RECURSIVELY ENUMERABLE SETS AND THE CHURCH-TURING THESIS

DAVID WHITE

Mention *Introduction to the Theory of Computation* by Michael Sipser

I want to encourage grad students to take computer science courses as I've already seen how useful it is in applying for jobs in industry and jobs in academia. It's also useful for doing research mathematics as any field can have the word "computational" tacked on to the beginning to make a new and hot field. The 4 core CS courses are 211 (Intro Programming), 212 (Data Structures), 312 (Algorithms), and 301 (Theory). The last two have NO PROGRAMMING so grads should definitely consider them.

IBM, Google, Wall Street (Pollack), Scheduling (e.g. airlines), and NSA all want to hire math people with some CS skills. The PhD in math shows you can do independent work at a high level. The CS means they don't have to train you and they think your work will apply very easily to their problems. It also suggests you might be able to come up with NEW ALGORITHMS and they really want this. I could mention dynamical systems, coding theory, probability/measure theory, finite fields, graph theory, etc.

1. Definitions

A **language** is a set of words (finite strings of letters and symbols) taken from an alphabet A. We denote the set of all possible words over A by A^* (if A is finite, then A^* is countable). A formal language is often defined by means of a formal grammar or as the set of all possible words outputted by some machine.

Examples: $\{0,1\}^*$, $L_1 = \{w \mid w \text{ is the empty string or ends in a } 0\}$, $L_2 = \{0^p 1^p \mid p \in \mathbb{N}\}$

What is their expressive power? (Can formalism X describe every language that formalism Y can describe? Can it describe other languages?)

What is their recognizability? (How difficult is it to decide whether a given word belongs to a language described by formalism X?)

What is their comparability? (How difficult is it to decide whether two languages, one described in formalism X and one in formalism Y, or in X again, are actually the same language?).

Operations: union, intersection, complement, Kleene star (all words that are concatenations of 0 or more words in the original language), Reversal, etc

A language is **regular** if it is accepted by some DFA. These are used to define search patterns. The class of such languages is closed under union, concatenation, and star (the so-called **regular operations**). Expressions built up by applying these operations are called **regular expressions**, e.g. $(0 \cup 1)0^*$. To prove closure requires the notion of an NFA.

Our first theoretical model of how a computer works is the **finite automaton** or DFA. This is a 5-tuple: set of states, alphabet, start state, accept state, and transition function $\delta : Q \times A \to Q$.

Date: February 18, 2010.

A **deterministic** computation is one where every step follows uniquely from the preceding step. So a **Markov Chain** is the probabilistic analog of a DFA.

Do some examples, e.g. accepting L_1. The machine takes an input and uses it to follow arrows. If it's in the accept state when the string ends, then it accepts that string. Even with only finitely many states it can accept infinite L

DO EXAMPLE HERE FROM Page 38

2. Non-determinism

A **nondeterministic** machine is one where several choices may exist for the next state at any point. A **nondeterministic finite state automata** or NFA is a DFA which allows the empty transition ϵ. So the key is $\delta : Q \times A_\epsilon \to \mathcal{P}(Q)$

The computation following this machine must split and follow all copies in parallel. If the next input symbol doesn't appear on a given pathway then that copy of the machine dies. If any copy reaches an accept state, the NFA accepts an input string. Theorem: L is accepted by a DFA iff L is accepted by an NFA.

DO EXAMPLE HERE FROM Page 51

3. Proving a language is not regular

Lemma 1 (The Pumping Lemma)**.** *If L is regular then there is some $p \in \mathbb{N}$ such that for any string s of length $< p$, $s = xyz$ for $|y| > 0, |xy| \leq p$, and for all $i \geq 0$ $xy^iz \in L$*

O^p1^p is not regular. 3 cases are $1 \notin y$, $0 \notin y$, and both are in y.

4. Context-Free Grammars

A language is context-free iff it is accepted by a **pushdown automata**, i.e. an NFA which has a stack (last in, first out) as in a cafeteria.

Example: $L_2 = \{0^n1^n \mid n \in \mathbb{N}\}$. Also $a^ib^jc^k$ where $i = j$ or $i = k$

DO EXAMPLE HERE FROM Page 115

There is a pumping lemma in this context also. $\{a^nb^nc^n \mid n \in \mathbb{N}\}$ is not context free.

5. Recursively Enumerable Languages

A Turing machine consists of:

1. A TAPE of cells, each containing a symbol from some finite alphabet Γ. The alphabet contains a special blank symbol and the tape is assumed to be arbitrarily extendable both ways. Cells that have not been written to before are assumed to be filled with the blank symbol.
2. A HEAD that can read and write symbols on the tape and move the tape left and right one cell at a time.

(3) A finite TABLE (a.k.a. transition function) of instructions that, given the current state q_i and the symbol a_j currently under HEAD, tells the machine to do the following in sequence: either erase or write a symbol, move the head (left, right, or not at all), and take on the new state. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$

(4) A STATE REGISTER that stores the state of the Turing table, one of finitely many. There is one special start state.

There are theorems that having many tapes doesn't give you any extra power over one tape. Also, non-determinism doesn't help you (i.e. $\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$. Allowing random-access Turing machines does get you more power. This is well-studied (e.g. Sipser).

DRAW PICTURE HERE AND DO EXAMPLE OF $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ with markings.

A set $S \subset \mathbb{N}$ is called **recursively enumerable**, computably enumerable, semidecidable, provable or Turing-recognizable if there is an algorithm that, when given an input number, eventually **halts if and only if** the input is an element of S. Or, equivalently, if there is an algorithm that enumerates the members of S. That means that its output is simply a list of the members of $S : s_1, s_2, s_3, \ldots$. If necessary, this algorithm may run forever.

Here is the Chomsky hierarchy:

Type-0 grammars (unrestricted grammars) include all formal grammars. They generate exactly all languages that can be recognized by a Turing machine. These languages are also known as the **recursively enumerable languages**. Note that this is different from the recursive languages which can be decided by an always-halting Turing machine.

Type-1 grammars (context-sensitive grammars) generate the context-sensitive languages. These grammars have rules of the form $\alpha A \beta \to \alpha \gamma \beta$ with A a nonterminal and α, β, γ strings of terminals and nonterminals. The strings α and β may be empty, but γ must be nonempty. The rule $S \to \epsilon$ is allowed if S does not appear on the right side of any rule. Equiv, languages that can be recognized by a nondeterministic Turing machine whose tape is bounded by a constant times the length of the input.

Type-2 grammars (context-free grammars) generate the context-free languages. Context-free languages are the theoretical basis for the syntax of most programming languages.

Type-3 grammars (regular grammars) generate the regular languages. Regular languages are commonly used to define search patterns and the lexical structure of programming languages.

6. Church-Turing

Informally the ChurchTuring thesis states that if an algorithm (a procedure that terminates) exists then there is an equivalent Turing machine, recursively-definable function, or applicable λ-function, for that algorithm. A more simplified but understandable expression of it is that "everything computable is computable by a Turing machine." Though not formally proven, today the thesis has near-universal acceptance.

In the following, the words "effectively calculable" will mean "produced by any intuitively 'effective' means whatsoever" and "effectively computable" will mean "produced by a Turing-machine or equivalent mechanical device". Turing's 1939 "definitions" are virtually the same:

The thesis served to define effective calculability by identifying the equivalent notions of λ calculus and "general" recursion. These were proven to be equivalent and some call this equivalence the thesis. Later Turing added Turing machines into the mix and they were proven to be equivalent to the other two.

According to the Church-Turing thesis, any effectively calculable function is calculable by a Turing machine, and thus a set S is recursively enumerable if and only if there is some algorithm which yields an enumeration of S. This cannot be taken as a formal definition, however, because the Church-Turing thesis is an informal conjecture rather than a formal axiom.

This is unsolvable: There is no generalized "effective calculation" (method, algorithm) that can determine whether or not a formula in either the recursive- or λ-calculus is "valid"

A set $S \subset \mathbb{N}$ is called **recursive**, computable or decidable if there is an algorithm which terminates after a finite amount of time and correctly decides whether or not a given number belongs to the set. A set which is not computable is called noncomputable or undecidable. A subset S of the natural numbers is called recursive if there exists a total computable function f such that $f(x) = 1$ if $x \in S$ and $f(x) = 0$ if $x \notin S$.

Examples: Any finite or cofinite set by just checking equality. The set of prime numbers is computable.

A set is recursive if and only if it is either the range of an increasing total recursive function or finite. Every recursive set is recursively enumerable, but it is not true that every recursively enumerable set is recursive.

Theorem 1. *A set A is a recursive set if and only if A and the complement of A are both recursively enumerable sets.*

The simple sets are recursively enumerable but not recursive. $S \subset \mathbb{N}$ is simple if $\mathbb{N} - S$ is infinite and contains no infinite recursively enumerable set, and if S is recursively enumerable.

A function is **computable** if it can be calculated using a mechanical calculation device given unlimited amounts of time and storage space. Equivalently, any function which has an algorithm is computable. Formally, $f : A^* \to A^*$ is computable if some Turing machine halts on every w with just $f(w)$ on its tape. These functions are the main object of the theory of computation. We define them as those finitary partial functions on the natural numbers that can be calculated by a Turing machine. There are many equivalent models of computation that define the same class of computable functions. These models of computation include Lambda calculus. The set S is recursive if and only if the indicator function 1_S is computable.

A machine that always halts is called a **decider**. This is a Turing machine that halts for every input. Because it always halts, the machine is able to decide whether a given string is a member of a formal language. The class of languages which can be decided by such machines is exactly the set of **recursive languages**. However, due to the Halting Problem, determining whether an arbitrary Turing machine halts on an arbitrary input is itself an undecidable decision problem.

A problem must have a yes or no answer. These can be encoded as languages, e.g. the acceptance problem for DFAs is whether or not a given DFA accepts a given string. This is $A_{DFA} = \{\langle B, w\rangle \mid B$ is a DFA which accepts $w\}$. This is decidable because a Turing Machine can simulate a DFA and so it can run on w to see if it's accepted.

The following blew my mind and made me want to study math when I was 18.

Theorem 2 (Halting Problem)**.** *Determine, given a program and an input to the program, whether the program will eventually halt when run with that input.*

Decidable problems: acceptance for a DFA, NFA, or CFG. Emptiness Testing.

Theorem 3. *A general algorithm to solve the halting problem for all possible program-input pairs cannot exist*

Proof. First, clearly some languages are not Turing recognizable because there are only countably many Turing machines but there are uncountably many languages. Suppose H is a decider for HALT, i.e.

$$H(M,w) = \begin{cases} \textit{accept} \text{ if } M \text{ accepts w} \\ \textit{reject} \text{ if } M \text{ does not accept w} \end{cases}$$

For a Turing machine M let $\langle M \rangle$ be the description of M. Consider the Turing machine $D =$ "On input M run H on $(M, \langle M \rangle)$ and output the opposite of what H outputs"

$$D(\langle M \rangle) = \begin{cases} \textit{accept} \text{ if } M \text{ does not accepts } \langle M \rangle \\ \textit{reject} \text{ if } M \text{ accepts } \langle M \rangle \end{cases}$$

Plugging in $M = D$ (i.e. running this new algorithm on itself) yields a contradiction.

$$D(\langle D \rangle) = \begin{cases} \textit{accept} \text{ if } D \text{ does not accepts } \langle D \rangle \\ \textit{reject} \text{ if } D \text{ accepts } \langle D \rangle \end{cases}$$

□

Theorem: A language is decidable iff it is Turing-recognizable (recursively enumerable) and co-Turing-recognizable.

7. Further Topics in COMP501

P vs. NP. P is the class of all decision problems (i.e. languages) which can be solved by a deterministic Turing machine using a polynomial amount of computation time, or polynomial time. NP is the set of all decision problems for which the 'yes'-answers have simple proofs of the fact that the answer is indeed 'yes'. More precisely, these proofs have to be verifiable in polynomial time by a deterministic Turing machine. In an equivalent formal definition, NP is the set of decision problems solvable in polynomial time by a non-deterministic Turing machine. So clearly $P \subset NP$

$$\text{NP} = \bigcup_{k \in \mathbb{N}} \text{NTIME}(n^k)$$

A verifier is an algorithm V such that $A = \{w \mid V \text{ accepts } (w,c) \text{ for some string } c\}$. We call c the certificate. HAMPATH, CLIQUE, COMPOSITE, 3SAT are NP.

The question of whether $NP \subset P$ is a HUGE open problem. It's a Clay Institute 1 million dollar question. The problem has been reduced to showing that any NP-complete problem lies in P. To understand "complete" we need the following:

Given two subsets A and B of N and a set of functions F from N to N which is closed under composition, A is called **reducible** to B under F if $\exists f \in F$ such that $\forall x \in \mathbb{N}$, $x \in A \Leftrightarrow f(x) \in B$. We write$A \leq_F B$. Note that if A reduces to B and B is decidable then A is decidable. Same goes for recursively enumerable.

Let S be a subset of $P(\mathbb{N})$ and $\leq$ a reduction, then S is called closed under $\leq$ if $\forall s \in S$ and $\forall A \in P(\mathbb{N})$ we know that $A \leq s \Rightarrow A \in S$

A subset A of $\mathbb{N}$ is called hard for S if $\forall s \in S$ we know that $s \leq A$. A subset A of $\mathbb{N}$ is called complete for S if A is hard for S and A is in S.

So to prove P=NP you just need to find some NP-complete problem and find a deterministic polynomial time algorithm to solve it. Note that Factoring is hard (RSA, cryptography, etc) but it's polynomial. Minesweeper is NP-complete. The first known NP-complete problem was Boolean satisfiability (SAT): determining if the variables of a given Boolean formula can be assigned in such a way as to make the formula evaluate to TRUE. It's NP-completeness is the Cook-Levin Theorem (see Sipser). The proof is two parts. First give a polynomial time verifier (easy). Next give a deterministic polynomial time reduction from any NP problem to SAT. For each input, I, specify a Boolean expression which is satisfiable if and only if the machine M accepts I.

Reduction is also used to prove certain languages are undecidable since a decider for them would solve the Halting Problem. There are many other complexity classes. Here are some known relationships for the big ones. There are also unknown relationships and many others I didn't list.

$$\text{L} \subseteq \text{AL} = \text{P} \subseteq \text{NP} \subseteq \text{PSPACE} \subseteq \text{EXPTIME}$$

The time hierarchy theorem states that

$$\text{DTIME}\big(f(n)\big) \subsetneq \text{DTIME}\big(f(n) \cdot \log^2(f(n))\big)$$

The space hierarchy theorem states that

$$\text{DSPACE}\big(f(n)\big) \subsetneq \text{DSPACE}\big(f(n) \cdot \log(f(n))\big)$$