Regular Languages and Finite Automata

Theorem: Every regular language is accepted by some finite automaton.

Proof: We proceed by induction on the (length of/structure of) the description of the regular language. We need to show that

- $\emptyset$ is accepted by a finite automaton
  - Easy: build an automaton where no input ever reaches a final state
- $A$ is accepted by a finite automaton
  - Easy: an automaton where the initial state accepts
- each $x \in I$ is accepted by a finite automaton
  - Easy: an automaton with two states, where $x$ leads from $s_0$ to a final state.
- if $A$ and $B$ are accepted, so is $AB$
  - Proof: Suppose that $M_A = (S_A, I, f_A, s_A, F_A)$ accepts $A$ and $M_B = (S_B, I, f_B, s_B, F_B)$ accepts $B$. Suppose that $M_A$ and $M_B$ and NFAs, and $S_A$ and $S_B$ are disjoint (without loss of generality).

  Idea: We hook $M_A$ and $M_B$ together.

- if $A$ is accepted, so is $A^*$.
  - $M_A^* = (S_A \cup \{s_0\}, I, f_A, s_0, F_A \cup \{s_0\})$, where
    - $s_0$ is a new state, not in $S_A$;
    - $f_A(s) = \begin{cases} f_A(s) & \text{if } s \in S_A \setminus F_A; \\ f_A(s_A) & \text{if } s = s_0; \end{cases}$
  - $M_A^*$ accepts $A^*$.

A Non-Regular Language

Not every language is regular (which means that not every language can be accepted by a finite automaton).

Theorem: $L = \{0^n1^n : n = 0, 1, 2, \ldots \}$ is not regular.

Proof: Suppose, by way of contradiction, that $L$ is regular. Then there is a DFA $M = (S, \{0, 1\}, f, s_0, F)$ that accepts $L$. Suppose that $M$ has $N$ states. Let $s_0, \ldots, s_{2N}$ be the set of states that $M$ goes through on input $0^N1^N$

- Thus $f(s_i, 0) = s_{i+1}$ for $i = 0, \ldots, N$.

Since $M$ has $N$ states, by the pigeonhole principle (remember that?), at least two of $s_0, \ldots, s_N$ must be the same. Suppose it’s $s_i$ and $s_j$, where $i < j$, and $j - i = t$.

Claim: $M$ accepts $0^N0^t1^N$, and $0^N0^t1^N$, $0^N0^t1^N$.

Proof: Starting in $s_0$, $0^t$ brings the machine to $s_t$; another $0^t$ bring the machine back to $s_t$ (since $s_j = s_{i+t} = s_t$); another $0^t$ bring machine back to $s_t$ again. After going around the loop for a while, the can continue to $s_N$ and accept.
The Pumping Lemma

The techniques of the previous proof generalize. If $M$ is a DFA and $x$ is a string accepted by $M$ such that $|x| \geq |S|$:
- $|S|$ is the number of states; $|x|$ is the length of $x$
- then there are strings $u$, $v$, $w$ such that
  - $x = uvw$,
  - $|uv| \leq |S|$,
  - $|v| \geq 1$,
  - $uv^iw$ is accepted by $M$, for $i = 0, 1, 2, \ldots$.

The proof is the same as on the previous slide.
- $x$ was $0^n1^n$, $u = 0^i$, $v = 0^t$, $w = 0^{N-t-i1N}$.

We can use the Pumping Lemma to show that many languages are not regular:
- $\{1^n^2 : n = 0, 1, 2, \ldots\}$: homework
- $\{0^n1^n : n = 0, 1, 2, \ldots\}$: homework
- $\{1^n : n \text{ is prime}\}$
- $\ldots$

More Powerful Machines

Finite automata are very simple machines.
- They have no memory
- Roughly speaking, they can’t count beyond the number of states they have.

Pushdown automata have states and a stack which provides unlimited memory.
- They can recognize all languages generated by context-free grammars (CFGs)
  - CFGs are typically used to characterize the syntax of programming languages
- They can recognize the language $\{0^n1^n : n = 0, 1, 2, \ldots\}$, but not the language $L' = \{0^n1^n2^n : n = 0, 1, 2, \ldots\}$

Linear bounded automata can recognize $L'$.
- More generally, they can recognize context-sensitive grammars (CSGs)
- CSGs are (almost) good enough to characterize the grammar of real languages (like English)

Coverage of Final

- everything covered by the first prelim
  - emphasis on more recent material
- Chapter 4: Fundamental Counting Methods
  - Permutations and combinations
  - Combinatorial identities
  - Pascal’s triangle
  - Binomial Theorem (but not multinomial theorem)
  - Balls and urns
  - Inclusion-exclusion
  - Pigeonhole principle
- Chapter 6: Probability:
  - 6.1–6.5 (but not inverse binomial distribution)
  - basic definitions: probability space, events
  - conditional probability, independence, Bayes Thm.
  - random variables
  - uniform and binomial distribution
  - expected value and variance
Ten Powerful Ideas

- **counting**: Count without counting (*combinatorics*)
- **Induction**: Recognize it in all its guises.
- **Exemplification**: Find a sense in which you can try out a problem or solution on small examples.
- **Abstraction**: Abstract away the inessential features of a problem.
  - One possible way: represent it as a graph
- **Modularity**: Decompose a complex problem into simpler subproblems.
- **Representation**: Understand the relationships between different possible representations of the same information or idea.
  - Graphs vs. matrices vs. relations
- **Refinement**: The best solutions come from a process of repeatedly refining and inventing alternative solutions.
- **Toolbox**: Build up your vocabulary of abstract structures.

Some Bureaucracy

- The final is on Friday, May 15, 2-4:30 PM, in Olin 155
- If you have a conflict and haven’t told me, let me know now
  - Also tell me the courses and professors involved (with emails)
  - Also tell the other professors
- Office hours go on as usual during study week, but check the course web site soon.
  - There may be small changes to accommodate the TA’s exams
- There will be two review sessions: May 12 (7 PM) and May 13 (4:45)

- **Optimization**: Understand which improvements are worth it.
- **Probabilistic methods**: Flipping a coin can be surprisingly helpful!
Connections: Random Graphs

Suppose we have a random graph with \( n \) vertices. How likely is it to be connected?

- What is a random graph?
  - If it has \( n \) vertices, there are \( C(n, 2) \) possible edges, and \( 2^{C(n, 2)} \) possible graphs. What fraction of them is connected?
  - One way of thinking about this. Build a graph using a random process, that puts each edge in with probability \( 1/2 \).

- Given three vertices \( a, b, \) and \( c \), what’s the probability that there is an edge between \( a \) and \( b \) and between \( b \) and \( c \)? \( 1/4 \)
- What is the probability that there is no path of length 2 between \( a \) and \( c \)? \( (3/4)^{n-2} \)
- What is the probability that there is a path of length 2 between \( a \) and \( c \)? \( 1 - (3/4)^{n-2} \)
- What is the probability that there is a path of length 2 between \( a \) and every other vertex? \( > (1-(3/4)^{n-2})^{n-1} \)

Now use the binomial theorem to compute \( (1-(3/4)^{n-2})^{n-1} \)
\[
1 - (n - 1)(3/4)^{n-2} + C(n - 1, 2)(3/4)^{2n-2} + \cdots
\]
For sufficiently large \( n \), this will be (just about) 1.

Bottom line: If \( n \) is large, then it is almost certain that a random graph will be connected.

**Theorem:** [Fagin, 1976] If \( P \) is any property expressible in first-order logic, it is either true in almost all graphs, or false in almost all graphs.

This is called a 0-1 law.

Connection: First-order Logic

Suppose you wanted to query a database. How do you do it?

Modern database query language date back to SQL (structured query language), and are all based on first-order logic.

- The idea goes back to Ted Codd, who invented the notion of relational databases.

Suppose you’re a travel agent and want to query the airline database about whether there are flights from Ithaca to Santa Fe.

- How are cities and flights between them represented?
- How do we form this query?

You’re actually asking whether there is a path from Ithaca to Santa Fe in the graph.

- This fact cannot be expressed in first-order logic!

(A Little Bit on) NP

(No details here; just a rough sketch of the ideas. Take CS 3810/4820 if you want more.)

**NP = nondeterministic polynomial time**

- a language (set of strings) \( L \) is in NP if, for each \( x \in L \), you can guess a witness \( y \) showing that \( x \in L \) and quickly (in polynomial time) verify that it’s correct.

**Examples:**
- Does a graph have a Hamiltonian path?
  * guess a Hamiltonian path
- Is a formula satisfiable?
  * guess a satisfying assignment
- Is there a schedule that satisfies certain constraints?
  * …

Formally, \( L \) is in NP if there exists a language \( L' \) such that

1. \( x \in L \) iff there exists a \( y \) such that \( (x, y) \in L' \), and
2. checking if \( (x, y) \in L' \) can be done in polynomial time
NP-completeness

- A problem is NP-hard if every NP problem can be reduced to it.

A problem is NP-complete if it is in NP and NP-hard
- Intuitively, if it is one of the hardest problems in NP.

There are lots of problems known to be NP-complete
- If any NP complete problem is doable in polynomial time, then they all are.
  - Hamiltonian path
  - satisfiability
  - scheduling
  - . . .
- If you can prove P = NP, you’ll get a Turing award.