Computational Complexity 1: Algorithms and Landscapes

Cristopher Moore
Santa Fe Institute
Computational complexity

Why are some problems qualitatively harder than others?
Computational complexity

Euler: at most two nodes can have an odd number of bridges, so no tour is possible!
What if we want to visit every vertex, instead of every edge?

A Hamiltonian cycle is one that visits each vertex exactly once and returns to the starting vertex. One such cycle for the dodecahedron is shown in Figure 1.5.

At first, Hamilton's puzzle seems very similar to the bridges of Königsberg. Eulerian paths cross each edge once, and Hamiltonian paths visit each vertex once. Surely these problems are not very different? However, while Euler's theorem allows us to avoid a laborious search for Eulerian paths or cycles, we have no such insight into Hamiltonian ones. As far as we know, there is no simple property—analogous to having vertices of even degree—to which Hamiltonianness is equivalent.

As a consequence, we know of no way of avoiding, essentially, an exhaustive search for Hamiltonian paths. We can visualize this search as a tree as shown in Figure 1.6. Each node of the tree corresponds to a part of a path, and branches into children nodes corresponding to the various ways we can extend the path.

In general, the number of nodes in this search tree grows exponentially with the number of vertices of the underlying graph, so traversing the entire tree—either finding a leaf with a complete path, or learning that every possible path gets stuck—takes exponential time.

To phrase this computationally, we believe that there is no program, or algorithm, that tells whether a graph with \( n \) vertices is Hamiltonian or not in an amount of time proportional to \( n \), or \( n^2 \), or a polynomial function of \( n \). We believe, instead, that the best possible algorithm takes exponential time, \( 2^{cn} \) for some constant \( c > 0 \). Note that this is not a belief about how fast we can make our computers. Rather, it is a belief that finding Hamiltonian paths is fundamentally harder than finding Eulerian ones. It says that these two problems differ in a deep and qualitative way.

While finding a Hamiltonian path seems to be hard, checking whether a given path is Hamiltonian is easy. Simply follow the path vertex by vertex, and check that it visits each vertex once. So if a computationally powerful friend claims that a graph has a Hamiltonian path, you can challenge him or her to Computational complexity
Computational complexity

As far as we know, the only way to solve this problem is (essentially) exhaustive search!
An exponential tree

Backtracking search: follow a path until you get stuck, then backtrack to your last choice

If there are $n$ nodes, this could take $2^n$ time

When can we avoid this kind of search?
Polynomials don’t grow too badly as $n$ grows...
...but exponentials explode.

\[ 2^n \]

- \( n = 1 \):
  - 1,048,576

- \( n = 2 \):
  - 1,073,741,824

- \( n = 3 \):

- \( n = 4 \):

Total:

18,446,744,073,709,551,615
Faster computers, bigger problems?

Say you can do $T$ steps in a week with your current computer...

According to Moore’s law, next year this will be $2T$

If $T=O(n^2)$, then doubling $T$ multiplies $n$ by $\sqrt{2}$

But if $T=O(2^n)$, doubling $T$ just changes $n$ to $n+1$
2.3 The Basics

Euler algorithm for \textsc{Eulerian Path}.

Algorithm:

\begin{itemize}
  \item \textbf{Input:} A graph \( G = (V, E) \)
  \item \textbf{Output:} “yes” if \( G \) is Eulerian, and “no” otherwise.
  
  \begin{algorithmic}
    \STATE \( y := 0 \);
    \FORALL {\( v \in V \)}
      \IF {\( \deg(v) \) is odd}
        \STATE \( y := y + 1 \);
      \ENDIF
    \ENDFOR
    \IF {\( y > 2 \)}
      \RETURN “no”;
    \ELSE
      \RETURN “yes”;
    \ENDIF
  \end{algorithmic}
\end{itemize}

2.4.2 Details, and Why They Don’t Matter

In the Prologue we saw that Euler’s approach to \textsc{Eulerian Path} is much more efficient than exhaustive search. But how does the running time of the resulting algorithm scale with the size of the graph? It turns out that a precise answer to this question depends on many details. We will discuss just enough of these details to convince you that we can and should ignore them in our quest for a fundamental understanding of computational complexity.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{until_end_world.png}
\caption{Running times of algorithms as a function of the size \( n \).}
\end{figure}
Divide and conquer

Tasks and subtasks

Figure 3.2: The tree corresponding to the recursive algorithm for the Towers of Hanoi with $n = 3$. Each node $\text{Hanoi}(n, i, j)$ corresponds to the subproblem of moving $n$ disks from peg $i$ to peg $j$. The root node, corresponding to the original problem, is at the left. The actual moves appear on the leaf nodes (the ellipses), and the solution goes from top to bottom.
Divide and conquer: mergesort

\[ T(n) = 2T(n/2) + n \]

\[ T(n) = n \log_2 n \]
Divide and conquer

The Fast Fourier Transform
When greed is good

Minimum Spanning Tree (Boruvka, 1920): add the shortest edge
When greed is good

For Minimum Spanning Tree, doing the best thing in the short term can never lead us down the wrong path.
The primrose path

The Traveling Salesman Problem
Landscapes

A single optimum, that we can find by climbing:
Landscapes

Many local optima where we can get stuck
Reorganizing the landscape: max flow

Each edge has a capacity

Greedy: push more flow along any path with excess capacity

But we can get stuck in local optima!
Reorganizing the landscape: max flow

Solution: allow reverse edges to cancel out previous flow

Theorem: now we can’t get stuck

Sometimes we can turn the landscape from Rockies to Mt. Fuji, by defining what moves are possible—but not always!
“Reducing” one problem to another

Translate new problem into one we already know how to solve

Bipartite Matching $\leq$ Max Flow

If Max Flow is easy, then so is Bipartite Matching
Duality: max flow and min cut

Cut the smallest set of edges that divides s from t

Find Max Flow from s to t, and cut the saturated edges
Computation and complexity

*Systems* aren’t simple or complex; questions about them are

Intrinsic complexity of a problem: the running time (or memory use, or other resource) of the *best possible algorithm* for it

Worst-case! Works for all instances = works for worst ones

Upper bounds are easy: just give an algorithm

Lower bounds are hard!
How can we tell if an algorithm is optimal?

Twenty questions: can only distinguish $2^{20}$ situations

To sort, we need $\log_2 n! \approx n \log_2 n$ comparisons
Lessons so far

We only know of a small number of families of algorithms that are guaranteed to be efficient:

- Greedy (local search)
- Divide and conquer
- Dynamic programming
- Linear programming, convex programming, and duality

Are there others we haven’t found yet? Are there fundamental limits to what efficient algorithms can do?
Computational Complexity 2:
NP-completeness and the P vs. NP question

Cristopher Moore
Santa Fe Institute
Needles in haystacks

**P**: we can find a solution efficiently

**NP**: we can *check* a solution efficiently
Complexity classes

NP
- Hamiltonian Path

P
- Eulerian Path
- Multiplication
NP-completeness

Some problems B have the amazing property that any problem in NP can be reduced to them: $A \leq B$ for all $A$ in NP

But if $A \leq B$ and $A$ is hard, then $B$ is hard too

So if $P \neq NP$, then $B$ can’t be solved in polynomial time

How can a single problem express every other problem in NP?
Any program that tests solutions (e.g. Hamiltonian paths) can be “compiled” into a Boolean circuit.

The circuit outputs “true” if an input solution works.

Is there a set of values for the inputs that makes the output true?

Satisfying a circuit
Add variables representing the truth values of the wires

The condition that each gate works, and the output is “true,” can be written as a Boolean formula:

\[
(x_1 \lor \overline{y}_1) \land (x_2 \lor \overline{y}_1) \land (\overline{x}_1 \lor \overline{x}_2 \lor y_1) \\
\land \cdots \land z .
\]
Our first NP-complete problem!

Given a set of clauses with 3 variables each,

\[
(x_1 \lor \overline{x}_2 \lor x_3) \land (x_2 \lor x_{17} \lor \overline{x}_{293}) \land \cdots
\]

does a set of truth values for the $x_i$ exist such that all the clauses are satisfied?

$k$-SAT (with $k$ variables per clause) is NP-complete for $k \geq 3$
If 3-SAT were easy...

we could take any problem in NP we want to solve,
write a program that checks solutions,
convert that program into a circuit,
convert that circuit to a 3-SAT formula which is satisfiable if a solution exists,
and use our efficient algorithm for 3-SAT to solve it!

So, if 3-SAT is in P, then all of NP is too, and P=NP

Conversely, if P≠NP, then 3-SAT cannot be solved in polynomial time: something like exhaustive search is needed
Given a set of countries and borders between them, what is the smallest number of colors we need?
From SAT to Coloring

“Gadgets” enforce constraints:

Graph 3-Coloring is NP-complete

Graph 2-Coloring is in $\mathbf{P}$ (why?)
Why are some problems NP-complete?

Can we tile a shape with these tiles?
Because we can use them to build a computer.
Even problems in continuous mathematics…

\[ \int_{-\pi}^{\pi} (\cos a_1 x)(\cos a_2 x) \cdots (\cos a_n x) \, dx \neq 0? \]
Thousands of NP-complete problems
Oh, cruel world!

NP-completeness is a worst-case notion...

We assume that instances are designed by a clever adversary to encode hard problems.

A good assumption in cryptography, but not in most of nature.

*The scientist is always working to discover the order and organization of the universe, and is thus playing a game against the arch-enemy, disorganization. Is this devil Manichaean or Augustinian? Is it a contrary force opposed to order or is it the very absence of order itself?*

— Norbert Wiener, *Cybernetics*
From theory to the real world

Many algorithms that take exponential time in the worst case are efficient in practice.

Optimization problems are like exploring a high-dimensional jewel.

If we add noise to the problem, the number of facets goes down, and the path to the top gets shorter.

Figure 9.18: The polytope on the left has 200 facets and 396 vertices. On the right, we perturb this polytope by adding 5% random noise to the right-hand sides of the inequalities, reducing the number of facets and vertices to 87 and 170. The angles between the facets have become sharper, so the polytopes' shadows, i.e., the two-dimensional polygons that form the outlines of these pictures, have fewer sides.

If the adversary has to carefully tune the parameters of an instance to make it hard, adding noise upsets his plans. It smooths out the peaks in the running time, and makes the problem easy on average.

Spielman and Teng considered a particular pivot rule called the shadow vertex rule. It projects a two-dimensional shadow of the polytope, forming a two-dimensional polygon, and tries to climb up the outside of the polygon. This rule performs poorly if the shadow has exponentially many sides, with exponentially small angles between them. They showed that if we add noise to the constraints, perturbing the entries of $A$ and $b$, then the angles between the facets—and between the sides of the shadow polygon—are $1/\text{poly}(m)$ with high probability. In that case, the shadow has $\text{poly}(m)$ sides, and the shadow vertex rule takes polynomial time. We illustrate this in Figure 9.18.

Smoothed analysis provides a sound theoretical explanation for the surprisingly good performance of the simplex algorithm on real instances of LP. But it still leaves open the question of whether LP is in $P$—whether it can be solved in polynomial time even in the worst case, when the adversary has the last word. We will answer that question soon. But first we explore a remarkable property of LP that generalizes the relationship between MAXFLOW and MINCUT, and which says something about LP’s worst-case complexity as well.
Alternatives

Probably Approximately Correct [Valiant]

Noise can foil the adversary, producing a smoother problem [Spielman and Teng]

Landscapes are not as bumpy as they could be: good solutions are close to the optimum [Balcan, Blum, and Gupta, clustering]

In nature, problems and algorithms coevolve (e.g. protein folding)
Beyond NP

Which of these puzzles are in NP? Which has a solution that is easy to check?
Logical hierarchies

I can win if there exists a move for me, such that for all of your replies, there exists a move for me...

Sam Loyd (1903)
Mate in 3

Lewis Stiller (1995)
Mate in 262
Undecidability

Suppose we could tell whether a program $p$ will ever halt. This would be really handy!

```plaintext
Fermat
begin
    t := 3;
    repeat
        for $n = 3$ to $t$ do
            for $x = 1$ to $t$ do
                for $y = 1$ to $t$ do
                    for $z = 1$ to $t$ do
                        if $x^n + y^n = z^n$ then return $(x, y, z, n)$;
                    end
                end
            end
        end
    end
    $t := t + 1$;
until forever;
end
```

I have discovered a marvelous proof that this program will run forever, but it is too small to fit on this slide...
Suppose $\text{halt}(p, x)$ can tell whether $p$, given input $x$, will halt. Then we could feed it to itself, and run this program instead:

\[
\text{trouble}(p): \\
\text{if \ halt}(p, p) \text{ loop forever} \\
\text{else \ halt}
\]

Will $\text{trouble}(\text{trouble})$ halt or not?

Undecidable problems $\Rightarrow$ unprovable truths!
“Computers play the same role in complexity that clocks, trains and elevators play in relativity.”

– Scott Aaronson
Questions, questions...

Suppose we have a cellular automaton. There are lots of questions we could ask about it:

- Given an initial state $s$, what will the state be at time $t$? \textbf{P}
- Does a state $s$ have a predecessor? \textbf{NP}
- On a lattice of size $n$, is $s$ on a periodic orbit? \textbf{PSPACE}
- On an infinite lattice, will $s$ ever die out? \textbf{Undecidable}

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure8.2.png}
\caption{A periodic orbit of cellular automaton rule 30, with time running from left to right. Each column is a state with $n$ bits, where 0 is white and 1 is black. The transition rule is illustrated above, showing how the current values of a bit and its neighbors determine its value on the next time step. We use cyclic boundary conditions so that the lattice wraps around. Here $n = 7$, the initial state is 0000101, and the period is $t = 63$. While the period may be exponentially long, we only need $n$ bits to remember the current state, so telling whether the initial state lies on a periodic orbit is in \textbf{PSPACE}.}
\end{figure}
Deep questions

Is finding solutions harder than checking them? When can we avoid exhaustive search?

How much memory do we need to find our way through a maze?

What if we only need good answers, instead of the best ones? Are there problems where even finding good answers is hard?

How much does it help if we can do many things at once?

If you and I are working together to solve a problem, how much do we need to communicate?

Are there good pseudorandom generators? Are there strong cryptosystems?

How much does quantum physics help?
The computational lens

Do brains compute? Do cells? Do societies? Do planets?

Maybe the wrong question…

Does focusing on flows and transformations of information (as opposed to e.g. energy) help me understand my system?
Shameless Plug

To put it bluntly: this book rocks! It somehow manages to combine the fun of a popular book with the intellectual heft of a textbook.

Scott Aaronson, MIT

This is, simply put, the best-written book on the theory of computation I have ever read; one of the best-written mathematical books I have ever read, period.

Cosma Shalizi, Carnegie Mellon

A creative, insightful, and accessible introduction to the theory of computing, written with a keen eye toward the frontiers of the field and a vivid enthusiasm for the subject matter.

Jon Kleinberg, Cornell

www.nature-of-computation.org