Traditional and Alternative Computing

William Gasarch

June 29, 2026

Abstract

LATER

Contents

1 Introduction

2 A Brief History of the Theory of Computation

2.1 Computability

In the 1930's, 1940's and 1950's, motivated by both concerns about constructive proofs in Mathematics, and the the existence of real computers, several people developed models of computation: These models of computation had very different motivations; however, they ended up all computing *the same set of functions!* In addition, the time loss in converting between them is bounded by a polynomial (though this last fact was not useful until around 1970).

For this exposition we do not need to know the formal definition of Turing machine. We only need to know the following:

1. Turing machines take as input a string of symbols from an alphabet Σ. Typically $\Sigma = \{0, 1\}$. The set of all strings is denoted Σ^*.

2. If M is a Turing machine and $x \in \Sigma^*$ then one can run M on x. This is denoted $M(x)$. The computation might not halt.

3. If the computation does halt it will output an element of Σ^*.

4. Any function that can be computed by a conventional digital computer can be computed by a Turing machine. Hence you may think of Turing machines as Java programs.

Turing machines give a *well defined notion of computability*. Hence questions like *are there functions that are not computable?* can be *asked*. And the answer is yes; the following problem is undecidable.

Definition 2.1 *The Halting Problem (HALT):* Given a program M (say in Java) and an input x, will $M(x)$ halt?

1. That HALT is undecidable is a potential problem for programmers who want to test their programs before running them.

2. For people who work on the problem of program testing and verification, it would seem that HALT being undecidable is an obstacle. However, the proof that HALT is undecidable involves contrive programs that are unlikely to come up in real life.

3. Fortunately, there are very few problems that are (1) important to solve, (2) undecidable, and (3) the relevant instances to solve are common.

Upshot The notion of decidability has been defined rigorously and hence questions of the form *is problem X decidable?* can be posed and often answered.

2.2 Polynomial Time

So far we have only been looking at whether a problem can be solved *at all*. We will now look at *How long* solving a problem takes. How long a problem takes to solve depends on *the size of the input*. The longer the input, the more time should be allowed. Our concern is the *rate of growth*.

Definition 2.2 A is in P if there exists a Turing machine M and a polynomial p such that $\forall x$

- If $x \in A$ then $M(x) = YES$.
- If $x \notin A$ then $M(x) = NO$.
- For all x $M(x)$ runs in time $\leq p(|x|)$.

Note 2.3 The definition of P seems to be tied to the definition of a Turing machine. Fortunately, all of the models of computation are equivalent up to a polynomial time; hence, this definition of P is not tied to Turing machines. More concretely, we could have replaced *Turing Machine* with *Java Program* in the definition of P.

We now consider two problems that illustrate the importance of *Polynomial Time*.

1. SAT: Given a Boolean formula $\phi(x_1, \ldots, x_n)$, does there exist $b_1, \ldots, b_n \in \{T, F\}$ such that $\phi(b_1, \ldots, b_n) = T$? This can clearly be solved in roughly 2^n steps, by going through all elements of $\{T, F\}^n$. If one could give an algorithm for SAT that ran in time 2^{n-10} or even $2^{n^{1/2}}$ you would think it is still brute force search with some tricks to speed it up a little. However, if someone had an algorithm in time n^{1000}, even though it would not be practical, it would *not* be a brute force search. It would require something very clever (it could also probably be modified to be practical). Hence P is a way of saying *Not Brute Force*.

2. TSP (Travelling Salesperson Problem): Given n cities and all of the distances between them (formally, a weighted graph with positive weights) what is the cheapest way to visit all of the cities exactly once and return to where you started? This can clearly be solved in roughly $n!$ steps, by going through all possible cycles.

The following is true:

SAT is P iff TSP is in P.

Both problems are called NP-complete which we won't go into here, except to say that all problems that are NP-complete are thought to not be in P. There are literally thousands of NP-complete problems.

Given that a problem is NP-complete, and hence unlikely to be in P, how do you write an efficient program for it? There are several approaches:

1. Settle for an approximation. For the TSP problem there are algorithms that, for some cases of it, return a route which costs ≤ 1.5 times the optimal solution.

2. Settle for heuristics and real world instances. For TSP there are algorithms that, on most real world data, return a route which is within a few percent of being optima.

3. Restrict the domain. For TSP, on planar graphs, there is a polynomial time algorithm.

4. This entire discussion has assumed that we are using Turing Machines. There are alternative models of computation that may be faster.

Upshot The theory of NP-completeness allows us to identify which problems seem to require, using traditional computing, exponential time.

3 Can Analog Computers Solve NP-Complete Problems?

Vergis, Steiglitz, and Dickson [7] prove that, assuming P $\neq$ NP, analog computers *cannot* solve NP-complete problems efficiently. The paper also indicates that approximate solutions would be hard for analog computer.

Their upshot is that analog computers should work on continuous problems such as solving differential equations.

4 Computing Over the Reals

4.1 The Pour-El & Richards Model for Computing over the Reals

Pour-El & Richards [4], building on ideas of Abeth, Grzegorczy, and Turing defined a model of computation over the reals. Here is the basic idea (the definition is informal).

Definition 4.1 Let $f: \mathsf{R} \to \mathsf{R}$. f is *computable* if there exists a Turing machine M with three tapes such that the following happens: If x is a real is on the input tape (there are an infinite number of digits) then the M will write $f(x)$ to the output tape. In some variants M will use n bits of x to produce some $g(n)$ bits of $f(x)$.

Even though a computable function of the reals has infinite input, infinite output, and takes infinite time, there are cases in the real world where this is desired. For example, if a sequence of requests comes to an operating sytem, it needs to assign those requests resources.

This model was used to prove certain desired tasks are *not computable*. The lack of concern for the amount of reseources used by the Turing machine makes the model hard to use for real computations.

4.2 The Blum-Shub-Smale Model For Computing over the Reals

Blum, Shub, and Smale [3] (see also [1], [2]) defined a model of computation over the reals. In this model it is assumed that one can store reals as reals and that an operation on reals, such as addition, can be done in unit cost.

They developed an analog of NP-completeness and proved several results on the *limits* of computation over the reals. We give two examples

1. They showed that the problem of, given a polynomial $f : \mathsf{R}^n \to \mathsf{R}$ of degree 4, determining of there is an $a_1, \ldots, a_n \in \mathsf{R}$ such that $f(a_1, \ldots, a_n) = 0$ is (the analog of) NP-complete. Hence this problem is likely hard to solve..

2. The Mandelbrot set is undecidable.

There are some problems that can be solved quickly in the BSS model. We list two.

1. Finding an approximate root for a system of complex polynomial equations in n unknowns can be solved in average polynomial time.

2. Given a matrix, find its permanent can be solved in polynomial time. This is striking since this problem is thought to be hard for traditional computing.

As of now there has been no physical realization of the BSS model. We list two obstacles to finding such a realization.

1. Formulating an input and and output that are real numbers seems difficult, if not impossible.

2. Treating operations on reals as unit cost is unrealistic.

4.3 The Siegelmann-Sontag Model For Computing over the Reals

BILL REGLI- I am looking at the paper

`http://binds.cs.umass.edu/papers/1995_Siegelmann_JComSysSci.pdf`

and I can't tell if its over the reals or not, realizable or not,

5 Factoring With an Optical Computer/Factoring with massive Parallelism

Most problems in NP are either in P or are NP-complete. One curious exception is factoring. There are reasons to think factoring is not NP-complete. However, As of 2026 there are no polynomial time algorithms for factoring. The algorithms that are known rely on number-theoretic conjectures to analyze their run times.

We state two known algorithms. L is the length of the input.

1. The Quadratic Sieve (QS) algorithm is believed to runs in time roughly $\exp(cL^{1/2}(\ln L)^{1/2})$.
2. The Number Field Sieve (NFS), is believed to have run time roughly $\exp(cL^{1/3}(\ln L)^{2/3})$.

For both algorithms that constant c is small enough This is small enough to make the algorithm practical for moderately large inputs.

Current techniques stopped being improved around 1988, and obtain run times of the form $\exp(L^t((\ln L)^{1-t}))$ where $0 < t < 1$. So current techniques might get factoring in time (say) $\exp(L^{1/10}((\ln L)^{9/10}))$, but not in P.

Since factoring is an important problem in cryptography, and seems to not be in polynomial time, it is of interest to see if some alternative computing device can solve it.

Shamir [5] proposed a way to do the most time-consuming part of both QS and NFS, the sieving, with an optical device that he named TWINKLE. At the time this approach looked promising; however, TWINKLE was never built for the following reasons:

1. It was expensive to build.
2. While making the sieve part fast, the rest of the algorithm still took some time.
3. The interface of TWINKLE for sieving, with the rest of the algorithm (which is classical) is difficult. This is a common problem for alt computing: the input and output to alt computing take up a lot of time.
4. Parallel computers for factoring got better, thus negating the advantage of TWINKLE.

Shamir and Tromer [6] recognized the issues with TWINKLE and used the lessons learned to design TWIRL which is a classical computing device inspired by TWINKLE. TWIRL is highly parallelized. Like TWINKLE, TWIRL is used for the sieving step. Since TWIRL is classical it avoids some of the problems that TWINKLE had.

Even so, TWIRL has never been built for the following reasons:

1. TWIRL would cost at least $20 million.

2. The part needed, notably wafer-scale integration, are hard to build.

3. If TWIRL was build to (say) crack RSA for 1024-bit keys, it is not possible to repurpose it for cracking RSA for 2048-bit keys.

4. Over time cryptosystems may move away from being based on number theory.

Upshot

1. Using an optical computer for factoring ran into a common problem for alt computing: input and output. (There were other issues as well.)

2. The failure of TWINKLE let to TWIRL. Even though TWIRL didn't work out either, this does show the value of alt computing as inspiring other research.

References

[1] L. Blum. Computing over the reals: Where turing meets newton, 2001. `https://www.cs.cmu.edu/~lblum/PAPERS/TuringMeetsNewton.pdf`.

[2] L. Blum, F. Cucker, M. Shub, and S. Smale. *Complexity and real computation*. Springer, 1997.

[3] L. Blum, M. Shub, and S. Smale. On a theory of computation over the real numbers; NP completeness, recursive functions and universal machines (extended abstract). In *29th Annual Symposium on Foundations of Computer Science, White Plains, New York, USA, 24-26 October 1988*, pages 387–397. IEEE Computer Society, 1988. `https://doi.org/10.1109/SFCS.1988.21955`.

[4] M. Pour-El and I. Richards. *Computability in analysis and physics*. Springer, New York, Heidelberg, Berlin, 1989.

[5] A. Shamir. Factoring large numbers with the TWINKLE device. In *Crytographic Hardware and Embedding Systems*, pages 2–12. Springer, 1999. `http://www.lia.deis.unibo.it/Courses/TecnologieSicurezzaAK/twinkle.pdf`.

[6] A. Shamir and E. Tromer. Factoring large numbers with the TWIRL device. In *Advances in Cryptology-CRYTPO2003*, pages 1–26. Springer, 2003. `https://cs-people.bu.edu/tromer/papers/twirl.pdf`.

[7] A. Vergis, K. Steiglitz, and B. Dickinson. The complexity of analog computation. *Mathematics and Computers in Simulation*, 28(2):91–113, 1986. `https://www.cs.princeton.edu/~ken/MCS86.pdf`.