SPACE HIERARCHY RESULTS FOR RANDOMIZED MODELS

JEFF KINNE [1] AND DIETER VAN MELKEBEEK [1]

[1] Department of Computer Sciences, University of Wisconsin-Madison, USA
E-mail address: {jkinne,dieter}@cs.wisc.edu

Abstract. We prove space hierarchy and separation results for randomized and other semantic models of computation with advice. Previous works on hierarchy and separation theorems for such models focused on time as the resource. We obtain tighter results with space as the resource. Our main theorems are the following. Let $s(n)$ be any space-constructible function that is $\Omega(\log n)$ and such that $s(an) = O(s(n))$ for all constants a, and let $s'(n)$ be any function that is $\omega(s(n))$.

There exists a language computable by *two-sided* error randomized machines using $s'(n)$ space and one bit of advice that is not computable by *two-sided* error randomized machines using $s(n)$ space and $\min(s(n), n)$ bits of advice.

There exists a language computable by *zero-sided* error randomized machines in space $s'(n)$ with one bit of advice that is not computable by *one-sided* error randomized machines using $s(n)$ space and $\min(s(n), n)$ bits of advice.

The condition that $s(an) = O(s(n))$ is a technical condition satisfied by typical space bounds that are at most linear. We also obtain weaker results that apply to generic semantic models of computation.

1. Introduction

A hierarchy theorem states that the power of a machine increases with the amount of resources it can use. Time hierarchy theorems on deterministic Turing machines follow by direct diagonalization: a machine N diagonalizes against every machine M_i running in time t by choosing an input x_i, simulating $M_i(x_i)$ for t steps, and then doing the opposite. Deriving a time hierarchy theorem for nondeterministic machines is more complicated because a nondeterministic machine cannot easily complement another nondeterministic machine (unless NP=coNP). A variety of techniques can be used to overcome this difficulty, including translation arguments and delayed diagonalization [4, 13, 16].

In fact, these techniques allow us to prove time hierarchy theorems for just about any *syntactic* model of computation. We call a model syntactic if there exists a computable

2000 ACM Subject Classification: F.1.3 [**Computation By Abstract Devices**]: Complexity Measures And Classes – *complexity hierarchies*.

Key words and phrases: Computations with Advice, Space Hierarchy, Randomized Machine, Promise Classes, Semantic Models.

First author supported and second author partially supported by NSF Career award CCR-0133693.

enumeration of all machines in the model. For example, we can enumerate all nondeterministic Turing machines by representing their transition functions as strings and then iterating over all such strings to discover each nondeterministic Turing machine.

Many models of computation of interest are not syntactic, but *semantic*. A semantic model is defined by imposing a promise on a syntactic model. A machine belongs to the model if it is output by the enumeration of the underlying syntactic model and its execution satisfies the promise on every input. Bounded-error randomized Turing machines are an example of a non-syntactic semantic model. There does not exist a computable enumeration consisting of exactly all randomized Turing machines that satisfy the promise of bounded error on every input, but we can enumerate all randomized Turing machines and attempt to select among them those that have bounded error. In general promises make diagonalization problematic because the diagonalizing machine must satisfy the promise everywhere but has insufficient resources to determine whether a given machine from the enumeration against which it tries to diagonalize satisfies the promise on a given input.

Because of these difficulties there has yet to be a single non-trivial proof of a time hierarchy theorem for any non-syntactic model. A recent line of research [1, 5, 7, 6, 11] has provided progress toward proving time hierarchy results for non-syntactic models, including two-sided error randomized machines. Each of these results applies to semantic models that take advice, where the diagonalizing machine is only guaranteed to satisfy the promise when it is given the correct advice. Many of the results require only one bit of advice, which the diagonalizing machine uses to avoid simulating a machine on an input for which that machine breaks the promise.

As opposed to the setting of time, fairly good space hierarchy theorems are known for certain non-syntactic models. In fact, the following simple translation argument suffices to show that for any constant $c > 1$ there exists a language computable by two-sided error randomized machines using $(s(n))^c$ space that is not computable by such machines using $s(n)$ space [10], for any space-constructible $s(n)$ that is $\Omega(\log n)$. Suppose by way of contradiction that every language computable by two-sided error machines in space $(s(n))^c$ is also computable by such machines in space $s(n)$. A padding argument then shows that in that model any language computable in $(s(n))^{c^2}$ space is computable in space $(s(n))^c$ and thus in space $s(n)$. We can iterate this padding argument any constant number of times and show that for any constant d, any language computable by two-sided error machines in space $(s(n))^d$ is also computable by such machines in $s(n)$ space. For $d > 1.5$ we reach a contradiction with the deterministic space hierarchy theorem because randomized two-sided error computations that run in space $s(n)$ can be simulated deterministically in space $(s(n))^{1.5}$ [12]. The same argument applies to non-syntactic models where $s(n)$ space computations can be simulated deterministically in space $(s(n))^d$ for some constant d, including one- and zero-sided error randomized machines, unambiguous machines, etc.

Since we can always reduce the space usage by a constant factor by increasing the work-tape alphabet size, the tightest space hierarchy result one might hope for is to separate space $s'(n)$ from space $s(n)$ for any space-constructible function $s'(n) = \omega(s(n))$. For models like nondeterministic machines, which are known to be closed under complementation in the space-bounded setting [8, 14], such tight space hierarchies follow by straightforward diagonalization. For generic syntactic models, tight space hierarchies follow using the same techniques as in the time-bounded setting. Those techniques all require the existence of an efficient universal machine, which presupposes the model to be syntactic. For that reason they fail for non-syntactic models of computation such as bounded-error machines.

In this paper we obtain space hierarchy results that are tight with respect to space by adapting to the space-bounded setting techniques that have been developed for proving hierarchy results for semantic models in the time-bounded setting. Our results improve upon the space hierarchy results that can be obtained by the simple translation argument.

1.1. Our Results

Space hierarchy results have a number of parameters: (1) the gap needed between the two space bounds, (2) the amount of advice that is needed for the diagonalizing machine N, (3) the amount of advice that can be given to the smaller space machines M_i, and (4) the range of space bounds for which the results hold. We consider (1) and (2) to be of the highest importance. We focus on space hierarchy theorems with an optimal separation in space – where any super-constant gap in space suffices. The ultimate goal for (2) is to remove the advice altogether and obtain uniform hierarchy results. As in the time-bounded setting, we do not achieve this goal but get the next best result – a single bit of advice for N suffices in each of our results. Given that we strive for space hierarchies that are tight with respect to space and require only one bit of advice for the diagonalizing machine, we aim to optimize the final two parameters.

1.1.1. *Randomized Models.* Our strongest results apply to randomized models. For two-sided error machines, we can handle a large amount of advice and any typical space bound between logarithmic and linear. We point out that the latter is an improvement over results in the time-bounded setting, in the sense that there tightness degrades for all super-polynomial time bounds whereas here the results remain tight for a range of space bounds.

Theorem 1.1. *Let $s(n)$ be any space-constructible function that is $\Omega(\log n)$ and such that $s(an) = O(s(n))$ for all constants a, and let $s'(n)$ be any function that is $\omega(s(n))$. There exists a language computable by two-sided error randomized machines using $s'(n)$ space and one bit of advice that is not computable by two-sided error randomized machines using $s(n)$ space and $\min(s(n), n)$ bits of advice.*

For $s(n) = \log(n)$, Theorem 1.1 gives a bounded-error machine using only slightly larger than $\log n$ space that uses one bit of advice and differs from all bounded-error machines using $O(\log n)$ space and $O(\log n)$ bits of advice. The condition that $s(an) = O(s(n))$ for all constants a is a technical condition needed to ensure the construction yields a tight separation in space. The condition is true of all natural space bounds that are at most linear. More generally, our construction works for arbitrary space bounds $s(n)$ and space-constructible $s'(n)$ such that $s'(n) = \omega(s(n + as(n)))$ for all constants a.

Our second result gives a separation result with similar parameters as those of Theorem 1.1 but for the cases of one- and zero-sided error randomized machines. We point out that the separation result for zero-sided error machines is new to the space-bounded setting as the techniques used to prove stronger separations in the time-bounded setting do not work for zero-sided error machines. In fact, we show a single result that captures space separations for one- and zero-sided error machines – that a zero-sided error machine suffices to diagonalize against one-sided error machines.

Theorem 1.2. *Let $s(n)$ be any space-constructible function that is $\Omega(\log n)$ and such that $s(an) = O(s(n))$ for all constants a, and let $s'(n)$ be any function that is $\omega(s(n))$. There exists a language computable by zero-sided error randomized machines using $s'(n)$ space and*

one bit of advice that is not computable by one-sided error randomized machines using $s(n)$ space and $\min(s(n), n)$ bits of advice.

1.1.2. *Generic Semantic Models.* The above results take advantage of specific properties of randomized machines that do not hold for arbitrary semantic models. Our final results involve a generic construction that applies to a wide class of semantic models which we term *reasonable*. We omit the precise definition due to lack of space; but besides randomized two-, one-, and zero-sided error machines, the notion also encompasses bounded-error quantum machines [15], unambiguous machines [2], Arthur-Merlin games and interactive proofs [3], etc. When applied to the logarithmic space setting, the construction gives a language computable within the model with $s'(n)$ space and one bit of advice that is not computable within the model using $O(\log n)$ space and $O(1)$ bits of advice, for any $s'(n) = \omega(\log n)$.

The performance of the generic construction is poor on the last two parameters we mentioned earlier – it allows few advice bits on the smaller space side and is only tight for $s(n) = O(\log n)$. Either of these parameters can be improved for models that can be simulated deterministically with only a polynomial blowup in space – models for which the simple translation argument works. In fact, there is a trade-off between (a) the amount of advice that can be handled and (b) the range of space bounds for which the result is tight. By maximizing the former we get the following.

Theorem 1.3. *Fix any reasonable model of computation for which space $O(\log n)$ computations can be simulated deterministically in space $O(\log^d n)$ for some rational constant d. Let $s'(n)$ be any function with $s'(n) = \omega(\log n)$. There exists a language computable using $s'(n)$ space and one bit of advice that is not computable using $O(\log n)$ space and $O(\log^{1/d} n)$ bits of advice.*

In fact, a tight separation in space can be maintained while allowing $O(\log^{1/d} n)$ advice bits for $s(n)$ any poly-logarithmic function, but the separation in space with this many advice bits is no longer tight for larger $s(n)$. By maximizing (b), we obtain a separation result that is tight for typical space bounds between logarithmic and polynomial.

Theorem 1.4. *Fix any reasonable model of computation for which space s computations can be simulated deterministically in space $O(s^d)$ for some constant d. Let $s(n)$ be a space bound that is $\Omega(\log n)$ and such that $s(n) \leq n^{O(1)}$; let $s'(n)$ be a space bound that is constructible in space $o(s'(n))$ and such that $s'(n+1) = O(s'(n))$. If $s'(n) = \omega(s(n))$ then there is a language computable in space $s'(n)$ with one bit of advice that is not computable in space $s(n)$ with $O(1)$ bits of advice.*

The first two conditions on $s'(n)$ are technical conditions true of typical space bounds in the range of interest – between logarithmic and polynomial. When applied to randomized machines, Theorem 1.4 gives a tight separation result for slightly higher space bounds than Theorems 1.1 and 1.2, but the latter can handle more advice bits.

1.2. Our Techniques

Recently, Van Melkebeek and Pervyshev [11] showed how to adapt the technique of delayed diagonalization to obtain time hierarchies for any reasonable semantic model of computation with one bit of advice. For any constant a, they exhibit a language that is computable in polynomial time with one bit of advice but not in linear time with a bits of

advice. Our results for generic models of computation (Theorems 1.3 and 1.4) follow from a space-efficient implementation and a careful analysis of that approach. The proofs of these results are omitted here but included in the full paper on our web pages.

Our stronger results for randomized machines follow a different type of argument, which roughly goes as follows. When N diagonalizes against machine M_i, it tries to achieve complementary behavior on inputs of length n_i by reducing the complement of M_i at length n_i to instances of some hard language L of length somewhat larger than n_i, say m_i. N cannot compute L on those instances directly because we do not know how to compute L in small space. We instead use a delayed computation and copying scheme that forces M_i to aid N in the computation of L if M_i agrees with N on inputs larger than m_i. As a result, either M_i differs from N on some inputs larger than m_i, or else N can decide L at length m_i in small space and therefore diagonalize against M_i at length n_i.

The critical component of the copying scheme is the following task. Given a list of randomized machines with the guarantee that at least one of them satisfies the promise and correctly decides L at length m in small space, construct a single randomized machine that satisfies the promise and decides L at length m in small space. We call a procedure accomplishing this task a space-efficient *recovery procedure* for L.

The main technical contributions of this paper are the design of recovery procedures for adequate hard languages L. For Theorem 1.1 we use the computation tableau language, which is an encoding of bits of the computation tableaux of deterministic machines; we develop a recovery procedure based on the local checkability of computation tableaux. For Theorem 1.2 we use the configuration reachability language, which is an encoding of pairs of configurations that are connected in a nondeterministic machine's configuration graph; we develop a recovery procedure from the proof that NL=coNL [8, 14].

We present the basic construction for Theorems 1.1 and 1.2 with the recovery procedures as black boxes in section 3. The recovery procedure for the computation tableau language is given in section 4, and the recovery procedure for the configuration reachability language is given in section 5. Resource analysis of the construction is given in section 6.

1.2.1. *Relation to Previous Work.* Our high-level strategy is most akin to the one used in [11]. In the time-bounded setting, [11] achieves a strong separation for bounded-error randomized machines using the above construction with satisfiability as the hard language L. Hardness of L follows from the fact that randomized machines can be time-efficiently deterministically simulated using a randomized two-sided error algorithm for satisfiability. We point out that some of our results can also be obtained using a different high-level strategy than the one in [11], which can be viewed as delayed diagonalization with advice. Some of the results of [11] in the time-bounded setting can also be derived by adapting translation arguments to use advice [1, 5, 7, 6]. It is possible to derive our Theorems 1.1 and 1.2 following a space-bounded version of the latter strategy. However, the proofs still rely on the recovery procedure as a key technical ingredient and we feel that our proofs are simpler. Moreover, for the case of generic semantic models, our approach yields results that are strictly stronger.

2. Preliminaries

We assume familiarity with standard definitions for randomized complexity classes, including two-, one-, and zero-sided error machines. For each machine model requiring

randomness, we allow the machine one-way access to the randomness and only consider computations where each machine always halts in finite time.

Our separation results apply to machines that take advice. We use α and β to denote infinite sequences of advice strings. Given a machine M, M/β denotes the machine M taking advice β. Namely, on input x, M is given both x and $\beta_{|x|}$ as input. When we are interested in the execution of M/β on inputs of length n, we write M/b where $b = \beta_n$.

We consider semantic models of computation, with an associated computable enumeration $(M_i)_{i=1,2,3,\ldots}$ and an associated promise. A machine falls within the model if it is contained in the enumeration and its behavior satisfies the promise on all inputs.

For a machine M/β^* that takes advice, we only require that M satisfies the promise when given the "correct" advice sequence β^*. We note that this differs from the Karp-Lipton notion of advice of [9], where the machine must satisfy the promise no matter which advice string is given. A hierarchy for a semantic model with advice under the stronger Karp-Lipton notion would imply the existence of a hierarchy without advice.

3. Randomized Machines with Bounded Error

In this section we describe the high-level strategy used to prove Theorems 1.1 and 1.2. Most portions of the construction are the same for both, so we keep the exposition general. We aim to construct a randomized machine N and advice sequence α witnessing Theorems 1.1 and 1.2 for some space bounds $s(n)$ and $s'(n)$. N/α should always satisfy the promise, run in space $s'(n)$, and differ from M_i/β for randomized machines M_i and advice sequences β for which M_i/β *behaves appropriately*, i.e., for which M_i/β satisfies the promise and uses at most $s(n)$ space on all inputs.

As with delayed diagonalization, for each M_i we allocate an interval of input lengths $[n_i, n_i^*]$ on which to diagonalize against M_i. That is, for each machine M_i and advice sequence β such that M_i/β behaves appropriately, there is an $n \in [n_i, n_i^*]$ such that N/α and M_i/β decide differently on at least one input of length n. The construction consists of three main parts: (1) reducing the complement of the computation of M_i on inputs of length n_i to instances of a hard language L of length m_i, (2) performing a delayed computation of L at length m_i on inputs of length n_i^*, and (3) copying this behavior to smaller and smaller inputs down to input length m_i. These will ensure that if M_i/β behaves appropriately, either N/α differs from M_i/β on some input of length larger than m_i, or N/α computes L at length m_i allowing N/α to differ from M_i/b for all possible advice strings b at length n_i. We describe how to achieve (1) for two-sided error machines in section 4 and for one- and zero-sided error machines in section 5. For now, we assume a hard language L and describe (2) and (3).

Let us first try to develop the construction without assuming any advice for N or for M_i and see why N needs at least one bit of advice. On an input x of length n_i, N reduces the complement of $M_i(x)$ to an instance of L of length m_i. Because N must run in space not much more than $s(n)$ and we do not know how to compute the hard languages we use with small space, N cannot directly compute L at length m_i. However, L can be computed at length m_i within the space N is allowed to use on much larger inputs. Let n_i^* be large enough so that L at length m_i can be deterministically computed in space $s'(n_i^*)$. We let N at length n_i^* perform a *delayed computation* of L at length m_i as follows: on inputs of the form $0^\ell y$ where $\ell = n_i^* - m_i$ and $|y| = m_i$, N uses the above deterministic computation of L on input y to ensure that $N(0^\ell y) = L(y)$.

Since N performs a delayed computation of L, M_i must as well – otherwise N already computes a language different than M_i. We would like to bring this delayed computation down to smaller padded inputs. The first attempt at this is the following: on input $0^{\ell-1}y$, N simulates $M_i(0^\ell y)$. If M_i behaves appropriately and performs the initial delayed computation, then $N(0^{\ell-1}y) = M_i(0^\ell y) = L(y)$, meaning that N satisfies the promise and performs the delayed computation of L at length m_i at an input length one smaller than before. However, M_i may not behave appropriately on inputs of the form $0^\ell y$; in particular M_i may fail to satisfy the promise, in which case N would also fail to satisfy the promise by performing the simulation. If M_i does not behave appropriately, N does not need to consider M_i and could simply abstain from the simulation. If M_i behaves appropriately on inputs of the form $0^\ell y$, it still may fail to perform the delayed computation. In that case N has already diagonalized against M_i at input length $m_i + \ell$ and can therefore also abstain from the simulation on inputs of the form $0^{\ell-1}y$.

N has insufficient resources to determine on its own if M_i behaves appropriately and performs the initial delayed computation. Instead, we give N one bit of advice at input length $m_i + \ell - 1$ indicating whether M_i behaves appropriately and performs the initial delayed computation at length $n_i^* = m_i + \ell$. If the advice bit is 0, N acts trivially at this length by always rejecting inputs. If the advice bit is 1, N performs the simulation so $N(0^{\ell-1}y)/\alpha = M_i(0^\ell y) = L(y)$.

If we give N one bit of advice, we should give M_i at least one advice bit as well. Otherwise, the hierarchy result is not fair (and is trivial). Consider how allowing M_i advice effects the construction. If there exists an advice string b such that M_i/b behaves appropriately and $M_i(0^\ell y)/b = L(y)$ for all y with $|y| = m_i$, we set N's advice bit for input length $m_i+\ell-1$ to be 1, meaning N should copy down the delayed computation from length $m_i + \ell$ to length $m_i + \ell - 1$. Note, though, that N does not know for which advice b the machine M_i/b appropriately performs the delayed computation at length $m_i + \ell$. N has at its disposal a list of machines, M_i with each possible advice string b, with the guarantee that at least one M_i/b behaves appropriately and $M_i(0^\ell y)/b = L(y)$ for all y with $|y| = m_i$. With this list of machines as its primary resource, N wishes to ensure that $N(0^{\ell-1}y)/\alpha = L(y)$ for all y with $|y| = m_i$ while satisfying the promise and using small space.

N can accomplish this task given a space-efficient recovery procedure for L at length m_i: on input $0^{\ell-1}y$, N removes the padding and executes the recovery procedure to determine $L(y)$, for each b simulating $M_i(0^\ell y')/b$ when the recovery procedure makes a query y'. As the space complexity of the recovery procedures we give in sections 4 and 5 is within a constant factor of a single simulation of M_i, this process uses $O(s(n))$ space. We point out that for Theorem 1.1, the recovery procedure may have two-sided error, while for Theorem 1.2, the recovery procedure must have zero-sided error.

Given a recovery procedure for L, N/α correctly performs the delayed computation on inputs of length $m_i + \ell - 1$ if there is an advice string causing M_i to behave appropriately and perform the initial delayed computation at length $m_i + \ell$. We repeat the process on padded inputs of the next smaller size. Namely, N's advice bit for input length $m_i + \ell - 2$ is set to indicate if there is an advice string b such that M_i/b behaves appropriately on inputs of length $m_i + \ell - 1$ and $M_i(0^{\ell-1}y)/b = L(y)$ for all y with $|y| = m_i$. If so, then on inputs of the form $0^{\ell-2}y$, N/α uses the recovery procedure for L to determine the value of $L(y)$, for each b simulating $M_i(0^{\ell-1}y')/b$ when the recovery procedure makes a query y'. By the correctness of the recovery procedure, N/α thus correctly performs the delayed

computation on padded inputs of length $m_i + \ell - 2$. If the advice bit is 0, N/α acts trivially at input length $m_i + \ell - 2$ by rejecting immediately.

We repeat the same process on smaller and smaller padded inputs. We reach the conclusion that either there is a largest input length $n \in [m_i + 1, n_i^*]$ where for no advice string b, M_i/b appropriately performs the delayed computation of L at length n; or N/α correctly computes L on inputs of length m_i. If the former is the case, N/α performs the delayed computation at length n whereas for each b either M_i/b does not behave appropriately at length n or it does but does not perform the delayed computation at length n. In either case, N/α has diagonalized against M_i/b for each possible b at length n. N's remaining advice bits for input lengths $[n_i, n-1]$ are set to 0 to indicate that nothing more needs to be done, and N/α immediately rejects inputs in this range. Otherwise N/α correctly computes L on inputs of length m_i. In that case N/α diagonalizes against M_i/b for all advice strings b at length n_i by acting as follows. On input $x_b = 0^{n_i - |b|}b$, N reduces the complement of the computation $M_i(x_b)/b$ to an instance y of L of length m_i and then simulates $N(y)/\alpha$, so $N(x_b)/\alpha = N(y)/\alpha = L(y) = \neg M_i(x_b)/b$.

We have given the major points of the construction, with the notable exception of the recovery procedures. We develop these in the next two sections. We save the resource analysis of the construction for the final section.

4. Two-sided Error Recovery Procedure – Computation Tableau Language

In this section we develop a space-efficient recovery procedure for the computation tableau language (hereafter written COMP), the hard language used in the construction of Theorem 1.1.

> COMP $= \{\langle M, x, t, j\rangle \mid M$ is a deterministic Turing machine, and in the tth time step of executing $M(x)$, the jth bit in the machine's configuration is equal to 1$\}$.

Let us see that COMP is in fact "hard" for two-sided error machines. For some input x, we would like to know whether $\Pr[M_i(x) = 1] < \frac{1}{2}$. For a particular random string, whether $M_i(x)$ accepts or rejects can be decided by looking at a single bit in M_i's configuration after a certain number of steps – by ensuring that M_i enters a unique accepting configuration when it accepts. With the randomness unfixed, we view $M_i(x)$ as defining a Markov chain on the configuration space of the machine. Provided $M_i(x)$ uses at most $s(n)$ space, a deterministic machine running in $2^{O(s)}$ time and space can estimate the state probabilities of this Markov chain to sufficient accuracy and determine whether a particular configuration bit has probability at most 1/2 of being 1 after t time steps. This deterministic machine and a particular bit of its unique halting configuration define the instance of COMP we would like to solve when given input x.

We now present the recovery procedure for COMP. We wish to compute COMP on inputs of length m in space $O(s(m))$ with bounded error when given a list of randomized machines with the guarantee that at least one of the machines computes COMP on all inputs of length m using $s(m)$ space with bounded error. Let $y = \langle M, x, t, j\rangle$ be an instance of COMP with $|y| = m$ that we wish to compute.

A natural way to determine COMP(y) is to consider each machine in the list one at a time and design a test that determines whether a particular machine computes COMP(y). The test should have the following properties:

(i) if the machine in question correctly computes COMP on all inputs of length m, the test declares success with high probability, and
(ii) if the test declares success with high probability, then the machine in question gives the correct answer of COMP(y) with high probability.

Given such a test, the recovery procedure consists of iterating through each machine in the list in turn. We take the first machine P to pass testing, simulate $P(y)$ some number of times and output the majority answer. Given a testing procedure with properties (i) and (ii), correctness of this procedure follows using standard probability arguments (Chernoff and union bounds) and the assumption that we are guaranteed that at least one machine in the list of machines correctly computes COMP at length m.

The technical heart of the recovery procedure is the testing procedure to determine if a given machine P correctly computes COMP(y) for $y = \langle M, x, t, j\rangle$. This test is based on the local checkability of computation tableaux – the jth bit of the configuration of $M(x)$ in time step t is determined by a constant number of bits from the configuration in time step $t-1$. For each bit (t, j) of the tableau, this gives a local consistency check – make sure that the value P claims for $\langle M, x, t, j\rangle$ is consistent with the values P claims for each of the bits of the tableau that this bit depends on. We implement this intuition as follows.

(1) For each possible t' and j', simulate $P(\langle M, x, t', j'\rangle)$ a large number of times and fail the test if the acceptance ratio lies in the range $[3/8, 5/8]$.
(2) For each possible t' and j', do the following. Let j'_1, ..., j'_k be the bits of the configuration in time step $t'-1$ that bit j' in time step t' depends on. Simulate each of $P(\langle M, x, t', j'\rangle)$, $P(\langle M, x, t'-1, j'_1\rangle)$, ..., $P(\langle M, x, t'-1, j'_k\rangle)$ a large number of times. If the majority values of these simulations are not consistent with the transition function of M, then fail the test. For example, if the bit in column j' should not change from time $t'-1$ to time t', but P has claimed different values for these bits, fail the test.

Each time we need to run multiple trials of P, we run $2^{O(s(m))}$ many. The first test checks that P has error bounded away from $1/2$ on input $\langle M, x, t, j\rangle$ and on all other bits of the computation tableau of $M(x)$. This allows us to amplify the error probability of P to exponentially small in $2^{s(m)}$. For some constants $0 < \gamma < \delta < 1/2$, the first test has the following properties: (A) If P passes the test with non-negligible probability then for any t' and j', the random variable $P(\langle M, x, t', j'\rangle)$ deviates from its majority value with probability less than δ, and (B) if the latter is the case with δ replaced by γ then P passes the test with overwhelming probability. The second test verifies the local consistency of the computation tableau claimed by P. Note that if P computes COMP correctly at length m then P passes each consistency test with high probability, and if P passes each consistency test with high probability then P must compute the correct value for COMP(y). This along with the two properties of the first test guarantee that we can choose a large enough number of trials for the second test so that properties (i) and (ii) from above are satisfied.

Consider the space usage of the recovery procedure. The main tasks are the following: (a) cycle over all machines in the list of machines, and (b) for each t' and j' determine the bits of the tableau that bit (t', j') depends on and for each of these run $2^{O(s(m))}$ simulations of P. The first requirement depends on the representation of the list of machines. For our application, we will be cycling over all advice strings for input length m, and this takes $O(s(m))$ space provided advice strings for M_i are of length at most $s(m)$. The second requirement takes an additional $O(s(m))$ space by the fact that we only need to simulate

P while it uses $s(m)$ space and the fact that the computation tableau bits that bit (t', j') depends on are constantly many and can be computed very efficiently.

5. Zero-sided error Recovery Procedure – Configuration Reachability

In this section we develop a space-efficient recovery procedure for the configuration reachability language (hereafter written CONFIG), the hard language used in the construction of Theorem 1.2.

> CONFIG $= \{\langle M, x, c_1, c_2, t\rangle \,|\, M$ is a nondeterministic Turing machine, and on input x, if M is in configuration c_1, then configuration c_2 is reachable within t time steps.

We point out that CONFIG is "hard" for one-sided error machines since a one-sided error machine can also be viewed as a nondeterministic machine. That is, if we want to know whether $\Pr[M_i(x) = 1] < \frac{1}{2}$ for M_i a one-sided error machine that uses $s(n)$ space, we can query the CONFIG instance $\left\langle M_i, x, c_1, c_2, 2^{O(s(|x|))}\right\rangle$ where c_1 is the unique start configuration, and c_2 is the unique accepting configuration.

We now present the recovery procedure for CONFIG. We wish to compute CONFIG on inputs of length m with *zero-sided error* and in space $O(s(m))$ when given a list of randomized machines with the guarantee that at least one of the machines computes CONFIG on all inputs of length m using $s(m)$ space with *one-sided error*. Let $y = \langle M, x, c_1, c_2, t\rangle$ be an instance of CONFIG with $|y| = m$ that we wish to compute.

As we need to compute CONFIG with zero-sided error, we can only output a value of "yes" or "no" if we are sure this is correct. The outer loop of our recovery procedure is the following: cycle through each machine in the list of machines, and for each execute a search procedure that attempts to verify whether configuration c_2 is reachable from configuration c_1. The search procedure may output "yes", "no", or "fail", and should have the following properties:

(i) if the machine in question correctly computes CONFIG at length m, the search procedure comes to a definite answer ("yes" or "no") with high probability, and
(ii) when the search procedure comes to a definite answer, it is always correct, no matter what the behavior of the machine in question.

We cycle through all machines in the list, and if the search procedure ever outputs "yes" or "no", we halt and output that response. If the search procedure fails for all machines in the list, we output "fail". Given a search procedure with properties (i) and (ii), the correctness of the recovery procedure follows from the fact that we are guaranteed that one of the machines in the list of machines correctly computes CONFIG at length m.

The technical heart of the recovery procedure is a search procedure with properties (i) and (ii). Let P be a randomized machine under consideration, and $y = \langle M, x, c_1, c_2, t\rangle$ an input of length m we wish to compute. Briefly, the main idea is to mimic the proof that NL=coNL to verify reachability and un-reachability, replacing nondeterministic guesses with simulations of P. If P computes CONFIG at length m correctly, there is a high probability that we have correct answers to all nondeterministic guesses, meaning property (i) is satisfied. Property (ii) follows from the fact that the algorithm can discover when incorrect nondeterministic guesses have been made. For completeness, we explain how the

nondeterministic algorithm of [8, 14] is used in our setting. The search procedure works as follows.

(1) Let k_0 be the number of configurations reachable from c_1 within 0 steps, i.e., $k_0 = 1$.
(2) For each value $\ell = 1, 2, ..., t$, compute the number k_ℓ of configurations reachable within ℓ steps of c_1, using only the fact that we have remembered the value $k_{\ell-1}$ that was computed in the previous iteration.
(3) While computing k_t, experience all of these configurations to see if c_2 is among them.

Consider the portion of the second step where we must compute k_ℓ given that we have already computed $k_{\ell-1}$. We accomplish this by cycling through all configurations c and for each one re-experiencing all configurations reachable from c_1 within $\ell-1$ steps and verifying whether c can be reached in at most one step from at least one of them. To re-experience configurations reachable within distance $\ell - 1$, we try all possible configurations and query P to verify a nondeterministic path to each. To check if c is reachable within one step of a given configuration, we use the transition function of M. If we fail to re-experience all $k_{\ell-1}$ configurations or if P gives information inconsistent with the transition function of M at any point we consider the search for reachability/un-reachability failed with machine P.

An examination of the algorithm reveals that it has property (ii) from above: if the procedure reaches a "yes" or "no" conclusion for reachability, it must be correct. Further, by using a large enough number of trials each time we simulate P, we can ensure that we get correct answers on every simulation of P with high probability if P correctly computes CONFIG at length m. This implies property (i) from above.

Consider the space usage of the recovery procedure. A critical component is to be able to cycle over all configurations and determine whether two configurations are "adjacent". As the instances of CONFIG we are interested in correspond to a machine which uses $s(n)$ space, these two tasks can be accomplished in $O(s(m))$ space. The remaining tasks of the recovery procedure take $O(s(m))$ space for similar reasons as given for the recovery procedure for the computation tableau language in the previous section.

6. Analysis

In this section we explain how we come to the parameters given in the statements of Theorems 1.1 and 1.2. First, consider the space usage of the construction. The recovery procedures use $O(s(m))$ space when dealing with inputs of size m, and the additional tasks of the diagonalizing machine N also take $O(s(m))$ space. For input lengths n where N is responsible for copying down the delayed computation of the hard language L, N executes the recovery procedure using M_i on padded inputs of one larger length. Thus for such input lengths, the space usage of N is $O(s(n + 1))$. For input length n_i, N produces an instance y of the hard language corresponding to complementary behavior of M_i on inputs of length n_i and then simulates $N(y)$. For two-sided error machines, we reduce to the computation tableau language COMP. When M_i is allowed $s(n)$ space, the resulting instance of COMP is of size $n+O(s(n))$. For one- and zero-sided error machines, we reduce to configuration reachability, and the resulting instance is also of size $n+O(s(n))$. In both cases, the space usage of N on inputs of length n_i is $O(s(n_i + O(s(n_i))))$. We have chosen COMP and CONFIG as hard languages over other natural candidates (such as the circuit value problem for Theorem 1.1 and st-connectivity for Theorem 1.2) because COMP and CONFIG minimize the blowup in input size incurred by using the reductions.

The constant hidden in the big-O notation depends on things such as the alphabet size of M_i. If $s'(n) = \omega(s(n + as(n)))$ for all constants a, N operating in space $s'(n)$ has enough space to diagonalize against each M_i for large enough n. To ensure the asymptotic behavior has taken effect, we have N perform the construction against each machine M_i infinitely often. We set N's advice bit to zero on the entire interval of input lengths if N does not yet have sufficient space. Note that this use of advice obviates the need for $s'(n)$ to be space constructible.

Finally consider the amount of advice that the smaller space machines can be given. As long as the advice is at most $s(n)$, the recovery procedure can efficiently cycle through all candidate machines (M_i with each possible advice string). Also, to complement M_i for each advice string at length n_i, we need at least one input for each advice string of length n_i. Thus, the amount of advice that can be allowed is $\min(s(n), n)$.

Acknowledgments

We thank Scott Diehl for many useful discussions, in particular pertaining to the proof of Theorem 1.2. We also thank the anonymous reviewers for their time and suggestions.

References

[1] B. Barak. A probabilistic-time hierarchy theorem for slightly non-uniform algorithms. In *Workshop on Randomization and Approximation Techniques in Computer Science*, 2002.

[2] G. Buntrock, B. Jenner, K.-J. Lange, and P. Rossmanith. Unambiguity and fewness for logarithmic space. In *Fundamentals of Computation Theory*, pages 168–179, 1991.

[3] A. Condon. The complexity of space bounded interactive proof systems. In S. Homer, U. Schöning, and K. Ambos-Spies, editors, *Complexity Theory: Current Research*, pages 147–190. Cambridge University Press, 1993.

[4] S. Cook. A hierarchy theorem for nondeterministic time complexity. *Journal of Computer and System Sciences*, 7:343–353, 1973.

[5] L. Fortnow and R. Santhanam. Hierarchy theorems for probabilistic polynomial time. In *IEEE Symposium on Foundations of Computer Science*, pages 316–324, 2004.

[6] L. Fortnow, R. Santhanam, and L. Trevisan. Hierarchies for semantic classes. In *ACM Symposium on the Theory of Computing*, pages 348–355, 2005.

[7] O. Goldreich, M. Sudan, and L. Trevisan. From logarithmic advice to singe-bit advice. Technical Report TR-04-093, Electronic Colloquium on Computational Complexity, 2004.

[8] N. Immerman. Nondeterministic space is closed under complementation. *SIAM Journal of Computing*, 17(5):935–938, 1988.

[9] R. Karp and R. Lipton. Turing machines that take advice. *L'Enseignement Mathématique*, 28(2):191–209, 1982.

[10] M. Karpinski and R. Verbeek. Randomness, provability, and the separation of Monte Carlo time and space. In *Computation Theory and Logic*, pages 189–207, 1987.

[11] D. van Melkebeek and K. Pervyshev. A generic time hierarchy for semantic models with one bit of advice. *Computational Complexity*, 16:139–179, 2007.

[12] M. Saks and S. Zhou. $BP_HSPACE(S) \subseteq DSPACE(S^{3/2})$. *Journal of Computer and System Sciences*, 58:376–403, 1999.

[13] J. Seiferas, M. Fischer, and A. Meyer. Separating nondeterministic time complexity classes. *Journal of the ACM*, 25:146–167, 1978.

[14] R. Szelepcsényi. The method of forced enumeration for nondeterministic automata. *Acta Informatica*, 26(3):279–284, 1988.

[15] J. Watrous. On the complexity of simulating space-bounded quantum computations. *Computational Complexity*, 12:48–84, 2003.

[16] S. Žàk. A Turing machine time hierarchy. *Theoretical Computer Science*, 26:327–333, 1983.