Computability and Complexity

Lecture 1, Tuesday August 1, 2023

Ari Feiglin

There are two main problems which we will focus on in this course: search problems, and decision problems. Search problems are problems where our goal is to find a solution or an answer to a question. Decision problems are problems where our goal is to verify if a given solution or answer to a question is valid (more generally if we should accept or reject a given input).

Examples of search problems are:

(1) Given a graph G and two vertices s and t, find a path from s to t in G.

(2) Given a graph G, find a three-coloring of the vertices (ie. a function $\sigma\colon V \longrightarrow \{1,2,3\}$ such that if $(v,u) \in E$ then $\sigma(v) \neq \sigma(u)$).

(3) Given a boolean formula in conjunctive normal form (of the form $\bigwedge_{i=1}^{n} \bigvee_{j=1}^{m} \varepsilon_{ij} x_j$ where ε_{ij} is either $\neg$ or nothing), find valuations for the variables $x_1, \dots, x_m$ which satisfies the boolean formula.

Search problems need not give solutions, if they do not exist.

Definition 1.1:
A **search problem** is specified by a relation R, where $(x,y) \in R$ if and only if y is a valid solution to x.

Definition 1.2:
If R is a relation, and x is an element of its domain, then we define

$$R(x) = \{y \mid (x,y) \in R\}$$

So in the above examples,

(1) For the finding paths problem, we have the relation

$$R_{st\text{-conn}} = \big\{\big((G,s,t),P\big) \bigm| G \text{ is a graph, } s \text{ and } t \text{ are vertices, and } P \text{ is a path from } s \text{ to } t \text{ in } G\big\}$$

(conn is short for connectivity.)

(2) For the three coloring problem, we have the relation

$$R_{3\text{col}} = \{(G,\sigma) \mid G \text{ is a graph, and } \sigma \text{ is a three-coloring of } G\}$$

(col is short for coloring.)

(3) For finding valuations for a CNF, we have the relation

$$R_{CNF\text{-sat}} = \{(\varphi,\vec{x}) \mid \varphi \text{ is a boolean formula in CNF, and } \vec{x} \text{ is a boolean vector which satisfies } \varphi\}$$

(CNF is short for conjunctive normal form, and sat is short for satisfiability.)

Definition 1.3:
A **decision problem** is specified by a set S where $x \in S$ if and only if x is a valid answer to the problem.

For example, we can ask the question "is the graph G three-colorable?" This is specified by the set

$$S_{3\text{col}} = \{G \mid G \text{ is three-colorable}\}$$

Definition 1.4:

An **algorithm** is a finite set of rules which defines a process in a computational model. Given an algorithm A, and an input x, we define $A(x)$ to be result of running A on the input x.

We say that an algorithm A solves a search problem R if for every x,

(1) If $R(x) \neq \varnothing$, then $A(x) \in R(x)$.

(2) And if $R(x) = \varnothing$, then $A(x) = \perp$ (the symbol for there being no solution).

And we say that A solves a deciscion problem S if

(1) If $x \in S$ then $A(x) = 1$.

(2) And if $x \notin S$ then $A(x) = 0$.

An algorithm may be the definition of a regular automaton, a pushdown automaton, a context-free grammar, a turing machine, etc. But we can assume that an algorithm is a single-tape turing machine.

Definition 1.5:

Let M be a turing machine. We denote $t_M(x)$ by the number of transitions M performs on the input x.

The **time complexity** of M is defined to be the function

$$T_M \colon \mathbb{N} \longrightarrow \mathbb{N}, \quad T_M(n) = \max\{t_M(x) \mid |x| = n\}$$

(recall that inputs to turing machines are finite strings.)

Definition 1.6:

We say that a turing machine M **runs in polynomial time** if there exists a polynomial p such that for every n, $T_M(n) \leq p(n)$. In other words, the time complexity of M is bound by some polynomial.

And a search or deciscion problem is **solvable in polynomial time** if there exists a single-tape turing machine which solves it and runs in polynomial time.

We consider efficient solutions to be solutions which run in polynomial time. Note that a solution which runs in n^{100} time is still considered efficient for the sake of this course, but in practicality it is of course not.

Note:

A turing machine runs in polynomial time if and only if $T_M(n) \in O(n^d)$ for some $d \in \mathbb{N}$. This is true since $T_M(n) \in O(n^d)$ if and only if there exists a $c > 0$ such that for every $n \geq n_0$, $T_M(n) \leq cn^d$. If we let $A = \max\{T_M(n) \mid n < n_0\}$, and then $T_M(n) \leq cn^d + A$, which means M runs in polynomial time.

And if M runs in polynomial time, then $T_M(n) \leq \sum_{k=0}^{d} a_k x^k$, and so $T_M(n) \in O(n^d)$.

Note:

Notice that the requirement of the turing machine to be single-tape may be significant. For example, given the deciscion problem $\mathsf{Palindrome} = \{x \mid x^R = x\}$, with a doube-taped turing machine we can solve it in $\Theta(n)$ time. We do this by copying x onto the second tape, then comparing the characters at each head and moving them left and right respectively until they reach the end of the string.

With a single-taped turing machine we need to move from the beginning to the end of the string over and over, and this takes $\Theta(n^2)$ time.

Conjecture 1.7 (Cobham-Edmonds Thesis):

Any problem which can be solved in $T(n)$ time on a "feasible" computational model can be solved in $p(T(n))$ time for some polynomial p with a single-taped turing machine.

"Feasible" here is not well-defined, but intuitively it means that the computational model doesn't do an extraordinary amount of computations at each step.

So by the **Cobham-Edmonds Thesis**, for a problem to be solvable in polynomial time, it is sufficient to provide a solution to it with any (feasible) computational model.

Definition 1.8:

We say that R is **polynomially bound** if there exists a polynomial p such that for every $(x, y) \in R$ then $|y| \leq p(|x|)$.

We define **PF** to be the class of all polynomially bound relations R such that R can be solved in polynomial time,

$$\mathbf{PF} = \{R \mid R \text{ is polynomially bound and } R \text{ can be solved in polynomial time}\}$$

PF is short for polynomial-find.

Note that $R_{st\text{-con}}$ is not necessarily in **PF**, as it is not polynomially bound. Since if there exists a cycle in G, then we can find arbitrarily large paths in G. If we require that solutions be simple paths, then the length of the path is less than $|E|$ and thus is polynomially bound. And we know we there exist polynomial time algorithms to solve $R_{st\text{-con}}$, so it is in **PF**.

The question of whether $R_{3\text{col}} \in \mathbf{PF}$ is open. It is obviously polynomially bound (since solutions have a size of n^3), but we do not know if there exists a solution which runs in polynomial time or not.

Definition 1.9:

We define **PC** to be the class of all polynomially bound search problems R such that there exists an algorithm A which runs in polynomial time and verifies solutions to R.

$$\mathbf{PC} = \left\{ R \;\middle|\; \begin{array}{c} R \text{ is polynomially bound, and there exists an algorithm } A \text{ which runs in polynomial time} \\ \text{such that for every } (x, y),\ A(x, y) = 1 \text{ if and only if } (x, y) \in R \end{array} \right\}$$

PC is short for polynomial-check.

Intuitively, we may think that a problem which we can easily find a solution for (in **PC**) would also be easy to verify a solution for. But this is not the case, because to verify solutions we must deal with *all* possible solutions, and to find a solution we need only find one.

Proposition 1.10:

$\mathbf{PF} \nsubseteq \mathbf{PC}$

Proof:

Let us define

$$R = \big\{\big((M, \omega), !\big) \;\big|\; M \text{ is a turing machine which halts on the input } \omega\big\} \\ \cup \big\{\big((M, \omega), ?\big) \;\big|\; M \text{ is a turing machine, and } \omega \text{ is any input}\big\}$$

$R \in \mathbf{PF}$ since we can simply define the algorithm A which returns ? on every input.

Now, $R \notin \mathbf{PC}$ since if $R \in \mathbf{PC}$ then suppose A is an algorithm where $A((M, \omega), \sigma) = 1$ if and only if $((M, \omega), \sigma) \in R$. Then we can define the algorithm B whose input is (M, ω) and returns $A((M, \omega), !)$. But then B decides if M halts on ω, which contradicts the halting problem being undecidable. ■

Now, is $\mathbf{PC} \subseteq \mathbf{PF}$, ie. if we can easily verify a problem, can we solve it? This is an open question.

Definition 1.11:

PC and **PF** relate to search problems. For decision problems, we define the class **P** to be the class of all search problems S which can be solved in polynomial time,

$$\mathbf{P} = \{S \mid S \text{ can be solved in polynomial time}\}$$

P is the equivalent of **PF** for deciscion problems.

But it is not immediately clear how to define the equivalent of **PC** for deciscion problems. After all, how do you verify a solution to a deciscion problem?

Definition 1.12:

A **verifier** for a decision problem S is an algorithm V such that

(1) V is entire: for every $x \in S$, there exists a y such that $V(x, y) = 1$.

(2) V is reliable: for every $x \notin S$ and for every y, $V(x, y) = 0$.

We say that S has a **polynomial proof system** if it has a verifier V which runs in polynomial time and there exists a polynomial p such that if $x \in S$ then there exists a y such that $|y| \leq p(|x|)$ and $V(x, y) = 1$.

Note that if V is a verifier, then $V(x, y) = 0$ does not mean $x \notin S$, it just means that x may not be in S. If we run $V(x, y)$ for every y and we get zero, only then do we know that $x \notin S$. Conversely, if $V(x, y) = 1$ then $x \in S$.

Definition 1.13:

We define $\mathbf{NP}$ to be the class of all decision problems which have a polynomial proof system,

$$\mathbf{NP} = \{S \mid S \text{ has a polynomial proof system}\}$$

$\mathbf{NP}$ is the parallel to $\mathbf{PC}$ for decision problems.

Note that $S_{3\text{col}} \in \mathbf{NP}$ since we can take the verifier V which accepts (G, σ) if and only if σ is a three-coloring of G (which is verifiable in linear time). Since σ has a size on the order of n^3, $S_{3\text{col}}$ has a polynomial proof system.

Proposition 1.14:

$\mathbf{P} \subseteq \mathbf{NP}$.

Proof:

Let $S \in \mathbf{P}$, suppose A solves S. Then we define the algorithm V where $V(x, y) = 1$ if and only if $A(x)$. So if $x \in S$ then $A(x) = 1$ and so $V(x, y) = 1$ for every y. And if $x \notin S$, then $A(x) = 0$ so for every y, $V(x, y) = 0$. Thus V is a polynomial proof system for S, meaning $S \in \mathbf{NP}$ as required. ■

Now comes the biggie question, does

$$\mathbf{P} = \mathbf{NP}?$$

This is a major open question in computer science.