SE 3310b: Theoretical Foundations of Software Engineering

Overview of Topics Covered

Part 1: Formal Languages

Formal language:

- A set of strings over some alphabet
- Computation is modelled as deciding if a given string x is a member of a given language, i.e., Yes or no: string x is a member of language L?
- Turing-decidable languages are those where a correct 'yes' and 'no' answers are always computable. Turing machine always halts.
- Turing-recognizable languages are those in which correct 'yes' answers are always computable. Turing machine always halts on 'yes' answers. May halt and reject, or loop otherwise.

Hierarchy of formal languages:

1. **Recursively enumerable/recognizable** (languages recognized by some TM)
2. **Recursive/decidable** (languages decided by some TM)
3. **Context-free** (languages recognized by some PDA, or generated by some CFG)
4. **Regular** (languages recognized by some DFA/NFA, or generated by some RE)

Regular Languages

1. Deterministic Finite Automata (DFA)
2. Non-deterministic Finite Automata (NFA)
3. Regular Expressions (RE)
4. Equivalence of models
5. Pumping Lemma for regular languages

Context-free Languages

1. Context-free Grammars (CFG)
2. Pushdown Automata (PDA)
3. Equivalence of models
4. Pumping Lemma for context-free languages

Suggested exercises (Sipser):

- DFAs/NFAs: 1.5 (a,b). 1.6 (a,f,g), 1.7 (a,e,f)
- Regular expressions: 1.20 (a,c,d,h)
- Pumping lemma: 1.29 (a, b, c). See also examples 1.74, 1.76.
- CFGs/PDAs: 2.4 (a,d,e), See also example 2.14, 2.18

Recursively-enumerable Languages

1. Turing machine (TM)
    - Finite state machine + read/write head + unbounded tape/memory
    - At each step: read from tape, update state, optionally write to tape, move head left or right. Continues until:
        - It reaches an accept state, then it halts and outputs *accept*
        - No next state is explicitly defined for the current state/symbol, then it halts and output *reject*
    - TMs either accept, reject or loop forever
2. Turing machine variants:
    - Deterministic Turing machine
        - One tape, one computational path
    - Multi-tape Turing machine
        - Multiple tapes, one computational path (polynomially faster)
    - Probabilistic Turing machine (PTM)
        - Only follows one of potentially many possible computational paths
        - Selects each path with a certain probability (e.g., flips a coin)
    - Non-deterministic Turing machine (NTM)
        - Can explore many computational paths simultaneously
        - Accepts if *any* path accepts

3. Computational equivalence of models
    - They all decide/recognize the same languages, the difference comes down to speed/efficiency.

Suggested exercises (Sipser):

- Turing machines: 3.8 (a, b).
- See also Example 3.7, 3.9

Part 2: Computability

Model of Computation

- Simulation
    - A Turing machine M can be encoded as a string < M >. It is possible to construct another Turing machine M' which can accept the string < M,w > as input and *simulate* the execution of M on input w, i.e., M' executed on < M,w > gives the same result as M executed on w.

- Universal Turing Machine (UTM)
    - Can simulate any Turing machine M on input w.
    - UTM is a "universal computer," M is a "program," w is the "data"

- Effective Method
    - Turing's intuitive notion of an *algorithm*: must take finite time to reach an answer, the answer should be correct for any input, a person with simple instructions, pencils and paper could perform the computation, etc.

- Effectively Calculable
    - Anything that can be computed using an effective method (i.e., anything algorithmically computable)

- Church-Turing Thesis
    - Claim: Any computational system you can come up with cannot decide more languages than what can be decided by Turing machines

- Turing Completeness
    - A computing system is Turing complete if it can be used to simulate any TM. For example, the C programming language is Turing Complete (as shown in class).

Undecidability

- Recognizability vs. decidability
  - Decidable languages
    - A TM machine M *decides* language L if, for all input strings w, M accepts if (w in L) and rejects if (w not in L).
  - Recognizable languages
    - A TM machine M *recognizes* language L if, for all input strings w, M accepts if (w in L)
  - Decider: A TM that decides a language
  - Recognizer: A TM that recognizes a language
- Example Turing-Undecidable languages
  - Acceptance problem for Turing machines A_TM
    - Given < M,w > will Turing machine M accept string w?
    - Proved undecidable with Cantor's diagonalization method
  - Halting problem
    - Given < M,w > will Turing Machine M halt on string w?
    - Proved undecidable by reducing A_TM to Halt.
- Example Turing-Unrecognizable languages
  - Complement of A_TM
    - Given < M,w > will Turing machine M *not* accept w?

Suggested exercises (Sipser):

- Decidability: 3.15, 3.16.
- See also Examples 4.1, Figure 4.10, Theorem 4.22

Part 3: Complexity

Concepts

- Running time
  - The maximum number of steps a Turing machine takes to halt for any input of a

given size
    - Polynomial time: running time of a DTM grows as a polynomial function of the input size
    - Exponential time: running time of a DTM grows as an exponential function of the input size
    - Non-deterministic polynomial time: running time of an NTM grows as a polynomial function of the input size

- **P** vs. **NP** and the $1M question
- Polynomial-time reductions
    - Using a solution to problem B as a subroutine for a solution to problem A
    - A <=_P B if:
        - There exists a function f that can turn the question "is string s in A?" into the question "is string t in B?" such that
        - For all s and t=f(s) it is the case that (s in A)=True if and only if (t in B)=True
        - The function f executes in polynomial time
        - Informally a polynomial-time reduction is the process efficiently of turing instances of A problems into instances B problems.
        - Useful because it places an upper bound on how hard "A" problems can be: If I can efficiently turn an A problem into a B problem, then solving an A problem is no harder than solving a B problem.

- Cook-Levin Theorem
    - SAT is **NP**-Complete

- Converting optimization problems into decision versions problems
    - Travelling salesman: Given weighted graph G, find the lowest cost cycle
        - NP-Hard, but not NP-complete since it's not a decision (i.e., yes/no) problem

    - Decision version of Travelling salesman: Given weighted graph G and cost c, decide if G contains a cycle with cost less than c.
        - NP-Hard and NP-complete since (a) A polynomial time reduction exists from SAT to this problem (proof exists, but not shown in class), and (b) is clearly an NP problem: Is clearly a yes/no problem, and 'yes' answers can be efficiently verified

- Quantum computing
    - Superposition and probabilistic nature of q-bits
    - Quantum circuits as Turing-complete computing model

Complexity Classes

- The class **P**
    - Languages that can be decided by a DTM in polynomial time
    - Informally: *Problems that are efficient to solve*
- The class **NP**
    - Languages that can be decided by a NTM in polynomial time
    - Alternatively, problems that can be *checked* by a DTM in polynomial time. If a string s is in language L, there exists a polynomial algorithm A and auxiliary information c (called a "certificate") that can prove s is a member of L.
    - Informally: *Problems whose solutions are easy to check*
- The class **NP**-complete
    - The hardest problems in **NP**
    - A problem X is NP-complete if:
        1. X is in NP
        2. All problems in NP are polynomial-time reducible to X
    - SAT is in NP-complete (Cook-Levin theorem), so if SAT polytime reduces to problem X, then problem X is NP-complete.
    - Corollary: If X and Y are both NP-complete, then both X and Y are polynomial-time reducible to each other.
- The class **NP**-hard
    - Same as NP-complete except that problem X need not be in **NP**, i.e., not necessarily a polynomial-time verifiable decision problem.
    - Real-world applications include optimization problems
    - Also includes more exotic languages like the Halting problem (polynomial-time reduction from SAT to the Halting problem was show in in class).
- The class **BPP**
    - Problems that can be solved by a probabilistic Turing machine in polynomial time with bounded error
    - Clearly contains class P, a special case that doesn't utilize randomness and has no error).
    - Open question: Does BPP=P? One famous BPP problem that turned out to be in

P is FACTOR (Miller-Rabin is a probabilistic primality test, but Agrawal, Kayal, Saxena found a polynomial time algorithm in 2002).

- The class **BQP**
    - Quantum analogue of BPP
    - Problems that can be solved by a quantum Turing machine in polynomial time with bounded error
    - Includes P, BPP, but also other problems not known to be in either, including FACTOR and the discrete logarithm problem
    - Quantum-Turing Machines are theoretically possible, but we have struggled to build them in practice.

Suggested exercises (Sipser):

- 7.2, 7.5, 7.22, 7.39,
- See also: Theorems 7.31, 7.32

Problems

Optimization problems vs. decision problems

- Decision problems: answer = Yes or No
- Optimization problems: Answer = Best / least / fastest / smallest / cheapest etc..
- How to turn optimization problems into decision problems:
    - E.g., "what's the cheapest solution?" --> "is there a solution that costs less than x?"

Example problems

- FACTOR
    - Given an integer n, and a bound b, output true if n has a factor less than b.
- PRIMES
    - Given an integer n, output true if n is prime
- SAT and 3-SAT
    - Given a Boolean expression in conjunctive normal form (i.e., the AND of many clauses of OR'ed variables), decide if there's *any* truth assignment of variables such that the expression evaluates to True. 3SAT is SAT with the restriction that

clauses contain at most 3 variables.

- CLIQUE
  - Given a graph G and an integer k, decide if G contains a subgraph G' of k nodes in which every nodes is connected to every other node.

- SUBSET-SUM
  - Given a set of integers S and a target integer t, decide if any subset of S sums to t

- Hamiltonian Path/Cycle
  - Given a graph G, decide if it contains a Hamiltonian path (resp. cycle)

- Graph coloring
  - Given a graph G, what is the least number of colors required to color each node such that no adjacent nodes have the same color?

- Traveling Salesman
  - Given a list of cities and distances (e.g., roads) between them, find the shortest path that visits each city exactly once and returns to the starting point. In other words, given a weighted graph, find the Hamiltonian cycle of least weight.

- Knapsack
  - Given n items, each of which has a weight and a value, and a knapsack that can carry up to some weight w, fill the knapsack with the highest valued collection of goods that weigh less than w.

- Box packing
  - Given n boxes of various sizes and values, and a container that can hold up to volume v, fill the container with the highest valued collection of boxes with a volume less than v.

Algorithms

- Algorithms:

  - Generalized number-field sieve (GNFS) and Quadratic sieve. Solves instances of FACTOR in super-polynomial time.
  - AKS test. Solves instances of PRIMES in polynomial time.

- Probabilistic algorithms:

    - Miller-Rabin. Solves instances of FACTOR in polynomial time with bounded error.

- Quantum algorithms:

    - Shor's algorithm. Solves instances of FACTOR in quantum polynomial time with bounded error.
    - Grover's algorithm. Searches an unordered list in O(n^1/2)