Theory of Computing
What is Feasible, Infeasible, and Impossible, Computationally

Richard Cole

November 22, 2023

Chapter 1

Mathematical Background

Broadly speaking, this text concerns what can and cannot be computed, and when something can be computed how simply and efficiently it can be done. As you will see later in this chapter, there are precisely defined problems which cannot be solved computationally, regardless of how fast or large a computer is used or how much time is spent on the computation. Likewise, as you will see in a later chapter, there are problems that are effectively infeasible although solvable in principle, for the resources needed would be prohibitive.

To formulate these issues precisely entails appropriate mathematical specifications. Accordingly, this chapter begins by reviewing notation and terminology that will be used throughout the book. This is followed by a quick overview of several proof techniques.

1.1 Definitions and Terminology

1.1.1 Sets

A set is a collection of items. To specify a set, the items are written as a list enclosed by braces, e.g. $A = \{1, 2, 4\}$. This means that A is the set containing the items 1, 2, and 4. There is no notion of order in the set listing, so $\{1, 2, 4\} = \{4, 1, 2\} = \{2, 4, 1\}$, etc. Also, there is no notion of duplicate items, so $\{1, 2, 1\} = \{1, 2\}$ for example.

The *empty* set is the set containing no items; it is written as $\{\ \}$ or $\emptyset$ for short.

Notation

- $x \in A$ means that x is one of the items contained in set A; x is called an *element* of A.
- $A \cap B$, the *intersection* of A and B, denotes the set containing those items that are in both A and B.
- $A \cup B$, the *union* of A and B, denotes the set containing those items that are in at least one of A or B.
- $A \subseteq B$ means that every item in A is also in B; A is called a *subset* of B. When, in addition, there is an item in B which is not in A, we write $A \subset B$, or for emphasis $A \subsetneq B$. A is then called a *strict subset* of B.

- $\overline{A}$ denotes the set containing those items not in A. For this definition to be meaningful we need a notion of the universe of items at hand, the set U of items, where $A \subseteq U$. Then $\overline{A}$ is the set of items in U but not in A.
- $|A|$ denotes the number of items in A.
- $A - B$ is the set consisting of items contained in A that are not contained in B. Note that $A - B = A \cap \overline{B}$.

Question 1. *What is the size of the empty set, $|\emptyset|$?*

Venn Diagrams. These are diagrams that are used to illustrate the intersections of collections of sets. They can be useful in helping us see what is going on. Typically, the *Universal Set*, U, if present, is shown as a large rectangle. Other sets appear as circular shapes inside this rectangle.

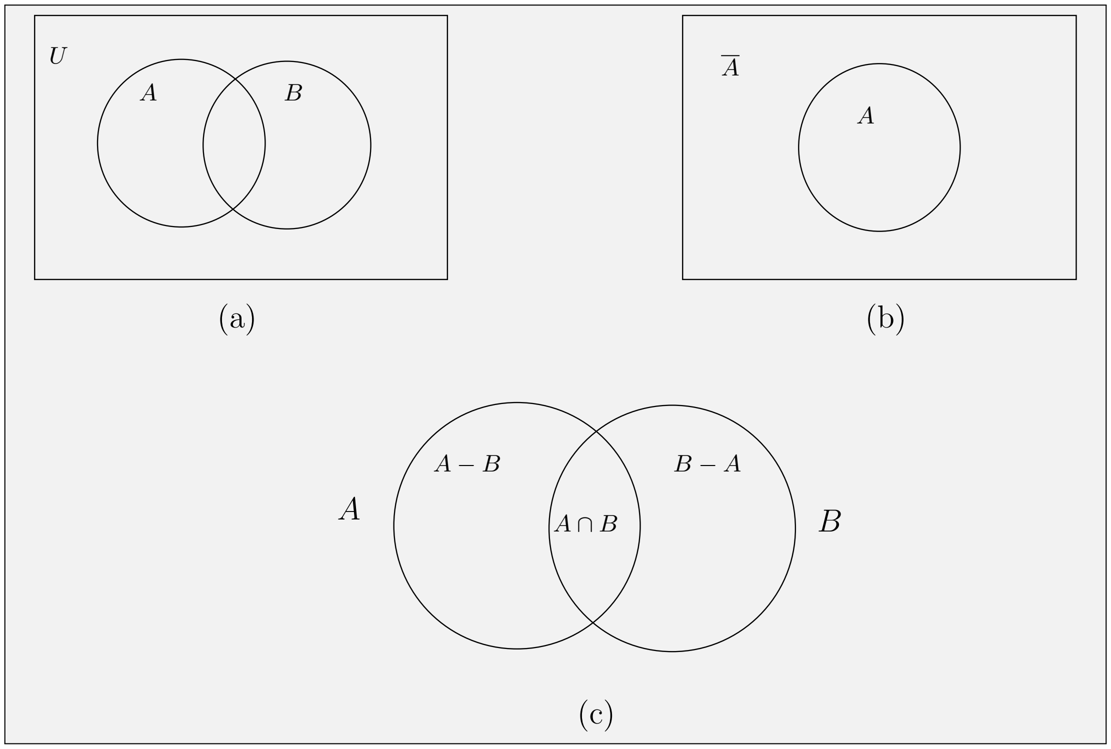


Figure 1.1: Venn Diagrams. (a) Sets A and B in a universal set U. (b) Sets A and $\overline{A}$. (c) Sets A and B (represented by circles) and subsets $A - B$, $A \cap B$, $B - A$ (represented by the regions they label).

Power Set. The power set of A is the collection (or set) of all subsets of A; it is written as 2^A. e.g. $A = \{a, b\}$. $\emptyset \subseteq A$, $a \subseteq A$, $b \subseteq A$, $\{a, b\} \subseteq A$. So $2^A = \textsc{PowerSet}(A) = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$.

Question 2. *Show that if $|A| = n$, then $|\textsc{PowerSet}(A)| = 2^n$. Note that this provides some justification for the notation 2^A.*

1.1.2 Sequences

A sequence is an ordered list of items in which there may be repetitions. For example, here are two sequences of three items each: $(10, 4, 13)$, and $(10, 9, 10)$; the second sequence has a repetition. Conventionally, the items are regarded as being ordered in the left to right order in which they are written. A k-item sequence is often called a *k-tuple*.

Let A and B be sets. $A \times B$ denotes the set of all possible 2-tuples whose first element is in A and whose second element is in B. So $|A \times B| = |A| \times |B|$. $A \times B$ is called the *cross-product* or *Cartesian product* of A and B. We can write

$$A \times B = \{(a, b) \mid a \in A, b \in B\}.$$

This is a first example of a notation for specifying sets. The expression to the left of the bar denotes a typical element of the set (in this case, a 2-tuple); the statement to the right of the bar explains what constraints such elements must obey (in this case that the first item in the tuple be an element of A, and the second an element of B). It is understood that all the elements obeying the constraints (or *condition* or *property*) are in the set being defined. So 2-tuple (e, f) will be an element of $A \times B$ exactly if $e \in A$ and $f \in B$.
e.g. $A = \{1, 3, 5\}, B = \{0, 1\}$. $A \times B = \{(1, 0), (3, 0), (5, 0), (1, 1), (3, 1), (5, 1)\}$.

1.1.3 Strings

A string is simply a sequence of characters, e.g. 01001001, CTAGCTTAG, the␣dog␣chased␣the␣cat. In the third example, note that the blank (␣) is a distinct character. Sometimes, to avoid ambiguity, we will write a character of string inside double quotes, e.g. "a", "cat".

Strings are everywhere. Text in documents provides an obvious example of strings. DNA sequences are another example. Programs are just (long) strings. Likewise, all inputs to programs are themselves strings. To be able to do anything useful with any of these various classes of strings one has to recognize and understand the useful units from which they are formed. Useful units could be individual words in text, subunits coding a protein in DNA, keywords in a program, mathematical operators (such as $+$), the digits forming a number, etc.

The study of strings forms a major part of this text: what collections of strings can be recognized, how easily this can be done for different sorts of collections, and what limits we face (for there are collections of strings that cannot be recognized as efficiently as we might wish, and other collections that cannot be recognized at all).

Alphabet This is a set of one or more characters, e.g., $\{0\}$, $\{0, 1\}$, $\{a, b, c, \cdots, z, \sqcup\}$, $\{A, C, G, T\}$, etc. Note the use of the $\cdots$ notation to indicate completion of the sequence in the natural way (in this case by including every letter of the English alphabet in the usual order). This notation is often used to specify infinite sets, e.g. $\{1, 2, 3, \cdots\}$, which denotes the set of all natural numbers, the integers greater than or equal to 1.

Strings A string is a sequence of zero or more characters. The zero character string, written λ, is a legitimate string. It has a role with strings analogous to the role 0 has with numbers as we will see in a bit. Incidentally, many authors use ϵ to denote the empty string. I prefer to avoid this notation as it is rather similar to the symbol for set membership.

Notation Conventions It is standard to use lower case letters from the end of the alphabet to name strings, typically u, v, w, x, y, z. Lower case letters from the beginning of the alphabet are used for individual characters. Both can be indexed: e.g. a_1, b_2, u_3, etc. Sometimes, though, u_i will indicate the ith character of string u (read from left to right). Typically, capital letters are used to name sets of strings.

More notation Σ usually indicates an alphabet.

Language L This is a set of strings. e.g. $L = \{aa, ab, ba, bb\}$, is the set of all 2-character strings over the alphabet $\{a, b\}$. (The term "over" indicates the alphabet from which the characters are drawn.) Σ^* denotes the set of all strings over the alphabet Σ; e.g. $\{a, b\}^* = \{\lambda, a, b, aa, ab, ba, bb, aaa, \cdots\}$.

Operations on Strings

Concatenation The concatenation of string u and string v is simply the string obtained by writing the characters of u followed by the characters of v. It is written as $u \circ v$ or simply uv.
Examples: $u = 01$, $v = 23$; then $uv = 0123$. $u = a_1a_2 \cdots a_i$, $v = b_1b_2 \cdots b_j$; then $uv = a_1a_2 \cdots a_ib_1b_2 \cdots b_j$.

Note that the concatenation of string u and the empty string yields string u: $u \circ \lambda = u$; e.g. $abc \circ \lambda = abc$. Likewise $\lambda \circ u = u$. This is analogous to the addition of zero to a number: $n + 0 = n$.

We also define the concatenation of two languages K and L as follows:

$$K \circ L = \{u \circ v \,|\, u \in K, v \in L\}.$$

Sometimes, we write KL for brevity. In words, the concatenation of K with L consists of all strings obtained by concatenating a string from K with a string from L. If K and L are finite, the concatenated language will have $|K| \times |L|$ strings. e..g $\{a, aa\} \circ \{b, c\} = \{ab, ac, aab, aac\}$.

Reversal The reversal of string u, denoted u^R, is the string obtained by writing the characters of u in reverse order. e.g. $u =$ stop, $u^R =$ pots; $u = a_1a_2 \cdots a_i$, $u^R = a_ia_{i-1} \cdots a_1$. Formally, $\lambda^R = \lambda$, and for v with $|v| \geq 1$, i.e. $v = au$ for some $a \in \Sigma$ and $u \in \Sigma^*$, $v^R = u^Ra$.

Substring String v is a substring of string z if z can be written as $z = uvw$ where u and w are also strings. Note that one or both of u and w could be the empty string. So z is a (trivial) substring of z, as is the empty string λ.
Examples: Let $z =$ "cat". "ca", "at" are both substrings of z, but "ct" is not.

Length of u This is the number of characters in u, and is denoted by $|u|$. So if $u = a_1a_2 \cdots\ a_i$, $|u| = i$.

The star operation L^* is the language obtained by concatenating 0 or more strings from L with the empty string; the strings chosen from L need not be distinct. Formally, $L* = \{w_1w_2 \ldots w_k \,|\, w_i \in L$, for $1 \leq i \leq k, k \geq 0\}$. e.g. $\{a\}^* = \{\lambda, a, aa, aaa, \ldots\}$; $\{a, bc\}^* = \{\lambda, a, bc, abc, bca, aabc, abca, bcaa, abcbc, bcabc, bcbca, \ldots\}$.

1.1.4 Graphs

A graph consists of two sets, a set of vertices and a set of edges. We draw a graph using little circles to denote vertices, and line segments joining pairs of vertices to denote edges. Vertices are named by little letters, typically u, v, w, or sometimes indexed, as in $v_1, v_2, \cdots$. Edges are specified by naming the pair of vertices they join.

In the example in Figure 1.2, the vertex set V is given by $V = \{u, v, w, x, y\}$; the edge set E is given by $E = \{(u, v), (v, x), (v, w), (w, u)\}$. $G = (V, E)$ denotes the graph.

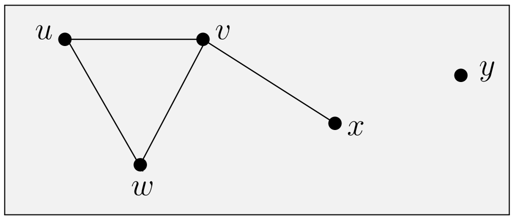


Figure 1.2: An Undirected Graph

There are two types of graphs, *undirected* and *directed*. In an undirected graph, the edges are unordered. In a directed graph, the edges are ordered, and here an edge $e = (u, v)$ indicates an edge from vertex u, called the *tail* of e, to vertex v, called the *head* of e. Edges are sometimes named using letters e, f, g, or using indices, as in $e_1, e_2, \cdots$.

Path A path in a graph in a sequence of vertices $v_1, v_2, \cdots, v_k$ where successive vertices are joined by edges, i.e. $(v_1, v_2), (v_2, v_3), \cdots, (v_{k-1}, v_k)$ are all edges in the graph, whether directed or undirected. Sometimes we specify this path as the edge sequence $e_1, e_2, \cdots, e_{k-1}$, where $e_i = (v_i, v_{i+1})$, for $1 \leq i < k$.

Cycle This is a path that begins and ends at the same vertex.

Simple path A path with no repeated vertex.

Simple cycle A cycle with exactly one repeated vertex (its first and last vertex).

Connected component This applies to undirected graphs only. It is a maximal collection of vertices such that for each pair of vertices in the component there is a path between them.

Strongly connected component This applies to directed graphs only. It is a maximal collection of vertices such that for each ordered pair of vertices (u, v) in the component, there is a path starting at u and ending at v.

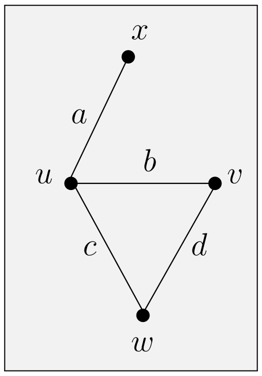


Figure 1.3: A Path: Path *wuvwux* has label *cbdca*.

In this text, by and large, the graphs we will be looking at are directed. Also, the edges will have labels, which will be either single characters, strings, or even sets of strings.

Path label let $P = (e_1, e_2, \cdots, e_r)$ be a path with edge e_i labeled by string s_i, for $1 \leq i \leq r$. Then P is said to have label $s_1 s_2 \cdots s_r$, the concatenation of strings $s_1, s_2, \cdots, s_r$.

1.1.5 Trees

Perhaps the easiest way to define a tree is recursively.

- A tree can be empty.
- A tree can consist of a single node called the *root*.
- A tree T can be built from two non-empty disjoint trees R and S as follows. Let r be the root of tree R. r will be the root of T also. T comprises a root r, a non-empty tree R rooted at r, together with another non-empty tree S which is a subtree of r; S is disjoint from R. In Figure 1.4, R would comprise root a and subtrees A, B, while S would be the subtree C.

Note the this construction allows r to have multiple subtrees (obtained by adding them one by one).

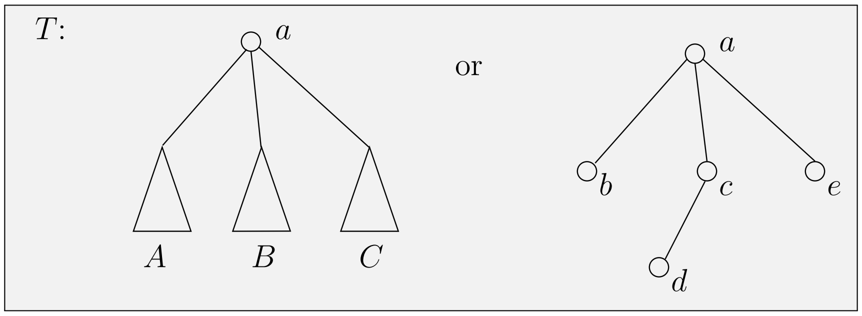


Figure 1.4: A Tree T. a is the root of T and it has 3 subtrees, named A, B, and C. B is formed from the nodes c and d, with c being the root of B.

Terminology

- The *children* of node u are the roots of u's subtrees. u is the *parent* of its children.
- A *leaf* is a node with no children.
- v is a *proper descendant* of u if v is a node in one of u's subtrees; likewise, u is a *proper ancestor* of v.
- v is a *descendant* of u if v is u or if v is a proper descendant of u; likewise, u is an *ancestor* of v.
- v and w are *siblings* if they have the same parent.

1.1.6 Logic

Logic is the algebra of variables that take on just one of two values: True and False (T and F for short).

Operations

- $x \wedge y$ (called x and y). This is True exactly if both x and y are True; otherwise it is False.
- $x \vee y$ (called x or y). This is True if at least one of x or y is True; otherwise (if both x and y are False) it is False.
- $\overline{x}$ (called not x, or sometimes "x bar"); it is also written as $\neg x$. This is True exactly if x is False; otherwise it is False.

Often, Boolean variables are defined with respect to a universe of objects. For example, the universe might be the set of integers and x the property of being prime; then $x(n)$ equals True exactly if n is a prime number.

There are two common and useful ways of expressing Boolean properties with respect to universes.

- $\forall x P(x)$. This is read as "for all x, $P(x)$ is true" and is itself TRUE exactly if, for every x in the relevant universe, $P(x)$ is TRUE.
- $\exists x P(x)$. This is read as "there exists an x for which $P(x)$ is true" and is itself TRUE exactly if, for some x in the relevant universe, $P(x)$ is TRUE.

Most assertions we wish to prove involve an implicit quantification. For example the assertion "The merge sort algorithm rearranges its input in sorted order" really means "For all inputs I, the merge sort algorithm on input I rearranges the input to put it in sorted order."

Observe that $\neg(\forall x P(x)) = \exists x(\neg P(x))$ and $\neg(\exists x P(x)) = \forall x(\neg P(x))$.

There are two more useful Boolean operators, implies ($\Rightarrow$) and if and only if (iff or $\Leftrightarrow$).

- $x \Rightarrow y$. This is read as "x implies y" and means that if x is TRUE then so is y. This is equivalent to $\overline{x} \vee y$. For if $x \Rightarrow y$ is TRUE there are two possibilities:

  (i) x is TRUE and hence so is y; then $\overline{x} \vee y$ is TRUE.
  (ii) x is FALSE; then there is no constraint on y. $x \Rightarrow y$ is still TRUE and so is $\overline{x} \vee y$.

  We now see that $x \Rightarrow y$ is FALSE only if x is TRUE and y is FALSE; but then $\overline{x} \vee y$ is also FALSE. Thus these two expressions ($x \Rightarrow y$ and $\overline{x} \vee y$) have the same truth value; they are then said to be *equivalent.*
- $x \Leftrightarrow y$. This is read as "x if and only if y." It means that x implies y and y implies x.

1.2 Proofs

This text is concerned with the demonstration, or *proof*, of mathematical claims, called theorems. The way this is done is to put together unassailable arguments, that is a flow of reasoning using clearly specified rules of deduction. At its most formal, this requires the specifications of axioms, which are the basic facts that are taken as given (e.g. the postulates seen in high school geometry). Then each step in the argument proceeds by appeal to a standard fact (e.g. the angles of a triangle sum to 180°), a previously proven result, and a given collection of rules of deduction. We will not be this formal, but in principle any argument we give could be fleshed out in (much) greater detail so as to be of this form.

There are several styles of proof you will be encountering repeatedly. We begin by reviewing their use, by showing a few examples.

1.2.1 Proof by Construction

A proof by construction is a direct demonstration that a claimed object exists (a construction). For example:

Theorem 1.2.1. *Suppose n is the product of three distinct primes, $n = pqr$ say, where $p \neq q \neq r \neq p$, and $p, q, r > 1$. Then n has (at least) 6 distinct factors.*

Proof. The factors are p, q, r, pq, pr, and qr and as p, q, r are all distinct primes, these 6 numbers are all distinct.

In fact, these are the only non-trivial factors of n (ruling out 1 and n as factors), for each possible factor must be a product using some but not all of p, q, and r, and there are 6 such choices. □

1.2.2 Proof by Contradiction

Suppose we want to prove an assertion A. In a proof by contradiction, we begin by supposing ("assuming") that in fact A is false. By a sequence of deductions we then manage to reach a contradiction, formally that FALSE = TRUE; an example would be that $0 = 1$. Since this is not possible (in any reasonable mathematical system), the only possibility is that the initial supposition was incorrect. That is, the assertion that A is false was incorrect, which means that A is actually true, and this constitutes a proof of that fact.

Theorem 1.2.2. *$\sqrt{3}$ is irrational.*

Proof. Recall that a number r is rational if it can be written as the ratio of two integers, $r = n/m$ say. As you can cancel any common factors in m and n, you can always assume that m and n have no common factors; such m and n are said to be *coprime*.

Now to the proof. Begin by assuming, for a contradiction, that $\sqrt{3}$ is in fact rational. This means that $\sqrt{3} = a/b$ for some coprime integers a and b. Squaring both sides gives $3 = a^2/b^2$ or $3b^2 = a^2$. But as 3 divides a^2, 3 must divide a (strictly, 3, being prime, must divide one of the factors of a^2, namely one of a or a). In other words, $a = 3c$, where c is also an integer. Substituting for a (in $3b^2 = a^2$) yields $3b^2 = 9c^2$; simplifying, this yields $b^2 = 3c^2$. By the same argument as before 3 must divide b. But then a and b are not coprime, for they are both divisible by 3. This contradicts the assumption that $\sqrt{3}$ is rational (namely that $\sqrt{3} = a/b$ where a and b are coprime integers). Thus $\sqrt{3}$ is not rational, that is $\sqrt{3}$ is irrational. □

The Pigeonhole Principle. This states that if there are $n-1$ slots and n objects, if each object is placed in a slot then some slot must receive two objects. This is a rather grand name for a statement of the obvious.

Lemma 1.2.3. *Suppose there are n objects and $n-1$ colors, and suppose each object is colored with one of the colors. Then at least two objects get the same color.*

Proof. Suppose for a contradiction that each color is used for at most one object. Then take the at most one object colored with each of the $n-1$ colors. This is at most $n-1$ objects. Thus at least one object is uncolored, a contradiction. It follows that at least one color is reused or in other words two objects receive the same color. □

As a rule, this level of painstaking precision is not needed. It would suffice to note that as there are n objects and only $n-1$ colors, some color is used for at least two objects.

Question 3. *What is the largest number of objects that can be colored with n colors where each color may be used twice, but not three times?*

1.2.3 Induction

This is a powerful technique for proving results that have an integer index, e.g.

$$P(n): \quad \sum_{i=1}^{n} i = \frac{1}{2}n(n+1).$$

The goal is to prove that $P(n)$ is true for every (sensible) value of n, say for integers $n \geq 1$. (By the way, what is the meaning of $\sum_{i=1}^{0} i$? This is viewed as a sum of no terms and hence equals 0.) Proving that $P(n)$ is true for all n is done in two steps.

In the first step, called the *base case*, we prove the claim for $n = 1$, that is we prove $P(1)$: $\sum_{i=1}^{1} i = \frac{1}{2} \cdot 1 \cdot 2$. But the left hand side (LHS for short) of this expression equals 1, and the right hand side (RHS) also equals 1. So the claim is true in this case.

In the next step, called the *inductive step*, we prove the claim for $n = k + 1$, assuming it is true for $n = k$, and we do this for all integer values $k \geq 1$. That is, we suppose that $P(k)$ is true, namely that $\sum_{i=1}^{k} i = \frac{1}{2}k(k+1)$: this is called the *inductive hypothesis.* We now need to show that $P(k+1)$ is true, namely that $\sum_{i=1}^{k+1} i = \frac{1}{2}(k+1)(k+2)$. But

$$\sum_{i=1}^{k+1} i = k + 1 + \sum_{i=1}^{k} i = (k+1) + \frac{1}{2}k(k+1),$$

using the inductive hypothesis (namely that $\sum_{i=1}^{k} i = \frac{1}{2}k(k+1)$), and

$$(k+1) + \frac{1}{2}k(k+1) = (k+1)(1 + k/2) = (k+1)(2+k)/2 = \frac{1}{2}(k+1)(k+2),$$

which shows that $\sum_{i=1}^{k+1} i = \frac{1}{2}(k+1)(k+2)$, as desired.

We claim this shows that $P(n)$ holds for all values of $n \geq 1$. Why is this? Well suppose that $P(n)$ did not hold for all values of n. For example, say it did not hold for $n = 5$. But we have shown that $P(1)$ is true, so $P(n)$ holds for $n = 1$. We have also shown that $P(1) \Rightarrow P(2)$ (since we have shown that $P(n) \Rightarrow P(n+1)$ for every $n \geq 1$), and as $P(1)$ is true it follows that $P(2)$ is also true. We have further shown that $P(2) \Rightarrow P(3)$, so $P(3)$ must also be true. As we have shown that $P(3) \Rightarrow P(4)$ and $P(4) \Rightarrow P(5)$ it follows that $P(4)$ and $P(5)$ are also true. So the result does hold for $P(5)$.

More generally, we have shown $P(1)$ and $P(1) \Rightarrow P(2) \Rightarrow P(3) \Rightarrow \cdots \Rightarrow P(n)$, from which we can conclude that $P(n)$ is true for any given integer $n \geq 1$.

Often, we will simple state the result without specifying the induction index. The above result would be written as $\sum_{i=i}^{n} i = \frac{1}{2}n(n+1)$ with the label $P(n)$ omitted. The statement is implicitly understood as holding for all sensible n (in this case, integers greater than or equal to 1).

The General Structure of a Proof by Induction

What is an effective way of presenting a proof by induction? I recommend the following approach.

- First, you state the indexed result that you are going to show, including the range of integers for which it will be shown to hold; e.g.

  We show $P(n)$ is true for $n \geq c$.

  Note that c is an integer.

- Second, you state and prove the result for the base case:

  Base Case: $n = c$ (or in more detail: we show $P(c)$).
  Then prove that $P(c)$ is true.

Sometimes you will need to have several base cases, e.g. you need to show both $P(c)$ and $P(c+1)$.

- Third, you state and show the inductive assertion.

  Inductive Step: We show, for all integers $k \geq c$, that if $P(k)$ is true, then so is $P(k+1)$, or for short $P(k) \Rightarrow P(k+1)$.
  Then give the proof.

- Finally, you state the result.

  Conclusion: It follows that $P(n)$ is true for all $n \geq c$.

Question 4. *Prove that $Q(n)$ is true for all $n \geq 1$, where $Q(n)$ is the assertion $\sum_{i=1}^{n} i(i+1) = \frac{1}{3}n(n+1)(n+2)$.*

A More General Form of Induction — Strong Induction

Consider the following problem: Let T be a non-empty n-node binary tree, in which every non-leaf or internal node has exactly two children. Let $e(T)$ denote the number of leaf nodes minus the number of non-leaf nodes in T, the *excess* for T.

Let $P(n)$ be the following assertion: $e(T) = 1$.

We would like to use a proof by induction to prove this result. To do this, we need a stronger assumption in the inductive step, namely we seek to prove that:

If $P(h)$ is true for all $h \leq k$ then $P(k+1)$ is also true.

Notice that this is fully consistent with the previous approach, for to prove $P(k)$ is true we needed that $P(k-1)$ be true, and to prove $P(k-1)$ is true we needed that $P(k-2)$ be true, and so forth. So in fact if we assume $P(k)$ is true we might as well assume all of $P(c)$, $P(c+1), \cdots, P(k)$ are true, where c is the base case value.

Now back to the problem. We prove the result by induction on n.

We define $e(T)$, the *excess* for tree T, to be the number of leaf nodes minus the number of non-leaf nodes in T. Our task is to show that $e(T) = 1$.

Base case. $n = 1$.
Here T is a tree with a single node. T has one leaf and zero non-leaf nodes, so in this case $e(T) = 1$.

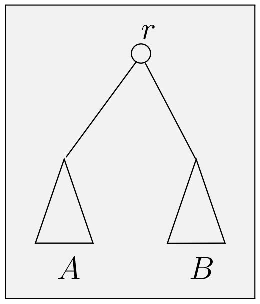


Figure 1.5: Tree T

Inductive step. We show, for $k \geq 1$, that if $P(h)$ is true for all $h \leq k$ then $P(k+1)$ is also true.
Let T be a tree of $k+1$ nodes. Then T's root must have two children (as it cannot have zero children). So T has the form shown in Figure 1.5, where r is the root and A and B are the subtrees rooted at r's children. A has at most $k-1$ nodes (for there are $k+1$ nodes in T, one of which is accounted for by T's root, and at least one more by the nodes(s)

in B); similarly, B has at most $k-1$ nodes. So we can apply the inductive hypothesis to each of A and B. It follows that $e(A) = 1$ and $e(B) = 1$. Now $e(T)$ is computed by summing the excesses for its subtrees and subtracting 1 to account for its non-leaf root node; i.e. $e(T) = e(A) + e(B) - 1$.

Conclusion. $P(n)$ is true for all $n \geq 1$, that is if T is a non-empty n-node binary tree in which all internal nodes have exactly two children, then $e(T) = 1$.

Structural Induction Often, when using strong induction, the value of n is not significant; rather, what matters is that the size of the sub-objects to which the inductive hypothesis is being applied are smaller than the size ($n+1$ or $k+1$) of the original object. In fact, this is the case for the above problem on binary trees. This type of induction is often called structural induction.

Recursion and Induction

Recursion is simply an implementation of induction.
Recall the Tower of Hanoi problem.

Input. n rings $r_1, r_2, \cdots, r_n$, where r_i has radius i, and 3 posts named A, B, and C.

Task. Suppose the rings are all initially on post A with larger radii rings beneath smaller radii ones, so from bottom to top they are in the order $r_n, r_{n-1}, \cdots, r_1$. The task is to move all the rings from post A to post B using post C to help as needed. There are two rules:

1. Only a ring at the top of a post may be moved.
2. A ring when moved may be placed only on either a larger radius ring or an empty post.

The algorithm ToH to move the n rings is shown below.

```
ToH($n, A, B, C$)
    **if** $n = 1$ **then** move ring 1 from Post $A$ to Post $B$
    **else do**
        ToH($n-1, A, C, B$) (* moves the top $n-1$ rings from Post $A$ to Post $C$ *)
        move ring $n$ from Post $A$ to Post $B$
        ToH($n-1, C, B, A$) (* moves the $n-1$ rings on Post $C$ to Post $B$ *)
    **end**
```

We can argue that ToH moves the rings correctly using a proof by induction on n, the number of rings.

To do this we need the correct inductive hypothesis, which is the following.

Suppose there is a legal arrangement of $n+m$ rings on the 3 posts (legal in the sense that smaller rings are always on top of larger ones), with the n smallest rings on post A. Then ToH(n, A, B, C) correctly moves the n smallest rings from Post A to Post B.

We call this inductive hypothesis $P(n)$. Note that for each fixed value of n, $P(n)$ holds for all possible values of m $(m = 0, 1, 2, \cdots)$.

Comment. The m rings play no role in the argument. However, in subproblems we have to account for the rings that do not move, and this is why the additional rings are present in the statement of the inductive hypothesis.

Base case. $n = 1$.
Here the algorithm correctly moves the smallest ring from Post A to Post B. Regardless of the positions of the other rings this is always a legal move.

Inductive step. Suppose that ToH works correctly for $n = k$, for any m. We show that it also works correctly for $n = k + 1$, for all m.

By the inductive hypothesis, the first recursive call, ToH(k, A, C, B), correctly moves the k smallest rings from Post A to Post C. So at this point ring $k + 1$, the $(k + 1)$th smallest ring, is at the top of Post A, and on Post B there are only larger radius rings. Therefore the subsequent action, the move of ring $k + 1$ from Post A to Post B is legal. Finally, by the inductive hypothesis, the second recursive call, ToH(k, C, B, A), correctly moves the k smallest rings from Post C to Post B.

Conclusion. This shows ToH$(k + 1, A, B, C)$ correctly moves the $k + 1$ smallest rings from Post A to Post B.

1.2.4 Zero Knowledge Proofs

This section is included so as to demonstrate that arguments can be compelling and incontrovertible, without falling into one of the categories you have likely encountered before.

We illustrate this proof style by means of two examples. We begin with a very simple problem. Suppose there are two envelopes, labeled E_A and E_B. Each envelope holds a card. A labels the back of the card in E_A, B the back of the card in E_B. Their fronts are numbered either 0 or 1; these numbers are called the cards' numbers. Suppose that a *prover*, Pam, knows the cards have the same number and that she wants to prove this to Chris, a *checker*.

Pam proceeds as follows. She needs one additional card, whose back is labeled H (for Helper), and whose number will be whichever of 0 and 1 is not used by Cards A and B; but card H's number will not be revealed. First, Pam puts Card H in envelope E_A. Then she removes the cards from E_A and shows their fronts to Chris; he sees that one front has the number 0 and other has the number 1. Next, after shuffling, their backs are shown to Chris and he sees that they are Cards A and H. Pam now moves card H to envelope E_B taking care not to show its front. Again, she removes the cards from E_B and shows their fronts to Chris; once more, he sees one front has the number 0 and other has the number 1. Again, after shuffling, Pam shows their backs to Chris and he sees that they are Cards B and H. Chris can conclude that Cards A and B have the same number, for their numbers are both unequal to the number on Card H. However, he has gained no information regarding their actual number. For what has he seen? (See Figure 1.6.) A pair comprising one card numbered 0 and one card numbered 1, twice. He would see this whether Cards A and B are both numbered 0 or both numbered 1. In other words, he has learned one bit of information: the fact that Cards A and B have the same number, and nothing more.

Comment. This procedure is called a *zero-knowledge proof.* A more accurate term might be *minimal knowledge*, but the former is the standard name.

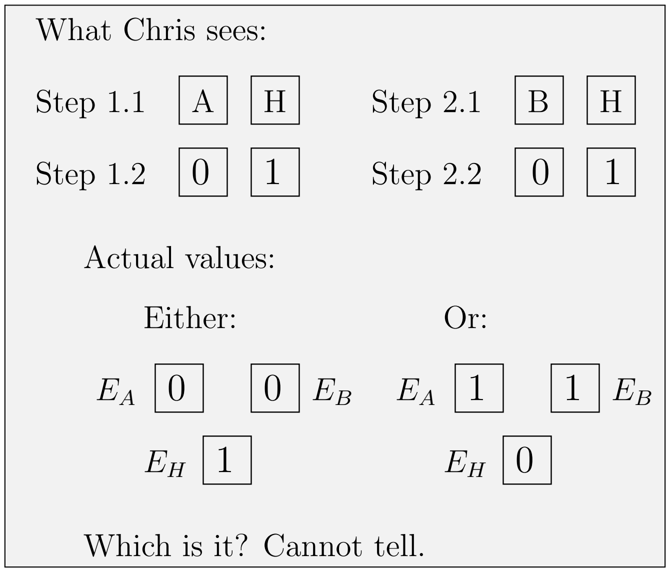


Figure 1.6: The Zero Knowledge Proof for Equal Values.

Next, we turn to a more substantial problem, namely graph coloring. The input is an undirected graph G. The task is to determine if the vertices of G can be colored using 3 colors (Red, Green, and Blue, say) in such a way that no two adjacent vertices have the same color. Note that the output is simply one of the answers "Yes" or "No."

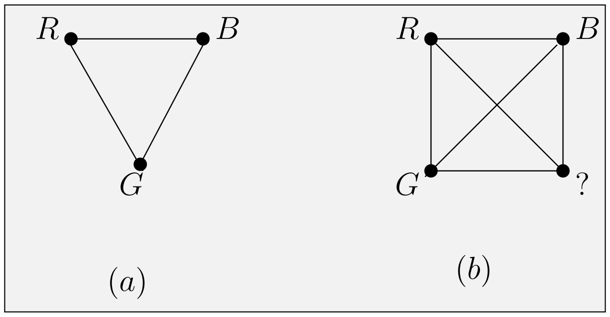


Figure 1.7: (a) The triangle is 3-colorable. (b) The complete graph on 4 vertices is not 3-colorable.

For example the triangle (see Figure 1.7a) can be 3-colored but the complete graph on 4 vertices (see Figure 1.7b) cannot be.

Now suppose that for a given n vertex graph G a *prover*, Pam, happens to know a 3-coloring (n is a million, say). Suppose Pam wants to prove to Chris, a *checker*, that G is indeed 3-colorable. She can use the following method, which does not reveal the 3-coloring of G, yet is still incontrovertible.

Suppose that G has n vertices and m edges. Pam's proof of 3-colorability uses $n + m$ physical cards, one card for each vertex and one card for each edge. On the back of each card, Pam writes the name of the corresponding edge or vertex. On the other side, if the card is for a vertex u, Pam writes u's color in the coloring she knows, and if for an edge (u, v) she writes the third color, the one not used by either of the vertices u or v.

Next, Pam will demonstrate, for each edge (u, v) in G, that u and v have been given distinct colors, but without revealing those colors. To do this, Pam simply takes the three cards for u, v and (u, v), shows Chris the backs, with the names u, v, (u, v) on them, then shuffles the three cards, turns them over and shows Chris the resulting 3 colors R,B,G, then takes the cards back, reshuffles them, and turns them back side up once more. As a result, Chris knows that u and v have distinct colors, but does not know anything beyond this about the colors.

Once this has been done for every edge Chris knows that only 3 colors have been used to color the vertices and that every pair of adjacent vertices have distinct colors, that is, that G is 3-colorable.

Yet, Chris knows nothing more. For Chris knows upfront that if the graph is 3-colorable then Pam's demonstration of this will result in his seeing the triple R,G,B m times, once for each edge, and this would be the case regardless of the details of the coloring, so long as it was a 3-coloring. So all that Chris knows is the following one bit of information: that the graph is 3-colorable. More formally, suppose that Chris can deduce D in addition to the fact that G is 3-colorable. Instead of watching the demonstration, Chris could carry out the following thought experiment. Chris supposes G is 3-colorable, and then imagines he is shown the 3-colors R,B,G a total of m times. Now he can deduce D. In other words, he knows that "G is 3-colorable" implies D. Thus the only additional information Chris learns from the demonstration is the one-bit fact that G is 3-colorable.

On the other hand, if the demonstration fails (because one of the triples of cards uses just two or even one color, e.g. R,B,R) then what Chris learns is that the proposed coloring of the graph is not a 3-coloring. However, this does not demonstrate that the graph is not 3-colorable. In sum:

Theorem 1.2.4. *There is a (physical) procedure to demonstrate the 1-bit fact that a 3-colorable graph is 3-colorable, without revealing anything beyond this 1-bit.*

1.3 Short Preview: Computationally Unsolvable Problems

Self-reference is well-known for creating paradoxes, as in the following conundrum:

This sentence is false.

For if the sentence is true then it is false, and if false then it is true.

Analogous difficulties arise with questions about programs, for programs (viewed as text) can be the input for other programs (e.g. compilers) including themselves (a compiler viewed as text could be the input to the same compiler viewed as a program).

In particular, this type of difficulty occurs with the *Halting Problem*, which seeks to answer the following class of questions:

Does the computation of program $P(\ )$ on input x eventually terminate where x is a possible input for P?

Note that P not terminating on input x might be the result of an infinite loop or an unbounded recursion.

We will prove that there is no algorithm which will answer this question correctly for every possible pair of P and x.

Such an algorithm, if it existed, on an input pair (P, x), answers one of "Yes" or "No." Note that an algorithm that simulates P on input x does not provide an adequate solution. For while if P does terminate on input x, the simulation will eventually discover this, and then an answer of "Yes" is justified, if P does not terminate this is never known for sure, and thus neither a "Yes" nor a "No" answer is justified.

The question is not whether for some particular P and x termination can be determined, but whether there is a systematic procedure than can determine it for every possible pair of P and x. As we will see, there is no such procedure.

Suppose our programs are written in binary and likewise for their inputs. (In practice they are written in the 256-character ASCII code, i.e. 8 bits per character.) Now imagine listing all strings of binary characters in lexicographic order: length 1 strings ordered alphabetically, then length 2 strings, and so forth. i.e., 0, 1, 00, 01, 10, 11, 000, 001, etc. Clearly, this listing includes all legitimate programs, as well as many other strings. It is convenient to number these strings, starting from 1 (or 0 if preferred); so 0 is string 1, 1 is string 2, 00 is string 3, etc.

To prove our result, let us assume, in order to obtain a contradiction, that there is a 2-input program H that solves the halting problem. H computes answers (Y, N) in the table shown in Figure 5.2. The ith row describes the behavior of the i-th program, P_i, on each of its possible inputs, $w_1, w_2, \ldots$. As we are using an integer to represent the corresponding string, rather than write $H(P_x, w_y)$, we will write $H(x, y)$ for short.

Using H as a subroutine we are going to construct a function D (computed by program P_d in the table) which has the opposite behavior to what is shown in the corresponding diagonal entry in the table. That is, if $H(x, x)$ = "Yes", then $D(x) = P_d(x)$ runs forever, while if $H(x, x)$ = "No" then $D(x)$ eventually terminates with output "Yes" (the particular output is unimportant; what matters is that it halts). D is defined as follows.

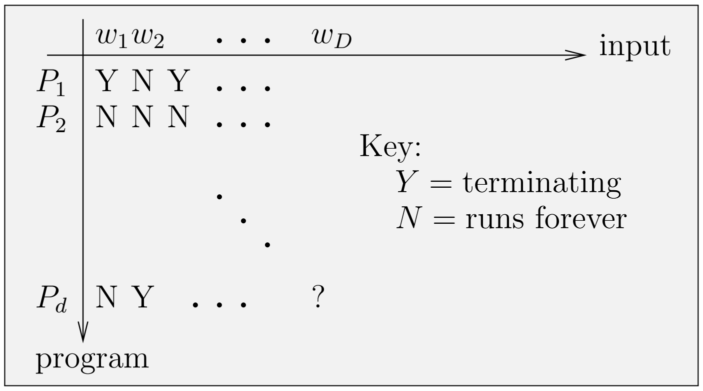


Figure 1.8: Function D.

```
D(x)
    Simulate H(x,x);
    **If** H(x,x) outputs "Yes" **then** loop forever
        **else** (* H(x,x) outputs "No" and *) D(x) outputs "Yes"
```

Side note: If x is not a valid program, we define $H(x, y)$ = "Yes."

Observe that

$$D(x) = P_d(x) \text{ terminates} \iff H(x,x) \text{ outputs "No"} \iff P_x(x) \text{ does not terminate.}$$

Setting $x = d$ yields

$$P_d(d) \text{ terminates} \iff P_d(d) \text{ does not terminate,}$$

which is the contradiction we seek.

This contradiction shows that the assumption that H exists must be false.

Comments. i. We have actually shown that there is no program H that can answer the question of "does program P_x on input x eventually halt?" for all possible programs P_x.

Notice that the difficulties occurred when considering the self-referential $P_d(d)$.

ii. One way of viewing this result is that it states that there is no algorithm to debug programs in general (by in general we mean that it works for every possible program). For if we cannot even determine in general if a program eventually stops on a given arbitrary input, how could we determine whether a program performs the computation that we want it to perform.

1.3.1 Overview of the Remainder of the Text

In Chapters 4 and 5, we will explore the boundary between computable and non-computable problems more thoroughly. We will also see that one proof of non-computability suffices: we will use the fact that the Halting Problem is unsolvable to show other problems are also unsolvable.

But knowing that a problem is solvable need not mean that it can be solved in practice. To be usable, an algorithm must terminate in a reasonable time period. In Chapter 6, we will formalize this notion via the class of **NP**-Complete problems and introduce the famous $\mathbf{P} \neq \mathbf{NP}$ conjecture.

But we will begin by looking at classes of computations that are much more feasible. These are classes of computations that can be carried out with quite simple computational devices, namely Finite Automata (Chapter 2) and Pushdown Automata (Chapter 3); we will be studying the classes of problems these devices can solve.

Exercises

1. What is the size of the empty set, $|\emptyset|$?

2. Show that if $A \subseteq B$ then $A \cap \overline{B} = \emptyset$.
   Hint. One way to show that $A \cap \overline{B} = \emptyset$, is to show the following two results: (i) if $x \in A$ then $x \notin \overline{B}$ and (ii) if $x \in \overline{B}$ then $x \notin A$.

3. Show that if $|A| = n$, then $|\textsc{PowerSet}(A)| = 2^n$.

4. Using the recursive definition of u^R ($\lambda^R = \lambda$, and if $v = au$ then $v^R = u^R a$) Show that $(u^R)^R = u$.
   Hint. i. Show by induction that $(ub)^R = bu^R$.
   ii. Use (i) to show that $(u^R)^R = u$.

5. Show that:

   i. $(x \vee y) \wedge z = (x \wedge z) \vee (y \wedge z)$.

Sample solution. We have to show that for every setting of the binary variables x, y, z the LHS and RHS of the equation evaluate to the same truth value. We proceed as follows.
If $z = \text{TRUE}$, then the LHS equals $(x \vee y) \wedge \text{TRUE} = x \vee y$. The RHS equals $(x \wedge \text{TRUE}) \vee (y \wedge \text{TRUE}) = x \vee y$. These are equal.
While if $z = \text{FALSE}$, the LHS equals $(x \vee y) \wedge \text{FALSE} = \text{FALSE}$, and the RHS equals $(x \wedge \text{FALSE}) \vee (y \wedge \text{FALSE}) = \text{FALSE} \vee \text{FALSE} = \text{FALSE}$. Again, the LHS and RHS are equal.

ii. $(x \wedge y) \vee z = (x \vee z) \wedge (y \vee z)$.
iii. $\neg(x \wedge y) = (\neg x \vee \neg y)$.
iv. $\neg(x \vee y) = (\neg x \wedge \neg y)$.

6. Show that:

i. $(A \cup B) \cap C = (A \cap C) \cup (B \cap C)$.

Sample solution. This result follows by showing both that $(A \cup B) \cap C \subseteq (A \cap C) \cup (B \cap C)$ and that $(A \cap C) \cup (B \cap C) \subseteq (A \cup B) \cap C$. We show each in turn.
The first result means the following: if $x \in (A \cup B) \cap C$ then $x \in (A \cap C) \cup (B \cap C)$. To see this is true we argue as follows. $x \in (A \cup B) \cap C$ means that $x \in A \cup B$ and $x \in C$. $x \in A \cup B$ means that $x \in A$ or $x \in B$ (and possibly both). If $x \in A$, as $x \in C$ also, $x \in A \cap C$; similarly, if $x \in B$, then $x \in B \cap C$. As $x \in A$ or $x \in B$, it follows that $x \in (A \cap C) \cup (B \cap C)$.
The second result is shown analogously. It means the following: if $y \in (A \cap C) \cup (B \cap C)$ then $y \in (A \cup B) \cap C$. Now $y \in (A \cap C) \cup (B \cap C)$ means that $y \in A \cap C$ or $y \in B \cap C$ (or possibly both). If $y \in A \cap C$ then $y \in A$ and $y \in C$. If $y \in A$ then $y \in A \cup B$. Thus, in this case, $y \in (A \cup B) \cap C$. Similarly, if $y \in B \cap C$ it follows that $y \in (A \cup B) \cap C$. Thus in either case $y \in (A \cup B) \cap C$.
Both subsidiary results have been shown; it follows that $(A \cup B) \cap C = (A \cap C) \cup (B \cap C)$.

ii. $(A \cap B) \cup C = (A \cup C) \cap (B \cup C)$.
iii. $\overline{A \cap B} = \overline{A} \cup \overline{B}$.
iv. $\overline{A \cup B} = \overline{A} \cap \overline{B}$.

7. Show that $3^{1/3}$ is not rational.

8. We will call a number r a square rational if it can be written in the form $r = a^2/b^2$ where a and b are integers. Show that 5 is not a square rational.

9. We will call a number r lopsided if it can be written in the form $r = a^2/b$ where a and b are integers, but with a and b having no common factors. Show that 7 is not lopsided.

10. Let $G = (V, E)$ be an undirected graph. Show that if every vertex of G has odd degree then G has an even number of vertices. (The degree of a vertex v is the number of edges having v as an endpoint.)

11. i. What is the largest number of objects that can be colored with n colors where each color may be used twice, but not three times?

    ii. What is the smallest number of objects that cannot be colored with n colors where each color may be used twice, but not three times?

12. Find the mistake in the following "proof."

> Consider the people in the world and suppose that if p is a friend of q then q is also a friend of p. Suppose further that everyone has at least one friend (not themselves).
>
> **Claim**. For any two distinct people r and s, there is a chain of friends connecting r and s, that is there is a sequence $r = p_1, p_2, \cdots, p_k = s$, with p_i and p_{i+1} being friends for $1 \le i < k$.
>
> **Proof**. We prove this by induction on n, the number of people in the world.
>
> **Base case**. $n = 1$.
> Then the claim is trivially true (as there cannot be two distinct people r and s).
>
> **Second base case**. $n = 2$.
> Let r and s be the two people in the world. By assumption, they are friends, so again the claim is trivially true.
>
> **Inductive step**. Suppose the claim is true for worlds of k or fewer people. We prove it for the world of $k + 1$ people.
> Let q be a person, other than r or s.
> Consider removing q from the world and for each pair u, v of friends of q, making u and v friends in the new world of the k remaining people. Clearly, for each pair of people, there is a chain connecting them in the new world if and only if there is a chain connecting them in the old world; the new chain may require using a new u—v link to replace a pair of links u—q and q—v. By induction, there is a chain between any pair of people in the new k-person world; consequently, there is also a chain in the old world connecting any pair of people.

What is incorrect in the above argument? Is it just the argument that is incorrect or is the claim false? A natural way to show a claim is false is to give a counterexample, an example for which the claim does not hold. Even if the claim is false, **you still need to explain what aspect of the argument is incorrect**; a natural way to do this is to explain where the argument fails on the counterexample.

13. Find the mistake in the following "proof."

Claim. Consider a girl scout camp attended by n girls, where $n \geq 8$. The camp counselors want to form teams for a scavenger hunt and believe that the ideal sizes for the teams are to have 4 or 5 girls per team. One of the counselors argues as follows that this is always possible with 8 or more girls.

Argument (Proof). Let $P(n)$ be the assertion that we can do this team formation with camps of n girls. We show that $P(n)$ is true for $n \geq 8$ by (strong) induction.

Base case. If there are 8 girls we make two teams of 4 girls each. If there are 9 girls we make a team of 4 and one of 5 girls. If there are 10 girls we make two teams of 5 girls each.

Inductive Step. Suppose that $P(8)$, $P(9)$, $\cdots$, $P(k)$ are true. We show that $P(k+1)$ is also true. Proceed as follows. Form one team of 4 girls; then, by the inductive hypothesis applied to $P(k-3)$, the remaining $n-3$ girls can be put into teams of 4 and 5 girls. Thus the $k+1$ girls can also be so divided.

What is incorrect in the above argument? Is it just the argument that is incorrect or is the claim false? A natural way to show a claim is false is to give a counterexample, an example for which the claim does not hold. Even if the claim is false, **you still need to explain what aspect of the argument is incorrect**; a natural way to do this is to explain where the argument fails on the counterexample.

14. Consider Problem 13. What is the smallest value c for which the claim is true for all $n \geq c$?

15. Suppose that $n \geq 2$ players take part in a tennis competition. Suppose the ith player plays games with $d_i \geq 0$ different players during the competition. Show that at least two players must play with the same number of other players, that is there are values j and k with $0 \leq j, k \leq n$ such that $d_j = d_k$.
Comment. Be careful; what are the possibilities when $n = 2$? There is more than one.

16. i. Show by induction that $Q(n)$ is true for all $n \geq 1$, where $Q(n)$ is the assertion $\sum_{i=1}^{n} i(i+1) = \frac{1}{3}n(n+1)(n+2)$.
    ii. Show that $\sum_{i=1}^{n} i(i+1)(i+2) = \frac{1}{4}n(n+1)(n+2)(n+3)$.
    iii. Show that $\sum_{i=1}^{n} i(i+1)(i+2)\cdots(i+h) = \frac{1}{h+2}n(n+1)(n+2)\cdots(n+h+1)$ for any $h \geq 0$.

17. Prove by induction that $\sum_{i=0}^{k} r^i = \frac{r^{k+1}-1}{r-1}$, for $r \neq 1$.

18. Let $H_k = \sum_{i=1}^{k} \frac{1}{i}$. H_k is called the kth harmonic number. Show by induction on m that $1 + m/2 \leq H_{2^m} \leq m+1$.

19. Show by induction that $e(\frac{n}{e})^n \leq n!$ for all integers $n \geq 1$. You may assume that $(1+1/x)^x \leq e$ for all $x \geq 1$. Recall that $e \approx 2.72$ is the natural logarithm base.
Hint. Assuming the inductive hypothesis for $n = k$, what do you need to show to obtain the result for $n = k+1$?

20. Let T be a non-empty tree with n internal (non-leaf) nodes of which n_i have i children, for $i = 1, 2, \ldots, k$ (and $\sum_{i=1}^{k} n_i = n$). Show by strong induction that T has $1 + \sum_{i=2}^{k}(i-1)n_i$ leaf nodes.

Hint: The cleanest way of doing this is to use an appropriate definition of excess for such trees, analogous to the one used for the similar result for binary trees. Also recall the way that trees are defined recursively.

21. Consider the recurrence equation

$$\begin{aligned} T(1) &= 1 \\ T(n) &= n + \max_{\substack{i+j=n \\ 1\le i,j<n}} \{T(i)+T(j)\} \qquad\qquad n > 1. \end{aligned}$$

Show by induction that $T(n)$ is maximized by choosing $i = 1$, $j = n - 1$.

Comment. This suggests that the worst case for a divide-and-conquer procedure is the most unbalanced case.

22. Let F be a Boolean formula constructed from Boolean variables and the operators "and," "or" and "not." Prove, by strong induction, that there is an equivalent formula F' in which all the not operators are applied solely to variables. e.g. The formula $\neg x \wedge \neg y$ is in the desired form, but $\neg(x \vee y)$ is not. Finally, recall that two formulae are equivalent if for every truth setting of the Boolean variables they evaluate to the same value.
Hint. F has one of the following forms: either a simple variable x, or one of $\neg G$, $G \vee H$, $G \wedge H$ where G and H are themselves Boolean formulas. There are two base cases, and for the inductive step, there are three possible forms for the formula $\neg G$ (i.e. you need to consider the expansion of G into each of three possible cases).

23. The game of Nim is played as follows by two players, A and B say.

n sticks are placed in a pile.

In turn, starting with player A, each player removes one or two sticks from the pile. The player to remove the last stick loses.

Show by induction on k that B can force a win if there are $3k + 1$ sticks in the pile initially; in addition, show that A can force a win otherwise.

b. In Variant-Nim each player can remove one, two, or three sticks. When can B force a win now? Justify your answer.

24. This is a Tower of Hanoi variant. Let R be a legal arrangement of the n rings. $R[i]$ denotes the post on which ring i is sitting. The task is to move all the rings from arrangement R to Post C using legal moves. Do this by giving a recursive algorithm VAR-TOH(P, k, Z), where P is the current arrangement of rings, and k indicates that the k smallest rings are being moved in this procedure call, with the k smallest rings ending up on post Z (Z is one of A, B, or C).
Hints: What is the base case? Consequently, what must the recursive call or calls do? Note

that you need to update P when moving a ring.
Comment: It is very hard to solve this problem without using recursion. But so long as you remember that recursive subproblems take care of themselves, this is not a difficult problem. A recursive call can be treated as a black box; you don't have to work out how it is carrying out its task.

25. This is another Tower of Hanoi variant. Let R and S be two legal arrangements of the n rings. $R[i]$ denotes the post on which ring i is sitting initially. $S[i]$ denotes the post on which ring i is to sit at the end of the algorithm. The task is to move the rings from arrangement R to arrangement S using legal moves. Do this by giving a recursive algorithm VAR2-TOH(P, Q), which given inputs P and Q moves the rings from arrangement P to arrangement Q using legal moves, where P is the current arrangement of rings, and Q is the desired arrangement at the end of the procedure call.
Hints: What is the base case? Consequently, what must the recursive call or calls do? What do you need to do with the largest ring? Hence what do you need to do with the other rings? Note that you need to update P when moving a ring.

26. Recall the Binary Search algorithm for testing if an item x lies in a sorted array $A[i:k]$.

```
BS($A, i, k, x$) – returns TRUE if $x = A[j]$ for some $j$, $i \le j \le k$,
        and returns FALSE otherwise.
    if $i > k$ then return FALSE
    else if $i = k$ then
        if $A[i] = x$ then return TRUE else return FALSE
    else do
        $mid \leftarrow \lfloor \frac{i+k}{2} \rfloor$;
        if $A[mid] > x$ then return BS($A, i, mid - 1$);
        else if $A[mid] = x$ then return TRUE
        else (* $A[mid] < x$ *) return BS($A, mid + 1, k$);
    end (* else do *)
```

Prove by induction that BS reports correctly whether x lies in array $A[i:k]$, using the following inductive hypothesis:

BS reports correctly on arrays of $n = \max\{0, k - i + 1\}$ items.

Sample solution. We use strong induction on n.

Base case. When $n = 0$, i.e. $i > k$, BS correctly returns FALSE.

Inductive step. Suppose that the inductive hypothesis holds for $n \leq l$; we show it also holds for $n = l + 1$.

Case 1. $A[mid] = x$, where $mid = \lfloor \frac{i+k}{2} \rfloor$. Then BS correctly returns TRUE.

Case 2. $A[mid] > x$. If x is present in $A[i : k]$, as A is in sorted order, x would be in $A[i : mid - 1]$. In this case BS recursively calls BS$(A, i, mid - 1)$. As $mid - 1 - i + 1 = mid - i < n$, by the inductive hypothesis, BS returns the correct answer.

Case 3. $A[mid] < x$. The argument is completely analogous to the one for Case 2.

In every case, BS returns the correct answer, which proves the inductive hypothesis for $n = l + 1$.

We conclude that BS reports correctly whether x lies in array $A[i : k]$.

27. Recall the Binary Search algorithm for locating an item x in a sorted array $A[i : k]$, when present.

```
FBS($A, i, k, x$) – returns $j$ if $x = A[j]$ for some $j$, $i \leq j \leq k$,
        and returns FALSE otherwise.
    if $i > k$ then return FALSE
    else if $i = k$ then
        if $A[i] = x$ then return $i$ else return FALSE
    else do
        $mid \leftarrow \lfloor \frac{i+k}{2} \rfloor$;
        if $A[mid] > x$ then return FBS($A, i, mid - 1$);
        else if $A[mid] = x$ then return $mid$
        else (* $A[mid] < x$ *) return FBS($A, mid + 1, k$);
    end (* elso do *)
```

Prove by induction that BS reports correctly: if $x = A[j]$ for some j, $i \leq j \leq k$, it returns j, and otherwise it returns FALSE, by using the following inductive hypothesis:

BS reports correctly on arrays of $n = \max\{0, k - i + 1\}$ items.

28. Recall the Merge Sort algorithm:

```
MERGESORT($A, i, j$) – sorts the items in array $A[i : j]$
    if $i \geq j$ then return
    else do
        $mid \leftarrow \lfloor \frac{i+j}{2} \rfloor$;
        MERGESORT($A, i, mid$);
        MERGESORT($A, mid + 1, j$);
        MERGE($A, i, mid, j$)
    end (* elso do *)
```

Prove by induction that MergeSort sorts correctly, using the following inductive hypothesis:

> MergeSort correctly sorts arrays of n items.

You may assume that Merge(A, i, mid, j) will merge sorted subarrays $A[i : mid]$ and $A[mid+1 : j]$ into sorted order in array $A[i : j]$.

29. Recall Dijkstra's single source shortest path algorithm. The input is a directed graph $G = (V, E)$ together with a source vertex $s \in V$ and for each edge $e = (i, j)$ a length $\ell(i, j)$ which gives the length of the edge. You may assume all edge lengths are positive values. The algorithm finds, for each vertex $v \in V$, the length of a shortest path from s to v.

```
Dijkstra($G, s$)
    initialize: $D(v) \leftarrow \infty$ for all $v \in V - \{s\}$; $D(s) \leftarrow 0$; $S \leftarrow \emptyset$;
    Create empty Priority Queue $Q$; for all $v \in V$ Insert($Q, v$) with key $D(v)$;
    while $Q$ not empty do
        $v \leftarrow$ DeleteMin($Q$); $S \leftarrow S \cup \{v\}$;
        for each edge $(v, w)$ do
            if $D(v) + \ell(v, w) < D(w)$ then do
                $D(w) \leftarrow D(v) + \ell(v, w)$; ReduceKey($Q, w, D(w)$)
                end (* then do *)
        end (* while *)
```

Show by induction that Dijkstra's algorithm is correct, using the following inductive hypothesis.

> After k iterations of the while loop S contains the k vertices in V nearest to s (with ties broken arbitrarily). Further, for every vertex $v \in V$, $D(v)$ is the length of a shortest path from s to v among the following paths: those paths all of whose vertices are in the set S, except possibly for v.

Recall that Insert(Q, v) inserts item v into the Priority Queue Q where v has an associated key value, that ReduceKey(Q, w, k) reduces the key value for item w in Q to k (the data structure works correctly only if k is less than or equal to the current value of w's key), and that DeleteMin(Q) removes and returns the item in Q with the smallest value key.

30. i. Suppose you are given two envelopes each holding a card whose front is numbered either 0 or 1. Describe a zero-knowledge proof to show that two cards have distinct numbers. Your proof should not reveal which card has which number if the numbers are distinct, but it may reveal the number if they are the same.

    ii. Now design a zero-knowledge proof that does not reveal the numbers in either case.

31. Recall the game of Sudoku. You are faced with a 9×9 grid of cells, which is further subdivided into nine 3×3 grids. Some of the cells have integers in the range 1–9 written in them, while others are blank. The task is to fill in the blank cells so that each row, each column, and each of the nine 3×3 grids contains each of the nine numbers 1–9 exactly once. Well constructed puzzles have exactly one solution.

    Suppose that Pam knows a solution. Describe a zero-knowledge proof she can use to convince Chris of this fact without revealing anything else about her solution.

32. Let $G = (V, E)$ be an undirected graph. $I \subset V$ is an independent set if every edge $e \in E$ has at most one endpoint in I.

    Suppose that Pam knows an independent set of size k. Describe a zero knowledge proof she can use to convince Chris of this fact without revealing anything else about the independent set.

33. Suppose Pam has two cards with, respectively, the numbers i and $i + 1 \mod n$ on their backs, where $0 \leq i < n - 1$. Give a zero-knowledge proof she can use to convince Chris of this fact without revealing anything about the number i.
    Hint. You may want to consider physical methods besides envelopes, though there is also a solution in which collections of envelopes are put in a larger envelope.

34. Let $G = (V, E)$ be an undirected graph. C is a Hamiltonian Cycle of G if C goes through each vertex of G exactly once.

    i. Suppose that Pam knows a Hamiltonian cycle C for graph G. Why is the following procedure not a zero-knowledge proof of this fact?

       The proof uses one card per edge. On its back, Pam writes the name of the corresponding edge. On the front, she writes a "Y" if the edge is in the circuit and she writes an "N" otherwise.

       For each vertex v in turn, Pam gathers up the cards for the inedges (the edges into vertex v), shows the backs of these cards to Chris (so that he sees she has taken exactly these cards), she shuffles them, turns them over and thereby demonstrates that exactly one of these cards has a "Y;" she then takes the cards back, reshuffles them and turns them back over. This process is repeated for the cards for the outedges from v (the edges leaving v).

       Pam thereby shows Chris that there is one edge entering and one edge leaving each vertex, and consequently that there is a circuit which goes through all the vertices exactly once.

    *ii. Give a correct zero-knowledge proof to show that G has a Hamiltonian cycle.
       Hint. One solution uses n copies of each edge, where G has n vertices. If $e_1, e_2, \cdots, e_n$ is a Hamiltonian cycle expressed as a path of n edges, then the proof will use the ith copy of e_i in demonstrating that there is a Hamiltonian cycle. The solution of Problem 33 provides a useful subroutine.

35. i. In Section 1.3 we showed that $H(x, x)$ is undecidable, that is there is no program that for all x can output TRUE if program P_x halts on input x and outputs FALSE if program P_x does not halt on input x. Now show that $H(x, 2x)$ is undecidable, i.e. show there is no program that for all x can output TRUE if program P_x halts on input $2x$ and outputs FALSE if program P_x does not halt on input $2x$.
       Hint: You will want to build a program E (analogous to D for the solution shown in Section 1.3), but what matters particularly is the action of E on even valued inputs.

    ii. Show that $H(x, x^2)$ is undecidable.

    iii. Explain why it is not possible to modify the argument from Section 1.3 to show that $H(2x, x)$ is undecidable.

Bibliographic Notes

Problem 31 and its solution are due to Michael Rabin. Problems 24 and 25 are based on Tower of Hanoi problems developed by Alan Siegel.

Chapter 2

Finite Automata

This chapter and much of the text concerns strings, which are always defined with respect to a specific alphabet. Recall that an *alphabet* is a set of characters. For example:

- Binary = {0,1}
- Boolean = {T,F}
- English = $\{a, b, \cdots, z\}$

The usual way of writing an unspecified k-character alphabet is as $\Sigma = \{a_1, a_2, \cdots, a_k\}$.

2.1 Regular Expressions: A Motivating Problem

String matching is a common task in text processing. In string matching there are two inputs, a text t and a pattern p, which are both strings of characters over the same alphabet. The task is to find all occurrences of p in t; for example, to find all occurrences of the string "Big Bang Theory" (the p) in an astronomy textbook (the t). A simple algorithm would be to check for an occurrence of p starting at each successive location in t. It is easy to implement this to run in time $O(|p| \cdot |t|)$. There are also several well-known linear time algorithms for this problem, i.e. running in time $O(|p| + |t|)$, most notably the Knuth-Morris-Pratt and the Boyer-Moore string matching algorithms; however, these lie outside the scope of this text.

Often, one would like to perform more complicated types of searches. For example, one might want to search for the singular or plural form of a word, and sometimes this is not just a matter of adding an "s", as with "mouse" and "mice"; of course, in this case, one could just perform two searches, one for each word. But, suppose one wants to search for every phone number in a text. One might describe this as consisting of any sequence of 10 digits in the range 1–9 (at least for US phone numbers), or one might give a more elaborate collection of patterns, which include parentheses, spaces and dashes, as in numbers of the form (123) 456-7890 and 123 456 7890, etc. Presumably, one is not going to search one by one for all 10^{10} or more possible patterns.

More generally, one can describe a class of strings that are specified as regular expressions (to be defined shortly). Interestingly, several Unix commands, such as grep, allow one to search in text files for sets of strings specified by regular expressions. Furthermore, these searches can be done efficiently. One of the questions we will be answering in this chapter is how this is possible.

Regular expressions allow one to specify sets (languages) built up by means of the following three operations: union, concatenation, and iteration (defined below). These are called the *Regular Operations.*

- Union: $A \cup B = \{w \mid w \in A \text{ or } w \in B\}$.
- Concatenation: $A \circ B = \{uv \mid u \in A \text{ and } v \in B\}$. This is the set of strings comprising a string from A followed by a string from B.
- Iteration: $A^* = \{x_1 x_2 \cdots x_k \mid k \geq 0 \text{ and } x_i \in A, \text{ for } i = 1, 2, \cdots, k\}$. Note that setting $k = 0$ shows $\lambda \in A^*$. This is the set of strings consisting of the concatenation of zero or more strings from A. One can also view this as starting with the empty string λ, and concatenating 0 or more strings from A to the initial λ. This is often called the *star* operation, for obvious reasons.

Regular expressions are simply a notation to represent the above operations. We will use small letters such as r, s, r_1, etc, to denote regular expressions. Each regular expression specifies a set or language of strings; we also say that the regular expression *represents* the corresponding language. Also, note that each regular expression is defined with respect to a specified alphabet Σ.

Regular expressions are built using the following six rules, comprising three base cases and three combining rules. (This can be thought of as being analogous to the building of arithmetic expressions using the operators "+", "−", "∗", and "/", along with parentheses.)

We begin with the base cases.

1. $\emptyset$ is a regular expression; it represents $\emptyset$, the empty set.
2. λ is a regular expression; it represents the language $\{\lambda\}$, the language containing the empty string alone.
3. a is a regular expression; it represents the language $\{a\}$.

   For the combining rules, let regular expressions r and s represent languages R and S, respectively. The rules follow.

4. $r \cup s$ is a regular expression; it represents language $R \cup S$.
5. $r \circ s$ is a regular expression; it represents language $R \circ S$. We write rs for short.
6. r^* is a regular expression; it represents language R^*.

Parentheses are used to indicate the scope of an operator. It is also convenient to introduce the notation r^+; while not a standard regular expression, it is shorthand for rr^*, which represents the language RR^*. Also, Σ is used as a shorthand for $a_1 \cup a_2 \cup \cdots \cup a_k$, where $\Sigma = \{a_1, a_2, \cdots, a_k\}$. Sometimes, to emphasize that r represents a language, we write $L(r)$ to denote the language represented by regular expression r. Finally, if $w \in L(r)$, w is said to be *represented* by r.

Examples. Let $\Sigma = \{a, b\}$.

1. $a^*ba^* = \{w \mid w \text{ contains exactly one } b\}$. This can be read as zero or more a's, followed by a b, followed by zero or more a's.

2. $\Sigma^* bab \Sigma^* = \{w \mid w \text{ contains } bab \text{ as a substring}\}$. This can be read as a possibly empty string, followed by the string aba, followed by a possibly empty string.

3. $(\Sigma\Sigma)^* = \{w \mid w \text{ has even length}\}$. This can be read as a sequence of zero of more pairs of characters.

4. $a\Sigma^* a \cup b\Sigma^* b \cup a \cup b = \{w \mid w \text{ has the same first and last character}\}$.

5. $(a \cup \lambda)b^* = ab^* \cup b^*$.

6. $a^*\emptyset = \emptyset$. (Why?[1])

One might ask why this extra notation has been introduced; why not simply stick with the set notation? The answer is that one can be terser, for example writing a^*ba^* rather than $\{a\}^* \circ \{b\} \circ \{a\}^*$.

Thinking recusively It is natural to approach regular expressions recursively. We illustrate this will an algorithm that given as input a regular expression r, tests whether $\lambda \in L(r)$.
Base cases:
i. $r = \phi$. Then $\lambda \notin L(r)$.
ii. $r = \lambda$. Then $\lambda \in L(r)$.
iii. $r = a$. Then $\lambda \notin L(r)$.
Recursive cases:
iv. $r = s \cup t$. Then $\lambda \in L(r)$ exactly if $\lambda \in L(s)$ or $\lambda \in L(t)$. This follows because r represents the language $L(s) \cup L(t)$.
v. $r = s \circ t$. Then $\lambda \in L(r)$ exactly if both $\lambda \in L(s)$ and $\lambda \in L(t)$. This follows because each string in $L(r)$ is the concatenation of a string in $L(s)$ with a string in $L(t)$, and the only way to produce λ with such a concatenation is to concatenate λ with λ.
vi. $r = s^*$. Then $\lambda \in L(r)$, as L^* includes λ for every language L.

Finding strings represented by regular expressions Now, given a regular expression r, which might represent an infinite set of strings, and given a text t, how can one find every substring s of t such that s lies in the set of strings specified by r, let alone do this efficiently? Some instances may seem quite doable; for example searching for aa^*, the set of strings containing one or more a's, seems quite feasible. But handling an arbitrary expression involving multiple concatenations, unions, and iterations (or stars) seems more daunting.

To answer this question we are going to start by looking at a class of devices or machines called finite automata, which as first sight will seem quite unrelated to this problem.

2.2 Finite Automata: The Basic Model

Finite automata model very simple computational devices or processes. These devices have a constant amount of memory and process their input in an online manner. By that we mean that the input is read a character at a time, and for each character, as it is read, a constant amount of processing is performed.

[1]As there is no string in $\emptyset$, to follow a string in a^* with a string from $\emptyset$ is not possible.

One might wonder whether such simple devices could be useful. In fact, they are widely used as controllers in mechanical devices such as elevators and automatic doors.

Example. A controller for the doors on the exit hatchway of a spaceship.

The rather simple spaceship in this example has an exit area with two doors: one leads out into space, the other leads into the spaceship interior, as shown in Figure 2.1.

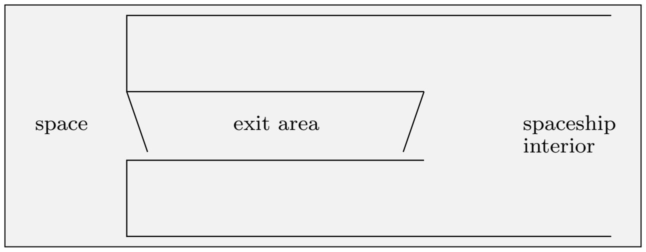


Figure 2.1: Spaceship Exit Doors

The critical rule is that at most one door may be open at any given time (for otherwise all the air in the spaceship could vent out into the vacuum of space). The state of the doors in managed by a controller, a simple device which we describe next. The controller can receive signals or requests to open or close a door. For simplicity, we suppose that only one request can be received at one time. The diagram in Figure 2.2 specifies the operation of the controller. It is simply a directed graph with labels on the edges.

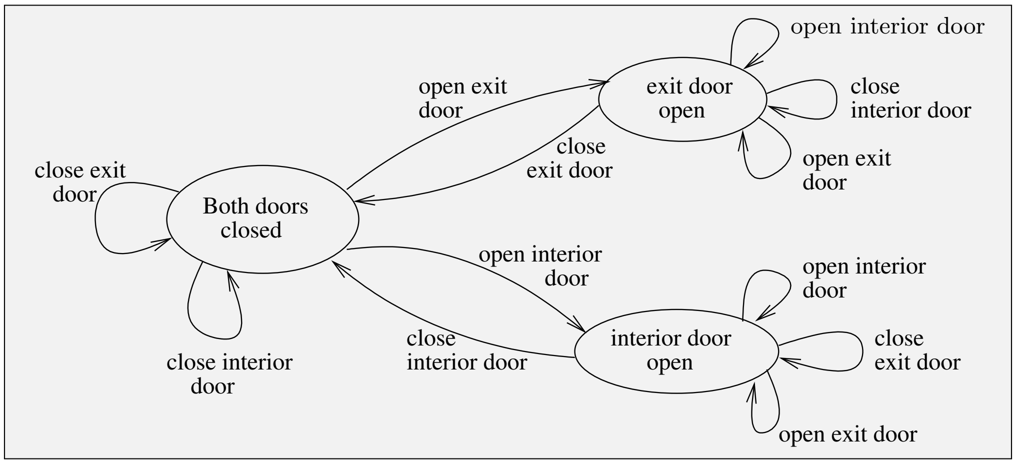


Figure 2.2: Spaceship Door Controller

The vertex names in this graph specify the door state to which the vertex corresponds. The edges exiting a vertex indicate the actions taken by the controller on receiving the request labeling the edge. So for example, if both doors are closed and a request to open the exit door is received, then the controller opens this door and the state changes accordingly. On the other hand, if currently the interior door is open and a request to open the exit door is made, then nothing changes; i.e. the interior door remains open and the exit door remains closed. Note that there are many requests that result in no change. We can simplify the diagram so as to show only those requests that cause a change of state, as shown in Figure 2.3.

As requests are received the controller cycles through the appropriate states. To implement the device it suffices to have a 2-bit memory to record the current state (2 bits can distinguish 4 states)

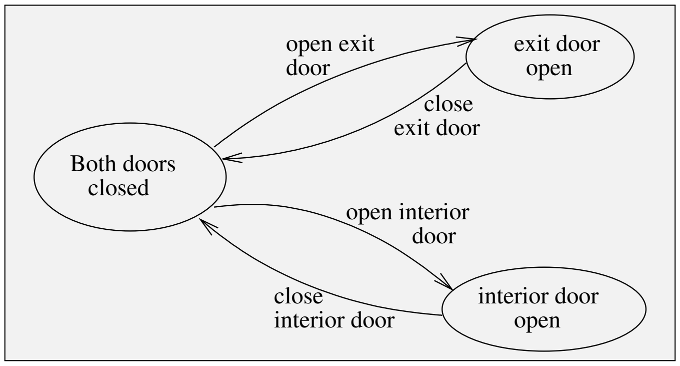


Figure 2.3: Spaceship Door Controller, Sparse Diagram

plus a little logic to decide what state change to make as requests are received.

Comment. The same design of controller would be needed for the doors on a torpedo hatchway in a submarine.

Many controllers need similarly modest amounts of memory and logic. They are more readily understood as simple devices, specified by means of a graph, rather than as general purpose computers.

Similar devices are also very useful as a front end in many programs, where they are used to partition the input into meaningful tokens. For example, a compiler, which is a program that processes input programs so they can then be executed, needs to partition the input into keywords, variables, numbers, operators, comments, etc.

For example, we could imagine recognizing the keywords *if, then, else, end* with the help of the following device or procedure, represented as a graph.

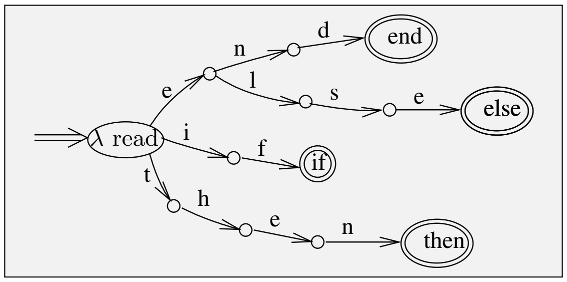


Figure 2.4: Keyword Recognizer

The automata always has at most one active vertex. Initially, the active vertex is the *start* vertex (the vertex labeled "λ read" in this example). On reading the next input character, the automata traverses the edge, if any, exiting the active vertex and labeled by this character; the vertex reached becomes the new active vertex. In other words, given an input string s, the automata follows the path whose concatenated edge labels form the string s. If at some point there is no edge to follow, then the computation fails; otherwise, if the active vertex when the computation finishes is a vertex represented by a double circle, then the computation has recognized a keyword; if is finishes at some other vertex, then the string read is not a keyword. For example, on reading the input string *if*, the finite automata goes from the start vertex to the vertex labeled *if*. Note that it is conventional for the start vertex, which need not be named *start*, to be indicated in the graph by means of a double arrow.

There are more details to handle in a compiler. In particular, the compiler has to be able to decide when it has reached the end of a keyword rather than being in the middle of a variable (e.g.

ended) whose name begins with the keyword. But these essentially straightforward and tedious, albeit important, details are not the focus here.

Variables are another collection of strings that compilers need to recognize. Let us suppose variables consist of all strings of (lower-case) letters other than keywords, and to simplify the illustration below let us suppose there is just one keyword: *if*. Then the following procedure, shown in Figure 2.5, will recognize variables.

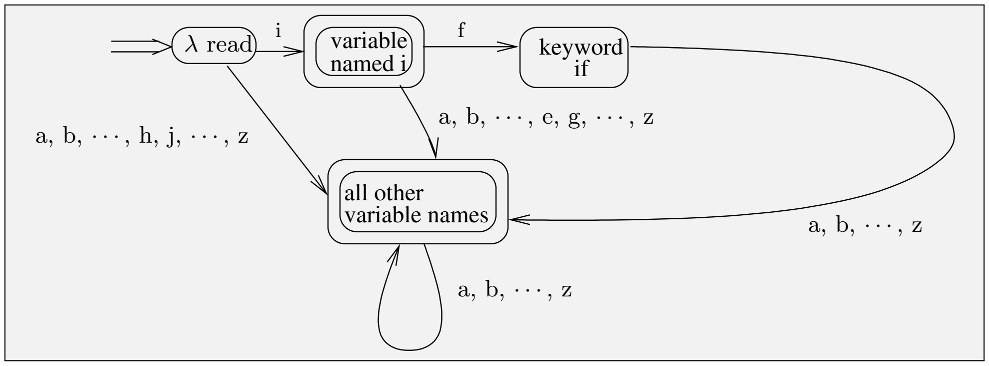


Figure 2.5: Variable Recognizer

Notice that the graph has a self-loop; also, some of its edges have more than one label (alternatively, we could think of this as a collection of parallel edges, each with a distinct label). Again, to use the variable recognizer, the compiler follows the path spelled out by the input. Any input other than the word *if* leads to a vertex represented as a double circle and hence represents a variable.

2.2.1 A Precise Specification

We need to introduce a little notation and terminology.

By convention, a finite automaton is named M, or M_1, M_2, $\cdots$ if there are several. As we shall see subsequently, other devices will also be named by M. A finite automaton is a directed graph plus a finite alphabet Σ in which:

- One vertex is designated as the start vertex.
- A subset of the vertices, conventionally called F, form the collection of *Recognizing* or *Final* vertices.
- Each edge is labeled by a character of Σ.
- Each vertex has $|\Sigma|$ outgoing edges, one for each character in Σ.

In the corresponding drawing of the automaton, the start vertex is indicated by a double arrow, and the vertices in F are indicated by drawing them with double circles.

One way to think about M's processing is procedurally. At any point in time, M will have one *currently reached* vertex, or *current* vertex for short. At the start of the computation, before any of its input u has been read, the current vertex is the start vertex. In general, having read a portion u of its input, the current vertex is the vertex p it reaches on reading string u. M then reads the next character of its input, a say, and follows the edge labeled a leaving vertex p, which brings it to vertex q. q is now its current vertex. M will read its input u character by character, advancing from vertex to vertex as described above, until all of u is read, which brings it to some last vertex r, say. Then u is recognized exactly if r is in the recognizing or final set of vertices, F. We call r the *destination* vertex for string u, i.e. the vertex the automata reaches after reading all of u.

We can also think of this computation as causing M to follow a path P, where the concatenation of the edge labels on P form the string u. We think of u as P's label; then M's computation can be viewed as finding the end vertex of the path starting at the start vertex and labeled by u. For example, in the Keyword Recognizer (see Figure 2.4), the input *if* specifies the path from the vertex *start* to the vertex named *if*. If the end of the path is a vertex in F then M *recognizes* or *accepts* u; otherwise M *rejects* u. Continuing the example, input *if* is recognized by the automaton as the vertex *if* is in F.

Thus we can view M as partitioning the set of all possible strings into those it recognizes and those it rejects. The set of strings recognized by M is often called the *language* recognized or accepted by M, and is sometimes written as $L(M)$ or L for short. The topic we will study in this chapter is what sorts of collections of strings finite automata can recognize.

Conventionally, the vertices of a finite automata are called its *states* and are written as $Q = \{q_1, q_2, \cdots, q_r\}$ rather than $V = \{v_1, v_2, \cdots, v_r\}$. The index n tends to be reserved for the length of the input string. However, we will seek to provide more descriptive specifications for the vertices in our examples.

For simplicity in drawing we replace multiple parallel edges by a single edge with multiple labels as shown in Figure 2.6.

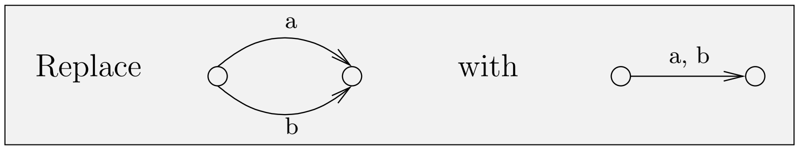


Figure 2.6: Simplifying Figures of Finite Automata

Also, it is often convenient to omit a “sink” vertex (a non-recognizing vertex which cannot be left); see Figure 2.7.

Note the descriptive specifications being used for the vertices. A description for a vertex p specifies the strings u that cause the automata to have destination p, the vertex reached after reading all of u when starting from the automata's start vertex. For example, in Figure 2.7, the vertex labeled ‘ye’ is the destination on reading the input string ‘ye’, but it is the destination for no other input string.

Finally, it's helpful to have a notation for representing labeled edges. We write $\delta(p, a) = q$ to mean that there is an edge labeled a from vertex p to vertex q; a more active interpretation is that starting at vertex (state) p, on reading a the automata goes to vertex q. In other words the *destination* on reading a when at vertex p is vertex q. Conventionally, δ is called the *transition* function. We will also refer to the labeled edge as a *transition rule*, because an edge (p, q) labeled by an a tells the finite automata what to do if the character a is read when it is at vertex p: namely to transit to vertex q.

It is also useful to extend the destination (or transition) function to the reading of strings;

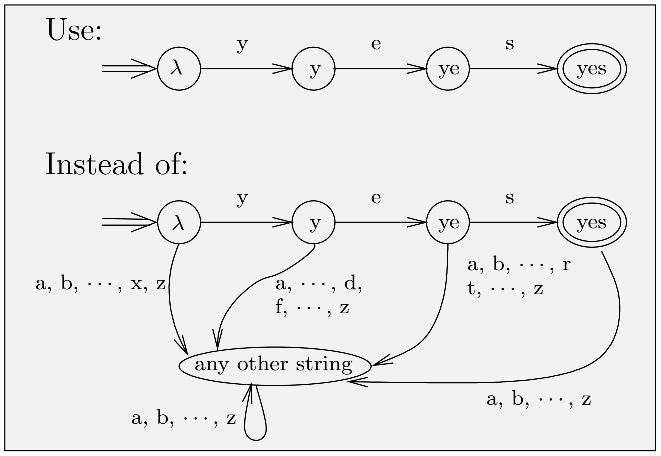


Figure 2.7: Omitting the Sink Vertex

this is denoted by δ^*. $\delta^*(p, u) = r$ means that vertex r is the destination on reading string u starting at vertex p. For example, in the Variable Recognizer (see Figure 2.5), $\delta^*(\text{“}\lambda \text{ read”}, it) =$ “all other variable names”.[2] Observe that $\delta^*(p, \lambda) = p$: reading the empty string, i.e. reading no characters, leaves the currently reached vertex unchanged.

Mathematical notation. A finite automata M can be written as $M = (Q, \Sigma, \delta, \text{start}, F)$, where Q is its set of vertices, start $\in Q$ is its start vertex, $F \subseteq Q$ is its set of recognizing vertices, Σ is the input alphabet, and δ specifies its edges and their labels.

Some Particular Languages

- $L(M) = \emptyset$ — This is the empty language, the language with no strings in it. So the finite automata M recognizes nothing.
- $L(M) = \{\lambda\}$ — This is the language comprising the single string λ. Note that $\emptyset \neq \{\lambda\}$; these two languages are distinct.

2.2.2 Examples of Automata

M_1, shown in Figure 2.8, recognizes the following strings: those strings with at least one a and an even number of b's following the last a.

M_2, shown in Figure 2.9, recognizes the set $L(M_2) = \{x \mid x \in \{a, b\}^* \text{ and } x \text{ ends with an } a\}$.

M_3, shown in Figure 2.10, recognizes the complement of the language recognized by M_2.

M_4, shown in Figure 2.11, keeps a running count mod 3 of the sum of the characters in its input that it has read, where the input alphabet $\Sigma = \{0, 1, 2\}$; it recognizes a string if the corresponding total is 0 mod 3.

Suppose that we want to maintain the above running count, but mod k, where the input alphabet is $\Sigma = \{0, 1, \cdots, k-1\}$. This can be done with a k-vertex automata. The automata has vertex set $Q = \{q_0, q_1, \cdots, q_{k-1}\}$; q_i will be its currently reached vertex if the running sum equals

[2]When a name or string includes spaces we will often put quote marks around it to avoid ambiguity.

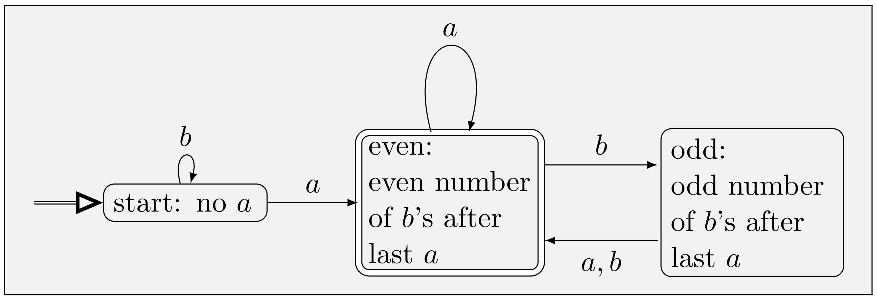

start: no a
even: even number of b's after last a
odd: odd number of b's after last a
a
b
a
b
a, b

Figure 2.8: Example Automata M_1

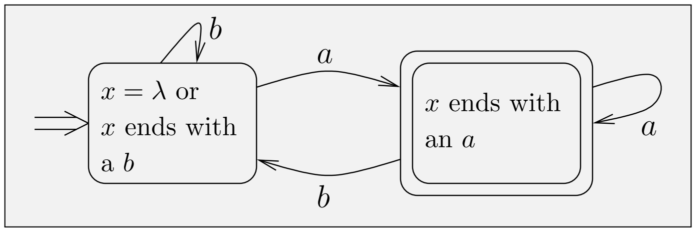


Figure 2.9: Example Automata M_2

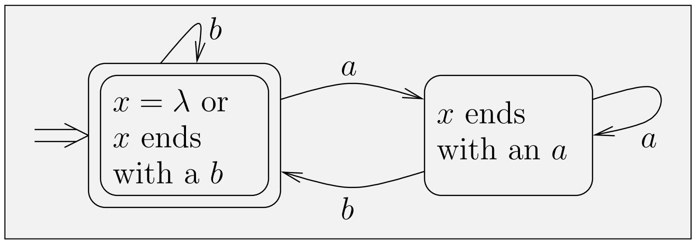


Figure 2.10: Example Automata M_3: $L(M_3) = \{x \mid x \text{ is the empty string or } x \text{ ends with a } b\}$

$i \bmod k$. It immediately follows that the start vertex is vertex q_0 and the recognizing vertex set is $F = \{q_0\}$. The transition function is also immediate: if the current running sum equals h (mod k) and an i is the next input character read, then the new running sum is $j = h + i \bmod k$; so the current vertex changes from q_h to q_j. We write $\delta(q_h, i) = q_j$ where $j = h + i \bmod k$. (See Figure 2.12.)

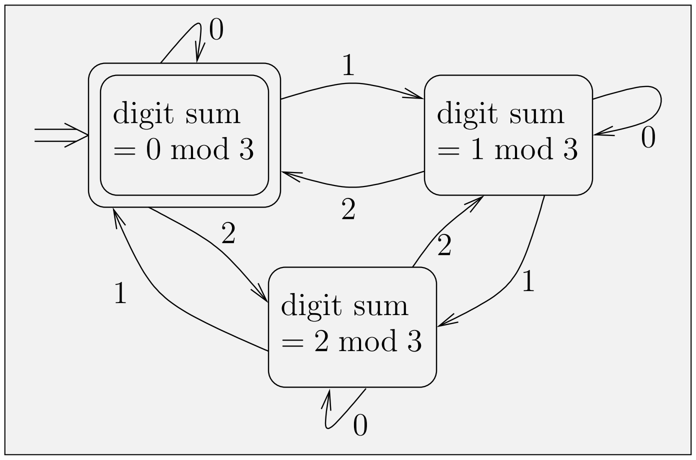


Figure 2.11: Example Automata M_4: $L(M_4) = \{x \mid \sum_{i=1}^{|x|} x_i = 0 \bmod 3,\ \text{where } x = x_1x_2\cdots x_{|x|}\}$. Note that $\lambda \in L(M_4)$.

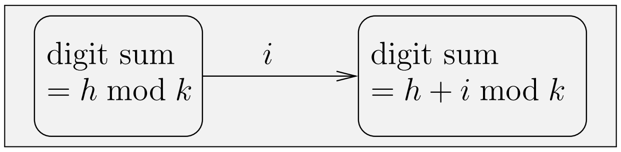


Figure 2.12: Example Automata M_5: The Transition Function

2.2.3 Designing Automata

The first issue is to decide what has to be remembered as the input is read. The easiest thing, it might seem, is to remember the whole string. But, in general, one cannot do this as the automata has only finite memory.

For example, with $\Sigma = \{a, b\}$, consider recognizing those strings with an odd number of b's. It suffices to keep track of whether an even or an odd number of b's have been read so far. This leads to the transition diagram shown in Figure 2.13(a). To obtain the full automata, it's enough to note that at the start zero b's have been read (an even number), and the recognizing vertex corresponds to an odd number of b's having been read; see Figure 2.13(b).

Next, consider the language $L(M) = \{w \mid w \text{ contains at least one } a\}$, with $\Sigma = \{a, b\}$. The natural vertices correspond to "no a read" and "at least one a read". The transitions are now evident and are shown in Figure 2.14.

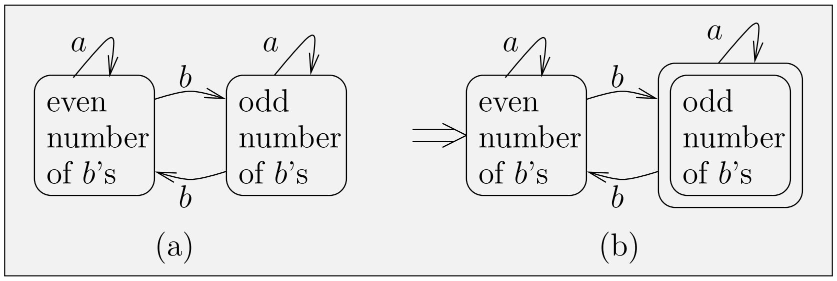


Figure 2.13: Automata recognizing strings with an odd number of b's

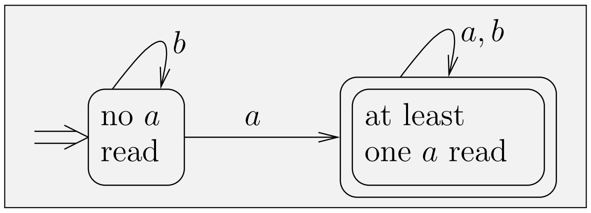


Figure 2.14: Automata recognizing strings with at least one a

Finally, consider the language $L(M) = \{w \,|\, w \text{ contains at least two } b\text{'s}\}$, with $\Sigma = \{a, b\}$. The natural three vertices correspond to: "no b read", "one b read", and "at least two b's read". Again, the transitions are evident and are shown in Figure 2.15.

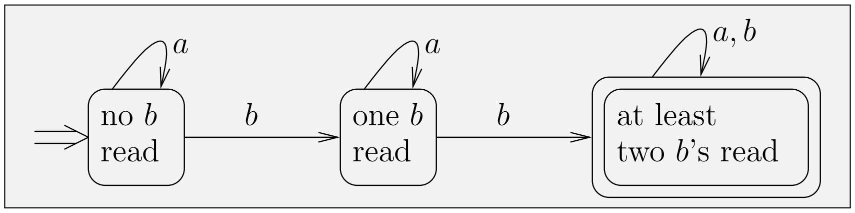


Figure 2.15: Automata recognizing strings with at least two b's

The vertex descriptors help to check that the automata is doing what it is supposed to do. In particular, when designing an automata, you should check that the edge labels and the descriptors are consistent.

More specifically, suppose that (p, q) is an edge with label a. Let S_p be p's descriptor and let S_q be q's. e.g. in Figure 2.8, the start vertex descriptor is the set of all strings with no a's. i.e. $\{\lambda, b, bb, bbb, \cdots\}$. Then you want to confirm that for each string $s \in S_p$, string $sa \in S_q$; i.e. appending an a to a string in S_p always yields a string in S_q.

This may seem very difficult to do, but in fact it tends to be straightforward. For example, consider Figure 2.8 again, and the edge (start, even) labeled a. We need to confirm that the descriptor 'even number of b's after last a' includes all strings of the form 'no a' (strings in S_{start}) followed by one a. Clearly, there are zero b's after the last (just read) a, and zero is an even number of b's. Of course it would be rather tedious to write out many such checks; one just wants to do them in one's head. However, it is all too easy to assume one has got the design right and not bother with the checks; this is how many simple mistakes are overlooked.

There are three other very simple checks.

1. When nothing has been read, the automata is at the start vertex. Consequently, the empty string must be in the descriptor for the start vertex.

2. Each string causes exactly one vertex to be reached. Consequently, the descriptors for distinct vertices are disjoint. In addition, the union of the descriptors is the set of all strings.

3. The union of the sets of strings specified by the descriptors for the set of recognizing vertices specifies exactly those strings that you want the automata to recognize. More formally, let $F = \{q_1, q_2, \cdots, q_l\}$; then $S_{q_1} \cup S_{q_2} \cup \cdots \cup S_{q_l}$ is the set of strings recognized by the automata.

In Section 2.6 we will show that performing the full set of checks ensures the correctness of the automata.

2.2.4 Recognizing More Complex Sets

We consider sets (languages) built up by means of the following three operations: union, concatenation, and star. Recall that these are called the *Regular Operations* (see Section 2.1). As we will see, if the sets A and B can be recognized by finite automata, then so can their union, concatenation, and the "star" of each set.

Recognizing $A \cup B$ The approach is natural: we use a device $M_{A\cup B}$ that comprises two finite automata: one, M_A, that recognizes A, and one, M_B, that recognizes B. The device $M_{A\cup B}$ runs M_A and M_B in parallel (simultaneously) on its input and recognizes the input exactly if, on reading this input, the destination in at least one of M_A and M_B is a recognizing vertex (state).

The only challenge is to show how to have $M_{A\cup B}$ be a finite automata. To describe this we need some more notation. Let $M_A = (Q_A, \Sigma, \delta_A, start_A, F_A)$, $M_B = (Q_B, \Sigma, \delta_B, start_B, F_B)$, and $M_{A\cup B} = (Q_{A\cup B}, \Sigma, \delta_{A\cup B}, start_{A\cup B}, F_{A\cup B})$. We illustrate the construction in Figure 2.16 for the pair of languages $A = \{\text{even length strings}\}$ and $B = \{w \mid \text{all } a\text{'s in } w \text{ precede all the } b\text{'s in } w\}$.

Consider what happens on input *aaba*. M_A and M_B go through the following sequence of pairs of vertices (the first vertex in the pair is M_A's vertex, the second is M_B's: $(p_1, q_1), (p_2, q_1), (p_1, q_1), (p_2, q_2)$, (p_1, q_3). Clearly, the information $M_{A\cup B}$ needs to record is the current pair of vertices reached by M_A and M_B, respectively. Thus, for this example, $M_{A\cup B}$ will need to have 6 vertices, one vertex for each possible pair (p_i, q_j). The edges in $M_{A\cup B}$ need to correspond to the joint moves made by M_A and M_B. For example, there is an edge labeled a from p_2 to p_1, and an edge labeled a from q_2 to q_3; thus in $M_{A\cup B}$ there will be an edge labeled a from (p_2, q_2) to (p_1, q_3). The remaining edges are obtained similarly, as shown in Figure 2.16. Clearly, the start vertex of $M_{A\cup B}$ is the vertex which corresponds to the pair of start vertices in M_A and M_B, respectively, namely, (p_1, q_1). To see how to identify the recognizing vertices in $M_{A\cup B}$ we argue as follows. A string s should be recognized by $M_{A\cup B}$ exactly if $s \in A$ or $s \in B$ (or both). The first of these occurs if the destination vertex p in M_A on input s is in F_A, and the second occurs if the destination vertex q in M_B on input s is in F_B. In other words, the (vertex of $M_{A\cup B}$ corresponding to the) vertex pair (p, q) needs to satisfy the condition that either $p \in F_A$ or $q \in F_B$ (or both). In our example, we see that every vertex pair apart from (p_2, q_3) satisfies this condition.

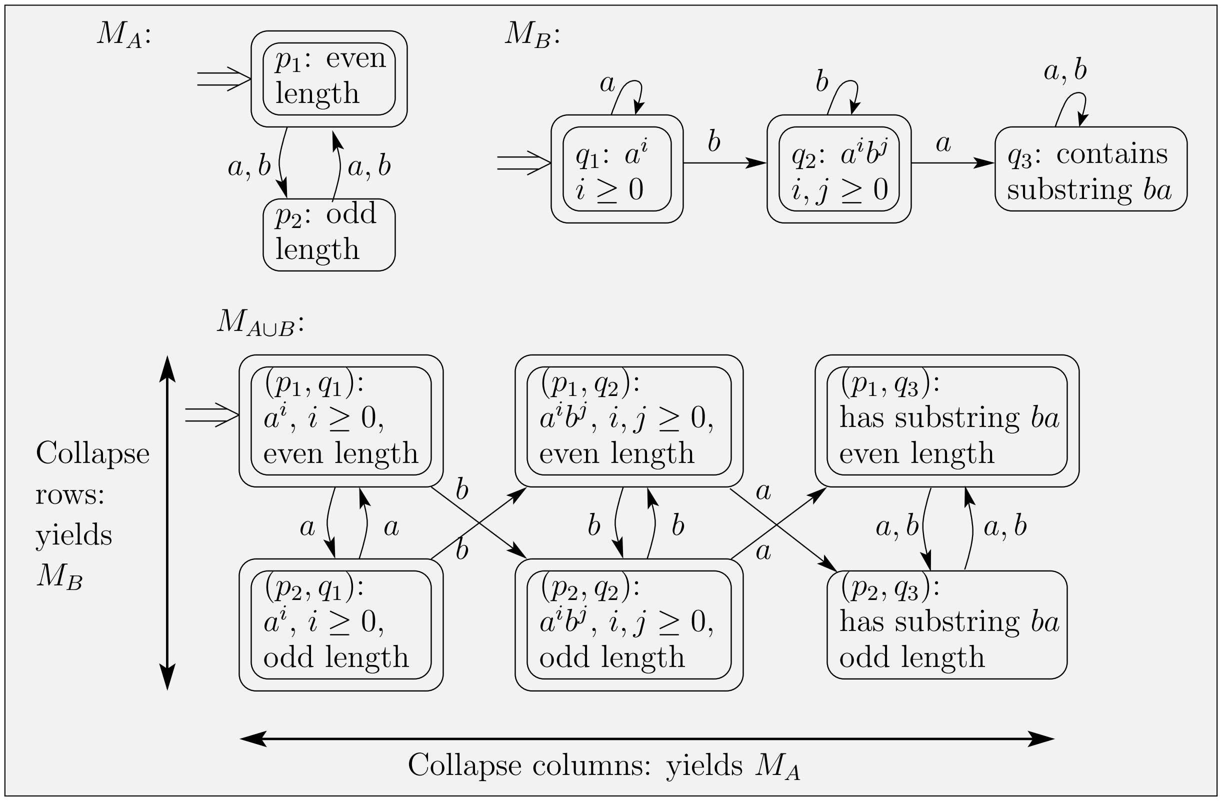

M_A:
p_1: even length
p_2: odd length
a, b
a, b
M_B:
q_1: a^i $i \geq 0$
q_2: $a^i b^j$ $i, j \geq 0$
q_3: contains substring ba
a
b
a, b
$M_{A \cup B}$:
(p_1, q_1): a^i, $i \geq 0$, even length
(p_1, q_2): $a^i b^j$, $i, j \geq 0$, even length
(p_1, q_3): has substring ba even length
(p_2, q_1): a^i, $i \geq 0$, odd length
(p_2, q_2): $a^i b^j$, $i, j \geq 0$, odd length
(p_2, q_3): has substring ba odd length
Collapse rows: yields M_B
Collapse columns: yields M_A

Figure 2.16: Automata $M_{A \cup B}$

Notice that in our example, if we overlay the columns this yields automata M_A; in other words, looking at just the first component of $M_{A\cup B}$'s vertex pairs on input w will yield the same actions as for M_A on input w. Similarly, overlaying the rows yields automata M_B.

Now we turn to the construction for general regular languages A, B. The vertices of $M_{A\cup B}$ are pairs, where each pair consists of a vertex of M_A and a vertex of M_B; we call these *vertex-pairs* for clarity. The purpose of the M_A component in the vertex-pair is to simulate the computation of M_A, and similarly that of the M_B component is to simulate M_B's computation. Formally, we define $Q_{A\cup B} = Q_A \times Q_B$: each vertex-pair of $M_{A\cup B}$ is a 2-tuple, consisting of a vertex of M_A and a vertex of M_B (in that order). The idea is that:

Assertion 2.2.1. *$M_{A\cup B}$ will have destination $q = (q_A, q_B)$ on reading input w exactly if M_A has destination q_A and M_B has destination q_B on reading input w.*

The definitions of $start_{A\cup B}$, $\delta_{A\cup B}$, and $F_{A\cup B}$ are now immediate.

- The start vertex-pair of M is the pair consisting of the start vertices of M_A and M_B: $start_{A\cup B} = (start_A, start_B)$.

- An edge of $M_{A\cup B}$, labeled a, connects a vertex-pair $p = (p_A, p_B)$ to vertex-pair $q = (q_A, q_B)$, exactly if p_A is connected to q_A by an a-labeled edge in M_A and p_B is connected to q_B by an a-labeled edge in M_B; in other words, $\delta_{A\cup B}(p, a) = \delta_{A\cup B}((p_A, p_B), a) = (\delta_A(p_A, a), \delta_B(p_B, a)) = (q_A, q_B) = q$.

- The recognizing vertex-pairs of $M_{A\cup B}$ are those pairs that include either a recognizing vertex of M_A or a recognizing vertex of M_B (or both): $F = (F_A \times Q_B) \cup (Q_A \times F_B)$.

Lemma 2.2.2. *$M_{A\cup B}$ recognizes the language $A \cup B$, $L(M_{A\cup B}) = A \cup B$ for short.*

Proof. We need to do two things:

1. Show that if w is recognized by $M_{A\cup B}$, i.e. if $w \in L(M_{A\cup B})$, then $w \in A \cup B$, and

2. show that if $w \in A \cup B$, then $w \in L(M_{A\cup B})$.

The first substep is to verify Assertion 2.2.1. But this is easy to see: consider the computation of $M_{A\cup B}$ on input w, but only pay attention to what is happening to the first component of its vertex-pair: this is identical to the computation of M_A on input w. Likewise, if we pay attention only to the second component of the vertex-pair, we see that the computation is identical to that of M_B on input w. In particular, $M_{A\cup B}$ has destination vertex-pair $q = (q_A, q_B)$ on reading input w exactly if M_A has destination q_A and M_B has destination q_B after reading input w, which is what Assertion 2.2.1 states.

Next, we show (1). So suppose that $M_{A\cup B}$ recognizes w, that is, on input w, $M_{A\cup B}$ reaches a recognizing vertex-pair, $q = (q_A, q_B)$ say. By Assertion 2.2.1, $M_{A\cup B}$ has destination $q = (q_A, q_B)$ on input w exactly if M_A has destination q_A on input w and M_B has destination q_B. Now, by definition, q is a recognizing vertex-pair of $M_{A\cup B}$ exactly if $q \in (F_A \times Q_B) \cup (Q_A \times F_B)$. Consider the case that $q = (q_A, q_B) \in F_A \times Q_B$; then $q_A \in F_A$, that is M_A has as destination a recognizing vertex on input w, which happens exactly if $w \in A$. Similarly, in the case that $q = (q_A, q_B) \in Q_A \times F_B$, $q_B \in F_B$, that is M_B has as destination a recognizing vertex on input w, which happens exactly if $w \in B$. This shows that if $M_{A\cup B}$ recognizes input w, then either $w \in A$ or $w \in B$ (or possibly both); this means that $w \in A \cup B$, which is the statement in (1) above.

Finally, we show (2). If $w \in A \cup B$, then either $w \in A$ or $w \in B$ (or possibly both). Consider the case that $w \in A$. Then w is recognized by M_A. That is, on input w, M_A has a recognizing vertex as its destination, q_A say, with $q_A \in F_A$. At the same time, M_B, on input w, has some vertex $q_B \in Q_B$ as its destination. By Assertion 2.2.1, $M_{A\cup B}$, on input w, has as its destination the vertex pair $(q_A, q_B) \in F_A \times Q_B$. So $M_{A\cup B}$ recognizes w since $(q_A, q_B) \in F_A \times Q_B \subset (F_A \times Q_B) \cup (Q_A \times F_B) = F_{A\cup B}$. An essentially identical argument shows that if $w \in B$ then $M_{A\cup B}$ again recognizes w. In sum, if $w \in A \cup B$, then $M_{A\cup B}$ recognizes w, that is $w \in L(M_{A\cup B})$. □

Corollary 2.2.3. *Let A and B be languages recognized by finite automata. Then there is another finite automata recognizing the language $A \cup B$.*

Recognizing $\boldsymbol{A \circ B}$ Again, we suppose we have finite automata M_A recognizing A and M_B recognizing B. We then build a device M to recognize $A \circ B$, or AB for short. $w \in AB$ exactly if we can divide w into two parts, u and v, with $u \in A$ and $v \in B$. This immediately suggests how M will try to use M_A and M_B. First it runs M_A on substring u and then it runs M_B on substring v. The requirements are that M_A recognize u and M_B recognize v; that is, on input u, M_A goes from its start vertex to a recognizing vertex, and on input v, M_B also goes from its start vertex to a recognizing vertex. The difficulty is for M to know when u ends, for there may be multiple possibilities. For example, consider the languages $A = \{\text{strings of one or more } a\text{'s}\}$ and $B = \{\text{strings of one or more } b\text{'s}\}$. On the string *aabb* it would not be correct to switch from simulating M_A to simulating M_B after reading a single a, while it would be correct on the string *abb*. The solution is to keep track of all possibilities. We do not explore this further at this point, as this is more readily understood using the technique of nondeterminism, which is the topic of the next subsection.

Likewise, we defer the description of how to recognize A^* to the next subsection.

2.3 Nondeterministic Finite Automata

Non-deterministic Finite Automata, NFAs for short, are a generalization of the machines we have already defined, which are often called *Deterministic Finite Automata* by contrast, or DFAs for short. The reason for the name will become clear later.

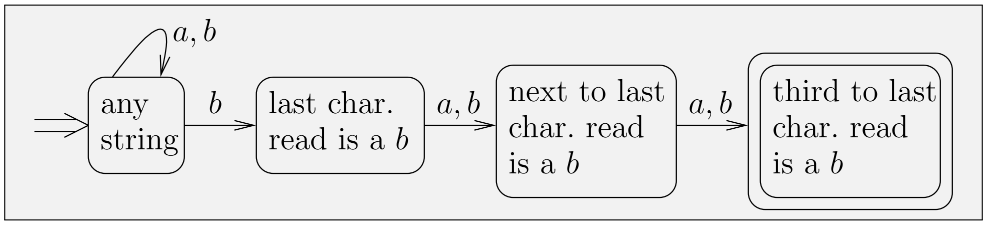
As with a DFA, an NFA is simply a graph with edges labeled by single letters from the input alphabet Σ. There is one structural change.

> For each vertex v and each character $a \in \Sigma$, the number of edges exiting v labeled with a is unconstrained; it could be 0, 1, 2 or more edges.

This obliges us to redefine what an automaton M is doing, given an input x. Quite simply, M on input x determines all the vertices at the end of paths labeled x that begin at the start vertex, *start*. If any of these destination vertices is in the set of Recognizing or Final vertices then M is defined to recognize x. Another way of looking at this is that M recognizes x exactly if there is some path (and possibly more than one) labeled x, going from *start* to a vertex (state) $q \in F$; such a path is called a *recognizing* or *accepting* path.

Example 2.3.1. Let $A = \{w \,|\, \text{the third to last character in } w \text{ is a "}b\text{"}\}$. The machine M with $L(M) = A$ is shown in Figure 2.17.

Note that on input *abbb* all four vertices are destination vertices, whereas on input *abab* the second from rightmost vertex is not a destination vertex.



a, b
any string
b
last char. read is a b
a, b
next to last char. read is a b
a, b
third to last char. read is a b

Figure 2.17: NFA recognizing $\{w \,|\, \text{the third to last character in } w \text{ is a } b\}$.

As we will see later, the collection of languages recognized by NFAs is exactly the collection of languages recognized by DFAs. That is, for each NFA N there is a DFA M with $L(M) = L(N)$. Thus NFAs do not provide additional computational power. However, they can be more compact and easier to understand, which makes them quite useful.

Indeed, the language from Example 2.3.1 is recognized by the DFA shown in Figure 2.18.

It is helpful to consider how one might implement an NFA N. One way is to maintain a bit vector over the vertices, so as to record which vertices are possible current destinations. That is, suppose that on reading input w, N can end up at any of the vertices in set R, say. The set S of possible destination vertices on reading a further character a is obtained by following all the edges labeled a that exit any of the vertices in R. So in the automata of Figure 2.17, the set of possible destination vertices on reading input b is {"any string", "last char read is a b"}, and the set of possible destination vertices on reading ba is {"any string", "next to last char read is a b"}.

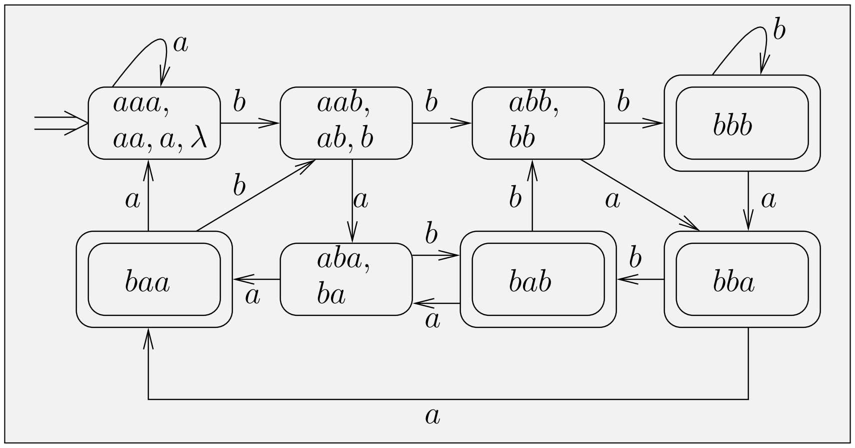


Figure 2.18: DFA recognizing $\{w \,|\, \text{the third to last character in } w \text{ is a } b\}$. By naming the last three characters or all the characters in the string, each vertex specifies the strings for which it is the destination vertex.

Definition 2.3.2 (NFA Recognition). *An NFA M recognizes input string w if there is a path from M's start vertex to a recognizing vertex (state) such that the labels along the edges of the path, when concatenated, form the string w. That is, were we to follow this path, reading the edge labels as we go, the string read would be exactly w; we say this path has label w. We call such a path a* recognizing *or* accepting *path for w.*

The implementation described above keeps track, in turn, of all possible endpoints of paths with labels $\lambda, w_1, w_1w_2, \ldots, w_1w_2\ldots w_n$, where $w = w_1w_2\ldots w_n$; to recognize w, one needs to have at least one path with label w leading to a recognizing vertex.

To define NFAs formally, we need to redefine the destination (transition) functions δ. Now δ takes as its arguments a set of vertices (states) $R \subseteq V$ and a character $a \in \Sigma$ and produces as output another set of vertices (states) $S \subseteq V$: $\delta(R, a) = S$. The meaning is that the set of vertices (states) reachable from R on reading a is exactly the set of vertices (states) S.

δ^* is also redefined. $\delta^*(R, w) = S$ means that the set of vertices (states) S are the possible destinations on reading w when starting from a vertex (state) in R. So, in particular, w is recognized by M, $w \in L(M)$, exactly if $\delta^*(\{start\}, w) \cap F \neq \emptyset$; i.e., when starting at vertex *start*, on reading w, at least one destination vertex (state) is a recognizing or final vertex (state).

δ is often defined formally as follows: $\delta : 2^Q \times \Sigma \to 2^Q$. In this context, 2^Q means the collection of all possible subsets of Q, and as Q is the set of vertices of the NFA, δ does exactly what it should. Its inputs are (i) a set of vertices (states), that is one of the elements of the collection 2^Q, and (ii) a character in Σ; its output is also a set of vertices (states).

Next, we add one more option to NFAs: edges labeled with the empty string, as shown in Figure 2.19. We call these edges λ-edges for short. The meaning is that if vertex (state) p is reached, then vertex (state) q can also be reached without reading any more input. Again, as we shall see later, the same languages as before are recognized by NFAs with λ-edges and by DFAs; however, the λ-edges are very convenient in creating understandable machines. The meanings of the destination functions δ and δ^* are unchanged.

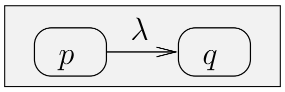


Figure 2.19: λ label on an edge.

Example 2.3.3. Let A and B be languages over the alphabet Σ that are recognized by NFAs M_A and M_B, respectively. Then the NFA shown in Figure 2.20 recognizes the language $A \cup B$.

Its first step, prior to reading any input, is to go to the start vertices for machines M_A and M_B. Then the computations in these two machines are performed simultaneously. The set of recognizing vertices for M is the union of the recognizing vertices for M_A and M_B; thus M reaches a recognizing vertex on input w exactly if at least one of M_A or M_B reaches a recognizing vertex on input w. In other words $L(M) = L(M_A) \cup L(M_B)$.

Deterministic vs. Nondeterministic Another way of viewing the computation of an NFA M on input x is that the task is to find a path labeled x from the start vertex to a recognizing vertex if there is one, and this is done correctly by (inspired) guessing. This process of correct guessing is called *non-deterministic* computation. Correspondingly, if there is no choice or uncertainty in

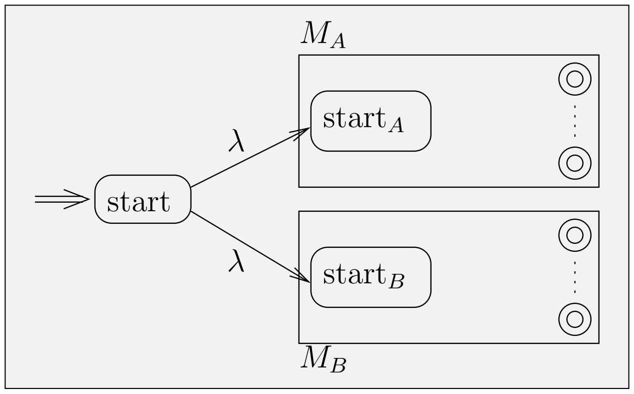


Figure 2.20: An NFA recognizing language $A \cup B$

the computation, it is said to be *deterministic*. This is not an implementable view; it is just a convenient way to think about what nondeterminism achieves.

2.3.1 Closure Properties

Definition 2.3.4. *A language is said to be* regular *if it can be recognized by an NFA.*

The regular languages obey three closure properties: if A and B are regular, then so are $A \cup B$, $A \circ B$ and A^*. We have already seen a demonstration of the first of these. We now give a simpler demonstration of the first property and then show the other two.

First, the following technical lemma is helpful.

Lemma 2.3.5. *Let M be an NFA. Then there is another NFA, N, which has just one recognizing vertex, and which recognizes the same language as M: $L(N) = L(M)$.*

Proof. The idea is very simple: N is a copy of M with one new vertex, *new_recognize*, added; *new_recognize* is N's only recognizing vertex. λ-edges are added from the vertices that were recognizing vertices in M to *new_recognize*, as illustrated in Figure 2.21.

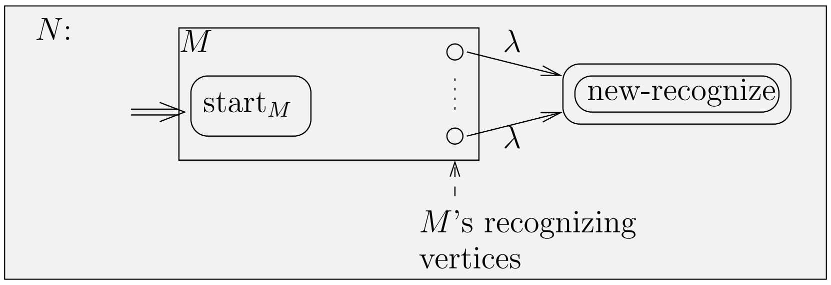


Figure 2.21: NFA N with a single recognizing vertex

Recall that a recognizing path in an NFA goes from its start vertex to a recognizing vertex. It is now easy to see that M and N recognize the same language. The argument has two parts.

First, any recognizing path in M, for string w say, can be extended by a single λ-edge to reach N's recognizing vertex, and thereby becomes a recognizing path in N; in addition, the edge addition leaves the path label unchanged (as $w\lambda = w$). It follows that if M recognizes string w then so does N. In other words, $L(M) \subseteq L(N)$.

Second, removing the last edge from a recognizing path in N yields a recognizing path in M having the same path label (for the removed edge has label λ). It follows that if N recognizes string w then so does M. In other words, $L(N) \subseteq L(M)$.

Together, these two parts yield that $L(M) = L(N)$, as claimed.

□

We are now ready to demonstrate the closure properties.

If A and B are regular then so is A ∪ B We show this using the NFA $M_{A\cup B}$ in Figure 2.22. $M_{A\cup B}$ is built using M_A and M_B, NFAs with single recognizing vertices recognizing A and B,

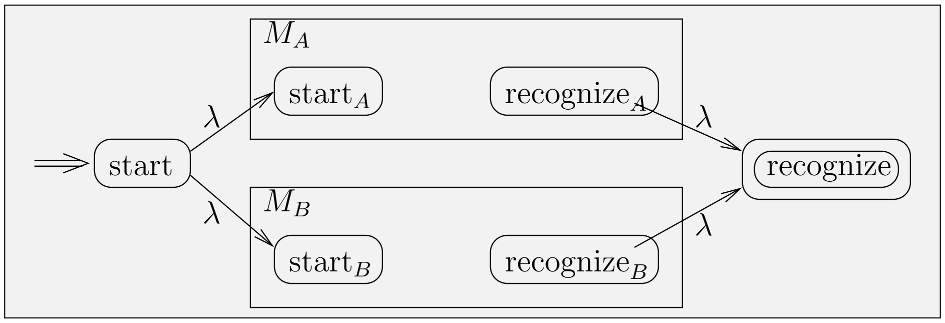


Figure 2.22: NFA $M_{A\cup B}$ recognizing $A \cup B$

respectively. It has an additional start vertex *start*, connected to M_A and M_B's start vertices by λ-edges, and an additional recognizing vertex *recognize*, to which M_A and M_B's recognizing vertices are connected by λ-edges.

As in the earlier construction, each recognizing path in $M_{A\cup B}$ corresponds to a recognizing path in one of M_A and M_B, and vice-versa, and thus $w \in L(M_{A\cup B})$ if and only if $w \in A \cup B$. We now show this more formally.

Lemma 2.3.6. *$M_{A\cup B}$ recognizes $A \cup B$.*

Proof. We show that $A \cup B = L(M_{A\cup B})$ by showing that $L(M_{A\cup B}) \subseteq A \cup B$ and that $A \cup B \subseteq L(M_{A\cup B})$.

To show that $L(M_{A\cup B}) \subseteq A \cup B$, it is enough to show that if $w \in L(M_{A\cup B})$ then $w \in A \cup B$ also. So let $w \in L(M_{A\cup B})$ and let P be a recognizing path for w in $M_{A\cup B}$. Removing the first and last edges of P, which both have label λ, yields a recognizing path in one of M_A or M_B, also having label w. This shows that if $w \in L(M_{A\cup B})$, then either $w \in L(M_A) = A$ or $w \in L(M_B) = B$ (of course, it might be in both, but this argument does not reveal this); in other words, if $w \in L(M_{A\cup B})$ then $w \in A \cup B$.

To show that $A \cup B \subseteq L(M_{A\cup B})$, it is enough to show that if $w \in A \cup B$ then $w \in L(M_{A\cup B})$ also. Now, if $w \in A \cup B$, then either $w \in A$ or $w \in B$ (or possibly both). Consider the case that

$w \in A$. Then there is a recognizing path P_A for w in M_A. Preceding P_A with the appropriate λ-edge from *start* and following it with the λ-edge to *recognize*, yields a recognizing path in $M_{A\cup B}$, also having label w. This shows that if $w \in L(M_A) = A$ then $w \in L(M_{A\cup B})$ also. Similarly, if $w \in B$ then $w \in L(M_{A\cup B})$ too. Together, this gives that if $w \in A \cup B$ then $w \in L(M_{A\cup B})$. $\square$

If A and B are regular then so is $\mathbf{A \circ B}$ We show this using the NFA M_{AB} displayed in Figure 2.23. M_{AB} is built using M_A and M_B, NFAs with single recognizing vertices, recognizing A and B,

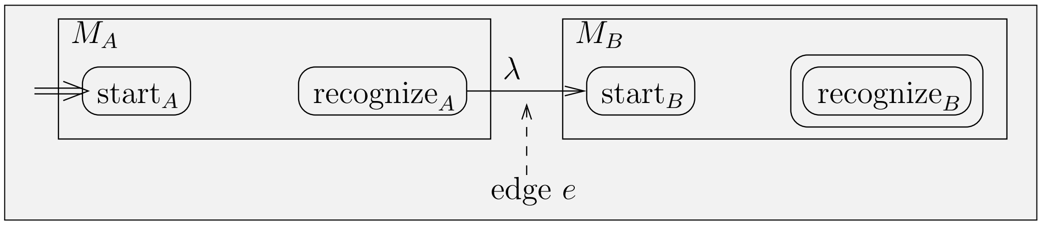


Figure 2.23: NFA M_{AB} recognizing $A \circ B$

respectively. M_{AB} comprises a copy of M_A plus a copy M_B, plus one additional edge. M_A's start vertex is also M_{AB}'s start vertex, and M_B's recognizing vertex is M_{AB}'s only recognizing vertex. Finally, the new edge, e, joins M_A's recognizing vertex to M_B's start vertex.

The idea of the construction is that a recognizing path in M_{AB} corresponds to recognizing paths in M_A and M_B joined by edge e. It then follows that $w \in L(M_{AB})$ if and only if w is the concatenation of strings u and v, $w = uv$, with $u \in A$ and $v \in B$. We now show this more formally.

Lemma 2.3.7. *M_{AB} recognizes $A \circ B$.*

Proof. We show that $L(M_{AB}) = A \circ B$ by showing that $L(M_{AB}) \subseteq A \circ B$ and that $A \circ B \subseteq L(M_{AB})$.

To show that $L(M_{AB}) \subseteq A \circ B$, it is enough to show that if $w \in L(M_{AB})$ then $w \in A \circ B$ also. So let $w \in L(M_{AB})$ and let P be a recognizing path for w in M_{AB}. Removing edge e from P creates two paths P_A and P_B, with P_A being a recognizing path in M_A and P_B a recognizing path in M_B. Let u and v be the path labels for P_A and P_B, respectively. So $u \in A$ and $v \in B$. As e is a λ-edge, $w = u\lambda v = uv$. This shows that $w \in A \circ B$.

To show that $A \circ B \subseteq L(M_{AB})$, it is enough to show that if $u \in A$ and $v \in B$ then $uv \in L(M_{AB})$. So let $u \in A$, $v \in B$, let P_A be a recognizing path for u in M_A, and let P_B be a recognizing path for v in M_B. Then form the path $P = P_A, e, P_B$ in M_{AB}; clearly, this is a recognizing path. Further, it has label $u\lambda v = uv$. So $uv \in L(M_{AB})$.

$\square$

If A is regular then so is $\mathbf{A}^*$ We show this using the NFA M_{A^*} in Figure 2.24.

M_{A^*} is built using M_A, an NFA with a single recognizing vertex recognizing A. M_{A^*} comprises a copy of M_A plus a new start vertex, plus two additional λ-edges, e and f. e joins M_{A^*}'s start vertex to M_A's start vertex, and f joints M_A's recognizing vertex to M_{A^*}'s start vertex. M_{A^*}'s start vertex is also its recognizing vertex.

The construction is based on the observation that removing all copies of e and f from a recognizing path in M_{A^*} yields subpaths, k of them say, each of which is a recognizing path in M_A. It

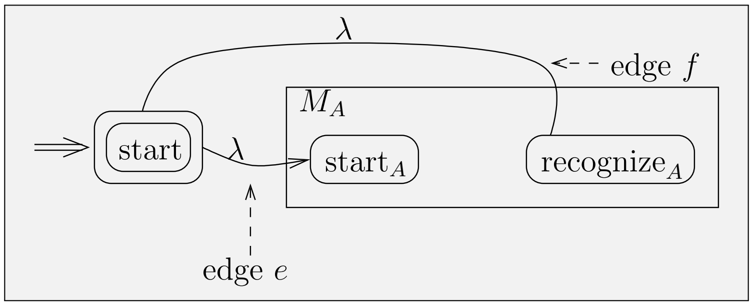


Figure 2.24: NFA M_{A^*} recognizing A^*

then follows that a string w is recognized by M_{A^*} if and only if it is the concatenation of k strings recognized by M_A, for some $k \geq 0$. Now we show this more formally.

Lemma 2.3.8. *M_{A^*} recognizes A^*.*

Proof. We show that $L(M_{A^*}) = A^*$ by showing that $L(M_{A^*}) \subseteq A^*$ and that $A^* \subseteq L(M_{A^*})$.

To show that $L(M_{A^*}) \subseteq A^*$, it is enough to show that if $w \in L(M_{A^*})$ then $w \in A^*$ also. So let $w \in L(M_{A^*})$ and let P be a recognizing path for w in M_{A^*}. Removing all instances of edges e and f from P creates k subpaths, for some $k \geq 0$, where each subpath is a recognizing path in M_A. Let the path labels on these k subpaths be $u_1, \cdots, u_k$, respectively. So $u_i \in A$, for $1 \leq i \leq k$. As e and f are λ-edges, $w = \lambda u_1 \lambda\lambda u_2 \lambda\lambda \cdots \lambda\lambda u_k \lambda = u_1 u_2 \cdots u_k$. Thus $w \in A^*$.

To show that $A^* \subseteq L(M_{A^*})$, it is enough to show that if $w \in A^*$ then $w \in L(M_{A^*})$. If $w \in A^*$ then $w = u_1 u_2 \cdots u_k$, with $u_1, u_2, \cdots, u_k \in A$, for some $k \geq 0$. As $u_i \in A$, there is a recognizing path P_i in M_A for u_i. Let P be the following path in M_{A^*}: $e, P_1, f, e, P_2, f, \cdots, e, P_k, f$ (for $k = 0$ we intend the path of zero edges). P is a recognizing path in M_{A^*}, and it has label $u_1 u_2 \cdots u_k = w$, as e and f are λ-edges. Thus $w \in L(M_{A^*})$.

□

2.3.2 Every regular language is recognized by a DFA

Let N be an NFA. We show how to construct a DFA M with $L(M) = L(N)$.

Recall that to implement the computation of an NFA N on an input w we keep track of the set of current destination vertices as w is read. Suppose that $S_0, S_1, \cdots S_n$ is the sequence of sets of destination vertices reached on reading $w = w_1 w_2 \cdots w_n$ character by character; i.e. S_i is the set of destination vertices for input $w_1 w_2 \cdots w_i$, and in particular S_0 is the set of vertices reachable without reading any input (i.e. on input λ). M will need to remember exactly this information: the set of current destination vertices in N. To do this, we make the vertices in M be the possible sets of destination vertices in N. To avoid confusion, henceforth we call the vertices in M supervertices. If N has q vertices, M has 2^q supervertices, one for each subset of N. In other words M's collection (set) of supervertices is the power set of Q, 2^Q.

The relation between the supervertices in M and the sets of vertices in N is specified by the following assertion.

Assertion 2.3.9. *On input w, N goes through the sequence of destination sets $S_0, S_1, S_2, \cdots, S_n$ if and only if M traverses the path $S_0, S_1, S_2, \cdots, S_n$.*

To achieve Assertion 2.3.9, we define the transition function δ for M to be identical to the transition function δ for N. Note that this specifies where M's edges go. More formally, if $U = \{u_1, u_2, \ldots, u_k\}$ is a supervertex of M, and in N $\delta(u_i, a) = W_i$, for $1 \leq i \leq k$, then in M, $\delta(U, a) = \cup_{1 \leq i \leq k} W_i$.

To make sure the assertion is correct initially, that is for $w = \lambda$, we set M's start supervertex to be the set of N's vertices that N can reach on input λ.

Finally, to ensure that $L(M) = L(N)$, we recall that N accepts input w if its destination set S on input w includes a vertex in F, the set of its recognizing vertices; accordingly, we define the recognizing supervertices of M to be those supervertices that include one or more of the recognizing vertices of N; i.e. R is a recognizing supervertex for M exactly if $R \cap F \neq \emptyset$.

We illustrate the construction in Figure 2.25.

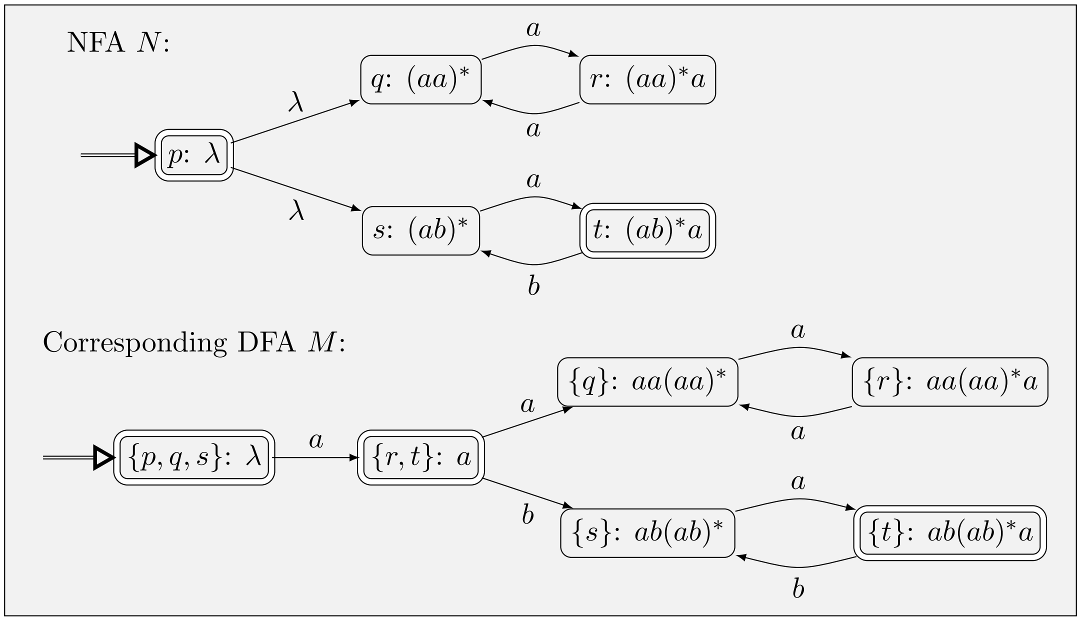


Figure 2.25: NFA and corresponding DFA. The strings that terminate at a vertex of the DFA, e.g. at the vertex labeled $\{p, q, s\}$, are exactly those strings that can terminate at all of the corresponding nodes in the NFA and no others, namely the vertices labeled p, q, and s. Continuing the example, the strings that terminate at the vertex labeled $\{r\}$ exclude the strings with destinations p, q, s, t in the NFA.

Lemma 2.3.10. *$L(M) = L(N)$.*

Proof. Clearly Assertion 2.3.9 is true, for N starts, prior to any reads, with destination set S_0, and M starts at supervertex S_0. Then, as the machines use the same transition function δ, the destination set that N reaches after reading each successive input character will be the destination supervertex for M after reading the same input.

It remains to consider which strings the two machines recognize. M recognizes input w if and only if on input w it reaches supervertex R where $R \cap F \neq \emptyset$; and N recognizes input w if and only if its set S of possible destinations satisfies $S \cap F \neq \emptyset$. But we have shown that $S = R$. So the two machines recognize the same collection of strings, that is $L(M) = L(N)$. $\square$

Implementation Remark The advantage of an NFA is that it may have far fewer vertices (states) than a DFA recognizing the same language, and thus use much less memory to store them. On the other hand, when running the machines, the NFA may be less efficient, as the set of reachable vertices had to be computed as the input is read, whereas in the DFA one just needs to follow a single edge. Which choice is better depends on the particulars of the language and the implementation environment.

2.4 Non Regular Languages

We turn now to a method for demonstrating that some languages are not regular. For example, as we shall see, $L = \{a^n b^n \mid n \geq 1\}$ is not a regular language. Intuition suggests that to recognize L we would need to count the number of a's in the input string; in turn, this suggests that any automata recognizing L would need an unbounded number of vertices. But how do we turn this into a convincing argument?

We use a proof by contradiction. So suppose for a contradiction that L were regular. Then there must be a DFA M that accepts L. M has some number k of vertices. Let's consider feeding M the input $a^k b^k$. Look at the sequence of $k+1$ vertices M goes through on reading a^k: *start* $= r_0, r_1, r_2, \cdots, r_k$, where some of the r_i's may be a repeated instance of the same vertex. This is illustrated in Figure 2.26.

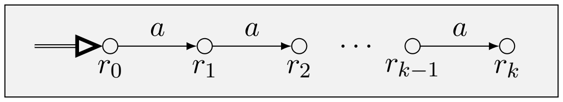


Figure 2.26: The path traversed on input a^k.

As there are only k distinct vertices, there is at least one vertex that is visited twice in this sequence, $r_h = r_j$, say, for some pair h,j, $0 \leq h < j \leq k$. This is shown in Figure 2.27.

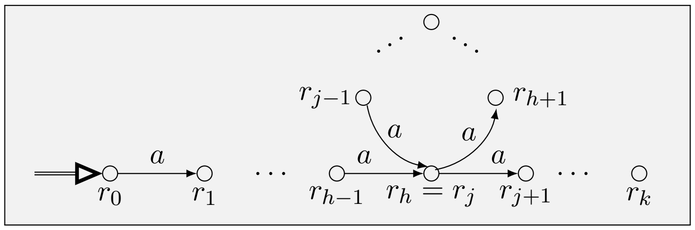


Figure 2.27: The path traversed on input a^k.

In fact we see that $r_{j+1} = r_{h+1}$ if $j+1 \leq k$, $r_{j+2} = r_{h+2}$ if $j+2 \leq k$, and so forth. But all we will need for our result is the presence of one loop in the path, so we will stick with the representation in Figure 2.27.

It is helpful to partition the input into four pieces: a^h, a^{j-h}, a^{k-j}, b^k. The first a^h takes M from vertex r_0 to r_h (to the start of the loop), the next a^{j-h} takes M from r_h to r_j (once around the loop), the final a^{k-j} takes M from r_j to r_k, and b^k takes M from r_k to a recognizing vertex, as shown in Figure 2.28.

What happens on input $a^h a^{j-h} a^{j-h} a^{k-j} b^k = a^{k+j-h} b^k$? The initial a^h takes M from r_0 to r_h, the first a^{j-h} takes M from r_h to $r_j = r_h$ (once around the loop), the second a^{j-h} takes M from

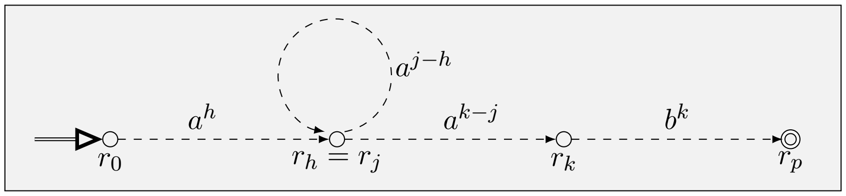


Figure 2.28: The path traversed on input $a^k b^k$.

r_h to r_j (around the loop again), the a^{k-j} takes M from r_j to r_k, and the b^k takes M from r_k to a recognizing vertex. So M accepts $a^{k+j-h}b^k$, which is not in L as $j - h > 0$. This is a contradiction, and thus the initial assumption, that L was regular, must be incorrect.

We now formalize the above approach in the following lemma.

Lemma 2.4.1. (Pumping Lemma) *Let L be a regular language. Then there is an integer $p = p_L \geq 1$, the* pumping length *for L, with the property that for each string s in L of length at least p, s can be* pumped*, that is s can be written as the concatenation of three substrings x, y, and z, that is $s = xyz$, and these substrings satisfy the following conditions.*

1. $|y| > 0$,
2. $|xy| \leq p$,
3. *For each integer $i \geq 0$, $xy^iz \in L$.*

Proof. The proof is very similar to the argument we just saw.

As L is regular there is a DFA M recognizing L. Now let p be the number of vertices (states) in M. p will be the pumping length for L.

Let s be a string in L of length $n \geq p$, if any. Write s as $s = s_1 s_2 \cdots s_n$, where each s_h, $1 \leq h \leq n$, is a character in Σ, the alphabet for L. Consider the substring $s' = s_1 s_2 \cdots\ s_p$. We look at the path M follows on input s'. It must go through $p+1$ vertices, and as M has only p vertices, at least one vertex must be repeated. Let *start* $= r_0$, r_1, $\cdots$, r_p be this sequence of vertices and suppose that $r_h = r_j$, where $0 \leq h < j \leq p$, is a repeated vertex, as shown in Figure 2.29.

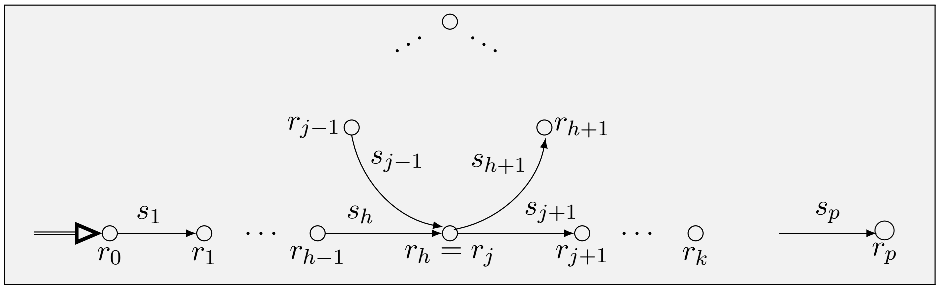


Figure 2.29: The path traversed on input $s' = s_1 s_2 \cdots s_p$.

Let x denote $s_1 \cdots s_h$, y denote $s_{h+1} \cdots s_j$, and z denote $s_{j+1} \cdots s_p s_{p+1} \cdots s_n$. The path traversed on input $s = xyz$ is illustrated in Figure 2.30.

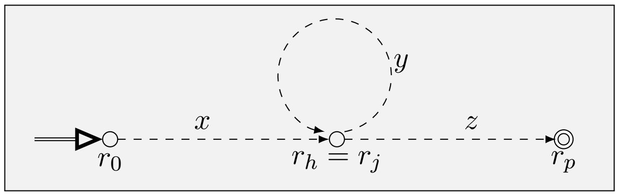


Figure 2.30: The path traversed on input $s = xyz$.

As $j > h$, $|y| = j - h > 0$. Also $|xy| = |s_1 \cdots s_j| = j \le p$. So (1) and (2) are true.

Clearly, xz is recognized by M, for x takes M from r_0 to $r_h = r_j$ and z takes M from r_j to a recognizing vertex.

Similarly xy^iz is recognized by M for any $i \ge 1$, for x takes M from r_0 to r_h, each repetition of y takes M from r_h (back) to $r_j = r_h$, and then the z takes M from r_j to a recognizing vertex. Thus (3) is also true, proving the result. □

Now, to show that a language L is non-regular we use the pumping Lemma in the following way. We begin by assuming that L is regular so as to obtain a contradiction. Next, we assert that there is a pumping length p such that for each string s in L of length at least p the three conditions of the Pumping Lemma hold. The next (and more substantial) task is to choose a particular string s to which we will apply the conditions of the Pumping Lemma, and condition (3) in particular, so as to obtain a contradiction.

Example 2.4.2. Let us look at the language $L = \{a^nb^n \mid n \ge 1\}$ again. The argument showing that L is not regular proceeds as follows.

Step 1. Suppose, so as to obtain a contradiction, that L were regular. Then L must have a pumping length $p \ge 1$ such that for any string $s \in L$ with $|s| \ge p$, s can be pumped.

Step 2. Choose s. Recall that we chose the string $s = a^pb^p$.

Step 3. By pumping s, obtain a contradiction.
As s can be pumped (for $s \in L$ and $|s| = 2p \ge p$), we can write $s = xyz$, with $|y| > 0$, $|xy| \le p$, and $xy^iz \in L$ for every integer $i \ge 0$. In particular, setting $i = 0$ shows that $xz \in L$.

Next, we show that in fact $xz \notin L$. As the first p characters of s are all a's, the substring xy must also be a string of all a's. By condition (3), with $i = 0$, we know that $xz \in L$; but the string xz is obtained by removing $|y|$ a's from s, that is $xz = a^{p-|y|}b^p$, and this string is not in L since $p - |y| \ne p$.

This is a contradiction for we have shown that $xz \in L$ and $xz \notin L$.

Step 4. Consequently the initial assumption is incorrect; that is, L cannot be recognized by a DFA, so L is not regular.

Let me stress the sequence in which the argument goes. First the existence of pumping length p for L is asserted (different regular languages L may have different pumping lengths, but each regular language containing arbitrarily long strings will have a pumping length). Then a suitable string s is chosen. The length of s will be a function of p. s is chosen so that when it is pumped a string outside of L is obtained. An important point about the pumped substring y is that while we know y occurs among the first p characters of s, *we do not know exactly which ones form the substring* y. Consequently, a contradiction must arise for every possible substring y of the first p characters in order to show that L is not regular.

There is a distinction here: for a given value of p, s is fully determined; for instance, in Example 2.4.2 for $p = 3$, $s = aaabbb$. By contrast, all that we know about x and y is that together they contain between 1 and 3 characters, and that y has at least one character. There are 6 possibilities in all for the pair (x, y), namely: (λ, a), (λ, aa), (λ, aaa), (a, a), (a, aa), (aa, a). In general, for $|xy| \le p$, there are $\frac{1}{2}p(p+1)$ choices of x and y. The argument leading to a contradiction must work for every possible choice of p and every possible partition of s into x, y, and z. Of course, you are not going to give a separate argument for each case given that there are infinitely many cases. Rather, the argument must work regardless of the value of p and regardless of which partition of s is being considered.

Next, we show an alternative Step 3 for Example 2.4.2.

Alternative Step 3. In applying Condition 3, use $i = 2$ (instead of $i = 0$), giving $xyyz \in L$. But $xyyz$ is obtained by adding $|y|$ a's to xyz, namely $xyyz = a^{p+|y|}b^p$, and this string is not in L since $p + |y| \neq p$. This is a contradiction for we have shown that $xyyz \in L$ and $xyyz \notin L$.

In Example 2.4.2 both *pumping down* ($i = 0$) and *pumping up* ($i = 2$) will yield a contradiction. This is not the case in every example. Sometimes only one direction works, and then only for the right choice of s.

A common mistake. Not infrequently, an attempted solution may try to specify how s is partitioned into x, y, and z. In Example 2.4.2, this might take the form of stating that $x = \lambda$, $y = a^p$, and $z = b^p$, and then obtaining a contradiction for this partition. This is an incomplete argument, however. All that the Pumping Lemma states is that there is a partition; it does not tell you what the partition is. *The argument showing a contradiction must work for every possible partition.*

Example 2.4.3. Let $K = \{ww^R \mid w \in \{a, b\}^*\}$. For example, $\lambda, abba, abaaba \in K$, $ab, a, aba \notin K$. We show that K is not regular.

Step 1. Suppose, so as to obtain a contradiction, that K were regular. Then K must have a pumping length $p \ge 1$ such that for any string $s \in K$ with $|s| \ge p$, s can be pumped.

Step 2. Choose s. We choose the string $s = a^p bba^p$.

Step 3. By pumping s, obtain a contradiction.
As s can be pumped (for $|s| = 2p + 2 \ge p$), we can write $s = xyz$, with $|y| > 0$, $|xy| \le p$, and $xy^iz \in K$ for every integer $i \ge 0$. Setting $i = 0$ shows that $xz \in K$.

We now show that $xz \notin K$. As the first p characters of s are all a's, the substring xy must also be a string of all a's. By condition (3), with $i = 0$, we know that $xz \in K$; but the string xz is obtained by removing $|y|$ a's from s, that is $xz = a^{p-|y|}bba^p$, and this string is not in K since it is not in the form ww^R for any w. (To see this, note that if it were in the form ww^R, as xz contains two b's, one of them would be in the w and the other in the w^R; this forces $w = a^{p-|y|}b$ and $w^R = ba^p$, and this is not possible as $p - |y| \neq p$. This is a very detailed explanation, which we will not spell out to this extent in future. Noting that $p - |y| \neq p$ will suffice.)

This is a contradiction for we have shown that $xz \in K$ and $xz \notin K$.

Step 4. Consequently the initial assumption is incorrect; that is, K cannot be recognized by a DFA, so K is not regular.

Another common mistake. I have seen attempted solutions for the above example, Example 2.4.3, that set $s = w^p(w^R)^p$. This is not a legitimate definition of s. For w is an arbitrary string, so such a definition does not fully specify the string s, that is it does not spell out the characters forming s. To effectively apply the Pumping Lemma you are going to need to choose an s in which only p is left unspecified. So if you are told, for example, that $p = 3$, then you must be able to write down s as a specific string of characters (in Example 2.4.3, with $p = 3$, $s = aaabbaaa$).

The assumption in the application of the Pumping Lemma is that the language L under consideration is recognized by a DFA. What is not known is the assumed size of the DFA. What the argument leading to a contradiction shows is that regardless of its size, the supposed DFA cannot recognize L, but this requires the argument to work for any value of p.

Example 2.4.4. $H = \{a^{i^2} \mid i \geq 0\}$. We show that H is not regular. To do this we introduce another new technique. Note that the gap between successive strings in H is growing (when we order these strings by increasing length). In particular, $|a^{(i+1)^2}| - |a^{i^2}| = (i+1)^2 - i^2 = 2i + 1$. We will use this to show that when pumping up, for large enough i, we will produce a string $s' \notin H$. We proceed as follows.

Step 1. Suppose, so as to obtain a contradiction, that H were regular. Then H must have a pumping length $p \geq 1$ such that for any string $s \in K$ with $|s| \geq p$, s can be pumped.

Step 2. Choose s. We choose the string $s = a^{p^2} \in H$.

Step 3. By pumping s, obtain a contradiction.
As s can be pumped (for $|s| = p^2 \geq p$), we can write $s = xyz$, with $|y| > 0$, $|xy| \leq p$, and $xy^iz \in H$ for every integer $i \geq 0$.

By condition (3), with $i = 2$, we know that $s' = xyyz \in H$. As $1 \leq |y| \leq p$, $|s| < |s'| \leq |s| + p = p^2 + p < (p+1)^2$. Thus $|s'|$ lies strictly between $|s|$ and the length of the next string in H in increasing length order. But this means $s' \notin H$.

This is a contradiction for we have shown that $s' \in H$ and $s' \notin H$.

Step 4. Consequently the initial assumption is incorrect; that is, H cannot be recognized by a DFA, so H is not regular.

Example 2.4.5. $J = \{w \mid w \text{ has equal numbers of } a\text{'s and } b\text{'s}\}$. We show that J is not regular. To do this we introduce a new technique. Note that if A and B are regular then so is $A \cap B = \overline{(\overline{A} \cup \overline{B})}$ (for the union and complement of regular languages are themselves regular; alternatively, see Problem 3).

Now note that $R = \{a^ib^j \mid i, j \geq 0\}$ is regular. Thus if J were regular, then $J \cap R = \{a^ib^i \mid i \geq 0\}$ would also be regular. But we have already shown that $J \cap R$ is not regular in Example 2.4.2. Thus J cannot be regular either. (Strictly, this is a proof by contradiction.)

We could also proceed as in Example 2.4.2, applying the Pumping Lemma to string $s = a^pb^p$. The exact same argument will work.

Question Does the proof of the Pumping Lemma work if we consider a p-vertex NFA that accepts L, rather than a p-vertex DFA? Justify your answer.

2.5 Regular Expressions

Regular expressions provide another, elegant and simple way of describing regular languages. The reader may wish to review Section 2.1 at this point.

Next we show that regular expressions represent exactly the regular languages.

Lemma 2.5.1. *Let r be a regular expression. There is an NFA N_r that recognizes the language represented by r: $L(N_r) = L(r)$.*

Proof. The proof is by induction on the number of operators (union, concatenation, and star) in regular expression r.

The base case is for zero operators, which are also the base cases for specifying regular expressions. It is easy to give DFAs that recognize the languages specified in each of these base cases and this is left as an exercise for the reader.

For the inductive step, suppose that r is given by one of the recursive definitions, $r_1 \cup r_2$, $r_1 \circ r_2$, or r_1^*. Since r_1, and r_2 if it occurs, contain fewer operators than r, we can assume by the inductive hypothesis that there are NFAs recognizing the languages represented by regular expressions r_1 and r_2. Then Lemmas 2.3.6–2.3.8 provide the NFAs recognizing the languages r.

We can conclude that there is an NFA recognizing $L(r)$. □

To prove the converse, that every regular language can be represented by a regular expression takes more effort. To this end, we introduce yet another variant of NFAs, called GNFAs (for *Generalized* NFAs).

In a GNFA each edge is labeled by a regular expression r rather than by one of λ or a character $a \in \Sigma$. We can think of an edge labeled by regular expression r being traversable on reading string x exactly if x is in the language represented by r. String w is recognized by a GNFA M if there is a path P in M from its start vertex to a recognizing vertex such that P's label, the concatenation of the labels on P's edges, forms a regular expression that includes w among the set of strings it represents.

In more detail, suppose P consists of edges $e_1, e_2, \cdots, e_k$, with labels $r_1, r_2, \cdots, r_k$, respectively; then P is a w-recognizing path if w can be written as $w = w_1 w_2 \cdots w_k$ and each w_i is in the language represented by r_i, for $1 \le i \le k$.

Clearly, every NFA is a GNFA, so it will be enough to show that any language recognized by a GNFA can also be represented by a regular expression.

We begin with two simple technical lemmas.

Lemma 2.5.2. *Suppose that GNFA M has two vertices u and v with h edges from u to v, edges $e_1, e_2, \cdots, e_h$. Suppose further that these edges are labeled by regular expressions $r_1, r_2, \cdots, r_h$, respectively. Then replacing these h edges by a new edge e labeled $r_1 \cup r_2 \cup \cdots \cup r_h$, or r for short, yields a GNFA N recognizing the same language as M.*

Proof. The edge replacement is illustrated in Figure 2.31.

First we show that $L(M) \subseteq L(N)$. Suppose that M recognizes string w. Let P be a w-recognizing path in M. Replacing each instance of edge e_i, for $1 \le i \le h$, by edge e yields a new path P' in N, with P' being w-recognizing (for a substring w_i read on traversing edge e_i, and hence represented by r_i, is also represented by r and so can be read on traversing edge e also). This shows $w \in L(N)$.

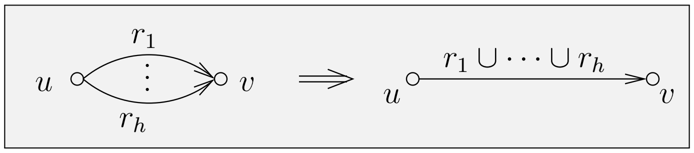


Figure 2.31: Edge replacement in Lemma 2.5.2.

Next we show that $L(N) \subseteq L(M)$. Suppose that N recognizes string w. let P be a w-recognizing path in N. Suppose that edge e is traversed j times in P, and let $w_1, w_2, \cdots, w_j$ be the substrings of w read on these j traversals. Each substring w_g, $1 \le g \le j$, is represented by regular expression r, and as $r = r_1 \cup \cdots \cup r_h$, each w_g is represented by one of r_1, r_2, $\cdots$, r_h; so let w_g be represented by r_{i_g}, where $1 \le i_g \le h$. Replacing the corresponding instance of e in P by e_{i_g} yields a path P' in M which is also w-recognizing. This shows that $w \in L(M)$. □

Lemma 2.5.3. *Suppose that GNFA M has three vertices u, v, q, connected by edges labeled as shown in Figure 2.32 and further suppose that they are the only edges incident on q. Finally, suppose that q is not a recognizing vertex.*

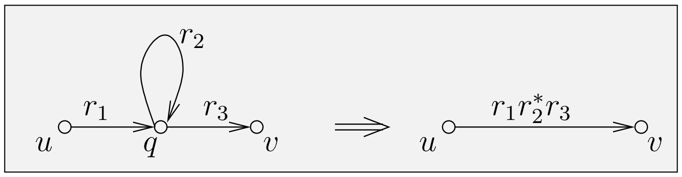


Figure 2.32: Edge replacement in Lemma 2.5.3.

Then removing vertex q and creating edge (u, v) with label $r_1 r_2^* r_3$ yields a GNFA N recognizing the same language as M.

Proof. First we show that $L(M) \subseteq L(N)$. Suppose that $w \in L(M)$, and let P be a w-recognizing path in M. We build a w-recognizing path P' in N. Consider a segment of the path from u to v going through q in P. It consists of edge (u, q), followed by some $k \ge 0$ repetitions of edge (q, q), followed by edge (q, v). This subpath has label $r_1 r_2^k r_3$. But all strings represented by $r_1 r_2^k r_3$ are represented by $r_1 r_2^* r_3$, and so we can replace this subpath by the new edge (u, v) in N. Thus w is recognized by N.

Next, we show that $L(N) \subseteq L(M)$. Suppose that $w \in L(N)$, and let P be a w-recognizing path in N. We build a w-recognizing path P' in M. Consider an instance of edge e with $e = (u, v)$ on P, if any. Suppose string w_i is read on traversing e. Then w_i is in the language represented by $r_1 r_2^k r_3$ for some $k \ge 0$ (for this is what $r_1 r_2^* r_3$ means: $r_1 r_2^* r_3 = r_1 r_3 \cup r_1 r_2 r_3 \cup r_1 r_2^2 r_3 \cup r_1 r_2^3 r_3 \cup \cdots$.) So we can replace this instance of edge e with the edge sequence (u, q) followed by k instances of edge (q, q), followed by edge (q, v). When all instances of e in P are replaced in this manner, the resulting path P' in M is still w-recognizing, so $w \in M$. □

Lemma 2.5.4. *Let M be a GNFA. Then there is a regular expression r representing the language recognized by M: $L(r) = L(M)$.*

Proof. We begin by modifying M so that its start vertex has no in-edge (if need be by introducing a new start vertex) and so that it has a single recognizing vertex with no out-edges (again, by adding a new vertex, if needed). Also, for each pair of vertices, if there are several edges between them, they are replaced by a single edge by applying Lemma 2.5.2. Let N_0 be the resulting machine and suppose that it has $n+2$ vertices $\{\textit{start}, q_1, q_2, \cdots, q_n, \textit{recognize}\}$.

We construct a sequence N_1, N_2, $\cdots$, N_n of GNFAs, where N_i is N_{i-1} with q_i removed and otherwise modified so that N_{i-1} and N_i recognize the same language, for $1 \le i \le n$.

Thus $L(N_n) = L(M)$. But L_n has two vertices, *start* and *recognize*, joined by a single edge labeled by a regular expression r, say. Clearly, the language recognized by N_n is $L(r)$. So it remains to show how to construct N_i given N_{i-1}.

Step 1. Let $q = q_i$ be the vertex being removed. For each pair of vertices $u, v \neq q$ such that there is a path u, q, v we make a new copy $q_{u,v}$ of q as shown in Figure 2.33, and then remove the vertex q and the edges incident on it.

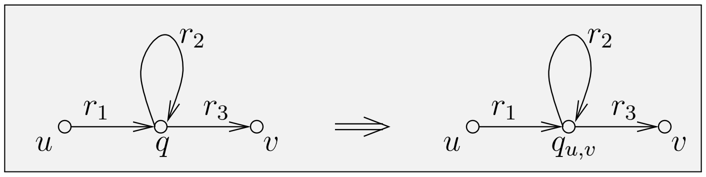


Figure 2.33: Vertex duplication in Step 1.

Clearly, a path segment (u, q repeated $k \ge 1$ times, v) in N_{i-i} has the same label as the path segment (u, $q_{u,v}$ repeated k times, v), and consequently the labels on the recognizing paths in N_{i-1} and the machine following the Step 1 modification are the same.

Step 2. In turn, for each vertex $q_{u,v}$, apply Lemmas 2.5.3 and 2.5.2 to the subgraph formed by u, $q_{u,v}$, and v, for each pair of vertices u, v. The resulting GNFA is N_i. Clearly, $L(N_i) = L(N_{i-1})$. The effect of the application of these lemmas is illustrated in Figure 2.34. Note that in fact we

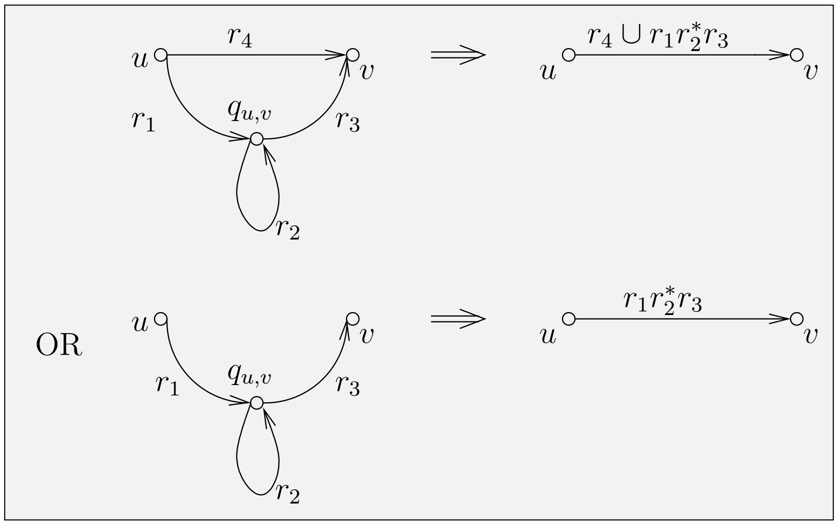


Figure 2.34: Change due to Step 2.

could have performed the change achieved by Step 2 without introducing the vertices $q_{u,v}$. They are here just to simplify the explanation. □

2.5.1 Finding a Pattern in a Text

We are now ready to return to the problem described at the start of the chapter. Let r be a regular expression and let t, the text, be a string over the same alphabet. How do we identify every occurrence of strings $s \in L(r)$ in the text t? In other words, we want to identify every substring s of t such that $s \in L(r)$. A typical way of identifying a substring is to give the indices in t of its first and last characters.

We have just shown that for every regular r, there is an NFA $N(r)$ such that $L(r) = L(N(r))$, so to test if a string is in $L(r)$ it suffices to test if it is recognized by $N(r)$. In particular, we could test whether $t \in L(N(r))$ in $O(|t|)$ time by giving t as input to $N(r)$. However, the question we want to solve is to identify all substrings of t that are in $L(N(r))$.

Suppose we modify the finite automata so that it can provide output whenever a vertex is reached. In particular, suppose that whenever a recognizing vertex is reached, the automata could output R (for recognize) and suppose we are independently tracking the index of the last character read. Then we would be able to identify every substring starting at the first character in t that lies in $L(N(r)) = L(r)$, and further this could still be done in $O(|t|)$ time. Another task is to find the first occurrence in t of a string represented by regular expression r in $O(|t|)$ time. This requires a further small modification of the automata. See exercise 22.

However, in general, we are not going to be able to report every substring in $L(r)$ that lies in t in time $O(|t|)$. For suppose $r = aa^*$, strings of one or more a's. If $t = a^n$, then t has $\frac{1}{2}n(n-1)$ substrings in $L(r)$ so reporting them (unless one finds a compact way of specifying them) will take $\Theta(n^2)$ time.

One solution is to run the just described algorithm $|t|$ times, starting, in turn, at the first character in t, at the second character, ..., at the $|t|$th character. One small but practical optimization is to stop each iteration once the only reachable vertices are sink vertices.

2.6 Correctness of Vertex Specifications

Given a Finite Automata, with a vertex specification S_v for each vertex v, how can one determine whether the vertex specifications are correct, in the sense of exactly specifying those strings for which the automata can reach v, when starting at the start vertex?

Definition 2.6.1. *A* specification S_v *for a vertex v in a DFA or NFA is a set of strings s.*

e.g. $S_v = \{\text{even length strings}\}$.

For the specifications to be correct they will need to satisfy three properties, including forward and backward consistency, which we define next. We begin by considering automata with no λ-edges.

Let $e = (u, v)$ be an edge of the finite automata labeled by a. Suppose that u and v have specifications S_u and S_v, respectively.

Definition 2.6.2. *S_u and S_v are* forward-consistent *with respect to edge e if for each $s \in S_u$, $sa \in S_v$.*

Definition 2.6.3. *S_v is* backward-consistent *if for each $a \in \Sigma$ and each non-empty string $s \in S_v$ ending in a, there is a vertex u and an edge (u, v) labeled a, with the property that $s' \in S_u$, where s' denotes the remainder of s, i.e. $s = s'a$.*

Definition 2.6.4. *The specifications for a finite automata are* consistent *if*

1. *The specifications are forward-consistent with respect to every edge.*

2. *The specifications are backward-consistent.*

3. *$\lambda \in S_{start}$ and $\lambda \notin S_v$ for $v \neq start$, where start denotes the start vertex.*

Lemma 2.6.5. *The specifications for a finite automata with no λ-edges are consistent exactly if they specify the strings that can reach each vertex.*

Proof. Suppose for a contradiction that the specifications are consistent yet some specification is incorrect, i.e. it fails to specify exactly those strings that can reach its vertex.
Case 1. There is a shortest string s that is incorrectly included in some specification S_v.
By (3), $s \neq \lambda$. So s can be written as $s = s'a$, where $a \in \Sigma$. By backward-consistency, $s' \in S_u$ for some vertex u where (u, v) is an edge labeled a. Now s' is correctly included in S_u as s was a shortest incorrectly placed string; but as s' is correctly in S_u, then in fact s is correctly in S_v. So this case does not arise.
Case 2. There is a shortest string s that is incorrectly missing from some specification S_v.
Again, $s \neq \lambda$. So s can be written as $s = s'a$, where $a \in \Sigma$. Let U be the set of vertices with an edge labeled a into v (i.e. for each $u \in U$, there is an edge (u, v) labeled by a). Then s' reaches some vertex in U, vertex u say. As $|s'| < |s|$, $s' \in S_u$. But then $s \in S_v$ by forward consistency. So this case does not occur either.

We conclude that if the specifications are consistent then all the specifications are correct.

To show that specification correctness implies consistency just entails checking that Conditions 1–3 above hold. This is immediate from the definitions of forward and backward consistency and is left to the reader. □

We account for the λ-edges by making appropriate small changes to the definition of consistency. What we want to do is to modify the automata by removing all λ-edges and introducing appropriate replacements. Let M be an NFA; we build an equivalent NFA M' with no λ edges, as follows.

1. Make all vertices reachable on input λ into start vertices.

2. For each pair of vertices (u, v) and for each label $a \in \Sigma$, if there is path from u to v labeled by a, introduce an edge (u, v) labeled by a, and then remove all λ-edges.

We need to explain what it means to have more than one start vertex: M' is simply understood as starting at all its start vertices simultaneously.

It is not hard to see that M' recognizes exactly the same strings as the original machine. The result of Lemma 2.6.5 applies to M'. Note that the vertex specifications used for M are the ones being used for M' too. Consequently, one can check the correctness of the specifications for M by checking that they are consistent in M'.

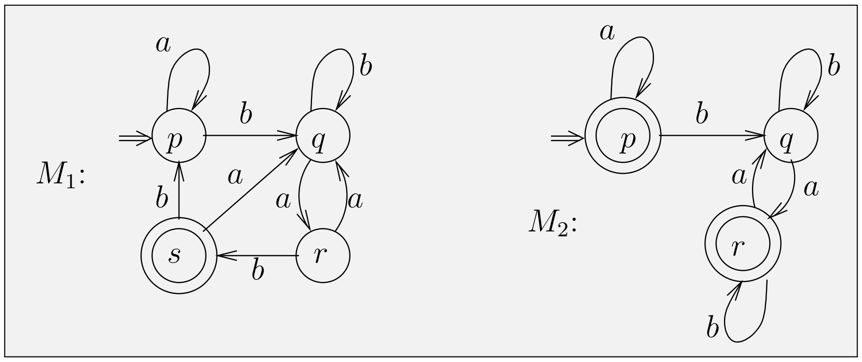


Figure 2.35: Automata for Problem 1.

Exercises

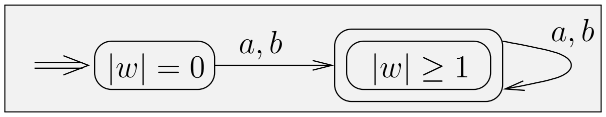
1. Consider the Finite Automata shown in Figure 2.35.

   Answer the following questions regarding each of the automata.

   i. Name its start vertex.
   ii. List its recognizing vertices.
   iii. List the series of vertices the finite automata goes through as the following input is read: *ababb*.
   iv. Is *ababb* recognized by the automata?

2. Draw the graphs of Deterministic Finite Automata recognizing the following languages. In each subproblem, the alphabet being used is $\Sigma = \{a, b\}$.

   i. The empty set $\emptyset$.
   ii. The set containing just the empty string: $\{\lambda\}$.
   iii. The set of all strings: Σ^*.
   iv. The set of all strings having at least one character: $\{w \mid |w| \geq 1\}$.
   Sample solution.



   v. The set of all strings of length two or more: $\{w \mid |w| \geq 2\}$.
   vi. The set of all strings that begin with an a.
   vii. The set of all strings that end with at least one b.
   viii. The set of all strings containing aa as a substring.
   ix. The set of all strings containing at least four a's.

x. The set of all strings either starting with an "a" and ending with a "b", or starting with a "b" and ending with an "a".

xi. The set of all strings such that no two b's are adjacent.

xii. The set of all strings excepting aba: $\{w \mid w \neq aba\}$.

xiii. The set of all strings with alternating a's and b's.

xiv. The set of all strings with a's in all the even positions and a's or b's in the odd positions (so λ, bab, aab, b are in this set, but ab is not).

xv. The set of all strings of even length for which the second character is a "b".

xvi. The set of all strings containing at least two a's.

xvii. The set of all strings that contain aba as a substring.

xviii. The set of strings in which all the a's come before all the b's.

3. Consider the construction from Section 2.2.4. Show how to modify this construction so as to give a (Deterministic) Finite Automata M recognizing $A \cap B$, assuming that A and B can both be recognized by Finite Automata. Explain briefly why your construction is correct; that is, explain why $L(M) = A \cap B$.

4. Each of the following languages can be obtained by applying set operations (one of union, intersection, or complement) to simpler languages. By building Deterministic Finite Automata recognizing the simpler languages and then combining or modifying them, build Deterministic Finite Automata to recognize the following languages. For intersections, Problem 3 may be helpful. In each subproblem, the alphabet being used is $\Sigma = \{a, b\}$.

   i. The set of all strings that contain at least one a or at least two b's.

   ii. The set of all strings that do not contain the substring aa.

   iii. The set of all strings other than the empty string: $\{w \mid w \neq \lambda\}$.

   iv. The set of all strings other than a or bb: $\{w \mid w \neq a, bb\}$.

   v. The set of all strings containing at least one a and at least one b.

   vi. The set of all strings containing at least one a and at most two b's.

   vii. The set of all strings with an even number of a's and an odd number of b's.

   viii. The set of all strings with an even number of a's and no adjacent a's.

   ix. The set of all strings that start with an a and end with the character b.

   x. The set of all strings that have either two or three b's and that have exactly two a's.

5. Draw the graphs of NFAs recognizing the following languages. In each subproblem, the alphabet being used is $\Sigma = \{a, b\}$. In all the problems below, the edge count refers to the number of multiedges—so two parallel edges, labeled by a and b respectively, count as one edge.

   i. $L = a^*$, using a 1-vertex NFA. Why is your solution not a DFA?

   ii. $L_2 = \{w \mid w \text{ ends with } bb\}$, using a 3-vertex, 3-edge NFA.

iii. $L_3 = \{w \mid w \text{ has both } aa \text{ and } bb \text{ as substrings}\}$, using a 9-vertex,13-edge NFA (8 vertices and 12 edges is doable).

iv. $L_4 = \{w \mid w \text{ is of even length or the second symbol in } w \text{ is a } b \text{ (or both)}\}$, using a 5-vertex NFA.

6. Using the method of Section 2.3.2, convert the following NFAs to DFAs.

   i. The solution to Problem 5(i).

   ii. The solution to Problem 5(ii).

   iii. The solution to Problem 5(iv).

   iv. The NFA in Figure 2.36(a).

   v. The NFA in Figure 2.36(b).

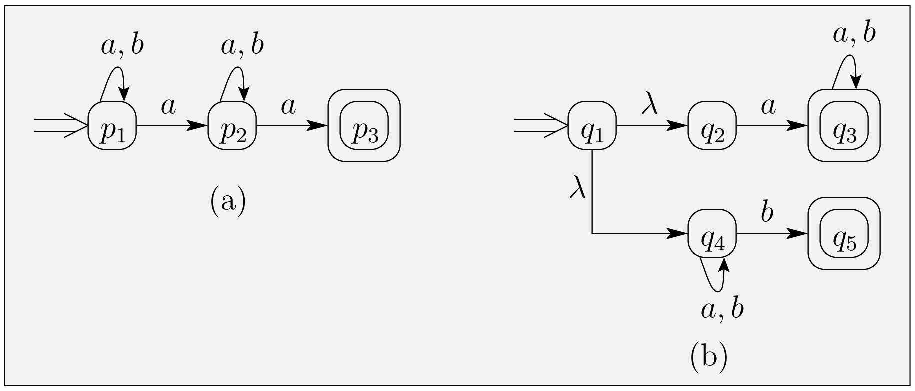


Figure 2.36: Automata for Problem 6.

7. Let the following languages all be subsets of $\{a, b\}^*$:

   $A = \{w \mid w \text{ begins with an } a\}$.

   $B = \{x \mid x \text{ ends with the character } b\}$.

   $C = \{w \mid |w| \geq 2\}$.

   $D = \{x \mid x \text{ contains } aa \text{ as a substring}\}$,

   $E = \{w \mid \text{all characters in even positions in } w \text{ are } a\text{'s and } w \text{ has even length}\}$, i.e. the second, fourth, sixth, etc. characters are all a's.

   $F = \{w \mid w \text{ contains at least one } a\}$.

   $G = \{w \mid w \text{ begins with the character } b\}$.

   $H = \{w \mid w \text{ contains } ba \text{ as a substring}\}$.

   $J = \{w \mid w\text{'s next to last character is an } a\}$.

   $K = \{w \mid \text{all characters in odd positions in } w \text{ are } a\text{'s and } w \text{ has even length}\}$. (See the definition of E also.)

Using the methods of Section 2.3.1, give the graphs of NFAs that recognize the following languages.

i. $A \cup B$.

ii. $F \cup G$.

iii. $C \circ D$.

iv. $H \circ J$.

v. E^*.

vi. K^*.

vii. $L = \emptyset^*$ (recall that $\emptyset$ is the empty language). Trust the construction. What strings, if any, are in $\emptyset^*$?

8. For each of the following languages, (a) construct an NFA recognizing the language, and then (b) Convert the NFA to a DFA recognizing the same language using the method of Section 2.3.2. You need show only the portion of the DFA reachable from the start vertex.

   i. $K = \{ba, bab\}^*$. The NFA must have a single recognizing vertex, and 4 vertices altogether.

   ii. $L = \{b, bab\}^*$. The NFA must have a single recognizing vertex, and 4 vertices altogether.

9. Let L be a regular language. Define the reverse of L, $L^R = \{w \mid w^R \in L\}$, i.e. L^R contains the reverse of strings in L. Show that L^R is also regular.
   Hint. Suppose that M is a DFA (or an NFA if you prefer) recognizing L. Construct an NFA M^R that recognizes L^R; M^R will be based on M. Remember to argue that $L(M^R) = L^R$.

10. Give regular expressions that represent the following languages. Suggestion: Test your solutions with a few strings both in and not in the language. In each subproblem, unless otherwise stated, the alphabet being used is $\Sigma = \{a, b\}$.

    i. $A = \{w \mid w$ has an even number of b's$\}$.

    ii. $B = \{w \mid$ every a in w is immediately preceded by one or more b's$\}$.

    iii. $C = \{w \mid w$ does not contain the substring $bb\}$.

    iv. $D = \{w \mid$ the third to last character in w is a "b"$\}$.

    v. $E = \{w \mid w$ contains at least two b's$\}$.

    $*$vi. Let $\Sigma = \{0, 1, 2\}$. Let $F = \{w \mid$ the sum of the digits in w mod 3 equals $0\}$. Give a direct construction without using Lemma 2.5.4. Explain why your solution is correct.

11. For each of the following regular expressions answer the following questions:

    i. $(a \cup b)^* a (a \cup b)^*$.

    ii. $(a \cup b)^* a (a \cup b)^* b (a \cup b)^*$.

    iii. $(\lambda \cup a) b^*$.

    iv. $(\lambda \cup aa) b^*$.

    a. Give two strings in the language represented by the regular expression.

b. Give two strings not in the language represented by the regular expression.

c. Describe in English the language represented by the regular expression.

12. Using the method of Lemma 2.5.1, give NFAs that recognize the languages represented by the following regular expressions.

    a. $(a \cup bb)^*(aba)$.

    b. $(bab)(a \cup bb)^*$.

    c. $((ab)^* \cup a)b$.

13. Using the method of Lemma 2.5.4, give regular expressions representing the languages recognized by the following NFAs.

    a. The NFA in figure 2.37.

    b. The DFA M_1 in Section 2.2.2.

    c. The DFA M_3 in Section 2.2.2.

    d. The DFA M_4 in Section 2.2.2.

    e. The DFA in Figure 2.13(b).

    f. The DFA in Figure 2.15.

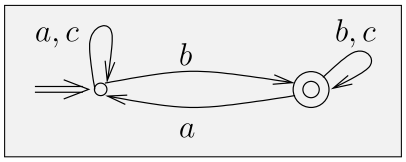


Figure 2.37: NFA for Problem 13.

14. Let r be a regular expression, and let $L(r)$ be the language generated by r. Define $L^R(r) = \{w \mid w^R \in L(r)\}$ to be the reversal of $L(r)$, i.e. $L^R(r)$ contains the reverse of strings in L.

    Give a regular expression to generate $L^R(r)$.
    Hint. You will need to proceed inductively using strong induction. The base cases will be for regular expressions defined by the first three rules for creating regular expressions, and the inductive cases will be for regular expressions defined by the remaining three rules.

15. Let r be a regular expression and let $L(r)$ be the language generated by r.

    i. Give an algorithm to test if $L(r) = \phi$.

    ii. Give an algorithm to test if $L(r) = a$.

16. Use the Pumping Lemma to show that the following languages are not regular.

i. $L = \{a^i b^i \mid i \geq 0\}$.
Sample solution. Suppose, so as to obtain a contradiction, that L were regular. Then L must have a pumping length $p \geq 1$ such that for any string $s \in L$ with $|s| \geq p$, s can be pumped.
Choose s to be the string $s = a^p b^p$.
As $s \in L$ and $|s| = 2p \geq p$, s can be pumped. In other words, we can write $s = xyz$, with $|y| > 0$, $|xy| \leq p$, and $xy^i z \in L$ for every integer $i \geq 0$.
By pumping s, we will obtain a contradiction, as follows.
As the first p characters of s are all a's, the substring xy must also be a string of all a's. As $xy^i z \in L$ for any integer $i \geq 0$, on setting $i = 0$, we have that $xz \in L$; but the string xz has removed $|y|$ a's from s, that is $xz = a^{p-|y|} b^p$, and this string is not in L since $p - |y| \neq p$. This is a contradiction for we have shown that both $xz \in L$ and $xz \notin L$.
Consequently the initial assumption is incorrect; that is, L is not regular.

ii. $A = \{a^i b a^i \mid i \geq 0\}$.

iii. $B = \{a^{2i} b^i \mid i \geq 1\}$.

iv. $C = \{a^i b^i c^i \mid i \geq 1\}$.

v. Let $\Sigma = \{(,)\}$ and let D be the language of legal balanced parentheses: i.e., for each left parenthesis there is a matching right parenthesis to its right, and pairs of matched parentheses do not interleave. For example, the following are in D: (), ()(), ((())()), and the following are not in D:)(, (()())),((.

vi. $E = \{ww \mid w \in \{a,b\}^*\}$.

vii. $F = \{w \mid w = w^R, w \in \{a,b\}^*\}$. Such w are called *palindromes.*

viii. $G = \{ww^R w, w \in \{a,b\}^*\}$.

ix. $H = \{a^{2^i} \mid i \geq 0\}$, strings with a repeated 2^i times for some i.

x. $I = \{a^q \mid q \text{ prime}\}$, strings containing a prime number of a's.

17. If for language L the pumping length is 4 and $a^4 b^4 \in L$, then if you choose $s = a^4 b^4$ and apply the Pumping Lemma to s, how many pairs x, y are there, and what are these pairs?

18. Does the proof of the Pumping Lemma work using a p-state NFA accepting regular language L instead of a p-state DFA? Justify your answer briefly.

19. Let $A = \{a^i w \mid w \in \{a,b\}^* \text{ and } w \text{ contains at most } i \text{ } a\text{'s}, i \geq 1\}$. Show that A is not regular. Remember that if R is regular and if A is regular then so is $A \cap R$. It may be helpful to choose a suitable R, and then show that $A \cap R$ is not regular.

20. Show that the following languages are not regular.

i. $A = \{a^i b^j \mid i \neq j\}$.

ii. $B = \{uvu \mid u, v \in \{a,b\}^* \text{ and } v \in a^*\}$.

*iii. $C = \{uvu \mid u, v \in \{a,b\}^* \text{ and } u, v \neq \lambda\}$.

iv. $D = \{w \mid w \neq w^R, w \in \{a,b\}^*\}$.

21. Use the Pumping Lemma to show that the following languages are not regular.

    i.
    $$A = \{m = n * r \mid m, n, r \in \{0,1\}^* \text{ and } m = n * r\}.$$
    The strings in A include the equal sign and the multiplication sign.

    ii.
    $$B = \{m = n + r \mid m, n, r \in \{0,1\}^* \text{ and } m = n + r\}.$$
    The strings in B include the equal sign and the addition sign.

22. Let r be a regular expression and let M be an NFA recognizing $L(r)$. Suppose M is modified to output R the first time it reaches a recognizing vertex. Let t denote an input to M. Design a machine N, based on M, which outputs R the first time a string in r starting anywhere in t is read. (See the discussion in section 2.5.1.)

23. Let $C = \text{shuffle}(A, B)$ denote the *shuffle* C of two languages A and B; it consists of all strings w of the form $w = a_1b_1a_2b_2\cdots a_kb_k$, for $k > 0$, with $a_1a_2\cdots a_k \in A$ and $b_1b_2\cdots b_k \in B$. Show that if A and B are regular then so is $C = \text{shuffle}(A, B)$.

24. Define $\text{ROC}(w)$ to be the collection of strings obtained from w by removing exactly one of w's characters.
    e.g. $\text{ROC}(bab) = \{ab, bb, ba\}$. Let $\text{ROC}(L) = \cup_{w\in L}\text{ROC}(w)$.
    Show that if L is regular then so is $\text{ROC}(L)$.
    Sample solution. Let M be a DFA recognizing L. We will construct NFA N to accept $\text{ROC}(L)$. The intuitive idea is that N will be obliged to follow an edge in M without reading a character exactly once. This is implemented as follows. The graph for N consists of two copies of the graph for M, with the first copy joined to the second by λ-labeled edges. Hence a path in N transits once from the first copy to the second. Having the start vertex for N in the first copy and the recognizing vertices in the second copy ensures that a string is recognized exactly if it omits one character that would be read on the corresponding path in M.

    The details follow. N comprises two copies of M, named M_1 and M_2. For each vertex p in M there will be a vertex p_1 in M_1 and a vertex p_2 in M_2. And for each a-edge (p, q) in M (an a-edge is an edge labeled by a), in N there will be a-edges (p_1, q_1) and (p_2, q_2) plus a λ-edge (p_1, q_2). We will say that edge (p_1, q_2) has pseudo-label a, meaning that (p_1, q_2) was obtained from a-edge (p, q). An edge (p_1, q_2) could have more than one pseudo-label. Finally, for each recognizing vertex r in M there will be a recognizing vertex r_2 in N, and if s is the start vertex for M, s_1 will be the start vertex for N.

    Suppose w is recognized by M. Let $w = uav$ for some character $a \in \Sigma$ and strings $u, v \in \Sigma^*$ (so $w \neq \lambda$). Then there is a recognizing path P for w which can be partitioned into path P_u labeled by u, followed by an a-edge (p, q), followed by a path P_v labeled by v. Consider the following path P' in N: It comprises path P_u in M_1, followed by edge (p_1, q_2), followed by path P_v in M_2. Clearly P' is a recognizing path in M' and P' has label $u\lambda v = uv$. This construction works for any of the characters in w, and thus the resulting strings $w' = uv$ form the set $\text{ROC}(w)$, which is therefore recognized by N. As this applies to any w recognized by M, it follows that $\text{ROC}(L) \subseteq L(N)$.

On the other hand, if w' is recognized by N, then there is a recognizing path P' for w' which comprises a path P_u in M_1, followed by an edge (p_1, q_2), followed by a path P_v in M_2. Let u be the label on P_u, v the label on P_v, and a the pseudo-label on (p_1, q_2) ((or one of them if there are several). Consider the path P in M comprising P_u, then edge (p, q), then path P_v. It has label $w = uav$ and is recognizing. Clearly $w' \in \text{ROC}(w)$. Thus for each $w' \in L(N)$ there is a $w \in L(M)$ with $w' \in \text{ROC}(w)$. In other words $L(N) \subseteq \text{ROC}(L)$.

This shows that N recognizes $\text{ROC}(L)$, and hence $\text{ROC}(L)$ is regular if L is regular.

25. Let $w' = \text{Substitute}(a, b, w)$, or $\text{Subst}(a, b, w)$ for short, if w' is obtained by replacing every "a" in w by a "b". For example, $\text{Subst}(a, b, cacaa) = cbcbb$.
Let $\text{Subst}(a, b, L) = \{w' \mid w' = \text{Subst}(a, b, w) \text{ for some } w \in L\}$.
Show that if L is regular then so is $\text{Subst}(a, b, L)$.

26. Let $w = w_1 w_2 \cdots w_k$ where each $w_i \in \Sigma$, $1 \le i \le k$. And let h be a function from the alphabet Σ into the set of strings Γ^*, so that $h(a) \in \Gamma^*$ for each $a \in \Sigma$.
Define $\text{FullSubst}(w, h)$, or $\text{FS}(w, h)$ for short, to be $h(w_1)h(w_2)\cdots h(w_k)$. Note that $\text{FullSubst}(\lambda, h) = \lambda$.
e.g. if $h(a) = cc$, $h(b) = de$, then $\text{FS}(aba) = ccdecc$.
For $L \subseteq \Sigma^*$, define $\text{FS}(L, h) = \{\text{FS}(w, h) \mid w \in L\}$.
Show that if L is regular then so is $\text{FS}(L, h)$.

27. Let $w = w_1 w_2 \cdots w_k$ where each $w_i \in \Sigma$, $1 \le i \le k$. And let r be a function mapping characters in Σ into regular expressions over the alphabet Γ.
Define $\text{SubstRegExpr}(w, r)$, or $\text{SRE}(w, r)$ for short, to be $r(w_1)r(w_2)\cdots r(w_k)$. Note that $\text{SRE}(\lambda, r) = \lambda$.
e.g. if $r(a) = c^*$, $r(b) = \lambda \cup d$, then $r(ab) = c^*(\lambda \cup d)$.
For $L \subseteq \Sigma^*$, define $\text{SRE}(L, r) = \cup_{w \in L} \text{SRE}(w, r)$.
Show that if L is regular then so is $\text{SRE}(L, r)$.

28. Define $\text{RemoveOneSymbol}(w, a)$, or $\text{ROS}(w, a)$ for short, to be the string w with each occurrence of a deleted.
e.g. $\text{ROS}(abac, a) = bc$, $\text{ROS}(bc, a) = bc$.
Let $\text{ROS}(L, a) = \{\text{ROS}(w, a) \mid w \in L\}$.
Show that if L is regular then so is $\text{ROS}(L, a)$.

29. Define $\text{ReplaceOneSymbol}(w, a, b)$, or $\text{RpOS}(w, a, b)$ for short, to be the collection of strings obtained from w by replacing exactly one occurrence of a with a b.
e.g. $\text{RpOS}(bb, a, b) = \{\}$; $\text{RpOS}(acca, ab) = \{bcca, accb\}$.
Let $\text{RpOS}(L, a, b) = \cup_{w \in L} \text{RpOS}(w, a, b)$.
Show that if L is regular then so is $\text{RpOS}(L)$.

30. The next problem introduces yet another way of defining regular languages.

    i. Let L be a language over the alphabet Σ. Let $R(x, y)$ be the following property defined with respect to L: $R(x, y)$ is TRUE exactly if for every string $z \in \Sigma^*$, $xz \in L \iff yz \in L$. Prove that R is an equivalence relation, that is that: (a) $R(x, x)$ is TRUE, (b) $R(x, y)$ being TRUE implies $R(y, x)$ is also TRUE, and (iii) if $R(x, y)$ and $R(y, z)$ are both TRUE, then so is $R(x,z)$. We write $x \equiv y$ if $R(x,y)$ is TRUE, or sometimes $x \equiv_L y$ to emphasize the dependence on the language L.

ii. X is an equivalence class with respect to R if (a) for every pair of strings $x, y \in X$, $x \equiv y$, and (b) for every $x \in X$, for every y such that $x \equiv y$, $y \in X$ also.
Suppose that there is a DFA M recognizing L.
(a) In addition, suppose that input strings x and y have the same destination vertex p in M. Then show that $x \equiv y$.
(b) Conclude that L has a finite number of equivalence classes.

iii. Show that if R partitions Σ^* into a finite number of equivalence classes then there is a DFA recognizing L.
Hint: What is the first thing to try as possible descriptors for the vertices in the DFA?

This question shows that a language is regular if and only if it has a finite number of equivalence classes.

31. Let Prefix(L) $= \{u \,|\, \text{there is a string } v \text{ such that } uv \in L\}$. Show that if L is regular then so is Prefix(L).
Hint: Let u be a prefix of w (i.e. there is a string v such that $uv = w$). Let M be a DFA recognizing L. What can you say about the relationship between the destination vertices reached in M by strings u and w?

32. Let Suffix(L) $= \{v \,|\, \text{there is a string } u \text{ such that } uv \in L\}$. Show that if L is regular then so is Suffix(L).

33. Let Continuation(L) or Cont(L) for short be defined by Cont(L) $= \{w \,|\, \text{there is a string } u \in L \text{ and a string } v \text{ such that } uv = w\}$. Show that if L is regular then so is Cont(L).

34. Let Inf-Cont(L) $= \{u \,|\, u \in L \text{ and for infinitely many } v,\ uv \in L\}$. Show that if L is regular then so is Inf-Cont(L).

35. Let All-Cont(L) $= \{u \,|\, \text{for all strings } w,\ uw \in L\}$. Show that if L is regular then so is All-Cont(L).

36. Let All-Pref(L) $= \{w \,|\, \text{for all prefixes } u \text{ of } w,\ u \in L\}$. Show that if L is regular then so is All-Pref(L). (Recall that u is a prefix of w if there is a string v such that $uv = w$.)

37. i. Let $\frac{1}{2}$-$L = \{u \,|\, \exists v \text{ with } |v| = |u| \text{ and } uv \in L\}$.
Suppose that L is regular; then show that $\frac{1}{2}$-L is also regular.
Hint. Think nondeterministically. You will need to "guess" v. Getting its length right can be done only as u is being read. What else do you need to guess, and what needs to be checked?

ii. Let Square-$L = \{u \,|\, \exists v \text{ with } |v| = |u|^2 \text{ and } v \in L\}$.
Show that if L is regular then so is Square-L.

iii. Let Power-$L = \{u \,|\, \exists v \text{ with } |v| = 2^{|u|} \text{ and } v \in L\}$.
Show that if L is regular then so is Power-L.

38. i. Let $L = \{a^3, a^5\}$. Show that L^* includes all strings of 15 or more a's. Hence conclude that L^* is regular.

ii. Let $L \subset a^*$ be finite. Show that L^* is regular.

39. Consider the following variant of a DFA, called a *Mealy-Moore machine*, that writes an output string as it is processing its input. Viewed as a graph, it is like a DFA but each edge is labeled with both an input character $a \in \Sigma$, as is standard, and an output character $b \in \Gamma$ or the empty string λ, where Γ is the output alphabet (possibly $\Sigma = \Gamma$).

    The output $M(x)$ produced by M on input x is simply the concatenation of the output labels on the path followed by M on reading input x. The output language produced by M, $O(M)$, is defined to be
$$O(M) = \{M(x) \mid x \in L(M)\},$$
    the output strings obtained on inputs x that M recognizes.

    Show that $O(M)$ is regular.

40. A 2wayDFA is a variant of a DFA in which it is possible to go back and forth over the input, with no limit on how often the reading direction is reversed.

    This can be formalized as follows. The input $x \in \Sigma^*$ to the DFA is sandwiched between symbols ¢ and \$, so the DFA can be viewed as reading string $x_0 x_1 x_2 \cdots x_n x_{n+1}$, where $x = x_1 x_2 \cdots x_n$, $x_0 = ¢$, and $x_{n+1} = \$$. The DFA is equipped with a *read head* which will always be over the next symbol to be read. At the start of the computation, the read head is over the symbol x_1, and the DFA is at its start vertex.

    In general, the DFA will be at some vertex v, with its read head over character x_i for some i, $0 \le i \le n+1$. On its next move the DFA will follow the edge leaving v labeled x_i. This edge will also carry a label L or R indicating whether the read head moves left (so that it will be over symbol x_{i-1}), or right (so that it will be over x_{i+1}). The latter is the only possible move for a standard DFA. There are two constraints: when reading ¢ only moves to the right are allowed and when reading \$ only moves to the left are allowed. If there is no move, the computation ends.

    A 2wayDFA M recognizes an input x if it is at a recognizing vertex when the computation ends. Show that the language recognized by a 2wayDFA is regular.

    Hint. Consider the sequence of vertices M is at when its read head is over character x_i. Observe that in a recognizing computation there can be no repetitions in such a sequence. Now create a standard NFA N to simulate M. A state or supervertex of N will record the sequence of vertices that M occupies when its read head is over the current input character. The constraints on N are that its moves must be between consistent supervertices (you need to elaborate on what this means). Also, you need to specify what are the recognizing supervertices of N, and you need to argue that M and N recognize the same language.

Chapter 3

Tree Structured Languages: Context Free Languages and Pushdown Automata

In the previous chapter we saw that many interesting languages are not regular, for example, $L = \{a^ib^i \mid i \geq 0\}$. In this chapter we will be looking at a larger class of languages, called Context Free Languages, which includes the above language L as well as all regular languages.

To gain some intuition, let's discuss how we might characterize the strings in L. One approach is recursive: a string $w \in L$ is either λ, the empty string, or it can be written as avb where v is a shorter string, also in L. This recursive structure can also be viewed as a tree, and this is why we say these languages are tree structured.

A second approach is to use a stack to recognize strings in L. Given an input string w, as we read the initial a's in w we push them on the stack; then, for each b we read, we pop one a from the stack. Clearly, the stack becomes empty precisely when we finish reading w only if the number of a's and b's are equal (and also only if w is of the form a^*b^*).

As it turns out, these two approaches are equivalent. The first approach is formalized via Context Free Grammars (CFGs), and the second approach via Pushdown Automata (PDAs). The latter are Finite Automata with the addition of a stack.

Context Free Grammars are widely used in practice. Perhaps the most important usage is to specify and recognize legal programs (there is more to this than can be managed by Context Free Grammars alone, but the for each programming language, the basic form of a legal program is specified by a corresponding Context Free Grammar). One part of this, which we will look at more carefully, is the specification of legal arithmetic expressions.

Interestingly, Context Free Grammars were first introduced to provide a specification of the structure of natural language sentences. We will begin the next section with an informal description of this perspective focusing on the structure of English sentences, keeping in mind that while the details vary from language to language, every natural language is believed to have a specification of this sort.

3.1 Tree Structured Languages

We begin with an intuitive overview, by looking at two examples: natural languages and arithmetic expressions.

Natural Languages Let's start by looking at some sentences in English. For example, consider the sentence "The hungry dog eagerly ate a crunchy biscuit". This is an example of a simple sentence, which consists of a subject ("The hungry dog"), followed by a verb phrase ("eagerly ate"), followed by an object ("a crunchy biscuit"). Both the subject and the object are noun phrases. These noun phrases take the form: an article, followed by an adjective, followed by a noun. The verb phrase consists of an adverb followed by a verb. We can show this structure using a tree, as shown in Figure 3.1. The strings in the language (namely English sentences here) are just the leaf labels in left-to-right order (incidentally, the alphabet here is the set of English words). Such trees can be used to generate strings comprising English sentences, and to recognize these strings requires the construction, explicitly or implicitly, of the same trees. This does not mean that every string of words that can be generated in this way is a legitimate sentence (e.g. "the rainbow walked a cow"); there are further constraints that go beyond the constraints of this tree structure.

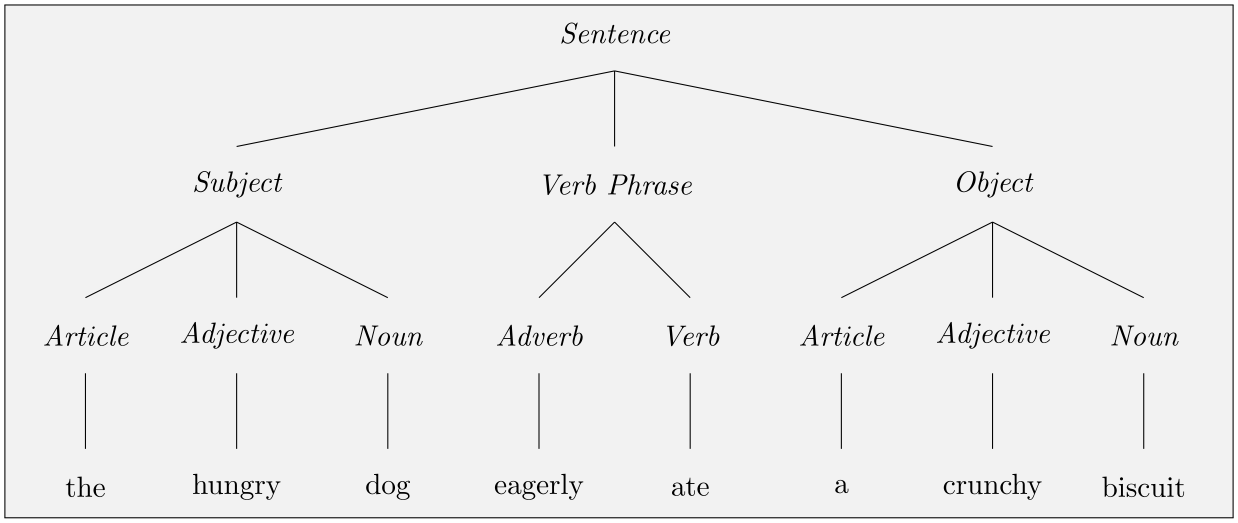


Figure 3.1: The tree generating the sentence "The hungry dog eagerly ate a crunchy biscuit".

We can create more complex sentences. For example: "The hungry dog eagerly ate a crunchy biscuit, which I had bought at the corner store." Here the sentence consists of a main clause ("The hungry dog eagerly ate a crunchy biscuit") and a dependent clause ("which I had bought at the corner store"). The tree structure for this sentence is a main clause followed by a connector ("which") followed by the dependent clause "I had bought at the corner store". The dependent clause has no explicit object — in fact its object is "the biscuit", but this is meant to be understood by the reader and thus is not explicitly spelled out; this clause also has an indirect object ("at the store").

We can add further dependent clauses, as in: "The hungry dog eagerly ate a crunchy biscuit, which I had bought at the corner store, which was owned by Jasmine Wolf, whom I had first met in kindergarten." In principle, we can keep adding dependent clauses indefinitely, pointing to the recursive nature of spoken languages, but in practice, if there are more than a few dependent clauses, the listener quickly loses track of what is being said (or written).

The language structure of English, let alone of the thousands of other spoken languages, is a huge topic and one we are not going to explore further here.

Arithmetic Expressions The basic form for an arithmetic expression is to have an operand, followed by an operator, followed by another operand; the operands are themselves arithmetic expressions. This recursive formulation needs a base case: it is provided by arithmetic expressions that are plain numbers; for simplicity, we limit ourselves to non-negative integers. The operators we consider are $+, -, \times, \div$.

Arithmetic expressions are readily presented as trees, called *expression trees*; a few small examples are shown in Figure 3.2.

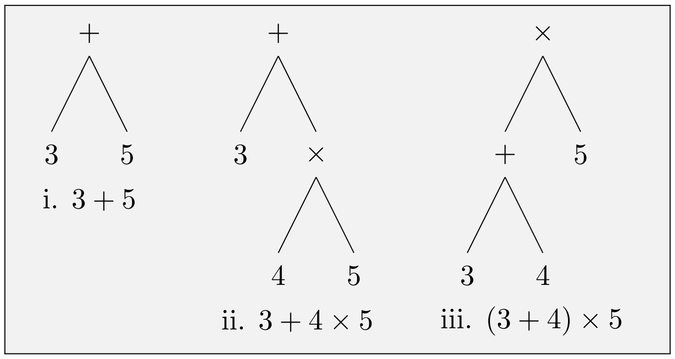


Figure 3.2: Example expression trees.

The expression trees in Figure 3.2 correspond respectively to the expressions $3+5$, $3+4\times 5$, and $(3+4)\times 5$ (and not to $3+4\times 5$ which equals $3+(4\times 5)$ because of operator precedence). We now see that an arithmetic expression could begin and end with matching left and right parentheses.

To match the way we presented the English sentence in tree form, we would like the text form of the arithmetic expression to be obtained from reading the leaves of the tree in left-to-right order. Accordingly, as shown in Figure 3.3, we create derivation trees for the arithmetic expressions in Figure 3.2.

Arithmetic expressions can be found in many, many programs. In order to execute these programs the arithmetic expressions have to be understood. This is done by implicitly building the corresponding derivation trees.

We are now ready to be more precise. In the next section we introduce Context Free Grammars, which provide the rules for creating the derivation trees of the sort we have been seeing.

3.2 Context Free Grammars

Context Free Grammars (CFGs) provide a way of specifying certain recursively defined languages.

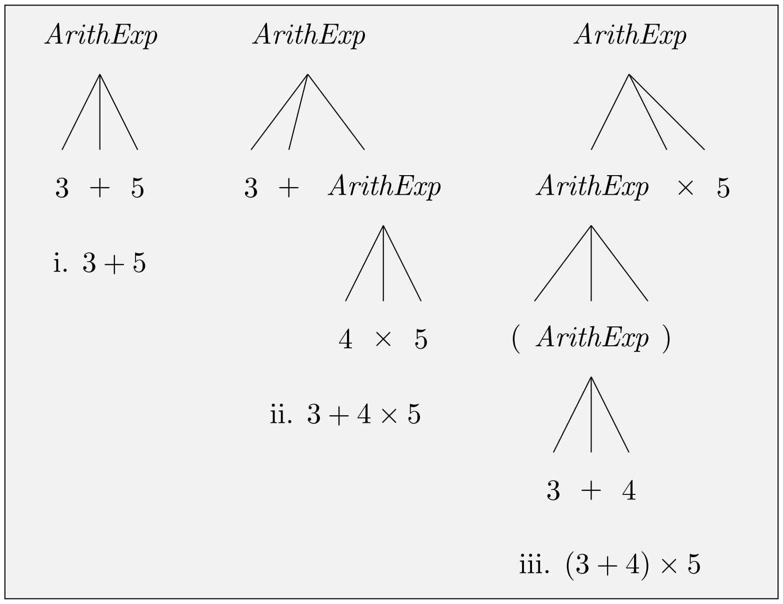


Figure 3.3: Trees generating the arithmetic expressions in Figure 3.2.

Let's begin by giving a recursive method for generating non-negative integers in decimal form. We will call this representation of integers *decimal numbers*. A decimal number is defined to be either a single digit (one of 0–9) or a single digit followed by another decimal number. For example, the derivation of the number 147 is shown in Figure 3.4. In fact, the strings of decimal numbers form a regular language, but we are using them to provide a simple example of the recursive structure that Context Free Grammars enable.

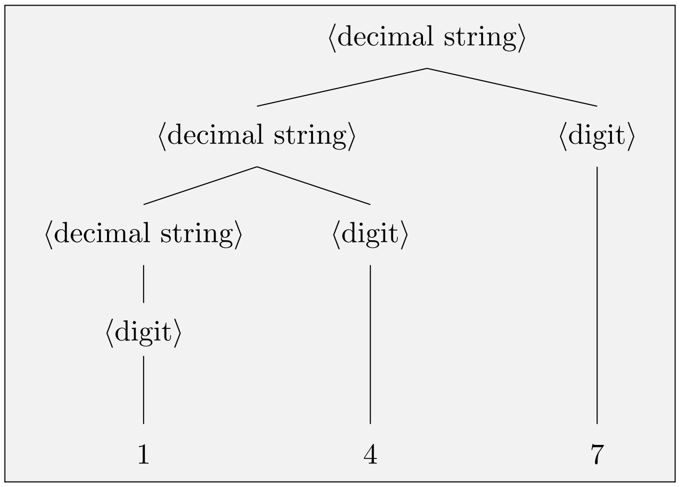


Figure 3.4: Parse Tree Generating 147.

This tree is called a *derivation tree* or *parse tree* for 147. We write the rules specifying the possible children of a node as follows.

$$\begin{aligned} \langle\text{decimal number}\rangle &\rightarrow \langle\text{digit}\rangle \mid \langle\text{decimal number}\rangle\langle\text{digit}\rangle \\ \langle\text{digit}\rangle &\rightarrow 0 \mid 1 \mid 2 \mid \cdots \mid 9 \end{aligned} \tag{3.1}$$

The vertical bar should be read as "or". So the meaning of

$$\langle\text{digit}\rangle \to 0 \mid 1 \mid 2 \mid \cdots \mid 9$$

is that a node labeled by ⟨digit⟩ will have a single child labeled by one of the digits 0–9. And the meaning of

$$\langle\text{decimal number}\rangle \to \langle\text{digit}\rangle \mid \langle\text{decimal number}\rangle\langle\text{digit}\rangle$$

is that a node labeled ⟨decimal number⟩ will either have a single child labeled by ⟨digit⟩, or two children, the left child being labeled by ⟨decimal number⟩ and the right child by ⟨digit⟩.

The angle brackets are not really necessary when the derivation of the string is presented in tree form. But often it is inconvenient to draw trees, and an inline presentation of the derivation is used instead. We show the inline derivation of the string 147 in Figure 3.5.

$$\begin{array}{rcl}
\langle\text{decimal string}\rangle & \Rightarrow & \langle\text{decimal string}\rangle\langle\text{digit}\rangle \\
& \Rightarrow & \langle\text{decimal string}\rangle 7 \\
& \Rightarrow & \langle\text{decimal string}\rangle\langle\text{digit}\rangle 7 \\
& \Rightarrow & \langle\text{decimal string}\rangle 47 \\
& \Rightarrow & \langle\text{digit}\rangle 47 \\
& \Rightarrow & 147
\end{array}$$

Figure 3.5: Inline Generation of 147.

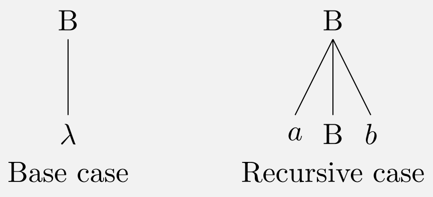
We write $\sigma \Rightarrow \tau$ if the string τ is the result of a single replacement in σ. The possible replacements are those given in Equation (3.1). Each replacement takes one occurrence of an item on the left-hand side of an arrow and replaces it with one of the items on the right-hand side; these are the items separated by vertical bars. Specifically, the possible replacements are to take one occurrence of one of:

⟨decimal string⟩	and replace it with one of ⟨digit⟩ or the sequence ⟨decimal string⟩⟨digit⟩.
⟨digit⟩	and replace it with one of 0–9.

Clearly, were we patient enough, in principle we could generate any number.

Next, let's look at an example of a non-regular language that can be defined recursively.

Example 3.2.1. $L = \{a^i b^i \mid i \geq 0\}$.
If $w \in L$ then either $i = 0$ and $w = \lambda$, or $i \geq 1$ and $w = avb$ with $v = a^{i-1}b^{i-1} \in L$. We can show these choices using a pair of trees as shown below.



It is convenient to write the recursive rules without drawing trees, which we do as follows. The term $\langle Balanced \rangle$ is the label for every internal node in the generating trees for L. Such a node either has a single child labeled λ, or it has three children labeled a, $\langle Balanced \rangle$, b, in left-to-right order. We write this as follows.

$$\langle Balanced \rangle \to \lambda \mid a \, \langle Balanced \rangle \, b.$$

We can then think of the string generation as a process of repeatedly replacing the $\langle Balanced \rangle$ term by either λ or $a \, \langle Balanced \rangle \, b$. For example, to generate $aabb$ we start (at the root) with the term $\langle Balanced \rangle$. We replace it by $a \, \langle Balanced \rangle \, b$; we replace the new instance of $\langle Balanced \rangle$ by $a \, \langle Balanced \rangle \, b$, yielding $aa \, \langle Balanced \rangle \, bb$; finally we replace this instance of $\langle Balanced \rangle$ by λ. For brevity, we will use B to denote $\langle Balanced \rangle$. We then write this string derivation as follows.

$$B \Rightarrow aBb \Rightarrow aaBbb \Rightarrow aabb.$$

Now we are ready to define *Context Free Grammars* (CFGs), G (which have nothing to do with graphs).

Definition 3.2.2. [CFG]. *A Context Free Grammar has four parts:*

1. *A set V of variables (such as $\langle Balanced \rangle$); note that V is not a vertex set here.*
   These are the labels for the internal nodes in a derivation tree.
   The individual variables are usually written using single capital letters, often from the end of the alphabet, e.g. X, Y, Z; no angle brackets are used here. This has little mnemonic value, but it is easier to write. If you do want to use longer variable names, I advise using angle brackets to delimit them.

2. *An alphabet T of terminals: these are the characters used to write the strings being generated. These, together with the empty string λ, are the labels for the leaves of the derivation tree. Usually, the terminals are written with small letters.*

3. *$S \in V$ is the start variable, the variable from which the string generation begins. This is the label for the root of every derivation tree.*
   The start variable does not have to be the letter S; this is simply the default choice.

4. *A set of rules R. Each rule has the form $X \to \sigma$, where $X \in V$ is a variable and $\sigma \in (T \cup V)^*$ is a string of variables and terminals, which could be λ, the empty string. In a derivation*

tree, if X is the label of a parent node, then the left-to-right sequence of labels for its children must be a string s, where s is the right-hand side of a rule with X on the left-hand side.

If we have several rules with the same left-hand side, for example $X \to \sigma_1, X \to \sigma_2, \cdots, X \to \sigma_k$, they can be written as $X \to \sigma_1 \mid \sigma_2 \mid \cdots \mid \sigma_k$ for short. The meaning is that an occurrence of X in a generated string can be replaced by any one of $\sigma_1, \sigma_2, \cdots, \sigma_k$. Different occurrences of X can be replaced by distinct σ_i, of course.

The generation of a string proceeds by a series of replacements starting from the string $s_0 = S$, and which, for $1 \le i \le k$, obtains s_i from s_{i-1} by replacing some variable X in s_{i-1} by one of the replacements $\sigma_1, \sigma_2, \cdots, \sigma_k$ for X, as provided by the rule collection R. We will write this as

$$S = s_0 \Rightarrow s_1 \Rightarrow s_2 \Rightarrow \cdots \Rightarrow s_k \text{ or } S \Rightarrow^* s_k \text{ for short.}$$

The language generated by grammar G, $L(G)$, is the set of strings of terminals that can be generated from G's start variable S:

$$L(G) = \{w \mid S \Rightarrow^* w \text{ and } w \in T^*\}.$$

In the next example, we look at the language of properly nested parentheses, namely strings in which each right parenthesis is matched to a distinct left parenthesis to its left. We can think of this language as consisting of those strings that can be obtained from legal arithmetic expressions from which we remove every character apart from the parentheses.

Example 3.2.3. Grammar G_2:
The grammar generating the language of properly nested parentheses. It has:
Variable set: $\{S\}$.
Terminal set: $\{(,)\}$.
Rules: $S \to (S) \mid SS \mid \lambda$.
Start variable: S.

Here are some example derivations.
$S \Rightarrow SS \Rightarrow (S)S \Rightarrow ()S \Rightarrow ()(S) \Rightarrow ()((S)) \Rightarrow ()(())$.
$S \Rightarrow (S) \Rightarrow (SS) \Rightarrow ((S)S) \Rightarrow (()S) \Rightarrow (()(S)) \Rightarrow (()())$.

The above examples are all reasonably straightforward. For more of a challenge let's look anew at the problem of generating all legal arithmetic expressions taking account of operator precedence.

Let's use E to denote the start variable for this language, that is starting from E we want to be able to generate all legal arithmetic expressions. We classify the possible expressions in terms of their priority.

- The highest priority expressions are of the form: Non-Negative Integer or (E)
  The second of these represents a legal arithmetic expression enclosed in parentheses.
- The mid-priority expressions are of the form: Operand $\times$ Operand or Operand $\div$ Operand
- And the low priority expressions are of the form: Operand $+$ Operand or Operand $-$ Operand

We let H, M and L be the variables that generate the high, mid, and low priority expressions, respectively.

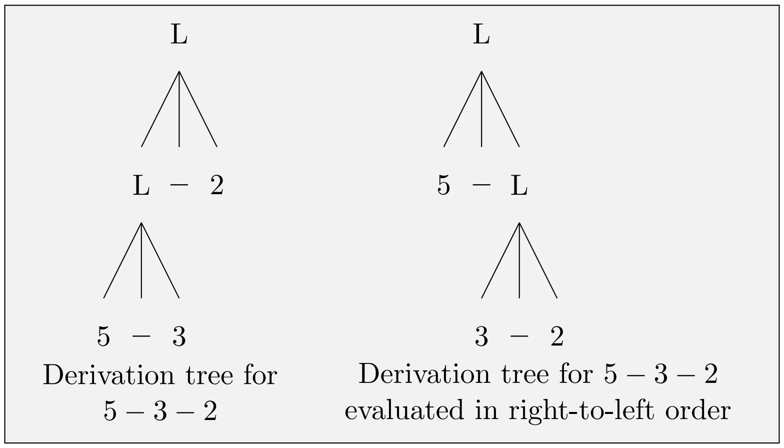

L
L − 2
5 − 3
Derivation tree for 5 − 3 − 2
L
5 − L
3 − 2
Derivation tree for 5 − 3 − 2 evaluated in right-to-left order

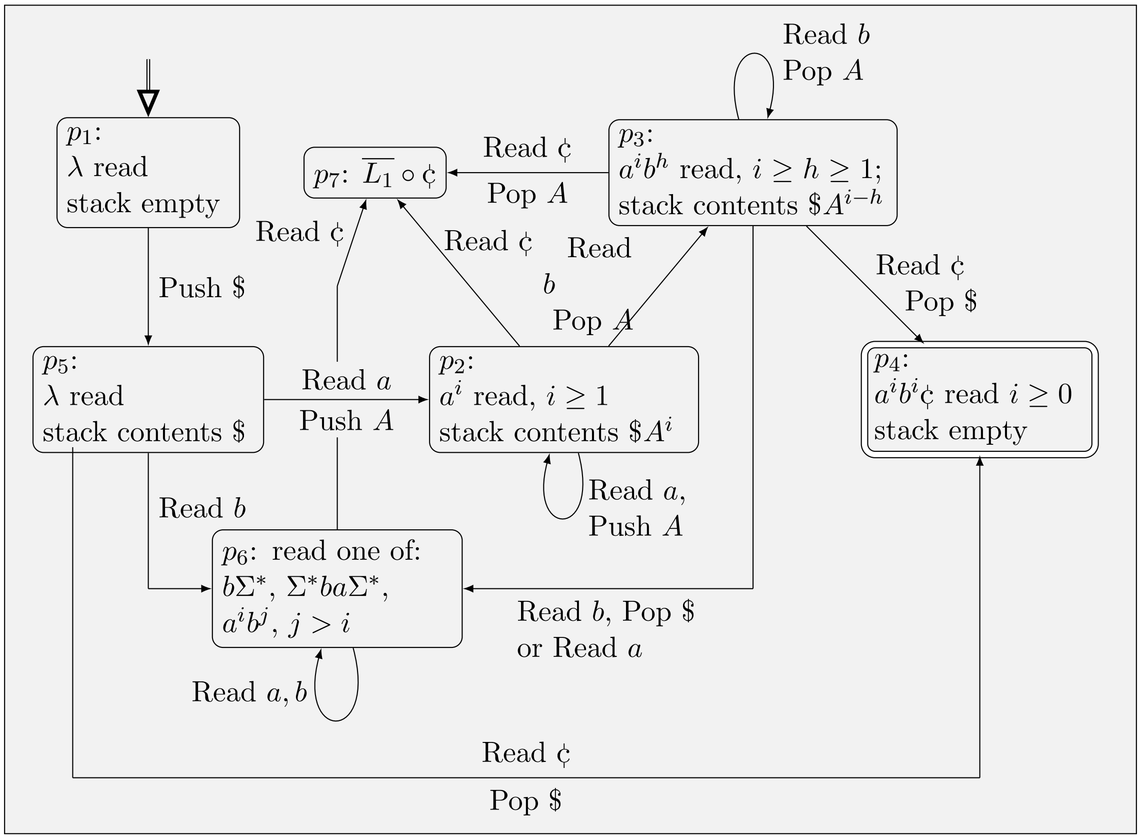
Figure 3.6: Derivation trees for arithmetic expressions.

We will need to place some constraints on the operands in order to ensure that the resulting expressions are legal arithmetic expressions. To provide some intuition, consider the expression $5-3-2$, which evaluates to 0 as the operators are evaluated in left-to-right order. We show this expression in a derivation tree in Figure 3.6 along with the tree for a right-to-left order of evaluation. As the latter expression it not legitimate, it has to be ruled out. The way we rule it out is to require the right operand of a plus or minus operation to be of higher priority, i.e. generated by H or M. Again, the right operand of a $\times$ or $\div$ has to have higher priority, i.e. be generated by an H. Similarly, its left operand cannot be of lower priority, i.e. it must be generated by an H or M. We specify the resulting grammar in Example 3.2.4 (also see Equation (3.1) for the grammar generating non-negative decimal numbers).

Example 3.2.4. Grammar G_3, which generates all legal arithmetic expressions involving non-negative integers and the operators $+, -, \times, \div$.
Variable set $\{E, N, D, H, M, L, A, P\}$, where N generates the non-negative integers, D generates the digits 0–9, H, M, L generate the high, mid, and low priority arithmetic expressions respectively, A generates $+$ and $-$, and P generates $\times$ and $\div$.
The start variable is E.
Terminal set $T = \{+, -, \times, \div, (,), 0\text{–}9\}$.
The rules are:

$$\begin{aligned}
E &\rightarrow H \,|\, M \,|\, L \\
H &\rightarrow N \,|\, (E) \\
M &\rightarrow MPH \,|\, HPH \\
L &\rightarrow EAM \,|\, EAH \\
A &\rightarrow + \,|\, - \\
P &\rightarrow \times \,|\, \div \\
N &\rightarrow D \,|\, ND \\
D &\rightarrow 0 \,|\, 1 \,|\, 2 \,|\, \cdots \,|\, 9
\end{aligned}$$

The derivation of $(1+2) \times 4 - 2$ is shown in Figure 3.7.



Read b
Pop A
p_1: λ read stack empty
p_7: $\overline{L_1} \circ ¢$
Read ¢
Pop A
p_3: $a^i b^h$ read, $i \geq h \geq 1$; stack contents $\$A^{i-h}$
Read ¢
Read ¢
Read b
Pop A
Read ¢
Pop \$
Push \$
p_5: λ read stack contents \$
Read a
Push A
p_2: a^i read, $i \geq 1$ stack contents $\$A^i$
p_4: $a^i b^i ¢$ read $i \geq 0$ stack empty
Read a, Push A
Read b
p_6: read one of: $b\Sigma^*$, $\Sigma^* ba \Sigma^*$, $a^i b^j$, $j > i$
Read b, Pop \$ or Read a
Read a, b
Read ¢
Pop \$

Figure 3.7: Derivation tree for $(1 + 2) \times 4 - 2$.

3.2.1 Closure Properties

Lemma 3.2.5. *Let G_A and G_B be CFGs generating languages A and B, respectively. Then there are CFGs generating $A \cup B$, $A \circ B$, A^*.*

Proof. Let $G_A = (V_A, T_A, R_A, S_A)$ and $G_B = (V_B, T_B, R_B, S_B)$. By renaming variables if needed, we can ensure that V_A and V_B are disjoint.

First, we show that $A \cup B$ is generated by the following grammar $G_{A\cup B}$.

$G_{A\cup B}$ has variable set $\{S_{A\cup B}\} \cup V_A \cup V_B$, terminal set $T_A \cup T_B$, start variable $S_{A\cup B}$, rules $R_A \cup R_B$ plus the rules $S_{A\cup B} \to S_A \mid S_B$.

To generate a string $w \in A$, $G_{A\cup B}$ performs a derivation with first step $S_{A\cup B} \Rightarrow S_A$, and then follows this with a derivation of w in G_A: $S_A \Rightarrow^* w$. So if $w \in A$, $w \in L(G_{A\cup B})$. Likewise, if $w \in B$, then $w \in L(G_{A\cup B})$ also. Thus $A \cup B \subseteq L(G_{A\cup B})$.

To show $L(G_{A\cup B}) \subseteq A \cup B$ is also straightforward. For if $w \in L(G_{A\cup B})$, then there is a derivation $S_{A\cup B} \Rightarrow^* w$. Its first step is either $S_{A\cup B} \Rightarrow S_A$ or $S_{A\cup B} \Rightarrow S_B$. Suppose it is $S_{A\cup B} \Rightarrow S_A$. Then the remainder of the derivation is $S_A \Rightarrow^* w$; this says that $w \in A$. Similarly, if the first step is $S_{A\cup B} \Rightarrow S_B$, then $w \in B$. Thus $L(G_{A\cup B}) \subseteq A \cup B$.

Next, we show that $A \circ B$ is generated by the following grammar $G_{A\circ B}$.

$G_{A\circ B}$ has variable set $S_{A\circ B} \cup V_A \cup V_B$, terminal set $T_A \cup T_B$, start variable $S_{A\circ B}$, and rules $R_A \cup R_B$ plus the rule $S_{A\circ B} \rightarrow S_A S_B$.

If $w \in A \circ B$, then $w = uv$ for some $u \in A$ and $v \in B$. So to generate w, $G_{A\circ B}$ performs the following derivation. The first step is $S_{A\circ B} \Rightarrow S_A S_B$; this is followed by a derivation $S_A \Rightarrow^* u$, which yields the string uS_B; this is then followed by a derivation $S_B \Rightarrow^* v$, which yields the string $uv = w$. Thus $A \circ B \subseteq L(G_{A\circ B})$.

To show $L(G_{A\circ B}) \subseteq A \circ B$ is also straightforward. For if $w \in L(G_{A\circ B})$, then there is a derivation $S_{A\circ B} \Rightarrow^* w$. The first step can only be $S_{A\circ B} \Rightarrow S_A S_B$. Let u be the string of terminals derived from S_A in the full derivation, and v be the string of terminals derived from S_B. So $uv = w$, $u \in A$ and $v \in B$. Thus $L(G_{A\circ B}) \subseteq A \circ B$.

The fact that there is a grammar G_{A^*} generating A^* we leave as an exercise for the reader. □

Lemma 3.2.6. $\{\lambda\}$*,* ϕ*,* $\{a\}$ *are all context free languages.*

Proof. The grammar with the single rule $S \rightarrow \lambda$ generates $\{\lambda\}$, the grammar with no rules generates ϕ, and the grammar with the single rule $S \rightarrow a$ generates $\{a\}$, where, in each case, S is the start variable. □

Corollary 3.2.7. *All regular languages have CFGs.*

Proof. It suffices to show that for any language represented by a regular expression r there is a CFG G_r generating the same language. This is done by means of a proof by induction on the number of operators in r. As this is identical in structure to the proof of Lemma 2.5.1, the details are left to the reader. □

3.3 Pushdown Automata

A Pushdown Automata, or PDA for short, is simply an NFA equipped with a single stack. As with an NFA, it moves from vertex to vertex as it reads its input, with the additional possibility of also pushing to and popping from its stack as part of a move from one vertex to another. As with an NFA, there may be several viable computation paths. In addition, as with an NFA, to recognize an input string w, a PDA M needs to have a recognizing path from its start vertex to a recognizing vertex, which it can traverse on input w.

Recall that a stack is an unbounded store which one can think of as holding the items it stores in a tower (or stack) with new items being placed (written) at the top of the tower and items being read and removed (in one operation) from the top of the tower. The first operation is called a *Push* and the second a *Pop*. For example, if we perform the sequence of operations Push(A), Push(B), Pop, Push(C), Pop, Pop, the 3 successive pops will return the items B, C, A. respectively. The successive states of the stack are shown in Figure 3.8.

Let's see how a stack allows us to recognize the following language $L_1 = \{a^i b^i \mid i \geq 0\}$. We start by explaining how to process a string $w = a^i b^i \in L$. As the PDA reads the initial string of a's in its input, it pushes a corresponding equal length string of A's onto its stack (one A for each a read). Then, as M reads the b's, it seeks to match them one by one against the A's on the stack (by popping one A for each b it reads). M recognizes its input exactly if the stack becomes empty on reading the last b.

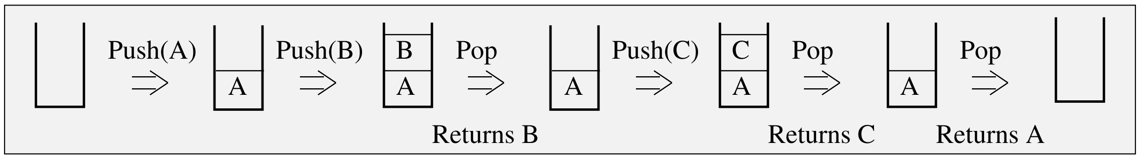


Figure 3.8: Stack Behavior.

In fact, PDAs are not allowed to use a *Stack Empty* test. We use a standard technique, which we call \$-*shielding*, to simulate this test. Given a PDA M on which we want to perform Stack Empty tests, we create a new PDA $\overline{M}$ which is identical to M apart from the following small changes. $\overline{M}$ uses a new, additional symbol on its stack, which we name \$. Then at the very start of the computation, $\overline{M}$ pushes a \$ onto its stack. This will be the only occurrence of \$ on its stack. Subsequently, $\overline{M}$ performs the same steps as M except that when M seeks to perform a Stack Empty test, $\overline{M}$ pops the stack and then immediately pushes the popped symbol back on its stack. The simulated stack is empty exactly if the popped symbol was a \$.

Next, we explain what happens with strings outside the language L_1. We do this by looking at several categories of strings in turn.

1. $a^i b^h$, $h < i$.
   After the last b is read, there will still be one or more A's on the stack, indicating the input is not in L_1.

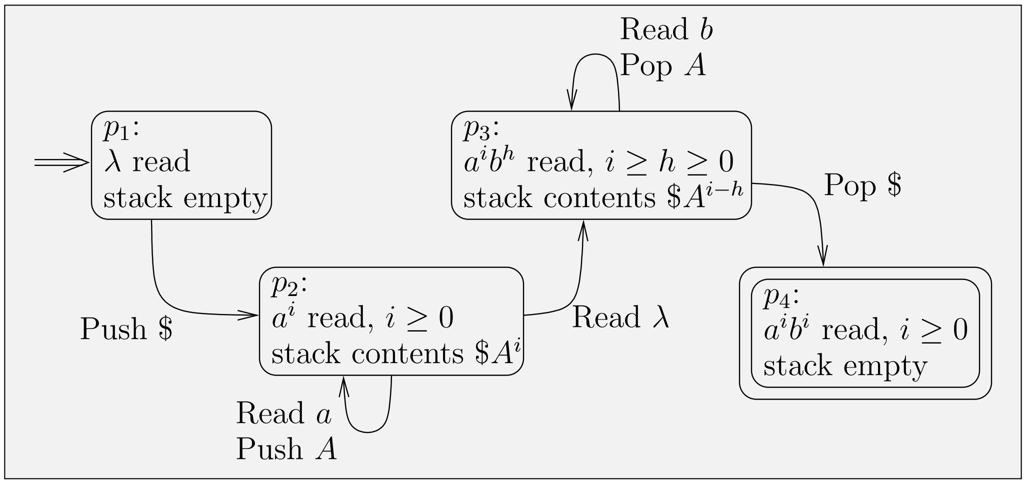
2. $a^i b^j$, $j > i$.
   On reading the $(i+1)$st b, there is an attempt to pop the now empty stack to find a matching A; this attempt fails, and again this indicates the input is not in L_1.

3. The only other possibility for the input is that it contains the substring ba; as already described, the processing consists of an a-reading phase, followed by a b-reading phase. The a in the substring ba is being encountered during the b-reading phase and once more this input is easily recognized as being outside L_1.

As with an NFA, we can specify the computation using a directed graph, with the edge labels indicating the actions to be performed when traversing the given edge. To recognize an input w, the PDA needs to be able to follow a path from its start vertex to a recognizing vertex starting with an empty stack, where the read labels along the path spell out the input, and the stack operations on the path are consistent with the stack's ongoing contents as the path is traversed. To emphasize the fact that a path traversal involves both input reading and stack operations, we will often call the traversed path a *computation path*. Also, as before, we say that a vertex v is a *destination* vertex for string s if the PDA can end up at vertex v after reading all of string s.

A PDA M_1 recognizing L_1 is shown in Figure 3.9. So as to be able to refer to the vertices, we have given them the names p_1–p_4. Their descriptors specify exactly those strings and the corresponding stack contents for which the vertex in question is a destination vertex.

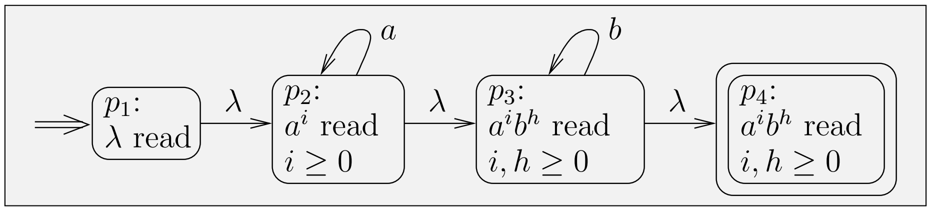
Definition 3.3.1. *(s, σ) is called a* data configuration *of PDA M at vertex p if M can end up at vertex p having σ on its stack and having read input string s (when starting the computation at its start vertex with an empty stack). We also refer to (p, s, σ) as a* configuration *of PDA M.*



p_1: λ read stack empty
Push \$
p_2: a^i read, $i \geq 0$ stack contents $\$A^i$
Read a Push A
Read λ
p_3: a^ib^h read, $i \geq h \geq 0$ stack contents $\$A^{i-h}$
Read b Pop A
Pop \$
p_4: a^ib^i read, $i \geq 0$ stack empty

Figure 3.9: PDA M_1 recognizing $L_1 = \{a^ib^i \,|\, i \geq 0\}$.

An initial understanding of what M_1 recognizes can be obtained by ignoring what the stack does and viewing the machine as just an NFA (i.e. using the same graph but with just the reads labeling the edges). See Figure 3.10 for the graph of the NFA N_1 derived from M_1. The significant



p_1: λ read
λ
p_2: a^i read $i \geq 0$
a
λ
p_3: a^ib^h read $i, h \geq 0$
b
λ
p_4: a^ib^h read $i, h \geq 0$

Figure 3.10: The NFA N_1 derived from M_1.

point is that if M_1 can reach vertex p on input w using computation path P then so can N_1 (for the same reads label P in both machines). It follows that any string recognized by M_1 is also recognized by N_1: $L(M_1) \subseteq L(N_1)$.

It is not hard to see that N_1 recognizes a^*b^*. If follows that M_1 recognizes a subset of a^*b^*. So to explain the behavior of M_1 in full it suffices to look at what happens on inputs the form a^ib^j, $i, j \geq 0$, which we do by examining five subcases that account for all such strings.

1. λ.
   M_1 starts at p_1. On pushing \$, p_2 and p_3 can be reached. Then on popping the \$, p_4 can be reached. Note that the specification of p_2 holds with $i = 0$, that of p_3 with $i = h = 0$, and that of p_4 with $i = 0$. Thus the specification at each vertex includes the case that the input is λ.

2. a^i, $i \geq 1$.
   To read a^i, M_1 needs to push \$, then follow edge (p_1, p_2), and then follow edge (p_2, p_2) i times. This puts $\$A^i$ on the stack. Thus on input a^i, p_2 can be reached and its specification

is correct. In addition, the edge to p_3 can be traversed without any additional reads or stack operations, and so the specification for p_3 with $h = 0$ is correct for this input.

3. $a^i b^h$, $1 \le h < i$.
   The only place to read b is on edge (p_3, p_3). Thus, for this input, M_1 reads a^i to bring it to p_3 and then follows (p_3, p_3) h times. This leaves $\$A^{i-h}$ on the stack, and consequently the specification of p_3 is correct for this input. Note that as $h < i$, edge (p_3, p_4) cannot be followed as \$ is not on the stack top.

4. $a^i b^i$, $i \ge 1$.
   After reading the i b's, M_1 can be at vertex p_3 as explained in (3). Now, in addition, edge (p_3, p_4) can be traversed and this pops the \$ from the stack, leaving it empty. So the specification of p_4 is correct for this input.

5. $a^i b^j$, $j > i$.
   On reading $a^i b^i$, M_1 can reach p_3 with the stack holding \$ or reach p_4 with an empty stack, as described in (4). From p_3 the only available move is to p_4, without reading anything further. At p_4 there is no move, so the rest of the input cannot be read, and thus no vertex can be reached on this input.

This is a very elaborate description which we certainly don't wish to repeat for each similar PDA. We can describe M_1's functioning more briefly as follows.

> M_1 checks that its input has the form a^*b^* (i.e. all the a's precede all the b's) using its underlying NFA (i.e. without using the stack). The underlying NFA is often called its *finite control*. In tandem with this, M_1 uses its \$-shielded stack to match the a's against the b's, first pushing the a's on the stack (it is understood that in fact A's are being pushed) and then popping them off, one for one, as the b's are read, confirming that the numbers of a's and b's are equal.

The detailed argument we gave above is understood, but not spelled out.

Now we are ready to define a PDA more precisely. As with an NFA, a PDA consists of a directed graph with one vertex, *start*, designated as the start vertex, and a (possibly empty) subset of vertices designated as the recognizing set, F, of vertices. As before, in drawing the graph, we indicate recognizing vertices using double circles and indicate the start vertex with a double arrow. Each edge is labeled with the actions the PDA performs on following that edge.

For example, the label on edge e might be: Pop A, read b, Push C, meaning that the PDA pops the stack, reads the next input character, and if the pop returns an A and the character read is a b, then the PDA can traverse e, which entails it pushing C onto the stack. Some or all of these values may be λ: Pop λ means that no Pop is performed, read λ that no read occurs, and Push λ that no Push happens. To avoid clutter, we usually omit the λ-labeled terms; for example, instead of Pop λ, read λ, Push C, we write Push C. Also, to avoid confusion in the figures, if there are multiple triples of actions that take the PDA from a vertex u to a vertex v, we use multiple edges from u to v, one for each triple.

In sum, a label, which specifies the actions accompanying a move from vertex u to vertex v, has up to three parts.

1. Pop the stack and check that the returned character has a specified value (in our example this is the value A).

2. Read the next character of input and check that it has a specified value (in our example, the value b).
3. Push a specified character onto the stack (in our example, the character C).

From an implementation perspective it may be helpful to think in terms of being able to peek ahead, so that one can see the top item on the stack without actually popping it, and one can see the next input character (or that one is at the end of the input) without actually reading forward.

One further rule is that an empty stack may not be popped.

A PDA also comes with an input alphabet Σ and a stack alphabet Γ (these are the symbols that can be written on the stack). It is customary for Σ and Γ to be disjoint, in part to avoid confusion. To emphasize this disjointness, we write the characters of Σ using lowercase letters and those of Γ using uppercase letters.

Because the stack contents make it difficult to describe the condition of the PDA after multiple moves, the transition function here will describe possible out-edges from single vertices only. Accordingly, $\delta(p, A, b) = \{(q_1, C_1), (q_2, C_2), \cdots, (q_l, C_l)\}$ indicates that the edges exiting vertex p and having both Pop A and read b in their label are the edges going to vertices $q_1, q_2, \cdots, q_l$ where the rest of the label for edge (p, q_i) includes Push C_i, for $1 \leq i \leq l$. That is $\delta(p, A, b)$ specifies the possible moves out of vertex p on popping character A and reading b. (Recall that one or both of A and b might be λ, as might some or all of the C_i's.)

We summarize this formally in the following definition.

Definition 3.3.2. [PDA]. A PDA M consists of a 6-tuple: $M = (\Sigma, \Gamma, V, \mathit{start}, F, \delta)$, where

1. Σ is the input alphabet,
2. Γ is the stack alphabet,
3. V is the vertex or state set,
4. $F \subseteq V$ is the recognizing vertex set,
5. *start* is the start vertex, and
6. δ is the transition function, which specifies the edges and their labels.

Recognition is defined as for an NFA, that is, PDA M recognizes input w if there is a path that M can follow on input w that takes M from its start vertex to a recognizing vertex. We call this path a *w-recognizing computation path* to emphasize that stack operations may occur in tandem with the reading of input w. More formally, M recognizes w if there is a path $\mathit{start} = p_0, p_1, \cdots, p_m$, where p_m is a recognizing vertex, the label on edge (p_{i-i}, p_i) is (Read a_i, Pop B_i, Push C_i), for $1 \leq i \leq m$, and the stack contents at vertex p_i is σ_i, for $0 \leq i \leq m$, where

1. $a_1 a_2 \cdots a_m = w$,
2. $\sigma_0 = \lambda$,
3. and Pop B_i, Push C_i applied to σ_{i-i} produces σ_i, for $1 \leq i \leq m$.

We write $L(M)$ for the language, or set of strings, recognized by M.

Next, we show some more examples of languages that can be recognized by PDAs.

Example 3.3.3. $L_2 = \{a^i c b^i \mid i \geq 0\}$.

PDA M_2 recognizing L_2 is shown in Figure 3.11. The processing by M_2 is similar to that of M_1. M_2 checks that its input has the form a^*cb^* using its finite control. In tandem, M_2 uses its \$-shielded stack to match the a's against the b's, first pushing the a's on the stack (actually A's are being pushed), then reads the c without touching the stack, and finally pops the a's off, one for one, as the b's are read, confirming that the numbers of a's and b's are equal.

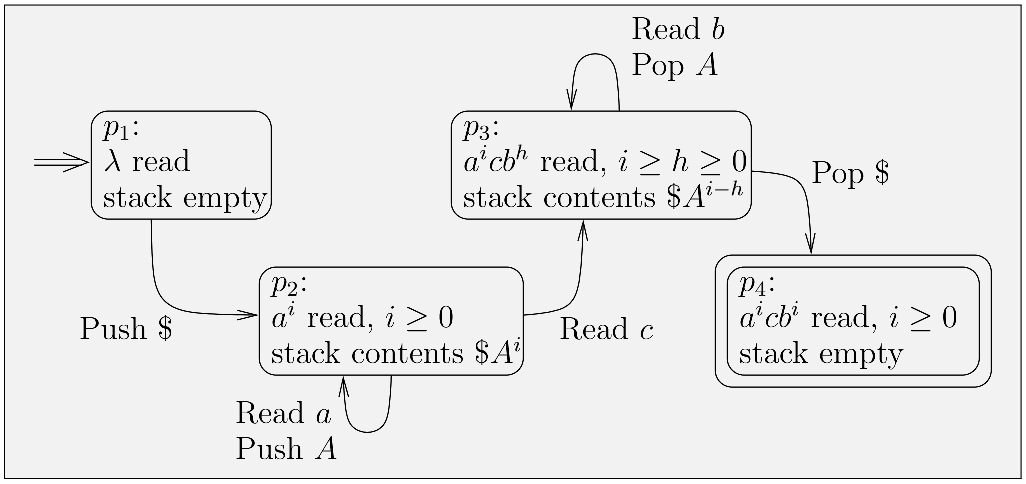


Figure 3.11: PDA M_2 recognizing $L_2 = \{a^i c b^i \mid i \geq 0\}$.

Example 3.3.4. $L_3 = \{wcw^R \mid w \in \{a,b\}^*\}$.

PDA M_3 recognizing L_3 is shown in Figure 3.12. M_3 uses its \$-shielded stack to match the w against the w^R, as follows. It pushes w on the stack (the end of the substring w being indicated by reaching the c). At this point, the stack content read from the top is $w^R\$$, so popping down to the \$ outputs the string w^R. This stack contents is readily compared to the string following the c. The input is recognized exactly if they match.

Example 3.3.5. $L_4 = \{ww^R \mid w \in \{a,b\}^*\}$.

PDA M_4 recognizing L_4 is shown in Figure 3.13. This is similar to Example 3.3.4. The one difference is that the PDA M_4 can decide at any point to stop reading w and begin reading w^R. Of course there is only one correct switching location, at most, but as M_4 does not know where it is, M_4 considers all possibilities by means of its nondeterminism.

Example 3.3.6. $L_5 = \{a^i b^j c^k \mid i = j \text{ or } i = k\}$.

PDA M_5 recognizing L_5 is shown in Figure 3.14. This is the union of languages $L_6 = \{a^i b^i c^k \mid i,k \geq 0\}$ and $L_7 = \{a^i b^j c^i \mid i,j \geq 0\}$, each of which is similar to L_2. M_5, the PDA recognizing L_5 uses submachines to recognize each of L_6 and L_7. M_5's first move from its start vertex is to traverse (Push \$)-edges to the start vertices of the submachines. The net effect is that M_5 recognizes the union of the languages recognized by the submachines. As the submachines are similar to M_2 they are not explained further.

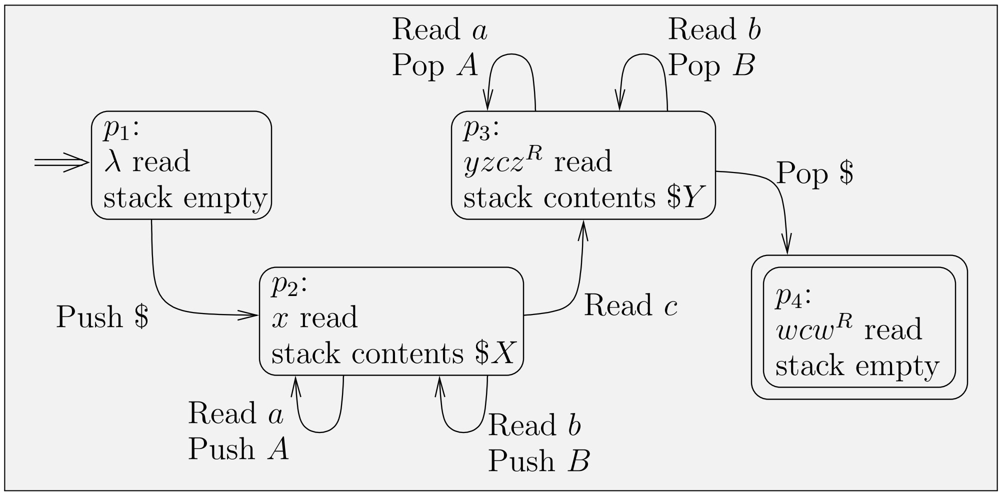


Figure 3.12: PDA M_3 recognizing $L_3 = \{wcw^R \,|\, w \in \{a,b\}^*\}$. Z denotes string z in capital letters.

3.3.1 Deterministic PDAs

Suppose that a PDA is constrained so that at each vertex, given the current character at the top of the stack and the next character of input, there is at most one possible move. It may be that this move does not read the next input character (i.e. it includes a Read λ) or it does not pop the stack (i.e. it includes a Pop λ). We call such a machine a *Deterministic PDA*, a DPDA for short.

This still leaves one possible ambiguity: whether to make additional moves once the input is fully read. To resolve this we treat the input as if there was an additional end-of-input character following the actual input. Thus instead of having an input w, the input will be of the form $w¢$ where ¢ is a character that appears only once at the very end of the input. The DPDA will be allowed to have edges whose labels include the action Read ¢. In addition, all recognizing vertices will be reached by following an edge labelled Read ¢, and further these vertices will have no out-edges. In fact, it is easy to see that one such vertex suffices.

Notice that if there is an edge labeled (Read λ, Pop B, Push C) exiting a vertex v, for some $B, C \in \Gamma \cup \{\lambda\}$, then there is no edge leaving v with label (Read a, Pop B, Push D) for any $a \in \Sigma$ and $D \in \Gamma \cup \{\lambda\}$. The following rules cover every possibility, and are exactly what is needed to ensure that there is always at most one move. Specifically, for each vertex v its outedges satisfy the following constraints:

- If there is an outedge labeled (Read λ, Pop λ, Push C) for some $C \in \Gamma \cup \{\lambda\}$, then this is v's only outedge.
- For all $a \in \Sigma \cup \{\lambda\}$, $B \in \Gamma \cup \{\lambda\}$, if there is an outedge labeled (Read a, Pop B, Push C) for some $C \in \Gamma \cup \{\lambda\}$, then there is no outedge labeled (Read a, Pop B, Push D), for any $D \neq C$, $D \in \Gamma \cup \{\lambda\}$.
- If there is an outedge labeled (Read λ, Pop B, Push C) for some $B \in \Gamma$ and $C \in \Gamma \cup \{\lambda\}$, then there is no outedge labeled (Read a, Pop B, Push D), for any $a \in \Sigma$ and $D \in \Gamma \cup \{\lambda\}$.
- If there is an outedge labeled (Read a, Pop λ, Push C) for some $a \in \Sigma$ and $C \in \Gamma \cup \{\lambda\}$, then

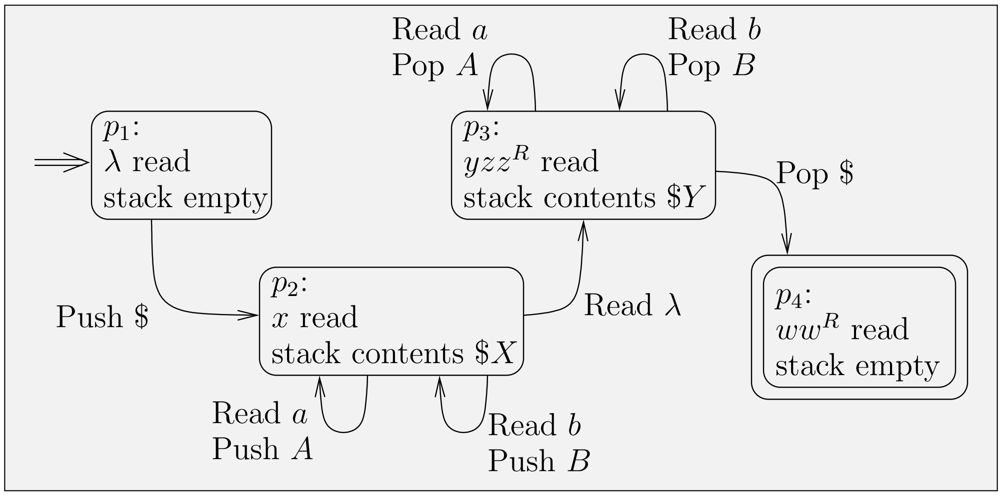


Figure 3.13: PDA M_4 recognizing $L_4 = \{ww^R \mid w \in \{a,b\}^*\}$. Z denotes string z in capital letters.

there is no outedge labeled (Read a, Pop B, Push D), for any $B \in \Gamma$ and $D \in \Gamma \cup \{\lambda\}$.

DPDAs are readily implemented, for as the input is read it suffices to keep track of the current destination vertex and the current stack contents. As the next move is always uniquely specified, the change to these values is always unambiguous.

Example 3.3.7. $L_1 = \{a^ib^i \mid i \geq 0\}$. The PDA in Figure 3.9, recognizing L_1, is not a DPDA, but a small modification turns it into a DPDA $\widetilde{M_1}$ recognizing L_1. The need for a modification is due to the λ-edge from p_2 to p_3, namely from the vertex where the a's are read to the vertex where the b's are read. We change the λ-edge to instead read the first b in the input, as shown in Figure 3.15. Also, to be able to distinguish λ from non-empty strings, we need to introduce a new vertex p_5 between p_1 and p_2. p_5 is reached by the initial Push \$. Then, if the input is λ, there is a direct transition to the recognizing vertex p_4. Otherwise an a is read (and an A pushed) to reach p_2.

Next we show that as with DFAs, we can have a sink vertex so as to ensure that there is always exactly one move to make.

Lemma 3.3.8. *Let M be a DPDA. There is another DPDA $\widetilde{M}$ with $L(M) = L(\widetilde{M})$ such that on every input, at every step of its computation, $\widetilde{M}$ has a unique move.*

Proof. For each possible action which is missing, we add an edge to the sink vertex labeled by that move, as specified by the following rule.
For each $a \in \Sigma$ and $B \in \Gamma$, add an edge from non-recognizing vertex v to the sink vertex labeled (Read a, Pop B, Push B) exactly if none of the following edges exiting v are present:

- an outedge labeled (Read a, Pop B, Push C) for any $C \in \Gamma \cup \{\lambda\}$,
- an outedge labeled (Read λ, Pop B, Push C) for any $C \in \Gamma \cup \{\lambda\}$,
- an outedge labeled (Read a, Pop λ, Push C) for any $C \in \Gamma \cup \{\lambda\}$,

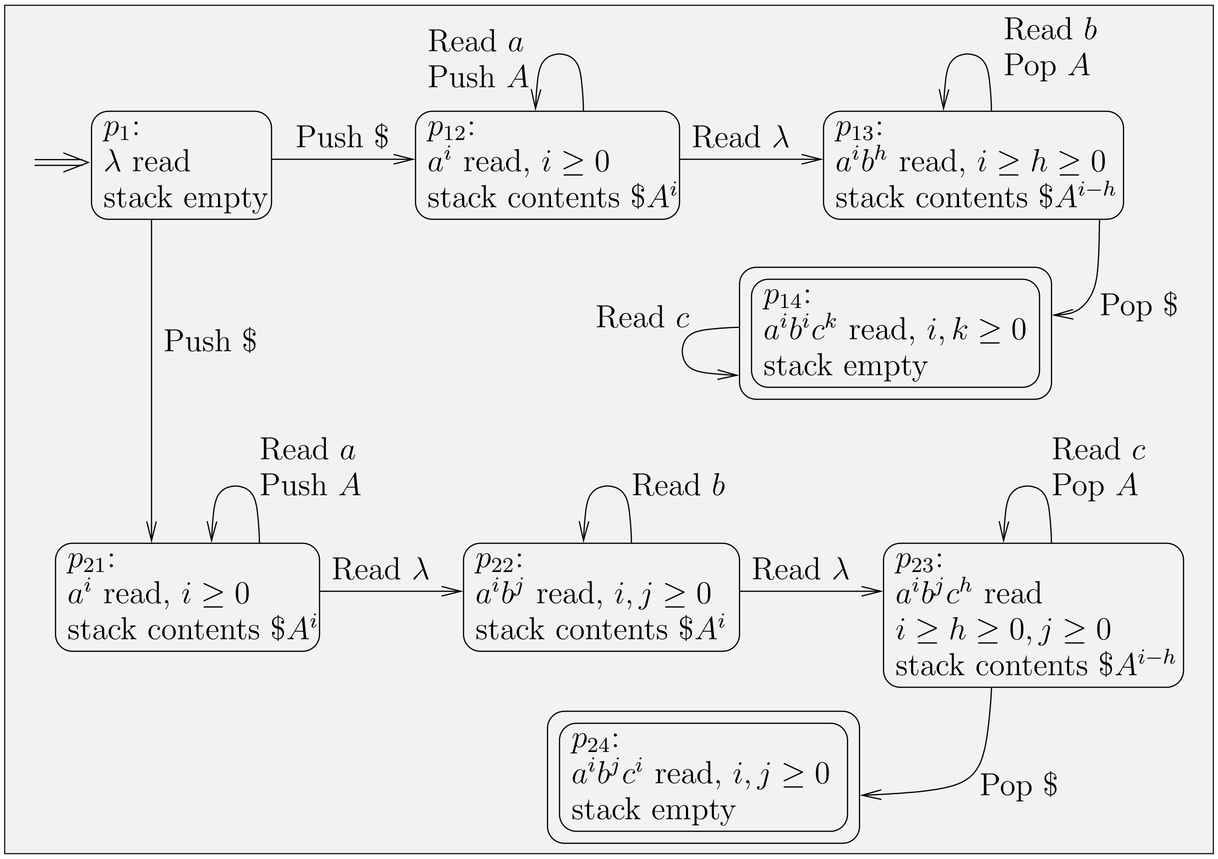

p_1: λ read stack empty
Push \$
Read a Push A
p_{12}: a^i read, $i \geq 0$ stack contents $\$A^i$
Read λ
Read b Pop A
p_{13}: a^ib^h read, $i \geq h \geq 0$ stack contents $\$A^{i-h}$
Pop \$
Read c
p_{14}: $a^ib^ic^k$ read, $i, k \geq 0$ stack empty
Push \$
Read a Push A
p_{21}: a^i read, $i \geq 0$ stack contents $\$A^i$
Read λ
Read b
p_{22}: a^ib^j read, $i, j \geq 0$ stack contents $\$A^i$
Read λ
Read c Pop A
p_{23}: $a^ib^jc^h$ read $i \geq h \geq 0, j \geq 0$ stack contents $\$A^{i-h}$
Pop \$
p_{24}: $a^ib^jc^i$ read, $i, j \geq 0$ stack empty

Figure 3.14: PDA M_5 recognizing $L_5 = \{a^ib^jc^k \mid i = j \text{ or } i = k\}$.

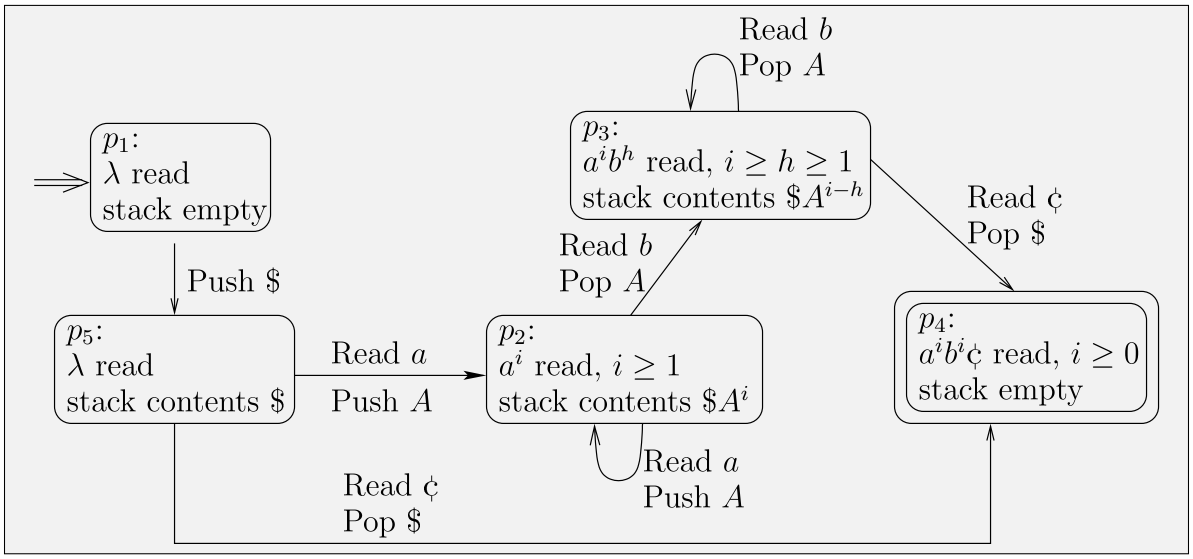

p_1: λ read stack empty
Push \$
p_5: λ read stack contents \$
Read a Push A
p_2: a^i read, $i \geq 1$ stack contents $\$A^i$
Read a Push A
Read b Pop A
p_3: a^ib^h read, $i \geq h \geq 1$ stack contents $\$A^{i-h}$
Read b Pop A
Read ¢ Pop \$
p_4: a^ib^i¢ read, $i \geq 0$ stack empty
Read ¢ Pop \$

Figure 3.15: DPDA recognizing $L_1 = \{a^ib^i \mid i \geq 0\}$.

- an outedge labeled (Read λ, Pop λ, Push C) for any $C \in \Gamma \cup \{\lambda\}$

In addition, for each $a \in \Sigma$, a self-loop is added to the sink vertex with label Read a. □

Example 3.3.9. DPDA for $L_1 = \{a^ib^i \mid i \geq 0\}$ with a unique move at every step.

In Figure 3.16 we show the result from applying Lemma 3.3.8 to the automata from Example 3.3.7.

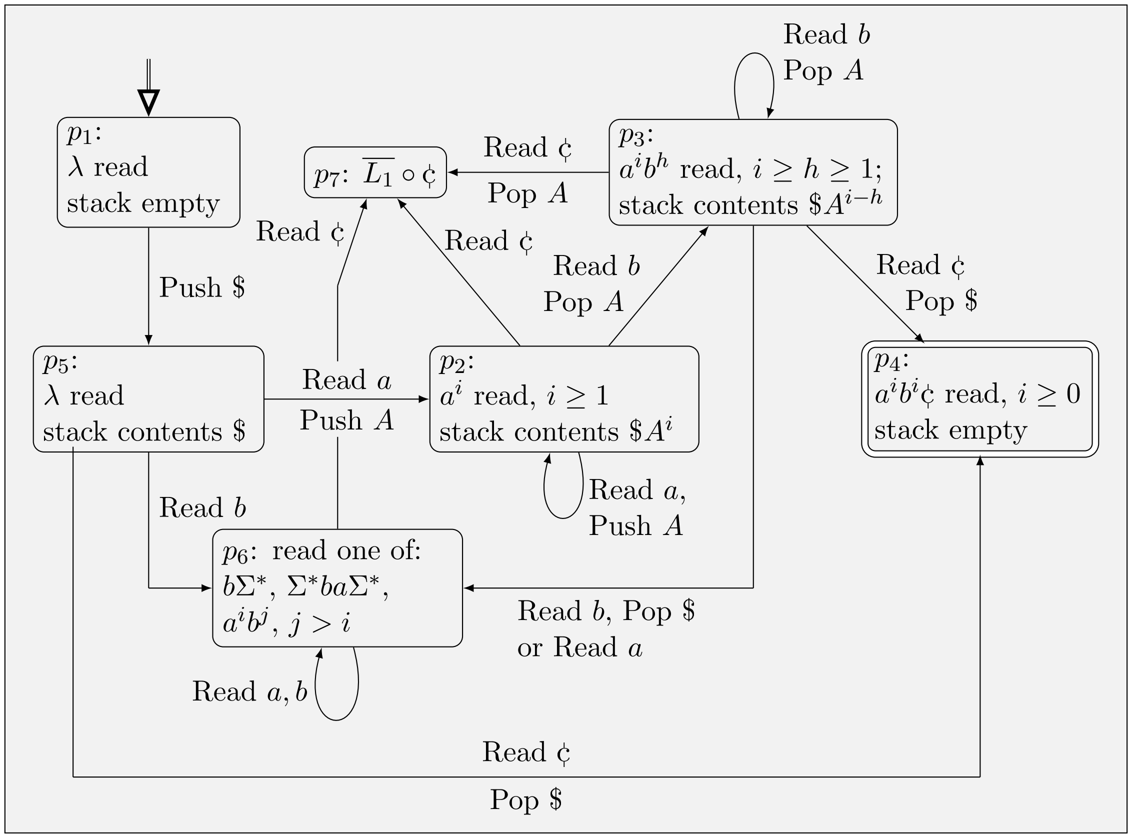

p_1: λ read stack empty
Push \$
p_5: λ read stack contents \$
Read a Push A
p_2: a^i read, $i \geq 1$ stack contents $\$A^i$
Read a, Push A
Read b Pop A
p_3: a^ib^h read, $i \geq h \geq 1$; stack contents $\$A^{i-h}$
Read b Pop A
Read ¢ Pop A
p_7: $\overline{L_1} \circ ¢$
Read ¢
Read ¢
Read ¢ Pop \$
p_4: $a^ib^i¢$ read, $i \geq 0$ stack empty
Read b
p_6: read one of: $b\Sigma^*$, $\Sigma^*ba\Sigma^*$, a^ib^j, $j > i$
Read a, b
Read b, Pop \$ or Read a
Read ¢
Pop \$

Figure 3.16: Unique move DPDA recognizing $L_1 = \{a^ib^i \mid i \geq 0\}$. Σ denotes the alphabet $\{a, b\}$.

3.4 Closure Properties

Lemma 3.4.1. *Let A and B be languages recognized by PDAs M_A and M_B, respectively, Then $A \cup B$ is also recognized by a PDA called $M_{A\cup B}$.*

Proof. The graph of $M_{A\cup B}$ consists of the graphs of M_A and M_B plus a new start vertex $\mathit{start}_{A\cup B}$, which is joined by λ-edges to the start vertices start_A and start_B of M_A and M_B, respectively. Its recognizing vertices are the recognizing vertices of M_A and M_B. The graph is shown in figure 3.17.

While it is clear that $L(M_{A\cup B}) = L(M_A) \cup L(M_B)$, we present the argument for completeness.

First, we show that $L(M_{A\cup B}) \subseteq L(M_A) \cup L(M_B)$, i.e. if $w \in L(M_{A\cup B})$ then $w \in L(M_A) \cup L(M_B)$. So suppose that $w \in L(M_{A\cup B})$. Then there is a w-recognizing computation path from $start_{A\cup B}$ to a recognizing vertex f. If f lies in M_A, then removing the first edge of P leaves a path P' from $start_A$ to f. Further, at the start of P', the stack is empty and nothing has been read, so P' is a w-recognizing path in M_A. That is, $w \in L(M_A)$. Similarly, if f lies in M_B, then $w \in L(M_B)$. In either case, $w \in L(M_A) \cup L(M_B)$.

Second, we show that $L(M_A) \cup L(M_B) \subseteq L(M_{A\cup B})$, i.e. if $w \in L(M_A) \cup L(M_B)$ then $w \in L(M_{A\cup B})$. Suppose that $w \in L(M_A)$. Then there is a w-recognizing computation path P' from $start_A$ to a recognizing vertex f in M_A. Adding the λ-edge $(start_{A\cup B}, start_A)$ to the beginning of P' creates a w-recognizing computation path in $M_{A\cup B}$, showing that $L(M_A) \subseteq L(M_{A\cup B})$. Similarly, if $w \in L(M_B)$, then $w \in L(M_{A\cup B})$. Thus if $w \in L(M_A) \cup L(M_B)$ then $w \in L(M_{A\cup B})$. □

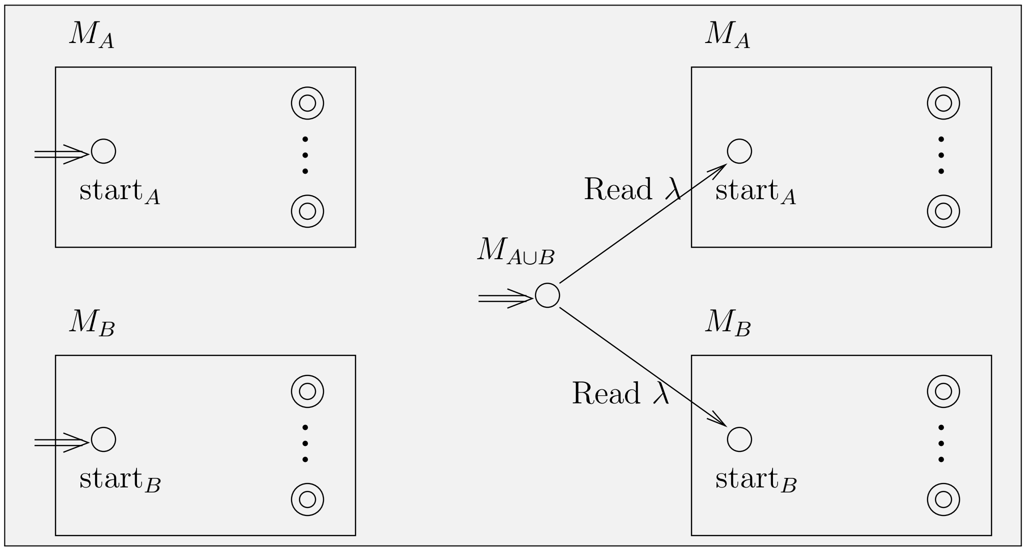


Figure 3.17: PDA $M_{A\cup B}$.

Our next construction is simplified by the following technical lemma.

Lemma 3.4.2. *Let PDA M recognize L. There is an L-recognizing PDA M' with the following properties: M' has only one recognizing vertex, $recognize_{M'}$, and M' will always have an empty stack when it reaches $recognize_{M'}$. M' is called a single-destination, empty-stack PDA.*

Proof. The idea is quite simple. M' simulates M using a \$-shielded stack. When M's computation is complete, M' moves to a new stack-emptying vertex, *stack-emptier*, at which M' empties its stack of everything apart from the \$-shield. To then move to $recognize_{M'}$, M' pops the \$, thus ensuring it has an empty stack when it reaches $recognize_{M'}$. M' is illustrated in Figure 3.18. More precisely, M' consists of the graph of M plus three new vertices; $start_{M'}$, *stack-emptier*, and $recognize_{M'}$. The following edges are also added: $(start_{M'}, start_M)$ labeled Push \$, λ-edges from each of M's recognizing vertices to *stack-emptier*, self-loops (*stack-emptier*, *stack-emptier*) labeled Pop X for each $X \in \Gamma$, where Γ is M's stack alphabet (so $X \neq \$$), and edge (*stack-emptier*, $recognize_{M'}$) labeled Pop \$.

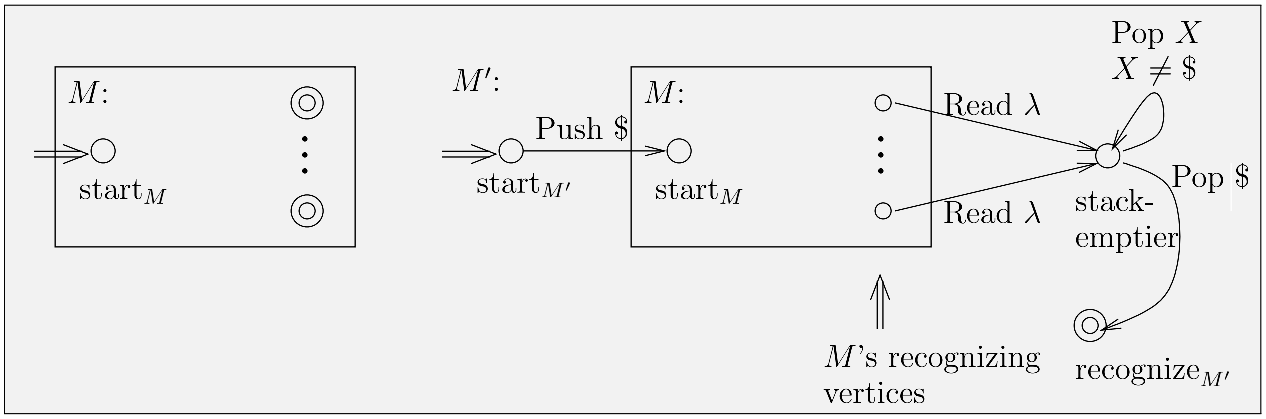


Figure 3.18: PDA M' for Lemma 3.4.2.

It is clear that $L(M) = L(M')$. Nonetheless, we present the argument for completeness.

First, we show that $L(M') \subseteq L(M)$, i.e. if $w \in L(M')$ then $w \in L(M)$ also. So let $w \in L(M')$. Let P' be a w-recognizing path in M' and let f be the recognizing vertex of M preceding *stack-emptier* on the path P'. Removing the first edge in P' and every edge including and after $(f, \textit{stack-emptier})$, leaves a path P which is a w-recognizing path in M. Thus $w \in L(M)$ and consequently $L(M') \subseteq L(M)$.

Now we show $L(M) \subseteq L(M')$, i.e. if $w \in L(M)$ then $w \in L(M')$ also. Let $w \in L(M)$ and let P be a w-recognizing path in M. Suppose that P ends with string s on the stack at recognizing vertex f. We add to P the edges $(\textit{start}_{M'}, \textit{start}_M)$, $(f, \textit{stack-emptier})$, $|s|$ self-loops at *stack-emptier*, and $(\textit{stack-emptier}, \textit{recognize}_{M'})$, yielding path P' in M'. By choosing the self-loops to be labeled with the characters of s^R in this order, which corresponds to popping s^R from the stack, we cause P' to be a w-recognizing path in M'. Thus $w \in L(M')$ and consequently $L(M) \subseteq L(M')$. □

Lemma 3.4.3. *Let A and B be languages recognized by PDAs. Then there is a PDA recognizing $A \circ B$.*

Proof. Let M_A and M_B be single-destination, empty-stack PDAs recognizing A and B, respectively.

We will construct PDA $M_{A\circ B}$ that recognizes $A \circ B$. It consists of M_A, M_B plus one λ-edge $(\textit{recognize}_A, \textit{start}_B)$. Its start vertex is $\textit{start}_A$ and its recognizing vertex is $\textit{recognize}_B$.

To see that $L(M_{A\circ B}) = A \circ B$ is straightforward.

First, we show that $L(M_{A\circ B}) \subseteq A \circ B$, i.e. if $w \in L(M_{A\circ B})$ then $w \in A \circ B$ also. So suppose that $w \in L(M_{A\circ B})$, Then there is a w-recognizing path P in $M_{A\circ B}$; P is formed from a path P_A in M_A going from $\textit{start}_A$ to $\textit{recognize}_A$ (and which therefore ends with an empty stack), λ-edge $(\textit{recognize}_A, \textit{start}_B)$, and a path P_B in M_B starting from $\textit{start}_B$ with an empty stack and going to $\textit{recognize}_B$. Let u be the sequence of reads labeling P_A and v those labeling P_B. It follows that P_A is u-recognizing, and P_B is v-recognizing (see Figure 3.19) and thus $u \in A$ and $v \in B$. In addition, $w = u\lambda v = uv$, which implies that $w = uv \in A \circ B$.

Next, we show that $A \circ B \subseteq L(M_{A\circ B})$. So suppose that $w = uv \in A \circ B$, where $u \in A$ and $v \in B$. Then there is a u-recognizing path P_A in M_A and a v-recognizing path P_B in M_B. We argue that the following path P is w-recognizing: P_A followed by the λ-edge $(\textit{recognize}_A, \textit{start}_B)$, followed

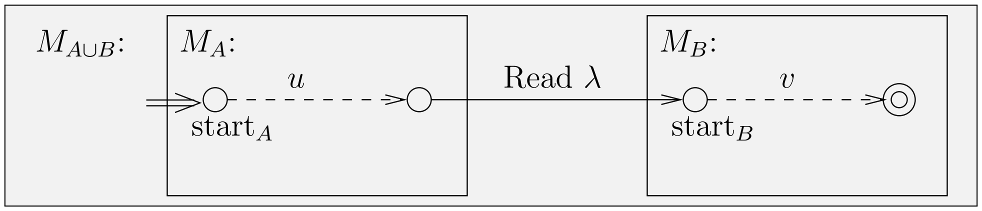


Figure 3.19: PDA $L(M_{A\circ B})$.

by P_B. Note that by construction the stack is empty when at node $recognize_A$, and hence it is also empty at node $start_B$. Consequently, computation path P in $M_{A\circ B}$ recognizes $u\lambda v = uv = w$, as claimed. It follows that $w = uv \in L(M_{A\circ B})$. □

Lemma 3.4.4. *Suppose that L is recognized by a PDA and suppose that R is a regular language. Then there is a PDA recognizing $L \cap R$.*

Proof. We illustrate the construction below in Example 3.4.5.

Let $M_L = (\Sigma, \Gamma_L, V_L, start_L, F_L, \delta_L)$ be a single-destination, empty-stack PDA recognizing M_L, and let R be recognized by DFA $M_R = (\Sigma, V_R, start_R, F_R, \delta_R)$. We will construct PDA $M_{L\cap R}$ that recognizes $L \cap R$. The vertices of $M_{L\cap R}$ will be 2-tuples, the first component corresponding to a vertex of M_L and the second component to a vertex of M_R. The computation of $M_{L\cap R}$, when looking at the first components along with the stack will be exactly the computation of M_L, and when looking at the second components, but without the stack, it will be exactly the computation of M_R. This leads to the following edges in $M_{L\cap R}$.

1. If M_L has an edge (u_L, v_L) with label (Pop A, Read b, Push C) and M_R has an edge (u_R, v_R) with label b, then $M_{L\cap R}$ has an edge $((u_L, u_R), (v_L, v_R))$ with label (Pop A, Read b, Push C).

2. If M_L has an edge (u_L, v_L) with label (Pop A, Read λ, Push C) then $M_{L\cap R}$ has an edge $((u_L, u_R), (v_L, u_R))$ with label (Pop A, Read λ, Push C) for every $u_R \in V_R$.

Example 3.4.5. PDA recognizing $L_1 \cap (aa)^*b^*$, where $L_1 = \{a^ib^i \mid i \geq 0\}$.

We take the PDA M_1 recognizing L_1 from Figure 3.9 and the DFA recognizing $(aa)^*b^*$ shown in Figure 3.20, and apply Lemma 3.4.4 to them. As it happens, only 5 vertices are reachable in the new automata; the other vertices are not shown.

The start vertex for $M_{L\cap R}$ is $(start_L, start_R)$ and its set of recognizing vertices is $F_L \times F_R$, the pairs of recognizing vertices, one from M_L and one from M_R, respectively.

Assertion. On input w, (v_L, v_R) is a destination vertex of $M_{L\cap R}$ if and only if v_L is a destination vertex of M_L and v_R a destination vertex of M_R.

Next, we argue that the assertion is true. For suppose that on input w, $M_{L\cap R}$ ends up at vertex (v_L, v_R) using computation path $P_{L\cap R}$. If we consider the first components of the vertices in $P_{L\cap R}$, we see that it is a computation path of M_L on input w with destination vertex v_L. Likewise, if we consider the second components of the vertices of $M_{L\cap R}$, we obtain a path P'_R. The only difficulty is that this path may contain repetitions of a vertex u_R corresponding to reads of λ by $M_{L\cap R}$. Eliminating such repetitions creates a path P_R in M_R with destination v_R and having the same label w as path P'_R.

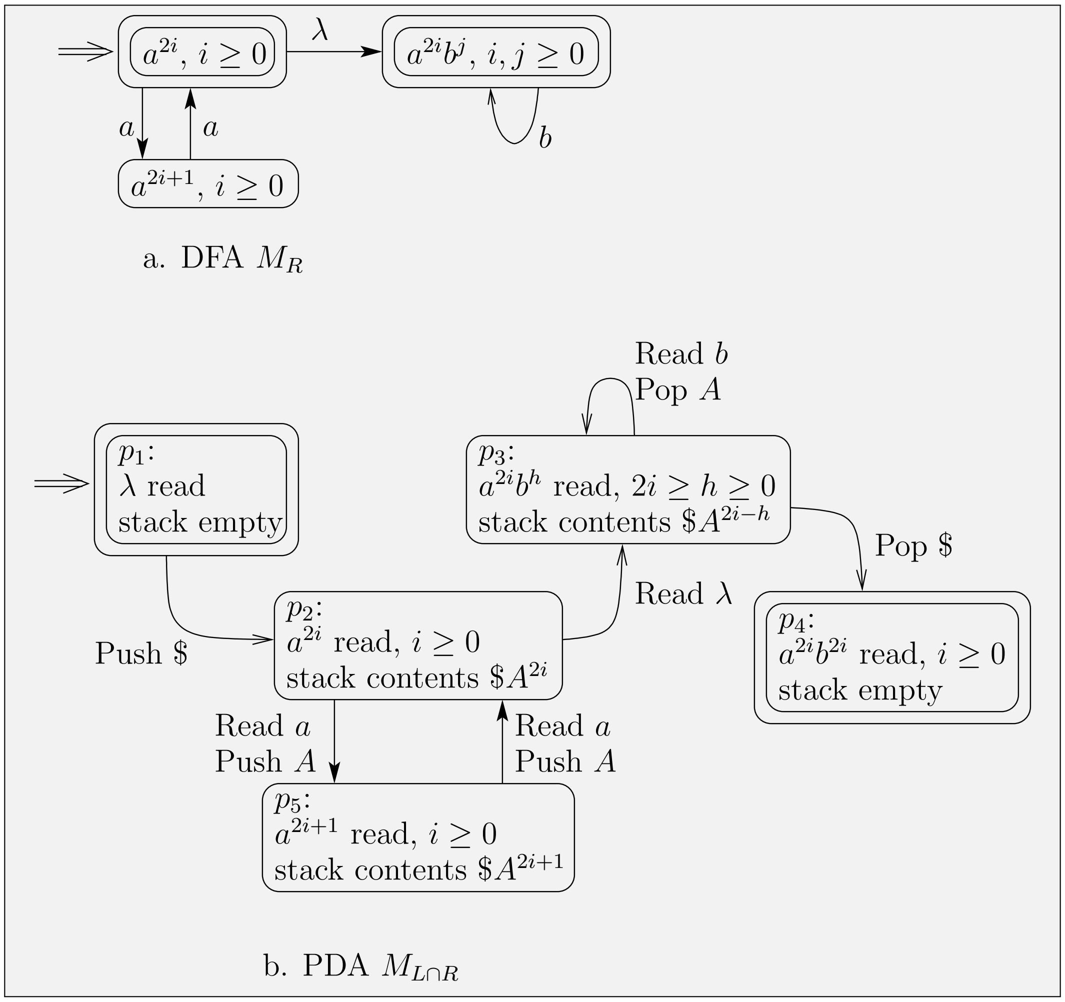


Figure 3.20: (a) DFA M_R recognizing $(aa)^*b^*$, (b) PDA $M_{L\cap R}$ recognizing $\{a^ib^i \mid i \geq 0\} \cap (aa)^*b^* = \{a^{2i}b^{2i} \mid i \geq 0\}$.

Conversely, suppose that M_L can end up at vertex v_L using computation path P_L and M_R can end up at vertex v_R using computation path P_R. Combining these paths, with care, gives a computation path P in $M_{L\cap R}$ on input w which ends at vertex (v_L, v_R). We proceed as follows. The first vertex is $(start_L, start_R)$. Then we traverse P_L and P_R in tandem. Either the next edges in P_L and P_R are both labeled by a Read b (simply a b on P_R) in which case we use Rule (1) above to give the edge to add to P, or the next edge on P_L is labeled by Read λ (together with a Pop and a Push possibly) and then we use Rule (2) to give the edge to add to P. In the first case we advance one edge on both P_L and P_R, in the second case we only advance on P_L. Clearly, the path ends at vertex (v_L, v_R) and reads input w.

It is now easy to see that $L(M_{L\cap R}) = L \cap R$. For on input w, $M_{L\cap R}$ ends up at a recognizing vertex $v \in F = F_L \times F_R$ if and only if on input w, M_L ends up at a vertex $v_L \in F_L$ and M_R ends up at a vertex $v_R \in F_R$. That is, $w \in L(M_{L\cap R})$ if and only if $w \in L(M_L) = L$ and $w \in L(M_R) = R$, or in other words $w \in L(M_{L\cap R})$ if and only if $w \in L \cap R$. □

Exercise. Show that if A is recognized by PDA M_A then there is a PDA M_{A^*} recognizing A^*.

3.4.1 Closure Properties for DPDAs

Lemma 3.4.6. *Let L be recognized by a DPDA M. Then L is also recognized by a PDA.*

Proof. This lemma is not quite immediate because of the addition of the ¢ to the DPDA input, but it is easy to simulate the missing ¢ in a PDA computation. We build PDA M' to recognize L as follows. Simply replace each Read ¢ in M with a Read λ. Clearly if $w \in L$ then there is a recognizing path P in M which goes from M's start vertex to a recognizing vertex. The Reads on this path, when concatenated, read the string $w¢$. The same path in M' will read the string w. Consequently M' recognizes w.

Conversely if M recognizes string w, the final edge on its recognizing path P has read label Read λ. In M this same final edge will have read label ¢, and as this is the only change, the path reads the string $w¢$, and thus M recognizes the string w. □

DPDAs have quite different closure properties from PDAs. As it happens, the complement of a language recognized by a DPDA is also recognized by a DPDA, but to prove this is quite difficult; the difficulties arise due to the possibility of a cycle on which nothing is read and which can be traversed infinitely often (see Problem 10). As we shall see later, this property does not hold for PDAs. Since every language recognized by a DPDA is recognized by a PDA, this implies PDAs recognize more languages that DPDAs, in contrast to the situation with DFAs and NFAs.

We now show that the intersection of two languages recognized by a DPDA need not be recognized by a DPDA, nor by any PDA.

Lemma 3.4.7. *There are languages K_1 and K_2 recognized by DPDAs, for which $K_3 = K_1 \cap K_2$ is not recognized by any DPDA, nor by any PDA.*

Proof. Let $K_1 = \{a^i b^j c^k \mid i = j \text{ and } i, j, k \geq 0\}$ and $K_2 = \{a^i b^j c^k \mid i = k \text{ and } i, j, k \geq 0\}$. Then $K_3 = K_1 \cap K_2 = \{a^i b^i c^i \mid i \geq 0\}$. K_1 and K_2 are recognized by DPDAs similar to the DPDA in Figure 3.15, but as we shall see in Section 3.6, K_3 is not a Context Free Language (CFL), and hence as we shall see in Section 3.7, is not recognized by any PDA, and hence not by any DPDA either.

□

Likewise, it need not be the case that DPDAs are closed under union (this is left as an exercise for the reader). However, all regular languages are recognized by DPDAs.

Lemma 3.4.8. *Every regular language is recognized by a DPDA.*

Proof. Let L be a regular language and let M be a DFA recognizing L. Add a new recognizing vertex to M with edges labeled ¢ from each of the old recognizing vertices, which cease to be recognizing. Then the modifed M is a DPDA recognizing L. □

3.5 Simpler Grammars

In this section, we show that we can impose some simple restrictions on the Context Free Grammars and still generate every CFL. The reason to do this is to simplify working with the grammars when we seek to prove certain properties of CFLs. The first will be a pumping Lemma for CFLs; the second is the demonstration that each CFL can be recognized by a PDA. We impose the following restrictions on the right-hand side of a rule.

i. Either a single terminal, e.g. $A \to a$.

ii. Or the empty string, e.g. $A \to \lambda$.

iii. Or one variable, e.g. $A \to B$.

iv. Or two variables, e.g. $A \to BC$.

Given a CFG G, we show how to obtain a simplified grammar G' obeying the above rules, while generating the same language, i.e. with $L(G) = L(G')$.

We proceed in a sequence of two steps which enforce the above criteria one by one; each step leaves the generated language unchanged.

Step 1 For each terminal a, we introduce a new variable, U_a say, add a rule $U_a \to a$, and for each occurrence of a in a string of length 2 or more on the right-hand side of a rule, replace a by U_a. Clearly, the generated language is unchanged.

Example: If we have the rule $A \to Ba$, this is replaced by $U_a \to a$, $A \to BU_a$.

This ensures that terminals on the right-hand sides of rules obey criteria (i) above.

Nothing needs to be done to enforce criteria (ii).

Step 2 For each rule with 3 or more variables on the right-hand side, we replace it with a new collection of rules obeying criteria (iii) above. Suppose there is a rule $U \to W_1W_2 \cdots W_k$, for some $k \geq 3$. Then we create new variables $X_2, X_3, \cdots, X_{k-1}$, and replace the prior rule with the rules:

$$U \to W_1X_2;\ X_2 \to W_2X_3;\ \cdots;\ X_{k-2} \to W_{k-2}X_{k-1};\ X_{k-1} \to W_{k-1}W_k$$

Clearly, the use of the new rules one after another, which is the only way they can be used, has the same effect as using the old rule $U \to W_1W_2 \cdots W_k$. Thus the generated language is unchanged.

This ensures criteria (iii) and (iv), namely that no right-hand side has more than 2 variables.

One can in fact constrain the right-hand sides of the rules even further while still being able to generate all CFLs. This is a form known as Chomsky Normal Form (CNF for short). In this form, the following additional criteria are imposed.

iv. The start symbol S can appear only on the LHS of a rule.

v. Every RHS of a rule has either no variables or two variables (i.e. a RHS consisting of one variable is not allowed).

vi. λ can appear on the RHS of a rule only for the start variable, i.e. the rule $S \to \lambda$ is allowed, but for $A \neq S$, $A \to \lambda$ is not allowed.

Demonstrating that every CFL has a CNF grammar is more of a challenge than the simplification we showed, and will be left to the exercises.

3.6 Showing Languages are not Context Free

We will do this with the help of a Pumping Lemma for Context Free Languages. To prove this lemma we need two results relating the height of derivation trees and the length of the derived strings, when using simplified grammars.

Lemma 3.6.1. *Let T be a derivation tree of height h for string $w \neq \lambda$ using simplified grammar G. Then $w \leq 2^{h-1}$.*

Proof. The result is easily confirmed by strong induction on h. Recall that the height of the tree is the length, in edges, of the longest root to leaf path.

The base case, $h = 1$, occurs with a tree of two nodes, the root and a leaf child. Here, w is the one terminal character labeling the leaf, so $|w| = 1 = 2^{h-1}$; thus the claim is true in this case.

Suppose that the result is true for trees of height k or less. We show that it is also true for trees of height $k + 1$. To see this, note that the root of T has at most two children, each one being the root of a subtree of height k or less. Thus, by the inductive hypothesis, each subtree derives a string of length at most 2^{k-1}, yielding that T derives a string of length at most $2 \cdot 2^{k-1} = 2^k$. This shows the inductive claim for $h = k + 1$.

It follows that the result holds for all $h \geq 1$. $\square$

Corollary 3.6.2. *Let w be the string derived by derivation tree T using simplified grammar G. If $|w| > 2^{h-1}$, then T has height at least $h + 1$ and hence has a root to leaf path with at least $h + 1$ edges and at least $h + 1$ internal nodes.*

Now let's consider the language $L = \{a^i b^i c^i \,|\, i \geq 0\}$ which is not a CFL as we shall proceed to show. Let's suppose for a contradiction that L were a CFL. Then it would have a simplified grammar G, with m variables say. Let $p = 2^m$.

To prove the pumping lemma for CFLs, we need the notion of a *minimal derivation.* A minimal derivation never derives the variable A from itself. For example, if the grammar included the rules $A \to B$ and $B \to A$ one could have the derivation $A \Rightarrow B \Rightarrow A$; but one could have a shorter derivation by simply omitting these two steps. Again, if one had the rules $A \to AC$ and $C \to \lambda$, one could have the derivation $A \Rightarrow AC \Rightarrow A$; again, one could have a shorter derivation by simply omitting these two steps.

Let's consider string $s = a^p b^p c^p \in L$, and look at a minimal derivation tree T for s. As $|s| > 2^{m-1}$, by Corollary 3.6.2, T has a root to leaf path with at least $m + 1$ internal nodes. Let P be a longest such path. Each internal node on P is labeled by a variable, and as P has at least $m + 1$ internal nodes, some variable must be used at least twice.

Working up from the bottom of P, let c be the first node to repeat a variable label. So on the portion of P below c each variable is used at most once. The derivation tree is shown in Figure 3.21.

Let d be the descendant of c on P having the same variable label as c, A say. Let w be the substring derived by the subtree rooted at d. Let vwx be the substring derived by the subtree rooted at c (so v, for example, is derived by the subtrees hanging from P to its left side on the portion of P starting at c and ending at d's parent). Let $uvwxy$ be the string derived by the whole tree.

Observation 1. The height of c is at most $m + 1$. This follows because P is a longest root to leaf path and because no variable label is repeated on the path below node c. Hence, by Lemma 3.6.1,

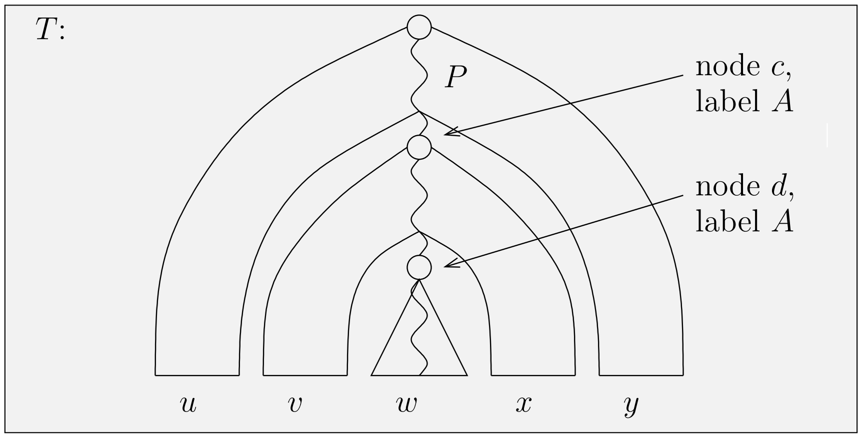


Figure 3.21: Derivation Tree for $s \in L$.

$|vwx| \leq 2^m = p$.

Observation 2. Either $|v| \geq 1$ or $|x| \geq 1$ (or both). We abbreviate this as $|vx| \geq 1$. This follows because the derivation is minimal, and if both $v = \lambda$ and $x = \lambda$ the derivation would not be minimal.

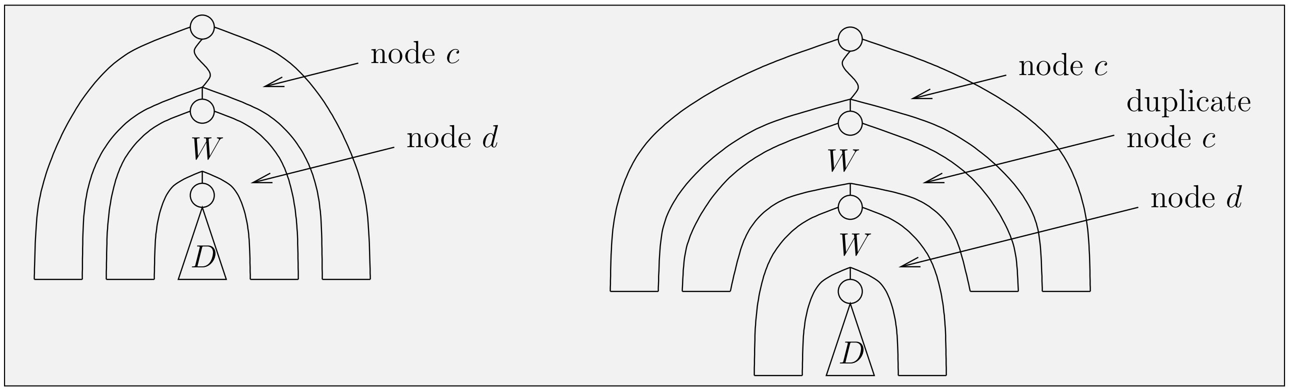


Figure 3.22: Duplicating wedge W.

Let's consider replicating the middle portion of the derivation tree, namely the wedge W formed by taking the subtree C rooted at c and removing the subtree D rooted at d, to create a derivation tree for a longer string, as shown in Figure 3.22.

Note that W's root has label A, and the vertex reached by the one edge descending down from W, W's child for short, is also labeled by A. Thus W's child could be the root of subtree D, but it could also be the root of another copy of W. Consequently, W plus a nested subtree D is a legitimate replacement for subtree D. The resulting tree, with two copies of W, one nested inside the other, is a derivation tree for $uvvwxxy$. Thus $uvvwxxy \in L$.

Clearly, we could duplicate W more than once, or remove it entirely, showing that all the strings $uv^iwx^iy \in L$, for any integer $i \geq 0$.

Now let's see why $uvvwxxy \notin L$. Note that we know that $|vwx| \leq p$ and $|vx| \geq 1$, by Observations 1 and 2. Further recall that $a^pb^pc^p = uvwxy$. As $|vwx| \leq p$, it is contained entirely in either one or two adjacent blocks of letters, as illustrated in Figure 3.23.

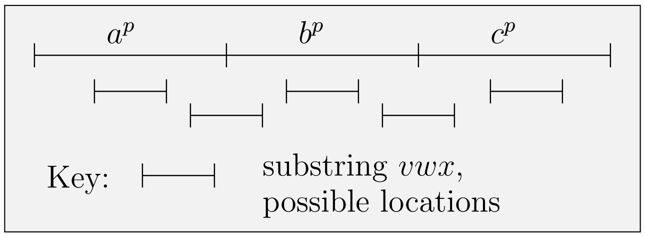


Figure 3.23: Possible Locations of vwx in $a^pb^pc^p$.

Therefore, when v and x are duplicated, as $|vx| \geq 1$, the number of occurrences of one or two of the characters increases, but not of all three. Consequently, in $uvvwxxy$ there are not equal numbers of a's, b's, and c's, and so $uvvwxxy \notin L$.

We have shown both that $uvvwxxy \in L$ and $uvvwxxy \notin L$. This contradiction means that the original assumption (that L is a CFL) is mistaken. We conclude that L is not a CFL.

We are now ready to prove the Pumping Lemma for Context Free Languages in the style of the above argument. It will provide a tool to show many languages are not Context Free.

Lemma 3.6.3. (Pumping Lemma for Context Free Languages.) *Let L be a Context Free Language. Then there is a constant $p = p_L$, such that if $s \in L$ and $|s| \geq p$, then s is* pumpable*, that is s can be written in the form $s = uvwxy$ with*

1. $|vx| \geq 1$.
2. $|vwx| \leq p$.
3. *For every integer i, $i \geq 0$, $uv^iwx^iy \in L$.*

Proof. Let G be a simplified grammar for L, with m variables say. Let $p = 2^m$. Let $s \in L$, where $|s| \geq p$, and let T be a derivation tree for s. As $|s| > 2^{m-1}$, by Corollary 3.6.2, T has a root to leaf path with at least $m+1$ internal nodes. Let P be a longest such path. Each internal node on P is labeled by a variable, and as P has at least $m+1$ internal nodes, some variable must be used at least twice.

Working up from the bottom of P, let c be the first node to repeat a variable label. So on the portion of P below c each variable is used at most one. Thus c has height at most $m+1$. The derivation tree is shown in Figure 3.21. Let d be the descendant of c on P having the same variable label as c, A say. Let w be the substring derived by the subtree rooted at d. Let vwx be the substring derived by the subtree rooted at c. Let $uvwxy$ be the string derived by the whole tree.

By Observation 1, c has height at most $m+1$; hence, by Lemma 3.6.1, vwx, the string c derives, has length at most $2^m = p$. This shows Property (2). Property (1) is shown in Observation 2, above.

Finally, we show property (3). Let's replicate the middle portion of the derivation tree, namely the wedge W formed by taking the subtree C rooted at c and removing the subtree D rooted at d, to create a derivation tree for a longer string, as shown in Figure 3.22.

We can do this because the root of the wedge is labeled by A and hence W plus a nested subtree D is a legitimate replacement for subtree D. The resulting tree, with two copies of W, one nested inside the other, is a derivation tree for $uvvwxxy$. Thus $uvvwxxy \in L$.

Clearly, we could duplicate W more than once, or remove it entirely, showing that all the strings $uv^iwx^iy \in L$, for any integer $i \geq 0$. □

Next, we demonstrate by example how to use the Pumping Lemma to show languages are not context free. The argument structure is identical to that used in applying the Pumping Lemma for regular languages.

Example 3.6.4. $J = \{ww \mid w \in \{a,b\}^*\}$.
We will show that J is not context free.

Step 1. Suppose, for a contradiction, that J were context free. Then, by the Pumping Lemma, there is a constant $p = p_J$ such that for any $s \in J$ with $|s| \geq p$, s is pumpable.

Step 2. choose $s = a^{p+1}b^{p+1}a^{p+1}b^{p+1}$. Clearly $s \in J$ and $|s| \geq p$, so s is pumpable.

Step 3. As s is pumpable we can write s as $s = uvwxy$ with $|vwx| \leq p$, $|vx| \geq 1$ and $uv^iwx^iy \in J$ for all integers $i \geq 0$. Also, by condition (3) with $i = 0$, $s' = uwy \in J$. We argue next that in fact $s' \notin J$. As $|vwx| \leq p$, vwx can overlap one or two adjacent blocks of characters in s but no more. Now, to obtain s' from s, v and x are removed. This takes away characters from one or two adjacent blocks in s, but at most p characters in all (as $|vx| \leq p$). Thus s' has four blocks of characters, with either one of the blocks of a's shorter than the other, or one of the blocks of b's shorter than the other, or possibly both of these. In every case $s' \notin J$. We have shown that both $s' \in J$ and $s' \notin J$. This is a contradiction,

Step 4. The contradiction shows that the initial assumption was mistaken. Consequently, J is not context free.

Comment. Suppose, by way of example, that vwx overlaps the first two blocks of characters. It would be incorrect to assume that v is completely contained in the block of a's and x in the block of b's. Further, it may be that $v = \lambda$ or $x = \lambda$ (but not both). All you know is that $|vwx| \leq p$ and that one of $|v| \geq 1$ or $|x| \geq 1$. Don't assume more than this.

The following example uses the yet to be proven result that if L is context free and R is regular then $L \cap R$ is also context free.

Example 3.6.5. $H = \{z \mid z \in \{a,b,c\}^*$ and z has equal numbers of a's, b's and c's$\}$.
Consider $H \cap a^*b^*c^* = L = \{a^ib^ic^i\}$. If H were context free, then L would be context free too. But we have already seen that L is not context free. Consequently, H is not context free either.
This could also be shown directly by pumping on string $s = a^pb^pc^p$.

The next example shows that there can be several cases which depend on the positioning of vwx, and the various cases may need to be handled differently in terms of whether to pump up or down.

Example 3.6.6. $K = \{a^i b^j c^k \mid i < j < k\}$.
We show that K is not context free.

Step 1. Suppose, for a contradiction, that K were context free. Then, by the Pumping Lemma, there is a constant $p = p_K$ such that for any $s \in K$ with $|s| \geq p$, s is pumpable.

Step 2. Choose $s = a^p b^{p+1} c^{p+2}$. Clearly, $s \in K$ and $|s| \geq p$, so s is pumpable.

Step 3. As s is pumpable we can write s as $s = uvwxy$ with $|vwx| \leq p$, $|vx| \geq 1$ and $uv^iwx^iy \in K$ for all integers $i \geq 0$.

As $|vwx| \leq p$, vwx can overlap one or two blocks of the characters in s, but not all three. Our argument for obtaining a contradiction depends on the position of vwx.

Case 1. vwx does not overlap the block of c's.
Then consider $s' = uvvwxxy$. As s is pumpable, by Condition (3) with $i = 2$, $s' \in K$. We argue next that in fact $s' \notin K$. As v and x have been duplicated in s', the number of a's or the number of b's is larger than in s (or possibly both numbers are larger); but the number of c's does not change. If the number of b's has increased, then s' has at least as many b's as c's, and then $s' \notin K$. Otherwise, the number of a's increases, and the number of b's is unchanged, so s' has at least as many a's as b's, and again $s' \notin K$.

Case 2. vwx does not overlap the block of a's.
Then consider $s' = uwy$. Again, as s is pumpable, by Condition (3) with $i = 0$, $s' \in K$. Again, we show that in fact $s' \notin K$. To obtain s' from s, the v and the x are removed. So in s' either the number of c's is smaller than in s, or the number of b's is smaller (or both). But the number of a's is unchanged. If the number of b's is reduced, then s' has at least as many a's as b's, and so $s' \notin K$. Otherwise, the number of c's decreases and the number of b's is unchanged; but then s' has at least as many b's as c's, and again $s' \notin K$.

In either case, a pumped string s' has been shown to be both in K and not in K. This is a contradiction.

Step 4. The contradiction shows that the initial assumption was mistaken. Consequently, K is not context free.

When applying the Pumping Lemma, it seems a nuisance to have to handle the cases where one of v or x may be the empty string, and fortunately, we can prove a variant of the Pumping Lemma in which both $|v| \geq 1$ and $|x| \geq 1$.

Lemma 3.6.7. (Variant of the Pumping Lemma for Context Free Languages.) *Let L be a Context Free Language. Then there is a constant $p = p_L$ such that if $s \in L$ and $|s| \geq p$ then s is* pumpable, *that is s can be written in the form $s = uvwxy$ with*

1. $|v|, |x| \geq 1$.
2. $|vwx| \leq p$.
3. *For every integer i, $i \geq 0$, $uv^iwx^iy \in L$.*

Proof. The structure of this proof is similar to that for the standard pumping lemma, but it is a bit more involved.

Again, let G be a simplified CFG grammar generating L and suppose G has m variables. Consider a minimal derivation tree T for s using grammar G. Let P be a longest root-to-leaf path in T. Suppose P has at least $2m+1$ internal nodes, and hence at least one variable repeated three times. Note that by Corollary 3.6.2, having $|s| \geq 2^{2m}$ ensures P is this long. So define $p = 2^{2m}$ here.

Traversing up P from its leaf end, let A be the first variable to be repeated twice. We will write $s = uv_2v_1wx_1x_2y$, where w is the string generated by the bottommost A on P, v_1wx_1 is the string generated by the next bottommost A on P, and $v_2v_1wx_1x_2$ is the string generated by the second bottommost A.

As before, we deduce the subtree rooted at the node labeled by the second bottommost A has edge-height at most $2m+1$ and hence the string it generates, $v_2v_1wx_1x_2$, has length at most $2^{2m} = p$.

Also, as the derivation tree is minimal, both $|v_2x_2| \geq 1$ and $|v_1x_1| \geq 1$.

Now we need to consider three cases to complete the result.

Case 1. Both $|v_2v_1| \geq 1$ and $|x_1x_2| \geq 1$.
As before, by duplicating the wedge between the top and bottom of the three A's, we see that $u(v_2v_1)^iw(x_1x_2)^iy \in L$ for all integer $i \geq 0$. Setting $v = v_2v_1$ and $x = x_1x_2$ gives the desired result.

Case 2. $|x_1x_2| = 0$.
Then both $|v_1| \geq 1$ and $|v_2| \geq 1$. In this case we duplicate two wedges separately, one between the top and middle A's and one between the middle and bottom A's. This yields that $u(v_2)^i(v_1)^i(wy) \in L$ for all $i \geq 0$. On setting $\tilde{u} = u$, $\tilde{v} = v_2$, $\tilde{w} = \lambda$, $\tilde{x} = v_1$, and $\tilde{y} = wy$, we obtain that $\tilde{u}\tilde{v}^i\tilde{w}\tilde{x}^i\tilde{y} \in L$, which is the desired result.

Case 3. $|v_2v_1| = 0$.
This is entirely analogous to Case 2. ☐

3.7 PDAs Recognize exactly the Context Free Languages

The notion of a leftmost derivation is useful for the next construction.

Definition 3.7.1. *A leftmost derivation of a string s by a CFG G is a derivation in which the leftmost variable in the currently derived string is always the one to be replaced.*

The derivations in Example 3.2.3 are both leftmost derivations. Also, a leftmost derivation corresponds to a depth first traversal of the derivation tree.

Lemma 3.7.2. *Let L be a CFL. Then there is a PDA M_L that recognizes L.*

Proof. Let L be generated by simplified grammar $G_L = (V_L, T, R_L, S_L)$. The corresponding M_L is illustrated in Figure 3.24. The computation centers on vertex *Main.* Each return visit to *Main* corresponds to the simulation of one step of a leftmost derivation in G_L. Specifically

Claim 3.7.3. *Let $s \in T^*$ and $\sigma \in V^*$. Then*

G_L generates string $s\sigma$
exactly if
M_L can reach vertex Main *with data configuration $(s, \$\sigma^R)$.*

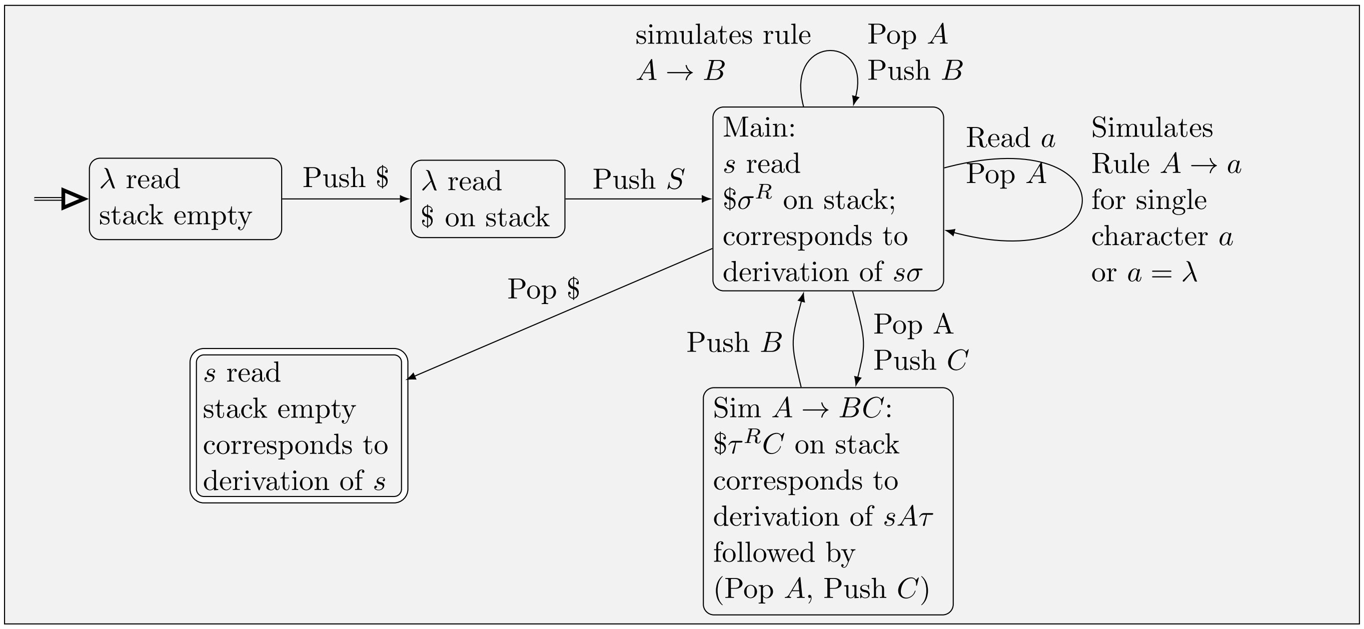

simulates rule $A \to B$
Pop A Push B
λ read stack empty
Push \$
λ read \$ on stack
Push S
Main: s read $\$\sigma^R$ on stack; corresponds to derivation of $s\sigma$
Read a Pop A
Simulates Rule $A \to a$ for single character a or $a = \lambda$
Pop \$
s read stack empty corresponds to derivation of s
Push B
Pop A Push C
Sim $A \to BC$: $\$\tau^R C$ on stack corresponds to derivation of $sA\tau$ followed by (Pop A, Push C)

Figure 3.24: PDA M_L simulating CFL G_L.

Note that the derivation is always replacing the leftmost symbol in σ. In order to simulate the derivation's use of rule $A \to a$, M_L has a self-loop labeled (Pop A, Read a). To simulate the use of rule $A \to \lambda$, M_L has a self-loop labeled (Pop A, Read λ). To simulate the use of rule $A \to B$, M_L has a self-loop labeled (Pop A, Push B). To simulate the use of rule $A \to BC$, M_L will execute the sequence Pop A, Push C, Push B (remember, $\$\sigma^R$ is on the stack, so the B needs to be at the top of the stack). To achieve this, M_L has an additional vertex called "Sim $A \to BC$" and edges (Main, "Sim $A \to BC$") and ("Sim $A \to BC$", Main), labeled (Pop A, Push C) and Push B, respectively. It will be helpful to refer to these actions, that take M_L from vertex Main back to itself, as *supermoves*. So each supermove of M_L corresponds to one derivation step in G_L.

A derivation of a terminal string s occurs if $\sigma = \lambda$. To allow this to be recognized, M_L uses a \$-shielded stack. Then if M_L is at vertex Main with data-configuration $(s, \$)$, it can pop its stack and move to its recognizing vertex. Thus if we can show the claim it is immediate that $G_L \Rightarrow^* w$ exactly if M_L can reach vertex *Main* with data configuration $(w, \$)$, which is the case exactly if M_L can reach its recognizing vertex on input w, i.e. exactly if $w \in L(M_L)$.

Proof. (Of Claim 3.7.3.) We show the claim in two steps. First, suppose that $G_L \Rightarrow^* s\sigma$. Then there is a leftmost derivation $S = s_1\sigma_1 \Rightarrow s_2\sigma_2 \Rightarrow \cdots \Rightarrow s_{k+1}\sigma_{k+1} = s\sigma$. (In a leftmost derivation, at step i, the leftmost variable in σ_i, for $1 \le i < k$, is always the one to be replaced using a rule of the grammar. It corresponds to a Depth First Traversal of the derivation tree.) The corresponding k supermove computation by M_L starts by moving to vertex Main with λ read and $\$S$ on its stack, i.e. it has data configuration $C_1 = (\lambda, \$S) = (s_1, \$\sigma_1^R)$. It then proceeds through the following k data configurations at vertex Main: $C_2 = (s_2, \$\sigma_2^R), \cdots C_{k+1} = (s_{k+1}, \$\sigma_{k+1}^R) = (s, \$\sigma^R)$, and it goes from C_i to C_{i+1}, for $1 \le i \le k$, by means of the supermove corresponding to the application of the rule taking $s_i\sigma_i$ to $s_{i+1}\sigma_{i+1}$.

Next suppose that M_L reaches vertex Main with configuration $(s, \Sigma\sigma^R)$. It does so by means of a computation using k supermoves, for some $k \geq 0$. The computation begins by moving to vertex Main, while reading λ and pushing $\$S$ on the stack, i.e. it is at configuration $C_1 = (\lambda, \$S) = (s_1, \$\sigma_1^R)$. It then moves through the following series of data configurations at vertex Main: $C_2 = (s_2, \$\sigma_2^R), \cdots, C_{k+1} = (s_{k+1}, \$\sigma_{k+1}^R)$, where, for $1 \leq i \leq k$, C_{i+1} is reached from C_i by means of a supermove. By construction, the ith supermove corresponds to the application of the rule that takes string $s_i\sigma_i$ to $s_{i+1}\sigma_{i+1}$. Thus, the following is a derivation in grammar G_L: $S = s_1\sigma_1 \Rightarrow s_2\sigma_2 \Rightarrow \cdots \Rightarrow s^{k+1}\sigma^{k+1}$. □

Now we conclude the proof of the lemma by showing that $L = L(M_L)$.

First, suppose that $w \in L$, i.e. that G_L can generate the string $w = w\sigma$ with $\sigma = \lambda$. Then, by the just proved claim, M_L can reach vertex Main with configuration $(w, \$)$. And from this configuration, M_L can follow the edge to its recognizing vertex, by popping the \$ now at the top of its stack. Thus $w \in L(M_L)$.

Next, suppose that $w \in L(M_L)$. This means that M_L reaches its recognizing vertex having read w. But then its last move must be a Pop \$ from vertex Main. This means that it had reached configuration $(w, \$)$ at vertex main. Then, by the claim, G_L can generate string w; i.e. $w \in L$. □

Lemma 3.7.4. *Let $L \subseteq \Sigma^*$ be context-free and let $R \subseteq \Sigma^*$ be regular. Then $L \cap R$ is context free.*

Proof. We illustrate the construction we give in this proof in Example 3.7.5 below.

Let $G_L = (V_L, \Sigma, R_L, S_L)$ be a simplified grammar generating L and let $M_R = (V_R, \mathit{start}, F_R, \delta_R)$ be a DFA recognizing R. We will build a grammar $G_{L\cap R} = (V_{L\cap R}, \Sigma, R_{L\cap R}, S_{L\cap R})$ to generate $L \cap R$. Let $V_R = \{q_1, q_2, \cdots, q_m\}$. For each variable $U \in V_L$, we create m^2 variables U_{ij} in $V_{L\cap R}$. The rules we create will ensure that:

$$U_{ij} \Rightarrow^* w \in \Sigma^*$$
$$\text{exactly if}$$
$$U \Rightarrow^* w \text{ and there is a path labeled } w \text{ in } M_R \text{ going from } v_i \text{ to } v_j.$$

Thus a variable in $G_{L\cap R}$ records the name of the corresponding variable in G_L and also records a "start" and a "finish" vertex in M_R. The constraint we are imposing is that U_{ij} can generate w exactly if both U can generate w and M_R when started at q_i will end up at q_j on input w. It follows that if q_f is a recognizing vertex of M_R and if $q_1 = \mathit{start}$, then

$$(S_L)_{1f} \Rightarrow^* w \text{ for some } q_f \in F_R$$
$$\text{exactly if}$$
$$w \in L \cap R.$$

If there is more than one $q_f \in F_R$ we cannot use $(S_L)_{1f}$ as the start variable in $G_{L\cap R}$, as this is not a single variable. Instead, we introduce an additional start variable $S_{R\cap L}$ and add the rules

- $S_{R\cap L} \to (S_L)_{1f}$ for all $q_f \in F_R$.

To simulate the use of a single rule in G_L, together with the corresponding move in M_R in the case that the rule generates a terminal, we create the following rules.

- $A_{ij} \to a$ if $A \to a$ is a rule in G_L and $\delta_R(q_i, a) = q_j$.

- $A_{ii} \to \lambda$ if $A \to \lambda$ is a rule in G_L, for all i, $1 \le i \le m$.
- $A_{ij} \to B_{ij}$ if $A \to B$ is a rule in G_L, for all i, j, $1 \le i, j \le m$.
- $A_{ik} \to B_{ij}C_{jk}$ if $A \to BC$ is a rule in G_L, for all i, j, k, $1 \le i, j, k \le m$.

To see why this works, we first consider any non-empty string $w \in L \cap R$ and look at a derivation tree T for w with respect to G_L. At the same time we look at a w-recognizing path P in M_R.

We label each leaf of T with the names of two vertices in M_R: the vertices that M_R is at right before and right after reading the input character corresponding to the character labeling the leaf. If we read across the leaves from left to right, recording vertex and character or λ labels, we obtain a sequence $p_1w_1p_2, p_2w_2p_3, \cdots, p_nw_np_{n+1}$, where each w_i is either a single character or the empty string, $p_1 = start$, $p_{n+1} \in F_R$, if w_i is a character then $\delta_R(p_i, w_i) = p_{i+1}$, and if $w_i = \lambda$ then $p_{i+1} = p_i$, for $1 \le i \le n$.

Next, we provide vertex labels to the internal nodes in T. A node receives as its first label the first label of its leftmost leaf and as its second label the second label of its rightmost leaf. Suppose that A is the variable label at an internal node with children having variable labels B and C (see Figure 3.25).

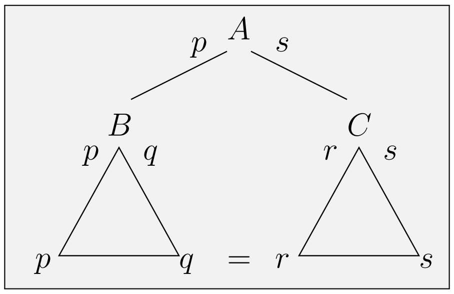


Figure 3.25: Vertex Labels in Derivation Tree T: first label on left, second label on right.

Suppose further that B receives vertex labels p and q (in that order), and C receives vertex labels r and s. Then $q = r$ and A receives vertex labels p and s. To obtain the derivation in $G_{L\cap R}$, we simply replace $A \Rightarrow BC$ by $A_{ps} \Rightarrow B_{pq}C_{qs}$. Similarly, we replace $A \Rightarrow B$ by $A_{pq} \Rightarrow B_{pq}$. In addition, at the leaf level, we replace $A \Rightarrow a$ by $A_{pq} \Rightarrow a$ where p and q are the vertex labels on the leaf (and on its parent), and $A \Rightarrow \lambda$ by $A_{pp} \Rightarrow \lambda$, where p is the start and finish vertex label for the leaf. Clearly, this is a derivation in $G_{L\cap R}$ and further it derives w. Thus if $w \in L \cap R$, then $S_{L\cap R} \Rightarrow^* w$, i.e. $w \in (G_{L\cap R})$.

On the other hand, suppose that $S_{L\cap R} \Rightarrow^* w$. Then consider the derivation tree for w in $G_{L\cap R}$. On removing the root and replacing each variable U_{ij} by U we obtain a derivation tree for w in G_L (for recall that the root has one child labeled by $(S_L)_{1f}$ for some $q_f \in F_R$; the variable replacement process turns this child's label to S_L). Thus $S_L \Rightarrow^* w$ also. On looking at the leaf level, and labeling each leaf with the vertex indices on the variable at its parent, we obtain a sequence $p_1w_1p_2, p_2w_2p_3, \cdots, p_nw_np_{n+1}$, where $w = w_1w_2 \cdots w_n$, each w_i is either a single character or the empty string, $p_1 =$ start and $p_{n+1} \in F_R$. For each w_i that is a single character, $\delta_R(p_i, w_i) = p_{i+1}$, and for each $w_i = \lambda$, $p_{i+1} = p_i$, for $1 \le i \le n$, by the first and second rules in the definition of $G_{L\cap R}$. Now we see that $p_1p_2p_i \cdots p_{n+1}$, minus the duplicated vertices for "reading" $w_i = \lambda$, is a w-recognizing path in M_R, and so $w \in R$. This shows that if $S_{L\cap R} \Rightarrow^* w$ then $w \in L \cap R$. □

Example 3.7.5. CFG for $L \cap R$
where $R = a^*bb^*$ and L has grammar G given below.

G is given by the rules:

$$\begin{aligned} S &\to WX \\ W &\to \lambda \,|\, aWb \\ X &\to \lambda \,|\, Xa \end{aligned}$$

First, we convert G to a simplified grammar, given the rules:

$$\begin{aligned} S &\to WX \\ W &\to \lambda \,|\, U^aY \\ Y &\to WU^b \\ X &\to \lambda \,|\, XU^a \\ U^a &\to a \\ U^b &\to b \end{aligned}$$

We use a 2-vertex automata M_R to recognize $R = a^*bb^*$, as shown in Figure 3.26.

This means that we need up to three versions of each variable, e.g. for W we may need W_{11}, W_{12}, W_{22}. W_{21} is not needed as one cannot go from vertex p_2 to vertex p_1 in M_R.

The new grammar has start vertex S_{12} and rules:

$$\begin{aligned} S_{12} &\to W_{12}X_{22} \\ W_{11} &\to \lambda \\ W_{12} &\to U^a_{11}Y_{12} \\ X_{22} &\to \lambda \\ Y_{12} &\to W_{11}U^b_{12} \,|\, W_{12}U^b_{22} \\ U^a_{11} &\to a \\ U^b_{12} &\to b \\ U^b_{22} &\to b \end{aligned}$$

None of the other variables can generate a string of terminals. For an a can be read by M_R only along the self-loop from p_1 to itself, so the only useful copy of variable U^a is U^a_{11}. Similarly only U^b_{12} and U^b_{22} are useful. Since X appears only on the RHS of the rule for S_{12} it must be in the form X_{12} or X_{22}. The variable X_{12} can only generate $X_{12}U^a_{22}$ or $X_{11}U^a_{12}$, but neither of these forms of U_a can generate terminals. Thus X appears only in the form X_{22}, meaning that the only RHS for the rule with S on the LHS is the rule shown. Similarly W_{12} can only be replaced by $U^a_{11}Y_{12}$. The two possible replacements for Y_{12} are as shown. Finally W_{11} can be replaced by λ or by $U^a_{11}Y_{11}$; but the only possible replacement for Y_{11} is $W_{11}U^b_{11}$, and we have already ruled out U^b_{11} as a useful variable, which rules out Y_{11} as a useful variable, leaving just the rules shown.

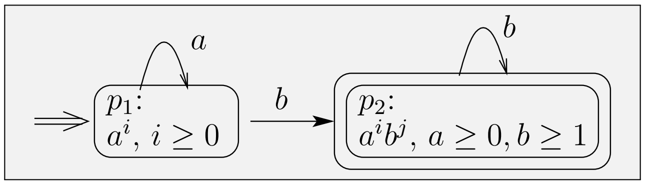


Figure 3.26: DFA recognizing a^*bb^*.

3.7.1 Constructing a CFG Generating a PDA-Recognized Language

Our final construction will show that if L is recognized by a PDA, then there is a CFG generating L. The first step in the construction is to represent the computation of the PDA in terms of matching pushes and pops to the stack.

A little more terminology will be helpful.

Definition 3.7.6. *A computation of M that goes from vertex p to vertex q and begins and ends with σ on the stack is called σ-preserving if throughout the computation σ remains on the stack (possibly with other characters pushed on top some of the time); i.e. none of σ is popped during this part of the computation.*

Matching Pushes and Pops: The Trapezoidal Decomposition

Let $M_L = (Q, T, \Gamma, F, s, \delta)$ be a \$-shielded PDA recognizing L that has a single recognizing vertex f which is reachable only with an empty stack (see Lemma 3.4.2). Let us further suppose that on each move M_L does either a Pop or a Push but not both (if there is a move in which neither a Pop nor a Push occurs, it can be replaced by two moves, the first being an unnecessary Push and the second being a Pop; likewise, a move which has both a Pop and a Push can be replaced by two moves: a Pop followed by a Push). Finally, we suppose that the first step of the computation goes from vertex s to vertex s', and this move pushes the \$-shield onto the stack, but does no read (i.e. it reads λ); in addition, s cannot be visited again, which is ensured by having no in-edges to s. Also note that s has just the one out-edge. Likewise, we suppose that the last step of a recognizing computation will go from vertex f' to vertex f, while popping the \$-shield and again reading λ; further, this is the only visit to f, which is ensured by having no other in-edges to f and no out-edges at all. Finally, note that s and f are distinct vertices (since one has a single in-edge and the other a single out-edge).

We begin with an example that illustrates the general construction. Consider the following PDA M_L which recognizes the set L of strings with equal numbers of a's and b's. The diagram for M_L is shown in Figure 3.27. Now consider a possible computation of M_L on input $a(ab)(bbaa)(ba)b$ (the brackets indicate one way of matching the a's and b's). It is helpful to view M_L's computation in terms of a *Stack Contents Diagram*, as shown in Figure 3.28. It shows the height of the stack evolving as the computation proceeds. The successive nodes in the diagram represent the successive vertices reached by M_L over the course of its computation. For clarity in our example, so as to distinguish the multiple instances of vertex p, we name them p^1, p^2, etc. In general, though, we will label a node in the diagram with the name of the corresponding vertex in M_L; thus multiple nodes could have the same label. When no confusion will result, we will often name a node by using its

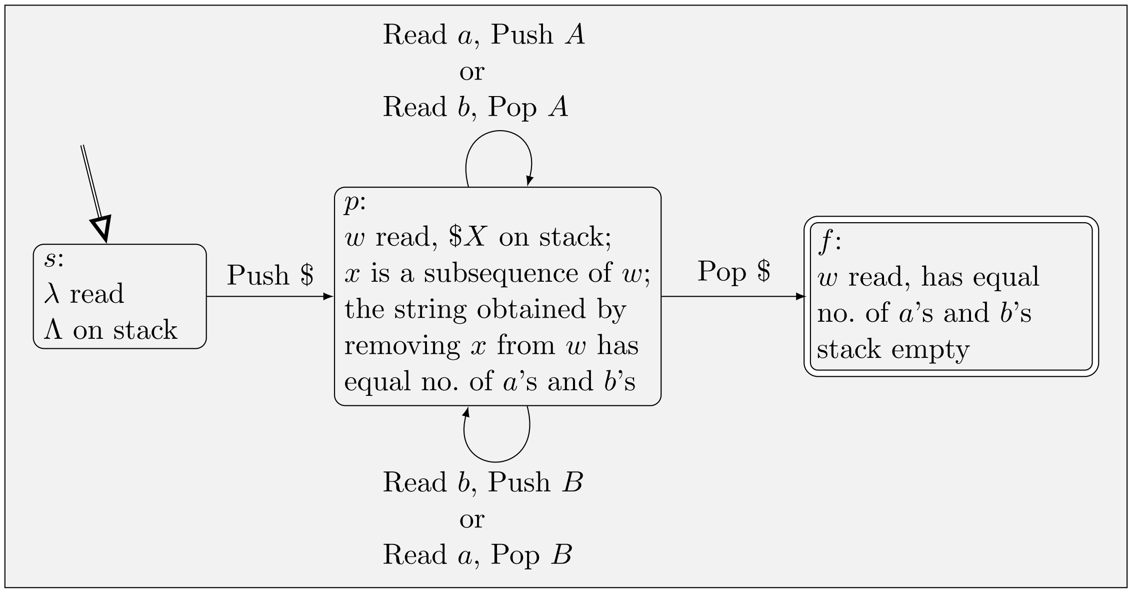


Figure 3.27: PDA recognizing strings with equal numbers of a's and b's.

label (even when this is not a unique identifier). Also, each read is shown as a label on an edge; so for example in going from p^1 to p^2 an a is read (i.e. in going from the first visit to vertex p to the second visit to p). In addition, the stack contents at any point in the computation can be obtained by reading vertically up the series of trapezoids; e.g. at p^6, the stack contents are $\$ABB$.

Notice that the string recognized by this computation can be obtained by concatenating the series of edge labels in left to right order. Our goal is to create a derivation tree that has one leaf for each edge, in the same left to right order, and with each leaf having the same label as the corresponding edge. One way to do this is to introduce one internal node for each trapezoid base and one leaf for each edge in the Stack Contents Diagram. Clearly, this derivation tree derives the string recognized by the computation of M_L represented in the Stack Contents Diagram. Of course, we have yet to specify the variables and rules of the grammar that would actually let us generate this derivation tree. But the goal is clear: we want to generate the derivation tree shown in Figure 3.29.

Now we are ready to describe how to create the grammar G_L generating L.

Forming the trapezoids We associate each node in the Stack Contents Diagram with a matching successor or predecessor node (or possibly both), as follows.

Definition 3.7.7. *Nodes p and q are* matched *if q is the first vertex following p for which the computation path in M_L from p to q is σ-preserving.*

If we draw the edges connecting matched vertices in the Stack Contents Diagram, this naturally partitions the diagram into trapezoids. We call this the *Trapezoidal Decomposition.* In our example, the first and last vertices are s and f, and they are matched. All the other vertices are instances of vertex p. Note that p^2 is matched with p^4, and p^4 is also matched with p^8. (The vertices at a

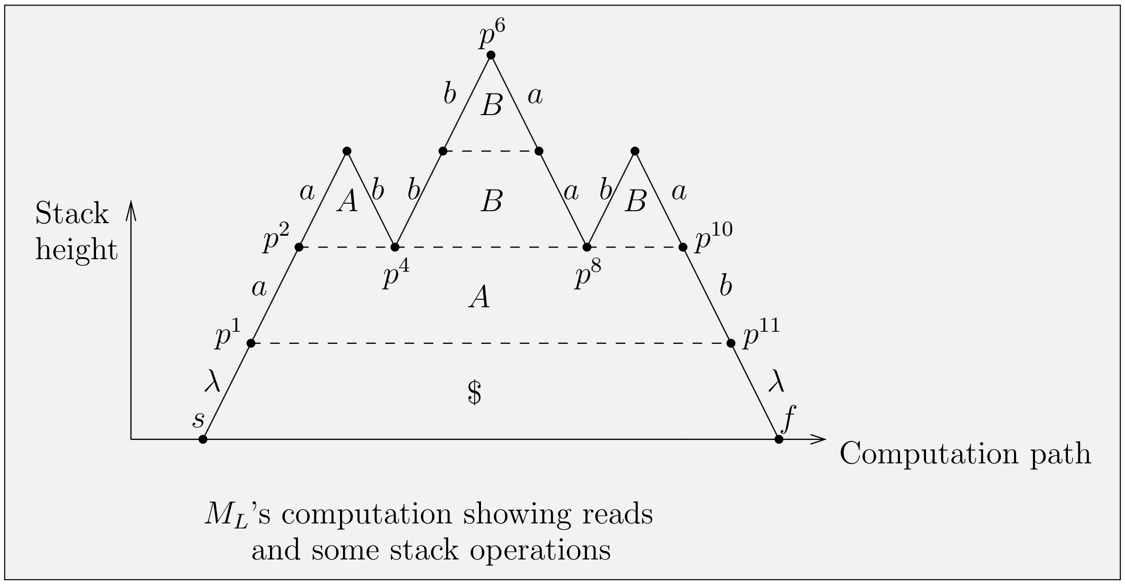


Figure 3.28: Stack Contents Diagram, showing characters read and stack contents.

point in the computation where M_L switches from doing Pops to Pushes will be the ones that are double matched.)

We give each trapezoid $\mathcal{T}$ a name that exactly specifies the two computation steps it represents. Each such name has the form $\mathcal{T}^{Aab}_{pqrs}$, where the first computation step comprises "Read(a), Push(A)" operations and goes from vertex p to vertex q, and the second computation step comprises "Read(b), Pop(A)" operations and goes from vertex r to vertex s. Notice that either or both of a and b could equal λ; indeed, in our example the bottommost trapezoid is $\mathcal{T}^{\$\lambda\lambda}_{sppf}$. We will only allow values for the parameters A, a, b, p, q, r, s that correspond to possible computation steps of M_L and to emphasize this point we refer to such trapezoids $\mathcal{T}^{Aab}_{pqrs}$ as *realizable trapezoids*. In our example, $\mathcal{T}^{Aab}_{pppp}$ is realizable but $\mathcal{T}^{Aaa}_{pppp}$ is not, for opposite characters must be read by the two computation steps represented by a trapezoid.

We also note that the intermediate computation that takes M_L from q to r must be stack-preserving. The consequence is that the Pop performed in $\mathcal{T}$'s second step pops the very A pushed in its first step. And so the full computation going from p to s is also stack-preserving. We call this computation a *phase*. Note that the phase starting at node p is simply the shortest stack-preserving computation beginning at node p.

Let $\mathcal{P}$ be the phase associated with trapezoid $\mathcal{T}$. We call $\mathcal{T}$ the base trapezoid for $\mathcal{P}$. If $\mathcal{T} = \mathcal{T}^{Aab}_{pqrs}$, then $\mathcal{P}$ consists of $\mathcal{T}$ plus a stack-preserving computation that goes from q to r. As this computation is stack-preserving, it consists of zero or more (sub)-phases, $\mathcal{P}_1, \mathcal{P}_2, \cdots, \mathcal{P}_k$, say, where $k \geq 0$ (the case $k = 0$ can occur only when $q = r$, though even when $q = r$ it could be that $k \geq 1$). In our example, the phase $\mathcal{P}$ that begins at p^1 ends at p^{11}. $p^1p^2p^{10}p^{11}$ is its base trapezoid. $\mathcal{P}$ contains three subphases that begin and end at, respectively, p^2 and p^4, p^4 and p^8, and p^8 and p^{10}.

If a phase consists of just two steps, we call its base trapezoid a *triangle*. In our example, the subphase which goes from p^2 to p^4 is a triangle.

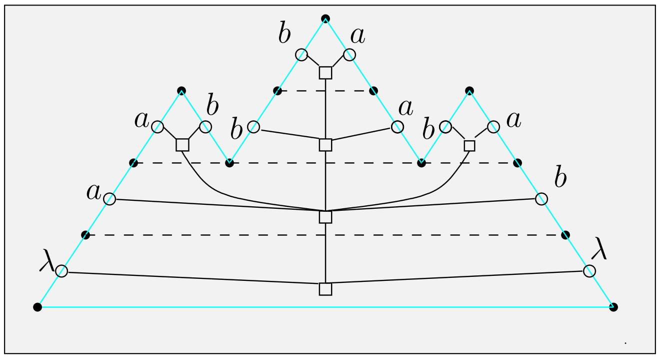


Figure 3.29: Derivation Tree for string $a(ab)(bbaa)(ba)b$ recognized by M_L.

A key property of the trapezoids (and phases) is their nesting. Suppose that realizable trapezoid $\mathcal{T}$ has k realizable trapezoids $\mathcal{T}_1, \mathcal{T}_2, \cdots, \mathcal{T}_k$ immediately above it in the trapezoidal decomposition. We will say that $\mathcal{T}_{i+1}$ follows $\mathcal{T}_i$, for $1 \le i < k$, meaning that the bottom right-hand corner of $\mathcal{T}_i$ has the same label as the bottom left-hand corner of $\mathcal{T}_{i+1}$. We will also say that $\mathcal{T}_1, \mathcal{T}_2, \cdots, \mathcal{T}_k$ are nested in $\mathcal{T}$, meaning that $\mathcal{T}$'s top left-hand corner has the same label as the bottom left-hand corner of $\mathcal{T}_1$, and $\mathcal{T}$'s top right-hand corner has the same label as the bottom right-hand corner of $\mathcal{T}_k$, and that $\mathcal{T}_{i+1}$ follows $\mathcal{T}_i$, for $1 \le i < k$. In terms of the trapezoid names, we have that $\mathcal{T}_i = \mathcal{T}^{\circ\circ\circ}_{q_i\circ\circ q_{i+1}}$, for $1 \le i \le k$, and $\mathcal{T} = \mathcal{T}^{\circ\circ\circ}_{\circ q_1 q_{k+1}\circ}$, for some $q_1, q_2, \cdots, q_{k+1}$, where each $\circ$ indicates an unspecified parameter value.

The Context Free Grammar For each realizable trapezoid $\mathcal{T} = \mathcal{T}^{Aab}_{pqrs}$ in the trapezoidal decomposition we will create a node in the derivation tree labeled by a variable U_{ps}, which we can think of as corresponding to the base of this trapezoid.

Example, continued. In our example, this yields the following two variables:

$$U_{sf}, U_{pp}.$$

The start variable in our example is U_{sf}; it corresponds to the initial and final vertices of a recognizing computation, namely vertices s and f, respectively.

Suppose Trapezoid $\mathcal{T}^{Aab}_{pqrs}$ has a single trapezoid $\mathcal{T}'$ nested inside it. Necessarily $\mathcal{T}'$ has base qr. So the natural rule for the corresponding variables would be

$$U_{ps} \to aU_{qr}b$$

However, this will not handle the case in which more than one trapezoid is nested inside $\mathcal{T}^{Aab}_{pqrs}$. As the number of nested trapezoids can be unbounded we need to introduce intermediate variables to allow us to create a sequence of variables corresponding to the bases of these intermediate variables.

To this end, for each pair of variables $p, q \in V$, we create the intermediate variable X_{pq} (possibly

$p = q$), and we add the following rules:

$$\begin{aligned}&X_{pr} \to X_{pq}X_{qr}, \quad \text{for all } p,q,r \in Q\\ &X_{pq} \to U_{pq}, \quad \text{for all } p,q \in Q\\ &U_{ps} \to aX_{qr}b, \quad \text{for all } p,q,r,s \in Q \text{ such that there is a trapezoid } \mathcal{T}^{Bab}_{pqrs} \text{ for some } B \in \Gamma\end{aligned}$$

Finally, if $\mathcal{T}^{Bab}_{pqqr}$ is a legal trapezoid, we add the rule

$$U_{pr} \to ab$$

The start variable for this grammar will be U_{sf}, where s and f are the names for the start and recognizing vertices in M.

Example, continued. For our example, this creates the following rules.

$$\begin{array}{rcl} U_{sf} & \to & U_{pp} \mid \lambda \\ U_{pp} & \to & a\,U_{pp}\,b \mid a\,X_{pp}\,b \mid ab \\ U_{pp} & \to & b\,U_{pp}\,a \mid b\,X_{pp}\,a \mid ba \\ X_{pp} & \to & X_{pp}X_{pp} \mid U_{pp} \end{array}$$

Now given a trapezoidal decomposition, we create a derivation tree as follows. We introduce a variable U_{pq} for each trapezoid base $\mathcal{T}^{Bab}_{p\circ\circ q}$, and a leaf node for each edge labeled by the character read on that edge. Next, we use the intermediate variables to connect the base of one trapezoid $\mathcal{T} = \mathcal{T}^{Bab}_{pq_0q_kr}$ to the sequence of bases of the trapezoids nested in $\mathcal{T}$. In particular, suppose $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_k$ are nested inside $\mathcal{T}$, with variables p, r at the base of $\mathcal{T}$, and variables q_{i-1}, q_i at the base of $\mathcal{T}_i$, for $1 \le i \le k$. Then, we can use the following series of replacements:

$$\begin{aligned} U_{pr} &\to aX_{q_0q_k}b \\ X_{q_0q_k} &\to X_{q_0q_1}X_{q_1q_k} \quad \text{and} \quad X_{q_0q_1} \to U_{q_0q_1} \\ X_{q_1q_k} &\to X_{q_1q_2}X_{q_2q_k} \quad \text{and} \quad X_{q_1q_2} \to U_{q_1q_2} \\ &\ldots \\ X_{q_{k-2}q_k} &\to X_{q_{k-2}q_{k-1}}X_{q_{k-1}q_k}, X_{q_{k-2}q_{k-1}} \to U_{q_{k-2}q_{k-1}}, \text{ and } X_{q_{k-1}q_k} \to U_{q_{k-1}q_k} \end{aligned}$$

to achieve $U_{pr} \Rightarrow^* aU_{q_0q_1}U_{q_1q_2}\ldots U_{q_{k-1}q_k}b$.

This then yields a derivation of the string w read by this recognizing computation of PDA M.

We conclude the argument by showing that given a derivation of w in our grammar, there is a corresponding trapezoidal decomposition and hence a recognizing computation of M on input w.

Given a derivation tree D, we shortcut all the nodes labeled by X variables to obtain a new tree D', where there is no particular bound on the node degree. The internal nodes are labeled by U variables and these correspond to trapezoid bases; the leaves are labeled by terminals or λ's and these will correspond to the reads performed in M's computation.

In particular for a node labeled U_{pr}, with children labeled by $a, U_{q_0q_1}, U_{q_1q_2}, \ldots, U_{q_{k-1}q_k}, b$, we create a trapezoid $\mathcal{T}^{Bab}_{q_0q_k}$, where B is one of the stack characters for which both the moves (Read a, Push B) from p to q_0 and (Read b, Pop B) from q_k to r are legal. And for a node labeled by U_{pr} with two children labeled by a and b, respectively, we create a trapezoid $\mathcal{T}^{Bab}_{pqqr}$, for which the moves (Read a, Push B) from p to q and (Read b, Pop B) from q to r are legal. This trapezoidal diagram specifies a recognizing computation of M which reads w, and consequently M recognizes w.

We have shown the following result.

Theorem 3.7.8. *Let PDA M_L recognize language L. Then there is a CFL generating L.*

Exercises

The following terminology is used in several of these questions. A string u is called a prefix of string w if there is a string v such that $w = uv$; similarly, v is called a suffix of w.

1. For the PDA in Figure 3.30 answer the following questions.https://www.overleaf.com/project/64e72514e757ece4

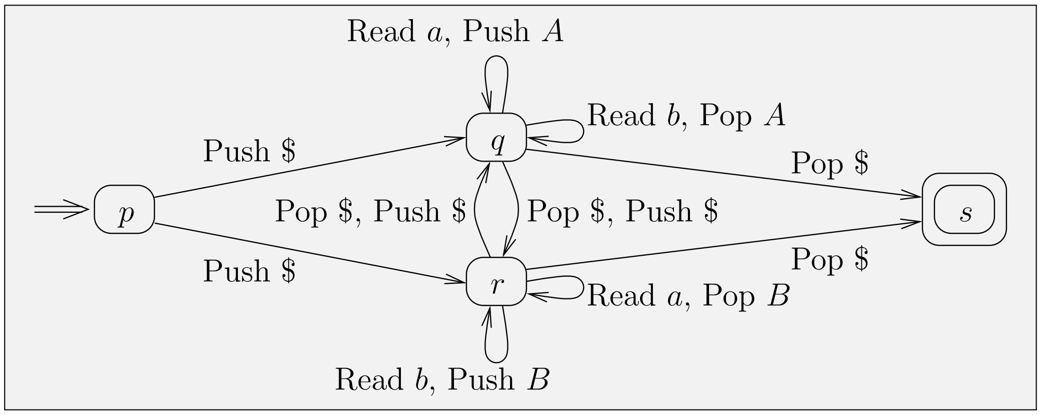


Figure 3.30: Figure for Problem 1.

   i. What is its start vertex?
   ii. What are its recognizing vertices?
   iii. Give the sequence of vertices and associated data configurations the PDA goes through in a recognizing computation on input $abba$. Are there any other computation paths that can be followed on this input, and if yes, give the sequence of vertices such a computation goes through.
   iv. What is the language accepted by this PDA?

2. For the PDA in Figure 3.27 answer the following questions.

   i. What is its start vertex?
   ii. What are its recognizing vertices?
   iii. What are the possible stack contents after reading input ab?

3. Give PDAs to recognize the following languages. You are to give both an explanation in English of what each PDA does plus a graph of the PDA; seek to label the graph vertices accurately (i.e. for each vertex, specify the possible data configurations reached at that vertex).

   i. $A = \{w \mid w \text{ has odd length, } w \in \{a,b\}^* \text{ and the middle symbol of } w \text{ is an } a\}$
   ii. $B = \{w \mid w \text{ has even length, } w \in \{a,b\}^* \text{ and the two middle characters of } w \text{ are equal}\}$.
   iii. $C = \{w \mid w \in \{a,b\}^* \text{ and } w \neq a^i b^i \text{ for any integer } i\}$.
   iv. $D = \{wcw^R x \mid w, x \in \{a,b\}^*\}$.
   v. $E = \{w \mid w \in \{a,b\}^* \text{ and } w \text{ contains equal numbers of } a\text{'s and } b\text{'s}\}$.
   vi. $F = \{w \mid w \in \{a,b\}^* \text{ and every prefix of } w \text{ contains at least as many } a\text{'s as } b\text{'s}\}$.
   vii. $H = E \cap F$.
   viii. $I = \{w \mid w \in \{(,),[,]\}^*$, w is a string of nested parentheses with each "(" matching a ")" to its right, and each "[" matching a "]" also to its right$\}$.
   ix. $J = \{w\#x \mid w, x \in \{a,b\}^* \text{ and } w^R \text{ is a prefix of } x\}$.
   Hint: x can be written as $x = w^R y$ for some $y \in \{a,b\}^*$. Also note that # is just another character.
   x. $K = \{x \mid x \in \{a,b\}^* \text{ and } x \neq ww^R \text{ for any } w \in \{a,b\}^*\}$.

4. Suppose that A is recognized by PDA M. Give a PDA to recognize A^*.

5. Let C be a language over the alphabet $\{a,b\}$.

   i. Let $\text{Suffix}(C) = \{w \mid \text{there is a } u \in \{a,b\}^* \text{ with } uw \in C\}$. Show that if C is recognized by a PDA then so is $\text{Suffix}(C)$.
   ii. Similarly, let $\text{Prefix}(C) = \{w \mid \text{there is an } x \in \{a,b\}^* \text{ with } wx \in C\}$. Show that if C is recognized by a PDA then so is $\text{Prefix}(C)$.

6. This problem aims to show that there is a PDA recognizing the following language: $\{s\#t \mid s, t \in \{a,b\}^* \text{ and } s \neq t\}$.

   i. Let $A = \{uav\#xby \mid u, v, x, y \in \{a,b\}^* \text{ and } (|u| - |v|) = (|x| - |y|)\}$. Give a PDA to recognize A.
   ii. Let $B = \{w\#z \mid w, z \in \{a,b\}^* \text{ and } |w| \neq |z|\}$. Give a PDA to recognize B.
   iii. Show that $A \cup B = \{s\#t \mid s, t \in \{a,b\}^* \text{ and } s \neq t\}$.

7. Consider the PDA in Figure 3.30. Add descriptors to the vertices. What language does this PDA recognize?

8. Give DPDAs to recognize the following languages. You are to give both an explanation in English of what each DPDA does plus a graph of the DPDA; seek to label the graph vertices accurately (i.e. for each vertex, specify the possible data configurations reached at that vertex).

   i. $A = \{wcx \mid w, x \in \{a,b\}^* \text{ and } |w| = |x|\}$
   ii. $B = \{w \mid w \in \{a,b\}^* \text{ and } w \neq a^i b^i \text{ for any integer } i\}$.

iii. $C = \{wcw^R \mid w \in \{a,b\}^*\}$.

iv. $D = \{wcx \mid w, x \in \{a,b\}^*$ and $w \neq x^R\}$.

v. $E = \{w \mid w \in \{a,b\}^*$ and w contains equal numbers of a's and b's$\}$.

vi. $F = \{w \mid w \in \{a,b\}^*$ and every prefix of w contains at least as many a's as b's$\}$.

vii. $G = E \cap F$.

viii. $H = \{w \mid w \in \{(,),[,]\}^*$, w is a string of nested parentheses with each "(" matching a ")" to its right, and each "[" matching a "]" also to its right$\}$.

ix. $I = \{w\#x \mid w, x \in \{a,b\}^*$ and w^R is a prefix of $x\}$.
Hint: x can be written as $x = w^R y$ for some $y \in \{a,b\}^*$. Also note that # is just another character.

9. Assuming that the complement of a language recognized by a DPDA is also recognized by a DPDA, show that DPDAs are not closed under union; i.e. give two languages L_1 and L_2 which are recognized by DPDAs, and show that $L_1 \cup L_2$ is not recognized by any DPDA.

10. This exercise will show that DPDAs are closed under complementation. Let M be a DPDA recognizing language L. The challenge to overcome is that there may be inputs on which M will traverse an infinite loop. This can happen only if there is a cycle of edges all of which have a label including a Read λ. The task is to modify M so as to remove all loops which could induce an infinite length computation, while leaving the language it recognizes unchanged. We now outline the needed modifications. This problem asks you to fill in missing details.

    The goal will be to build a DPDA with one recognizing vertex and one reject vertex, both reached by reading ¢. All input strings will have one of these two vertices as their destination.

    i. In this step, the DPDA is modified so as to remove all edges labeled by (Read λ, Pop λ, Push λ), λ-edges for short. Note that any vertex having a λ-out-edge has no other out-edge. If some subset of these edges form a cycle, they are replaced by a sink vertex (a vertex at which the rest of the input is read) followed by an edge to the reject vertex. Then all remaining λ-edges are removed, with other edges being added as necessary (explain what needs to be done). Argue that the language recognized by the machine following there changes is still $L(M)$.

    ii. We make multiple copies of each vertex so as to partition its in-edges. One copy receives all the in-edges whose label includes a Read a for some $a \in \Sigma$. Every other copy has inedges with a label including a Read λ. Each copy v_B is associated with edges whose label includes a Push B, for $B \in \Gamma \cup \{\lambda\}$. The out-edges of all these copies are the same as for vertex v. For the start vertex, we make an extra copy with no in-edges; this will be the new start vertex.

        Observe that the language recognized by the machine following there changes is still $L(M)$.

    iii. Suppose two successive edges (u,v) and (v,w) have the following edge labels. Make the indicated changes and observe that the language recognized by the machine following these changes is still $L(M)$.

        - (Read λ, Pop A, Push B), (Read λ, Pop C, Push D) where $A, B, C \neq \lambda$, and $B \neq C$. Then remove edge (v,w). If v has no out-edges now, remove v also.

- (Read λ, Pop A, Push B), (Read λ, Pop B, Push D), where $A, B \neq \lambda$, but perhaps $A = B$.
  Note that because of the previous step this is the only out-edge for vertex v. Then remove vertex v and its incident edges, and add edge (u, w) labeled by (Read λ, Pop A, Push D).
- (Read λ, Push B), (Read λ, Pop C, Push D) where $B, C \neq \lambda$, and $B \neq C$.
  Then remove the edge (v, w).
- (Read λ, Push B), (Read λ, Pop B, Push D), where $B \neq \lambda$.
  Again, because of the previous step, this is the only out-edge for vertex v. Then for each of v's in-edges (u, v) (which all have the same label) make a new edge (u, w) with label (Read λ, Pop λ, Push D), and remove vertex v and its incident edges. In addition, if $D = \lambda$, remove the new λ-edge as in step (i).

iv. Argue that this process terminates.

v. Now conclude that each cycle of edges with labels that all include Read λ must all have labels with a Push but no Pop, or a Pop but no Push (where we mean these operations to be on actual characters, and not λ).

Replace each cycle of Pushes with a sink vertex as in Step (i). Argue that the language recognized by the machine following there changes is still $L(M)$.

Conclude that there are no infinite computational loops in the resulting machine.

Explain what further modifications are needed to make it recognize $\overline{L}$.

11. Consider the following context free grammar: $S \to (S) \,|\, [S] \,|\, SS \,|\, (\,) \,|\, [\,]$

i. What are its terminals?

ii. What are its variables?

iii. What are its rules?

iv. Show the derivation tree for string ([ ] ( )).

v. Describe in English the set of strings generated by this grammar.

12. Consider the following context free grammar:

$$\begin{aligned} S &\to XY \\ X &\to aXb \,|\, \lambda \\ Y &\to cYd \,|\, \lambda \end{aligned}$$

i. What are its terminals?

ii. What are its variables?

iii. What are its rules?

iv. Show the derivation tree for string $abcd$.

v. Describe in English the set of strings generated by this grammar.

13. Consider the following context free grammar:

$$\begin{aligned} S &\to XY \\ X &\to aXb \,|\, aYb \,|\, \lambda \\ Y &\to cYd \,|\, \lambda \end{aligned}$$

    i. What are its terminals?
    ii. What are its variables?
    iii. What are its rules?
    iv. Show the derivation tree for string $acdb$.
    v. Describe in English the set of strings generated by this grammar.

14. Consider the following context free grammar:

$$\begin{aligned} S &\to XY \,|\, YX \\ X &\to aXb \,|\, \lambda \\ Y &\to cXd \,|\, cYd \,|\, \lambda \end{aligned}$$

    i. What are its terminals?
    ii. What are its variables?
    iii. What are its rules?
    iv. Show the derivation tree for string $cabd$.
    v. Describe in English the set of strings generated by this grammar.

15. Give CFG's to generate the following languages. Remember to specify for each variable the set of strings it generates. Of course, S, the start variable, should generate the language in question.

    i. $A = \{w \mid w$ has odd length, $w \in \{a, b\}^*$ and the middle character of w is an $a\}$.
    ii. $B = \{w \mid w$ has even length, $w \in \{a, b\}^*$ and the two middle characters of w are equal$\}$.
    iii. $C = \{w \mid w \in \{a, b\}^*$ and $w \neq a^i b^i$ for any integer $i\}$.
    iv. $D = \{w \,|\, w \in \{a, b\}^*$ and $w = w^R\}$. D is the language of palindromes, strings that read the same forward and backward.
    Hint: Be sure to handle strings of all possible lengths.
    v. $E = \{wcw^R x \,|\, w, x \in \{a, b\}^*\}$.
    vi. $F = \{w \,|\, w \in \{a, b\}^*$ and w contains an equal number of a's and b's$\}$.
    Hint: suppose that the first character in w is an a. Let x be the shortest prefix of w having an equal number of a's and b's. If $|x| < |w|$, then w can be written as $w = xy$; what can you say about y? Otherwise, $x = w$ and w can be written as $w = azb$; what can you say about z?
    vii. $H = \{w \,|\, w \in \{a, b\}^*$ and every prefix of w contains at least as many a's as b's$\}$.

viii. $I = \{w \mid w \in \{(,),[,]\}^*$, w is a string of nested parentheses with each "(" matching a ")" to its right, and each "[" matching a "]" also to its right$\}$.

ix. $J = \{w\#x \mid w, x \in \{a,b\}^*$ and w^R is a prefix of x $\}$.
Hint: x can be written as $x = w^R y$ for some $y \in \{a,b\}^*$.

16. i. Let $E = \{a^i b^j \mid i < j\}$. Give a CFG to generate E.

ii. Let $F = \{a^i b^j \mid 2i > j\}$. Give a CFG to generate F.

iii. Let $J = \{a^i b^j \mid i < j < 2i\}$. Give a CFG to generate J.
Hint: Let $i = h + l$ and $j = h + 2l$. what can you say about h and l?

17. Give CFGs to generate the following languages. Remember to specify for each variable the set of strings it generates.

i. $L_1 = \{a^i \# b^{i+j} \$ a^j \mid i, j \geq 0\}$. (Recall that # is just another character.)

ii. $L_2 = \{w\#x\$y \mid w, x, y \in \{a,b\}^*$ and $|x| = |w| + |y|\}$.

iii. $L_3 = \{uv \mid |u| = |v|$ but $u \neq v\}$.
Hint: think of the # and the \$ from part (2) as a pair of aligned yet unequal characters in u and v; what is the relation among the lengths of the remaining pieces of u and v?
Further hint: Use the ideas from the first two parts.

18. Let A be a CFL generated by a CFG G_A. Give a CFG grammar G_{A^*}, based on G_A, to generate A^*. Show that $L(G_{A^*}) = A^*$.

19. Convert the following grammars to simplified form—see Section 3.5.

i. G_1 has start variable S, terminal set $\{a,b\}$ and rule $S \to ab$.

ii. G_2 has start variable S, terminal set $\{a,b,c,d\}$ and rules $S \to ABCD$, $A \to a$, $B \to b$, $C \to c$, $D \to d$.

20. This exercise derives an algorithm to convert a grammar to Chomsky Normal Form (CNF)—see Section 3.5. Let G be a grammar in simplified form.

i. We start by removing S from the RHS of any rule in which it appears. Simply add a new variable S' and use it to replace S in every rule in which S appeared, and then add the rule $S \to S'$.

Argue that the resulting grammar G' generates the same language as G: $L(G') = L(G)$.

ii. Next, we will remove λ from the RHS of all rules except possibly a rule with S on the LHS. Begin by using the following method to determine the set of variables that can generate λ:

For each rule of the form $A \to \lambda$ add A to the set.

For each rule of the form $A \to B$ with B in the set, add A to the set.

For each rule of the form $A \to BC$ with both B and C in the set, add A to the set.

Iterate this process until the set of these variables grows no further.

Then we modify the grammar as follows.

Remove all rules of the form $A \to \lambda$.

If S is in the set include the rule $S \to \lambda$ in the grammar.

Finally, for each rule $A \to BC$, if B is in the set, include $A \to C$ in the grammar, and if C is in the set, include $A \to B$ in the grammar.

Argue that the resulting grammar G'' generates the same language as G': $L(G'') = L(G')$.

iii. Now we give an algorithm to remove all rules of the form $A \to B$. We say the variables $A_1, A_2, \ldots, A_k$ are equivalent if there is sequence of rules $A_1 \to A_2, A_2 \to A_3, \ldots, A_{k-1} \to A_k, A_k \to A_1$. We will identify all equivalent variables, and replace each collection of equivalent variables by a new single variable. So in the above instance we would replace each occurrence of $A_1, A_2, \ldots, A_k$ in any rule by the new variable A say. We then remove all the copies of the resulting rule $A \to A$.

Give an algorithm to identify all equivalent variables.

Argue that the resulting grammar generates the same language as the original grammar.

Next, for each sequence of rules $A_1 \to A_2, A_2 \to A_3, \ldots, A_{k-1} \to A_k$, we add the rules $A_1 \to A_2, A_1 \to A_3, A_1 \to A_k$ (and $A_2 \to A_4, \ldots A_2 \to A_k$, etc.).

Give an algorithm to perform these changes, and argue that the resulting grammar generates the same language as the original grammar.

Finally, for each pair of rules $A \to B, B \to b$ we add the rule $A \to b$, and for each pair of rules $A \to B, B \to CD$ we add the rule $A \to CD$. We finish by removing all rules of the form $A \to B$.

Give an algorithm to perform these changes, and argue that the resulting grammar generates the same language as the original grammar.

At this point the grammar is in CNF form.

21. Convert the following CFGs to CNF form—see Section 3.5.

i. G_1 has start variable S, terminal set $\{a, b\}$ and rules $S \to A$, $A \to B \,|\, SA \,|\, a$, $B \to C \,|\, b$, $C \to a$.

ii. G_2 has start variable S, terminal set $\{a, b\}$ and rules $S \to AB$, $A \to a \,|\, \lambda$, $B \to b \,|\, \lambda$.

iii. G_3 has start variable S, terminal set $\{a, b\}$ and rules $S \to AB$, $A \to X \,|\, BB \,|\, a$, $X \to Y \,|\, AA$, $Y \to A \,|\, AB$, $B \to b$.

22. Show that the following languages are not context free.

i. $A = \{a^h b^j c^k b^j c^k a^h \,|\, h, j, k \geq 0\}$.

ii. $B = \{a^m b^n c^m d^n \,|\, m, n \geq 0\}$.

iii. $C = \{z \,|\, z \in \{a, b, c\}^*$ and the number of a's, b's and c's in z are all equal$\}$.

iv. $D = \{z_1\#z_2\#z_3 \,|\, z_1, z_2, z_3 \in \{a, b\}^*$, the number of a's in z_1 equals $|z_2|$,
and the number of b's in z_2 equals $|z_3|\}$.

v. $E = \{z \,|\, z \in \{a, b, c, d\}^*$ and z contains equal numbers of a's and b's, and equal
numbers of c's and d's$\}$.

vi. $F = \{a^{h^2} \,|\, h \geq 1\}$.

Comment. Any CFL over a 1-character alphabet is a regular language. Give a proof without using this fact.

vii. $H = \{a^{2^h} \mid h \geq 0\}$.
Comment. Any CFL over a 1-character alphabet is a regular language. Give a proof without using this fact.

viii. $J = \{z_1\#z_2\#\cdots\#z_m \mid z_\ell \in \{a,b\}^*,\ 1 \leq \ell \leq m$, and for some $h, j, k,\ 1 \leq h < j < k \leq \ell$, $|z_h| = |z_j| = |z_k|\}$.

ix. $K = \{z_1\#z_2\#\cdots\#z_\ell \mid z_k \in \{a,b\}^*,\ 1 \leq k \leq \ell$, and for some $h, j,\ 1 \leq h < j \leq \ell, z_h = z_j\}$.

x. $L = \{a^h b^j \mid h$ is an integer multiple of $j\}$.

xi. N is the language consisting of all palindromes over the alphabet $\Sigma = \{a, b, c\}$ having equal numbers of a's and b's.

xii. $P = \{a^h b^h c^j \mid j > h\}$.

xiii. $Q = \{a^h b^j c^k \mid k = \max\{h, j\}\}$.

xiv. $R = \{a^j b^h c^j \mid j > h\}$.

23. This exercise concerns a variant of the Pumping Lemma, called Ogden's Lemma. Ogden's Lemma is useful for it can be used to show certain languages are not CFLs, languages for which the standard pumping lemma does not suffice.

i. Let L be a CFL and let $s \in L$ be a string in which some characters have been marked. Ogden's Lemma states that there is a constant $p = p_L$ such that if $s \in L$ is a string with at least p marked characters then s can be written as $s = uvwxy$, where

a. vx contains at least one marked character.

b. vwx contains at most p marked characters.

c. For all $i \geq 0$, $uv^iwx^iy \in L$.

Prove Ogden's Lemma.
Hint. Proceed in the same way as for the proof of the standard pumping lemma. In order to obtain analogs of Observations 1 and 2 you will need to count just the marked characters; you will also need to redefine the notion of "height" to count just those internal nodes with two subtrees both of which have marked characters.

ii. Prove a variant of Ogden's Lemma in which both v and x contain at least one marked character.

iii. Using Ogden's Lemma prove that the following languages are not context free.

a. $L = \{a^h b^j c^k d^\ell \mid h = 0$ or $j = k = \ell\}$.
Why will the ordinary Pumping Lemma fail to prove that L is not a CFL, i.e. show that any string $s \in L$ can be pumped so that all the resulting strings are in L.

b. $K = \{a^h b^h c^j \mid j \neq h\}$.
Hint. Consider the string $s = a^p b^p c^{p!}$.
Again, why will the ordinary Pumping Lemma fail to prove that L is not a CFL, i.e. show that any string $s \in L$ can be pumped so that all the resulting strings are in L.

24. Show the PDAs generated by applying the construction of Lemma 3.7.2 to the following CFGs.

i. $S \to AB, A \to AA, A \to a, B \to b$.

ii. $S \to XX \mid \lambda;\ X \to a$.

25. Consider the PDA shown in Figure 3.9.

    i. Modify this PDA so that it meets the requirements given in Section 3.7.1. (Do this carefully.)

    ii. Draw the trapezoidal diagram for the computation of the modified PDA recognizing input *aabb*.

    iii. Give the CFG generated by applying the construction of Section 3.7.1 to the modified PDA.

26. Repeat Question 25 for PDA M_2 shown in Figure 3.11 but use input *aacbb* for part (ii).

27. Repeat Question 25 for PDA M_3 shown in Figure 3.12 but use input *abcba* for part (ii).

28. For each of the language transformations T defined in the parts below, answer the following two questions.

    a. Suppose that L is a CFL. Show that $T(L)$ is also a CFL by giving a CFG to generate $T(L)$. Remember to explain why your solution is correct.

    b. Now suppose that L is recognized by a PDA. Show that $T(L)$ is also recognized by a PDA. Again, remember to explain why your solution is correct.

    Comment: The two parts are equivalent; nonetheless, you are being asked for a separate construction for each part.

    i. Let $w \in \{a,b,c\}^*$. Define $\text{Sbst}(w,a,b)$ to be the string obtained by replacing all instances of the character "a" in w with a "b". e.g. $\text{Sbst}(ac,a,b) = bc$, $\text{Sbst}(cc,a,b) = cc$, $\text{Sbst}(abc,a,b) = bbc$, $\text{Sbst}(acacac,a,b) = bcbcbc$.

       Let L be a language over the alphabet $\{a,b,c\}$. Define $T(L) = \text{Sbst}(L,a,b) = \{x \mid x = \text{Sbst}(w,a,b) \text{ for some } w \in L\}$.

    ii. Let $w \in \{a,b,c\}^*$. Define $\text{OneSubst}(w,a,b)$, or $\text{OS}(w,a,b)$ for short, to be the set of strings obtained by replacing one instance of the character "a" from w with a "b". e.g. $\text{OS}(acacac,a,b) = \{bcacac, acbcac, acacbc\}$.

       Let L be a language over the alphabet $\{a,b,c\}$.
       Define $T(L) = \text{OS}(L,a,b) = \{x \mid x \in \text{OS}(w,a,b) \text{ for some } w \in L\}$.

    iii. Let $w \in \{a,b,c\}^*$. Define Remove-c(w) to be the string obtained by deleting all instances of the character "c" from w. e.g. Remove-c(ab) $= ab$, Remove-c(cc) $= \lambda$, Remove-c(abc) $= ab$, Remove-c($acacac$) $= aaa$.

       Let L be a language over the alphabet $\{a,b,c\}$. Define $T(L) =$ Remove-c(L) $= \{x \mid x =$ Remove-c(w) for some $w \in L\}$.

    iv. Let $w \in \{a,b,c\}^*$. Define Remove-One-c(w) to be the set of strings obtained by deleting one instance of the character "c" from w. e.g. Remove-One-c($acacac$) $= \{aacac, acaac, acaca\}$.

       Let L be a language over the alphabet $\{a,b,c\}$.
       Define $T(L) =$ Remove-One-c(L) $= \{x \mid x \in$ Remove-One-c(w) for some $w \in L\}$.

    v. Let h be a mapping from Σ to Σ^*, that is h maps each character in Σ to a string of zero or more characters. Define $h(s)$ for a string $s = s_1 s_2 \cdots s_k$ to be the string $h(s_1)h(s_2)\cdots h(s_k)$.
       Define $T(L) = \{h(w) \mid w \in L\}$.

vi. Let h be a mapping from Σ to R where R is the set of regular expressions over alphabet Σ; i.e. $h(a)$ is a regular expression for each $a \in \Sigma$. Define $h(s)$ for a string $= s_1 s_2 \cdots s_k$ to be the set of strings specified by the regular expression $h(s_1)h(s_2)\cdots h(s_k)$.
Define $T(L) = \bigcup_{w \in L} h(w)$.

vii. Let Σ be an alphabet, and for each $a \in \Sigma$ let L_a be a CFL. Let $w = a_{i_1} a_{i_2} \cdots a_{i_k}$ be a string in Σ^*. Define $h(w) = L_{a_{i_1}} \circ L_{a_{i_2}} \circ \cdots \circ L_{a_{i_k}}$.
Define $T(L) = \bigcup_{w \in L} h(w)$; i.e. each $x \in T(L)$ can be written as $x = x_1 x_2 \ldots x_k$ for some $k \geq 0$, where each $x_i \in L_{a_i}$ for some $a_i \in \Sigma$.

29. i. Show that $\{b^j c^j d^j \mid j \geq 0\}$ is not a CFL.

ii. Conclude that $\{a^i b^j c^k d^l \mid i \geq 0 \text{ and } j = k = l\}$ is not a CFL.

iii. Let $L = \{a^i b^j c^k d^l \mid i = 0 \text{ or } j = k = l\}$. See Problem 28v. By using a suitable mapping h from Σ to Σ^*, show that L is not a CFL.
Comment. See Problem 23.iii.a. Note that Ogden's Lemma is not needed to show L is not a CFL.

30. A 2wayPDA is a variant of a PDA in which it is possible to go back and forth over the input, with no limit on how often the reading direction is reversed.

This can be formalized as follows. The input $x \in \Sigma^*$ to the PDA is sandwiched between symbols ¢ and \$, so the PDA can be viewed as reading string $x_0 x_1 x_2 \cdots x_n x_{n+1}$, where $x = x_1 x_2 \cdots x_n$, $x_0 = ¢$, and $x_{n+1} = \$$. The PDA is equipped with a *read head* which will always be over the next symbol to be read. At the start of the computation, the read head is over the symbol x_1, and the PDA is at its start vertex.

In general, the PDA will be at some vertex v, with its read head over character x_i for some i, $0 \leq i \leq n+1$. On its next move the PDA will follow an edge leaving v whose label includes Read x_i. This edge will also carry a label L or R indicating whether the read head moves left (so that it will be over symbol x_{i-1}), or right (so that it will be over x_{i+1}). Note that as we can do two moves in succession say to the left and then to the right, there is no need for a Read λ operation, which would leave the read head in the same place. There are two constraints: when reading ¢ only moves to the right are allowed and when reading \$ only moves to the left are allowed. If there is no move, the computation ends.

A 2wayPDA M recognizes an input x if it is at a recognizing vertex when the computation ends.

Give descriptions in English of 2way-PDAs for the following languages.

i. $L = \{a^i b^i c^i \mid i \geq 0\}$.
Note that by contrast with 2wayDFAs, this shows 2wayPDAs can recognize languages not recognized by PDAs.

ii. Show that if L_1 and L_2 are recognized by a 2wayPDA then so are $L_1 \cup L_2$ and $L_1 \cap L_2$.

31. Define 2wayDPDAs analogous to the 2wayPDAs in Question 30. Again, there is a read head, but the 2wayDPDA always has at most one move.

Show that 2wayDPDAs are closed under intersection, union and complement.

32. Show that the following languages can be recognized by 2wayDPDAs.

   $*$i. $H = \{a^i b^{2^i} \mid i \geq 1\}$.
   Hint. An intermediate data configuration will have $\$a^{i-h}b^{2^h}$ for $h \leq i$ on the stack.

   $*$ii. $J = \{a^i b^{i^2} \mid i \geq 0\}$.
   Hint. Find another helpful intermediate data configuration.

   $*$iii. $J_c = \{a^i b^{i^c} \mid i \geq 0\}$, where $c \geq 2$ is a fixed integer.

   $*$iv. $K = \{a^i b^j c^{i \cdot j} \mid i, j \geq 0\}$.
   Hint. You will need to count in two ways. At times the stack contents will be used as a counter. At other times the position of the read head in the string of c's will serve as a counter.

Chapter 4

Turing Machines

Alan Turing completed the invention of what became known as Turing Machines in 1936. He was seeking to describe in a precise way what it means to systematically calculate or compute; we might think of this as a process that can be described by one person and carried out by another. Before going further let me note that Turing Machines are conceptual devices; while they are easy to describe on paper, and in principle could be built, Turing had no intention of building them or having others build them. (Turing was not simply an "ivory-tower" mathematician; in the late 1940s he was one of the earliest users of one of the few computers then in existence — at Manchester University in England).

Let's consider what systematic computation meant in 1936. At the time there were machines and tools for helping with calculations: adding machines and slide rules; slide rules continued to be in widespread use until the early 1970s when calculators became relatively cheap. Also, there were card readers and sorters supporting data management (IBM was a leading supplier). There were even computers; but these were people whose job was to compute. A description of a computing room at Los Alamos in 1944 full of human computers can be found in Richard Feynman's semi-autobiography "Surely You're Joking Mr. Feynman," in the chapter "Los Alamos from Below"). Also, the central characters at the beginning of the movie "Hidden Figures" (who are African-American women) are human computers; they subsequently become computer programmers.

What Turing was trying to formalize were the processes followed when calculating something: the orbit of a planet, the load a bridge could bear, the next move to play in a game of chess, etc., though subsequently he proposed a much broader view of what could be computed (at least in principle).

The key question Turing asked was how do people compute? Essentially people could do three things: they could think in their minds, they could write down something (on a sheet of paper), or they could read something they had written down previously. As we all know, with too many sheets of paper it is hard to keep track of what is where, and consequently it is helpful to number or label the pages. One way of organizing this is to keep the pages bound in a book in numbered order. However, for reasons of conceptual simplicity, Turing preferred to think of the pages as being on a roll, or tape as he called it, with the pages being accessed in consecutive order by rolling or unrolling the roll (this is the way in which long documents were made and read in ancient times).

The actions with the roll or tape, as we shall call it henceforth, were very simple: one could read the current page, one could decide to erase the current page and write something new (pencil, not ink), one could decide to advance to the next page or go back to the previous page. But does this really capture everything? Suppose one wanted to go to page 150 and one was currently on page 28: well, then one needs to advance one page 122 times — a tedious process, for sure, but one that achieves the desired result in the end. The implicit assumption is that one can hold the count in one's mind. But what if the numbers are too large to hold in mind? What one would like to do is to write them on scraps of paper; however, this is cheating. What one has to do is use the tape pages as the scraps of paper. How this might be done is left as an exercise to the reader (see Exercise 2).

How did Turing abstract the notion of thinking? The crucial hypothesis was that a mind can hold only a finite number of different thoughts, albeit extremely large, not an infinite number. Why is this plausible? At the level of neurons (not that this was properly understood in Turing's day) simplifying a bit, each neuron is either sending a signal or not; as their number is finite, this yields only a finite number of possibilities — albeit a ridiculously large number of possibilities. At the level of elementary particles (electrons, protons, etc.), quantum physics states that there are only a finite number of states for each particle even if they are not fully knowable, and again this yields only a finite number of possibilities. So abstractly, thinking can be viewed as follows: the mind has a (large) finite number of states in which it could currently be, conventionally denoted by $q_0, q_1, ..., q_{s-1}$. Computation takes the following form: when in state q, on reading the current page take an action as already described (rewrite the page, move to an adjacent page) and enter a new state q' (possibly $q' = q$); this action is fully determined by the state q and the contents of the current page.

Turing proposed one more conceptual simplification. Just as the states of a mind are finite, what one can write on a single page is also finite. Accordingly he proposed representing the possible one-page writings by a finite set of symbols $a_0, a_2, ..., a_{r-1}$, which he called *characters*; he called the set $A = \{a_0, a_1, ..., a_{r-1}\}$ the *alphabet*. Now that "only" a single character was being written on each page, he called the pages *cells*. So an action consists of

1. reading the character written in the current cell;
2. possibly writing a new character in the current cell;
3. possibly moving one position to the left or right on the tape thereby changing which cell is the current cell;
4. possibly changing states.

A computational process can then be described by giving the particular rules to be followed for the calculation at hand. To specify this fully, a few more details are needed. One begins with the information on which one is calculating, the *input*, written on a leftmost segment of the tape, and the mind (now called the *finite control*) in an initial state (conventionally q_0). The computation finishes with the result, the *output*, written on a leftmost segment of the tape, or sometimes right after the input, and the finite control in a final state (conventionally q_f).

Turing's thesis was that these machines captured everything that could be done computationally. As we shall see, anything that can be done on a computer can in principle be done on a Turing Machine, albeit rather inefficiently (the slowdown is by "only" a polynomial factor). But why study them if they are horribly inefficient? The reason is that they are conceptually much simpler, and so they are very useful in understanding what is computationally possible, both in terms of whether something can be computed at all, and whether something can be computed efficiently (e.g. in polynomial rather than exponential time).

There is one more detail to mention. As Turing did not want to put any fixed bound on the length of a computation, although any computation to be useful would have to end eventually, he imagined that the tape of cells was unending or infinite to the right, and the computation would use as much of the tape as needed.

Now we proceed to make this more formal.

4.1 Turing Machines — A Formal Definition

As illustrated in Figure 4.1, a Turing Machine M comprises a finite control, an unbounded writable tape, and a read/write head. We sometimes use the abbreviation TM for a Turing Machine.

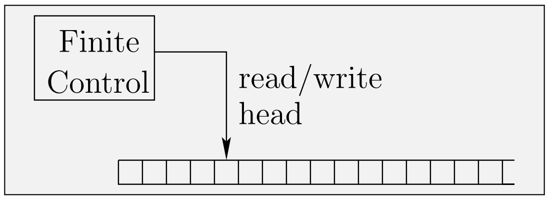


Figure 4.1: The Components of a Turing Machine

As with all the other machines we have seen, the finite control is simply a labeled, directed graph $G = (Q, E)$. $Q = \{q_0, q_1, \ldots, q_{s-1}\}$ are the possible states (or vertices) of M. We will specify the edges E shortly.

Each cell of the tape holds a single character drawn from the alphabet $A = \{a_0, a_1, \ldots, a_{r-1}\}$, which includes the symbols ¢ and $\not b$ (the symbol for blank). It is convenient to let $a_0 =$ ¢ and $a_1 = \not b$.

An edge (p, q) of the finite control is labeled by the triple (a, b, m) where p is the current state/vertex and a is the character in the cell under the read/write head. The action taken is to write b in the cell under the read/write head (replacing the a), move the read/write head one cell in direction m (one of Left or Right, L or R for short), and update the finite control to go to vertex q. M is deterministic if it has at most one possible move at each step and otherwise it is non-deterministic. In delta notation, $\delta(p, a) = \{(q_1, b_1, m_1), (q_2, b_2, m_2), \ldots, (q_l, b_l, m_l)\}$, where these are the l possible actions of the Turing Machine when at vertex p with an a in the cell under the read/write head.

To keep the read/write head from trying to move off the left end of the tape, we require that only the character ¢ can be written in the leftmost cell, and furthermore the only possible move when reading this cell is to move the read/write head to the right while leaving the ¢ unchanged, i.e. $\delta(p, ¢) = \{(q, ¢, R)\}$, for some q (see Figure 4.2). Also, to avoid confusing any other cell with the leftmost one, we prohibit the writing of ¢ in any other cell.

The computation starts with the read/write head over the leftmost cell, and with ¢$x\not b\not b\ldots$ on the tape, for some $x \in \Sigma^*$, where $\Sigma \subseteq A \setminus \{¢, \not b\}$. x is called the input to M; it is written using an alphabet Σ which is a strict subset of the character set used by TM M, and at the very least excludes characters ¢ and $\not b$.

A deterministic computation ends when M has no further move to make. There are two modes of computation. The first is to recognize an input, and the second is to compute a function $f(x)$.

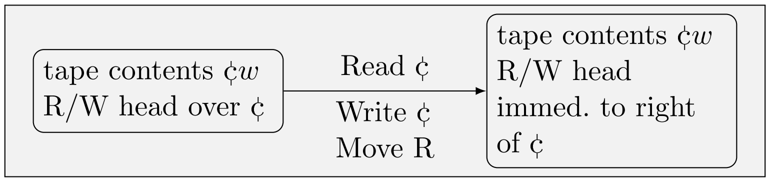


Figure 4.2: Moving from the left end of the tape

As we will argue shortly, the first mode provides the same computational power as the second mode. First, we will explain how each form of computation is carried out. An input x is recognized if the computation ends with M's finite control at a vertex (state) $q \in F \subseteq Q$, where F is the set of recognizing vertices. While to compute a function $f(x)$, M will write the string $f(x)$ at the left end of the tape, overwriting the input; the computation will end with ¢$f(x)$\$ at the left end of the tape, with the control being at a specified vertex q_f. The purpose of the \$ is to mark the end of the output, which may be needed in the case that the tape has been written to the right of where $f(x)$ is written. We require that $x, f(x) \in \Sigma^*$, where $\Sigma \subseteq A \setminus \{¢, b\!\!\!/, \$\}$.

We can also view function computation as a recognition problem. Suppose the input is a tuple (x, y) and it is recognized exactly if $f(x) = y$. Clearly, if we could test the inputs $(x, 0), (x, 1), (x, 2), \ldots$ in turn, stopping when $(x, f(x))$ was reached, we would in effect be computing $f(x)$. So from the perspective of what can and cannot be computed, there is no meaningful difference between language recognition and function computation. (See Exercise 9.)

In order to describe exactly the information held by the Turing Machine at each step of its computation, we introduce the notion of a *configuration.* This consists of the vertex (or state) of the Turing Machine, the tape contents (up to but not including the all blank portion of the tape), amd the position of the read/write head. We will write this as (q, uHv), where q is the current vertex, $uvb\!\!\!/b\!\!\!/\ldots$ is the current tape contents, and the read/write head is over the leftmost character in v, or if $v = \lambda$, it is over the $b\!\!\!/$ immediately to the right of u. Some authors use the more compact notation uqv.

A computation consists of a sequence of configurations $C_0 \vdash C_1 \vdash \ldots \vdash C_m$, where $C_i \vdash C_{i+1}$ denotes that configuration C_{i+1} follows configuration C_i by a single step of the Turing Machine's computation. We write $C_i \vdash^* C_j$ if C_j can be reached from C_i in 0 or more steps of computation. $C_0 = (q_0, H¢x)$ is the initial configuration on input x, and $C_m = (q, uHv)$ is a recognizing configuration if $q \in F$.

The language recognized by M is defined to be

$$L(M) = \{x \,|\, \text{there is a recognizing configuration } C_m \text{ such that } C_0 \vdash^* C_m \text{ and } C_0 = (q_0, H¢x)\}.$$

Formally, a Turing Machine M is specified by an 6-tuple $M = (\Sigma, A, Q, \text{start}, F, \delta)$, where A is the tape alphabet, $\Sigma \subseteq A - \{¢, b\!\!\!/\}$ is the input alphabet, Q is the set of vertices (or states) in the finite control, $F \subseteq Q$ is the set of recognizing vertices, and δ specifies the edges and their labels (i.e. what is read and the resulting action), and start is the start vertex.

We can always ensure that M has a single recognizing vertex and no move from the recognizing vertex if desired (see Exercise 4).

Writing Turing Machines in full detail is somewhat painstaking, so we limit ourselves to a few examples to build some intuition.

Example 4.1.1. TM M takes input $x \in \{0,1\}^+$, where x is viewed as an integer written in binary. The output is the integer $x+1$ also in binary. To make our task a little simpler, we will write x in left to right order going from the least to the most significant bit. Thus, for example 6 is written as the binary string 011 ($0 \times 1 + 1 \times 2 + 1 \times 4$).

In general, if $x = 1^h0y$, with $y \in \{0,1\}^*$, then $x+1 = 0^h1y$; while if $x = 1^h$ then $y = 0^h1$.

This suggests a fairly simple implementation: read from left to right over the initial all 1s prefix of x, changing the 1s to 0s and change the next character, either a 0 or a $\not b$, to a 1. The finite control is shown below.

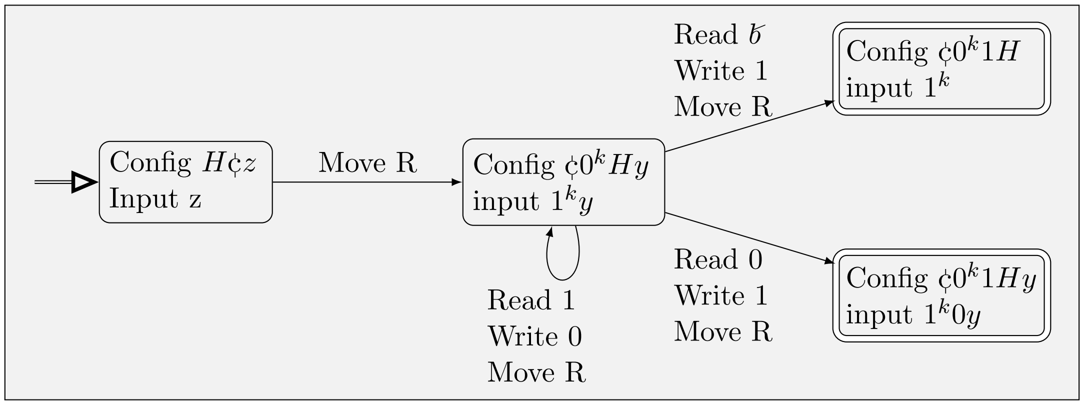


Figure 4.3: Turing Machine that adds 1 to its input; the nodes at which the computation ends are shown as recognizing nodes.

4.1.1 Multi Tape Turing Machines

To facilitate the construction of more complex Turing Machines, we make them more versatile.

A multi-tape Turing Machine is very similar to the 1-tape Turing Machine we previously described, except that it can have several tapes — this is a fixed number, specified for each individual machine. The number of tapes does not depend on the input.

Suppose the Turing Machine at hand has k tapes. Each tape will have its own read/write head. Each action will depend on the current vertex (state), and on the contents of the k cells under the read/write heads. The move will rewrite the content of these cells (which might be the previous contents), shift each read/write head left or right or allow it to stay put as desired, and advance to a new vertex (which might be the same vertex). We denote the move of each read/write head by one of L, R, P for a move to the left, right, or staying put, respectively. (With multiple tapes it is no longer possible to simulate staying put by a move to the right followed by a move to the left, for we might want some R/W heads to stay put and others to move right, for example). As before each tape will have a ¢ as its leftmost character, with the rules w.r.t. ¢ unchanged.

We begin with an example.

Example 4.1.2. Input: $x\#p$
Output: Index in x of the leftmost occurrence of p in x, and 0 if there is no occurrence. i.e. if $x = x_1 x_2 \dots x_n$ and if $x_{i+1}x_{i+2}\dots x_{i+m} = p$ is the first occurrence of p in x then output $i+1$ in binary on tape 2; if there is no such i, output 0.

We use a 4-tape machine which proceeds as follows.

```
Step 1. Write 1 on Tape 2.
Step 2. Copy $p$ to Tape 3.
Step 3. for $i = 1$ to $n - m + 1$ do (* $i$ is the number on Tape 2 *)
            if $p = x_i x_{i+1} \dots x_{i+m-1}$
                then{move to a recognizing vertex; i.e. end the computation}
                else {add 1 to the value on Tape 2} (see Example 4.1.1)
Step 4. Write 0 on Tape 2 (* the case that there is no match *)
```

Next, we show that having multiple tapes provides no additional recognition power. Specifically, given a k-tape Turing Machine M, we show there is a 1-tape Turing Machine $\widetilde{M}$ that recognizes the same language.

We use $2k$ *tracks* on $\widetilde{M}$'s single tape to simulate M's k tapes. The tracks are organized in pairs, with each pair being used to hold the contents of one of M's tapes and the position of its read/write head. Specifically, if the tape configuration is (uHv), then the first track will store the string uv at its left end, and the second track will store the string $(\not b)^{|u|}H(\not b)^{|v|-1}$ — the parentheses are present for clarity. (See Figure 4.4.) The i-th cell on $\widetilde{M}$'s single tape stores the contents of the i-th cell

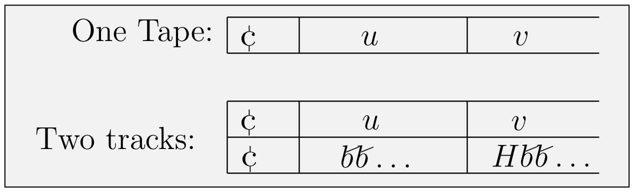


Figure 4.4: Two tracks simulate one tape

on each of the $2k$ tracks. Thus, if we represent the tape as a semi-infinite array with $2k$ rows, each of its rows corresponds to a single track, each entry represents a single cell on a single track, while each column corresponds to a single one of $\widetilde{M}$'s cells. See Figure 4.5.

To simulate a single one of M's moves or actions, $\widetilde{M}$ begins with its read/write head on the cell storing ¢ at the left end of its tape knowing the action it is going to simulate, i.e. the k writes and the k moves that M will perform on its k tapes. The simulation sweeps across $\widetilde{M}$'s tape from left to right going as far as necessary and then sweeps back, recording M's k updates to its k tapes in the correct locations on its $2k$ tracks, and also gathering the values of the k characters under the new positions for M's read/write heads.

In more detail, $\widetilde{M}$'s finite control will consist of supervertices with $4k + 2$ components. The first component records whether $\widetilde{M}$ is performing its left or right sweep. The second component records M's current vertex. The remaining $4k$ components are used differently on the two sweeps. Right sweep: The $(4i - 1)$-th component records the value to be written under the read/write head on M's i-th tape, and the $4i$-th component records the direction in which this read/write head

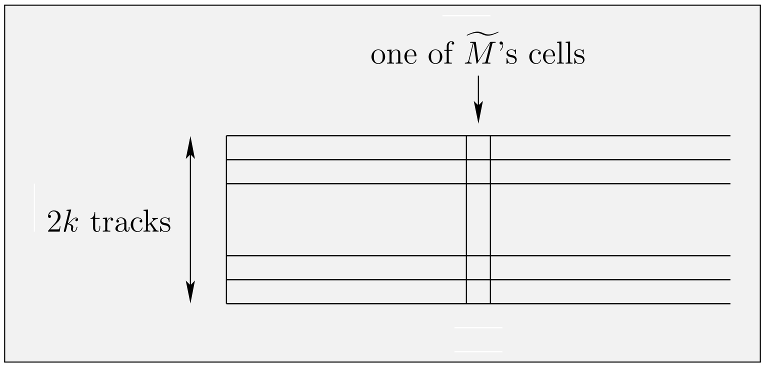


Figure 4.5: $\widetilde{M}$'s cells

is to move (L, R, or P). The $(4i+1)$-st component records whether the H on track $2i$ has been encountered yet. We explain the role of the $(4i+2)$-nd component later. What this means is that each time an H is encountered, there is a move in $\widetilde{M}$'s finite control from the current vertex to a new vertex which represents the now correct information. When an H is encountered on track $2i$, the value recorded by the $(4i-1)$-th component is written on track $2i-1$, and if the move is R, then the H on track $2i$ is replaced by $\not b$, and on the next move right in this sweep, an H is written on track $2i$; we use the $(4i+2)$-nd component to record when this update is needed and not yet made. When k H's have been encountered, after one more move to the right has been made, the right sweep is complete.

Left sweep: By the end of the sweep, the $(4i-1)$-th component will record the value under the read/write head on M's i-th tape following M's current move. The $4i$-th component continues to record the move being made in M's current step, and the $(4i+1)$-st component is not used. When the H on the $2i$-th track is encountered, the value on the $(2i-1)$-st track is recorded in the $(4i-1)$-th component by changing vertices in $\widetilde{M}$'s finite control, and if the move is an L it is implemented as for the R move in the right sweep, but in the case the recording of the value under the R/W head is done after the move of the R/W is carried out.

When the left end of the tape is reached, the values in the components are updated to reflect the next move, which again is done by $\widetilde{M}$ moving to a new vertex.

If the non-blank contents on M's longest tape cover m cells, then simulating one step of M's computation will take $\widetilde{M}$ up to $2m$ steps.

$\widetilde{M}$ will recognize its input exactly if it reaches a vertex v which corresponds to one of M's recognizing vertices (i.e., the second entry in v's tuple is one of M's recognizing vertices).

Comment. Strictly speaking, $\widetilde{M}$ begins with tape contents $¢x\tilde{\not b}\tilde{\not b}\ldots$ which it rewrites to $(¢H)^k x_1 \not b^{2k-1} x_2 \not b^{2k-1} \ldots x_n \not b^{2k-1}$, where $x = x_1 x_2 \ldots x_n$, and each sequence of $2k$ symbols in the rewritten tape contents denotes the contents of the $2k$ tracks in one of $\widetilde{M}$'s cells.

We have shown:

Lemma 4.1.3. *Let M be a k-tape Turing Machine. Then there is a 1-tape Turing Machine $\widetilde{M}$ such that $L(\widetilde{M}) = L(M)$.*

Finally, we comment on the number of steps performed by $\widetilde{M}$ on input x, denoted by $\widetilde{M}(x)$ for brevity, assuming $M(x)$ performs t steps in its computation.

Lemma 4.1.4. *Suppose k-tape Turing Machine M on input x performs t steps of computation. Then $\widetilde{M}$ on input x performs at most $2|x| + 2 + 2t \cdot (1 + \max\{|x|, t\}) = O(t(|x| + t))$ steps of computation.*

Proof. The initial rewriting of $\widetilde{M}$'s input requires $2|x| + 2$ steps to rewrite each of the $|x| + 1$ cells in use and return to having the R/W over the leftmost cell (remember that the first $\not b$ cell has to be visited in order to ensure that all the input has been seen).

M can use at most $\max\{|x|+1, t+1\}$ cells over the course of t steps of computation. Therefore each step of simulation by $\widetilde{M}$ uses at most $2\max\{|x| + 1, t + 1\}$ of $\widetilde{M}$'s steps. Consequently, simulating the entire t steps of M's computation takes $\widetilde{M}$ at most $2|x| + 2 + 2t \cdot (1 + \max\{|x|, t\})$ steps. □

4.1.2 The Universal Turing Machine

So far we have needed to create a separate Turing Machine for each computation we wish to perform. But now we will show that a single general purpose Turing Machine, akin to a computer, suffices. This Universal Turing Machine U will take two inputs: A description of a Turing Machine M and an input x to M. It will then simulate M's computation on input x. We provide a description of how to simulate deterministic Turing Machines. The extension to non-deterministic machines is left as an exercise.

We start by describing a general purpose 3-tape Turing Machine $\widetilde{U}$, which we can then simulate using a 1-tape Turing Machine U, by applying Lemma 4.1.3. Tape 1 holds $\widetilde{U}$'s input. Tape 2 will hold a copy of M's tape. Tape 3 will hold M's current vertex/state. $\widetilde{U}$ will then repeatedly simulate the next step of M's computation.

Since the possible M that $\widetilde{U}$ needs to be able to simulate can have arbitrarily large finite controls (graphs) and arbitrarily large tape alphabets, $\widetilde{U}$ cannot use M's alphabet to carry out the simulation, nor can it store M's finite control in its own possibly smaller finite control. Instead, $\widetilde{U}$ will use a single fixed alphabet which it uses to encode (i.e. write) a precise description of M's finite control and tape contents. There are many ways of doing this; we proceed as follows. We can write a Turing Machine M as a tuple $M = (\Sigma, A, Q, \text{start}, F, \delta)$ where Σ is the input alphabet, A is the tape alphabet, with $\Sigma \subset A - \{¢, \not b\}$, Q is its set of vertices or states, start $\in Q$ is the start vertex, $F \subseteq Q$ is the set of recognizing vertices, and δ is the transition function.

We will describe how to write out the description of a deterministic TM M in full detail. Let $A = \{a_0, a_1, \dots, a_{r-1}\}$, and recall that $¢ = a_0, \not b = a_1$. We will represent a_i by the $\lceil \log_2 r \rceil$-bit binary string that denotes the integer i. A can be written using the following 5-character alphabet: 0, 1, (,), and "," (i.e. the comma itself), in the form $(\text{binary}(a_1), \text{binary}(a_2), \dots, \text{binary}(a_{r-1}))$, where $\text{binary}(a_i)$ denotes the $\log r$-bit representation of a_i in binary. Σ can be written in the same way. Let $Q = \{q_0, q_1, \dots, q_{s-1}\}$. Then we will represent q_j by the $\lceil \log_2 s \rceil$-bit binary string that denotes the integer j. Again, we can use the same 5-character encoding for Q. F can be written in the same way, while the vertex start just requires the listing of a single vertex in binary. We need to ensure that $F \subseteq Q$ and start $\in Q$, of course. Each entry in δ comprises a 5-tuple (p, q, a, b, m), where p denotes the current vertex, a the character in the cell under the read/write head, q the vertex being moved to, b the character being written, and m the move being made (L or R). This tuple can be written as a sequence of five binary strings separated by commas and enclosed in parentheses. M itself can be written as the comma separated encoding of its 6 constituent elements, enclosed in parentheses.

Thus we obtain an encoding of M as a string using 5 characters. In turn, this can be rewritten as a binary string by encoding these 5 characters in binary, using 3 bits for each character. Thus each 1-tape deterministic Turing Machine can be described by a distinct binary string.

It is also straightforward to write an input x to M in binary using the encoding for alphabet A: this requires $3\lceil \log_2 r \rceil$ bits per character in x.

The input to $\widetilde{U}$ is a string s which is supposed to be the encoding of a Turing Machine M followed by the encoding of an input x to M. So first stage of $\widetilde{U}$'s computation will be to check that its input does have this form. This requires many instances of pattern matching. For example, $\widetilde{U}$ needs to confirm that the encodings for all the characters in A have the same length and that they are all distinct. It needs to confirm that $\Sigma \subset A$ and $a_0, a_1 \in A - \Sigma$. In more detail, to confirm $¢ = a_0 \in A - \Sigma$, $\widetilde{U}$ confirms that a_0 does not match any of the characters in Σ. Similar checks need to be made for Q, F and start, and for δ. $\widetilde{U}$ also needs to check that M cannot make any illegal moves: the only time it can write ¢ is when it is reading ¢ and then it must move to the right. We leave these further painstaking but straightforward details to the reader.

Now $\widetilde{U}$ is ready to carry out the simulation. To this end, $\widetilde{U}$ will use three tapes. Tape 1 holds its input. Tape 2 is used to maintain an encoding in binary of the characters on M's single tape; to facilitate identifying the start and end of each character encoding, they will be separated by a third character, # say ($\widetilde{U}$ is allowed to use a constant number of additional characters beyond 0,1, its ¢ and its $\not b$). The read/write head for Tape 2 will be used to keep track of the position of M's read/write head. Finally, $\widetilde{U}$ stores the encoding of M's current vertex on Tape 3. To perform one step of simulation, $\widetilde{U}$ has to find the 5-tuple in the encoding of δ that represents the current legal move, if any. Thus, if a is under M's read/write head and p is M's current vertex, $\widetilde{U}$ has to find the at most one tuple that contains p as its first entry and a as its third entry. This can be done by matching the contents of Tape 3 and the encoded character under the read/write head for the second tape with the encoding of δ on Tape 1.

If no match is found, then the simulation ends. A transition to $\widetilde{U}$'s recognizing vertex occurs if M's current vertex is in F. Again, to make this determination requires a matching process.

On the other hand, if a match with tuple (p, q, a, b, m) is found then the contents of $\widetilde{U}$'s Tape 2 and Tape 3 are updated. On Tape 3 p is overwritten by q, on Tape 2 a is overwritten by b, and finally on Tape 2, the read/write head is moved one encoded character in the direction m (here the # separators make this task easy). There is one more detail to add regarding this move: a move to the right may reach a previously unwritten portion of M's tape, in which case $\widetilde{U}$ has to write the encoding of $\not b$, M's blank, followed by a # at the current right end of the written portion of Tape 2. Once this is all done, M proceeds to the next step of simulation.

Theorem 4.1.5 (Turing, 1936)**.** *There is a 1-tape Universal Turing Machine U that given an input $w = \langle M, x \rangle$ recognizes its input w exactly if M, a 1-tape Turing Machine, recognizes its input x.*

Proof. The above description showed there is a 3-tape Universal Turing Machine $\widetilde{U}$. But by Lemma 4.1.3, $\widetilde{U}$ can be simulated by a 1-tape Turing Machine U which recognizes the same language. □

The Universal Turing Machine can be thought of as either a general purpose computer or as a compiler. Historically, its significance is that it showed that a general purpose computing machine could carry out all the tasks being done by individual specialized devices. It is worth noting that this has already been realized a century earlier in Babbage's work.

Another important point is that U's control has a fixed size. What gives it its computational power is its unbounded memory (in the form of its tape).

Finally, we comment on the number of steps needed by U to simulate a computation of a Turing Machine M on an input x, supposing that computation $M(x)$ runs in t steps.

Lemma 4.1.6. *Suppose the encoding of Turing Machine M uses b bits, and suppose that M on input x runs in t steps. Then U on input (M, x) runs in $O(b^2(x+t)^2)$ steps.*

Proof. We assert without further proof that the pattern matching algorithm, when comparing a pattern of length ℓ to a text of the same length, takes $O(\ell)$ steps. We first bound the number of steps performed by $\widetilde{U}$, the multi-tape version of the universal machine. The initial set-up of $\widetilde{U}$'s tapes takes $O(b|x|)$ steps. Following this, each step of simulation will take at most $O(b)$ steps (as this involves a sequence of comparisons of strings representing the encoding of a character or a vertex, which each run in time linear in the length of these encodings). Thus $\widetilde{U}$ performs at most $O(b|x| + bt)$ steps.

$\widetilde{U}$ will use at most $b(t + |x| + 1)$ cells on each of its tapes. Thus, by Lemma 4.1.4, the 1-tape universal machine U will use at most $O(b(|x| + t) \cdot b(t + |x| + 1)) = O(b^2(t+x)^2)$ steps. □

4.2 Programs and Turing Machines are Equivalent

Next, we wish to argue that the computational power of programs and Turing Machines are the same. To this end, we consider a very simple computing model and programming language. In this language, a program consists of a sequence of incrementally numbered instructions, i.e., instruction 1, instruction 2, etc. Our programs run on a computer with a memory $M[0, m-1]$ of w-bit integer values (we discuss the issue of integer size below). For readability we allow variable names such as x, y, z, but each variable corresponds to a specific location in M. There are eight different instructions in all, which involve variables and constants c which are integer values such as 0, 1, -5, etc.

1. $x \leftarrow c$; (assign x a value specified in the program).
2. $x \leftarrow M[c]$; (assign x the value stored in $M[c]$).
3. $x \leftarrow M[y]$; (assign x the value stored in $M[y]$).
4. $M[c] \leftarrow x$; (store in $M[c]$ the value of variable x).
5. $M[y] \leftarrow x$; (store in $M[y]$ the value of variable x).
6. $x \leftarrow y + z$; (store in x the value $y + z$).
7. $x \leftarrow -y$; (store in x the value $-y$).
8. **if** $x \geq 0$ **then go to** l; (see below).

A program executes starting at its first instruction, and performs the instructions in the program sequentially except when it performs an instruction "**if** $x \geq 0$ **then go to** l", and then if $x \geq 0$ it instead executes instruction l next. When it seeks to execute a non-existent instruction subsequent to the last instruction, the computation terminates.

We note that we could replace instructions 2–5 with the single instruction $M[z] \leftarrow M[y]$. For example, to carry out instruction 2, we would proceed as follows. We let $M[a]$ be the memory location storing the value of variable x (this is what we mean by a variable: it is a shorthand for a specific memory location, or equivalently, each variable has its own memory location). We assign a to variable z (assign $z \leftarrow a$); next we assign value c to variable y ($y \leftarrow c$); finally we perform the operation $M[z] \leftarrow M[y]$. Note that this is the same as $M[a] \leftarrow M[c]$, which amounts to $x \leftarrow M[c]$. Implementing the remaining assignments is left as an exercise (see Exercise 7).

Note that there is no output, and the input is implicit: it is provided via the initial values in the memory M. We therefore define recognition of an input v by whether the program's computation terminates on input v or not.

For the program to be able to access a memory location $l \leq m - 1$ it must be able to reference it, and if l is not a constant named in the program then this location can be accessed only via a reference of the form $M[y]$, where y is a variable, which is itself stored in some memory location. This means that $0 \leq y \leq m-1$, and consequently the memory location storing y, and by symmetry each memory location, must be able to store $\lceil \log m \rceil$-bit numbers. Consequently, we will need the word length w of a memory location to satisfy $w \geq \log m$. Indeed, this is a feature of the RAM model, the standard model used in analysis of algorithms, which requires $w = \Omega(\log m)$, where m is the size of the memory being used.

To enable simulation by a Turing Machine we assume the input to the Turing Machine comprises a pair (w, x), where w indicates the size of integers being stored in each memory location, namely up to $m - 1 = 2^w - 1$, and x indicates the contents of the memory in locations $M[0 : m - 1]$. As we are storing both positive and negative integers, a further bit is required to indicate the sign of each integer.

Also, we need to specify what happens if the computation ever seeks to use more than w bits for an integer: we can have the program increase its counter to a value one larger than the number of lines in the program, which causes the computation to end; i.e. the input is recognized. (It might be more appealing if the input were not recognized in these circumstances; this could be achieved by branching to an infinite loop, but this seems less simple, which explains the choice we made.)

We now describe a Turing Machine simulation of a program P on input (w, x). Tape 1 is used to hold the input (w, x). Tape 2 will be used to store the contents of $M[0, m - 1]$, as a series of pairs of $w + 1$ bit numbers separated by # markers; the first number in each pair is the index of the location, and for simplicity the pairs are stored in increasing order based on the location index; the second number is the value stored in the location. Tape 3 is used to store the number of the next instruction. Three more tapes are used to carry out the instructions. We explain how to simulate a few of the instructions; the others are handled similarly.

$M[y] \leftarrow x$: The value stored in x, i.e. in the memory location holding variable x, is copied to Tape 4; similarly the value stored in y is copied to Tape 5. Now, x, the value on Tape 4, is copied to memory location $M[y]$, the location being provided by the contents of Tape 5. Finally, the instruction counter is incremented by 1.

$x \leftarrow y + z$: The values stored in y and z are copied to Tapes 4 and 5, respectively. They are then added together, with the result being written on Tape 6. This value is then copied to the memory location for variable x. Finally, the instruction counter is incremented by 1.

if $x \geq 0$ **then go to** l: The value stored in x is copied to Tape 4. If it non-negative, the instruction counter is reset to l and otherwise it is incremented by 1.

Once again, we can simulate the 6-tape simulating Turing Machine with a 1-tape Turing Machine.

We have shown:

Theorem 4.2.1. *Every language L recognized by a program P is recognized by a 1-tape Turing Machine.*

Generality of the Programming Language It is straightforward to implement standard programming language instructions using our limited instruction set, such as logic operations, the "if then else" command, while loops, procedure calls, multiplication, etc.

Also it should be clear that we could write a program to simulate a Turing Machine. In particular, we could write a program P_U that simulates the Universal Turing Machine U, and thus P_U can be thought of as a universal program. It is a program akin to a compiler; that is it takes two inputs: a Turing Machine M (or equivalently a program P) and an input x for M and it simulates the execution of $M(x)$.

Exercises

1. Show that a binary alphabet suffices. That is, given Turing Machine M with alphabet A, create a new Turing Machine M_b with a binary alphabet that carries out the same computation as M. Furthermore, $x \in L(M)$ exactly if $b(x) \in L(M_b)$, where $b(x)$ denotes the encoding of x in binary. Hint. Encode each character of A using $\lceil \log A \rceil$ binary symbols, and make M_b's finite control sufficiently larger than that of M so that it can "read" and "write" a character of M.

2. This exercise shows how to count out a number larger than can be kept in mind.

   Design a Turing Machine with a 4-character alphabet $A = \{0, 1, *, \not{b}\}$. Suppose the input is a number n written in binary at the left end of the tape; at this point the rest of the tape is blank, i.e. every other cell stores a $\not{b}$. Design a Turing Machine to write n $*$ characters after the input number. You may alter the input number as the computation proceeds. Also assume you are given a procedure (a collection of vertices and actions) that subtracts 1 from a positive number written in consecutive cells at the left end of the tape. Finally, describe the Turing Machine in reasonably high level terms such as: move to the right while reading cells with a '$*$' until a cell without a '$*$' is reached.

3. Describe a 3-tape Turing Machine that adds two binary numbers x and y. They are written at the left ends of Tapes 1 and 2, respectively, in the form $¢x$ and $¢y$. The result, the binary number $z = x + y$, is to be written on Tape 3, in the form $¢z$. All three numbers are written, in left to right order, from least significant to most significant bit.

4. Let M be a 1-tape Turing Machine.

   i. Describe a 1-tape Turing Machine M' which has a single recognizing vertex and such that $L(M') = L(M)$.

   ii. Describe a 1-tape Turing Machine M'' which has a single recognizing vertex, no move from the recognizing vertex, and such that $L(M'') = L(M)$.

5. Describe how to modify the Universal Turing Machine described in this chapter so that it can simulate non-deterministic Turing Machines. The modified Universal Turing Machine is to be

non-deterministic. To ensure you create a suitable non-deterministic universal Turing Machine U, we will place the following constraint on the number of steps it performs. Suppose U is simulating non-deterministic Turing Machine M running on input x, and suppose M has a recognizing computation which completes in t steps. Furthermore suppose the encoding of M uses b bits. Then U, when running on input (M, x), needs to have a recognizing computation that uses $O(b^2(|x| + t)^2)$ steps.

6. Let N be a non-deterministic 1-tape Turing Machine. Describe a deterministic 1-tape Turing Machine M such that $L(M) = L(N)$.
   Hint. To recognize an input x, N needs to have a recognizing computation. Thus M's task on input x is to determine if N has a recognizing computation on this input. M will need to explore the possible computations in a breadth-first fashion, for some non-recognizing computations may have infinite length.

7. Consider the version of the programming language with a single assignment statement $M[z] \leftarrow M[y]$, in place of statements 2–5. Show how to implement statements 3–5.

8. Consider the 8-instruction programming language given in the text. Show how to implement the following instructions in this language.

   i. **go to** l, meaning execute the statement with label l next.
   ii. $z = x \wedge y$, where x, y and z are variables that can take the values 0 or 1. You may use the instruction **go to** l.
   iii. $z = x \vee y$, where x, y and z are variables that can take the values 0 or 1. You may use the instruction **go to** l.
   iv. $z = \neg y$, where x and y are variables that can take the values 0 or 1. You may use the instruction **go to** l.
   v. **if** $x \leq 0$ **then go to** l.
   vi. **if** $x = 0$ **then go to** l.
   vii. The "**if** $x = 0$ **then** {first sequence of statements} **else** {second sequence of statements}" construct, where the sequences of statements are given. You may assume the first sequence of statements comprises ℓ_1 statements, and the second sequence ℓ_2 statements.
   viii. The construct "**while** $x = 0$ **do** {sequence of statements}, where the sequence of statements is given. You may use the instruction **if** $x = 0$ **then go to** l. You may assume the sequence of statements comprises m statements.
   ix. Multiplication: the input comprises the variables x and y storing positive integers and the task is to compute $z = x \times y$.
   Hint: Implement the standard long multiplication algorithm.

9. Let f be a function mapping non-negative integers to non-negative integers. Suppose you are given a Turing Machine M that takes inputs (x, y), where x and y are non-negative integers, and that eventually halts if $y = f(x)$ and runs forever if $f(x) \neq y$. Give a Turing Machine $\widetilde{M}$ that on input x computes the value $f(x)$.

Chapter 5

Decidability and Undecidability

In the previous chapter we saw that Turing Machines and programming languages are equivalent in terms of the languages they can recognize. However, recognition is not the same thing as a membership test. If a language L is recognized by program P it means that on an input $x \in L$, $P(x)$ will eventually terminate (i.e. have its program counter take on a value greater than the number of instructions in the program), but if $x \notin L$ then this does not happen, which means that $P(x)$ runs forever.

In this chapter we are going to study two classes of languages: those that can be recognized by programs (or Turing Machines), and the smaller class of languages which have membership tests. The latter class are called *computable* or *decidable* languages.

To do this it will be helpful to have programs that can produce output. To this end we add the following instructions to our programming language:

Write(c), where c is a suitable constant, e.g. 0, 1, "Recognize", etc.

Write(x), where x is a variable.

5.1 Computable Languages

A computable language is one for which there is an algorithmic membership test, that is there is a program which determines whether an input string x is in the language or not.

Definition 5.1.1. *$L \subseteq \Sigma^*$ is* computable *or* decidable *if there is a program P with two possible outputs, "Recognize" and "Reject" and on input $x \in \Sigma^*$, P outputs "Recognize" if $x \in L$ and "Reject" if $x \notin L$. We will also say that P* decides *L (in the sense of* decides *set membership).*

Remark. Often, one wants to compute some function $f(x)$ of the input x. By allowing 2-input programs we can define computable functions f as follows: f is computable if there is a 2-input program P, such that on input (x, y), P outputs "Recognize" if $f(x) = y$ and "Reject" if $f(x) \neq y$.

5.1.1 Encodings

Often, we will want to treat the text of one program as a possible input to another program. To this end, we take the conventional approach that the program symbols are encoded in ASCII or more simply in binary. We do the same for the input alphabet. Of course, if we are not trying to treat a program as a possible input to another program, we can have distinct alphabets for programs and inputs.

We already described how to encode Turing Machines in the previous chapter. As this is rather tedious, henceforth we will leave the description of correct encodings to the reader (to imagine). We will simply use angle brackets to indicate a suitable encoding in a convenient alphabet (binary say). So $\langle M\rangle$ denotes an encoding of machine M, $\langle w\rangle$ an encoding of string w (note that w is over some alphabet Σ, while $\langle w\rangle$ may be in binary). We also use $\langle M,w\rangle$ to indicate the encoding of the pair M and w. But even this can prove somewhat elaborate, so sometimes we will simply write M when we mean $\langle M\rangle$. As a rule, when M is the input to another program, then what we intend is $\langle M\rangle$, a proper encoding of the description of M.

We are now ready to look at the decidability of some languages.

5.1.2 Decidability of Regular Language Properties

Example 5.1.2. Rec-DFA $= \{\langle M,w\rangle \mid M$ is a DFA and M recognizes input $w\}$.

Claim 5.1.3. Rec-DFA *is decidable.*

Proof. To prove the claim we simply need to give an algorithm $\mathcal{A}$ that on input $\langle M,w\rangle$ determines whether M recognizes its input. Such an algorithm is easily seen at a high level: $\mathcal{A}$ first checks whether the input is legitimate and if not it rejects. By legitimate we mean that $\langle M,w\rangle$ is the encoding of a DFA, followed by the encoding of a string w over the input alphabet for M. If the input is legitimate $\mathcal{A}$ continues by simulating M on input w: $\mathcal{A}$ keeps track of the the current destination vertex as M reads its input w. When (in $\mathcal{A}$'s simulation) M has read all of w, $\mathcal{A}$ checks whether the destination vertex M has reached is a recognizing vertex and outputs accordingly: $\mathcal{A}$ outputs "Recognize" if M's destination on input w is a recognizing vertex, and $\mathcal{A}$ outputs "Reject" otherwise.

$$\mathcal{A}(\langle M,x\rangle) = \begin{cases} \text{"Recognize"} & \text{if } M \text{ recognizes } w \\ \text{"Reject"} & \text{if } M \text{ does not recognize } w \end{cases}$$

The details are a bit more painstaking: given the current vertex reached by M and the next character in w, $\mathcal{A}$ looks up the vertex reached by scanning the edge descriptions (the triples $(p,a)\to q$). In yet more detail, $\mathcal{A}$ stores the current vertex in a variable, the input w in an array, an index indicating the next character of w to be read, and the DFA in adjacency list format.

The details of how to implement algorithm $\mathcal{A}$ should be clear at this point. As they are not illuminating we are not going to spell them out further. □

Note. Henceforth, a description at the level of detail of the first paragraph of the above proof will suffice. Further, we will take it as given that there is an initial step in our algorithms to check that the inputs are legitimate.

Let P_{RecDFA} be the program implementing the just described algorithm $\mathcal{A}$ deciding the language of Example 5.1.2. So P_{RecDFA} takes inputs (M,x) (strictly, $\langle M,x\rangle$) and outputs "Recognize" if M recognizes x and outputs "Reject" if M does not recognize x. This is a notation we will use repeatedly. If Prop is a decidable property (i.e. the language $L=\{w \mid \text{Prop}(w) \text{ is true}\}$ is decidable) then P_{Prop} will be a program that decides L; we will also say that P_{Prop} decides Prop.

Example 5.1.4. Rec-NFA $= \{\langle M,w\rangle \mid M$ is a description of an NFA and M decides $w\}$.

Claim 5.1.5. Rec-NFA *is decidable.*

Proof. The following algorithm $\mathcal{A}_{\text{Rec-NFA}}$ decides Rec-NFA. Given input $\langle M, w\rangle$, it simulates M on input w by keeping track of all possible destination vertices on reading the first i characters of w, for $i = 0, 1, 2, \cdots$ in turn. M recognizes w exactly if one of its destinations on reading all of w is a recognizing vertex, and consequently $\mathcal{A}_{\text{Rec-NFA}}$ outputs "Recognize" if it finds M on input w has a destination vertex which is a recognizing vertex and outputs "Reject" otherwise. □

Example 5.1.6. Rec-RegExp $= \{\langle r, w\rangle \mid r$ is a regular expression that generates $w\}$.

Claim 5.1.7. Rec-RegExp *is decidable.*

Proof. The following algorithm $\mathcal{A}_{\text{Rec-RegExp}}$ decides Rec-RegExp. Given input $\langle r, w\rangle$, it begins by building an NFA M_r recognizing the language $L(r)$ described by r, using the procedure from Chapter 2, Lemma 2.5.1. $\mathcal{A}_{\text{Rec-RegExp}}$ then forms the encoding $\langle M_r, w\rangle$ and simulates the program $P_{\text{Rec-NFA}}$ from Example 5.1.4 on input $\langle M_r, w\rangle$. $\mathcal{A}_{\text{Rec-RegExp}}$'s output ("Recognize" or "Reject") is the same as the one given by $P_{\text{Rec-NFA}}$ on input $\langle M_r, w\rangle$.

This is correct for M_r recognizes w if and only if w is one of the strings described by r. □

This procedure is taking advantage of an already constructed program and using it as a subroutine. This is a powerful tool which we are going to be using repeatedly.

Example 5.1.8. Empty-DFA $= \{\langle M\rangle \mid M$ is a DFA and $L(M) = \phi\}$.

Note that $L(M) = \phi$ exactly if no recognizing vertex of M is reachable from its start vertex. It is easy to give a graph search algorithm to test this.

Claim 5.1.9. Empty-DFA *is decidable.*

Proof. The following algorithm $\mathcal{A}_{\text{Empty-DFA}}$ decides Empty-DFA. Given input $\langle M\rangle$, it determines the collection of vertices reachable from M's start vertex. If this collection includes a recognizing vertex then the algorithm outputs "Reject" and otherwise it outputs "Recognize". □

Example 5.1.10. Equal-DFA $= \{\langle M_A, M_B\rangle \mid M_A$ and M_B are DFAs and $L(M_A) = L(M_B)\}$.

Claim 5.1.11. Equal-DFA *is decidable.*

Proof. Let $A = L(M_A)$ and $B = L(M_B)$. We begin by observing that there is a DFA $\widetilde{M}_{AB}$ such that $L(\widetilde{M}_{AB}) = \phi$ exactly if $A = B$. For let $C = (A \cap \overline{B}) \cup (\overline{A} \cap B)$. Clearly, if $A = B$, $C = \phi$. While if $C = \phi$, $A \cap \overline{B} = \phi$, so $A \subseteq \overline{\overline{B}} = B$; similarly $\overline{A} \cap B = \Phi$, so $B \subseteq A$; together, these imply $A = B$.

But given DFAs M_A and M_B, we can construct DFAs $\overline{M}_A$ and $\overline{M}_B$ to recognize $\overline{A}$ and $\overline{B}$ respectively. Then using M_A and $\overline{M}_B$ we can construct DFA $M_{A\overline{B}}$ to recognize $A \cap \overline{B}$, and can similarly construct $M_{\overline{A}B}$ to recognize $\overline{A} \cap B$. Given $M_{A\overline{B}}, M_{\overline{A}B}$ we can construct $\widetilde{M}_{AB}$ to recognize $(A \cap \overline{B}) \cup (\overline{A} \cap B)$.

So the algorithm $\mathcal{A}_{\text{Equal-DFA}}$ to decide Equal-DFA, given input $\langle M_A, M_B\rangle$, constructs $\widetilde{M}_{AB}$ and forms the encoding $\langle \widetilde{M}_{AB}\rangle$. $\mathcal{A}_{\text{Equal-DFA}}$ then simulates program $P_{\text{Empty-DFA}}$ from the preceding example on input $\langle \widetilde{M}_{AB}\rangle$. $\mathcal{A}_{\text{Equal-DFA}}$ outputs the result of the simulation of $P_{\text{Empty-DFA}}$ on input $\langle \widetilde{M}_{AB}\rangle$.

This is correct for $P_{\text{Empty-DFA}}$ outputs "Recognize" exactly if $L(\widetilde{M}_{AB}) = \phi$ which is the case exactly if $A = B$. □

We have now given two examples of a use of a subroutine in a very particular form. More specifically, program P has used program Q as a subroutine and then used the answer of Q to compute its own output. In these two examples, the calculation of P's output has been the simplest possible: the output of Q has become the output of P.

We call this form of algorithm design a *reduction*. If we have a program (or algorithm) Q that decides a language A (e.g. Empty-DFA), and we give a program to decide language B (e.g. Equal-DFA) using Q as a subroutine then we say we have *reduced* language B to language A. What this means is that if we know how to decide language A we have now also demonstrated how to decide language B.

Example 5.1.12. Inf-DFA $= \{\langle M\rangle \mid M$ *is a DFA and* $L(M)$ *is infinite*$\}$.

Claim 5.1.13. Inf-DFA *is decidable.*

Proof. Note that $L(M)$ is infinite exactly if there is a path which includes a cycle and which goes from M's start vertex to some recognizing vertex. This property is readily tested by the following algorithm $\mathcal{A}_{\text{Inf-DFA}}$ shown below. □

Algorithm $\mathcal{A}_{\text{Inf-DFA}}$

Step 1. $\mathcal{A}_{\text{Inf-DFA}}$ identifies the non-trivial strong components[a] of M's graph, that is those that contain at least one edge (so any vertices with a self-loop will be in a non-trivial strong component).
Step 2. $\mathcal{A}_{\text{Inf-DFA}}$ forms the reduced graph, in which every strong component is replaced by a single vertex, and in addition it marks each non-trivial strong component (or rather the corresponding vertices). Further, there is an edge in the reduced graph from component C to component C' exactly if in the original graph there is an edge (u, v) where $u \in C$ and $v \in C'$.
Step 3. $\mathcal{A}_{\text{Inf-DFA}}$ checks whether any path from the start vertex to a recognizing vertex includes a marked vertex (and thus can contain a cycle drawn from the corresponding strong component).
Step 3.1. By means of any graph traversal procedure (e.g. DFS, BFS), $\mathcal{A}_{\text{Inf-DFA}}$ determines which marked vertices are reachable from the start vertex. It doubly marks these vertices.
Step 3.2. By means of a second traversal, $\mathcal{A}_{\text{Inf-DFA}}$ determines whether any recognizing vertices can be reached from the doubly marked vertices. If so, there is a path with a cycle from the start vertex to a recognizing vertex in M's graph, and then $\mathcal{A}_{\text{Inf-DFA}}$ outputs "Recognize"; otherwise $\mathcal{A}_{\text{Inf-DFA}}$ outputs "Reject."

[a]Recall that a strong component is a maximal set C of vertices, such that for every pair of vertices $u, v \in C$, there are directed paths from u to v and from v to u.

Example 5.1.14. All-DFA $= \{\langle M\rangle \mid L(M) = \Sigma^*$ *where* Σ *is* M*'s input alphabet*$\}$.

$L(M) = \Sigma^*$ exactly if $\overline{L(M)} = \phi$. So to test if $L(M) = \Sigma^*$, the decision algorithm $\mathcal{A}_{\text{All-DFA}}$ simply constructs the encoding $\langle\overline{M}\rangle$ and then uses the program $P_{\text{Empty-DFA}}$ to test if $L(\overline{M}) = \phi$, where $\overline{M}$ is the DFA recognizing $\overline{L}$, and then outputs the same answer, namely:

$$\mathcal{A}_{\text{All-DFA}}(\langle M\rangle) = \begin{cases} \text{"Recognize"} & \text{if } P_{\text{Empty-DFA}}(\langle\overline{M}\rangle) = \text{"Recognize"} \\ \text{"Reject"} & \text{if } P_{\text{Empty-DFA}}(\langle\overline{M}\rangle) = \text{"Reject"} \end{cases}$$

5.1.3 Decidability of Context Free Language Properties

Chomsky Normal Form Grammars

For the algorithms that follow it will be convenient to use Chomsky Normal Form (CNF) grammars for the CFLs. Recall that CNF grammars observe the following restrictions.

1. The only rule, if any, with λ on the RHS is $S \rightarrow \lambda$.
2. S appears only on the LHS of rules.
3. Every rule, except for the one in (1) if present, has one of two forms: $A \rightarrow b$ of $A \rightarrow BC$.

Lemma 5.1.15. *Every Context Free Language has a CNF grammar.*

Proof. To establish this result we start with a simplified grammar and then enforce the CNF properties one by one.

We establish Property 2 by introducing a new variable S' which replaces S on the RHS of all rules in which S appears, and for each rule $S \rightarrow \sigma$, we add the rule $S' \rightarrow \sigma$.

To establish Property 1 we add new rules which allow us to shortcut the use of rules of the form $A \rightarrow \lambda$, as follows. If there is a rule $A \rightarrow \lambda$, then for each rule $B \rightarrow A$ or $B \rightarrow AA$ we add the rule $B \rightarrow \lambda$, and for each rule $B \rightarrow AC$ or $B \rightarrow CA$ we add the rule $B \rightarrow C$ (which we also do even when $C = A$). If originally a derivation used the two-step sequence $B \Rightarrow AC \Rightarrow C$, we can replace these two steps with the single step $B \Rightarrow C$, and similarly for any other two-step sequence ending with the use of rule $A \rightarrow \lambda$. Each of these changes shortens the derivation by one step. We repeatedly add rules in this manner until no additional rules can be added. Since the number of possible rules of the above types is at most $|V|^3 + |V|^2 + |V|$, where $|V|$ is the number of variables in the grammar, this is clearly a finite process. Once all these rules have been included, any shortest derivation comprises either the single step $S \rightarrow \lambda$ (if λ is in the language) or it does not use any rule of the form $A \rightarrow \lambda$, since the use of all such rules can be shortcut. Thus at this point we can safely remove all rules of the form $A \rightarrow \lambda$.

Property 3 is established similarly. If a shortest derivation uses a rule $B \rightarrow A$, we will shortcut this step as follows. If there is a rule $B \rightarrow A$, then for each rule $C \rightarrow B$ we add the rule $C \rightarrow A$, for each rule $D \rightarrow BC$ we add the rule $D \rightarrow AC$, and for each rule $D \rightarrow CB$ we add the rule $D \rightarrow CA$. If originally a derivation used the two-step sequence $D \Rightarrow BC \Rightarrow AC$, we can replace these two steps with the single step $D \Rightarrow AC$, and similarly for any other two-step sequence ending with the use of rule $B \rightarrow A$. Each of these changes shortens the derivation by one step. We repeatedly add rules in this manner until no additional rules can be added. As before this is clearly a finite process. Once all these rules have been included, any shortest derivation does not use any rule of the form $B \rightarrow A$, since all such rules can be shortcut. Thus at this point we can safely remove all rules of the form $B \rightarrow A$. □

Decidable Properties

Next, we will demonstrate some decidable properties of Context Free languages.

Example 5.1.16. Rec-CFG $= \{\langle G, w\rangle \mid G$ *is a CFG which can generate* $w\}$.

Claim 5.1.17. Rec-CFG *is decidable.*

Proof. The algorithm $\mathcal{A}_{\text{Rec-CFG}}$ shown below decides Rec-CFG. □

Algorithm $\mathcal{A}_{\textbf{Rec-CFG}}$

Step 1. $\mathcal{A}_{\text{Rec-CFG}}$ converts G to a CNF grammar $\widetilde{G}$, with start symbol S.
Step 2. If $w = \lambda$, $\mathcal{A}_{\text{Rec-CFG}}$ checks if $S \to \lambda$ is a rule of $\widetilde{G}$ and if so outputs "Recognize" and otherwise outputs "Reject."
Step 3. If $w \neq \lambda$, the derivation of w in $\widetilde{G}$, if there is one, would take $2|w| - 1$ steps. $\mathcal{A}_{\text{Rec-CFG}}$ simply generates, one by one, all possible derivations in $\widetilde{G}$ of length $2|w| - 1$. If any of them yield w then it outputs "Recognize" and otherwise it outputs "Reject." (This is not intended to be an efficient algorithm.)

Example 5.1.18. Empty-CFL $= \{\langle G\rangle \mid G$ *is a CFG and* $L(G) = \phi\}$.

Claim 5.1.19. Empty-CFL *is decidable.*

Proof. Note that $L(G) \neq \phi$ if and only if G's start variable can generate a string in T^*, where T is G's terminal alphabet. We simply determine this property for each variable A in G: can A generate a string in T^*? This can be done by means of the following algorithm $\mathcal{A}_{\text{Empty-CFL}}$, which marks each such variable.

Step 1. $\mathcal{A}_{\text{Empty-CFL}}$ converts the grammar to CNF form (this just simplifies the rest of the description).
Step 2. $\mathcal{A}_{\text{Empty-CFL}}$ marks each variable A for which there is a rule $A \to a$ or $A \to \lambda$ (the latter could apply only to S, the start variable).
Step 3. Iteratively, $\mathcal{A}_{\text{Empty-CFL}}$ marks each variable A such that there is a rule $A \to BC$ and B and C are already marked. (We leave an efficient implementation to the reader). $\mathcal{A}_{\text{Empty-CFL}}$ stops when no more variables can be marked.
Step 4. $\mathcal{A}_{\text{Empty-CFL}}$ outputs "Reject" if S is marked and "Recognize" otherwise.

□

Next, we describe a more efficient algorithm $\mathcal{A}_{\text{Eff-Rec-CFL}}$, for determining if a CNF grammar G can generate a string w. It runs in time $O(mn^3)$, where m is the number of rules in G and $n = |w|$.

First, we introduce a little notation. Let $w = w_1 w_2 \cdots w_n$, where each $w_i \in T$, the terminal alphabet, for $1 \leq i \leq n$. w_i^l denotes the length l substring of w beginning at w_i: $w_i^l = w_i w_{i+1} \cdots w_{i+l-1}$.

Algorithm $\mathcal{A}_{\textbf{Eff-Rec-CFL}}$

It uses dynamic programming. Specifically, in turn, for $l = 1, 2, \cdots, n$, it determines, for each variable A, whether A can generate w_i^l, for each possible value of i, i.e. for $1 \le i \le n - l + 1$. This information suffices, for G can generate w exactly if $S \Rightarrow^* w_1^n$, when S is G's start variable.
For $l = 1$, the test amounts to asking whether $A \to w_i$ is a rule.
For $l > 1$, the test amounts to the following question:

> Is there a rule $A \to BC$, and a length k, with $1 \le k < l$, such that B generates the length k substring of w beginning at w_i and such that C generates the remainder of w_i^l (i.e. $B \Rightarrow^* w_i^k$ and $C \Rightarrow^* w_{i+k}^{l-k}$). Note that the results of the tests involving B and C have have already been computed, so for a single rule and a single value of k, this test runs in $O(1)$ time.

Summing the running times over all possible values of i, k, l, and all m rules yields the overall running time of $O(mn^3)$. This shows that:

Lemma 5.1.20. *The decision procedure for language* Rec-CFL *runs in time $O(mn^3)$ on input $\langle G, w\rangle$, where $n = |w|$ and m is the number of rules in G.*

Algorithm $\mathcal{A}_{\textbf{Inf-CFL}}$

Step 1. This step identifies *useful* variables, variables that can generate non-empty strings of terminals.
This can be done using a marking procedure, First, $\mathcal{A}_{\text{Inf-CFL}}$ marks the variables A for which there is a rule of the form $A \to a$. Then, iteratively, for each rule $A \to BC$, if both B and C are marked, it also marks A, continuing until no additional variables can be marked. The marked variables are exactly the useful variables.
If the start variable S is not useful, the algorithms stops, answering "Reject". Otherwise, it continues with Step 2.
Step 2. Let U be the set of G's useful variables. $\mathcal{A}_{\text{Inf-CFL}}$ now identifies the *reachable useful* variables, i.e. those useful variables for which there is a derivation $S \Rightarrow^* \sigma A \tau$, where $\sigma, \tau \in U^*$. This is done via the following marking process.
Step 2.0. $\mathcal{A}_{\text{Inf-CFL}}$ removes all rules that contain one or more useless (non-useful) variables.
Step 2.1. $\mathcal{A}_{\text{Inf-CFL}}$ marks S.
Step 2.2. For each unprocessed marked variable A, $\mathcal{A}_{\text{Inf-CFL}}$ marks all variables on the RHS of a rule with A on the LHS.
When this process terminates, the marked variables are exactly the reachable useful variables.
Step 3. Finally, $\mathcal{A}_{\text{Inf-CFL}}$ identifies the repeating, reachable useful variables, namely the variables that can repeat on a derivation tree path.
To do this, $\mathcal{A}_{\text{Inf-CFL}}$ uses a procedure analogous to the one used in Step 2: For each reachable useful variable A, $\mathcal{A}_{\text{Inf-CFL}}$ determines the variables reachable from A; if this collection includes A, then A is repeating. (Note that Step 2 found the useful variables reachable from S.)
$\mathcal{A}_{\text{Inf-CFL}}$ answers "Recognize" exactly if some variable is repeating.

Example 5.1.21. Inf-CFL $= \{G \mid G$ *is a CNF grammar and* $L(G)$ *is infinite*$\}$.

Claim 5.1.22. Inf-CFL *is decidable.*

Proof. Note that $L(G)$ is infinite exactly if there is a path in a derivation tree with a repeated variable. The following algorithm, $\mathcal{A}_{\text{Inf-CFL}}$ identifies the variables that can be repeated in this way; $L(G)$ is infinite exactly if there is at least one such variable. $\mathcal{A}_{\text{Inf-CFL}}$ proceeds in several steps, as shown above. □

5.2 Undecidability

The *Barber of Seville* is a classic puzzle. The barber of Seville is said to shave all men who do not shave themselves. So who shaves the barber of Seville? To make this into a puzzle the words have to be treated unduly logically. In particular, one has to interpret it to mean that anyone shaved by the barber of Seville does not shave himself. Then if the barber of Seville shaves himself it is because he does not shave himself and in turn this is because he does shave himself.

One way out of this conundrum occurs if the Barber of Seville is a woman. But our purpose here is to look at how to set up this type of conundrum, or contradiction.

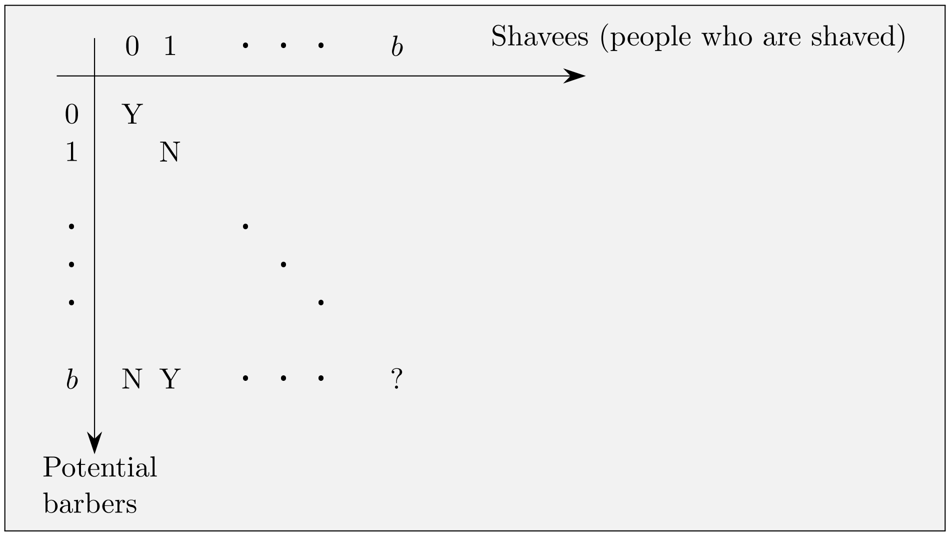


Figure 5.1: Who Shaves the Barber of Seville?

Let us form a table, on one side listing people in their role as people who are shaved, on the other as potential shavers (or barbers). For simplicity, we just name the people 1, 2, $\cdots$. So the entries in row i show who is shaved by person i, with entry (i, j) being Y ("yes") if person i shaves person j and N ("no") otherwise. Let row b be the row for the barber of Seville. Then, for every j, entries (b, j) and (j, j) are opposite (one Y and one N). This leads to a contradiction for entry (b, b) cannot be opposite to itself. See Figure 5.1.

Now we are ready to show that the halting problem is undecidable by means of a similar argument. We define the halting problem function H as follows.

$$H(P, w) = \begin{cases} \text{"Recognize"} & \text{if program } P \text{ halts on input } w \\ \text{"Reject"} & \text{if program } P \text{ does not halt on input } w \end{cases}$$

By halt, we mean that a program completes its computation and stops.

Recall that the program P and its input are encoded as binary strings. Let i denote the ith such string in lexicographic order, namely the order 0, 1, 00, 01, 10, 11, 000, $\cdots$. Let P_i denote the ith string when it is representing a program, and w_i denote it when it is representing an input. When considering the pair (P_i, w_j), if P_i is not an encoding of a legal program, or if w_j is not an encoding of a legal input to P_i, then the output of program P_i on input w_j is deemed to be "Halting".

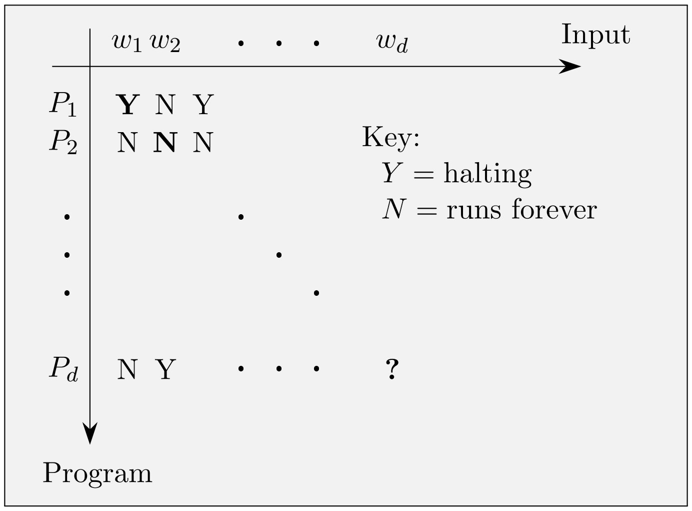


Figure 5.2: Function D.

The output of program H is illustrated in Figure 5.2. By analogy with the Barber of Seville, our aim is to create a new program which has output the opposite of the entries on the diagonal of the output table for H. Program P_d in Figure 5.2, if it exists, provides this opposite.

More precisely, we define D as follows.

$$D(P_i) = \begin{cases} \text{output "Done"} & \text{if } P_i \text{ runs forever on input } w_i \\ \text{loop forever} & \text{if } P_i \text{ on input } w_i \text{ eventually halts} \end{cases}$$

If H is computable then clearly there is a program P_d to compute D. But then, as with the Barber of Seville, $P_d(w_d)$ is not well defined, for:

$$P_d(w_d) = \begin{cases} \text{output "Done"} & \text{if } P_d \text{ runs forever on input } w_d \\ \text{loop forever} & \text{if } P_d \text{ on input } w_d \text{ halts} \end{cases}$$

But this is a contradiction. We have shown:

Lemma 5.2.1. *There is no program to compute the function D.*

We have also shown:

Theorem 5.2.2. *There is no (2-input) program that computes H, the halting function.*

The technique we have just used is called *diagonalization*. It was first developed to show that the real numbers are not countable. We will show this result next.

Definition 5.2.3. *A set A is* listable *or* countable *if it is finite or if there is a function f such that $A = \{f(1), f(2), \cdots\}$.*

In other words, A can be listed by the function f ($f(1) = a_1$ is the first item in A, $f(2) = a_2$ is the second item, and so on).

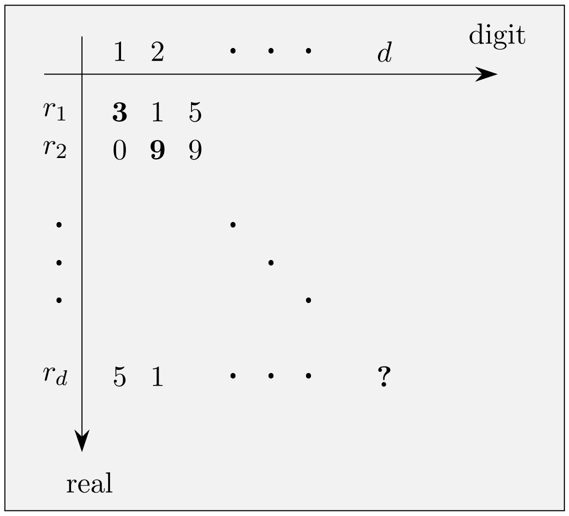


Figure 5.3: d is not in the listing of reals.

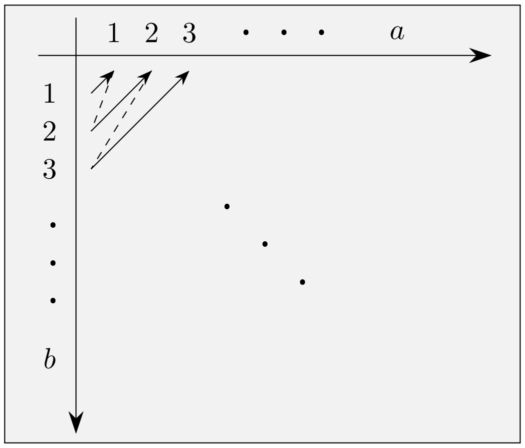


Figure 5.4: Listing the Rationals.

If this listing can be carried out by a program, A is said to be *effectively* or *recursively enumerable*. As we will see later, the halting set $H = \{\langle P, w\rangle \mid P \text{ halts on input } w\}$ is recursively enumerable.

Lemma 5.2.4. *The real numbers are not countable.*

Proof. We will show that the real numbers in the range $[0, 1]$ are not countable. This suffices to show the result, for given a listing of all the real numbers, one could go through the list, forming a new list of the reals in the range $[0, 1]$.

For a contradiction, suppose that there were a listing of the real numbers in the range $[0, 1]$, $r_1, r_2, \cdots$, say. Imagine that each real number is provided as an infinite decimal: $r_i = 0.r_{i1}r_{i2}\cdots$, when each r_{ij} is a digit ($r_{ij} \in \{0, \cdots, 9\}$). Note that $1 = 0.999\cdots$. This could mean that each real r is presented using a program P_r that on input i generates the ith digit of r.

We create a new decimal, $d = d_1d_2\cdots$, such that $d \in [0, 1]$ yet $d \neq r_i$ for all i; so the listing of reals in the range $[0, 1]$ would not include d; but it must do so as it is a listing of all the reals in this range. This is a contradiction, which shows that the reals are not countable.

It remains to define d, which we do as follows:

$$d_i = r_{ii} + 2 \bmod 10 \quad \text{for all } i.$$

Suppose that $d = r_j$ for some j. As $d_j \neq r_{jj}$, the only way d and r_j could be equal is if one of them had the form $0.sx99\cdots$ and the other had the form $0.s(x + 1)00\cdots$, where x is a single digit and s is a string of digits. But as the shift on the jth digit is by 2 this is not possible.

Again, this is a construction by diagonalization, with the fact that $0.99\cdots = 1.00\cdots$ creating a small complication. The construction of d is illustrated in Figure 5.3. □

By contrast the rationals are countable. By a rational we mean a ratio a/b where a and b are positive integers, but not necessarily coprime.

So for the purposes of listing the rationals we will view a/b and $2a/2b$ as distinct rationals (it is a simple exercise to modify the listing function to eliminate such duplicates).

Lemma 5.2.5. *The rationals are countable.*

Proof. The rationals can be displayed in a 2-D table, as shown in Figure 5.4. The rows and columns are indexed by the positive integers, and entry (a, b) represents rational a/b. So the task reduces to listing the table entries, which is done by going through the forward diagonals, one by one, in the order of increasing $a + b$. That is, first the entry with $a + b$ value 2 is listed, namely the entry (1,1); then the entries with $a + b$ value 3 are listed, namely the entries (2,1), (1,2); next the entries with $a + b$ value 4 are listed, namely the entries (3,1), (2,2), (1,3); and so forth. Within a diagonal, the entries are listed in the order given by increasing the second coordinate.

Clearly every rational is listed eventually. ((a, b) will be the $(a + b - 1)(a + b - 2)/2 + a$th item listed, in fact.) Also every rational is listed exactly once. Thus the rationals are countable. □

The same idea can be used to list the items in a d-dimensional table where each coordinate is indexed by the positive integers. Let the coordinate names be x_1, x_2, $\cdots$, x_d, respectively. Then, in turn, the entries with $x_1 + x_2 + \cdots + x_d = k$, for $k = d, d + 1, d + 2, \cdots$ are listed. For a given value of k, in turn, the entries with $x_d = 1, 2, \cdots, k - d + 1$ are listed recursively.

So for $d = 3$ the listing begins (1,1,1), (2,1,1), (1,2,1), (1,1,2), (3,1,1), (2,2,1), (1,3,1), (2,1,2), etc. We call this the *diagonal listing*, and we use it to show that H is recursively enumerable.

Lemma 5.2.6. *H is recursively enumerable.*

Proof. The listing program explores a 3-dimensional table in the diagonal listing order. At the (i, j, k)th table entry, it simulates program P_i on input w_j for k steps. If $P_i(w_j)$ runs to completion in exactly k steps then the listing program outputs the encoding $\langle i, j\rangle$ of the pair (i, j).

Clearly, if P_i halts on input w_j, then $\langle i, j\rangle$ occurs in this listing, and further it occurs exactly once. As this listing is produced by a program it follows that H, the set listed, is recursively enumerable. □

Next, we relate recursive enumerability and decidability.

Lemma 5.2.7. *If L and $\overline{L}$ are both recursively enumerable then L is decidable.*

Proof. It suffices to give an algorithm $\mathcal{A}_L$ to decide L. $\mathcal{A}_L$ will use the listing procedures for L and $\overline{L}$. Let $\text{List}_L(x)$ be the program that on input x returns the xth item in a listing of L, and let $\text{List}_{\overline{L}}(x)$ be the analogous program with respect to $\overline{L}$. Then, on input w, $\mathcal{A}_L$ simply runs $\text{List}_L(x)$ and $\text{List}_{\overline{L}}(x)$ for increasing values of x ($x = 1, 2, \cdots$ in turn) until one of them returns the value w, thereby showing in which set w occurs. At this point, $\mathcal{A}_L$ outputs "Recognize" or "Reject", as appropriate. In more detail, $\mathcal{A}_L$ is the following program.

```
$\mathcal{A}_L(w)$:
    found $\leftarrow$ FALSE; $x \leftarrow 1$
    while (not found) do
        $w_L \leftarrow \text{List}_L(x)$
        $w_{\overline{L}} \leftarrow \text{List}_{\overline{L}}(x)$
        if $w = w_L$ or $w = w_{\overline{L}}$
            then found $\leftarrow$ TRUE
            else $x \leftarrow x + 1$
        end while
    if $w = w_L$ then return("Recognize")
        else return("Reject")
```

Note that w occurs in one of the lists L or $\overline{L}$. Consequently, w must be listed eventually, and therefore the loop in the above program also terminates eventually (when it reaches the item w in the list containing w). ☐

We can now show that $\overline{H}$ is not recursively enumerable.

Lemma 5.2.8. *$\overline{H}$ is not recursively enumerable.*

Proof. Recall that H is recursively enumerable (by Lemma 5.2.6). Were $\overline{H}$ also recursively enumerable, then, by Lemma 5.2.7, H would be decidable, which is not the case (by Theorem 5.2.2). This shows that $\overline{H}$ cannot be recursively enumerable. (Strictly, this was a proof by contradiction.) ☐

Notice that membership and non-membership are not symmetric in a recursively enumerable but non-decidable set such as H. While membership can be demonstrated simply by listing the set and encountering the item, there is no test for non-membership.

5.2.1 Undecidability via Reductions

We turn to showing undecidability via reductions. The form of the argument will be as follows.

Suppose that we are given an algorithm (or program) $\mathcal{A}_L$ which is claimed to decide set L. Suppose that using $\mathcal{A}_L$ as a subroutine, we create another algorithm $\mathcal{A}_J$ to decide set J. But suppose that we already know set J to be undecidable, for example if $J = H$, the halting set. We can then conclude that $\mathcal{A}_L$ does not decide set L, and in fact that there is no algorithm to decide set L. The latter claim follows by a proof by contradiction: assume that there is such an algorithm, and call it $\mathcal{A}_L$; then the previous argument shows that $\mathcal{A}_L$ does not decide L, a contradiction.

Example 5.2.9. $W = \{\langle Q \rangle \mid Q \text{ on input string 0 eventually halts}\}$.

Lemma 5.2.10. *If there is an algorithm $\mathcal{A}_W$ to decide W, then there is also an algorithm to decide H, the halting set.*

Proof. Here is the algorithm $\mathcal{A}_H$ deciding H. It will use $\mathcal{A}_W$ as a subroutine.

On input $\langle P, w\rangle$:

$\mathcal{A}_H$ will build (compute the encoding of) a one-input program $R_{P,w}$[a] which it then inputs to the algorithm $\mathcal{A}_W$. $\mathcal{A}_H$ concludes by reporting the result of running $\mathcal{A}_W$ on input $\langle R_{P,w}\rangle$.

[a]The notation $R_{P,w}$ is meant to indicate that program R is determined in part by P and w. (Thus P might be a subroutine in R, and w might be the initial value of one of R's variables. w is not an input to R. x denotes R's input.)

This means that we want $R_{P,w}$ to behave as follows:

$$\mathcal{A}_H(\langle P, w\rangle) = \mathcal{A}_W(\langle R_{P,w}\rangle) = \begin{cases} \text{“Recognize”} & \text{if } P \text{ eventually halts on input } w \\ \text{“Reject”} & \text{if } P \text{ runs forever on input } w \end{cases}$$

Now by the definition of W:

$$\mathcal{A}_W(\langle R_{P,w}\rangle) = \begin{cases} \text{“Recognize”} & \text{if } R_{P,w}(0) \text{ halts} \\ \text{“Reject”} & \text{if } R_{P,w}(0) \text{ runs forever} \end{cases}$$

This means that we want:

$$\begin{array}{ll} R_{P,w}(0) \text{ halts} & \text{if } P \text{ eventually halts on input } w \\ R_{P,w}(0) \text{ runs forever} & \text{if } P \text{ runs forever on input } w. \end{array}$$

Here is a program $R_{P,w}$ that meets the above requirement:

```
$R_{P,w}(x)$:
    run $P$ on input $w$ (note that $R_{P,w}$ ignores its input).
```

Clearly, for any x, $R_{P,w}(x)$ halts if P eventually halts on input w, and $R_{P,w}(x)$ runs forever if P runs forever on input w, and hence it does so for $x = 0$ in particular.

Thus our algorithm for deciding H proceeds as follows:

```
On input $\langle P, w\rangle$:
    Step 1. $\mathcal{A}_H$ computes $\langle R_{P,w}\rangle$, the encoding of program $R_{P,w}$.
    Step 2. $\mathcal{A}_H$ simulates algorithm $\mathcal{A}_W$ on input $\langle R_{P,w}\rangle$ and outputs the result of $\mathcal{A}_W(\langle R_{P,w}\rangle)$.
```

$\square$

Corollary 5.2.11. *W is undecidable, that is there is no algorithm to decide W.*

Example 5.2.12. Let $X = \{\langle Q\rangle \mid$ either Q on input 0 eventually halts, or Q on input 1 eventually halts, or both$\}$.

Lemma 5.2.13. *If there is an algorithm $\mathcal{A}_X$ to decide X, then there is also an algorithm $\mathcal{A}_H$ to decide H.*

Proof. This proof is very similar to the previous one. Again, $\mathcal{A}_H$ will build a program $R'_{P,w}$ with the following characteristics:

$$\mathcal{A}_X(\langle R'_{P,w}\rangle) = \begin{cases} \text{“Recognize”} & \text{if } P \text{ eventually halts on input } w \\ \text{“Reject”} & \text{if } P \text{ runs forever on input } w \end{cases}$$

$\mathcal{A}_H$ then simulates $\mathcal{A}_X(\langle R'_{P,w}\rangle)$, reporting the result of this simulation as its output.

This means that we want:

at least one of $R'_{P,w}(\langle 0\rangle)$ and $R'_{P,w}(\langle 1\rangle)$ halts	if P eventually halts on input w
both $R'_{P,w}(\langle 0\rangle)$ and $R'_{P,w}(\langle 1\rangle)$ run forever	if P runs forever on input w

It is easy to check that $R'_{P,w} = R_{P,w}$ meets this requirement. Thus we can use the following algorithm $\mathcal{A}_H$ to decide H.

```
On input $\langle P, w\rangle$:
    Step 1. $\mathcal{A}_H$ constructs the encoding $\langle R_{P,w}\rangle$.
    Step 2. $\mathcal{A}_H$ simulates $\mathcal{A}_X(\langle R_{P,w}\rangle)$ and give its result as the output of $\mathcal{A}_H$.
```

□

Corollary 5.2.14. *X is undecidable.*

We now give another proof of the result that W is undecidable using the fact that X is undecidable. This shows that reductions can be to languages other than H, and also this particular argument introduces a somewhat different construction.

Lemma 5.2.15. *If there is an algorithm $\mathcal{A}_W$ to decide W, then there is also an algorithm $\mathcal{A}_X$ to decide X.*

Proof. For this result, we create an algorithm to decide X. Given an input Q, $\mathcal{A}_X$ will build a program R_Q with the following characteristics:

$$\mathcal{A}_W(\langle R_Q\rangle) = \begin{cases} \text{“Recognize”} & \text{if } Q \text{ eventually halts on either input 0 or input 1, or both} \\ \text{“Reject”} & \text{if } Q \text{ runs forever on both input 0 and input 1} \end{cases}$$

$\mathcal{A}_X$ then simulates $\mathcal{A}_W(\langle R_Q\rangle)$, reporting the result of this simulation as its output.

This means that we want:

$R_Q(\langle 0\rangle)$ halts	if Q eventually halts on at least one of inputs 0 and 1
$R_Q(\langle 0\rangle)$ runs forever	if Q runs forever on both inputs 0 and 1

Consider the following procedure R_Q.

```
$R_Q(x)$:
    interleave runs of $Q$ on input 0 and on input 1 (i.e. for $t = 1, 2, \ldots$,
    run the $t$-th step of $Q(0)$ and then of $Q(1)$), and halt as soon as either
    $Q(0)$ or $Q(1)$ halts
```

Clearly, $R_Q(0)$ eventually halts if either $Q(0)$ or $Q(1)$ halts, and otherwise it runs forever, thereby meeting the requirements.

Thus we can use the following algorithm $\mathcal{A}_X$ to decide X.

```
On input $\langle Q \rangle$:
    **Step 1**. $\mathcal{A}_X$ constructs the encoding $\langle R_Q \rangle$.
    **Step 2**. $\mathcal{A}_X$ simulates $\mathcal{A}_W(\langle R_Q \rangle)$ and give its result as the output of $\mathcal{A}_W$.
```

□

Example 5.2.16. $Y = \{\langle Q \rangle \mid$ there is a variable v in Q that is never assigned a value when Q is run on input $1\}$.

Lemma 5.2.17. *If there is an algorithm $\mathcal{A}_Y$ to decide Y, then there is also an algorithm $\mathcal{A}_H$ to decide H.*

Proof. Here is the algorithm $\mathcal{A}_H$ to decide H.

```
On input $\langle P, w \rangle$:
    $\mathcal{A}_H$ will compute the encoding of a program $R''_{P,w}$ which it then inputs to $\mathcal{A}_Y$.
```

What we want is for:

$$\mathcal{A}_H(\langle P, w \rangle) = \mathcal{A}_Y(\langle R''_{P,w} \rangle) = \begin{cases} \text{“Recognize”} & \text{if } P \text{ eventually halts on input } w \\ \text{“Reject”} & \text{if } P \text{ runs forever on input } w \end{cases}$$

This means that we want:

- If P eventually halts on input w then, when $R''_{P,w}$ is run on input 1, $R''_{P,w}$ has a variable that is never assigned a value, and
- If P runs forever on input w then, when $R''_{P,w}$ is run on input 1, every variable of $R''_{P,w}$ is assigned a value.

Let's try the following program for $R''_{P,w}$:

```
On input $x$:
    **Step 1**. $R''_{P,w}$ simulates $P(w)$.
    **Step 2**. For every variable $z$, that appears in $P$ or the simulation environment, do:
    $z \leftarrow 0$.
    **Step 3**. $v \leftarrow 1$, where $v$ is a variable that does not appear in $P$ or the simulation
    environment.
```

When $R''_{P,w}$ is run on input x and on $x = 1$ in particular, if P halts on input w, then every variable appearing in $R''_{P,w}$ is assigned a value, while if P runs forever on input w, at the very least, variable v in $R''_{P,w}$ is not assigned a value.

Oops, this is back to front. Unfortunately, this is unavoidable. So let's change our goal for $\mathcal{A}_Y$. Let's require:

$$\begin{aligned}\mathcal{A}_Y(\langle R''_{P,w}\rangle) &= \begin{cases}\text{“Reject”} & \text{if } P \text{ eventually halts on input } w\\ \text{“Recognize”} & \text{if } P \text{ runs forever on input } w\end{cases}\\ \text{so }\ \mathcal{A}_H(\langle P,w\rangle) &= \text{Opposite } (\mathcal{A}_Y(\langle R''_{P,w}\rangle)).\end{aligned}$$

This means that we want:

- If P eventually halts on input w then, when $R^{''}_{P,w}$ is run on input 1, every variable of $R^{''}_{P,w}$ is assigned a value.
- If P runs forever on input w then, when $R^{''}_{P,w}$ is run on input 1, $R^{''}_{P,w}$ has a variable that is never assigned a value.

But this is achieved by the above program $R^{''}_{P,w}$.

Now, our algorithm $\mathcal{A}_H$ for deciding H simply reports the opposite answer to $\mathcal{A}_Y(\langle R^{''}_{P,w}\rangle)$. So the algorithm is the following:

On input $\langle P,w\rangle$:
  Step 1. $\mathcal{A}_H$ constructs the encoding $\langle R^{''}_{P,w}\rangle$.
  Step 2. $\mathcal{A}_H$ simulates $\mathcal{A}_Y(\langle R^{''}_{P,w}\rangle)$.
  Step 3. $\mathcal{A}_H$ reports the opposite answer to that given by $\mathcal{A}_Y$ in Step 2.

□

Corollary 5.2.18. *Y is undecidable.*

Example 5.2.19. Never-Halt $= \{\langle Q\rangle \mid Q$ runs forever on every input$\}$.

Lemma 5.2.20. *If there is an algorithm $\mathcal{A}_{\text{NH}}$ to decide* Never-Halt*, then there is also an algorithm $\mathcal{A}_H$ to decide H.*

Proof. The algorithm $\mathcal{A}_H$ deciding H will use $\mathcal{A}_{\text{NH}}$ as a subroutine. On input $\langle P,w\rangle$, $\mathcal{A}_H$ will compute the encoding of a one-input program $R^3_{P,w}$ which it then inputs to the algorithm $\mathcal{A}_{\text{NH}}$. What we want is that:

$$\mathcal{A}_H(\langle P,w\rangle) = \mathcal{A}_{\text{NH}}(\langle R^3_{P,w}\rangle) = \begin{cases}\text{“Recognize”} & \text{if } P \text{ eventually halts on input } w\\ \text{“Reject”} & \text{if } P \text{ runs forever on input } w\end{cases}$$

Now by the definition of Never-Halt:

$$\mathcal{A}_{\text{NH}}(\langle R^3_{P,w}\rangle) = \begin{cases}\text{“Recognize”} & \text{if } R^3_{P,w} \text{ never halts}\\ \text{“Reject”} & \text{if } R^3_{P,w} \text{ halts on some input}\end{cases}$$

This means that we want:

$$\begin{array}{ll} R^3_{P,w} \text{ never halts} & \text{if } P \text{ eventually halts on input } w\\ R^3_{P,w} \text{ halts on some input} & \text{if } P \text{ runs forever on input } w.\end{array}$$

It does not seem possible to build such an R. Let's try switching the outputs given by $\mathcal{A}_{\text{NH}}$ to be:

$$\begin{aligned}\mathcal{A}_{\text{NH}}(\langle R^3_{P,w}\rangle) &= \begin{cases}\text{“Reject”} & \text{if } P \text{ eventually halts on input } w\\ \text{“Recognize”} & \text{if } P \text{ runs forever on input } w\end{cases}\\ \text{so}\ \ \mathcal{A}_H(\langle P,w\rangle) &= \text{Opposite } (\mathcal{A}_{\text{NH}}(\langle R^3_{P,w}\rangle)).\end{aligned}$$

And then we want:

$$\begin{array}{ll} R^3_{P,w} \text{ never halts} & \text{if } P \text{ runs forever on input } w\\ R^3_{P,w} \text{ halts on some input} & \text{if } P \text{ eventually halts on input } w.\end{array}$$

The program $R^3_{P,w} = R_{P,w}$ from Example 5.2.10 meets the above requirement.

Thus our algorithm for deciding H proceeds as follows:

On input $\langle P, w\rangle$:
Step 1. $\mathcal{A}_H$ computes $\langle R_{P,w}\rangle$, the encoding of program $R_{P,w}$.
Step 2. $\mathcal{A}_H$ simulates algorithm $\mathcal{A}_{\text{NH}}(\langle R_{P,w}\rangle)$ and outputs the opposite of its result.

□

Corollary 5.2.21. Never-Halt *is undecidable.*

Example 5.2.22. Equal-Prog $= \{\langle Q_1, Q_2\rangle \mid Q_1$ and Q_2 halt on exactly the same inputs$\}$. We write $Q_1 \equiv Q_2$ if $\langle Q_1, Q_2\rangle \in$ Equal-Prog and $Q_1 \not\equiv Q_2$ if $\langle Q_1, Q_2\rangle \notin$ Equal-Prog, for short.

Lemma 5.2.23. *If there is an algorithm $\mathcal{A}_{EP}$ to decide* Equal-Prog*, then there is also an algorithm $\mathcal{A}_{NH}$ to decide* Never-Halt.

Proof. The algorithm $\mathcal{A}_{\text{NH}}$ deciding Never-Halt will use $\mathcal{A}_{\text{EP}}$ as a subroutine. On input $\langle P\rangle$, $\mathcal{A}_{\text{NH}}$ will compute the encoding of programs R^4_P and R^5_P which it then inputs to the algorithm $\mathcal{A}_{\text{EP}}$. What we want is that:

$$\mathcal{A}_{\text{NH}}(\langle P\rangle) = \mathcal{A}_{\text{EP}}(\langle R^4_P, R^5_P\rangle) = \begin{cases}\text{“Recognize”} & \text{if } P \text{ runs forever on every input}\\ \text{“Reject”} & \text{if } P \text{ halts on some input}\end{cases}$$

Now by the definition of Equal-Prog:

$$\mathcal{A}_{\text{EP}}(\langle R^4_P, R^5_P\rangle) = \begin{cases}\text{“Recognize”} & \text{if } R^4_P \equiv R^5_P\\ \text{“Reject”} & \text{if } R^4_P \not\equiv R^5_P\end{cases}$$

This means that we want:

$$\begin{array}{ll} R^4_P \equiv R^5_P & \text{if } P \text{ runs forever on every input}\\ R^4_P \not\equiv R^5_P & \text{if } P \text{ halts on some input.}\end{array}$$

This suggests the following choice for R^4_P and R^5_P. Set $R^4_P = P$ and $R^5_P = N$, a program that never halts. Here is $N(x)$: loop forever. This pair does meet the above criteria. This yields the following algorithm $\mathcal{A}_{\text{NH}}$ for deciding *Never-Halt*.

$\mathcal{A}_{\text{NH}}$ simulates algorithm $\mathcal{A}_{\text{EP}}(\langle P, N\rangle)$ and outputs its result. (Note that $\langle P\rangle$ is the input to $\mathcal{A}_{\text{NH}}$, and $\langle N\rangle$ can be stored as the initial value of one of the variables of $\mathcal{A}_{\text{NH}}$.)

□

Corollary 5.2.24. Equal-Prog *is undecidable.*

Exercises

1. Show that L is computable if and only if both L and $\overline{L}$ are recognizable.

2. Let Rec-GNFA $= \{\langle M, w\rangle \mid M$ is a GNFA and $w \in L(M)\}$.
   Show that Rec-GNFA is decidable.

3. i. Show that $R \subseteq S$ if and only if $R \cap \overline{S} = \phi$.

   ii. Let Reg-Contain $= \{\langle M_R, M_S\rangle \mid M_R$ and M_S are DFAs recognizing regular languages R and S respectively, and $R \subseteq S\}$.
   Show that Reg-Contain is decidable.
   Hint: Use a reduction to Empty-DFA.

4. Let Eq-DFA-NFA $= \{\langle M, N\rangle \mid M$ is a DFA and N is an NFA with $L(M) = L(N)\}$.
   Show that Eq-DFA-NFA is decidable.

5. Let Eq-Rev $= \{\langle M\rangle \mid M$ is a DFA, $L = L(M)$, and $L = L^R\}$, where L^R denotes the language containing the reversals of the strings in L. Show that Eq-Rev is decidable. You may assume the result of Chapter 2, No. 9.

6. Let CFL-Int-a^* $= \{\langle G\rangle \mid G$ is a context free grammar, and $L(G)$, the language it generates, satisfies $L(G) \cap a^* \neq \phi\}$.
   Show that CFL-Int-a^* is decidable by means of a reduction to Empty-CFL.

7. $A = \{\langle G\rangle \mid G$ is a CNF grammar with start variable S, terminal alphabet $\{a, b\}$, and there is a string $x \in a^*$ such that $S \Rightarrow^* x\}$.
   That is, there is a string containing only a's in $L(G)$.
   Show that A is decidable by means of an algorithm that analyzes the rules in G.

8. Let Inf-PDA $= \{\langle M_L\rangle \mid M_L$ is a PDA recognizing language L, and L is infinite$\}$.
   Show that Inf-PDA is decidable.

9. Let CFL-Int-Reg = CFL-IR $= \{\langle G, M\rangle \mid G$ is a context free grammar, M is a DFA and $L(G) \cap L(M) \neq \phi\}$.
   Show that CFL-IR is decidable.
   Hint. Use a reduction to Empty-CFL.

10. Let $L(x)$ be a computable function that lists *always halting integer output programs*: a program is an always halting integer output program if for every possible input it eventually outputs an integer.

    So L is computed by a program, P_L say. Specifically, $L(1), L(2), \cdots = P_{j_1}, P_{j_2}, \cdots$ is a list of always halting integer output programs.

    Prove, by diagonalization, that there is an always halting integer output program Q not on the list generated by L.

11. Let A and B be recursively enumerable sets. Suppose that $\overline{A} \cap \overline{B} = \phi$. Show that there is a decidable set C such that $\overline{A} \subseteq C$ and $\overline{B} \subseteq \overline{C}$.
    Hint. Modify the algorithm for deciding membership in set L if both L and $\overline{L}$ are recursively

enumerable. (A and B will replace L and $\overline{L}$; you will need to define L suitably. This will involve three cases for your membership test: when $x \in A - B$, when $x \in B - A$, and when $x \in A \cap B$. Which case(s) apply when $x \in \overline{A}$?)

12. Let Halt-Exactly-Once = HEO = $\{\langle Q \rangle \mid Q$ halts on exactly one input$\}$.

    Suppose that you are given an algorithm $\mathcal{A}_{\text{HEO}}$ that decides HEO. Using $\mathcal{A}_{\text{HEO}}$ as a subroutine, give an algorithm $\mathcal{A}_H$ to decide H.

13. Let Mixed = $\{\langle Q \rangle \mid Q$ halts on input 1 and does not halt on input 2$\}$.

    Suppose that you are given an algorithm $\mathcal{A}_{\text{Mixed}}$ that decides Mixed. Using $\mathcal{A}_{\text{Mixed}}$ as a subroutine, give an algorithm $\mathcal{A}_H$ to decide H.

14. i. Let Dead-Code = DC = $\{\langle Q, l \rangle \mid$ line l of Q's code is not executed on any input to $Q\}$.
    Suppose that you are given an algorithm $\mathcal{A}_{\text{DC}}$ that decides DC. Using $\mathcal{A}_{\text{DC}}$ as a subroutine, give an algorithm $\mathcal{A}_H$ to decide H.

    *ii.

    Let Any-Dead-Code = ADC =
    $$\{\langle Q \rangle \mid Q \text{ contains a line of code that is not executed on any input to } Q\}.$$

    Suppose that you are given an algorithm $\mathcal{A}_{\text{ADC}}$ that decides ADC. Using $\mathcal{A}_{\text{ADC}}$ as a subroutine, give an algorithm $\mathcal{A}_H$ to decide H.
    Hint: In some circumstances algorithm $\mathcal{A}_H$ is going to have to be written in such a way that it executes every line of its code.

15. Let Useless-Var = UV = $\{\langle Q \rangle \mid Q$ contains a variable that remains unassigned whatever the input to $Q\}$.

    Suppose that you are given an algorithm $\mathcal{A}_{\text{UV}}$ that decides UV. Using $\mathcal{A}_{\text{UV}}$ as a subroutine, give an algorithm $\mathcal{A}_H$ to decide H.

16. Let Even-Length-Halt = ELH = $\{\langle Q \rangle \mid Q$ eventually halts on all input strings of even length$\}$.

    Suppose that you are given an algorithm $\mathcal{A}_{\text{ELH}}$ that decides ELH. Using $\mathcal{A}_{\text{ELH}}$ as a subroutine, give an algorithm $\mathcal{A}_H$ to decide H.

17. Let Equal-Prog = EQ = $\{\langle Q_1, Q_2 \rangle \mid Q_1$ and Q_2 halt on exactly the same inputs$\}$.

    Suppose that you are given an algorithm $\mathcal{A}_{\text{EQ}}$ that decides EQ. Using $\mathcal{A}_{\text{EQ}}$ as a subroutine, give an algorithm $\mathcal{A}_{\text{ELH}}$ to decide ELH. See Problem 16 for the definition of ELH.

18. Let Inf-Mixed = IM = $\{\langle Q \rangle \mid Q$ eventually halts on infinitely many inputs and fails to halt on infinitely many inputs$\}$.

    Suppose that you are given an algorithm $\mathcal{A}_{\text{IM}}$ that decides IM. Using $\mathcal{A}_{\text{IM}}$ as a subroutine, give an algorithm $\mathcal{A}_H$ to decide H.

19. Let Sometime-Halt = SH = $\{\langle Q \rangle \mid Q$ halts on at least one input$\}$.

    Suppose that you are given an algorithm $\mathcal{A}_{\text{SH}}$ that decides SH. Using $\mathcal{A}_{\text{SH}}$ as a subroutine, give an algorithm $\mathcal{A}_H$ to decide H.

20. Let Inf-Halt = IH = $\{\langle Q\rangle \mid Q$ halts on infinitely many inputs$\}$.

    Suppose that you are given an algorithm $\mathcal{A}_{\text{IH}}$ that decides IH. Using $\mathcal{A}_{\text{IH}}$ as a subroutine, give an algorithm $\mathcal{A}_H$ to decide H.

21. Let Inf-Not-Halt = INH = $\{\langle Q\rangle \mid Q$ fails to halt on infinitely many inputs$\}$.

    Suppose that you are given an algorithm $\mathcal{A}_{\text{INH}}$ that decides INH. Using $\mathcal{A}_{\text{INH}}$ as a subroutine, give an algorithm $\mathcal{A}_H$ to decide H.

22. Two programs P and Q are *functionally equivalent* if for every possible input x, $P(x) = Q(x)$, where this is taken to mean that if $P(x)$ does not halt then $Q(x)$ does not halt either.

    A set S of programs is said to be *functionally defined* if for any pair of functionally equivalent programs P and Q, $P \in S$ if and only if $Q \in S$.

    e.g. the set $S = \{\langle Q\rangle \mid Q$ halts on input 0$\}$ is functionally defined, but the set $T = \{\langle Q\rangle \mid Q$ has 10 lines of code$\}$ is not functionally defined.

    Rice's Theorem states that there is no algorithm to decide S if S is a non-trivial functionally defined set (i.e. $S \neq \phi$ and $S \neq \Sigma^*$, where Σ is the alphabet used for strings in S).

    i. Prove that if there is an algorithm $\mathcal{A}_{\text{S}}$ to decide some non-trivial functionally defined set S then there is also an algorithm to decide H.
    Hint. Let Q_{N} be a program that never halts. Suppose that $Q_{\text{N}} \in \overline{S}$ (if not, simply switch the roles of S and $\overline{S}$). Let Q_{H} be a program in S. Now use Q_{N} and Q_{H} to construct a program $R_{P,\,x}$ such that $R_{P,\,x} \in S$ exactly if $P(x)$ halts.

    ii. By applying Rice's Theorem, show that the problems in Questions 12, 13, 16–21 are all undecidable.

    iii. Why does Rice's Theorem not apply to the undecidability of the problems in Questions 14 and 15?

Chapter 6

NP Completeness

6.1 Introduction

This chapter studies the boundary between feasible and infeasible computations, that is between problems that can be solved with a reasonable amount of resources, time being the most critical resource, and those problems that require an inordinate amount of time to solve.

In order to be precise, we need to specify what feasibility means. We define feasibility to be Deterministic Polynomial Time, $\mathcal{P}$ for short. This is by contrast with another class, Non-Deterministic Polynomial Time, $\mathcal{NP}$ for short. As we will see, $\mathcal{P} \subseteq \mathcal{NP}$, and further it is an essentially universal belief that $\mathcal{P} \subsetneq \mathcal{NP}$; the hardest problems in $\mathcal{NP} \setminus \mathcal{P}$, and there are many important problems among these, are believed to be infeasible. Before discussing this further we need to define these classes.

We begin by defining running time in terms of the number of steps a program performs. By a step we intend a basic operation such as an addition, a comparison, a branch (due to a goto, in a while loop, in an if-statement, to perform a procedure call, etc.). We do not allow more complex steps, such as $B \leftarrow 0$ where B is an $n \times n$ array, to count as a single step. Finally, we also limit the word size—the contents of a single memory location—to $O(\log n)$ bits, where n is the input size. This implies that repeated squaring of a number would not be a legitimate series of steps, for it would soon result in arithmetic overflow.

More specifically, we define the running time with respect to the programming language defined in Chapter 4, modified as in Chapter 5 to include write instructions. This allows us to have terminating programs which decide a language (an output of 1 corresponds to recognition of an input, and an output of 0 to rejection). It also allows programs that compute functions.

Definition 6.1.1. *A program or algorithm has worst case running time $T(n)$, running time $T(n)$ for short, if for all inputs of size n it completes its computation in at most $T(n)$ steps.* (*Strictly, in addition, on some input of size n, the full $T(n)$ steps are performed.*)

6.1.1 The Class $\mathcal{P}$

Definition 6.1.2. *An algorithm runs in polynomial time if its running time $T(n)$ is bounded by a polynomial function of n.*

Definition 6.1.3. *A problem or language is in $\mathcal{P}$ (polynomial time) if there is an algorithm solving the problem or deciding the language that runs in polynomial time.*

We view polynomial time as corresponding to the class of problems having efficient algorithms. Of course, if the best algorithm for a problem ran in time n^{100} this would not be efficient in any practical sense. However, in practice, the polynomial time algorithms we find tend to have modest running times such as $O(n), O(n \log n), O(n^2), O(n^3)$.

One reason for defining $\mathcal{P}$ as the class of efficient algorithms is that there is no principled way of partitioning among polynomial time algorithms. Is $\Theta(n^3)$ OK but $\Theta(n^{3.5})$ too slow? Then what about $\Theta(n^{3.1})$, etc.? A second reason is that the class $\mathcal{P}$ is closed under composition: if the output of one polynomial time algorithm is the input to a second polynomial time algorithm, then the overall combined algorithm also runs in time polynomial in the size of the original input.

Another interesting way of viewing polynomial time algorithms is that a moderate increase in resources suffices to double the size of the problems that can be solved. For example, given that algorithm $\mathcal{A}$ can solve a problem of size n in one hour on a particular computer, if $\mathcal{A}$ runs in linear time, then given a new computer that runs twice as fast, in the one hour one could solve twice as large a problem with algorithm $\mathcal{A}$. Running twice as fast means that it performs twice as many operations in the given time. While if $\mathcal{A}$ runs in quadratic time ($\Theta(n^2)$), then give a new computer that runs four times as fast, in one hour one could solve a problem that is twice as large; again, if $\mathcal{A}$ runs in cubic time ($\Theta(n^3)$), then given a new computer that runs eight times as fast, in one hour one could solve twice as large a problem; and so forth.

This contrasts with the effect of an exponential running time ($\Theta(2^n)$, for instance). Here doubling the speed of the computer increases the size of the problem that can be solved in one hour from k to $k+1$, where k was the previously solvable size. As a result, exponential time algorithms are generally considered to be infeasible.

6.1.2 Polynomial Time Defined via Turing Machines

We could also define running time in general, and polynomial time in particular in terms of the number of steps performed by a Turing Machine computation. It will turn out that these two definitions (w.r.t. our programming language and w.r.t. Turing Machines) are equivalent in the sense that they define the same class of problems.

Definition 6.1.4. *A Turing Machine runs in polynomial time if its running time $T(n)$ on an input of length n, i.e. an input that occupies the first n cells of its tape, is bounded by a polynomial function of n.*

Definition 6.1.5 (Turing Machine version)**.** *A problem or language is in $\mathcal{P}$ (polynomial time) if there is a Turing Machine solving the problem or deciding the language which runs in polynomial time.*

We now prove the equivalence of the two definitions of $\mathcal{P}$.

Lemma 6.1.6. *Let Q be a program on w-bit words that runs in time T on an input x. Then there is a Turing Machine simulation of Q which runs in time $O(w^2 \cdot (T + |x|)^2 \cdot T)$.*

Proof. We will use the simulation given in Section 4.2, with two small but important modifications. We begin by analyzing a simulation of Q on the 6-tape Turing Machine M described there. We note that in T steps, Q can access at most $3T$ distinct memory locations. Accordingly, we will store on one of M's tapes the contents of up to $3T$ of Q's memory locations (strictly speaking, the memory of the machine on which Q executes). We call this the *memory tape.* For each of these memory locations, M stores a pair in the form (location, value). These need not be contiguous locations (this is the first modification). Each pair will use $O(w)$ bits and hence $O(w)$ cells on its memory tape.

The second modification concerns M's input. As the input need not be written in contiguous locations of Q's memory, we will assume that M is provided the relevant memory locations as its input in the form of a series of pairs of the form (location, value). We will copy this input to the memory tape. Altogether, therefore, storing P's memory will use $O(w \cdot (T + |x|))$ cells.

Each step of the simulation requires up to three sweeps across M's memory tape, to retrieve and store the up to three variables that are used by the instruction being simulated. A further $O(w)$ steps are needed to perform the instruction. Three tapes are used for these three variables.

Tape 6 is used to hold Q's instruction counter.

Thus M simulates one step of Q's computation using at most $O(w \cdot (T + |x|))$ of its steps, and therefore it simulates Q's T-step computation using $O(w \cdot (T+|x|) \cdot T)$ of its steps, i.e. in this time.

We now simulate this 6-tape Turing Machine M using the 1-tape Machine $\widetilde{M}$ described in the proof of Lemma 4.1.3. Recall that simulating one step of M requires one sweep forward and back across $\widetilde{M}$'s tape, and takes a number of steps equal to the length of tape traversed by $\widetilde{M}$. Also recall that if M uses C cells then so does $\widetilde{M}$. Next, note that M uses $O(w \cdot (T+|x|))$ cells. Thus $\widetilde{M}$ simulates one of M's steps in $O(w \cdot (T + |x|))$ steps. Overall, $\widetilde{M}$ simulates Q's computation using $O(w^2 \cdot (T + |x|)^2 \cdot T)$ steps. ☐

Theorem 6.1.7. *If Q is a program that runs in polynomial time, then the Turing Machine simulating Q described in Lemma 6.1.6 also runs in polynomial time.*

Proof. We make the standard assumption that a polynomial time algorithm uses a memory of polynomial size, and that it uses words of size $c \log n$ for some constant $c \geq 1$. Let n be the size of the input. Then, in Lemma 6.1.6 set $w = c \log n$ and $T = p(n)$, where $p(n)$ is the polynomial bounding the running time of Q. Then by Lemma 6.1.6, the simulating Turing Machine runs in time $O(\log^2 n \cdot (n + p(n))^2 \cdot p(n)) = O(n \cdot (n + p(n))^3)$, which is also a polynomial bound. ☐

It is easy to show the complementary result, namely given a Turing Machine M that runs in polynomial time, there is a program P that simulates M and also runs in polynomial time.

6.1.3 The Class $\mathcal{NP}$

This class of languages is characterized by being "verifiable" in polynomial time. We begin with some examples.

Example 6.1.8. Hamiltonian Cycle.
Input: A directed graph $G = (V, E)$.

Question: Does G have a Hamiltonian Cycle, that is a cycle that goes through each vertex in V exactly once?

Output: "Yes" or "no" as appropriate.

The "Yes" (or "Recognize") answer is polynomial time verifiable in the following sense. Given a sequence of $n = |V|$ vertices which is claimed to form a Hamiltonian Cycle this is readily checked in linear time (it suffices to check that each vertex appears exactly once in the list and that if the list is the sequence $v_1, v_2, \cdots, v_n$, then $(v_i, v_{(i+1) \bmod n}) \in E$ for each $i, 1 \le i \le n$.)

If this sequence of vertices forms a Hamiltonian Cycle, as claimed, it is called a *certificate*.

By definition, a Hamiltonian graph has a certificate, namely the list of vertices forming (one of) its Hamiltonian Cycle(s). On the other hand, if the graph is not Hamiltonian, i.e. it does not have a Hamiltonian Cycle, then no attempted certificate will check out, for no proposed sequence of vertices will form a Hamiltonian Cycle. However, it might be that you are given a proposed certificate for a Hamiltonian graph which fails because it is not a Hamiltonian Cycle. This failure does not show anything regarding whether the graph is Hamiltonian; it only shows that the sequence of vertices did not form a Hamiltonian Cycle for this particular graph. You cannot conclude whether or not the graph is Hamiltonian from this.

Example 6.1.9. Clique.
Input: (G, k) where G is an undirected graph and k an integer.

Question: Does G have a clique of size k? A clique is a subset of vertices such that every pair of vertices in the subset is joined by an edge.

Output: "Yes" or "no" as appropriate.

Again the "Yes" (or "Recognize") answer is polynomial time verifiable, given the following additional information, or *certificate*: a list of k vertices which are claimed to form a clique. To verify this it suffices to check that the list comprises k distinct vertices and that each pair of vertices in this list are joined by an edge. This is readily done in $O(k^2 n)$ time, and indeed in $O(n^3)$ time (for if $k > n = |V|$, then the graph does not have a k-clique).

Again, if the graph does not have a k-clique, then any set of k vertices will fail to be fully connected; there will be a pair of vertices in the given k-vertex set with no edge between them. So any proposed certificate will fail to demonstrate that the graph has a k-clique.

However, just because you are given a set of k-vertices that does not happen to form a k-clique, you cannot then conclude that the graph has no k-clique.

Example 6.1.10. Satisfiability.
Input: F, a CNF Boolean formula.

F is in CNF form if $F = C_1 \wedge C_2 \wedge \cdots \wedge C_m$, where each clause $C_i, 1 \le i \le m$, is an 'or' of literals:

$$C_i = l_{i1} \vee l_{i2} \vee \cdots \vee l_{ij_i},$$

where each l_{ik} is a Boolean variable or its complement (negation).
e.g. $F_1 = (x_1 \vee x_2) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee x_3) \wedge (x_3 \vee x_2); F_2 = x; F_3 = x \wedge \overline{x}$.

Question: Is F satisfiable? That is, is there an assignment of truth values to F's variables that causes F to evaluate to TRUE?
e.g. For F_1, setting $x_1 =$ TRUE, $x_2 =$ FALSE, $x_3 =$ TRUE, causes F_1 to evaluate to TRUE; For F_2, setting $x =$ TRUE causes F_2 to evaluate to TRUE; for F_3, no setting of the variables will cause F_3 to evaluate to TRUE.

Here, the additional information, or certificate, is the assignment of truth values to the formula's variables that causes the formula to evaluate to TRUE. Notice that if the formula never evaluates to TRUE, then any proposed truth assignment for the Boolean variables will cause the formula to evaluate to FALSE. In other words, any proposed certificate fails.

Again, just because you are given a truth assignment that cause the formula to evaluate to FALSE, you cannot then conclude that the formula is not satisfiable. All you know is that this particular collection of truth assignments to the Boolean variables did not work (i.e. they caused the Formula to evaluate to FALSE in this case).

Definition 6.1.11. *An assignment σ for the variables of a Boolean formula F is said to be* satisfying *if it causes the formula to evaluate to* TRUE.

Definition 6.1.12. *A language $L \in \mathcal{NP}$ if there are polynomials p, q, and an algorithm V called the verifier, such that:*

- *If $x \in L$ there is a* certificate *$C(x)$, of length at most $p(|x|)$, such that $V(x, C(x))$ outputs "Yes".*
- *While if $x \notin L$, for every string y of length at most $p(|x|)$, $V(x, y)$ outputs "No".*

In addition, $V(x, y)$ runs in time $q(|x| + |y|) = O(q(n + p(n)))$, where $|x| = n$.

Comment. If $x \notin L$, there is no certificate for x; in other words, any string claiming to be a certificate is readily exposed as a non-certificate.

However, a non-certificate does not determine whether $x \in L$ or $x \notin L$.

6.1.4 Discussion

It is not known whether $\mathcal{P} = \mathcal{NP}$ or $\mathcal{P} \subsetneq \mathcal{NP}$ ($\mathcal{P} \subseteq \mathcal{NP}$ as every problem Q in $\mathcal{P}$ has a polynomial time verifier: the verifier uses the empty string as the certificate and runs the polynomial time recognizer for Q).

However, due to the lack of success in finding efficient algorithms for problems in $\mathcal{NP}$, and in particular for the so-called $\mathcal{NP}$-Complete problems, or $\mathcal{NPC}$ problems for short, it is essentially universally believed that $\mathcal{P} \neq \mathcal{NP}$. $\mathcal{NPC}$ problems are the hardest problems in the class $\mathcal{NP}$ and they have the interesting feature that if a polynomial time algorithm is found for even one of these problems, then they will all have polynomial time algorithms. Given that many of these problems have significant practical importance, considerable effort has gone into attempting to find efficient algorithms for these problems. The lack of success of this effort has reinforced the belief that there are no efficient algorithms for these problems. In some sense, such an efficient algorithm would provide an efficient implementation of the "process" of inspired "guessing," which strikes many as implausible.

Factoring is an example of a problem in $\mathcal{NP}$ for which no efficient algorithm is known. By factoring, we mean the problem of reporting a non-trivial factor of a composite number. As it happens, there is a polynomial time algorithm to determine whether a number is composite or prime, but this algorithm does not reveal any factors in the case of a composite number. Indeed, the assumed hardness of the factoring problem underlies the security of the RSA public key scheme, which at present is a key element of the security of much of e-commerce and Internet communication.

It is also worth noting that factoring is not believed to be an $\mathcal{NPC}$ problem, that is it is not believed to be a "hardest" problem in the class $\mathcal{NP}$.

However, in the end, none of the above evidence demonstrates the hardness of $\mathcal{NPC}$ problems. Rather, it indicates why people believe these problems to be hard.

6.2 Reductions Between $\mathcal{NP}$ Problems

As with undecidable problems, the tool to show problems are $\mathcal{NPC}$ problems is a suitable type of reduction. The basic form is the following. Given a polynomial time decision or membership algorithm for language A we use it as a subroutine to give a polynomial time algorithm for problem B. Recall that a decision algorithm reports "Recognize" ("Yes") or "Reject" ("No") as appropriate. This is called a *Polynomial Time Reduction.*

Our goal is to show reductions among all $\mathcal{NPC}$ problems, for this then leads to the conclusion that if one of them can be solved by a polynomial time algorithm, then they all can be solved in polynomial time. We begin with several examples of such reductions.

6.2.1 Independent Set and Clique

Example 6.2.1. Independent Set.
Input: Undirected Graph G, integer k.
Question: Does G have an independent set of size k, that is a subset of k vertices with no edges between them?

Claim 6.2.2. *Given a polynomial time algorithm for Clique there is a polynomial time algorithm for Independent Set.*

Proof. Let $G = (V, E)$ be the input to the Independent Set problem. Consider the graph $\overline{G} = (V, \overline{E})$. We note that $S \subseteq V$ is an independent set in G if and only if it is a clique in $\overline{G}$. (For if a subset of k vertices have no edges among themselves in graph G, then in graph $\overline{G}$ all possible edges between them are present, i.e. the k vertices form a k-clique in $\overline{G}$. The converse is true also: a set of k vertices forming a k-clique in $\overline{G}$ also form an independent set in G. An example is shown in Figure 6.1.

Thus the Independent Set algorithm is simply to run the Clique algorithm on the pair $(\overline{G}, k)$, and report its result. $\square$

Terminology We will use the name of a problem to indicate the set of objects satisfying the problem condition. Thus Indpt-Set $= \{(G, k) \mid G$ has an independent set of size $k\}$.

The algorithm $\mathcal{A}_{\text{IS}}$, for Independent Set, performs 3 steps.

1. On input (G, k), compute the input $(\overline{G}, k)$ for the Clique algorithm, $\mathcal{A}_{\text{clique}}$.
2. Run $\mathcal{A}_{\text{Clique}}(\overline{G}, k)$.
3. Use the answer from Step 2 to determine its own answer (the same answer is this case).

Demonstrating the algorithm is correct also entails showing:

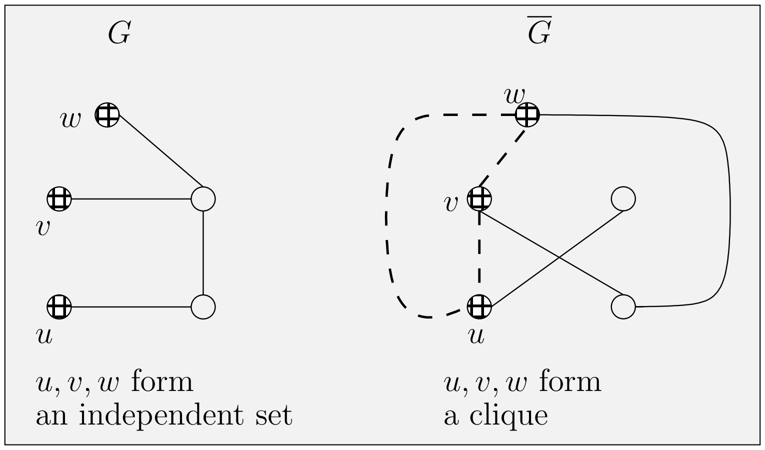


Figure 6.1: Example: An Independent Set in G forms a clique in $\overline{G}$.

4. $\mathcal{A}_{\text{IS}}$'s answer is correct. In this case that means showing that

$$(G, k) \in \text{Indpt-Set} \iff (\overline{G}, k) \in \text{Clique}.$$

   This was argued in the proof of Claim 6.2.2.

   Often, we will ague Step 4 right after Step 1 in order to explain why the constructed input makes sense.

5. Assuming that $\mathcal{A}_{\text{Clique}}$ runs in polynomial time, then showing that $\mathcal{A}_{\text{IS}}$ also runs in polynomial time.
   This is straightforward. For $(\overline{G}, k)$ can be computed in $O(n^2)$ time, so Step 1 runs in polynomial time. For Step 2, as $\mathcal{A}_{\text{Clique}}$ receives an input of size $O(n^2)$ and as $\mathcal{A}_{\text{Clique}}$ runs in polynomial time by assumption, Step 2 also runs in polynomial time. Finally, Step 3 takes $O(1)$ time, so the whole algorithm runs in time polynomial in the size of its input.

   As a rule, we will not spell out these arguments in such detail as they are straightforward.

The next claim is similar and is left to the reader.

Claim 6.2.3. *Given a polynomial time algorithm for Independent Set there is a polynomial time algorithm for Clique.*

6.2.2 Hamiltonian Path and Cycle

Example 6.2.4. Hamiltonian Path (HP).
Input: (G, s, t), where $G = (V, E)$ is a directed graph and $s, t \in V$.

Question: Does G have a Hamiltonian Path from s to t, that is a path that goes through each vertex exactly once?

Claim 6.2.5. *Given a polynomial time algorithm for Hamiltonian Cycle (HC) there is a polynomial time algorithm for Hamiltonian Path.*

Proof. Let (G, s, t) be the input to the Hamiltonian Path Problem. $\mathcal{A}_{\text{HP}}$, the algorithm for the Hamiltonian Path problem, proceeds as follows.

1. Build a graph H with the property that H has a Hamiltonian Cycle exactly if G has a Hamiltonian Path from s to t.
2. Run $\mathcal{A}_{\text{HC}}(H)$.
3. Report the answer given by $\mathcal{A}_{\text{HC}}(H)$.

H consists of G plus one new vertex, z say, together with new edges $(t, z), (z, s)$.

4. We argue that G has a Hamiltonian Path from s to t exactly if H has a Hamiltonian Cycle. For if H has a Hamiltonian Cycle it includes edges $(z, s), (t, z)$ as these are the only edges incident on z; we write $s = v_1$ and $t = v_n$. So the cycle has the form $z, v_1, v_2, \cdots, v_n$, and then $v_1, v_2, \cdots, v_n$ is the corresponding Hamiltonian Path in G. Conversely, if G has Hamiltonian Path $v'_1, v'_2, \cdots, v'_n$, where $s = v'_1$ and $t = v'_n$, then H has Hamiltonian Cycle $z, v'_1, v'_2, \cdots, v'_n$. An example of this construction is shown in Figure 6.2.
5. Clearly $\mathcal{A}_{\text{HP}}$ runs in polynomial time if $\mathcal{A}_{\text{HC}}$ runs in polynomial time.

□

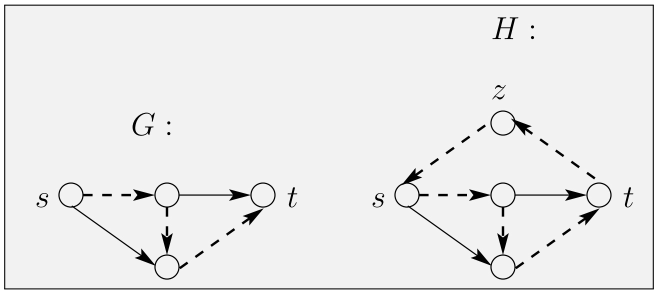


Figure 6.2: G has a Hamiltonian Path from s to t iff H has a Hamiltonian Cycle. A corresponding path and cycle are shown as dashed edges.

Example 6.2.6. Degree d Bounded Spanning Tree (DBST).
Input: (G, d), where G is an undirected graph G and d is an integer.

Question: Does G have a spanning tree T such that in T each vertex has degree at most d?

Example 6.2.7. Undirected Hamiltonian Path (UHC).
Input: (G, s, t), where $G = (V, E)$ is an undirected graph, and $s, t \in V$.

Question: Does G have a Hamiltonian Path from s to t, that is a path going through each vertex exactly once?

Claim 6.2.8. *Given a polynomial time algorithm for Degree d Bounded Spanning Tree there is a polynomial time algorithm for Undirected Hamiltonian Path.*

Proof. Let (G, s, t) be the input to the Hamiltonian Path Problem. $\mathcal{A}_{\text{UHP}}$, the algorithm for the Hamiltonian Path problem, proceeds as follows.

1. Build a graph H with the property that H has a degree 3 bounded spanning tree exactly if G has a Hamiltonian Path from s to t.
2. Run $\mathcal{A}_{\text{DBST}}(H, 3)$.
3. Report the answer given by $\mathcal{A}_{\text{DBST}}(H, 3)$.

H is the following graph: G plus vertices v' for each $v \in V$, plus vertices s'', t'', together with edges (v', v) for each $v \in V$ and edges $(s'', s), (t'', t)$. An example is shown in Figure 6.3.

4. We argue that G has a Hamiltonian Path from s to t exactly if H has a spanning tree with degree bound 3. Note that all the new vertices in H have degree 1 and consequently all the new edges must be in any spanning tree T of H. Removing these edges and the new vertices leaves a tree in which each vertex has degree at most 2, and in which s and t both have degree at most 1. As the resulting graph is still connected, this means that the remaining edges in T form a Hamiltonian Path in G from s to t.
5. Clearly $\mathcal{A}_{\text{HP}}$ runs in polynomial time if $\mathcal{A}_{\text{DBST}}$ runs in polynomial time.

$\square$

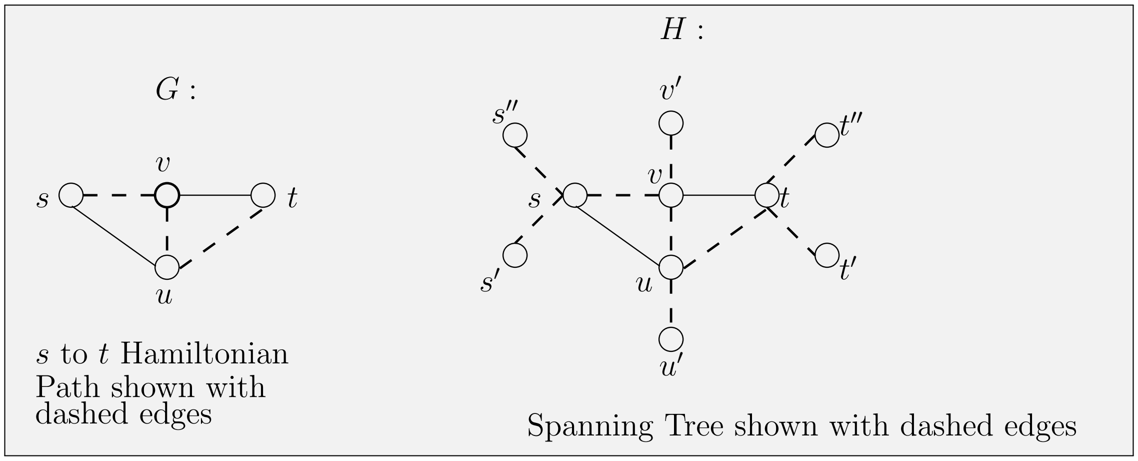

H :
G :
v
s
t
u
s to t Hamiltonian Path shown with dashed edges
s''
v'
t''
s
v
t
s'
u
t'
u'
Spanning Tree shown with dashed edges

Figure 6.3: G has a Hamiltonian Path from s to t if and only if H has a Degree 3 Bounded Spanning Tree.

6.3 The Structure of Reductions

We observe that all these constructions have the following form: given a polynomial time algorithm $\mathcal{A}_B$ for membership in B we construct a polynomial time algorithm $\mathcal{A}_A$ for membership in A as follows.

Input to $\mathcal{A}_A$: I.

1. Construct $f_{A\leq B}(I)$ in polynomial time.
2. Run $\mathcal{A}_B(f_{A\leq B}(I))$.
3. Report the answer from Step 2.

If there is such a function $f_{A\leq B}$, we write $A \leq_P B$, or $A \leq B$ for short. It is read as: A can be reduced to B in polynomial time. We call the function $f_{A\leq B}$ the reduction.
In order for Step 3 to be correct we need that:

$$I \in A \Longleftrightarrow f_{A\leq B}(I) \in B.$$

As $f_{A\leq B}(I)$ is computed in polynomial time it produces a polynomial sized output. Thus $\mathcal{A}_B(f_{A\leq B}(I))$ runs in time polynomial in $|I|$, as $\mathcal{A}_B$ runs in polynomial time. This uses the fact that if p and q are bounded degree polynomials, then so is $p(q(\cdot))$.

This is called a polynomial time reduction of problem A to problem B. It shows that if there is a polynomial time membership (decision) algorithm for B then there is also a polynomial time membership algorithm for A. The converse is true also:

> If there is no polynomial time membership algorithm for A then there is not one for B either.

6.4 More Examples of Reductions

6.4.1 Undirected Hamiltonian Path and Cycle

Example 6.4.1. *Undirected Hamiltonian Path* (UHP) and *Undirected Hamiltonian Cycle* (UHC) are the same problems as the corresponding problems for directed graphs, but the new problems are for undirected graphs.

Claim 6.4.2. *Given a polynomial time algorithm for* Undirected Hamiltonian Path (UHP) *there is a polynomial time algorithm for* Directed Hamiltonian Path*.*

Proof. Let (G, s, t) be the input to the Directed Hamiltonian Path algorithm, $\mathcal{A}_{\text{DHP}}$. The algorithm proceeds as follows.

1. Build the triple (H, p, q), where H is an undirected graph, and p and q are vertices in H with the property that
$$(G, s, t) \in \text{DHP} \iff (H, p, q) \in \text{UHP}.$$
2. Run $\mathcal{A}_{\text{UHP}}(H, p, q)$.
3. Report the answer given by $\mathcal{A}_{\text{UHP}}(H, p, q)$.

Now, we need to describe the construction and observe its correctness.

Let $G = (V, E)$ and $H = (F, W)$. Let $n = |V|$. (H, p, q) is constructed as follows. For each vertex $v \in V$, H has 3 vertices $v^{\text{in}}, v^{\text{mid}}, v^{\text{out}}$, plus the edges $(v^{\text{in}}, v^{\text{mid}}), (v^{\text{mid}}, v^{\text{out}})$. And for each edge $(v, w) \in E$, H receives edge $(v^{\text{out}}, w^{\text{in}})$. See Figure 6.4 for an illustration. Then $\mathcal{A}_{\text{DHP}}$ sets $p = s^{\text{in}}$ and $q = t^{\text{out}}$.

An example of the reduction is shown in Figure 6.5.

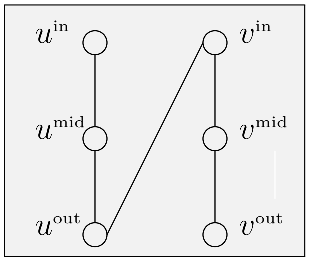


Figure 6.4: Edges in H corresponding to edge (u, v) in G

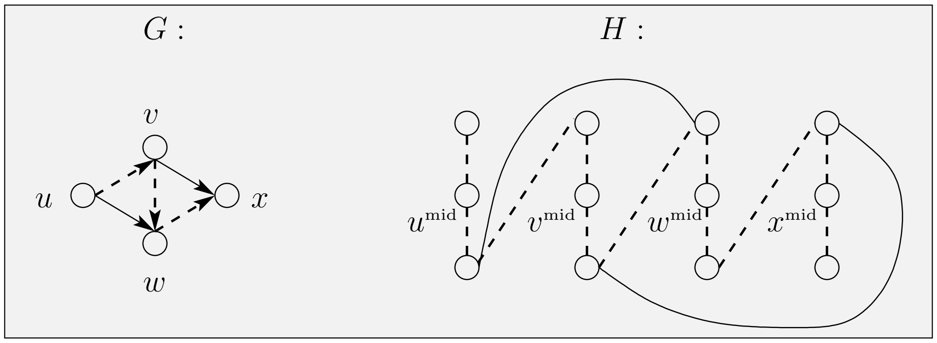


Figure 6.5: An example of the reduction from HP to UHP.

4. We argue that G has a Hamiltonian Path from s to t exactly if H has a Hamiltonian Path from p to q.

   First, suppose that G has a Hamiltonian Path from s to t. Suppose that the path is $u_1, u_2, \cdots, u_n$, where $s = u_1, t = u_n$, and $u_1, u_2, \cdots, u_n$ is a permutation of the vertices in V. Then the following path is a Hamiltonian Path from p to q in H.

$$p = s^{\text{in}} = u_1^{\text{in}}, u_1^{\text{mid}}, u_1^{\text{out}}, u_2^{\text{in}}, u_2^{\text{mid}}, u_2^{\text{out}}, \cdots, u_n^{\text{in}}, u_n^{\text{mid}}, u_n^{\text{out}} = t^{\text{out}} = q.$$

   Next, suppose that H has a Hamiltonian Path P from p to q. For P to go through vertex v^{mid}, P must include edges $(v^{\text{in}}, v^{\text{mid}})$ and $(v^{\text{mid}}, v^{\text{out}})$. Thus the path has the form

$$s^{\text{in}} = p = v_1^{\text{in}}, v_1^{\text{mid}}, v_1^{\text{out}}, v_2^{\text{in}}, v_2^{\text{mid}}, v_2^{\text{out}}, \cdots, v_n^{\text{in}}, v_n^{\text{mid}}, v_n^{\text{out}} = q = t^{\text{out}},$$

   where $|V| = n$, $s = v_1$, $t = v_n$, and $v_1, v_2, \cdots, v_n$ is a permutation of the vertices in V. Then $v_1, v_2, \cdots, v_n$ is a Hamiltonian Path from s to t in G.

   This shows the claim.

5. Clearly $\mathcal{A}_{\text{DHP}}$ runs in polynomial time if $\mathcal{A}_{\text{UHP}}$ runs in polynomial time.

$\square$

6.4.2 Vertex Cover, Independent Set, and 3-SAT

Example 6.4.3. Vertex Cover (VC).

Input: (G, k), where $G = (V, E)$ is an undirected graph and k is an integer.

Question: Does G have a vertex cover of size k, that is a subset $U \subseteq V$ of size at most k with the property that every edge in E has at least one endpoint in U?

Claim 6.4.4. *Given a polynomial time algorithm for* Independent Set (IS) *there is a polynomial time algorithm for* Vertex Cover.

Proof. This is immediate from the following observation. If I is an independent set of G then $V - I$ is a vertex cover, and conversely. See Figure 6.6 for an example.

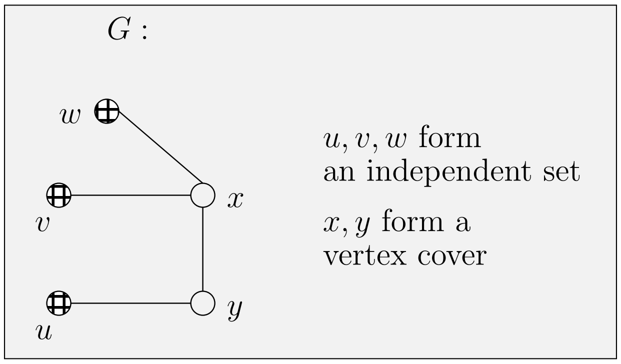


Figure 6.6: Example: The complement of an Independent Set is a Vertex Cover.

For if I is an independent set, then as no edge joins pairs of vertices in I, any edge has at most one endpoint in I, and hence at least one endpoint in $V - I$. Conversely, if $V - I$ is a vertex cover, then any edge has at least one endpoint in $V - I$, and so there is no edge with two endpoints in I, meaning that I is an independent set.

We can conclude that $(G, k) \in \text{VC} \iff (G, |V| - k) \in \text{IS}$.

We leave the details of creating the algorithm for Vertex Cover to the reader. □

Example 6.4.5. 3-SAT.

Input: F, a CNF Boolean Formula in which each clause has at most 3 variables.

Question: Does F have a satisfying assignment (of Boolean values to its variables)?

Claim 6.4.6. *Given a polynomial time algorithm for* Independent Set (IS) *there is a polynomial time algorithm for* 3-SAT.

Proof. Let F be the input to the 3-SAT algorithm, $\mathcal{A}_{\text{3-SAT}}$. The algorithm proceeds as follows.

1. $\mathcal{A}_{\text{3-SAT}}$ computes (G, k), where G is a graph and k is an integer, with the property that

$$F \in \text{3-SAT} \iff (G, k) \in \text{IS}.$$

2. It runs $\mathcal{A}_{\text{IS}}(G,k)$.
3. It reports the answer given by $\mathcal{A}_{\text{IS}}(G,k)$.

Let $F = C_1 \wedge C_2 \wedge \cdots \wedge C_\ell$, and suppose that F has variables $x_1, x_2, \cdots, x_n$. G receives the following vertices. For each clause $C_j = a \vee b \vee c$ it has vertices u_a^j, u_b^j, u_c^j connected in a triangle (if the clause is $C_j = a \vee b$ it has vertices u_a^j, u_b^j joined by an edge, and if $C_j = a$ it has the vertex u_a^j alone). In addition, for every pair j, k of distinct clauses, for each variable x appearing in C_j as x and in C_k as $\overline{x}$, the pair of vertices u_x^j and $u_{\overline{x}}^k$ are joined together. See Figure 6.7 for an example. Finally, $\mathcal{A}_{\text{3-SAT}}$ sets $k = \ell$.

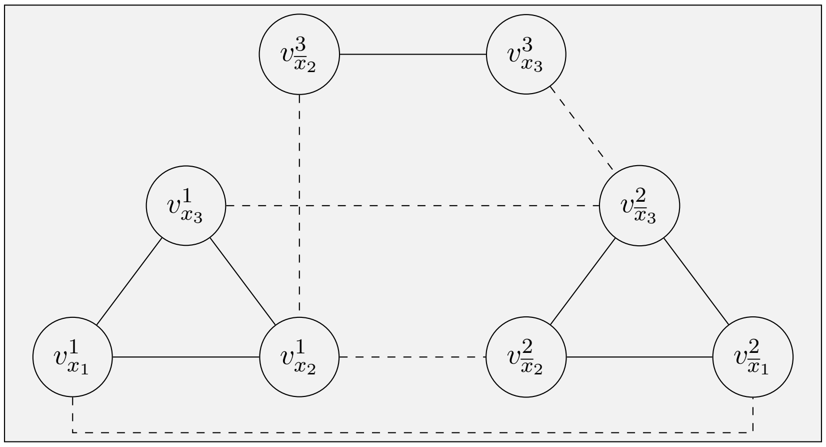


Figure 6.7: The graph for the Independent Set construction for $F = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3) \wedge (\overline{x}_2 \vee x_3)$. The edges joining nodes in different triangles are shown as dashed lines for greater visibility.

4. We argue that F has a satisfying assignment exactly if G has an independent set I of size ℓ.

   Suppose that F has a satisfying assignment σ. For each clause C_j, we identify one literal l_{ij} that has value TRUE in assignment σ. We add the corresponding vertex $v^j_{l_{ij}}$ to I. This puts ℓ vertices in I. We observe that there are no edges among the vertices added to I for the following two reasons: first, only one vertex per triangle is added. Second, for each variable x, if it is set to TRUE, then only vertices corresponding to the literal x are added to I, and there are no edges between these vertices; similarly, if x is set to FALSE, the added vertices correspond to the literal $\overline{x}$ and again there are no edges among these vertices. Consequently, the vertices in I form an independent set of ℓ vertices.

   Conversely, suppose that G has an independent set I of size ℓ. We describe a satisfying assignment σ for F. For each clause $C_j = a \vee b \vee c$, I can include at most one of u_a^j, u_b^j, u_c^j as these three vertices are all connected (and similarly for a clause of two or one literals). This provides at most ℓ vertices. As no other vertices are available, I must include one vertex for each clause.

   We set the variable truth values to match the vertices in I. If a vertex v_x is in I, then x is set to TRUE, and if a vertex $v_{\overline{x}}$ is in I, then x is set to FALSE. Note that both cannot occur as there would then be an edge between the relevant vertices. If any variables x remain unset, they can be set to TRUE. Note that in each clause at least one literal has been set to TRUE and therefore the formula evaluates to TRUE.

This shows the claim.

5. Clearly $\mathcal{A}_{\text{3-SAT}}$ runs in polynomial time if $\mathcal{A}_{\text{IS}}$ runs in polynomial time.

□

Next, we show a reduction from IS to SAT. This introduces the technique of expressing NP problems in terms of Boolean formulas.

Claim 6.4.7. *Given a polynomial time algorithm for* SAT *there is a polynomial time algorithm for* Independent Set (IS).

Proof. Let (G, k) be the input to IS where $G = (V, E)$. Suppose $V = \{v_1, v_2, \ldots, v_n\}$. We will construct a CNF Boolean formula F such that $F \in$ SAT if and only if $(G, k) \in$ IS.

F will have kn variables x_i^j, for $1 \le i \le n$ and $1 \le j \le k$. We think of $x_i^j =$ TRUE as indicating that v_i is the jth vextex in the independent set (the order of the vertices in an independent set can be chosen arbitrarily). More precisely, if I is a size k independent set for G, containing vertices $v_{h_1}, v_{h_2}, \ldots, v_{h_k}$, then the variables $x_{h_j}^j$ will be set to TRUE, for $1 \le j \le k$, and all the other variables will be set to FALSE.

Now we specify the clauses needed to enforce this meaning for the variables x_i^j. First, we require that for each j, exactly one of the x_i^j be set to TRUE; this corresponds to exactly one vertex being chosen as the jth vertex in I. To this end, we use two collections of clauses.

$$F_{j1} = (x_1^j \vee x_2^j \vee \ldots \vee x_n^j).$$

If satisfied, this ensures at least one of the x_i^j is set to TRUE.

$$F_{j2} = \wedge_{1 \le h < i \le n} (\overline{x}_h^j \vee \overline{x}_i^j).$$

If satisfied, the clause for h and i ensures that at most one of x_h^j and x_i^j is set to TRUE. Together, if satisfied, these two collections of clauses ensure exactly one of the x_i^j is set to TRUE.

Second, we require that the chosen truth values correspond to the jth and ℓth vertices in I being distinct, for $1 \le j < \ell \le k$. This follows if, for each i, at most one of the variables $x_i^1, x_i^2, \ldots, x_i^k$ is set to TRUE. To this end, we use the clauses

$$F_3 = \wedge_{1 \le i \le n} \wedge_{1 \le j < \ell \le k} (\overline{x}_i^j \vee \overline{x}_i^\ell).$$

The clause $\overline{x}_i^j \vee \overline{x}_i^\ell$ ensures that at most one of x_i^j and x_i^ℓ is set to TRUE for $j \neq \ell$.

Third, we ensure that the chosen truth values correspond to there being no edges between the vertices in I. This follows if, for each edge (v_h, v_i), at most one of x_h^j and x_i^ℓ is set to TRUE, for $j \neq \ell$ and $1 \le j, \ell \le k$. To this end, we use the clauses

$$F_4 = \wedge_{(v_h, v_i) \in E} \wedge_{1 \le j, \ell \le k, j \neq \ell} (\overline{x}_h^j \vee \overline{x}_i^\ell).$$

The final formula F is given by

$$F = \wedge_{1 \le j \le k} F_{j1} \wedge_{1 \le j \le k} F_{j2} \wedge F_3 \wedge F_4.$$

Clearly, F can be built in time $O(n^3)$, where $n = |V|$.

Suppose G has a size k independent set containing vertices $v_{h_1}, v_{h_2}, \ldots, v_{h_k}$. Then, as specified above, the variables $x^j_{h_j}$ aree set to TRUE, for $1 \le j \le k$, and all the other variables are set to FALSE. The just described construction ensures F evaluates to TRUE.

Conversely, suppose that F evaluates to TRUE. For each variable x^j_i = TRUE, we add v_i to the independent set I. The construction of F ensures that exactly k variables are set to TRUE, at most one for each value of i, and exactly one for each value of k. Thus k distinct vertices are added to I. Furthermore, F_4 ensures there are no edges among these vertices. Thus the set I is indeed a size k independent set.

□

6.4.3 3-SAT, SAT, and G-SAT

Example 6.4.8. G-SAT.
Input: F, a Boolean Formula.
Question: Does F have a satisfying assignment (of Boolean values to its variables), that is, an assignment that causes it to evaluate to TRUE?

Claim 6.4.9. *Given a polynomial time algorithm for* 3-Satisfiability (3-SAT) *there is a polynomial time algorithm for* G-SAT*.*

Proof. Let F be the input to the G-SAT algorithm, $\mathcal{A}_{\text{G-SAT}}$. The algorithm proceeds as follows.

1. $\mathcal{A}_{\text{G-SAT}}$ computes E, where E is a 3-CNF formula, with the property that
$$F \in \text{G-SAT} \iff E \in \text{3-SAT}.$$
2. It runs $\mathcal{A}_{\text{3-SAT}}(E)$.
3. It reports the answer given by $\mathcal{A}_{\text{3-SAT}}(E)$.

$\mathcal{A}_{\text{G-SAT}}$ computes a CNF formula E such that E is satisfiable if and only if F is satisfiable. To help understand the process, let's view F as a binary expression tree. The clause $x_1 \vee x_2 \vee \ldots \vee x_k$ can be written as $x_1 \vee (x_2 \vee (\ldots \vee (x_{k-1} \vee x_k) \ldots)$ and similarly for the clause $x_1 \wedge x_2 \wedge \ldots \wedge x_k$. E receives a new variable for each internal node in the expression tree. Note that the leaves retain their literal labels (either x or $\overline{x}$). Let r be the variable at the root. The variables labeling the internal nodes are intended to take on the values the corresponding nodes would take on when evaluating the expression tree. An example is shown in Figure 6.8.

For a parent node with operator $\neg$ and corresponding variable z, with its child having corresponding variable y, several clauses are added to E, clauses that evaluate to TRUE exactly if $z = \overline{y}$, which is the value z should take on. The following clauses suffice:

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

For if B evaluates to TRUE (as is needed for E to evaluate to TRUE) then one of the following two situations applies:

- z is set to TRUE; then $\overline{z}$ is FALSE and y must be set to TRUE.

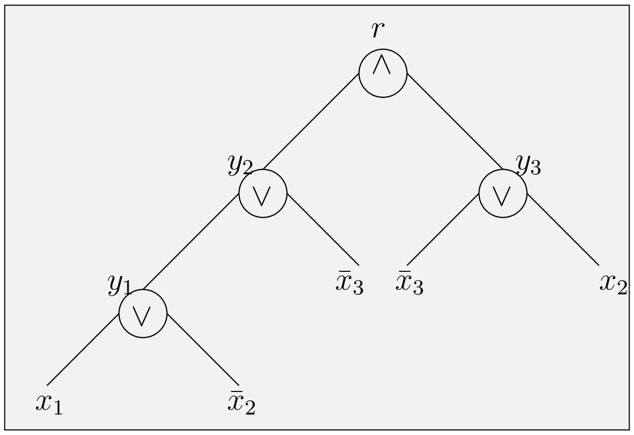


Figure 6.8: Expression Tree Representation of $F = [(x_1 \vee \overline{x}_2) \vee \overline{x}_3] \wedge (\overline{x}_3 \vee x_2)$.

- z is set to FALSE; then $\overline{y}$ must evaluate to TRUE, i.e. y is set to FALSE.

In sum, $z = \overline{x}$.

Similarly, for a parent node with operator $\vee$ and corresponding variable z, with children having corresponding variables x and y, the following clauses are added to E:

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

For if B evaluates to TRUE then one of the following two situations applies:

- z is set to TRUE; then $\overline{z}$ is FALSE and $x \vee y$ must evaluate to TRUE.
- z is set to FALSE; then both $\overline{x}$ and $\overline{y}$ must be set to TRUE, and consequently $x \vee y$ evaluates to FALSE.

In sum, $z = x \vee y$.

Similarly, for a parent node with operator $\wedge$ and corresponding variable z, with children having corresponding variables x and y, the following clauses are put into E:

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

For if B evaluates to TRUE then one of the following two situations applies:

- z is set to TRUE; then both x and y must be set to TRUE.
- z is set to FALSE; then $\overline{x} \vee \overline{y}$ must evaluate to TRUE, i.e. $x \wedge y$ evaluates to FALSE.

In sum, $z = x \wedge y$.

E consists of the 'and' of these collections of clauses, one collection per internal node in the expression tree, 'and'-ed with the clause (r).

4. Now we show that $F \in$ G-SAT $\iff E \in$ SAT.

   First, suppose that E is satisfiable. Let τ be a satisfying assignment of Boolean values for E. Choose the variables in F to receive the Boolean values given by τ. By construction, if E

evaluates to True, the Boolean values of the variables in E correspond to an evaluation of the expression tree for F. In particular, the Boolean value of r in E is the Boolean value of the root of the expression tree for F, or more simply of F itself. As E evaluates to True by assumption, so must the clause (r), and consequently F evaluates to True.

Now suppose that F is satisfiable. Let σ be a satisfying assignment of Boolean values for F. Then the following Boolean assignment causes E to evaluate to True: Use σ as the truth assignment for these variables in E too. The remaining unassigned variables in E correspond to the internal nodes of the expression tree for F. These variables are given the Boolean values obtained in the evaluation of F's expression tree when the leaves have Boolean values given by σ.

As F evaluates to True this means that the root of F's expression tree evaluates to True, and hence so does clause (r) in E. The remaining clauses in E evaluate to True exactly if the variable at each parent node v has Boolean value equal to the $\neg$, $\vee$, or $\wedge$, as appropriate, of the Boolean values of the variables for v's children; but this is exactly the Boolean values we have assigned these variables. Consequently, E evaluates to True.

This shows the claim.

5. Clearly $\mathcal{A}_{\text{G-SAT}}$ runs in polynomial time if $\mathcal{A}_{\text{SAT}}$ runs in polynomial time.

$\square$

6.5 NP-Completeness

We begin by showing that reductions compose.

Lemma 6.5.1. *If $J \le_P K \le_P L$ then $J \le_P L$.*

Proof. As $J \le_P K$ there is a polynomial time computable function f such that $x \in J \iff f(x) \in K$. And as $K \le_P L$ there is a polynomial time computable function g such that $y \in K \iff g(y) \in L$. So $x \in J \iff g(f(x))) \in L$, and $g \circ f$ is also polynomial time computable. That is, $J \le_P L$. $\square$

Definition 6.5.2. *A language L is NP-Complete ($L \in \mathcal{NPC}$) if*

1. *$L \in \mathcal{NP}$, and*
2. *For every $J \in \mathcal{NP}$, $J \le_P L$.*

Lemma 6.5.3. *If K is NP-Complete and we show that*

1. *$L \in \mathcal{NP}$, and*
2. *$K \le_P L$*

then we can conclude L is NP-Complete also.

Proof. As $L \in \mathcal{NP}$, it suffices to show that for all $J \in \mathcal{NP}$, $J \le_P L$. Now, for any $J \in \mathcal{NP}$, as K is NP-complete, $J \le_P K$, and we know that $K \le_P L$. By Lemma 6.5.1, we conclude that $J \le_P L$. $\square$

Lemma 6.5.3 means that once we have shown one problem is NP-Complete, e.g. General Satisfiability, then subsequent problems can be shown NP-Complete by means of reductions from G-SAT, as already given in earlier sections.

Example 6.5.4. Universal NPC (U-NPC).
Input: $\langle V, x, 1^s, 1^t \rangle$, where V is a verification algorithm taking inputs x, C.
Question: Is there is a certificate $C(x)$, with $|C(x)| \le s$, such that in at most t steps of computation $V(x, C(x))$ outputs "Yes"?

Claim 6.5.5. *U-NPC is NP-Complete.*

Proof. The verification algorithm for U-NPC, given a candidate certificate C, checks $|C| \le s$, and then simulates $V(x, C)$ for up to t steps, outputting "Yes" if $V(x, C)$ outputs "Yes" in this time and outputting "No" otherwise.

U-NPC $\in \mathcal{NP}$ since any $\langle V, x, 1^s, 1^t \rangle \in$ U-NPC has a certificate $C(x)$ of size at most s, which is less than the length of the input instance $\langle V, x, 1^s, 1^t \rangle$. In addition, the verification algorithm for U-NPC runs in time polynomial in the length of its input, for it needs only $O(t)$ time for the simulation of V, as it will take $O(1)$ time to simulate one step of V.

To show $L \le_P$ U-NPC for any $L \in \mathcal{NP}$ we argue as follows. As $L \in \mathcal{NP}$, there are polynomials p and q, and L has a polynomial time verifier V_L with $V_L(x, y)$ running in time at most $p(|x| + |y|)$, such that each $x \in L$ has a certificate $C(x)$ of length at most $q(|x|)$. Thus $x \in L$ exactly if $\langle V, x, 1^{q(|x|)}, 1^{p(|x|+q(|x|))} \rangle \in$ U-NPC.

So the algorithm $\mathcal{A}_{\text{L}}$ to recognize L, given a polynomial time algorithm $\mathcal{A}_{\text{U-NPC}}$ for U-NPC, proceeds as follows.

1. Construct $\langle V_L, x, 1^{q(n)}, 1^{p(n+q(n))} \rangle$, where $|x| = n$.
2. Run $\mathcal{A}_{\text{U-NPC}}(\langle V_L, x, 1^{q(n)}, 1^{p(n+q(n))} \rangle)$.
3. Report the answer from (2).

It remains to argue that $\mathcal{A}_{\text{L}}$ runs in time polynomial in $n = |x|$. For the purposes of this reduction $|V_L|$ can be viewed as a constant, as it is a fixed value for all $x \in L$, so $|\langle V_L, x, 1^{q(n)}, 1^{p(n+q(n))} \rangle| = O(p(n + q(n)))$, a polynomial in n. Thus $\langle V_L, x, 1^{q(n)}, 1^{p(n+q(n))} \rangle$ can be constructed in time polynomial in n. Consequently, $L \le_P$ U-NPC, and this is true for every $L \in \mathcal{NP}$. □

6.5.1 Characterizing $\mathcal{P}$

In order to prove more problems are NP-Complete it will be helpful to first show that every language in $\mathcal{P}$ can be reduced to the following Circuit Evaluation (CE) problem.

Example 6.5.6. Circuit Evaluation (CE).

Input: C, a circuit comprising *and*, *or*, and *not* gates, with Boolean inputs $x_1, x_2, \cdots, x_n$ and a Boolean output y. (The circuit can be viewed as a directed graph, where the inputs are the names of vertices with no in-edges, the output is the name of a vertex with no out-edge, and every other vertex is labeled by one of the operators (*and*, *or*, or *not*); the latter vertices have one or two inedges as appropriate, but there are no limits on the number of outedges they could have.) See Figure 6.9 for an example of such a circuit.

Question: Given an assignment of Boolean values to the inputs, does the circuit output equal TRUE?

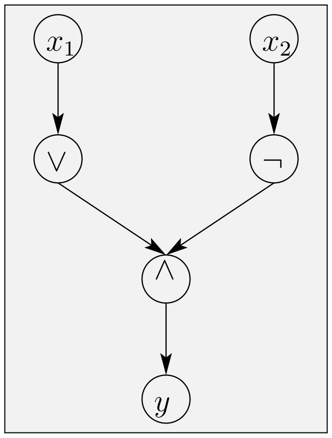


Figure 6.9: Example: A Boolean Circuit.

We now relate the Circuit Value Problem to the computation of a polynomial time Turing Machine.

Lemma 6.5.7. *Let $M = (\Sigma, A, V, start, F, \delta)$ be a 1-tape Turing Machine. Suppose that for each input x of length n, M terminates within T steps of computation. Then there is a circuit C, which takes inputs corresponding to the strings x of length n, whose size is $O(T^2) \cdot |A| \cdot |V|)$, and which outputs 1 if M recognizes x and 0 otherwise.*

Proof. Let $A = \{a_0, a_1, \ldots, a_{r-1}\}$ and $V = \{q_0, q_1, \ldots, q_{s-1}\}$.

C will have $T+1$ layers of nodes. Layer 0 will represent M's input configuration, and layer $t \geq 1$ will represent M's configuration after t steps of computation. In order to allow for the possibility that M's computation ends in fewer that T steps, we introduce the possibility of null moves once M has reached a recognizing vertex (vertices refer to locations in M's finite control, nodes to locations in C). A null move will leave M's current vertex unchanged, the current cell contents unchanged, and move the read/write head one cell to the right.

Note that as the computation now lasts exactly T steps, it uses at most $T+1$ cells (and then only if the read/write head moves right on every step of the computation). For convenience, we number the cells from 0 to T, in left to right order. Now, we describe C's nodes in more detail.

Specification of C's nodes For each cell k, $0 \le k \le T$, for each time t, $0 \le t \le T$, there are r nodes c_{ik}^t, for $0 \le i < r$, corresponding to the possible values in the cell.

More specifically, $c_{ik}^t = 1$ (i.e. TRUE) exactly if after t steps of computation cell k holds character a_i.

Also, for each t, $0 \le t \le T$, there are an additional s nodes v_j^t, for $0 \le j < s$, corresponding to the possible current vertices.

More specifically, $v_j^t = 1$ exactly if after t steps of computation, the current vertex is q_j.

Finally, for each t, $0 \le k \le T$, there are an additional $T+1$ nodes h_k^t, for $0 \le k \le T$, corresponding to the possible current positions for the read/write head.

More specifically, $h_k^t = 1$ exactly if after t steps of computation, the read/write head is over cell k.

For level $t = 0$, we ensure these properties by setting the node values to match M's starting configuration. The next step in the construction is to connect the nodes in level t to those in level $t+1$ to preserve the above properties.

Each of the nodes we have described above, for $t > 0$, will be taking an or over its inputs. There are two cases to consider.

Case 1. The read/write head is over cell k, i.e. $h_k^t = 1$.
Suppose that $c_{ik}^t = 1$, $v_j^t = 1$, and $\delta(a_i, q_j) = (a_{i'}, q_{j'}, R)$. Then the circuit needs to ensure that $c_{i'k}^{t+1} = 1$, $v_{j'}^{t+1} = 1$, and $h_{k+1}^{t+1} = 1$. This is achieved by the edges shown in Figure 6.10.

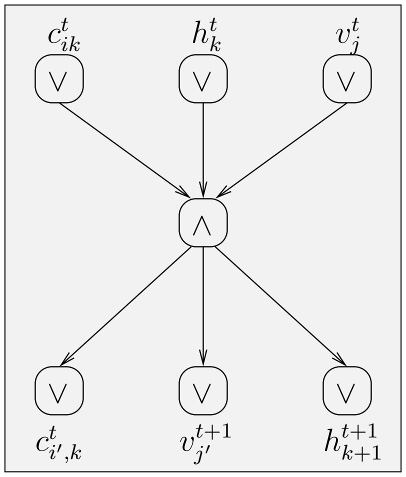


Figure 6.10: The circuit edges when the read/write head is over a cell.

If $\delta(a_i, q_j) = (a_{i'}, q_{j'}, L)$, one need make only a small change to the above construction, so as to ensure that $h_{k-1}^{t+1} = 1$ rather than $h_{k+1}^{t+1} = 1$.

Note that each of the nodes $c_{i'k}^{t+1}$, $v_{j'}^{t+1}$, h_{k+1}^{t+1} may have multiple inputs, corresponding to the multiple predecessor configurations that cause them to have value 1.

Of course, we don't know where the read/write head might be (at least not without simulating M). Thus the above construction is carried out for every triple (i, j, k), with a distinct $\wedge$-node for each triple.

Case 2. The read/write head is not over cell k, i.e. $h_k^t = 0$.
In this case the circuit needs to ensure that $c_{ik}^t = c_{ik}^{t+1}$. This is achieved by the nodes and edges shown in Figure 6.11. This part of the construction provides one more input to the node c_{ik}^{t+1}.

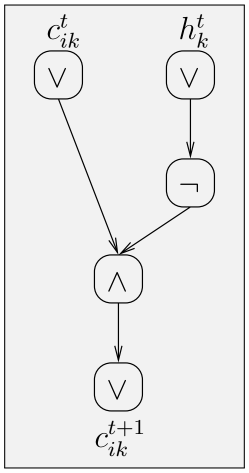


Figure 6.11: The circuit edges when the read/write head is not over a cell.

Again, the construction is carried out for every pair (i, k).

These constructions maintain the properties described in the specification, which ensures that they describe a configuration: For each k and t, there is exactly one $c_{ik}^t = 1$; i.e. cell k has content a_i after t steps of computation. For each t there is exactly one $v_j^t = 1$; i.e. the current vertex after t steps of computation is q_j. For each t there is exactly one $h_k^t = 1$; i.e. the read/write head is over cell k. And further the configuration C_t after t steps of computation is succeeded by the configuration C_{t+1} after $t + 1$ steps of computation: $C_t \vdash C_{t+1}$.

Finally, we need C to produce an output. This output should be 1 if M has reached a recognizing vertex, i.e. a vertex $q_j \in F$, and 0 otherwise. This is obtained by taking the or of the outputs of all the nodes v_j^T for $q_j \in F$.

It follows that C outputs 1 exactly if M recognizes its input x. □

Theorem 6.5.8. *Let $L \in \mathbf{P}$. Then there is a polynomial $r(n)$ and a family of circuits $C_1, C_2, \cdots$ with sizes $|C_n| \le r(n)$, such that for each $x \in L$ with $|x| = n$,*

$$C_n(x) = \text{TRUE} \iff x \in L.$$

Proof. Let $M = (\Sigma, A, V, \text{start}, F, \delta)$ be a Turing Machine recognizing L that runs in polynomial time $p(n)$, i.e. on each input of length n its computation uses at most $p(n)$ steps. Now we apply Lemma 6.5.7 with $T = p(n)$ to obtain a circuit C_n for inputs to M of size n. C_n's size can be deduced as follows. There are $(T + 1)^2|A|$ nodes c_{ik}^t, $(T + 1)|V|$ nodes v_j^t, and $(T + 1)^2$ nodes h_k^t. The number of $\wedge$ nodes not yet counted is $T(T+1) \cdot |A| \cdot |V| + T(T+1)|A|$, the number of $\neg$ nodes is $T(T+1)|A|$ (actually $T(T+1)$ would suffice), and there is one more $\vee$ node; the number of edges is $6T(T+1) \cdot |A| \cdot |V| + 4T(T+1)|A| + |F|$, where F is the set of recognizing vertices (necessarily, $|F| \le |V|$). Thus $r(n) = O((p(n))^2 \cdot |A| \cdot |V|)$. As A and V are independent of n, $r(n) = O(p(n)^2)$, which is a polynomial in n. □

6.6 More NP Complete Problems & Cook's Theorem

Example 6.6.1. Circuit Value (CV).

Input: C, a circuit comprising *and*, *or*, and *not* gates, with Boolean inputs $x_1, x_2, \cdots, x_n$ and a Boolean output y. (The circuit can be viewed as a directed graph, where the inputs are the names of vertices with no in-edges, the output is the name of a vertex with no out-edge, and every other vertex is labeled by one of the operators (*and*, *or*, or *not*); the latter vertices have one or two inedges as appropriate, but there are no limits on the number of outedges they could have. We allow some of the inputs to be preset to the values TRUE or FALSE.

Question: Is there an assignment of Boolean values to the unset inputs that causes the circuit output to evaluate to TRUE?

In the example in Figure 6.9, setting $x_1 =$ TRUE and $x_2 =$ FALSE causes the circuit output to evaluate to TRUE.

Claim 6.6.2. *Given a polynomial time algorithm for* 3-SAT *there is a polynomial time algorithm for* Circuit Value (CV)*:* CV $\le_P$ 3-SAT.

Proof. Let C be the input to the CV algorithm, $\mathcal{A}_{\text{CV}}$. The algorithm proceeds as follows.

1. Compute E, where E is a Boolean formula with the property that

$$C \in \text{CV} \iff E \in \text{3-SAT}.$$

2. Run $\mathcal{A}_{\text{3-SAT}}(E)$.
3. Report the answer from (2).

$\mathcal{A}_{\text{CV}}$ constructs E using the same method as in Claim 6.4.9.

4. As argued in Claim 6.4.9, E can be satisfied exactly if C can be satisfied.
5. Clearly $\mathcal{A}_{\text{CV}}$ runs in polynomial time if $\mathcal{A}_{\text{3-SAT}}$ runs in polynomial time.

□

Theorem 6.6.3 (Cook's Theorem, 1972)**.** *CV is NP-Complete.*

Proof. Let $L \in \mathcal{NP}$. We need to show that $L \le_P$ CV. That is, given a polynomial time algorithm for CV, we give a polynomial time algorithm for L.

As $L \in \mathcal{NP}$, we know that L has a polynomial time verifier, V_L say. By Theorem 6.5.8, there is a polynomial r, a circuit family $C_1, C_2, \cdots$, recognizing the inputs to V_L, where $|C_m| = O(r(m))$, and m is the size of V_L's input measured in bits.

Let x be an input string to be tested for membership in L. Suppose that the n input bits for x are preset in circuit $C_{n+q(n)}$, where $q(n)$ is the polynomial bound on the bit-length of certificates used by V_L. This circuit can be simplified by computing the ouputs of all gates that are determined by these preset input bits, yielding a new circuit, $C(x, n)$ say. $C(x, n)$ has $q(n)$ unspecified inputs, corresponding to the bits for candidate certificates. So $C(x, n)$ is a legitimate input for the CV problem, and in addition, $x \in L$ exactly if $C(x, n) \in$ CV.

Thus the algorithm $\mathcal{A}_L$ to recognize L proceeds as follows.

1. $\mathcal{A}_L$ builds circuit $C(x, n)$.
2. $\mathcal{A}_L$ runs $\mathcal{A}_{\text{CV}}(C(x, n))$.
3. $\mathcal{A}_L$ reports the answer given by $\mathcal{A}_{\text{CV}}(C(x, n))$.

As already noted, this algorithm computes the correct output ($x \in L \iff C(x, n) \in$ CV). Further, it runs in polynomial time, for given x, $C(x, n)$ can be constructed in time polynomial in $|x|$. □

Corollary 6.6.4. GSAT, SAT, 3-SAT, VC, Clique, IS *are all NP-Complete.*

Proof. We have shown all these languages are in NP. Cook's Theorem, Lemma 6.5.1, and the individual reductions already shown yield the claimed result. □

Definition 6.6.5. *Language L is NP-hard if for all $K \in \mathcal{NP}$, $K \le_P L$.*

Lemma 6.6.6. *If J is NP-hard and $J \le_P L$, then L is NP-hard.*

Proof. This is proved in the same way as Lemma 6.5.3. □

6.7 Reduction Techniques

There are several techniques which arise repeatedly in making reductions from one NP-Complete problem to another, namely:

1. Generalization.
2. Restriction.
3. Local substitution.
4. Component construction.

The boundaries between one category and another are not always clearcut, but nonetheless they can be helpful in organizing the various constructions.

6.7.1 Generalization

Quite often, a problem one wants to show NP-Complete (NPC) is a generalization of a known NPC problem. For example, G-SAT is a generalization of SAT. Showing that the generalized problem is NP-hard is trivial: the identity reduction suffices. Or to put it another way, a polynomial time algorithm for the generalized problem is also a polynomial time algorithm for the more specialized problem.

6.7.2 Restriction

This is a complement to generalization: here one wants to show a restricted version of an NPC problem is also NPC. This is not always the case, so there is some work to do. One example is 3-SAT, which is a restricted version of SAT, which in turn is a restricted version of G-SAT. What is needed in the reduction is a way of encoding an instance of the general problem in the more restricted version. This is best seen by example.

Example 6.7.1. *Degree 4 Directed Hamiltonian Cycle* (4-DHC) is the Directed Hamiltonian Cycle problem on directed graphs where each vertex has degree (indegree plus outdegree) bounded by 4.

Claim 6.7.2. *Given a polynomial time algorithm for* 4-DHC *there is a polynomial time algorithm for* DHC (DHC $\le_P$ 4-DHC)*.*

Proof. Given a graph G, as input to $\mathcal{A}_{\text{DHC}}$, the main step (Step 1) is to construct a graph H, with degree bound 4, with the property that

$$G \in \text{DHC} \iff H \in \text{4-DHC}$$

and to do this in polynomial time.

Let d be the largest degree of any node in G, and suppose that $2^{h-1} < d \le 2^h$. H is the following graph. First, there is a copy of each vertex in G. However, the edges are replaced by a more elaborate construction. For each node v in G with inedges $e_1, e_2, \cdots, e_k$, H receives cycles $C_h, C_{h-1}, \cdots C_2$ of $2^h, 2^{h-1}, 2^{h-2}, \cdots, 4$ vertices respectively, as illustrated in Figures 6.12 and 6.13. Each edge $e_i = (u, v)$ is replaced by an edge $e_i' = (u', v')$; the new edges are paired, and each pair will be incident on distinct alternate vertices in C_h. In turn, the remaining 2^{h-1} vertices in C_h are incident, two by two, on alternate vertices in C_{h-1}. This construction repeats, down to C_2, whose two outedges go to v.

A symmetric construction is used for the outedges.

Clearly all the vertices in H have degree at most 4. Further H has at most $O(|E|+|V|)$ vertices, where $G = (E, V)$, so $|H| = O(|G|)$, and further H is readily constructed in time $O(|G|)$.

Next, we argue that $G \in \text{DHC} \iff H \in \text{4-DHC}$.

First, suppose that G has a Hamiltonian Cycle C. Suppose it uses some edge $e = (u, v)$. We describe the path taken in H corresponding to edge (u, v). It consists of the edge $e' = (u', v')$ preceded by a path from u to u' and followed by a path from v' to v. We describe the latter path in more detail. e' enters cycle C_h at vertex v'. The path in H then goes round all the vertices in C_h, leaving it at the 2^hth vertex visited, taking an edge into C_{h-1}. C_{h-1} is then traversed, being left at the 2^{h-1}th vertex to go to C_{h-2}, and so forth, until eventually C_2 is traversed and departed by an edge into v. This causes all the vertices on the cycles on the "into" side of v to be traversed; similarly, on leaving v, all the vertices on the cycles on v's "out" side will be traversed. Thus this process creates a Hamiltonian Cycle for H.

Second, suppose that H has a Hamiltonian Cycle D. Let $v_1, v_2, \cdots, v_n$ be the order in which the original vertices of G are visited by D. Then $v_1, v_2, \cdots, v_n$ form a Hamiltonian Cycle of G. This follows because once the path reaches the "in" vertices of v_i, it goes through v_i and then visits the "out" vertices of v_i before proceeding to v_{i+1}'s vertices. As v_i can be traversed only once, all

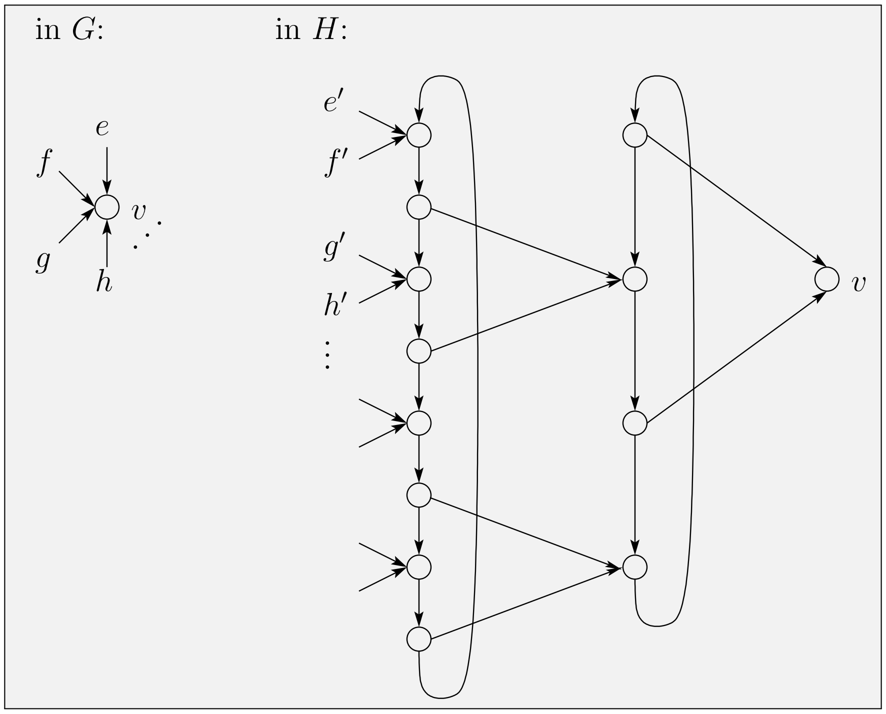


Figure 6.12: The Inedge Gadget for the 4-DHC Construction.

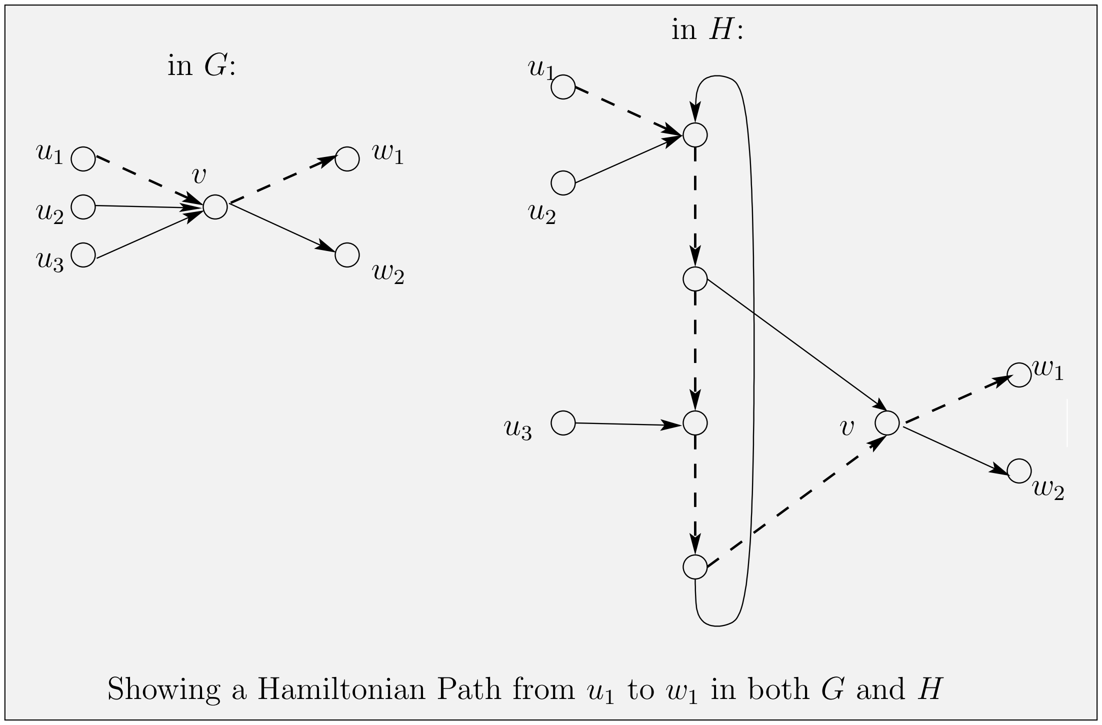


Figure 6.13: The Inedge Gadget for an Example Vertex.

its "in" and "out" vertices must be visited right before and right after v_i is visited. Thus the path from v_i to $v_{i+1 \bmod n}$ in D must go directly from the "out" vertices for v_i to the "in" vertices for $v_{i+1 \bmod n}$ and hence must use edge $(v'_i, v'_{i+1 \bmod n})$, which implies that $(v_i, v_{i+1 \bmod n})$ is an edge of G.

This demonstrates the claim that $G \in \text{DHC} \iff H \in \text{4-DHC}$. □

6.7.3 Local Replacement

In local replacement, each element of the original problem instance is replaced by a small structure in the new problem instance. We have already seen several examples of this, including the constructions showing DHP $\leq_P$ UHP and SAT $\leq_P$ 3-SAT. The latter is also an instance of restriction.

Example 6.7.3. *Dominating Set* (DS).
Input: (G, k), where $G = (V, E)$ is an undirected graph and $k \leq |V|$ is an integer.
Question: Does G have a dominating set $U \subseteq V$ of size k, that is a set U of vertices of size k, such that every vertex is either in U or adjacent to a vertex in U.
Note that this problem seems quite similar to the Vertex Cover problem, but it is a different question that is being asked.

Claim 6.7.4. *Given a polynomial time algorithm for* DS *there is a polynomial time algorithm for* VC.

Proof. Given an input (G, k) to $\mathcal{A}_{\text{VC}}$, the main step (Step 1) is to construct a pair (H, h) with the property that

$$(G, k) \in \text{VC} \iff (H, h) \in \text{DS}$$

and to do this in polynomial time.

H is obtained as follows. For each vertex v in G a copy, also called v, is put in H. For each edge $e = (u, v)$ in G, an "edge" vertex e is put in H; in addition, edges (u, e) and (v, e) are created in H. Finally, H receives two more vertices: y and z; y is connected to every vertex v, where v is a copy of a vertex in G, and z is connected only to y. h is set equal to $k + 1$. An example of graph H is shown in Figure 6.14.

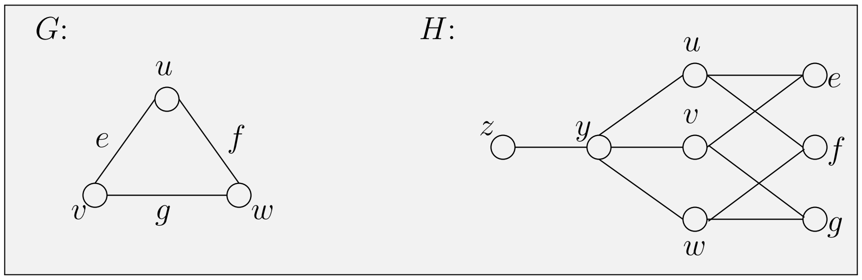


Figure 6.14: The Vertex Cover to Dominating Set Reduction; An Example Graph.

Clearly $|H| = O(|G|)$ and H can be constructed in $O(|G|)$ time.

Next, we argue that $(G, k) \in \text{VC} \iff (H, h) \in \text{DS}$.

First, suppose that G has a vertex cover U of size k. Then we claim that U plus the vertex y forms a Dominating Set for H. Since U is a vertex cover it covers every edge in G. Thus U covers all the "edge" vertices in H. y covers the remaining vertices in H.

Next, suppose that H has a dominating set U' of size h. We modify U' as follows. If $z \in U'$ then it is replaced by y; this ensures all the non-edge vertices in H are covered. Likewise, if any edge vertex $(u, v) = e \in U'$ then it is replaced by either one of u or v; e continues to be covered. Let U'' be the resulting set; U'' is also a dominating set for H and it has size at most h. Next, we note that $y \in U''$ as z is covered. Let $U = U'' - \{y\}$. So $|U| \leq h - 1 = k$. We claim that U is a vertex cover for G. This follows because in H every edge vertex is covered by U, and thus in G every edge is covered by U.

This demonstrates the claim that $(G, k) \in \text{VC} \iff (H, h) \in \text{DS}$. □

6.7.4 Component or Gadget Construction

In gadget construction, each element of the original problem instance is replaced by a possibly quite elaborate structure in the new problem instance. We have seen an example of such a reduction, namely 3-SAT $\leq_P$ IS; the next section gives another example. The separation of local replacement and gadget construction is not clearcut, but by the latter we intend constructions which are not just a modest tweak of the original input.

6.8 Examples of More Elaborate Reductions

6.8.1 Vertex Cover to Directed Hamiltonian Cycle

Claim 6.8.1. *Given a polynomial time algorithm for* Directed Hamiltonian Cycle (DHC) *there is a polynomial time algorithm for* Vertex Cover (VC)*:* VC $\leq_P$ DHC.

Proof. Let (G, k) be the input to the VC algorithm, $\mathcal{A}_{\text{VC}}$. The algorithm proceeds as follows.

1. Compute H, where H is a directed graph, with the property that

$$(G, k) \in \text{VC} \iff H \in \text{DHC}.$$

2. Run $\mathcal{A}_{\text{DHC}}(H)$.
3. Report the answer given by $\mathcal{A}_{\text{DHC}}(H)$.

Let $G = (V, E)$. $\mathcal{A}_{\text{VC}}$ obtains H as follows. For each edge $e = (u, v) \in E$, it adds four vertices to H, namely $(u, e, 0), (u, e, 1), (v, e, 0), (v, e, 1)$, connected as shown in Figure 6.16(a); this is called an *edge gadget.* Then, for each vertex $u \in V$, with incident edges $e_1, e_2, \cdots, e_l$, the corresponding vertices, two per edge, are connected as shown in Figure 6.16(b). This can be thought of as corresponding to u's adjacency list, except that there are two entries per edge; we call this sequence of $2l$ nodes the *adjacency list* for u (in H). Finally, k *selector* vertices $s_1, s_2, \cdots, s_k$ are added to H, together with edges from each s_i to the first vertex on each vertex u's "adjacency list" (i.e. edges $(s_i, (u, e_1, 0))$), and edges from the last vertex on each vertex u's adjacency list to each s_i (i.e. edges $((u, e_l, 1), s_i)$. This completes the description of graph H. An example of a pair $(G, 2)$ and the corresponding H are shown in Figure 6.15.

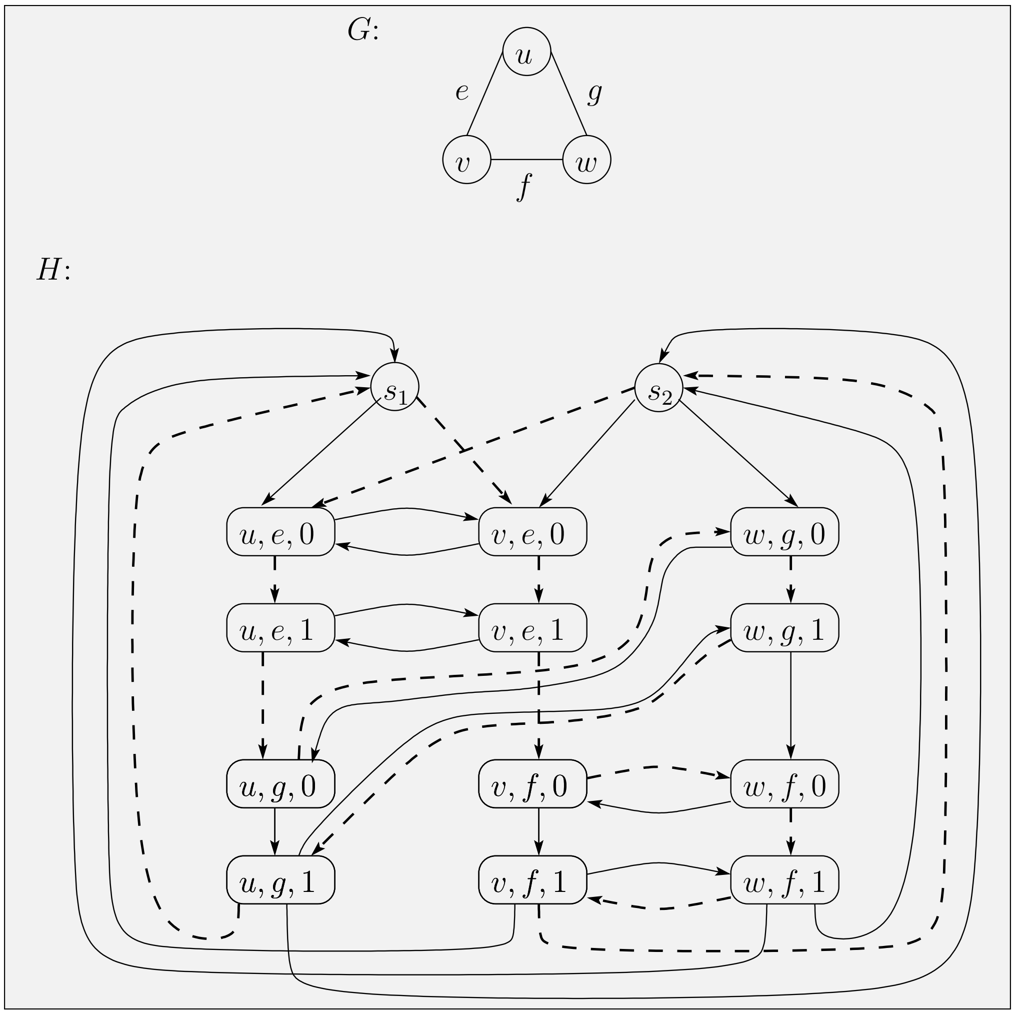


Figure 6.15: The graph H corresponding to $(G, 2)$. A Hamiltonian Cycle is shown with dashed edges.

4. We argue that H has a Hamiltonian Cycle exactly if G has a vertex cover of size k.

   First, suppose that G has a vertex cover of size k, $U = \{u_1, u_2, \cdots, u_k\}$ say. A preliminary, and incomplete Hamiltonian Cycle C' is given by the following cycle: It starts at s_1 and then goes through the adjacency list for vertex u_1, then goes to s_2, then traverses the adjacency list for u_2, and so on, eventually traversing the adjacency list for u_k, following which it completes the cycle by returning to s_1. C' needs to be modified to include the nodes on the adjacency lists of vertices outside the vertex cover U. The nodes in H which have not yet been traversed all correspond to the v end of edges (u, v) with just one endpoint, u, in U. (For as U forms a vertex cover, there is no edge with zero endpoints in U.) In the example in Figure 6.15, the vertices not on C' are $(w, g, 0)$, $(w, g, 1)$, $(w, f, 0)$, and $(w, f, 1)$. To include them, it suffices to replace the edge $((u, g, 0), (u, g, 1))$ with the sequence of three edges obtained by traversing $(u, g, 0), (w, g, 0), (w, g, 1), (u, g, 1)$ in that order, and the edge $((v, f, 0), (v, f, 1))$ with the sequence of three edges obtained by traversing $(v, f, 0), (w, f, 0), (w, f, 1), (v, f, 1)$ in that order. The result is still a cycle, but now it goes through every vertex of H exactly once, so it forms a Hamiltonian Cycle.

   Next, suppose that H has a Hamiltonian Cycle C. Let $(u_i, e_{u_i,1}, 0)$ be the vertex following s_i on C, for $1 \le i \le k$. Then we claim that $U = \{u_1, u_2, \cdots, u_k\}$ forms a vertex cover for G. To see this we need to understand how C traverses the adjacency list of each of the u_i.

   We claim that there are three ways for C to traverse the edge gadget for edge $e = (u, v)$ in G, as illustrated in Figure 6.17. For there are two vertices by which the gadget can be entered ($(u, e, 0)$ and $(v, e, 0)$), and two by which it can be exited ($(v, e, 1)$ and $(u, e, 1)$). Either it is entered twice and exited twice, in which case the corresponding entry and exit vertices are joined by single edges (Figure 6.17(a)) or it is entered and exited once, in which case all four vertices lie on the path between the corresponding entry and exit vertices (Figure 6.17(b) or (c)); the latter 4-vertex paths are called *zig-zags*.

   Let $e_1, e_2, \cdots, e_l$ be the edges incident on some $u = u_i \in U$, and suppose the edges appear in that order on u's adjacency list in H. As the first edge gadget on u's adjacency list in H is entered by edge $(s_i, (u, e_1, 0))$, C must go directly or by zig-zag from $(u, e_1, 0)$ to $(u, e_1, 1)$, then directly to $(u, e_2, 0)$, then directly or by zig-zag from $(u, e_2, 0)$ to $(u, e_2, 1)$, then directly to $(u, e_3, 0)$, and so on until it reaches $(u, e_l, 1)$, from where it goes to some $s_{i'}$, $i' \neq i$. By renumbering indices if needed, we can allow $i' = i + 1 \bmod k$.

   Thus C traverses k adjacency lists directly; any other nodes on C are reached by means zig-zags. These other nodes correspond to edge endpoints for edges with one endpoint in the traversed adjacency lists. This shows that for each such edge one endpoint is in U. As all nodes in H are traversed, it follows that every endpoint in a non-traversed adjacency list must have its other endpoint in U. Consequently U forms a vertex cover.

   This shows the claim.

5. Clearly $\mathcal{A}_{\text{VC}}$ runs in polynomial time if $\mathcal{A}_{\text{DHC}}$ runs in polynomial time.

□

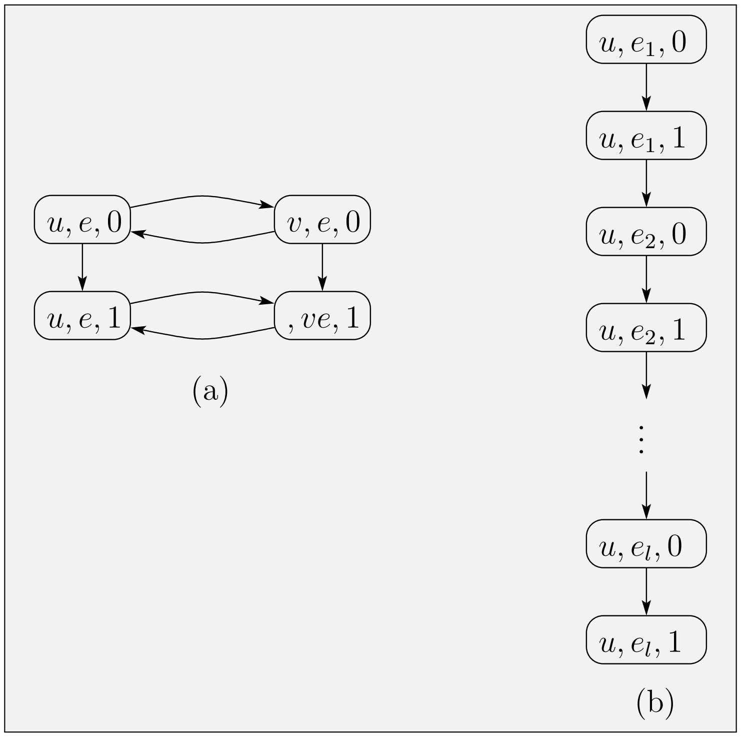


Figure 6.16: The structure in H due to edge (u, v) in G and to u's adjacency list for edges $e_1, e_2, \cdots, e_l$.

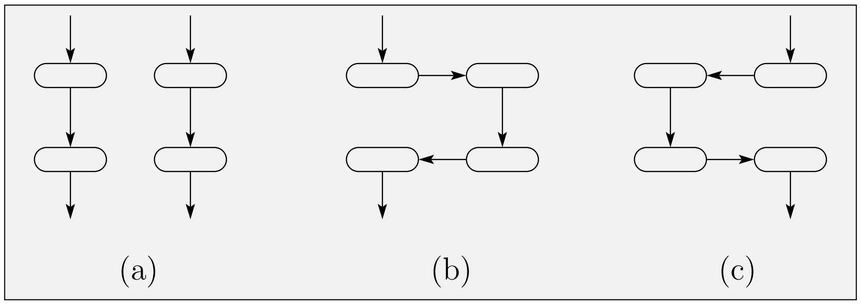


Figure 6.17: The Three Possible Ways of Traversing the Edge Gadget.

Exercises

1. Let Half-SAT be the following language:

$$\text{Half-SAT} = \{F \mid F \text{ is a CNF formula with } 2n \text{ variables and there is a satisfying assignment in which } n \text{ variables are set to } \textsc{True} \text{ and } n \text{ variables are set to } \textsc{False}\}.$$

Show that Half-SAT is NP-Complete, that is (i) show that Half-SAT has a polynomial time verifier, and (ii) Supposing that you were given a polynomial time algorithm for Half-SAT, use it to give a polynomial algorithm for SAT.

Sample Solution. i. The certificate comprises an assignment σ of truth values to the variables in F. Since $|\sigma| = O(|F|)$, the certificate has length linear in $|F|$.

To check that a candidate certificate is valid, the verifier checks (a) that F is a CNF formula with an even number of variables; (b) that the certificate has exactly n variables set to True and n set to False; (c) that the assignment of truth values in the certificate causes F to evaluate to True. It is straightforward to implement the verifier to run in polynomial time, and clearly it outputs "Yes" exactly when $F \in$ Half-SAT.

Finally, note that if there is a valid certificate then the input is in Half-SAT, and if the input $I \in$ Half-SAT, then there is a satisfying assignment σ with half the variables set to True and half set to False, and σ is a valid certificate.

ii. Let F' be the input to SAT. The algorithm $\mathcal{A}_{\text{SAT}}$ to test if $F' \in$ SAT builds a CNF formula F with the property that

$$F' \in \text{SAT} \iff F \in \text{Half-SAT}.$$

It then runs the algorithm for Half-SAT, $\mathcal{A}_{\text{H-SAT}}$, on input F and reports the answer as its own result.

F is built as follows. Suppose that F' has variables $x_1, x_2, \cdots, x_n$. F will have variables $x_1, x_2, \cdots, x_n$ and $y_1, y_2, \cdots, y_n$. The idea is that $x_i = \overline{y_i}$ for all i, $1 \le i \le n$. This is enforced by including clauses $(x_i \vee y_i) \wedge (\overline{x_i} \vee \overline{y_i})$ for $1 \le i \le n$. For $(x_i \vee y_i) \wedge (\overline{x_i} \vee \overline{y_i})$ evaluates to True only if $x_i =$ True and $\overline{y_i} =$ True or if $x_i =$ False and $\overline{y_i} =$ False, i.e. only if $x_i = \overline{y_i}$.

Clearly F can be built in polynomial time. Also, clearly, if $\mathcal{A}_{\text{H-SAT}}$ runs in polynomial time, then so does $\mathcal{A}_{\text{SAT}}$.

Now we argue that $F' \in \text{SAT} \iff F \in \text{Half-SAT}$.

It is readily seen that if F' has a satisfying assignment σ, setting truth values for the x_i in F according to σ, and then for the y_i according to the rule $y_i = \overline{x_i}$ produces a satisfying assignment for F with exactly n variables set to True. Thus $F' \in$ SAT implies $F \in$ Half-SAT.

Likewise, a satisfying assignment for F restricted to the x variables is a satisfying assignment for F'. Thus $F \in$ Half-SAT implies $F' \in$ SAT.

This shows that $F' \in \text{SAT} \iff F \in \text{Half-SAT}$.

2. Let Non Tautology be the following problem.
   Input: A DNF formula F (i.e., a boolean formula which is an "or" of clauses, where each clause is an "and" of boolean variables and their complements).
   Question: Does F have a non-satisfying assignment, that is an assignment of truth values to its Boolean variables that causes the formula to evaluate to FALSE?
   e.g. $F_1 = (x_1) \vee (x_2)$ has the non-satisfying assignment $x_1 =$ FALSE, $x_2 =$ FALSE; $F_2 = (x_1 \wedge x_2) \vee (\overline{x}_1 \wedge \overline{x}_2)$ has the non-satisfying assignment $x_1 =$ FALSE, $x_2 =$ TRUE.

   Show that Non Tautology has a polynomial time verifier. That is, given a candidate certificate C of size polynomial in n, where n is the input size, there is a polynomial time algorithm to check whether C certifies that the input F is in the language Non Tautology, and furthermore, for each $F \in$ Non Tautology, there is a polynomial-sized certificate C for F.

3. Let Subset Sum be the following problem.
   Input: A collection of n not necessarily distinct positive integers and a target integer t, where the integers are given in binary form.
   Question: Is there a subset of the collection, such that the numbers in the subset sum to exactly t?
   e.g. Let the collection of integers be $\{1, 2, 2, 6\}$ and the target be 5. The subset adding up to the target is 1,2,2.

   Show that Subset Sum has a polynomial time verifier.

4. The Traveling Peddler (TP) is the following problem.
   Input: A directed graph $G = (V, E)$, where each edge has a non-negative integer length, and an integer b.
   Question: Does G have a Hamiltonian Cycle of length at most b?

   Show that Traveling Peddler has a polynomial time verifier.
   Note: This problem is often called the Traveling Salesman problem.

5. Let Composites be the following problem.
   Input: An n-bit integer m.
   Question: Is m a composite, that is a non-trivial product of two integers; in other words are there integers r and s, with $r, s \neq 1$, where $m = rs$?

   Show that Composites has a polynomial time verifier.
   Comment. Interestingly, there is a polynomial time algorithm (called a *primality test*) to test this property; however, this algorithm does this without identifying a pair r and s of divisors for m.

6. Suppose that you were given a polynomial time algorithm for Directed Hamiltonian Path (DHP). Using it as a subroutine, give a polynomial time algorithm for Undirected Hamiltonian Path (UHP). (Claim 6.4.2 demonstrates the reduction in the opposite direction).

7. Suppose that you were given a polynomial time algorithm for Undirected Hamiltonian Path (UHP). Using it as a subroutine, give a polynomial time algorithm for Undirected Hamiltonian Cycle (UHC).

8. Suppose that you were given a polynomial time algorithm for Traveling Peddler. Using it as a subroutine, give a polynomial time algorithm for Directed Hamiltonian Cycle. See Problem 4 for the definition of the Traveling Peddler problem.

9. Let k-Color be the following problem.
   Input: An undirected graph G.
   Question: Can the vertices of G be colored using k distinct colors, so that every pair of adjacent vertices are colored differently?

   Suppose that you were given a polynomial time algorithm for $(k+1)$-Color. Use it to give a polynomial algorithm for k-Color. This means that you need to provide a polynomial time algorithm that on input G, a graph, constructs another graph H, with the property that $G \in k\text{-Color} \iff H \in (k+1)\text{-Color}$.

10. Let Equal Subset Sum be the following problem.
    Input: A collection of n not necessarily distinct positive integers, $a_1, a_2, \cdots, a_n$, given in binary form.
    Question: Is there a subset of the collection, such that the numbers in the subset sum to exactly $\frac{1}{2}\sum_{i=1}^{n} a_i$?

    Suppose that you were given a polynomial time algorithm for Equal Subset Sum. Use it as a subroutine to give a polynomial time algorithm for Subset Sum. See Problem 3 for the definition of the Subset Sum problem.

11. Let Half-Clique be the following problem.
    Input: An undirected graph $H = (W, F)$, with $n = |W|$ being an even integer.
    Question: Does H have a size $n/2$ clique?

    Suppose that you were given a polynomial time algorithm for Half-Clique. Using it as a subroutine, give a polynomial time algorithm for Clique. This means that you need to provide a polynomial time algorithm that on input (G, k), G an undirected graph and k an integer, constructs another graph H, with the property that $(G, k) \in \text{Clique} \iff H \in \text{Half-Clique}$.

12. Let 2-Satisfying Assignments be the following problem.
    Input: A CNF formula F (i.e., a Boolean formula which is an "and" of clauses, where each clause is an "or" of Boolean variables and their complements).
    Question: Does F have two distinct satisfying assignments?
    e.g. $F_1 = x_1 \vee x_2$ has the satisfying assignments $x_1 = \text{True}$, $x_2 = \text{False}$ and $x_1 = \text{False}$, $x_2 = \text{True}$; $F_2 = (x_1 \vee x_2) \wedge (\overline{x}_1)$ has just one satisfying assignment, namely $x_1 = \text{False}$, $x_2 = \text{True}$.

    Suppose that you were given a polynomial time algorithm for 2-Satisfying Assignment. Using it as a subroutine, give a polynomial time algorithm for SAT. This means that you need to provide a polynomial time algorithm that on input F, a Boolean CNF formula, constructs a Boolean CNF formula $\widetilde{F}$ such that

    $$F \in \text{SAT} \iff \widetilde{F} \in \text{2-Satisfying Assignments}$$

    i.e. F is satisfiable if and only if $\widetilde{F}$ has at least 2 distinct satisfying assignments.

13. Let Kernel be the following problem.
    Input: A directed graph $G = (V, E)$.
    Question: Does G have a kernel? A kernel is a subset K of vertices such that no edge joins two vertices in the kernel and for every vertex $v \in V - K$ there is a vertex $u \in K$ such that $(u, v) \in E$.

    Suppose that you were given a polynomial time algorithm for Kernel. Using it as a subroutine, give a polynomial time algorithm for 3-SAT.
    Hint: Create a directed triangle for each clause with one vertex per literal (even if the clause has fewer than 3 literals), and a directed cycle for each variable, with one vertex for each truth assignment. Connect the vertices in the cycles to the vertices in the triangles appropriately. If F is satisfiable the kernel will contain one vertex from each triangle and one vertex from each cycle.

14. Suppose that you were given a polynomial time algorithm for Satisfiability, that is an algorithm that reports whether or not the input is a satisfiable CNF formula.

    Use it to give a polynomial time algorithm to find a satisfying assignment σ if the formula is satisfiable.
    Hint. Let $x_1, x_2, \cdots, x_m$ be the variables in the formula. Consider an algorithm that proceeds as follows: First, determine if F is satisfiable. If so, construct a satisfying assignment σ in n iterations, with each iteration determining the truth value for one more variable. The first iteration determines if there is a satisfying assignment with $x_1 = \text{TRUE}$; if so it sets $x_1 = \text{TRUE}$ and otherwise it sets $x_1 = \text{FALSE}$. Be careful as to how you continue, for note that even if each of $x_1 = \text{TRUE}$ and $x_2 = \text{TRUE}$ occur in satisfying assignments, it need not be the case that they both occur in a single satisfying assignment (e.g. $F = (x_1 \vee x_2) \wedge (\overline{x}_1 \vee \overline{x}_2)$).

15. Let 3-Color be the following problem.
    Input: A undirected graph $G = (V, E)$.
    Question: Does G have a 3-coloring, that is can the vertices of G be colored using 3 colors in such a way that every pair of adjacent vertices have distinct colors.

    Suppose that you were given a polynomial time algorithm for 3-Color. Using it as a subroutine, give a polynomial time algorithm for 3-SAT.

    To do this you need to provide a polynomial time algorithm that on input F, a Boolean 3-CNF formula, constructs a graph G such that F is satisfiable if and only if G is 3-colorable.

    G will contain one instance of the Color Defining Gadget (Definer for short) shown in Figure 6.18 (a triangle). The colors given to the Definer (T, F and N) are viewed as corresponding to TRUE, FALSE, and NEUTRAL. For each variable in F, G will contain a pair of vertices in a Variable gadget connected to the Definer as shown. Supposing that the Definer vertices are colored T, F, N, as shown, what colors do the two vertices in the Variable gadget corresponding to the literals x and $\overline{x}$ take on?

    Now consider the OR gadget. Suppose that its bottom two nodes are both colored T; what color(s) can its top node take on? What if they are both F? What if one is T and one is F? Use two OR gadgets to simulate the evaluation of a clause. By connecting the top node of the combined two OR gadgets to the Definer, and identifying the bottom nodes with the nodes in the Variable gadgets, make the OR-gadgets used for the clause 3-colorable exactly if the clause

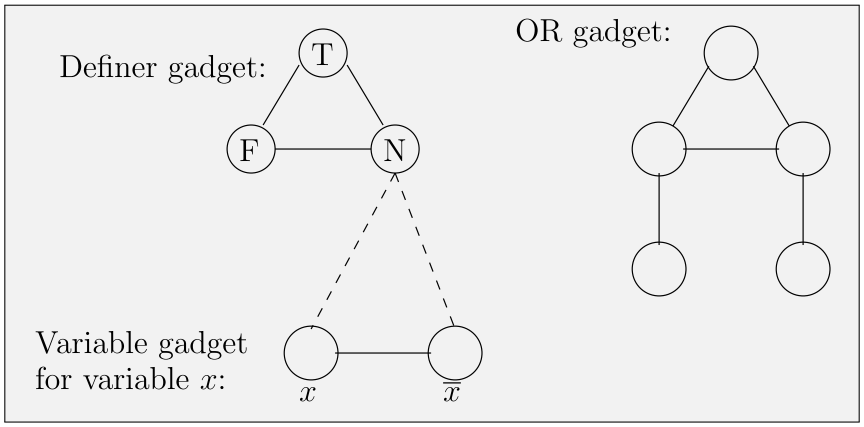


Figure 6.18: The Gadget for problem 15.

evaluates to TRUE on the corresponding truth setting for the Boolean variables. Remember to handle clauses of two and one literals also.

16. Suppose that you were given a polynomial time algorithm for G-SAT. Use it to give a polynomial algorithm for Clique.

    That is, given a graph G and an integer k, the task is to give an algorithm to determine if G has a clique of size k. To this end, your algorithm will construct a Boolean formula $F_{G,k}$ such that $F_{G,k}$ is satisfiable if and only if G has a clique of size k. The main task is to describe how to construct $F_{G,k}$ in polynomial time. Broad-brush, this construction is analogous to that of Claim 6.5.7.

17. Let No-Tri-VC be the vertex cover problem limited to undirected graphs which have no triangles (collections of 3 vertices u, v, w with all three edges $(u, v), (v, w), (w, v)$ present).

    Suppose that you were given a polynomial time algorithm for No-Tri-VC. Use it to give a polynomial algorithm for Vertex Cover. This means that you need to provide a polynomial time algorithm that on input (G, k), where G is an undirected graph and $k \geq 0$ is an integer, constructs an undirected triangle-free graph H and an integer l, with the property that $(G, k) \in$ VC $\iff$ $(H, l) \in$ No-Tri-VC.

18. The Timetable Problem. The input has several parts: an integer t, a list $F_1, F_2, \cdots, F_k$ of k final exams to schedule, a list of students $S_1, S_2, \cdots, S_n$; in addition, each student is taking some subset of exams, specified in a list SL_i for student i, $1 \leq i \leq n$. The task is to schedule the exams so that a student is scheduled for at most one exam in any given time slot. The problem is to determine if there is a schedule using only t time slots.

    Show that the Timetable Problem is NP-Complete.

    Hint: Try reducing 3-Color to the Timetable Problem (see Problem 15). That is, you have the following task. Given a graph $G = (V, E)$, an input to the 3-Color problem, you need to create an input T for the Timetable problem, such that G is 3-colorable if and only if T is schedulable. The main issues in the construction are to decide what in the timetable corresponds to a vertex of G and what corresponds to an edge. Don't forget to choose a suitable value for t.