COMPUTATION: DAY 7

BURTON ROSENBERG
UNIVERSITY OF MIAMI

Contents

1. Complexity

Associated with each Turing Machine is a "run time", most easily associated with the number of transitions taken in the calculation. 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.

Various definitions lead us from a fact of one particular Turing Machine on one particular input to a conclusion about the complexity of the language.

(1) The step count is calculated as the maximum number of steps needed to decide, given an input length n.
(2) The step count, as a function of n, is considered only for the order of growth of the function.

This still is only about programs, not about language. Different programs with different run times can recognize the same language.

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.

The definition of complexity indeed ends here. And we have languages that have complexity $O(n)$, or $O(n^2)$, for instance. However, 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 Turing Machine 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 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, there must be an infinite number of problem instances, and each must be stated as a string over the language alphabet, whose string length is important in calculating the run time. The number of steps can depend on exactly the stated instance. The step bound is the maximum number of steps for any instance of a given size. This is a maximum over a finite set. If the bound is $t(n)$, then the an algorithm A is a Turing machine M_A such that the

language is,

$$\mathcal{L}(A) = \{\, x \mid M_A(x; t(|x|)) = T \,\}$$

Here are two algorithms to calculate the greatest common divisor of two positive integers,

```
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
```

They are both algorithms, but they have very different run times. We will associate with the language of greatest common divisors the run time of the Euclidean algorithm, the second code block. In fact, this code is the fastest possible known calculation of the GCD when the run time is stated as a big–oh function class.

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. □

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.

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. Polynomial Reduction

Definition 4.1. 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 4.2. 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 4.1. 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. $\square$.

5. 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 5.1. A language A is in NP if there is a polynomial time machine V taking two inputs and,

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

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.

6. NP Completeness

Definition 6.1. A lanuage $A \subseteq \Sigma^*$ is *NP complete* if,

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

Theorem 6.1 (Cook–Levin)**.** There exists NP complete languages.

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.

7. The Class BPP

Another viewpoint of NP is when the witness string, the string that helps the verifier determine if a string is in the language or not, is a randomly chosen string.

The computation is then a random outcome depending on the choice of the witness string. Furthermore, we loosen the requirement, so that the verifier need not always be correct in its determination. We allow error but require that in a middle majority for the choices of the witness string, the machine does decide correctly.

Definition 7.1. A problem ins in BBP if thre is a two input machine

$$T : \Omega \times \Sigma^* \to \{ T, F \}$$

that that decides a language $A \subseteq \Sigma^*$ by,

(1) For $a \in A$, the probability that $T(\omega, a) = T$ is greater than $2/3$, where the probability is over the randomly chosen string $\omega \in \Omega$,
(2) For $a \notin A$, the probability that $T(\omega, a) = T$ is less than $1/3$, where the probability is over the randomly chosen string $\omega \in \Omega$,

BPP problems are problems that we might not yet know a way to solve them in polynomial time, but can be verified in polynomial time, and in addition, any random string is likely a witness.

An important example of such a problem is the testing if an integer is prime. Actually, it is known that this is in P. However, the algorithm is impractical, and what is important in practice that it is in BPP.

To know the principle of the algorithm might help elucidate what is BPP. Given an $n \in \mathbb{Z}$, consider the modular arithmetic system $\mathbb{Z}_n$. This system has very different properties if n is a prime or not, and those differences are evident "everywhere". Let $w \in \mathbb{Z}_n$ be randomly chosen. The probability is at least a $1/2$ that if n is not a prime, then,

$$w^{n-1} \neq 1 \pmod{n}.$$

Hence a randomly drawn w is likely to be a witness of non-primality, when n is not prime. This equation is never true if n is. a prime. The algorithm has no witness for primality, what is has is after repeated choices of w, none is a witness to non-primality.