Chapter 3

Reversible computing theory

In the previous chapter, we set the stage for our research by reviewing the known physical limits on computation, including the entropic cost of logically irreversible information loss. We saw that avoiding this cost requires the use of computational primitives that possess the special property of logical reversibility. This observation leads naturally to the question: What implications would logical reversibility have in the context of the traditional theory of computation?

This chapter addresses that question, while also introducing the related question of how the traditional measures of complexity and models of computation will need to be adjusted to more effectively cope with thermodynamic issues and other important physical considerations. That line of study is continued in chapters 5 and 6.

One reason that computer designers have not yet rushed to adopt reversible computing principles is that purely reversible operation is not necessarily optimal in all circumstances. For many applications of computer technology today and in the future, energy dissipation may not be a limiting factor. In such circumstances, purely reversible operation appears to incur significant computational overheads compared to irreversible operation. If energy dissipation is modeled as costing exactly nothing, then it seems that the total cost overhead factor for pure reversible computing becomes unboundedly large as problem sizes increase.

In §3.4 of this chapter, we will rigorously prove a technical theorem in computational complexity theory which suggests that such overheads are inevitable, and that no amount of clever improvements of reversible algorithms can avoid these overheads on all problems. This result indicates that if we wish to be able to perform asymptotically optimally *even under cost models in which the energy cost is zero*, then our computer models should at least include the *option* of not being completely reversible.

However, in chapter 6, we will show that if energy dissipation has *any* non-zero cost, then our physical model of computation must also include the option to have an arbitrarily *high* degree of reversibility, if it is to achieve asymptotically optimal speed and cost-efficiency on all problems.

But first, in §3.1 and §3.2, we will review general concepts of models of computation, computability, and complexity, and introduce a few new measures of complexity that attempt to better capture important physical considerations. Then in §3.3 we review the major results of existing reversible computing theory, leading up to our own contributions in §3.4. Finally, §3.5 sums up the comparison of traditional reversible and irreversible computing models.

3.1 Models of computation

Discussions of the theory of computation often start with the definition of a particular model of computation to work with, such as, for example, Turing machines. However, in this thesis, we do not wish to pick a particular model, since our interest is in comparing the relative efficiency of different models. If we wished to pursue a completely formal mathematical approach, we would need to give a precise definition of what a model of computation *is* in general, describe various particular models in terms of that general framework, define what it means to compare two models under that framework, and then prove various theorems comparing the different models. This would be straightforward but tedious, and it is unclear whether we would learn anything important from that highly formal approach that is not already sufficiently clear using our more informal understanding of the situation.

Therefore, in this thesis we will refrain from presenting a detailed formal explication of the concept of a "model of computation," and instead we will rest our discussion on the intuitive understanding of the phrase that the reader will be expected to have, given a general background in computer science. To refresh the reader's memory, a partial list of existing models of computation may be helpful (table 3.1).

Informally speaking, a model of computation merely delineates a space of abstract computing machines, and the computations that run on them. Most models were originally introduced as an attempt to approximate some class of physical machines; however, the existing models unusually end up ignoring one or another of the important realities of physical law that we saw in chapter 2. Sections 5.2 and 5.3 of ch. 5 review some of the problems with the existing models, and discusses candidates for a new model (which we might call PM, the "physical machine") intended to exactly represent the computing capabilities of physics.

Physically realistic or not, any abstract model of computation needs to be reduced to a physical implementation in order to actually run. In chapter 6 we will compare the power of two fairly realistic classes of models of physically-implemented machines: the FIA (fully irreversible architectures) and the TPRA (time-proportionally reversible architectures), and we show that the TPRAs are strictly more efficient, in several

Notation	Model name	Example references
PRF	Primitive recursive functions	Rogers 1987 [141], §1.2, pp. 5–9
RF	Recursive functions	[141], ch. 1
FA	Finite automata	Hopcroft & Ullman 1979 [86], ch. 2
RFA	Reversible finite automata	Pin 1987 [136]
TM	Turing machines	[169]; [86], ch. 7
RTM	Reversible Turing machines	[99, 16, 98]
NTM	Nondeterministic Turing machines	[86], §7.5
CA	Cellular automata	von Neumann 1966 [182], Toffoli & Margolus 1987 [164]
BBM	Billiard-ball model	Fredkin [74]
RAM	Random access machines	Papadimitriou 1994 [133], §2.6
PRAM	Parallel random access machines	Papadimitriou 1994 [133], §15.2, pp. 371–375
BLC	Boolean logic circuit	Papadimitriou 1994 [133], §4.3
3dM	3-d mesh	Leighton 1992 [101], ch. 1

Table 3.1: Some existing theoretical models of computation.

physically-relevant senses.

3.1.1 Computability

For any model of computation, an obvious first question is "What computations can it possibly perform?" (Given unlimited resources.) This question was the subject of much early research on computation, but eventually it was realized that a large variety of physically reasonable models of computation can all compute exactly the same set of functions, namely the *recursive* (now just called *computable*) functions (cf. [141]), and so the issue became less interesting. The famous "Church's thesis" is the conjecture that the recursive functions *are* indeed exactly the functions that real physically-realizable machines can compute; the conjecture is true as far as anyone knows, and it would be extremely surprising if physical machines were to turn out to be able to compute non-recursive functions.

Of course, there also exist weaker models of computation that cannot even compute all recursive functions, such as finite automaton (FA) models.

With computability turning out to be mostly a non-issue, the next natural issue in computing theory is to discover how difficult or *complex* one finds various computational tasks to be under a given model of computation, or (in complementary terms), how *efficiently* the model can perform on various tasks.

3.2 Computational complexity and efficiency

Now we review some of the basic concepts used in traditional computational complexity theory, and extend them to capture some new, more general measures of computational complexity and cost-efficiency that will help us better address real-world concerns in later sections.

3.2.1 Computational efficiency vs. computational complexity

The focus of this thesis is on how to achieve maximum *computational efficiency*, which can mean several things, but most often we will use it to mean *cost efficiency*, defined as follows.

Given some way ¥ of characterizing the *cost* of a computation (or any process), one very general notion of efficiency is the fraction of the cost that is actually well-spent. In other words, if the minimum *possible* cost to perform some task is $\$_{\min}$, and the actual costs incurred by a particular computation that performs that task are $\$$, then we can say that the cost-efficiency $\%_{\$}$ of the computation (under the cost measure ¥) is

$$\%_{\$} = \frac{\$_{\min}}{\$} \tag{3.1}$$

because only $\$_{\min}$ out of the total cost $\$$ was really warranted; the remainder $\$ - \$_{\min}$ was wasted.

Thus, whatever the minimum cost $\$_{\min}$ for a task, in order to maximize the efficiency $\%_{\$}$, one should try to minimize the actual cost $\$$. This leads to the frequent emphasis in computer science on characterizing and studying various abstract measures of cost, which are often referred to in theoretical computer science as measures of *computational complexity*.

3.2.2 Characterizing computational complexity

In this section we examine how measures of computational complexity are traditionally characterized, and propose the use of some new complexity measures that may allow different computational models to be compared in a more realistic way.

3.2.2.1 Scaling with problem size

When comparing the cost-efficiency of two algorithms or two models of computation, it is sometimes difficult to make a definitive distinction as to which candidate is better,

Our Notation for the Cost Measure	Meaning
Computational cost measures.	
N_{ops}	Number of primitive operations.
T	Number of computational clock "ticks" (called "time" in traditional complexity theory).
S	Maximum memory used at any time (called "space" in traditional complexity theory).
(S, T)	Computational "space" paired with "time" (p. 57).
ST	Computational "space" times "time" (p. 57).
Physical cost measures.	
t_{phys}	Physical time taken.
$\mathcal{V}_{\text{max}}$	Maximum physical volume of space used.
S_{tot}	Total entropy generated.
$\$_{\text{c}}$	Comprehensive physical cost complexity (p. 58).
$\$_{\text{s}}$	Simplified physical cost complexity (p. 58).

Table 3.2: Some measures of cost or complexity. We distinguish the non-physical, "computational" cost measures from the physical cost measures. The physical measures can be accurately determined only for models of computation that realistically take into account physical constraints on computation such as we discussed in chapter 2.

---

if one of them is more efficient on some problems, and the other one is more efficient at others. Even within a particular class of problems, one machine may be better at small problems and the other at large ones.

However, if we look at how the performance of the two machines scales as the problem size increases, it may often be the case that one machine performs better than the other at problems of *all* sizes above a certain size, and the ratio between the efficiency of the two machines may even grow unboundedly large as problem sizes increase. Asymptotic order-of-growth analysis (see table F.4, p. 409) is the traditional tool for determining if such relationships hold, because it allows ignoring the many details of algorithm design that cause constant-factor differences in complexity, which often end up being irrelevant in an asymptotic determination of which machine is better.

Table 3.2.2.1 lists several measures of complexity which we will now discuss.

3.2.2.2 Traditional measures of complexity

Traditionally in computer science, theoreticians study only very simple measures of complexity, in order to make their analysis easier. Two of the most popular measures are time complexity and space complexity.

Time complexity. The "time complexity" of a computation can be characterized simply as the amount of physical time t_{phys} that the computation takes (from its start to its end), or as the number N_{ops} of computational "operations" (at whatever level of interest) that are performed, which is proportionally equivalent to real time if, for example, operations are performed serially and take $\Theta(1)$ (*i.e.*, constant) time each. If operations are performed in parallel, a better approximation to time would be the number of "ticks" T of some (real or imagined) computational "clock" that is thought of as synchronizing the operations of all the processing elements.

The problem with using time complexity alone as a cost measure is that it ignores the cost of the computer that is needed to solve a problem with the minimum time complexity. The minimum time complexity might only be achieved by a computer that is unfeasibly expensive.

One may reply that the machine cost is negligible because it may be amortized over arbitrarily many uses of the machine into the future, but one can counter with the point that whenever the computer is fully occupied with solving the given problem, its components can not meanwhile be used for another problem, so there is an opportunity cost inherent in using a large machine that must be considered as well.

Thus, minimizing only time complexity may completely miss the solution that minimizes cost in the real world.

Space complexity. Another measure of computational complexity which attempts to take the machine cost into account is space complexity, that is, the maximum amount of digital storage (in bits, say) that is in use at any point during the computation (we will denote this as S).

Given fixed lower bounds to the physical size and mass-energy required for a bit's worth of storage, space complexity can also be equated (within a constant factor) to the amount of physical volume ($\mathcal{V}_{\mathrm{max}}$) or mass in the computer, assuming there are no cost advantages in storing bits with an asymptotically increasing mass-per-bit or volume-per-bit. We conjecture that asymptotically, this assumption is true.

Of course, like time complexity, space complexity by itself is also inaccurate for real-world situations. Most significantly, it ignores the impact of the length of time during which the given amount of storage needs to be used. If the storage requirements for a computation are large, but the computation is rather short, or even if just the time during which the bulk of the storage is in use is short, then the computation may actually be less costly, in real terms, than a computation that has a smaller formal space complexity but which occupies that space for an extremely long time.

3.2.2.3 Some new measures of complexity

Given the inadequacies of the most popular traditional measures of complexity, we now describe some new alternative measures which attempt to more closely approximate the real-world economics of computing.

Joint space-time complexity. We saw earlier that both space complexity and time complexity, although they each took important cost factors into account, were individually incomplete. We can try to improve on the situation by combining both space and time complexity into a single measure of complexity.

One way to combine a space complexity measure s and a time complexity measure t is to simply group them into a pair (s, t). We can define a partial order $\succsim$ between pairs (s_1, t_1) and (s_2, t_2) by saying, for example, that $(s_1, t_1) \succsim (s_2, t_2)$ iff $s_1 \succsim s_2$ and $t_1 \succsim t_2$. (The $\succsim$ notation is defined in table F.4, p. 409.) However, this approach suffers from the problems that two complexity measurements may be incomparable (for example if $s_1 \prec s_2$ but $t_1 \succ t_2$), and that it is difficult to define a numerical measure of overall efficiency in this system. However, this simple complexity measure still suffices for some purposes, such as for our proof in §3.4.

Space-time product complexity. One interesting, improved measure of complexity is the product of space and time complexity. This comes closer to a true measure of cost because it increases monotonically with both space and time and allows comparisons between any two instances. It can be viewed as a measure of *rental cost*, the cost of renting a computer having storage capacity s for a period of time t; we might expect such a cost to be roughly linear in both storage capacity and time. Another way to look at the product is as a measure of the total volume of spacetime (as in the theory of relativity) that is dedicated to the computation.

However, even the space-time product is still somewhat inaccurate, since it does not take into account that a particular algorithm may not have constant space usage over time, and that the resources that are unused by the algorithm during a particular period of time can (in an appropriate machine architecture) be used for solving other problems during that time, thus reducing the effective cost of the program whose complexity we are measuring.

Another point is that besides spacetime volume, there is another resource that a computation uses up: namely, free energy. Energy that is dissipated by the computer is forever unavailable for use in other computations, because it is in a disorganized, maximum-entropy form that cannot do useful work. (We discussed these issues in much more detail in §2.5.) So this dissipation has a cost. In fact, in contexts such as battery-powered portable computers, the energy costs may be fairly high because the readily-available supply of energy is so limited. So a comprehensive model ought to take energy costs into account. One way to characterize free energy loss is by the total amount of entropy that is generated during the computation (S_{tot}).

Finally, there is the point that the storage space itself can be separated into several constituent entities that separately contribute to the total rental costs: the mass-energy of the computation/storage medium, the volume of physical space it occupies, and perhaps even its surface area (real estate it occupies). Mass-energy can be further broken down into free energy and rest mass, which can be further decomposed into the cost of various types of constituent components and the raw materials that they are made of; but we will not go this far in our modeling.

These observations lead to the following new complexity measures.

Comprehensive physical cost complexity. For a computation (or really, any) process that increases total entropy by S, takes total real time t, and that at times $0 \leq \tau \leq t$ (between the start and end of the computation) occupies spatial volume $\mathcal{V}(\tau)$, contains free energy $E(\tau)$, rest mass $M(\tau)$, and has a minimum surface area of $A(\tau)$, we define the *comprehensive physical cost* $\$_{\text{c}}$ of the process as

$$\$_{\text{c}} \;\equiv\; \pounds_{\text{S}} S + \int_0^t \Big[\pounds_{\text{V}} \mathcal{V}(\tau) + \pounds_{\text{E}} E(\tau) + \pounds_{\text{M}} M(\tau) + \pounds_{\text{A}} A(\tau) \Big] \, d\tau \tag{3.2}$$

where the various $\pounds_X \geq 0$ are *cost coefficient* constants whose values are parameters of the cost model. The $\pounds_X$ convert all cost elements to some canonical cost unit, perhaps even a monetary unit.

This cost model is very comprehensive, probably more so than needed. In our explorations of the efficiency of reversible and irreversible machines in chapter 6, we have found that not all of the above terms need to be included in the cost model in order to find the optimal machine configurations for the kinds of computational tasks we have considered so far. So we also suggest a simplified version of this model.

Simplified physical cost complexity. For a computation process that generates entropy S, takes total real time t, and that at times $0 \leq \tau \leq t$ requires a free energy allocation of $E(\tau)$, we define the *simplified physical cost* $\$_{\text{s}}$ of the process as

$$\$_{\text{s}} \;\equiv\; \pounds_{\text{S}} S + \int_0^t \pounds_{\text{E}} E \, d\tau \tag{3.3}$$

where the $\pounds \geq 0$ constants are parameters of the cost model. Given the Margolus-Levitin bound on computation rate from §2.4, the second term in this cost measure can be considered a measure of the maximum number of states that could be traversed using the given energy profile over the given time.

We propose that cost models like the above are appropriate for exploring the asymptotic physical limits of computation.

3.2.3 Complexity classes

A complexity measure tells us how to assign a cost to a particular instantiation of a computation process. In chapter 5 we will discuss a variety of models of computation processes. Given a complexity measure and a model of computation, we can characterize the complexity of any program written for that model, as a function of the length n_{in} of its input (in bits, say). The program complexity for length n_{in} is often defined as the worst-case complexity of the program over all the inputs of length n_{in}.

Further, we can define the complexity of a given *task* under a model of computation as the complexity of the program that performs that task with the lowest program complexity, on that model.

A *complexity class* is the set of all problems that can be solved under a given model of computation within given bounds on asymptotic complexity, according to a given complexity measure.

3.3 Review of existing reversible computing theory

In this section we review the past developments in reversible computing theory. Much of our predecessors' work can be interpreted as an attempt to compare the computational efficiency of reversible and irreversible machines under various complexity measures and models of computation. In this section we will show how each of the existing results can be interpreted in this way, and then in §3.4 and ch. 6 we will carry this effort onward to the new complexity measures that we proposed in §3.2.2.3.

3.3.1 Reversible models of computation

Reversible models of computation can be easily defined in general as models of computation in which the transition function between machine configurations has a single-valued inverse. In other words, the directed graph showing allowed transitions between states has in-degree 1. In this thesis we will always deal with machines that are deterministic, so that the configuration graph always has out-degree one as well. See figure 3.1, p. 60.

3.3.2 Computability in reversible models

As we already noted in §3.1.1, one of the most important questions to answer for any new kind of computation is "What functions it can compute at all?" This comes before efficiency questions, since obviously a machine's efficiency at a task is meaningless if the machine cannot even perform the task.

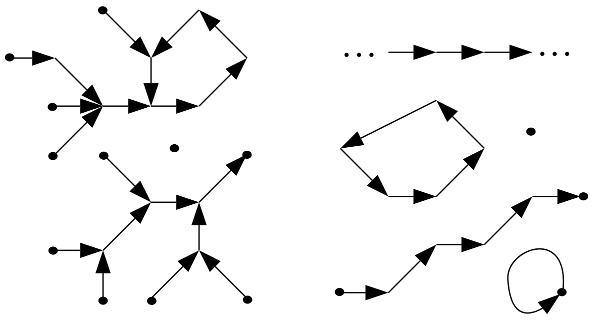

Figure 3.1: Machine configuration graphs in (deterministic) reversible and irreversible models of computation.

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, if we wish, permit configurations having 0 predecessors (initial states) and/or 0 successors (final states).

3.3.2.1 Unbounded-space reversible machines are Turing-universal

In his 1961 paper [97], Landauer had already pointed out that arbitrary irreversible computations could be embedded into reversible ones by simply saving a record of all the information that would otherwise be thrown away (*cf.* §3 of [97]). This observation makes it obvious that reversible machines with unbounded memory can certainly compute all the Turing-computable functions.

We will call this idea, of embedding an irreversible computation into a reversible one by saving a history of garbage, a "Landauer embedding," since Landauer seems to have been the first to suggest it.

3.3.2.2 Reversible finite automata are especially weak

In contrast, in 1987 Pin [136] investigated reversible *finite* automata, which he defined as machines with fixed memory reading an unbounded-length one-way stream of data, and found that they cannot even decide all the regular languages, which means that technically they are strictly less powerful than normal irreversible finite automata, which are in turn strictly less powerful than unbounded-space Turing machines.

So there are functions computable by an irreversible machine with fixed memory that no purely reversible machine with fixed memory can compute, given an external one-way stream of input. We should note, however, that this incapacity might be due solely to the non-reversible nature of the input flow, rather than to the finiteness of the automaton memory itself. Conceivably, if a finite reversible machine was permitted to read backwards as well as forwards through its read-only input, and perform some sort of "unread" operations, it might then be able to recognize any regular language. But we have not investigated that possibility in detail.

That issue aside, in the rest of this thesis we consider models of computation that permit access to arbitrarily large amounts of memory as input sizes increase. For such machines, pure computability is no longer an issue, and we turn to questions of computational efficiency.

3.3.3 Time complexity in reversible models

One of the most common simple measures of computational cost we have seen is "time complexity," which in a theoretical computer science context often means the number of primitive operations performed. Landauer's suggestion (*cf.* §3 of [97]) of embedding each irreversible operation into a reversible one makes it clear that the number of such operations in a reversible machine need not be larger than the number for an irreversible machine, as was demonstrated more explicitly by many later embeddings *e.g.*, [99, 16]. So under the time complexity measure by itself, reversibility does not hurt.

Can a reversible machine perform a task using *fewer* computational operations than an irreversible one? Obviously not, if we take reversible operations to just be a special case of irreversible operations. However, physically speaking, actually it is the converse that is true: so-called "irreversible" operations, implemented physically, are really just a special case of reversible operations, since physics is *always* reversible at a low level. We will see the implications of this for *physical* time complexity in ch. 6. But, using the usual computer-science definition of time as the number of *computational* operations required, clearly reversible machines can be no more "time"-efficient than irreversible ones.

Although Lecerf and Bennett explicitly discussed their time-efficient reversible simulations only in the context of Turing machines, the approach is easily generalized to any model of computation in which we can give each processing element access to an unbounded amount of auxiliary unit-access-time stack storage. For example, based on Toffoli's embedding [160], one could use essentially the same trick to create a time-efficient simulation of irreversible cellular automata on reversible ones, by using an extra dimension in the cell array to serve as a garbage stack for each cell of the original machine. (To actually recycle the garbage in a CA, we would also need a boundary condition that applies globally after an appropriate amount of time in order to reverse the simulation.)

3.3.4 Reversible entropic complexity

The original point of reversibility was not to reduce time but to reduce energy dissipation, or in other words entropy production. Can this be done by reversible machines? In 1961 Landauer [97] argued that it could not, since if we cannot get rid of the "garbage" bits that are accumulated in memory, they just constitute another form on entropy, no better in the long term than the kind produced if we just irreversibly dissipated those bits into physical entropy right away.

3.3.4.1 Lecerf reversal

However, in 1963, Lecerf [99] formally described a construction in which an irreversible machine was embedded into a reversible one that first simulated the irreversible machine running forwards, then turned around and simulated the irreversible machine in reverse, uncomputing all of the history information and returning to a state corresponding to the starting state. If anyone familiar with Landauer's work had noticed Lecerf's paper in the 1960's, it would have seemed tantalizing, because here was Lecerf showing how to reversibly get rid of the garbage information that was accumulated in Landauer's reversible machine in lieu of entropy. So maybe the entropy production can be avoided after all!

Unfortunately, Lecerf was apparently unaware of the thermodynamic implications of reversibility; he was concerned only with determining whether certain questions about reversible transformations were decidable. Lecerf's paper did not address the issue of how to get useful results out of a reversible computation. In Lecerf's embedding, by the time the reversible machine finishes its simulation of the irreversible machine, any outputs from the computation have been uncomputed, just like the garbage. This is not very useful!

3.3.4.2 The Bennett trick

Fortunately, in 1973, Charles Bennett [16], who was unaware of Lecerf's work but knew of Landauer's, independently rediscovered Lecerf reversal, and moreover added the ability to retain useful output. The basic idea was simple: one can just reversibly copy the desired output into available memory before performing the Lecerf reversal! As far as we can tell, this trick had not previously occurred to anyone.

Bennett's idea suddenly implied that reversible computers could in principle be *more* efficient than irreversible machines under at least one cost measure, namely entropy production. To compute an output on an irreversible machine, one must produce an amount of entropy roughly equal to the number of (irreversible) operations performed; whereas the reversible machine in principle can get by with *no* new entropy production, and with an accumulation of only the desired output in memory.

3.3.4.3 Entropy proportional to speed

Unfortunately, absolutely zero entropy generation per operation is achievable in principle only in the ideal limit of a perfectly-isolated ballistic (frictionless) system, or in a Brownian-motion-based system that makes zero progress forwards through the computation on average, and takes $\Theta(n^2)$ expected time before visiting the nth computational step. In useful systems that progress forwards at a positive constant speed, the entropy generation per operation appears to be, at minimum, proportional to the speed. (We do not yet know how necessary this relationship is, but it appears to be the case empirically.) A cost analysis that takes both speed and entropy into account will need to recognize this tradeoff. We do this is chapter 6.

3.3.5 Reversible space complexity

In addition to the number of computational operations performed and the entropy produced, another important element of cost is the number S of memory cells that are required to perform a computation.

3.3.5.1 Initial estimates of space complexity.

As Landauer pointed out [97], his simple strategy of saving all the garbage information appears to suffer from the drawback that the amount of garbage that must be stored in digital form is as large as the amount of entropy that would otherwise have been generated. If the computation performs on average a constant number of irreversible bit-erasures per computational operation, then this means that the memory usage becomes proportional to the number of operations. This means a large asymptotic increase in memory usage for many problems; up to exponentially large. Even if the garbage is uncomputed using Lecerf reversal, this much space will still be needed temporarily during the computation.

3.3.5.2 Bennett's pebbling algorithm

In 1989, Bennett [19] introduced a new, more space-efficient reversible simulation for Turing machines. This new algorithm involved doing and undoing various-sized portions of the computation in a recursive, hierarchical fashion. Figure 3.2 is a schematic illustration of this process. We call this the "pebbling" algorithm because the algorithm can be seen as a solution to a sort of "pebble game" or puzzle played on a one-dimensional chain of nodes, as described in detail by Li and Vitányi '96 [105]. (Compare figure 3.2(a) with fig. 3.7 on page 79.) We will discuss the pebble game interpretation and its implications in more detail in §3.4.2.

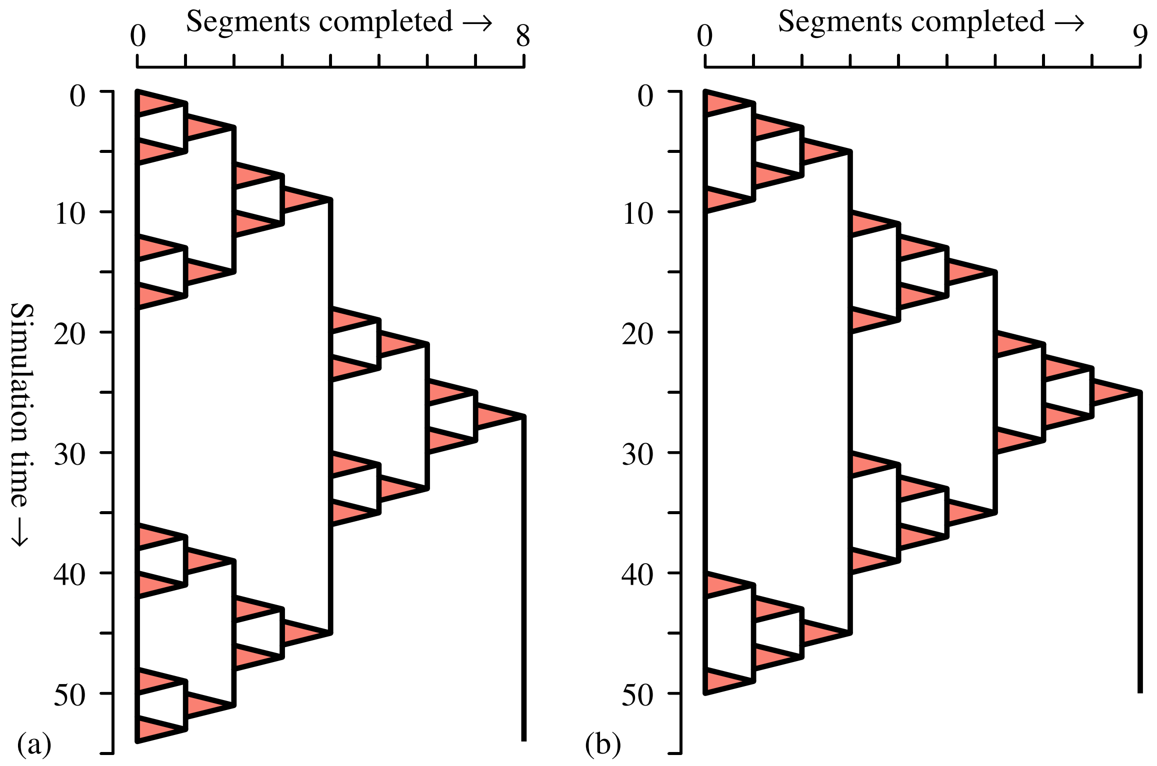

Figure 3.2: Illustration of two versions of Bennett's 1989 algorithm for reversible simulation of irreversible machines. Diagram (a) illustrates the version with $k = 2$, diagram (b) the version with $k = 3$. (See text for explanation of k.)

In both diagrams, the horizontal axis indicates which segment of the original irreversible computation is being simulated, whereas the vertical axis tracks time taken by the simulation in terms of the time required to simulate one segment. The black vertical lines represent times during which memory is occupied by an image of the irreversible machine state at the indicated stage of the irreversible computation, whereas the shaded areas within the triangles represent memory occupied by the storage of garbage data for a particular segment of the irreversible computation being simulated.

Note that in (b), where $k = 3$, the 9th stage is reached after only 25 time units, whereas in (a) 27 time units are required to only reach stage 8. But note also that in (b), at time 25, five checkpoints (after the initial state) are stored simultaneously, whereas in (a) at most four are stored at any given time. This illustrates the general point that higher-k versions of the Bennett algorithm run faster, but consume more memory.

The overall operation of the algorithm is as follows. The irreversible computation to be simulated is broken into fixed-size segments, whose run time is proportional to the memory required by the irreversible machine. The first segment is reversibly simulated using a Landauer embedding (§3.3.2.1). Then the state of the irreversible machine being simulated is checkpointed using the Bennett trick of reversibly copying it to free memory. Then, we do a Lecerf reversal (§3.3.4.1) to clean up the garbage from simulating the first segment.

We proceed the same way through the second segment, starting from the first checkpoint, to produce another checkpoint. After some number k of repetitions of this procedure, all the previous checkpoints are then removed by reversing everything done so far except the production of the final checkpoint. Now we have only a single checkpoint which is k segments along in the computation. We repeat the above procedure to create another checkpoint located another k segments farther along, and then again, and again k times, then reverse everything again at the higher level to proceed to a point where we only have checkpoint number k^2 in memory. The procedure can be applied indefinitely at higher and higher levels.

In general, for any number n of recursive higher-level applications of this procedure, k^n segments of irreversible computation are be simulated by $(2k-1)^n$ reversible simulations of a single segment, while having at most $n(k-1)$ intermediate checkpoints in memory at any given time [19].

The upshot is that if the original irreversible computation takes time T and space S, then the reversible simulation via this algorithm takes time $O(\mathsf{T}^{1+\epsilon})$ and space $O(\mathsf{S}\log\mathsf{T}) = O(\mathsf{S}^2)$. As k increases, the ϵ approaches 0 (very gradually), but unfortunately the constant factor in the space usage increases at the same time [103].

Li and Vitányi '96 [105] proved that Bennett's algorithm (with $k = 2$) is the most space-efficient possible pebble-game strategy for reversible simulation of irreversible machines.

Crescenzi and Papadimitriou '95 [42] later extended Bennett's technique to provide space-efficient reversible simulation of *nondeterministic* Turing machines as well.

3.3.5.3 Achieving linear space complexity

Bennett's results stood for almost a decade as the most space-efficient reversible simulation technique known, but in 1997, Lange, McKenzie, and Tapp [98] showed how to simulate Turing machines reversibly in linear space—but using worst-case exponential time. Their technique is very clever, but simple in concept: Given a configuration of an irreversible machine, they show that one can reversibly enumerate its possible predecessors. Given this, starting with the initial state of the irreversible machine, the reversible machine can traverse the edge of the irreversible machine's tree of possible configurations in a reversible "Euler tour." (See figure 3.3.) This is

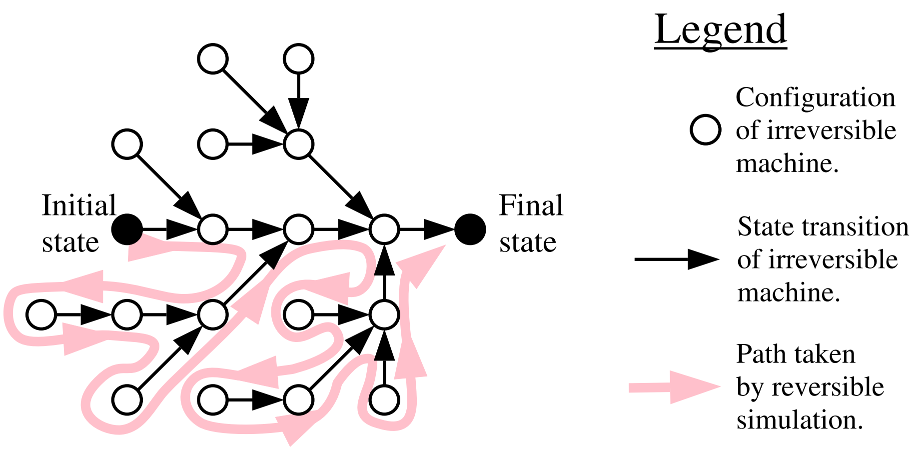

Figure 3.3: Illustration of an Euler tour of an irreversible machine's computation tree. Although the tree has branches, the Euler tour is itself both forward- and reverse-deterministic, and so can be traversed in purely reversible fashion, using no more space than is needed to keep track of the current irreversible machine configuration [98].

---

analogous to using the "right-hand rule" technique (move forward while keeping your right hand on the wall) to find the exit of a planar non-cyclical maze. The search for the final state is kept finite, and the space usage is kept small, by cutting off exploration whenever the configuration size exceeds some limit. Unfortunately, the size of the pruned tree, and thus the time required for the search, is still, in the worst case, exponential in the space bound.

Lange *et al.* originally thought that a limit on the size of the final state was required to be known in advance of the computation in order to guarantee finding the final state, but after seeing a draft of their paper, I pointed out to them (in personal discussions) that in fact, one could determine the appropriate limit dynamically by simply traversing repeatedly around and around the tree, advancing to a successively higher size limit each time the initial state is re-encountered, until the size limit is made large enough that the final state is found. This approach does not increase the worst-case asymptotic run-time, because that time is dominated anyway by the final traversal around the tree, due to the exponential nature of the worst-case branching.

As with Bennett's techniques, the Lange-McKenzie-Tapp technique was defined explicitly only in terms of Turing machines, but it is easily generalized to many different models of computation.

The above time and space complexity results for reversible simulation (§3.3.3 & §3.3.5) are very interesting in themselves, but to our knowledge, no one has yet directly addressed the question of whether a single reversible simulation can run in linear time like Bennett's 1973 technique *and* in linear space like the new Lange *et al.* technique. Li and Vitányi's analysis [105] of Bennett's 1989 algorithm [19] leads to our proof in §3.4 that if such an ideal simulation exists, it would not relativize to oracles, or work in cases where the space bound is much less than the input length.

3.3.6 Miscellaneous developments

Here, we mention in passing a few more miscellaneous developments in reversible computing theory, but we do not go into them in detail.

Coppersmith and Grossman (1975, [37]) proved a result in group theory which implies that reversible boolean circuits only 1 bit wider than a fixed-length input can compute arbitrary boolean functions of that input. (Thanks to Alain Tapp for bringing this paper to our attention.)

Toffoli (1977, [160]) showed that reversible cellular automata can simulate irreversible ones in linear time using an extra spatial dimension. Fredkin and Toffoli also developed much reversible circuit theory (1980–1982, [161, 162, 74]).

As we already mentioned in §3.3.2.2, Pin (1987 [136]) showed that reversible finite automata (defined in a certain way) cannot decide all regular languages.

3.4 Reversible vs. irreversible space-time complexity

In this section we prove that reversible machine models require higher asymptotic complexity on some problems than corresponding irreversible models, if a certain new reversible black-box operation is made available to both models. Thus, no *completely* general technique can exist for simulating irreversible machines on reversible ones with no asymptotic overhead.

However, the new primitive operation that we defined in order to make this proof go through is not itself physically realistic. The operation implements a computable function, but the operation is modeled as taking constant ($\mathbf{\Theta}(1)$) time to perform independent of the size of its input, which violates physical locality (ref. §2.1) and the asymptotically very large number of steps that it would take to compute the operation using the algorithm that corresponds directly to the operation's definition.

Therefore, technically, even given our proof, it is still an open question whether a perfect simulation technique might still exist that works in the case of reversible machines simulating irreversible machines that are composed only of primitives that are physically realistic in the sense of obeying locality. However, if one wishes to progress to *complete* physical realism, then irreversible machines are themselves already reversible at the micro-level (§2.5), and therefore are efficiently implementable on reversible machines, as we will see in ch. 6.

Nevertheless, we conjecture that if the constraint of physical reversibility is ignored, then reversible machines are strictly less efficient on some problems than irreversible machines, even if the machines are constrained to be physically realistic in all other respects. If this conjecture is true, then in combination with our results of ch. 6, it would follow that the constraint of physical reversibility is not independent of other physical constraints from a computational complexity perspective, and that it *must* be taken into account in order to have a realistic physical model of computational complexity, as we will discuss in ch. 5.

If our conjecture were false, and irreversible models can be simulated with no overhead on reversible machines, then one would not necessarily have to explicitly incorporate reversibility in a model of computation in order for it to qualify as an accurate model for predicting problem complexity, such as we advocate in ch. 5. But as a matter of opinion, we consider that possibility *a priori* to be very unlikely.

In this section, we will prove our results in both oracle-relativized and non-oracle forms for serial (uniprocessor) machines. The oracle results cover a large family of possible asymptotic bounds on the joint space and time requirements of machines. For all bounding functions within this family, we show that there exist an oracle and a language such that the language is decidable within the given bounds by serial machines that can query the oracle only if the machines are *ir*reversible. This result is non-trivial (compared to Pin's, for example) because the individual oracle calls are themselves reversible and easy to undo.

A similar result, not involving an oracle, covers cases where the space bound is much smaller than the length of the randomly (and reversibly) accessible input. Corollaries to both the oracle and non-oracle results give loose lower bounds on the amount of extra space required for a reversible machine to decide the language within the time bounds.

Another contribution of our proof is to illustrate ways to use incompressibility arguments in analyzing reversible machines. It is conceivable that similar techniques might increase the range of reversible and irreversible space-time complexity classes that we can separate without resorting to the oracle.

Acknowledgment. Some ideas in the proof below originated with M. Josephine Ammer, who was an undergraduate research assistant in our group at the time this work was done. Ms. Ammer also assisted with the writing of the original manuscript [69] from which this section is derived. That manuscript has not yet been formally published, but some version of it may be in the future.

3.4.1 General definitions

Space-time complexity classes. Given any reversible model of computation (*e.g.*, reversible Turing machines), and given any computational space and time bounding functions $\mathsf{S}(n_{\text{in}}), \mathsf{T}(n_{\text{in}})$, we define the *reversible space-time* S, T *complexity class*, abbreviated $\mathbf{RST}(\mathsf{S}, \mathsf{T})$, to be the set of languages that are accepted by reversible machines that take worst-case space of $\mathcal{O}(\mathsf{S}(n_{\text{in}}))$ memory bits and worst-case time $\mathcal{O}(\mathsf{T}(n_{\text{in}}))$ ticks, where n_{in} is the length of the input. Similarly, we define the unrestricted *space-time* S, T *complexity class*, abbreviated $\mathbf{ST}(\mathsf{S}, \mathsf{T})$, 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 in-degree of the transition graph. For oracle-relativized complexity classes, we use the notation C^O, as is standard in complexity theory, to indicate the class of problems that can be solved by the machines that define the class C if they are allowed to query oracle O.

We want to know whether $\mathbf{RST}(\mathsf{S}, \mathsf{T}) \stackrel{?}{=} \mathbf{ST}(\mathsf{S}, \mathsf{T})$, for all S, T, in normal sorts of serial computational models such as multi-tape Turing machines or RAM machines.

Unfortunately, we have found this question, in its purest form, very difficult to definitively 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 proven to require extra space or time to recognize reversibly in ordinary machine models. The difficulty is in constructing a proof that rules out all reversible algorithms, no matter how subtle or clever.

But is the $\mathbf{RST}(\mathsf{S}, \mathsf{T}) \stackrel{?}{=} \mathbf{ST}(\mathsf{S}, \mathsf{T})$ 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, **P** and **NP**) are equivalent. Traditionally (as in [9]), one way to indicate that such an equivalence might really 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 be proven 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 any 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. Reputedly, this rules out a large number of proof techniques from recursion theory, and means that resolving the question will be more difficult.

In this section we will demonstrate, for any given S, T in a large class, an oracle

A relative to which we prove $\mathbf{RST}(\mathsf{S},\mathsf{T})^A \neq \mathbf{ST}(\mathsf{S},\mathsf{T})^A$, for the case of serial machine models with a certain kind of oracle interface. For these same S,T we have not yet found an alternative oracle B for which $\mathbf{RST}(\mathsf{S},\mathsf{T})^B = \mathbf{ST}(\mathsf{S},\mathsf{T})^B$. It may be that none exists, but this is uncertain.

Reversible oracle interface. First, we define an oracle interface that allows a reversible machine to call an oracle. Ordinarily, oracle queries are irreversible, and thus impossible in reversible machines. For example, a bit of the oracle's answer cannot just overwrite some storage location, because regardless of whether the location contained `0` or `1` before the oracle call, after the call it would contain the oracle's answer. The resulting configuration would thus have two predecessors, and the machine would be irreversible.

Our reversible oracle-calling protocol is as follows. Machines will have reversible read and 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 new contents, according to some fixed invertible mapping $A : \mathcal{C} \to \mathcal{C}$ over the space $\mathcal{C}$ of possible tape contents. The function A is called a *permutation oracle*. Further, if A is its own inverse, $A = A^{-1}$, it will be called *self-reversible*. Presented more formally:

Definition 3.1. A *permutation oracle* A is an invertible (bijective) function $A : \mathcal{C} \to \mathcal{C}$, where $\mathcal{C}$ is the space of possible contents of a semi-infinite *oracle tape*.

Definition 3.2. A *self-reversible (permutation) oracle* is a permutation oracle A such that $A = A^{-1}$.

In the below, we will deal only with self-reversible oracles. Self-reversibility ensures that machines can easily undo oracle operations, just as they can easily undo their own internal reversible primitives. If oracle calls were hard to undo, then the oracle model would be unlikely to teach us anything meaningful about ordinary machines.

ST-constructibility. In order for our proof to go through, we will need to restrict our attention to space and time functions $\mathsf{S}(n_{\text{in}}), \mathsf{T}(n_{\text{in}})$ which are *ST-constructible*, meaning that given any input of length n_{in}, an irreversible machine can construct binary representations of the numbers $\mathsf{S}(n_{\text{in}})$ and $\mathsf{T}(n_{\text{in}})$ using only space $\mathcal{O}(\mathsf{S}(n_{\text{in}}))$ and time $\mathcal{O}(\mathsf{T}(n_{\text{in}}))$. We state here without proof that many reasonable pairs of functions are indeed ST-constructible. For example, $\mathsf{S} = n_{\text{in}}^2$, $\mathsf{T} = n_{\text{in}}^3$ can both be computed in time $\mathcal{O}(\log^2 n_{\text{in}})$ plus $\mathcal{O}(n_{\text{in}})$ to count the input bits, and space $\mathcal{O}(\log n_{\text{in}})$ plus $\mathcal{O}(n_{\text{in}})$ if we include the input.

Next, we need some basic definitions to support the notion of incompressibility that will be crucial to the proof of our theorem. The following definition and lemma follow

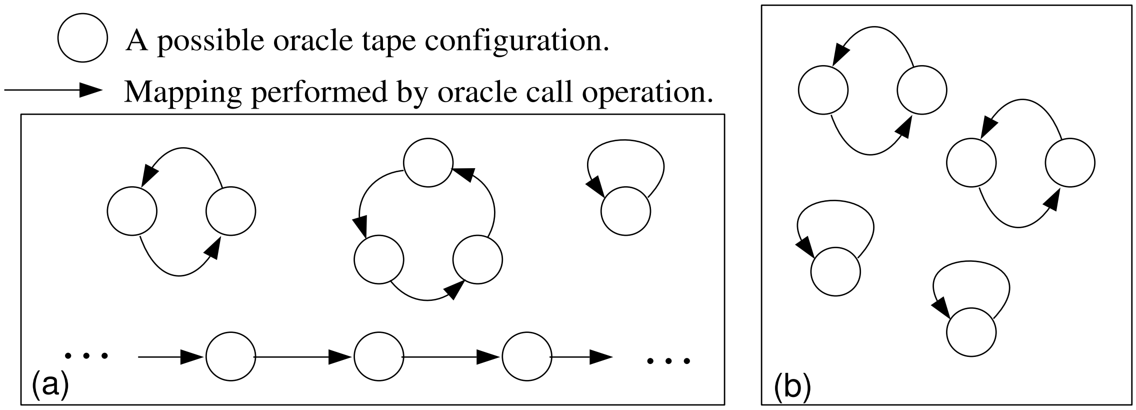

Figure 3.4: Illustration of the structure of (a) a permutation oracle, and (b) a self-reversible permutation oracle.

In either case, 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, $A = A^{-1}$.

the spirit of the discussions of incompressibility in Li and Vitányi's excellent book on Kolmogorov complexity [104].

Description systems and compressibility. A *description system* s is any function $s\colon \{0,1\}^* \to \{0,1\}^*$ from bit-strings to bit-strings, that is, from *descriptions* to the bit-strings they describe. We say that a bit-string d *describes* bit-string x in description system s if $s(d) = x$. We say that a bit-string x is *compressible* in description system s if there is a shorter bit-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. □

In our main proof, we will be selecting incompressible strings from a series of computable description systems.

Notational conventions. In the following, we will often abbreviate the space and time function values $\mathsf{S}(n_{\text{in}})$ and $\mathsf{T}(n_{\text{in}})$ by just S and T, respectively; likewise for other functions of n_{in}. For comparing orders of growth, we will use both the standard $\boldsymbol{\Theta}$, $\mathcal{O}$, $\boldsymbol{\Omega}$, $\mathbf{o}$, $\boldsymbol{\omega}$ notations, and our mnemonic custom $\sim$, $\precsim$, $\succsim$, $\prec$, $\succ$ notation, defined in table F.4 on p. 409.

3.4.2 Oracle results

Theorem 3.1. Relative separation of reversible and irreversible space-time complexity classes. Let S, T be any two non-decreasing functions over the non-negative integers. Then the following are true:

(a) If $\mathsf{S} \succsim \mathsf{T}$ or $\mathsf{T} \succsim 2^{\mathsf{S}}$, then $\mathbf{RST}(\mathsf{S},\mathsf{T})^O = \mathbf{ST}(\mathsf{S},\mathsf{T})^O$ for any self-reversible oracle O.

(b) If $\mathsf{S} \prec \mathsf{T} \prec 2^{\mathsf{S}}$, and if S, T are ST-constructible, then there exists a computable, self-reversible oracle A such that $\mathbf{RST}(\mathsf{S},\mathsf{T})^A \neq \mathbf{ST}(\mathsf{S},\mathsf{T})^A$.

Proof.

Part (a). (Cases $\mathsf{S} \succsim \mathsf{T}$ and $\mathsf{T} \succsim 2^{\mathsf{S}}$.) First, if $\mathsf{S} \succ \mathsf{T}$, then obviously we have both $\mathbf{RST}(\mathsf{S},\mathsf{T})^O = \mathbf{RST}(\mathsf{T},\mathsf{T})^O$ and $\mathbf{ST}(\mathsf{S},\mathsf{T})^O = \mathbf{ST}(\mathsf{T},\mathsf{T})^O$ simply because in time T no more than $\mathsf{S} \sim \mathsf{T}$ memory cells can be accessed on a machine that performs $\boldsymbol{\Theta}(1)$

operations per time step. Similarly, if $\mathsf{T} \succ 2^{\mathsf{S}}$, then $\mathbf{RST}(\mathsf{S},\mathsf{T})^O = \mathbf{RST}(\mathsf{S},2^{\mathsf{S}})^O$ and $\mathbf{ST}(\mathsf{S},\mathsf{T})^O = \mathbf{ST}(\mathsf{S},2^{\mathsf{S}})^O$, because no computation using only S bits of memory can run for more than 2^{S} steps without repeating. So part (a) reduces to proving $\mathbf{RST}(\mathsf{S},\mathsf{T})^O = \mathbf{ST}(\mathsf{S},\mathsf{T})^O$ only for the case where $\mathsf{S} \sim \mathsf{T}$ or $\mathsf{T} \sim 2^{\mathsf{S}}$.

From here, the result follows due to the existing relativizable simulations. When $\mathsf{S} \sim \mathsf{T}$, Bennett's simple reversible simulation technique [16] can be applied because it takes time $\mathcal{O}(\mathsf{T})$ and space $\mathcal{O}(\mathsf{T})$. Similarly, when $\mathsf{T} \sim 2^{\mathsf{S}}$ the simulation of Lange *et al.* [98] can be used because it takes time $\mathcal{O}(2^{\mathsf{S}})$ and space $\mathcal{O}(\mathsf{S})$. Both techniques can be easily seen to relativize to any self-reversible oracle O. Thus, in both cases, any irreversible machine can be simulated reversibly in $\mathcal{O}(\mathsf{T})$ and space $\mathcal{O}(\mathsf{S})$, and therefore $\mathbf{RST}(\mathsf{S},\mathsf{T})^O = \mathbf{ST}(\mathsf{S},\mathsf{T})^O$.

Part (b). (Case $\mathsf{S} \prec \mathsf{T} \prec 2^{\mathsf{S}}$.) **Outline:** We will construct A to be a permutation oracle that can be interpreted as specifying 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 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 will prove this by equating the machine's operation with the "pebble game" for which Li and Vitányi [105] have already proven lower bounds, and by showing 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.

For the formal proof of part (b), we need some special definitions.

Definition 3.3. A *graph oracle* is a self-reversible permutation oracle with the following property: There exists a partial function $f\colon \{\mathtt{0},\mathtt{1}\}^* \to \{\mathtt{0},\mathtt{1}\}^*$, called a *successor function*, such that for any bit string (node) $b \in \{\mathtt{0},\mathtt{1}\}^*$ for which f is defined, the oracle's permutation function maps the tape contents b to the tape contents $b\mathtt{\#}f(b)$, and also maps $b\mathtt{\#}f(b)$ back to b, where $\mathtt{\#}$ is a special separator character in the oracle tape alphabet. For all tape contents x not of either of these forms, the oracle's permutation function maps them to themselves. See fig. 3.5.

Given that we will be working only with graph oracles, we can now specify an oracle by specifying just the successor function f that it embodies. But before we actually construct the special oracle A that proves our theorem, let us define, relative to A, the language that we claim separates $\mathbf{RST}(\mathsf{S},\mathsf{T})^A$ from $\mathbf{ST}(\mathsf{S},\mathsf{T})^A$.

Definition 3.4. Given two ST-constructible functions $\mathsf{S}(n)$, $\mathsf{T}(n)$, and graph oracle

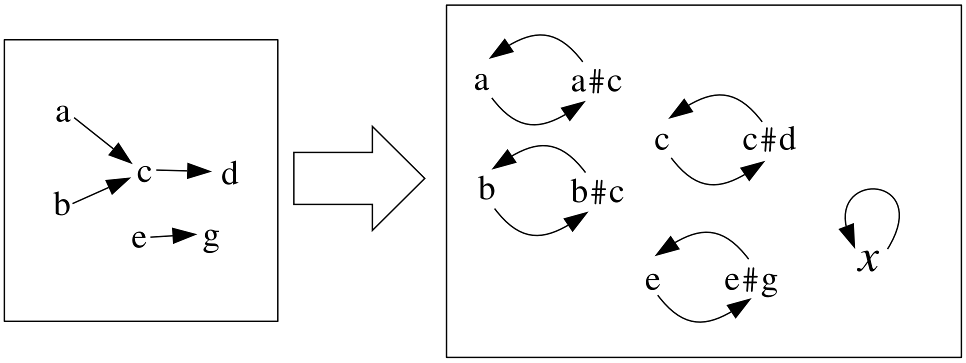

Figure 3.5: Encoding outdegree-1 directed graphs in self-reversible permutation oracles. Letters stand for nodes represented as bit-strings, except for x which represents any other bit-string not explicitly shown. The # is a special separator character.

On the left, we show an example of an outdegree-1 directed graph with bit-string nodes abbreviated 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. This f is a partial function; *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 self-reversible 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 (including those for terminal nodes of the graph) is simply mapped to itself. In this way the permutation oracle allows easily and reversibly looking up a node's successor, or uncomputing a node's successor given the node and its successor. But finding a node's predecessor(s), given just the node itself, is designed to be hard. Thus the oracle call resembles the reversible computation of a "one-way" invertible function that is easy to compute, but whose inverse is difficult to compute.

A with successor 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 $n = |w|$, compute $\mathsf{S} = \mathsf{S}(n), \mathsf{T} = \mathsf{T}(n)$.
  Let bit-string $b = \mathtt{0}^{\mathsf{S}}$.
  Repeat the following, $t = \lfloor \mathsf{T}/\mathsf{S} \rfloor$ times:
    Write b on oracle tape, and call oracle.
    If result is of the form $b\#c$, with c a bit-string,
      assign $b \leftarrow c$ (note $c = f(b)$),
    else, quit loop early.
  Accept iff $b[0] = \mathtt{1}$.

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

We will be explicitly constructing the successor function f so that it always returns a string of the same length as its input. Given the corresponding oracle, the above algorithm requires only space $\mathcal{O}(\mathsf{S})$ and time $\mathcal{O}(\mathsf{T})$ on on irreversible machine in any standard serial model of computation. (Recall that S, T are ST-constructible.) Therefore the language $L(A)$ will be in the class $\mathbf{ST}(\mathsf{S}, \mathsf{T})^A$.

Now, we will specify how to construct f so that the language $L(A)$ will not be computable by any reversible machine that takes space $\mathcal{O}(\mathsf{S})$ and time $\mathcal{O}(\mathsf{T})$. The way we will do this is to make each of the node identifiers be a different incompressible string. Intuition suggests that the only way to decide $L(A)$ is to actually follow the entire chain of nodes, to see what the final one is. But having obtained a node's successor, the reversible machine cannot easily get rid of its incompressible records of the prior nodes. The graph oracle provides no convenient way to compute f^{-1} and find a node's predecessor, even if the successor function f happens to be invertible. Thus the reversible machine will tend to accumulate records of previous nodes, of size $\mathsf{S}(n_{\text{in}})$ each, and thus, for sufficiently long enough chains, it will take more than a constant factor times $\mathsf{S}(n_{\text{in}})$ space. The reversible machine could conceivably find and uncompute 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 iterating a one-way function, *i.e.* an invertible function whose inverse much is harder to compute than the function itself (*e.g.*, MD5). Public-key cryptography depends on the (unproven, but empirically reasonable) assumption that some functions are one-way. The same assumption might allow us to show that $\mathbf{RST}(\mathsf{S}, \mathsf{T}) \neq \mathbf{ST}(\mathsf{S}, \mathsf{T})$ without an oracle, by using a one-way function instead.

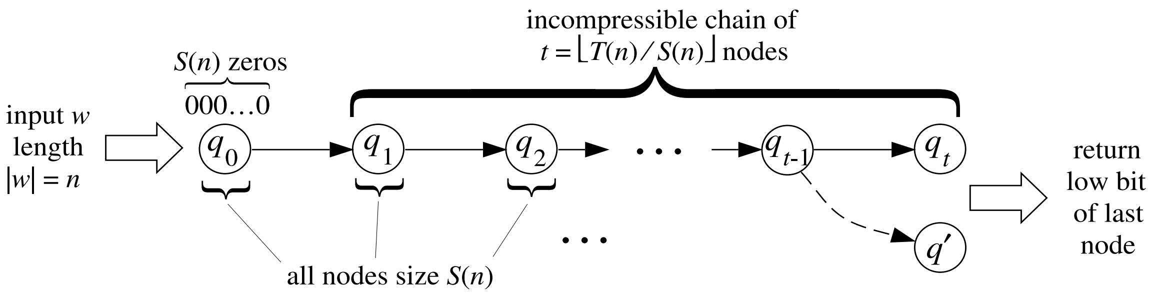

Figure 3.6: The problem graph defined by our oracle for inputs of size n. The "correct answer" is just the first bit of the final node q_t. If the reversible machine M_i that we are trying to foil happens to get the right answer, but never asks for the successor of node q_{t-1}, we redefine q_{t-1}'s successor to be a new node q' having a different initial bit.

Oracle construction. We now construct a particular oracle A and prove that $L(A) \notin \mathbf{RST}(\mathsf{S}, \mathsf{T})^A$.

First, fix some standard enumeration of all reversible oracle-querying machines. The enumeration is possible because reversible Turing machines, for example, can be characterized by local syntactic restrictions on their transition function, as in Lange *et al.*, so we can enumerate all machines and pick out the reversible ones. Let $(M_1, c_1), (M_2, c_2), \ldots$ be this enumeration dovetailed together with an enumeration of the positive integers. If a given machine always runs in space $\mathcal{O}(\mathsf{S})$ and time $\mathcal{O}(\mathsf{T})$ then it will eventually appear in the enumeration paired with a large enough c_i so that the machine M_i takes space less than $c_i + c_i S(n_{\text{in}})$ and time less than $c_i + c_i \mathsf{T}(n_{\text{in}})$ for any input length n_{in}.

We will construct the oracle A so that each machine M_i will fail to decide $L(A)$ within these bounds. When considering M_i, $f(q)$ will have already been specified 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, choose n_i (henceforth called n), the input length for which our oracle definition will foil M_i, to be such that $\mathsf{S}(n)$ is greater than the maximum length z of any of those earlier machines' oracle queries. Some other lower bounds on the size of n will be mentioned as we go along.

Later we will specify 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 $\mathsf{S}(n)$). The description system will be a total computable function, *i.e.*, there is an algorithm that computes $s_i(d)$ for any d and always halts. We will use this description system to define $f(q)$ for bit-strings of length $\mathsf{S}(n)$, as follows:

Let x be a bit-string of length $\mathsf{T}(n)$ that is incompressible in description system s_i (to be defined as we go along). This x will be used as the sequence of size-$\mathsf{S}(n)$

node identifiers that will define our graph for inputs of size n.

Break x up into a sequence of $t(n) \equiv \lfloor \mathsf{T}/\mathsf{S} \rfloor$ bit-strings of length $\mathsf{S}(n)$ each; call these our graph 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 $q_j = q_k$, $j < k$, 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 j and k would take $\mathcal{O}(\log(\mathsf{T}/\mathsf{S}))$ space, which is $\mathcal{O}(\log \mathsf{T})$ space, which is $o(\mathsf{S})$ space, whereas we are saving $\mathsf{S}(n)$ 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 would be less than $\mathsf{T}(n)$. With x being incompressible in 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 $q_0 = \mathtt{0}^{\mathsf{S}}$ (a string of S $\mathtt{0}$-bits). Provisionally, set $f(q_{j-1}) = q_j$ for all $1 \le j \le t$. 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 ensure as well.) Given these assignments, all strings of length n are in the language $L(A)$ if and only if $q_t[0] = \mathtt{1}$, due to the earlier definition of $L(A)$. (Definition 3.2.)

Suppose temporarily that our oracle definition was completed by letting f remain undefined over all strings w for which we have not yet specified $f(w)$. (*I.e.*, let $A(w) = w$ for these strings.) Under that assumption, simulate M_i's behavior on the input $\mathtt{0}^n$. If M_i runs for more than $c_i + c_i T$ steps, then it takes too much time, and we are through addressing it. Otherwise, M_i either accepts ($\mathtt{1}$) or rejects ($\mathtt{0}$). If this answer is different from $q_t[0]$, then M_i already fails to accept the language $L(A)$, and we are through with it.

Alternatively, suppose M_i's answer is correct with the given q_j's and it halts within $c_i + c_i T$ steps. But now suppose that M_i never asked any query dependent on $f(q_{t-1})$ during its run on input $\mathtt{0}^n$. That is, suppose M_i never asked either query q_{t-1} or query $q_{t-1}\mathtt{\#}q_t$. 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 whose successor was never requested in any query by M_i, and whose first bit is the opposite of M_i's answer. To ensure such strings exist, note there are $\frac{1}{2}2^{\mathsf{S}}$ bit-strings of length S having the desired initial bit, but M_i can make at most $c_i + c_i T$ queries since that is its running time. We know $\mathsf{T} \prec 2^{\mathsf{S}}$, so with sufficiently large n, $\frac{1}{2}2^{\mathsf{S}} > c_i + c_i\mathsf{T}$, and we can find our node q'. Now, given q', we change $f(q_{t-1})$ to be q'. This cannot possibly affect the behavior of M_i since it never asked about $f(q_{t-1})$. But the correct answer is changed to the first bit of q', the new node number t in the chain. Thus with this new partial specification of f, M_i fails to correctly decide $L(A)$, and we can go on to foil other machines.

Finally, suppose M_i does ask query q_{t-1}. We now show how to complete the definition of our description system s_i, source of our incompressible x, so that if M_i does ask query q_{t-1}, then it must at some point take more than $c_i + c_i S$ space.

To do this, we show that M_i can always be interpreted as following the rules of Bennett's reversible "pebble game," introduced in [19] and analyzed by Li and Vitányi in [105].

Pebble game rules. The game is played on a linear list of nodes, which we will identify with query strings $q_1, \ldots, q_t$. At any time during the game some set of nodes is *pebbled.* 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 any node q_j for which the previous node q_{j-1} is pebbled. Only one such move may be made at a time.

The idea of the pebbled set is that we will make it correspond to the set of nodes that is currently "stored in memory" by M_i. Pebbling or unpebbling node q_j will require querying the oracle with query string q_{j-1} or $q_{j-1}\texttt{\#}q_j$, respectively. 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 the oracle can be constructed to foil it trivially.

Li and Vitányi's analysis of the pebble game [105] 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. We will show that our machine M_i and its space usage can be modeled using the pebble game, so that for some sufficiently large n, the space required to store the necessary number of pebbled nodes will exceed M_i's allowable storage capacity $c_i + c_i\mathsf{S}$.

For the oracle A as defined so far, consider the complete sequence of configurations of M_i given input $\mathtt{0}^n$, notated $C_1, C_2, \ldots, C_{\mathsf{T}'}$, where $\mathsf{T}' \leq c_i + c_i\mathsf{T}$ is M_i's total running time, in terms of the number of primitive operations (including oracle calls) performed.

Now, for any time point τ, $1 \leq \tau \leq \mathsf{T}'$, and for any node q_j in the chain of nodes $q_1, \ldots, q_t$, define *the previous query involving* q_j (written $\mathrm{prev}(q_j)$)to mean the most recent oracle query in M_i's history before time τ in which the query string (the one that is present on the oracle tape at the start of the query) is one of $\{q_{j-1}, q_j, q_j\#q_{j+1}, q_{j-1}\#q_j\}$. There may of course be no such query in which case $\mathrm{prev}(q_j)$ does not exist. Similarly define *the next query involving* q_j (written $\mathrm{next}(q_j)$) to mean the most imminent such query in M_i's future after time τ.

Definition 3. Node q_j *is pebbled at time* τ iff at time τ either (a) $\mathrm{prev}(q_j)$ exists and is either (a1) q_{j-1}, (a2) q_j, or (a3) $q_j\texttt{\#}q_{j+1}$, or (b) $\mathrm{next}(q_j)$ exists and is (b1) q_j, (b2) $q_j\texttt{\#}q_{j+1}$, or (b3) $q_{j-1}\texttt{\#}q_j$. (Exception: the final node q_t is only considered pebbled in cases (a1) and (b3).)

Note that this definition implies that q_j is *not* pebbled iff $\mathrm{prev}(q_j) = q_{j-1}\#q_j$ (or

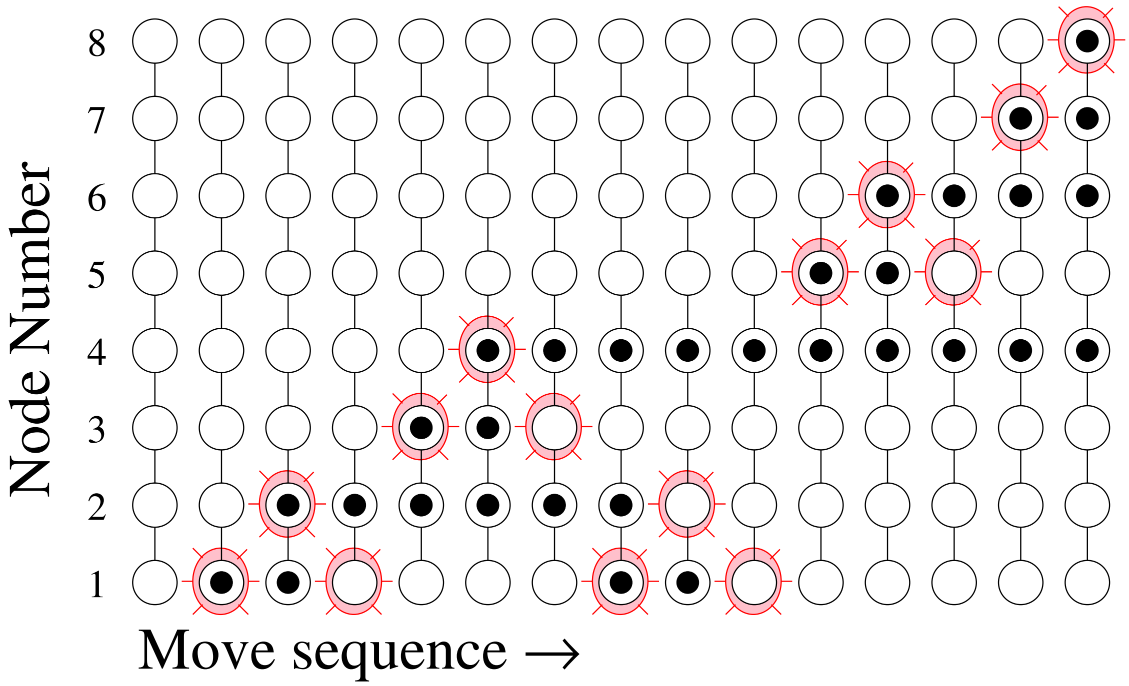

Figure 3.7: Bennett's reversible pebble game strategy. Highlights point out the move made at each step. (Compare with fig. 3.2(a), page 64, rotated 90°.)

A node q_j can be pebbled or unpebbled only if it is node q_1 or if the previous node q_{j-1} is pebbled. The strategy invented by Bennett [19], illustrated here, was shown by Li and Vitányi to be optimal [106] 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 outwards along the chain a constant distance is crucial to our proof.

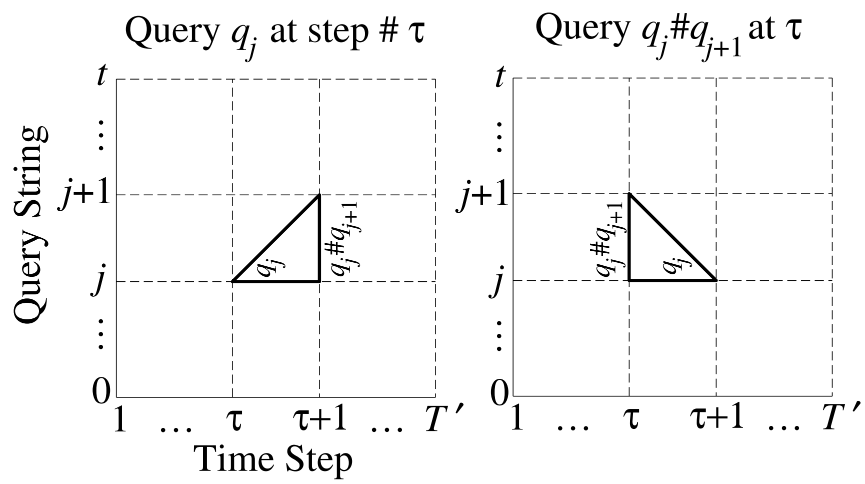

Figure 3.8: Triangle representation of oracle queries.

The shape and direction of the triangle is meant to evoke 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 set of triangles defines the set of pebbled nodes at any time, as illustrated in figure 3.9.

---

nonexistent) and $\text{next}(q_j) = q_{j-1}$.

Figure 3.9 illustrates the intuition behind this definition using the graphical notation introduced in fig. 3.8. This graphical notation is especially nice because it evokes the image of playing the pebble game or running Bennett's algorithm (compare fig. 3.9 with figs. 3.7 and 3.2).

The times at which a node is to be considered "pebbled" during a machine's execution are indicated by the solid horizontal lines on 3.9. These times are determined, according to definition 3 above, solely by the arrangement of triangles (representing oracle queries, see fig. 3.8) 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. Query string 0 is never considered pebbled because it is not considered to be a node.

Let p denote the number of distinct nodes out of $q_1, \ldots, q_t$ that are pebbled at time τ. We now lower bound the size of C_τ, *i.e.* M_i's space usage at time τ.

Lemma 2. *Space to pebble p nodes. $|C_\tau| > \frac{1}{4}pS$.*

Proof. Suppose C_τ were no larger than $\frac{1}{4}pS$ 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_i will be defined to process descriptions of the required form.

First, note that for each node q_j that is pebbled at time τ, that node is pebbled

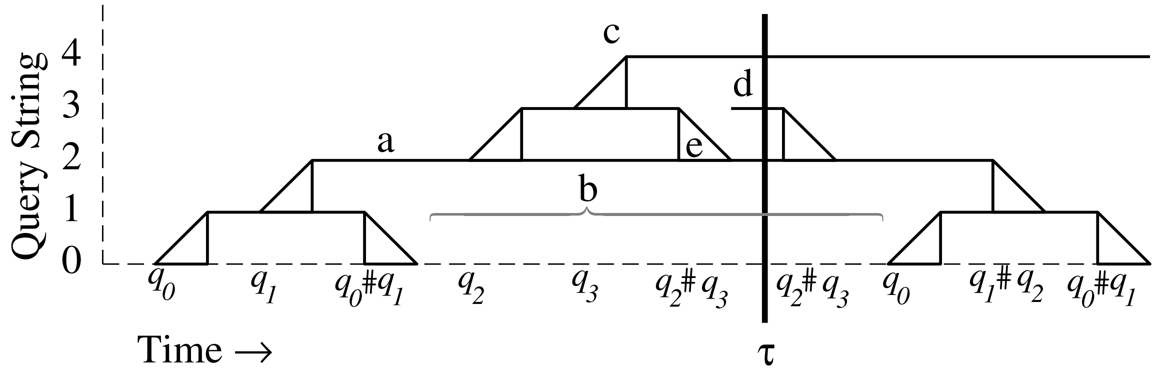

Figure 3.9: Visualizing the definition of the set of pebbled nodes. The times at which a node is pebbled (indicated by solid horizontal lines on the chart) are determined, by definition, solely by the identities and timing of oracle queries and the corresponding arrangement of triangles (see fig. 3.8) 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 encounters 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 [19] optimal pebble game strategy. (Compare with figs. 3.7 and 3.2.)

Node 2 is considered 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 the previous and next queries are $q_0\#q_1$ and q_0 respectively. 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 (q_3). Node 3 is pebbled at time (d) because although the previous query (e) is of the wrong form ($q_2\#q_3$), 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.

either because of the previous query involving q_j, because of the next query involving q_j, or both. Therefore, either at least $\frac{1}{2}p$ nodes are pebbled because of their previous query, or at least $\frac{1}{2}p$ nodes are pebbled because of their next query. Let D be a direction (forwards or backwards) from time τ in which one can find queries causing $h \geq \frac{1}{2}p$ nodes to be pebbled.

We now specify the shorter description d that describes x. It will contain an explicit description of C_τ, which by our assumption is no longer than $\frac{1}{4}p\mathsf{S}$. 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. (Space: $(t-h)\mathsf{S}$.) For each of the h nodes q_j that *are* pebbled because of a query in direction D, the description d will contain the node index j and an integer $\Delta\tau_j$ giving the number of steps from step τ to the time of the query. 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 $\mathcal{O}(\log t) \prec \log \mathsf{T} \prec \mathsf{S}$, and each $\Delta\tau_j$ takes space $\mathcal{O}(\log \mathsf{T}) \prec \mathsf{S}$. The tag is constant size. Thus for sufficiently large n, all h of the $(j, \Delta\tau_j, k_j)$ tuples together take less than $\frac{1}{2}h\mathsf{S}$ space. Total space so far: less than tS. If $t\mathsf{S} < \mathsf{T}$, 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 $\mathsf{T} = |x|$.

We now demonstrate that the description d is sufficient to reconstruct x, and give an algorithm for doing so. The function computed by this algorithm tells how our description system s will handle descriptions of the form outlined above.

The algorithm will work by simulating M_i's operation in direction D starting from configuration C_τ, and reading the identifiers of pebbled nodes from M_i's simulated oracle tape as it proceeds. We can figure out which oracle queries correspond to which nodes by referring to the stored times $\Delta\tau_j$ and tags k_j. Once we have extracted the identifiers of all nodes pebbled in direction D, we print all the nodes out in the proper order.

As an example, refer again to fig. 3.9. In the machine configuration marked 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 known, 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 at least half of the pebbled nodes is what makes our incompressibility argument work. The algorithm is described and verified in more detail in the appendix.

Given d, the algorithm produces x, and with n chosen large enough, the length of the description will be smaller than x itself, contradicting the assumption of x's incompressibility relative to s. Therefore for these sufficiently large n, all configurations in which p nodes are pebbled must actually be larger than $\frac{1}{4}p\mathsf{S}$. This completes the proof of lemma 2. ■

Now, given the definition of the set of pebbled nodes from earlier (defn. 3), it is easy to see how M_i's execution history can be interpreted as the playing of a pebble game. Whenever M_i performs a query q_j and node q_{j+1} was not already pebbled immediately prior to this query, we 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. For example, in fig. 3.9, query (e) (the first occurrence of $q_2\#q_3$ is not considered a move in the pebble game, since it doesn't change the set of pebbled nodes as defined by definition 3.

It is obvious that under the above interpretation, all moves must obey the main pebble game rule, *i.e.* that the pebbled status of node q_j can only change if $j = 1$ or if node q_{j-1} is pebbled during the change. The move is a query, and the presence of the query means the node q_{j-1} is pebbled both before and after the query, by definition 3, unless $j = 1$; q_0 is not considered to be a node.

To show that no nodes are *initially* pebbled 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 in lemma 2, 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 the 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.

Thus M_i exactly obeys all the rules of the Bennett pebble game. Now, Li and Vitányi have shown [105] that any strategy for the pebble game 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 $k \geq 4(c_i + 1)$, and also so that $S \geq c_i$. Then at times τ when p is maximum, M_i's space usage is $|C_\tau| > \frac{1}{4}p\mathsf{S} > \frac{1}{4}kS \geq (c_i + 1)\mathsf{S} \geq c_i + c_i S$.

The above discussion establishes that machine M_i takes more than space $c_i + c_i\mathsf{S}$ if it correctly decides membership in $L(A)$ for inputs of length $n_i = n$ and takes only time $c_i + c_i\mathsf{T}$, so long as the oracle A is consistent with the definition above. Since machine M_i's behavior on the input $\mathtt{0}^n$ only depends on the values of the successor 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 and constant c_i in the entire infinite enumeration, the machine will either get the wrong answer or take more than time $c_i + c_i\mathsf{T}$ or space $c_i + c_i\mathsf{S}$ on input 0^{n_i}. Thus, no reversible machine can actually decide $L(A)$ in time $\mathcal{O}(\mathsf{T})$ and space $\mathcal{O}(\mathsf{S})$, and so $L(A) \notin \mathbf{RST}(\mathsf{S},\mathsf{T})^A$.

Note that this entire oracle construction, as described, is computable. If we are given procedures for computing $\mathsf{S}(n)$ and $\mathsf{T}(n)$, we can write an effective procedure that, given any finite oracle query, returns A's response to the query. The details of the oracle construction algorithm follow directly from the above definition of A, but would be too tedious to present here.

This concludes our proof of theorem 3.1.

□

Note that in the above proof, we used the fact that the number of pebbles required to get to the final node grows larger than any constant as n increases. But the actual rate of growth can be used as well, to give us an interesting lower bound.

Corollary 1. *Lower bound on space for linear-time relativizable reversible simulation of irreversible machines.* For all ST-constructible S,T and computable S' such that $\mathsf{S} \prec \mathsf{T} \prec 2^{\mathsf{S}}$ and $\mathsf{S}' \prec \mathsf{S}\log(\mathsf{T}/\mathsf{S})$, there exists a computable, self-reversible oracle A such that $\mathbf{RST}(\mathsf{S}',\mathsf{T})^A \neq \mathbf{ST}(\mathsf{S},\mathsf{T})^A$.

Proof. Essentially the same as for Theorem 1 part (b), but with S' in place of S in appropriate places. In the last part of the proof, M_i is shown to take more than $c_i + c_i\mathsf{S}'$ space by using Lemma 2 together with the fact that $p > \lfloor \lg\lfloor \mathsf{T}/\mathsf{S} \rfloor \rfloor$ pebbles are required to reach the final node. □

This result implies that any general linear-time simulation of irreversible machines by reversible ones that is relativizable with respect to all self-reversible oracles must take space $\mathbf{\Omega}(\mathsf{S}\log(\mathsf{T}/\mathsf{S}))$.

The most space-efficient linear-time reversible simulation technique that is currently known was provided by Bennett ([19], p. 770), and analyzed by Levine and Sherman [103] to take space $\mathcal{O}(\mathsf{S}(\mathsf{T}/\mathsf{S})^{1/(0.58\lg(\mathsf{T}/\mathsf{S}))})$. Bennett's simulation can be easily seen to work with all self-reversible oracles, so it gives a relativizable upper bound on space. There is a gap between it and our lower bound, 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 *linear* time pebble game strategies would allow us to expand our lower bound on space, hopefully to converge with the existing upper bound.

3.4.3 Non-relativized separation

We now explain how the same type of proof can be applied to show a non-relativized separation of $\mathbf{RST}(\mathsf{S},\mathsf{T})$ and $\mathbf{ST}(\mathsf{S},\mathsf{T})$ in certain cases, when inputs are accessed in a specialized way that is similar to an oracle query.

Input framework. Machine inputs will be provided in the form of a random-access read-only memory I, which may consist of 2^b b-bit words for any integer $b \geq 0$. The length of this input may be considered to be $n(b) = b2^b$ bits; let $b(n)$ be the inverse of this function. The machine will have a special *input access tape* which is unbounded in one direction, initially empty, and is used for reversibly accessing the input ROM via the following special operations.

Get input size. If the input access tape is empty before this operation, after the operation it will contain b written as a binary string. If the tape contains b before the operation, afterwards it will be empty. In all other circumstances, the query is a no-op.

Access input word. If the input access tape contains a binary string a of length b before the operation, afterwards it will contain the pair $(a, I[a])$ where $I[a]$ is a length-b binary string giving the contents of the input word located at address a. If the tape contains this pair before the operation, afterwards it will contain just a. Otherwise, nothing happens.

Theorem 2. *Non-relativized separation of reversible and irreversible spacetime.* For models using the above input framework, and for $\mathsf{S}(n) = b(n)$ and any ST-constructible $\mathsf{T}(n)$ such that $\mathsf{S} \prec \mathsf{T} \prec 2^{\mathsf{S}}$, $\mathbf{RST}(\mathsf{S},\mathsf{T}) \neq \mathbf{ST}(\mathsf{S},\mathsf{T})$.

Proof. (Sketch following proof of theorem 1.) For input I of length $n = b2^b$, define result bit $r(I)$ to be the first bit in the b-bit string given by

$$\underbrace{I[I[\ldots I[}_{\lfloor \mathsf{T}/\mathsf{S} \rfloor} \mathtt{0}^b] \ldots]].$$

Let language $L = \{I : r(I) = \mathtt{1}\}$. $L \in \mathsf{ST}$ because an irreversible machine can simply follow the chain of $\lfloor \mathsf{T}/\mathsf{S} \rfloor$ pointers from address $\mathtt{0}^b$, using space $\mathcal{O}(\mathsf{S})$ (not counting the input) and time $\mathcal{O}(\mathsf{T})$.

Assume there is a reversible machine M that decides L in $c + c\mathsf{S}$ space and $c + c\mathsf{T}$ time for some c. Let b be sufficiently large for the proof below to work. Let s be a certain description system to be defined. Let $t = \lfloor \mathsf{T}/\mathsf{S} \rfloor$. Let x be a length-$t\mathsf{S}$ string incompressible in s. Let $w_1 \ldots w_t = x$ where all w_i are size b. Restrict s so that all the words w_i must be different from each other and from 0^b. Let I be an input of length $n = b2^b$ such that $I[\mathtt{0}^b] = w_1$, and $I[w_i] = w_{i+1}$ for $1 \leq i < t$, and $I[a] = \mathtt{0}^b$ for every other address a. M must at some time access $I[w_{t-1}]$ because otherwise we could change the first bit of $I[w_{t-1}]$ to be the opposite of whatever M's answer is, and M would give the wrong answer. Assign a set of pebbled nodes to each configuration of M's execution on input I like in the oracle proof, except that this time, input access operations take the place of oracle calls. Show, as in lemma 2, that the size of a configuration is at least $\frac{1}{4}p\mathsf{S}$ where p is the number of pebbled nodes, by defining s to allow descriptions that are interpreted by simulating M and reading pebbled nodes from the input access tape. As before, the machine must therefore take space $\mathbf{\Omega}(\mathsf{S}\log(\mathsf{T}/\mathsf{S}))$ which for sufficiently large n contradicts our assumption that the space is bounded by $c + c\mathsf{S}$. Thus $L \notin \mathbf{RST}(\mathsf{S},\mathsf{T})$. ∎

Corollary 2. *Non-relativized lower bound on space for linear-time reversible simulations.* For $\mathsf{S} = b(n)$, computable $\mathsf{S}' \prec \mathsf{S}\log(\mathsf{T}/\mathsf{S})$, and ST-constructible $\mathsf{T}(n)$ such that $\mathsf{S} \prec \mathsf{T} \prec 2^{\mathsf{S}}$, $\mathbf{RST}(\mathsf{S}',\mathsf{T}) \neq \mathbf{ST}(\mathsf{S},\mathsf{T})$.

Proof. As in corollary 1 but with theorem 2. □

Such a T exists because b can be found in space and time $\mathcal{O}(\log b)$ using the "get input size" operation, after which $\mathsf{T} = b^2$, for example, can be found in space $\mathcal{O}(\log b)$ and time $\mathcal{O}(\log^2 b)$. Thus, any reversible machine that simulates irreversible ones without slowdown takes $\mathbf{\Omega}(\mathsf{S}\log(\mathsf{T}/\mathsf{S}))$ space in some cases.

3.4.4 Decompression algorithm

It is probably not obvious to the reader that the algorithm that we briefly mentioned in the proof of lemma 2 in §3.4.2 can be made to work properly. In this section we give the complete algorithm and explain why it works.

The algorithm, shown in figure 3.10, essentially 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 is part of the definition of 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 $|b| < \mathsf{S}$, is disjoint from the portion of f that depends on this algorithm, that is, only values of $f(b)$ for $|b| \geq \mathsf{S}(n_i)$. Thus there is no circularity.

The $f()$ values for the entire infinite oracle can be enumerated by enumerating all values of i in sequence, and for each one, computing the appropriate values of M_i and c_i, and choosing an n_i that satisfies all the explicit and implicit lower bounds on n that we mentioned above. Then, n_i is used in the above algorithm to allow us to define s_i and choose the appropriate x, which determines $f(b)$ for all b where $|b| = \mathsf{S}(n_i)$; these values of f can then be added to the table for use in the algorithm later when running on higher values of i.

We now explain why the simulation carried out by the (oracle-less) decompression algorithm 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. Either q_{j+1} 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 been before this one, since this query is not of the type that would have caused the node to be pebbled in direction D. In either case we will already have a value in array entry $q[j+1]$.

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 algorithm prints x, as required for the proof of lemma 2.

3.4.5 Can this proof be carried farther?

Given the work above, an obviously desirable next step would be to show that $\mathbf{RST}(\mathsf{S},\mathsf{T}) \neq \mathbf{ST}(\mathsf{S},\mathsf{T})$ for a larger class of space-time functions S,T in a reasonable serial model of computation *without* an oracle. A similar problem of following a chain of nodes may still be useful for this. But when there is no oracle, and when the required time is larger than the input length $\mathsf{T} \succ n$, there is no opportunity to specify an incompressible chain of nodes to follow. Instead, the function f mapping nodes to their successors must be provided by some actual computation that is specified by the relatively short input. It will be helpful if f is non-invertible or is a one-way invertible function, whose inverse might be hard to compute. But the function will still have some structure, and so it may be very difficult to prove that there are no shortcuts that might allow the result of repeated applications of the function to be computed reversibly using little time or space.

3.5 Summary of reversible complexity results for traditional models

Let us summarize the above results on complexity in reversible machines. In sec. 3.2.2, p. 54, we described a variety of measures of the cost or "complexity" of a computation. Now we will summarize how reversible and irreversible models compare under different measures of computational complexity.

Let $\mathcal{M}$ denote an arbitrary (not necessarily reversible) abstract model of computation such as a Turing machine or RAM machine, in which primitive operations may

Given description d as described in the text,
Let $q[1] \dots q[t]$ be a table of node values,
initially all NULL.
Initialize all $q[j]$'s not pebbled in direction D,
as specified by description d.
Simulate M_i in direction D from configuration C_τ,
as follows:
To simulate a single operation of M_i:
If it's a non-query operation, simulate it
straightforwardly, and proceed.
Otherwise, it's an oracle query.
Examine oracle tape.
If it's not of the form b or $b\#c$ for
bit-strings b, c, $|b| = |c|$, do nothing.
If $|b| < \mathsf{S}$, look up $f(b)$ in a finite table,
and set the oracle tape appropriately.
If $|b| > \mathsf{S}$, do nothing for this operation.
If the query is of the form b, then
If current time matches some $\Delta\tau_j$,
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
For each $\Delta\tau_j$ matching current time,
set $q[j]$ to b or c depending on 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.
Repeat until time exceeds largest $\Delta\tau_j$.
Print all $q[j]$'s.

Figure 3.10: Algorithm to print the incompressible chain of nodes x via simulation of the reversible machine M_i.

be reversible or irreversible, and unbounded memory is available. Let $\mathrm{R}(\mathcal{M})$ denote the corresponding reversible model of computation, like $\mathcal{M}$ but with all primitive operations constrained to be perfectly logically reversible, and with an extra stack for history information made available to each active processing element from the original machine.

Number of ticks. Under the cost measure T (number of "ticks" of some synchronous, computational "clock"), a reversible model $\mathrm{R}(\mathcal{M})$ is exactly as efficient as an arbitrary model $\mathcal{M}$. This follows from the Landauer-Lecerf-Bennett ideas as we discussed in §3.3.3, p. 61, which apply to parallel models as well as to serial models.

Memory requirement. Under the cost measure S (maximum memory used at any time during the computation), $\mathrm{R}(\mathcal{M})$ is exactly as efficient as $\mathcal{M}$. This follows from the Lange-McKenzie-Tapp technique we discussed in §3.3.5.3, p. 65. However, if the model includes an input stream that can only flow one way and whose length is not included in S, then Pin's proof [136] applies, and the reversible model is structly less space-efficient, because there are regular languages that cannot be recognized by constant-space reversible machines.

Memory and number of ops. Under the cost measure (S, T), without additional assumptions, we only know that $\mathrm{R}(\mathcal{M})$ is no more efficient than $\mathcal{M}$. However, given one-way external inputs, reversible models are less efficient by our argument in the previous paragraph. Moreover, this is also true given a certain oracle, or given random-access external inputs, by our two new theorems from §3.4. We conjecture that reversible models are less (S, T)-efficient even given only internal inputs and simple primitive operations, but this has not yet been proven.

Memory times num. ops. The statements of the previous paragraph also apply to the cost measure ST (the product of the traditional space and time). Therefore, probably reversible models are in general strictly less space-time efficient, in traditional complexity-theory terms.

Note that all the above comparisons deal in measures of complexity that are characterized in abstract, computational terms, such as the number of operations performed, rather than as real physical quantities; and the models of computation that were compared were all idealized abstract models, rather than models of machines as they could be physically implemented.

Normally in computer science it is often assumed that such abstract models are a good enough approximation to reality so that correct conclusions can be inferred from the resulting abstract theory. In our case, the theory would seem to indicate that machines that are constrained to be reversible are strictly inferior to unconstrained machines, including machines that are completely irreversible.

However, in chapter 6 we will show that this conclusion is actually in error, in the sense that if one uses more physically realistic models of machines and of costs, machines that are completely *ir*reversible are instead strictly inferior to machines that are allowed to be reversible to some degree, and are sometimes even inferior to fully reversible machines. We anticipate that this inferiority will make itself felt in a variety of of present-day and projected future computing technologies.

Therefore, for the purpose of deciding between reversible and irreversible modes of computation in the real world, we see that traditional computer science and traditional complexity theory are inadequate; they give the opposite of the correct answer in some cases!

This underscores our overall point, which is that computer scientists must not become mired in the traditional models of computation, but instead should strive to keep their models up-to-date with all the new factors that become important as technology improves.

We believe that computer science, as a field, should look ahead and try to anticipate the ultimate physical limits of computer technology, and begin studying models that are accurate enough to give the right answer even in that limit.

In that spirit, chapter 4 reviews quantum computation, which will stretch the bounds of computational efficiency, if it turns out to be feasible. Chapter 5 further motivates our quest for an ultimate physical model of computation, and outlines some plausible candidates that should remain valid at least through the foreseeable future. Chapter 6 shows why the ultimate model will need to permit an arbitrarily high (if not perfect) level of logical and physical reversibility. Then, Part II of this thesis will present a variety of engineering designs and analyses demonstrating that reversible computation is indeed quite feasible, and that most concepts from ordinary irreversible computation remain applicable to reversible computing as well.