15. Limitations to Computing

15.1 Introduction

We conclude the book with a discussion of what is very difficult or even not possible in various models, or in computation as a whole.

In the course up to this point, we have concentrated on what is "doable", with what models, what speed, etc. Other questions asked in computer science focus on what is not doable, or not doable in less than certain time, etc. Here we briefly survey the kinds of things that are known.

15.2 Algorithm lower bounds

For any given problem, the growth rate of an algorithm to solve the problem will have an absolute minimum. Unlike upper bounds on algorithmic performance, which are demonstrated for specific algorithms, lower bounds are for problems, and span all possible algorithms. Thus establishing them is much more difficult.

An example of a fairly accessible lower-bound argument is for sorting using two-way comparisons between data objects without regard to their internal structure (this excludes radix sort, which relies on a radix representation of the objects). For this problem, we have a lower bound of

$$\Omega\ (N \log N)$$

to sort N data objects, meaning that N log N is O(the growth rate for *any* algorithm for the problem). This means that sorts such as heapsort and mergesort are "optimal" to within a constant factor.

In order to derive this bound, we take an abstract view of what it means to compute by comparisons. Each time a program makes a comparison, that provides new information about the data. Since each comparison has two outcomes, we can cast the information states about the data as *an information tree*. The root of the tree represents the state where we are completely ignorant of the data since we have not yet made any comparisons. From a general information state, a comparison will take us to one of two other states. At some point, we will have made sufficiently many comparisons to have determined the original order of the data, in other words to determine which *permutation* of the data is needed to put the elements into sorted order. Such an unambiguous state is a *leaf* of the information tree. Thus there will be one leaf for each permutation of the data. The sequence of comparisons made during a given run of the program will correspond to a path from root to leaf. A lower bound on the time to sort is thus the minimum length of all such paths.

We know the following: For N data elements, there are P = N! permutations, hence P leaves. Secondly, in a binary tree with P leaves, *some* path must have length at least log P. With all paths less than log P, we would have fewer than $2^{\log P} = P$ leaves, a contradiction. The worst case sorting time is thus at least log P = log N!. An approximation known as Stirling's formula says

$$\log N! \approx k\, N \log N + \text{lower order constants}$$

for appropriate k, which is what we need for our lower bound.

There are many areas where tight lower bounds are not known, e.g. in the area of NP-complete problems that includes the traveling salesman problem, proposition logic satisfiability, and many others.

15.3 Limitations on Specific Classes of Machines

For example, finite-state machines are limited in the kinds of functions they can compute. Some examples of languages that are not finite-state acceptable are:

matched-parenthesis languages

$\{0^n1^n \mid n \text{ a natural number}\} = \{\lambda, 01, 0011, 000111, ...\}$

$\{1^p \mid p \text{ a prime number}\} = \{11, 111, 11111, 1111111, ...\}$

The first two of these can be represented by a context-free grammar, but context-free grammars have their own limitations, e.g. $\{0^n1^n2^n \mid n \text{ a natural number}\}$ is not generated by any context-free grammar. All of the above are acceptable by Turing machines.

To see that the second language mentioned above is not acceptable by a finite-state acceptor, suppose to the contrary that it is. Let *N* be the number of states of a machine accepting $\{0^n1^n \mid n \text{ a natural number}\}$. Now consider the input 0^N1^N. Since the machine makes 2N > N state transitions in the process of reading this input, some state must recur. (This is called the **pigeon-hole principle**; if one puts *M* pigeons (states occurring in a sequence) into *N* < *M* holes (states in the machine), then some of holes must have more than one pigeon in them.) Suppose that q is such a repeated state. If q occurs both times during the processing of 0^n, then we have a problem: it would then also be the case that 0^p1^n would be accepted for some p < n. Similarly, we also have a contradiction if a repeated state occurs during processing of 1^n. Finally, if q occurs the first time during the processing of 0^n, and the second time during the processing of 1^n, then we get a contradiction in that a sequence where some 1's precede 0's is also acceptable.

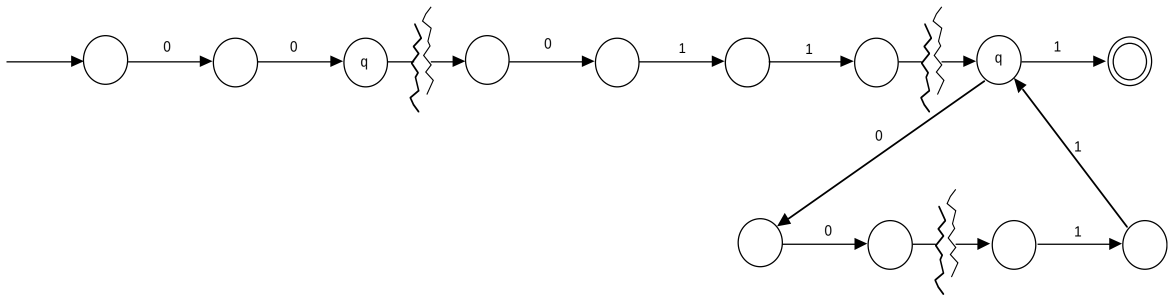

Figure 312: ***Anomalous acceptance arising from repeated states in a long sequence***

15.4 The Halting Problem

Consider any class of reasonably-powerful machines (Turing machines, programs written in C++, partial recursive functions, etc.) It is provable that there are well-defined functions that cannot be computed in the framework. Consider the program-analysis function H where

H(P) = 1 if program P diverges with P as input

H(P) diverges otherwise

[Here "diverges" means "never halts".] H is well defined. Many programs can accept programs as input (compilers, text formatters, etc.) Most of these sorts of programs always halt, and could be modified to halt with answer 1 if desired.

The problem is that there is no program for the particular function H. To see why, suppose P is the program for H. Then H(P) either diverges or it doesn't. But since P is a program for H, the first line of the definition says that if H(P) diverges, then H(P) halts. The second line says that if H(P) does not diverge, then H(P) diverges. Thus we have an absurdity. We are forced to conclude that our assumption was wrong that a program for H exists. This is usually summarized as

The halting problem is unsolvable.

Principle of Diagonalization

The proof that the halting problem is unsolvable relies on the *Principle of Diagonalization*. A classical mathematical use of this principle is to prove that the set of all subsets of the natural numbers is not countable. As promised in the first chapter, we will now discuss this proof. Let 2^{ω} represent the set of all subsets of ω. Suppose that 2^{ω}

were countable, i.e. there is an enumeration of it. Let the sets in the enumeration be $\{S_0, S_1, S_2, \ldots\}$. Now we construct a new subset of ω, call it S. The definition of S is

$$S = \{ n \in \omega \mid n \notin S_n \}$$

That is, S is the set of all natural numbers n such that n is not a member of S_n in the supposed enumeration. The funny thing about S is that it is obviously a subset of ω. Therefore, it should appear in the enumeration. But where does it appear? It must be S_n for some n. However, then we have a problem: For the index n of S_n, is $n \in S_n$ or not? If we assume $n \in S_n$, then we get from the definition of S (a synonym for S_n) that $n \notin S_n$. If we assume $n \notin S_n$, then we get from the same definition that $n \in S_n$. We are snookered either way. Therefore we must backtrack to where we made an assumption, and that was that 2^ω could be enumerated. Thus 2^ω is not countable.

The reason this is called diagonalization is that if we create an infinite two-dimensional array, listing the elements of along the top and the supposed enumeration along the side, then put a 1 for the entry in column n, row S_m, if $n \in S_m$ and put a 0 if $n \notin S_m$. Consider the diagonal of this array. Flip the values on the diagonal, i.e. interchange 0 and 1. If this diagonal is flattened out as a if a row, its 0's and 1's correspond to a subset of ω, and it should therefore appear as one of the rows. However it cannot, because it differs from every row in at least one position, according to its construction.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…
S_0	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	…
S_1	1	**0**	1	0	1	0	1	0	1	0	1	0	1	0	1	…
S_2	0	1	**0**	1	0	1	0	1	0	1	0	1	0	1	0	…
S_3	1	1	0	**1**	0	0	0	0	0	0	0	0	0	0	0	…
S_4	0	0	1	0	**1**	0	0	0	0	0	0	0	0	0	0	…
S_5	1	0	1	0	0	**0**	1	0	1	0	1	0	1	0	0	…
S_6	0	1	1	0	0	0	**1**	0	0	0	0	0	0	0	0	…
S_7	1	1	1	0	0	0	0	**0**	0	0	0	0	0	0	0	…
S_8	0	0	0	1	0	1	0	1	**0**	1	0	1	0	0	0	…
S_9	1	0	0	1	0	0	0	0	1	**0**	1	0	1	0	1	…
S_{10}	0	1	0	1	0	0	0	1	1	0	**1**	0	1	1	0	…
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	

Figure 313: Diagonalization construction

The row obtained by inverting the diagonal, 11100101110... , cannot appear as any row in the enumeration.

In terms of Turing machines, we can list all of the Turing machines in the same way we listed the subsets of the natural numbers. Across the top, we list the inputs to those Turing machines, which as we have already discussed, can be enumerated. The diagonal of the Turing machine array corresponds to those machines that halt on the tape with their own description. By inverting that diagonal, we have a representation of the function in the halting problem.

Although the definition of H might seem peculiar at first, other desirable, less peculiar functions can be similarly shown to be unsolvable. For example, it turns out that we don't need to rely on P being fed itself as input. We can *fix the input*, for example, to be a totally blank tape in the case of Turing machines, and still get unsolvability. In this kind of argument, we say that we have *reduced* the original halting problem to the blank-tape halting problem: if the latter is solvable, then the former is as well.

Note that throughout such a discussion, the domain of interest is functions that range over a wide set of inputs (e.g. all programs of a given model). We cannot prove unsolvability of the halting question for a particular single program on a particular input. Such a function would be a constant function (with a 0 or 1 answer) and would thus be computable.However, we might not know *which* program to use for it: one that always gives result 1 or one that always gives result 0.

Other Uncomputable Problems

The halting problem may sound very esoteric, something we'd never consider computing. But there are other problems that sound more down-to-earth, but which can be proved unsolvable by either an argument similar to the original halting problem, or by showing that arbitrary Turing machines can be represented in the model:

Busy Beaver Problem

In 1962, T. Rado showed this problem to be uncomputable.

> Fix the alphabet to {1, _} (_ is blank). Let BB(N) be the largest number of 1's that will be printed by any halting N-state Turing machine when started on an all-blank tape. BB is not computable. It can be shown that BB has a faster-growth rate than any computable function. To get some idea of how large BB can be,
>
> BB(5) is at least 4098.
> BB(100) is at least ((((7!)!)!)!)

Domino Problems

In 1961, Hao Wang showed the following to be unsolvable.

> Is there an algorithm for determining whether the infinite half-plane can be completely tiled (such that regions adjacent between dominos have the same color) with an input set of types of 4-sided "dominoes"?

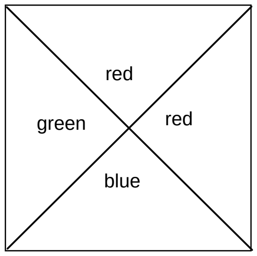

Figure 314: One of a finite set of dominoes to be used to tile an infinite plane

An algorithm for the domino problem would yield an algorithm for telling whether a TM halts. (The dominoes can simulate the possible successive configurations of a tape on the TM.)

For the quarter-plane, the question is "semi-decidable". It can be shown that the quarter-plane is tileable iff every finite "prefix" square is tileable. If some prefix square is untileable, this can be shown by enumerating all possible potential tilings of the square. We can thus determine whether the quarter plane is untileable by an expanding series of enumerations.

Rice's Theorem

The ultimate heartbreak of unsolvability is captured generally in an elegant theorem known as Rice's theorem. It asserts:

> Any property of computable functions that holds for some functions, but not for all, is undecidable for the corresponding programs of those functions. (includes halting, specialized halting, equivalence to a given function, etc.)

Basically, this means that if we are dealing with models that are general enough to represent any computable function, we can forget about developing algorithms that analyze those models for any functional property precisely.

15.5 NP-Complete Problems

Around 1970, computer scientists Edmunds, Cook, and Karp made a series of observations about various computational problems. Some of these problems seemed to be "easy", in the sense that there is a known polynomial-time algorithm for them. Others seemed "hard" in that no polynomial-time algorithm was known (the best known algorithms were exponential-time). It was observed that there were inter-conversions among the hard algorithms. Indeed, there were conversion algorithms that ran in polynomial time that would convert one hard problem into another. The implication was that if we could find a polynomial algorithm for one such problem, we would immediately have a polynomial time algorithm for several others, according to the known conversions. Eventually the class of hard problems called "NP complete" (NPC) was formulated. This was a set of problems all of that are interconvertible by polynomial time algorithms. The "NP" stands for "non-deterministic polynomial" and alludes to the fact that these problems can be run on a non-deterministic Turing machine in polynomial time. Non-deterministic Turing machines are related to Turing machines in the same way that non-deterministic finite-state machines are related to finite-state machines. Unfortunately, unlike finite-state machines, there is no known general simulation of a polynomial-time non-deterministic Turing machine that runs in polynomial time on a deterministic one. To the present, the question of whether the problems in the class NPC yield to any polynomial algorithm is open. Lacking is either a demonstration of such an algorithm or a proof that no such algorithm exists. Either one of these would resolve the question of the large family of NP complete problems. Meanwhile, simply showing a problem to be a member of this family is an indication that the problem is computationally quite difficult in terms of its complexity. See [Garey and Johnson 1975] or most any algorithms text for further discussion.

15.6 Amdahl's Law

Many computer scientists are interested in possible ways of speeding up computation by doing several things simultaneously, or "in parallel". Gene Amdahl observed that the success of such attempts is limited by the amount of the workload that is inherently sequential. Suppose that a program has a number N basic steps such that a fraction F of those steps cannot be done in parallel with any other steps, while the remainder of the steps can be done in parallel on P processors. Thus the time to do those remaining steps can be sped up by a factor of P.

The overall time to do the work with the parallel processing capability, counting 1 time unit per step, is:

$$F*N + (1-F)*N / P$$

The speedup is the time to do the work on 1 processor (N) divided by this time. In other words, the speedup is

$$\frac{1}{F + (1-F)/P}$$

or

$$\text{speedup} \le \frac{1}{F + (1-F)/S_{max}}$$

where S_{max} is the maximum speedup if everything were done in parallel.

This puts some limits on speedup that are perhaps surprising. If F were only 10%, then a speedup of at most 10 can be obtained. If F = 50%, then a speedup of at most 2 can be obtained, even if S_{max} were infinity. In other words, if a very fast technique is applicable to only half of the work, then the entire job will never be reduced by more than a factor of 2. Therefore parallel computers are best applicable to tasks that have a low degree of inherent sequentiality.

Amdahl's law is applicable to any speedup method, not just parallel computing. It indicates one of the uses of program profiling: effort to improve a program's performance is most effective if concentrated on the parts that take the most time.

15.7 The Glitch Phenomenon

We saw earlier how it is sometimes convenient to leave the digital abstraction. For example, we used a three-state buffer to achieve multiplexing via a bus. The outputs of such a buffer are 0, 1, and high-impedance, which is neither 0 or 1. When dealing with the interface between synchronous (clocked) systems and asynchronous (unclocked) systems, it should be understood that the digital abstraction is not quite sufficient.

Up to this point, we assumed that the signal to be gated into a latch was never changing during the interval in which the clock is changing. But when interfacing to the outside world, we have no control over when the sampled signal changes. This can cause problems with latch behavior. The following diagram shows such an interface, typically called a "synchronizer".

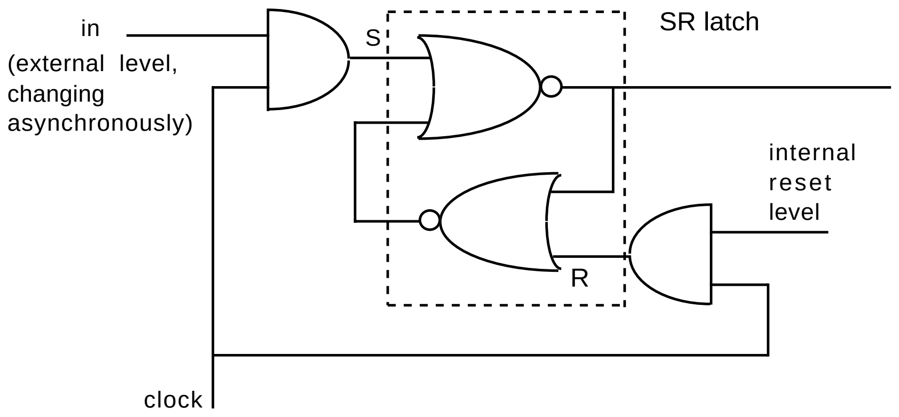

Figure 315: An attempted synchronizer

The set-reset latch formed of NOR gates has an input that is the AND of the clock and an unclocked external signal. The purpose of the circuit is to determine if the external signal has been raised since the latch was last reset. When the external is changing at the same time as the clock, the outputs of the top-left AND gate can be "runts", something in between 0 and 1. A runt can be sufficiently in between 0 and 1 that it causes the latch not to switch to the set state, but rather to "hang" mid way.

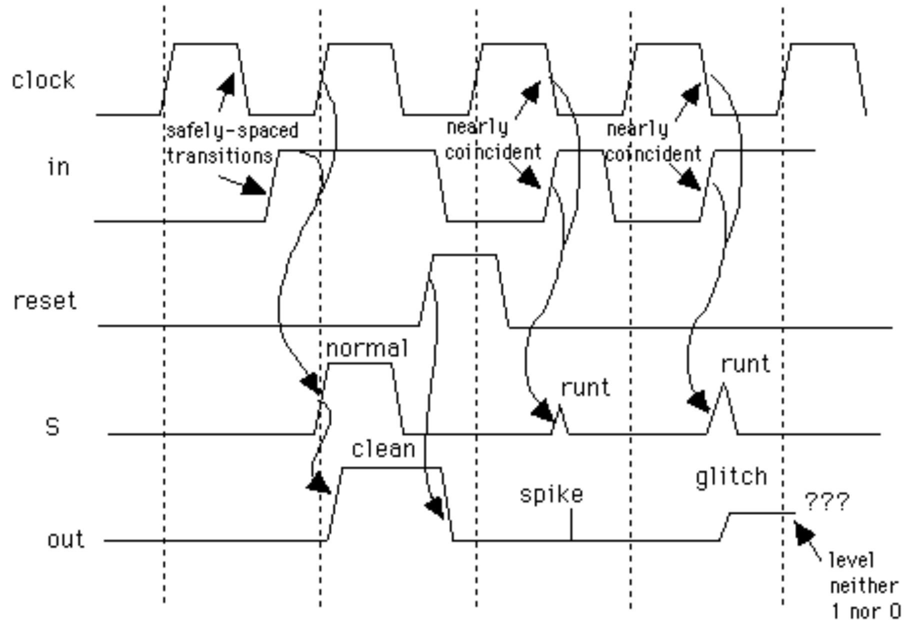

Figure 316: Possible timing of the interplay of the signals in the synchronizer

Metastable Behavior in Physical Terms

Three events of interest are shown. In the first, the external input rises between clock changes, and is detected when the clock rises. In the second, the falling clock ANDed with the rising external input produces a small runt pulse that is not enough to change the state of the latch. A spike (perhaps harmless) results in the output of the latch. In the third event, a similar coincidence creates a runt pulse with enough energy to bring the latch into a mid-way or metastable state, resulting in a glitch: a signal that is neither a digital 0 nor a 1.

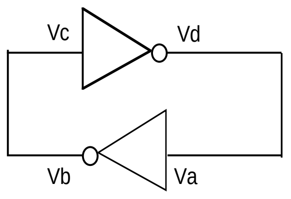

Figure 317: Simplified situation representing the opposing NOR gates in the latch

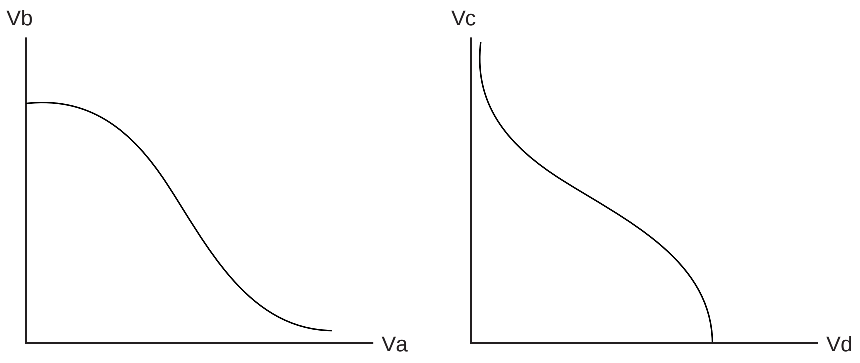

Figure 318: Response curves of the two inverters in isolation (the curves are the same, but one is rotated and reflected

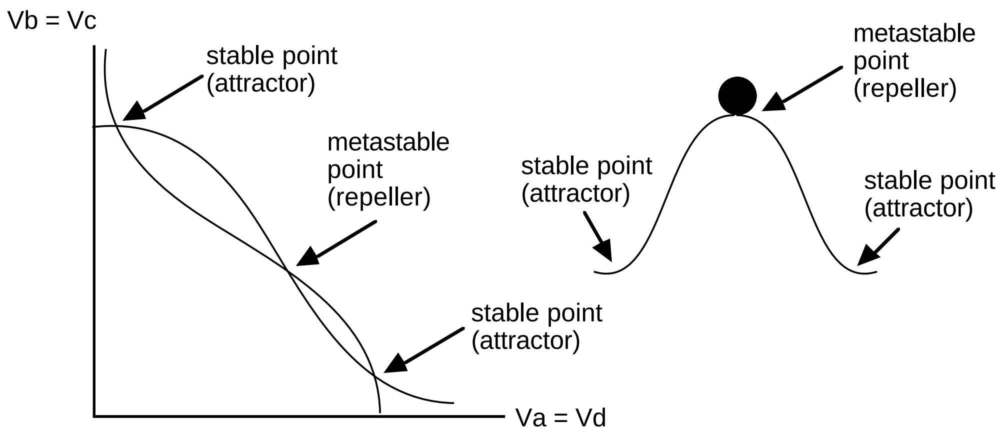

Figure 319: *Left*: Superimposed response curves reflect the interconnection with Vb = Vc and Va = Vd. The stable or equilibrium points are those where the curves intersect. The metastable point is a "repeller" in the sense that points on either side to move away from the repeller and toward one of the two attractors. *Right*: The ball-on-hill analogy. The latch operation is analogous to pushing the ball from one stable point to the other. With insufficient energy, the ball will not make it over the hill and will roll back down. With just the right amount of energy, the ball will sit atop the hill indefinitely (i.e. glitch).

Solution for the glitch problem: A perfectly glitch-free system cannot be built if asynchronous interaction is necessary. By waiting sufficiently long before sampling the output of a synchronizer latch, an adequately-small probability of a glitch can be achieved, e.g. one with expected time-to-failure of several centuries.

15.8 Chapter Review

Define the following terms:

- Amdahl's law
- diagonalization
- glitch
- halting problem
- lower bound
- metastable state
- pigeon-hole principle
- synchronizer

15.9 Further Reading

Michael R. Garey and David S. Johnson, *Computers and intractability*, W.H. Freeman, San Francisco, 1979.

David Harel, *Algorithmics – The Spirit of Computing*, Addison-Wesley, Reading, Massachusetts, 1987. [Further discussion of unsolvable problems. Easy to moderate.]

J.L. Hennessy and D.A. Patterson. *Computer Architecture - A quantitative approach*, Morgan Kauffman, 1991. [Moderate.]

R. Machlin and Q.F. Stout, *The complex behavior of simple machines*, in S. Forrest (ed.), *Emergent Computation*, MIT Press, 1991. [Recent results on the Busy Beaver problem.]

T. Rado, *On non-computable functions*, Bell Systems Technical Journal, May 1962, pp 877-884. [Introduces the Busy Beaver problem.]

H. Rogers, *Theory of recursive functions and effective computability*, McGraw-Hill, 1967. [Rice's theorem and related arguments. Difficult.]

J.F. Wakerly. *Digital design principles and practices*, Prentice-Hall, 1990. [Easy to moderate.]

Hao Wang, *Proving theorems by pattern recognition*, Bell Systems Technical Journal, 40, pp. 1-42, 1961. [Relationship of a domino problem to the halting problem.]