

Probabilistic Computation and Linear Time

Lance Fortnow
Michael Sipser*
MIT Math Dept.
Cambridge, MA 02139

April 1, 1997

Key Words: Probabilistic Computation, Time Hierarchies, Relativization

AMS (MOS) Subject Classification: 68C25

Abstract

In this paper, we give an oracle under which BPP is equal to probabilistic linear time, an unusual collapse of a complexity time hierarchy. In addition, we also give oracles where Δ_2^P is contained in probabilistic linear time and where BPP has linear sized circuits, as well as oracles for the negation of these questions. This indicates that these questions will not be solved by techniques that relativize. We also note that probabilistic linear time can not contain both NP and BPP, implying that there are languages solvable by interactive proof systems that can not be solved in probabilistic linear time.

1 Introduction

According to general belief, a problem has an efficient deterministic algorithm when the algorithm runs in time polynomial in the size of the problem. At first this seems natural, though algorithms that take time n^{10} , where n is the size of the problem, strain the notion of efficiency. Most of the natural problems that have polynomial time algorithms have algorithms that require only a small polynomial number of steps such as n^2 . We can then ask the natural question: Do all problems with polynomial time algorithms have algorithms with a small polynomial time bound?

To answer that question, we go back to the beginning. In what is commonly believed to be the seminal paper in computational complexity theory [6], Hartmanis and Stearns show for any $k \geq 1$ there exist problems with deterministic n^{k+1} algorithms but no deterministic algorithms exist which run in n^k steps. This hierarchy theorem answered the question as soon as it was asked. So theoretical computer scientists believed—until they discovered the coin.

Several problems have polynomial time probabilistic algorithms where no efficient deterministic algorithms are known, primality being the best known example [14]. We can then ask the same question as before: Do all problems with polynomial time probabilistic algorithms have probabilistic algorithms with a small polynomial time bound?

In contrast with the deterministic case, this question remains unsolved. The simple diagonalization that established the deterministic hierarchy theorem fails in the probabilistic case. The main result of this paper shows that this happens for a fundamental reason.

*Both authors are supported in part by NSF Grant DCR-8602062 and Air Force Grant AFOSR-86-0078. Fortnow's current address: Computer Science Dept., University of Chicago, Chicago, IL 60637

We show this by exhibiting an oracle relative to which probabilistic linear time equals BPP, probabilistic polynomial time, as well as an oracle for which they differ. Thus techniques that relativize will not answer this question. Virtually all known techniques for solving problems of this type relativize, particularly the techniques that separate the deterministic time classes.

Our result suggests the surprising possibility that a complexity time hierarchy has a non-trivial collapse. Results of this nature show some fundamental differences in probabilistic computation versus other forms of computation such as deterministic and non-deterministic.

We also show some other relativized results in this paper relating to randomness and linear time.

2 Definitions

A probabilistic Turing machine M is a standard Turing machine with a special *coin toss* state. When the machine enters this state, with probability $1/2$ it will immediately enter a special *heads* state; otherwise it will enter a special *tails* state. This definition is equivalent to other probabilistic Turing machine models such as a standard Turing machine having a random tape [3].

A probabilistic machine M accepts a language $L \subseteq \Sigma^*$ if:

1. For all $x \in L$ we have $\Pr(M \text{ accepts } x) > 2/3$.
2. For all $x \notin L$ we have $\Pr(M \text{ accepts } x) < 1/3$.

A probabilistic machine M is *proper* if it accepts some language L . A probabilistic machine M is *improper* if for any string x , M on x accepts with probability between $1/3$ and $2/3$. We say a proper probabilistic Turing machine M has *one-sided error* if for $x \notin L$, the probability that M accepts x is zero, *i.e.* M does not accept x on any computation path.

We assume throughout the paper that $\Sigma = \{0, 1\}$ and all logarithms are base two. We also will use n freely to represent $|x|$, the length of the input.

We say $L \in \text{BPTIME}[f(n)]$ if there is a proper probabilistic machine for L that runs in $O(f(n))$ steps. $\text{BPP} = \cup_{k>0} \text{BPTIME}[n^k]$. We use RTIME and R respectively for one-sided error probabilistic machines. $\text{RTIME}[f(n)] \subseteq \text{NTIME}[f(n)]$, because one can just guess the coin tosses. We use ZPTIME and ZPP for zero-sided error, *i.e.* a language L has $\text{ZPTIME}[f(n)]$ machine M if the probabilistic machine M runs in $O(f(n))$ steps, outputs an answer with probability greater than two-thirds and when it outputs an answer the answer is always correct. $\text{ZPTIME}[f(n)] = \text{RTIME}[f(n)] \cap \text{co-RTIME}[f(n)]$. For a more thorough introduction to probabilistic computation see Gill's seminal paper this subject [3].

An oracle A is a subset of Σ^* . A *relativized* Turing machine is a machine with a special query tape on which it writes a string $x \in \Sigma^*$. The Turing machine then enters an oracle query state that immediately goes to a special *yes* state if $x \in A$ and to a *no* state otherwise. In shorthand, we say the machine makes the oracle query x . Equivalently, we can think of an oracle as its characteristic function $A : \Sigma^* \rightarrow \{0, 1\}$ where $A(x) = 1$ iff $x \in A$. We relativize a circuit by having a special oracle gate that takes as input a query to the oracle and returns true if that string is in the oracle.

When we use recursion theoretic techniques to prove a statement true, the proof will work even if all the machines involved have access to the same oracle. For example, we can prove $\text{P} \neq \text{EXP}$ using diagonalization, *i.e.* we create a language in exponential time defined to be different than each possible polynomial-time machine. This proof works even if we allow the exponential-time machine and the polynomial-time machines access to the same arbitrary oracle. Thus $\text{P} \neq \text{EXP}$ is true under any oracle A .

Suppose we show a complexity statement is true under a certain oracle B . If we could prove the statement false using recursion theoretic methods then the statement would be false under all oracles. This contradiction tells us recursion theoretic techniques will not work to prove the statement false. If we can find two oracles A and B such that the statement is true under oracle A and false under oracle B then recursion theoretic techniques will not work to settle the statement true or false. We will need other techniques, techniques that do not relativize, to settle this statement. Most complexity statements relativized both true and false have remained unsettled by any techniques.

3 Our Results and Related Results

We show the existence of oracles under which the following hold:

1. $\text{BPP} = \text{BPTIME}[n]$ (actually we will show $\text{BPP} = \text{RTIME}[n]$, $\text{BPP} \subseteq \text{NTIME}[n]$ and $\text{BPP} = \text{ZPTIME}[n]$)
2. BPP has linear-size circuits.
3. $\Delta_2^P \subseteq \text{BPTIME}[n]$ (Δ_2^P is defined as P with access to an NP oracle)
4. The negation of each of the above.

We also show there must exist a language in either BPP or NP but not in $\text{BPTIME}[n]$. This result implies there are languages accepted by interactive proof systems that are not accepted in probabilistic linear time. This is the first result showing a separation for languages accepted by interactive proof systems not known to hold for NP or BPP. (See [5] for the definition of interactive proof systems.)

Hartmanis and Stearns [6] with Hennie and Stearns [7] show for “nice” f and g such that $\lim_{n \rightarrow \infty} \frac{g(n)}{f(n) \log f(n)} = 0$ then $\text{DTIME}[f(n)] \not\subseteq \text{DTIME}[g(n)]$, and thus $\text{DTIME}[n^j] \not\subseteq \text{DTIME}[n^k]$ for $1 \leq k < j$. Cook [2] showed the latter result for non-deterministic time, which was improved by Seiferas, Fischer and Meyer [11]. All of these results relativize to all oracles.

Wilson [15] showed Δ_2^P has linear-size circuits with an appropriate oracle. We extend Wilson’s result to show $\text{BPTIME}[n]$ contains Δ_2^P relative to an oracle. However, his techniques fail to help prove our main theorem since they rely on the fact that languages in Δ_2^P can be forced by setting only a polynomial number of oracle answers for each input. BPP does not afford us that luxury.

Kannan [8] showed $\Sigma_2^P \cap \Pi_2^P$ does not have n^k -size circuits for any fixed k . Using standard techniques, one can show $\text{BPTIME}[n]$ has n^4 -size circuits. These results relativize to all oracles. Combining these facts with the above results, we find that there are oracles that collapse Δ_2^P to $\text{BPTIME}[n]$ and linear-size circuits and oracles that collapse BPP in $\text{BPTIME}[n]$ and linear-size circuits but such oracles do not exist for $\Sigma_2^P \cap \Pi_2^P$. The class $\Sigma_2^P \cap \Pi_2^P$ contains BPP [12] and Δ_2^P though the relationship between Δ_2^P and BPP is unknown.

One can get a trivial separation of $\text{BPTIME}[n]$ and $\text{BPTIME}[2^n]$ by simulating all possible coin tosses. Karpinski and Verbeek [9] improved this result to show $\text{BPTIME}[n^{\log n}]$ does not contain $\text{BPTIME}[2^{n^\epsilon}]$ for any $\epsilon > 0$.

4 Deterministic, Non-deterministic and Probabilistic Linear Time

Why does probabilistic computation behave differently than deterministic and non-deterministic computation for separating the time classes? In this section we will describe the proof techniques for separating $\text{DTIME}[n^2]$ from $\text{DTIME}[n]$ and $\text{NTIME}[n^2]$ from $\text{NTIME}[n]$ and show why these techniques fail for probabilistic computation.

The proof that $\text{DTIME}[n^2] \not\subseteq \text{DTIME}[n]$ works roughly as follows: Let M_1, M_2, \dots be an enumeration of all linear-time deterministic Turing machines. Define a machine M that on input i does the following: Simulate M_i on input i and accept if and only if M_i rejects. In quadratic time, M has more than enough time to simulate M_i on input i . However, if a linear-time machine M_j accepts $L(M)$ then we have a contradiction by the definition of M .

At first glance this proof seems to work for probabilistic machines. However, the proof fails because of the enumeration of the machines. If we choose a standard enumeration of the BPP machines, one of the M_i will accept with probability $1/2$ (for example the machine that just flips a coin and accepts if heads). Since a proper probabilistic machine must accept with probability below $1/3$ or above $2/3$, M will not accept any languages even though it runs in quadratic time. We could try to have an enumeration of proper probabilistic linear-time machines but such an enumeration may be computationally infeasible.

The deterministic proof does not work for the non-deterministic case either. Cook [2] proved $\text{NTIME}[n^2] \not\subseteq \text{NTIME}[n]$ using a translation lemma: If $\text{NTIME}[f(n)] \subseteq \text{NTIME}[g(n)]$ then for all “reasonable” superlinear h , $\text{NTIME}[h(f(n))] \subseteq \text{NTIME}[h(g(n))]$. A similar lemma holds for deterministic and probabilistic computation. Cook’s proof proceeds as follows: Assume $\text{NTIME}[n^2] = \text{NTIME}[n]$. Then by using the translation lemma $\text{NTIME}[n^4] = \text{NTIME}[n^2]$ and thus $\text{NTIME}[n^4] = \text{NTIME}[n]$. If we repeat this process k times, we get $\text{NTIME}[n^{2^k}] = \text{NTIME}[n]$ for any constant k . Cook shows by careful analysis that this process can actually be repeated $\log n$ times to get $\text{NTIME}[n^n] = \text{NTIME}[n]$ which can be shown false by diagonalization.

Even though the translation lemma holds, this proof still fails for probabilistic computation. The difficulty comes when we try to make a universal proper probabilistic machine. A universal proper probabilistic machine would be a proper probabilistic machine that can simulate other proper probabilistic machines; a very difficult task as we have already seen. Thus we can not keep the constants in check, and therefore can only repeat the translation process a constant number of times.

We will exploit these difficulties to create an oracle to collapse BPP to $\text{BPTIME}[n]$.

5 Proof of the Main Theorem

In this section, we prove the main theorem of this paper:

Theorem 5.1 *There exists an oracle A such that $\text{BPP}^A = \text{BPTIME}^A[n]$.*

In order to construct such an A , we encode within A the answers to whether BPP^A machines M accepts inputs w (for each M , and almost all w) in a way that a $\text{BPTIME}^A[n]$ machine can find the encoding and thus perform the simulation quickly. The difficulty that arises is that the BPP^A machine also has access to A and so can use it to try to ensure that however we try to encode the answers the simulating machine will be incorrect. This is in fact why analogous oracles for P and NP can not exist since the theorems of [6, 2] relativize for all oracles. Our ability to construct the oracle in this case rests on a balancing act between the power of probabilistic over deterministic

computation on one side, its still limited ability on the second side and the “forbidden” region of acceptance probability (between 1/3 and 2/3) lastly.

We present the proof in several sections as follows:

1. We describe the structure of the oracle.
2. We define influential, weak and encoding strings.
3. We examine a simple case in which machines and inputs only look at their own encodings.
4. We describe the encoding process for a restrictive BPP machine that uses the oracle non-adaptively.
5. We create a dependency graph for a machine and an input.
6. We process the dependency graph encoding that machine and input.
7. We generalize the proof to all BPP machines.

5.1 Structure of the Oracle

Let M_1, M_2, \dots be an enumeration of polynomial-time Turing machines that can make random choices and ask queries of an oracle. M_i^A designates the machine with index i using oracle A . Without loss of generality we can assume $M_i^A(x)$ runs in at most n^i steps. Clearly if $L \in \text{BPP}^A$ then $L = L(M_i^A)$ for some i with M_i^A being a proper probabilistic Turing machine.

Without loss of generality we can assume $M_i^A(x)$ flips all its coins before it does any other computation. Once $M_i^A(x)$ has flipped these coins it becomes a deterministic machine whose acceptance depends only on the oracle questions it asks. We call the computation after $M_i^A(x)$ flips its coins a *computation path* of $M_i^A(x)$. We also assume $M_i^A(x)$ flips the same number of coins on each computation path, so that each path has the same probability of occurring. Since $M_i^A(x)$ runs in polynomial time it can ask only a polynomial number of oracle queries on any computation path.

We will create the oracle A such that for each machine M_i^A one of the two following statements will be true:

1. M_i^A is improper.
2. There will exist a probabilistic linear-time machine S_i^A that accepts exactly the same language as M_i^A .

Suppose BPP^A contains a language L . Then some machine M_j^A must accept exactly the language L . The machine M_j^A must be proper, otherwise it could not accept any language at all, certainly not L . If we have set up the oracle A as described above then we have a linear-time machine S_j^A accepting the same language as M_j^A , *i.e.* the language L . Thus all of BPP collapses to probabilistic linear time under the oracle A .

We could encode whether $M_i^A(x)$ accepts by putting the string (i, x) in the oracle A if and only if $M_i^A(x)$ accepts. The linear-time machine S_i^A accepts on input x if (i, x) is in A . This idea fails to work because $M_i^A(x)$ can look at its own encoding of (i, x) in A .

Instead, we will encode in the oracle A whether or not M_i^A accepts using strings of the form (i, x, r) where r may be any of the $2^{5|x|}$ strings of length $5|x|$.

We say A *properly encodes* $M_i^A(x)$ if

1. If M_i^A accepts x then for at least $2/3$ of the possible r 's, $(i, x, r) \in A$.
2. If M_i^A rejects x then for at most $1/3$ of the possible r 's, $(i, x, r) \in A$.

We now have the simulating machine S_i^A do the following for input x :

1. Pick a random r of length $5|x|$.
2. Accept if (i, x, r) is in the oracle A .

Notice that if we have properly encoded A for M_i^A then S_i^A will accept exactly the same language as M_i^A .

5.2 Influential, Weak and Encoding Strings

For each $M_i^A(x)$ we will look at all the oracle queries it can possibly ask on every computation path—potentially a very large set of oracle strings. We call the oracle queries that affect the probability of $M_i^A(x)$ accepting by a certain non-negligible probability *influential strings*. The remaining oracle queries we call *weak strings*. We will show that $M_i^A(x)$ can not have too many influential strings.

More specifically we make the following definitions:

Definition: For a given machine M_i^A and input x , a string z is a β -*influential* string if $M_i^A(x)$ queries z on at least β of its computation paths.

Definition: For a given machine M_i^A and input x , a string z is a β -*weak* string if $M_i^A(x)$ queries z on some computation path but less than β of its computation paths.

We show a given $M_i^A(x)$ can not have too many β -influential strings by the following lemma:

Lemma 5.2 *For any machine M^A and input x , if M^A runs in time $t(n)$ then $M^A(x)$ has at most $\frac{t(n)}{\beta}$ β -influential strings.*

Proof Suppose $M^A(x)$ has c computation paths. If $M^A(x)$ has more than $\frac{t(n)}{\beta}$ β -influential strings then these influential strings account for more than $ct(n)$ oracle queries. However we can have at most $ct(n)$ oracle queries because at most $t(n)$ queries can be asked on each of the c computation paths. \square

From this point on we will let $\beta = 1/6$ and we will drop β and only refer to *influential* and *weak* strings.

We also use the term *encoding strings* to refer to the strings used to encode $M_i^A(x)$. More specifically:

Definition: An *encoding* string for a machine M_i^A and input x is a string of the form (i, x, r) with $|r| = 5|x|$.

Encoding strings for a machine M_i^A are all encoding strings for $M_i^A(x)$ for all $x \in \Sigma^*$. An encoding string in general is any encoding string for any machine-input pair, *i.e.* all strings of the form (j, y, r) with $|r| = 5|y|$.

5.3 A Simple Case

Let us examine the case when $M_i^A(x)$ only looks at strings of its own encoding, *i.e.* strings of the form (i, x, r) .

By lemma 5.2 there are at most $6n^i$ influential strings for $M_i^A(x)$ and thus they make up a very small fraction of all the possible (i, x, r) .

We will properly encode $M_i^A(x)$ in A as follows:

Initially we will set all of the strings of the form (i, x, r) to zero, *i.e.* not in A. We will then determine the probability of $M_i^A(x)$ accepting. There are three cases:

1. $M_i^A(x)$ accepts with probability less than one third.
2. $M_i^A(x)$ accepts with probability between one third and two thirds.
3. $M_i^A(x)$ accepts with probability greater than two thirds.

In the first case $M_i^A(x)$ rejects by definition and we have set less than a third of the (i, x, r) in A; in fact, none of the (i, x, r) are in A. In this case we have already properly encoded A.

In the second case $M_i^A(x)$, being improper, can not accept any languages, so we no longer need to encode A for M_i^A , even for other inputs.

In the third case, $M_i^A(x)$ accepts, but there are less than two thirds of the (i, x, r) in A. We will properly encode A using the following algorithm:

Pick a non-influential string from the (i, x, r) and put that string in the oracle A. Now only one string of the form (i, x, r) is in the oracle. Once again determine the probability of $M_i^A(x)$ accepting.

If $M_i^A(x)$ accepts with probability between one third and two thirds, then M_i^A is improper and we no longer need to encode M_i^A .

$M_i^A(x)$ can not accept with probability less than one third. Since we chose an weak string occurring on at most one sixth of the computation paths of $M_i^A(x)$ it can not change its probability by more than one sixth. However $M_i^A(x)$ would have to change its probability by more than one third to accept with probability less than one third.

Thus if $M_i^A(x)$ is still proper then it must accept with probability at least two thirds. Pick a new non-influential encoding string (the set of influential strings might have changed with the changed in the oracle) and add this new string to the oracle A along with the first string.

Once again either $M_i^A(x)$ is improper or it accepts with probability more than two thirds. We continue this process until M_i^A becomes improper or we have added to A enough strings such that two-thirds of the (i, x, r) are in A. We can continue this far since the influential strings form only a tiny fraction of all the encoding strings. At this point we will have properly encoded A for $M_i^A(x)$.

Unfortunately, this does not finish the proof because $M_i^A(x)$ may ask questions of other machines and inputs. To handle this case we must look carefully at the dependencies among the machine-input pairs and the encoding strings they query. The remainder of the proof handles these dependencies.

5.4 Order of Encoding

We also make the following restriction on how $M_i^A(x)$ works: We assume $M_i^A(x)$ does not make oracle queries dependent on the answers of previous queries. In other words, for any given computation path, $M_i^A(x)$ has a fixed set of oracle queries to decide whether to accept or reject. This set can have at most $|x|^i$ oracle queries, the running time of M_i^A on x . We will show in section 5.7

how to extend this proof to the general case where $M_i^A(x)$'s oracle queries can depend on previous queries.

For any given proper M_i^A , we need only properly encode oracle A for all but a finite number of inputs for M_i^A . The linear-time probabilistic machine S_i^A that merely chooses a random r and accepts if the oracle A contains (i, x, r) will work for all the inputs properly encoded by A. We create a linear-time machine T_i^A that accepts the same language as M_i^A by "hardwiring" the answers of the finite number of inputs not properly encoded by A.

We will use a *finite injury* argument. For a given M_i^A , we might not properly encode M_i^A on some finite number of inputs. We will determine which inputs for M_i^A we will not properly encode as the construction happens, making sure only that a finite number of inputs are not properly encoded. For example, suppose we can set some finite subset of the encoding strings of M_i^A in order to make M_j^A improper for $i > j$. Then we do not have to consider M_j^A again. Since there are only $i - 1$ machines with a lower index than i , we can only set the encoding strings in this way for a finite number of times.

We will encode machine-input pairs in the following order: First pick any total ordering of the inputs with the property that $|x| < |y|$ implies $x < y$. For each input x , we will encode machines $M_1, M_2, \dots, M_{\log \log \log n}$. Thus for machine i , we will not encode any of the finite number of inputs of size smaller than 2^{2^i} .

For each $M_i^A(x)$ during the process of the proof we will be able to either

1. Set enough of the oracle A to determine whether $M_i^A(x)$ accepts and appropriately set the encoding strings of $M_i^A(x)$ in A.
2. Make M_i^A improper.
3. Make M_j^A improper for some $j < i$.

At all times we carefully set the oracle strings in A so as not to use too many encoding questions for any machine and/or input except for finite injury in cases 2 and 3. Lemma 5.7 guarantees that we will succeed.

We encode $M_i^A(x)$ in two phases. First we will determine which encoding strings $M_i(x)$ depends on. Then we properly encode these machines until we have encoded $M_i^A(x)$. We do this through use of a dependency graph.

Before we encode any machines we set all oracle strings not used for encoding to zero, *i.e.* all non-encoding strings and all encoding strings (j, y, r) with $|y| < 2^{2^{2^j}}$.

Since we encode in order of input size, when we process machine-input pair (i, x) we have previously determined all of the following oracle strings:

1. Encoding strings of all machines-input pairs (j, y) with $|y| < |x|$ since these string have been previously encoded.
2. Encoding strings of all machines previously made improper.

As we will see in this proof, we may have determined strings in A in addition to those listed above.

5.5 Creating the Dependency Graph

We will create a finite *dependency graph* to help us properly encode $M_i^A(x)$. Each node of the dependency graph is an ordered pair $\langle j, y \rangle$ where j is an index of a machine and y is an input. Directed edges go from $\langle j, y \rangle$ to $\langle k, z \rangle$ if $M_j^A(y)$ ask oracle strings which are encoding strings of

$M_k^A(z)$. We call j the *index* of node $\langle j, y \rangle$ and y the *input*. We define the following order on the nodes of the dependency graph:

Definition: The *graph ordering* is a total ordering of the nodes such that $\langle j, y \rangle < \langle k, z \rangle$ if $j < k$ or $j = k$ and $y < z$.

In section 5.3 we assumed the dependency graph had only self loops, *i.e.* machines and inputs only look at their own encoding strings. If the dependency graph had no cycles, we could properly encode all the nodes by encoding the leaves and then work our way back to the root. However, the dependency graph may have cycles, in which case we will require more work to process this graph.

To create the dependency graph G for $M_i^A(x)$ we will start with the single node $\langle i, x \rangle$ and calling $\text{CREATE}(\langle i, x \rangle)$ using the following procedure:

Procedure $\text{CREATE}(\langle i, x \rangle)$:

Initialize G as the single node $\langle i, x \rangle$.

$\text{EXPAND}(\langle i, x \rangle)$

While there exists an expanded node $\langle k, z \rangle$ and an unexpanded node $\langle j, y \rangle$

such that $j < k$ and either $|y| < |z|$ or there exists an edge from $\langle k, z \rangle$ to $\langle j, y \rangle$

$\text{EXPAND}(\langle j, y \rangle)$

Procedure $\text{EXPAND}(\langle j, y \rangle)$:

For all $\langle k, z \rangle$ such that an encoding string of $M_k^A(z)$ is queried by

$M_j^A(y)$ on some computation path do

Add node $\langle k, z \rangle$ to G if it's not already there

Add an edge from $\langle j, y \rangle$ to $\langle k, z \rangle$

An *expanded* node of G is a node $\langle j, y \rangle$ such that $\text{EXPAND}(\langle j, y \rangle)$ was called during the creation of G .

We will see this graph contains the structure of the recursive encodings necessary to encode $M_i(x)$. A node $\langle j, y \rangle$ *depends* on node $\langle k, z \rangle$ if there is an edge from $\langle j, y \rangle$ to $\langle k, z \rangle$. Likewise, $M_j^A(y)$ *depends* on $M_k^A(z)$ if $\langle j, y \rangle$ depends on $\langle k, z \rangle$.

Note that all nodes of G represent machines with as yet undetermined encoding strings, strings that we have yet to decide whether or not to place them in A . If $\langle j, y \rangle$ is a node of G then $1 \leq j \leq \log \log \log n$ and $|y| \geq n$. If $\langle j, y \rangle$ is an expanded node of G other than $\langle i, x \rangle$ then $j < i$.

Lemma 5.3 *If $\langle j, y \rangle \in G$ then $|y| < n^{\log n}$.*

Proof A Turing machine can not ask an oracle question longer than the amount of time it has to write it down. If $\langle k, z \rangle$ is a node of G then M has running time at most $|z|^{\log \log \log |z|}$ since $k \leq \log \log \log n$ and M_k runs in time at most n^k for inputs of size n . We only create expanded nodes towards smaller indices and the only way we can get to a larger input is if there is a direct arrow from a higher index expanded node. If $f(j)$ is the maximum size of the inputs of all the nodes of index at least j then $f(j-1) \leq f(j)^{\log \log \log f(j)}$. We know $f(i) = n$ since $\langle i, x \rangle$ is the only expanded node of G of index i . We can bound the recurrence using lemma 5.4 to show $f(j) < n^{\log n}$ for all j such that $1 \leq j \leq i$. Thus for all expanded nodes $\langle j, y \rangle$, $|y| < n^{\log n}$. Since we create unexpanded nodes only from expanded nodes, we can actually show for all nodes $\langle j, y \rangle$ of G , $|y| < n^{\log n}$. \square

Lemma 5.4 *Suppose we have a function $f(j)$ with the following conditions:*

1. $f(i) = n$ for some i , $1 \leq i \leq \log \log \log n$

2. $f(j-1) \leq f(j)^{\log \log \log f(j)}$ for all j , $1 \leq j \leq i$

Then $f(j) < n^{\log n}$ for all j , $1 \leq j \leq i$.

Proof By induction on j . True for $j = i$ by assumption. Assume $f(k) < n^{\log n}$ for all k , $j < k \leq i$. We will show the lemma true for j .

For all k with $j < k \leq i$,

$$f(k-1) \leq f(k)^{\log \log \log f(k)} < f(k)^{\log \log \log (n^{\log n})} = f(k)^{g(n)}$$

for $g(n) = \log \log \log (n^{\log n}) = 1 + \log \log \log n$. Then we have

$$f(j) \leq f(i)^{g(n)^{i-j}} \leq n^{g(n)^{\log \log \log n}}$$

by repeatedly exponentiating $f(i) = n$ by $g(n)$ for $i - j$ times. To bound the exponent, we note

$$\log(g(n)^{\log \log \log n}) = (\log \log \log n)(\log g(n)) = (\log \log \log n)(\log(1 + \log \log \log n)) < \log \log n.$$

Thus $g(n)^{\log \log \log n} < \log n$ and thus $f(j) < n^{\log n}$ \square

Since j and $|y|$ is bounded, the graph G is finite.

5.6 Processing the Dependency Graph

After we create the dependency graph G for $M_i^A(x)$ as described above, we will process each expanded node $\langle j, y \rangle$ of G from smallest node to $\langle i, x \rangle$ in the graph ordering described in section 5.5, possibly changing G at each step.

As we process each node $\langle j, y \rangle$ of G we will either

1. Set enough of the oracle A to determine $M_j(y)$. We then encode $M_j(y)$ in A appropriately with its unset encoding strings. We remove node $\langle j, y \rangle$ from G , along with all its associated edges.
2. Make M_j improper. At this point we stop trying to encode $M_i(x)$, invoking the finite injury argument since $j \leq i$.
3. Put node $\langle j, y \rangle$ on *hold*. We will restructure the dependency graph so $\langle j, y \rangle$ only has edges to nodes $\langle k, z \rangle$ with $k > j$ and $|y| > 2|z|$. These held nodes are the key to handling cycles in the dependency graph.

As we process each node $\langle j, y \rangle$ of G all previously processed nodes not removed will be on hold. These held nodes depend only on a larger index or input than $\langle j, y \rangle$. We will combine node $\langle j, y \rangle$ with all the held nodes of smaller indices it depends on into one single node. We will show the probabilistic machine corresponding to this node can not have too much power. This machine-input pair also depends only on encoding strings of $M_k^A(z)$ with $k \geq j$. We show we can apply one of the three actions above and then we process the next node. When we process node $\langle i, x \rangle$ it can not be put on hold because $|x| = n$ and no nodes of G have input of length less than n . We have then succeeded in encoding $M_i^A(x)$ making it improper or making some machine with a smaller index improper.

If when we process node $\langle j, y \rangle$ we decide to place $\langle j, y \rangle$ on hold and if a previously held node $\langle k, z \rangle$ has an edge to $\langle j, y \rangle$ we will “forward” the edges of $\langle k, z \rangle$ to the nodes $\langle j, y \rangle$ depend on to keep the following invariant: After we process a node of the dependency graph, all held nodes only have edges to unprocessed nodes.

We now give a more precise description of how we process node $\langle j, y \rangle$:

1. Convert $M_j(y)$ to $\hat{M}_j(y)$, a new machine that combines $M_j(y)$ with all the machines related to the held nodes of a smaller index that $\langle j, y \rangle$ depends on. We describe the combining process below. We remove all edges from $\langle j, y \rangle$ to the held nodes. The machine $\hat{M}_j(y)$ depends only on encodings relating to nodes of the same or larger index. [Backward Combine]
2. We try to set the oracle to make $\hat{M}_j^A(y)$ accept with an improper probability. At this point we have finished processing this dependency graph.
3. If unsuccessful, we examine the influential and weak strings of $\hat{M}_j(y)$. We argue that the weak strings can not affect the output of $\hat{M}_j^A(y)$, and we remove the related edges from G . We set all the influential strings of encodings of inputs at least as long as $|y|/2$ to zero, also removing those edges from G . Recompute every machine related to a held node with an edge to one of these influential strings. If we have now determined those machines, properly encode the oracle and remove those nodes from G .
4. If there are no more influential strings then we have determined $\hat{M}_j^A(y)$ (Lemma 5.9). Properly encode the oracle. Recompute every machine related to a held node with an edge to $\langle j, y \rangle$. If we have now determined those machines, properly encode the oracle and remove those nodes from G . Finally, remove node $\langle j, y \rangle$ from G and all its remaining associated edges. We then process then next node of G .
5. If influential edges remain, we then place node $\langle j, y \rangle$ on hold. First we combine $\hat{M}_j(y)$ with all previously processed nodes it still depends on (nodes $\langle j, z \rangle$ with $|z| < |y|/2$). [Downward Combine]
6. After we process all nodes of G with index j , we then combine every machine with a hold on a node $\langle j, z \rangle$ with $\hat{M}_j(z)$ replacing edges to $\langle j, z \rangle$ with edges to the edges of node $\langle j, z \rangle$. [Forward Combine]

5.6.1 Combining Machines

We combine a machine M with a set of machines \mathcal{M} as follows: We simulate M choosing the random coin tosses at random. When M asks an oracle query about the encoding of a machine $M' \in \mathcal{M}$, we would like to respond with whether M' accepts. We could simulate M' and respond to the oracle query with the output of M' . However with any given set of coin tosses, M may ask several oracle queries and each simulation could fail with probability up to one third. Thus we could have a large probability of getting wrong answers on some of the oracle queries. Instead we simulate M' m times independently, where m is the length of the input of the machine to be combined, and take the majority answer, which gives us an exponentially small error.

We call this combined machine \hat{M} with a running time of $h(n)$. Suppose M runs in time $f(m)$, the longest question M asks of a machine in \mathcal{M} is of length $l(m)$ and the maximal running time of a machine in \mathcal{M} is $g(m)$. We can bound $h(m)$ by $mf(m)g(l(m))$: A maximum of $f(m)$ oracle queries of length at most $l(m)$ each simulated m times. For backward combining we will have $l(m) \leq f(m)$, the running time of M so $h(m) \leq mf(m)g(f(m))$. For downward and forward combining steps we will have $l(m) \leq m/2$ and thus $h(m) \leq mf(m)g(m/2)$.

Lemma 5.5 *If the combined machine is encoded into the oracle, it will accept the same language as the though it used the oracle queries of A directly.*

Proof By the construction of the oracle, unless the finite injury argument is invoked or we do a recomputation in step 3, all queries we make will be eventually set to the correct value of whether or not the machine accepts.

The only potential problem happens if a simulation outputs a different answer than correct value of whether or not the machine accepts. The simulation is designed to let that happen with an exponentially small chance of error. Lemma 5.6 shows that every computation path has a subexponential length and thus the error caused can only affect a tiny number of computation paths and thus the overall probability only slightly. \square

Lemma 5.6 *Let $t(k, m, j)$ be the maximum running time of the combined machine $\hat{M}_k^A(y)$ with $|y| = m$ after we have processed all nodes of G of index j . For all but a finite number of m , we have*

$$t(k, m, j) \leq m^{(\log m)^{2^j}} \leq m^{(\log m)^{\log \log m}}$$

Proof By the construction of the dependency graph G we have that $\log \log \log m$ bounds both k and j . The second inequality follows from this fact.

The first inequality follows from the following recurrence:

1. $t(k, m, j) = m^k$ for $k > j$ (never combined)
2. $t(k, n, j) = n^k$ (never combined)
3. $t(k, m, j) = t(k, m, j-1) * t(j, \lfloor m/2 \rfloor, j) * m$ for $k < j$ (forward combine)
4. $t'(m, j) = m^j * (\max_{k < j} (t(k, m^j, j-1)) * m$ (backward combine)
5. $t(j, m, j) = t'(m, j) * t(j, \lfloor m/2 \rfloor, j) * m$ (downward combine)

We will prove the first inequality by induction. The inequality clearly holds for cases 1 and 2. We will show the inequality holds for case 4 and 5. Case 3 has a similar proof.

Combining cases 4 and 5 and applying the inductive argument we get

$$t(j, m, j) \leq m^{j+2} * (m^j)^{(\log m^j)^{2^{j-1}}} * (m/2)^{(\log m/2)^{2^j}}$$

Noticing that $2^{(\log m/2)^{2^j}}$ dominates m^{j+2} we need only show

$$m^{j(j \log m)^{2^{j-1}}} * m^{((\log m)-1)^{2^j}} \leq m^{(\log m)^{2^j}}$$

Letting $r = \log m$ and combining exponents means we need only show

$$j(jr)^{2^{j-1}} + (r-1)^{2^j} \leq r^{2^j}$$

We have $j \leq \log \log \log m$ so $j \leq \log \log r$. We then have

$$j^{2^{j-1}} \leq j^{2^j} = (2^{2^j})^{\log j} \leq r^{\log j}$$

We also have

$$(r-1)^{2^j} \leq r^{2^j} - 2^j r^{2^j-1} + 2^{2^j} r^{2^j-2}$$

Since $j r^{\log j} r^{2^{(j-1)}}$ is clearly much less than $2^j r^{2^j-1}$ the lemma follows. \square

5.6.2 Analysis

In this subsection we will argue that the above procedures will properly encode $M_i^A(x)$ in our oracle or the finite injury argument will apply. Then the construction given in Section 5.4 will create the oracle we desire.

Lemma 5.7 *After we process the dependency graph G for $M_i^A(x)$ we will either have properly encoded $M_i^A(x)$ or made some machine M_j improper for some $j \leq i$.*

Proof We will show below that we can always process the dependency graph as described in Section 5.6. Node $\langle i, x \rangle$ can not be put on hold since there are no nodes with an input of length less than $|x|$ in G . The lemma follows. \square

Lemma 5.8 *We can always properly encode $\hat{M}_j^A(y)$ as required when processing G as described in Section 5.6.*

Proof Let $m = |y| \geq n$. Except for the finite injury case, the only encoding strings of $M_j^A(y)$ are those influential strings of inputs of length at most twice m . By Lemma 5.2, the number of influential strings set by any of these (combined) machines is at most six times its running time. Thus any machine $M_k^A(z)$ that has set its influential strings in the encoding strings of $M_j^A(y)$ must have $k \leq \log \log \log n \leq \log \log \log m$ and $|z| \leq 2|y| = 2m$. By Lemma 5.6 the running time of $M_k^A(z)$ is at most $(2m)^{(\log 2m)^{\log \log 2m}}$ and thus the number of influential strings is six times this amount. There are a total of $\log \log \log n$ possible such machines on at most 2^{2m} inputs for a total number of strings set of:

$$(\log \log \log n)(2^{2m})(6(2m)^{(\log 2m)^{\log \log 2m}}) < 2^{3m}$$

which is less than the 2^{-2m} of the 2^{5m} encoding strings available. The unset strings consist of more than $1 - 2^{-2m} > \frac{2}{3}$ of the encoding strings available and thus we can properly encode whether or not $M_j^A(y)$ accepts

Lemma 5.9 *If we can not set the oracle to make $\hat{M}_j^A(y)$ accept with an improper probability and we have set all the influential strings of $\hat{M}_j^A(y)$ then we have determined $\hat{M}_j^A(y)$.*

Proof Identical to the proof given in Section 5.3. \square

5.7 Generalizing the Proof for All BPP Machines

We give a sketch of the modifications of this proof necessary if we allow the probabilistic machines to base their oracle queries on answers to previous oracle queries. The computation paths on a machine M will now contain branches both for coin tosses and oracle queries. We create G using all possible branches of both types. When we process each node $\langle j, y \rangle$ of G we do the following:

1. We create $\hat{M}(j, y)$ as before.
2. We will try all possible oracle settings of A to try to make $\hat{M}_j^A(y)$ improper. If we succeed then we no longer need to continue processing G . Note that this process will only affect nodes (k, z) for $k \geq j$.

3. Look at the machine where it takes undetermined oracle query branches as though they were zero. We argue the weak strings of this model can not affect the output of the original machine. As before, we set to zero influential strings of an encoding of an input of length at least $|y|/2$. If there are other influential strings, we will put the machine on hold and combine the appropriate machines.
4. We encode $M_j^A(y)$. In this case we may have introduced ones into A. We then recompute as well as combine all machines that had a hold on $\langle j, y \rangle$.

The remainder of the proof follows as before.

6 Other Results

Corollary 6.1 $BPP = ZPTIME[n] = RTIME[n] \subseteq NTIME[n]$ for some oracle A .

Proof Note in the proof of the main theorem we only introduce ones into the oracle when we make a machine improper or when we encode a machine. Except for a finite number of injuries, if $M_i^A(x)$ rejects then we will encode $M_i^A(x)$ entirely with zeros. A linear-time machine that picks an encoding string at random will never accept in this case so the oracle we created actually collapses BPP to $RTIME[n]$. If $BPP = RTIME[n]$ then $BPP = \text{co-}RTIME[n]$ and thus $BPP = ZPTIME[n]$. \square

Corollary 6.2 For some oracle A , BPP has linear-size circuits.

Proof Fix a probabilistic machine M_i . Look at all 2^n inputs of length n . For any (i, x, r) all but 2^{-2n} of the r 's correctly encode $M_i^A(x)$. Thus there exists some r' such that (i, x, r') correctly encodes $M_i^A(x)$ for all x of length n . We can easily build a linear-size circuit that determines if $(i, x, r') \in A$. \square

Theorem 6.3 For some oracle A , $\Delta_2^P \subseteq BPTIME[n]$.

Proof Let M_1, M_2, \dots be a list of Δ_2^P machines, *i.e.* polynomial-time deterministic machines with access to an NP oracle. We set up the oracle A as in the proof of the main theorem but we encode our machines in a different way.

Look at the computation of $M_i(x)$ on undetermined oracle queries. Using techniques of Wilson [15] we note there exists a setting of polynomial many oracle questions that determines $M_i^A(x)$. We set the oracle in this way to determine $M_i^A(x)$. We then encode (i, x, r) properly. There are $\log \log \log n$ machines on 2^n inputs each requiring at most a polynomial number of oracle queries to be set. We have hardly used any of the 2^{5n} oracle questions available. Thus we will have no problem encoding $M_i^A(x)$. \square

Note $\Delta_2^P \not\subseteq RTIME[n]$ under any oracle, because $\Delta_2^P \subseteq RTIME[n]$ would imply $NP = NTIME[n]$, which contradicts Cook's result [2].

Clearly there exists an oracle such that $BPTIME[n]$ does not contain BPP or Δ_2^P created by diagonalizing with strings too long for the $BPTIME[n]$ machines to read. We can create similar oracles such that $NTIME[n]$ does not contain BPP and BPP does not have linear-size circuits. In fact these results hold for random and generic oracles.

We now show the impossibility of simultaneously collapsing both BPP and NP to probabilistic linear time even though we can do either individually.

Theorem 6.4 The following two statements can not both be true:

$$\begin{aligned} \text{BPP} &= \text{BPTIME}[n] \\ \text{NP} &\subseteq \text{BPTIME}[n] \end{aligned}$$

Proof Assume both statements are true. Then NP would be contained in BPP. By a result of Zachos [16], $\text{NP} \subseteq \text{BPP}$ implies the entire polynomial-time hierarchy collapses to BPP and thus to $\text{BPTIME}[n]$. Then all languages in the polynomial-time hierarchy have n^4 -size circuits which contradicts Kannan's result [8] that $\Sigma_2^P \cap \Pi_2^P$ does not have n^k -size circuits for any fixed k . \square

Note this proof relativizes; thus no oracle A exists that collapses both BPP and NP to probabilistic linear time even though we can collapse each individually.

Any complexity class that contains both NP and BPP can not be collapsed to $\text{BPTIME}[n]$. In particular, the set of all languages accepted by interactive proof systems must contain languages not recognizable in probabilistic linear time since IP [5, 1] contains both NP and BPP.

We can use the recent series of results showing $\text{IP}=\text{PSPACE}$ [10, 13] to get a stronger result answering an open question raised in a previous version of this paper. Let $\text{IPTIME}[f(n)]$ be the class of languages accepted by an interactive proof system with a verifier restricted to run in $O(f(n))$ time. For example, $\text{IP}=\bigcup_{k>0}\text{IPTIME}[n^k]$.

Theorem 6.5 *There exists a language L in IP but not in $\text{IPTIME}[n]$.*

Proof A careful analysis of the proof that PSPACE contains IP shows $\text{IPTIME}[n]\subseteq\text{DSPACE}[n^2]$. However we know that PSPACE properly contains $\text{DSPACE}[n^2]$ [6] and $\text{IP}=\text{PSPACE}$. The theorem follows. \square

Fortnow and Lund [4] have shown a strong time and space hierarchy for public-coin interactive proof systems.

Also, IP does not have linear-size circuits because PSPACE does not [8]. Since the proof that $\text{IP}=\text{PSPACE}$ does not relativize, there could be an oracle A such that $\text{IP}^A=\text{IPTIME}[n]^A$ or IP^A has linear-size relativized circuits.

7 Conclusions and Further Research

This paper shows the collapse of both BPP and Δ_2^P to both $\text{BPTIME}[n]$ and linear-size circuits with appropriate oracles. Also, $\Sigma_2^P \cap \Pi_2^P$ can not be collapsed to either $\text{BPTIME}[n]$ or linear-size circuits. Since $\text{IP}=\text{PSPACE}$, IP properly contains $\text{BPTIME}[n]$ and IP does not have linear-size circuits.

Some of these questions remain open for the class AM of interactive proof systems limited to a constant number of rounds of interaction. By theorem 6.4, AM properly contains $\text{BPTIME}[n]$ since AM contains both NP and BPP. Still open is whether AM has linear-size circuits and/or AM properly contains AM with a linear-time verifier under any reasonable definition of a constant-round interactive proof system with a linear-time verifier.

Ideally the questions in this paper should be resolved in the unrelativized world. Although this paper shows solving such problems will be hard, there have recently been results, like $\text{IP}=\text{PSPACE}$, that do solve problems previously thought hard for similar reasons. However it will require techniques that do not relativize to solve the problems in this paper. We conjecture none of the collapses occur in the unrelativized world.

This paper introduces several techniques for oracle construction. These methods may be useful in the construction of oracles for other problems.

This paper gives an example of how probabilistic computation appears different than deterministic and non-deterministic computation. Perhaps there exist other results that may help to

understand the differences and similarities in the nature of probabilistic computation and other types of computation such as interactive proof systems.

Acknowledgments

The authors would like to thank Eric Schwabe, Richard Beigel, Eric Allender and Rutger Verbeek for their useful comments on this paper. We would also like to thank the anonymous referees for their many helpful comments.

References

- [1] L. Babai, *Trading Group Theory for Randomness*, Proc. 17th STOC(1985), pp. 421-429.
- [2] S. Cook, *A Hierarchy for Nondeterministic Time Complexity*, JCSS 7(1973), pp.343-353.
- [3] Gill, J., *Computation Complexity of Probabilistic Turing Machines*, SIAM J. Comp. 6(1977), pp. 675-695.
- [4] L. Fortnow and C. Lund, *Interactive Proof Systems and Alternating Time-Space Complexity*, Theoretical Computer Science, to appear. Previous version in Proc. of the 8th STACS, Springer-Verlag LNCS 480(1991), pp. 263-274.
- [5] S. Goldwasser, S. Micali and C. Rackoff, *The Knowledge Complexity of Interactive Proof Systems*, SIAM Journal on Computing 18(1989), pp. 186-208. Extended abstract available in Proc. 17th STOC (1985), pp. 291-304.
- [6] J. Hartmanis and R. Stearns, *On the Computational Complexity of Algorithms*, Trans. AMS 117(1965), pp. 285-306.
- [7] F. Hennie and R. Stearns, *Two-tape Simulation of Multitape Turing Machines*, JACM 13(1966), pp. 533-546.
- [8] R. Kannan, *Circuit-Size Lower Bounds and Non-reducibility to Sparse Sets*, Information and Control 55(1982), pp. 40-56.
- [9] M. Karpinski and R. Verbeek, *Randomness, Provability, and the Separation of Monte Carlo Time and Space*, LNCS 270(1988), pp. 189-207.
- [10] C. Lund, L. Fortnow, H. Karloff and N. Nisan, *Algebraic Methods for Interactive Proof Systems*, J. ACM, to appear. Earlier version in Proc. 31st FOCS(1990), pp. 2-10.
- [11] J. Seiferas, M. Fischer and A. Meyer, *Separating Nondeterministic Time Complexity Classes*, JACM 25(1978), pp. 146-167.
- [12] M. Sipser, *A Complexity Theoretic Approach to Randomness*, Proc. 15th STOC (1983), pp. 330-335.
- [13] A. Shamir, *IP=PSPACE*, J. ACM, to appear. Earlier version in Proc. 31st FOCS(1990), pp. 11-15.
- [14] S. Solovay and V. Strassen, *A Fast Monte-Carlo Test for Primality*, SIAM J. Comp. 6(1977), pp.84-85.

- [15] C. Wilson, *Relativized Circuit Complexity*, JCSS 31(1985), pp. 169-181.
- [16] S. Zachos, *Probabilistic Quantifiers, Adversaries, and Complexity Classes: An Overview*. LNCS 223(1986), pp. 383-400.