Lecture Notes for CSCI 311: Algorithms
Set 20-P, NP, and Undecidability

Professor Talmage

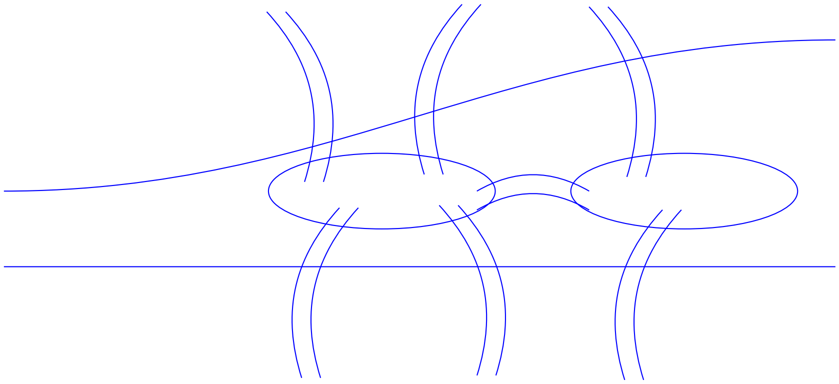
May 2, 2025

1 Tours

To wrap up our course, we want to consider some of the limits of computation, both in terms of what problems computers can solve and in terms of complexity. To that end, we will look at a couple of similar graph problems.

Definition 1. An *Euler cycle* in a graph G is a path that crosses each edge exactly once, ending where it began. This is also known as an *Euler tour*.

Exercise: In the following map, can you cross each of the bridges exactly once, ending where you began?



If we represent this map as a graph, where each landmass is a vertex and bridges are edges, we get the Euler tour problem:

Input: Undirected graph G

Output: Euler tour of G

This problem was formulated, and solved, for the above (simplified) map of Königsberg, Prussia, by the mathematician Leonhard Euler in the 18th century. There is actually a very simple algorithm for this problem:

1. Start at any vertex
2. Follow any unused edge from the current vertex (as long as removing that edge would not disconnect the graph)

3. Repeat until all edges are used

4. If you have used every edge and are currently at the start vertex, there is an Euler tour (and you have traversed it). Otherwise, there is no Euler tour.

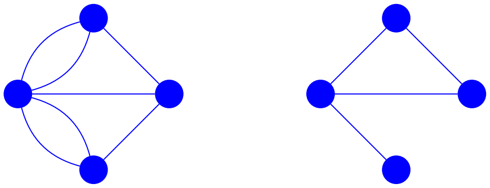
The idea of the proof is that if there is an Euler tour, you cannot get stuck in any node, until you have traversed the entire tour. This follows because the tour must use every edge, and thus must arrive at every node the same number of times the algorithm does, and must be able to leave each node the same number of times.

This actually suggests a much simpler solution to the corresponding *decision problem*, which simply asks whether an Euler tour exists. Given an undirected graph G, there is an Euler tour of G iff G is connected and every node has even degree. In general, decision problems are simply Boolean problems, often asking whether an input has a particular property. Here, we ask whether a given graph has the property of containing an Euler tour.

Consider another, similar, problem:

Definition 2. A *Hamiltonian cycle or tour* in a graph G is a path which visits every vertex exactly once, then returns to the start vertex.

Exercise: Can you find a Hamiltonian cycle in each of the following graphs?



Can you prove that there is no Hamiltonian tour in the second graph?

You may notice that this is really just the Traveling Salesman Problem from recitation. There is no known efficient algorithm to find a Hamiltonian tour, or even to determine whether one exists! All we know to do is try all possible paths, which can be $O(|V|!)$. Ouch. We are actually pretty confident that there is no sub-exponential algorithm, but no one has been able to prove it.

One thing we can do efficiently is *verify* that a claimed solution is correct. If someone gives you a graph and a claimed Hamiltonian tour $[u_1, u_2, \ldots, u_n]$, you can verify that each (u_i, u_{i+1}) is in the graph, as well as (u_n, u_1). This takes only $|V|$ time in an adjacency matrix, plus a bit to verify that you are not repeating vertices, but no more than $O(|V|^2)$.

Definition 3. A problem is *verified* by checking that a claimed solution is valid.

Exercise:

1. Give a polynomial-time verifier for the decision version of the Longest Path problem: Does graph G have a path of length at least k?

2. Give a polynomial-time verifier for the Subset-Sum problem: Given a set of numbers S and a target sum t, is there a subset of S which sums to t?

2 Complexity Classes

We can divide the set of all computing problems into classes based on how efficient their best solutions are.

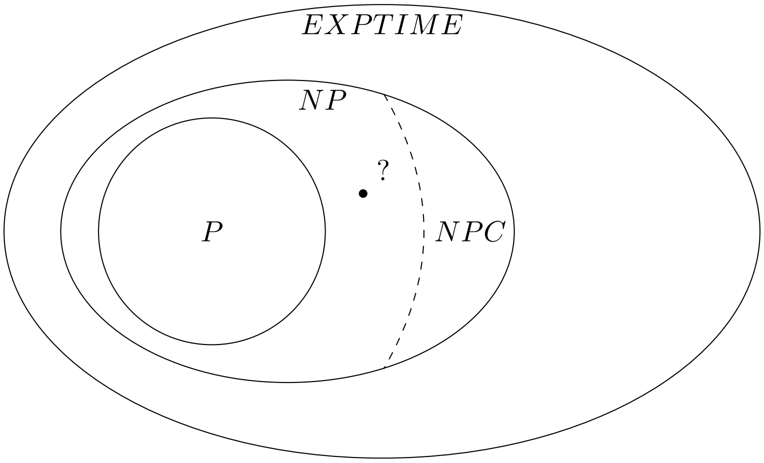
- P: The set of all problems with polynomial-time solutions.
- NP: The set of all problems with polynomial-time verifiers. Alternately, NP is the set of all problems solvable in polynomial time using a non-deterministic computer, which gives the class its name: $\underline{\text{N}}$on-deterministic $\underline{\text{P}}$olynomial.

Aside: NP does **not** stand for "Non-Polynomial". In fact, $P \subseteq NP$, so NP contains all problems with polynomial solutions. This is one of those silly "Are you a 'real' computer scientist?" trivia points where it is easy to misuse a term, which might come up in an interview.

The problem is, we have now divided computational problems into two classes, P and NP. We have shown that at least some problems which appear to require exponential time are in NP, as they are verifiable in polynomial time. So $P \neq NP$, right? Well...we think so. But no one has figured out how to prove that the two classes are not equal, so it is technically still possible, as far as we know, that $P = NP$. If they are equal, then it turns out that many apparently hard problems are actually efficiently solvable, such as decrypting many of our most common encryption systems. Oops!

Also, while it might seem that the real answer could be somewhere in between, there are certain problems, known as *NP-Complete*, which are the hardest problems in NP. We know that if we can show that **any** NP-Complete problem is solvable in polynomial time, then all NP problems are also solvable in polynomial time.

We can draw this complexity hierarchy as a Venn diagram. If the point marked "?" exists, then $P \neq NP$. If any problem in NPC is in P, then $P = NP$. We also refer to problems in NPC and beyond NP as *NP-Hard*.



3 (Un)Decidability

As computer scientists, we tend to live with an implicit assumption: Every problem can be solved with a computer. Or, at least, reasonably-well defined problems, maybe restricted to those of a somewhat mathematical or analytical nature. We need formal models of computation to properly define what counts as a "problem", and thus what the set of "all problems" is.

Definition 4. A *Turing Machine* (TM) is a memory (*"tape"*) and a machine (*"automaton"*) that manipulates the memory according to a specific set of rules.

- The memory is a one-way infinite array. That is, there is a first position, but no last position.
- The machine reads one element of the array at a time.

- For a given state of the machine and a given value in the current memory location, the machine has exactly one next action, which can move to a new state, write a new value to that memory location, and move left or right to an adjacent memory location.

We say that we can *solve* a problem if there is a Turing Machine that halts on all inputs and accepts all inputs whose output should be True.

Note that this definition of computation only considers Boolean functions. That is actually sufficiently expressive, since we can use these solutions to build any more general solution.

Theorem 1 (Church-Turing Thesis)**.** *Turing Machines accurately capture the notion of computation.*

What this says is that any model we might define for "computation" is equivalent to TMs–we can solve a problem in that model if and only if we can solve it with a TM. Turing and Church proved that all then-known general models of computation (Turing Machines, Lambda Calculus, General Recursive Functions) were equivalent, and hypothesized that any other model would also be equivalent. That thesis has held true so far (excluding unreasonable models that allow things like infinite work in a single step).

Claim 1. *The implicit assumption of the general usefulness of computers is False.*

That is, not every computational problem is solvable. We will look at this in two ways. One is very abstract, but simple. The other is more roundabout, but shows a specific useful problem computers cannot solve. For the first proof, we need to use the concept of different sizes of infinity. The "smallest" infinity is *countably* infinite sets–we can assign an order to the elements such that there is a first, second, third, ... element. The important part is to guarantee that we will count any particular element in finite time. Then there are *uncountably* infinite sets, which cannot be counted this way, as any way you try to order them, you will miss some elements when counting. Uncountably infinite sets are larger than countably infinite sets, which means there is no surjection (onto function) from a countably infinite set to an uncountably infinite set.

Theorem 2. *Not all functions are computable.*

Proof 1.

Claim 2. *The set of all Turing Machines (programs) is countably infinite.*

Proof. First, note that every program description is finite (since algorithms must terminate in finite time). There are 2^n binary strings of length n, and we can represent any program in binary, so we can count all strings of length n, for any particular n. We can further count all strings of any length by first counting all strings of length 1, then all of length 2, and so on. Since there are finitely many strings of length less than n, we will count every string of finite length n in finite time.

We have thus counted every possible program, since every one is a binary string. The set of valid programs (TMs) is smaller, so it is also countable. We technically should argue that there are infinitely many valid programs, but we can see that by considering the family of programs which add a constant. There is a program which adds 1 to its input, one which adds 2, etc. to infinity. □

Claim 3. *The set of all functions $f : \mathbb{Z}^+ \to \mathbb{Z}^+$ is uncountably infinite.*

Proof. Assume in contradiction that this set of functions is countable. That is, we can assign a positive integer to each one, counting them as $f_1, f_2, f_3, \ldots$. We will now construct a function which we failed to count, showing that it is not possible to count all functions.

Define our new function $f' : \mathbb{Z}^+ \to \mathbb{Z}^+$ as $f'(i) = f(i) + 1$. Thus, for all $i \in \mathbb{Z}^+, f'(i) \neq f_i(i)$, so f' is not any of the functions we counted.

Aside: This is a use of a general technique called *Cantor Diagonalization.*

□

Any uncountable set is strictly larger than any countable set (we could match program p_i with function f_i, and show as above that some function did not get a matching program), so there is at least one function $f : \mathbb{Z}^+ \to \mathbb{Z}^+$ for which there is no program which computes it. □

This is a *non-constructive proof.* We know there is an uncomputable function, but we do not know what that function is, or whether it is important. We will next prove that a specific problem we would very much like to solve is uncomputable.

Proof 2. Consider the *Halting Problem*:

Input: Code for a program P and an input x for P.

Output: Boolean, True iff $P(x)$ finishes in a finite amount of time.

Note that x can be any binary string, up to and including code for a program. Having code as input to another program is normal: compilers, optimizers, debuggers, test code, browsers, etc. all take code as input. Further, it would be very useful to be able to compute `Halt`(), as this is basically an infinite loop detector. Without it, we have to manually reason through our code, and sometimes make mistakes.

Now, assume that `Halt` is computable. Then we can use it as a subroutine to build other functions. Let D be the function

```
function D(M)
    $x = Halt(M, M)$
    if $x$ then
        while true do
        end while
    else
        return True
    end if
end function
```

> **Aside:** It is normal to run a compiler with its own code as input. When you first develop a new language, you have to compile it more or less manually, so one of the first programs you write tends to be a compiler, which you compile by hand, then you can build a more advanced compiler, which you compile with your first compiler, and so on, building better and more complicated compilers and eventually you will compile a compiler with itself to get a more optimized version of the executable. This is known as *bootstrapping*.

Consider the behavior of program `D`:

$$\mathtt{D}(M) = \begin{cases} \text{loop forever} & \text{if } \mathtt{M}(M) \text{ halts} \\ \text{halt and return} & \text{if } \mathtt{M}(M) \text{ loops forever} \end{cases}$$

Now, for no apparently sane reason beyond the fact we can, run `D` on itself! Consider its behavior:

$$\mathtt{D}(D) = \begin{cases} \text{loop forever} & \text{if } \mathtt{D}(D) \text{ halts} \\ \text{halt and return} & \text{if } \mathtt{D}(D) \text{ loops forever} \end{cases}$$

Now, `D`(D) halts if and only if it does not halt. This is a contradiction, so `D` cannot exist, which means that `Halt`() cannot exist, either. That is, there can never be a program on any computer which solves the Halting problem. We call such a problem *undecidable.* □

There are many undecidable problems, but one of the most important results is a generalization of the Halting problem. This theorem basically says that we cannot have nice things.

Theorem 3 (Rice's Theorem)**.** *Any non-trivial, functional property of programs is undecidable.*

Let us define these terms:

- "non-trivial" means that some programs have the property and some do not, so the question of whether a program has the property is interesting.
- "functional property of programs" means that we are talking about a property of the external behavior (function) of the program, not the specific code or implementation details. That is, any two programs which do the same thing will both have the property, or both not have it.
- "undecidable" means that we cannot compute whether or not a given piece of code has the property.

The Halting Problem is asking whether a piece of code has the property of always halting in finite time.

Exercise: Think of another useful property you would like to know whether some code you have written has or does not have.