Randomization and Derandomization in Space-Bounded Computation *

Michael Saks†
Department of Mathematics
Rutgers University
New Brunswick, NJ 08903

Abstract

This is a survey of space-bounded probabilistic computation, summarizing the present state of knowledge about the relationships between the various complexity classes associated with such computation. The survey especially emphasizes recent progress in the construction of pseudorandom generators that fool probabilistic space-bounded computations, and the application of such generators to obtain deterministic simulations.

1 Introduction

Inspired in part by the then-surprising use of randomness in algorithms for "non-random" problems such as primality testing ([47, 40]), probabilistic computation emerged as a major subfield of complexity theory during the late 1970's. Beginning with Gill's seminal paper [15], researchers defined models and built the foundations for a rigorous study of probabilistic computation and, in particular, of probabilistic time- and space-bounded complexity classes. Despite the considerable success in understanding the structure of and relationships between such classes, the two central questions, "Does randomness ever provide more than a polynomial advantage in time over determinism?" and "Does randomness provide more than a constant factor advantage in space over determinism?", remain unsolved. However, recent progress in the realm of space-bounded computation gives promise that a negative answer to (at least one form of) the second question may be in sight, or, short of that, that some less sweeping but still fundamental questions are accessible. This survey of probabilistic space-bounded computation aims at providing a thorough overview of the developments in this area, and at highlighting some of the many questions that remain.

Early work on space-bounded probabilistic complexity classes was based on connections between these classes and matrix computations. These connections led to two of the main results: that (unbounded error) probabilistic space s is contained in deterministic space s^2 [9], and that the power of unbounded error probabilistic space s is not diminished by imposing a time bound of $2^{O(s)}$ [22].

Much of the more recent progress in the area grew out of efforts to place the theory of pseudorandom generation on a firm theoretical foundation, which, motivated largely by considerations from cryptography, began in the mid 1980's. A fundamental insight here, originating in [50] and expanded upon and systematized in [36, 29], is the connection between "hardness" and "randomness": the existence of problems whose instances can be generated efficiently, and that are sufficiently hard to approximate within a particular computation model, can be used as the basis for a pseudorandom generator. Such generators can then be used to construct deterministic simulations of randomized computation within that model. Lacking any provable hardness results for time-bounded computation, this insight led to a number of "conditional" results of the form: if two time-bounded complexity classes A and B are distinct, then probabilistic time-bounded class C can be simulated in deterministic time-bounded class D.

Now, inasmuch as space-bounded computation suffers from the same absence of provable hardness results as time-bounded computation does, one might at first expect that this approach would be limited to conditional results here as well. However, it turns out that to construct provably good pseudorandom generators for space-bounded computation, it is enough to prove hardness results for a model of space-bounded computation in which the input is accessible only by a

*This paper appeared in the Proceedings of the 11th Annual Structure in Complexity Theory Conference, 1996, copyright IEEE.

†Supported in part by NSF contracts CCR-9215293 and STC–91–19999 and by DIMACS

one-way tape. This observation opened the way for a sequence of papers [2, 7, 30, 19, 37, 6], presenting ingenious constructions of pseudorandom number generators that can be proved unconditionally to look random to space-bounded machines. These constructions provide the basis for some significant deterministic simulations of randomness: that any bounded error randomized log space, polynomial time computation can be simulated by a deterministic polylog space, polynomial time computation [31], and also by an $O((\log n)^{3/2})$ space deterministic computation ([42]), and that if the randomized computation uses only polylog many random bits then it has a log space deterministic simulation [37].

In this survey, we focus on language membership problems and on complexity classes corresponding to space functions $s(n)$ that are at least $\log n$ (thus omitting the notable body of work on very small space classes and probabilistic automata, e.g, [39, 13, 17, 11, 24, 23]). Within these restrictions, the aim is to be reasonably comprehensive, and apologies are offered in advance for the inevitable omissions.

Section 2 presents definitions of the relevant models and complexity classes and their connection with Markov chains and matrix computations. Section 3 summarizes the known relationships between these complexity classes and some of the main open problems. Section 4 provides a closer look at the recent developments in pseudorandom generators for space bounded computation, and their application to deterministic simulation. A final section briefly considers some other related research directions.

2 Models, complexity classes and equivalent problems

2.1 Some Basics

In this paper, deterministic Turing machines (DTMs) have three tapes: a read-only input tape with a two-way head, a work tape with a two-way head, and a write-only output tape with a one-way head. Nondeterministic and probabilistic Turing machines (PTMs) have an auxiliary read-only tape with a one-way head that, for non-deterministic machines, contains an infinite string of non-deterministic bits, and for probabilistic machines, contains an infinite string of unbiased independent bits. Informally, we think of these bits as "flips" of a random coin, and the value of past coin flips can only be recalled if they have been written on the work tape.

A *configuration* of a machine consists of (1) the contents of the work tape and the position of the head (2) the position of the head on the input tape, and (3) the state of the final state control. Note that it does not include any information about the output tape or (for non-deterministic or randomized machines) the auxiliary tape.

The *execution* $M(x)$ of a DTM M on an input x is the (possibly infinite) sequence of configurations through which M passes on input x. For a PTM M, the execution of $M(x)$ is a random process, which depends on the string stored in the auxiliary tape. Under this probability distribution, all events associated with an execution, such as "the machine does not halt" are assigned a probability. For a PTM M and input x, a *run* of $M(x)$ is a specific outcome of this random process, i.e., a *run* is obtained by specifying a particular setting of the bits on the random tape. For a DTM, a run and an execution are synonomous.

We will usually be considering Turing Machines for language membership. Without loss of generality, we will usually assume that such a machine either produces no output (and either halts or not), or outputs a single 1 and then halts immediately. If a run of $M(x)$ produces a 1 output, then we say that the run accepts x. For a PTM M, we define $p_M(x)$ to be the probability that $M(x)$ accepts x.

We adopt the following notions of time and space for probabilistic computations. For functions $s(n)$ and $t(n)$, a PTM M is said to operate in space $s(n)$ if for every input x, every run of $M(x)$ requires at most space $s(|x|)$, and is said to operate in time $t(n)$ if for every input x the *expected* time of $M(x)$ is at most $t(n)$. As explained in [15], there are some problems with these definitions, and the less restrictive definitions given there are, in general, better behaved. However, for our purposes, the definitions given above will serve just as well. Also, there are the usual technicalities that s and t should be suitably well-behaved functions (see, e.g., the discussion of "proper" complexity functions in [38]), and we always assume that the functions s and t satisfy whatever conditions are necessary.

The usual definition of the space of a run is the number of distinct cells that are accessed on the work tape. We adopt the following convenient (and common) abuse of terminology: we will use the term space to denote the number of work tape cells accessed *plus* the number of bits to encode the state of the finite state control, the positions of the heads on the input tape and work tape. This abuse allows us to say that a run using space S has at most 2^S distinct configurations. Given our restriction to space functions $s(n)$ that are at least $\log(n)$, this notion of space differs from the usual one by at most a multiplicative constant.

We will indulge in a few other notational abuses. We will be very casual about constant factors: when we say s, we may well mean $\theta(s)$ or $O(s)$. Also, if machine M has space bound s, and x is an input we may say that the execution $M(x)$ operates in space s, when properly we should say that $M(x)$ operates in space $s(|x|)$.

2.2 Randomized space-bounded complexity classes

Notational conventions for denoting probabilistic space-bounded complexity classes have not been completely standardized, and there are some incompatibilities in the choice of notation by various researchers. The most serious instance is the complexity class RL, which depending on the author of the paper, has one of two distinctly different meanings. This section proposes a reasonably comprehensive and consistent set of notation that does not differ too much from common usage; the conventions are inspired by and adapted from the taxonomy proposed in [8] for randomized log space complexity classes, with some modifications that seem appropriate for extending these definitions to general space-bounded classes. At the end of this section we include a short discussion relating the notation here to notation elsewhere.

If M is a PTM, the language L computed by M is the set of strings x such that $p_M(x) > 1/2$. Probabilistic Turing machines can be classified according to their behavior:

- M operates in *bounded error* if $p_M(x) \leq 1/3$ for all $x \notin L$.
- M operates with *1-sided error* if $x \notin L$ implies that $p_M(x) = 0$.
- M *halts almost surely* if for all x, the probability that $M(x)$ does not halt is 0.
- M *halts absolutely* if for all x, M halts on every run (i.e., for every setting of the random tape). For example, a machine that reads random bits and halts the first time it reads a 1, halts almost surely, but not absolutely.

The first two properties are very familiar and arise in the study of both time- and space-bounded computation. The distinction between the third and fourth properties can be shown to be unimportant in the context of time-bounded complexity, but, as will be seen, is crucial in the context of space-bounded complexity. It turns out that the condition "halts almost surely" can be imposed without loss of generality (see Theorem 2.1). However, the condition "halts absolutely", which appears at first to be only a slightly stronger condition, seems to make a huge difference in the power of randomized machines. This will be discussed in some detail in section 3.3.

These conditions on probabilistic machines give rise to eight randomized space-bounded complexity classes, listed in figure 1. The class $PrSPACE(s)$ (for "probabilistic space s") contains all languages computed by a space s probabilistic Turing machine. $BPSPACE(s)$ (for "bounded error probabilistic space s"), and $RSPACE(s)$ are, respectively, those languages computed by a space s machine that operate with bounded error and with one-sided error. The class $ZPSPACE(s) = RSPACE(s) \cap co - RSPACE(s)$ is the class of languages L such that both L and its complement $\bar{L}$ are computed by a space s one-sided error PTM. The notation $ZPSPACE$ stands for zero-sided error and is motivated by the following alternative characterization:

Proposition 2.1 *A language L is in $ZPSPACE(s)$ if and only if there is a PTM M satisfying: (i) on any input x, either M produces no output, or it outputs "accept", or it outputs "reject" (ii) for $x \in L$ it outputs "accept" with probability greater than 1/2 and never outputs "reject", and (iii) for $x \notin L$, it outputs "reject" with probability greater than 1/2 and never outputs "accept".*

For each of these classes, the corresponding class with subscript H is defined by modifying the definition of the class so that the accepting machine for the language is required to halt absolutely. We refer to these classes as the *halting classes* and to the others as the *non-halting classes*.

From the definitions it is clear that the containments indicated in the figure hold.

In addition to these classes and the standard classes $DSPACE(s)$ and $NSPACE(s)$, we will also refer to the class $SSPACE(s)$, of languages computed by a *symmetric Turing machine* [25] running in space s. For our purposes it suffices to know that any language in $SSPACE(s)$ can be reduced in $DSPACE(s)$ to an undirected (s,t)-connectivity ($USTCON$) problem for a graph on $2^{O(s)}$ vertices.

For the special case where $s(n) = \log n$, we write BPL, RL, SL, BP_HL, etc. for the various classes. By this convention, the class $PrSPACE(\log n)$ would be denoted PrL, but we will use the standard name PL for this class.

We will occasionally consider complexity classes corresponding to computation that is both time-

	Unbounded Error		*Bounded Error*		*One-sided Error*		*Zero-sided Error*
Non-halting	$PrSPACE(s)$	$\supseteq$	$BPSPACE(s)$	$\supseteq$	$RSPACE(s)$	$\supseteq$	$ZPSPACE(s)$
	$\cup\mid$		$\cup\mid$		$\cup\mid$		$\cup\mid$
Halting	$Pr_HSPACE(s)$	$\supseteq$	$BP_HSPACE(s)$	$\supseteq$	$R_HSPACE(s)$	$\supseteq$	$ZP_HSPACE(s)$

Figure 1: Probabilistic space-bounded complexity classes

and space-bounded. For $X \in \{BP, Pr, R, D, N, S\}$, $XTISP(t, s)$ denotes the class of languages recognizable by Turing machines of type X running within time bound $t(n)$ and space bound $s(n)$.

Remark: In most of the early papers on randomized computation from the late '70s and early 80s, the focus was on computations that do not necessarily halt absolutely, and the complexity class definitions were made for this case only. The definitions here of $PrSPACE(s)$, $BPSPACE(s)$ and $ZPSPACE(s)$ follow the usage introduced in these papers. The prefix VP appears in some of the early literature in place of the prefix R to denote one-sided error computation, but R seems to gradually have become the preferred choice.

As will be seen in Proposition 2.3, the condition of halting absolutely for space s computation is essentially equivalent to imposing a bound of 2^s on either the maximum or expected time of the computation. For the case $s = \log n$, the corresponding time bound is thus polynomial. In this case, notation for, e.g., one-sided classes has included $RSPACE^{poly}(\log n)$ and the more widely used RLP. The "sub H" notation proposed here seems more natural and convenient for general values of the space parameter.

The fact (see Proposition 3.1 below) that the class $RSPACE(s)$ coincides with $NSPACE(s)$ led to diminishing reference to $RSPACE(s)$ (and RL in particular) as a distinct class. Gradually it became clear that the "interesting" one-sided probabilistic log space class to study, was the one with a polynomial time bound. This led to the co-opting of the term RL to refer to this class, with ensuing notational inconsistencies. This notational ambiguity is particularly bothersome in the case of bounded error computation, since based on current knowledge, the classes $BP_HSPACE(s)$ and $BPSPACE(s)$ are distinct from each other and from other standard classes.

2.3 The robustness of probabilistic classes

We have followed the convention that the language recognized by a probabilistic machine is the set of strings that are accepted with probability strictly greater than 1/2. The choice of 1/2 is, of course, arbitrary. It is easy to show that the classes $PrSPACE(s)$ and $Pr_HSPACE(s)$ are unchanged if we set the acceptance threshold β to be any dyadic number in $(0, 1)$ (dyadic means having a terminating binary representation; in fact β can be a function of the input, provided that β is computable in space s, and so, in particular is at most 2^s bits long). We'll say that the PTM M β-computes L if for $x \in L$, $p_M(x) > \beta$ and for $x \notin L$, $p_M(x) \leq \beta$. Given any such PTM, we can construct M' that computes L relative to acceptance threshold 1/2 by having M' first flip a coin. If the coin is heads, M' runs M on x and accepts if M does. If the coin is tails, M' flips a sequence of at most 2^s coins in order to simulate a single coin with bias $1 - \beta$ and accepts if that coin is heads. Conversely, it is not hard (by a similar type of construction) to convert a computation with acceptance threshold 1/2 to any dyadic threshold β with at most 2^s bits.

Similarly, for bounded error computation, we can replace both the upper and lower thresholds 1/3 and 1/2 by arbitrary dyadic rationals (possibly functions of the input) α and β provided that $\alpha \leq \beta - 2^{-\theta(s)}$ We'll say that M (α, β)-computes L if for $x \in L$, $p_M(x) > \beta$ and for $x \notin L$, $p_M(x) < \alpha$. Given a machine M that (α, β)-computes L, define the machine M' that on input x does 2^{cs} independent trials of $M(x)$ for some large enough c, and accepts if the fraction of accepting trials exceeds $(\alpha + \beta)/2$. By standard probability estimates, one shows that M' $(2^{-\theta(s)}, 1 - 2^{-\theta(s)})$-computes L, i.e., the probability of error can be made exponentially small in s.

There is one subtle point here that is easy to miss. Based on the definition of a general probabilistic computation, the computation need not halt with probability 1. If it doesn't, then we can not reliably do repeated trials, since there is a nontrivial chance that the computation stalls during one trial. Thus, we need the following result of Simon:

Theorem 2.1 *[45] If M is a PTM running in space s with unbounded error (resp., bounded error, one-sided error) then there is a machine M' running in space s that computes the same language as L and runs in expected time $2^{2^{O(s)}}$ (and hence halts almost surely).*

A simple construction of M' given in [41] is: choose constants c and d appropriately and perform the following loop: while the computation has not halted, do (i) run M for d^s steps, and if M does not halt, (ii) toss $(c+d)^s$ coins and if all come up tails then halt and reject.

The second part of the loop provides a "probabilistic clock" which will cause the computation to halt with probability 1, but whose expectation is very large, i.e., $2^{(c+d)^s}$. Thus M' clearly halts within the required expected time, and it suffices to show that it computes the same language as M. Clearly for any input x, $p_{M'}(x) \leq p_M(x)$, and thus the language computed by M' is a subset of that computed by M. On the other hand, it can be shown that the fact that M and M' run in space s and $p_{M'}(x) < p_M(x)$ imply that $p_{M'}(x) \geq p_M(x) - 2^{-c^s}$ and combined with a fact that for a space s PTM M, $p_M(x) > 1/2$ implies $p_M(x) \geq 1/2 + 2^{-c^s}$ ([15], Lemma 6.6), one concludes that $p_M(x) > 1/2$ implies $p_{M'}(x) > 1/2$.

The classes $R_HSPACE(s)$ and $RSPACE(s)$ are similarly robust with respect to the acceptance threshold. In discussing the various classes, we freely use whatever thresholds are convenient.

2.4 Markov Chains and Matrix Computations

In understanding a computational model, two fundamental questions that are asked are: "what problems can be solved in this model?", and "are there natural mathematical structures for representing computation in this model?" For randomized space-bounded computation, answers to both of these questions can be found within the related domains of finite state Markov chains and matrix computation. We will assume familiarity with the most elementary facts about finite Markov Chains (see, e.g., [28]).

Associated to every finite Markov chain C on state space S is its transition probability matrix $P = P_C$, where for $i, j \in S$, $P[i, j]$ is the probability, given that the chain is in state i at a particular step, of moving to state j. P is a stochastic matrix, i.e., it is nonnegative with row sums 1. Any submatrix of P (defined by eliminating rows and/or columns) is *substochastic* (nonnegative with row sums at most 1). Such a matrix Q is *strictly substochastic* if all row sums are strictly less than 1.

We will begin by formulating some related computational problems for Markov chains and for stochastic, substochastic and strictly stochastic matrices, and then will discuss the direct connection of these problems with probabilistic space-bounded computation.

For a Markov chain C with transition probability P, we define the following matrix-valued functions of C (or, equivalently P). For an integer $t \geq 1$, the t-step transition probability matrix P^t is the matrix with $P^t[i, j]$ equal to the probability that, given the chain is in state i at a particular time, that the chain is in state j exactly t steps later. Of course, as the notation suggests, P^t is equal to the t^{th} power of P. The *hitting probability matrix* P^H, is the matrix with $P^H[i, j]$ equal to the probability that, given the chain is in state i at a particular time, that the chain will visit state j at some later time. It is a straightforward exercise to show that the j^{th} column of this matrix can be computed as follows: let v be the j^{th} column of P and Q be the matrix P with column j and row j replaced by 0's. Define Q^* to be the matrix (with possibly infinite entries) $\sum_{i \geq 0} Q^i$; Q is called the *completion of* Q [9]. Then it can be shown that (i) the matrix-vector product Q^*v is well defined even if Q^* has infinite entries since any infinite entry in Q^* must coincide with a 0 entry of v, (ii) Q^*v is equal to the j^{th} column of P^H and (iii) Q^* is finite if and only if $I - Q$ is invertible, in which case $Q^*v = (I - Q)^{-1}$, (iv) For any positive $\delta < 1$, the matrix δQ is strictly substochastic which implies that $(I - \delta Q)$ is invertible and so $(\delta Q)^* = (I - \delta Q)^{-1}$ is finite. Furthermore Q^*v is equal to the limit as δ tends to 1 of $(I - \delta Q)^{-1}v$.

The two functions above map matrices to matrices, and by specifying a particular entry $[i, j]$ of the output, we may view them as mapping matrices to real numbers (and in our case the range will be $[0, 1]$). For any real valued function f on some domain D, we define the threshold language L_f associated to f to be the set of inputs $x \in D$ such that $f(x) > 1/2$. We will consider three versions of the membership problem for this language, the *exact*, *approximate* and *one-sided* versions. In each we require that on input x, if $x \in L_f$, then x is accepted. In the exact version we require that if $x \notin L_f$, then x is not accepted. In the approximate version, false acceptance of $x \notin L$ is permitted for those x such that $f(x) > 1/3$ and in the one-sided version we allow false acceptance of $x \notin L$ for those x for which $f(x) > 0$.

Figure 2 shows a correspondence between space-bounded complexity classes and the threshold problem for the Markov chain functions and matrix functions described above. In each of the two rows, unbounded error computation corresponds to the exact version of the threshold problem for the function, bounded error computation corresponds to the approximate version and one-sided error computation to the one-sided version.

As will be seen below, for a given type of prob-

Complexity Classes	*Markov Chain Problems*	*Matrix Problems*
Halting	t-step Transition Probability	Stochastic Matrix Exponentiation
Non-halting	Hitting Probability	$Q^* = \lim_{\delta \to 1}(I - \delta Q)^{-1}$, for Q stochastic $(I - R)^{-1}$, for R strictly substochastic

Figure 2: Matrix and Markov chain problems corresponding to space-bounded probabilistic computation

abilistic space bounded computation, we can reduce a space s computation to the corresponding matrix problem for a $2^s \times 2^s$ matrix whose entries are 0,1/2 or 1. On the other hand, each matrix problem of the given type (where the matrix entries can be arbitrary dyadic rationals) can be solved in the corresponding probabilistic space bounded complexity class, where the space required is logarithmic in the size of instance of the matrix problem.

Various early papers in the area established that the problems listed within each row, in their various versions, play the role of complete problems for the corresponding set of complexity classes in that row. (Strictly speaking the approximate and one-sided versions of these problems are not complete for their corresponding class, since such a problem is not really a pure language membership problem, but is a "promise problem" [38]. One can reformulate the definitions to make it all precise, but we won't do that here.)

We first discuss how to reduce arbitrary probabilistic space-bounded computation to Markov chain and matrix problems. As first observed in [15], the execution $M(x)$ of a probabilistic machine M with space bound s, on a given input x, is a Markov chain whose state space is the set of 2^s possible configurations. Each step of $M(x)$ is either a deterministic change in the configuration or involves reading the next random bit ("tossing a coin") and changing the configuration to one of two configurations depending on that bit. Thus, for a given configuration i, the i^{th} row of the transition probability matrix either has a single nonzero entry, which is 1 (in the case that the next step from i is deterministic) or two nonzero entries which are each 1/2. By standard transformations we may assume that the Markov chain has exactly one state that corresponds to accepting x, labeled ACC, and that ACC is an absorbing state; i.e., $P[ACC, ACC] = 1$. We label the initial configuration of $M(x)$ by $START$.

With these definitions, it is clear that the probability that $M(x)$ accepts is just the entry $P^H[START, ACC]$ of the hitting probability matrix. As explained earlier, this computation boils down to the matrix computation $Q^* = \lim_{\delta \to 1}(I - \delta Q)^{-1}$ for some substochastic Q. In [21], it is shown that if we choose $\delta \geq 1 - 2^{-Cs}$ for some $C > 0$, then $Q^*[i, j] > T$ if and only if $(I - \delta Q)^{-1}[i, j] > T$. Thus we can reduce the problem $Q^*[i, j] > T$ to the problem $(I - R)^{-1}[i, j] > T$ for some strictly substochastic matrix R.

For the halting classes, we can reduce them to the same problems as for the non-halting classes, but in fact we can reduce to the threshold problem for the t-step transition matrix. We say that a Markov chain is *acyclic* with respect to a designated state $START$ if the underlying directed graph of the Markov chain, when restricted to the set of states accessible from $START$ has no directed cycles except for self loops at absorbing states. Then we have:

Proposition 2.2 *The Markov chain associated to a computation that halts absolutely is acyclic with respect to the START state.*

This is clear, since the existence of a cycle that uses only accessible states implies that there is a setting of the random coins on the tape that will cause the chain to traverse this cycle indefinitely so that the computation will not halt.

Acyclicity can easily be seen to imply that for any t exceeding the number of states of the chain, the chain starting from $START$ must occupy an absorbing state at step t. This means that the probability of eventually hitting ACC is equal to the probability of being in ACC after exactly t steps. From this the reductions in the top half of figure 2 follow.

The acyclicity of the Markov chain immediately implies that a space s PTM that halts absolutely runs with time bound 2^s. Formally, we have:

Proposition 2.3 *1. A probabilistic space s machine that halts absolutely, must halt within 2^s steps.*

2. For each $X \in \{Pr, BP, R, ZP\}$,

$$X_H SPACE(s) = XTISP(2^{O(s)}, s).$$

The first part implies the forward inclusion of the second part; the reverse inclusion of the second part is easy but reflects a subtlety in the precise definition ([15]) of time-bounded randomized computation; i.e., time bound $t(n)$ does not in general mean that the computation is guaranteed to halt in time $t(n)$. However, one can easily show that for a computation in $XTISP(2^{O(s)}, s)$ if one uses an $O(s)$-bit clock to halt the computation after $2^{O(s)}$ steps, the resulting computation recognizes the same language.

Next, we sketch the other direction of the correspondences represented in figure 2: that the various matrix computations can be solved within the corresponding complexity class.

Consider first the halting classes. It is easy to see that each variant of the transition probability problem (and hence the matrix exponentiation problem) can be solved in the corresponding complexity class. Basically, this is because we can build a probabilistic machine M that on input a $2^s \times 2^s$ stochastic matrix and index i, simulates the run of the associated Markov chain starting from i, where the simulation runs in space s. (Notice that since we restrict to the case that all probabilities are dyadic rationals we can simulate a single transition exactly by a finite sequence of coin flips). To compute or approximate the threshold language associated to the t-step transition probability for states i and j and threshold T, we simply simulate the chain for t steps starting at i, and accept if and only if the simulation ends in state j.

Now consider the non-halting classes. We argue similarly that the various versions of the hitting time problem for Markov chains (and the corresponding completion problem for matrices) can be solved in these classes. Now the algorithm consists of running the Markov chain starting from i for an indefinite number of steps and halting and accepting when and if the chain arrives in state j.

2.5 Randomness-bounded computation

Yet another way to view the restriction "halts absolutely" is as a restriction on the number of random bits used by the algorithm. By proposition 2.3 any space s halting machine runs for time at most $2^{O(s)}$ and hence uses at most this number of random bits. Conversely, any machine that satisfies a restriction on the number of random bits can be converted to one that is halting and accepts the same language, as follows: add a counter that increments with each step of the computation and is reset to 0 each time a new random bit is read, and if the counter ever reaches 2^s, halt and reject. This modification clearly guarantees that the machine halts absolutely, and it is easy to see that it accepts the same language as the original machine.

This interpretation of absolute halting suggests a further refinement of the probabilistic complexity classes, according to the number of random bits used.

If M is a PTM, we say that M *operates within randomness bound* $r = r(n)$ if on any input x, $M(x)$ uses at most $r(|x|)$ random bits. If there is such a function, we say the machine is *randomness-bounded*, which by the above remarks, is the same as saying that the machine halts absolutely. For $X \in \{Pr, BP, R, ZP\}$, and $r = r(n), s = s(n)$, we define $X_r SPACE(s)$ to be the subclass of $XSPACE(s)$ consisting of those languages that are recognizable by a PTM M that operates within randomness bound $r(n)$ and satisfies the restrictions specified by $XSPACE(s)$. In analogy to Proposition 2.3 we have:

Proposition 2.4 *For any* $X \in \{Pr, BP, R, ZP\}$, $X_H SPACE(s) = X_{2^{O(s)}} SPACE(s)$.

On the other hand, the trivial deterministic simulation of a randomized machine (try all values of the random bits and count the number of accepting computations) yields:

Proposition 2.5 $Pr_r SPACE(s) \subseteq DSPACE(r + s)$, *in particular* $Pr_s SPACE(s) = DSPACE(s)$.

Thus, space s probabilistic computation with randomness bound $\theta(s)$ provides no advantage over determinism, while randomness bound 2^s gives the full power of absolutely halting probabilistic computation.

3 Relationships between complexity classes

As mentioned in the introduction, there has been considerable success in relating probabilistic classes to each other and to deterministic and non-deterministic classes. In figure 3 we summarize the present state of knowledge regarding the relationships between the various classes. In figure 4 we present a plausible guess of the "true" map of this part of the complexity theory world. In it there are only three distinct classes, deterministic space s, nondeterministic space s, and unbounded probabilistic space s, and all the other classes are equivalent to one of those three. The complete lack of lower bound techniques for complexity classes above NC_1 means, of course, that proving the *distinctness* of these three classes remains a wistful dream. However, the underlying theme of the rest of this paper is that there collapsing the picture in 3 down to the picture in 4 may indeed be possible in the foreseeable future.

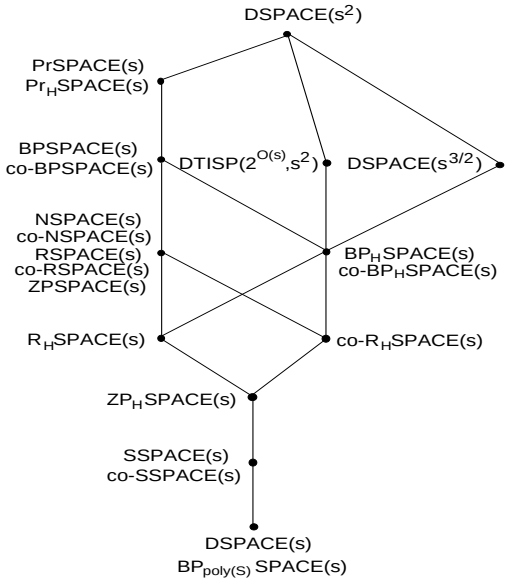

Figure 3: Relationships among space-bounded complexity classes

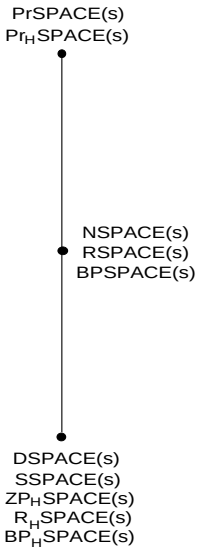

Figure 4: A conjectural view of the space-bounded complexity world

In this section, we review in detail the results summarized in figure 3. At the same time, we highlight various questions that point in the direction of figure 4.

3.1 Solving $USTCON$ in R_HL

The most well known use of probabilism in space-bounded computation is the result of Aleliunas et al. [3] showing how $USTCON$ can be decided in R_HL. By our earlier remark that $SSPACE(s)$ can be reduced to a $USTCON$ problem on a graph of size 2^s, this implies that $SSPACE(s) \subseteq R_HSPACE(s)$.

Given the n vertex graph G and vertices s and t, the randomized algorithm for $USTCON$ is to start at vertex s and take a random walk on the graph G: when the walk is at vertex v choose one of the neighbors of v uniformly at random and move to that neighbor. It can be shown that if s and t are in the same component then after at most $poly(n)$ steps the random walk will reach t with high probability. So simply walk for that number of steps and accept if and only if the vertex t is visited during the walk.

3.2 Probabilism vs. non-determinism

One of the earliest observations made about probabilistic space-bounded computation is that, when there is no halting restriction, one-sided error is equivalent to non-determinism:

Proposition 3.1 *[15]* $RSPACE(s) = NSPACE(s)$

The forward inclusion is trivial, while the reverse inclusion is obtained by noting that given a NTM M, we can construct a PTM M' that on input x, performs an infinite loop, where in each iteration it simulates M, using random bits in place of the nondeterministic ones. Such a machine will never falsely accept $x \notin L$, and will accept any $x \in L$ with probability 1, hence operates with one-sided error.

This equivalence, together with the trivial inclusions between probabilistic classes, imply that $R_HSPACE(s) \subseteq NSPACE(s) \subseteq BPSPACE(s) \subseteq PrSPACE(s)$. Figure 4 suggests that the following two open questions have affirmative answers.

Question 3.1 *Is* $NSPACE(s) = BPSPACE(s)$*?*

Question 3.2 *Is* $BP_HSPACE(s) \subseteq NSPACE(s)$*?*

In light of the celebrated result [18, 48] that the alternating space hierarchy collapses to $NSPACE(s)$, the second question represents an analog to the result [46] that BPP is contained in the polynomial time hierarchy. However, a straightforward attempt to adapt the simulation for BPP to question 3.2 fails for at least two reasons: the simulation seems to require two-way access to non-deterministic bits and also to require the computation of universal hash functions in small space.

3.3 Halting versus non-halting computations

Proposition 3.1 makes apparent the potentially drastic affect that the halting condition has on one-

sided error computation: without the halting condition one gets the full power of non-determinism, while it seems implausible that $R_H SPACE(s) = NSPACE(s)$. In contrast, for unbounded error computation, Jung showed that the two classes coincide:

Theorem 3.1 *([22])*

$$PrSPACE(s) = Pr_H SPACE(s).$$

The proof of the non-trivial inclusion as simplified in [4], follows from (1) the fact listed in figure 2 that $PrSPACE(s)$ can be reduced to the threshold problem for the entry of some matrix inverse, (2) the fact that the threshold problem for a matrix inverse can be converted (via Kramer's rule) to the question of which of two matrices has a larger determinant, and (3) this latter problem is solvable in $Pr_H SPACE(s)$.

What about the relationship between $BPSPACE(s)$ and $BP_H SPACE(s)$? Since $NSPACE(s) \subseteq BPSPACE(s)$, and we suspect that $BP_H SPACE(s)$ equals $DSPACE(s)$, we certainly expect the two classes to be different. One reasonable intuition is that the power of $BPSPACE(s)$ is simply the combination of the power of $BP_H SPACE(s)$ and $NSPACE(s)$:

Question 3.3 *Is $BPSPACE(s)$ equal to the class of languages accepted by a deterministic space s machine with access to oracles for $NSPACE(s)$ and $BP_H SPACE(s)$?*

Note also that if question 3.3 is shown to have an affirmative answer, then questions 3.1 and 3.2 would be equivalent.

In an effort to refine this question, one can define a class that contains both $BP_H SPACE(s)$ and $NSPACE(s)$ and is contained in $BPSPACE(s)$, which we will call, for lack of a better term, $SBP_H SPACE(s)$, (for *slightly bounded error*). (This class arose in discussions with Eric Allender, Shiyu Zhou and David Zuckerman). A PTM M is said to accept a language L with slightly bounded error if there is an integer valued function $K(n)$, computable in space $s(n)$, such that for $x \in L$, $p_M(x) > 2^{-K(|x|)}$ and for $x \notin L$, $p_M(x) \leq 2^{-K(|x|)-1}$. This class differs from $BP_H SPACE(s)$ because the function K might be as big as $2^{\theta(s)}$ and so the error probability is potentially doubly exponential in s and thus too small to amplify by a halting computation (which can only perform 2^s iterations.) However, using a probabilistic clock similar to the one that appears in the proof of Theorem 2.1, one can amplify the acceptance probability by performing a random number of trials where the expected number of trials is $2^{2^{K(|x|)}}$, and show that $SBP_H SPACE(s) \subseteq BPSPACE(s)$. It is possible that the inclusion also goes the other way, which would be interesting on its own, and also might help to resolve question 3.3.

3.4 Complementary classes

Theorem 3.2 *The following space-bounded complexity classes are closed under complementation: $ZPSPACE(s)$, $RSPACE(s)$, $BPSPACE(s)$, $PrSPACE(s)$, $ZP_H SPACE(s)$, $BP_H SPACE(s)$, $Pr_H SPACE(s)$, $SSPACE(s)$, and $NSPACE(s)$.*

For each of the classes $ZPSPACE(s)$, $ZP_H SPACE(s)$, $BP_H SPACE$, and $Pr_H SPACE(s)$ the result follows trivially from the definition. For $BPSPACE(s)$ and $PrSPACE(s)$ closure under complement was first shown in [45]; it requires Theorem 2.1, which shows that we can assume that a probabilistic computation halts almost surely, and this then permits the roles of acceptance and non-acceptance to be interchanged. For $PrSPACE(s)$ one can also see this as a consequence of the equivalence of $PrSPACE(s)$ and $Pr_H SPACE(s)$.

For $NSPACE(s)$ (and hence for $RSPACE(s)$), closure under complementation is, of course, from [18, 48].

Finally for $SSPACE(s)$, the result follows from a very clever direct reduction [35] from the problem $co - USTCON$ to $USTCON$. This implies an earlier result of [8], which was proved using inductive counting arguments in the spirit of [18, 48], that $SSPACE(s) \subseteq co - R_H SPACE(s)$, i.e., $SSPACE(s) \subseteq ZP_H SPACE(s)$.

Conspicuous by its omission from the list of classes in Theorem 3.2 is the class $R_H SPACE(s)$. Indeed, the following is a very tantalizing open question, which appears in [8]:

Question 3.4 *Is $R_H SPACE(s)$ closed under complement, i.e., is $R_H SPACE(s) = ZP_H SPACE(s)$?*

3.5 Deterministic Simulations

The first deterministic simulation results for bounded-space probabilistic computation were for the most powerful class $PrSPACE(s)$. Gill [15] showed that $PrSPACE(s) \subseteq DSPACE(2^{O(s)})$. Simon [44] was the first to show that $PrSPACE(s)$ can be simulated deterministically with only a polynomial increase in space, specifically, in $DSPACE(s^6)$. This result was improved independently by Borodin, Cook and Pippenger, and Jung:

Theorem 3.3 *([9, 20])*

$$PrSPACE(s) \subseteq DSPACE(s^2).$$

This strengthens Savitch's fundamental theorem [43] that $NSPACE(s) \subseteq DSPACE(s^2)$.

The proofs of Theorem 3.3 and its antecedents were based on small space solutions to one of the associated matrix problems discussed in section 2.4. In particular, in [9], the problem of computing the limit as $\delta \longrightarrow 1$, of $(I - \delta Q)^{-1}$ is solved by computing the inverse matrix treating δ as a formal variable and then passing to the limit. These computations are shown to be performable in $NC^2 \subseteq DSPACE((\log m)^2)$ (as a function of the matrix size $m = 2^s$).

Any improvement in Theorem 3.3 would, in particular, imply an improvement in Savitch's theorem, and would be a remarkable achievement.

The other known deterministic simulations are for bounded error halting computations. Each approaches the conjectured collapse $BP_HSPACE(s) = DSPACE(s)$ from a different direction.

The matrix problems that were shown to be equivalent to $PrSPACE(s)$ are all solvable in polynomial time, which implies that $PrSPACE(s) \subseteq DTIME(2^{O(s)})$. It is not known, however, whether problems in $PrSPACE(s)$ (or even $NSPACE(s)$) can be solved by a DTM that runs simultaneously in polylog space and polynomial time. For $BP_HSPACE(s)$, such a result was obtained by Nisan:

Theorem 3.4 *([31])*

$$BP_HSPACE(s) \subseteq DTISP(2^{O(s)}, s^2).$$

When specialized to the case $s = \log n$, this says that bounded error log-space poly-time probabilistic computation (BP_HL) can be simulated in deterministic polylog-space poly-time (SC, or "Steve's class"). In contrast, the known deterministic simulations of NL in polylog space require $n^{\log n}$ time.

In another direction, Ajtai, et al. [2] looked at the problem of deterministically simulating space s PTMs that operate with some randomness bound r. As noted in Proposition 2.4, if $r = \theta(s)$, then such a simulation is trivial and for $r = 2^s$ this the problem is equivalent to simulating any halting computation. They showed that a one-sided probabilistic computation operating with randomness bound $s^2/\log s$ can be made deterministic. This was extended by Nisan and Zuckerman, to the case that r is any polynomial in s, and to bounded error (not just one-sided error) computation:

Theorem 3.5 *([37])*

$$BP_{poly(s)}SPACE(s) \subseteq DSPACE(s).$$

The best result known for deterministic simulations of halting bounded error computations with no nontrivial restriction on the number of random bits was proved by Saks and Zhou:

Theorem 3.6 *([42])*

$$BP_HSPACE(s) \subseteq DSPACE(s^{3/2}).$$

This result generalized the previous result of Nisan, Szemerédi and Wigderson [34] that $USTCON$ could be solved in $DSPACE((\log n)^{3/2})$.

The central ingredient in the deterministic simulations that achieve Theorems 3.4, 3.5 and 3.6 are pseudorandom generators for space-bounded computation. The discussion of such generators and their use in these results is the subject of the next section.

4 Pseudorandom number generators for space-bounded computation

A pseudorandom generator (PRG) for a class of probabilistic machines is, intuitively, a probabilistic machine that uses only a "small" number of random bits from it's tape and outputs a long "pseudorandom" string that "fools" any machine in the class, in the sense that no machine in the class can distinguish the output of the PRG from a truly random string. Clearly, such PRGs can be used to reduce the amount of randomness needed by a computation. In this section we discuss the construction and application of PRGs for the class of halting space-bounded PTMs ($HSBPTM$s).

Formally, a *generator* is a function G that maps $\{0,1\}^{R_2}$ to $\{0,1\}^{R_1}$ for some integers R_2 and R_1. The parameter R_2 is referred to as the *seed length* of the generator, and the parameter R_1 is the *output length* of the generator. We use the notation $G\{R_2, R_1\}$ to indicate that G is a generator with these parameters. If $G\{R_2, R_1\}$ is a generator, then G induces a probability distribution D_G on $\{0,1\}^{R_1}$ called the *output distribution of* G: for $z \in \{0,1\}^{R_1}$, $D_G(z)$ is equal to the probability, for w selected uniformly from $\{0,1\}^{R_2}$, that $G(w) = z$.

Now suppose that $M(x)$ is a PTM execution that uses at most R_1 random bits. We defined $p_M(x)$ to be the probability that M accepts x, under the assumption that the R_1 random bits it uses are selected uniformly from all R_1 bit strings. If $G\{R_2, R_1\}$ is a generator, define $p_M^G(x)$ to be the probability that M accepts x under the assumption that it uses R_1 bits selected according to the output distribution of G. We come to the first key definition:

Definition 4.1 *Let $G\{R_2, R_1\}$ be a generator and $\epsilon \in (0,1)$.*

1. *If $M(x)$ is a PTM execution using at most R_1 random bits, then G is said to* ϵ-fool *$M(x)$ if $|p_M^G(x) - p_M(x)| \leq \epsilon$.*

2. *Let S be a positive integer. G is said to be an* ϵ-PRG for space S *if for any PTM execution $M(x)$ that runs in space S and with randomness bound R_1, G ϵ-fools $M(x)$.*

Intuitively, G is an ϵ-PRG if it is impossible for any space S computation to statistically distinguish the output of G from a random string, with probability more than ϵ.

As defined, generators are "non-uniform" objects: each generator is a single function mapping strings of one fixed length to another. To be useful for deterministic simulation of $HSBPTM$ computations, we want to construct an ensemble of generators whose seed length and output length are variable and that are, in some appropriate sense, efficiently computable. This leads to the next set of definitions.

First, let us establish a notational convention: we will continue to use $s = s(n)$ to refer to the space-bound of some $HSBPTM$ M and $r = r(s)$ to denote the randomness bound. We will use capital letters S and R to denote specific values that s and r take on. For instance, if x is an input to M, then when we refer to the execution $M(x)$, we will use $S = s(|x|)$ and $R = r(S)$.

We define a $HSBPTM$ *generator ensemble* $\hat{G}$ to be a $HSBPTM$ that takes as input two parameters S and R, where S represents the space bound of the computation that the generator is intended to fool, and R is the number of pseudorandom bits being requested from the generator. The generator outputs R bits. The *seed length* of $\hat{G}$ on input S, R, $L(S,R) = L_{\hat{G}}(S,R)$ is defined to be the maximum number of random bits that $\hat{G}$ reads from its random tape during it's computation. The behavior of $\hat{G}$ on input S and R can be viewed as a generator $G^{S,R}\{L(S,R), R\}$. By definition $\hat{G}$ is itself space-bounded. The space-bound $s_{\hat{G}}$ of $\hat{G}$ is expressed as a function of S and R (not of the input length).

Let $\hat{G}$ be a $HSBPTM$ generator ensemble. If M is a PTM with space bound s and randomness bound r, we define the $\hat{G}$ *simulation of* M, $M' = \hat{G} \circ M$, to be a PTM that on input x, simulates $M(x)$, but instead of directly using random bits from its auxiliary tape, it uses the output of $G^{S,R}$, where $S = s(|x|)$ and $R = r(s)$. (Notice that we are assuming (without loss of generality) that M' comes "equipped" with the functions s and r). $M'(x)$ executes $M(x)$, until the first time it needs a random bit. Instead of reading a bit from its auxiliary tape, it simulates $\hat{G}$ on inputs S and R until $\hat{G}$ outputs a bit, at which point the execution of $M(x)$ resumes, using the first output bit of $G^{S,R}$ in place of $M(x)$. Each time that $M(x)$ needs another random bit, the execution of $G^{S,R}$ is resumed where it left off until it produces the next output bit, which is used in $M(x)$. The space needed for $\hat{G} \circ M(x)$ is the sum of the space needed for $G^{S,R}$ and $M(x)$, and thus:

Proposition 4.1 *Let M be a $HSBPTM$ with space bound s and randomness bound $r(s)$, and let $\hat{G}$ be a $HSBPTM$ generator ensemble with space bound $s_{\hat{G}}(S,R)$. Then $\hat{G} \circ M$ is a PTM with space bound $s + s_G(s, r(s))$ and randomness bound $L_{\hat{G}}(s, r(s))$. Furthermore, for any input x and for $S = s(|x|)$, $R = r(S)$, the probability $p_{\hat{G} \circ M}(x)$ that $\hat{G} \circ M$ accepts x is equal to $p_M^{G^{S,R}}(x)$.*

The next definition is the analog of definition 4.1 for $HSBPTM$ generator ensembles.

Definition 4.2 *Let $\hat{G}$ be an $HSBPTM$ generator, and let $\epsilon = \epsilon(S,R)$ be a function whose range is in the interval $(0,1)$.*

1. *Let M be a $HSBPTM$. Let x be an input and let S and R be the space and randomness bound for $M(x)$. The ensemble $\hat{G}$ is said to* ϵ-fool *the execution $M(x)$ if the generator $G^{S,R}$ $\epsilon(S,R)$-fools $M(x)$.*

2. *Let $r = r(S)$ be a (suitably well behaved) function with $r(S) \leq 2^S$. $\hat{G}$ is an* ϵ-PRG for randomness bound r, *or (ϵ, r)-PRGE (pseudorandom generator ensemble) if for any integer $S \geq 1$, $G^{S,r(S)}$ is an $\epsilon(S, r(S))$-PRG for space S. In other words, for any $HSBPTM$ M with randomness bound r, and for any input x, $\hat{G}$ ϵ-fools $M(x)$.*

Note that since every $HSBPTM$ has randomness bound 2^s, an $(\epsilon, 2^s)$-PRGE $\hat{G}$ ϵ-fools every $HSBPTM$ computation, and so we say, simply that $\hat{G}$ is an ϵ-PRGE.

The existence of appropriate PRGEs implies the following deterministic simulation result for $BP_H SPACE(s)$:

Proposition 4.2 *Suppose that $\hat{G}$ is an $(1/20, r)$-PRGE for some suitably well behaved function $r = r(s)$, and that $\hat{G}$ has seed length function L and runs in space $s_{\hat{G}}$. Then:*

$$\begin{aligned} BP_r SPACE(s) &\subseteq BP_{L(s,r)} SPACE(s + s_{\hat{G}}(s,r)) \\ &\subseteq DSPACE(s + s_{\hat{G}}(s,r) + L(s,r)) \end{aligned}$$

Indeed, the first containment follows from Proposition 4.1 and the fact that by the definition of an ϵ-PRGE, if M is a $HSBPTM$ that computes L with bounded error, then $\hat{G} \circ M$ $(1/3+\epsilon, 1/2-\epsilon)$-computes L, as defined in section 2.3. The second containment follows immediately from the naive derandomization of $\hat{G} \circ M$ as in Proposition 2.5.

What do we need in this proposition to prove $BP_H SPACE(s) \subseteq DSPACE(s)$? For an arbitrary $HSBPTM$, we may take $r(s) = 2^s$. If the seed length, $L(s, 2^s)$ is $\theta(s)$ and the space $s_{\hat{G}}(s, 2^s)$ required to produce 2^s bits is also $\theta(s)$, then we would have this optimal deterministic simulation of $BP_H SPACE(s)$.

Now, it is an elementary exercise in the probabilistic method to show that if we omit the requirement that the generator be computable, and take, for each $S \geq 1$, $G^{S,2^S}$ to be a random function mapping $c_1 S$ bits to 2^S bits, then the result would fool all probabilistic computations running in space S. This fact can be viewed as a rough analog to Adleman's result ([1]) that BPP can be computed by non-uniform polynomial circuits.

Of course, we do not now know how to construct efficiently computable generators that achieve the optimal simulation (otherwise, this would be quite a different paper!). The crux of the problem, then, is to find a PRGE whose seed length and space requirement are sufficiently small functions of S and R.

The literature contains six constructions of generators of various quality which we will discuss. The last of the six generators, which builds heavily on previous ones, provides the best parameters currently known: it gives an ϵ-PRG $\hat{G}$ for some $\epsilon = R^{-\theta(1)}$, such that the seed length $L(S, R)$ is $O(\frac{S \log R}{2 \log S - \log \log R})$, and the space required for the generator is linear in the size of the seed. The two most interesting special cases are when $R = 2^S$, in which case the seed length is $O(S^2)$, and the case that $R = poly(S)$ in which case the seed length is $O(S)$. For each of these two special cases, previous constructions had given the same bounds.

The remainder of this section consists of six subsections each devoted to one of the generator constructions, presented in historical order. These subsections summarize what the various generators accomplish, and describe their constructions. The question of *why* these constructions are generators is beyond the scope of this survey. As stated in the previous section, the three deterministic simulations stated in Theorems 3.4,3.5 and 3.6 were obtained using these generators. For Theorem 3.5, the simulation is constructed directly from one of the generators by using Proposition 4.2. For each of the other two, the derandomization uses Nisan's generator inside a more intricate simulation. We will discuss these simulations together with Nisan's generator in Section 4.3.

Two final remarks on terminology and notation before proceeding to the descriptions of the generators. As described above, we are really interested in defining generator ensembles, not just generators. However, it is more convenient to define a generator ensemble $\hat{G}$ by describing a component generator $G^{S,R}$ for generic parameters S and R. The fact that these generators are part of an ensemble is almost always implicit. Typically, it is also more convenient to first define the generic function in terms of parameters other than S and R, and then to relate the chosen parameters to S and R.

The second remark is that, in the case that we are only trying to build an (ϵ, r)-generator for some particular randomness bound r, it suffices to assume that the parameter R is equal to $r(S)$; since we don't care what the generator does for other values of R. In this case, we can and will view $\hat{G}$ as taking a single parameter S, and, if needed for clarity, we refer to $\hat{G}$ as a *single parameter $HSBPTM$ generator ensemble*. We then write $L(S)$ for the seed length function, and G^S for the function from $L(S)$ bits to $r(S)$ bits defined by $\hat{G}$ with input S.

4.1 The AKS pseudorandom sampler

The first work in the direction of constructing PRGs for small space was by Ajtai, Komlós and Szemerédi [2]. Although the formalism they used differs considerably from the one given above, what they presented can be interpreted as a construction of a single parameter $HSBPTM$ generator that, on input S, produces a string of length $r(S) = \theta(\frac{S^2}{\log S})$ from a seed of size $\theta(S)$ in space $\theta(S)$. Their generator satisfies the following condition, which is a relaxation of the condition of being a PRGE.

Definition 4.3 *Let $G\{R_2, R_1\}$ be a generator and $\epsilon \in (0, 1)$.*

1. *If $M(x)$ is a PTM execution using at most R_1 random bits, then G is said to* ϵ-sample *$M(x)$ if $p_M(x) \geq \epsilon$ implies that $p_M^G(x) > 0$.*
2. *Let S be a positive integer. G is said to be an* ϵ-PRS (pseudorandom sampler) for space S *if for any PTM execution $M(x)$ that runs in space S, G ϵ-samples $M(x)$.*

In the same way that PRGEs were defined from PRGs, we can define a pseudorandom sampler ensemble (PRSE) from PRS. The definition of a pseudorandom sampler ensemble is exactly what is needed for

use in a deterministic simulation of a one-sided error computation (provided that $\epsilon \leq 1/2$). If M computes L with one-sided error, then given a 1/2-PRSE, we can deterministically decide whether $x \in L$ by running the computation $M(x)$ using the output of the generator for each possible seed. If $x \notin L$ no computation will accept x (since the computation has one-sided error), while if $x \in L$, there will be at least one seed which causes the execution to accept.

We remark (1) that an (ϵ, r)-PRGE for space bounded computation is automatically an (ϵ, r)-PRSE (2) the definition of an (ϵ, r)-PRSE makes sense even if G_S is a partial mapping from $L(S)$ bits to $r(S)$, rather than a total mapping.

We briefly describe the AKS construction of a pseudorandom sampler, which introduced an important technique known as "expander walks". We present a simplified version of their that was given in [26] for another purpose. For $\alpha \in (0,1)$ and positive integers N, and $d > 0$, an undirected graph H is an (N, d, α)-expander if H has N vertices, maximum degree d and for any subset A of vertices, the fraction of vertices in $V(G) - A$ that have a neighbor in A is at least $\alpha|A|/N$. We will think of a d-regular expander as a directed graph in which each edge is replaced by a directed edges in each direction, and the edges coming out of a vertex are indexed from 1 to d. The generator G^S uses an expander with $N = 2^S$ vertices, whose vertices all have the same degree, where both the degree d and the parameter α are constants independent of S. Such expanders exist and are explicitly constructible in the following sense: there is an algorithm that given an S-bit index to a vertex v and an integer i between 1 and d, runs in space $\theta(S)$ and returns the index of the i^{th} neighbor of v (see, e.g. [14, 27]).

Given a d-regular expander H and arbitrary positive integers $J \leq K$ we define a partial mapping sending $S + (\log d)(K-1) + K$ bits to JS bits as follows. Fix a (possibly redundant) encoding of the J element subsets of $\{1, 2, \ldots, K\}$ by K bits. Given the input, use the first K bits to specify such a J element subset I. Use the remaining bits to specify a path $(v_1, v_2, \ldots, v_K)$ in H of K vertices as follows: use S bits to specify a vertex and use each of $K-1$ successive blocks of $\log d$ bits to specify an edge label, and construct the path following those labels. Finally, use the subset $I = \{i_1 < i_2 < \ldots < i_J\}$ to extract a subsequence $(v_{i_1}, v_{i_2}, \ldots, v_{i_J})$ from $(v_1, v_2, \ldots, v_k)$ and interpret this sequence as a sequence of JS bits.

The main result of [2] is that for each $\epsilon > 0$ there are constants C_1, C_2, for $K = C_1 S$ and $J = C_2 S / \log S$, this mapping is an (ϵ, r)-PRS, for $r(S) = \theta(S^2/\log S)$. The result is proved by a delicate induction using a carefully selected (and quite technical) induction hypothesis. Vaguely, the idea is that a randomized computation defines a random walk through the configuration space. If the machine is such that with a substantial probability (at least ϵ) the random walk accepts, then the expander property guarantees that the sequences produced by the generator are sufficiently "spread" out that they are guaranteed to include an accepting path.

4.2 The BNS generator

The first formalization of the notion of a PRG for small space and the first construction of such a generator, was done by Babai, Nisan and Szegedy [7]. Their generator is most easily described in terms of three parameters k,t and u. For now, view these parameters as fixed. The construction needs a function $f = f^{k,u}$ that maps ku bits to 1 bit. We write f as $f(x_1, x_2, \ldots, x_k)$ where each $x_i \in \{0,1\}^u$. For the given t, if $I = \{i_1 < i_2 < \ldots < i_k\}$ is a subset of $\{1, 2 \ldots, t\}$, define the function $f_I : \{0,1\}^{ut} \longrightarrow \{0,1\}$ by $f_I(x_1, \ldots, x_t) = f(x_{i_1}, x_{i_2}, \ldots, x_{i_k})$. Let $B = \binom{t}{k}$ and let $I_1, I_2, \ldots, I_B$ be the enumeration of the k element subsets of t in *colex* order (i.e., $I < J$ if the highest element in their symmetric difference is in J). Finally, define $G(x_1, \ldots, x_t)$ to be the concatenation of the bits $f_{I_j}(x_1, \ldots, x_t)$ for $1 \leq j \leq B$.

The function G maps ut bits to $\binom{t}{k}$ bits. If f is chosen to be the generalized inner product function, whose output is the sum for $1 \leq i \leq u$ of the product of the i^{th} bits from each input, then it is shown that the resulting generator is an ϵ-PRG (for any $\epsilon > 0$) for space s, provided that $s \leq u/c^k$ for some constant $c = c(\epsilon)$. For a given space S, we can choose the parameters as follows: $u = 2^{\theta(\sqrt{S})}$, $k = \theta(\sqrt{S})$, $t = 2^k$, to obtain a generator that maps $2^{\theta(\sqrt{S})}$ bits to $2^{\theta(S)}$ bits and ϵ-fools all space S computations, for some $\epsilon = 2^{-\theta(\sqrt{S})}$.

The proof that their generator is a PRG for space bounded computation is based on a connection between small space computation and a model of multiparty communication complexity introduced in [10]. Let $f(x_1, x_2, \ldots, x_k)$ be a boolean function as above. In the multi-party communication model, there are k parties, and the i^{th} party knows every input *except* x_i. They communicate by means of a blackboard readable by all. The communication complexity of f is the minimum number of bits that must be written in worst case in order that one of them can evaluate f. The ϵ-complexity of f, $C_\epsilon(f)$ is the minimum number of bits required so that f is correctly evaluated on at least a $1/2 + \epsilon$ fraction of the inputs.

Now fix such a function (family) $f = f_{k,u}$, and consider the above generator G constructed from f. What Babai, et al. show is that if there is a space S execution $M(x)$ that is not ϵ-fooled by the generator, then one can used this machine to construct a multi-party protocol using kS bits that is correct on at least $1/2+\epsilon$ of the inputs, and therefore $C_\epsilon(f)/k < S$. Thus if we find a function f for which a lower bound on $C_\epsilon(f)$ can be proved, we get a corresponding lower bound on the smallest space needed to distinguish the output of the generator from random. In the case of the generalized inner product function, they showed a $\Omega(\frac{u}{4^k} + \log \epsilon)$ lower bound on $C_\epsilon(f)$, which implies the results stated above for G.

What is the limit of this approach? For multi-party communication complexity there is a trivial upper bound of u (since one party can simply announce its input to the others), so we can't hope for a bound better than $C_\epsilon(f) \geq u$. If we did get such a bound, the corresponding lower bound on the space needed to beat the generator would be u/k. Thus to produce 2^s bits that fool space s machines, the best we could do would be to take $k = \theta(s)$, $t = 2k$ and $u = \theta(s^2)$ giving a generator that needs a seed of $\theta(s^3)$ bits. This would be a considerable advance over the above generator, but still falls short of the $\theta(s)$ size seed that we'd like in order to derandomize space s computations.

No explicit functions f are known for which such a strong lower bound, or even a lower bound of $u/poly(k)$ can be proved (although a random function f can be shown to satisfy such a bound). However, as will be seen in succeeding section, other approaches to generator construction lead to PRGs that produce 2^s bits from a seed of size $\theta(s^2)$, which is better than the theoretical limit to the above approach.

4.3 Nisan's generator and its application to derandomization

The next construction of a PRG for small space was Nisan's, which was a major breakthrough. First of all, this construction substantially reduced the size of the seed needed to produce 2^s bits fooling space s executions, from $2^{\theta(\sqrt{s})}$ to only $\theta(s^2)$. This is a remarkable achievement, but unfortunately, it seems to fall just short of providing an improved deterministic simulation of $BP_HSPACE(s)$. This is because if we simulate $BP_HSPACE(s)$ by running over all seeds, we will need $\theta(s^2)$ space just to store the seed. As we have seen, there is already a $\theta(s^2)$-space deterministic simulation not just for $BP_HSPACE(s)$ but for $PrSPACE(s)$.

However, it turns out that Nisan's generator has some subtly stronger properties than being a PRG. These properties can be carefully exploited to obtain two deterministic simulations of $BP_HSPACE(s)$, one in $DTISP(2^{O(s)}, s^2)$ (Theorem 3.4) and the other in $DSPACE(s^{3/2})$(Theorem 3.6).

For fixed S and R, Nisan's construction involves constructing not one function, but a family of functions. The property of this construction is that the family of generators is not too large and has the property that every space S computation using R bits is fooled by most of them. The following Theorem is implicit in Nisan's work:

Theorem 4.1 *[30, 31] Let k, b be positive integers with $b \leq 2^k$. There is a family $\mathcal{G} = \mathcal{G}_{b,k} = \{G_h | h \in \{0,1\}^{2bk}$ satisfying for each b and k:*

1. *Each of the functions G_h maps b bits to $R = b2^k$ bits.*
2. *Given as input $h \in \{0,1\}^{2bk}$ and $x \in \{0,1\}^b$, $G_h(x)$ can be computed in space $O(b+k)$.*
3. *Suppose that $S = \gamma k$ and $R = b2^k$ where $\gamma > 0$ is a suitably small positive real number. Suppose M is a PTM and x in an input such that the computation $M(x)$ runs with space bound S and randomness bound R. Then for all but an $\epsilon = R^{-\Omega(1)}$ fraction of choices of $h \in \{0,1\}^{2bk}$, G_h ϵ-fools $M(x)$. Moreover, given the PTM M and input x it is possible to compute such an index $h \in \{0,1\}^{2bk}$ in time $O(R^{O(1)})$ and space $O(bk)$.*

It is easy to transform such a generator family into a single "amalgamated" generator G mapping $(2k+1)b$ bits to $b2^k$ bits: $G = G_{b,k}$ interprets its input as (h, x) where $h \in \{0,1\}^{2bk}$ and $x \in \{0,1\}^b$ and computes $G(h, x) = G_h(x)$. Furthermore, it is easy to see that condition 3 above implies that this amalgamated generator $R^{-\Omega(1)}$ fools every computation running in space $S = \gamma k$. Restricting to a particular choice of parameters we get:

Corollary 4.1 *Let b, k, S be integers with $b = S = \gamma k$ for some suitable $\gamma > 0$. The amalgamated generator built from the family $\mathcal{G}_{b,k}$ is a $2^{-\theta(S)}$-PRG that for space S that produces $2^{\theta(S)}$ bits from a seed of size $\theta(S^2)$.*

Before sketching the construction of Nisan's generator, we discuss the way Theorem 4.1 is used to prove Theorems 3.4 and 3.6. Theorem 3.4 follows almost immediately. First, notice that, while the amalgamated generator requires seeds of length S^2, most of the bits are used only to index the generator within the family.

The generators in the underlying family each require seeds of size only $O(S)$. We know by the last part of the Theorem that for any execution $M(x)$ running in space S, almost all of the generators in the family will fool $M(x)$. If we could get our hands on one such member of the family we could, by running over all seeds, derandomize the computation in space S. The second assertion in the last part of the Theorem implies that we can indeed identify such a generator in $DTISP(2^S, S^2)$. Thus, by first finding the generator and then running over all seeds to it, we obtain a deterministic simulation of $M(x)$ that runs in time polynomial in 2^S and space $\theta(S^2)$. This proves Theorem 3.4.

Theorem 3.6 does not seem to follow as easily from Theorem 4.1. We now sketch the proof. Note that this proof sketch is a digression from the main discussion of generators and can be skipped without loss of continuity.

4.3.1 $BP_HSPACE(s) \subseteq DSPACE(s^{3/2})$

Recall that in order to tell whether a bounded error space S computation accepts, it suffices to estimate a particular entry of the $n = 2^S$ power of the $n \times n$ stochastic matrix P for the associated Markov chain. Thus, to prove Theorem 3.6 it would suffice to find a deterministic algorithm for this approximate exponentiation problem that runs in space $O((\log n)^{3/2})$. More generally, the algorithm given in [42] approximates $P^t[i,j]$ to within $1/n$ and runs in space $O(\log n(\log t)^{1/2})$, for $t \leq n$ where P is an $n \times n$ stochastic matrix. For purposes of the description here we'll assume the entries of P are all 0,1/2 or 1, which is the case for stochastic matrices that model probabilistic machines, and also assume that $t = 2^q$ is a power of 2, but the algorithm can be made to work for arbitrary rational entries and integer exponents.

In the following discussion, we will consider some randomized algorithms that operate in the following way: first they flip all of their random bits, which are all written to memory and then they proceed deterministically. For such an algorithm we distinguish between the space used to store the random bits, and the rest of the space used. We refer to the latter use of space as *processing space*.

At the highest level, the approximation algorithm for matrix exponentiation is obtained by derandomizing a particular randomized algorithm. This randomized algorithm has the following properties: (1) On input P and $t = 2^q$ and indices i, j, the algorithm outputs a value $z \in (0, 1)$ such that with probability at least $1 - 1/n$, z is within $1/n$ of $P^t[i,j]$, (2) The algorithm uses $O(q^{1/2} \log n)$ random bits, and (3) The processing space of the algorithm is at most $O(q^{1/2} \log n)$.

Given such an algorithm, we can easily derandomize it in space $O(q^{1/2} \log n)$ by running the algorithm on the given input for each of the possible random strings, and averaging the outputs obtained. Properties (2) and (3) of the randomized algorithm guarantee that the space of the deterministic algorithm is as desired, and property (1) guarantees that the output of the algorithm is within $2/n$ of $P^t[i,j]$.

The construction of this randomized algorithm satisfying (1),(2) and (3) is developed in several steps. The starting point is an algorithm, algorithm A, that simply simulates the Markov chain defined by P for t steps starting from state i, and outputs $z = 1$ if the final state is j and $z = 0$ otherwise. Note that property (1) is not satisfied but we do have that the expected value of z is $P^t[i,j]$. Furthermore, the processing space used is only $\theta(\log n)$, so property (3) is more than satisfied. However, the number of random bits used is t (one for each step of the Markov chain), which is far more than what is allowed by property (2).

Next, we modify algorithm A using Nisan's generator family $\mathcal{G}_{b,k}$ with $k = q$ and $b = \theta(\log n)$. Recall that the members G_h of $\mathcal{G}_{b,k}$ are indexed by $h \in \{0,1\}^{2bk}$. For each such h, define the algorithm B_h as follows: Select $x \in \{0,1\}^b$ and run algorithm A using the first t bits of the output of $G_h(x)$ in place of the random coins. Next, define the algorithm C_h to be the one obtained by derandomizing B_h: run 2^b trials, one for each choice of x and average the results. Finally, define the algorithm C, that chooses h at random and then performs C_h.

Now the third property of the generator family implies that, if we fix P, t, i, j, then for all but $1/poly(n)$ choices of h, the fraction of accepting trials of algorithm A is within $1/poly(n)$ of the number of accepting trials of B_h (where we can adjust the size of the polynomial by adjusting b by a constant factor). For any such h, the algorithm C_h will output a value that is within $1/poly(n)$ of $P^t[i,j]$. Thus the algorithm C has the property that with probability at least $1 - 1/poly(n)$ it outputs a value that is within $1/poly(n)$ of $P^t[i,j]$, which is stronger than that needed for condition (1).

The number of random bits needed by algorithm C is $\theta(q \log n)$ (to choose h), which exceeds the number required by property (2) by a factor of $q^{1/2}$. On the other hand, the processing space of this algorithm, i.e., the space over that used to record h, is only $\theta(\log n)$ which is better than that required by property (3), by

a factor $q^{1/2}$. The goal now is to try to somehow trade-off the excess number of random bits used against the unused processing space.

The next step is to express the computation of $P^t[i,j]$ recursively. Assume without loss of generality that $q = w^2$ is a perfect square and define $P_0 = P$ and $P_i = P_{i-1}^{2^w}$. Then $P_w = P^{2^q}$ and we may use the recurrence for P_i to get a recursive algorithm for computing P^t. Now, at each level of recursion, instead of computing $P_i = P_{i-1}^{2^w}$ exactly, we can use the algorithm C to estimate it. The resulting recursive algorithm is algorithm D. At each level of recursion, algorithm D is estimating the 2^w power of a matrix, so the call to algorithm C needs $\theta(w \log n) = \theta(q^{1/2} \log n)$ random bits for each level of recursion and $\theta(\log n)$ processing space for each level of recursion. Multiplying by $q^{1/2}$, the number of levels of recursion, we get that the number of random bits use is $\theta(q \log n)$ and the processing space is $\theta(q^{1/2} \log n)$. Comparing this to algorithm C, we see that the number of random bits has not decreased, but the processing space has increased by a factor of $q^{1/2}$.

But notice that the random bits used in algorithm D consist of w strings $h_1, h_2, \ldots, h_w$ each $\theta(w \log n)$ bits, and each used at a different level of recursion. What we'd like to do is to choose one random h of $\theta(w \log n)$ bits and use it in place of all the h_i. However, this introduces dependencies between the various recursive levels, and invalidates the proof of correctness of the algorithm. It is possible that these dependencies are not consequential, but there is currently no way known to prove this.

The last step, which we do not describe in detail, involves introducing a small amount of additional randomness (at most $\theta(q^{1/2} \log n)$) into the algorithm, in the form of small perturbations of the matrix entries at all levels of recursion. These perturbations can be shown to be too small to affect the quality of the answer, but are sufficient to, in effect, overcome the dependencies between recursive levels, and to make it possible to prove that if the same h is used at all recursive levels then the resulting algorithm still works. The number of random bits in the resulting algorithm is $\theta(q^{1/2} \log n)$ as required. Since this final algorithm satisfies properties (1),(2), and (3), it can be derandomized as described above.

4.3.2 A description of Nisan's generator

The construction of the family of generators $\mathcal{G}_{b,k}$ of Theorem 4.1 consists of two steps. The first step is to take $X = \{0,1\}^b$ and define a family $\mathcal{H}_b$ of maps from X to itself. Each map in $\mathcal{H}_b$ is indexed by a $2b$-bit string.

The second step is the definition of a transformation which, given a set X and a sequence $f = (f_1, f_2, \ldots, f_k)$ of functions from X to itself, yields a function G_f mapping X to X^{2^k}.

Using these two parts we construct $\mathcal{G}_{b,h}$. We need to associate to each $2bk$-bit sequence h a function G_h mapping b bits to $b2^k$ bits. Given $h \in \{0,1\}^{2b}$, interpret h as indexing a sequence $(h_1, h_2, \ldots, h_k)$ of function from the family $\mathcal{H}_b$ given in the first step; the function associated with h is then G_h as defined in the second step.

Let us then describe these two steps. For the first step, we recall the well known fact that for every $b \geq 1$, there is a field on 2^b elements and that there is a representation of the elements by b bit binary strings such that addition and multiplication can be carried out in $O(b)$ space. Fix such a representation, and define U_b to be the class of affine linear functions, i.e., functions of the form $x \longrightarrow \alpha x + \beta$ where $\alpha, \beta \in X$. This family maps X to itself and each map is indexed by the $2b$ bit string (α, β).

Now for the second step. For convenience, let $F(X)$ be the set of maps from X to itself. Let $\oplus$ denote the concatenation of strings. Given $f = (f_1, f_2, \ldots, f_k)$ where $f_i \in F(X)$, define the function $G_{f_1,\ldots,f_k}$ recursively by $G_{f_1}(x) = x \oplus f_1(x)$ and for $k \geq 2$,

$$G_{f_1,\ldots,f_k}(x) = G_{f_1,f_2,\ldots,f_{k-1}}(x) \oplus G_{f_1,f_2,\ldots,f_{k-1}}(f_k(x))$$

For example, when $k = 3$, $f = (f_1, f_2, f_3)$ we have

$$\begin{aligned} G_f(x) \quad = \quad & x \oplus f_1(x) \oplus f_2(x) \oplus f_1(f_2(x)) \oplus f_3(x) \oplus \\ & f_1(f_3(x)) \oplus f_2(f_3(x)) \oplus f_1(f_2(f_3(x))). \end{aligned}$$

4.4 The INW generator for distributed networks

Impagliazzo, Nisan and Wigderson [19] initiated an investigation into generators that can be used for distributed computation. In this setting, processors, arranged in some network, are running a randomized protocol. The random bits they use come from a common source. Assume that each processor needs r random bits and that there are n processors altogether. The problem is to construct a generator that maps a small m bit seed to rn bits that can be distributed to the various processors, so that the resulting computation behaves as though the distributed bits were truly random. They construct such a generator whose seed length depends on two parameters: the *width* of the graph (which is a rough measure of the "communication bandwidth" the graph provides) and the max-

imum number c of bits sent or received by any one processor.

As one of the applications of their generator they give an alternative construction of a generator that, for each S, maps a seed length of size $O(S^2)$ bits to a string of $2^{\theta(S)}$ bits that fools any space S computation. This generator, however, does not give a generator family, and therefore can not be substituted for Nisan's generator in the proofs of Theorems 3.4 and 3.6.

The connection between PRGs for space-bounded computation and for networks is in the same spirit, but is more straightforward than, the connection between space-bounded computation and multiparty communication. Given a HSBPTM execution $M(x)$ with space bound S, we know that the computation takes at most 2^S steps. We can simulate the computation on a network consisting of 2^S nodes arranged in a line. The first processor begins by simulating the first step of the computation and then sends the current configuration to the next processor who does the second step of the computation, etc. The maximum communication per processor is $\theta(S)$.

In the special case corresponding to PRGs for space-bounded computation, the INW generator is most easily described in terms of three parameters p, m, i. The generator $F^i_{p,m}$ takes an $p + im$ bit seed and maps it to $2^i p$ bits. For any S, the generator $i2^{-m/2+S}$-fools all space S computations. Taking $p = i = m = 4S$ yields a generator mapping $\theta(S^2)$ bits to $2^{\theta(S)}$ bits that $S2^{-S}$ fools all space S computations.

For $i = 1$, the generator $F_{p,m} = F^1_{p,m}$ mapping $p+m$ to $2m$ bits is defined in terms of two functions $L_{p,m}$ and $R_{p,m}$, each mapping $p+m$ bits to m bits. $F^1_{p,m}(x)$ is defined to be the concatenation $L_{p,m}(x) \oplus R_{p,m}(x)$. To define $L_{p,m}$ and $R_{p,m}$, let $H^{p,m}$ be an efficiently constructible expander graph with 2^p vertices and degree 2^m. As in the description of the AKS sampler, we view each edge as a pair of oppositely directed edges, and assume that the edges out of each vertex are labeled 1 to 2^m. On input a string of $p + m$ bits, interpret the string as a pair (v, i) where v is the vertex indexed by the first p bits and i is an integer (between 1 and 2^m) indexing one of the edges out of v. Define $L_{p,m}(v, i) = v$ and $R_{p,m}(v, i) = w$ where w is the other endpoint of the indicated edge.

For $i \geq 2$, the generator $F^i_{p,m}$ is defined recursively in a way that closely resembles the recursive definition of $G_{f_1,f_2,\ldots,f_k}$ in Nisan's generator. For $i \geq 2$, $F^i_{p,m}$ is supposed to map $p + im$ bits to $2^i m$ bits. For $v \in \{0,1\}^{p+(i-1)m}$ and $i \in \{0,1\}^m$, $F^i_{p,m}(v, i)$ is defined to be $F^{i-1}_{p,m}(L_{p+im,m}(v, i)) \oplus F^{i-1}_{p,m}(R_{p+im,m}(v, i))$.

4.5 Extractors and the Nisan-Zuckerman generator

Recall that the AKS construction of a PRS for space S, mapped $\theta(S)$ bits to $\theta(S^2/\log S)$ bits. Because the size of the seed was only $\theta(S)$, this led to a deterministic simulation of a non-trivial part of $R_H SPACE(s)$ in $DSPACE(s)$.

If one examines the other generators described above, it can be seen that these generators only magnify a seed of size $\theta(S)$ by a constant factor. one only gets a non-trivial expansion of the random seed if the seed is asymptotically larger than S. This means that these generators do not provide a simulation of a non-trivial part of $BP_H SPACE(s)$ in $DSPACE(s)$, at least not in any obvious way. To get such a simulation, we would need to build generators for space S that nontrivially magnify a $\theta(S)$ bit seed. Such generators were found by Nisan and Zuckerman. Their construction gave generators that mapped a $\theta(S)$ bit seed to a $poly(S)$ bit string, for any fixed polynomial, fooling space S computations. As mentioned earlier, this construction together with the naive deterministic simulation, yields Theorem 3.5.

At the center of their construction is a combinatorial construction called an *extractor*. A thorough survey of extractors is given by Nisan [33]; we content ourselves with a short discussion. Extractors first arose in the following context. Suppose we have a random source that outputs bits that are "faulty"; a k-bit string produced by the source may have some biased bits, or dependencies among its bits. We would like to find a mapping that maps the k-bit output of the source to an m-bit string for some m that is less, but not too much less than k. We want this mapping to have the property that when applied to a k-bit string from the faulty source, its output is uniformly (or nearly uniformly) distributed over all m-bit strings. We make the assumption that the faulty source comes from some distribution that is "not too concentrated". This assumption is formalized by requiring that the probability distribution of the source assigns probability at most $2^{-\delta k}$ to any k-bit string, where $\delta \in (0, 1)$ is some constant. Such a random source is called a *δ-source*. It is not hard to see that there is no single mapping that will work for all δ-sources. However, suppose that in addition to the k-bit faulty source we have access to a "short" string of t bits from a genuine source. Then it turns out that it is possible to find a function E that maps a pair (x, y) where x is a k-bit string from the faulty source and y is a perfectly random t bit string to an m-bit string $E(x, y)$ (for some m reasonably close to k) such that the output is very

close to the uniform distribution, assuming only that x comes from some δ-source. Here we measure the distance of the output from the uniform distribution by *statistical distance*: for two distributions D_1 and D_2 on the same set X, the distance between them is defined to be the maximum over $Y \subseteq X$ of $|D_1(Y) - D_2(Y)|$.

Such a function E is called an *extractor* because it enables one to "extract" nearly uniform randomness from faulty randomness. Formally, a $(k, t, m, \delta, \epsilon)$-extractor is a function $E : \{0,1\}^k \times \{0,1\}^t \longrightarrow \{0,1\}^m$ such that if D_k is any δ-source on k bits, then for x selected according to D_k and y a t-bit string chosen uniformly at random, the string $y \oplus E(x, y)$ (the concatenation of y and $E(x, y)$), is within statistical distance ϵ of the uniform distribution on $t+m$-bit strings.

The generator they construct uses an extractor in the following way. Given a $(k, t, m, \delta, \epsilon)$-extractor E, and an integer r, we can define a generator $G_{E,r}$ mapping $k + tr$ bits to tm bits as follows: interpret the input as a sequence $(x, y_1, y_2, \ldots, y_r)$ where x is a k-bit string and each y_i is a t-bit string. The output of the generator is the concatenation $E(x, y_1) \oplus E(x, y_2) \oplus \ldots \oplus E(x, y_k)$. The intuition behind why this generator fools small space computations is as follows. Suppose that the computation has used the first j blocks of random bits, and we compute the next block by selecting y_{j+1} and computing $E(x, y_{j+1})$. Then the current state of the computation encodes some information about the random string x, and conditioned on this information, x becomes a δ-source, for some δ. The string y_{j+1} provides "fresh" random bits, and so the output $E(x, y_{j+1})$ of the extractor is very close to being a uniform string, even when conditioned on the current state of the computation.

To build a generator that fools space S computations, they choose some $\gamma > 0$, $k = \theta(S)$, $t = \theta(S^{1-\gamma})$, and $r = \theta(S^\gamma)$. Then the input to the generator has $\theta(S)$ bits. Nisan and Zuckerman show that, for these parameters, there is an extractor for these values of k and t, $\delta = 1/2$ and $\epsilon = 2^{-\theta(S)}$, whose output size is $m = \theta(S)$. Thus the output of the generator is $\theta(S^{1+\gamma})$ bits. It can further be shown that this is a $2^{-\theta(S)}$-PRG for space S.

Thus, we can magnify $\theta(S)$ random bits to $\theta(S^{1+\gamma})$ pseudorandom bits that fool space S machines. By composing generators of this form some constant number of times, we get a PRG that stretches $\theta(S)$ bits to $poly(S)$ bits for any desired polynomial.

We make one final remark. The Nisan and INW generators for space S stretch $\theta(S^2)$ bits to $2^{\theta(S)}$ bits, while the Nisan-Zuckerman generator stretches $\theta(S)$ bits to $poly(S)$ bits. One might hope that one could compose these generators to get a generator mapping S bits to 2^S bits (first apply the NZ generator and then apply Nisan's generator to the output). This does not work for two reasons. First, the resulting generator is not computable in space S, because when the inner generator is evaluated on S bits, the S^2 bits it produces must be written down in order to evaluate the outer generator. The second problem, which really stems from the first, is that it is not clear that the resulting generator fools space S computations. It turns out that in general, for the composition of two generators to fool space S, one seems to need the condition that the outer generator is computable in space S, which is not the case here.

4.6 The Armoni-Wigderson generator

The Armoni-Wigderson generator [6], is a modification of the Nisan-Zuckerman generator, and provides a single construction for fooling space s computation that (1) stretches $\theta(S)$ bits to $poly(S)$ bits (matching the Nisan Zuckerman generator), (2) stretches $\theta(S^2)$ bits to $2^{\theta(S)}$ bits, (matching the Nisan and INW generators) and (3) for seed sizes in the intermediate range $\theta(S^{1+\gamma})$ for $0 < \gamma < 1$, does slightly better than any of the other generators, producing $S^{\theta(S^\gamma)}$ output bits as compared to the Nisan and INW generators which produces $2^{\theta(S^\gamma)}$ bits.

The generator is built recursively by composing the same extractor-based generators used by Nisan-Zuckerman. The improvement comes from two sources: first, they use an improved extractor construction of Zuckerman [51]. The more significant difference comes in how they compose the generators. In the Nisan-Zuckerman generator, the seed size grows exponentially in the number of nested levels of composition, while in this generator, the seed size only grows linearly in the number of nested levels of composition. The composition method used for this generator is related to that used in both the Nisan and INW generators.

5 Other Directions

We conclude this paper with a brief discussion of some additional topics in space-bounded probabilistic computation not covered earlier.

5.1 Read-once versus multi-access random bits

In specifying the model of probabilistic space-bounded computation, we made the assumption that the random bits for the computation are on a separate tape that is read by a one-way head. This means that each bit, once accessed, can not be reread, unless it is written on the work tape. An alternative model is to have the random bits written on a two-way readable

tape, so that each bit can be accessed arbitrarily often. Such a model was proposed in [7] and studied further in [32]. Following [32], for each probabilistic complexity class $XSPACE(s)$, we denote the corresponding class with multi-access random bits by $X^*SPACE(s)$.

For non-halting probabilistic computation, multi-access is extremely powerful; it can be shown, for instance, that $R^*SPACE(s)$ contains NP. Many of the results that hold for read-once random bits are not clear in the multi-access case, for instance, in this case it is not clear that a computation that has an upper bound on the number of random bits used can be assumed to halt absolutely.

There are two main results about these new classes in the literature. The first, which is obtained by plugging Nisan's generator into a simulation given in [7], says that $BP_HSPACE(s)$ can be simulated in $BP^*_{s^2}SPACE(s)$, i.e., by space s PTMs that use at most $\theta(s^2)$ multi-access random bits. The second [32] says that $BP_HSPACE(s)$ can be simulated in $ZP^*_HSPACE(s)$, i.e., that multi-access computation can be used to convert bounded-error machines to zero-error machines.

5.2 Graph Connectivity and Universal Traversal sequences

The $USTCON$ problem is the most important combinatorial problem known to be in R_HL but not known to be in DL, and there has been considerable effort to find space efficient deterministic algorithms for the problem. One major approach is based on the idea of a *universal traversal sequence*(UTS). Universal traversal sequences can be viewed, roughly, as a type of pseudorandom generator that is specially designed to be used to derandomize the random walk algorithm for $USTCON$. The goal is to explicitly construct such sequences that are as short as possible, preferably polynomial in n, the size of the graph on which the $USTCON$ problem is being solved. As noted in [7], a good PRG for log-space can be used to construct a short UTS, and, the shortest explicit UTS known, having size $n^{\theta(\log n)}$, is built in this way from Nisan's generator.

A thorough overview of the complexity theoretic aspects of the graph connectivity problem, including universal traversal sequences, can be found in Wigderson's survey [49].

5.3 Hitting sets and generators for combinatorial rectangles

Another special case of the problem of constructing a small space pseudorandom generator can be formulated as follows. Let $m = 2^q, d$ be integers and let X denote the set $\{0,1\}^q$, so $|X| = m$. A combinatorial rectangle in X^d, or (m,d)-rectangle is a subset of X^d of the form $R = R_1 \times R_2 \times \ldots \times R_d$, where $R_i \subseteq X$. The *volume* of R, $vol(R)$ is the fraction of points of X^d in R, i.e., $|R|/m^d$. Given a generator G whose output length is qd we can view the output of G as specifying a point in X^d. A generator G is said to *ϵ-fool combinatorial rectangles* if for any such rectangle R, the probability that $G(w) \in R$ is within ϵ of $vol(R)$ and is said to *ϵ-samples (m,d)-rectangles* if for any such rectangle R, if $vol(R) \geq \epsilon$, then the probability that $G(w) \in R$ is strictly positive. Note that this latter condition says that the set H which is the range of $G(w)$ intersects or *hits* all (m,d)- rectangles of volume at least ϵ.

Versions of this problem were discussed in [30] and [12]. The problem of constructing such generators can be viewed as a special case of the problem of constructing small space pseudorandom generators as follows. Suppose we have a space $O(S)$ computation which is decomposed as the AND of a sequence of d independent subcomputations, where each subcomputation requires at most S bits. The entire computation requires Sd bits, and the set of accepting strings is a combinatorial rectangle whose volume is equal to the probability of acceptance.

Using this correspondence, it can be shown that the problem of building an efficient pseudorandom sampler for space bounded computation that maps $\theta(S)$ bits to $2^{\theta(S)}$ bits, includes as a special case the following as a subproblem: for each m, d, ϵ construct an efficient generator for that ϵ-samples (m,d)-rectangles and whose seed length is at $O(\log m + \log d + \log(1/\epsilon))$. This subproblem was solved in [26]. The corresponding problem of building such a generator that ϵ-fools (m,d)-rectangles is open; the best known constructions are (1) a generator with seed length $\log m + O(\log^2 d + \log d \log(1/\epsilon))$ which can be constructed using the INW generator, and (2) a construction in [5], which gives such a generator whose seed length is $O(\log m + \log d + \log^2(1/\epsilon))$.

5.4 Amplification and computing with weak random sources

Some of the most interesting recent work in *time-bounded* probabilistic computation has concerned two related problems, the problem of deterministic amplification, and the problem of computing with weak random sources.

As discussed earlier, given a probabilistic computation that runs with bounded error, it is possible, by repeated trials, to make the probability of error arbitrarily small. Suppose that the algorithm uses r random bits. If one performs k independent trials, us-

ing rk random bits, the error probability is reduced to probability to $2^{-\theta(k)}$. The deterministic amplification problem is to find ways to reduce the error probability by a similar amount using fewer random bits.

The problem of computing weak random sources is as follows. Suppose we are running a BPP algorithm, but the source of bits we have is "faulty". We assume that the random source is a δ-source for some δ (see Section 4.5). Depending on the nature of the faultiness of the source, the algorithm may not work properly. Is it possible to convert the algorithm to one that still works in polynomial time, but is robust in the sense that it works properly with any δ-source?

These two questions have been studied extensively in the context of poly-time computation and there are very strong results for both of them (see [33] for a survey). One can ask the same questions in the context of space-bounded computation. Here there is essentially nothing non-trivial known. All of the methods known in the poly-time case (for either problem) involve generating a large set of random bits and using the same bits repeatedly (but in different ways) over many trials. Nothing like this seems possible in small space, because a large set of random bits can not be stored.

For space-bounded computation, these problems have added significance. It turns out that one can show, for instance, that if there was a general "black box" procedure for converting $BP_HSPACE(s)$ algorithms so that they run with a δ-source, for some $\delta < 1$, then in fact $BP_HSPACE(s) = DSPACE(s)$.

Acknowledgements. Much of my understanding of this area has developed as a result of working with my student and collaborator, Shiyu Zhou. Shiyu also helped in the preparation of this survey by, among other things, collecting the references and preparing the figures. I am very grateful to him for his help. I also benefited greatly from discussions with and comments from Eric Allender, Allan Borodin, Noam Nisan, Avi Wigderson, and David Zuckerman, and gratefully acknowledge their contributions to this paper.

References

[1] L. Adleman. Two theorems on random polynomial time. In *19th IEEE Symposium on Foundations of Computer Science*, pages 75–83, 1978.

[2] M. Ajtai, J. Komlós, and E. Szemerédi. Deterministic simulation of logspace. In *19th ACM Symposium on Theory of Computing*, pages 132–140, 1987.

[3] R. Aleliunas, R. Karp, R. Lipton, L. Lovasz, and C. Rackoff. Random walks, universal sequences and the complexity of maze problems. In *20th IEEE Symposium on Foundations of Computer Science*, pages 218–223, 1979.

[4] E. Allender and M. Ogihara. Relationships among **pl**, #**l**, and the determininant. In *9th Annual Structure in Complexity Theory Conference*, pages 267–279, 1994.

[5] R. Armoni, M. Saks, A. Wigderson, and S. Zhou. Pseudorandom generators for combinatorial rectangles. Manuscript, 1996.

[6] R. Armoni and A. Wigderson. Pseudorandomness for space bounded computations. 1995.

[7] L. Babai, N. Nisan, and M. Szegedy. Multiparty protocols and logspace-hard pseudorandom sequences. In *21st ACM Symposium on Theory of Computing*, pages 1–11, 1989.

[8] A. Borodin, S. Cook, P. Dymond, W. Ruzzo, and M. Tompa. Two applications of inductive counting for complementation problems. *SIAM Journal of Computing*, 18:559–578, 1989.

[9] A. Borodin, S. Cook, and N. Pippenger. Parallel computation and well-endowed rings and space-bounded probabilistic machines. *Information and Control*, 58:113–136, 1983.

[10] A. Chandra, M. Furst, and R. Lipton. Multi-party protocols. In *15th ACM Symposium on Theory of Computing*, pages 94–99, 1983.

[11] C. Dwork and L. Stockmeyer. A time-complexity gap for two-way probabilistic finite state automata. *SIAM Journal of Computing*, 19:1011–1023, 1990.

[12] G. Even, O. Goldreich, M. Luby, N. Nisan, and B. Velicković. Approximations of general independent distributions. In *24th ACM Symposium on Theory of Computing*, pages 10–16, 1992.

[13] R. Freivalds. Probabilistic two-way machines. *Proceedings of the International Symposium on Mathematical Foundations of Computer Science, Springer-Verlag Lecture Notes in Computer Science*, 188:33–45, 1981.

[14] O. Gabber and Z. Galil. Explicit construction of linear-sized superconcentrators. *Journal of Computer and System Science*, 22:407–420, 1981.

[15] J. Gill. Computational complexity of probabilistic turing machines. *SIAM Journal of Computing*, 6:675–695, 1977.

[16] J. Gill, J. Hunt, and J. Simon. Deterministic simulation of tape-bounded probabilistic turing machine transducers. *Theoretical Computer Science*, 12:333–338, 1980.

[17] G. Greenberg and A. Weiss. A lower bound for probabilistic algorithms for finite state machines. *Journal of Computer and System Science*, 33:88–105, 1986.

[18] N. Immerman. Nondeterministic space is closed under complementation. *SIAM Journal of Computing*, 17:935–938, 1988.

[19] R. Impagliazzo, N. Nisan, and A. Wigderson. Pseudorandomness for network algorithms. In *26th ACM Symposium on Theory of Computing*, pages 356–364, 1994.

[20] H. Jung. Relationships between probabilistic and deterministic tape complexity. *10th Symposium on Mathematical Foundations of Computer Science, Lecture Notes in Computer Science*, 118:339–346, 1981.

[21] H. Jung. On probabilistic tape complexity and fast circuits for matrix inversion problems. *11th International Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science*, 172:281–291, 1984.

[22] H. Jung. On probabilistic time and space. *12th International Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science*, 194:310–317, 1985.

[23] J. Kaneps. Regularity of one-letter languages acceptable by 2-way finite probabilistic automata. *Proceedings of Fundamentals of Computation Theory, Springer-Verlag Lecture Notes in Computer Science*, 529:287–296, 1991.

[24] J. Kaneps and R. Freivalds. Minimal nontrivial space complexity of probabilistic one-way turing machines. *Proceedings of the Conference on Mathematical Foundations of Computer Science, Springer-Verlag Lecture Notes in Computer Science*, 452:355–361, 1990.

[25] H. Lewis and C. Papadimitiou. Symmetric space-bounded computation. *Theoretical Computer Science*, 19:161–187, 1982.

[26] N. Linial, M. Luby, M. Saks, and D. Zuckerman. Efficient construction of a small hitting set for combinatorial rectangles in high dimension. Manuscript. A preliminary version of this paper appeared in the proceedings of the 25th ACM Symposium on Theory of Computing, pages 258-267, 1993.

[27] A. Lubotzky, R. Phillips, and P. Sarnak. Explicit expanders and the ramanujan conjectures. In *18th ACM Symposium on Theory of Computing*, pages 240–246, 1986.

[28] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[29] N. Nisan. *Using hard problems to create pseudorandom generators*. The MIT Press, 1991. (An ACM Distinguished Dissertation).

[30] N. Nisan. Pseudorandom generators for space-bounded computation. *Combinatorica*, 12(4):449–461, 1992.

[31] N. Nisan. $RL \subseteq SC$. In *24th ACM Symposium on Theory of Computing*, pages 619–623, 1992.

[32] N. Nisan. On read-once vs. multiple access to randomness in logspace. *Theoretical Computer Science*, 107:135–144, 1993.

[33] N. Nisan. Extracting randomness: How and why. In *11th Annual Conference on Computational Complexity*, 1996.

[34] N. Nisan, E. Szemerédi, and A. Wigderson. Undirected connectivity in $o(log^{1.5}n)$ space. In *30th IEEE Symposium on Foundations of Computer Science*, pages 24–29, 1992.

[35] N. Nisan and A. Ta-Shma. Symmetric logspace is closed under complement. In *27th ACM Symposium on Theory of Computing*, pages 140–146, 1995.

[36] N. Nisan and A. Wigderson. Hardness vs. randomness. In *29th IEEE Symposium on Foundations of Computer Science*, pages 2–12, 1988.

[37] N. Nisan and D. Zuckerman. More deterministic simulation in logspace. In *25th ACM Symposium on Theory of Computing*, pages 235–244, 1993. Revised version *Randomness is linear in space* to appear in JCSS.

[38] C. Papadimitriou. *Computational Complexity*. Addison Wesley, 1994.

[39] M. Rabin. Probabilistic automata. *Information and Control*, 6:230–245, 1963.

[40] M. Rabin. Probabilistic algorithms. In J.F.Traub, editor, *Algorithms and Complexity: Recent Results and New Directions*, pages 21–39. Academic Press, 1976.

[41] W. Ruzzo, J. Simon, and M. Tompa. Space-bounded hierarchies and probabilistic computation. *Journal of Computer and System Science*, 28:216–230, 1984.

[42] M. Saks and S. Zhou. $RSPACE(s) \subseteq DSPACE(s^{3/2})$. In *36th IEEE Symposium on Foundations of Computer Science*, pages 344–353, 1995.

[43] W.J. Savitch. Relationships between nondeterministic and deterministic space complexities. *Journal of Computer and System Science*, 4(2):177–192, 1970.

[44] J. Simon. On tape-bounded probabilistic turing machine acceptors. *Theoretical Computer Science*, 16:75–91, 1981.

[45] J. Simon. Space-bounded probabilistic turing machine complexity classes are closed under complement. In *13th ACM Symposium on Theory of Computing*, pages 158–167, 1981.

[46] M. Sipser. A complexity theoretic approach to randomness. In *15th ACM Symposium on Theory of Computing*, pages 330–335, 1983.

[47] R. Solovay and V. Strassen. A fast monte-carlo test for primality. *SIAM Journal of Computing*, 6(1):84–85, 1977.

[48] R. Szelepcsényi. The method of forced enumeration for nondeterministic automata. *Acta Informatica*, 26:279–284, 1988.

[49] A. Wigderson. The complexity of graph connectivity. *Mathematical Foundations of Computer Science: Proceedings, 17th Symposium, Lecture Notes in Computer Science*, 629:112–132, 1992.

[50] A. C. Yao. Theory and applications of trapdoor functions. In *23rd IEEE Symposium on Foundations of Computer Science*, pages 80–91, 1982.

[51] D. Zuckerman. Randomness-optimal sampling, extractors and constructive leader election. In *28th ACM Symposium on Theory of Computing*, 1996.