Hard Problems

Computability and Algorithmic Complexity

Contents

Last Updated December 3, 2025

1 Hilbert's Second Problem: Entscheidungsproblem

Is there an algorithm that takes as input a logical statement and tells us if the statement is true or false?

(Algorithms: Hat tip to Euclid and Al-Khwarizmi.)

A solution to this problem would make mathematics automatic (and mathematicians redundant).

Unfortunately for Hilbert, in 1930, Godel showed that the answer is no in his famous incompleteness theorems.

(A set of axioms is *complete* (of a particular kind) if for each sentence in the formal system, either the statement or its negation is provable from the system's axioms. A *consistent* system is free of contradictions.)

Godel used an encoding of statements as numbers to state a claim analogous to: "This statement is not provable".

So, any formal system powerful enough to express the facts of arithmetic is incomplete. In particular, there are true statements that cannot be proved.

1.1 Formalizing Algorithms (1935-1936)

Alonzo Church Lambda Calculus

```
L, M, N <-  x
         |  lambda x. N
         |  L M
```

Kurt Godel Recursive Functions

Computable functions on (say) Nat -> Nat defined from some basic functions (constants, successor, projection) via composition and (for-loop-style) recursion.

Alan Turing Turing Machines (many related variants)

Basic formulation: a machine with a

- **Tape** divided into cells on which symbols can be written
- A **tape head** that can read and write symbols on the current cell and move the tape one cell in either direction

- A **Finite State Machine** with a current state in a register at any time
- A **table** of instructions: state x symbol -> action (Actions include: move tape, erase or write symbol, change state)

Automata of increasing power:

1. Fixed Boolean circuits
2. Finite-State Machine
3. Pushdown Automaton (FSM plus stack) During transitions, it can use the top of the stack and manipulate it too.
4. Turing Machine

Universal Turing Machines:

> It is possible to invent a single machine which can be used to compute any computable sequence. If this machine U is supplied with the tape on the beginning of which is written the string of quintuples separated by semicolons of some computing machine M, then U will compute the same sequence as M.
> – Alan Turing, The Undecidable

(Later) Haskell Curry Combinators

(Later) Many other systems (cellular automata, rewrite systems, ...)

All of these systems have been proved to have equivalent computational power – they can all compute the same functions/algorithms. This is the **Church-Turing Thesis**: a function is computable if and only if it can be encoded in any of these systems.

1.2 The Halting Problem

Turing provided a concrete version of Godel's answer to Hilbert's problem: the **Halting problem**:

> Can you devise a procedure that given an arbitrary computer program can tell you whether that program will ever terminate?

The answer is no. To see this, suppose we have a program `halt()` that does this job. What happens if we feed `halt` to itself?

```
def paradox():
    if halts(paradox):
        loop_forever()
    return True
```

The halting problem is **undecidable**! Note the similarity to Godel's question: "This statement is not provable."

A practical point: many hard problems are the halting problem in disguise.

1.3 Two directions

These ideas take us in two distinct and interesting directions:

1. Computational Complexity

   What makes a problem hard? What are the hard problems and how do they relate? What can we do with computers?

2. Propositions as Types (aka The Curry-Howard Isomorphism)

   There is a formal correspondence between Types/Programs, Logical Proposition/Proofs, and objects/operations in particular kinds of Categories.

We'll look at the first today. The second is touches on the foundations of mathematics and is at the heart of dependent types and formal verification of programs.

2 What Makes a Problem Hard?

2.1 Bridges of Konigsburg

```
North bank ---------------------
   |  |                        \
West Island -------------------- East Island
   |  |                        /
South Island -------------------
```

2.1.1 Eulerian and Hamiltonian Circuit / Cycle

Eulerian a cycle in the graph that traverses each edge exactly once

Hamiltonian a cycle in the graph that visits each node exactly once

Why is one so much harder than the other? And what does that mean?

2.1.2 Insights and Exhaustive Searches

Euler's insight about the Bridges of Konigsburg reduces an exhaustive search to a simpler procedure. And given a proposed solution to the problem, we can check that it works.

For example, factoring integers is hard, but *checking* a factorization is easy. Just multiply pq and see if it equals r. The product is a **witness** to the solution.

Not all problems are like this. If I told you have have a Chess position that leads to mate in 300 moves, how would you check beyond exhaustive search? Chess is an adversarial game; we would have to be sure that we accounted for the best opposing moves...

2.2 Problems and Solutions

What do we mean by *problem* and what does it mean to *solve* the problem?

We think of specific cases, like the literal Bridges of Konigsburgh, as **instances** of problem, and solutions to specific cases as **witnesses** or **certificates** for *that particular instance.*

We are instead interested in families of instances specified by a particular input configuration and a solution is a function that produces a witness for every instance (or indicates that no such witness exists).

We typically consider two kinds of problems:

1. Decision problems :: Want to find whether or not a witness exists for any given instance.

   Solutions to these problems are function of type `Instance -> Boolean`.

2. Search problems :: Want to find a specific witness for any given instance, or determine that no such witness exists.

Solutions to these problems are functions of type `Instance -> Maybe Witness`.

Given a problem, we are interested in how the "complexity" of a problem scales with the **size of the input**.

Note: given a solution to a decision problem, we can often construct an efficient solution to the search problem, as we will see below. (The other direction is trivial.)

Problems are often named with words or phrases in ALL CAPS (or small caps).

2.2.1 Example: Euclid's Algorithm for GCD

```
EUCLID(a,b)

Input: Positive numbers a and b with a >= b

Decision problem: Are positive integers a and b relatively prime?

Search problem: Given positive integers a and b, find gcd(a,b)
```

Suppose $a > b$

$$\gcd(a, b) = \gcd(b, a \bmod b).$$

```
def euclid(a, b):
    if b == 0:
        return a
    return euclid(b, a % b)
```

What is the **size** of the input for this algorithm? Or put another way: how does the running time scale with the inputs?

We can let n be the **total number of digits** in a and b, or roughly $2 \log a$

Because $a \bmod b < a/2$, we know that EUCLID(a,b) has running time $O(n)$.

2.2.2 Example: Chess

```
CHESS(n)

Input: An n by n chess board with suitably generalized pieces and rules.
```

```
Decision problem: Does White have a winning strategy?

Search problem: Find White's winning strategy or indicate if none exists.
```

2.3 Measuring Complexity

We are interested in the resources used by a solution algorithm, typically computation time and memory. We'll focus on time.

How to measure the time complexity of an algorithm?

Vague answer: we count *basic operations* performed during the algorithm (e.g., addition, moving a memory location, making a comparison)

Specific answer: we count operations performed using a comprehensive model of computation (e.g., Turing machine, lambda calculus)

Important point: we generally compute this **up to a constant**. So if your computer is half as fast as mine, we don't worry about that difference.

Thus, in particular, we evaluate the *order of magnitude* of the time complexity, e.g., with big oh.

The **intrinsic complexity** of a problem (in time or whatever target resource we are considering) as the complexity of the *most efficient algorithm* for that problem.

In general, we use *asymptotic* criteria to understand the intrinsic difficulty of a problem, much as we do in statistical inference problems. In practice this gets expressed in terms of big-Oh, big-Theta, and their cousins.

A **complexity class** is a class of problems with a certain (asymptotic) performance bound in a particular criterion (e.g., time, space)

Key example: P is the class of problems for which an algorithm exists that solves instances of size n in time $O(n^c)$ for some constant c.

More generally, $\mathsf{TIME}(f(n))$ is the class of problems for which an algorithm exists that solves instances of size n in time $O(f(n))$.

Let $\mathsf{poly}(n)$ stand for $O(n^c)$ for some $c > 0$. Then $\mathsf{P} = \mathsf{TIME}\,(\mathsf{poly}(n))$ and $\mathsf{EXP} = \mathsf{TIME}\left(2^{\mathsf{poly}(n)}\right)$.

We can build whole hierarchies of complexity classes (the complexity zoo) this way, and we can use other criteria (e.g., SPACE) as well.

2.4 On the Importance of Being Polynomial

We use polynomial time, $O(n^c)$ for some contant c, as a baseline standard for "tractable" problems. This is sometimes shortned to **polytime**.

1. Polytime algorithms are those that are feasible in practice, though in reality only for small c.
2. If we convert the "basic operations" from one computer to another, polytime is preserved.
3. Using polytime lets us ignore some of the low-level details of an algorithm (e.g., specific representation of a graph) and focus on the essence.

2.5 Reductions

We say that problem `A` reduces to `B` when we can write a solution for `A` in terms of a solution for `B`:

```
A = translate_solution_BtoA . B . translate_instance_AtoB
```

where `.` stands for composition, `translate_instance_AtoB` converts *instances* of `A` into instance of `B`, and `translate_solution_BtoA` converts a solution of the `B` instance into a solution of the `A` instance.

Here, insist that the translation steps are polytime!

For a **decision problem**, the solution maps `Instance -> Boolean`, so no inverse translation is needed:

```
A = B . translate_instance_AtoB
```

For a search problem, the solution maps `Instance -> Maybe Witness`, so we have

```
def A(instance):
    solB = B(translate_instance_AtoB(instance))
    if solB is not None:
        return translate_solution_BtoA(solB)
    return None
```

or equivalently

```
A instance = do
  solB <- B (translate_instance_AtoB instance)
  return (fmap translate_solution_BtoA solB)
```

Saying that A can be reduced to B tells us that B is **at least as hard as A**. We write this as A $\leq$ B. If A can be reduced to B *and vice versa*, then A and B are equivalent (up to polynomial translation).

How can we show that a problem B is outside P? If A $\leq$ B and A is outside P, then so is B. More loosely, we often have many problems that can be reduced to B for which we can find no polynomial solution. . . is this evidence? We will see such reductions when we tour the class NP.

3 P and NP: The Nature of Mathematical Insight

3.1 Complexity classes

Recallthat $\mathsf{TIME}(f(n))$ is the class of problems for which an algorithm exists that solves instances of size n in time $O(f(n))$.

$$\mathsf{P} = \bigcup_{c>0} \mathsf{TIME}(n^c) = \mathsf{TIME}(\mathsf{poly}(n)).$$

Similarly, EXP is just $\mathsf{TIME}(2^{\mathsf{poly}(n)})$.

We have a hierarchy:

$$P \subset \mathsf{EXP} \subset \mathsf{EXPEXP} \subset \cdots .$$

This is not the only important hierarchy.

3.2 NP

Consider the EULERIAN CYCLE vs. HAMILTONIAN CYCLE. The first is in P, the second is not.

But they do have one thing in common: if I give you a candidate witness, you can **check it in polynomial time**.

The complexity class `NP` consists of those problems for which we have a polytime *verifier* V, such that for any instance i and candidate witness w, $V(i, w)$ indicates whether w is a valid witness for the instance i and can be computed in polynomial time. (Notice that the witness must be representable in a polynomial number of bits.)

HAMILTONIAN CYCLE is in NP.

Notice that $\mathsf{P} \subset \mathsf{NP} \subset \mathsf{EXP}$.

Alternative definition: polytime on **non-deterministic** machine

In general, we can define $\mathsf{NTIME}(f(n))$ analogously using $\mathsf{TIME}(f(n))$.

A decision problem is in NP if, whenever the answer to a particular instance is yes, there is a simple (polytime) proof of that fact.

3.3 P vs. NP: What does it mean?

P is the subset of problems that can be solved via some **insight** without requiring an **exhaustive search**.

The nature of mathematical insight/creativity: P ne NP means that some problems *require* insight or creativity to solve efficiently, that mathematical results require some creativity to prove. If P were equal to NP, this could all in principle be automated.

- EULERIAN CYCLE vs. HAMILTONIAN CYCLE as an example
- Proof
- Though unsolvable problems remain, e.g., the halting problem

3.4 A Brief Tour of NP

3.4.1 GRAPH k-Coloring

```
Input: A graph G
Problem: Is there a proper k-coloring of G?
```

Question: Show that GRAPH 2-COLORING is in P.

Question: Show that GRAPH k-COLORING <= GRAPH (k+1)-COLORING

Question: Show that PLANAR GRAPH k-COLORING <= GRAPH k-COLORING

What about the other direction? Is GRAPH k-COLORING <= PLANAR GRAPH k-COLORING?

Yes. Given a graph G, write in the plane with crossings (in polynomial time, why?)

Then replace crossings with "gadgets" that make the graph planar but do not change the colorability.

3.4.2 SAT

```
SAT
Input: A Boolean formula phi(x_1,...,x_n) in Conjunctive Normal Form
Problem: Is phi satisfiable?
```

That is, can we assign values to the boolean variables x_i that make the formula true.

Conjunctive Normal Form means an `and` of a series of clauses, where each clause is an `or` of variables and their complements.

A special case is `k-SAT`, in which each clause must have k variables.

Question: Show that GRAPH k-COLORING <= SAT.

($k = 3$: Three variables for each node, for colors say red, green, blue.)

(In general let v_{nc} be the variable that node n is color c.)

We have ands of the following clauses

1. $\text{or}_c\ v_{nc}$ for each n
2. $!v_{nc}$ or $!v_{nc'}$ for colors c, c'
3. $!v_{nc}$ or $!v_{n'c}$ for $\{n, n'\} \in E$

Now 1-SAT is in P quite directly, but interestingly so is 2-SAT.

$x_1 \mid x_2 <=> (!x_1 => x_2)\ \&\ (!x_2 => x_1)$

Convert a 2-SAT formula to a directed graph with nodes x_i and $!x_i$ for all i and an edge x_1 -> x_2 whenever $!x_1 => x_2$.

Then the formula is satisfiable iff there is no variable with paths from x to !x and !x to x

```
while there is an unset variable:
    choose an unset variable x
    if there is a path x to !x:
        set x = false
    elif there is a path !x to x:
        set x = true
    else:
        set x arbitarily
    iteratively propagate clauses !x | y and x | y with these settings
```

2-SAT can be solved with multiple calls to FIND PATH IN DIGRAPH which is in P.

For 3-SAT, it's not so simple. And indeed `SAT <= 3-SAT` and `k-SAT <= 3-SAT`.

Conversion: use dummy variables, e.g., x | y | z | u | w <=> (x | y | a_1) & (!a_1 | z | a_2) & (!a_2 | u | w).

3.4.3 Balancing Numbers

```
INTEGER PARTITIONING
Input: A list of k positive integers
Problem: Is there a partition of the list into
two subsets that have the same sum?
```

```
SUBSET SUM
Input: A set of k positive integers and a number t
Problem: Is there a subset of the numbers that sums to t?
```

Dynamic programming can give an integer partition with $O(wk)$ subproblems where w is proportional to the total length of the input. But if we scale by the number of digits in the input this is actually exponential time.

In practice, the number of bits is usually fixed, so the algorithm is practical.

3.4.4 Rivalries and Cliques

An **independent set** in a graph G is a subset of nodes where no two nodes in S are adjacent. (Ex: nodes are people, edges are rivalries)

A subset S of nodes is a **vertex cover** if every edge has at last one endpoint in S.

A subset S of nodes is a **clique** if every node in S is adjacent to every other node in S.

These three notions are related. TFAE:

1. S is an independent set
2. N - S is a vertex cover
3. S is a clique in the complementary graph G' (flips adjacency)

```
INDEPENDENT SET

Input: A graph G and an integer k
Problem: Does G have an independent set of size k or more?
```

```
VERTEX COVER

Input: A graph G and an integer k
Problem: Does G have a vertex cover of size k or less?
```

```
CLIQUE

Input: A graph G and an integer k
Problem: Does G have a clique of size k or more?
```

All these problems are in P if k is bounded, but if k is part of the input, they are in NP.

Show: If a graph with n nodes is k-colorable, it has an independent set of size at least n/k.

4 Another View of NP

For decision problems, we can express them as "properties", where say A(x) is true if x is a true-instance of A. Properties in P, we can check in polynomial time.

Properties in NP simply add an **existential quantifier**:

```
NP is the class of properties A of the form

  A(x) = There Exists w such that B(x, w)

where B is in P and |w| = poly(|x|).
```

This makes clear why NP treats true and false instances asymmetrically.

Also leads to coP and coNP complexity classes, and a hierarchy of classes with various numbers of interleaved "there exists" and "for alls".

5 NP-Completeness

A problem B is **NP-complete** if A <= B for all A in NP

We can see that at least one NP-complete problem exists, trivially

```
WITNESS EXISTENCE


Input: A Program P(x,w), an input x, and an integer t (given in unary)
Problem: Does there exist a w of size |w| <= t such that P(x,w) returns true
after t or fewer steps?
```

This is just the definition of NP, hidden in plain sight. (The unary specification ensures that t is <= the total input size.)

If any NP-complete problem is in P, then P = NP.

5.1 A more interesting example: CIRCUIT SAT

```
CIRCUIT SAT


Input: A boolean circuit C
Problem: Is C satisfiable?
```

We can reduce WITNESS EXISTENCE to CIRCUIT SAT by "compiling" the witness-verifying program to a circuit.

CIRCUIT SAT is NP-complete

And then

CIRCUIT SAT <= 3-SAT

so 3-SAT is NP-complete, and this continues to grow (. . . , Coloring, Tiling, Max Cut, Diophantine Equations, . . .).