CSCE 551 Course Notes

Stephen A. Fenner

February 11, 2019

1 Lecture 1

Course website: `https://cse.sc.edu/~fenner/csce551/index.html`

1.1 Automata, Computability, Complexity

Complexity Why are problems easy? Hard?

Computability What problems are solvable? Solvable vs. insolvable. An initial motiviation (early 1900s): Given a mathematical statement, is it true? (Is this solvable via algorithmic computation?) Framework for classifying problems by their difficulty. Time (e.g., cryptography) and space.

Automata [*automata* is plural for *automaton*] This is a simple formal mathematical model of a computational device that can do rudimentary (but still useful) pattern matching in a string. It is good formal practice for dealing with "real" models.

1.2 Mathematical Preliminiaries

A *set* is a collection of things (the *members* or *elements* of the set). Sometimes, we can describe a set just by listing its members, separated by commas and enclosed in braces (squiggly brackets). For example, the statement,

$$S = \{7, 15, 19\}$$

says that S is a set containing exactly three members: the numbers 7, 15, and 19. The statement implies, for example that $7 \in S$ (the $\in$ relation means, "is a member of") and that $10 \notin S$ (the $\notin$ relation means, "is not a member of").

Although a set may contain objects, it itself is a single object that is necessarily distinct from its members. This means, for example, that sets can be elements of other sets. The set

$$\{\{1, 5\}, \{2\}, \{4, 9, 10, 15, 17\}\}$$

has exactly three members, namely, $\{1, 5\}$, $\{2\}$, and $\{4, 9, 10, 15, 17\}$. Each of these members is itself a set of numbers.

A set is determined entirely by its members, i.e., which objects are elements of the set. This principle is called the Axiom of Extensionality, and its precise statement is: If A and B are sets such that for all z, $z \in A \iff z \in B$, then $A = B$. That is, two sets are equal if they have exactly the same members.

The order that elements are listed in a set does not matter, neither does it matter that an element is listed more than once (duplicates). Thus $\{7, 15, 19\} = \{15, 7, 19\} = \{19, 7, 7, 15, 7\}$.

The set with no members is called the *empty set* and is denoted either by $\{\}$ or by $\emptyset$. (Note that we can say "*the* empty set" instead of just "*an* empty set" because by Extensionality there can only be one empty set.)

In some situations, when we list elements, we do want duplicates to matter (but still not order). These things are called *multisets* or *bags*. E.g., $\{7, 15, 7\} = \{7, 15\}$ as sets, but $\{7, 15, 7\} \neq \{7, 15\}$ as multisets.

Let A and B be sets. We say that $A \subseteq B$ (read, "A is a subset of B") iff (if and only if) every element of A is also an element of B. Thus $\{7, 19\} \subseteq \{7, 15, 19\}$ but $\{7, 19\} \not\subseteq \{7, 15\}$. If $A = B$ then clearly $A \subseteq B$ and $B \subseteq A$. The converse holds by Extensionality: If $A \subseteq B$ and $B \subseteq A$, then $A = B$. (So $A = B$ if and only if $A \subseteq B$ and $B \subseteq A$.) This is the most useful way to show that two sets are equal: show that each is a subset of the other.

Note that $\emptyset \subseteq A$ for any set A.

We say that $A \subset B$ (read, "A is a proper subset of B") to mean that $A \subseteq B$ but $A \neq B$, i.e., every element of A is an element of B but there is at least one element of B that is not an element of A.

Note: "iff" means "if and only if"

Natural numbers. The Sipser book defines the natural numbers to be $\mathcal{N} = \{1, 2, 3, \ldots\}$ as is common practice in mainstream mathematics, especially algebra and number theory. I (and most of my colleagues in computer science, logic, and set theory) define the natural numbers as $\mathbb{N} = \omega = \{0, 1, 2, \ldots\}$, that is, the same as $\mathcal{N}$ but including zero. Because of the potential confusion, I will try to avoid the term, "natural number" altogether and instead use the term "positive integer" to refer to an arbitrary element of $\mathcal{N}$ and "nonnegative integer" to refer to an arbitrary element of $\mathbb{N}$. (Some people use the term, "whole number" to mean "positive integer.")

The *integers* are defined as

$$\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}.$$

Sets given by rules. The list-of-members approach to describing a set is getting unwieldy, especially for large or infinite sets. An alternative way to define a set is by giving some criterion for membership in the set:

$$\{n \mid \text{rule about } n\} \text{ or } \{n : \text{rule about } n\},$$

where "rule about n" is some statement whose truth or falsehood may depend on the value of n. The set described this way contains as members exactly those n that satisfy the rule, i.e., those n for which the rule is true. You can read it literally as, "the set of all n such that [rule about n]." For example,

$$\{n \mid n \text{ is an even integer}\} = \{\ldots, -4, -2, 0, 2, 4, \ldots\}$$

defines the set of n such that n is an even integer, i.e., the set of all even integers.

Sometimes for shorthand, we include part of the rule to the left of the vertical bar (|). The expression

$$\{n \in \mathbb{Z} \mid n \text{ is even}\}$$

also describes the set of all even integers. You can read it literally as, "the set of all integers n such that n is even."

Another variation is to give some expression to the left of the bar that denotes some object, and to the right of the bar put some rule regarding the component(s) of the expression. Thus,

$$\{2n \mid n \text{ is an integer}\}$$

(literally, "the set of all things of the form $2n$ where n is an integer") also denotes the set of all even integers.

Every set can be given in a rule-based form. For example,

$$\{7, 15, 19\} = \{n \mid n = 7 \vee n = 15 \vee n = 19\},$$

where "$\vee$" means inclusive OR.

1.2.1 Operations on Sets

Let A and B be sets. $A \cup B$ (read, "A union B" or, "the union of A and B) is the set of all things that are either in A or in B (or both). $A \cap B$ (read, "A intersect B" or, "the intersection of A and B) is the set of all things common to A and B, i.e., things that are both elements of A and elements of B.

$$\begin{aligned} \{7, 15, 19\} \cup \{7, 19, 21\} &= \{7, 15, 19, 21\}, \\ \{7, 15, 19\} \cap \{7, 19, 21\} &= \{7, 19\}. \end{aligned}$$

[Venn diagram of $A \cup B$ and $A \cap B$]

$A - B$ (read, "the complement of B in A" or "the relative complement of B with respect to A, also denoted $A \backslash B$) is the set of all things that are in A but not in B. [Venn diagram]

For any set A, the *powerset* of A, denoted $\mathcal{P}(A)$, is the set of all subsets of A. That is, an object z is a member of $\mathcal{P}(A)$ just in case that $z \subseteq A$. For example,

$$\mathcal{P}(\{1, 2, 3\}) = \{S \mid S \subseteq \{1, 2, 3\}\} = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}.$$

Sequences. A *sequence* is a list of things. In a sequence, both order and duplicates matter. We can give a sequence by listing its elements surrounded by parentheses:

$$(7, 19, 15, 20, \ldots)$$

Two sequences are equal only if they have the same length and each element of one sequence is equal to the element *in the same position* in the other sequence.

A finite sequence is called a *tuple*. A sequence with k elements is a *k-tuple*. A 3-tuple is also called a *triple*, and a 2-tuple is also called a *pair* (or *ordered pair*). Note: $(x, y) = (y, x)$ only if $x = y$.

Cartesian product. For sets A and B, we define

$$A \times B = \{(x, y) \mid x \in A \wedge y \in B\},$$

where "$\wedge$" means AND. Alternatively,

$$A \times B = \left\{ p : \begin{array}{l} p \text{ is a pair whose first element is in } A \text{ and} \\ \text{whose second element is in } B \end{array} \right\}.$$

The *cardinality* of a set A, denoted $|A|$, is the number of elements in A (the size of A). Fact: $|A \times B| = |A| \cdot |B|$ for sets A and B.

For sets $A_1, A_2, \ldots, A_k$, we can define the Cartesian product $A_1 \times A_2 \times \cdots \times A_k$ to be the set of all k-tuples $(a_1, a_2, \ldots, a_k)$ where $a_i \in A_i$ for all $1 \le i \le k$. Also,

$$A^k = \underbrace{A \times A \times \cdots \times A}_{k \text{ times}}.$$

More general unions and intersections. If X is a set (of sets), then

$$\cup X = \{z \mid (\exists A \in X)[z \in A]\}$$

is the union of all elements of X. If $X \neq \emptyset$, then we can also define

$$\cap X = \{z \mid (\forall A \in X)[z \in A]\},$$

which is the intersection of all elements of X. For example, if

$$X = \{\{6\}, \{4, 6, 7\}, \{2, 3, 6\}\},$$

then

$$\begin{aligned} \cup X &= \{2, 3, 4, 6, 7\}, \\ \cap X &= \{6\}. \end{aligned}$$

1.3 Scopes of names

One aspect of mathematical discourse that I notice confuses many students is the concept of a *free* variable versus a *bounded* variable. A name occuring in an expression or equation may be of either kind, and to follow a mathematical argument it is crucial to discern which kind it is and what it means.

Consider the set former we used above to express the set of all even integers:

$$\{n \in \mathbb{Z} \mid n \text{ is even}\}.$$

In this expression, n occurs as a bound variable, or a *dummy variable.* That means that its scope is limited to the expression. Outside the expression n has no meaning (unless it is given one elsewhere). It is akin to a local variable inside a function definition in a computer program. You cannot refer to a local variable from outside the function. Just as changing the name of a local variable consistently throughout a function definition does not change the behavior of the function, changing n to another variable name consistently thoughout the expression does not change the meaning of the expression at all. For example,

$$\{x \in \mathbb{Z} \mid x \text{ is even}\}$$

denotes exactly the same set—of even integers. The only caveat is that you should not change a bound variable's name to that of an existing variable in the same expression.

The first occurrence of n in the expression above is called the *binding* occurrence, akin to the declaration of a local variable in a function; the other occurrence(s) of n are its *uses.* There are several ways to bind a variable, making it a dummy/local variable. In the expression,

$$\sum_{i=1}^{n} i^2$$

The variable i is bound. The binding occurrence is in the subscript below the $\sum$ symbol, and here there is one of i—where it is being squared. The name i means nothing outside this expression. The expression has exactly the same meaning as $\sum_{j=1}^{n} j^2$.

The occurrence n in the expression $\sum_{i=1}^{n} i^2$ is not a bound occurrence; it is a *free occurrence.* That means that n must be introduced ("declared") outside the expression (in the previous text), and the value of the expression now depends on the value of n. You cannot change n to another name in the expression without changing the meaning of the expression. This situation is analogous to a function definition using an externally declared (e.g., global) variable.

Another way to bind a variable is with a quantifier to form an assertion. Let O be the set of all positive odd integers. The following sentence asserts (falsely) that all positive odd integers are prime:

$$(\forall x \in O)[\, x \text{ is prime} \,] .$$

In this sentence, x is a bound variable (with no meaning outside the sentence), and its binding occurrence is right after the "$\forall$" quantifier. This sentence has a definite truth value—FALSE—independent of the value of x. You can change all x's to y's to get an equivalent false sentence. In the formula, "x is prime" by itself, x occurs freely (unbound). The truth value of this formula depends on the (externally supplied) value of x. The formula is false if $x = 6$, but true if $x = 7$. Prepending a quantifier for x binds it, making it a dummy variable.

By the way, the type of quantifier used may influence the truth value of the sentence. For example, the sentence

$$(\exists x \in O)[\, x \text{ is prime} \,]$$

asserts that there exists an odd number which is prime. This is obviously TRUE.

2 Lecture 2

2.1 More Mathematical Preliminaries

I'm assuming you've picked up on the following:

- Graphs (undirected graphs)
  - vertices (nodes), edges (arcs)

  - incidence, adjacency, neighbors
  - degree, labeled graph
  - path, simple path, connected graph, cycle, simple cycle
  - acyclic graph
  - tree, leaf, root

- Digraphs (directed graphs)
  - indegree, outdegree, directed path
  - strongly connected (directed paths in both directions)

- Strings and languages
  - alphabet = any finite set, members are symbols
  - string (over an alphabet)
  - uv is concatenation of strings u and v
  - length of w is denoted $|w|$, empty string is ε (identity under concatenation), reverse of w is $w^{\mathcal{R}}$
  - substring,
  - lexicographic ordering is length first then dictionary order between strings of the same length
  - language (set of strings over a given alphabet)

- Boolean logic
  - Boolean values true and false (1 = true and 0 = false)
  - Boolean operations: conjunction (and, $\wedge$), disjunction (or, $\vee$), negation (not, $\neg$), exclusive or (xor, $\oplus$)
  - book: equality $\leftrightarrow$; me and others: equivalence (biconditional)
  - book: implication $\rightarrow$; me and others: conditional
  - operands
  - Both $\{\wedge, \neg\}$ and $\{\vee, \neg\}$ are complete set of Boolean connectives
  - distributive laws

2.2 Definitions, theorems, and proofs (and lemmas, corollaries)

A *formal proof* (of statement S) is a sequence of mathematical statements written in a rigorously defined syntax, such that each statement is either an axiom or follows from some previous statements in the sequence by a rule of inference, and whose last statement is S. S is then considered a theorem.

- The syntax specification is a *formal language.*

- The specification of axioms and rules of inference is a *formal system.*

The idea is that, at least in principle, a proof can be checked for correctness by a computer using a purely algorithmic procedure, based solely on the proof's syntactical form without regard to its meaning. Formal proofs, even of simple theorems, are almost always long and difficult for a human to read and digest; therefore, they almost never appear in the mathematical literature. Instead, what appear are informally written (but still rigorous) proofs.

Such an informal proof is written basically as prose, but may include mathematical formulas. It is a piece of rhetoric meant to convince the mathematically intelligent reader beyond any doubt that the proposed assertion is true (a theorem), that is, that a formal proof exists. The informal proof can appeal to previous theorems. It can also appeal to the reader's mathematical understanding and intuition. In this case, the prover must be prepared to explain or "fill in" more formally these appeals if challenged to do so. Writing a proof is more of an art than a mechanical exercise. It contains elements of style, as with any good writing. The best way to learn to write good proofs is to read good proofs that others have written.

2.2.1 Proof methods: construction, contradiction, induction

Theorem 2.1. *There are (real) irrational numbers $a, b > 0$ such that a^b is rational.*

Proof. We know that $\sqrt{2}$ is irrational. If $\sqrt{2}^{\sqrt{2}}$ is rational we are done (set $a = b = \sqrt{2}$). Otherwise, $\sqrt{2}^{\sqrt{2}}$ is irrational. Set $a = \sqrt{2}^{\sqrt{2}}$ and $b = \sqrt{2}$. Then

$$a^b = \left(\sqrt{2}^{\sqrt{2}}\right)^{\sqrt{2}} = \sqrt{2}^{\sqrt{2}\sqrt{2}} = \sqrt{2}^2 = 2,$$

which is certainly rational, so the theorem is proved. □

Proof that $\pi > 3$.
Proof that area of circle is πr^2.

Lemma 2.2. *Every acyclic graph with $n > 0$ vertices contains a vertex of degree 0 or 1.*

Proof. (By contradiction and the pigeonhole principle.) Suppose there is a nonempty acyclic graph every vertex of which has degree at least 2. Let G be such a graph. Choose any vertex v_1 of G. v_1 has at least two neighbors, so choose a neighbor v_2 of v_1 arbitrarily. Now v_2 has at least one neighbor besides v_1, so choose such a neighbor v_3 arbitrarily. Continue in this way to pick $v_4, v_5, \ldots$ such that v_i is adjacent to v_{i+1} and $v_i \neq v_{i+2}$ for every $i \geq 1$. Since G has only finitely many vertices, by the pigeonhole principle there must be some $i < j$ such that $v_i = v_j$ and $v_i, v_{i+1}, \ldots, v_{j-1}$ are all distinct. Then $(v_i, v_{i+1}, \ldots, v_j)$ forms a simple cycle in G, which contradicts the fact that G is acyclic. Thus G must have at least one vertex of degree at most 1. □

Intuition is crucial in forming a proof! Pictures and diagrams are very helpful, however, they cannot replace a formal argument.

Theorem 2.3. *No graph with $n \geq 2$ vertices and fewer than $n - 1$ edges can be connected.*

Proof. (Uses the "minimum counterexample" idea, which combines induction with contradiction. Also uses cases.) Suppose there is a connected graph with $n \geq 2$ vertices and $m < n - 1$ edges. Let G be such a graph with the least number of edges of all such graphs. There are two cases:

Case 1: G contains a cycle. We select an edge e of G that lies on some cycle c and remove it. The resulting graph G' is still connected (any path joining two vertices of G that used to go through e can be rerouted around the rest of the cycle c, giving a path in G'). But G' has fewer edges than G, which contradicts the minimality of G.

Case 2: G is acyclic. Let n be the number of vertices of G and let m be the number of edges of G. By assumption, $m < n-1$. The graph with two vertices and no edges is clearly disconnected, so G must have at least three vertices, i.e., $n \geq 3$. By Lemma 2.2, G has some vertex v with degree 0 or 1. If $\deg(v) = 0$, then v is isolated, making G disconnected, and so we must have $\deg(v) = 1$. Remove v and its only incident edge from G. The resulting graph G' is clearly still connected, and furthermore, G' has $n-1 \geq 2$ vertices and $m-1 < m \leq (n-1)-1$ edges. Since G' has fewer edges than G, this contradicts the minimality of G.

The cases are exhaustive, and in either case we arrive at a contradiction. Thus no such graph can exist. □

Theorem 2.4. *Any graph with $n \geq 3$ vertices and $m \geq n$ edges must contain a cycle.*

Proof. (Same "minimal counterexample" technique.) Suppose there is some acyclic graph with $n \geq 3$ vertices and at least n edges. Let G be such a graph whose n value (the number of vertices) is least among all such graphs. The graph only graph with three vertices and at least three edges is the triangle, which is clearly cyclic, so we must have $n \geq 4$. By Lemma 2.2, G has a vertex v with $\deg(v) \leq 1$. Remove v and its incident edge (if there is one) from G to obtain a new graph G'. G' is clearly acyclic because G is acyclic. Furthermore, G' has $n-1 \geq 3$ vertices and at least $n-1$ edges, which contradicts the minimality of G. Thus no such graph can exist. □

Definition 2.5. A *tree* is a connected, acyclic graph.

Corollary 2.6. *Any tree with $n > 0$ vertices has exactly $n-1$ edges.*

3 Lecture 3

3.1 Hall's theorem

First a few definitions. A graph $G = (V, E)$ is *bipartite* iff the vertices can be partitioned into two nonempty sets V_1 and V_2 (that is, $V_1 \cup V_2 = V$ and $V_1 \cap V_2 = \emptyset$ and $V_1 \neq \emptyset$ and $V_2 \neq \emptyset$), so that every edge in E has one endpoint in V_1 and the other in V_2. The sets V_1 are V_2 together are called a *bipartition* of G. Lots of real-world situations can be modeled by bipartite graphs. For example, V_1 is the set of students at a university, V_2 is the set of courses taught at that university, and we include an edge connecting student s with course c iff s has taken course c. For another example, V_1 is the set of kids at a daycare center, V_2 is the set of candies available at a local candy store, and we have an edge from kid k to candy c iff k likes c.

Hall's theorem relates to the following question, related to the second example above: Can you buy a separate piece of candy for each kid so that each kid gets a candy that (s)he likes? There's no possibility for sharing, so you must by a different candy for each kid. Suppose there is some group of n kids such that the set of all candies liked by at least one kid in the group has size less than n. Then the task is impossible; you can't buy a separate piece of candy liked by each of the n kids in the group, because there are fewer than n acceptable candy choices for the group (pigeonhole principle!).

Now suppose that there is no such group of kids. That is, for any group of kids in the daycare center, there is a set of candies—each liked by at least one member of the group—with at least as many candies as there are kids in the group. Hall's theorem says that in this case, it *is* possible to buy a separate piece of candy for each kid in the daycare center that (s)he likes.

We state and prove the theorem in terms of bipartite graphs. Let G be bipartite with bipartition V_1, V_2. Given any set $S \subseteq V_1$, we define the *neighborhood* $\Gamma(S)$ of S in G as the vertices in V_2 that are adjacent to vertices in S. That is,

$$\Gamma_G(S) := \{y \in V_2 : y \text{ is adjacent to some vertex in } S\} .$$

We will say that G has the *neighborhood size property* iff $|\Gamma_G(S)| \geq |S|$ for all nonempty $S \subseteq V_1$. A *full matching* in G (with respect to V_1) is any function $g : V_1 \to V_2$ such that x is adjacent to $g(x)$ for all $x \in V_1$.

Theorem 3.1 (Hall's Theorem)**.** *Let G be a bipartite graph. If G has the neighborhood size property, then G has a full matching.*

The theorem and its proof are not at all obvious, but the proof is a good illustration of induction/minimality combined with proof-by-cases.

Proof. Suppose (for the sake of contradiction) that there is some graph G that violates the theorem. That is, G is bipartite with the neighborhood size property, but G has no full matching. By the minimality principle, we can assume that G has the fewest vertices in V_1 of any graph violating the theorem. (In other words, any bipartite graph with a strictly smaller V_1 and with the neighborhood size property has a full matching.) We now show that G actually does have a full matching, contradicting our assumption.

Case 1: For every $S \subseteq V_1$ such that $0 < |S| < |V_1|$, we have $|\Gamma_G(S)| > |S|$. Since G has the neighborhood property, it must have at least one edge (Why?[1]). Let $\{x, y\}$ be some edge of G with $x \in V_1$ and $y \in V_2$. Now remove x and y (along with all incident edges) from the graph G to obtain a graph H with bipartition $V_1 - \{x\}$ and $V_2 - \{y\}$. Now H has the neighborhood size property: for any nonempty $S \subseteq V_1 - \{x\}$, we have $\Gamma_H(S) = \Gamma_G(S) - \{y\}$, and so

$$|\Gamma_H(S)| = |\Gamma_G(S) - \{y\}| \geq |\Gamma_G(S)| - 1 \geq |S| ,$$

that last inequality following from the Case 1 assumption. Therefore, by the minimality of G, the graph H must have a full matching $h : (V_1 - \{x\}) \to (V_2 - \{y\})$. But now we can just extend h to a full matching g in G by matching x with y:

$$g(u) := \begin{cases} h(u) & \text{if } u \neq x, \\ y & \text{if } u = x \end{cases}$$

for all $u \in V_1$.

[1]Because $|\Gamma_G(V_1)| \geq |V_1| > 0$.

Case 2: There exists $S \subseteq V_1$ such that $0 < |S| < |V_1|$ and $|\Gamma_G(S)| = |S|$ (that is, not Case 1).[2] We chop G up into two smaller graphs H and J. H has bipartition S and $\Gamma_G(S)$ and all edges of G running between them. J has bipartition $V_1 - S$ and $V_2 - \Gamma_G(S)$ and all edges of G running between them. Now H has the neighborhood size property: for any $T \subseteq S$, we have $\Gamma_H(T) = \Gamma_G(T) \subseteq \Gamma_G(S)$, and so

$$|\Gamma_H(T)| = |\Gamma_G(T)| \geq |T|$$

by the neighborhood size property of G.

Now we show that J also has the neighborhood size property. Let T be any subset of $V_1 - S$. Then $\Gamma_G(T \cup S) = \Gamma_J(T) \cup \Gamma_G(S)$, which implies $|\Gamma_G(T \cup S)| \leq |\Gamma_J(T)| + |\Gamma_G(S)| = |\Gamma_J(T)| + |S|$ (Case 2 assumption). By the neighborhood size property of G, we then have

$$|\Gamma_J(T)| \geq |\Gamma_G(T \cup S)| - |S| \geq |T \cup S| - |S| = |T| + |S| - |S| = |T| \ .$$

By the minimality of G, we therefore know that H and J have full matchings $h : S \to \Gamma_G(S)$ and $j : (V_1 - S) \to (V_2 - \Gamma_G(S))$, respectively. There are no collisions between h and j, so we can simply combine them to get a full matching g of G:

$$g(u) := \begin{cases} h(u) & \text{if } u \in S, \\ j(u) & \text{if } u \in V_1 - S \end{cases}$$

for all $u \in V_1$.

So in either case, G satisfies the theorem. □

4 Lecture 4

4.1 Overview of computation

The theory was mapped out in 20s, 30s, before electronic computers (Church, Hilbert, Kleene, Goedel), to clarify the concept of mathematical proof. Turing's and von Neumann's ideas led to first electronic computer.

4.2 Regular languages

The finite state machine (automaton—simplest computational model) models limited memory computers.

4.2.1 Examples

Door opener.

states: closed, open

input conditions: front, rear, both, neither

[2]Note that we cannot have $|\Gamma_G(S)| < |S|$, because G has the neighborhood size property.

nonloop transitions:

closed $\to$ open on front

open $\to$ closed on neither

(probabilistic counterpart: Markov chain)

Other examples: number of 1s is even, number of 0s is multiple of 4, automaton for strings ending in 00.

4.2.2 Formal Definitions

Definition 4.1. Let Σ be a finite set. A *string over* Σ is a finite sequence of elements of Σ.

We may call Σ an *alphabet* and it elements *symbols*. We write a string by writing its constituent symbols one after another. If w is a string, we let $|w|$ denote the *length* of w (number of symbols of w). There is a unique string of length 0, which we call the *empty string*, denoted by ε. We can identify the symbols of Σ with strings of length 1. If x and y are two strings over Σ, we let xy denote the *concatenation* of x and y, which is the string starting with x and immediately followed by y.

For example, the concatenation of 00110 and 111011 is 00110111011.

The empty string contatenated with any string is just the latter string: $w\varepsilon = \varepsilon w = w$ for any string w. So ε is the *identity under concatenation.* Concatation is a binary operator (like multiplication) that takes two strings as arguments and produces a string as a result. Concatenation is associative, i.e., $(xy)z = x(yz)$ for any strings x, y, and z, and we denote this string by xyz. We can also drop parentheses in the usual way when concatenating more than three strings, etc.

Definition 4.2. A *deterministic finite automaton (DFA)* is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

- Q is a finite set (members of Q are called *states*),
- Σ is a finite set called the *alphabet* (members of Σ are called *symbols*),
- $\delta : Q \times \Sigma \to Q$ is the *transition function*,
- $q_0 \in Q$ is the *start state*, and
- $F \subseteq Q$ is the set of *accepting states* (or *final states*).

Here's an example of a DFA M_1 with state set $Q = \{q_1, q_2, q_3\}$, alphabet $\Sigma = \{0, 1\}$, start state q_1, final state set $F = \{q_2\}$, and transition function δ given by the following table:

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_1

Draw corresponding transition diagram (a directed graph).

Describe computation informally using a tape, cells with symbols, and a finite-state control with a read head advancing over the input. The DFA accepts the input if its state after reading all the symbols is in F. Otherwise it rejects. (Consider the empty string as input, too.)

Definition 4.3. Let Σ be an alphabet. We let Σ^* denote the set of all strings over Σ. A *language over* Σ is any subset of Σ^*, i.e., any set of strings over Σ.

We use languages to encode decision problems. Given an input (string) is the answer "yes" or "no"? Strings for which the answer is "yes" are called *yes-instances* and the others are *no-instances*. The language corresponding to a decision problem is just the set of strings encoding yes-instances.

Definition 4.4. Let M be a DFA with alphabet Σ and let $A \subseteq \Sigma^*$ be a language over Σ. We say that M *recognizes* A iff M accepts every string in A and rejects every string in $\Sigma^* - A$. We let $L(M)$ denote the language recognized by M.

Thus recognizing a language means being able to distinguish membership from nonmembership in the language, thus solving the corresponding decision problem. Every DFA recognizes a unique language.

Example: DFA recognizing $B = \{x \in \{0,1\}^* \mid |x| \geq 2$ and the 1st symbol of x equals the last symbol of $x\}$.

Draw a seven-state DFA for B. Whoops, there is a five-state DFA as well. Is five states the fewest possible? We'll see how to answer this question later.

5 1/30/2008

5.1 Formal Definition of a DFA Computation

We now define a DFA computation formally.

Definition 5.1. Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA and let $w \in \Sigma^*$ be a string over Σ. Suppose $w = w_1 w_2 \cdots w_n$ where $w_i \in \Sigma$ for all $1 \leq i \leq n$. The *computation of M on input w* is the unique sequence of states $(s_0, s_1, \ldots, s_n)$ where

- each $s_i \in Q$,
- $s_0 = q_0$, the start state, and
- $s_i = \delta(s_{i-1}, w_i)$ for all $1 \leq i \leq n$.

The computation $(s_0, \ldots, s_n)$ is *accepting* if $s_n \in F$, and is *rejecting* otherwise. If the former holds, we say that M *accepts* w, and if the latter holds, we say that M *rejects* w.

Thus $s_0, s_1, \ldots$ is the sequence of states that M goes through while reading w from left to right, starting with the start state.

Example: DFA that recognizes multiples of 3 in unary, binary.

Example: DFA that accepts strings over $\{a, b, c\}$ containing *abacab* as a substring. (Idea is useful for text search.)

Example: Strings over $\{0,1\}$ with an even number of 1s. Strings over $\{0,1\}$ with an even number of 1s or an odd number of 0s.

Definition 5.2. A language $A \subseteq \Sigma^*$ is *regular* iff some DFA recognizes it, i.e., $A = L(M)$ for some DFA M.

5.2 Nondeterminism

In a DFA, the next state is completely determined by the current state and the symbol being scanned, according to the transition function δ. In a *nondeterministic finite automaton*, there may be a choice of several states to transition into at each step, and we may decide not to advance the read head. The machine accepts the input iff there is *some* series of choices that can lead to an accepting state having read the entire input.

Transition diagram may have any number of edges with the same label leaving the same state. There can also be ε-edges (meaning: make the transition without reading the symbol or advancing the read head).

Example: The substring recognizer above. Add an ε-transition and explain it.

6 2/4/2008

Given an alphabet Σ, we define

$$\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\},$$

which is the set of all strings over Σ of length 0 or 1.

Recall from set theory: For any set A, the set $\mathcal{P}(A)$, called the *power set of A* is the set of all subsets of A. For example,

$$\mathcal{P}(\{1,2,3\}) = \{\emptyset, \{1\}, \{2\}, \{1,2\}, \{3\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}.$$

Definition 6.1. A *nondeterministic finite automaton (NFA)* is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

- Q is a finite set (the set of states),
- Σ is a finite set (the alphabet),
- $\delta : Q \times \Sigma_\varepsilon \to \mathcal{P}(Q)$ is the transition function,
- $q_0 \in Q$ is the start state, and
- $F \subseteq Q$ is the set of final states (accepting states).

Notice that $\delta(q, a)$ is now a set of states instead of just a single state. It is the set of possible successor states of q reading a. Note also that $\delta(q, \varepsilon)$ is also defined to be a set of states. This is the set of possible successors to q via an ε-transition. Such a transition, when taken, does not advance the read head.

Given an NFA M and an input string w, there may now be many possible computations (or computation paths) of M on w. We say that M accepts w iff *at least one of these computation paths ends in a final state after reading all of w*. Informally, M accepts if there is a correct set of transition "guesses" that leads to an accepting state and reads the entire input.

Example with detecting a substring in a larger string. Easier with an NFA. Explain getting stuck (that computation path rejects, even if it gets stuck in a final state).

Definition 6.2. Let $M = (Q, \Sigma, \delta, q_0, F)$ be an NFA and let $w \in \Sigma^*$ be a string. A *complete computation* of M on input w is a pair of tuples $((s_0, \ldots, s_m), (w_1, \ldots, w_m))$ where

- all the $s_i \in Q$ for $0 \le i \le m$,
- all the $w_i \in \Sigma_\varepsilon$ for $1 \le i \le m$ (that is, each w_i is either a symbol in Σ or the emptystring ε),
- $w = w_1 \cdots w_m$,
- $s_0 = q_0$, and
- $s_i \in \delta(s_{i-1}, w_i)$ for all $1 \le i \le m$.

The complete computation above is *accepting* iff $s_m \in F$; otherwise, it is *rejecting*. We say that M *accepts* w if there exists some accepting complete computation of M on w.

A complete computation represents a legal path through the NFA while reading the entire input. ε-transitions correspond to steps where $w_i = \varepsilon$.

As with DFAs before, we define the *language* $L(M)$ *recognized by NFA* M to be the set of all input strings accepted by M.

We say that two automata (NFAs or DFAs) are *equivalent* iff they recognize the same language.

6.1 Equivalence of DFAs and NFAs

Here we show that DFAs and NFAs are equally powerful models of computation, that is, they recognize the same class of languages, i.e., the regular languages. We show this in two steps:

1. For every DFA there is an equivalent NFA.
2. For every NFA there is an equivalent DFA.

One direction is obvious: for any DFA, make an equivalent NFA with the same transition diagram. Each $\delta(q, a)$ is a singleton for $a \in \Sigma$, and $\delta(q, \epsilon) = \emptyset$. More formally, if $M = (Q, \Sigma, \delta, q_0, F)$ is a DFA, define the NFA $N = (Q, \Sigma, \delta', q_0, F)$, where

- $\delta'(q, a) = \{\delta(q, a)\}$ for all $q \in Q$ and $a \in \Sigma$, and
- $\delta'(q, \varepsilon) = \emptyset$ for all $q \in Q$.

It's easy to see that $L(N) = L(M)$, i.e., that N is equivalent to M.

Other direction is nontrivial and (somewhat) surprising.

Theorem 6.3. *For every NFA there is an equivalent DFA.*

We give a sketch of a proof, which includes the construction of the equivalent DFA but does not verify that it is correct. The idea is that, as we simulate a given NFA, we keep track of the set of all states we could be in after reading each successive symbol of the input. Fix an NFA M. We'll build an equivalent DFA N. If S is the set of all states of M we could possibly get to by reading the input up to but not including a symbol a, then the set S' of states of M that we could possibly be in *after* reading a only depends on S and a, and can be computed from the transition diagram of M. We make each possible set S of states in M a single state of the new DFA N, and the transitions of N correspond to shifting the set of possible states of M upon reading each symbol. For example, the transition $S \xrightarrow{a} S'$ would be a transition in N. There are only finitely many subsets of M, so N is really a deterministic *finite* automaton.

M may have ε-transitions, so to find S' from S, we first follow any a-transitions from states in S, then follow any ε-transitions thereafter (possibly several in a row). You may think that we should also follow any ε-transitions *before* the a-transitions, but this won't be necessary, as we'll see below.

The start state of N corresponds to the set of states of M that one could possibly be in before reading any input symbols. This is exactly the set of states reachable from the start state of M via ε-transitions only. A set S of states of M constitutes an accepting state of N iff it includes at least one accepting state of M, i.e., the computation could possibly be in a final state of M.

Here is the formal construction. Fix an NFA $M = (Q, \Sigma, \delta, q_0, F)$. For any set $S \subseteq Q$ of states of M, we define the *ε-closure* of S, denoted $E(S)$ to be the set of states reachable from S via zero or more ε-transitions in a row. More formally, for $i \geq 0$ we define $E_i(S)$ inductively to be the set of states reachable from S via exactly i many ε-transitions:

$$E_0(S) = S,$$

and for all $i \geq 0$,

$$E_{i+1}(S) = \{r \in Q \mid (\exists q \in E_i(S))[r \in \delta(q, \varepsilon)]\} = \bigcup_{q \in E_i(S)} \delta(q, \varepsilon).$$

This lets us define

$$E(S) = \bigcup_{i \geq 0} E_i(S),$$

namely, the set of all states reachable from S via zero or more ε-transitions.

Note that $E(S)$ is *closed under ε-transitions*, i.e., there are no ε-transitions from inside $E(S)$ to outside $E(S)$.

We are now ready to define precisely the DFA N equivalent to M: we let $N = (\mathcal{P}(Q), \Sigma, \Delta, S_0, \mathcal{F})$, where

- $\mathcal{P}(Q)$ is the powerset of Q,
- $S_0 = E(\{q_0\})$, the set of states reachable from q_0 via ε-transitions,
- $\mathcal{F} = \{S \subseteq Q \mid S \cap F \neq \emptyset\}$, the set of sets of states of M which contain final states of M, and
- for every $S \subseteq Q$ and $a \in \Sigma$,

$$\Delta(S, a) = E\left(\bigcup_{q \in S} \delta(q, a)\right).$$

From the above definition, every state of N that is reachable from the start state S_0 is closed under ε-transitions, so there is no need to follow ε-transitions before a-transitions in the definition of Δ.

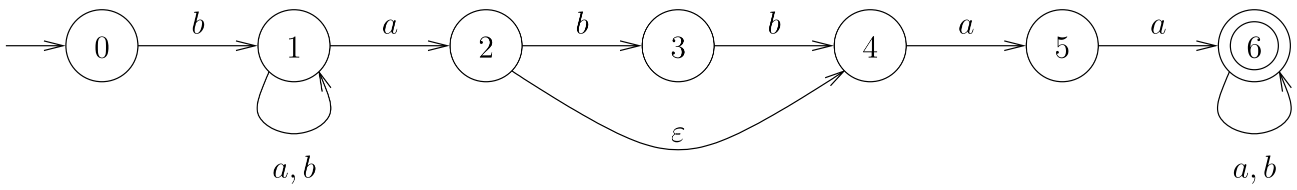
7 2/6/2008

Corollary 7.1. *A language is regular iff it is recognized by some NFA.*

Quick example: three states with an epsilon move.

When an NFA is converted to an equivalent DFA, often not all the possible sets of states of the NFA are used, so not all the DFA states are reachable from the start state, and they can be eliminated (e.g., any state sets which are not closed under ε-transitions). As a practical matter, instead of including all possible subsets of states of the NFA, it's better to "grow" the DFA states from the start state by following transitions. You will usually wind up with a lot fewer DFA states this way.

Example. Here's an NFA that recognizes the language of strings over $\{a, b\}$ that start with b and contain either aaa or $abbaa$ as a substring:



To construct the equivalent DFA, we build its states one by one, starting with its start state, then following transitions. Whenever we encounter a new set of states of the NFA, we add it as a new state to the DFA. The general algorithm is as follows:

```
Input: NFA $M = (Q, \Sigma, \delta, q_0, F)$
Output: equivalent DFA $N = (\mathcal{Q}, \Sigma, \Delta, S_0, \mathcal{F})$
    Set $\Delta := \emptyset$ (no transitions yet)
    Set $\mathcal{F} := \emptyset$ (no final states yet)
    Set $S_0 := E(\{q_0\})$, the states reachable from $q_0$ via paths of $\varepsilon$-transitions
    Set $\mathcal{Q} := \{S_0\}$
    While there is an unmarked $S \in \mathcal{Q}$, do
        For each $a \in \Sigma$, do
            Let $T$ be the set of states reachable from states in $S$ via
                a single $a$-transition followed by any number of $\varepsilon$-transitions
            Add the transition $\Delta(S, a) = T$ to $\Delta$
            If $T \notin \mathcal{Q}$ then
                $\mathcal{Q} := \mathcal{Q} \cup \{T\}$
                If $T \cap F \neq \emptyset$, then $\mathcal{F} := \mathcal{F} \cup \{T\}$
        End-for
        Mark $S$ as "finished"
    End-while
    Return $N = (\mathcal{Q}, \Sigma, \Delta, S_0, \mathcal{F})$
End-algorithm
```

Let's apply this algorithm to the transition diagram above.

The start state of the DFA is $S_0 = E(\{0\}) = \{0\}$.

Compute the transitions from S_0. First the a-transition: there is nothing reachable from state 0 following a, so the new state of the DFA is $\emptyset$ and we have $\Delta(S_0, a) = \emptyset$, which is a new state, so we call it S_1 and add it to $\mathcal{Q}$. We now have $\mathcal{Q} = \{S_0, S_1\}$, where $S_1 = \emptyset$, and the diagram looks like this:

$$\begin{tikzcd}
{} \arrow[r] & S_0 \arrow[r, "a"] & S_1
\end{tikzcd}$$

Starting back at state 0 and following b, the only state we can reach is 1, so we have a new state $S_2 = \{1\}$ and set $\Delta(S_0, b) = S_1$. How about transitions from S_1? S_1 has no states of the NFA, so we cannot get anywhere from S_1 following any symbol. Thus

$$\Delta(S_1, a) = \Delta(S_1, b) = \emptyset = S_1,$$

and we should add two self-loops from S_1 to itself. The diagram becomes:

$$\begin{tikzcd}
{} \arrow[r] & S_0 \arrow[r, "a"] \arrow[d, "b"] & S_1 \arrow[loop right, "{a,b}"] \\
& S_2 &
\end{tikzcd}$$

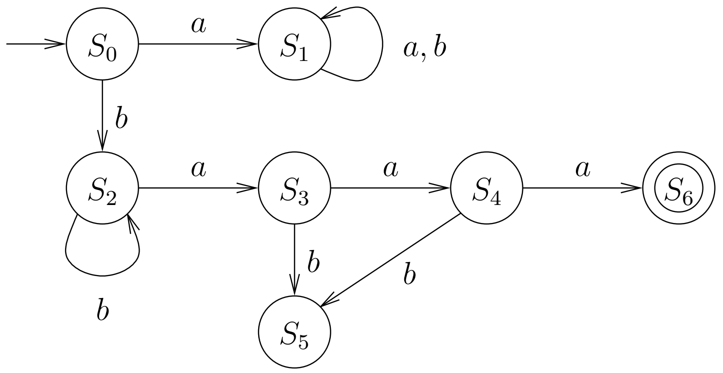
S_0 and S_1 are both finished at this point. Now S_2. Following a from state 1 of the NFA, we can get to states 1 and 2, then following the ε-transition from 2 we can get to 4. Thus we have a new state $S_3 = \{1, 2, 4\} = \Delta(S_2, a)$. Following b from state 1, we can only get back to state 1, so $\Delta(S_2, b) = \{1\} = S_2$, and we put a self-loop $S_2 \xrightarrow{b} S_2$. Here's the new diagram:

$$\begin{tikzcd}
{} \arrow[r] & S_0 \arrow[r, "a"] \arrow[d, "b"] & S_1 \arrow[loop right, "{a,b}"] \\
& S_2 \arrow[r, "a"] \arrow[loop below, "b"] & S_3
\end{tikzcd}$$

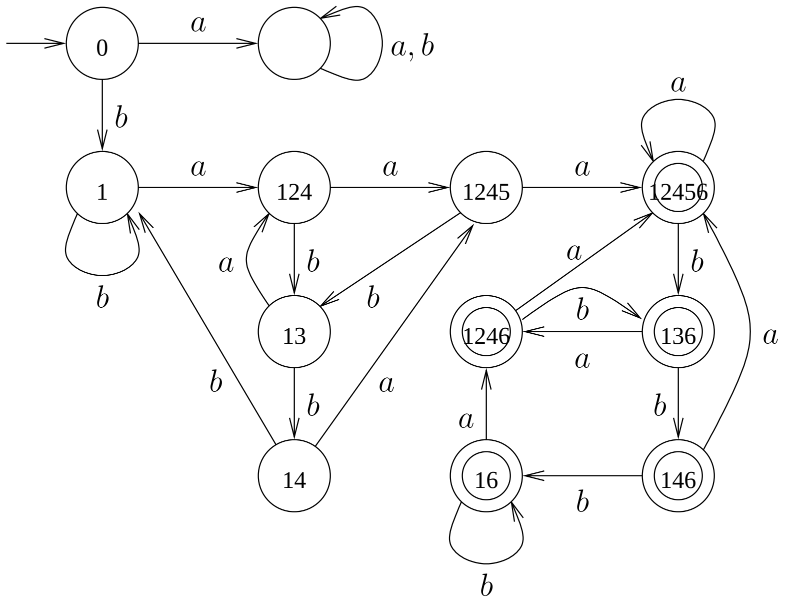
Now we follow transitions from S_3. For the a-transition, the NFA has transitions $1 \xrightarrow{a} 1$, $1 \xrightarrow{a} 2$, and $4 \xrightarrow{a} 5$. In addition, we can get from 2 to 4 again by following ε. So we have a new state $S_4 = \{1, 2, 4, 5\} = \Delta(S_3, a)$. For the b-transition, the NFA has $1 \xrightarrow{b} 1$ and $2 \xrightarrow{b} 3$, with no additional ε-moves. So we have a new state $S_5 = \{1, 3\} = \Delta(S_3, b)$, and the diagram for the DFA is now

$$\begin{tikzcd}
{} \arrow[r] & S_0 \arrow[r, "a"] \arrow[d, "b"] & S_1 \arrow[loop right, "{a,b}"] & \\
& S_2 \arrow[r, "a"] \arrow[loop below, "b"] & S_3 \arrow[r, "a"] \arrow[d, "b"] & S_4 \\
& & S_5 &
\end{tikzcd}$$

Following a from S_4, the NFA has a-transitions to states 1, 2, 5, and 6, with an ε-move to 4. So we get a new state $S_6 = \{1, 2, 4, 5, 6\} = \Delta(S_4, a)$. Since $S_6 \cap F = \{6\} \neq \emptyset$, S_6 is our first final state. Following b from states in S_4, we can reach states 1 and 3 and no others, and there are no additional ε-transitions possible. Since $S_5 = \{1, 3\}$, we have $\Delta(S_4, b) = S_5$.



We continue this way until each state of the DFA has a single outgoing edge for every alphabet symbol. The complete DFA looks like this:



States are labeled with the states of the NFA that they contain.

Note that this is not an optimal solution. For example, all the accepting states can be coalesced into a single accepting state.

7.1 Operations on languages

Fix an alphabet Σ.

Given two languages $A, B \subseteq \Sigma^*$, we have already defined $A \cup B$ and $A \cap B$. Here are some other useful operations:

- $\overline{A} = \Sigma^* - A$, the *complement* of A (in Σ^*). This is the set of all strings over Σ that are not in A.
- $AB = \{xy \mid x \in A \text{ and } y \in B\}$ the set of all concatenations of a string in A followed by a string in B. (Book: $A \circ B$) This operation on languages is associative.
- $A^k = A \cdots A$ (k times). By convention, $A^0 = \{\varepsilon\}$.
- $A^* = A^0 \cup A^1 \cup A^2 \cup \cdots$. This is called the *Kleene closure* of A.

All these operations preserve regularity, i.e., if A and B are regular, then so are all the resulting languages above. (Draw Cartesian product construction for DFAs for intersection.) A DFA for $\overline{A}$ is the same as a DFA for A but with the final status of every state swapped (final states become nonfinal states and vice versa). By DeMorgan's laws, we get that union preserves regularity. This will be easier to see with NFAs in a minute.

This is not in the book. The notion of a clean NFA pops up on several occasions.

Definition 7.2. Let M be an NFA. We say that M is *clean* iff

- M has exactly one final state, which is distinct from the start state,
- M has no transitions into its start state (even self-loops), and
- M has no transitions out of its final state (even self-loops).

Proposition 7.3. *For any NFA M there is an equivalent clean NFA N.*

Proof. If M is not clean, then we can "clean it up" by adding two additional states:

- a new start state with a single ε-transition to M's original start state (which is no longer the start state), and
- a new final state with ε-transitions from all of M's original final states (which are no longer final states) to the new final state.

The new NFA is obviously clean, and a simple, informal argument shows that it is equivalent to the original M. □

For concatenation and union, we give the NFA construction (series and parallel connections, respectively). Define the clean version of an NFA and do the loop construction for the Kleene closure.

Point: You can't get the Kleene closure "for free" just via an infinite union of finite concatenations. Only finite unions preserve regularity; infinite ones do not preserve regularity in general.

7.2 Regular expressions (regexps)

We fix an alphabet Σ for the following discussion. Here, a *language* is any set of strings over the alphabet Σ.

A *regular expression*, or *regexp* for short, is an expression that denotes a particular language over Σ. Primitive (atomic) regular expresions are

- $\emptyset$ (denoting the empty language),
- ε (denoting the language $\{\varepsilon\}$ consisting of just the empty string), and
- a (denoting the language $\{a\}$ consisting of just the string a of length 1) for any symbol $a \in \Sigma$.

If r is a regular expression, we write $L(r)$ for the language denoted by r. We may also just use the regexp itself in place of the language it denotes.

Additionally, regular expressions can be combined with three possible operators: union, concatenation, and Kleene closure. If r and s are regexps, then so are

- $(r \cup s)$ (denoting the language $L(r) \cup L(s)$, the union of $L(r)$ with $L(s)$),
- (rs) (denoting the language $L(r)L(s)$, the concatenation of $L(r)$ with $L(s)$),
- (r^*) (denoting the language $L(r)^*$, the Kleene closure of $L(r)$).

No other regular expressions are possible.

This is an inductive (aka recursive) definition, since it defines new regular expressions in terms of shorter ones (subexpressions).

Parentheses are only used for grouping and can be dropped if we assume the following syntactic precedences: (r) before r^* before rs before $r \cup s$. Both binary operations are associative, so we don't care how they associate.

Examples: Pascal-like floating-point constants, etc.

8 2/11/2008

The language denoted by a regular expression is always regular, because the union, concatenation, and Kleene closure operations preserve regularity. (Draw DFAs for the atomic regexps.) An actual proof could be by induction on the length of the regexp.

Proposition 8.1. *Every regular regular expression denotes a regular language.*

The proof is by induction on the complexity (or length) of a regular expression, using the closure properties of regular languages.

Somewhat surprisingly, the converse is true. The next theorem is the converse of Proposition 8.1.

Theorem 8.2. *Every regular language is denoted by some regular expression.*

The proof (given below) is by construction of a regular expression for a language A given an NFA recognizing A. The construction uses a generalized form of NFA. Two operations in terms of transition diagrams: parallel edge combination and vertex removal.

Definition 8.3. Let REGEXP be the class of all regular expressions. A *generalized nondeterministic finite automaton* (GNFA) is a 5-tuple $N = (Q, \Sigma, \delta, q_0, F)$, where

- Q is a finite set (the set of states),
- Σ is a finite alphabet,
- $q_0 \in Q$ (the start state),

- $F \subseteq Q$ (the set of final states), and
- $\delta : Q \times Q \to \text{REGEXP}$ is the transition function.

The transition diagram for a GNFA is the same for an NFA except that there is exactly one edge from each state q to each state r, and the edges are labeled with regular expressions, the edge (q, r) being labeled with $\delta(q, r)$. Any edge labeled with the regexp $\emptyset$ can be omitted from the diagram (they can never be followed—see below).

A complete accepting computation can be defined for GNFAs analogously to the case of NFAs. The idea is that you can follow an edge by reading a prefix of the remaining input that matches the edge's label, i.e., you read a string that is in the language denoted by the label (which is a regexp). If you really want it, here's a formal definition. Note how similar it is the corresponding definition for NFAs.

Definition 8.4. Let $N = (Q, \Sigma, \delta, q_0, F)$ be a GNFA, and let $w \in \Sigma^*$ be a string. We say that N *accepts* w if there exist strings $w_1, \ldots, w_k \in \Sigma^*$ and states $s_0, \ldots, s_k \in Q$ for some $k \geq 0$ such that

- $w = w_1 \cdots w_k$,
- $s_0 = q_0$,
- $s_k \in F$, and
- $w_i \in L(\delta(s_{i-1}, s_i))$ for all $1 \leq i \leq k$.

The *language recognized by* N (denoted $L(N)$) is the set of all strings over Σ accepted by N.

[If $k = 0$ above, then the right-hand side of the first item is the empty concatenation, which is ε by convention (so $w = \varepsilon$). Also, $q_0 = s_0 = s_k \in F$, and the last item is satisfied vacuously.]

Note that there is no *a priori* length restrictions on the strings w_i for a GNFA, whereas each w_i must have length at most 1 for an NFA. Also note that acceptance only depends on the *languages* denoted by the various regexps on the transitions, not the syntax of the regexps themselves. That is, if any regular expression $\delta(p, q)$ is replaced with an equivalent regular expression (denoting the same language), then the resulting GNFA is equivalent to the original one. Finally, note that along any accepting path above, each language $L(\delta(s_{i-1}, s_i))$ is nonempty (it contains the string w_i), and so no edges labeled with $\emptyset$ are ever traversed.

We can define a clean GNFA analogously to a clean NFA: A GNFA is *clean* iff (1) it has exactly one final state, which is distinct from the start state, (2) all transitions from the final state are labeled $\emptyset$, and (3) all transitions into the start state are labeled $\emptyset$.

Given an NFA, it is easy to construct an equivalent GNFA: Let $N = (Q, \Sigma, \delta, q_0, F)$ be any NFA. Define the GNFA G to be $(Q, \Sigma, \delta', q_0, F)$, where for any $p, q \in Q$ we define the regexp $\delta'(p, q)$ to be the union of all strings in Σ_ε labelling transitions from p to q in N. That is,

$$\delta'(p, q) := \bigcup_{w \in \Sigma_\varepsilon : q \in \delta(p, w)} w.$$

[Each string $w \in \Sigma_\varepsilon$ is clearly a regexp. It is possible the union on the right is empty, in which case an empty union of regexps is naturally $\emptyset$ by convention.] G allows exactly the same state-to-state transitions as N does, reading the same strings, so it is clear that N and G are equivalent. Also, if N is clean, then G is certainly also clean.

Now for the **proof** of Theorem 8.2. Let $A \subseteq \Sigma^*$ be any regular language over Σ. Let N be some NFA recognizing A. We can assume WLOG that N is clean, thus $N = (Q, \Sigma, \delta, q_0, \{q_f\})$ for some unique final state $q_f \neq q_0$. Let $G = (Q, \Sigma, \delta', q_0, \{q_f\})$ be the equivalent (clean) GNFA constructed from N as in the previous paragraph. Starting with G, we will repeatedly remove states not in $\{q_0, q_f\}$, producing a series of equivalent GNFAs, until we get a GNFA with only two states q_0 and q_f and a single nonempty transition—labeled by some regexp R—going from q_0 to q_f. This last GNFA clearly recognizes the language $L(R)$, and it is equivalent to the original NFA N, so we will have $A = L(N) = L(R)$, which proves the theorem.

All we have to do now is describe how to remove a (nonstarting, nonfinal) state from a clean GNFA $G = (Q, \Sigma, \delta', q_0, \{q_f\})$ to get an equivalent GNFA H. Suppose we remove a state $q \in Q - \{q_0, q_f\}$ from G (it doesn't matter which q we pick). We need to define H to accommodate any accepting path in G that may have gone through q, allowing it to bypass q. Such a path would enter q from some other state $s \neq q$, loop some number of times at q, then exit to some other state $t \neq q$. (The path could neither start nor end at q, because q is not the start state or a final state). Let $x = \delta'(s, q)$, let $y = \delta'(q, q)$, and let $z = \delta'(q, t)$ (x, y, and z are regexps). Any path from s to t that goes through q as above reads a string matching x, followed by zero or more strings matching y, followed by a string matching z, all concatenated together. Equivalently, this path reads a string matching xy^*z, so we need to include xy^*z as a possible transition in H from s to t. We do this for all pairs of states $s, t \in Q - \{q\}$, and it follows from our general considerations that H must equivalent to G. (We can prove this last fact formally using Definition 8.4, but the proof is rather tedious and so we won't bother.)

Formally, let G and q be as above. We define H to be the GNFA $(Q - \{q\}, \Sigma, \delta'', q_0, \{q_f\})$, where, for each $s, t \in Q - \{q\}$,

$$\delta''(s, t) = \delta'(s, t) \cup xy^*z,$$

where $x = \delta'(s, q)$, $y = \delta'(q, q)$, and $z = \delta'(q, t)$.

We may simplify $\delta''(s, t)$ if possible. In particular, if $L(x) = \emptyset$ or $L(z) = \emptyset$, then $L(xy^*z) = \emptyset$, and so in this case we can set $\delta''(s, t) := \delta'(s, t)$. If we simplify in this way, then we see that we make H clean provided G is clean. Also, H has one fewer states than G, so repeating this process will lead to a clean equivalent GNFA with state set $\{q_0, q_f\}$ after a finite number of steps. This concludes the proof of Theorem 8.2.

9 2/18/2008

9.1 Proving languages nonregular

Let

$$A = \{0^n 1^n \mid n \geq 0\}.$$

This language is "easy" to recognize: given an input string $w \in \{0, 1\}^*$:

1. reject if there is a 1 followed by a 0 somewhere in w, else

2. count the number of 0's and 1's; if the numbers are equal, then accept; else reject.

Yet we'll see in a minute that A is not regular, i.e., no DFA (equivalently NFA) can recognize A. Intuitively, a DFA can only scan from left to right and has a fixed finite amount of memory (its state). When it passes from the 0's to the 1's, it can't remember exactly how many 0's it has seen, so it can't compare this with the number of 1's. That A is not regular speaks more to the weakness of the DFA model than it does to the difficulty of A.

Lemma 9.1 (Pumping Lemma (NFA version))**.**

If A is a regular language, then
  there exists an integer $p > 0$ such that
    for any string $w \in A$ with $|w| \geq p$,
      there exist strings x, y, z such that
        $xyz = w$,
        $|xy| \leq p$,
        $|y| > 0$, and
        for all $i \geq 0$,
          $xy^iz \in A$.

The Pumping Lemma says that if A is regular, then every sufficiently long string $w \in A$ may be "pumped," that is, some substring y in w may be repeated any number of times, and the resulting pumped string is still in A. We call this "pumping on y."

Applications:

- $\{0^n1^n \mid n \geq 0\}$ is not regular.
- $\{0^m1^n \mid 0 \leq m \leq n\}$ is not regular.
- $\{s \in \{0,1\}^* \mid s$ has the same number of 0's as 1's$\}$ is not regular.
- $\{0^{n^2} \mid n \geq 0\}$ is not regular.
- $\{ww \mid w \in \{0,1\}^*\}$ is not regular.
- $\{0^{2^n} \mid n \geq 0\}$ is not regular.

10 2/20/2008

10.1 Minimal DFAs

A DFA M is *minimal* if there is no DFA N equivalent to M with fewer states.

We will assume a fixed alphabet Σ throughout.

For convenience, we make the following definition: suppose $N = \langle Q, \Sigma, \delta, q_0, F\rangle$ is a DFA. For any state $q \in Q$ and string w over Σ, define $\hat{\delta}(q, w)$ to be the unique state resulting from starting in state q and reading w on the input. That is, $\hat{\delta}$ extends δ to strings of any length.

More formally, we can define $\hat{\delta} : Q \times \Sigma^* \to Q$ inductively:

Base case: $\hat{\delta}(q, \varepsilon) = q$.

Inductive case: $\hat{\delta}(q, aw) = \hat{\delta}(\delta(q, a), w)$ for $a \in \Sigma$ and $w \in \Sigma^*$.[3]

[3]Alternatively, one could define $\hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a)$ for $a \in \Sigma$ and $w \in \Sigma^*$. The two definitions give rise to the same function.

It is clear by the definition that for any strings x and y and state q, $\hat{\delta}(q, xy) = \hat{\delta}(\hat{\delta}(q, x), y)$. Note also that a string w is in $L(N)$ iff $\hat{\delta}(q_0, w) \in F$ (i.e., $\hat{\delta}(q_0, w)$ is an accepting state).

Let L be any language (not necessarily regular) over Σ. For every string w, define $L_w := \{x \mid wx \in L\}$. Note that $L_\varepsilon = L$, and if $L_w = L_{w'}$ then $L_{wa} = L_{w'a}$ for any $a \in \Sigma$, because $x \in L_{wa} \iff wax \in L \iff ax \in L_w \iff ax \in L_{w'} \iff w'ax \in L \iff x \in L_{w'a}$. Define

$$\mathcal{C}_L := \{L_w \mid w \in \Sigma^*\}.$$

Theorem 10.1 (Myhill, Nerode)**.** *Let L be any language. L is regular if and only if $\mathcal{C}_L$ is finite, and in this case, there is a (unique) minimal DFA recognizing L with $|\mathcal{C}_L|$ many states.*

We'll prove Theorem 10.1 via three lemmas.

The first lemma shows that $|\mathcal{C}_L|$ is a lower bound on the number of states of any DFA recognizing a language L.

Lemma 10.2. *If a language L is regular, recognized by some DFA N with state set Q, then $|\mathcal{C}_L| \leq |Q|$.*

Proof. Suppose $N = \langle Q, \Sigma, \delta, q_0, F\rangle$. For any state $p \in Q$, define N_p to be the DFA that is the same as N but whose start state is p. That is, $N_p = \langle Q, \Sigma, \delta, p, F\rangle$. (So, for example, $N = N_{q_0}$.) Now let w be any string over Σ, and let $q := \hat{\delta}(q_0, w)$. We claim that

$$L_w = L(N_q).$$

To see this, note that, for any string $x \in \Sigma^*$, we have

$$\begin{aligned}
x \in L_w &\iff wx \in L && \text{(by definition of } L_w\text{)} \\
&\iff wx \in L(N) && \text{(since } L = L(N)\text{)} \\
&\iff \hat{\delta}(q_0, wx) \in F && \text{(definition of acceptance)} \\
&\iff \hat{\delta}(\hat{\delta}(q_0, w), x) \in F && \text{(since this is the same state)} \\
&\iff \hat{\delta}(q, x) \in F && \text{(definition of } q\text{)} \\
&\iff x \in L(N_q).
\end{aligned}$$

So, with N fixed, we see that L_w only depends on the state obtained by starting in q_0 and reading w, and thus the number of distinct L_w is no more than the number of states of N. □

The next lemma gives $|\mathcal{C}_L|$ (if $\mathcal{C}_L$ is finite) as an upper bound on the number of states necessary for a DFA to recognize L.

Lemma 10.3. *Let L be a language. If $\mathcal{C}_L$ is finite, then L is regular, recognized by a DFA M_L with exactly $|\mathcal{C}_L|$ many states.*

Proof. We construct M_L having state set $\mathcal{C}_L$. We let $M_L = \langle \mathcal{C}_L, \Sigma, \delta, q_0, F\rangle$ where

- $q_0 = L_\varepsilon = L$,
- $\delta(L_w, a) = L_{wa}$ (this is well-defined!), and
- $F = \{q \in \mathcal{C}_L \mid \varepsilon \in q\}$.

Note that if we start in the start state q_0 of M_L and read a string w, then we wind up in state L_w; that is, $\hat{\delta}(q_0, w) = L_w$. You can see this by induction on $|w|$. The base case is when $|w| = 0$, that is, when $w = \varepsilon$. In this case, reading w keeps us in q_0. But $q_0 = L_\varepsilon = L_w$ by definition, so we wind up in state L_w in this case. Now suppse $|w| > 0$ and so $w = ya$ for some string y and alphabet symbol a. By the inductive hypothesis, reading y lands us in state L_y, but then further reading a moves us to state $\delta(L_y, a) = L_{ya} = L_w$.

Intuitively, the state L_w coincides with the set of all strings x such that starting in this state and reading x would lead us to accept wx. This is why we make L_w itself to be an accepting state when we do: if reading ε should make us accept w. Immediately after this proof, we give a concrete example of an M_L.

We must show that $L = L(M_L)$. For any $x \in \Sigma^*$ we have (below, q_0, $\hat{\delta}$, and F are all with respect to M_L):

$$\begin{aligned} x \in L &\iff \varepsilon \in L_x \ \text{ (by definition of } L_x) \\ &\iff L_x \in F \ \text{ (by definition of } F) \\ &\iff \hat{\delta}(q_0, x) \in F \ \text{ (since } L_x = \hat{\delta}(q_0, x) \text{ from above)} \\ &\iff x \in L(M_L) \ \text{ (by definition of acceptance).} \end{aligned}$$

Thus M_L recognizes L. $\square$

Example. Let $\Sigma = \{a, b\}$ and let L be the set of all strings that contain three consecutive b's somewhere in the string. So L is denoted by the regular expression $(a \cup b)^* bbb(a \cup b)^*$. With a little thought, we can see that $|\mathcal{C}_L| = 4$. In fact, $\mathcal{C}_L = \{L_\varepsilon, L_b, L_{bb}, L_{bbb}\}$. These four languages are all distinct:

- $L_\varepsilon = L$.
- L_b contains bb (which is not in L_ε), but does not contain b.
- L_{bb} contains b (which is neither in L_ε nor in L_b), but does not contain ε.
- L_{bbb} contains all strings over $\{a, b\}$, including ε.

Furthermore, any L_w is equal to one of these four. For example, $L_{abaabb} = L_{bb}$ and $L_{bbbaaaaa} = L_{bbb}$. (For any w, how do you tell which of the four L_w is?) We thus get the following DFA M_L for L, as constructed in the proof of Lemma 10.3:

- The state set is $\mathcal{C}_L$.
- The start state is L_ε.
- The sole accepting state is L_{bbb}, as this is the only one of the four languages that contains ε.

- The transition function δ is as follows:

$$\begin{aligned}
\delta(L_\varepsilon, a) &= L_a = L_\varepsilon, \\
\delta(L_\varepsilon, b) &= L_b, \\
\delta(L_b, a) &= L_{ba} = L_\varepsilon, \\
\delta(L_b, b) &= L_{bb}, \\
\delta(L_{bb}, a) &= L_{bba} = L_\varepsilon, \\
\delta(L_{bb}, b) &= L_{bbb}, \\
\delta(L_{bbb}, a) &= L_{bbba} = \{a,b\}^* = L_{bbb}, \\
\delta(L_{bbb}, b) &= L_{bbbb} = L_{bbb}.
\end{aligned}$$

□

The third and final lemma proves that M_L is unique, thus completing the proof of Theorem 10.1. First, we will say that a DFA $N = \langle Q, \Sigma, \delta, q_0, F\rangle$ is *economical* if every state of N is reachable from the start state, that is, for every $q \in Q$ there is a string w such that $q = \hat{\delta}(q_0, w)$. Clearly, a DFA N is equivalent to an economical DFA with the same number or fewer states—just remove any states of N that are unreachable from q_0. These states will never be visited in a computation, so they are completely irrelevant.

Lemma 10.4. *Let L be a regular language. The DFA M_L of Lemma 10.3 is the unique (up to renaming of states) minimal DFA recognizing L.*

Proof. Let $N = \langle Q, \Sigma, \delta, q_0, F\rangle$ be any economical DFA recognizing L. We can map the states of N surjectively onto the states of M_L in a way that preserves the transition function, start state, and accepting states. For every $q \in Q$, define $f(q) = L(N_q)$, where N_q is as in the proof of Lemma 10.2. This defines a mapping f with domain Q, mapping states to languages. We will verify the following properties of f:

1. The range of f is exactly $\mathcal{C}_L$, that is, f maps Q onto $\mathcal{C}_L$ (the state set of M_L).

2. $f(q_0) = L_\varepsilon = L$ (the start state of M_L).

3. Let $q \in Q$ be a state, and let w be any string such that $f(q) = L_w$ (such a w exists by item (1)). For any $a \in \Sigma$, we have $f(\delta(q, a)) = L_{wa}$ (which is the result of the transition function of M_L being applied to the state $L_w = f(q)$ and a).

4. For any $q \in Q$, we have $q \in F$ if and only if $\varepsilon \in f(q)$ (that is, if and only if $f(q)$ is an accepting state of M_L).

For (1): First, let q be any state in Q. Since N is economical, there is some string w such that $q = \hat{\delta}(q_0, w)$. In the proof of Lemma 10.2 we showed that $L_w = L(N_q)$. Since $L(N_q) = f(q)$ by definition, this shows that $f(q) \in \mathcal{C}_L$, and thus the range of f is a subset of $\mathcal{C}_L$. Lastly, we must show that every element of $\mathcal{C}_L$ is equal to $f(q)$ for some $q \in Q$. Given any string w, we define $q = \hat{\delta}(q_0, w)$. Then again by the proof of Lemma 10.2, we have $L_w = L(N_q)$, which also equals $f(q)$. Thus every L_w is in the range of f, and so f maps Q surjectively onto $\mathcal{C}_L$.

For (2): Clearly, $f(q_0) = L(N_{q_0}) = L(N) = L = L_\varepsilon$.

For (3): Let $r = \delta(q, a)$. For any string x, we have

$$\begin{aligned} x \in f(r) &\iff x \in L(N_r) \\ &\iff \hat{\delta}(r, x) \in F \\ &\iff \hat{\delta}(q, ax) \in F \\ &\iff ax \in L(N_q) \\ &\iff ax \in f(q) \\ &\iff ax \in L_w \\ &\iff wax \in L \\ &\iff x \in L_{wa}. \end{aligned}$$

Therefore $f(r) = L_{wa}$.

For (4): Clearly, $q \in F$ if and only if $\varepsilon \in L(N_q)$, because $\hat{\delta}(q, \varepsilon) = q$.

Now we're ready to prove the lemma. Let $M_L = \langle \mathcal{C}_L, \Sigma, \delta', L_\varepsilon, F' \rangle$ be the minimal DFA for L constructed in the proof of Lemma 10.3. Suppose $N = \langle Q, \Sigma, \delta, q_0, F \rangle$ is *any* minimal DFA recognizing L. We show that N and M_L are the same DFA after relabeling states. Since N is minimal, it must also be economical (otherwise, we could remove at least one unreachable state to get a smaller equivalent DFA). Thus we have the surjective mapping $f : Q \to \mathcal{C}_L$ defined above. Also, by Lemmata 10.2 and 10.3 we must also have $|Q| = |\mathcal{C}_L|$ (since N is minimal). Thus f must be a bijection (perfect matching) between Q and $\mathcal{C}_L$. This is the relabeling we want: f maps the start state of N to the start state of M_L (by (2)); it maps accepting states to accepting states and nonaccepting states to nonaccepting states (by (4)); finally, it preserves the transition function, that is, for any $q \in Q$ and $a \in \Sigma$,

$$f(\delta(q, a)) = L_{wa} = \delta'(f(q), a)$$

by (3), where w is any string such that $f(q) = L_w$.

This shows that N and M_L are identical up to the relabeling f, which proves the lemma. □

10.2 Minimizing a DFA

Nice as they are, the results in the last section do not give an explicit general procedure to construct a minimal DFA equivalent to a given DFA. Often one can determine M_L by inspection, as we did in the Example, above. Nevertheless, we would like a general algorithm that, given a DFA $N = \langle Q, \Sigma, \delta, q_0, F \rangle$, constructs M_L, where $L = L(N)$.

The algorithm to do this works in two steps: (1) remove all states of N that are not reachable from q_0, making N economical, and (2) collapse remaining states together that we find are *equivalent*. We'll assume that we have already performed (1), so that N is economical.

For any state $q \in Q$, define N_q as in the proof of Lemma 10.2, above.

10.2.1 State equivalence and distinguishability

We say that two states p and q in Q are *distinguishable* if $L(N_p) \neq L(N_q)$, that is, there is some string x such that one of $\hat{\delta}(p, x)$ and $\hat{\delta}(q, x)$ is accepting and the other is not. Such a string x, if it exists, *distinguishes* p from q. (Such an x is in $L(N_p) \triangle L(N_q)$, the symmetric difference of $L(N_p)$ and $L(N_q)$.[4])

If p and q are indistinguishable (i.e., not distinguishable by any string, i.e., $L(N_p) = L(N_q)$), then we say p and q are *equivalent* and write $p \approx q$. This is obviously an equivalence relation. Intuitively, $p \approx q$ means that our acceptance or rejection of any string w does not depend on whether we start in state p or in state q before reading w. (Although it's not necessary for the current development, you may observe that $p \approx q$ iff p and q are mapped to the same state by the f defined in the proof of Lemma 10.4.)

The meat of the algorithm is to determine which pairs of states are equivalent. This is done using a table-filling technique. We methodically find *distinguishable* pairs of states; when we can't find any more, the pairs of states left over must be equivalent. For convenience, assume that $Q = \{1, \ldots, n\}$. We use a two-dimensional array T with entries $T[p, q]$ for all $1 \le p, q \le n$. We'll keep T symmetric in what follows, i.e., any value assigned to $T[p, q]$ will automatically also be assigned to $T[q, p]$ implicitly. (Also, we will have no use for diagonal entries $T[p, p]$, so actually, only the proper upper triangle of T need be stored: those entries $T[p, q]$ where $p < q$.) We proceed as follows:

```
Initially, all entries of $T$ are blank;
FOR each pair $(p, q)$ of states with $p < q$, DO
    IF $p \in F$ or $q \in F$ but not both, THEN
        $T[p, q] \leftarrow$ X;
REPEAT
    FOR each pair $(p, q)$ with $p < q$ and each $a \in \Sigma$, DO
        IF $T[p, q]$ is blank and $T[\delta(p, a), \delta(q, a)] =$ X, THEN
            $T[p, q] \leftarrow$ X
UNTIL no entries are marked X in one full iteration of this loop
```

When finished, it will be the case that an entry $T[p, q]$ is blank if and only if $p \approx q$.

To see the "if" part, notice that we only mark an entry with X if we have *evidence* that the two states in question are distinguishable. If $T[p, q]$ gets marked in the initial FOR-loop, it is because ε distinguishes p from q. For entries marked in the REPEAT-loop, we can proceed by induction on the number of steps taken by the algorithm when an entry is marked to show that the corresponding states are distinguishable: if $T[p, q]$ is marked in the REPEAT-loop, it is because $T[\delta(p, a), \delta(q, a)]$ was marked sometime previously, for some $a \in \Sigma$. By the inductive hypothesis, $\delta(p, a)$ and $\delta(q, a)$ are distinguished by some string w. But then, aw clearly distinguishes p from q, so marking $T[p, q]$ is correct. This proves the "if" part.

To show the "only if" part, we need to show that all distinguishable pairs of states are eventually marked with X. Suppose this is not the case. We call (p, q) a *bad pair* if $p \not\approx q$ but $T[p, q]$ is left blank by the algorithm. Our assumption is that there is at least one bad pair. Let w be a string distinguishing some bad pair (p, q), and assume that w is as short as any string distinguishing any bad pair. It must be that $w \neq \varepsilon$, since otherwise, ε distinguishes p from q, which in turn implies that either $p \in F$ or $q \in F$ but not both; but then $T[p, q]$ is marked X in the initial FOR-loop, so (p, q) is not a bad pair. So we must have $w = ay$ for some $a \in \Sigma$ and $y \in \Sigma^*$. Then, y distinquishes $r = \delta(p, a)$ from $s = \delta(q, a)$, and since $|y| < |w|$, (r, s) is not a bad pair (by the minimality of w).

[4]For any sets A and B, we define $A \triangle B = (A - B) \cup (B - A) = (A \cup B) - (A \cap B)$, i.e., the set of all z such that $z \in A$ or $z \in B$ but not both.

So $T[r,s]$ is marked with an X at some point. But then, on the first iteration of the REPEAT-loop following the marking of $T[r,s]$, we mark $T[p,q]$ with X, which contradicts the assumption that (p,q) is a bad pair. Thus there are no bad pairs.

Now that we can tell whether any two states are equivalent, we are ready to construct the minimum DFA M for L. Let the state set of M be the set $Q/\approx$ of equivalence classes of Q under the $\approx$-relation. For any $q \in Q$ we let $[q]$ denote the equivalence class containing q.

We define the transition function δ_M for M as follows: for any $q \in Q$ and $a \in \Sigma$, let

$$\delta_M([q], a) = [\delta(q, a)].$$

This is a legitimate definition, because $p \approx q$ clearly implies $\delta(p,a) \approx \delta(q,a)$, and so the transition does not depend on which representative of the equivalence class we use. Also note that when we extend δ_M to $\hat{\delta}_M$ acting on all strings, we can routinely check that

$$\hat{\delta}_M([q], w) = [\hat{\delta}(q, w)]$$

for all $q \in Q$ and $w \in \Sigma^*$.

We define the start state of M to be $[q_0]$, and the set of accepting states of M to be $F_M = \{[r] \mid r \in F\}$.

We must verify two things:

1. M recognizes L, and
2. M is minimal.

For (1), let w be any string.

$$\begin{aligned} w \in L &\iff \hat{\delta}(q_0, w) \in F \\ &\iff [\hat{\delta}(q_0, w)] \in F_M \\ &\iff \hat{\delta}_M([q_0], w) \in F_M \\ &\iff w \in L(M). \end{aligned}$$

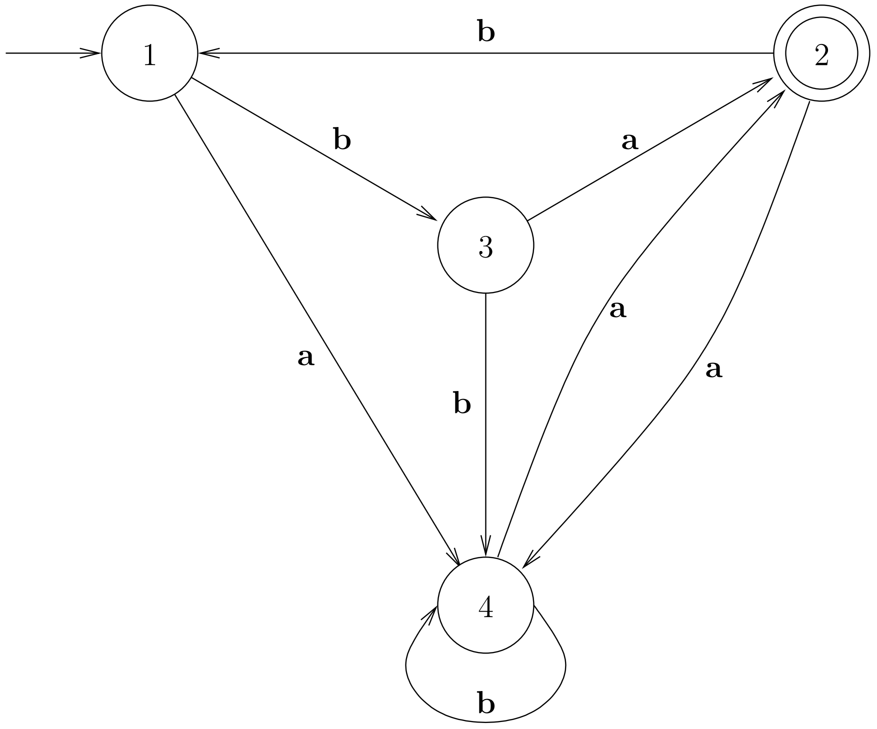
Thus $L = L(M)$.

For (2), first note that since N is economical, so is M: if $[r]$ is any state of M, let w be such that $\hat{\delta}(q_0, w) = r$; then $\hat{\delta}_M([q_0], w) = [\hat{\delta}(q_0, w)] = [r]$, and thus $[r]$ is reachable from the start state $[q_0]$ via w. Now we show that $\mathcal{C}_L = \left\{L(M_{q'}) \mid q' \in Q/\approx\right\}$. By adapting the proof of Lemma 10.2 we get that $L_w = L(M_{q'})$ for any string w, where $q' = \hat{\delta}_M([q_0], w)$. But since M is economical, every state of M is of this form for some w, so indeed $\mathcal{C}_L = \left\{L(M_{q'}) \mid q' \in Q/\approx\right\}$. We'll be done if we can show that any two distinct states of M are distinguishable, for then $L(M_{p'}) \neq L(M_{q'})$ for all $p', q' \in Q/\approx$, and so it must be that $|Q/\approx| = |\left\{L(M_{q'}) \mid q' \in Q/\approx\right\}| = |\mathcal{C}_L|$, which implies that M is minimal by Lemmata 10.2 and 10.3.

To see that all distinct states of M are distinguishable, notice that if $[p] \neq [q]$, then $p \not\approx q$, and so p and q are distinguished (in N) by some string x. We claim that the same string x also distinguishes $[p]$ from $[q]$ in M. Let $r = \hat{\delta}(p, x)$ and let $s = \hat{\delta}(q, x)$. Then either $r \in F$ or $s \in F$ but not both. We also have $\hat{\delta}_M([p], x) = [r]$ and $\hat{\delta}_M([q], x) = [s]$ by definition, so we're done if either $[r]$ or $[s]$ is accepting (in M) but not both. Suppose, WLOG, that $r \in F$. Then $[r]$ is accepting in M by definition. Further, $s \notin F$. This implies that $[s]$ cannot be accepting in M: otherwise, there is some $s' \in [s] \cap F$, but then $s' \approx s$, and so $s' \notin F$ because $s \notin F$—contradiction. Thus $[s]$ is not accepting in M. (Any equivalence class in $Q/{\approx}$ either contains only accepting states or only nonaccepting states.) A similar argument works assuming $s \in F$. Thus $[p]$ and $[q]$ are distinguishable in M.

11 2/25/2008

Example. Consider the following DFA, which is not minimal:



We use the table-filling method described above to find all distinguishable pairs of states. Initially, T is all blank:

T	1	2	3	4
1				
2				
3				
4				

We first look for pairs of states (p, q) that are ε-distinguisable, i.e., where one of p and q is a final state but not both. Such pairs are $(1, 2)$, $(2, 3)$, and $(2, 4)$. We put an X in the corresponding entries of T, making sure that if we mark $T[x, y]$ with an X, we also mark $T[y, x]$ with an X.

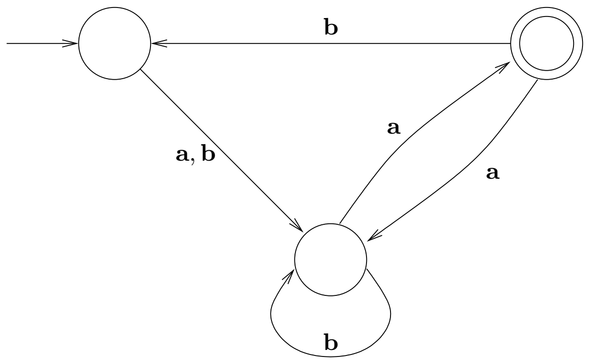
T	1	2	3	4
1		X		
2	X		X	X
3		X		
4		X		

Entering the REPEAT-loop, first iteration, we look for transitions into previously marked (i.e., ε-distinguishable) pairs of states. We only look for transitions from pairs of states for which T has a blank entry. We find the pair $(1,3)$, because $\delta(1,\mathbf{a}) = 4$, $\delta(3,\mathbf{a}) = 2$, and $T[4,2] = T[2,4] = X$. Transitioning on the symbol **a** again, we also find $(1,4)$, because $T[\delta(1,\mathbf{a}),\delta(4,\mathbf{a})] = T[4,2] = T[2,4] = X$. Thus we set $T[1,3] = T[3,1] = T[1,4] = T[4,1] = X$. We don't find anything else looking for **b**-transitions, so the table after the first full iteration of the REPEAT-loop is

T	1	2	3	4
1		X	X	X
2	X		X	X
3	X	X		
4	X	X		

We then run the second iteration of the REPEAT-loop, again looking for transitions into previously marked pairs of states. We don't find anything new, so we break out of the loop and out of the table-building algorithm, with the final table as it is above.

The only missing X is for the pair $(3,4)$. So states 3 and 4 are indistinguishable, and all other pairs of distinct states are distinguishable. So for the minimum DFA, we collapse states 3 and 4 into a single state and also collapse transitions as you would expect. Here is the equivalent minimum DFA:



11.1 Introduction to Turing machines

Turing machines (TMs) are formal devices just like DFAs, but they are much more powerful, and their behavior is correspondingly more subtle. They can be used as formal renderings of what we informally call, "algorithms."

Turing machines have a

- one-way infinite read/write tape (head can move in both directions),
- finite-state control,
- accepting and rejecting states (two halting states; take effect immediately),
- start state, and
- finite transition function $(q,a) \overset{\delta}{\mapsto} (q',a',d)$, where d is either L (left) or R (right).

Turing "program" M_1 for $B = \{w\#w|win\{0,1\}^*\}$:

On input w:

1. Scan input to make sure it contains a single # symbol. If not, reject.
2. Zig-zag across #, crossing off left symbol, and checking corresponding symbol on right side. If different, reject. Otherwise, cross off right symbol.
3. When all left symbols crossed off, check right to see if any symbols left. If so, reject; else, accept.

Definition 11.1. [Formal Definition of a TM] A *Turing machine* (TM) is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where

- Q is a finite set (the set of *states*),
- Σ is a finite alphabet (the *input alphabet*),
- Γ is a finite alphabet (the *tape alphabet*) where $\Sigma \subseteq \Gamma$,
- $q_0, q_{accept}, q_{reject} \in Q$ and $q_{accept} \neq q_{reject}$ (q_0 is the *start state*, and q_{accept} and q_{reject} are the two *halting states*—the *accepting state* and *rejecting state*, respectively, and
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is a function (the *transition function*, where L and R are distinct objects denoting *left* and *right*, respectively).

Also, $\Gamma \cap Q = \emptyset$ (not in the textbook), and Γ contains a special *blank symbol* (_) that is not in Σ.

A TM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ computes as follows: Start with input string $w \in \Sigma^*$ on the leftmost part of the tape (the rest of the tape blank), state q_0, the tape head scanning the leftmost square. Repeatedly apply the transition δ function based on the current state q and the symbol a currently being scanned. The value of $\delta(q, a)$ is a triple (r, b, d) that says what to do next: change the state to r, write b on the tape (where the head is, thus overwriting a), then move the head one cell in direction d (either left or right). (If M tries to move off of the left end of the tape, the head stays put instead.)

Note the differences between a TM and a DFA. A TM can exhibit much more subtle and complex behavior:

- A TM's head can move in both directions (one cell at a time).
- A TM can alter the contents of its tape.
- A TM can use as much of the blank portion of its tape (beyond the input) as it wants.
- A TM can use any symbols in its tape alphabet Γ, but it only computes on input strings over the input alphabet Σ.
- A TM's computation ends just when it enters one of the two halting states. This can happen at any time, e.g., before or long after it has read the entire input.
- A TM may *never* enter either of its halting states. In this case, the TM runs forever, and we say that the TM "loops."

[Give a diagram for M_1.] Show how it computes on 1011#10111.

Some conventions: Omit q_{reject} and any transitions to it. Omit any transitions out of q_{accept}. Here is a simplified transition table for M_1 (0 is the start state):

$$\begin{array}{rcl}
0,0 & \mapsto & 1,x,R \\
0,1 & \mapsto & 2,x,R \\
0,\# & \mapsto & 7,\#,R \\
1,0 & \mapsto & 1,0,R \\
1,1 & \mapsto & 1,1,R \\
1,\# & \mapsto & 3,\#,R \\
2,0 & \mapsto & 2,0,R \\
2,1 & \mapsto & 2,1,R \\
2,\# & \mapsto & 4,\#,R \\
3,x & \mapsto & 3,x,R \\
3,0 & \mapsto & 5,x,L \\
4,x & \mapsto & 4,x,R \\
4,1 & \mapsto & 5,x,L \\
5,x & \mapsto & 5,x,L \\
5,\# & \mapsto & 6,\#,L \\
6,0 & \mapsto & 6,0,L \\
6,1 & \mapsto & 6,1,L \\
6,x & \mapsto & 0,x,R \\
7,x & \mapsto & 7,x,R \\
7,\_ & \mapsto & \text{accept}
\end{array}$$

M_1 actually has ten states. We don't list the rejecting state at all, and we don't bother listing any transitions out of the accepting state. Any "missing" transitions above go to the rejecting state (in that case we don't care what is written or which way the head moves).

12 2/27/2008

We need to formalize the notion of a TM computation. First we need the concept of a configuration.

A configuration of a TM M is a "snapshot" of M's computation at a given time. We can represent configurations by strings. More formally,

Definition 12.1. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ be a TM. A *configuration* of M is represented by a string of the form uqv, where $u, v \in \Gamma^*$ and $q \in Q$. We call q the *state* of the configuration.

The string uqv represents the state of M's computation at some point in time. It means that M's current state is q, the entire current tape contents to the left of the cell being scanned is u, and v represents the contents of the tape starting with the scanned cell and extending (at least) through the rightmost nonblank symbol. Note that the same configuration can be represented by more than one string, namely, by padding it on the right with blanks. Thus $1101q00$, $1101q00\_$, $1101q00\_\_$, etc. all represent the same configuration.

Let M be as in Definition 12.1, above, and let C_1 and C_2 be configurations of M. We say that C_1 *yields* C_2 if M can legally go from configuration C_1 to configuration C_2 in a single step. More formally, let $a, b \in \Gamma$ be any symbols over Γ, let $u, v \in \Gamma^*$ be any strings over Γ, and let $p \in Q$ be any nonhalting state. There are two cases, one for each possible direction the head can move.

1. Suppose $\delta(p, a) = (q, c, R)$ for some $q \in Q$ and $c \in \Gamma$. Then any configuration represented by $upav$ yields the configuration represented by $ucqv$.
2. Suppose $\delta(p, b) = (q, c, L)$ for some $q \in Q$ and $c \in \Gamma$. Then
   (a) any configuration represented by $uapbv$ yields the configuration represented by $uqacv$, and
   (b) any configuration represented by pbv yields the configuration represented by qcv.

You should check that these cases match up with your intuition about M's computation.

Definition 12.2. Let M be as in Definition 12.1 and let $w \in \Sigma^*$ be any string over M's input alphabet.

- The *start configuration* of M on input w is represented by $q_0 w$.
- Any configuration with state q_{accept} is an *accepting configuration* of M.
- Any configuration with state q_{reject} is a *rejecting configuration* of M.
- A *halting configuration* of M is either an accepting or rejecting configuration of M.

Note that a halting configuration does not yield any further configurations, and a nonhalting configuration yields exactly one configuration.

Definition 12.3. Let M and w be as in the previous definition. The *computation* of M on input w is the unique sequence $C_0, C_1, C_2, \ldots$ of configurations of M such that C_0 is the start configuration of M on w, each C_i yields C_{i+1}, and either

1. the sequence is finite, ending with an accepting configuration,
2. the sequence is finite, ending with a rejecting configuration, or
3. the sequence is infinite, containing no halting configuration.

In Case (1) we say that the computation is *accepting* and that *M accepts w*. In Case (2) we say that the computation is *rejecting* and that *M rejects w*. In Case (3) we say that the computation is *infinite* (or *looping*) and that *M runs forever* (or *loops*) on w. In Cases (1) or (2), we also say that *M halts* on w.

For each TM and input w there is a unique computation, and it is one of the three types, above. [Also, you can generate any finite amount of it!] We can define language recognition for TMs just as we did for DFAs.

Definition 12.4. Let M be as in the previous definition. The language *recognized* by M (or the *language of* M, denoted $L(M)$) is defined as

$$L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}.$$

The next definition is analogous to the way we define a language to be regular via DFAs.

Definition 12.5. A language is *Turing-recognizable* (aka *computably enumerable* or *recursively enumerable*) iff some TM recognizes it.

If a TM M does not accept an input string w, then it either rejects or loops on w. If the latter, is often difficult or impossible to tell in advance that this will happen. So we have the strange and uncomfortable situation where $w \notin L(M)$ but we never know for sure that this is the case. We can start simulating the computation of M, but we can only discover finitely much of it in any given finite amount of time, and that may not be enough to predict M's eventual behavior. It's generally desirable when this situation can never arise.

Definition 12.6. A TM is a *decider* iff it halts on all its inputs. If TM M recognizes language A (that is, $A = L(M)$) and is a decider, then we also say that M *decides* A.

Definition 12.7. A language is *decidable* (aka *computable* or *recursive*) iff some TM (necessarily a decider) decides it.

Box-and-lightbulb picture of TMs.

Transducers. A TM can do more than accept, reject, or loop on an input. We can use TMs to compute functions $\Sigma^* \to \Sigma^*$. Such TMs are called *transducers.*

Definition 12.8. A function $f : \Sigma^* \to \Sigma^*$ is *computable* if there is a Turing machine M that, on every input $w \in \Sigma^*$ halts with just $f(w)$ on its tape.

13 3/3/2008

13.1 Hacking TMs

You can think of TMs as programs in some rudimentary programming language, such as assembly language or machine language. Despite their simplicity, they are just as powerful as any higher-level general-purpose programming language, or any other mathematical model of computation (this is also true of assembly and machine language).

The next couple of lectures is meant to convince you of this.

What are the building blocks of general-purpose programming? Data types and structures (numbers (integers and finite-precision floating point), character strings, arrays, lists, records, trees, etc.), operations on data structures (assignment, arithmetic operations on numbers, array indexing, string/list splicing and concatenation, etc.) and high-level algorithm-building components (sequences of statements, if-then-else constructs, while- and other kinds of loops, procedures and functions, recursion). All of these can be implemented with TMs.

Example: Maintaining a list Given strings $x_1, \dots, x_n \in \Sigma^*$, we can represent the list (n-tuple) $(x_1, \dots, x_n)$ by the string $\#x_1\# \cdots \#x_n$, where $\#$ is a "separator" symbol not in Σ.

Putting an end marker on the left (exercise).

Compute the length of a list (n). Given 0^i, find x_i.

Go to higher-level descriptions of TMs.

13.2 Multitape TMs

Let $k \geq 1$. A k-tape TM has k separate 1-way infinite tapes, numbered 1 to k, with a head scanning each one. The machine changes state and moves each head in some direction depending on the combination of k symbols being scanned.

Definition 13.1. Let $k \geq 1$. A k-tape Turing machine is a tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where all components are as with a standard (1-tape) TM, except that $\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, S, R\}^k$. Here, L, S, and R denote the head movement directions "left," "stationary," and "right," respectively.

Let M be a k-tape machine as above, and let $w \in \Sigma^*$. The computation of M on w starts in state q_0 with w being in the leftmost portion of tape 1 (also called the *input tape*). The rest of tape 1 is blank and all the other tapes are completely blank initially, and the heads are all scanning the leftmost cells of their respective tapes. At any point in the computation, M will be in some state q and the heads will be scanning symbols $a_1, \dots, a_k \in \Gamma$ on tapes $1, \dots, k$, respectively. If $\delta(q, (a_1, \dots, a_k)) = (r, (b_1, \dots, b_k), (d_1, \dots, d_k))$, say, and q is not a halting state, then for all $1 \leq i \leq k$:

- each a_i is overwritten with b_i on tape i,
- the ith head either moves one cell right if $d_i = R$, stays put without moving if $d_i = S$, or moves one cell left if $d_i = L$ (with the usual exception of not moving if scanning the leftmost cell), and
- M's state becomes r.

This all constitutes one *step* of the computation. Note that the transition depends on the state and *all* k symbols being scanned.

More examples: Reversing a string (exercise). Addition of unsigned integers. Basic list operations: head, tail, empty test, list construction.

14 3/5/2008

14.1 Encoding of Finite Objects

By *finite object* I mean any piece of data that's representable in computer memory. Such an object contains a finite amount of information and can be encoded as a finite binary string in some reasonable way. When you store data in a file, you are essentially encoding it as a binary string, because a file is just a finite sequence of bits. You store it in a way that you can easily retrieve it later. This makes the encoding *reasonable.*

Lots of different types of finite objects can be easily encoded as binary strings:

- natural numbers (e.g., usual binary representation),
- integers (e.g., two's complement),
- strings, possibly over a bigger alphabet (e.g., replace each symbol with a fixed-length binary string),
- finite lists of objects (e.g., first encode each object as a binary string, then build the list using separators as we did in the previous lecture with alphabet $\{0, 1, \#\}$, then re-encode this string as a binary string as in the last item above),
- finite sets (e.g., as lists of their members),
- relations over finite sets (e.g., as finite sets of ordered pairs, which themselves are 2-element lists)
- functions with finite domain and codomain (e.g., as relations, above)
- graphs, trees, etc.

The reason to encode objects as strings is so that TMs can compute with them. Formally, TMs only compute with strings, but under reasonable encodings, we can imagine that TMs are computing with these other kinds of finite objects as well.

For any finite object $\mathcal{O}$, we let $\langle \mathcal{O} \rangle$ denote the encoding of $\mathcal{O}$ as a binary string under some reasonable encoding. If $\mathcal{O}_1, \ldots, \mathcal{O}_n$ are finite objects, we use $\langle \mathcal{O}_1, \ldots, \mathcal{O}_n \rangle$ to denote the string encoding the list of the objects.

Here are some more types of easily encodable objects that we've been introduced to in this class: DFAs, NFAs, regexps, and TMs. These are all finite objects built out of components in the list above. For example, a TM M can be encoded as a string $\langle M \rangle$ that is input to another TM. What can TMs do with TMs as inputs? One answer: simulate them.

14.2 TMs Simulating TMs

One of the most important capabilities of a TM is that it can simulate other TMs. One can easily conceive of a 3-tape TM U that takes as input a string of the form $\langle M, w \rangle$, where M is a (1-tape) TM and w is a string over M's input alphabet, and proceeds to simulate the computation of M on input w. For instance, it may copy $\langle M \rangle$ to tape 2, leaving just an encoding of w on tape 1, then repeatedly scan the string encoding M on tape 2 to see what M's next move is, based on the (encoded) symbol currently being scanned on tape 1 and the current state of M, which is maintained (in encoded form) on tape 3. It would be quite tedious to write down a complete formal description of U, but you could do it if pressed. (We haven't completely specified U because we haven't said what U does with input strings that are not of the form $\langle M, w \rangle$ where M is a TM and w is a string. We often don't care what U does, but just for completeness: In this and in all future cases, we'll assume that the TM first checks the syntax of its input string and rejects if the string is not of the assumed form.)

The machine U is called a *universal Turing machine*, because it can mimic the behavior of any Turing machine on any input whatsoever, given a string describing the machine and its input. It was Alan Turing who first showed that universal TMs exist, and they served as the chief inspiration for the stored-program (M is the program, w its input) electronic computers that came later and that are ubiquitous today.

Can U simulate itself? U has three tapes and it only simulates TMs with one tape. But below we show that any k-tape machine can be simulated by a one-tape machine, so we can assume WLOG that U is a standard, 1-tape machine.

Exercise 14.1. Given U as above and any string w, show how to generate an infinite sequence of distinct strings $w_1, w_2, \ldots$ such that U behaves the same on inputs $w_1, w_2, \ldots$ as it does on w.

SPOILER: The solution to the exercise is instructive. Given a string w, let

$$\begin{aligned} w_1 &= \langle U, w\rangle, \\ w_2 &= \langle U, w_1\rangle, \\ &\vdots \\ w_{i+1} &= \langle U, w_i\rangle, \\ &\vdots \end{aligned}$$

This solution does not even attempt to analyze the innards of U or how its computation proceeds, and such an attempt would usually be futile. Turing machines can be devious that way. In general, TMs can defy any attempt to determine beforehand their eventual behavior.

There are lots of variations on the universal machine concept. There is a Turing machine C that, given input $\langle M, w, t\rangle$ where M is a TM, w is a string over M's input alphabet, and t is a nonnegative integer, simulates M on input w for t steps or until M halts, whichever comes first. If M does not halt on w within t steps, then C rejects $\langle M, w, t\rangle$. I'll call such a machine C a *clocked universal Turing machine.*

You can also imagine a TM that takes encodings of two or more TMs with inputs and simulates them all simultaneously, alternating one step at a time for each.

Recall the definitions of decidability and Turing recognizability (T-recognizability). It is obvious that if a language A is decidable, then it is T-recognizable. We'll see later that the converse does not hold. But we do have the following facts.

Proposition 14.2. *If a language A is decidable, then its complement $\overline{A}$ is decidable.*

Given a decider for A, just swap the accepting and rejecting states to get a decider for $\overline{A}$.

Proposition 14.3. *Let A be any language. If both A and its complement $\overline{A}$ are Turing recognizable, then A is decidable.*

Proof. Suppose $A = L(M_1)$ and $\overline{A} = L(M_2)$ for some TMs M_1 and M_2. To decide whether or not some string w is in A, run the two machines M_1 and M_2 simultaneously on input w. Exactly one of these machines will eventually accept w. If M_1 accepts, then $w \in A$; if M_2 accepts, then $w \notin A$. Here's the algorithm:

"On input w:

1. Run M_1 and M_2 simultaneously on input w until one of them accepts.
2. If M_1 accepts, then accept; else reject."

Let M be a TM that implements this algorithm. It is clear that M is a decider and that $A = L(M)$; therefore M decides A. □

We will often describe algorithms in an informal, high-level way like this. It should come as no surprise that the algorithm above can be implemented by a Turing machine, but formally describing such a machine is astoundingly tedious and not particularly illuminating.

14.3 Simulating a k-tape TM with a standard (1-tape) TM

Book method: maintain a k-tuple on the tape whose entries are the contents of the k-tapes of the machine being simulated. "Mark" the location in each entry where the head should be. Each step of the k-tape machine is simulated by scanning and updating the entire list (left to right then right to left).

Alternate method: use bigger tape alphabet and multitrack tape. (Each tape alphabet symbol encodes corresponding contents of the k tapes, and for each tape, whether or not the head is scanning that cell.)

Technicality: must first prepare the "active" region of the tape, mark beginning and end.

Each step of the k-tape machine is simulated by a full pass of the active region of the 1-tape machine:

- going left to right: gather information about which symbol is being scanned on which tape,
- going right to left: simulate the writing and head movement for each tape.

All additional information, including the state of the k-tape machine, constitutes a fixed finite amount of information and so can be stored in the state of the simulating machine.

On to a formal description. This is the last time I'll describe a Turing machine formally. We fix a k-tape TM

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject}).$$

We'll refer to the cells on each tape as cell $0, 1, 2, \ldots$ starting with the leftmost cell. We build a standard, 1-tape Turing machine

$$\mathcal{M} = (\mathcal{Q}, \Sigma, \widetilde{\Gamma}, \Delta, Q_0, Q_{accept}, Q_{reject})$$

to simulate M, where the $(i+1)$st cell of $\mathcal{M}$ contains within a single "multitrack" symbol information about all the contents of the ith cells of the k tapes of M, along with whether or not each cell is currently being scanned by one of M's heads. This makes $\mathcal{M}$'s tape alphabet rather large:

$$\widetilde{\Gamma} = \Gamma \mathbin{\dot\cup} \{\$\} \mathbin{\dot\cup} (\{\to, \_\} \times \Gamma)^k,$$

where $\dot\cup$ represents *disjoint union*—the union of sets assumed to be disjoint. As well as containing all of M's tape alphabet and a new end marker \$, $\widetilde{\Gamma}$ also contains the multitrack symbols which we will depict as $k \times 2$ arrays. For example, a typical multitrack symbol looks like

$$\begin{bmatrix} \to & a \\ & b \\ \to & c \\ & b \end{bmatrix}$$

for a 4-tape machine. Here, $a, b, c \in \Gamma$, and the arrows indicate which cells are being scanned. This symbol represents the situation where some cell on tape 1 contains an a and is being scanned, the cell right below it on tape 2 contains b (not scanned), the cell below that on tape 3 has c (scanned), and below that there is a b (not scanned) on tape 4.

The transition diagram of $\mathcal{M}$ consists of a preparation phase followed by the main simulation loop. The preparation phase starts with some input string $w \in \Sigma^*$ on the tape and ends with the tape recording the starting configuration of M. The main loop is composed of two subloops: the reading subloop (where the head moves from left to right gathering transition information), and the writing subloop (where the head alters the tape accordingly while moving from right to left. For simplicity, our $\mathcal{M}$ will only mimic M in terms of accepting, rejecting, or looping, and we won't care about what is left on $\mathcal{M}$'s tape at the end.

The states of $\mathcal{Q}$ are as follows:

- the start state Q_0,
- preparation states $P_1(a)$ (for all $a \in \Gamma$) and P_2 and P_3,
- reading states $R(q, \vec{v})$ where $q \in Q$ and each $\vec{v}$ is a length-k column vector of symbols in Γ that may be only partially filled in,
- writing states $W(r, m)$ where $r \in Q$ and each m is a $k \times 2$ array where the first entry in each row is a symbol from Γ and the second entry is a direction in $\{L, S, R\}$. Only some of m's rows may be filled in.
- head placement states $H(r, m, i, d)$ for all r and m as in the previous item, all $1 \le i \le k$, and all $d \in \{L, R\}$, and finally
- halting states Q_{accept} and Q_{reject}.

14.3.1 The Transitions of $\mathcal{M}$

We now describe $\mathcal{M}$'s transition function Δ from the start state onward. You should draw the transition diagram from the information below.

The preparation phase. Remember that initially, $\mathcal{M}$ is given some input string $w \in \Sigma^*$ on its tape. The first job is to prepare the initial configuration of M, along with an end marker to delineate the nonblank portion of $\mathcal{M}$'s tape. As usual, any missing transitions are assumed to go directly to Q_{reject}.

1. For all $a \in \Gamma$, set $\Delta(Q_0, a) := (P_1(a), \$, R)$. (Actually, we only need to do this for all $a \in \Sigma \cup \{\_\}$, since that's all we would ever see.)
2. For all $a \in \Sigma$ and $b \in \Gamma$, set $\Delta(P_1(a), b) = \left(P_1(b), \left[\begin{array}{c} \quad\quad a \\ \\ \end{array}\right], R\right)$, where $\left[\begin{array}{c} \quad\quad a \\ \\ \end{array}\right]$ is the multitrack symbol corresponding to an a on tape 1, blanks on all the other tapes, and no scanning heads.
3. Set $\Delta(P_1(\_), \_) := \left(P_2, \left[\begin{array}{c} \quad\quad \\ \\ \end{array}\right], L\right)$. Here, $\left[\begin{array}{c} \quad\quad \\ \\ \end{array}\right]$ is the multitrack symbol representing all blanks and no scanning with heads.

4. Set $\Delta(P_2, A) := (P_2, A, L)$ for all $A \in \widetilde{\Gamma} - \{\$\}$.
5. Set $\Delta(P_2, \$) := (P_3, \$, R)$.
6. For all $a \in \Gamma$, set $\Delta\left(P_3, \begin{bmatrix} & a \end{bmatrix}\right) := \left(P_3, \begin{bmatrix} \rightarrow & a \\ \vdots & \\ \rightarrow & \end{bmatrix}, L\right)$, where $\begin{bmatrix} \rightarrow & a \\ \vdots & \\ \rightarrow & \end{bmatrix}$ is the multitrack symbol with a on tape 1, blanks elsewhere, and all cells being scanned with heads (all arrows in the first column).
7. Set $\Delta(P_3, \$) = \left(R\left(q_0, \begin{bmatrix} \; \end{bmatrix}\right), R\right)$, where $\begin{bmatrix} \; \end{bmatrix}$ is the empty column vector of symbols (no symbols filled in).

This ends the preparation phase. Note that the current state of $\mathcal{M}$ is $R\left(q_0, \begin{bmatrix} \; \end{bmatrix}\right)$, which records that the current state of M is q_0 (its start state). $\mathcal{M}$'s head is scanning the leftmost multitrack symbol just to the right of the \$ in the leftmost cell.

The reading phase. A state of the form $R(q, \vec{v})$ records the current state of M as q and records the scanned symbols seen thus far in the reading phase. The main simulation loop starts with state $R\left(q_0, \begin{bmatrix} \; \end{bmatrix}\right)$, which begins the reading phase and records the current state of M as q_0 and the fact (with a completely blank vector) that we have not found any scanned symbols on M's tapes yet.

We start moving to the right, gathering information about scanned symbols. We do this by setting, for every nonhalting state $q \in Q - \{q_{accept}, q_{reject}\}$, partially filled k-vector $\vec{v}$, and multitrack symbol $A = \begin{bmatrix} & a_1 \\ \vdots & \vdots \\ & a_k \end{bmatrix}$,

$$\Delta(R(q, \vec{v}), A) := (R(q, \vec{u}), A, R),$$

where $\vec{u}$ is the same as $\vec{v}$ except that for every row i of A that has an arrow, we fill in the (empty) ith entry of $\vec{u}$ with a_i. This effectively records in $\mathcal{M}$'s state the symbol being scanned on the ith tape of M. Of course, if there are no arrows in A, then $\vec{u} = \vec{v}$, so we keep the same state and just move to the right. When all entries of the vector $\vec{v}$ are filled in, we have complete information about what transition M will take, and we know we will not see any more arrows in the multitrack symbols, so the reading phase ends.

Transition to the writing phase. Given a reading state $R(q, \vec{v})$ of $\mathcal{M}$ where all the k entries of $\vec{v}$ are filled in with symbols from Γ, we make a transition to a writing state that depends on q, $\vec{v}$, and the fixed transition function δ of M. This is the only place in the simulation where we use δ. Suppose that $\vec{v} = \begin{bmatrix} a_1 \\ \vdots \\ a_k \end{bmatrix}$ for some $a_1, \dots, a_k \in \Gamma$, and that $\delta(q, (a_1, \dots, a_k)) = (r, (b_1, \dots, b_k), (d_1, \dots, d_k))$, where $r \in Q$, $b_1, \dots, b_k \in \Gamma$, and $d_1, \dots, d_k \in \{L, S, R\}$. Then we set

$$\Delta(R(q, \vec{v}), A) := \left(W\left(r, \begin{bmatrix} b_1 & d_1 \\ \vdots & \vdots \\ b_k & d_k \end{bmatrix}\right), A, L\right)$$

for any $A \in \widetilde{\Gamma}$. The new writing state records the output of M's transition function δ, i.e., the actions to be performed on each of M's tapes. For example, if $d_i = R$ for some i, then when we find an arrow in the ith row of a multitrack symbol, we must change the corresponding Γ symbol to b_i and "move" the arrow to the neighboring multitrack symbol to the right.

The writing phase. We scan $\mathcal{M}$'s tape from right to left, altering symbols and moving arrows according to the information contained in $\mathcal{M}$'s writing state. Moving an arrow to the right presents no difficulty, but there is a slight technical difficulty if an arrow moves to the left: we will encounter that arrow again in our scan, and we don't want to be confused about whether we already moved that arrow or not. To deal with this, whenever we perform the action on the ith tape in state $W(r,m)$, we remove the entries in the ith row of m, making that row empty. We then only take action on tape i if the ith row of m is nonempty. This guarantees that we only act on each of M's tapes once.

Let $W(r,m)$ be any writing state, where $r \in Q$ and m is any $k \times 2$ array whose first column contains symbols from Γ and whose second column contains directions from $\{L,S,R\}$, and were some of m's rows may be empty. Let $A \in \widetilde{\Gamma}$ be any multitrack symbol, and let i be least such that the ith row of m is nonempty and there is an arrow in the ith row of A. If no such i exists, then there is no action to be performed at this step, and we just set $\Delta(W(r,m),A) := (W(r,m),A,L)$.

Otherwise, we perform the required action on M's ith tape. Let (b,d) be the ith row of m (so $b \in \Gamma$ and $d \in \{L,S,R\}$). There are two cases depending on whether the ith head moves or not. In either case, let m' be m with its ith row emptied out.

$d = S$: The head does not move. We set $\Delta(W(r,m),A) := (W(r,m'),A')$, where A' is the same as A except the second entry of its ith row is b.

$d \in \{L,R\}$: The head moves. Let A' be the same as A except that the arrow in the i row is removed and the second entry in the ith row is b. Set $\Delta(W(r,m),A) := (H(r,m',i,d),A',d)$.

Head placement states. The job of the head placement state $H(r,m,i,d)$ is to place an arrow in the ith row of the multitrack symbol being scanned by $\mathcal{M}$, then move $\mathcal{M}$'s head back to where it just came from (the direction opposite d). Let d' be the opposite of d, that is, $d' \in \{L,R\} - \{d\}$. (Note that this uniquely defines d' since $d \in \{L,R\}$.) Let A be any multitrack symbol, and let A' be the same as A except that there is an arrow in the ith row of A'. Set $\Delta(H(r,m,i,d),A) := (W(r,m),A',d')$.

There are two exceptional transitions out of $H(r,m,i,d)$. One is where $d = R$ and $\mathcal{M}$ is scanning the regular blank symbol $\_ \in \Gamma$. This occurs when M's ith head moves further to the right than any of M's heads have moved before, and thus we must lay down a new multitrack symbol, increasing the range of our tape simulation. In this case, set

$$\Delta(H(r,m,i,R),\_) := \left(W(r,m), \begin{bmatrix} \rightarrow & \end{bmatrix}, L\right),$$

where the new multitrack symbol has all blanks in the second column and a single arrow in the first column at the ith row. The other case is where M's ith head tries to move left but can't because it is already scanning the leftmost cell of its tape. We notice this when we scan the end marker \$. In this case, all we need to do is move $\mathcal{M}$'s head one to the right before planting the arrow. Set

$$\Delta(H(r,m,i,L),\$) := (H(r,m,i,L),\$,R).$$

You should convince yourself that this quick fix works.

The writing phase ends when we are in some writing state $W(r, m)$ and scanning the end-marker \$ in the leftmost cell. We then transition to the start of the next iteration of the simulation loop:

$$\Delta(W(r, m), \$) := \left(R\left(r, \left[\quad\right]\right), R\right)$$

for all $r \in Q$ and $k \times 2$ arrays m. This concludes the description of the writing phase and the whole simulation loop.

Halting transitions. Finally, when we are at the start of a simulation loop and we notice that M is in a halting state, then we immediately halt in the same way. For any $\vec{v}$ and $A \in \widetilde{\Gamma}$, set

$$\begin{aligned} \Delta(R(q_{accept}, \vec{v}), A) &:= (Q_{accept}, A, L), \\ \Delta(R(q_{reject}, \vec{v}), A) &:= (Q_{reject}, A, L). \end{aligned}$$

All this guarantees is that $\mathcal{M}$ will halt iff M halts, and if so, in the same way (accepting or rejecting). This is fine for language recognition and language decision, but we may have reason to simulate M more closely. Suppose M is being used as a transducer (i.e., to compute a function $\Sigma^* \to \Sigma^*$, and one of M's tapes (the kth tape, say) is designated as the *output* tape, which when M halts contains the output of the function being computed. Our 1-tape $\mathcal{M}$ would then need to leave its tape contents (and perhaps its head location) identical to that of M's output tape before halting. I'll leave it as an exercise to add such an output-preparation module to $\mathcal{M}$.

Exercise 14.4. Modify the machine $\mathcal{M}$ above so that if M halts, $\mathcal{M}$ halts with its tape contents and head location identical to M's kth tape.

This completes the description of $\mathcal{M}$.

15 3/17/2008

15.1 Nondeterministic Turing machines

Nondeterministic TMs (NTMs as opposed to DTMs): analogous with NFAs: transition function gives a set of possible actions.

Nondeterministic computation best viewed as a tree (nodes = configurations, root is start configuration). Machine accepts if there is some computation path that leads to the accept state.

Theorem 15.1. *For every NTM N there is an equivalent DTM D (i.e., such that $L(D) = L(M)$).*

Proof idea: D uses breadth-first search on N's tree (why not depth-first search?). Three tapes for D:

tape 1 always contains the input and is never altered,

tape 2 maintains a copy of N's tape contents on some branch of its computation tree, and

tape 3 keeps track of D's location in N's tree, specified by a sequence of numbers in $\{1, \ldots, b\}$, where b is the maximum branching of N.

Tape 3 cycles through all sequences in length-first lexicographical order. For each sequence, the corresponding branch of N's computation branch is simulated (using tape 2) from the beginning for as many steps are there are elements of the sequence. We accept if we ever notice that N reaches an accepting configuration.

An NTM is a *decider* iff it halts on all branches on all inputs (König's Lemma implies that the whole computation tree is finite for any input).

NTMs are equivalent to DTMs at deciding languages.

15.2 Enumerators

Define enumerators. A language is *computably enumerable* (aka *recursively enumerable*) iff some enumerator enumerates it.

Theorem 15.2. *A language is Turing recognizable iff it is computably enumerable.*

Proof. First the forward direction. Suppose $A \subseteq \Sigma^*$ is a T-recognizable language. Let M be a TM recognizing A ($A = L(M)$). Let E be an enumerator that runs as follows:

For all strings x in length-first lexicographical order:

1. If x is not of the form $\langle w, t\rangle$ where $w \in \Sigma^*$ and $t \in \mathbb{N}$, then go on to the next x.
2. Otherwise, $x = \langle w, t\rangle$ for some $w \in \Sigma^*$ and $t \in \mathbb{N}$. Run M on input w for t steps or until M halts, whichever comes first.
3. If M accepts w within t steps, then print w.
4. Go on to the next x.

Note that E spends a finite amount of time with each string x, so it manages to cycle through all possible strings. We need to show that E prints a string w iff $w \in A$. E prints a string w only if it notices that M accepts w, and thus $w \in L(M) = A$. Conversely if $w \in A$, then M accepts w after some finite number of steps t. Letting $x = \langle w, t\rangle$, we see that when E gets to the string x, it will run M on w for t steps, notice that M accepts w, then print w. This shows that E enumerates A, and so A is computably enumerable.

For the reverse direction, suppose that A is enumerated by an enumerator E. Let M be a TM implementing the following algorithm:

"On input $w \in \Sigma^*$:

1. Run E indefinitely. If w is ever printed, then accept."

Clearly, M accepts exactly those strings that are printed by E, and thus $L(M) = A$. Therefore, A is T-recognizable. □

Here's another way to look at T-recognizability:

Theorem 15.3. *A language $A \subseteq \Sigma^*$ is Turing-recognizable iff either $A = \emptyset$ or A is the range of a computable function.*

Recall that the range of a function is the set of things that are actually output by the function on some input.

Proof. ($\Longrightarrow$) Let $A = L(M)$ for some TM M. If $A = \emptyset$, we're done, so assume $A \neq \emptyset$ and fix an element $x_0 \in A$ (our "fall-back" string). Let f be the function computed by the following algorithm:

```
"On input $x$:
  1. If $x$ is not of the form $\langle w, t\rangle$ for some $w \in \Sigma^*$ and $t \in \mathbb{N}$, then output $x_0$ and halt.
  2. Otherwise $x = \langle w, t\rangle$. Run $M$ on input $w$ for up to $t$ steps.
  3. If $M$ accepts $w$ within $t$ steps, then output $w$ and halt.
  4. Otherwise, output $x_0$ and halt."
```

Clearly, f either outputs x_0 or some element w accepted by M. Thus f only outputs strings in A. To see that all strings in A are output by f, let $w \in A$ be arbitrary. Since $A = L(M)$, M accepts w after some finite number of steps t. But f is defined so that $f(\langle w, t'\rangle) = w$ for any $t' \geq t$. So w is in the range of f. This concludes showing that A is the range of f.

($\Longleftarrow$) Suppose that A is either empty or the range of a computable function f. We want to show that A is T-recognizable. The empty set $\emptyset$ is recognized by a TM defined as follows:

```
"On input w:
  1. Reject."
```

Now assume A is the range of computable f. Let E be an enumerator that runs as follows:

```
"For every string $x$:
  1. Compute $y = f(x)$.
  2. Print $y$."
```

Clearly, E enumerates the range of f, which is A, so A is computably enumerable and hence T-recognizable. □

16 3/19/2008

16.1 The Church-Turing Thesis

Lots of different mathematical models of computation were developed in the first half of the 20th century—well before the advent of electronic computers. There are TMs of various sorts (Alan Turing 1936), the λ-calculus (Alonzo Church 1936), Post production systems, Herbrand-Gödel-Kleene systems, Kleene's six schemata, combinatory logic, etc. With electronic computers have come various reasonable, general-purpose programming languages that can also be described formally as models of computation (Fortran, C, Perl, C++, Ruby, ML, Java, Haskell, etc.) as well as theoretical models that closely mimic the behavior of typical hardware (the RAM model (random access machine)). All these models are equivalent, describing exactly the same class of algorithms, because they all can simulate each other. It was Turing who really discovered this, showing that his Turing machines could simulate (and be simulated by) all the other models that existed at the time. Before that, it was not at all clear that these models were equivalent, or which was the "right" model of computation. Unlike the other models at the time, which could be quite abstract, Turing machines had a clear, down-to-earth, physically intuitive semantics that made them a convincing model of computation. The existence of a universal Turing machine also leant much credibility to the following principle:

Church-Turing Thesis: All these (equivalent) formal definitions precisely capture the intuitive (or even physical) notion of an algorithm.

Further evidence: lots of attempts to strengthen the TM model (more tapes, multidimensional tapes, several heads, etc.) don't provide any additional power, i.e., they don't make things computable that weren't before with regular TMs. Similarly, attempts to weaken TMs (restricting head movements, what can be written, etc.) either lead to ridiculously and obviously weak models of computation or else give the same power as the original TMs. The difference is never subtle.

One of my favorite examples: Read-only TMs with two auxillary blank tapes with end-markers. This is just as powerful as the original TM model. Data for intermediate calculations is stored in the positions of the auxillary heads, which must move out doubly exponentially far in the number of steps of the standard TM being simulated. If you only have one auxillary tape, then such a device is so weak that it cannot even recognize some simple (context-free) languages. These results date back to the early 1960s. Each blank tape acts like a counter that holds a natural number (corresponding to the head position). A counter can be incremented, decremented, left alone, and tested for zero. These machines are also called two-counter automata.

All computational models that are equivalent to Turing machines are said to be capable of *universal computation.*

Here are some more models of universal computation, some rather exotic:

- Boolean circuits (directed acyclic graphs with nodes labeled with Boolean operators such as AND, OR, or NOT). These are appealing because they can closely mimic actual hardware. Issue: a single Boolean circuits can only take inputs of some fixed length; for universal computation, one must take a uniform family of circuits, one for each input length; there are various ways one can define "uniform" here.

- Type-0 (unrestricted) grammars. These form one of the levels of the Chomsky hierarchy.

- Tag systems. These describe processes by which a FIFO queue is modified in a simple way.

- John Conway's Game of Life. This is a specific two-dimensional two-state cellular automaton, meant to crudely simulate the behavior of bacteria in a Petri dish.

- Rule 110. This is a specific, very simple, one-dimensional two-state cellular automaton, conjectured by Stephen Wolfram and proved by Matthew Cook (around the year 2000) to be universal.

- brainf__k (pardon the bowdlerization). This is an extremely simple programming language that has an extremely tiny interpreter. It is universal, but almost unreadable and very inefficient.

Emphasis switches from TM to algorithm. Three levels of description: formal (TM), implementation level (informal TM), and high-level (algorithmic).

Two current research papers of ours, both mostly accessible:

- "The complexity of finding SUBSEQ(A)"
(`http://www.cse.sc.edu/~fenner/papers/finding-subseq.pdf`)

- "The complexity of learning SUBSEQ(A)" (`http://www.cse.sc.edu/~fenner/papers/learning-subseq.pdf`)

16.2 Chapter 4: Decidable languages

The acceptance problem for DFAs—given a DFA B and a string w, does B accept w?—can be encoded as the following language:

$$A_{\text{DFA}} = \{\langle B, w\rangle \mid B \text{ is a DFA accepting } w\}.$$

A_{DFA} is decidable (via TM M): Implementation details with multitape machine: write description of B on separate work tape, keep track of current state on another work tape. Read the input w as B would read it.

A_{NFA}, the acceptance problem for NFAs, is defined analogously. It is decidable via TM N using M as subroutine: On input $\langle B, w\rangle$, N first converts B into an equivalent DFA C, then runs M on $\langle C, w\rangle$.

The acceptance (or matching) problem for regular expressions,

$$A_{\text{REX}} = \{\langle R, w\rangle \mid w \text{ matches regexp } R\}$$

is decidable: Convert R into an equivalent NFA B (Thm 1.28 is *effective*, i.e., algorithmic or computable), then run N on input $\langle B, w\rangle$.

The emptiness problem for DFAs

$$E_{\text{DFA}} = \{\langle A\rangle \mid A \text{ is a DFA and } L(A) = \emptyset\}$$

is decidable: Search from start state of A for a final state:

```
$T$ = "On input $\langle A\rangle$ where $A$ is a DFA:
  1. Mark the start state of $A$.
  2. Repeat until no new states get marked:
     (a) Mark any state that has a transition coming into it from any state that is already marked.
  3. If no accept state is marked, accept, else reject."
```

The equivalence problem for DFAs

$$EQ_{\text{DFA}} = \{\langle A, B\rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$$

is decidable: Construct DFA C recognizing the language $L(A) \triangle L(B)$. Run T on input $\langle C\rangle$. ($L \triangle L'$ is the *symmetric difference* of L and L', defined as $(L - L') \cup (L' - L)$.)

$E_{\text{NFA}}, E_{\text{REX}}, EQ_{\text{NFA}}, EQ_{\text{REX}}$ are defined analogously and are all decidable. But: think about how to decide these more efficiently than just converting everything to DFAs, which can be exponentially large compared with the original NFAs or regexps.

Regular $\Longrightarrow$ decidable $\Longrightarrow$ Turing-recognizable. We know the first $\Longrightarrow$ is strict. What about the last one?

16.3 An undecidable language: The Halting Problem (4.2)

The acceptance problem for TMs is

$$A_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM that accepts string } w\}.$$

Proposition 16.1. *A_{TM} is Turing-recognizable.*

Proof. Recall the universal Turing machine

$U =$ "On input $\langle M, w\rangle$, where M is a TM and w is a string:

1. Simulate M on input w.
2. If M ever enters its accept state, accept; if M ever enters its reject state, reject."

Clearly by its definition, $A_{\text{TM}} = L(U)$, and thus A_{TM} is T-recognizable. □

U loops on $\langle M, w\rangle$ iff M loops on w, so U does not decide A_{TM}. If U had some way of finding out that M would not halt on w, then it could reject.

Theorem 16.2. *A_{TM} is undecidable.*

Proof. The proof uses the *diagonalization* method (Georg Cantor, 1873): the same method used to prove that there are uncountably many reals, etc. (We'll skip over this mostly.)

Assume (for the purposes of contradiction) that there is a TM H that is a decider for A_{TM}. That is, for all TMs M and strings w:

$$\begin{aligned} M \text{ accepts } w \quad &\Longrightarrow \quad H \text{ accepts } \langle M, w\rangle, \\ M \text{ does not accept } w \quad &\Longrightarrow \quad H \text{ rejects } \langle M, w\rangle. \end{aligned}$$

Let D be the following machine:

$D =$ "On input $\langle M\rangle$ where M is a TM:

1. Run H on input $\langle M, \langle M\rangle\rangle$ until it halts.
2. If H accepts $\langle M, \langle M\rangle\rangle$ then reject; else accept."

Thus for every TM M,

- D accepts $\langle M\rangle$ if M does not accept $\langle M\rangle$;
- D rejects $\langle M\rangle$ if M accepts $\langle M\rangle$.

Now suppose we run D on input $\langle D\rangle$. Substituting D for M above, we have:

- D accepts $\langle D\rangle$ if D does not accept $\langle D\rangle$;
- D rejects $\langle D\rangle$ if D accepts $\langle D\rangle$.

Clearly this is impossible. Therefore no such machine H exists, and thus A_{TM} is undecidable. □

The proof of Theorem 16.2, as well as being an example of diagonalization, uses a paradox of future forcasting: You cannot reliably fortell the future to someone who has the ability to change it. D, when run on input $\langle D\rangle$, is able to "ask" the "fortune teller" H, "Will I eventually accept?" That is, D on input $\langle D\rangle$ asks H whether or not the computation D on input $\langle D\rangle$ will eventually accept. H must commit to an answer in a finite amount of time, whereafter, D just does the opposite of what H predicts, making H wrong for this computation at least.

Corollary 16.3. $\overline{A_{\mathrm{TM}}}$ *is not Turing-recognizable.*

Proof. We know that A_{TM} is T-recognizable. If $\overline{A_{\mathrm{TM}}}$ were also T-recognizable, then A_{TM} would be decidable by Proposition 14.3, but this is not the case by Theorem 16.2. □

The *halting problem* is defined as the language

$$HALT_{\mathrm{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM that halts on input string } w\}.$$

17 3/24/2008

17.1 Reducibility (Chapter 5)

Informally, a problem A *reduces* to a problem B if you can easily solve A given a solution for B. If A and B are computational problems, then A reduces to B means that there is an easy way to transform a given instance of A into one or more instance of B so that, given solutions to those instances of B, one can easily solve the original instance of A. A reduction from A to B says that (in some sense) A is no more difficult to solve than B, or equivalently that B is at least as difficult as A. In particular, if A and B are languages, and A reduces to B, then if B is decidable then so is A, and (equivalently) if A is undecidable then so is B. Our most common method of showing a language to be undecidable is to reduce some previously known undecidable language to it.

There are many different kinds of reductions. The most general kind that we will see is a Turing reduction. A Turing reduction (from language A to language B) is an algorithm that decides A by using answers to questions about membership in B "for free."

Definition 17.1. Let B be any language. A *Turing reduction to* B is an algorithm that at any point may include a test of the form,

"If $w \in B$, then ..."

where w is some string prepared by the algorithm. Furthermore, assuming the tests are always answered truthfully, the algorithm halts (either accepts or rejects) on all inputs. A Turing reduction to B is also called a *decision procedure that uses (or queries) oracle* B.

We think of B as a black box, or oracle, that we can feed strings (queries) to and it will say "yes" or "no" according to whether or not the string is in B. The definition says nothing about whether B is decidable, i.e., whether the black box can be replaced by an actual algorithmic decision procedure for B as a subroutine. If R is a Turing reduction to B, we define the *language decided by R with oracle B* as you'd expect: the set of all strings accepted by R.

We won't go into details here, but a Turing reduction can be modeled formally by an *oracle Turing machine* (OTM). An OTM is a Turing machine equipped with an extra read/write tape—its *oracle query tape*—and three specially designated states: $q_?$ (the *query state*), q_{yes} (the *"yes"* *answer state*), and q_{no} (the *"no" answer state*). Given some language $B \subseteq \Sigma^*$, the OTM computing on input string w with oracle B acts just like a regular multitape TM with input w, except that at any point during a computation the OTM may enter the state $q_?$ with some string $x \in \Sigma^*$ sitting on the oracle query tape. Then in one step, the machine either enters state q_{yes}, if $x \in B$, or enters state q_{no} if $x \notin B$. (Nothing is written and the heads don't move in this step.) In this way, the OTM can ask a membership question to the oracle and receive the answer in its state. The computation of the OTM with oracle B may loop on some input, in which case the OTM does not compute a Turing reduction to B. Note that an OTM can be described as a finite object, independent of any oracle it computes with, but any computation of the OTM may vary depending on the oracle. If M is an OTM and B is a language, we may write M^B for the machine M equipped with oracle B. This leads to definitions of OTMs analogous to regular TMs, e.g., $L(M^B)$ is the language of all input strings accepted by M^B, M^B is a decider (i.e., Turing reduction to B) iff it halts on all input strings, etc.

Definition 17.2. Let A and B be languages. We say that A *Turing reduces*, or *T-reduces* to B (denoted $A \leq_{\mathrm{T}} B$) if there is a Turing reduction (oracle Turing machine) that decides A using oracle B. We may also say that A is *decidable in B* or *decidable relative to B* in this case.

Let A, B, and C be any languages. Here are some easy but important facts.

- $A \leq_{\mathrm{T}} A$ ($\leq_{\mathrm{T}}$ is reflexive).
- If $A \leq_{\mathrm{T}} B$ and B is decidable, then A is decidable.
- If $A \leq_{\mathrm{T}} B$ and A is undecidable, then B is undecidable (this follows from the previous statement by pure logic).
- If $A \leq_{\mathrm{T}} B$ and $B \leq_{\mathrm{T}} C$, then $A \leq_{\mathrm{T}} C$ ($\leq_{\mathrm{T}}$ is transitive). This relativizes the previous item to oracle C, replacing "decidable" with "decidable in C."

Use T-reducibility to show the undecidability of $\mathit{HALT}_{\mathrm{TM}}$, E_{TM}, *HELLO WORLD*, $\mathit{REGULAR}_{\mathrm{TM}}$, $\mathit{EQ}_{\mathrm{TM}}$. The second and third require an algorithm to produce (the description of) a Turing machine. (More on that later.)

18 3/26/2008

Another fact about Turing reducibility:

Easy Fact 18.1. *$\overline{A} \leq_{\mathrm{T}} A$ for any language A.*

Proof. Given A, here's the Turing reduction:

```
"On input $w$:
    1. If $w \in A$ then reject; else accept."
```

□

Let's apply this with $A = A_{\mathrm{TM}}$. Then $\overline{A_{\mathrm{TM}}} \leq_{\mathrm{T}} A_{\mathrm{TM}}$. Recall that A_{TM} is Turing recognizable but $\overline{A_{\mathrm{TM}}}$ is not. Thus Turing reductions do not preserve Turing recognizability (although they do preserve decidability).

We now introduce a more restricted reduction that preserves both decidability and Turing recognizability.

18.1 Mapping reducibility

Definition 18.2. Let $A, B \subseteq \Sigma^*$ be languages. We say that A *is mapping reducible to* B or that A *is m-reducible to* B ($A \leq_{\mathrm{m}} B$) if there exists a computable function $f : \Sigma^* \to \Sigma^*$ such that, for all $w \in \Sigma^*$,

$$w \in A \iff f(w) \in B.$$

We also say in this case that f is a *mapping reduction* or *m-reduction* from A to B, and that f *mapping reduces* or *m-reduces* A to B.

A mapping reduction f provides a more restrictive special case of a Turing reduction. It only applies to decision problems (encoded by languages), where a given instance w of a decision problem A is transformed into a single instance $f(w)$ of the decision problem B with the same answer.

Easy Fact 18.3. *Let A and B be languages. If $A \leq_{\mathrm{m}} B$ then $A \leq_{\mathrm{T}} B$.*

Proof. Let f m-reduce A to B. Then the following oracle algorithm T-reduces A to B:

"On input w:

1. Compute $x = f(w)$.
2. If $x \in B$ then accept; else reject."

Since $w \in A$ iff $x \in B$, this correctly decides A with oracle B. □

Since m-reducibility implies T-reducibility, it preserves decidability, i.e., if $A \leq_{\mathrm{m}} B$ and B is decidable, then A is decidable. The same is true for Turing recognizability (unlike with T-reductions):

Proposition 18.4. *Let A and B be languages. If $A \leq_{\mathrm{m}} B$ and B is Turing recognizable, then A is Turing recognizable.*

Proof. Suppose that f m-reduces A to B and that $B = L(M)$ for some TM M. Let

$N =$ "On input w:

1. Compute $x = f(w)$.
2. Simulate M on input x (and do whatever M does)."

Clearly, for any input string w, N accepts w iff M accepts $f(w)$ iff $f(w) \in B$ iff $w \in A$. Thus $A = L(N)$ and so A is Turing recognizable. □

We'll use this fact in the contrapositive to show that various languages A are not Turing recognizable, usually by showing that $\overline{A_{\mathrm{TM}}} \leq_{\mathrm{m}} A$. For if this is the case, then A cannot be T-recognizable; otherwise, $\overline{A_{\mathrm{TM}}}$ would be T-recognizable, which we know is not true.

Another consequence of Proposition 18.4 is that $\overline{A_{\mathrm{TM}}} \not\leq_{\mathrm{m}} A_{\mathrm{TM}}$. This is because the right-hand side is Turing recognizable, but the left-hand side is not.

Most of the Turing reductions we built to show the undecidability of various languages above can be turned into m-reductions.

Proposition 18.5. $A_{\mathrm{TM}} \leq_{\mathrm{m}} \overline{E_{\mathrm{TM}}}$.

Proof. This works as before, except that we output the machine R explicitly as the value of the function.

Fix the TM

$M_0 =$ "On input w: reject."

Clearly, $L(M_0) = \Sigma^*$, and so $\langle M_0 \rangle \in E_{\text{TM}}$, or, more to the point, $\langle M_0 \rangle \notin \overline{E_{\text{TM}}}$. Let the function f be computed as follows:

```
f = "On input x:
  1. If x is not of the form ⟨M, w⟩ where M is a TM and w is a string, then output
     ⟨M_0⟩ (and halt).
  2. Otherwise, we have x = ⟨M, w⟩ where M is a TM and w is a string. Extract M
     and w from x.
  3. Let
        R = "On input x:
        (a) Run M on input w (and do whatever M does)."
  4. Output ⟨R⟩."
```

Note that R ignores its own input, so if M accepts w, then $L(R) = \Sigma^*$, and otherwise, $L(R) = \emptyset$.

We show that f m-reduces A_{TM} to $\overline{E_{\text{TM}}}$. Evidently, f is computable. Let x be any string. We need to show that $x \in A_{\text{TM}} \iff f(x) \in \overline{E_{\text{TM}}}$. First, suppose x is not of the form $\langle M, w \rangle$ for some TM M and string w. Then $x \notin A_{\text{TM}}$, and $f(x) = \langle M_0 \rangle \notin \overline{E_{\text{TM}}}$, so that's ok. Now suppose that $x = \langle M, w \rangle$ for some TM M and string w. Let $f(x) = \langle R \rangle$ for TM R as above. We have

$$\begin{aligned} \langle M, w \rangle \in A_{\text{TM}} &\iff M \text{ accepts } w \\ &\iff L(R) \neq \emptyset \\ &\iff \langle R \rangle \in \overline{E_{\text{TM}}}. \end{aligned}$$

So in all cases, $x \in A_{\text{TM}} \iff f(x) \in \overline{E_{\text{TM}}}$, so f m-reduces A_{TM} to $\overline{E_{\text{TM}}}$. □

Easy Fact 18.6. *If $A \leq_{\text{m}} B$, then $\overline{A} \leq_{\text{m}} \overline{B}$.*

Proof. If f m-reduces A to B, then *the same* f m-reduces $\overline{A}$ to $\overline{B}$. Why? For all x,

$$\begin{aligned} x \in A &\iff f(x) \in B \\ x \notin A &\iff f(x) \notin B \\ x \in \overline{A} &\iff f(x) \in \overline{B}. \end{aligned}$$

□

Applying this to the reduction we just did, we see that

$$\overline{A_{\text{TM}}} \leq_{\text{m}} \overline{\overline{E_{\text{TM}}}} = E_{\text{TM}},$$

and it follows that E_{TM} is not Turing recognizable.

Exercise 18.7. Show that $\overline{E_{\text{TM}}}$ is, in fact, Turing recognizable.

So far, the only non-T-recognizable languages we've seen are complements of T-recognizable languages. Now we'll see a language such that neither it nor its complement is Turing recognizable.

The equivalence problem for Turing machines is encoded by the language

$$EQ_{\text{TM}} = \{\langle M, N\rangle \mid M \text{ and } N \text{ are TMs and } L(M) = L(N)\}.$$

We show first that $E_{\text{TM}} \leq_{\text{m}} EQ_{\text{TM}}$. Since $\overline{A_{\text{TM}}} \leq_{\text{m}} E_{\text{TM}}$, we know that E_{TM} is not Turing recognizable, so m-reducing E_{TM} to EQ_{TM} will show that the latter is not T-recognizable either.

Recall the TM M_0 recognizing $\emptyset$ from above. Define

M_1 = "On input w: accept."

Clearly, $L(M_1) = \Sigma^*$.

For this m-reduction, we use the fact that a TM recognizes $\emptyset$ just in case that it is equivalent to M_0. Let

f = "On input x:

1. If x is not of the form $\langle M\rangle$ for any TM M, then output $\langle M_0, M_1\rangle$.
2. Otherwise, $x = \langle M\rangle$ where M is a TM. Output $\langle M, M_0\rangle$."

Clearly, f is computable. For any TM M we have

$$\begin{aligned} \langle M\rangle \in E_{\text{TM}} &\iff L(M) = \emptyset \\ &\iff L(M) = L(M_0) \\ &\iff \langle M, M_0\rangle \in EQ_{\text{TM}} \\ &\iff f(\langle M\rangle) \in EQ_{\text{TM}}. \end{aligned}$$

The exceptional case where the input to f does not encode a TM is handled in Step 1. Thus f m-reduces E_{TM} to EQ_{TM}.

Now we show that $\overline{A_{\text{TM}}} \leq_{\text{m}} \overline{EQ_{\text{TM}}}$, or equivalently, $A_{\text{TM}} \leq_{\text{m}} EQ_{\text{TM}}$. Like the previous reduction, this one does not need much effort.

Recall the computable function f defined in the proof of Proposition 18.5. For any input x, it was the case that f output the encoding of some TM R such that

$$\begin{aligned} x \in A_{\text{TM}} &\implies L(R) = \Sigma^*, \\ x \notin A_{\text{TM}} &\implies L(R) = \emptyset. \end{aligned}$$

Thus $x \in A_{\text{TM}}$ iff $L(R) = \Sigma^*$; in other words, $x \in A_{\text{TM}}$ iff R is equivalent to M_1. Now let

g = "On input x:

1. Compute $\langle R\rangle = f(x)$.
2. Output $\langle R, M_1\rangle$."

Clearly, g is computable and $x \in A_{\text{TM}}$ iff $g(x) \in EQ_{\text{TM}}$. Thus $A_{\text{TM}} \leq_{\text{m}} EQ_{\text{TM}}$ via g.

18.2 Linearly Bounded Automata

A linearly bounded atomaton (LBA) is a kind of one-tape Turing machine, where the tape is *finite* for each input, having just enough cells to contain the input string. Otherwise, the machine can read and write cells, change state, move the head, and enter halting states just as with a regular TM, except that now if the head tries to go off of *either* end of the tape, it stays put instead. Since the tape alphabet can be larger than the input alphabet (but still constant size), the machine can store a constant number of bits of information in each cell, and thus its memory is limited to growing linearly with the input size.

Exercise 18.8. With a regular TM, we saw that you can add an end marker to the beginning of the tape. Show how an LBA can do the functional equivalent of adding two end markers to each side of the input.

Let

$$A_{\text{LBA}} = \{\langle M, w\rangle \mid M \text{ is an LBA that accepts string } w\}.$$

Proposition 18.9. *A_{LBA} is decidable.*

Proof. Let $q = |Q|$ and $g = |\Gamma|$. There are exactly qng^n possible distinct configurations of M for a tape of length n. Run M on input w for qng^n steps (where $n = |w|$) or until it halts. If M has accepted, then accept, else reject. This works because if M does not halt within qng^n steps, then it must repeat a configuration and thus loop forever (not accepting). □

19 3/31/2008

Let

$$E_{\text{LBA}} = \{\langle M\rangle \mid M \text{ is an LBA and } L(M) = \emptyset\}.$$

Proposition 19.1. *E_{LBA} is undecidable. In fact, E_{LBA} is not T-recognizable.*

Proof. (Mapping reduction from $\overline{A_{\text{TM}}}$ using computational history method): let

f = "On input $\langle M, w\rangle$ where M is a TM and w is an input string:

1. Construct an LBA $B_{M,w}$ that accepts an input string x iff x is the complete history of an accepting computation of M on input w, i.e.,

$$x = \$C_1\#C_2\#\cdots\#C_k\$,$$

where the C_i are the successive confugurations of M on input w:

B = 'On input x:

(a) (B can find $C_1, \ldots, C_k$ when necessary, using the delimiters)
(b) Check that C_1 is the start configuration of M on input w, that is, $C_1 = q_0 w$.
(c) Check that each C_{i+1} legally follows from C_i in one step by the rules of M.
(d) Check that C_k is an accepting configuration, that is, $C_k = \cdots q_{accept} \cdots$.'

2. Output $\langle B_{M,w} \rangle$."

(Tape alphabet is $Q \cup \Gamma \cup \{\#, \$\}$, where Γ is the tape alphabet of M. Assume these sets are disjoint.) ☐

If you're still uncomfortable with algorithms that output other algorithms (in the form of descriptions of TMs), then it may help to know that all such constructions are just applications of the following general theorem, which is also called the s-m-n Theorem for historical reasons:

Theorem 19.2 (Parameter Theorem)**.** *There is a computable function $s : \Sigma^* \to \Sigma^*$ such that, for any TM M and string w, $s(\langle M, w \rangle) = \langle R \rangle$, where R is a TM which, on any input string x, behaves the same as M on input $\langle w, x \rangle$.*

[Intuitive explanation]
[Proof]
[Application to m-reduce ATM to ETM-bar]

20 4/2/2008

Midterm II (Chapters 3,4,5)

21 4/7/2008

Two exercises that will be useful later on:

Exercise 21.1. Show that a language A is Turing recognizable iff there is a decidable D such that, for all strings w,

$$w \in A \iff (\exists x)\ \langle w, x \rangle \in D.$$

A language that satisfies the part after the "iff" is called Σ_1. Thus the exercise is to show that being Turing recognizable is the same as being Σ_1. Intuition: If $w \in A$, then there is some string x that acts as a "proof" or "witness" that $w \in A$. The proof is verified algorithmically by the decider for D. If $w \notin A$, then there is no proof, and the verifier can never be fooled into accepting.

[Do the exercise.]

Exercise 21.2. Show that if A is Turing recognizable, then $A \leq_{\mathrm{m}} A_{\mathrm{TM}}$.

The converse of this is true (easy), so we have: A is Turing recognizable iff $A \leq_{\mathrm{m}} A_{\mathrm{TM}}$. So these exercises give us two more characterizations of Turing recognizability.

[Do the exercise. This is the easiest m-reduction you'll see.]

21.1 Time Complexity (Chapter 7)

We haven't cared about efficient use of resources so far. We should care, because there are decidable problems that cannot be decided efficiently. For such problems, finding solutions for large inputs is possible in principle, but not in practice, because it would just take too long. Whereas computability categorizes problems by decidability/undecidability, computational complexity theory categorizes decidable problems by feasibility/infeasibility. It gives a finer classification within the realm of decidable languages.

Two standard measures of resources: time and space (there are others). Time is the number of steps of a Turing machine before it halts, space is the maximum rightward extent of the head during the computation.

[Review big-O and big-Θ notation and definitions. We only care about time and space resources up to asymptotic equivalence (big Θ). Useful fact: If $p : \mathbb{N} \to \mathbb{N}$ is given by a polynomial of degree $d \geq 0$, then $p(n) = \Theta(n^d)$.]

Example: 1-tape TM deciding $\{0^n 1^n \mid n \geq 0\}$ in $\Theta((2n)^2) = \Theta(n^2)$ steps. Takes linear time on a 2-tape machine. Is there a faster 1-tape algorithm? Yes: $\Theta(n \lg n)$, and this is asymptotically optimal for a 1-tape machine.

Definition 21.3. Let $t : \mathbb{N} \to \mathbb{N}$ be a function and let M be a Turing machine that halts on all inputs. We say that M *has time complexity* t or that M *runs in time* t if for every $n \in \mathbb{N}$, $t(n)$ is the maximum number of steps that M takes to halt on any input string of length n.

TIME(t) is the class of all languages that are decidable in time $O(t)$, i.e., that have deciders running in time $O(t(n))$.

All measures of complexity are taken with respect to the length of the input, and we will only consider worst-case complexity.

We really only care about time complexity up to a polynomial. [Define $f = \mathcal{P}oly(g)$.] We say that a TM runs in *polynomial time* if its running time is bounded (from above) by some polynomial in the length of the input. In this case, the number of Turing machine tapes does not matter, since a k-tape machine running in time t can be simulated by a 1-tape machine running in time $O(t^2)$, provided $t \geq n$, where n is the length of the input.

Functions that are not polynomially bounded include exponentially growing functions such as $f(n) = 2^n$.

21.2 The class P

Definition 21.4. **P** is the class of all languages that are decidable in polynomial time, i.e.,

$$\mathbf{P} = \mathrm{TIME}(\mathcal{P}oly(n)) = \bigcup_{k>0} \mathrm{TIME}(n^k).$$

22 4/9/2008

We take **P** to capture the intuition of efficiently decidable languages. Examples of languages in **P**: PATH (e.g., BFS), REL-PRIME (e.g., Euclidean Algorithm), PRIMES (not obvious; only known since 2002), A_{DFA}, E_{DFA} and E_{NFA} (both applications of PATH), A_{NFA} (How? Constructing the equivalent DFA like we did before make take too long (exponential time), but there is another way: only construct the path taken by the DFA on the input string w).

There are languages that can be decided in exponential time, but not polynomial time (i.e., not in **P**). [Draw a Venn diagram of classes seen so far.]

22.1 The class NP

P is the class of languages where membership can be *decided* efficiently (polynomial time). NP (which stands for "nondeterministic polynomial time") is the class of languages where membership can be *verified* in polynomial time.

Definition 22.1. NP is the class of all languages A for which there is a polynomial p and Turing machine V (the *verifier*) such that, for all input string w,

$$w \in A \iff (\exists x)[V \text{ accepts input } \langle w, x\rangle],$$

and $V(\langle w, x\rangle)$ halts in at most $p(|w|)$ steps. If there exists such a string x, then we call x a *proof* or *witness* that $w \in A$.

The verifier V must run in time polynomial in $|w|$, which is stronger than merely requiring that V just run in polynomial time (in the length of its input). This prevents V from being able to spend a long long time verifying a long long proof. V only has time to view the first $p(|x|)$ many symbols of the proof string x at most, so if there is any proof x that $w \in A$, there must exist a proof that is reasonably short ($|x| = \mathcal{P}oly(|w|)$). This gives a slightly alternate definition of NP:

Proposition 22.2. *A language A is in* NP *if and only if there is a polynomial p and a language $D \in \mathbf{P}$ such that, for all strings w,*

$$w \in A \iff (\exists x)[\ |x| \le p(|w|)\ \&\ \langle w, x\rangle \in D.$$

Thus we can explicitly require the proof string to be reasonably short. Notice how similar this is to the exercise that shows that T-recognizability is the same as Σ_1. Here, we just have polynomial bounds of the length of the proof and the time taken to verify it. Indeed, NP is sometimes denoted Σ_1^p.

[Examples: CLIQUE, COMPOSITES]

[Discussion: How about verifying nonmembership? coNP]

[Discussion: Clearly, $\mathbf{P} \subseteq$ NP (the verifier ignores the proof and just runs the decider). Is it the case that $\mathbf{P} =$ NP? This is a celebrated and important open problem in mathematics and computer science.]

23 4/14/2008

23.1 Polynomial reductions and NP-completeness

NP contains loads of problems of wide interest to many fields: scheduling, routing, partitioning, constraint satisfaction, graph coloring, etc. We want to classify these problems with regard to feasibility. We use a polynomially bounded version of the mapping reduction.

Definition 23.1. [FP, Polynomial mapping reduction]

[Basics: Reflexive, transitive. If $A \le_{\mathrm{m}}^{\mathrm{p}} B$, then ...]

[NP-completeness.]

[An NP-complete set K. In the exercises, the book denotes this set U.]

24 4/16/2008

24.1 Boolean formulæ and satisfiability

A *Boolean formula* is an expression made from Boolean variables $x, y, z, \ldots$, binary infix Boolean connectives "$\wedge$" (AND), "$\vee$" (OR), and the unary prefix $\neg$ (NOT). Boolean variables can be assigned the values 0 (FALSE) or 1 (TRUE), and if some truth value (i.e., TRUE or FALSE) is assigned to each variable in a formula, then the truth value of the formula is determined in the usual way. For example, let φ be the formula

$$\varphi = (x \vee \overline{y}) \wedge (\overline{x} \vee z \vee y) \wedge \overline{z}.$$

[Give some sample values for x, y, z. When is φ true?] A *truth assignment* of φ is some setting of values for the variables in φ (and this will make φ either TRUE or FALSE). Computing the truth value of φ given a truth assignment of its variables is an easy bottom-up tree traversal, which takes no more than quadratic time (thus polynomial time).

A Boolean formula φ is *satisfiable* if there is some truth assignment to the variables occuring in φ that makes φ true. [Is the formula φ above satisfiable?] Such an assignment is called a *satisfying assignment.* A natural, interesting, and fundamental computational decision problem in logic is: given a Boolean formula, is it satisfiable? This is called the SATISFIABILITY problem, and as usual we can encode it as a language:

$$\mathrm{SAT} = \{\langle\varphi\rangle \mid \varphi \text{ is a satisfiable Boolean formula}\}.$$

Clearly, SAT $\in$ NP. Why? If φ is satisfiable, then an easily checkable proof of this would be an actual satisfying assignment, which you can check in polynomial time.

We will only worry about Boolean formulæ that are in *conjunctive normal form (CNF).* A formula φ is in CNF iff it is the *con*junction of *clauses*

$$\varphi = C_1 \wedge C_2 \wedge \cdots \wedge C_k,$$

where each *clause* C_i is a *dis*junction of *literals*

$$C_i = (\ell_{i,1} \vee \ell_{i,2} \vee \cdots \vee \ell_{i,j_i}),$$

where each *literal* is either a variable or the negation of a variable: $\ell = x$ or $\ell = \overline{x}$ for some variable x. (It is customary in computer science to write a negated variable x as $\overline{x}$ instead of $\neg x$.) The formula above is in CNF.

The language CNF-SAT is the restriction of SAT to CNF formulæ:

$$\text{CNF-SAT} = \{\langle\varphi\rangle \mid \varphi \text{ is a satisfiable CNF formula}\}.$$

Today we'll prove the Cook-Levin Theorem—arguably the most important theorem in complexity theory:

Theorem 24.1 (Cook, Levin)**.** *CNF-SAT is* NP*-complete.*

Proof. We already know that CNF-SAT $\in$ NP (just as SAT $\in$ NP). It remains to show that CNF-SAT is NP-hard. We'll do this directly. Let B be any language in NP. We show that $B \leq^{\mathrm{p}}_{\mathrm{m}}$ CNF-SAT.

Let V be some verifier for B such that $V(\langle w, x\rangle)$ runs in at most $p(|w|)$ steps (p is some polynomial), and $w \in B$ iff $(\exists x)[V$ accepts $\langle w, x\rangle]$. Given a string w we will to construct (in polynomial time) a Boolean formula φ_w in CNF such that $w \in B \iff \varphi_w$ is satisfiable. The map $w \mapsto \varphi_w$ will then be a ptime m-reduction from B to CNF-SAT, which finishes the proof.

Basically, the formula φ_w "says" that $V(\langle w, x\rangle)$ accepts, for some string x. The variables of φ_w will encode an entire computation of V on some input. The variables will make φ_w if and only if they encode an accepting computation of V on input $\langle w, x\rangle$ for some x.

Now the details. Let $V = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ (V is a standard 1-tape TM). Fix a string $w \in \Sigma^*$ and let $w = w_1 \cdots w_n$, where each $w_i \in \Sigma$ (so $n = |w|$. For technical convenience, we will make some assumptions:

- for any string x, $\langle w, x\rangle$ is always encoded as the string $\$w\#x$, where $\$, \# \in \Sigma$ are two special symbols,
- neither \$ nor # occurs in w or in x,
- V never overwrites \$ with another symbol,
- V never writes \$ anywhere else on the tape, and
- V never attempts to move left when scanning \$.

These assumptions do not place any undue hardship on V.

The entire computation of V on some input $\$w\#x$ can be described entirely with a two-dimensional grid (a *tableau*). The vertical axis represents the time step $t = 0, 1, 2, \ldots$, and the horizontal axis represents the tape contents (cell $0, 1, 2, \ldots$). Since V halts within $p = p(n)$ steps, we only need to make the tableau $(p + 1) \times (p + 1)$. So time runs from 0 to $p(n)$ inclusive, and the only tape cells that can ever be scanned in this time are cells 0 through $p(n)$.

Here are the variables of φ_w. They come in three types:

- $s_{q,t}$ for all $0 \le t \le p$ and all $q \in Q$ ($s_{q,t} = 1$ means "V is in state q at time t"),
- $x_{a,i,t}$ for all $0 \le i, t \le p$ and all $a \in \Gamma$ ("Cell i contains symbol a at time t"),
- $h_{i,t}$ for all $0 \le i, t \le p$ ("V's head is scanning cell i at time t").

φ_w contains clauses ensuring that:

1. V is in exactly one state at any time,
2. V's head is in exactly one place at any time,
3. each cell has exactly one symbol at any time,
4. the initial conditions are correct,
5. V enters q_{accept} at some point in time,
6. the configuration at time $t + 1$ is the legal successor to the configuration at time t, for all $0 \le t < p$.

We'll construct these clauses in order.

1. For each $0 \le t \le p$, add the clause
$$\bigvee_{q \in Q} s_{q,t}$$
("V is in at least one state at time t"), and for each $q, r \in Q$ with $q \neq r$, add the clause
$$\overline{s_{q,t}} \vee \overline{s_{r,t}}$$
("V is not simultaneously in state q and r at time t").

2. For each $0 \le t \le p$, add the clause
$$\bigvee_{i=0}^{p} h_{i,t}$$
("V's head is in at least one position at time t"), and for each $0 \le i < j \le p$ add the clause
$$\overline{h_{i,t}} \vee \overline{h_{j,t}}$$
("V's head is not simultaneously scanning cells i and j at time t").

3. For each $0 \le i, t \le p$, add the clause
$$\bigvee_{a \in \Gamma} x_{a,i,t}$$
("Cell i contains at least one symbol at time t"), and for each $a, b \in \Gamma$ with $a \neq b$, add the clause
$$\overline{x_{a,i,t}} \vee \overline{x_{b,i,t}}$$
("Cell i does not simultaneously contain symbols a and b at time t").

4. Add the following clauses:
   - $s_{q_0,0}$ ("V is in the start state at time 0"),
   - $h_{0,0}$ ("V's head is scanning cell 0 at time 0"),
   - $x_{\$,0,0}$ ("Cell 0 contains \$ at time 0"),
   - $x_{w_i,i,0}$ for each $1 \le i \le n$ ("Cells 1 through n contain w at time 0"), $x_{\#,n+1,0}$ ("Cell $n+1$ contains # at time 0"),
   - $\bigvee_{a \in (\Sigma - \{\$,\#\}) \cup \{\_\}} x_{a,i,0}$ for all $n+2 \le i \le p$ ("Cells $n+2$ through p either contain blanks or symbols in Σ except \$ or # at time 0),
   - $\overline{x_{\_,i,0}} \vee x_{\_,i+1,0}$ for all $n+2 \le i < p$ ("At time 0, starting at cell $n+2$, no blank cell can be followed by a nonblank cell immediately to its right").

5. Add the clause
$$\bigvee_{t=0}^{p} s_{q_{accept},t}$$
("V enters q_{accept} at some time t between 0 and p).

6. We add two types of clauses: (i) clauses ensuring that an unscanned symbol remains unaltered in the next time step, and (ii) clauses ensuring that the correct symbol is written, the head moves properly, and the state changes correctly, based on the current state and the scanned symbol—all according to the transition function of V.

   For (i), for each time $0 \leq t < p$, cell $0 \leq i \leq p$, and symbol $a \in \Gamma$, we add a clause that says,

   > "At time t, if the head is not scanning cell i and cell i contains a, then cell i contains a at time $t + 1$."

   This can be schematized as

   $$(\overline{h_{i,t}} \wedge x_{a,i,t}) \rightarrow x_{a,i,t+1}, \tag{1}$$

   where "$\rightarrow$" is the if-then conditional. A standard rule of propositional logic says that $P \rightarrow Q$ is equivalent to $\overline{P} \vee Q$ for any propositions P and Q. Thus (1 becomes

   $$\overline{\overline{h_{i,t}} \wedge x_{a,i,t}} \vee x_{a,i,t+1},$$

   which becomes, using DeMorgan's Laws,

   $$h_{i,t} \vee \overline{x_{a,i,t}} \vee x_{a,i,t+1}.$$

   This clause has the right syntactic form, so we add it to φ_w for each time $0 \leq t < p$, cell $0 \leq i \leq p$, and symbol $a \in \Gamma$.

   For (ii), we add the following clauses for each state $q \in Q$ and symbol $a \in \Gamma$: Suppose $\delta(q, a) = (r, b, d)$ for some $r \in Q$, $b \in \Gamma$, and $d \in \{L, R\}$. Then for each time $0 \leq t < p$ and cell $0 \leq i \leq p$ we add the two clauses

   $$\begin{aligned} h_{i,t} \wedge x_{a,i,t} \wedge s_{q,t} &\rightarrow s_{r,t+1} \\ h_{i,t} \wedge x_{a,i,t} \wedge s_{q,t} &\rightarrow x_{b,i,t+1} \end{aligned}$$

   ("If at time t the head is scanning cell i which contains symbol a, and V's state is q, then at time $t + 1$, the ith cell contains b and the new state is r"). Using transformations similar to before, we actually add the following equivalent clauses, which have the right syntax:

   $$\begin{aligned} &\overline{h_{i,t}} \vee \overline{x_{a,i,t}} \vee \overline{s_{q,t}} \vee s_{r,t+1}, \\ &\overline{h_{i,t}} \vee \overline{x_{a,i,t}} \vee \overline{s_{q,t}} \vee x_{b,i,t+1}. \end{aligned}$$

   It remains only to add clauses describing the head movement. There are two cases:

   - If $d = L$, then for all $0 \leq t < p$ and $0 < i \leq p$ add the clause

     $$\overline{h_{i,t}} \vee \overline{x_{a,i,t}} \vee \overline{s_{q,t}} \vee h_{i-1,t+1}$$

     ("If at time t the head is scanning cell $i > 0$ which contains symbol a, and V's state is q, then at time $t+1$ the head is in position $i - 1$"). Notice that we do not need to worry about $i = 0$ because we are assuming that V never tries to move left when scanning the \$ symbol, which is in cell 0.

- If $d = R$, then for all $0 \le t < p$ and $0 \le i < p$ add the clause

  $$\overline{h_{i,t}} \vee \overline{x_{a,i,t}} \vee \overline{s_{q,t}} \vee h_{i+1,t+1}$$

  ("If at time t the head is scanning cell $i < p$ which contains symbol a, and V's state is q, then at time $t+1$ the head is in position $i-1$").

By our construction of it, φ_w is evidently satisfiable if and only if there is a string x such that V accepts $\langle w, x\rangle = \$w\#x$. $\square$

25 4/21/2008

25.1 CLIQUE is NP-complete

Once we showed that A_{TM} was undecidable, showing other languages undecidable became easier because it only sufficed to reduce A_{TM} to them. Something similar is going on here with the Cook-Levin Theorem regarding CNF-SAT. This theorem makes our job of showing other problems NP-hard much easier, since it suffices to reduce CNF-SAT to them in polynomial time. This technique of showing problems NP-complete (by reducing known NP-complete problems to them) is widely applicable and has proved hundreds of computational problems from various fields of science, engineering, and mathematics to be NP-complete (or at least NP-hard), and thus intractable unless $\mathbf{P} = \mathrm{NP}$. We'll illustrate this technique here with a single example.

Theorem 25.1. *CLIQUE is* NP*-complete.*

Proof. First, CLIQUE $\in$ NP as we discussed earlier. Next, to establish that CLIQUE is NP-hard, we show that CNF-SAT $\le_{\mathrm{m}}^{\mathrm{p}}$ CLIQUE.

Recall that a literal is either a Boolean variable or the negation of a Boolean variable. We say that two literals are *contradictory* if one is a variable and the other is the negation of the same variable. Note that two literals are contradictory iff they have opposite truth values in any truth assignment.

To ptime m-reduce CNF-SAT to CLIQUE, we describe how to map (in ptime) any CNF Boolean formula φ to a string $\langle G, k\rangle$ where G is a graph and $k \in \mathbb{N}$ such that φ is satisfiable iff G has a clique of size k. (If the input string does not encode a CNF formula—something we can easily check in ptime by looking at its syntax—then we map it to some trival string in $\overline{\text{CLIQUE}}$.) Let φ be any CNF formula. Then

$$\varphi = C_1 \wedge \cdots \wedge C_m,$$

where each clause C_i is a disjunction of literals. We map φ to $\langle G, m\rangle$, where m is the number of clauses of φ and G is the following graph:

- The vertices of G are all the literal occurrences in φ. (If the same literal occurs several times in φ, then each occurrence counts as a separate vertex of G.)
- We put an edge between every pair of vertices of G except (i) vertices corresponding to literals occuring in the same clause and (ii) vertices corresponding to contradictory literals.

This completes the description of the reduction. It remains to show that this construction is ptime and that φ is satisfiable iff G has a clique of size m.

Given φ, we can clearly construct $\langle G, m\rangle$ in polynomial time; there is really nothing more to say about this.

Now suppose first that φ is satisfiable. Let $\vec{a}$ be a satisfying assignment of φ. For each of the m clauses $C_1, \ldots, C_m$, choose some literal made true by $\vec{a}$. These m literals form a clique, because they all occur in distinct clauses and no two can be constradictory (since they are all true). Thus G has a clique of size m.

Conversely, suppose that G has a clique $C \subseteq V(G)$ of size m. Notice first that no two literals in C can be from the same clause, since no two literals from the same clause are adjacent. So all the vertices of C must be from different clauses of φ, and since there are exactly m clauses, C much include exactly one literal from each clause. Now since no two literals of C are contracdictory (because they are adjacent), there is a truth assignment $\vec{a}$ that makes all literals in C true simultaneously. But then $\vec{a}$ makes at least one literal true in each clause of φ, and thus $\vec{a}$ makes φ true. This shows that φ is satisfiable. This completes the proof. $\square$

25.2 Space complexity (Chapter 8)

The *space* used by a TM M on some input is the maximum s such that M's head scans cell s (on at least one of its tapes) at some point in the computation. We can then define space complexity just as we defined time complexity. The space complexity of M is, as a function of $n \in \mathbb{N}$, the maximum space used by M on all inputs of length n. For $s : \mathbb{N} \to \mathbb{N}$, we define SPACE($s$) to be the class of all languages decided by TMs with space complexity $O(s)$. Assuming $s(n) \geq n$, the class SPACE(s) is the same whether we use single-tape or multitape TMs, because we can simulate a multitape TM with a single-tape TM that uses no additional space (although it uses a bigger tape alphabet).

Analogously with **P**, we define the class *polynomial space* as

$$\mathrm{PSPACE} = \bigcup_{k>0} \mathrm{SPACE}(n^k) = \mathrm{SPACE}(\mathcal{P}oly(n)).$$

PSPACE contains all those decision problems that are decidable using a "reasonably modest" amount of memory.

Space complexity has some characteristics similar to time complexity. However, space-bounded machines can apparently do more than similarly time-bounded machines can. The intuitive reason for this is that limited space can be reused over and over again, whereas limited time cannot.

Recall that a natural way to define the *exponential time* complexity class is

$$\mathrm{EXP} = \bigcup_{k>0} \mathrm{TIME}(2^{n^k}) = \mathrm{TIME}(2^{\mathcal{P}oly(n)}).$$

Here's a basic proposition relating time and space complexity classes:

Proposition 25.2. $\mathbf{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE} \subseteq \mathrm{EXP}$.

It is known that $\mathbf{P} \neq \mathrm{EXP}$, but none of the other containments above are known to be proper. It is widely believed, however, that they are all proper.

We've already seen that $\mathbf{P} \subseteq$ NP. To see that NP $\subseteq$ PSPACE, just notice that, given an NP-style verifier, a polynomial space-bounded machine can cycle through all possible proof strings and simulate the verifier on each one. If the verifier is seen to reject a proof, the pspace simulator erases the computation and tries the next proof. This takes a long time (exponential time), but only polynomial space because it only needs to store the verifier's computation for one proof at a time.

To see that PSPACE $\subseteq$ EXP, we use the same configuration-counting technique that we used to show that A_{LBA} is decidable. A TM using space s must always be in one of only $2^{O(s)}$ many possible configurations. (The constants hidden in the big-O depend on the size of the tape alphabet, the number of tapes, and the number of states.) If the machine takes longer than this for some computation, then it must repeat a configuration and hence run forever, contradicting our general assumption that it is a decider. Therefore, a machine using space $\mathcal{P}oly(n)$ (and halting on all inputs) must run in at most $2^{\mathcal{P}oly(n)}$ time.

25.3 Savitch's Theorem

[Probably not enough time for this.]

25.4 PSPACE-completeness

PSPACE-hardness and PSPACE-completeness for a language are defined entirely analogously to the case of NP. A language A is PSPACE-hard (under p-m-reductions) if $B \leq_{\text{m}}^{\text{p}} A$ for every $B \in$ PSPACE. If a PSPACE-hard language is itself in PSPACE, then we say that is is PSPACE-complete. PSPACE has complete languages, and we'll see one shortly. Since NP $\subseteq$ PSPACE, every PSPACE-hard language is also NP-hard. Since this containment is most likely proper, we'd expect PSPACE-complete languages to be "harder" than merely NP-complete languages.

We'll describe a language now that bears some resemblance to to CNF-SAT, but it is PSPACE-complete instead of NP complete. Given a Boolean formula φ (not necessarily in CNF), we can quantify any or all of its variables with either $\exists$ ("there exists ...") or $\forall$ ("for all ..."). The former is called *existential* quantification, and the latter is called *universal* quantification. Let φ be a Boolean formula and let v is some Boolean variable (which may or may not occur in φ). Let φ_0 be the formula resulting from φ by replacing all occurrences of v in φ with the constant 0 (FALSE). Similarly, let φ_1 be the formula resulting from φ by replacing all occurrences of v in φ with the constant 1 (TRUE). Thus v occurs neither in φ_0 nor in φ_1, although both of these may contain other Boolean variables. The existentially quantified Boolean formula

$$(\exists v)\varphi$$

is then equivalent to $\varphi_0 \vee \varphi_1$. It says, in effect, that "there exists a truth value of v such that φ." Likewise, the universally quantified Boolean formula

$$(\forall v)\varphi$$

is equivalent to $\varphi_0 \wedge \varphi_1$. It says that "for each truth value of v, it is the case that φ." The truth values of these two quantified formulæ depend in general on the other variables occurring in them. We can add several quantifiers in a row, quantifying over different variables. For example, let φ be the quantified formula

$$(\forall x)(\exists y)(\forall z)[(\overline{x} \vee z) \wedge (x \vee \overline{y} \vee \overline{z}) \wedge (y \vee z)]$$

has all its variables quantified over, so its truth value does not depend on anything, i.e., it is either true or false outright (which?). A quantified Boolean formula with all its variables quantified is called a *closed formula*. Closed formulas are either true or false *per se*.

Here, then, is our first PSPACE-complete language:

Definition 25.3. The language TQBF is defined as

$$\mathrm{TQBF} = \{\varphi \mid \varphi \text{ is a closed formula that is true}\}.$$

(TQBF stands for True Quantified Boolean Formulæ.)

Theorem 25.4. TQBF *is* PSPACE-*complete.*

We'll accept this as given for now.

25.5 Games

A closed quantified Boolean formula can be thought of in terms of a game between two players, Alice and Bob. The quantifiers are read from left to right. If a variable is existentially quantified (e.g., "$(\exists x)$"), then Alice gets to choose a truth value for that variable. If the variable is universally quantified (e.g., "$(\forall y)$"), then Bob gets to choose a truth value for that variable. After the players have chosen truth values for all the variables, the resulting truth assignment makes the remaining unquantified portion of the formula either true or false. If true, Alice wins; if false, Bob wins. Since the game is finite, either Alice or Bob (not both) has a winning strategy. The original quantified formula is true if and only if Alice has a winning strategy.

Because of this game-like aspect to TQBF, many PSPACE-complete decision problems are about who has a winning strategy in some game.

[Describe Generalized Geography (GG). Instances are $\langle G, s\rangle$, where G is a digraph and $s \in V(G)$ is the starting vertex.]

26 4/23/2008

Proposition 26.1. GG *is* PSPACE-*complete.*

Proof. We need to show that GG $\in$ PSPACE and that GG is PSPACE-hard. For the former, we give an algorithm deciding GG and verify that the algorithm can be implemented using a polynomial amount of space. Probably the easiest way to do this is via a recursive algorithm.

If G is a directed graph and $s \in V(G)$ is a vertex of G, then let $N(s)$ be the set of all vertices t of G adjacent to s via an edge $s \to t$ from s to t. Also, let $G - s$ be the graph obtained from G by removing the vertex s and all edges to and from s. Note that, starting with the game configuration $\langle G, s\rangle$, the next player moves by choosing some $t \in N(s)$ (if one exists), and the game configuration then becomes $\langle G - s, t\rangle$. (s cannot be used again, so we might as well remove it.)

Here's the algorithm deciding GG:

> "On input $\langle G, s\rangle$ where G is a digraph and $s \in V(G)$:
>
> 1. If NextPlayerWins(G, s) then accept; else reject. (That is, accept iff Alice (who is the next player) can win the game $\langle G, s\rangle$.)"

NextPlayerWins is a Boolean-valued function that returns TRUE for a game $\langle G, s \rangle$ iff the next player to move in the game (which could be either Alice or Bob) can win it, i.e., has a winning strategy. We define it recursively:

```
Function NextPlayerWins($G, s$) : Boolean

  1. For every $t \in N(s)$ do:
     (a) If NOT NextPlayerWins($G - s, t$) then return TRUE
         (We've found a move for the next player from which his or her opponent cannot
         guarantee a win (i.e., can be forced to lose); this is the first move in a winning
         strategy for the next player.)
  2. If we get here, then return FALSE
     (Every one of the next player's possible moves leads to a game winnable by his or
     her opponent, so the next player has no winning strategy.)
```

The algorithm is intuitively correct. How much space does it use? If we implement it on a multitape TM, we can use one of the tapes to act as a control stack for the recursion. There are a few other auxillary variables to keep track of, but the space is dominated by the size of the control stack. Suppose the input is $\langle G, s \rangle$ and $n = |\langle G, s \rangle|$. Each stack frame only needs to contain some string $\langle G', s' \rangle$, where G' is some subgraph of G and $s' \in V(G')$. Thus the size of each stack frame is at most n. Since each recursive call is made on a graph with one fewer vertices, the maximum depth of the recursion (the maximum number of frames on the stack at any one time) is bounded by $|V(G)| + 1$, which is certainly at most $n + 1$. Thus the total space used is the max size of a stack frame times the max number of stack frames, and this is $O(n^2) = \mathcal{P}oly(n)$.

Thus the algorithm uses polynomial space, which puts GG in PSPACE. Another way to see it is to notice that the algorithm just traverses the game tree generated by $\langle G, s \rangle$, and it only needs to remember one (polynomial length) path at a time.

[This next part is currently a sketch.] PSPACE-hardness of GG is shown by a p-m-reduction from TQBF. Given some quantified Boolean formula as input, we can assume that the formula is of the form

$$(\exists x_1)(\forall x_2)(\exists x_3) \cdots (\exists x_n)\varphi$$

(alternating quantifiers, beginning and ending with $\exists$, so n is odd), where φ is in CNF, i.e., φ is of the form

$$C_1 \wedge C_2 \wedge \cdots \wedge C_k,$$

where each clause C_i is a disjunction of literals. We can do this by inserting new quantifications into the string of quantifications. The new quantifications use fresh variables that don't occur anywhere else and hence don't affect the truth value of φ. We do this purely for technical reasons so that in the TQBF game, Alice and Bob alternate picking values for one Boolean variable apiece, and Alice picks the first and last values. We can then translate this game directly into an instance of GG.

[Draw the output graph G with start vertex s. After the variables' truth values have been chosen, Bob picks an edge corresponding to one of the clauses C_i, and then Alice responds by choosing an edge corresponding to some literal in C_i. She can do this (and then Bob can't move so he loses) iff the literal is true.] □

Theorem 26.2. TQBF *is* PSPACE-*complete.*

Proof. TQBF is in PSPACE via either a recursive algorithm (as with GG) or one that explicitly traverses the (game) tree of all possible truth assignments to the variables, at each leaf checking whether the unquantified portion is true for that assignment.

Since this is our "first" PSPACE-hardness proof, we need to give a p-m-reduction to TQBF from an arbitrary language $A \in$ PSPACE. Some of the details of this proof are similar to that of the Cook-Levin Theorem above, so we will only sketch those parts when they arise, concentrating on the features unique to the current proof.

We are given a language $A \in$ PSPACE decided by a TM M that uses at most $p(n)$ space for inputs of length n, where p is some polynomial. By picking a larger polynomial p if necessary, we can also assume that

- M halts in at most $2^{p(n)}$ steps on any input of length n, and
- all configurations reachable by M on inputs of length n can be described using exactly $p(n)$ bits.

We show that $A \leq_{\mathrm{m}}^{\mathrm{p}}$ TQBF.

Fix a string w, let $n = |w|$, and let $p = p(n)$. Then M halts in at most p steps on input w. To show the reduction, we construct (in polynomial time) a quantified Boolean formula φ_w that is true iff M accepts w. For simplicity, we'll make one more harmless assumption about M: We assume that just before M accepts, it erases the entire contents of its tape and leaves the head scanning the leftmost cell. This implies that there is one unique accepting configuration $c_{acc} = q_{accept}\text{\textvisiblespace}\text{\textvisiblespace}\cdots$ for M.

For any nonnegative integer t we use the two-place predicate $\mathrm{Reach}_t(c_1, c_2)$ to mean,

> "c_1 and c_2 are configurations of M (represented by binary strings of length p each), and starting in configuration c_1, M reaches configuration c_2 after at most 2^t steps."

We can assume that c_1 and c_2 are both binary strings of length p.

Notice that M accepts w just in case $\mathrm{Reach}_p(q_0 w, c_{acc})$ is true, where $q_0 w$ is the initial configuration of M on input w. That is, M accepts w iff M can reach c_{acc} in at most 2^p steps starting from $q_0 w$.

We can now translate $\mathrm{Reach}_t(c_1, c_2)$ into a quantified Boolean formula recursively. The formula will not be in quantified CNF to begin with, but we can fix that later.

Base Case: If $t = 0$, then $\mathrm{Reach}_0(c_1, c_2)$ is true just in case M can go from c_1 to c_2 within $2^0 = 1$ step. This in turn is true iff either $c_1 = c_2$ (M takes zero steps) or c_2 is a legal successor to c_1 (M takes one step). To express "$c_1 = c_2$," we need a conjuction of clauses that come in p pairs: $(c_{1,i} \vee \overline{c_{2,i}}) \wedge (\overline{c_{1,i}} \vee c_{2,i})$, where $c_{1,i}$ and $c_{2,i}$ are the ith bits of c_1 and c_2, respectively, for all $1 \leq i \leq p$. The resulting formula is in CNF and is true iff the ith bits are equal for all i.

As in the proof of the Cook-Levin Theorem, we can express the condition that c_2 is a successor to c_1 as an unquantified Boolean CNF formula. (We omit the details.)

So we translate $\mathrm{Reach}_0(c_1, c_2)$ into the disjunction of these two formulæ. This is an unquantified Boolean formula over $2p$ variables: the first p encoding c_1 and the rest encoding c_2. To get it into CNF, we can use the fact that $\vee$ distributes over $\wedge$ and take the conjunction of all possible disjunctions of a clause from one formula with a clause from the other. This may increase the size of the formula quadratically, but it is still polynomial size.

Recursive Case: Suppose $t \geq 1$. Clearly, M goes from c_1 to c_2 in at most 2^t steps iff there is a configuration c such that M goes from c_1 to c in at most 2^{t-1} steps and from c to c_2 in at most 2^{t-1} steps. So,

$$\mathrm{Reach}_t(c_1, c_2) \iff (\exists c)[\,\mathrm{Reach}_{t-1}(c_1, c) \wedge \mathrm{Reach}_{t-1}(c, c_2)\,]. \tag{2}$$

Here, $(\exists c)$ is shorthand for a string of p many existential quantifications over Boolean variables: $(\exists c_1)(\exists c_2)\cdots(\exists c_p)$ where the c_i are single bits (Boolean variables) and $c = c_1c_2\cdots c_p$.

Starting with $\mathrm{Reach}_p(q_0w, c_{acc})$ and repeatedly substituting the right-hand side of (2) for the left-hand side recursively, we wind up with a quantified Boolean formula. Unfortunately it is too large. When we do the substitution given in (2), we decrease t by 1 but roughly double the size of the formula. So in the end, we get a formula of size roughly 2^p, which is exponentially large and thus cannot be generated in polynomial time.

To prevent exponential blow-up in the formula size, we really need to have only one recursive "call" to $\mathrm{Reach}_{t-1}(\cdot,\cdot)$ on the right-hand side of (2) instead of two. We can accomplish this by replacing the conjunction with universal quantification:

$$\mathrm{Reach}_t(c_1, c_2) \iff (\exists c)(\forall c_3)(\forall c_4)[\,(c_3, c_4) \in \{(c_1, c), (c, c_2)\} \rightarrow \mathrm{Reach}_{t-1}(c_3, c_4)\,]. \tag{3}$$

The term "$(c_3, c_4) \in \{(c_1, c), (c, c_2)\}$" expands to $(c_3 = c_1 \wedge c_4 = c) \vee (c_3 = c \wedge c_4 = c_2)$, and we use the standard equivalence $P \rightarrow Q \iff \neg P \vee Q$ to remove the "$\rightarrow$." It almost goes without saying that c_3 and c_4 are binary strings of length p.

Applying (3) recursively p times starting with $\mathrm{Reach}_p(q_0w, c_{acc})$ yields a polynomial-size quantified Boolean sentence (i.e., a closed formula), and it is easy to see that this expansion can be done in time polynomial in n. (q_0w and c_{acc} are strings not of variables but rather of Boolean constants.) It remains to get the sentence into the form we want, namely, a string of quantifiers followed by a CNF formula. By standard rules of logic, all the quantifiers can simply be moved to the left without changing the truth value of the sentence. The resulting sentence is said to be in *prenex form.* Getting a CNF formula after the quantifiers is an exercise for the reader. □