COMPUTATION: DAY 7

BURTON ROSENBERG
UNIVERSITY OF MIAMI

Contents

1. Complexity

Associated with each deciding Turing Machine is a "run time" which the number of transitions taken during the computation. Complexity Theory uses an abstracted version of this step count to associate with problems a complexity. More complex languages require more steps to come to a decision whether the presented string is in the language.

A bound will be placed on the run time of the algorithm.

(1) The bound will be a function of the length of the input. Longer inputs are given more steps.
(2) The bound will be the maximum number of steps needed to decide a string, over all strings of a given length.
(3) The bound, a function $t(n)$ where n is the input length, will be categorized by its asymptotic order of growth (Big-Oh notation).

This still is only about this or that program that decides the language. Different programs might require different bounds. Those with significantly larger bounds will be considered less efficient, although still correct as a decision algorithm.

Date: Tuesday, April 22, 2025.

Definition 1.1. The *time complexity* of a language A is the function $t(n)$ of minimum order so that there is a TM with runtime $t(n)$ that decides A.

While this quantifies over all algorithms it does so in a certain model of computation. With this definition of complexity, the complexity of the problem is not truly a property of the language only — there is a dependence of the computational model.

We previously described a k-tape TM, and a simulation of a k-tape machine with a one tape machine. What this simulation result achieved was the show that the definition of computability was not changed by the additional abilities of a k-tape machine. However, the run time was. We showed that a k-tape machine of run time $O(f(n))$, when simulated by a 1-tape machine, that 1-tape machine ran in time $O(f(n)^2)$. So, to say a problem has complexity $O(n^d)$ will depend on whether we mean the complexity on a k-tape machine or on a 1-tape machine.

Likewise, we previously described the simulation of a non-deterministic TM with run time $O(f(n))$ by a deterministic machine of run time $O(2^{O(f(n))})$.

The exact order of run time is important, once the machine model is fixed. For this section of the course we would rather disregard some details of this order so as to understand the larger outline of complexity theory. We would like, as an example, to not be concerned about the difference between k and one tape machines, but retain the more substantial difference between deterministic and non-deterministic machines.

The Church–Turing hypothesis is that any intuitively computable function is computable on a Turing Machine. We will go on to say something more speculative. That any reasonable model of computation computes functions in a time polynomially related to the time to compute on the standard model one tape Turing Machine. We consider adding tapes to be a reasonable variant, but non-determinism to be unreasonable. This statement has been challenged (but not decimated) by the creation of Quantum Computing.

2. Algorithms

An algorithm is the idea of a program on a Turing machine or otherwise reasonably related model of computation. The notion of reasonably related is the strong Turing-Church hypothesis concerning the number of steps taken by the algorithm compared between the two computational models. While what is being counted is the number of steps, this is referred to as "time" as it is considered each step, if the model were physically run, takes a certain unit of time.

For a problem to fit into our schema, a problem (such as sorting) as instances (such as sorting a particular array of numbers). The problem is the membership problem for the set $A \subseteq \Sigma^*$ which encodes all instances, and identifies which $a \in A$ are accepted and which are not. The number of steps depends on the problem size,

```
def gcd_exp(x,y):
    for d in range(min(x,y),1,-1):
        if x%d==0 and y%d==0: return d
    return 1

def gcd_euclid(x,y):
    while y!=0: x,y = y,x%y
    return x
```

FIGURE 1. Greatest Common Divisor

$|a|$, and solving a problem in time t is to solve it in time $t(|a|)$,

$$a \in A \iff M_A(a; t(|a|)) = T$$

Example:
Two algorithms to calculate the greatest common divisor of two positive integers are given in Figure 1. They are both algorithms solving the problem, but `gcd_exp` runs in time exponential in the size of the inputs, the length in digits of values stored in variables `a` and `b`; where as `gcd_euclid`[1] has time no more than cubic in this size.

2.1. **Asymptotic Order of Growth.** You should be familiar with Big-Oh notation. It puts functions into a linear order according to how fast they grow, disregarding constant factors and behavior on small inputs.

Definition 2.1. Let $f(n)$ be a function on the naturals, $f : \mathbb{N} \to \mathbb{N}$. The order of $f(n)$ is the set,

$$O(f(n)) = \{\, g : \mathbb{N} \to \mathbb{N} \mid \exists\, c, n_o > 0, \forall n \geq n_o, c\, f(n) \geq g(n) \,\}$$

Customarily we write $g = O(f(n))$ when more formally we would write $g \in O(f(n))$.

Lemma 2.1. For functions $f, g, h : \mathbb{N} \to \mathbb{N}$,

- $f = O(f)$,
- If $g = O(f)$ and $h = O(g)$ then $h = O(f)$.
- If $g, h = O(f)$ then $g + h = O(f)$ and $g\, h = O(f^2)$.
- $0 = O(f)$, for any f.

Proof: Left as an exercise for the reader. □

[1]This algorithm is found in *Euclid's Elements*, (300 B.C.) and is called the Euclidean Algorithm.

3. The class P

Definition 3.1. An algorithm is called polynomial if its run time is of the form $O(n^d)$ for some non-negative integer d. The class P is the class of all problems that can be solved with a polynomial time algorithm on any reasonable model of computation.

The following properties of languages have polynomial-time algorithms that decide the property.

- The acceptance problem for regular languages.
- The emptiness problem for regular languages,
- The equality problem for regular languages.
- The acceptance problem for context free languages.

Theorem 3.1. Let $A \subseteq \Sigma^*$ be a regular language. The problem of determining if $a \in A$ is in the class P.

Proof: Since the language is regular, there is a FA which accepts the language. Write down a description of this language and simulate it on a TM on the input a. The TM will halt with a decision.

Since the FA is a fixed machine, the length of its description is fixed, and we consider inputs of length longer than the length of the description. Each character of the input causes a bounded number of machine steps in the simulator. Hence the run time is $O(n)$. □

Theorem 3.2. Let $\langle A \rangle$ be the description of a FA A. The set,

$$E_{FA} = \{ \langle A \rangle \mid \mathcal{L}(A) = \emptyset, \}$$

is in the class P.

Proof: We give a polynomial time algorithm to decide if machine A accepts any string at all. The algorithm searches backwards from all accept states to states that can lead to an accept state. Repeatedly iterate over all states. In the first iteration, mark the accept states; in the second and subsequent iterations mark any state with a transition to a marked state. When an iteration does not mark any states halt and return that the language is not empty if and only if the start state is marked.

The description will be of size proportional to the number of states. Each iteration will take states proportional to the number of states; and after as many iterations as states the algorithm must have marked all the states it ever can mark. So the algorithm runs in time $O(n^2)$, where n can be taken to be the number of states in A. □

Theorem 3.3. Let $\langle A \rangle$ and $\langle B \rangle$ be the descriptions of two FA 's. The set,

$$EQ_{FA} = \{ \langle A \rangle, \langle B \rangle \mid \mathcal{L}(A) = \mathcal{L}(B), \}$$

is in the class P.

Proof: We observe that,

$$\mathcal{L}(A_1) = \mathcal{L}(A_2) \Leftrightarrow \mathcal{L}(A_1 \oplus A_2) = \emptyset$$

The machine for $\mathcal{L}(A_1 \oplus A_2)$ can be constructed from $\langle A \rangle$ and $\langle B \rangle$, by the product construction which has as accepting states when only one of the machines would accept. Then we use the algorithm above to see if this machine accepts nothing. □

Theorem 3.4. Let $A \subseteq \Sigma^*$ be a context free language. The problem of determining if $a \in A$ is in the class P.

Proof: Since the language is context free, there is a Chomsky Normal Form which accepts the language. Since this is a fixed grammar, consider inputs of length longer than the length of the grammar. The CYK algorithm decides in time $O(n^3)$ in the length of the string. □

4. The class NP

The extended Church–Turing hypothesis kept as distinct those languages decided by a nondeterministic TM. The search for advice required exponential time in the length of the calculation, as all computation paths might be explored. This distinction continues in the case of polynomial algorithms, but considering problems that might not be solved in polynomial time, but give a solution, that solution can be verified correct in polynomial time. The connection with nondeterministic machines is that can consider this solution by guess. But we must be able to prove the guess correct.

Any P time algorithm is NP. Rather than guess we construct the answer, then verify our own answer. If the algorithm to construct the answer is correct there is no need to verify, but I present it this way just to continue in the format of an NP algorithm.

Definition 4.1. A language A is in NP if there is a polynomial time machine V taking two inputs and,

$$a \in A \iff \exists\, w \text{ s.t. } V(a, w) = T.$$

and w is polynomial length in the length of a.

An example of an NP problem is the problem if a Hamiltonian Circuit. Given a graph $G = \langle V, E \rangle$, a path between s and t, $s, t \in V$ is a sequence of edges in E that begins a s and ends at t, and subsequent edges share exactly one vertex. The problem of a Hamiltonian Circuit is given a presentation of G, s and t, is there are path from s to t that visits every vertex V in G exactly once.

Given a path, it is very quick, a polynomial of low degree in the description of the graph G to determine if the path goes from s to t, following edges in appropriately, and visiting each vertex exactly once. However, at this moment, no P time algorithm is know to in general find such a path.

Because one has a polynomial time verifier, disregarding time, there is a solution to Hamiltonian Circuit which follows a pattern common to all NP class problems. A Hamiltonian Circuit for a graph of n vertices will have $n-1$ edges. Write a program to enumerate all $n-1$ sized subsets from graph's set of edges. As they are being enumerated, test them one by one with the verifier. If and when the verifier accepts, the solution has been found. If the enumeration completes with no proposed set of $n-1$ edges being a Hamiltonian Circuit, then reject the graph, as it has no Hamiltonian Circuit.

The string w can be called the witness. The class NP is the class of problems for which a solution has a polynomial sized witness. The witness is a membership proof.

5. NP Completeness

All P languages are in NP. Whether there are NP languages not in P is an open problem. Most people think that there are. That these language classes are distinct. The progress to solving the P versus NP problem involves the theory of *NP completeness*. In this theory a reduction is defined that organizes NP problems into those which are at least as hard, and then shows that this ordering has "hardest problems" in NP. Those hardest problems are called NP complete, and under this ordering they are known to be all as hard as each other.

5.1. **Polynomial Reduction.**

Definition 5.1. A *polynomial time function* $f : A \to B$ is a Turing machine (in any reasonable mode of computation) that when started with an $a \in A$ on the time, halts in time $t(|a|)$ with only a $b \in B$ written on the tape, and t is of order $O(n^d)$ for some $d \in \mathcal{N}$.

Definition 5.2. A *polynomial time reduction* between two languages $A \subseteq \Sigma^*$ and $B \subset \Sigma^*$ is a polynomial time function $f : \Sigma^* \to \Sigma^*$ such that $f(A) \subseteq B$ and $f(\overline{A}) \subseteq \overline{B}$. It is denoted $A \leq_P B$.

Lemma 5.1. If $A \leq_P B$ and B can be solved in polynomial time, than A can be solved in polynomial time.

Proof: Because B is solvable in polynomial time, there is a polynomial time function $g : \Sigma^* \to \{ T, F \}$, that decides the set $B \subseteq \Sigma^*$. Let f the reduction. Then the composition function $g \circ f$ is a polynomial time function that decides A. $\square$.

Definition 5.3. A language $A \subseteq \Sigma^*$ is *NP complete* if,

(1) The language A is in NP, and
(2) for every language B in NP, there is a reduction $B \leq_P A$.

5.2. **Cook–Levin Theorem.**

Theorem 5.1 (Cook–Levin)**.** Given A, a language in NP, then $A \leq_P SAT$, where SAT is the general problem of finding a satisfying assignment to a boolean formula.

Proof: Given a problem in NP, there is a polynomial time Turing machine that decides it. From this machine and a given string s, in polynomial time a set of boolean variables can be defined that represent all stages of the computation of the machine, and equations written down that verify that the setting of the variables corresponds to a correct computation of the the machine which accepts or rejects s.

Therefore this problem of finding an assignment to boolean variables to satisfy boolean equations is universal for the class NP. It is also in NP, since a satisfying assignment, if it exists, can be verified in polynomial time. □

It was more specifically shown that the collection of boolean equations can be collected into a single equation in what is known as 3 Conjunctive Normal Form. In this form the problem is called *3-SAT*, so the statement is: 3-SAT in NP Complete.

We then showed that 3-SAT can be reduced to k-CLIQUE and the HAMILTONIAN-PATH. So both these are also NP Complete problems. The reductions are clever and in a certain sense very direct.

Theorem 5.2. SAT, k-CLIQUE and k-COVER are NP-complete.

Proof: A language is NP complete is it is an NP language, and any NP language can be reduced the that problem. The Cook-Level theorem shows SAT is NP complete. It can be shown (not done here) that any SAT can be written as a 3-SAT in polynomial time and (consequently) without more than a polynomial increase in size. . Hence,

$$\text{SAT} \leq_P \text{3-SAT}$$

By the properties of polynomial reductions, 3-SAT is NP complete. 3-SAT is reducible to the other languages in the theorem statement, and hence as also NP complete. □

A problem such that an NP language can reduce to it, but the problem is not necessarily in NP is called *NP hard*.