trace the path back to the root and evaluate the formula. This verification is polynomial in the size of the number of boolean variables.

Any problem solved by a DTM in polynomial time can also be solved by an NTM in polynomial time by simply constructing a tree that has exactly one path from the starting state to a halting state. The famous P *vs* NP asks any problem solved by an NTM in polynomial time can also be solved in deterministic polynomial time. In other words, is the class P equal to the class NP or is it a strict subset of NP. This problem is the most famous problem in theoretical computer science and is still unsolved. It is part of the 7 millennium problems, each of which offer a million dollars to the solver. In fact, if someone were able to show that $P = NP$, they would walk away with not just 1 million dollars, but 6 million(since

the Poincaré conjecture is already solved) because this would mean that proving every other one of these problems is computationally easy.

2.3 Reductions and Completeness

Definition 2.4. A language A has a *polynomial time reduction* to another language B if there exists a DTM M that runs in polynomial time such that for all strings $w \in A$, $M(w) \in B$. If there exists a string $w \notin A$, then $M(w) \notin B$. M is a DTM that takes in input w, writes the string $M(w)$ on the tape, and halts.

A reduction is a way of solving one computational problem by solving another. Let's say we want to solve problem A on input x. If there exists a polynomial time reduction from problem A to problem B, this means that we can simply check if the Turing machine that solves problem B accepts the input string $M(x)$. If this is true, then $x \in A$ and the Turing machine that solves A accepts the input x.

Assume we have a reduction from problem A to problem B and B can be solved by a DTM M_B in polynomial time. This means we have a DTM M that runs in polynomial time as described in Definition 2.4. If we want to check if x is in A, we just need to check if B accepts $M(x)$ by passing $M(x)$ as an input to machine M_B. Since M and M_B are DTMs that run in polynomial time, we have a DTM that runs in polynomial time that also solves A.

Definition 2.5. A problem A is said to be *NP complete* if every NP problem has a polynomial time reduction to A.

Theorem 2.6. *(Cook Levin Theorem) Boolean satisfiability problem is NP complete. [1] [8]*

If we are able to find a DTM that solves some NP complete problem such as SAT in polynomial time, then we have DTM that can solve any NP problem in polynomial time, and we have shown that $P = NP$. If we can show that some NP complete problem is not solvable by a DTM in polynomial time, then $P \neq NP$.

3 Average Case Complexity Class

Languages in worst case complexity theory are simply described by the set of accepting strings. However, we need more detail when describing problems in average case complexity. An average case complexity class consists of pairs, (D, μ), which are called distributional problems. The first element of the pair is the decision problem, and the second element of the pair is the probability distribution of the inputs. The probability distribution describes the likelihood of each input and it takes in binary strings as input. We can define any kind of efficient ordering in binary strings. For the sake of simplicity, we will use the standard lexicographic ordering of binary strings.

Definition 3.1. A cumulative distribution function μ is a non-decreasing function from binary input strings to the interval $[0, 1]$ which must converge to one. The density function associated with μ is written as μ' and is defined as $\mu'(0) = \mu(0)$ and $\mu'(x) = \mu(x) - \mu(x-1)$.

The function $\mu'(x)$ outputs the exact probability density of x. Note that the cumulative distribution of a string x is the sum of the probability densities of any other string $y \leq x$. We can define any probability distribution we want when we formalize a distributional problem. We can decide to give certain inputs a much higher probability than others, and this will change the efficiency of the algorithm. Intuitively, if our problem is "slower" on certain inputs than others, and we give these inputs a higher probability density than the "faster" inputs, our running time will be much slower than if we had swapped these densities.

4 AvgP and DistNP

We now have definitions for distributional problems. Before exploring the hierarchies of distributional problems, we will need to define a special class of distributions.

Definition 4.1. A distribution is said to be P *computable* if there exists a deterministic polynomial time Turing machine that can compute $\mu(x)$ for any input x.

If a distribution is P computable, we can efficiently compute the expansion of the cumulative and density probability distributions for any input. This also means that the length of the binary expansion of the probability distribution must be polynomial in the size of the input.

Definition 4.2. A distributional problem is said to be in the class *DistNP*, if the decision problem is in NP and the probability distribution is P computable.

Now, we can define an equivalent of the P class for distributional problems. Levin's original definition [9] may seem counterintuitive to most readers. We will first examine the more intuitive definition of an efficient average case distributional problem, and discuss why this definition has a major shortcoming.

Let the running time of a particular problem on input x be $t(x)$. Intuitively, we may think of defining an efficient distributional algorithm to be one that has running time $t(x)$ polynomial in the size of x. Now, we will discuss a very important shortcoming of this definition.

Let us define a distributional problem A. Let A have running time 2^n on $\frac{1}{2^n}$ of the inputs, and running time n^2 on $1 - \frac{1}{2^n}$ of the inputs. A will have expected running time $2^n \frac{1}{2^n} + n^2 \cdot (1 - \frac{1}{2^n}) = O(n^2)$. Now, let us define another distributional problem B, which has running time A^2. In other words, every distribution will have a running time squared of that of A in the same disribution. Therefore, B will have running time 2^{2n} on on $\frac{1}{2^n}$ of the inputs, and running time n^4 on $1 - \frac{1}{2^n}$. of the inputs. If we evaluate the expected running time of B, we find that it is $O(2^n)$.

This doesn't make sense, since we took the square of a polynomial running time and got an exponential running time. This tells us that if we have a computation that is expected polynomial time and then compute a worst case polynomial time function on this result, the entire running time may not be expected polynomial time. We will see that Leonid Levin's definition [9] resolves this conflict.

Definition 4.3. A function $t : \{0,1\}^* \to \mathbb{N}$ is polynomial on average with respect to a distribution μ if there exists some constant $\varepsilon > 0$ and k such that

$$\sum_{x \in \{0,1\}^*} \mu'(x) \frac{t(x)^\varepsilon}{|x|} = k.$$

A distributional problem D on distribution μ is said to be in $AvgP$ if there exists an algorithm to compute D whose running time is polynomial on average with respect to μ.

We will often think of this definition as the expected value of the running time raised to some small power must be polynomial in the size of the input. The sum $\sum_{x \in \{0,1\}^*} \mu'(x) t(x)$ represents the expected running time over all binary inputs x. The difference between this definition and the more intuitive definition we discussed earlier is the restricting ε in Levin's definition. We will see how Levin's definition of a polynomial on average running time resolves the conflict posed above.

We will consider the same distributional problems of A and B mentioned above. Let us assume that A has a polynomial on average running time according to Levin. According to definition 4.3, if $t(x)$ denotes the running time of A, $\sum_{x \in \{0,1\}^*} \mu(x) \frac{t(x)^\varepsilon}{|x|} = k$ for some $\varepsilon > 0$. If B has a function $t'(x)$ describing the running time on an input x, $t'(x) = t(x)^2$. Therefore,

$$\begin{aligned}
\mathbb{E}(t'(x)) &= \sum_{x \in \{0,1\}^*} \mu'(x) \frac{t'(x)^{\frac{\varepsilon}{2}}}{|x|} \\
&= \sum_{x \in \{0,1\}^*} \mu'(x) \frac{(t(x)^2)^{\frac{\varepsilon}{2}}}{|x|} \\
&= \sum_{x \in \{0,1\}^*} \mu'(x) \frac{t(x)^\varepsilon}{|x|} = k
\end{aligned}$$

Therefore, we have shown that B has polynomial on average running time by Levin's definition.

Definition 4.4. Let A be an algorithm that computes the answer to a decision problem. An algorithm $A(x, \varepsilon)$ with a running time polynomial in $\frac{|x|}{\varepsilon}$ is said to compute the algorithm A on input x with *benign faults* if it either computes the correct answer, or fails (exceeds the running time) by outputting '?'. A polynomial time benign algorithm is an algorithm $A'(x, \varepsilon)$ so that

$$\mathbb{P}[A'(x, \varepsilon) = ?] \leq \varepsilon.$$

We allow A' to have a maximum running time T which is polynomial in $\frac{|x|}{\varepsilon}$. If the running time of A' on input x exceeds T the algorithm will fail. When we want A' to have a lower probability of failing, we will need to decrease ε and increase $\frac{|x|}{\varepsilon}$. This makes sense because we have to allow the algorithm to have a higher running time in order for it compute the correct answer on a higher distribution of inputs.

Proposition 4.5. *A distributional problem D on distribution μ is in AvgP iff there exists a polynomial time benign algorithm that computes D.*

To prove this, we will begin with a lemma.

Lemma 4.6. *(Markov's Inequality) If X is a nonnegative random variable and $a > 0$, then*

$$\mathbb{P}[X \geq a] \leq \frac{\mathbb{E}(X)}{a},$$

where $\mathbb{E}(X)$ is the expected value of X.

Proof. Assume algorithm A computes D which is in AvgP. Let $t(x)$ represent the running time of algorithm A on input x. We will represent the timeout time of algorithm A as T. For any $\varepsilon > 0$, if $\mathbb{P}[t(x) \geq T] \leq \varepsilon$, then we need to show that T is polynomial in $\frac{|x|}{\varepsilon}$. Since A is in AvgP, there exists some constants c, k such that $\mathbb{E}_{x \in \{0,1\}^*} \frac{t(x)^{\frac{1}{c}}}{|x|} = k$. We know that $\mathbb{P}[t(x) \geq T] = \mathbb{P}[t(x)^{\frac{1}{c}} \geq T^{\frac{1}{c}}]$ by raising both sides to $\frac{1}{c}$. Now, we can use Markov's Inequality.

$$\begin{aligned}
\mathbb{P}[t(x) \geq T] &= \mathbb{P}[t(x)^{\frac{1}{c}} \geq T^{\frac{1}{c}}] \\
&= \mathbb{P}\left[\frac{t(x)^{\frac{1}{c}}}{|x|} \geq \frac{T^{\frac{1}{c}}}{|x|}\right] \\
&\leq \frac{\mathbb{E}\left(\frac{t(x)^{\frac{1}{c}}}{|x|}\right)}{\frac{T^{\frac{1}{c}}}{|x|}} \\
&= \frac{k}{\frac{T^{\frac{1}{c}}}{|x|}} = \frac{k|x|}{T^{\frac{1}{c}}} \leq \varepsilon
\end{aligned}$$

Rearranging this, we get that $T \geq \frac{|x|^c k^c}{\varepsilon^c}$, which shows that T is polynomial in $\frac{|x|}{\varepsilon}$.

Conversely, we need to show that if $A'(x, \varepsilon)$ is a benign algorithm scheme solving D, there exists an AvgP algorithm A to solve D. Let A be the algorithm that solves B with the parameters $\varepsilon = \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \ldots$ until A' doesn't fail. As we lower the probability that the algorithm will fail, we will in turn increase the running time of the algorithm.

We can think of A as repeatedly applying A' to solve D. Since the first parameter was $\frac{1}{2}$, A' will fail with probability $\frac{1}{2}$. A' will fail the next iteration with probability $\frac{1}{4}$. Since A' is a benign algorithm scheme, it has running time polynomial to $\frac{|x|}{\varepsilon}$. Therefore, the expected running time of producing the correct answer on the first iteration is $\left(\frac{|x|}{\frac{1}{2}}\right)^c = (2|x|)^c$, for some constant c. The second iteration only takes place with probability $\frac{1}{2}$(if the first iteration fails), so the expected running time of producing the correct answer in the second iteration is $\frac{1}{2}(4|x|)^c$. We need to show that the expected running time of A raised to some small power must be polynomial in the input size. We will choose this small power to be $\frac{1}{3c}$.

$$\begin{aligned}
\mathbb{E}(t(x)^{\frac{1}{3c}}) &\leq \left((2|x|)^c + \frac{1}{2}(2|x|)^c + \frac{1}{4}(8|x|)^c) \ldots\right)^{\frac{1}{3c}} \\
&= |x|^{\frac{1}{3}} \cdot \sum_{k=1}^{\infty} \frac{1}{2^k}(2^{k+1})^{\frac{1}{3}}
\end{aligned}$$

The large sum can be evaluated to converge to 1, so the overall expression is polynomial in $|x|$. Therefore, A, which was constructed from A', which was a polynomial time benign algorithm scheme, is in AvgP.

■

The consequence of this bijective relation is that any efficient algorithm in Levin's definitions can be made to be as accurate as possible, simply by increasing the resources available to the algorithm. This amount of resources is always polynomial in nature, and always preserves the structure of the algorithm to be in AvgP. This illustrates an analogue of the time hierarchy theorem in worst case complexity, which loosely states that given more time, a Turing machine can solve more problems.

5 Reducibility Between Distributional Problems

Definition 2.4 tell us about reductions between worst case computational problems. We will now define an equivalent notion of reductions for distributional problems.

Let us say we have a polynomial reduction based on definition 2.4 between the distributional problems (D_1, μ_1) and (D_2, μ_2). This doesn't mean this is a valid reduction in the average case. Recall that distributional problems have a probability distribution of inputs. Let us say that our polynomial function that performs the reduction converts an input x for D_1 into an input y for D_2. The probability $\mu_1(x)$ may not be equal to the probability $\mu_2(y)$. If x has a very high probability with respect to μ_1 than y with respect to μ_2, then having a polynomial time on average algorithm for D_2 does not necessarily imply that D_1 can be solved with a polynomial time on average algorithm. This is because the rare inputs(which are difficult to solve) in D_2 are mapped to very common inputs(which are meant to be easier to solve) in D_1. The non-efficient algorithm used to solve these more difficult inputs in D_2 will be used to solve the common inputs for D_1, and the overall algorithm may not be efficient. Thus, we also need a way of "preserving" the probability distribution of μ_1 in μ_2.

We will ensure that our definition for average polynomial time reduction is efficient in running time, valid for computing the answer to the original problem, and preserves the probability distributions of the original problem. We will call these three conditions efficiency, validity, and domination. We will have a Turing machine M that performs the reduction from inputs to D_1 to inputs of D_2.

Definition 5.1. We will say that there is a polynomial time reduction from the distributional problem (D_1, μ_1) to (D_2, μ_2) if there exists a DTM M such that the following conditions hold.

1. Efficiency: The Turing machine M must have running time polynomial time on average over the distribution μ_1. Let $t(x)$ represent the running time of M on input x of D_1. The machine M is polynomial time on average if there exists some $\varepsilon > 0$ such that

$$\sum_{x \in \{0,1\}^*} \mu_1'(x) \frac{t(x)^\varepsilon}{|x|} = k,$$

for some constant k.

2. Validity: Let M^{D_2} be the output of M first performing the reduction on input x and then querying the oracle machine of D_2. This output must be accurate with probability at least $\frac{2}{3}$. For every $x \in \{0,1\}^*$,

$$\mathbb{P}[D_2(M(x)) = D_1(x)] \geq \frac{2}{3},$$

where $D_1(x)$ is the answer to the decision problem D_1 on input x.

3. Domination: The distribution μ_2 must "dominate" the distribution μ_1. If machine M produces an input y for D_2 from an input x for D_2, the probability of y in μ_2 must be at most some polynomial order in the input size times the probability of x in μ_1. This can be written as,

$$\mu_2'(y) \geq \frac{1}{|y|^c} \cdot \sum_{x \in \{0,1\}^*} \mathbb{P}[M(x) = y] \cdot \mu_1'(x),$$

for some constant c, where $\mathbb{P}[M(x) = y]$ is the probability that M outputs y from input x.

The efficiency condition is to ensure that the Turing machine M actually does a polynomial on average reduction. The entire purpose of the polynomial reduction is the ensure that if we have an efficient solution for the reduced problem, we also have an efficient solution for the original problem. If the reduction is not in polynomial time, then this statement is false.

The validity condition is to ensure that the reduction actually yields the correct answer. Breaking down the formulaic definition, if the original input to D_1 is x, $M^{D_2}(x)$ is the answer of D_2 after the reduction. The probability that this answer is equal to the original answer of problem D_1 should be greater than $\frac{2}{3}$. The value of $\frac{2}{3}$ is arbitrary, and any constant over $\frac{1}{2}$ will do. We will eventually end up with exponential accuracy by applying this function a polynomial amount of times.

The domination condition is probably the most interesting and potentially confusing. The key idea here is to preserve the distributions of inputs. In other words, if an input x of μ_1 is being reduced to input y of μ_2, the probability of obtaining x, or $\mu_1'(x)$ should roughly be equal to the probability of $\mu_2'(y)$. The gold standard is obviously that the two probabilities being exactly equal. However, as long as $\mu_2'(y)$ is not too much lower than $\mu_1'(x)$, we can say the reduction is feasible. We see in the definition that $\sum_{x \in \{0,1\}^*} \mathbb{P}[M(x) = y] \cdot \mu_1'(x)$ is the expected value of $\mu_1'(x)$ such that $M(x) = y$. The probability $\mu_2'(y)$ should not be more than an inverse polynomial in $|y|$ smaller than this expected value.

We saw in worst case complexity that we can solve a computational problem by solving another problem that the first one reduces to. More importantly, if we have problem A reducing to problem B, and problem B is solvable in polynomial time, so is problem A. Therefore, if we have a polynomial time algorithm for an NP complete problem, we then have a polynomial time algorithm for any NP problem, and we can show that $P = NP$.

We were able to show this fact for worst case problems by simply applying the reduction to the input and solving problem B. This was exactly the answer to problem A. If problem B was solvable in polynomial time, the entire process has polynomial running time, since the reduction was polynomial.

This is not quite so simple for distributional problems. We have to show the entire process is polynomial on average, and proving that this is true using Levin's definition, and this is not so trivial as the worst case analogue.

Proposition 5.2. *If (D_1, μ_1) is reducible to (D_2, μ_2) through a deterministic polynomial time oracle Turing machine and (D_2, μ_2) is solvable by a deterministic Turing machine with a polynomial on average running time, then so is (D_1, μ_1).*

Proof. We will start with a classic technique used to prove reduction results. We will create an intermediate distribution μ_I, which precisely reflects the distribution of μ_1. The only difference is that μ_I consists of inputs to D_2 rather than D_1. For example, if for some $x \in \{0,1\}$, $M(x)$, or the reduced input of x, is y, then $\mu_1'(x) = \mu_I'(y)$. We can define this as,

$$\mu_I'(y) = \sum_{x \in \{0,1\}^*} \mathbb{P}[M(x) = y] \cdot \mu_1'(x).$$

The domination condition of 5.1 involved an inequality and a polynomial factor of $\frac{1}{|y|^c}$. We now make μ_I perfectly resemble the distribution μ_1. This allows us to directly compare μ_I and μ_2.

If (D_2, μ_I) is solvable in polynomial time on the average, then so is (D_1, μ_1). This is because μ_I is an exact copy of μ_1. Therefore, any time we want to solve D_1 on an input x, we can simply solve the corresponding problem D_2 on the reduced input. If the latter process takes place in polynomial time on the average, we can be sure that D_1 is solvable in x, since the distributions are exactly the same.

We now need to show that if (D_2, μ_2) is solvable in polynomial time on the average, then (D_2, μ_I) is solvable in polynomial time on the average. By our hypothesis, we know that (D_2, μ_2) is in AvgP, and by definition 4.3, for some $\varepsilon > 0$, there exists a function $t : \{0,1\}^* \to \mathbb{N}$ such that

$$\sum_{y \in \{0,1\}*} \mu_2'(y) \frac{t(y)^\varepsilon}{|y|} = k,$$

for some constant k. If we can show that t is polynomial on average with respect to μ_I as well, then we have shown that there (D_2, μ_I) is solvable in polynomial on the average running time. We only need to show that

$$\sum_{y \in \{0,1\}*} \mu_I'(y) \frac{t(y)^\varepsilon}{|y|} = k_1,$$

for some constant k_1. Since (D_2, μ_I) reduces to (D_2, μ_2), we know that μ_2 dominates μ_I. We follow the exact same formula as the domination condition in definition 5.1, but we omit the section related to the probability for the machine M querying an input in the target distribution from the original distribution. We omit this part because the machine performing the reduction from μ_I to μ_2 does not need to convert the inputs at all since the inputs are exactly the same. For any $y \in \{0,1\}^*$, $\mu_I'(y)$ should roughly be equal to $\mu_2'(y)$. Thus, the definition tells us that $\mu_2'(y) \geq \frac{1}{|y|^c}\mu_I'(y)$. Rearranging, we can write

$$\mu_I'(y) \leq |y|^c \mu_2'(y),$$

for some constant c.

Next, we will define a set G that consists of a specific set of inputs $y \in \{0,1\}^*$. G only contains the inputs y such that $t(y) \leq |y|^{\frac{2c}{\varepsilon}}$, with the same assumptions about ε and c as above. Formally, we have

$$G = \{y : t(y) \leq |y|^{\frac{2c}{\varepsilon}}\}.$$

We will now split the sum $\sum_{y\in\{0,1\}^*} \mu_I'(y)\frac{t(y)^{\frac{\varepsilon}{2c}}}{|y|}$ based on whether $y \in G$ or not. Note that we are dividing ε by $2c$ in this sum. This is perfectly reasonable since we are allowed to choose any value for ε that we want. This change is necessary for the proof to work out.

If $y \in G$, can show that this sum is at most 1. We can raise both sides of $t(y) \leq |y|^{\frac{2c}{\varepsilon}}$ to the power $\frac{\varepsilon}{2c}$ to get $t(y)^{\frac{\varepsilon}{2c}} \leq |y|$. We can plug this into our original sum to get

$$\begin{aligned}\sum_{y\in G} \mu_I'(y)\frac{t(y)^{\frac{\varepsilon}{2c}}}{|y|} &\leq \sum_{y\in G} \mu_I'(y)\frac{|y|}{|y|} \\ &= \sum_{y\in G} \mu_I'(y) \\ &= 1.\end{aligned}$$

We were able to make the last conclusion by definition 3.1 which says that every density function of a distribution must converge to 1. Therefore, we have shown that if $y \in G$, the sum is at most 1, and t is polynomial on the average with respect to μ_I.

Now, we need to show if $y \notin G$, t is still polynomial on the average with respect to μ_I. By the domination condition, we have $\mu_I'(y) \leq |y|^c\mu_2'(y)$, for some constant c. We then plug in and simplify,

$$\sum_{y\notin G} \mu_I'(y)\frac{t(y)^{\frac{\varepsilon}{2}}}{|y|} \leq \sum_{y\in G} |y|^c\mu_2'(y)\frac{t(y)^{\frac{\varepsilon}{2c}}}{|y|}.$$

We also know that since $y \notin G$, $t(y) > |y|^{\frac{2c}{\varepsilon}}$, which means that $|y|^c < t(y)^{\frac{\varepsilon}{2}}$. Substituting, we have

$$\begin{aligned}\sum_{y\notin G} |y|^c\mu_2'(y)\frac{t(y)^{\frac{\varepsilon}{2}}}{|y|} &< \sum_{y\notin G} t(y)^{\frac{\varepsilon}{2}}\mu_2'(y)\frac{t(y)^{\frac{\varepsilon}{2c}}}{|y|} \\ &< \sum_{y\notin G} t(y)^{\frac{\varepsilon}{2}}\mu_2'(y)\frac{t(y)^{\frac{\varepsilon}{2}}}{|y|} \\ &= \sum_{y\notin G} \mu_2'(y)\frac{t(y)^{\varepsilon}}{|y|} \\ &< k.\end{aligned}$$

Since we have proved that $\sum_{y\in\{0,1\}^*} \mu_I'(y)\frac{t(y)^{\varepsilon}}{|y|} < k$, we know that t is polynomial on the average with respect to μ_I. Therefore, we have shown that if (D_2, μ_2) is solvable in polynomial time on the average, then so is (D_2, μ_I), which means that (D_1, μ_1) is solvable in polynomial time on the average. ■

We will need to mention one, small constraint with these reductions. Given an input x to (D_1, μ_1), the reduction machine M must output a result that is some polynomial in the size of x, or $|x|^k$ for some k. This is important because running time is a reflection on the size of the input. If M reduces x to an input very small or large, the running time will be drastically different for the reduced problem compared to the original. Although we will have an efficient algorithm by Levin's definition when we divide by $|x|$ in the end of the analysis, this algorithm is not really efficient for normal purposes. Therefore, we need some sort of a bound for the size of the input produced by the reduction.

With this result, we can perform analysis on one problem by considering the feasibility of a reducible problem. We will also be able to define the notion of distributional NP complete problems.

6 DistNP Complete problems

We saw the notion of NP complete problems in worst case complexity theory. We want a DistNP complete problem to reflect definition 2.4 except for distributional problems.

Definition 6.1. A distributional problem A is said to be *distNP complete* if $A \in$ DistNP and every distributional NP problem is reducible to A.

As seen in reductions with distributional problems, we need the probability distributions to be somewhat preserved. Therefore, even if we have a distributional problem with an NP predicate, meaning the problem is a worst case NP problem, we cannot simply assume that this is a distNP complete problem for any given distribution. We will need to define a uniform distribution that we would like to use in every distNP complete problem.

Intuitively, we want a uniform distribution to assign probability equally for every input of a given size. An input of a higher size should be assigned a lower probability than an input of a lower size, since the larger inputs are more rare on average. One other important condition we need to meet is that the distribution converges to 1. Keeping these restrictions in mind, we will precisely define a uniform distribution.

Definition 6.2. A cumulative uniform distribution function μ_0 is defined in terms of the density function μ_0' such that

$$\mu_0'(x) = \frac{1}{|x| \cdot (|x|+1)} \cdot 2^{-|x|}.$$

This definition will definitely assign equal probabilities to all inputs of the same size, since $|x|$ is constant. This will also assign less probability to inputs of higher size. We now just need to verify that this definition converges to 1 over all values of x. We have,

$$\sum_{x \in \{0,1\}^*} \frac{1}{|x|(|x|+1)} \cdot 2^{-|x|} = \sum_{n=0}^{\infty} 2^n \cdot \frac{1}{n(n+1)} \cdot 2^{-n}.$$

The value n is going through every possible size of the input. Since the inputs are in binary, there are 2^n possible inputs of size n. Since $|x| = n$, we substitute n for $|x|$. After simplifying, we have

$$\sum_{n=0}^{\infty} 2^n \cdot \frac{1}{n(n+1)} \cdot 2^{-n} = \sum_{n=0}^{\infty} \frac{1}{n(n+1)},$$

which we can verify is a telescoping series that converges to 1. Therefore, this definition of a uniform distribution satisfies all the intuitive requirements we posed earlier. For the purposes of this paper, we will simplify this definition to just $\mu_0'(x) = \frac{1}{|x|^2} \cdot 2^{-|x|}$, to make computation easier.

We will now consider an example of a distributional NP complete problem. The *halting problem* is a famous problem in theoretical computer science which asks whether a program will halt on a given input. This problem is *undecidable,* which means there does not exist any Turing machine that can decide this problem over all possible programs and inputs. A possible way of solving this problem may be to consider whether the program halts in an arbitrarily large number of steps, but if this is not true, we cannot be sure that the program will never halt.

The *Bounded Halting problem*, is a similar problem which asks whether a program will halt on a given input within k steps, where k written in unary is polynomial in the size of the input. We will use a more rigorous definition of the bounded halting problem in this paper.

Definition 6.3. (Bounded halting problem) The bounded halting problem(BH) takes as input (M, x, t), where M is a nondeterministic Turing machine, x is the input string, and t is the allowable time written in unary length polynomial in the size of x. The problem is to decide whether machine M accepts input x in time at most t.

Bounded halting is definitely in NP because it is decidable by a nondeterministic Turing machine in polynomial time.

Proposition 6.4. *The bounded halting problem is NP complete.*

Proof. Let problem A be a generic NP problem. By the definition of an NP problem, there exists a nondeterministic Turing machine M_a that decides A. The problem is now to check if M_A accepts a particular input x' in polynomial time. We claim that we can solve any such A given that we are able to solve a specific input of BH.

For this specific input for BH, we will pass in $(M_A, x, |x|^k)$ for some constant k. If M_A validates input x in time $|x|^k$, which is polynomial in the size of x, this input is accepted. This precisely means that we have checked if A is solvable on input x in polynomial time. Since, we can solve any generic NP problem A through bounded halting problem, this BH is NP complete. ■

7 DistNP Completeness of Bounded Halting Problem

We first need to come up with a distributional analogue of bounded halting problem that is in distNP. We will denote this distributional problem as Π_{BH}.

Definition 7.1. $\Pi_{BH} = (BH, \mu_{BH})$, where $\mu'_{BH}(M, x, 1^k) = \frac{1}{|M|^2 \cdot 2^{|M|}} \cdot \frac{1}{|x|^2 \cdot 2^{|x|}} \cdot \frac{1}{k^2}$.

This is a little different than our general definition of a uniform distribution. We can check that it still does satisfy our intuitive requirements we stated earlier. The main difference is the structure of the term involving $|k|$. This is not too surprising, since k is comprised of only 1's, and we cannot reorder k in any way. To make our definition more specific, we can write that $\mu'_{BH}(M, x, 1^k) = \frac{1}{|M|(|M|+1)\cdot 2^{|M|}} \cdot \frac{1}{|x|(|x|+1)\cdot 2^{|x|}} \cdot \frac{1}{k(k+1)}$, and in fact verify that this will converge to 1 over all values of x.

It was fairly easy to show that BH was in worst case NP complete. However, we cannot simply extend this proof for distributional problems. For different distNP problems, there will be different distributions. We can reduce each of these problems and have a different μ_{BH}, but this will mean we have different instances of a distributional bounded halting problem. Additionally, some of these distributions will be very much skewed compared to the uniform distribution defined above, and will not preserve the domination condition.

The overall sketch of the proof that will follow is to have an efficient way of compressing any input x for the generic distNP problem into a string $c(x)$ that will ensure that the domination condition of polynomial on the average reductions is true. Let M be the Turing machine solving a general distNP problem A. We will pass in $(M', c(x), t)$, as input to BH, where M' decompresses $c(x)$ and then checks whether M is able to solve x in time at most t polynomial in the size of x.

Our expression for the uniform distribution is now

$$\mu_{BH}\left(M, c(x), 1^{\text{poly}(|x|)}\right) = \frac{1}{|M|^2 \cdot 2^{|M|}} \cdot \frac{1}{|c(x)|^2 \cdot 2^{|c(x)|}} \cdot \frac{1}{\text{poly}(|x|)^2}.$$

The exponential value in the denominator of this expression is $2^{|c(x)|}$. Therefore, we will want the size of $c(x)$ to be at most the base 2 logarithm of $\frac{1}{\mu'(x)}$. This is the constraint that will result in the domination condition of the reduction being fulfilled.

Proposition 7.2. *Π_{BH} is distributional NP complete.*

We need to show that a general distNP problem (D, μ) is reducible to (BH, μ_{BH}). We will begin with a lemma.

Lemma 7.3. *For every input $x \in \{0,1\}^*$, there exists an encoding $c(x)$ such that*

$$|c(x)| \leq \log_2 \frac{1}{\mu'(x)} + 1,$$

such that this function is polynomial time computable and it is one-to-one.

Proof. We will assume that every input string $x \in \{0,1\}^*$ has some positive probability. In case there exists some $x \in \{0,1\}^*$ such that $\mu'(x) = 0$, we can simply omit this value of x from our consideration.

We know that every input x has a cumulative binary probability distribution $\mu(x)$. Since every probability is positive, we know that $\mu(x) > \mu(x-1)$ for all $x > 0$, simply because $\mu'(x) = \mu(x) - \mu(x-1) > 0$. We will compress x into the smallest prefix of $\mu(x)$ such that it is different from the same size prefix of $\mu(x-1)$.

More formally, let $\mu(x) = 0.a_1a_2a_3 \ldots a_n a_{n+1} \ldots$ and $\mu(x-1) = 0.b_1b_2b_3 \ldots b_n b_{n+1} \ldots$ Let $a_1 = b_1, a_2 = b_2 \ldots a_n = b_n$, but $a_{n+1} \neq b_{n+1}$. Then we will let $c(x) = a_1a_2 \ldots a_n a_{n+1}$. This can be seen more clearly in the following representation,

$$\begin{aligned}\mu(x-1) &= 0.101101010010\\ \mu(x) &= 0.101101011110\\ \mu(x+1) &= 0.101101101010,\end{aligned}$$

where the red and blue numbers denote $c(x)$ and $c(x+1)$ respectively.
We now need to show that the three conditions of the lemma are true.

- This encoding is efficient because by definition 4.2, $\mu(x)$ and $\mu(x-1)$ can be computed in polynomial time.
- This encoding is also one-to-one because $c(x) \neq c(x-1)$. If this was the case, then $c(x)$ would not be defined with this prefix, since it is equal to the same size prefix of $\mu(x-1)$. We can extend this chain of inequalities to show that for all $i, j \in \{0,1\}^*$, $c(i) \neq c(j)$. Thus, if $c(i) = c(j)$, then $i = j$, making the function one-to-one.
- Now, we need to show that $|c(x)| \leq \log_2 \frac{1}{\mu'(x)} + 1$. In order to identify how long the prefix of $c(x)$ needs to be, we identify what is the first bit that is different between $\mu(x)$ and $\mu(x-1)$. This means every other bit to the left of this bit is the same between the two cumulative probabilities. Therefore, $\mu'(x) = \mu(x) - \mu(x-1)$ will consist of many zeroes in the beginning, and finally have a nonzero digit in the place of the first difference. When we take $\log_2 \frac{1}{\mu'(x)}$, we will get the number of digits before this first nonzero digit. We add a one to the logarithm to account for rounding errors since $|c(x)|$ must always be an integer.

■

We will now prove proposition 7.2

Proof. The input $\left(M', c(x), 1^{\text{poly}(|x|)}\right)$ is passed as input to BH, where machine M' decompresses x and computes $M(x)$, and M is the Turing machine solving the distributional problem (D, μ). We need to verify that this reduction satisfies the efficiency, validity, and domination conditions of definition 5.1.

- This reduction is efficient because $c(x)$ can be computed in polynomial time. For more efficient decompressing $c(x)$ to get x, we can binary search through the values of $c(x)$ to find x. This binary search is possible because $c(x) > c(x-1)$ for all $x > 0$, which makes it a monotonous function. Further, machine M' has access to the oracle machine for problem (D, μ), this computation is a single operation.
- The reduction is valid because BH accepts $\left(M', c(x), 1^{\text{poly}(|x|)}\right)$ if and only if the machine M accepts x in time polynomial in the size of x.
- To see that μ is dominated by μ_{BH}, we will write out the formula for the uniform distribution

$$\mu_{BH}\left(M, c(x), 1^{\text{poly}(|x|)}\right) = \frac{1}{|M|^2 \cdot 2^{|M|}} \cdot \frac{1}{|c(x)|^2 \cdot 2^{|c(x)|}} \cdot \frac{1}{\text{poly}(|x|)^2}.$$

We know that $\frac{1}{|M|^2 \cdot 2^{|M|}}$ is a constant only depending on the size of the machine that solves (D, μ), so we will represent this value as a constant k. We also know from lemma 7.3 that $|c(x)| \leq \log_2 \frac{1}{\mu'(x)} + 1$. Therefore, we know that $\mu'(x) \leq 2 \cdot 2^{-|c(x)|}$. We have,

$$\begin{aligned}\mu_{BH}\left(M, c(x), 1^{\text{poly}(|x|)}\right) &\geq k \cdot \frac{\mu'(x)}{|c(x)|^2 \cdot 2} \cdot \frac{1}{\text{poly}(|x|)^2} \\ &\geq \frac{1}{\text{poly}\left(M, c(x), 1^{\text{poly}(|x|)}\right)} \cdot \mu'(x).\end{aligned}$$

This satisfies the domination condition and we have proved that Π_{BH} is DistNP complete.

■

8 Current State of Research

We have shown that BH is distNP complete over a uniform distribution. This uniform distribution reflects the way inputs are usually passed into a problem, which is completely random. Therefore, this is the kind of distribution we prefer for our distribution problems. We have some classical NP complete problems such as SAT and Travelling Salesman which we would like to find distributional analogues of. Leonid Levin and Russel Impagliazzo have shown in their paper [6] that if there is an efficient average-case algorithm for a distNP-complete problem under the uniform distribution, then there is an average-case algorithm for every problem in NP under any polynomial-time computable distribution. This means if we are able to find a distNP complete analogue for a classical NP complete problem, then we can extend this to any distribution we want. Noam Livne showed that all NP complete problems have distributional versions that are distNP complete in his paper [10], but the distributions of these problems may be unnatural. For example, the distribution for a distNP complete SAT problem might give higher probability to certain inputs over others to make it reducible to every other distNP problem. One of the current aims of researchers is to find distributional versions of classical NP complete problems under the uniform distribution.

One might notice that we have already found that BH is distNP complete under the uniform distribution. However, we don't consider BH to be a natural problem. There are several opinions on this topic, but Livne's opinion [10] is that we can think of a problem as natural with respect to a result if it has other important implications other than the proof of the result. SAT is a natural problem because it existed as a logical problem before the proof of the Cook Levin theorem. However, BH would not be considered a natural problem because it mostly represents a theoretical simulation of a process and seems like an artificial construction just to prove the proposition.

Researchers are also trying to tackle the simpler problem of finding reductions between distributional problems. The classical reductions between worst case problems are not easily extendable to distributional problem with uniform distributions.

The current state of research is very active, and researchers are still trying to find answers that will give us a complete theory of average case complexity.

9 Acknowledgements

The author would like to thank Sawyer Anthony Dobson and Simon Rubinstein-Salzedo for helpful conversations.

References

[1] S. A. Cook. The complexity of theorem-proving procedures. In *Logic, Automata, and Computational Complexity: The Works of Stephen A. Cook*, pages 143–152. 2023.

[2] J. Edmonds. Paths, trees, and flowers. *Canadian Journal of mathematics*, 17:449–467, 1965.

[3] O. Goldreich. Notes on levin's theory of average-case complexity. *Studies in Complexity and Cryptography. Miscellanea on the Interplay between Randomness and Computation: In Collaboration with Lidor Avigad, Mihir Bellare, Zvika Brakerski, Shafi Goldwasser, Shai Halevi, Tali Kaufman, Leonid Levin, Noam Nisan, Dana Ron, Madhu Sudan, Luca Trevisan, Salil Vadhan, Avi Wigderson, David Zuckerman*, pages 233–247, 2011.

[4] J. Hartmanis and R. E. Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965.

[5] R. Impagliazzo. A personal view of average-case complexity. In *Proceedings of Structure in Complexity Theory. Tenth Annual IEEE Conference*, pages 134–147. IEEE, 1995.

[6] R. Impagliazzo and L. LA. No better ways to generate hard np instances than picking uniformly at random. In *Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science*, pages 812–821. IEEE, 1990.

[7] R. M. Karp. *Reducibility among combinatorial problems*. Springer, 2010.

[8] L. A. Levin. Universal sequential search problems. *Problemy peredachi informatsii*, 9(3):115–116, 1973.

[9] L. A. Levin. Average case complete problems. *SIAM Journal on Computing*, 15(1):285–286, 1986.

[10] N. Livne. All natural np-complete problems have average-case complete versions. *computational complexity*, 19(4):477–499, 2010.

[11] A. M. Turing et al. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.