COSC 341
Theory of Computing
Lecture 26
Exam contents and review

Stephen Cranefield
stephen.cranefield@otago.ac.nz

The exam in 2025

- Worth 70% of your final grade.
- Overall structure similar to previous years: one question worth roughly half the total marks on categorising languages and then a mix of other material.
- Questions that ask you to ‘explain’ or ‘describe’ something (for example) are intended to elicit medium length answers: not single sentences but not a page either. Beware of using bullet points—they often seem to encourage overly brief statements. Link up the points you make to form a connected argument.
- Definitions of the two pumping lemmas are given as reference material.

Previous exams

- Exams since 2001 are available from the library.
- Pre-2019 there's a very standard structure, which has varied a bit since then but the general contents are still quite similar.
- Do remember that 2020 was a bit of a special case!
- There has been a gradual shift of emphasis away from languages and towards machines and algorithms (but languages are still important!)
- Solution sets will not be provided (don't ask!), but if you've tried a question or questions and want feedback, or you are confused about what's being asked, do please get in touch.
- I can arrange a pre-exam question and answer session if there is interest (take poll!)

Two and a half machines

- Finite state automata
- Pushdown automata (the half - we really didn’t cover these in great detail)
- Turing machines

The fundamental question for each type is “What kind of things can (or can’t) this compute?"

Deterministic finite state automata (DFAs)

- Fundamental clockwork mechanism
- See a symbol, make a transition (i.e., update internal state)
- Red light/Green light (accepting states)
- What kinds of sets of words do they accept?

Regular languages

- Recursive closure of singleton (symbol or empty word) languages under:
  - Concatenation
  - Union
  - Kleene-star (concatenation of a language with itself 0 or more times)

Non-deterministic finite state automata (NFAs)

- May change state spontaneously (ϵ-transitions)
- May choose from a set of options on a given symbol
- Easier to glue together or modify
- Surprisingly(?), they accept the same languages as DFAs

Identifying regular languages

- (Positive) Find a DFA, NFA, regular expression, or regular grammar for it
- (Negative) Show that the **pumping lemma** is violated
- (Both) Use the **Myhill-Nerode theorem**

Theorem

A language $L \subseteq \Sigma^*$ is regular if and only if the equivalence relation $\sim_{suffix}$ on Σ^* has finitely many equivalence classes. The definition of $w \sim_{suffix} v$ is that the set of $x \in \Sigma^*$ such that $wx \in L$ ***is the same as*** *the set of $y \in \Sigma^*$ such that $vy \in L$*

Pushdown automata

- Augments a finite state machine with a stack for memory
- Transitions based on symbol read, and top symbol of stack (optional)
- Transition can cause push to or pop from stack (or both)
- Acceptance by final state, with **all input consumed** and **an empty stack**
- Languages accepted have context-free grammars (CFGs) – proof not given in this paper.
- A pumping lemma for languages described by CFGs exists

Turing machines

- Internal control by finite state
- Access to a tape containing symbols
- Transitions based on current state and symbol on tape
- Can write in current location and move tape one position left or right (and in some versions, stay)

Universal Turing machine

There is a way of representing Turing machines as binary strings (machine M, string $R(M)$)

There is a Turing machine U that:

- Processes input tapes of the form $R(M)\#w$
- Simulates the operation of M on w

In more familiar terms, U is an interpreter, $R(M)$ is source code for a program, and w is the input.

Non-deterministic Turing machines

- Can use the same model as NFA (spontaneous transitions or options)
- More useful model is to have a second tape on which a genie writes some sort of hint before the computation begins
- To accept a language means that for every word in the language there is at least one good hint, but for all words not in the language, no hints are good.

HALT

Instance: A Turing machine M (represented as $R(M)$) and an input word w
Problem: Does M halt on input w?

- Observe that if the answer is "yes" we can determine that just by running U on $R(M)\#w$
- To show that we can't solve the problem, imagine we could
  - Create an "anti-halting" machine (on input $R(M)\#w$ loops if M would halt and halts if M would loop)
  - Create the machine D that, on input $R(M)$ runs the anti-halting machine on input $R(M)\#R(M)$.
  - Ask what does D do on input $R(D)$?

Turing-reducibility

- If we can mechanically convert instances of one decision problem, A, to instances of another, B, in such a way that the answer is preserved, we say that A is reducible to B.
- If B is easy (i.e., solvable) then A must be easy too.
- So if A is undecidable (e.g., A could be HALT) then B must be too.
- *Rice's theorem* All non-trivial questions about "properties of the languages that a TM accepts" are undecidable because of a reduction from HALT.

Timing matters

- It's reasonable to say that a problem is feasible if there's a polynomial time algorithm to resolve it.
- The model of computing doesn't matter (much).
- Except, we don't know whether or not non-determinism helps.
- $\mathbf{P}$ is the class of all languages that are recognised by some **deterministic** TM with a polynomial-time bound, meaning it halts on all inputs of size n after at most $A \times n^c$ steps for some fixed constants A and c.
- For $\mathbf{NP}$ change **deterministic** to **non-deterministic**
- The million-dollar question: *Does* $P = NP$*?*

All about $\mathbf{NP}$

- A problem is *NP-complete* if it's in $\mathbf{NP}$ and there's a polynomial-time reduction of any problem in $\mathbf{NP}$ to it (intuitively, it's at least as hard as any other problem in NP).
- A problem is *NP-hard* if such a reduction exists, but it may not itself be in $\mathbf{NP}$ (meaning, we know no way of verifying correct answers efficiently)
- SATISFIABILITY is $\mathbf{NP}$-complete via the "model snapshots of a TM in action as a set of boolean constraints" argument (Cook-Levin theorem).
- From there, via reductions, *many* natural problems in graph theory and other areas are $\mathbf{NP}$-complete.

Fixed-parameter tractability and approximation

- “What can we do with hard problems?” (aside from hoping that the instances we’re trying to solve might be easy)
- A problem is *fixed-parameter tractable* if its difficulty can be confined to an auxiliary parameter k leading to time bounds of the form $O(f(k)n^c)$ where f might be very badly behaved, but c is fixed. So, for small values of k we might still be able to handle a wide variety of problem sizes.
- A c-*approximation algorithm* finds a feasible solution to an optimisation problem that is guaranteed to be within a factor of c of the true optimum.
- In other optimisation situations (not discussed in detail - but TSP is such an example) there are practical methods that produce feasible solutions which *post facto* can be shown to be very nearly optimal, but do not provide such guarantees *a priori*.

Some good news

From 1999 onwards, the Conflict-Driven Clause Learning (CDCL) family of algorithms revolutionised the solution of SAT problems. These can solve *practical instances* of SAT with millions of variables. SAT-solvers can now be used as components of solvers for complex problems such as software and hardware verification.