MIT Reversible Computing Project

Memo #M6

Relativized separation of reversible and irreversible space-time complexity classes

WORKING DRAFT MANUSCRIPT

Revision: 1.35

Michael P. Frank
http://www.ai.mit.edu/~mpf

M. Josephine Ammer
josie@ai.mit.edu

MIT AI Lab
545 Technology Sq.
Cambridge, MA 02139

Started February 28, 1997.
Revision Date: 1997/04/16 17:28:15 GMT
Typeset July 10, 1997.

A current version is available online at
http://www.ai.mit.edu/~mpf/rc/memos/M06_oracle.html

Abstract

We are interested in finding models of computation that accurately express the potential computational power afforded by the laws of physics. The accepted laws of both classical and quantum mechanics are reversible, and this leads naturally to an interest in the efficiency of reversible models of computation. We conjecture that all proposed irreversible models of computation would be asymptotically less powerful (spacetime-efficient) if constrained to be reversible.

We would like to prove our conjecture at least for the special case of normal serial models of computation. As a prelude to this, in this paper we use an incompressibility argument to prove that there exists an oracle relative to which irreversible serial machines are indeed strictly more powerful than reversible ones, when both types of machines can query the oracle as a black-box operation.

This result implies that any disproof of our conjecture for normal serial models would have to make use of the fact that none of the operations available to normal machines are oracle queries. Also, our proof illustrates some techniques that might help to prove our conjecture in the non-oracle case, and perhaps eventually also for other models such as 3-D parallel meshes, which might more accurately express the raw computational capabilities of physics.

1 Introduction

We believe that an important goal of research in the physics of computation is to describe and study models of computation that reflect as closely as possible the ultimate information-processing capabilities offered by the laws of physics as they are currently understood. It is expected that experience with such computing models will facilitate the eventual design of computers that approach the maximum speeds physically possible. See [19] for one kind of speed limit.

The laws of both classical and quantum mechanics are reversible; the complete description of the microscopic state of a system uniquely determines what state the system was in at some earlier time. In contrast, most models of computation that computer scientists have studied are irreversible.

In a separate paper [7], we conjecture that any physically realizable computing architecture, so long as it does not utilize quantum superposition effects, can be simulated with at most constant factor slowdown by a reversible 3-dimensional parallel mesh architecture (like that described in [12], §1.9, pp. 222–236), which is equivalent to a reversible 3-D cellular automaton. This model consists of a grid of finite processors that communicate locally with their immediate neighbors. See [18] for a discussion of the ability of reversible cellular automata to model physics.

Moreover, we expect that this model can be efficiently implemented physically on arbitrarily large scales, as technology improves, at least until gravity or a lack of raw material prohibits further scaling. If these conjectures are correct, then the reversible 3-D mesh is an appropriate model of the computing capabilities of physics (at least when quantum effects aren't exploited).

Under that assumption, a natural question is whether the reversibility constraint is necessary in order for the 3-D mesh model to be accurate. We conjecture that it is; that an irreversible 3-D mesh would be strictly more powerful than a reversible one, and therefore that the irreversible 3-D mesh is not physically implementable in a scalable way and therefore is not ultimately an appropriate model for physical computation.

To see why we would not expect an irreversible 3-D mesh model to be physically implementable, recall that any physical system is actually reversible at the level of its microscopic dynamics. Therefore, any information that is nominally irreversibly discarded will remain present, manifesting itself in the form of heat. Within a sufficiently large 3-D mesh that is performing irreversible computations, the rate of internal heat (garbage information) production will exceed the maximum flow capacity through the 2-D surface of the mesh. The mesh's internal temperature will increase to the point where correct functionality is impossible. So in general, a run-time analysis of an irreversible 3-D mesh algorithm would not necessarily apply in the real world.

In contrast, in any algorithm for a reversible 3-D mesh, the entire flow of information is accounted for explicitly, so there is no danger of forgetting to account some discarded information that will eventually gum up the works. If the algorithm works in the reversible model, then it will work in the real world. Of course, there may be engineering difficulties that make scaling difficult, but one can anticipate technological fixes for these. But there is no possible technological fix for the fact that in the real world, information always must go somewhere.

We would like to prove that irreversible 3-D meshes are more powerful than reversible ones. In fact, we would like to prove that for any irreversible model of computation that has been proposed so far, a reversible version of the model would be less powerful.

But since we do not yet have much experience with algorithms for and analysis of 3-D parallel mesh models of computation, we start by considering the more traditional serial computational models, such as Turing machines and RAM machines.

Previous work by Bennett [2] and Lange/McKenzie/Tapp [10] has shown that reversible machines can always imitate the time-efficiency or space-efficiency of traditional computing models, when the two kinds of efficiency are studied separately. However, when space and time requirements are taken into account *simultaneously*, we conjecture that reversibility necessarily incurs inefficiencies.

In this paper, for all space and time bounds in a wide range, we describe an oracle and a language that can be recognized within a constant factor of the bounds by oracle-querying machines that are irreversible, but not by ones that are reversible. We assume a serial machine model, but suspect that a similar proof would go through for parallel models as well.

This result implies that an attempt to disprove the conjecture that reversible machines require more than constant factor overhead would have to take into account the limited set of operations that real machines may actually perform, since otherwise, adding an extra operation (namely the call to our oracle) would falsify the disproof.

For example, this result makes it unlikely that there could be any general constant-overhead simulation of irreversible machines by reversible ones, since there seems to be no reason why the general simulation would not apply to the oracle machines as well. (However, this argument is not rigorous.)

Another contribution of our proof is to illustrate some ways to apply incompressibility methods in the analysis of reversible machines. It seems likely that similar techniques might be used to show that a related problem *without* an oracle, such as searching through an incompressible graph, might allow a provable separation of reversible and irreversible space-time complexity classes.

2 Related Work

Brief history of reversible computing theory. Rolf Landauer '61 [9] was probably the first to point out that irreversible computer operations cause unavoidable energy dissipation due to the laws of thermodynamics, a fact that has been the primary motivation for the study of reversible models. Reportedly, a paper by Lecerf '63 [11] first established that reversible Turing machines are universal. (Lecerf's paper was not cited in English-language reversible computing literature until recently.) Bennett '73 [2] rediscovered the universality result by describing a linear-time (but space-inefficient) simulation of irreversible Turing machines by reversible ones. Coppersmith and Grossman '75 [5] proved a result in group theory which implied that reversible circuits only 1 bit wider than their input can still compute arbitary boolean functions. Toffoli '77 [21] showed that reversible cellular automata can simulate irreversible ones in linear time by using an extra spatial dimension. Continuing through the early 80's, Toffoli and Fredkin developed much reversible circuit theory [22, 8]. Pin '87 [20] showed that reversible finite automata cannot decide all regular languages. Bennett '89 [3] introduced a new, more space-efficient reversible simulation technique for Turing machines. Crescenzi and Papadimitriou '95 [6] extended this to nondeterministic machines. Li and Vitányi [17, 16, 14] characterized the benefits of partial reversibility. Most recently, Lange, McKenzie, and Tapp [10] showed that linear-space reversible simulations of Turing machines are possible.

Comparisons. Although the Bennett and Lange *et al.* results [2, 10] showed that linear time or space simulations of irreversible Turing machines by reversible ones are indeed possible, to our knowledge the current paper is the first to address the question of whether a general reversible simulation can take both

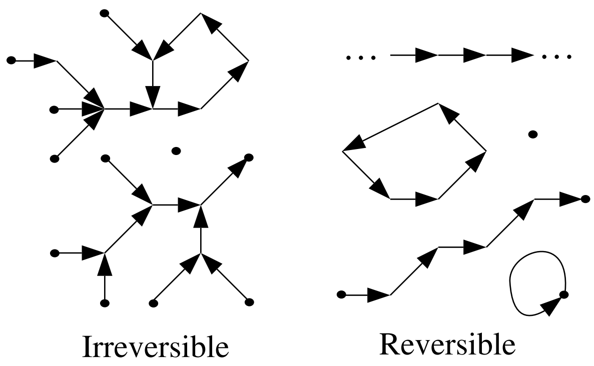

Figure 1: Machine configuration graphs in (deterministic) reversible and irreversible models of computation.

In the configuration graphs of irreversible machines, configurations may have many different predecessor configurations. In reversible models of computation, each configuration may have at most one predecessor. The configuration graph therefore consists of disjoint loops and chains, which may be infinite. In both reversible and irreversible models we may permit configurations having 0 predecessors (initial states) and/or 0 successors (final states).

---

linear time *and* linear space. Our proof begins to confirm our conjecture that the answer is no.

3 General Definitions

Reversible models of computing. As is by now standard in the theory of reversible computing, we define a *reversible* model of computation as one in which the infinite graph of machine states has indegree one. In this paper we will always assume that our machines are deterministic, so that the configuration graph always has outdegree one as well. Figure 1 illustrates the difference between configuration graphs in reversible and irreversible models.

Space-time complexity classes. Given any reversible model of computation (*e.g.*, reversible Turing machines), and given any space and time bound functions $S(n), T(n)$, we define the *reversible space-time S, T complexity class*, abbreviated $\mathbf{R}ST$, to be the set of languages that are accepted by some reversible machine that takes worst-case space $O(S(n))$ and time $O(T(n))$, where n is the length of the input. Similarly, we define the unrestricted *space-time S, T complexity class*, abbreviated ST, to be the set of languages accepted in that same order of space and time on the corresponding normal machine model, without the restriction on the indegree of the transition graph.

The question we would like to answer is whether $\mathbf{R}ST =_? ST$, for all S, T, in normal sorts of serial computational models such as multi-tape Turing machines or RAM machines. Unfortunately, we find this question very difficult to resolve. We do not see any general way to simulate normal machines on reversible machines without suffering asymptotic increases in either the time or space required. But neither do we know of a language that can be proved to require extra space or time to recognize reversibly. The difficulty is in constructing a proof that rules out all reversible algorithms, no matter how subtle or clever.

But is the $\mathbf{R}ST{=_?}ST$ question truly difficult to resolve, or have we just been unlucky in our search for a proof? Often in computational complexity theory, we find ourselves unable to prove whether or not two complexity classes (for example, $\mathbf{P}$ and $\mathbf{NP}$) are equivalent. Traditionally (as in [1]), one way to indicate that such an equivalence might be difficult to prove is to show that if the machine model defining each class is augmented with the ability to perform a new type of operation (a query to a so-called "oracle"), then the classes may become provably either equal or unequal, depending on the behavior of the particular oracle. This shows that any proof equating or separating the two classes must make use of the fact that normal machine models are only capable of performing a particular limited set of primitive operations. Otherwise, we could just add the appropriate oracle call as a new primitive operation, and invalidate the supposed proof. In complexity theory, it is said that the proof of the equivalence or inequivalence of the two classes must not "relativize," that is, it does not remain valid relative to models that are augmented with oracles. Apparently this rules out a large number of proof techniques from recursion theory, and means that resolving the question will be more difficult.

In this paper will will demonstrate an oracle with which $\mathbf{R}ST$ and ST are unequal. We have not yet found an oracle for which they are equal. It may be that none exists, but this is uncertain.

Oracle-relativized complexity classes. For a complexity class C, and oracle O, the standard notation in complexity theory is to write C^O to indicate the class of problems that can be solved by the class of machines that define C, if those machines are also given the ability to query the oracle O as a new primitive operation.

If we can prove there exists an oracle A such that $\mathbf{R}ST^A \neq ST^A$, then we will have shown that any argument that might exist to prove that $\mathbf{R}ST = ST$ must not relativize.

In the remainder of the paper, we show how to construct such an oracle, for the case of serial machine models with a certain kind of oracle interface. For other types of machine models, the proof may be similar.

Reversible oracle interface. First, we must define our oracle interface, and work out exactly what it means for a reversible machine to call an oracle. Normally oracle queries are defined in such that the act of querying the oracle makes an irreversible change to machine state; we must avoid this if we want our reversible machines to be able to call the oracle and still be reversible. For example, the oracle's answer cannot just be written into some location in the machine's memory, overwriting it, because that would not be a reversible operation. To speak more precisely, no matter whether the storage location originally contained `0` or `1` before the oracle call, after the oracle call it would contain the oracle's answer. Thus the configuration after the oracle call would have two different possible predecessor configurations, violating our definition that the configuration graph of a reversible machine has indegree one.

One way to avoid this sort of problem is to restrict ourselves to using "permutation oracles." A permutation oracle takes as input the state of some part of the machine and changes it to a new state according to a mapping function that is a permutation over all the possible states. (See figure 2.) Thus, the oracle operation is reversible, by the definition of a permutation, and it will be callable from both reversible and irreversible machines with equal ease.

Further, if the permutation function is its own inverse, the oracle will be called *self-reversible*. In this paper will we deal only with self-reversible oracles. Self-reversibility ensures that it is easy for a machine to undo an oracle operation, just as with the normal primitive operations of a typical reversible machine architecture, such as reversible Turing machines or the Pendulum RISC instruction set being developed in our project [23]. Thus, by using a self-reversible oracle, we maximize the chance that our result might carry over to normal reversible machine models, in which all primitive operations can be trivially undone.

To make things more concrete, we will define our oracle-calling protocol as follows. Machines will have read/write access to a special "oracle tape" which has a definite start, unbounded length, and is initially clear. At any time the machine is allowed to perform an "oracle call," a special primitive operation which in a single step replaces the entire contents of the oracle tape with some new contents, according to some fixed invertible mapping between possible tape contents. The mapping is defined once and for all time by the choice of oracle.

$\mathbf{R}ST$-constructibility. As another basic definition, in order for our proof to go through, we will need to restrict our attention to space and time functions $S(n), T(n)$ which are *$\mathbf{R}ST$-constructible*, which we define here as meaning that given any input of length n, a reversible machine in our model can construct binary representations of $S(n)$ and $T(n)$ using only space $O(S(n))$ and time $O(T(n))$. We state here without proof that most familiar arithmetic functions that people might care about are indeed $\mathbf{R}ST$-constructible.

Finally, we will need some basic definitions to support the notion of incompressibility that will be crucial to the proof of our theorem. This definition

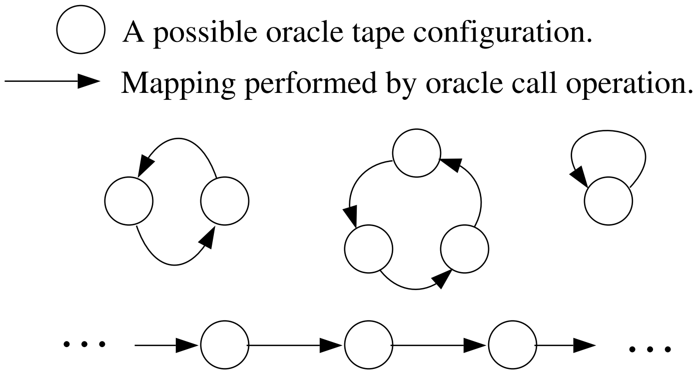

Figure 2: A permutation oracle.

The oracle call operation replaces the old contents of the oracle tape with new contents according to a transition function $A : \mathcal{C} \to \mathcal{C}$ that is a permutation mapping—a bijective function—over the space $\mathcal{C}$ of possible tape contents. The bijectivity of this function means that a call to a permutation oracle is always a reversible operation. After an oracle call, the previous oracle tape contents can be uniquely determined by applying the inverse mapping A^{-1}. In self-reversible oracles, $A = A^{-1}$.

---

was inspired by the discussions of incompressibility in Li and Vitányi's excellent book on Kolmogorov complexity [15].

Description systems and compressibility. A *description system* s is a function $s : \{0,1\}^* \to \{0,1\}^*$ from bit-strings to bit-strings, that is, from *descriptions* to the bit-strings that they describe. We say that a bit-string d describes bit-string x in description system s if $s(d) = x$. We say that x is *compressible* in description system s if there is a shorter string that describes it; *i.e.* if there exists a string d such that $s(d) = x$ and $|d| < |x|$, where the notation $|b|$ denotes the number of bits in bit-string b.

Lemma 1. *Existence of incompressible strings.* For any description system s, and any string length ℓ, there is at least one bit-string x of length ℓ that is not compressible in s.

Proof. (Trivial counting argument.) There are 2^ℓ bit-strings of length ℓ, but there are only $\sum_{i=0}^{\ell-1} 2^i = 2^\ell - 1$ descriptions that are shorter than ℓ bits long. Each description d can describe at most one bit string of length ℓ, namely the string $s(d)$ if that string's length happens to be ℓ. Therefore there must be at least one remaining bit-string of length ℓ that is not described by any shorter description. $\square$

In our main proof we will be constructing our oracle using incompressible strings that we draw from a series of computable description systems.

Concise notation for space-time functions. To be concise, throughout the rest of this paper, we will sometimes abbreviate the space and time function values $S(n)$ and $T(n)$ by just S and T, respectively; similarly for other functions of n introduced later. All these quantities will be understood to be dependent on the value of n (input length) even when this dependence is not made explicit.

Notation for asymptotic comparisons between functions. For convenience we adapt standard order-of-growth notation as follows. If f and g are functions of n, we write $f <_o g$ to mean $f = o(g)$, *i.e.* $f(n)$ asymptotically grows strictly slower than $g(n)$. Similarly define $\leq_O$, $\geq_\Omega$, $>_\omega$, and $=_\Theta$.

4 Main Theorem

Theorem 1. *Relative separability of reversible and irreversible spacetime.* For any $\mathbf{R}ST$-constructible functions S, T such that $S <_o T <_o 2^S$, there exists a computable, self-reversible oracle A such that $\mathbf{R}ST^A \neq ST^A$.

The intuitive reason for the restriction that $S <_o T$ is that for $S \not<_o T$, a reversible machine could simulate any irreversible one without increasing the space or time complexity, by using Bennett's [2] simple expedient of keeping a complete history of all irreversibly erased data. (Doing this efficiently on Turing machines requires an extra tape or stack on which to store the history, but if our model allows multi-tape Turing machines, that is acceptable.) This simulation applies equally well to models with permutation oracles. Thus, whenever $S \not<_o T$, we can show that $\mathbf{R}ST^A = ST^A$ for any permutation oracle A, so it would be impossible for our theorem to cover that case. Similarly, if $T \not<_o 2^S$, then the simulation technique of Lange *et al.* [10] can be used (with self-reversible oracles) without decreasing efficiency since its worst-case time for simulations that halt is $O(2^S)$, and so in that case also, the reversible and irreversible space-time classes are equal, and our theorem would not apply.

The reason for the $\mathbf{R}ST$-constructibility requirement will become apparent later.

Outline of proof. We will construct A to be a permutation oracle that can be interpreted as defining an infinite directed graph of nodes with outdegree at most 1. We will also define a corresponding language-recognition problem, which will be to report the contents of a node that lies T/S nodes down an incompressible linear chain of nodes that have size-S identifiers, starting from a node that is determined by the input length. The oracle will be explicitly constructed via a diagonalization, so that for each possible reversible machine, there will be a particular input length for which our oracle makes that particular reversible machine take too much space or else get the wrong answer. In the cases where the reversible machine takes too much space, we prove this by equating the machine's operation with the "pebble game" for which Li and Vitányi [17] have already proven lower bounds, and by proving that if the machine does not take too much space, then we can build a shorter description of the chain of nodes using the machine's small intermediate configurations, thus contradicting our choice of an incompressible chain.

4.1 Special definitions.

First we will define the general class of permutation oracles we will be using. We will call them "graph oracles" since we will be interpreting them as representing graphs, in particular, they will represent directed graphs of maximum outdegree 1. Since our graph oracles will be permutation oracles, we will define them by defining the permutation function that maps each possible oracle tape contents to the new contents that would result from a query operation.

In the interpretation of these oracles as graphs, nodes of the graph are represented by and identified with bit-strings. Queries on the oracle tape will consist of nodes (which are identified with bit-strings), or of pairs of two nodes. The permutation performed by the graph oracle will be to map a node to the pair of the node and its unique successor, and to map that pair back to the the first node in the pair. All other possible tape contents will be mapped to themselves (an oracle call will have no effect). In this way, an oracle defines a graph: for any node q, its successor node q' is defined as the node that appears as the second node in the pair of nodes resulting from giving the oracle the node q as a query. More formally:

Definition 1. *Graph oracles.* A graph oracle is a permutation oracle with the following property: there exists a partial function f called a "graph function" which maps bit strings to bit strings, $f : \{\mathtt{0},\mathtt{1}\}^* \to \{\mathtt{0},\mathtt{1}\}^*$, such that for any bit string w for which f is defined, the oracle's permutation function maps the tape contents w to the tape contents $w\#f(w)$, and also maps $w\#f(w)$ back to w, where # is a separator character in the oracle tape alphabet. For all tape contents not of either of these forms, the oracle's permutation function maps them to themselves. See figure 3.

The graph function f is interpreted as mapping nodes to their unique successor nodes. f is undefined for nodes with no successors.

Given that we will be working only with graph oracles, we can now specify an oracle by specifying just the graph function f that it embodies.

Next, before we actually construct the special oracle A that proves our theorem, let us define, relative to A, the language that we claim will separate $\mathbf{R}ST^A$ from ST^A.

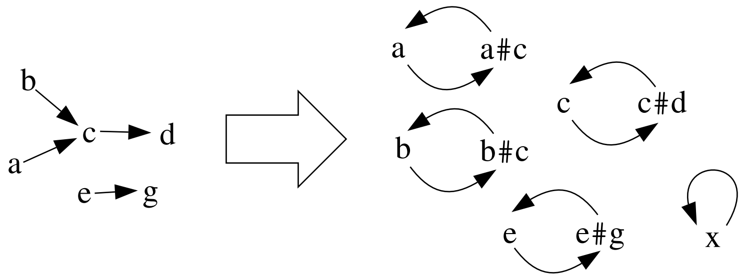

Figure 3: Encoding outdegree-1 directed graphs in permutation oracles.

On the left we show an example of an outdegree-1 directed graph with nodes named a,b,c,d,e,g. The graph function f gives the successor of each node: $f(\mathrm{a}) = \mathrm{c}$, $f(\mathrm{c}) = \mathrm{d}$, etc. f is partial; *e.g.* $f(\mathrm{d})$ is undefined. For each edge in this graph, there is a corresponding pair of strings that are mapped to each other by the permutation oracle. To represent the edge $a \rightarrow c$, for example, the permutation oracle maps tape contents "a" to "a#c" and maps "a#c" back to "a". Any other string x is simply mapped to itself. In this way the permutation oracle allows easily and reversibly looking up a node's successor. But finding a node's predecessor(s) is designed to be hard. Thus the oracle call resembles a reversible computation of an invertible function that is easy to compute but whose inverse is difficult to compute.

Definition 2. *The difficult language.* Given $\mathbf{R}ST$-constructible functions $S(n), T(n)$, and graph oracle A with graph function f, we define the *difficult language* $L(A)$ to be the language decided by the irreversible machine described by the following pseudocode:

Given input string w,
  let $n = |w|$
  let bit-string $b = \mathtt{0}^{S(n)}$
  repeat $\lfloor T(n)/S(n) \rfloor$ times until $f(b)$ is undefined:
    assign $b \leftarrow f(b)$ (found by querying A)
  accept iff $b[0] = \mathtt{1}$. □

In other words, given a string of length n, construct a string of zeros of length $S(n)$, treat it as a node identifier, and use oracle queries to proceed down its chain of successors for up to $\lfloor T/S \rfloor$ nodes, and return the first bit of the final node's bit-string identifier.

We will be explicitly constructing the oracle's graph function f so that it always returns a string of the same length as its input. Given such an oracle, the above algorithm can obviously be implemented so as to take worst-case space $O(S(n))$ and time $O(T(n))$ on on irreversible machine in any standard serial model of computation. Therefore the language $L(A)$ will be in the class ST^A.

Now we need to specify how to construct the graph function f defining our oracle A so that the language $L(A)$ will not be computable by any reversible machine in space $O(S(n))$ and time $O(T(n))$. The way we will do this is to make each of the node identifiers be an incompressible string. Intuitively speaking, it seems the only way to decide $L(A)$ accurately is to follow the entire chain of nodes, to see what the final one is. But having obtained a node's successor, the reversible machine will generally be unable to efficiently erase its incompressible records of the prior nodes, since the oracle will provide no convenient way to compute f^{-1} and find a node's predecessor—even if the graph function f happens to be invertible. Thus the reversible machine will tend to accumulate records of previous nodes, of size $S(n)$ each, and thus, for sufficiently long enough chains, it will take more than a constant factor times $S(n)$ space. The reversible machine could conceivably find and erase predecessor nodes by searching them all exhaustively, but this would take too much time.

The situation with this oracle language resembles the non-oracle problem of repeatedly applying a one-way function, *i.e.* an invertible function whose inverse is harder to compute than the function itself. Public-key cryptography depends on the assumption that some functions are one-way, but this has not yet been proven. If it was proven, this might allow us to show that $\mathbf{R}ST \neq ST$ by using a one-way function in place of the oracle described here.

In any case, although our intuition tells us that it's easy for the oracle A to be such that $L(A)$ cannot be decided in $\mathbf{R}ST^A$, intuition is not a proof. We now construct a particular A and prove that $L(A) \notin \mathbf{R}ST^A$.

4.2 Proof of reversible difficulty.

First, fix some standard enumeration of all reversible space-$O(S(n))$-bounded, time-$O(T(n))$-bounded oracle-querying machines: $M_1, M_2, \ldots$. Let $c_1, c_2, \ldots$ be a parallel enumeration of constants such that M_i is restricted to always take space less than $c_i + c_i S(n)$ and time less than $c_i + c_i T(n)$. (Such constants must exist if M_i takes space $O(S(n))$ and time $O(T(n))$.)

This enumeration is possible since one can easily define a universal reversible programming language, and enumerate programs in that language. Also Lange *et al.* [10] have shown how to characterize reversible Turing machines by local restrictions on their transition function, so that in an enumeration of Turing machines one can effectively identify reversible ones. Bounding the space and time that a machine takes is easy, since $S(n)$ and $T(n)$ are by assumption $\mathbf{R}ST$-constructible. The positive integer constant factors c_i in the space and time bounds can easily be enumerated in dovetail fashion along with the machines, so that if some reversible machine solves the problem class in time $O(T(n))$ and space $O(S(n))$, we will eventually list it together with a large enough constant c_i in its space/time bound to allow the machine to always work.

Now define A to be the permutation oracle which we define recursively as follows.

When considering machine M_i, we will have already recursively defined $f(q)$ for all oracle queries q asked by machines $M_1, M_2 \ldots, M_{i-1}$ when given certain inputs of lengths $n_1, n_2, \ldots, n_{i-1}$, respectively. Now, we choose n_i (henceforth referred to as just n), the input length for which our oracle definition will foil M_i, to be such that $S(n)$ is greater than the length of any of those earlier machines' oracle queries, and also to be large enough that $T(n) > c_i + c_i S(n)$ (we know that such an n exists because the statement of the theorem assumes that $S(n) <_o T(n)$). We will mention some other lower bounds on the size of n as we go along; it would be too confusing to introduce them all now.

Later we will define a description system s_i based on M_i, c_i, the value of n, and all the $f(q)$ values defined so far (for bit-strings smaller than $S(n)$). The description system will be a total computable function, *i.e.*, there is an algorithm that computes $s_i(d)$ for all d and always halts. We will use this description system to define $f(q)$ for bit-strings of length $S(n)$, as follows:

Let x be a bit-string of length $T(n)$ that is incompressible in description system s_i. We know at least one such string exists because of the standard counting argument we mentioned earlier. We can find it effectively by enumerating all descriptions d shorter than $T(n)$, computing the described strings $s_i(d)$, remembering those of length $T(n)$, and setting x to (say) the lexicographically first length-$T(n)$ bit-string that is not equal to any of those $s_i(d)$.

This x will be used as the sequence of size-$S(n)$ node identifiers that will define our graph for inputs of size n; its incompressibility in system s_i will allow us to ensure that machine M_i cannot take any "shortcuts" to get to the answer.

Now break x up into a sequence of $t(n) \equiv \lfloor T(n)/S(n) \rfloor$ bit-strings of length

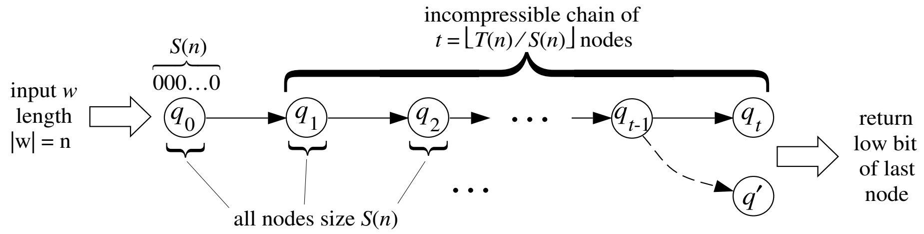

Figure 4: The problem graph defined by our oracle for inputs of size n consists of a linear chain of nodes $q_0 \ldots q_t$.

The correct answer is just the first bit of the final node q_t. If the reversible machine M_i that we are trying to foil happens to get the right answer, but never asks for the successor of node q_{t-1}, we redefine q_{t-1}'s successor to be a new node q' having a different initial bit.

$S(n)$ each; call these our nodes or "query strings" $q_1, \ldots, q_t$. We will design our description system s_i so that all the q_j's must be different. How? By allowing descriptions of the form (j, k, x'), where j and k are the indices of two equal nodes $q_j = q_k$, and x' is x with the q_k substring spliced out. The description system would be defined to generate x from such a description by simply looking up the string q_j in x' and inserting a copy of it in the kth position. The indices i and k would take $O(\log(T(n)/S(n)))$ space, which is $O(\log T(n))$ space, which is $o(S(n))$ space, whereas we are saving $S(n)$ space by not explicitly including the repetition of q_j. Therefore as long as n is sufficiently large, the total length of this description of x will be less than $T(n)$, violating our assumption that x was incompressible. Thus, with x having been generated from a description system that permits such descriptions, we know that $q_1, \ldots, q_t$ includes no repetitions.

Now we can specify exactly how the oracle defines our problem graph for inputs of size n, as follows. Define query string $q_0 = \mathtt{0}^S(n)$. Let us provisionally set $f(q_{j-1}) = q_j$ for all $1 \leq j \leq t$. These assignments are possible since all the q_j's are different, as we just proved. (They also must be different from q_0, but this is easy to add.) With these assignments, if $q_t[0] = \mathtt{1}$, then all strings of length n are in the language $L(A)$, otherwise all strings of length n are not in $L(A)$, according to our earlier definition of the language.

Now, let us suppose temporarily that our oracle definition was completed by letting $f(w)$ remain undefined for all strings w for which we have not yet defined $f(w)$ (this excludes w's for which $f(w)$ was already defined in the course of handling earlier M_is). Let us then, under that assumption, simulate M_i's behavior on the input $\mathtt{0}^n$. It will ask some sequence of oracle queries, and then (because it is time-bounded) will eventually either accept (1) or reject (0). If its answer is different from $q_t[0]$, then M_i already fails to accept the language $L(A)$ and we can go ahead and permanently define the oracle responses for $q_0, \ldots, q_t$

as above, leave $f(b)$ undefined for all other length-$S(n)$ bit-strings b, and go on to address the next machine M_{i+1}.

Alternatively, perhaps M_i's answer is correct with the given q_j's. But suppose that M_i never asked for $f(q_{t-1})$ in any query during its run on input 0^n. That is, suppose M_i never asked either query q_{t-1} or query $q_{t-1}\#q_t$. (We can discover this by straightforward simulation.) In that case, let us change our definition of $f(q_{t-1})$ as follows, to change the correct answer to be the opposite of what M_i gave. Let q' be a bit-string other than $q_0, \ldots, q_t$ that was never asked in any query by M_i, and whose first bit is the opposite of M_i's answer. We know plenty of such strings exist to choose from since there are $\frac{1}{2}2^S$ bit-strings of length $S(n)$ having the desired initial bit, but M_i can make at most T queries since that is its running time, and there are only $t+1$ queries in $q_0, \ldots, q_t$, and $t <_o T <_o 2^S$. (n should have been chosen large enough that $\frac{1}{2}2^S(n)$ was larger than the time bound $c_i + c_i T(n)$ plus $t+1$.) Given this new q', we let $f(q')$ stay undefined, and change $f(q_{t-1})$ to be q'. None of this can possibly affect the behavior of M_i since it never asked about $f(q_{t-1})$. Thus with the new partial definition of f, M_i fails to accept the language, and we can go on to define f for larger n to foil M_{i+1} and so forth.

We will next show how to complete our definition of our description system s, which we used to find an incompressible $x = q_1 q_2 \ldots q_t$, so that it is impossible by construction for M_i to ask query q_{t-1}, and thus the above technique for foiling M_i will always apply.

To do this, we would like to show how M_i can be interpreted as following the rules of Bennett's reversible "pebble game," introduced in [4] and analyzed by Li and Vitányi in [17]. We briefly describe the game here. See fig. 5 for an illustration.

In the pebble game there is a linear list of nodes, which we will identify with our query strings $q_1, \ldots, q_t$. At any time during the game some set of nodes is "pebbled." (We visualize that each such node has a pebble resting on it.) The idea of the pebbled set is that it will correspond to the set of nodes that is currently "stored in memory." However, not knowing how M_i might choose to represent nodes in memory, we will need another definition of what it means for a node to be pebbled. We will define pebbling in terms of M_i's execution history, and then prove that the number of pebbled nodes at a given time under this definition will indeed set a lower bound on the space usage at that time.

But first, let us finish describing the rules of the pebble game. Initially, no nodes are pebbled. At any time, the *player* (in our case, M_i) may, as a move in the game, change the pebbled vs. unpebbled status of node q_1, or of any node q_j for which the previous node q_{j-1} is pebbled at that time. Only one such move may be made at a time.

In our context, pebbling node q_j will usually correspond to querying the oracle with query string q_{j-1}, and unpebbling q_j will usually correspond to querying the oracle with query string $q_{j-1}\#q_j$. We say "usually" because some such queries of this form may not actually correspond to changing a node's

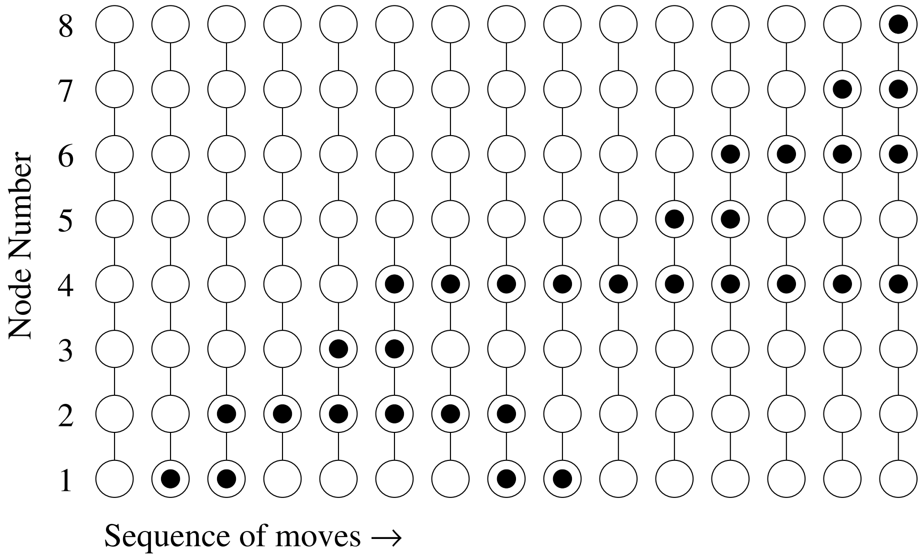

Figure 5: Bennett's reversible pebble game.

A node can be pebbled or unpebbled only if it is node 1 or if the previous node is pebbled. The strategy invented by Bennett [4], illustrated here, was shown by Li and Vitányi to be optimal [17] in terms of the number of pebbles required. But even with this optimal strategy, to pebble node 2^k we must at some time have more than k nodes pebbled. In this example, we reach node $2^3 = 8$ but must use 4 pebbles to do so. (After pebbling node 8, we can remove all pebbles by undoing the sequence of moves.) The fact that a constant-size supply of pebbles can only reach out a constant distance is crucial to our proof.

pebbled status, for example if during the query, q_j is being stored in some form elsewhere in the machine besides on the oracle query tape. Our mapping of pebble game concepts onto M_i will need to cover such cases.

The goal of the pebble game is to eventually place a pebble on the final node q_t. This corresponds to the fact already established that M_i must at some point ask query q_{t-1} or else we can easily construct our oracle so as to foil M_i.

Furthermore, in the usual characterization of the pebble game, the player must, after pebbling node q_t, subsequently remove all pebbles. In our case this would correspond to requiring machines to halt in a final state consisting of just the input and the output bit (accept or reject). But this is not really a restriction, because it is clear that if a reversible machine exists that can get the right answer in space-time order S and T, then a reversible machine exists that does this and also halts with its memory in a "clean" final state. This is done by simply copying the output bit and then running the original computation in reverse until everything but the copied output bit is back to its original state, as Bennett described in [2]. This reverse phase of the computation does not increase the order of S or T, but rather only doubles the size of the constant factor c_i that is required in the time bound. Therefore, we can simply assume, without loss of generality, that M_i does always halt in a clean state with all pebbles removed.

Li and Vitányi's analysis of the pebble game [17] showed that no strategy can win the game for 2^k nodes or more without at some time having more than k nodes pebbled at once. If we can indeed show that our machine M_i and its space usage can be modeled using the pebble game, then this will mean that since the number of nodes $t(n) >_o 1$ grows without bound as n increases, then for some sufficiently large n, the number of nodes that must be pebbled at once will exceed M_i's maximum node storage capacity due to whatever constant factor c_i is involved in its $O(S(n))$ space bound.

Now let us proceed to define exactly how to map pebble game concepts onto our situation of a reversible machine M_i that is trying to determine whether its input $\mathtt{0}^n$ is a member of the language $L(A)$.

For the oracle A as we have defined it so far (with $f(w)$ undefined for all heretofore undefined $f(w)$) let us consider the complete sequence of configurations of M_i on input $\mathtt{0}^n$, notated $C_1, C_2, \ldots, C_{T'}$, where $T' \leq c_i + c_i T(n)$ is M_i's total running time, in terms of the number of primitive operations performed.

Now, for any time point τ, $1 \leq \tau \leq T'$, and for any node q_j in the chain of nodes $q_1, \ldots, q_t$, let us define "the previous query involving q_j" to mean the most recent oracle query in M_i's history before time τ in which the query string is either q_{j-1}, q_j, $q_j\#q_{j+1}$, or $q_{j-1}\#q_j$. There may of course be no such query. Similarly define "the next query involving q_j" to mean the most imminent such query in M_i's future after time τ.

Now, we will say "q_j is pebbled at time τ" iff either (a) the previous query involving q_j exists and is either (case a1) q_{j-1}, (a2) q_j, or (a3) $q_j\#q_{j+1}$, or (b) the next query involving q_j exists and is (b1) q_j, (b2) $q_j\#q_{j+1}$, or (b3)

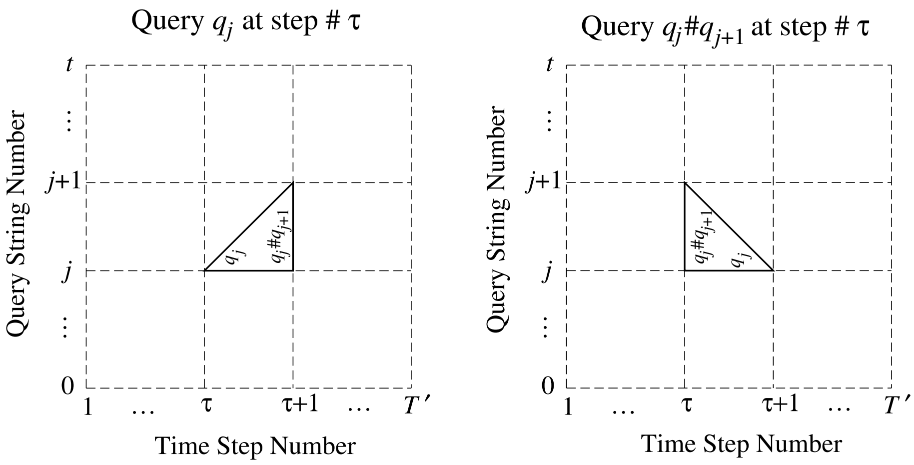

Figure 6: Triangle representation of oracle queries.

The shape and direction of the triangle reflects the fact that at the times just before and after an oracle query, the oracle tape contains the shorter string q_j at one of the times, and the longer string $q_j\#q_{j+1}$ at the other time. The triangles determine the set of pebbled nodes, as illustrated in figure 7.

$q_{j-1}\#q_j$. Exception: the final node q_t is only considered pebbled in cases (a1) and (b3). Figure 7 illustrates the intuition behind this definition using the graphical notation introduced in figure 6.

Let p be the number of distinct nodes out of $q_1, \ldots, q_t$ that are pebbled at time τ. We now show that the size of C_τ (*i.e.*, M_i's space usage at time τ) must be greater than $pS(n)/2$.

Lemma 2. *Space required to pebble nodes.* $|C_\tau| > pS/2$.

Proof. Suppose C_τ were no larger than $pS/2$ bits. Then we can show that x (the sequence of all q_j's) is compressible to a shorter description d which we will now specify. Our description system $s = s_i$ will be defined to process descriptions of the required form.

First, let us note that for each node q_j that is pebbled at time τ, that node is pebbled either because of the previous query involving q_j, because of the next query involving q_j, or both. Therefore, either at least $p/2$ nodes are pebbled because of their previous query, or at least $p/2$ nodes are pebbled because of their next query (or both). Let D be a direction (forwards or backwards) from time τ in which one can find queries causing at least $p/2$ nodes to be pebbled.

Now, we specify the shorter description d that describes x. d will contain an explicit description of C_τ, which by our assumption is smaller than $pS/2$.

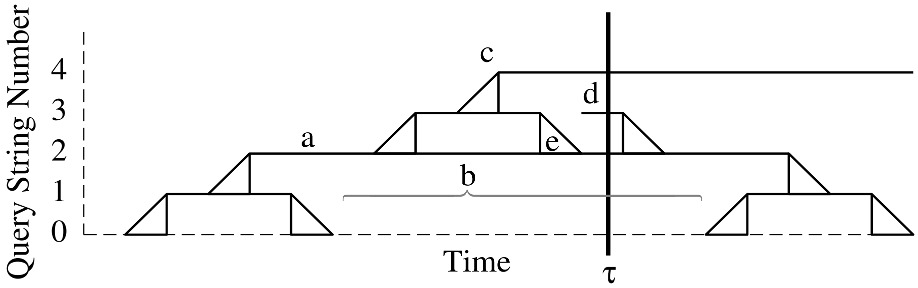

Figure 7: Visualizing the definition of the set of pebbled nodes.

The times at which a node is pebbled (indicated by solid horizontal lines on the chart) are determined, by definition, solely by the arrangement of triangles (see fig. 6) on the chart. Each vertex of a triangle generates a line of pebbled times for the corresponding node, extending horizontally away from the triangle until it hits another triangle. Except query string 0 is never pebbled, because it is not considered to be a node.

The above example shows a pattern of queries similar to the one that would occur if one tried to apply Bennett's [4] pebble game strategy. Node 2 is pebbled at time (a) both because of the previous and next queries (triangles) involving node 2. Node 1 is not pebbled at times (b) because both the previous and next queries are of the wrong form. Node 4 is pebbled at all times after (c) because even though there is no next query involving node 4, the previous query involving node 4 exists and is of the right form. Node 3 is pebbled at time (d) because although the previous query (e) is of the wrong form, the next query is okay.

Query (e) does not change the set of pebbled nodes and so is not considered to be a move in the pebble game. All the other queries are considered to be pebbling or unpebbling moves in the pebble game, depending on the direction of the corresponding triangle.

In the machine configuration C_τ at time τ, nodes 2, 3, and 4 are pebbled. But note that the query string for node 2 can be found by simulating the machine backwards from time τ until query (e), and reading q_2 off of the oracle tape. And if q_3 is given, we can continue simulating backwards until we get to time (c), and read q_4 off the oracle tape as well. The ability to perform this sort of simulation, for any arrangement of triangles, either forwards or backwards in time as needed to find out more than a constant number of the pebbled nodes is what makes our incompressibility argument work.

It will also specify the direction D and contain a concatenation of all the q_j's that are not pebbled because of queries in direction D. Finally, for each q_j that *is* pebbled because of a query in direction D, d will contain its node index j and an integer $\Delta\tau_j$ giving the number of steps from step τ to the time of the next query in direction D involving q_j which causes C_τ to pebble q_j by our above definition. Also we include a short tag k_j indicating which of the 3 possible cases of queries causes the node to be pebbled. Each of the indices j takes space $O(\log t) = O(\log T(n)) = o(S(n))$, and each $\Delta\tau_j$ takes space $O(\log T(n)) = o(S(n))$. The tag is constant size. Thus for sufficiently large n, all the $p/2$ $(j, \Delta\tau_j)$ pairs together take less than $pS/2$ space. Total space so far: less than tS. If $tS < T$, then x will contain some additional bits beyond the concatenation of $q_1 q_2 \ldots q_t$, in which case d includes those extra bits as well. The total length of d will still be less than $T(n) = |x|$.

We now need to show that the description d is actually sufficient to reconstruct x, and give an effective algorithm for doing so. The function computed by this algorithm will define our description system s for descriptions of the form described in the above paragraph. The description system may have any behavior when given descriptions that are not either of that form or of the form specified earlier when we showed how to guarantee that all the q_j's were unique.

The algorithm shown in figure 8 basically just simulates M_i's operation in direction D starting from configuration C_τ, and reads the identifiers of the pebbled nodes off of M_i's simulated oracle tape. The bulk of the algorithm is in the details showing how to simulate all oracle queries correctly.

There is a small subtlety in the fact that this algorithm has, built into it, some of the values of f that are defined by the oracle. Yet the algorithm defines our description system s_i, which is used to pick x and define the $f(q_j)$ values. This would be a circularity that might prevent the oracle from being well-defined, if not for the fact that the portion of f that is built in, that is, $f(b)$ for $|b| < S$, is disjoint from the portion of f that depends on this algorithm, that is, only values of $f(b)$ for $|b| \geq S$. Thus there is no circularity.

We now explain why the simulation performed by the program in figure 8 imitates the real oracle-calling program exactly. When we come to an oracle query operation where the queried bit-string(s) do not appear in our $q[j]$ array and do not have a matching $\Delta\tau_j$, then we know the bit-string(s) must not correspond to a real node in $q_1, \ldots, q_t$, because if they did, then either they were not pebbled due to queries in direction D, in which case they would have been in the description d and would have been present in the initial q array, or else the first query that involved them must have been before the current one (or else some $\Delta\tau_j$ would match), in which case they would have been added to the q array earlier.

Moreover, when we get to a single query q_j, we know we can look up q_{j+1} to answer the query, because it must already have been stored, because either it was not pebbled in direction D in which case it was stored originally, or it was pebbled in direction D in which case the first query involving it must have

Given description d of the form described in the text,
Let $q[1] \ldots q[t]$ be a table of node values, initially all NULL.
Initialize all $q[j]$'s not pebbled in direction D, as given by d.
Simulate M_i in direction D from configuration C_τ, as follows:
    To simulate a single operation of M_i:
        If a non-query operation, simulate it straightforwardly,
            and go ahead to next operation.
        Otherwise, it's an oracle query. Examine the oracle tape.
        If it's not of the form b or $b\#c$ for
            bit-strings b, c, $|b| = |c|$, skip it and go on.
        If $|b| < S$, look up $f(b)$ in a finite table,
            set the oracle tape appropriately, and go on.
        If $|b| > S$, go ahead to the next operation.
        If the query is of the form b, then
            If current time matches a $\Delta\tau_j$,
                set $q[j] = b$.
            If $b = q[j]$ for some $j < t$,
                Set oracle tape to $b\#q[j+1]$,
            else go ahead to the next operation.
        If query is of the form $b\#c$, then
            If current time matches some $\Delta\tau_j$,
                Set $q[j]$ to b or c depending on the tag k_j.
            If $b = q[j]$ and $c = q[j+1]$ for some j,
                Set oracle tape to b,
            else do nothing for this operation.
    Increment time counter after each simulated operation.
    Stop simulation when counter exceeds the largest stored $\Delta\tau_j$.
Print all $q[j]$'s. $\square$

Figure 8: Algorithm to derive the incompressible x via simulation of the reversible machine M_i.

been before this one (since this query is not of the type that would have caused the node to be pebbled in direction D), and so we will already have a value for $q[j+1]$.

Now, given any description d derived from the execution history of a real M_i, the simulation will eventually find values for all nodes, since either they were given initially or they are found eventually as we simulate. Thus the above program given a description d prints out x, and with n chosen large enough, as explained earlier, the length of the description will be smaller than x itself, contradicting the assumption of x's incompressibility relative to s. But we know that some incompressible x's do exist, by the counting argument, and that x was one of them. Therefore for these sufficiently large n, all configurations in which p nodes are pebbled must actually be larger than $\frac{1}{2}pS$. This completes the proof of our lemma. □

Continuing with the definition of the set of pebbled nodes from earlier, we now complete our description of the correspondence between M_i's execution history and the playing of a pebble game. Whenever M_i performs a query q_j and node q_{j+1} was not pebbled immediately prior to this query, we will say that M_i "is pebbling node q_{j+1}" as a move in the pebble game. Similarly, whenever M_i performs a query $q_j\#q_{j+1}$ and node q_{j+1} is not pebbled immediately after this query, we say that M_i "is unpebbling node q_{j+1}." All other oracle queries and computations by M_i are considered as pauses between pebble game moves of these two forms.

It is immediately obvious from this definition that all pebbling/unpebbling moves must obey the pebble game rule that the pebbled status of node q_j can only change if $j=1$ or if node q_{j-1} is pebbled before and after the change. The move is a query, and the presence of the query means the node q_{j-1} is pebbled by definition (unless $j-1=0$, because q_0 is not considered to be a node).

Can we also show that initially no nodes are pebbled? This is a only a little bit harder. Suppose that some node q_j was pebbled in M_i's initial configuration. Then a shorter description of x (for sufficiently large n) can be given as $(j, \Delta\tau_j, x')$, where x' is x with q_j spliced out. This description could be processed via simulation of M_i to produce x in the same way as described earlier, except that this time, the starting configuration C_1 can be produced directly from the known values of M_i and n, and need not be explicitly included in the description. Of course our description system s needs to be able to process descriptions of this form. Then the incompressibility of x in s shows that the assumption that q_j is initially pebbled is inconsistent.

Similarly, if we require our reversible machines to halt in a clean final state, we can show that no nodes are pebbled in that state either. (As we observed earlier, the requirement for a clean final state can be made without loss of generality.)

Finally, M_i eventually pebbles node q_t (by definition) if and only if at some point it asks query q_{t-1} or $q_{t-1}\#q_t$. Thus if M_i does not pebble q_t then we can foil it trivially, as we observed earlier.

So under the above interpretation, M_i exactly obeys all the rules of the Bennett pebble game, starting and finishing with no nodes pebbled, pebbling and unpebbling nodes according to the rules, and winning (being able to decide whether $0^n \in L(A)$) if node q_t is ever pebbled.

Now, Li and Vitányi have shown [17] that any system that obeys the rules of the pebble game and that eventually pebbles a node at or beyond node 2^k must at some time have at least $k+1$ nodes pebbled at once. So let us simply choose n large enough so that $t(n) \geq 2^k$ for some $k \geq 2(c_i+1)$, and also so that $S \geq c_i$. Then we have proven that M_i's space usage at some point is larger than $c_i + c_i S$, because at any time τ when p is maximum,

$$|C_\tau| > \frac{1}{2}pS > \frac{1}{2}kS \geq (c_i+1)S = S + c_i S \geq c_i + c_i S.$$

Therefore, M_i cannot actually obey its enforced space restriction on these large enough n, on the assumption that it eventually pebbles the last node q_t. So that assumption is inconsistent, and so actually the space-restricted M_i must never pebble node q_t, and thus must never ask query q_{t-1}, and so we can indeed foil any such M_i by choosing $f(q_{t-1})$ so that whatever answer M_i gives is wrong.

The reversibility of the machine M_i comes into this proof solely through the observation that if M_i were not reversible, then if the queries that cause nodes to be pebbled at some time τ occur prior to τ, then the description d specifying distances to the relevant queries prior to τ would not actually provide enough information to simulate M_i in reverse from configuration C_τ, because in an irreversible machine, many reverse paths may be possible from a given configuration, and so we could not actually generate x from d, and the proof would not go through.

In any case, the above discussion completes our proof that machine M_i would not correctly decide membership in $L(A)$ for inputs of length $n_i = n$, so long as the oracle A is consistent with the definition above. Now, since machine M_i's behavior on those inputs only depends on the values of the graph function $f(b)$ for bit-strings b up to a certain size (call it z), we are free to extend the oracle definition to similarly foil machine M_{i+1} by picking n_{i+1} so that $S(n_{i+1}) > z$. If one continues the oracle definition process in this fashion for further M_i's *ad infinitum*, then for the resulting oracle, it will be the case that for any M_i, we will have foiled it by the construction of our graph function f for nodes of some size n_i.

Note that this entire oracle construction, as described, is computable, in the sense that if we are given procedures for computing $S(n)$ and $T(n)$, we can write an effective procedure that, given any finite oracle query, returns the result of that query given by A as defined above. The details of the algorithm follow directly from the above definition of A, but would be too tedious to present here.

This concludes our proof of the main theorem. ■

As a small corollary of our theorem, we note that essentially the exact same oracle construction and proof also works for an enumeration of reversible machines that includes all machines that take time $O(T)$ and space $o(S\log(T/S))$, since the pebble game analysis actually shows not only that no constant number of pebbles will suffice, but also that more than $k = \lg 2^k \geq \lg t = \lg(T/S)$ pebbles are required to reach the final node. In this case, however, it is unclear whether we can effectively characterize the machines taking $o(S\log(T/S))$ space, and keep the oracle computable.

Corollary 1. *Lower bound on space for linear-time relativizable reversible simulation of irreversible machines.* For any $\mathbf{R}ST$-constructible functions S, T such that $S <_o T <_o 2^S$, there exists a self-reversible oracle A such that for all $S'(n) <_o S\log(T/S)$, we know that $\mathbf{R}S'T^A \neq ST^A$. Contrapositively, if we are given that $\mathbf{R}S'T^A \supseteq ST^A$ for all self-reversible A, then it follows that $S' \geq_\Omega S\log(T/S)$. Thus, any general linear-time simulation of irreversible machines by reversible ones that is relativizable with respect to self-reversible oracles must take space $\Omega(S\log(T/S))$.

Proof. Essentially the same as for Theorem 1, except that here we know no effective procedure for actually finding the value of n_i for which a given reversible machine M_i necessarily fails if it takes space $o(S\log(T/S))$. (But one must exist.) $\square$

The most space-efficient general linear-time reversible simulation technique currently known was provided by Bennett ([4], p. 770), and analyzed by Levine and Sherman [13] to take space $O(S(T/S)^{1/(0.58\lg(T/S))})$. Bennett's technique relativizes easily to machines with self-reversible oracles.

Thus we see there is still a gap between the known lower and upper bounds. The gap is due to the fact that the space-optimal pebble-game strategy referred to in our proof takes more than linear time in the number of nodes. A lower bound on the number of pebbles used by any asymptotically linear-time pebble game strategy would allow us to similarly expand our lower bound on space used, hopefully to converge with the existing upper bound.

Still, at this point it remains uncertain whether there might still be some general-purpose way for reversible machines to simulate irreversible machines using a technique that does *not* relativize even to self-reversible oracles, in which case the above results would say nothing substantive about the ultimate limits of real reversible machines.

5 Future Work

Given the work above, an obviously desirable next step would be to show that $\mathbf{R}ST \neq ST$ for a fairly reasonable serial model of computation that lacks any oracle. It seems likely that a language and incompressibility argument similar to those used in the above proof would be useful for this next step as well. For example, suppose that machine inputs are incompressible directed outdegree-1 graphs, and are provided in the form of a random-access read-only memory that allows instantly looking up the ID of a node's successor given the ID of the node. With this sort of specialized input framework, the situation is virtually the same as in the oracle problem we constructed above. Given an input ROM of size n, with words of size $O(\log n)$, an irreversible machine can completely follow any chain of pointers in space $O(\log n)$ and time linear in the length of the chain. It seems likely that if a reversible machine could do this in all cases, it would contradict the incompressibility of the graph, similarly to our oracle proof. But we have not yet investigated this possibility in detail.

Another approach would be to try to prove rigorously that if one-way functions exist, then the problem of returning the result of repeatedly applying the one-way function to an input would have essentially the same structure and the same reversible difficulty as the problem of following the directed chain of nodes to decide the oracle language $L(A)$. Then a non-relativized separation of reversible and irreversible space-time complexity classes would be at least as certain as the security of public-key cryptography.

Possibly more difficult will be to derive an $\mathbf{R}ST^A \neq ST^A$ result for models in which the inputs can be accessed bit-by-bit (such as a normal Turing machine model), rather than only in size $S(n)$ chunks. Our technique for reconstructing an incompressible input from reversible machine configurations seems on the face of it inapplicable to such a case, but perhaps some related technique would apply. At this point it is hard to say. A good starting point might be to try to rederive our oracle result with an oracle that returns only a single bit at a time.

Given oracles that reversibly return only a single bit at a time, it would also be of interest to consider random oracles R of that type, and hopefully show that $\mathbf{R}ST^R \neq ST^R$ with probability 1.

If we find that it is difficult to prove our desired separation of classes for the non-relativized case, this may mean that any proof of the inequality of the classes might not relativize. It is an interesting question to consider whether there might exist another oracle B relative to which $\mathbf{R}ST^B = ST^B$. One idea for this might be to have B identify the unique predecessors of states of irreversible machines having a given initial state, but it seems that no B that does exactly this can exist for irreversible machines that are allowed to query B, because the machines could falsify B's predictions in a fashion reminiscent of the standard proof of the undecidability of the halting problem. However, some other choice of B might exist that would avoid this problem but still equate the two classes.

It may turn out that we can prove $\mathbf{R}ST \neq ST$ in the non-relativized world only for some range of functions S and T that is more limited than the range dealt with here ($S <_o T <_o 2^S$). But ideally, we would like to prove it for the entire range.

Finally, as stated in the introduction, the main practical motivation for all this is to make progress towards showing that reversible models of computation will be necessary to accurately model computer architectures that take full advantage of the laws of physics while recognizing their limitations.

In particular, we suspect that many parallel models, such as irreversible 3-D meshes, predict better scaling for some classes of problems than is actually physically possible. If we could show that irreversible 3-D meshes are more powerful than reversible ones (that $\mathbf{R}ST \neq ST$ for these meshes), that would be a strong clue that the irreversible meshes are unrealistic, since any computers in the physical world (unless they utilize aspects of quantum computation) should be simulatable on an entirely reversible and non-quantum 3-D mesh model with only a constant factor slowdown, since the physics of the world is itself reversible.

Of course, ultimately we would like to also know for certain whether quantum models of computation are really physically implementable as well, and whether they are more powerful than any physically-implementable classical model. But we are not specialists in quantum systems, and so for now we leave those deep and important questions to others.

Acknowledgments

Thanks to Dan Spielman for his lectures in the Advanced Complexity Theory course at MIT, especially for his excellent explanation of the Baker/Gill/Solovay relativized separation of **P** and **NP** [1], from which our proof draws some ideas.

Gratitude to Alain Tapp of the University of Montreal for visiting us, telling us about his own work on reversible computing theory including [10], and commenting on an early version of our proof idea.

Thanks very much to Michael Sipser for his expert guidance and repeated encouragement, and especially for suggesting the use of an incompressibility argument to make the proof go through.

Our appreciation to Ming Li and Paul Vitányi for their earlier analysis of the pebble game [17] on which the present result depends, and for their excellent writings on the incompressibility method in their book on Kolmogorov complexity [15], which we found extremely helpful.

And finally, many thanks to Tom Knight, Norm Margolus and all my fellow students on the MIT Reversible Computing Project for providing a stimulating and supportive work environment.

References

[1] Theodore Baker, John Gill, and Robert Solovay. Relativizations of the $\mathcal{P} =? \mathcal{NP}$ question. *SIAM J. Computing*, 4(4):431–442, December 1975.

[2] C. H. Bennett. Logical reversibility of computation. *IBM J. Research and Development*, 6:525–532, 1973.

[3] C. H. Bennett. The thermodynamics of computation, a review. *Int'l J. Theoretical Physics*, 21(12):905–940, 1982.

[4] C. H. Bennett. Time/space trade-offs for reversible computation. *SIAM J. Computing*, 18(4):766–776, 1989.

[5] Don Coppersmith and Edna Grossman. Generators for certain alternating groups with applications to cryptography. *SIAM J. Appl. Math.*, 29(4):624–627, December 1975.

[6] Pierluigi Crescenzi and Christos H. Papadimitriou. Reversible simulation of space-bounded computation. *Theoretical Computer Science*, 143:159–165, 1995.

[7] Michael P. Frank. Physically-motivated models of computation for complexity theory. Manuscript under development, March 1997. `http://www.-ai.mit.edu/~mpf/rc/memos/M05_physical.html`.

[8] E. F. Fredkin and T. Toffoli. Conservative logic. *Int'l J. Theoretical Physics*, 21(3/4):219–253, 1982.

[9] R. Landauer. Irreversibility and heat generation in the computing process. *IBM J. Research and Development*, 5:183–191, 1961.

[10] Klaus-Jörn Lange, Pierre McKenzie, and Alain Tapp. Reversible space equals deterministic space. Manuscript, December 1996. To be published in proc. 12th Annual IEEE Conf. on Computational Complexity (CCC '97).

[11] Y. Lecerf. Machines de turing réversibles. insolubilité récursive en $n \in N$ de l'équation $u = \theta^n$, où θ est un "isomorphisme de codes". *Comptes Rendus hebdomadaires des seances de l'academie des sciences*, 257:2597–2600, 1963.

[12] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays · Trees · Hypercubes.* Morgan Kaufmann Publishers, San Mateo, California, 1992.

[13] Robert Y. Levine and Alan T. Sherman. A note on Bennett's time-space tradeoff for reversible computation. *SIAM J. Computing*, 19(4):673–677, 1990.

[14] Ming Li, John Tromp, and Paul Vitányi. Reversible simulation of irreversible computation by pebble games. Manuscript submitted to *Physica D*, 1997. `http://www.cwi.nl/~paulv/physics.html`.

[15] Ming Li and Paul Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications.* Graduate Texts in Computer Science. Springer-Verlag, 2nd edition, 1997.

[16] Ming Li and Paul M. B. Vitányi. Reversibility and adiabatic computation: trading time and space for energy. *Proc. Royal Society of London, Ser. A*, 452:1–21, 1996.

[17] Ming Li and Paul M. B. Vitányi. Reversible simulation of irreversible computation. In *Proc. 11th IEEE Conference on Computational Complexity*, Philadelphia, Pennsylvania, May 24–27, 1996.

[18] N. H. Margolus. *Physics and Computation.* PhD thesis, MIT Artificial Intelligence Laboratory, 1988.

[19] Norman Margolus and Lev B. Levitin. The maximum speed of dynamical evolution. In T. Toffoli, M. Biafore, and J. Leão, editors, *PhysComp96*, pages 208–211. New England Complex Systems Institute, 1996. Also online in the *InterJournal*.

[20] J. E. Pin. On the languages accepted by finite reversible automata. In Thomas Ottman, editor, *Automata, Languages and Programming, Proc. 14th Int'l Colloq. (ICALP)*, volume 267 of *Lecture Notes in Computer Science*, pages 237–249. Springer-Verlag, 1987.

[21] Tommaso Toffoli. Computation and construction universality of reversible cellular automata. *J. Computer and System Sciences*, 15:213–231, 1977.

[22] Tommaso Toffoli. Reversible computing. Technical memo MIT/LCS/TM-151, MIT Lab for Computer Science, February 1980. Out of print; available from NTIS.

[23] Carlin J. Vieri. Pendulum: A reversible computer architecture. Master's thesis, MIT Artificial Intelligence Laboratory, 1995.