Fundamentals of Computer Science
Fundamenten voor de Computerwetenschappen

$LastChangedDate$: $2013-02-0614:26:48+0100(Wed, 06Feb2013)$
Revision: Rev : 6968

Dave Clarke

January 30, 2014

Contents

Part I

Introduction

Chapter 1

Introduction

Aim

The aim of the course is to introduce students to the study of some of the formal aspects of computer science. First and foremost, we will study formal languages and their recognisers. Formal languages consist of collections of finite sequences of symbols (known as strings or words) from a finite set. Arguably all problems of computer science can be defined as a suitable formal language and solving such a problem reduces to recognising whether a given word is in the language or not. The recognisers we will study consist of finite state automata, pushdown automata, and Turing machines. As all algorithmic problems can be seen as language recognition problems, issues of complexity and computability will be studied in this context. In addition, students will be introduced to the language, methods, and terminology of graph theory, and acquire an understanding the different classes of graphs and of modelling using graphs. Students will appreciate the applicability of graph theory in other areas of computer science, such as networking and optimisation.

Objectives

At the end of the course the student should be able to:

- Determine the language recognised by a given finite state machine; Convert between finite state machines and regular expressions; Minimise a finite state machine. Determinise a non-deterministic finite state machine.
- Prove that certain problems are NP-complete or uncomputable.
- Describe major complexity classes and understand their practical consequences.
- Model problems using graphs and apply graph algorithms appropriately.
- Prove properties about automata, Turing machines, graphs, and their algorithms.
- Manipulate and prove properties of previously unseen definitions of related concepts.

Expected Audience

These notes are aimed at masters students of computer science and engineering who have some experience with discrete mathematics, though not necessarily a solid grounding in theoretical computer science.

Writing these notes is necessarily a balancing act: selecting enough material to give a decent but short introduction to any given area, selecting the right depth of material with as few assumptions as possible, and presenting a challenging enough collection of topics.

Acknowledgements

These notes borrowed heavily from *Automata and Computability* by Dexter C. Kozen, Springer, 1997; *Introduction to the Theory of Computation* 2nd Edition, Michael Sipser, Course Technology, Cengage Learning, 2006; *Elements of the Theory of Computation*, Harry R. Lewis and Christos H. Papadimitriou, Prentice Hall, 1981; *Automata, Computability, and Complexity: Theory and Applications*, Elaine Rich, Pearson Prentice Hall, 2008; *Computability and Logic* 5th Edition, George S. Boolos, John P. Burgess, Richard C. Jeffrey, Cambridge University Press, 2007; *Discrete Mathematics* 3rd Edition, Richard Johnsonbaugh, MacMillen, 1993; *Graph Theory: Modeling, Applications, and Algorithms*, Geir Agnarsson, Raymond Greenlaw, Pearson International, 2007; and *Graph Theory: An Advanced Course*, Adrian Bondy, U.S.R. Murty, Springer; 3rd Corrected Printing, 2007, along with numerous scientific articles, cited throughout the text.

These notes are only a beginning. Topics fundamental for the foundations of computer science not covered include logic, semantics, lattice theory, domain theory, category theory, and the lambda calculus. Some of this material can be found in other courses.

Chapter 2

Mathematical Preliminaries

Here we give some mathematical concepts that will be used throughout these notes. Many of these concepts should already be familiar; in this case, this section serves to establish the notation used throughout.

2.1 The Basics

2.1.1 Sets

A *set* is a collection of objects. An object in a set is called an *element* of the set. The unique set containing no elements is called the *empty set* and is denoted by $\emptyset$. Let A and B be sets. If a is an element contained in a set A, we denote this by $a \in A$. If for every $a \in A$, it is also true that $a \in B$, we say that A is a *subset* of B and denote this by $A \subseteq B$. Two sets A and B are *equal*, denoted by $A = B$, when each element in A is also an element in B and vice versa—that is, we have both $A \subseteq B$ and $B \subseteq A$.

We have the following set operations: The *union* of A and B, denoted by $A \cup B$, is the set of those elements that are either in A or in B. The *intersection* of A and B, denoted by $A \cap B$, is the set consisting of those elements that are contained in both A and B. If $A \cap B = \emptyset$, then we say that A and B are *disjoint*. The *difference* of A and B, denoted by $A \setminus B$, consists of those elements that are contained in A and are not in B. The *symmetric difference* of A and B, denoted by $A \triangle B$, is equal to $(A \setminus B) \cup (B \setminus A)$. It consists of those elements that are contained in exactly one of the sets A and B. The notion of union, intersection, and symmetric difference can be generalised to a collection of finitely many sets rather than just two; that is, one can show that if $A_1, A_2, \ldots, A_n$ is a finite collection of sets, then the notions of $A_1 \cup \cdots \cup A_n$, $A_1 \cap \cdots \cap A_n$, and $A_1 \triangle \cdots \triangle A_n$ are all unambiguous and sensible, since the operations $\cup$, $\cap$, and $\triangle$ are all associative. The last expression denotes the set of elements that are contained in an odd number of the sets $A_1, \ldots, A_n$.

A set S containing finitely many elements is called a *finite set*. The number of elements in the set S is denoted by $|S|$ and is often called the *cardinality* of S. For example, the set $\{0, 1, 2, 3, 4\}$ has five elements and hence $|\{0, 1, 2, 3, 4\}| = 5$. If S is infinite, that is, if S is not finite, then we write $|S| = \infty$. The set of natural numbers, $\mathbb{N} = \{1, 2, 3, \ldots\}$, is an infinite set. If S is a set, then the set of all subsets of S, denoted by $\mathscr{P}(S)$, is called the *power set* of S. For example, if $S = \{1, 2, 3\}$, then

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

2.1.2 Tuples

Let $k \in \mathbb{N}$ and $a_1, a_2, \ldots, a_k$ be any k objects. Then $(a_1, a_2, \ldots, a_k)$ is an *ordered k-tuple* or just k-tuple for short. For each $i \in \{1, 2, \ldots, k\}$, we say that a_i is the ith *component* of $(a_1, a_2, \ldots, a_k)$. If $A_1, \ldots, A_k$ are sets, then their *Cartesian product*, denoted by $A_1 \times \cdots \times A_k$, is the set of all the k-tuples $(a_1, \ldots, a_k)$, where $a_i \in A_i$, for each i.

2.1.3 Induction

We will often use *induction* to prove theorems and statements about automata, graphs, etc. If $n \in \mathbb{N}$ is a natural number and $P(n)$ is a statement involving n as the only unknown parameter, the truth value of $P(n)$ depends only on n. The general concept of induction states that to prove that $P(n)$ is true for all $n \geq 0$, it suffices to prove the following two statements:

1. $P(0)$ is true.
2. For every $n \geq 0$, if $P(k)$ is true for each $k \in \{0, 1, \ldots n\}$, then so is $P(n+1)$.

In most cases, a simplified version of induction can be used. Namely, to prove that $P(n)$ is true for each $n \geq 0$, we prove the following statements:

1. $P(0)$ is true.
2. For every $n \geq 0$, if $P(n)$ is true, then so is $P(n+1)$.

2.1.4 Functions

For a function (also called a mapping) $f : X \to Y$, the set X is called the *domain* of f and the set Y is called the *codomain* of f. The set $f(X) = \{f(x) \mid x \in X\} \subseteq Y$ is called the *range* of f. The function $f : X \to Y$ is *injective* or *one-to-one*, if $f(x_1) = f(x_2)$ always implies that $x_1 = x_2$. Also, f is *surjective* or *onto* if the range equals the codomain $f(X) = Y$—that is, if for every $y \in Y$, there is an $x \in X$ such that $y = f(x)$. A map that is both injective and surjective is called *bijective*. Such functions are also called *bijections* or *one-to-one correspondences*.

The map from X to X that has a value of x for each $x \in X$ is called the *identity map*. The identity map of X is denoted by $\iota_X : X \to X$. The function ι_X is bijective for any set X. Let $f : X \to Y$ and $g : Y \to Z$ be any two maps. The *composition* of g with f is denoted by $g \circ f : X \to Z$ is the mapping such that $(g \circ f)(x) = g(f(x))$ for each $x \in X$. If $f : X \to Y$ is a bijective map, then there is a unique map, denoted by $f^{-1} : Y \to X$, such that $f^{-1} \circ f = \iota_X$ and $f \circ f^{-1} = \iota_Y$. The map f^{-1} is called the *inverse* of the map f. Finally, if $f : X \to Y$ is a map and $X' \subseteq X$, then the *restriction* of f to X' is the map $f_{X'} : X' \to Y$ defined by $f_{X'}(x) = f(x)$ for each $x \in X'$.

For a function $f : A \to B$, where both A and B are finite, we have the following:

1. If f is injective, then $|A| \leq |B|$.
2. If f is surjective, then $|A| \geq |B|$.
3. If $|A| = |B|$, then f is injective if and only if f is surjective.

2.1.5 Pigeon-Hole Principle

Note the contrapositive of the first statement above: if $|A| > |B|$, then f is *not* injective. This forms a powerful principle called the *Pigeon-Hole Principle*. It can be stated in the following form: "If m pigeons are put into n holes, where $m > n$, then one hole must contain at least two pigeons."

2.1.6 Relations

A (binary) *relation* R between two sets A and B is a set of tuples of elements of A and B. Thus $R \subseteq A \times B$. We write $(a,b) \in R$ or aRb to indicate that a is related to b by R.

Relations can be generalised to relate 3 or more sets. Generally we will consider binary relations over some set A. A relation $R \subseteq A \times A$ is *reflexive* iff for all $a \in A$ we have $(a,a) \in R$. A relation $R \subseteq A \times A$ is *symmetric* iff for all $(a,b) \in R$ we also have $(b,a) \in R$. A relation $R \subseteq A \times A$ is *transitive* iff if $(a,b) \in R$ and $(b,c) \in R$, then $(a,c) \in R$.

The *transitive closure* R^+ of relation $R \subseteq A \times A$ is the smallest transitive relation containing R. This can be calculated as follows:

- $R^1 = R$.
- $R^i = R^{i-1} \cup \{(s_1,s_3) \mid \exists s_2 \cdot (s_1,s_2) \in R^{i-1} \wedge (s_2,s_3) \in R^{i-1}\}$.
- $R^* = \bigcup_{i \geq 1} R^i$.

The *reflexive, transitive closure* R^* of relation $R \subseteq A \times A$ is the smallest reflexive, transitive relation containing R. This can be calculated as follows:

- $R^0 = \{(a,a) \mid a \in A\}$.
- $R^i = R^{i-1} \cup \{(s_1,s_3) \mid \exists s_2 \cdot (s_1,s_2) \in R^{i-1} \wedge (s_2,s_3) \in R^{i-1}\}$.
- $R^+ = \bigcup_{i \geq 0} R^i$.

A relation $R \subseteq A \times A$ is an *equivalence relation* if it is reflexive, symmetric and transitive. Equivalence relations are often denoted $\sim$, $\equiv$, or $\approx$. Given a set A and an equivalence relation $\sim \subseteq A \times A$, the *equivalence class* of element $a \in A$ is the subset of all elements of A that are equivalent to a:

$$[a] \widehat{=} \{b \in A \mid a \sim b\}.$$

The equivalences classes of $\sim$ form a partition of the set A, meaning that they are disjoint and their union is A. Note that if $b \in [a]$, then $[b] = [a]$. The set of all equivalence classes in X given an equivalence relation $\sim$ is usually denoted as $X/\sim$ and called the *quotient set* of X by $\sim$. This operation can be thought of (very informally) as the act of "dividing" the input set by the equivalence relation, hence both the name "quotient" and the notation. The quotient set is to be thought of as the set X with all the equivalent points identified.

2.1.7 Matrices

An matrix with n rows and m columns, or an $n \times m$-matrix, or n-by-m-matrix, with elements a_{ij} for $i \in 1..n$ and $j \in 1..m$ is denoted (a_{ij}). A vector is an $n \times 1$-matrix.

Addition of two $n \times m$-matrices is performed elementwise. That is, if

$$(a_{ij}) + (b_{ij}) = (a_{ij} + b_{ij}).$$

For example,

$$\begin{bmatrix} 4 & 12 \\ 2 & -1 \end{bmatrix} + \begin{bmatrix} 3 & -3 \\ 2 & 0 \end{bmatrix} = \begin{bmatrix} 7 & 9 \\ 4 & -1 \end{bmatrix}$$

Multiplication of an $n \times p$-matrix by an $p \times m$-matrix results in an $n \times m$-matrix and is defined as

$$(a_{ij})(b_{ij}) = \left(\sum_{k=1}^{p} a_{ik} b_{kj} \right).$$

For example,

$$\begin{bmatrix} 4 & 12 \\ 2 & -1 \\ 6 & 7 \end{bmatrix} \begin{bmatrix} 3 & -1 \\ 2 & 2 \end{bmatrix} = \begin{bmatrix} 4 \times 3 + 12 \times 2 & 4 \times (-1) + 12 \times 2 \\ 2 \times 3 + (-1) \times 2 & 2 \times (-1) + (-1) \times 2 \\ 6 \times 3 + 7 \times 2 & 6 \times (-1) + 7 \times 2 \end{bmatrix} = \begin{bmatrix} 36 & 20 \\ 4 & -4 \\ 32 & 8 \end{bmatrix}$$

Matrices are typically defined over integers $\mathbb{Z}$, rational $\mathbb{Q}$, reals $\mathbb{R}$, or complex numbers $\mathbb{C}$, though the definitions apply to any field (or semiring, if inverses are not considered).

2.2 Analysis of Algorithms

Analysis of algorithms refers to the process of deriving estimates for the time and space needed to execute an algorithm.

Suppose we had an algorithm to examine all subsets of a given set X of n elements. As the number of subsets is 2^n, the program would need at least 2^n time units to execute. It does not matter what the units of time are, 2^n grows so fast as n increases that, except for small values of n, it would be infeasible to run the program.

Algorithms are analysed in terms of the *size* of their inputs. We can ask what the minimum time needed to execute an algorithm is for all inputs of size n—this is called the *best-case time* for inputs of size n. We can also ask what the maximum time needed to execute the algorithm for inputs of size n—this is called the *worst-case time*. Another important measure is the *average-case time*—the average time needed to execute the algorithm over some finite set of inputs all of size n.

Generally we are not interested in precise instruction counts, but want to find more coarse measures, such as the number of comparisons performed, if we are interested in sorting algorithms. We generally want to know how the best, worst or average time required for an algorithm grows as the size of the input increases.

For example, if the worst-case time of an algorithm is

$$t(n) = 60n^2 + 5n + 1$$

for input of size n. For large n, $t(n)$ grows like $60n^2$, so the other factors can be ignored. But because we are not actually concerned with the time unit, we can ignore the constant factor 60 (we could eliminate it by changing from seconds to minutes). Thus we would consider the worst case complexity of the algorithm as being of *order* n^2, as $T(n)$ grows like n^2 as n increases, and we write

$$t(n) = \Theta(n^2).$$

Definition 2.2.1 (Big oh notation, omega notation, theta notation) *Let f and g be functions with domain $\mathbb{N}^+$ (that is, $1,2,3,\ldots$).*

Write

$$f(n) = O(g(n))$$

and say that $f(n)$ is of order at most *$g(n)$ if there exists a positive constant C_1 such that*

$$|f(n)| \leq C_1|g(n)|$$

for all but finitely many positive integers n.

Write

$$f(n) = \Omega(g(n))$$

and say that $f(n)$ is of order at least *$g(n)$ if there exists a positive constant C_2 such that*

$$|f(n)| \geq C_2|g(n)|$$

for all but finitely many positive integers n.

Write

$$f(n) = \Theta(g(n))$$

and say that $f(n)$ is of order *$g(n)$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.*

In words, $f(n) = O(g(n))$ states that f is bounded above by g, $f(n) = \Omega(g(n))$ states that f is bounded below by g, and $f(n) = \Theta(g(n))$ states that f is bounded above and below by g.

Example 2.2.2 *Since,*

$$60n^2 + 5n + 1 \leq 60n^2 + 5n^2 + n^2 = 66n^2 \qquad \text{for } n \geq 1,$$

we may take $C_1 = 66$ to obtain

$$60n^2 + 5n + 1 = O(n^2).$$

Since,

$$60n^2 + 5n + 1 \geq 60n^2 \qquad \text{for } n \geq 1,$$

we may take $C_2 = 60$ to obtain

$$60n^2 + 5n + 1 = \Omega(n^2).$$

Since $60n^2 + 5n + 1 = O(n^2)$ and $60n^2 + 5n + 1 = \Omega(n^2)$, we conclude

$$60n^2 + 5n + 1 = \Theta(n^2).$$

Note that any polynomial in n of degree k with nonnegative coefficients is $\Theta(n^k)$.
Other examples include:

- $2n + 3\log_2 n = \Theta(n)$.
- $1 + 2 + \cdots + n \leq n + n + \cdots + n = n^2$, hence $1 + 2 + \cdots + n = O(n^2)$. Now $1 + 2 + \cdots + n \geq \lceil (n+1)/2 \rceil \lceil n/2 \rceil \geq (n/2)(n/2) = n^2/4$. Thus $1 + 2 + \cdots + n = \Omega(n^2)$. Therefore $1 + 2 + \cdots + n = \Theta(n^2)$.
- More generally, $1^k + 2^k + \cdots + n^k = \Theta(n^{k+1})$.

- $\log_2 n! = \Theta(n \log_2 n)$. Start by reasoning that $\log_2 n! = \log_2(n(n-1)(n-2)\cdots 1) = \log_2 n + \log_2(n-1) + \log_2(n-2) + \cdots + \log_2 1$.

Definition 2.2.3 (Best-case, worse-case, average-case time) *If an algorithm requires $t(n)$ units of time to terminate in the best case for an input of size n and*

$$t(n) = O(g(n))$$

we say that the best-case time required by the algorithm is $O(g(n))$.

If an algorithm requires $t(n)$ units of time to terminate in the worst case for an input of size n and

$$t(n) = O(g(n))$$

we say that the worst-case time required by the algorithm is $O(g(n))$.

If an algorithm requires $t(n)$ units of time to terminate in the average case for an input of size n and

$$t(n) = O(g(n))$$

we say that the average-case time required by the algorithm is $O(g(n))$.

We can also replace O by Ω or Θ so that we can have upper bounds and precise descriptions of the best-case, worst-case, and average-case time of an algorithm.

Example 2.2.4 *Consider the following loops*

```
1.  for i = 1 to n do
2.    for j = 1 to i do
3.      x = x + 1
```

The total number of times line 3 is executes is:

$$1 + 2 + \cdots + n = \Theta(n^2).$$

Example 2.2.5 *Consider the following loops:*

```
1.  j = n
2.  while j >= 1 do
3.    begin
4.      for i = 1 to j do
5.        x = x + 1
6.      j = j/2
7.    end
```

Let $t(n)$ denote the number of times we execute `x = x + 1`*. The first time we arrive at the body of the while loop, the statement* `x = x + 1` *is executed n times. Therefore $t(n) \geq n$ and $t(n) = \Omega(n)$.*

Now for the big oh notation for $t(n)$. The first time through the while loop `x = x + 1` *is executed n times. In line 6 j is replaced with $j/2$, which means* `x = x + 1` *will be executed $n/2$ additional times in the next iteration of the loop and so on. If we let k denote the number of times we execute the body of the while loop, then the number of times we execute* `x = x + 1` *is at most:*

$$n + \frac{n}{2} + \frac{n}{4} + \cdots + \frac{n}{2^{k-1}}.$$

This is equal to[1]

$$\frac{n\left(1-\frac{1}{2^k}\right)}{1-\frac{1}{2}}.$$

Now

$$t(n) \leq \frac{n\left(1-\frac{1}{2^k}\right)}{1-\frac{1}{2}} = 2n\left(1-\frac{1}{2^k}\right) \leq 2n,$$

so $t(n) = O(n)$*. Thus a theta notation for the number of times we execute* `x = x + 1` *is* $\Theta(n)$.

Observe that in the previous example we never actually provided the value of $t(n)$.

Example 2.2.6 (Searching an Unordered Sequence) *Given the sequence*

$$s_1, s_2, \ldots, s_n,$$

and a value key*, this algorithm finds the location of* key*. If* key *is not found, the algorithm outputs a* 0*.*

```
1.  for i = 1 to n do
2.    if key == s(i) then
3.      return i  // successful search
4.  return 0      // unsuccessful search
```

The best-case time of this algorithm is straightforward: key *is located in position* 1*. Thus the best-case time is* $\Theta(1)$.

The worst case-time is analysed similarly. In this case key *is not in the sequence, and thus the loop is executed n times. Hence, the worst-case time is* $\Theta(n)$.

Finally, the average-case time: if the key *is found at the ith position, line 2 has executed i times. If the* key *is not found, line 2 is executed n times. Thus the average number of times line 2 is executed is:*

$$\frac{(1+2+\cdots+n)+n}{n+1}.$$

Now

$$\begin{aligned}\frac{(1+2+\cdots+n)+n}{n+1} &\leq \frac{n^2+n}{n+1}\\ &= \frac{n(n+1)}{n+1}\\ &= n.\end{aligned}$$

Therefore the average-case time is $O(n)$*. Also,*

$$\begin{aligned}\frac{(1+2+\cdots+n)+n}{n+1} &\geq \frac{n^2/4+n}{n+1}\\ &\geq \frac{n^2/4+n/4}{n+1}\\ &\geq \frac{n}{4}\end{aligned}$$

Therefore the average-case time is $\Omega(n)$*, and hence* $\Theta(n)$.

[1] Due to equivalence $\sum_{k=0}^{n-1} ar^k = a\frac{1-r^n}{1-r}$, for $r \neq 1$.

The following table presents the common names for various Theta forms:

Theta Form	Name
$\Theta(1)$	Constant
$\Theta(\log\log n)$	Log log
$\Theta(\log n)$	Logarithmic
$\Theta(n)$	Linear
$\Theta(n\log n)$	n log n
$\Theta(n^2)$	Quadratic
$\Theta(n^3)$	Cubic
$\Theta(n^m)$	Polynomial
$\Theta(m^n), m \geq 2$	Exponential
$\Theta(n!)$	Factorial

Algorithms with theta forms lower down the table will eventually have longer running times than algorithms higher with theta forms higher up the table. It is worth experimenting with these formulas for larger and larger values of n. An algorithm that requires 2^n steps for inputs of size n is feasible for only small values of n, though algorithms requiring n^2 or n^3 steps also become infeasible, but for relatively larger input sizes. Notice also the dramatic improvement moving from n^2 steps to $n\log n$ steps. Often, nonetheless, we consider a problem that has a worse-case polynomial time algorithm to be considered to have a good algorithm; the interpretation is that such a problem has an efficient solution.

A problem that does not have a worst-case polynomial time algorithm is said to be *intractable*. Such an algorithm is guaranteed to take a long time to execute in the worst case for even modest sizes of input.

2.3 Recurrence Relations

A recurrence relation describes a sequence of numbers by relating the nth element of a sequence to its predecessors. Because recurrence relations are closely related to recursive algorithms, they naturally arise in the analysis of recursive algorithms. Recurrence relations are defined inductively, so mathematical induction is a commonly used technique for reasoning about them and recursive algorithms.

Example 2.3.1 *Consider the sequence*

$$5, 8, 11, 14, 17, \ldots.$$

If we denote the sequence as $a_1, a_2, \ldots$, it may be rephrased as the recurrence relation:

$$\begin{aligned} a_1 &= 5 \\ a_n &= a_{n-1} + 3, \qquad n \geq 2. \end{aligned}$$

Definition 2.3.2 (Recurrence Relation) *A* recurrence relation *for the sequence $a_0, a_1, \ldots$ is an equation that relates a_n to certain of its predecessors $a_0, a_1, \ldots, a_{n-1}$.*

Initial conditions *for the sequence $a_0, a_1, \ldots$ are given explicitly values for a finite number of the terms of the sequence.*

Example 2.3.3 (Fibonacci Numbers) *The Fibonacci numbers are defined by the recurrence relation*

$$f_n = f_{n-1} + f_{n-2}, \qquad n \geq 3$$

and initial conditions

$$f_1 = 1, \qquad f_2 = 2.$$

Example 2.3.4 (Compound Interest) *The yearly value of an investment of* \$1000 *at 12% compounded annually is described by the following recurrence relation:*

$$\begin{aligned} A_0 &= 1000 \\ A_n &= A_{n-1} + 0.12A_{n-1} \\ &= 1.12A_{n-1}, \qquad n \geq 1. \end{aligned}$$

Example 2.3.5 (Subset Number) *Let S_n denote the number of subsets of an n-element set. This can be represented by the recurrence relation:*

$$\begin{aligned} S_0 &= 1 \\ S_n &= 2S_{n-1}, \qquad n \geq 1. \end{aligned}$$

One of the main reasons to use recurrence relations is that sometimes it is easier to determine the nth term of a sequence of its predecessors than it is to find an explicit formula for the nth term in terms of n.

Example 2.3.6 *Let S_n denote the number of n-bit strings that do not contain the pattern* 111*. We will develop a recurrence relation for S_n.*

We will count the number of n-bit sequences that do not contain the pattern 111 *by partitioning into strings that begin with (a)* 0*, (b)* 10*, and (c)* 11*. If we can calculate S_n for strings satisfying these constraints, we can simply add them together.*

Suppose that an n-bit string begins with 0 *and does not contain pattern* 111*. Then the $(n-1)$-bit string following the initial* 0 *does not contain the pattern* 111*. Thus there are S_{n-1} strings of type (a).*

Similarly, if an n-bit string begins with 10 *and does not contain the pattern* 111*, then the $(n-2)$-bit string following the initial* 10 *cannot contain the pattern* 111*. Thus there are S_{n-2} strings of type (b).*

Finally, if an n-bit string begins with 11 *and does not contain the pattern* 111*, then the third bit must be a* 0*. The $(n-3)$-bit string following the initial* 110 *cannot contain the pattern* 111*; therefore there are S_{n-3} strings of type (c).*

Thus

$$S_n = S_{n-1} + S_{n-2} + S_{n-3}, \qquad n \geq 4.$$

By inspection, we find the initial conditions:

$$S_1 = 2, \quad S_2 = 4, \quad S_3 = 7.$$

Example 2.3.7 (Catalan Numbers) *The Catalan number C_n is equal to the number of routes from the lower left corner of an $n \times n$ square grid to the upper right corner if we are restricted to travelling only to the right or upwards and if we are allowed to touch but not go above the diagonal line from the lower left corner to the upper right corner. Such a route is called a* good route*.*

The recurrence relation giving the number of good routes in an $n \times n$ grid is

$$\begin{aligned} C_0 &= 1 \\ C_n &= \sum_{k=1}^{n} C_{k-1} C_{n-k}. \end{aligned}$$

Try to determine why?

A closed formula for Catalan numbers exists:

$$C_n = \frac{(2n)!}{(n+1)!n!}.$$

Example 2.3.8 (Ackermann's Function) *Ackermann's function is a function that grows extremely rapidly as its arguments increase. It appears in the time complexity of certain algorithms.*

$$\begin{aligned} A(m,0) &= A(m-1,1) && m \geq 1 \\ A(m,n) &= A(m-1,A(m,n-1)) && m,n \geq 1 \\ A(0,n) &= n+1 && n \geq 0 \end{aligned}$$

Experiment with this function for small values.

Argue that it terminates for all $n, m \geq 0$.

2.3.1 Solving Recurrence Relations

Solving a recurrence relation involving the sequence $a_0, a_1, \ldots$ involves finding an explicit formula for the general term a_n. We discuss two methods for solving such recurrence relations: *iteration* and a method that applies to *linear homogeneous recurrence relations with constant coefficients*. In general, solving recurrence relations is *hard*, though sophisticated techniques do exist to find exact formulas and upper and lower bounds.

To solve a recurrence relation by iteration, we use the recurrence relation to write the nth term a_n in terms of its predecessors $a_{n-1}, \ldots, a_0$. We then successively use the recurrence relation to replace each $a_{n-1}, \ldots$ by certain of their predecessors. We continue until an explicit formula is obtained.

Example 2.3.9 *Given the recurrence relation:*

$$\begin{aligned} a_1 &= 2 \\ a_n &= a_{n-1} + 3. \end{aligned} \tag{2.1}$$

This is solved by iteration as follows. Replace n by $n-1$ in (2.1) to obtain:

$$a_{n-1} = a_{n-2} + 3.$$

Substituting this for a_{n-1} in (2.1) gives:

$$a_n = a_{n-1} + 3 = (a_{n-2} + 3) + 3 = a_{n-2} + 3 \cdot 2. \tag{2.2}$$

Similarly we have from (2.1):

$$a_{n-2} = a_{n-3} + 3.$$

Substituting this into (2.2), results in

$$a_n = (a_{n-2} + 3) + 3 = a_{n-2} + 2 \cdot 3 = a_{n-3} + 3 \cdot 3. \tag{2.3}$$

In general, we have

$$a_n = a_{n-k} + k \cdot 3.$$

Setting $k = n - 1$ in this last expression, gives:

$$a_n = a_1 + (n-1) \cdot 3.$$

Since $a_1 = 2$, the explicit formula for a_n is:

$$a_n = 2 + 3(n-1).$$

Example 2.3.10 *We can solve the recurrence relation*

$$\begin{aligned} S_0 &= 1 \\ S_n &= 2S_{n-1} \end{aligned}$$

by iteration:

$$S_n = 2S_{n-1} = 2(2S_{n-2}) = \cdots = 2k(S_{n-k}) = \cdots = 2^n S_0 = 2^n.$$

Example 2.3.11 *The recurrence relation for the Tower of Hanoi puzzle*[2] *giving the number of moves required when n disks are present is:*

$$\begin{aligned} c_1 &= 1 \\ c_n &= 2c_{n-1} + 1 \end{aligned}$$

Applying the iterative method, we obtain

$$\begin{aligned} c_n &= 2c_{n-1} + 1 \\ &= 2(2c_{n-2} + 1) + 1 \\ &= 2^2 c_{n-2} + 2 + 1 \\ &= 2^3 c_{n-3} + 2^2 + 2 + 1 \\ &\vdots \\ &= 2^{n-1} c_1 + 2^{n-2} + 2^{n-3} + \cdots + 2 + 1 \\ &= 2^{n-1} + 2^{n-2} + 2^{n-3} + \cdots + 2 + 1 \\ &= 2^n - 1 \end{aligned}$$

The last step results from the formula for geometric sum.[3]

We now turn to a special class of recurrence relations.

[2] `http://en.wikipedia.org/wiki/Tower_of_Hanoi`

[3] $a + ar^1 + ar^2 + \cdots + ar^n = \frac{a(r^{n+1}-1)}{r-1}$.

Definition 2.3.12 *A* linear homogeneous recurrence relation of order k with constant coefficients *is a recurrence relation of the form*

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \ldots + c_k a_{n-k}, \quad c_k \neq 0 \tag{2.4}$$

A linear homogeneous recurrence relation of order k with constant coefficients, together with k initial conditions

$$a_0 = C_0, \quad a_1 = C_1, \cdots, a_{k-1} = C_{k-1},$$

uniquely defines a sequence $a_0, a_1, \ldots$.

The following are *not* linear homogeneous recurrence relation with constant coefficients:

- $a_n = 3a_{n-1}a_{n-2}$. It is *nonlinear*.
- $a_n - a_{n-1} = 2n$. $2n$ is not a constant. This kind is dubbed *inhomogeneous*.
- $a_n = 3na_{n-1}$. $2n$ is not a constant.

We now illustrate the general method of solving linear homogeneous recurrence relation with constant coefficients by finding an explicit formula for the sequence defined by the recurrence relation.

Theorem 2.3.13 *Let*

$$a_n \quad = \quad c_1 a_{n-1} + c_2 a_{n-2} \tag{2.5}$$

be a second-order linear homogeneous recurrence relation with constant coefficients.

If S and T are solutions of (2.5), then $U = bS + dT$ is also a solution of (2.5).

If r is a root of

$$t^2 - c_1 t - c_2 \quad = \quad 0, \tag{2.6}$$

then the sequence r^n, $n \geq 0$, is a solution of (2.5).

If a is a sequence defined by (2.5) and

$$a_0 = C_0, \quad a_1 = C_1,$$

and r_1 and r_2 are roots of (2.6), then there exists constants b and d such that

$$a_n = br_1^n + dr_2^n, \qquad n \geq 0.$$

Example 2.3.14 *Find an explicit formula for the Fibonacci sequence.*

The Fibonacci sequence is defined by the linear, homogeneous, second-order recurrence relation

$$f_n - f_{n-1} - f_{n-2} = 0, \qquad n > 0$$
$$f_1 = 1, \qquad f_2 = 2.$$

We begin by solving the quadradic formula:

$$t^2 - t - 1 = 0.$$

The solutions are

$$t = \frac{1 \pm \sqrt{5}}{2}.$$

Thus the solution is of the form

$$f_n = b\left(\frac{1+\sqrt{5}}{2}\right)^n + d\left(\frac{1-\sqrt{5}}{2}\right)^n.$$

To satisfy the initial conditions, we must have:

$$\begin{aligned} b\left(\frac{1+\sqrt{5}}{2}\right) + d\left(\frac{1-\sqrt{5}}{2}\right) &= 1 \\ b\left(\frac{1+\sqrt{5}}{2}\right)^2 + d\left(\frac{1-\sqrt{5}}{2}\right)^2 &= 2. \end{aligned}$$

Solving these equations for b and d, we obtain

$$b = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right), \qquad d = -\frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right).$$

Therefore, an explicit formula for the Fibonacci sequence is

$$f_n = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^{n+1}.$$

Surprisingly, even though f_n is an integer, the preceding formula involves the irrational number $\sqrt{5}$.

For the general linear homogeneous recurrence relation of order k with constant coefficients (2.4), if r is a root of

$$t_k - c_1 t^{k-1} - c_2 t^{k-2} - \cdots - c_k = 0$$

of multiplicity m, it can be shown that

$$r^n, nr^n, \ldots, n^{m-1}r^n$$

are solutions of (2.4). This fact can be used to solve linear homogeneous recurrence relations of order k with constant coefficients. This result is beyond the scope of this course.

2.3.2 Application to the Analysis of Algorithms

We can use recurrence relations to analyse the time required by algorithms. The technique is to develop a recurrence relation and initial conditions that define a sequence $a_1, a_2, \ldots$, where a_n is the time (best-case, average-case, or worst-case time) required for an algorithm to execute an input of size n. By solving the recurrence relation, we can determine the time needed by the algorithm.

Example 2.3.15 (Selection Sort) *This algorithm sorts the sequence*

$$s_1, s_2, \ldots, s_n$$

in increasing order by first selecting the largest item and placing it last and then recursively sorting the remaining elements.

```
 1.  procedure selection_sort(s,n)
 2.  // base case
 3.  if n = 1 then return
 4.  // find largest
 5.  max_index = 1 // assume initially that s[1] is largest
 6.  for i = 2 to n do
 7.    if s[i] > s[max_index] then // found larger, so update
 8.      max_index = i
 9.  // move largest to end
10.  swap(s[n],s[max_index])
11.  call selection_sort(s, n-1)
```

As a measure of the time required by this algorithm, we count the number of comparisons b_n at line 7 required to sort n items. We immediately obtain initial condition

$$b_1 = 0.$$

To obtain a recurrence relation for the sequence $b_1, b_2, \ldots$, we simulate the execution of the algorithm for arbitrary input of size $n > 1$, counting the number of comparisons, to obtain b_n. At line 7, there are $n-1$ comparisons, as line 6 causes line 7 to be executed $n-1$ times. In line 11, the recursive call requires b_{n-1} comparisons for input of size $n-1$. Therefore, the total number of comparisons is

$$b_n = n - 1 + b_{n-1},$$

which yields the desired recurrence relation.

This recurrence relation can be solved by iteration:

$$\begin{aligned} b_n &= b_{n-1} + n - 1 \\ &= (b_{n-2} + n - 2) + (n-1) \\ &= (b_{n-3} + n - 3) + (n-2) + (n-1) \\ &\vdots \\ &= b_1 + 1 + 2 + \cdots + (n-2) + (n-1) \\ &= 0 + 1 + 2 + \cdots + (n-2) + (n-1) = \frac{(n-1)n}{2} = \Theta(n^2). \end{aligned}$$

The following algorithm is *binary search*. It uses a divide-and-conquer approach. The sequence is divided into two nearly equal parts (line 4). If the item is found at the dividing point (line 5), the algorithm terminates. If the item is not find, because the sequence is sorted, an additional comparison (line 7) will locate the half of the sequence in which the item appears if it is present. The algorithm is recursively invoked (line 11) to continue the search.

Example 2.3.16 (Binary Search) *This algorithm looks for a value in an increasing sequence and returns the index of the value if it is found, or 0, if it is not found. It takes as input a sequence $s_i, s_{i+1}, \ldots, s_j$, $i \geq 1$, sorted in increasing order, plus* key*, i, and j.*

```
 1.  procedure binary_search(s, i, j, key)
 2.    if i > j then        // not found
```

```
 3.      return 0
 4.    k = (i + j)/2  // integer division
 5.    if key == s[k] then  // found
 6.      return k
 7.    if key < s[k] then   // search left half
 8.      j = k - 1
 9.    else                 // search right half
10.      i = k + 1
11.    return binary_search(s, i, j, key)
```

The worst-case time required by binary search is the number of times the algorithm is invoked in the worst case for a sequence containing n items. Let a_n denote the worst-case time.

Suppose that n is 1. In the worst case, the item will not be found in line 5 and the algorithm will be invoked a second time at line 11. The second call will have $i > j$ and the algorithm with terminate unsuccessfully at line 3. Thus

$$a_1 = 2.$$

Now for $n > 1$. In the worst case, the item will not be found at line 5, so the algorithm will be invoked at line 11. By definition, line 11 will require a total of a_m invocations, where m is the size of the input sequence input at line 11. Since the sizes of the left and right sides of the original sequence are $\lfloor n/2 \rfloor$ and $\lfloor (n-1)/2 \rfloor$ and the worst case occurs with the larger sequence, the total number of invocations at line 11 will be $a_{\lfloor n/2 \rfloor}$. The original invocation together with the invocations at line 11 gives all the invocations; thus

$$a_n = 1 + a_{\lfloor n/2 \rfloor}.$$

This is not easy to solve explicitly, so we will do so assuming that n is a power of 2. Thus the recurrence becomes:

$$a_{2^k} = 1 + a_{2^{k-1}}, \qquad k = 1, 2, \ldots.$$

If we let $b_k = a_{2^k}$, we obtain the recurrence relation

$$b_k = 1 + b_{k-1}, \qquad k = 1, 2, \ldots.$$

and the initial condition

$$b_0 = 2.$$

Solving using the iterative method:

$$\begin{aligned} b_k &= 1 + b_{k-1} \\ &= 2 + b_{k-2} \\ &\vdots \\ &= k + b_0 \\ &= k + 2. \end{aligned}$$

Thus, if $n = 2^k$,

$$a_n = 2 + \log_2 n.$$

If an arbitrary value of n falls between two powers of 2, say

$$2^{k-1} < n \leq 2^k.$$

Since the sequence a is not decreasing

$$a_{2^{k-1}} \leq a_n \leq a_{2^k}.$$

Note also that

$$k-1 < \log_2 n \leq k.$$

We can now deduce that

$$\log_2 n < 1+k = a_{2^{k-1}} \leq a_n \leq a_{2^k} = 2+k < 3+\log_2 n = O(\log_2 n).$$

Therefore $a_n = \Theta(\log n)$, and so binary search is $\Theta(\log n)$ in the worse case.

For our last example, we will analyse *merge sort*. In merge sort the sequence to be sorted is divided into two nearly equal sequences, each of which is recursively sorted, after which they are combined to produce a sorted arrangement of the original sequence. The process of combining two sorted sequences is called *merging*.

Example 2.3.17 (Merging Two Sequences) *This algorithm combines two increasing sequences into a single increasing sequence.*

Input is two increasing sequences: $s_i, \ldots, s_m$ and $s_{m+1}, \ldots, s_j$, and indexes i, m, and j. Output is a sequence $c_i, \ldots, c_j$ consisting of the elements $s_i, \ldots, s_m$ and $s_{m+1}, \ldots, s_j$ combined into one increasing sequence.

```
 1.  procedure merge(s, i, m, j, c)
 2.    // p is the position in the sequence s[i],...,s[m]
 3.    // q is the position in the sequence s[m+1],...,s[j]
 4.    // r is the position in the sequence c[i],...,c[j]
 5.    p = i
 6.    q = m + 1
 7.    r = i
 8.    // copy smaller of s[p] and s[q]
 9.    while p <= m and q <= j do
10.      if s[p] < s[q] then
11.        c[r] = s[p]
12.        p = p + 1
13.      else
14.        c[r] = s[q]
15.        q = q + 1
16.      r = r + 1
17.    // copy remainder of first sequence
18.    while q <= j do
19.      c[r] = s[q]
20.      q = q + 1
21.      r = r + 1
```

In the worst case this algorithm requires $j-i$ comparisons. In particular, in the worst case, $n-1$ comparisons are needed to merge two sequences the sum of whose lengths is n. This follows because the while loop will execute as long as $p \leq m$ and $q \leq j$, which in the worst case requires $j-i$ comparisons.

Example 2.3.18 (Merge Sort)

```
 1.  procedure merge_sort(s, i, j)
 2.    // base case: i = j
 3.    if i == j then
 4.      return
 5.    // divide sequence and sort
 6.    m = (i + j) / 2
 7.    call merge_sort(s, i, m)
 8.    call merge_sort(s, m+1, j)
 9.    // merge
10.    call merge(s, i, m, j, c)
11.    // copy c, the output of merge, into s
12.    for k = 1 to j do
13.      s[k] = c[k]
```

This algorithm is $\Theta(n \log n)$ in the worst case.

Let a_n be the number of comparisons require to sort n items in the worst case. Then $a_1 = 0$. If $n > 1$, a_n is at most the sum of the numbers of comparisons in the worst case resulting from the recursive calls at lines 7 and 8, and the number of comparisons in the worst case required by merge at line 10. That is,

$$a_n \leq a_{\lceil n/2 \rceil} + a_{\lceil (n-1)/2 \rceil} + n - 1.$$

In fact, this upper bound is achievable, so that

$$a_n = a_{\lceil n/2 \rceil} + a_{\lceil (n-1)/2 \rceil} + n - 1.$$

First we solve the preceding recurrence relation in case n is a power of 2, say $n = 2^k$. The equation becomes

$$a_{2^k} = 2a_{2^{k-1}} + 2^k - 1.$$

We may solve this last equation using iteration

$$\begin{aligned}
a_{2^k} &= 2a_{2^{k-1}} + 2^k - 1 \\
&= 2(2a_{2^{k-2}} + 2^{k-1} - 1) + 2^k - 1 \\
&= 2^2 a_{2^{k-2}} + 2 \cdot 2^k - 1 - 2 \\
&= 2^2(2a_{2^{k-3}} + 2^{k-2} - 1) + 2 \cdot 2^k - 1 - 2 \\
&= 2^3 a_{2^{k-3}} + 3 \cdot 2^k - 1 - 2 - 2^2 \\
&\vdots \\
&= 2^k a_{2^0} + k \cdot 2^k - 1 - 2 - 2^2 - \cdots - 2^{k-1} \\
&= k \cdot 2^k - (2^k - 1) \\
&= (k-1) \cdot 2^k + 1.
\end{aligned}$$

An arbitrary value of n falls between two powers of 2, say

$$2^{k-1} < n \leq 2^k.$$

which gives also

$$k - 1 < \log_2 n \leq k.$$

Since the sequence a is not decreasing,

$$a_{2^{k-1}} \leq a_n \leq a_{2^k}.$$

We can now deduce that

$$\begin{aligned} \Omega(n\log n) &= (-2+\log n)\frac{n}{2} < (k-2)2^{k-1}+1 = a_{2^{k-1}} \leq a_n \\ &\leq a_{2^i} \leq k2^k \leq (1+\log n)2n+1 = O(n\log n) \end{aligned}$$

Therefore $a_n = \Theta(n\log n)$, so merge sort is $\Theta(n\log n)$ in the worst case.

Note that any comparison based sorting algorithm is $\Theta(n\log n)$ in the worst case.

Part II

Automata and Formal Languages

Chapter 3

Basic Topics in Formal Languages

3.1 Formal Languages

Generally, we are interested in questions of the form: "Given some string s and some language L, is s in L?" We now define terms required to study what we mean by this kind of question.

An *alphabet*, often denoted by Σ, is a finite set. The members of Σ are called *symbols* or *characters*.

A *string* is a finite sequence, possibly empty, of symbols drawn from some alphabet Σ. The empty string is written as ϵ. The set of all possible strings over an alphabet Σ is written Σ^*. (The operation denoted by $*$ is called *Kleene star*.)

The *length* of a string s, which we shall write as $|s|$, is the number of symbols in s. For example, $|\epsilon| = 0$ and $|\texttt{howdy}| = 5$. For any symbol c and string s, we can define the function $\#_c s$ to be the number of times that the symbol c occurs in s. So, for example, $\#_a(\texttt{abbaaa}) = 4$.

The concatenation of two strings s and t, written st, is the string formed by appending t to s. For example, if $x = \texttt{good}$ and $y = \texttt{bye}$, then $xy = \texttt{goodbye}$. So, $|xy| = |x| + |y|$.

Next we define string *replication*. For each string w and each natural number i, the string w^i is defined as:

$$\begin{aligned} w^0 &= \epsilon \\ w^{i+1} &= w^i w \end{aligned}$$

For example, $\texttt{a}^3 = \texttt{aaa}$, $(\texttt{bye})^2 = \texttt{byebye}$, and $\texttt{a}^0\texttt{b}^3 = \texttt{bbb}$.

For each string w, the *reversal* of w, which we shall write as w^R, is defined as:

$$\begin{aligned} &\text{if } w = \epsilon, \text{ then } w^R = \epsilon \\ &\text{if } w = ua, \text{ then } w^R = au^R. \end{aligned}$$

Note that here we are relying on the fact that if $|w| \geq 1$, then $\exists a \in \Sigma \cdot \exists u \in \Sigma^* \cdot w = ua$. That is, a non-empty string can be split into a front part and the final character. One can similarly split a string as $w = au$, for some (other) $a \in \Sigma$ and $u \in \Sigma^*$. This idea can be used as the basis for inductive arguments on strings.

Exercise 3.1.1 *If w and x are strings, prove that $(wx)^R = x^R w^R$. (Hint: use induction on $|x|$.)*

A string s is a *substring* of string t iff s occurs contiguously as a part of t, that is iff $\exists x, y \in \Sigma^* \cdot t = xsy$. For example, aaa is a substring of aaabbbaaa, whereas aaaaaa is not a substring of aaabbbaaa.

A string s is a *proper substring* of a string t iff s is a substring of t and $s \neq t$. Every string is a substring of itself (though not a proper substring). The empty string, ϵ, is a substring of every string.

A string s is a *prefix* of t iff $\exists x \in \Sigma^* \cdot t = sx$. A string s is a *proper prefix* of a string t iff s is a prefix of t and $s \neq t$. Every string is a prefix of itself (though not a proper prefix). The empty string is a prefix of every string. For example, the prefixes of abba are ϵ, a, ab, abb and abba.

A string s is a *suffix* of t iff $\exists x \in \Sigma^* \cdot t = xs$. A string s is a *proper suffix* of a string t iff s is a suffix of t and $s \neq t$. Every string is a suffix of itself (though not a proper suffix). The empty string is a suffix of every string. For example, the suffixes of abba are ϵ, a, ba, bba and abba.

A *language* is a (finite or infinite) set of strings over an alphabet Σ. When we are talking about more than one language, we will use the notation Σ_L to mean the alphabet from which the strings in language L are formed.

Example 3.1.2 *Let $\Sigma = \{a, b\}$. Some example languages over Σ are $\emptyset$, $\{\epsilon\}$, $\{\mathtt{a}, \mathtt{b}\}$, $\{\epsilon, \mathtt{a}, \mathtt{aa}, \mathtt{aaa}, \mathtt{aaaa}, \mathtt{aaaaa}\}$, and $\{\epsilon, \mathtt{a}, \mathtt{aa}, \mathtt{aaa}, \mathtt{aaaa}, \mathtt{aaaaa}, \ldots\}$.*

Example 3.1.3 *Let $L = \{w \in \{\mathtt{a}, \mathtt{b}\}^* \mid$ all a's proceed all b's in $w\}$.*

Exercise 3.1.4 *Define the language in Example 3.1.3 more formally.*

Example 3.1.5 (Strings that end in a) *Let $L = \{x \mid \exists y \in \{\mathtt{a}, \mathtt{b}\}^* \cdot x = y\mathtt{a}\}$.*

Example 3.1.6 (The Empty Language) *Let $L = \{\} = \emptyset$. L is the language that contains no strings.*

Example 3.1.7 (The Empty Language is Different from the Empty String) *Let $L = \{\epsilon\}$, the language that contains a single string ϵ. Note that L is different from $\emptyset$.*

Example 3.1.8 (A Halting Problem Language) *Let $L = \{w \mid w$ is a C program that halts on all inputs$\}$.*

We can use the relations we have defined on strings as a way to define languages.

Example 3.1.9 (Using the Prefix Relation) *We define the following languages in terms of the prefix relation on strings:*

$$\begin{aligned}
L_1 &= \{w \in \{\mathtt{a}, \mathtt{b}\}^* \mid \text{no prefix of } w \text{ contains } \mathtt{b}\} \\
&= \{\epsilon, \mathtt{a}, \mathtt{aa}, \mathtt{aaa}, \mathtt{aaaa}, \mathtt{aaaaa}, \ldots\}. \\
L_2 &= \{w \in \{\mathtt{a}, \mathtt{b}\}^* \mid \text{no prefix of } w \text{ starts with } \mathtt{b}\} \\
&= \{w \in \{\mathtt{a}, \mathtt{b}\}^* \mid \text{the first character of } w \text{ is } \mathtt{a}\} \cup \{\epsilon\} \\
L_3 &= \{w \in \{\mathtt{a}, \mathtt{b}\}^* \mid \text{every prefix of } w \text{ starts with } \mathtt{b}\} \\
&= \emptyset.
\end{aligned}$$

L_3 is equal to $\emptyset$ because ϵ is a prefix of every string. Since ϵ does not start with b*, no strings meet L_3's requirements.*

Example 3.1.10 (Using Replication to Define a Language) *Let $L = \{a^n \mid n \geq 0\}$. $L = \{\epsilon, \mathtt{a}, \mathtt{aa}, \mathtt{aaa}, \mathtt{aaaa}, \mathtt{aaaaa}, \ldots\}$.*

Languages are sets. So, if we want to provide a computational definition of a language, we could specify either:

- a language generator, which enumerates (lists) the elements of the language, or
- a language recogniser, which describes whether or not a candidate string is in the language and returns *True* if it is and *False* if it isn't.

In some cases, when considering an enumerator for a language L, we may care about the order in which the elements of L are generated. If there exists a total order D of the elements of Σ_L, then we can use D to define on L a useful total order called *lexicographic order* (written $<_L$):

- Shorter strings precede longer ones: $\forall x \forall y \cdot (|x| < |y| \rightarrow x <_L y)$, and
- For strings that are of the same length, sort them in dictionary order using D.

We will say that a program *lexicographically enumerates* the elements of L iff it enumerates them in lexicographic order.

- A program that lexicographically enumerates a language can be used to recognise words from the language. How?
- A program that recognises words from a language can be used to enumerate words from the language. How?

How large is a language? The smallest language over any alphabet is $\emptyset$, whose cardinality is 0. The largest language over any alphabet Σ is Σ^*. What is $|\Sigma^*|$? Suppose that $\Sigma = \emptyset$. Then $\Sigma^* = \{\epsilon\}$ and $|\Sigma^*| = 1$. But what about the far more useful case when Σ is not empty?

Theorem 3.1.11 (Cardinality of Σ^*) *If $\Sigma \neq \emptyset$, then Σ^* is countably infinite.*

Proof. The elements of Σ^* can be lexicographically enumerated by a straightforward procedure that:

- Enumerates all strings of length 0, then length 1, then length 2, and so forth.
- Within strings of a given length, enumerates them in dictionary order.

This enumeration is infinite since there is no longest string in Σ^*. As there exists an infinite enumeration of Σ^*, it is countably infinite.[1] □

Let Σ be an alphabet. How many languages are defined on Σ? The set of languages defined on Σ is $\mathscr{P}(\Sigma^*)$, the powerset of Σ^*. If $\Sigma = \emptyset$, then $\Sigma^* = \{\epsilon\}$ and $\mathscr{P}(\Sigma^*)$ is $\{\emptyset, \{\epsilon\}\}$. What about the case when Σ is not empty?

Theorem 3.1.12 *If $\Sigma \neq \emptyset$, then the set of languages over Σ is uncountably infinite.*

[1] Based on well-known result: a set A is countably infinite iff there exists an infinite enumeration of it. An infinite enumeration of A is a bijection from $\mathbb{N}$ to A.

Proof. The set of languages defined on Σ is $\mathscr{P}(\Sigma^*)$. By Theorem 3.1.11, Σ^* is countably infinite. By well-known result,[2] if S is a countably infinite set, then $\mathscr{P}(S)$ is uncountably infinite. So $\mathscr{P}(\Sigma^*)$ is uncountably infinite. □

Since languages are sets, all of the standard set operations are well-defined on languages. In particular, we will find union, intersection, set difference, and complement to be useful. Complement will be defined with Σ^* as the universe unless we explicitly state otherwise. Three other useful operations on languages we consider are concatenation, Kleene star, and reverse.

Let L_1 and L_2 be two languages defined over some alphabet Σ. Then their *concatenation*, written L_1L_2, is:

$$L_1L_2 = \{w \in \Sigma^* \mid \exists s \in L_1 \cdot \exists t \in L_2 \cdot w = st\}.$$

The language $\{\epsilon\}$ is the identity for concatenation of languages. So, for all languages L, $L\{\epsilon\} = \{\epsilon\}L = L$.

The language $\emptyset$ is a zero for concatenation of languages. So, for all languages L, $L\emptyset = \emptyset L = \emptyset$.

Concatenation is also associative. So, for all languages L_1, L_2, and L_3:

$$(L_1L_2)L_3 = L_1(L_2L_3).$$

Let L be a language defined over some alphabet Σ. Then the *Kleene star* of L, written L^*, is:

$$L^* = \{\epsilon\} \cup \{w \in \Sigma^* \mid \exists k \geq 1 \cdot \exists w_1, \ldots, w_k \in L \cdot w = w_1 \cdots w_k\}$$

In other words, L^* is the set of strings that can be formed by concatenating together zero or more strings from L. That is, $L^* = \{\epsilon\} \cup L^1 \cup L^2 \cup L^3 \cup \cdots$.

It is sometimes useful to require that at least one element of L be selected, so we define:

$$L^+ = LL^*.$$

Another way to describe L^+ is that it is the closure of L under concatenation. Note that $L^+ = L^* \setminus \{\epsilon\}$ iff $\epsilon \notin L$.

Let L be a language defined over some alphabet Σ. Then the *reverse* of L, written L^R, is:

$$L^R = \{w \in \Sigma^* \mid w = x^R \text{ for some } x \in L\}.$$

In other words, L^R is the set of strings that can be formed by taking some string in L and reversing it.

Exercise 3.1.13 *Prove the following: If L_1 and L_2 are languages, then $(L_1L_2)^R = L_2^R L_1^R$.*

3.1.1 Formal Grammars

Languages are often specified by formal grammars. This notion should be familiar from compilers and programming language descriptions.

A *formal grammar* of this type consists of:

- a finite set of terminal symbols

[2] The proof of this result relies on Diagonalisation, which we cover in Section 6.3.2

- a finite set of nonterminal symbols
- a finite set of production rules with a left and a right-hand side consisting of a sequence of these symbols
- a start symbol.

A formal grammar defines (or *generates*) a *formal language*, which is a (usually infinite) set of finite-length sequences of symbols (i.e. strings) that may be constructed by applying production rules to another sequence of symbols which initially contains just the start symbol. A rule may be applied to a sequence of symbols by replacing an occurrence of the symbols on the left-hand side of the rule with those that appear on the right-hand side. A sequence of rule applications is called a derivation. Such a grammar defines the formal language: all words consisting solely of terminal symbols which can be reached by a derivation from the start symbol.

Nonterminals are usually represented by uppercase letters, terminals by lowercase letters, and the start symbol by S. For example, the grammar with terminals $\{a, b\}$, nonterminals $\{S, A, B\}$, production rules:

$$\begin{aligned} S &\to ABS \\ S &\to \epsilon \\ BA &\to AB \\ BS &\to b \\ Bb &\to bb \\ Ab &\to ab \\ Aa &\to aa \end{aligned}$$

and start symbol S, defines the language of all words of the form $a^n b^n$ (i.e., n copies of a followed by n copies of b). The following is a simpler grammar that defines the same language: Terminals $\{a, b\}$, Nonterminals $\{S\}$, Start symbol S, Production rules:

$$\begin{aligned} S &\to aSb \\ S &\to \epsilon \end{aligned}$$

3.1.2 Chomsky's Hierarchy

One way of classifying grammars was defined by Noam Chomsky. The classification captures roughly how easy it is to determine whether a given word can be generated by the grammar. This information is provided for interest—we will cover many of the aspects mentioned in the table, and it is good to know how they are related to each other.

The Chomsky hierarchy consists of the following levels:

- Type-0 grammars (unrestricted grammars) include all formal grammars. They generate exactly all languages that can be recognised by a Turing machine. These languages are also known as the *recursively enumerable languages*. Note that this is different from the recursive languages which can be decided by an always-halting Turing machine. See Chapter 6.

- Type-1 grammars (context-sensitive grammars) generate the context-sensitive languages. These grammars have rules of the form $\alpha A\beta \rightarrow \alpha\gamma\beta$, with A a nonterminal and α, β and γ strings of terminals and nonterminals. The strings α and β may be empty, but γ must be nonempty. The rule is allowed if S does not appear on the right side of any rule. The languages described by these grammars are exactly all languages that can be recognised by a linear bounded automaton (a nondeterministic Turing machine whose tape is bounded by a constant times the length of the input). Context-sensitive grammars are not covered in these notes.

- Type-2 grammars (context-free grammars) generate the context-free languages. These are defined by rules of the form $A \rightarrow \gamma$ with A a nonterminal and γ a string of terminals and nonterminals. These languages are exactly all languages that can be recognised by a non-deterministic pushdown automaton. Context-free languages are the theoretical basis for the syntax of most programming languages. See Chapter 5.

- Type-3 grammars (regular grammars) generate the regular languages. Such a grammar restricts its rules to a single nonterminal on the left-hand side and a right-hand side consisting of a single terminal, possibly followed (or preceded, but not both in the same grammar) by a single nonterminal. The rule $S \rightarrow \epsilon$ is also allowed here if S does not appear on the right side of any rule. These languages are exactly all languages that can be decided by a finite state automaton. Additionally, this family of formal languages can be obtained by regular expressions. Regular languages are commonly used to define search patterns and the lexical structure of programming languages. Regular grammars are the topic of the present chapter, though we do not explicitly consider grammars of the presented form, but rather use an equivalent notion.

The set of grammars corresponding to recursive languages is not a member of this hierarchy.

Every regular language is context-free, every context-free language not containing the empty string is context-sensitive and every context-sensitive language is recursive and every recursive language is recursively enumerable. These are all proper inclusions, meaning that there exist recursively enumerable languages which are not context-sensitive, context-sensitive languages which are not context-free and context-free languages which are not regular.

The following table summarises each of Chomsky's four types of grammars, the class of language it generates, the type of automaton that recognises it, and the form its rules must have.

Grammar	Languages	Automaton	Production rules (constraints)
Type-0	Recursively enumerable	Turing machine	$\alpha \rightarrow \beta$ (no restrictions)
Type-1	Context-sensitive	Linear-bounded non-deterministic Turing machine	$\alpha A\beta \rightarrow \alpha\gamma\beta$
Type-2	Context-free	Non-deterministic pushdown automaton	$A \rightarrow \gamma$
Type-3	Regular	Finite state automaton	$A \rightarrow a$ and $A \rightarrow aB$

Chapter 4

Finite Automata

4.1 Deterministic Finite State Machines

We now introduce machines (automata) for recognising strings in a language. These read a string one character at a time and determine whether the string is in the language by entering some final or accepting state. Firstly, we will begin with deterministic machines: every step of a computation follows in a unique way from the preceding step. When the machine is in a given state and reads the next input symbol, we know what the next state will be—it is determined.

We now provide a formal definition of a *finite automaton*.

Definition 4.1.1 (Deterministic Finite Automaton) *A* deterministic finite automaton *(DFA) is a 5-tuple* $(Q, \Sigma, \delta, q_0, F)$*, where*

1. Q *is a finite set called the* states*,*
2. Σ *is a finite set called the* alphabet*,*
3. $\delta : Q \times \Sigma \to Q$ *is called the* transition function*,*
4. $q_0 \in Q$ *is the* start state*, and*
5. $F \subseteq Q$ *is the* set of accepting states*.*

Sometimes the start state is called the *initial* state, and accepting states are called *accept* or *final* states.

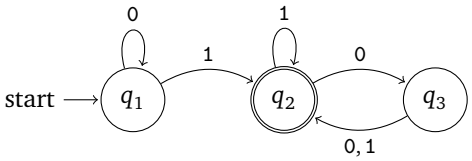


Figure 4.1: The finite automaton M_1

Consider the automaton in Figure 4.1. We can describe M_1 formally by writing $M_1 = (Q, \Sigma, \delta, q_1, F)$, where

1. $Q = \{q_1, q_2, q_3\}$,
2. $\Sigma = \{0, 1\}$,
3. δ is described as

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

4. q_1 is the start state, and
5. $F = \{q_2\}$.

When this automaton receives an input string such as 1101, it processes that string and produces an output which is either *accept* or *reject*. The processing begins in M_1's start state. The automaton receives the symbols from the input string one by one, from left to right. After reading each symbol, M_1 moves from one state to another along the transition that has that symbol as its label. When it reads the last symbol, M_1 produces its output. The output is *accept* if M_1 is now in an accepting state and *reject* if not.

When we feed 1101 into M_1 in Figure 4.1, the processing proceeds as follows:

1. Start in state q_1.
2. Read 1, follow transition from q_1 to q_2.
3. Read 1, follow transition from q_2 to q_2.
4. Read 0, follow transition from q_2 to q_3.
5. Read 1, follow transition from q_3 to q_2.
6. *Accept* because M_1 is in accepting state q_2 at the end of the input.

If A is the set of all strings that machine M accepts, we say that A is the *language of the machine* M and write $L(M) = A$. We say that M *recongnises* A or M *accepts* A. A machine may accept several strings, but it always recognises only one language. If the machine accepts no strings, it still recognises one language, namely the empty language $\emptyset$.

This can be formalised as follows.

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton and let $w = y_1 y_2 \cdots y_n$ be a string where each y_i is a member of the alphabet Σ. Then M *accepts* w if a sequence of states $r_0, r_1, \ldots, r_n$ in Q exists satisfying three conditions:

1. $r_0 = q_0$,
2. $\delta(r_i, y_{i+1}) = r_{i+1}$, for $i \in 0, \ldots, n-1$, and
3. $r_n \in F$.

Condition 1 says that the machine starts in the start state. Condition 2 says that the machine goes from state to state according to the transition function. Condition 3 says that the machine accepts its input if it ends up in an accepting state. We say that M *recognises* A if $A = \{w \mid M \text{ accepts } w\}$.

Definition 4.1.2 *A language is called a* regular language *if some finite automaton recognises it.*

4.2 Nondeterministic Finite State Machines

Nondeterminism is a useful concept that has had great impact on the theory of computation. So far in our discussion, every step of a computation follows in a unique way from the preceding step. When the machine is in a given state and reads the next input symbol, we know what the next state will be—it is determined. In a *nondeterministic* machine, several choices may exist for any state at any point.

Nondeterminism is a generalisation of determinism, so every deterministic finite automaton is automatically a nondeterministic finite automaton. As Figure 4.2 shows, nondeterministic finite automata may have additional features.

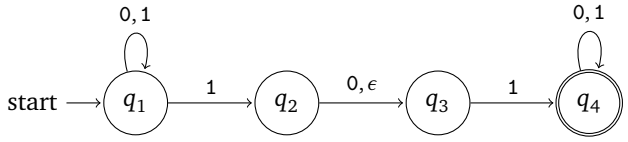


Figure 4.2: The nondeterministic finite automaton N_1

The differences are immediately apparent. Firstly, every state of a DFA always has exactly one outgoing transition arrow for each symbol in the alphabet. NFAs need not follow this rule: It may have zero, one or more outgoing transitions for each symbol. Secondly, in a DFA, the labels of the transitions are symbols from the alphabet. In an NFA transitions may be marked with members of the alphabet or with ϵ.

Notation 4.2.1 *For any alphabet Σ we write Σ_ϵ to denote $\Sigma \cup \{\epsilon\}$.*

Definition 4.2.2 (Nondeterministic Finite Automaton) *A* nondeterministic finite automaton *(NFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where*

1. *Q is a finite set called the* states*,*
2. *Σ is a finite set called the* alphabet*,*
3. *$\delta : Q \times \Sigma_\epsilon \to \mathscr{P}(Q)$ is called the* transition function*,*
4. *$q_0 \in Q$ is the* start state*, and*
5. *$F \subseteq Q$ is the* set of accepting states*.*

Recall the automaton in Figure 4.2. We can describe N_1 formally by writing $N_1 = (Q, \Sigma, \delta, q_1, F)$, where

1. $Q = \{q_1, q_2, q_3, q_4\}$,
2. $\Sigma = \{0, 1\}$,
3. δ is described as

	0	1	ϵ
q_1	$\{q_1\}$	$\{q_1, q_2\}$	$\emptyset$
q_2	$\{q_3\}$	$\emptyset$	$\{q_3\}$
q_3	$\emptyset$	$\{q_4\}$	$\emptyset$
q_4	$\{q_4\}$	$\{q_4\}$	$\emptyset$

4. q_1 is the start state, and

5. $F = \{q_4\}$.

The formal definition of computation for an NFA is similar to that for a DFA. Let $N = (Q, \Sigma, \delta, q_0, F)$ be a nondeterministic finite automaton and let $w = y_1 y_2 \cdots y_n$ be a string where each y_i is a member of the alphabet Σ_ϵ. Then N *accepts* w if a sequence of state $r_0, r_1, \ldots, r_n$ in Q exists satisfying three conditions:

1. $r_0 = q_0$,

2. $r_{i+1} \in \delta(r_i, y_{i+1})$ for $i \in 0, \ldots, n-1$, and

3. $r_n \in F$.

Condition 1 says that the machine starts in the start state. Condition 2 says that r_{i+1} is one of the allowable next states when N is in state r_i and reading y_i according to the transition function. Condition 3 says that the machine accepts its input if it ends up in an accepting state.

4.3 Equivalence of NFAs and DFAs

Deterministic and nondeterministic finite automata recognise the same class of languages. This equivalence is both surprising and useful. It is surprising because NFAs appear to have more power than DFAs, so we might expect that NFAs recognise more languages. It is useful because describing an NFA for a given language sometimes is much easier than describing a DFA for that language.

Theorem 4.3.1 *Every nondeterministic language has an equivalent deterministic finite automaton.*

Proof. Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA recognising some language A. We construct a DFA $M = (Q', \Sigma, \delta', q_0', F')$ recognising A. Before doing the full construction, let's first consider the easier case where N has no ϵ transitions. Later we will take these into account.

1. $Q' = \mathscr{P}(Q)$.
   Every state of M is a set of states of N.

2. For $R \in Q'$ and $a \in \Sigma$ let $\sigma'(R, a) = \{q \in Q \mid q \in \delta(r, a) \text{ for some } r \in R\}$.
   If R is a state of M, it is also a set of states of N. When M reads a symbol a in state R, it shows where a takes each state in R. Because each state may go to a set of states, we take the union of all of these sets. Another way to write this is

$$\delta'(R, a) = \bigcup_{r \in R} \delta(r, a).$$

3. $q_0' = \{q_0\}$.
   M starts in the state corresponding to the collection containing just the start state of N.

4. $F' = \{R \in Q' \mid R \text{ contains an accepting state of } N\}$.
   The machine M accepts if one of the possible states of N could be in at this point an accepting state.

Now we need to consider the ϵ transitions. To do so we set up an extra bit of notation. For any state R of M we define $E(R)$ to be the collection of states that can be reached from R by going only along ϵ arrows, including members of R themselves. Formally, for $R \subseteq Q$, let

$$E(R) = \{q \mid q \text{ can be reached from } R \text{ by travelling along 0 or more } \epsilon \text{ transitions}\}$$

Then we modify the transition function of M to record all states that can be reached by going along ϵ transitions after every step. Replacing $\delta(r,a)$ by $E(\delta(r,a))$ achieves this effect. Thus

$$\delta'(R,a) = \{q \in Q \mid q \in E(\delta(r,a)) \text{ for some } r \in R\}$$

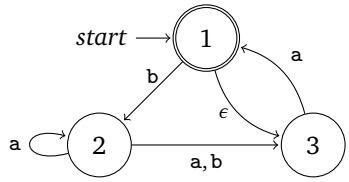
In addition, we need to modify the start state of M to record all possible states that can be reached from the start state of N along ϵ transitions. Changing q_0' to $E(\{q_0\})$ achieves this effect. We now have a complete construction of the DFA M that simulates the NFA N.

The construction of M obviously works correctly. At every step in the computation of M on an input, it clearly enters a state that corresponds to the subset of states that N could be in at that point. Thus our proof is complete. □

This construction is referred to as the *subset construction*.

If this construction had been any more complex, we would have to actually prove that it works as claimed. Such a proof would proceed by induction on the number of steps in the computation. Note that we have specified that we follow the ϵ transitions after each input symbol is read. An alternative procedure based on following the ϵ transitions before reading each input symbol works equally well.

Example 4.3.2 *We now illustrate the construction used in the proof of Theorem 4.3.1 for converting an NFA into a DFA. We will consider the following automata:*



A formal description of this automata is $N = (Q, \{\mathtt{a}, \mathtt{b}\}, \delta, 1, \{1\})$*, where* $Q = \{1,2,3\}$ *and* δ *is described as:*

	a	b	ϵ
1	$\emptyset$	$\{2\}$	$\{3\}$
2	$\{2,3\}$	$\{3\}$	$\emptyset$
3	$\{1\}$	$\emptyset$	$\emptyset$

To construct a DFA D that is equivalent to N, we first determine D's states. N has three states, so we construct D with 8 states, one for each subset of N's states. We label each of D's states with the corresponding subset. Thus D's state set is:

$$\{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}.$$

Next, we determine the start and accepting states of D. The start state is $E(\{1\})$*, the set of states that are reachable from* 1 *by travelling along* ϵ *transitions, plus* 1 *itself. A transition goes from* 1 *to* 3*, so* $E(\{1\}) = \{1,3\}$*. The new accepting states are those containing N's accepting state, thus* $\{\{1\}, \{1,2\}, \{1,3\}, \{1,2,3\}\}$.

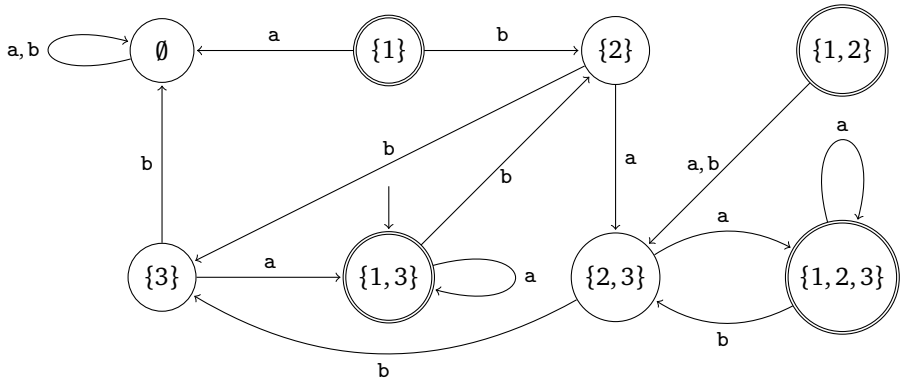
Finally, we need to determine D's transition function. Each of D's states goes to one place on input a *and one place on input* b*. We illustrate the process of determining of D's transitions with a few examples.*

In D, state $\{2\}$ *goes to* $\{2,3\}$ *on input* a*, because in N, state* 2 *goes to both* 2 *and* 3 *on input* a *and we cannot go further from* 2 *or* 3 *along* ϵ *transitions.*

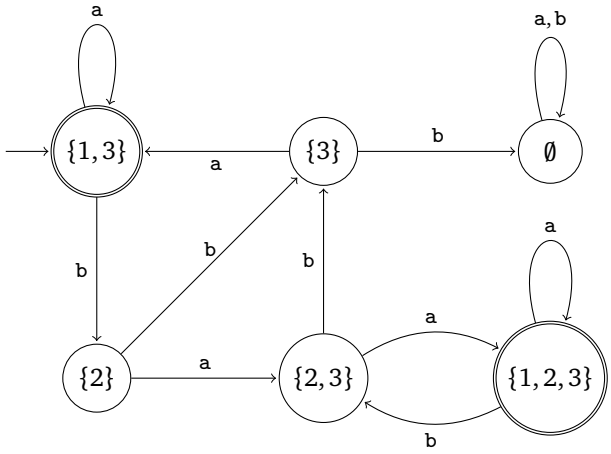
State $\{1\}$ *goes to* $\emptyset$ *on* a*, because no* a *transitions exit it. It goes to* $\{2\}$ *on* b*.*

State $\{3\}$ *goes to* $\{1,3\}$ *on* a*, because in N state* 3 *goes to state* 1 *on* a *and in turn goes to* 3 *with an* ϵ *transition. State* $\{3\}$ *on* b *goes to* $\emptyset$*.*

State $\{1,2\}$ *goes to* $\{2,3\}$ *because* 1 *points at no states with* a *transitions and* 2 *has* a *transitions to* 2 *and* 3 *and neither goes anywhere with an* ϵ *transition. State* $\{1,2\}$ *on* b *goes to* $\{2,3\}$*. Continuing in this way we obtain the following diagram for D:*



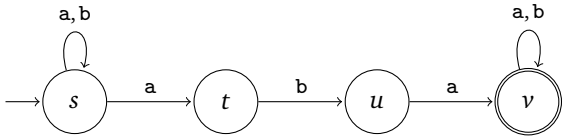
We may simplify this machine by observing that no transitions enter states $\{1\}$ *and* $\{1,2\}$*, thus they are unreachable and may be removed without affecting the behaviour of the automaton. Doing so yields the following automaton:*



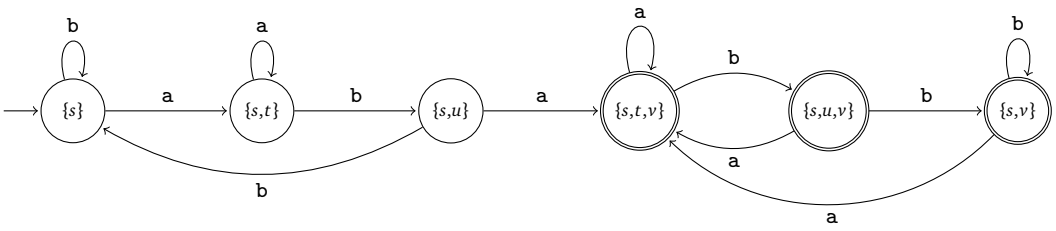
4.4 Minimisation of Finite State Machines

Often you come across situations where some automaton could be simplified simply by deleting states that are inaccessible from the start state of by collapsing states that were equivalent in some sense. For

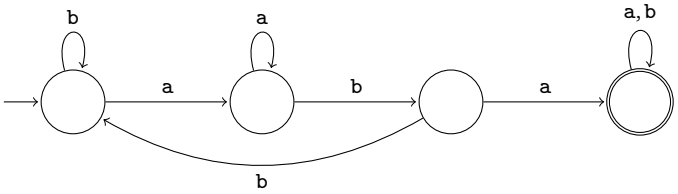
example, if we apply the subset construction to the following NFA



accepting the set of all strings containing the substring `aba`, we would obtain a DFA with $2^4 = 16$ states (Try it!). However, all but six of those states are inaccessible. Deleting them, results in the following DFA



Note that the rightmost three states of this DFA can be collapsed into a single state, since they are all accepting states, and once the machine enters one of them it cannot escape. Thus this DFA is equivalent to



This is a simple example in which the equivalence of states is obvious, but sometimes it is not so obvious. We will now develop a mechanical method to find all equivalent states of any given DFA and collapse them. This will give a DFA for any regular set A that has as few states as possible. An amazing fact is that every regular set has a minimal DFA that is unique up to isomorphism, and there is a purely mechanical method for constructing it from any DFA for A.

Before proceeding, we introduce some notation. Given a DFA $M = (Q, \Sigma, \delta, s, F)$, define $\widehat{\delta}(p, w)$, where $p \in Q$ and $w \in \Sigma^*$, as follows:

$$\begin{aligned} \widehat{\delta}(p, \epsilon) &= p \\ \widehat{\delta}(p, aw) &= \widehat{\delta}(\delta(p, a), w), \quad \text{where } a \in \Sigma, w \in \Sigma^*. \end{aligned}$$

Say we are given a DFA $M = (Q, \Sigma, \delta, s, F)$ for A. The minimisation process consists of two steps:

- Remove inaccessible states; that is, states q for which there exists no string $w \in \Sigma^*$ such that $\widehat{\delta}(s, w) = q$.
- Collapse "equivalent" states.

Removing inaccessible states surely does not change the set accepted. It is quite straightforward to see how to do this mechanically using depth-first search on the transition graph. Let us assume that this has been done. For state 2, we need to say what we mean by "equivalent" and how we do the collapsing.

4.4.1 The Quotient Construction

How do we know in general when two states can be collapsed safely without changing the set accepted? How do we do the collapsing formally? Is there a fast algorithm for doing it? How can we determine whether any further collapsing is possible?

Surely we never want to collapse an accepting state p and a rejecting state q, because if $p = \widehat{\delta}(s,x) \in F$ and $q \notin \widehat{\delta}(s,y) \in F$, then x must be accepted and y must be rejected, even after collapsing, so there is no way to declare the collapsed state to be both an accepting and a rejecting state. Also, if we collapse p and q, then we had better collapse $\delta(p,a)$ and $\delta(q,a)$ to maintain determinism. These observations together imply inductively that we cannot collapse p and q if $\widehat{\delta}(p,x) \in F$ and $\widehat{\delta}(q,x) \notin F$ for some string x.

It turns out that this criterion is necessary and sufficient for deciding whether a pair of states can be collapsed. That is, if there exists a string x such that $\widehat{\delta}(p,x) \in F$ and $\widehat{\delta}(q,x) \notin F$ or vice versa, then p and q cannot be safely collapsed; and if no such x exists, then they can.

Here's how we show this formally. We first define an equivalence relation $\approx$ on Q by

$$p \approx q \quad \widehat{=} \quad \forall x \in \Sigma^* \cdot (\widehat{\delta}(p,x) \in F \iff \widehat{\delta}(q,x) \in F).$$

It is not hard to argue that the relation $\approx$ is indeed an equivalence relation (namely, $\approx$ is reflexive, symmetric and transitive).

As with all equivalence relations, $\approx$ partitions the set on which it is defined into disjoint *equivalence classes*:

$$[p] \quad \widehat{=} \quad \{q \mid q \approx p\}.$$

Every element $p \in Q$ is contained in exactly one equivalence class $[p]$, and

$$p \approx q \iff [p] = [q].$$

We now define a DFA $M/\approx$ called the *quotient automaton*, whose states correspond to the equivalence classes of $\approx$. This construction is called a *quotient construction* and is quite common in algebra.

There is one state of $M/\approx$ for each $\approx$-equivalence class. In fact, formally, the states of $M/\approx$ are the equivalence classes; this is the mathematical way of "collapsing" equivalent states.

Define

$$\begin{array}{rcl} M/\approx & \widehat{=} & (Q', \Sigma, \delta', s', F') \\ \textit{where} & & \\ Q' & \widehat{=} & \{[p] \mid p \in Q\} \\ \delta'([p],a) & \widehat{=} & [\delta(p,a)] \\ s' & \widehat{=} & [s] \\ F' & \widehat{=} & \{[p] \mid p \in F\} \end{array}$$

There is a subtle point involving the definition of δ' above: we need to show that it is *well-defined*. Note that the action of δ' on the equivalence class $[p]$ is defined in terms of p. It is conceivable that a different choice of representative of the class $[p]$ (that is, some q such that $q \approx p$) might lead to a different right-hand side in the definition of δ' above. Lemma 4.4.1 says exactly that this cannot happen:

Lemma 4.4.1 *If $p \approx q$, then $\delta(p,a) \approx \delta(q,a)$. Equivalently, if $[p] = [q]$, then $[\delta(p,a)] = [\delta(q,a)]$.*

Proof. Suppose $p \approx q$. Let $a \in \Sigma$ and $y \in \Sigma^*$.

$$\begin{array}{rcll} \widehat{\delta}(\delta(p,a),y) \in F & \iff & \widehat{\delta}(p,ay) \in F & \\ & \iff & \widehat{\delta}(q,ay) \in F & \text{since } p \approx q \\ & \iff & \widehat{\delta}(\delta(q,a),y) \in F. & \end{array}$$

Since y was arbitrary, $\delta(p,a) \approx \delta(q,a)$ by definition of $\approx$. □

Lemma 4.4.2 *$p \in F$ if and only if $[p] \in F'$.*

Proof. The direction $\Rightarrow$ is immediate from the definition of F'. For the direction $\Leftarrow$, we need to show that if $p \approx q$ and $p \in F$, then $q \in F$. In other words, every $\approx$-equivalence class is either a subset of F or disjoint from F. This follows immediately by taking $x = \epsilon$ in the definition of $p \approx q$. □

Lemma 4.4.3 *For all $x \in \Sigma^*$, $\widehat{\delta'}([p],x) = [\widehat{\delta}(p,x)]$.*

Proof. By induction on $|x|$.

Base case. For $x = \epsilon$:

$$\begin{array}{rcll} \widehat{\delta'}([p],\epsilon) & = & [p] & \text{definition of } \widehat{\delta'} \\ & = & [\widehat{\delta}(p,\epsilon)] & \text{definition of } \widehat{\delta}\,. \end{array}$$

Induction step. Assume $\widehat{\delta'}([p],x) = [\widehat{\delta}(p,x)]$, and let $a \in \Sigma$.

$$\begin{array}{rcll} \widehat{\delta'}([p],xa) & = & \delta'(\widehat{\delta'}([p],x),a) & \text{definition of } \widehat{\delta'} \\ & = & \delta'([\widehat{\delta}(p,x)],a) & \text{induction hypothesis} \\ & = & [\delta(\widehat{\delta}(p,x),a)] & \text{definition of } \delta' \\ & = & [\widehat{\delta}(p,xa)] & \text{definition of } \widehat{\delta}\,. \end{array}$$

□

Theorem 4.4.4 *$L(M/\approx) = L(M)$.*

Proof. For $x \in \Sigma^*$,

$$\begin{array}{rcll} x \in L(M/\approx) & \iff & \widehat{\delta'}(s',x) \in F' & \text{definition of acceptance} \\ & \iff & \widehat{\delta'}([s],x) \in F' & \text{definition of } s' \\ & \iff & [\widehat{\delta}(s,x)] \in F' & \text{Lemma 4.4.3} \\ & \iff & \widehat{\delta}(s,x) \in F & \text{Lemma 4.4.2} \\ & \iff & x \in L(M) & \text{definition of acceptance.} \end{array}$$

□

$M/\approx$ cannot be collapsed further

It is conceivable that after doing the quotient construction once, we might be able to collapse even further by doing it again. It turns out that once is enough. To see this, let's do the quotient construction a second time. Define

$$[p] \sim [q] \quad \widehat{=} \quad \forall x \in \Sigma^* \cdot (\widehat{\delta}'([p], x) \in F' \iff \widehat{\delta}'([q], x) \in F').$$

This is exactly the same definition as $\approx$ above, only applied to the quotient automaton $M/\approx$. We use the notation $\sim$ for the equivalence relation on Q' to distinguish it from $\approx$ on Q. Now

$$\begin{array}{lll}
[p] \sim [q] & \Rightarrow & \forall x \in \Sigma^* \cdot (\widehat{\delta}'([p], x) \in F' \iff \widehat{\delta}'([q], x) \in F') \quad \text{definition of } \sim \\
& \Rightarrow & \forall x \in \Sigma^* \cdot ([\widehat{\delta}(p, x)] \in F' \iff [\widehat{\delta}(q, x)] \in F') \quad \text{Lemma 4.4.3} \\
& \Rightarrow & \forall x \in \Sigma^* \cdot (\widehat{\delta}(p, x) \in F \iff \widehat{\delta}(q, x) \in F \quad \text{Lemma 4.4.2} \\
& \Rightarrow & p \approx q \quad \text{definition of } \approx \\
& \Rightarrow & [p] = [q]
\end{array}$$

Thus any two equivalent states of $M/\approx$ are in fact equal, and the collapsing relation $\sim$ on Q' is just the identity relation $=$.

4.4.2 A Minimisation Algorithm

Here is an algorithm for computing the collapsing relation $\approx$ for a given DFA M with no inaccessible states. Our algorithm will mark (unordered) pairs of states $\{p, q\}$. A pair $\{p, q\}$ will be marked as soon as a reason can be discovered why p and q are *not* equivalent.

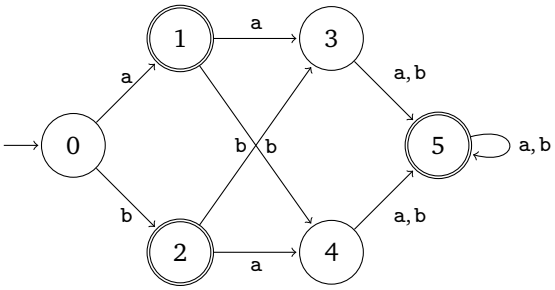
1. Write down a table of all pairs $\{p, q\}$, initially unmarked.
2. Mark $\{p, q\}$ if $p \in F$ and $q \notin F$ or vice versa.
3. Repeat the following until no more changes occur: if there exists an unmarked pair $\{p, q\}$ such that $\{\delta(p, a), \delta(q, a)\}$ is marked for some $a \in \Sigma$, then mark $\{p, q\}$.
4. When done, $p \approx q$ iff $\{p, q\}$ is not marked.

Here are some things to note about the algorithm:

- If $\{p, q\}$ is marked in step 2, then p and q are surely not equivalent: take $x = \epsilon$ in the definition of $\approx$.
- We may have to look at the pair $\{p, q\}$ many times in step 3, since any change in the table may suddenly allow $\{p, q\}$ to be marked. We stop only after we make an entire pass through the table with no new marks.

- The algorithm runs for only a finite number of steps, since there are only $\binom{n}{2}$ possible marks that can be made,[1] and we have to make at least one new mark in each pass to keep going.
- Step 4 is really a statement of the theorem that the algorithm correctly computes $\approx$. This requires proof.

Example 4.4.5 *Let's minimise the following automaton:*



Here is its transition relation written more compactly:

		a	b
$\rightarrow$	0	1	2
	$1F$	3	4
	$2F$	4	3
	3	5	5
	4	5	5
	$5F$	5	5

Here is the table built in step 1. Initially all pairs are unmarked.

0					
$-$	1				
$-$	$-$	2			
$-$	$-$	$-$	3		
$-$	$-$	$-$	$-$	4	
$-$	$-$	$-$	$-$	$-$	5

After step 2, all pairs consisting of one accepting state and one non-accepting state have been marked:

0					
$\surd$	1				
$\surd$	$-$	2			
$-$	$\surd$	$\surd$	3		
$-$	$\surd$	$\surd$	$-$	4	
$\surd$	$-$	$-$	$\surd$	$\surd$	5

[1] $\binom{n}{k} \mathrel{\widehat{=}} \frac{n!}{k!(n-k!)}$, the number of subsets of size k in a set of size n.

Now look at an unmarked pair, say $\{0,3\}$*. Under input* a*,* 0 *and* 3 *go to* 1 *and* 5*, respectively. (Write* $\{0,3\} \to \{1,5\}$*). The pair* $\{1,5\}$ *is not marked, so we do not mark* $\{0,3\}$*, at least not yet. Under input* b*,* $\{0,3\} \to \{2,5\}$*, which is not marked, so we do not mark* $\{0,3\}$*. We then look at unmarked pairs* $\{0,4\}$ *and* $\{1,2\}$ *and find out that we cannot mark them yet for the same reasons. But for* $\{1,5\}$*, under input* a*,* $\{1,5\} \to \{3,5\}$*, and* $\{3,5\}$ *is marked, so we mark* $\{1,5\}$*. Similarly, under input* a*,* $\{2,5\} \to \{4,5\}$ *which is marked, so we mark* $\{2,5\}$*. Under both inputs* a *and* b*,* $\{3,4\} \to \{5,5\}$*, which is never marked (it's not even in the table), so we do not mark* $\{3,4\}$*. After the first pass of step 3, the table looks like*

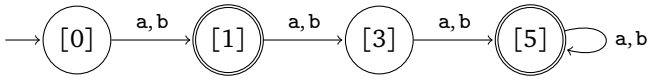
0					
√	1				
√	−	2			
−	√	√	3		
−	√	√	−	4	
√	√	√	√	√	5

Now we make another pass through the table. As before, $\{0,3\} \to \{1,5\}$ *under input* a*, but this time* $\{1,5\}$ *is marked, so we mark* $\{0,3\}$*. Similarly,* $\{0,4\} \to \{2,5\}$ *under input* b*, and* $\{2,5\}$ *is marked so we mark* $\{0,4\}$*. This gives:*

0					
√	1				
√	−	2			
√	√	√	3		
√	√	√	−	4	
√	√	√	√	√	5

Now we check the remaining unmarked paris and find out that $\{1,2\} \to \{3,4\}$ *and* $\{3,4\} \to \{5,5\}$ *under both* a *and* b*, and neither* $\{3,4\}$ *nor* $\{5,5\}$ *is marked, so there are no new marks. We are left with unmarked pairs* $\{1,2\}$ *and* $\{3,4\}$*, indicating that* $1 \approx 2$ *and* $3 \approx 4$*.*

The resulting quotient automaton is:



4.4.3 Correctness of the Collapsing Algorithm

Theorem 4.4.6 *The pair* $\{p,q\}$ *is marked by the above algorithm if and only if there exists* $x \in \Sigma^*$ *such that* $\widehat{\delta}(p,x) \in F$ *and* $\widehat{\delta}(q,x) \notin F$ *or vice versa; that is, if and only if* $p \not\approx q$*.*

Proof. Suppose first that $\{p,q\}$ is marked. Then proceed by induction on the stages of the algorithm. If $\{p,q\}$ is marked in step 2, then either $p \in F$ or $q \notin F$ or vice versa, therefore $p \not\approx q$ (take $x = \epsilon$ in the definition of $\approx$). If it is marked in step 3, then for some $a \in \Sigma$, $\{\delta(p,a), \delta(q,a)\}$ was marked at some earlier stage. By the induction hypothesis, $\delta(p,a) \not\approx \delta(q,a)$, therefore $p \not\approx q$ by Lemma 4.4.1.

Conversely, suppose $p \not\approx q$. By definition, there exists an $x \in \Sigma^*$ such that either $\widehat{\delta}(p,x) \in F$ and $\widehat{\delta}(q,x) \notin F$ or vice versa. We proceed by induction on the length of x. If $x = \epsilon$, then $p \in F$ and $q \notin F$ or vice versa, so $\{p,q\}$ is marked in step 2. If $x = ay$, then either $\widehat{\delta}(\delta(p,a),y) \in F$ and $\widehat{\delta}(\delta(q,a),y) \notin F$ or vice versa. By the induction hypothesis, $\{\delta(p,a), \delta(q,a)\}$ is eventually marked by the algorithm and $\{p,q\}$ will be marked in the following step. □

A nice way to look at the algorithm is as a finite automaton itself. Let

$$\mathcal{Q} = \{\{p, q\} \mid p, q \in Q, p \neq q\}.$$

There are $\binom{n}{2}$ elements of $\mathcal{Q}$, where n is the size of Q. Define a nondeterministic "transition function"

$$\Delta : \mathcal{Q} \to \mathcal{P}(\mathcal{Q})$$

on $\mathcal{Q}$ as follows:

$$\Delta(\{p, q\}, a) = \{\{p', q'\} \mid p = \delta(p', a), q = \delta(q', a)\}.$$

Define a set of "start states" $\mathcal{S} \subseteq \mathcal{Q}$ as follows:

$$\mathcal{S} = \{\{p, q\} \mid p \in F, q \notin F\}.$$

(We don't need to write "... or vice versa" because $\{p, q\}$ is an unordered pair.) Step 2 of the algorithm marks the elements of $\mathcal{S}$, and step 3 marks in $\Delta(\{p, q\}, a)$ when $\{p, q\}$ is marked for any $a \in \Sigma$. In these terms, Theorem 4.4.6 says that $p \not\approx q$ iff $\{p, q\}$ is accessible in this automaton.

4.5 Closure Properties

The class of regular languages is closed under the operations of union, concatenation, and Kleene star, described in Section 3.1. We demonstrate these properties using constructions on NFAs.

Theorem 4.5.1 *The class of regular languages is closed under the union operation.*

Proof. Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognise A_1 and $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ recognise A_2.

The following construction adds a new start state and ϵ transitions from this state to the start states of N_1 and N_2.

Construct $N = (Q, \Sigma, \delta, q_0, F)$ to recognise $A_1 \cup A_2$.

1. $Q = \{q_0\} \cup Q_1 \cup Q_2$.
   The states of N are all the states of N_1 and N_2, with the addition of a new start state q_0.

2. The state q_0 is the start state of N.

3. Accepting states $F = F_1 \cup F_2$.
   The accepting states of N are all the accepting states of N_1 and N_2. That way N accepts if either N_1 accepts or N_2 accepts.

4. Define δ so that for any $q \in Q$ and any $a \in \Sigma_\epsilon$,

$$\delta(q, a) = \begin{cases} \delta_1(q, a) & \text{if } q \in Q_1 \\ \delta_2(q, a) & \text{if } q \in Q_2 \\ \{q_1, q_2\} & \text{if } q = q_0 \text{ and } a = \epsilon \\ \emptyset & \text{if } q = q_0 \text{ and } a \neq \epsilon. \end{cases}$$

□

Theorem 4.5.2 *The class of regular languages is closed under the concatenation operation.*

Proof. Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognise A_1, and $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ recognise A_2.

The following construction adds ϵ transitions from the accepting states of N_1 to the start states of N_2.

Construct $N = (Q, \Sigma, \delta, q_1, F_2)$ to recognise A_1A_2.

1. $Q = Q_1 \cup Q_2$.
   The states of N are all the states of N_1 and N_2.
2. The state q_1 is the same as the start state of N_1.
3. The accepting states F_2 are the same as the accepting states of N_2
4. Define δ so that for any $q \in Q$ and any $a \in \Sigma_\epsilon$,

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & \text{if } q \in Q_1 \text{ and } q \notin F_1 \\ \delta_1(q,a) & \text{if } q \in F_1 \text{ and } a \neq \epsilon \\ \delta_1(q,a) \cup \{q_2\} & \text{if } q \in F_1 \text{ and } a = \epsilon \\ \delta_2(q,a) & \text{if } q \in Q_2. \end{cases}$$

□

Theorem 4.5.3 *The class of regular languages is closed under the Kleene star operation.*

Proof. Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognise A_1.

The following construction adds a new start state, ϵ transitions from that state to the start states of N_1 and ϵ transitions from the accepting states of N_1 to the new start state.

Construct $N = (Q, \Sigma, \delta, q_0, F)$ to recognise A_1^*.

1. $Q = \{q_0\} \cup Q$.
   The states of N are the states of N_1 plus a new start state.
2. The state q_0 is the new start state.
3. $F = \{q_0\} \cup F_1$.
   The accepting states are the old accepting states plus the new start state.
4. Define δ so that for any $q \in Q$ and any $a \in \Sigma_\epsilon$,

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & \text{if } q \in Q_1 \text{ and } q \notin F_1 \\ \delta_1(q,a) & \text{if } q \in F_1 \text{ and } a \neq \epsilon \\ \delta_1(q,a) \cup \{q_1\} & \text{if } q \in F_1 \text{ and } a = \epsilon \\ \{q_1\} & \text{if } q = q_0 \text{ and } a = \epsilon \\ \emptyset & \text{if } q = q_0 \text{ and } a \neq \epsilon. \end{cases}$$

□

Exercise 4.5.4 *Consider whether regular languages (over some alphabet Σ) are closed under*

- *complementation*
- *set difference*
- *intersection.*

4.6 Regular Expressions

The regular operations can be used to build expressions describing languages, which are called *regular expressions*. An example is:

$$(0 \cup 1)0^*$$

The value of a regular expression is a language. For the example above, the language is the set of all string starting with either a 0 or a 1 followed by any number of 0s. First, the symbols 0 and 1 are short for the sets $\{0\}$ and $\{1\}$. So $(0 \cup 1)$ means $(\{0\} \cup \{1\})$, which is the language $\{0, 1\}$. The part 0^* means $\{0\}^*$, and its value is the language consisting of all the strings containing any number of 0s. The final ingredient is the concatenation of $(0 \cup 1)$ and 0^*, which attaches the strings from the two parts to obtain the value for the entire expression.

Definition 4.6.1 (Regular Expression) *We say that R is a* regular expression *if R is*

1. $\mathtt{a}$ *for some* $\mathtt{a}$ *in the alphabet* Σ*,*
2. ϵ*,*
3. $\emptyset$*,*
4. $(R_1 \cup R_2)$*, where R_1 and R_2 are regular expressions,*
5. $(R_1 R_2)$*, where R_1 and R_2 are regular expressions, or*
6. (R_1^*)*, where R_1 is a regular expression.*

In items 1 and 2, the regular expressions $\mathtt{a}$ *and ϵ represent the languages* $\{\mathtt{a}\}$ *and $\{\epsilon\}$, respectively. In item 3, the regular expression $\emptyset$ represents the empty language. In items 4, 5, and 6, the expressions represent the languages obtained by taking the union or concatenation of the languages R_1 and R_2, or the Kleene star of the language R_1, respectively.*

Note that the regular expressions ϵ and $\emptyset$ denote different languages. ϵ denotes the language containing a single string, namely, the empty string, thus ϵ denotes the language $\{\epsilon\}$, whereas $\emptyset$ denotes the language that does not contain any strings.

Parentheses in an expression may be omitted. If they are, evaluation is done in the precedence order: Kleene star, then concatenation, then union.

For convenience, we let R^+ be shorthand for RR^*, which is the language of 1 or more concatenations of strings from R. So $R^* = R^+ \cup \epsilon$. In addition, we let R^k be shorthand for the concatenation of k Rs with each other.

When we want to distinguish between a regular expression R and the language it describes, we write $L(R)$ for the language of R.

The language of a regular expression can be calculated using the follow recursive function:

$$\begin{aligned} L(a) &= \{a\} \\ L(\epsilon) &= \{\epsilon\} \\ L(\emptyset) &= \emptyset \\ L(R_1 \cup R_2) &= L(R_1) \cup L(R_2) \\ L(R_1 R_2) &= L(R_1) L(R_2) \\ L(R^*) &= L(R)^* \end{aligned}$$

On the left hand side are syntactic elements, from the language of regular expressions, and on the right hand side are semantic elements, namely, operations on languages.

4.7 Equivalence of Regular Expressions and Finite State Machines

In this section we show that regular expressions and finite automata are equivalent in their descriptive power. This fact is surprising because finite automata and regular expressions superficially appear to be rather different. However, any regular expression can be converted into a finite automaton that recognises the language it describes and vice versa.

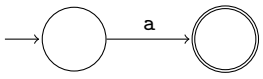
Theorem 4.7.1 *A language is regular if and only if some regular expression describes it.*

This theorem has two directions. We state and prove each as a separate lemma.

Lemma 4.7.2 *If a language is described by a regular expression, then it is regular.*

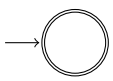
Proof. Let's convert R into an NFA N. We consider the six cases in the formal definition of regular expressions.

- $R = \mathtt{a}$ for some $\mathtt{a} \in \Sigma$. Then $L(R) = \{\mathtt{a}\}$, and the following NFA recognises $L(R)$.



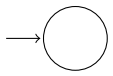
  Formally, $N = (\{q_1, q_2\}, \Sigma, \delta, q_1, \{q_2\})$, where we describe δ by saying that $\delta(q_1, a) = \{q_2\}$ and that $\delta(r, b) = \emptyset$ for $r \neq q_1$ and $b \neq a$.

- $R = \epsilon$. Then $L(R) = \{\epsilon\}$, and the following NFA recognises $L(R)$.



  Formally, $N = (\{q_1\}, \Sigma, \delta, q_1, \{q_1\})$, where we describe δ by saying that $\delta(r, b) = \emptyset$ for any r and b.

- $R = \emptyset$. Then $L(R) = \emptyset$, and the following NFA recognises $L(R)$.



  Formally, $N = (\{q_1\}, \Sigma, \delta, q_1, \emptyset)$, where we describe δ by saying that $\delta(r, b) = \emptyset$ for any r and b.

- $R = R_1 \cup R_2$, $R = R_1 R_2$, $R = R_1^*$. For these cases, we use the constructions given in the proofs (Section 4.5) that the class of regular languages is closed under the regular operations.

□

Lemma 4.7.3 *If a language is regular, then it is described by a regular expression.*

In order to do this proof, we introduce the following notation. Let $M = (Q, \Sigma, \delta, q_1, F)$ be a DFA. Define a *configuration* of M to be an element of $Q \times \Sigma^*$. The relation $\vdash_M$ between configurations (*yields in one step*) is defined as follows: $(q, w) \vdash_M (q', w')$ if and only if $w = aw'$ and $\delta(q, a) = q'$. Now define $\vdash_M^*$ as the reflexive transitive closure of $\vdash_M$.[2]

Proof. Let $M = (Q, \Sigma, \delta, q_1, F)$ be a DFA. We need to show that there is a regular language R such that $R = L(M)$. We represent $L(M)$ as the union of many (but a finite number of) simple languages. Assume that $Q = \{q_1, \ldots, q_n\}$. For $i, j = 1, \ldots, n$ and $k = 1, \ldots, n+1$, we let $R(i, j, k)$ be the set of all strings in Σ^* that drive M from q_i to q_j without passing through any state numbered k or greater. Formally,

$$\begin{aligned} R(i,j,k) \quad = \quad & \{x \in \Sigma^* \mid (q_i, x) \vdash_M^* (q_j, \epsilon) \ \wedge \\ & ((q_i, x) \vdash_M^* (q_l, y) \Rightarrow \ (l < k \ \ \vee \ \ y = \epsilon \wedge l = j \ \ \vee \ y = x \wedge l = i))\}. \end{aligned}$$

When $k = n + 1$, it follows that

$$R(i, j, n+1) \quad = \quad \{x \in \Sigma^* \mid (q_i, x) \vdash_M^* (q_j, \epsilon)\}.$$

Therefore,

$$L(M) = \bigcup_{q_j \in F} R(1, j, n+1)$$

The crucial point is that each set $R(i, j, k)$ is regular, and hence so is $L(M)$. The proof is by induction on k. For $k = 1$, we have the following:

$$R(i,j,1) \quad = \quad \begin{cases} \{a \in \Sigma \mid \delta(q_i, a) = q_j\} & \text{if } i \neq j \\ \{\epsilon\} \cup \{a \in \Sigma \mid \delta(q_i, a) = q_i\} & \text{if } i = j. \end{cases}$$

Each of these sets is finite and therefore regular. For $k = 1, \ldots, n$, provided that all the sets $R(i, j, k)$ have been defined, each set $R(i, j, k+1)$ can be defined in terms of the previously defined languages as follows:

$$R(i,j,k+1) \quad = \quad R(i,j,k) \cup R(i,k,k) R(k,k,k)^* R(k,j,k).$$

This equation states that to get from q_i to q_j without passing through a state numbered greater than k, M may either:

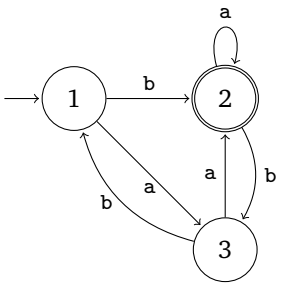
[2] A variant for NFA can easily be defined by stating: $(q, w) \vdash_M (q', w')$ if and only if $\exists u \in \Sigma_\epsilon$ such that $w = uw'$ and $q' \in \delta(q, u)$.

1. go from q_i to q_j without passing through a state numbered greater than $k-1$; or
2. go
   (a) from q_i to q_k; then
   (b) from q_k to q_k repeatedly; then
   (c) from q_k to q_j;

   in each case without passing through a state numbered greater than $k-1$.

Therefore if the language $R(i,j,k)$ is regular, so is each language $R(i,j,k+1)$. This completes the induction. □

Example 4.7.4 *Here is a complete worked example. Let* $\Sigma = \{\mathtt{a}, \mathtt{b}\}$.



Now, let's exhaustively compute most functions $R(i,j,k)$*:*

$$\begin{aligned}
R(1,1,1) &= \epsilon \\
R(1,2,1) &= \mathtt{b} \\
R(1,3,1) &= \mathtt{a} \\
R(2,1,1) &= \emptyset \\
R(2,2,1) &= \epsilon \cup \mathtt{a} \\
R(2,3,1) &= \mathtt{b} \\
R(3,1,1) &= \mathtt{b} \\
R(3,2,1) &= \mathtt{a} \\
R(3,3,1) &= \epsilon
\end{aligned}$$

$$\begin{aligned}
R(1,2,2) &= R(1,2,1) \cup R(1,1,1)R(1,1,1)^*R(1,2,1) \\
&= \mathtt{b} \cup \epsilon\epsilon^*\mathtt{b} \\
&= \mathtt{b}
\end{aligned}$$

$$\begin{aligned}
R(2,2,2) &= R(2,2,1) \cup R(2,1,1)R(1,1,1)^*R(1,2,1) \\
&= (\epsilon \cup \mathtt{a}) \cup \emptyset\epsilon^*\mathtt{b}
\end{aligned}$$

$$
\begin{aligned}
&= \epsilon \cup \mathtt{a} \\
\\
R(1,3,2) &= R(1,3,1) \cup R(1,1,1)R(1,1,1)^*R(1,3,1) \\
&= \mathtt{a} \cup \epsilon\epsilon^*\mathtt{a} \\
&= \mathtt{a} \\
\\
R(2,3,2) &= R(2,3,1) \cup R(2,1,1)R(1,1,1)^*R(1,3,1) \\
&= \mathtt{b} \cup \emptyset\epsilon^*\mathtt{a} \\
&= \mathtt{b} \\
\\
R(3,2,2) &= R(3,2,1) \cup R(3,1,1)R(1,1,1)^*R(1,2,1) \\
&= \mathtt{a} \cup \mathtt{b}\epsilon^*\mathtt{b} \\
&= \mathtt{a} \cup \mathtt{bb} \\
\\
R(3,3,2) &= R(3,3,1) \cup R(3,1,1)R(1,1,1)^*R(1,3,1) \\
&= \epsilon \cup \epsilon\mathtt{b}^*\mathtt{a} \\
&= \mathtt{ba} \\
\\
R(1,2,3) &= R(1,2,2) \cup R(1,2,2)R(2,2,2)^*R(2,2,2) \\
&= \mathtt{b} \cup \mathtt{b}(\epsilon \cup \mathtt{a})^*(\epsilon \cup \mathtt{a}) \\
&= \mathtt{b} \cup \mathtt{ba}^* \\
\\
R(1,3,3) &= R(1,3,2) \cup R(1,2,2)R(2,2,2)^*R(2,3,2) \\
&= \mathtt{a} \cup \mathtt{b}(\epsilon \cup \mathtt{a})^*\mathtt{b} \\
&= \mathtt{a} \cup \mathtt{ba}^*\mathtt{b} \\
\\
R(3,2,3) &= R(3,2,2) \cup R(3,2,2)R(2,2,2)^*R(2,2,2) \\
&= (\mathtt{a} \cup \mathtt{bb}) \cup \mathtt{a} \cup \mathtt{bb}(\epsilon \cup \mathtt{a})^*(\epsilon \cup \mathtt{a}) \\
&= (\mathtt{a} \cup \mathtt{bb}) \cup \mathtt{a} \cup \mathtt{bba}^* \\
\\
R(3,3,3) &= R(3,3,2) \cup R(3,2,2)R(2,2,2)^*R(2,3,2) \\
&= \mathtt{ba} \cup (\mathtt{a} \cup \mathtt{bb})(\epsilon \cup \mathtt{a})^*\mathtt{b} \\
&= \mathtt{ba} \cup (\mathtt{a} \cup \mathtt{bb})\mathtt{a}^*\mathtt{b} \\
\\
L(M) &= R(1,2,4) \\
&= R(1,2,3) \cup R(1,3,3)R(3,3,3)^*R(3,2,3) \\
&= (\mathtt{b} \cup \mathtt{ba}^*) \cup (\mathtt{a} \cup \mathtt{ba}^*\mathtt{b})(\mathtt{ba} \cup (\mathtt{a} \cup \mathtt{bb})\mathtt{a}^*\mathtt{b})^*((\mathtt{a} \cup \mathtt{bb}) \cup \mathtt{a} \cup \mathtt{bba}^*)
\end{aligned}
$$

$$= \quad \mathtt{ba}^* \cup (\mathtt{a} \cup \mathtt{ba}^*\mathtt{b})(\mathtt{ba} \cup (\mathtt{a} \cup \mathtt{bb})\mathtt{a}^*\mathtt{b})^*(\mathtt{a} \cup \mathtt{bba}^*)$$

Note that the simplification steps require individual verification. This could be done in an ad hoc fashion on a per case basis or by using Kleene algebra (Section 4.10).

4.8 Pumping Lemmas

In order to understand a class of languages (such as regular languages), you must also understand their limitations. In this section we show that certain languages cannot be recognised by any finite automaton.

Consider the language $B = \{0^n 1^n \mid n \geq 0\}$. If we attempt to find a DFA that recognises B, we discover that the machine seems to need to remember how many 0s have been seen so far as it reads the input. Because the number of 0s is not limited, the machine will have to keep track of an unlimited number of possibilities. But it cannot do so with any finite number of states.

Next we present a method for proving that languages such as B are not regular. Note that the above argument is not a proof. Just because it appears that the automata needs to account, does not mean that this is the case. Consider the following two languages over the alphabet $\Sigma = \{0, 1\}$:

$$\begin{aligned} C &= \{w \mid w \text{ has an equal number of 0s and 1s}\} \\ D &= \{w \mid w \text{ has an equal number of occurrences of 01 and 10 as substrings}\} \end{aligned}$$

At first glance a recognising machine appears to need to count in each case, and therefor neither language appears to be regular. As expected, C is not regular, but surprisingly D is regular. Thus our intuition can sometimes lead us astray, which is why we need mathematical proofs for certainty. In this section we show how to prove that certain languages are not regular.

Exercise 4.8.1 *Let* $\Sigma = \{0, 1\}$ *and let*

$$D = \{w \mid w \text{ has an equal number of occurrences of } 01 \text{ and } 10 \text{ as substrings}\}.$$

Thus $101 \in D$*, because* 101 *contains a single* 01 *and a single* 10*, but* $1010 \notin D$*, because* 1010 *contains two* 10*s and only one* 01*. Show that D is a regular language.*

Our techniques for proving non-regularity stems from a theorem about regular languages, traditionally called the *pumping lemma*. The theorem states that all regular languages have a special property. If we can show that a language does not have this property, we are guaranteed that it is not regular. The property states that all strings in the language can be "pumped" if they are at least as long as a certain value, called the *pumping length*. This means that each such string contains a section that can be repeated any number of times with the resulting string remaining in the language.

Theorem 4.8.2 (Pumping Lemma) *If A is a regular language, then there is at least a number p (the* pumping length*) where, if s is any string in A of length at least p, then s may be divided into three pieces,* $s = xyz$*, satisfying the following conditions:*

1. *for each* $i \geq 0$*,* $xy^iz \in A$*,*
2. $|y| > 0$*, and*

3. $|xy| \leq p$.

Proof Idea. Let $M = (Q, \Sigma, \delta, q_1, F)$ be a DFA that recognises A. We assign the pumping length p to be the number of states of M. We show that any string s in A length at least p can be broken into three pieces xyz satisfying our three conditions. What if no strings in A are of length at least p? Then our task is even easier, because the theorem becomes vacuously true: obviously the three conditions hold for all strings of length at least p if there are no such strings.

If s in A has length at least p, consider the sequence of states that M goes through when computing with input s. It starts with q_1 the start state, then goes to, say, q_2, then say q_3, then q_4, and so on, until it reaches the end of s in state q_n. With s in A, we know that M accepts s, so q_{n+1} is an accepting state, where n is the length of s.

Consider the sequence of states $q_1, q_2, \ldots, q_{n+1}$. Because n is a least p, we know that $n+1$ is greater than p, the number of states of M. Therefore the sequence must contain a repeated state. This result is an example of the *pigeonhole principle*, the fact that if p pigeons are placed into few than p holes, some hole has to have more than one pigeon in it.

Let q' be some state appearing twice in $q_1, q_2, \ldots, q_{n+1}$. We now divide s into three pieces, x, y, and z. The piece x is the part of s appearing before the first occurrence of q', y is the part between the two occurrences of q' and z is the remaining part of s, coming after the second occurrence of q'.

Let's see why this division of s satisfies the three conditions. Suppose that we run M on input $xyyz$. We know that x takes M from q_1 to q', and then the first y takes it back to q', as does the second, and then z takes it to accepting state q_{n+1}. Similarly, it will accept xy^iz for any $i > 0$. For the case $i = 0$, $xy^iz = xz$, which is accepted for similar reasons. This establishes condition 1.

Checking condition 2, we see that $|y| > 0$, as it was the part of s that occurred between two difference occurrences of state q'.

In order to get condition 3, we make sure that q' is the first repetition in the sequence. By the pigeon hole principle, the first $p+1$ states in the sequence must contain a repetition, therefore $|xy| \leq p$. □

Proof. Let $M = (Q, \Sigma, \delta, q_1, F)$ be a DFA recognising A and p be the number of states of M.

Let $s = s_1 s_2 \cdots s_n$ be a string in A of length n, where $n \geq p$. Let $r_1, \ldots, r_{n+1}$ be the sequence of states that M enters while processing s, so $r_{i+1} = \delta(r_i, s_i)$ for $1 \leq i \leq n$. This sequence has length $n+1$, which is at least $p+1$. Among the first $p+1$ elements in the sequence, two must be the same state, by the pigeonhole principle. We call the first of these r_j and the second r_l. Because r_l occurs among the first $p+1$ places in a sequence starting at r_1, we have $l \leq p+1$. Now let $x = s_1 \cdots s_{j-1}$, $y = s_j \cdots s_{l-1}$, and $z = s_l \cdots s_n$.

As x takes M from r_1 to r_j, y takes M from r_j to r_j, and z takes M from r_j to r_{n+1}, which is an accepting state, M must accept xy^iz for i. We know that $j \neq l$, so $|y| \geq 0$; and $l \leq p+1$, so $|xy| \leq p$. Thus we have satisfied all the conditions of the pumping lemma. □

4.9 Brzozowski's Derivative

Here we introduce an alternative approach to constructing a finite state machine from a regular expression. This approach is due to Brzozowski [1] (see also [2]) and is based on the idea of taking *derivatives of regular expressions*.

Definition 4.9.1 (Derivative) *If $S \subseteq \Sigma^*$ is a set of strings and $s \in \Sigma^*$, then define the* derivative of S with respect to s, *denoted D_sS, as $D_sS = \{t \mid st \in S\}$.*

The derivative D_sS of set S with respect to string s is the set of suffixes of strings that start with s, after we have removed the prefix s. For example, if $S = \{\mathtt{a}, \mathtt{ab}, \mathtt{ac}, \mathtt{ba}, \mathtt{abc}, \mathtt{bac}\}$, then $D_{\mathtt{a}}S = \{\epsilon, \mathtt{b}, \mathtt{c}, \mathtt{bc}\}$.

We now develop the notion of derivative of a regular expression. Firstly, we need to know whether the regular expression contains the empty string ϵ.

Definition 4.9.2 *Given a set of strings $S \subseteq \Sigma^*$, define $\delta(S)$ to be*

$$\delta(S) \quad = \quad \begin{cases} \epsilon & \epsilon \in S \\ \emptyset & \epsilon \notin S. \end{cases}$$

It is clear that $\delta(a) = \emptyset$ for any $a \in \Sigma$, $\delta(\epsilon) = \epsilon$, and $\delta(\emptyset) = \emptyset$. Furthermore, $\delta(R^*) = \epsilon$, $\delta(R_1R_2) = \delta(R_1)\delta(R_2)$, and $\delta(R_1 \cup R_2) = \delta(R_1) \cup \delta(R_2)$.

Exercise 4.9.3 *Prove that the above definition of δ is correct for regular expressions. That is, for any regular expression R, $\delta(R) = \epsilon$ if and only if $\epsilon \in L(R)$.*

Theorem 4.9.4 *If R is a regular expression, the derivative of R with respect to $a \in \Sigma$ is defined recursively as follows:*

$$\begin{aligned} D_a(a) &= \epsilon \\ D_a(b) &= \emptyset, \qquad a \neq b \\ D_a(\epsilon) &= \emptyset \\ D_a(\emptyset) &= \emptyset \\ D_a(R^*) &= (D_aR)R^* \\ D_a(R_1R_2) &= (D_aR_1)R_2 \cup \delta(R_1)D_aR_2 \\ D_a(R_1 \cup R_2) &= D_aR_1 \cup D_aR_2 \end{aligned}$$

Exercise 4.9.5 *Prove Theorem 4.9.4.*

The derivative of a regular expression R with respect to a finite string $s = a_1a_2 \cdots a_n$ is defined recursively as follows:

$$\begin{aligned} D_\epsilon R &= R \\ D_{aw}R &= D_w(D_aR) \end{aligned}$$

Example 4.9.6 *Consider the regular expression* $\mathtt{ab} \cup (\mathtt{abc})^*$.

$$\begin{aligned} D_{\mathtt{a}}(\mathtt{ab} \cup (\mathtt{abc})^*) &= D_{\mathtt{a}}(\mathtt{ab}) \cup D_{\mathtt{a}}((\mathtt{abc})^*) \\ &= D_{\mathtt{a}}(\mathtt{a})\mathtt{b} \cup \delta(\mathtt{a})D_{\mathtt{a}}(\mathtt{b}) \cup D_{\mathtt{a}}(\mathtt{abc})(\mathtt{abc})^* \\ &= \epsilon\mathtt{b} \cup \emptyset\emptyset \cup (D_{\mathtt{a}}(\mathtt{a})\mathtt{bc} \cup \delta(\mathtt{a})D_{\mathtt{a}}(\mathtt{bc}))(\mathtt{abc})^* \\ &= \mathtt{b} \cup \mathtt{bc}(\mathtt{abc})^* \end{aligned}$$

We now give some properties of derivatives.

Theorem 4.9.7 *The derivative D_s of any regular expression R with respect to any string s is a regular expression.*

Proof. By construction, using induction on the length of s. □

Theorem 4.9.8 *A string s is contained in a regular expression R if and only if ϵ is contained in D_sR.*

Proof. If $\epsilon \in D_sR$, then $s\epsilon = s \in R$ from Definition 4.9.1. Conversely, if $s \in R$, then $s\epsilon \in R$ and $\epsilon \in D_sR$, again from Definition 4.9.1. □

Theorem 4.9.8 reduces the problem of testing whether a string s is contained in a regular expression R to the problem of testing whether ϵ is contained in D_sR. The latter is solved through the use of $\delta(D_sR)$.

Two regular expressions that are equal (but not necessarily identical in form) will be said to be of the same *type*.

Theorem 4.9.9 *(a) Every regular expression R has a finite number d_R of types of derivatives. (b) At least one derivative of each type must be found among the derivatives with respect to sequences of length not exceeding d_{R-1}.*

Proof. See Brzozowski [1]. □

This assumes that it is possible to decide when two derivatives are of the same type. This is not always an easy problem, but it can be resolved, as we show shortly.

Theorem 4.9.10 *Every regular expression R can be written in the form*

$$R \quad = \quad \delta(R) \cup \sum_{a \in \Sigma} aD_aR,$$

where the terms in the sum are disjoint.

Proof. See Brzozowski [1]. □

Theorem 4.9.11 *The relationship between the d_R characteristic derivatives of R can be represented by a unique set of d_R equations of the form*

$$D_sR \quad = \quad \delta(D_sR) \cup \sum_{a \in \Sigma} aD_{u_a}R,$$

where D_sR is a characteristic derivative and $D_{u_a}R$ is a characteristic derivative equal to D_{sa}. Such equations are called the characteristic equations of *R.*

Proof. Theorem follows directly from Theorems 4.9.9 and 4.9.10. □

Theorem 4.9.12 *An equation of the form $X = AX \cup B$, where $\delta(A) = \emptyset$, has the solution $X = A^*B$, which is unique (up to equality of regular expressions).*

Theorem 4.9.13 *The set of characteristic equations of R can be solved for R uniquely (up to equality).*

Proof. See Brzozowski [1]. □

Thus the regular expression can always be reconstructed from the characteristic equations (although it may be in a different form, depending on the order of elimination of the derivatives from the equation).

Now we move on to showing how to use derivatives to construct a finite state machine for a regular expression.

Definition 4.9.14 *A string s is accepted by state q_i of a finite state automaton M iff when M is started in q_i the state at the end of s is accepting.*

Two states q_j and q_k of M are *indistinguishable* iff every string s applied to M started in state q_j produces the same output (accepting or not) as that produced by applying s to M started in q_k.

Theorem 4.9.15 *Two states q_j and q_k of M are indistinguishable iff R_j and R_k denoting the sets of strings accepted by q_j and q_k are equal.*

This theorem can equivalently be stated as follows:

Theorem 4.9.16 *Two states q_j and q_k of an automaton M characterised by the regular expression R are indistinguishable iff their derivatives are equal, that is $D_{s_j}R = D_{s_k}R$, where s_j and s_k are any two strings taking M from the initial state to q_j and q_k respectively.*

There is a very close relationship between derivatives of a regular expression and the states of the corresponding finite automaton. These results are summarised as follows:

1. To obtain a regular expression from a finite state machine, write the characteristic set of equations and solve for R. There is one equation for each state. If the transition under input a takes state q_j to q_k, then the equation for R_j contains the term aR_k; and ϵ is a term of R_j if and only if q_j is an accepting state.

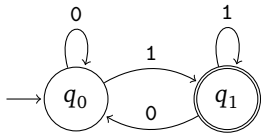
2. To obtain the minimal finite state machine from a regular expression, find the characteristic derivatives and associate one internal state with each characteristic derivative. Whether a state is accepting corresponds to whether the characteristic derivative contains ϵ.

This procedure is illustrated in the following example.

Example 4.9.17 *Let $R = (0 \cup 1)^*1$. Then:*

$D_\epsilon R = R$: *introduce q_0, not accepting as $\delta(R) = \emptyset$,*
$D_0 R = R$: *return to q_0 under input 0,*
$D_1 R = R \cup \epsilon$: *introduce q_1, accepting as $\delta(D_1 R) = \epsilon$, and a transition from q_0 to q_1 under input 1.*
D_{00} and D_{01} need not be considered as D_0 does not correspond to a new characteristic derivative, that is, 0 returns the state graph to q_0.
Continuing, we find:
$D_{10}R = R$: *go from q_1 to q_0 under input 0,*
$D_{11}R = R \cup \epsilon$: *return from q_1 to q_1 under input 1.*

This completes the process. The resulting state machine is depicted as:



We have implicitly assumed that it is always possible to recognise the equality of two regular expressions. If this is always the case, then the state machine constructed by associating one internal state with each type of derivative is always minimal. However, it is often quite difficult to determine whether two regular expressions are equal. We now show that this difficulty can be overcome, and a state machine can always be constructed, but not necessarily with the minimum number of states.

Definition 4.9.18 *Two regular expressions are* similar *if one can be transformed to the other using only the identities:*

$$\begin{aligned} R \cup R &= R \\ R \cup Q &= Q \cup R \\ (P \cup Q) \cup R &= P \cup (Q \cup R). \end{aligned}$$

Two regular expressions are dissimilar *iff they are not similar.*

It is clear that similarity implies equality, but equality does not imply similarity. Similarity can be easily recognised and is much weaker than equality.

Theorem 4.9.19 *Every regular expression has only a finite number of dissimilar derivatives.*

Proof. See Brzozowski [1]. □

A consequence of this result is that a state machine can be constructed even if only similarity among the derivatives is recognised. However, such a method has a serious disadvantage since, in general, the automaton constructed will be far from minimal. This arises because of the frequent appearance of ϵ and $\emptyset$ in the derivatives. For example, consider $R = (0 \cup 1)^* 01$ and the derivative $D_1 R$.

$$\begin{aligned} D_1 R &= D_1((0 \cup 1)^*))01 \cup D_1(01) \\ &= (D_1(0 \cup 1))(0 \cup 1)^* 01 \cup (D_1 0)1 \\ &= (D_1 0 \cup D_1 1)(0 \cup 1)^* 01 \cup (D_1 0)1 \\ &= (\emptyset \cup \epsilon)(0 \cup 1)^* 01 \cup \emptyset 1 \end{aligned}$$

In this case, using only similarity we are forced to conclude that R and $D_1 R$ are dissimilar. However, the expression for $D_1 R$ can be easily simplified by the identities:

$$\begin{aligned} R \cup \emptyset &= \emptyset \cup R = R \\ R\emptyset &= \emptyset R = \emptyset \\ R\epsilon &= \epsilon R = R. \end{aligned}$$

These identities are very useful and will be incorporated into the method.

Exercise 4.9.20 *Determine the derivatives of the following operations on regular expressions:*

1. $R \cap S$, *where* $L(R \cap S) = L(R) \cap L(S)$.
2. $\neg R$, *where* $L(\neg R) = \Sigma^* \setminus L(R)$.

As mentioned above, we can use derivatives for matching a string against a regular expression. Suppose that R is a regular expression and w is a string. To determine whether $w \in L(R)$ we test whether $\epsilon \in L(D_w R)$, which is true exactly when $\epsilon = \delta(D_w R)$. Combining this fact with the definition of D_w leads to an algorithm for testing if $w \in L(R)$. We express the algorithm in terms of the relation $R \sim u$ (R matches string u), defined as the smallest relation satisfying (where $a \in \Sigma$):

$$\begin{aligned} R \sim \epsilon &\iff \delta(R) = \epsilon \\ R \sim aw &\iff D_a R \sim w \end{aligned}$$

It is straightforward to show that $R \sim w$ if and only if $w \in L(R)$.

Example 4.9.21 *When a regular expression matches a string, we compute a derivative for each of the characters in the string. Consider the derivation of* $\mathtt{ab}^* \sim \mathtt{abb}$*:*

$$\begin{aligned} \mathtt{ab}^* \sim \mathtt{abb} &\iff D_{\mathtt{a}}(\mathtt{ab}^*) \sim \mathtt{bb} \\ &\iff \mathtt{b}^* \sim \mathtt{bb} \\ &\iff D_{\mathtt{b}}(\mathtt{b}^*) \sim \mathtt{b} \\ &\iff \mathtt{b}^* \sim \mathtt{b} \\ &\iff D_{\mathtt{b}}(\mathtt{b}^*) \sim \epsilon \\ &\iff \mathtt{b}^* \sim \epsilon \\ &\iff \delta(\mathtt{b}^*) = \epsilon. \end{aligned}$$

Example 4.9.22 *When the regular expression does not match the string, we reach a derivative that is the regular expression of* $\emptyset$ *and stop. For example,*

$$\begin{aligned} \mathtt{ab}^* \sim \mathtt{aba} &\iff D_{\mathtt{a}}(\mathtt{ab}^*) \sim \mathtt{ba} \\ &\iff \mathtt{b}^* \sim \mathtt{ba} \\ &\iff D_{\mathtt{b}}(\mathtt{b}^*) \sim \mathtt{a} \\ &\iff \mathtt{b}^* \sim \mathtt{a} \\ &\iff D_{\mathtt{a}}(\mathtt{b}^*) \sim \epsilon \\ &\iff \emptyset \sim \epsilon \\ &\iff \delta(\emptyset) = \epsilon \qquad (\mathit{false}). \end{aligned}$$

4.10 Kleene Algebra

The structure underlying regular expressions and regular sets can be generalised into an algebraic structure known as *Kleene algebra*, named after Stephen Cole Kleene. Every Boolean algebra[3] is a

[3] A *Boolean algebra* is a six-tuple consisting of a set A, equipped with two binary operations $\wedge$ (called 'meet' or 'and'), $\vee$ (called 'join' or 'or'), a unary operation $\neg$ (called 'complement' or 'not') and two elements 0 and 1 (sometimes denoted by $\bot$ and $\top$), such

Kleene algebra, but most Kleene algebras are not Boolean algebras. Just as Boolean algebras are related to classical propositional logic, Kleene algebras relate to Kleene's three-valued logic—though we will not explore these ideas in this course.

Kleene algebras were not defined by Kleene; he introduced regular expressions and asked for a complete set of axioms which would allow derivation of all equations among regular expressions. The problem was first studied by John Horton Conway under the name *regular algebras*. The axioms of Kleene algebras solved this problem, as was first show by Dexter Kozen.

Kleene algebras can be combined with matrices to solve linear equations involving sets of strings. The result is a method for producing a regular expression for an automaton.

A Kleene algebra is a set A together with two operations $+ : A \times A \to A$ and $\cdot : A \times A \to A$ (usually omitted in expressions) and one function $^* : A \to A$, written as $a + b$, ab, and a^*, respectively, so that the following axioms are satisfied:

- Associativity of $+$ and $\cdot$: $a + (b + c) = (a + b) + c$ and $a(bc) = (ab)c$ for all a, b, c in A.
- Commutativity of $+$: $a + b = b + a$ for all a, b in A.
- Distributivity: $a(b + c) = (ab) + (ac)$ and $(b + c)a = (bc) + (ba)$ for all a, b, c in A.
- Identity elements for $+$ and $\cdot$: There is an element 0 in A such that for all a in A: $a + 0 = 0 + a = a$. There exists an element 1 in A such that for all a in A: $a1 = 1a = a$.
- $a0 = 0a = 0$ for all a in A.

The above axioms define a structure called a *semiring*. These above properties are the same as those for ordinary addition and multiplication.

We further require:

- $+$ is idempotent: $a + a = a$ for all a in A.

Idempotence is a property satisfied by set union, for example.

It is now possible to define a partial order $\leq$ on A by setting $a \leq b$ iff $a + b = b$ (or equivalently, $a \leq b$ iff there exists an x such that $a + x = b$).

With this order, we can formulate the remaining axioms of the operation $*$:

- $1 + a(a^*) \leq a^*$ for all a in A.
- $1 + (a^*)a \leq a^*$ for all a in A.
- if a and x are in A such that $ax \leq x$, then $a^*x \leq x$.
- if a and x are in A such that $xa \leq x$, then $xa^* \leq x$.

---

that for all elements a, b and c of A, the following axioms hold:

$a \vee (b \vee c) = (a \vee b) \vee c$	$a \wedge (b \wedge c) = (a \wedge b) \wedge c$	associativity
$a \vee b = b \vee a$	$a \wedge b = b \wedge a$	commutativity
$a \vee (a \wedge b) = a$	$a \wedge (a \vee b) = a$	absorption
$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$	$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$	distributivity
$a \vee \neg a = \top$	$a \wedge \neg b = \bot$	complements

Intuitively, one should think of $a + b$ as "union" or "least upper bound" of a and b, and of ab as some multiplication which is monotonic, in the sense that $a \leq b$ implies $ax \leq bx$. The idea behind the star operator is $a^* = 1 + a + aa + aaa + \cdots$. From the standpoint of programming language theory, one may also interpret $+$ as "choice", $\cdot$ as "sequencing" and $*$ as "iteration".

The rules are summarised in Table 4.1.

Table 4.1: Axioms of Kleene Algebra

(A.1) $a + (b + c) = (a + b) + c$

(A.2) $a + b = b + a$

(A.3) $a + a = a$

(A.4) $a + 0 = a$

(A.5) $a(bc) = (ab)c$

(A.6) $a1 = 1a = a$

(A.7) $a0 = 0a = 0$

(A.8) $a(b + c) = ab + ac$

(A.9) $(a + b)c = ac + bc$

(A.10) $1 + aa^* = a^*$

(A.11) $1 + a^*a = a^*$

(A.12) $b + ac \leq c \Rightarrow a^*b \leq c$

(A.13) $b + ca \leq c \Rightarrow ba^* \leq c$

(A.14) $ac \leq c \Rightarrow a^*c \leq c$

(A.15) $ca \leq c \Rightarrow ca^* \leq c$

Exercise 4.10.1 *Show that the following two statements are equivalent (in a Kleene algebra):*

- $a \leq b$ *iff* $a + b = b$, *and*
- $a \leq b$ *iff there exists an* x *such that* $a + x = b$.

There are numerous examples of Kleene algebras. We present some.

Example 4.10.2 *Let* Σ *be a finite set (an "alphabet") and let* A *be the set of all regular expressions over* Σ. *We consider two such regular expressions equal if they describe the same language. Then* A *forms a Kleene algebra. In fact, this is a free Kleene algebra in the sense that any among regular expressions follows from the Kleene algebra axioms and is therefore valid in every Kleene algebra. More explicitly, an equation*

$\alpha = \beta$ is a theorem of Kleene algebra, that is, derivable from the axioms described above, if and only if α and β are equivalent as regular expressions.

Example 4.10.3 *Again let Σ be an alphabet. Let A be the set of all regular languages over Σ (or the set of all context-free languages (Chapter 5) over Σ; or the set of all recursive languages (Chapter 6) over Σ; or the set of* all *languages over Σ). Then the union (written as $+$) and the concatenation (written as $\cdot$) of two elements of A again belongs to A, and so does the Kleene star operation applied to any element of A. We obtain a Kleene algebra A with 0 being the empty set and 1 being the set that contains the empty string.*

Example 4.10.4 *Let M be a monoid with identity element e and let A be the set of all subsets of M. For each two subsets S and T, let $S + T$ be the union of S and T and set $ST = \{st \mid s \in S, t \in T\}$. S^* is defined as the submonoid of M generated by S, which can be described as $\{e\} \cup S \cup SS \cup SSS \cup \cdots$. Then A forms a Kleene algebra with 0 being the empty set and 1 being $\{e\}$.*

Example 4.10.5 *Suppose M is a set and A is the set of all binary relations on M. Take $+$ to be union, $\cdot$ to be composition,*

$$R \circ S \widehat{=} \{(u,w) \mid \exists v \in M (u,v) \in R \text{ and } (v,w) \in S\}$$

and $*$ to be reflexive transitive closure, we obtain a Kleene algebra. Here 0 corresponds to the empty relation, and 1 is the identity relation on M, namely $\{(x,x) \mid x \in M\}$. Kleene algebras of binary relations are used to model programs (in Dynamic Logic). M is the power set of states, primitive elements of A are basic programs, $+$ is non-deterministic choice, $\cdot$ is sequential composition, and Kleene star is looping.

Example 4.10.6 *Every Boolean algebra with operations $\vee$ and $\wedge$ turns into a Kleene algebra if we use $\vee$ for $+$, $\wedge$ for $\cdot$, and set $a^* = 1$ for all a.*

Example 4.10.7 *A quite different Kleene algebra is useful when computing shortest paths in weighted direct graphs: let A be the extended real line (that is, $\mathbb{R} \cup \{-\infty, +\infty\}$), and take $a + b$ to be the minimum of a and b, and ab to be the ordinary sum of a and b (with the sum of $+\infty$ and $-\infty$ being defined as $+\infty$). a^* is defined to be the real number zero for nonnegative a and $-\infty$ for negative a. This is a Kleene algebra with zero element $+\infty$ and one element the real number zero.*

Example 4.10.8 *The family of $n \times n$ Boolean matrices with the zero matrix for 0, the identity matrix for 1, element-wise Boolean matrix addition and multiplication for $+$ and $\cdot$, respectively, and reflective, transitive closure for $*$.*

Exercise 4.10.9 *Prove the following:*

1. $a^* a^* = a^*$.
2. $a^{**} = a^*$.
3. *The de-nesting rule:* $(a^* b)^* a^* = (a + b)^*$.
4. *The shifting rule:* $a(ba)^* = (ab)^* a$.
5. $a^* = (aa)^* + a(aa)^*$.
6. *Zero is the smallest element:* $0 \leq a$ *for all a in A.*

7. *The sum $a+b$ is the least upper bound of a and b: we have $a \leq a+b$ and $b \leq a+b$ and if x is an element of A with $a \leq x$ and $b \leq x$, then $a+b \leq x$. Similarly, $a_1 + \cdots + a_n$ is the least upper bound of the elements $a_1, \ldots, a_n$.*

8. *Multiplication and addition are monotonic: if $a \leq b$, then $a+x \leq b+x$, $ax \leq bx$, and $xa \leq xb$ for all x in A.*

9. *Regarding the $*$ operation, we have $0^* = 1$ and $1^* = 1$, that $*$ is monotonic ($a \leq b$ implies $a^* \leq b^*$), and that $a^n \leq a^*$ for every natural number n. Furthermore, $(a^*)(a^*) = a^*$, $(a^*)^* = a^*$, and $a \leq b^*$ iff $a^* \leq b^*$.*

The de-nesting rule and the shifting rule turn out to be particularly useful in simplifying regular expressions.

4.10.1 Matrices

We now examine matrices of Kleene algebras in detail. Given an arbitrary Kleene algebra $\mathscr{K}$, the set of $n \times n$ matrices over $\mathscr{K}$, which we will denote by $\mathscr{M}_n(\mathscr{K})$, also forms a Kleene algebra. In $\mathscr{M}_2(\mathscr{K})$, for example, the identity elements for $+$ and $\cdot$ are:

$$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \text{ and } \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix},$$

respectively, and the operations $+$, $\cdot$, and * are given by

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} + \begin{bmatrix} e & f \\ g & h \end{bmatrix} \;\widehat{=}\; \begin{bmatrix} a+e & b+f \\ c+g & d+h \end{bmatrix}$$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} e & f \\ g & h \end{bmatrix} \;\widehat{=}\; \begin{bmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{bmatrix}$$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^* \;\widehat{=}\; \begin{bmatrix} (a+bd^*c)^* & (a+bd^*c)^*bd^* \\ (d+ca^*b)^*ca^* & (d+ca^*b)^* \end{bmatrix}$$

In general, $+$ and $\cdot$ in $\mathscr{M}_n(\mathscr{K})$ are ordinary matrix addition and multiplication, respectively, the identity for $+$ is the zero matrix, and the identity for $\cdot$ is the identity matrix.

To define E^* for a given $n \times n$ matrix E over $\mathscr{K}$, we proceed by induction on n. If $n = 1$, the structure $\mathscr{M}_n(\mathscr{K})$ is just $\mathscr{K}$, so we are done. For $n > 1$, break E up into four submatrices

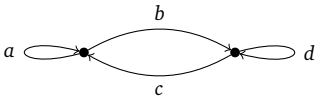
$$E = \left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right]$$

such that A and D are square, say $m \times m$ and $(n-m) \times (n-m)$, respectively. By the induction hypothesis, $\mathscr{M}_m(\mathscr{K})$ and $\mathscr{M}_{n-m}(\mathscr{K})$ are Kleene algebras, so it makes sense to define Kleene star using these, and the result will satisfy the required axioms for *. Define:

$$E^* \;\widehat{=}\; \left[\begin{array}{c|c} (A+BD^*C)^* & (A+BD^*C)^*BD^* \\ \hline (D+CA^*B)^*CA^* & (D+CA^*B)^* \end{array}\right]$$

Compare with the definition of * above.

The expressions on the right-hand sides of the definitions of * may look like they were pulled out of thin air. You may well discover where they came from if you stare really hard at the following mandala:



It can be shown that $\mathscr{M}_n(\mathscr{K})$ is a Kleene algebra under these definitions:

Lemma 4.10.10 *If $\mathscr{K}$ is a Kleene algebra, then so it $\mathscr{M}_n(\mathscr{K})$.*

Proof. Verify that $\mathscr{M}_n(\mathscr{K})$ satisfies the axioms above assuming only that $\mathscr{K}$ does. *Exercise*. □

4.10.2 Systems of Linear Equations

It is possible to solve systems of linear equations over a Kleene algebra $\mathscr{K}$. Suppose we are given a set of n variable $x_1, \ldots, x_n$ ranging over $\mathscr{K}$ and a sets of n equations of the form

$$x_i = a_{i1}x_1 + \cdots a_{in}x_n + b_i, \qquad 1 \leq i \leq n$$

where the a_{ij} and b_i are elements of $\mathscr{K}$. Arranging the a_{ij} into an $n \times n$ matrix A, the b_i in a vector b of length n, and the x_i in a vector of length n, we obtain the matrix-vector equation

$$x \quad = \quad Ax + b. \tag{4.1}$$

It is now not hard to show:

Theorem 4.10.11 *The vector A^*b is a solution to (4.1); moreover, it is the $\leq$-least solution in $\mathscr{K}^n$.*

Proof. Exercise. □

Now we can use this to give a regular expression equivalent to an arbitrarily given deterministic finite automaton:

$$M = (Q, \Sigma, \delta, s, F).$$

Assume without loss of generality that $Q = \{1, 2, \ldots, n\}$. For each $q \in Q$ let X_q denote the set of strings in Σ^* that would be accepted by M if q were the start state; that is,

$$X_q \quad \widehat{=} \quad \{x \in \Sigma^* \mid \widehat{\delta}(q, x) \in F\}.$$

The X_q satisfy the following system of equations:

$$X_q \quad = \quad \begin{cases} \sum_{a \in \Sigma} aX_{\delta(q,a)} & \text{if } q \notin F, \\ \sum_{a \in \Sigma} aX_{\delta(q,a)} + 1 & \text{if } q \in F. \end{cases}$$

Moreover, the X_q give the least solution with respect to $\subseteq$. As above, these equations can be arranged in a single matrix-vector equation of the form

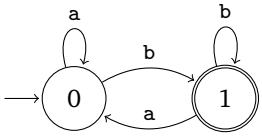
$$X = AX + b \tag{4.2}$$

where A is an $n \times n$ matrix containing sums of elements of Σ, b is a $0-1$ vector of length n, X is a vector consisting of $X_1, \ldots, X_n$. The vector X is the least solution of (4.2). By Theorem 4.10.11,

$$X = A^*b.$$

Compute the matrix A^* symbolically, so that its entries are regular expressions, then multiply by b. A regular expression for $L(M)$ can then be read off from the sth entry of A^*b, where s is the start state of M.

Example 4.10.12 *Given the following finite state automaton.*



This can be encoded as equations:

$$\begin{aligned} X_0 &= aX_0 + bX_1 \\ X_1 &= aX_0 + bX_1 + 1 \end{aligned}$$

This in turn can be seen as matrix-vector equation:

$$\begin{bmatrix} X_0 \\ X_1 \end{bmatrix} = \begin{bmatrix} a & b \\ a & b \end{bmatrix} \begin{bmatrix} X_0 \\ X_1 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

Now the solution to this is given by

$$\begin{aligned} \begin{bmatrix} X_0 \\ X_1 \end{bmatrix} &= \begin{bmatrix} a & b \\ a & b \end{bmatrix}^* \begin{bmatrix} 0 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} (a+bb^*a)^* & (a+bb^*a)^*bb^* \\ (b+aa^*b)^*aa^* & (b+aa^*b)^* \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} (a+bb^*a)^*bb^* \\ (b+aa^*b)^* \end{bmatrix}. \end{aligned}$$

Thus the regular expression, corresponding to start state 0*, is* $(a+bb^*a)^*bb^*$.

Chapter 5

Context-Free Languages

5.1 Context-Free Grammars

We now move on to *context-free grammars*, a more powerful method for describing languages. Such grammars can describe certain features that have a recursive structure, which makes them useful in a variety of applications, such as the study of human languages and in the specification and compilation of programming languages. The collection of languages associated with context-free grammars are called the *context-free languages*. They include all regular languages and many additional languages. Context-free languages are recognised by a class of machines known as *pushdown automata*.

A grammar consists of a collection of *production rules*. Each rule comprises a symbol and a string separated by an arrow. The symbol is called a *non-terminal*. The string consists of non-terminals and other symbols called *terminals*. Non-terminals are often represented by capital letters. The terminals are analogous to the input alphabet and are often represented by lowercase letters, numbers, or special symbols. One non-terminal is designated as the *start symbol*, which usually occurs on the left-hand side of the topmost rule.

Example 5.1.1 *The following is an example of a context-free grammar, which we call* G_1.

$$\begin{aligned} A &\to 0A1 \\ A &\to B \\ B &\to \# \end{aligned}$$

The non-terminals are $\{A, B\}$. *A is the start symbol. The terminals are* $\{0, 1, \#\}$.

A grammar is used to generate strings in the language as follows:

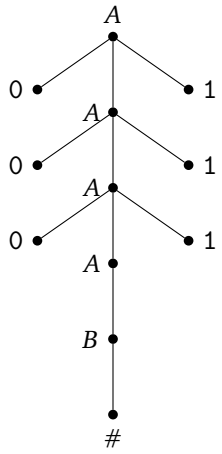
1. Write down the start symbol.
2. Find a non-terminal that has been written down and a rule that starts with that non-terminal. Replace the written down non-terminal with the right-hand side of the rule.
3. Repeat step 2 until no non-terminals remain.

The sequence of substitutions to obtain a string is called a *derivation*.

Example 5.1.2 *The grammar G_1 generates the string* 000#111. *A derivation of this string is:*

$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111.$$

The same information may be represented pictorially with a *parse tree*.



All strings generated in this way constitute the *language of the grammar*. We write $L(G_1)$ for the language of grammar G_1. Any language that can be generated by some context-free grammar is called a *context-free language*. For convenience, we abbreviate several rules with the same left-hand side, such as $A \rightarrow 0A1$ and $A \rightarrow B$, into a single line $A \rightarrow 0A1 \mid B$.

Example 5.1.3 $L(G_1) = \{0^n\#1^n \mid n \geq 0\}$.

Definition 5.1.4 (Context-Free Grammar) *A* context-free grammar *is a 4-tuple (V, Σ, R, S), where*

1. *V is a finite set called the* non-terminals*,*
2. *Σ is a finite set, disjoint from V, called the* terminals*,*
3. *R is a finite set of* rules*, with each rule being a non-terminal and a string of non-terminals and terminals, and*
4. *$S \in V$ is the start symbol.*

If u, v and w are strings of non-terminals and terminals, and $A \rightarrow w$ is a rule of the grammar, we say that uAv *yields* uwv, written $uAw \Rightarrow uwv$. We say that u *derives* v, written $u \Rightarrow^* v$, if $u = v$ or if a sequence $u_1, \ldots, u_k$ exists for $k \geq 0$ and $u \Rightarrow u_1 \Rightarrow \cdots \Rightarrow u_k \Rightarrow v$.

The *language of the grammar* is $\{w \in \Sigma^* \mid S \Rightarrow^* w\}$.

Example 5.1.5 (Nested Parentheses) *Consider the grammar $G_2 = (\{S\}, \{(,)\}, R, S)$. The set of rules, R, is*

$$S \quad \rightarrow \quad (S) \mid SS \mid \epsilon.$$

This grammar generates strings such as ()()*,* (((())) *, and* (()()) *. $L(G_2)$ is thus the language of all strings of properly nested parentheses.*

We now describe *parse trees*. These have ample applications in compiler construction, though we won't explore that topic here.

Given a context-free grammar $G = (V, \Sigma, R, S)$, we define its *parse trees* and their *roots*, *leaves*, and *yields*, as follows.

1.

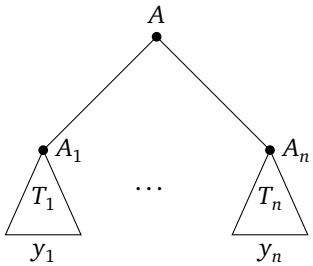
   This is a parse tree for each $A \in V$. The single node of this parse tree is the root and a leaf. The yield of this parse tree is A.

2. If $A \to \epsilon$ is a rule in R, then

   is a parse tree; its root is the node labelled A, its sole leaf is the node labelled ϵ, and its yield is ϵ.

3. If

   are parse trees ($n \geq 1$) with roots labelled $A_1, \ldots, A_n$, respectively, and with yields $y_1, \ldots, y_n$, and $A \to A_1 \ldots A_n$ is a rule in R, then



   is a parse tree. Its root is the new node labelled A, its leaves are the leaves of $T_1, \ldots, T_n$, and its yield is $y_1 \ldots y_n$.

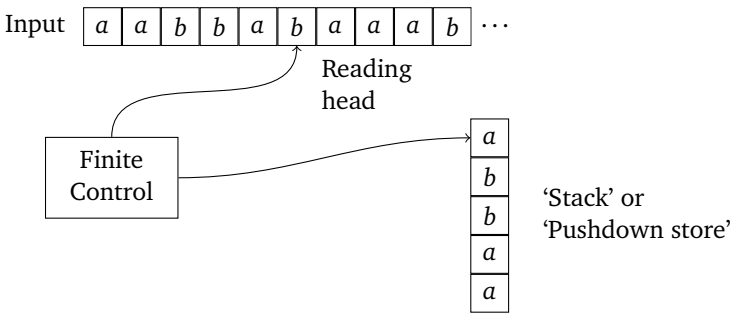
4. Nothing else is a parse tree.

A *path* in a parse tree is a sequence of distinct nodes, each connected to the previous one by a line segment; the first node is the root node and the last node is a leaf. The *length* of a path is the number of line segments, which is one less than the number of nodes. The *height* of a parse tree is the length of the longest path.

5.2 Pushdown Automata

Finite state automata are not able to recognise context-free languages in general. In order to do so, the smallest change required to finite state automata is to add a *stack* or *pushdown store*, which keeps track of information about what symbols have been visited up until this point in time.

Consider the language $\{ww^R \mid w \in \Sigma^*\}$. It seems that any automaton that recognises strings in this language from left to right needs to do so by remembering the first half of the input string so that it can check it—in reverse order—against the second half of the input. It is not surprising that this is not possible for a finite state machine, but if the machine is capable of accumulating its input string as it is read, appending symbols one at a time to a stored string, then it could non-deterministically guess when the centre of the input has been reached and check the symbols off from its memory one at a time.



Definition 5.2.1 (Pushdown Automaton) *A* pushdown automaton *is a sextuple* $M = (Q, \Sigma, \Gamma, \Delta, s, F)$*, where*

- Q *is a finite set of* states*,*
- Σ *is an alphabet (the* input symbols*),*
- Γ *is an alphabet (the* stack *symbols),*
- $s \in Q$ *is the* initial state*,*
- $F \subseteq Q$ *is the set of* final states*, and*
- Δ*, the* transition relation*, is a finite subset of* $(Q \times \Sigma^* \times \Gamma^*) \times (Q \times \Gamma^*)$.

Intuitively, if $((p, u, \beta), (q, \gamma)) \in \Delta$, then M, whenever it is in state p with β at the top of the stack, may read u from the input tape, replace β by γ on the top of the stack, and enter state q. Such a pair $((p, u, \beta), (q, y))$ is called a *transition* of M; since several transitions of M may be simultaneously applicable, the machines we are describing are non-deterministic in operation.

To *push* a symbol is to add it to the top of the stack; to *pop* a symbol is to remove it from the top of the stack. For example, the transition $((p, u, \epsilon), (q, a))$ pushes a, while $((p, u, a), (q, \epsilon))$ pops a.

As is the case with finite automata, during a computation the portion of the input already read does not affect the subsequent operation of the machine. Accordingly, a configuration is defined to be a member of $Q \times \Sigma^* \times \Gamma^*$: the first component is the state of the machine, the second is the portion of the

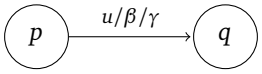
input yet to be read, and the third is the contents of the pushdown store, read top-down. For example, if the configuration were (q, w, abc), the a would be on the top of the stack and the c on the bottom.

For every transition $((p, u, \beta), (q, \gamma)) \in \Delta$, and for every $x \in \Sigma^*$ and $\alpha \in \Gamma^*$, we define $(p, ux, \beta\alpha) \vdash_M (q, x, \gamma\alpha)$; moreover $\vdash_M$ (*yields in one step*) holds only between configurations that can be represented in the form of some transition, some x, and some α.

We denote the reflexive, transitive closure of $\vdash_M$ by $\vdash_M^*$, and say that M *accepts* a string $w \in \Sigma^*$ if and only if $(s, w, \epsilon) \vdash_M^* (p, \epsilon, \epsilon)$ for some state $p \in F$. To put it another way, M accepts a string w if and only if there is a sequence of configurations $C_0, C_1, \ldots, C_n$ $(n \geq 0)$ such that $C_0 \vdash_M C_1 \vdash_M \cdots \vdash_M C_n$, $C_0 = (s, w, \epsilon)$ and $C_n = (p, \epsilon, \epsilon)$ for some $p \in F$. This a string is accepted if computation using the pushdown automaton results in a **final state with an empty stack.** Any sequence of configurations $C_0, C_1, \ldots, C_n$ such that $C_i \vdash_M C_{i+1}$ for $i = 0, \ldots, n-1$ will be called a *computation* of M; it will be said to have *length* n or to have n *steps*. The *language accepted* by M, denotes $L(M)$, is the set of all strings accepted by M.

Note that because pushdown automata are defined based on a transition relation, not transition function, they are non-deterministic.

Pushdown automata can be concisely represented as an automaton, where a transition $((p, u, \beta), (q, \gamma))$ is denoted as

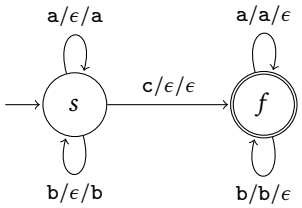


The label $u/\beta/\gamma$ is read as 'consume u from the string, pop β from the stack and push γ.'

Initial and final states are marked as before, namely with a small arrow and a double circle, respectively.

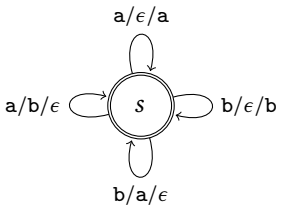
Example 5.2.2 *The following PDA accepts the language of odd-length palindromes with centre character* c*, namely,*

$$L = \{w\,\mathtt{c}\,w^R \mid w \in \{\mathtt{a}, \mathtt{b}\}^*\}$$



Example 5.2.3 *The following PDA accepts the language of strings with equal numbers of* a*'s and* b*'s, namely,*

$$L = \{w \in \{\mathtt{a}, \mathtt{b}\}^* \mid \#_a(w) = \#_b(w)\}$$



5.2.1 Context-Free Languages and Pushdown Automata

In this section we tie together context-free grammars and pushdown automata, showing that they correspond to the same class of languages (namely, the context-free languages). Beyond its mathematical significance—it ties together two different formalisms—this result has practical significance, as it lays the foundation for the study of syntax analysers for 'real' context-free languages such as programming languages.

Let's now introduce some preliminaries. Recall that if $G = (V, \Sigma, R, S)$ is a context-free grammar then $w \in L(G)$ iff $w \in \Sigma^*$ and there is a derivation

$$S \Rightarrow_G w_1 \Rightarrow_G w_2 \Rightarrow_G \cdots \Rightarrow_G w_{n-1} \Rightarrow_G w$$

for some strings $w_1, w_2, \ldots, w_{n-1} \in (V \cup \Sigma)^*$ $(n > 0)$. This derivation will be called a *leftmost* derivation if the nonterminal symbol replaced at every step is the leftmost nonterminal in the string. Formally, we write $\alpha \Rightarrow_G^L \beta$, where $\alpha, \beta \in (V \cup \Sigma)^*$, if and only if $\alpha = \alpha_1 A \alpha_2$, $\beta = \alpha_1 \gamma \alpha_2$, $A \to \gamma$ is a rule of G, and $\alpha_1 \in \Sigma^*$. We write $\Rightarrow_G^{L*}$ for the reflexive, transitive closure of $\Rightarrow_G^L$, and define a *leftmost derivation* to be a sequence of the form

$$S \Rightarrow_G^L w_1 \Rightarrow_G^L w_2 \Rightarrow_G^L \cdots \Rightarrow_G^L w_{n-1} \Rightarrow_G^L w_n$$

for any $n \geq 0$.

The crucial fact about leftmost derivations is given in the following lemma.

Lemma 5.2.4 *For any context-free grammar $G = (V, \Sigma, R, S)$ and any string $w \in \Sigma^*$, $S \Rightarrow_G^* w$ if and only if $S \Rightarrow_G^{L*} w$.*

Proof. In one direction is obvious, as any leftmost derivation is also a derivation. To prove the other direction, let

$$w_0 \Rightarrow_G w_1 \Rightarrow_G w_2 \Rightarrow_G \cdots \Rightarrow_G w_n$$

be a derivation, not necessarily leftmost, where $w_0 = S$ and $w_n = w$. We show how to transform it little by little into a leftmost derivation. Let k be the least number such that the step $w_k \Rightarrow_G w_{k+1}$ is *not* a leftmost derivation of w_{k+1} from w_k; if no such k exists, we are done. We define another derivation

$$w'_0 \Rightarrow_G w'_1 \Rightarrow_G w'_2 \Rightarrow_G \cdots \Rightarrow_G w'_n$$

also of n steps and such that $w'_0 = w_0$ and $w'_n = w_n$. This new derivation is either leftmost or has the property that if $w'_{k'} \Rightarrow_G w'_{k'+1}$ is the earliest step in *this* derivation that is not leftmost, then $k' > k$. Since both derivations have the same length, the lemma will follow by induction.

Since $w_k \Rightarrow_G w_{k+1}$ is not leftmost, w_k is of the form $\alpha A \beta B \gamma$ $(\alpha \in \Sigma^*; \beta, \gamma \in (V \cup \Sigma)^*; A, B \in V)$ and $w_{k+1} = \alpha A \beta \delta \gamma$, where $B \Rightarrow_G \delta$. Since $w_n \in \Sigma^*$, $k + 1 < n$, and there is some latter step where the indicated occurrence of A is replaced. Thus let l be the least number greater than k such that $w_l = \alpha A \zeta$ for some $\zeta \in (V \cup \Sigma)^*$ and $w_{l+1} = w_l = \alpha \eta \zeta$, where $A \Rightarrow_G \eta$. Then $\beta B \gamma \Rightarrow_G^* \zeta$ in $l - k$ steps. Then we can switch the uses of $B \Rightarrow_G \delta$ and $A \Rightarrow_G \eta$ as follows:

$$\begin{array}{lll} w_0 \Rightarrow_G^{L*} w_k = \alpha A \beta B \gamma & & k \text{ steps} \\ \Rightarrow_G^L \alpha \eta \beta B \gamma & & 1 \text{ step} \\ \Rightarrow_G^* \alpha \eta \zeta & & l - k \text{ steps} \\ \Rightarrow_G^* w_n & & n - l - 1 \text{ steps.} \end{array}$$

The new derivation is of length n and begins with at least $k + 1$ leftmost steps. □

This lemma permits us to restrict our attention to leftmost derivations.

It will also be useful to combine two computations by a pushdown automaton M into a single computation. It states that during a computation a pushdown automata neither looks ahead beyond the input it actually reads nor peeks below the part of the stack it actually uses, and the operation is unaffected by any such unseen part of the input or hidden part of the pushdown store.

Lemma 5.2.5 *If M is a pushdown automata and $(q_1, w_1, \alpha_1) \vdash_M^* (q_2, \epsilon, \alpha_2)$ and $(q_2, w_2, \alpha_2\alpha_3) \vdash_M^* (q_3, \epsilon, \alpha_4)$, then $(q_1, w_1w_2, \alpha_1\alpha_3) \vdash_M^* (q_3, \epsilon, \alpha_4)$. In particular, if $(q_1, w_1, \alpha_1) \vdash_M^* (q_2, \epsilon, \epsilon)$ and $(q_2, w_2, \alpha_3) \vdash_M^* (q_3, \epsilon, \epsilon)$, then $(q_1, w_1w_2, \alpha_1\alpha_3) \vdash_M^* (q_3, \epsilon, \epsilon)$*

Proof. Exercise. □

Theorem 5.2.6 *The classes of languages accepted by pushdown automata is exactly the class of context-free languages.*

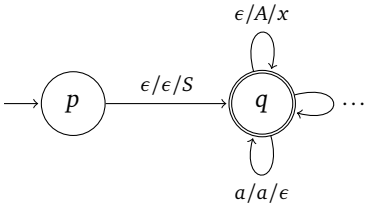
Proof. We break this proof into two parts, of which the first is easier. □

Lemma 5.2.7 *Each context-free language is accepted by some pushdown automaton.*

Proof Idea. Let $G = (V, \Sigma, R, S)$ be a context-free grammar; we must construct a pushdown automaton M such that $L(M) = L(G)$. The machine we construct has only two states, p and q, and remains permanently in state q after its first move. Also, M uses $(V \cup \Sigma)$, the set of terminals and nonterminals, as its stack alphabet. We let $M = (\{p, q\}, \Sigma, V \cup \Sigma, \Delta, p, \{q\})$, where Δ contains the following transitions.

1. $((p, \epsilon, \epsilon), (q, S))$,
2. $((q, \epsilon, A), (q, x))$ for each rule $A \to x$ in R, and
3. $((q, a, a), (q, \epsilon))$ for each $a \in \Sigma$.

This construction can be visualised as follows:



The pushdown automaton M begins by pushing S, the start symbol of G, on its initially empty pushdown store, entering state q (transition 1). On each subsequent step, it either replaces the topmost symbol A on the stack, provided that it is a nonterminal, by the right-hand side x of some rule $A \to x$ in R (transitions of type 2), or pops the topmost symbol from the stack, provided that it is a terminal symbol that matches the next input symbol (transitions of type 3). The transitions of M are designed so that the pushdown store during an accepting computation mimics a leftmost derivation of the input string: M intermittently carries out a step of such a derivation on the stack, and between such steps it

strips away from the top of the stack any terminal symbols and matches them against symbols from the input string.

The lemma can be obtained by proving the following two claims:

- If $S \Rightarrow_G^{L*} \alpha_1\alpha_2$, where $\alpha_1 \in \Sigma^*$ and $\alpha_2 \in V(V \cup \Sigma)^* \cup \{\epsilon\}$, then $(q, \alpha_1, S) \vdash_M^* (q, \epsilon, \alpha_2)$.
- If $(q, \alpha_1, S) \vdash_M^* (q, \epsilon, \alpha_2)$, where $\alpha_1 \in \Sigma^*$ and $\alpha_2 \in (V \cup \Sigma)^*$, then $S \Rightarrow_G^{L*} \alpha_1\alpha_2$.

□

Lemma 5.2.8 *If a language is accepted by a pushdown automaton, it is a context-free language.*

Proof Idea. It will be helpful to restrict somewhat the pushdown automata under consideration. Call a pushdown automaton $(Q, \Sigma, \Gamma, \Delta, s, F)$ *simple* if (1) whenever $((q, u, \beta), (p, \gamma))$ is a transition in Δ, $|\beta| \leq 1$; and (2) whenever $((q, u, \epsilon), (p, \gamma)) \in \Delta$, then also $((q, u, A), (p, \gamma A))$ for each $A \in \Gamma$. In other words, (1) if the machine consults the stack at all, it looks at—and possibly removes—only the topmost symbol; and (2) whenever the machine can move without consulting or removing any symbol from the stack, it may also achieve the same effect by removing the top symbol from the stack and then immediately putting it back. We claim that no generality is lost in considering only simple automata; that is, if a language is accepted by a unrestricted pushdown automaton, then it is accepted by a simple pushdown automaton. To see this, let $M = (Q, \Sigma, \Gamma, \Delta, s, F)$ by any pushdown automaton; we construct a simple pushdown automaton that also accepts $L(M)$. We first eliminate from Δ any transition $((q, u, \beta), (p, \gamma))$ with $|\beta| > 1$. This is done by modifying M to pop sequentially the individual symbols of β, rather than removing them all in a single step. Specifically, let $\beta = B_1 \ldots B_n$, where $n > 1$ and $\beta = B_1 \ldots B_n \in \Gamma$; then add to Q new states $t_1, \ldots, t_{n-1}$ and replace in Δ the single transition $((q, u, \beta), (p, \gamma))$ by these transitions:

$$\begin{array}{l} ((q, \epsilon, B_1), (t_1, \epsilon)) \\ ((t_1, \epsilon, B_2), (t_2, \epsilon)) \\ \quad \vdots \\ ((t_{n-2}, \epsilon, B_{n-1}), (t_{n-1}, \epsilon)) \\ ((t_{n-1}, u, B_n), (p, \gamma)) \end{array}$$

A similar modification is carried out for each transition in Δ that violates condition (1). To satisfy condition (2), we merely add to Δ the transitions $((q, u, A), (p, \gamma A))$ for each $A \in \Gamma$, whenever $((q, u, \epsilon), (p, \gamma))$ is a transition already in Δ. The resulting automaton is simple, and accepts the same language as M accepts.

Now we show that if $M = (Q, \Sigma, \Gamma, \Delta, s, F)$ is any *simple* pushdown automaton, then $L(M)$ is the language generated by some context-free grammar G. We let $G = (V, \Sigma, R, S)$, where V contains, in addition to the new symbol S, a new symbol $\langle q, A, p \rangle$ for all $q, p \in Q$ and each $A \in \Sigma \cup \{\epsilon\}$. To understand the role of the nonterminals $\langle q, A, p \rangle$, remember that G is supposed to generate all strings accepted by M. Therefore the nonterminals of G stand for different parts of the input strings that are accepted by M. In particular, if $A \in \Gamma$ then the nonterminal $\langle q, A, p \rangle$ denotes a portion of the input string that might be read between a point in time when M is in state q with A on top of its stack, and a point in time when M removes that occurrence of A from the stack and enters state p. If $A = \epsilon$, then $\langle q, \epsilon, p \rangle$ denotes a portion of the input string that might be read between a time when M is in state q and a time when it is in state p with the same stack, without in the interim changing or consulting that part of the stack.

The rules in R are of four types:

1. For each $f \in F$, the rule $S \to \langle s, \epsilon, f \rangle$.

2. For each transition $((q,u,A),(r,B_1 \ldots B_n)) \in \Delta$ where $q, r \in Q$, $u \in \Sigma^*$, $n > 0$, $B_1, \ldots, B_n \in \Gamma$, and $A \in \Gamma \cup \{\epsilon\}$, and for all $p, q_1, \ldots, q_{n-1} \in Q$, the rule

$$\langle q,A,p \rangle \to u\langle r,B_1,q_1 \rangle \langle q_1,B_2,q_2 \rangle \cdots \langle q_{n-1},B_n,p \rangle.$$

3. For each transition $((q,u,A),(r,\epsilon)) \in \Delta$, where $q, r \in Q$, $u \in \Sigma^*$, and $A \in \Gamma \cup \{\epsilon\}$, and for each $p \in Q$, the rule

$$\langle q,A,p \rangle \to u\langle r,\epsilon,p \rangle.$$

4. For each $q \in Q$, the rule $\langle q,\epsilon,q \rangle \to \epsilon$.

Note that, because M is simple, either type 2 or type 3 applies to each transition of M.

A rule of type 1 says essentially that the goal is to pass from the initial state to a final state, while leaving the stack in the same condition at the end as it was at the beginning. A rule of type 4 says that no computation is needed to go from a state to itself. A rule of type 2 describes a long computation, with the net effect of going from state q to state p, while removing A (possibly ϵ) from the stack. The right-hand side of this single rule of R represents $n+1 \geq 2$ shorter computations of M, the first of which is a single move to state r while reading input u, and the last n of which are sequences of moves between intermediate states that effect the removal of $B_1, \ldots, B_n$ from the stack. Rules of type 3 are analogous to rules of type 2 for the case in which $n = 0$, that is, a symbol (or ϵ) is removed from the stack and no symbol is added.

These intuitive remarks are formalised in the following claim:

- For any $q, p \in Q$, $A \in \Gamma \cup \{\epsilon\}$, and $x \in \Sigma^*$,

$$\langle q,A,p \rangle \Rightarrow_G^* x \text{ if and only if } (q,x,A) \vdash_M^* (p,\epsilon,\epsilon).$$

The lemma follows readily from the claim, since then $\langle s,\epsilon,f \rangle \Rightarrow_G^* x$, for some $f \in F$, if and only if $(s,x,\epsilon) \vdash_M^* (f,\epsilon,\epsilon)$; that is $x \in L(G)$ if and only if $x \in L(M)$. □

5.2.2 Closure Properties

Theorem 5.2.9 *The context-free languages are closed under union, concatenation, and Kleene star.*

Proof. Let $G_1 = (V_1, \Sigma_1, R_1, S_1)$ and $G_2 = (V_2, \Sigma_2, R_2, S_2)$, and without loss of generality assume that V_1 and V_2 are disjoint.

Union. Let S be a new symbol and let $G = (V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, R_1 \cup R_2 \cup \{S \to S_1, S \to S_2\}, S)$. Then $L(G_1) \cup L(G_2) = L(G)$. For the only rules involving S are $S \to S_1$ and $S \to S_2$, so $S \Rightarrow_G^* w$, where $w \in (\Sigma_1 \cup \Sigma_2)^*$, if and only if either $S_1 \Rightarrow_G^* w$ or $S_2 \Rightarrow_G^* w$; and since G_1 and G_2 have disjoint sets of nonterminals, $S_1 \Rightarrow_G^* w$ if and only if $S_1 \Rightarrow_{G_1}^* w$, and similarly for G_2.

Concatenation. The construction is similar: $L(G_1)L(G_2)$ is generated by the grammar

$$(V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, R_1 \cup R_2 \cup \{S \to S_1 S_2\}, S).$$

Kleene Star. $L(G_1)^*$ is generated by

$$(V_1, \Sigma_1, R \cup \{S_1 \to \epsilon, S_1 \to S_1 S_1\}, S_1).$$

□

The class of context-free languages is *not* closed under intersection or complementation. However, it is closed under intersection with regular sets.

Theorem 5.2.10 *The intersection of a context-free language with a regular language is a context-free language.*

Proof. We prove using finite and pushdown automata, as this approach is simpler than using grammars. If L is a context-free language and R is a regular set, then $L = L(M_1)$ for some pushdown automaton $M_1 = (Q_1, \Sigma_1, \Gamma_1, \Delta_1, s_1, F_1)$ and $R = L(M_2)$ for some *deterministic* finite automaton $(Q_2, \Sigma_2, \delta_2, s_2, F_2)$. The idea is to combine these machines into a single pushdown automaton M that carries out computations by M_1 and M_2 in parallel and accepts only if both would have accepted. Specifically, let $M = (Q, \Sigma, \Gamma, \Delta, s, F)$, where

- $Q = Q_1 \times Q_2$, the Cartesian product of the state sets of M_1 and M_2
- $\Sigma = \Sigma_1 \cup \Sigma_2$
- $\Gamma = \Gamma_1$
- $s = (s_1, s_2)$
- $F = F_1 \times F_2$

and Δ, the transition relation, is defined by

$$(((q_1, q_2), u, \beta), ((p_1, p_2), \gamma)) \in \Delta$$

if and only if

$$((q_1, u, \beta), (p_1, \gamma)) \in \Delta_1 \tag{5.1}$$

and

$$(q_2, u) \vdash^*_{M_2} (p_2, \epsilon). \tag{5.2}$$

That is, M passes from state (q_1, q_2) to state (p_1, p_2) in the same way that M_1 passes from state q_1 to p_1, except that in addition M keeps track of the change in the state of M_2 caused by reading the same input. Condition 5.2 is easy to check, since a finite automaton reads a single symbol on each move. Thus, in practice, we would construct M by repeatedly choosing a transition $((q_1, u, \beta), (p_1, \gamma))$ of M_1 and a state q_2 of M_2 and simulating M_2 for $|u|$ steps on input u to determine what state p_2 it would reach after reading that input. □

5.2.3 Pumping Lemma for Context-Free Languages

Theorem 5.2.11 *Let G be a context-free grammar. Then there is a number K, depending on G, such that any string w in $L(G)$ of length greater than K can be rewritten as $w = uvxyz$ in such a way that either v or y is non-empty and $uv^n xy^n z$ is in $L(G)$ for every $n \geq 0$.*

Proof. Let $G = (V, \Sigma, R, S)$. It suffices for the theorem to show that there is a K such that any terminal string in $L(G)$ of length greater than K has a derivation of the form

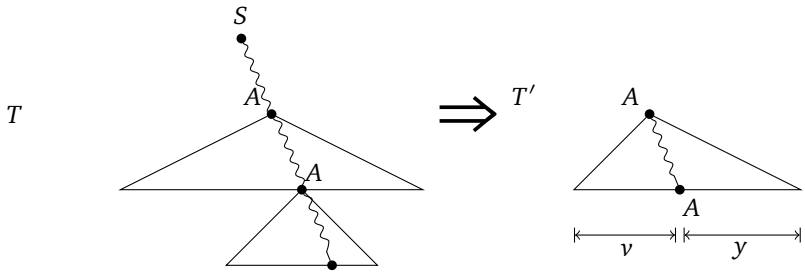
$$S \Rightarrow^* uAz \Rightarrow^* uvAyz \Rightarrow^* uvxyz$$

where $u, v, x, y, z \in \Sigma^*$, $A \in V$, and either v or y is non-empty. Then the derivation $A \Rightarrow^* vAy$ can be repeated any number of times to obtain the various strings $uv^n xy^n z$.

Let p be the largest number of symbols on the right-hand side of any rule in R, that is,

$$p = \max\{|\alpha| \mid A \to \alpha \text{ is a rule of } G\}.$$

For any $m \geq 1$, a parse tree of height m can have at most p^m leaves, as can easily be shown by induction on m, and can therefore have a yield of length at most p^m. To put it another way, if T is a parse tree with yield of length greater than p^m, then T has some path of length greater than m. Now let $m = |V|$, $K = p^m$, and suppose w has length exceeding K. Let T be a parse tree with root labelled S and with yield w. Then T has at least one path with more than $|V| + 1$ nodes, and therefore at least one path including two nodes labelled with the same member A of V. Let us look at such a path in more detail (consider the following figure).



Consider a parse tree T' whose root is one node labelled A on this path, and whose leaves are the leaves of T, except that some subsequent node labelled A on the path is a leaf. This parse tree T' has root labelled A and has a yield of the form vAy for some $v, y \in \Sigma^*$. Now it is possible that $v = y = \epsilon$, but this cannot be the case for every choice of the path and the two nodes on it with the same label. For if $v = y = \epsilon$, then the tree T' can be removed from T without changing the yield of the tree as a whole, by attaching the parse tree with the lower node as its root at the upper node labelled A. If all paths of length exceeding m could be shortened in this way, without changing the yield of the parse tree, we could obtain a parse tree with yield w and with height less than m, which is impossible. But then we are done; A, u, v, x, y and z can be determined from T.

□

This theorem is most useful for showing that certain languages are *not* context-free.

Theorem 5.2.12 *$L = \{a^n b^n c^n \mid n \geq 0\}$ is not context-free.*

Proof. We prove by contradiction. Suppose $L = L(G)$ for some context-free grammar G. Let K be a constant for G as specified by Theorem 5.2.11 and let $n > K/3$. Then $w = a^n b^n c^n$ is in $L(G)$ and has a representation $w = uvxyz$ such that v or y is nonempty and $uv^i xy^i z$ is in $L(G)$ for each $i = 0, 1, 2, \ldots$. But this is impossible; for if either u or v contains two symbols from $\{a, b, c\}$, then uv^2xy^2z contains a

b before an a or a c before a b, and if v and y each contain only a's, only b's, or only c's, then uv^2xy^2z cannot contain an equal number of a's, b's, and c's. □

We can now show that context-free languages are *not* closed under certain operations.

Theorem 5.2.13 *The context-free languages are not closed under intersection or complementation.*

Proof. Clearly $\{a^n b^n c^m \mid m, n \geq 0\}$ and $\{a^m b^n c^n \mid m, n \geq 0\}$ are context-free. The intersection of these two context-free languages is the language $\{a^n b^n c^n \mid n \geq 0\}$ just shown not to be context-free. And if context-free languages were closed under complementation, they would be closed under intersection, since

$$L_1 \cap L_2 = \Sigma^* \setminus ((\Sigma^* \setminus L_1) \cup (\Sigma^* \setminus L_2)).$$

□

Part III

Computability Theory

Chapter 6

Turing Machines

6.1 Turing Machines

We will now introduce the most powerful of the automata studied in this course: Turing Machines (TMs), named after Alan Turing, who invented them in 1936. Turing machines can compute any function normally considered computable. TMs were invented to come to grips with the notion of *effective computation*, long before computers were even invented. TMs are the one of a variety of competing, yet ultimately equivalent, formalisms (including Post systems, μ-recursive functions, λ-calculus, combinatory logic) that most resembles a modern computer. A TM consists of a deterministic finite state controller and an infinite tape (representing the memory). Many custom variations exist (nondeterministic, multi-tape, multi-dimensional type, two-way infinite tapes, and so on), which all turn out to be computationally equivalent in the sense that they can all simulate one another.

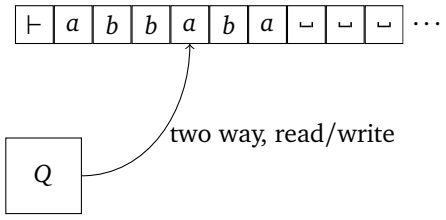
6.1.1 Church's Thesis

All of these vastly different formalisms are computationally equivalent, the common notion of computability that they embody is extremely robust, which is to say that it is invariant under fairly radical perturbations in the model. All of these models embody the elusive notion of *effective computability* that had been sought for so long. Alonzo Church gave voice to this thought and it has since become known as *Church's thesis* (or the *Church-Turing thesis*). It is not a theorem, as such, but rather a declaration that all these formalisms capture precisely our intuition about what it means to be effectively computable in principle. The most compelling development leading to the acceptance of Church's thesis was the Turing machine. It was the first model that could be considered readily programmable, whereas this was hard to see with the other models, and hence they were met with skepticism. But it is hard to argue with Turing machines. One can rightly challenge Church's thesis on the grounds that there are aspects of computation that are not addressed by Turing machines (for example, randomised or interactive computation), but no one could dispute that the notion of effective computability as captured by Turing machines is robust and important.

6.1.2 Informal Description of Turing Machines

We describe here a deterministic, one-tape Turing machine. This is the standard off-the-shelf model. There are many variations, apparently more powerful or less powerful, but in reality not. We will consider some of these later.

A TM has a finite set of states Q, a semi-infinite tape that is delimited on the left end by an endmarker $\vdash$ and is infinite to the right, and a head that can move left and right over the tape, reading and writing symbols.



The input string is of finite length and initially written on the tape in contiguous tape cells snug up against the left endmarker. The infinitely many cells to the right of the input contains a special blank symbol ␣.

The machine starts in its start state s with its head scanning the left endmarker. In each step it reads the symbol on the tape under its head. Depending on that cell and the current state, it writes a new symbol on that tape cell, moves its head either left or right one cell, and enters a new state. The action it takes in each situation is determined by a transition function δ. It *accepts* its input by entering a special accepting state t and *rejects* by entering a special rejecting state r. On some inputs it my run infinitely without ever accepting or rejecting, in which case it is said to *loop* on that input.

6.1.3 Formal Definition of Turing Machines

Formally, a *deterministic one-tape Turing machine* is a 9-tuple

$$M = (Q, \Sigma, \Gamma, \vdash, \text{␣}, \delta, s, t, r)$$

where

- Q is a finite set (the *states*);
- Σ is a finite set (the *input alphabet*);
- Γ is a finite set (the *tape alphabet*) containing Σ as a subset;
- $\text{␣} \in \Gamma \setminus \Sigma$, the *blank symbol*;
- $\vdash \in \Gamma \setminus \Sigma$, the *left endmarker*;
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$, the *transition function*;
- $s \in Q$, the *start state*;
- $t \in Q$, the *accept state*; and

- $r \in Q$, the *reject state*, $r \neq t$.

Intuitively, $\delta(p,a) = (q,b,d)$ means "when in state p scanning symbol a, write b on the tape cell, move the head in the direction d, and enter state q." The symbols L and R stand for left and right, respectively.

We restrict TMs so that the left endmarker is never overwritten with another symbol and the machine never moves of the tape to the left of the endmarker; that is, we require that for all $p \in Q$ there exists $q \in Q$ such that

$$\delta(p,\vdash) \quad = \quad (q,\vdash,R). \tag{6.1}$$

We also require that once the machine enters its accept state, it never leaves it, and similarly for its reject state; that is, for all $b \in \Gamma$ there exists $c,c' \in \Gamma$ and $d,d' \in \{L,R\}$ such that

$$\begin{aligned} \delta(t,b) &= (t,c,d) \\ \delta(r,b) &= (r,c',d'). \end{aligned} \tag{6.2}$$

We sometimes refer to the state set and transition function collectively as the *finite control*.

Remark 6.1.1 *A non-deterministic Turing machine can be defined by replacing the transition function by a* transition relation $\delta \subseteq Q \times \Gamma \to Q \times \Gamma \times \{L,R\}$. *Such a Turing machine can accept, reject, fail, or loop. It fails when there is no transition for the current configuration. Failing is treated as rejection.*

Example 6.1.2 *Here is a TM that accepts the non-context-free set* $\{a^n b^n c^n \mid n \geq 0\}$. *Informally, the machine starts in its start state s, then scans right over the input string, checking that it is of the form* $a^*b^*c^*$. *It doesn't write anything on the way across (formally, it writes the same symbol it reads). When it sees the first blank symbol* $\sqcup$, *it overwrites it with a right endmarker* $\dashv$. *Now it scans left, erasing the first c it sees, as well as one b and one a, Then scans right, erasing one a, one b, and one c. It continues to sweep left and right over the input, erasing one occurrence of each letter in each pass. If on some pass it sees at least one occurrence of one of the letters and no occurrences of another, it rejects. Otherwise, it eventually erases all the letters and makes one pass between* $\vdash$ *and* $\dashv$ *seeing only blanks, at which point it accepts.*

Formally, this machine has

$$\begin{aligned} Q &= \{s, q_1, \ldots, q_{10}, t, r\}, \\ \Sigma &= \{a,b,c\}, \\ \Gamma &= \Sigma \cup \{\vdash, \sqcup, \dashv\}. \end{aligned}$$

There is nothing special about $\dashv$; *it is just an extra useful symbol in the tape alphabet. The transition*

function δ is specified by the following table:

	$\vdash$	a	b	c	$\sqcup$	$\dashv$
s	$(s,\vdash,R)$	(s,a,R)	(q_1,b,R)	(q_2,c,R)	$(q_3,\dashv,L)$	$-$
q_1	$-$	$(r,-,-)$	(q_1,b,R)	(q_2,c,R)	$(q_3,\dashv,L)$	$-$
q_2	$-$	$(r,-,-)$	$(r,-,-)$	(q_2,c,R)	$(q_3,\dashv,L)$	$-$
q_3	$(t,-,-)$	$(r,-,-)$	$(r,-,-)$	$(q_4,\sqcup,L)$	$(q_3,\sqcup,L)$	$-$
q_4	$(r,-,-)$	$(r,-,-)$	$(q_5,\sqcup,L)$	(q_4,c,L)	$(q_4,\sqcup,L)$	$-$
q_5	$(r,-,-)$	$(q_6,\sqcup,L)$	(q_5,b,L)	$-$	$(q_5,\sqcup,L)$	$-$
q_6	$(q_7,\vdash,R)$	(q_6,a,R)	$-$	$-$	$(q_6,\sqcup,L)$	$-$
q_7	$-$	$(q_8,\sqcup,R)$	$(r,-,-)$	$(r,-,-)$	$(q_7,\sqcup,R)$	$(t,-,-)$
q_8	$-$	(q_8,a,R)	$(q_9,\sqcup,R)$	$(r,-,-)$	$(q_8,\sqcup,R)$	$(r,-,-)$
q_9	$-$	$-$	(q_9,b,R)	$(q_{10},\sqcup,R)$	$(q_9,\sqcup,R)$	$(r,-,-)$
q_{10}	$-$	$-$	$-$	(q_{10},c,R)	$(q_{10},\sqcup,R)$	$(q_3,\dashv,L)$

The symbol $-$ in the table above means "don't care"—any suitable value will do. The transitions for t and r are not included in the table—just define them to be anything satisfying the restrictions (6.1) and (6.2).

6.1.4 Configurations and Acceptance

At any point in time, the read/write tape of the Turing machine M contains a semi-finite string of the form $y\sqcup^\omega$, where $y\in\Sigma^*$ (y is a finite-length string) and $\sqcup^\omega$ denotes the semi-infinite string

$$\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\cdots.$$

Although the string is infinite, it always has a finite representation, since all but finitely many of the symbols are the blank symbol $\sqcup$.

We define a *configuration* to be an element of $Q\times\{y\sqcup^\omega \mid y\in\Gamma^*\}\times\mathbb{N}$, where $\mathbb{N}=\{0,1,2,\ldots\}$. A configuration is a global state giving a snapshot of all the relevant information about a TM computation at some instant in time. The configuration (p,z,n) specifies the current state p of the finite control, current tape contents z, and current position of the read/write head $n\geq 0$. We usually denote configurations by α,β,γ.

The *start configuration* on input $x\in\Sigma^*$ is the configuration

$$(s,\vdash x\sqcup^\omega,0).$$

The last component 0 means that the machine is initially scanning the left endmarker $\vdash$.

One can define a *next configuration relation* $\rightarrow_M^1$ as with PDAs. For a string $z\in\Gamma^\omega$, let z_n be the nth symbol of z (the leftmost symbol is z_0), and let $s_b^n(z)$ denote the string obtained from z by substituting b for z_n at position n. For example,

$$s_b^4(\vdash baaacabca\cdots) \quad = \quad \vdash baabcabca\cdots.$$

The relation $\rightarrow_M^1$ is defined by

$$(p,z,n)\rightarrow_M^1 \begin{cases} (q,s_b^n(z),n-1) & \text{if } \delta(p,z_n)=(q,b,L),\\ (q,s_b^n(z),n+1) & \text{if } \delta(p,z_n)=(q,b,R).\end{cases}$$

Intuitively, if the tape contains z and if M is in state p scanning the nth tape cell, and δ says to print b, go left, and enter state q, then after that step the tape will contain $s_b^n(z)$, the head will be scanning the $(n-1)$th tape cell, and the new state will be q.

We define the reflexive transitive closure $\to_M^*$ of $\to_M^1$ inductively, as usual:

$$\begin{aligned} \alpha &\to_M^0 \alpha, \\ \alpha &\to_M^{n+1} \beta \quad \text{if} \quad \alpha \to_M^n \gamma \to_M^1 \beta \text{ for some } \gamma, \text{ and} \\ \alpha &\to_M^* \beta \quad \text{if} \quad \alpha \to_M^n \beta \text{ for some } n \geq 0. \end{aligned}$$

The machine M is said to *accept* input $x \in \Sigma^*$ if

$$(s, \vdash x \sqcup^\omega, 0) \to_M^* (t, y, n)$$

for some y and n, and *reject* x if

$$(s, \vdash x \sqcup^\omega, 0) \to_M^* (r, y, n)$$

for some y and n. It is said to *halt* on input x if it either accepts x or rejects x. It is possible that it neither accepts nor rejects, in which case it is said to *loop* on input x. A Turing machine is said to be *total* if it halts on all inputs; that is, if for all inputs it either accepts or rejects. The set $L(M)$ denotes the set of strings accepted by M.

We call a set of strings

- *recursively enumerable* (r.e.) if it is $L(M)$ for some Turing machine M,
- *co-recursively enumerable* if its complement is recursively enumerable, and
- *recursive* if it is $L(M)$ for some *total* Turing machine M.

'Recursive' in this context is just the name for a set accepted by a Turing machine that always halts.

6.1.5 Examples

Example 6.1.3 *Consider the non-context-free language $\{ww \mid w \in \{a,b\}^*\}$. It is a recursive set, because we can give a total TM M for it. The machine M works as follows. On input x, it scans out to the first blank symbol $\sqcup$, counting the number of symbols mod 2 to make sure x is of even length and rejecting immediately if not. It lays down a right endmarker $\dashv$, then repeatedly scans back and forth over the input. In each pass from right to left, it marks the first unmarked a or b it sees with $\acute{}$. In each pass from left to right, it marks the first unmarked a or b it sees with $\grave{}$. It continues until all symbols are marked. For example, on input*

$$aabbaaabba$$

the initial tape contents are

$$\vdash aabbaaabba \sqcup\sqcup\sqcup\cdots$$

and the following are the tape contents after the first few passes:

$$\begin{aligned} &\vdash aabbaaabb\acute{a} \dashv \sqcup\sqcup\sqcup\cdots \\ &\vdash \grave{a}abbaaabb\acute{a} \dashv \sqcup\sqcup\sqcup\cdots \\ &\vdash \grave{a}abbaaab\acute{b}\acute{a} \dashv \sqcup\sqcup\sqcup\cdots \\ &\vdash \grave{a}\grave{a}bbaaab\acute{b}\acute{a} \dashv \sqcup\sqcup\sqcup\cdots \\ &\vdash \grave{a}\grave{a}bbaaa\acute{b}\acute{b}\acute{a} \dashv \sqcup\sqcup\sqcup\cdots \end{aligned}$$

Marking a with ` formally means writing the symbol $\grave{a} \in \Gamma$*; thus*

$$\Gamma = \{a, b, \vdash, \sqcup, \dashv, \grave{a}, \grave{b}, \acute{a}, \acute{b}\}$$

When all symbols are marked, we have the first half of the input string marked with ` and the second half marked with ´.

$$\vdash \grave{a}a\grave{b}\grave{b}\grave{a}a\acute{a}\acute{b}\acute{b}\acute{a} \dashv \sqcup\sqcup\sqcup\cdots$$

The reason we did this was to find the centre of the input string.

The machine then repeatedly scans left to right over the input. In each pass it erases the first symbol it sees marked with ` but remembers that symbol in its finite control (to "erase" really means to write the blank symbol $\sqcup$*). It then scans forward until it sees the first symbol marked with ´, checks that that symbol is the same, and erases it. If the two symbols are no the same, it rejects. Otherwise, when it has erased all the symbols, it accepts.*

In our example, the following would be the tape contents after each pass.

$$\begin{aligned}
&\vdash \grave{a}a\grave{b}\grave{b}\grave{a}a\acute{a}\acute{b}\acute{b}\acute{a} \dashv \sqcup\sqcup\sqcup\cdots\\
&\vdash \sqcup a\grave{b}\grave{b}\grave{a}\sqcup\acute{a}\acute{b}\acute{b}\acute{a} \dashv \sqcup\sqcup\sqcup\cdots\\
&\vdash \sqcup\sqcup\grave{b}\grave{b}\grave{a}\sqcup\sqcup\acute{b}\acute{b}\acute{a} \dashv \sqcup\sqcup\sqcup\cdots\\
&\vdash \sqcup\sqcup\sqcup\grave{b}\grave{a}\sqcup\sqcup\sqcup\acute{b}\acute{a} \dashv \sqcup\sqcup\sqcup\cdots\\
&\vdash \sqcup\sqcup\sqcup\sqcup a\sqcup\sqcup\sqcup\sqcup\acute{a} \dashv \sqcup\sqcup\sqcup\cdots\\
&\vdash \sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup \dashv \sqcup\sqcup\sqcup\cdots
\end{aligned}$$

Example 6.1.4 *We want to construct a total TM that accepts its input string if the length of the string is prime. This language is neither regular nor context-free. We will give a TM implementation of the* sieve of Eratosthenes*, which can be described informally as follows. Say we want to check whether* n *is prime. We write down all the numbers from* 2 *to* n *in order, then repeat the following: find the smallest number in the list, declare it prime, then cross off all multiples of that number. Repeat until each number in the list has been either declared prime or crossed off as a multiple of a smaller prime.*

For example, to check whether 23 *is prime, we would start with all the numbers from* 2 *to* 23*:*

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23

In the first pass, we cross of multiples of 2:

~~2~~ 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ 9 ~~10~~ 11 ~~12~~ 13 ~~14~~ 15 ~~16~~ 17 ~~18~~ 19 ~~20~~ 21 ~~22~~ 23

The smallest number remaining is 3, and this is prime. In the second pass, we cross off multiples of 3:

~~2~~ ~~3~~ ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~ 17 ~~18~~ 19 ~~20~~ ~~21~~ ~~22~~ 23

Then 5 *is the next prime, so we cross off multiples of* 5*; and so forth. Since* 23 *is a prime, it will never be crossed off as a multiple of anything smaller, and eventually we will discover that fact when everything smaller has been crossed off.*

Now we show how to implement this on a TM. Suppose we have a^p *written on the tape. We illustrate the algorithm with* $p = 23$*.*

$$\vdash aaaaaaaaaaaaaaaaaaaaaaa\sqcup\sqcup\sqcup\cdots$$

If $p = 0$ or $p = 1$, reject. We can determine this by looking at the first three cells of the tape. Otherwise, there are at least two a's. Erase the first a, scan right to the end of the input, and replace the last a in the input string with the symbol \$. We now have an a in positions $2, 3, 4, \ldots, p-1$, and a \$ at position p.

$$\vdash \text{␣}aaaaaaaaaaaaaaaaaaaa\$\text{␣␣␣}\cdots$$

Now we repeat the following loop. Starting with the left endmarker $\vdash$, scan right and find the first non-blank symbol, say occurring at position m. Then m is prime (this is the invariant of the loop). If this symbol is the \$, we are done: $p = m$ is prime, so we halt and accept. Otherwise, the symbol is an a. Mark it with $\widehat{\ }$ and everything between there and the left endmarker with $\acute{}$.

$$\vdash \acute{\text{␣}}\widehat{a}aaaaaaaaaaaaaaaaaaa\$\text{␣␣␣}\cdots$$

We will now enter an inner loop to erase all the symbols occurring at positions that are multiples of m. First, erase the a under the $\widehat{\ }$. (Formally, just write the symbol $\widehat{\text{␣}}$.)

$$\vdash \acute{\text{␣}}\widehat{\text{␣}}aaaaaaaaaaaaaaaaaaa\$\text{␣␣␣}\cdots$$

Shift the marks to the right one at a time a distance equal to the number of marks. This can be done by shuttling back and forth, erasing marks on the left and writing them on the right. We know when we are done because the $\widehat{\ }$ is the last mark moved.

$$\vdash \text{␣␣}\acute{a}\widehat{a}aaaaaaaaaaaaaaaaaaa\$\text{␣␣␣}\cdots$$

When this is done, erase the symbol under the $\widehat{\ }$. This is the symbol occurring at position $2m$.

$$\vdash \text{␣␣}\acute{a}\widehat{\text{␣}}aaaaaaaaaaaaaaaaaaa\$\text{␣␣␣}\cdots$$

Keep shifting the marks and erasing the symbol under the $\widehat{\ }$ in this fashion until we reach the end.

$$\vdash \text{␣␣}a\text{␣}a\text{␣}a\text{␣}a\text{␣}a\text{␣}a\text{␣}a\text{␣}a\text{␣}a\text{␣}\acute{\$}\widehat{\text{␣␣␣}}\cdots$$

If we find ourselves at the end of the string wanting to erase the \$, reject—p is a multiple of m not equal to m. Otherwise, go back to the left and repeat. Find the first non-blank symbol and mark it and everything to its left.

$$\vdash \acute{\text{␣}}\acute{\text{␣}}\widehat{a}\text{␣}a\text{␣}a\text{␣}a\text{␣}a\text{␣}a\text{␣}a\text{␣}a\text{␣}a\text{␣}\$\text{␣␣␣}\cdots$$

Alternately erase the symbol under the $\widehat{\ }$ and shift the marks until we reach the end of the string.

$$\vdash \text{␣␣␣␣}a\text{␣}a\text{␣␣␣}a\text{␣}a\text{␣␣␣}a\text{␣}a\text{␣␣␣}\acute{\text{␣}}\acute{\$}\widehat{\text{␣␣␣}}\cdots$$

Go back to the left and repeat.

$$\vdash \acute{\text{␣}}\acute{\text{␣}}\acute{\text{␣}}\acute{\text{␣}}\widehat{a}\text{␣}a\text{␣␣␣}a\text{␣}a\text{␣␣␣}a\text{␣}a\text{␣␣␣}\$\text{␣␣␣}\cdots$$

If we ever try to erase the \$, reject—p is not prime. If we manage to erase all the a's, accept.

6.1.6 Recursive and R.E. Sets

Recall that a set A is *recursively enumerable* (r.e.) if it is accepted by a TM and *recursive* if it is accepted by a *total* TM (one that halts on all inputs).

The recursive sets are closed under complement. (The r.e. sets are not, as we will see later.) That is, if A is recursive, then so it $\sim A = \Sigma^* \setminus A$. To see this, suppose that A is recursive. Then there exists a total TM M such that $L(M) = A$. By switching the accept and reject states of M, we get a total machine M' such that $L(M') = \sim A$.

This construction does not give the complement if M is not total. This is because "rejecting" and "not accepting" are not synonymous for non-total machines. To reject, a machine must enter its reject state. If M' is obtained from M by just switching the accept and reject states, then M' will accept the strings that M rejects and reject the strings that M accepts; but M' will still loop on the same strings that M loops on, so these strings are not accepted or rejected by either machine.

Every recursive set is r.e. but not necessarily vice versa. In other words, not every TM is equivalent to a total TM. (We shall prove this later.) However, if both A and $\sim A$ are r.e., then A is recursive. To see this, suppose that both A and $\sim A$ are r.e. Let M and M' be TMs such that $L(M) = A$ and $L(M') = \sim A$. Build a new machine N that on input x runs both M and M' simultaneously on two different tracks of its tape. Formally, the tape alphabet of N contains symbols

$$\begin{array}{|c|}\hline a \\ \hline c \\ \hline \end{array} \quad \begin{array}{|c|}\hline \widehat{a} \\ \hline c \\ \hline \end{array} \quad \begin{array}{|c|}\hline a \\ \hline \widehat{c} \\ \hline \end{array} \quad \begin{array}{|c|}\hline \widehat{a} \\ \hline \widehat{c} \\ \hline \end{array}$$

where a is a type symbol of M and c is a tape symbol of M'. Thus N's tape may contain a string of the form:

$$\begin{array}{|c|c|c|c|c|c|c|c|}\hline b & \widehat{a} & b & a & b & a & b & a \\ \hline c & c & c & d & d & c & \widehat{c} & d \\ \hline \end{array} \cdots$$

for example. The extra marks $\widehat{\ }$ are placed on the tape to indicate the current positions of the simulated read/write heads of M and M'. The machine N alternately performs a step of M and a step of M', shuttling back and forth between the two simulated tape head positions of M and M' and updating the tape. The current states and transition information of M and M' can be stored in N's finite control. If the machine M ever accepts, then N immediately accepts. If M' ever accepts, then N immediately rejects. Exactly one of these two events must eventually occur, depending on whether $x \in A$ or $x \in \sim A$, since $L(M) = A$ and $L(M') = \sim A$. Then N halts on all inputs and $L(N) = A$.

6.1.7 Decidability and Semi-decidability

A property P of strings is said to be *decidable* if the set of all strings having property P is a recursive set; that is, if there is a total Turing machine that accepts input strings that have property P and rejects those that do not. A property P is said to be *semi-decidable* if the set of strings having property P is an r.e. set; that is, if there is a Turing machine that on input x accepts if x has property P and rejects or loops if not. For example, it is decidable whether a string is of the form ww, because we can construct a Turing machine that halts on all inputs and accepts exactly the strings of this form.

Although you often hear them switched, the adjectives *recursive* and *r.e.* are best applied to sets and *decidable* and *semi-deciable* to properties. The two notions are equivalent, since

$$\begin{aligned} P \text{ is decidable} &\iff \{x \mid P(x)\} \text{ is recursive,} \\ A \text{ is recursive} &\iff \text{“}x \in A\text{” is decidable,} \end{aligned}$$

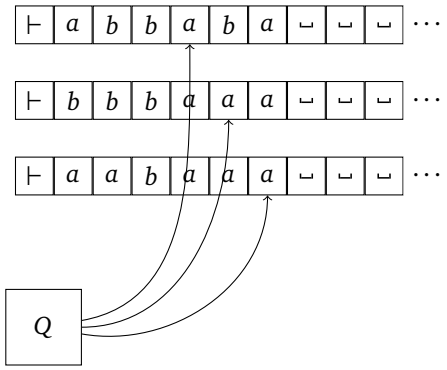
$$P \text{ is semi-decidable} \iff \{x \mid P(x)\} \text{ is r.e.,}$$
$$A \text{ is r.e.} \iff \text{“}x \in A\text{” is semi-decidable.}$$

6.2 Equivalent Models

As mentioned, the concept of computability is remarkably robust. As evidence of this, we will present several different flavours of Turing machines that at a first glance appear to be significantly more or less powerful than the basic model, but in fact are computationally equivalent.

6.2.1 Multiple Tapes

First, we show how to simulate multi-tape Turing machines with single-tape Turing machines. Thus extra tapes do not add any power. A three-tape machine is similar to a one-tape TM except that it has three semi-infinite tapes and three independent read/write heads. Initially, the input occupies the first head and the other two are blank. In each step, the machine reads the first three symbols under its head, and based on this information and the current state, it prints a symbol on each tape, moves the heads (they don't have to move in the same direction), and enters a new state.



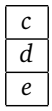
Its transition function is of type

$$\delta : Q \times \Gamma^3 \to Q \times \Gamma^3 \times \{L, R\}^3.$$

Say we have such a machine M. We build a single-tape machine N simulating M as follows. The machine N will have an expanded tape alphabet allowing us to think of its tape as divided into three tracks. Each track will contain the contents of one of M's tapes. We also mark exactly one symbol on each track to indicate that this is the symbol currently being scanned on the corresponding tape of M. The configuration of M illustrated above might be simulated by the following configuration of N.

⊢	⊢	a	b	b	$\widehat{a}$	b	a	␣	␣	␣	⋯
	⊢	b	b	b	a	$\widehat{a}$	a	␣	␣	␣	
	⊢	a	a	b	a	a	$\widehat{a}$	␣	␣	␣	

A type symbol of N is either $\vdash$, an element of Σ, or a triple

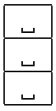


where c, d, e are tape symbols of M, each either marked or unmarked. Formally, we want to take the tape alphabet of N to be

$$\Sigma \cup \{\vdash\} \cup (\Gamma \cup \widehat{\Gamma})^3,$$

where $\widehat{\Gamma} \mathrel{\widehat{=}} \{\widehat{c} \mid c \in \Gamma\}$.

The three elements of $\Gamma \cup \widehat{\Gamma}$ stand for the symbols in corresponding positions on the three tapes of M, either marked or unmarked, and

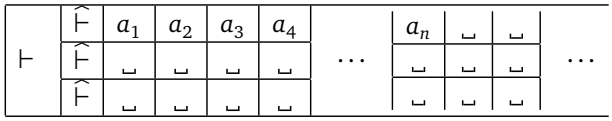


is the blank symbol of N.

On input $x = a_1 a_2 \cdots a_n$, N starts with tape contents

$$\vdash\ a_1\ a_2\ \cdots\ a_n \sqcup\ \sqcup\ \sqcup\ \cdots.$$

It first copies this input to its top track and fills in the bottom two tracks with blanks. It also shifts everything right one cell so that it can fill in the leftmost cells on the three tracks with the simulated left endmarker of M, which in marks with $\widehat{\ }$ to indicate the position of the heads in the starting configuration of M.

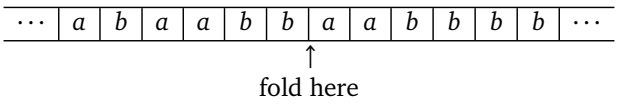


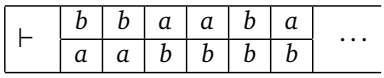
Each step of M is simulated by several steps of N. To simulate one step of M, N starts at the left of the tape, then scans out until it sees all three marks, remembering the marked symbols in its finite control. When it has seen all three, it determines what to do according to M's transition function δ, which it has encoded in its finite control. Based on this information, it goes back to all three marks, rewriting the symbols on each track and moving the marks appropriately. It then returns to the left end of the tape to simulate the next step of M.

Note that all of these steps require only a finite amount of memory to encode, and can therefore be encoded in the finite controller.

6.2.2 Two-Way Infinite Tapes

Two-way infinite tapes do not add any power. Just fold the tape someplace and simulate it on two tracks of a one-way tape:

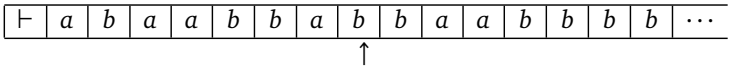




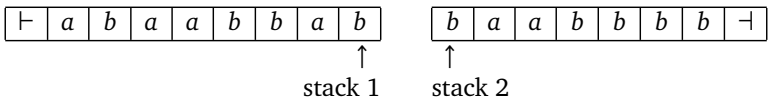
The bottom track is used to simulate the original machine when its head is to the right of the fold, and the top track is used to simulate the machine when its head is to the left of the fold, moving in the opposite direction.

6.2.3 Two Stacks

A machine with a two-way, *read-only* input head and two stacks is as powerful as a Turing machine. Intuitively, the computation of a one-tape TM can be simulated with two stacks by storing the tape contents to the left of the head on one stack and the tape contents to the right of the head on the other stack. The motion of the head is simulated by popping a symbol of one stack and pushing it onto the other. For example,



is simulated by



6.2.4 Counter Automata

A *k-counter automaton* is a machine equipped with a two-way read-only input head and k integer counters. Each counter can store an arbitrary non-negative integer. In each step, the automaton can independently increment or decrement its counters and test them for 0 and can move its input head one cell in either direction. It cannot write on the tape.

A stack can be simulated with two counters as follows. We can assume without loss of generality that the stack alphabet of the stack to be simulated contains only two symbols, say 0 and 1. This is because we can encode finitely many stack symbols as binary numbers of fixed length, say m; then pushing or popping m binary digits can be simulated by pushing and popping m binary digits. Then the contents of the stack can be regarded as a binary number whose least significant bit is on top of the stack. The simulation maintains this number in the first of the two counters and uses the second to effect the stack operations. To simulate pushing a 0 onto the stack, we need to double the value in the first counter. This is done by entering a loop that repeatedly subtracts one from the first counter and adds two to the second until the first counter is 0. The value in the second counter is then twice the original value in the first counter. We can then transfer that value back to the first counter, or just switch the roles of the two counters. To push 1, the operation is the same, except the value of the second counter is incremented once at the end. To simulate popping, we need to divide the counter value by two; this is done by decrementing one counter while incrementing the other counter every second step. Testing the parity of the original counter contents tells whether a simulated 1 or 0 was popped.

Since a two-stack machine can simulate an arbitrary TM, and since two counters can simulate a stack, it follows that a four-counter automaton can simulate an arbitrary TM.

However, we can do even better: a two-counter automaton can simulate a four-counter automaton. When the four-counter automaton has values i, j, k, l in its counters, the two-counter automaton will have the value $2^i 3^j 5^k 7^l$ in its counter. It uses the second counter to effect the counter operations of the four-counter automaton. For example, if the four-counter automaton wants to add one to k (the value of the third counter), then the two-counter automaton would have to multiply the value in its first counter by 5. This is done the same way as above, adding 5 to the second counter for every 1 we subtract from the first counter. To simulate a test for zero, the two-counter automaton has to determine whether the value in its first counter is divisible by 2, 3, 5, or 7, respectively, depending on which counter of the four-counter automaton is being tested.

Combining these simulations, we see that two-counter automata are as powerful as arbitrary Turing machines. However, as you can imagine, it takes an enormous number of steps of the two-counter automaton to simulate one step of the Turing machine.

One-counter automata are not as powerful as arbitrary TMs, although they can accept non-CFLs. For example, the set $\{a^n b^n c^n \mid n \geq 0\}$ can be accepted by a one-counter automaton.

6.2.5 Enumeration Machines

We defined the recursively enumerable (r.e.) sets to be those sets accepted by Turing machines. The term *recursively enumerable* actually comes from a different but equivalent formalism embodying the idea that the elements of an r.e. set can be enumerated one at a time in a mechanical fashion.

Define an *enumeration machine* as follows. It has a finite control and two takes, a read/write *work tape* and a write-only *output tape*. The work tape head can move in either direction and can read and write any element of Γ. The output tape head moves right one cell when it writes a symbol, and it can only write symbols in Σ. There is no input and no accept or reject state. The machine starts in its start state with both tapes blank. It moves according to its transition function like a TM, occasionally writing symbols on the output tape as determined by the transition function. At some point it may enter a special *enumeration state*, which is just a distinguished state of its finite control. When that happens, the string currently being written on the out put take is said to be *enumerated*. The output tape is then automatically erased and the output head moved back to the beginning of the tape (the work tape is left intact), and the machine continues from that point. The machine runs forever. The set $L(E)$ is defined to be the set of all strings in Σ^* that are ever enumerated by the enumeration machine E. The machine might never enter its enumeration state, in which case $L(E) = \emptyset$, or it might enumerate infinitely many strings. The same string may be enumerated more than once.

Enumeration machines and Turing machines are equivalent in computational power.

Theorem 6.2.1 *The family of sets enumerated by enumeration machines is exactly the family of r.e. sets. In other words, a set is $L(E)$ for some enumeration machine E if and only if it is $L(M)$ for some Turing machine M.*

Proof. We show first that given an enumeration machine E, we can construct a Turing machine M such that $L(M) = L(E)$. Let M on input x copy x to one of the three tracks of its tape, then simulate E, using two tracks to record the contents of E's work tape and output tape. For every string enumerated by E, M compares this string to x and accepts if they match. Then M accepts its input x iff x is ever enumerated by E, so the set of strings accepted by M is exactly the set of strings enumerated by E.

Conversely, given a TM M, we can construct an enumeration machine E such that $L(E) = L(M)$. We would like E somehow to simulate M on all possible strings in Σ^* and enumerate those that are accepted.

Here is an approach that doesn't quite work. The enumeration machine writes down the strings in Σ^* one by one on the bottom track of its work tape in some order. For every input string x, it simulates M on input x, using the top track of its work tape to do the simulation. If M accepts x, E copies x to its output tape and enters its enumeration state. It then goes on to the next string.

The problem with this procedure is that M might not halt on some input x, and then E would be stuck simulating M on x forever and would never move on to strings later in the list (and it is impossible to determine in general whether M will ever halt on x, as we shall see later). Thus E should not just list the strings in Σ^* in some order and simulate M on those inputs one at a time, waiting for each simulation to halt before going on to the next, because the simulation might never halt.

The solution to this problem is *timesharing*. Instead of simulating M on the input strings one at a time, the enumeration machine should run several simulations at once, working a few steps on each simulation and then moving on to the next. The work tape of E can be divided into segments separated by a special marker $\# \in \Gamma$, with a simulation of M on a different input string running in each segment. Between passes, E can move way out to the right, create a new segment, and start up a new simulation in that segment on a new input string. For example, we might have E simulate M on the first input for one step, then the first and second inputs for one step each, then the first, second, and third inputs for one step each, and so on. If any simulation needs more space than initially allocated in its segment, the entire contents of the tape to its right can be shifted to the right one cell. In this way M is eventually simulated on all input strings, even if some of the simulations never halt. □

6.3 Universal Machines and Diagonalisation

6.3.1 A Universal Turing Machine

Turing machines are powerful. There exists a Turing machine that can simulate other Turing machines whose descriptions are presented as part of the input. This is no great mystery. One can imagine writing a Java interpreter in Java.

First we need to fix a reasonable encoding scheme for Turing machines over the alphabet $\{0, 1\}$. This encoding should be simple enough that all the data associated with a machine M—the set of states, the transition function, the input and tape alphabets, the endmarker, the blank symbol, and the start, accept, and reject states—can be determined easily by another machine reading the encoded description of M. For example, if the string begins with the prefix

$$0^n 1 0^m 1 0^k 1 0^s 1 0^t 1 0^r 1 0^u 1 0^v 1,$$

this might indicate that the machine has n states represented by the numbers 0 to $n-1$; it has m tape symbols represented by the numbers 0 to $m-1$, of which the first k represent the input symbols; the start, accept, and reject states are s, t and r, respectively; and the endmarker and blank symbol are u and v, respectively. The remainder of the string can consist of a sequence of substrings specifying the transitions in δ. For example, the substring

$$0^p 1 0^a 1 0^q 1 0^b 1 0$$

might indicate that δ contains the transition

$$((p, a), (q, b, L))$$

the direction of the head encoded by the final digit. The exact details of the encoding scheme are not important. The only requirements are that it should be easy to interpret and able to encode all Turing machines up to isomorphism.

Once we have a suitable encoding of Turing machines, we can construct a *universal Turing machine* U such that

$$L(U) \mathrel{\widehat{=}} \{M\#x \mid x \in L(M)\}$$

In other words, presented with (an encoding over $\{0,1\}$ of) a Turing machine M and (an encoding over $\{0,1\}$ of) a string x over M's input alphabet, the machine U accepts $M\#x$ iff M accepts x. The $\#$ symbol is just a symbol in U's input alphabet other than 0 or 1 used to delimit M and x.

The machine U first checks its input $M\#x$ to make sure that M is a valid encoding of a Turing machine and x is a valid encoding of a string over M's input alphabet. If not, it immediately rejects.

If the encodings of M and x are valid, the machine U does a step-by-step simulation of M. This might work as follows. The tape of U is partitioned into three tracks. The description of M is copied to the top track and the string x to the middle track. The middle track will be used to hold the simulated contents of M's tape. The bottom track will be used to remember the current state of M and the current position of M's read/write head. The machine U then simulates M on input x one step at a time, shuttling back and forth between the description of M on its top track and the simulated contents of M's tape on the middle track. In each step, it updates M's state and simulated tape contents as dictated by M's transition function, which U can read from the description of M. If ever M halts or accepts or halts and rejects, then U does the same.

As we have observed, the string x over the input alphabet of M and its encoding over the input alphabet of U are two different things, since the two machines may have different alphabets. If the input alphabet of M is bigger than that of U, then each symbol of x must be encoded as a string of symbols over U's input alphabet. Also, the tape alphabet of M may be bigger than that of U, in which case each symbol of M's tape alphabet must be encoded as a string of symbols over U's tape alphabet. In general, each step of M may require many steps of U to simulate.

6.3.2 Diagonalisation

We now show how to use the universal Turing machine in conjunction with a technique called *diagonalisation* to prove that the halting and membership problems for Turning machines are undecidable. In other words, the sets

$$\begin{aligned} HP &\mathrel{\widehat{=}} \{M\#x \mid M \text{ halts on } x\} \\ MP &\mathrel{\widehat{=}} \{M\#x \mid x \in L(M)\} \end{aligned}$$

are not recursive.

The technique of diagonalisation was first used by Cantor at the end of the nineteenth century to show that there does not exist a one-to-one correspondence between the natural numbers $\mathbb{N}$ and its *power set*

$$2^{\mathbb{N}} = \{A \mid A \subseteq \mathbb{N}\},$$

the set of all subsets of $\mathbb{N}$. In fact, there does not even exist a function

$$f : \mathbb{N} \to 2^{\mathbb{N}}$$

that is onto. This is how Cantor's argument went.

Suppose (for a contradiction) that such an onto function f did exist. Consider an infinite two-dimensional matrix indexed along the top by the natural numbers $0, 1, 2, \ldots$ and down the left by the sets $f(0), f(1), f(2), \ldots$. Fill in the matrix by placing a 1 in position i, j if j is in the set $f(i)$ and 0 if $j \notin f(i)$.

	0	1	2	3	4	5	6	7	8	9	$\cdots$
$f(0)$	1	0	0	1	1	0	1	0	1	1	
$f(1)$	0	0	1	1	0	1	1	0	0	1	
$f(2)$	0	1	0	1	0	1	0	1	1	1	
$f(3)$	1	0	1	1	0	0	0	1	0	1	
$f(4)$	1	1	1	0	1	0	0	1	1	1	$\cdots$
$f(5)$	0	0	1	0	1	1	0	1	0	1	
$f(6)$	0	1	1	0	0	1	0	1	1	0	
$f(7)$	1	1	1	1	0	0	0	0	0	0	
$f(8)$	0	1	0	1	1	0	0	0	1	0	
$f(9)$	1	0	0	1	1	1	0	1	0	1	
$\vdots$					$\vdots$						$\ddots$

The ith row is a bit string describing the set $f(i)$. In the above example, $f(0) = \{0, 3, 4, 6, 8, 9, \ldots\}$ and $f(1) = \{2, 3, 5, 6, 9, \ldots\}$. By our (soon to be proved fallacious) assumption that f is onto, every subset of $\mathbb{N}$ appears as a row of this matrix.

But we can construct a new set that does not appear in the list by complementing the main diagonal of the matrix (hence the term *diagonalisation*). Look at the infinite bit string down the main diagonal (in this example, $001110011\cdots$) and take its Boolean complement (in this example, $110001100\cdots$). This new bit string represents a set B (in this example, $B = \{0, 1, 4, 5, \ldots\}$). But the set B does not appear anywhere in the list down the left side of the matrix, since it differs from every $f(i)$ on at least one element, namely i. This is a contradiction, since every subset of $\mathbb{N}$ was supposed to occur as a row of the matrix, by our assumption that f was onto.

This argument not only works for the natural numbers $\mathbb{N}$, but for any set A whatsoever. Suppose (for contradiction) there existed an onto function from A to its power set:

$$f : A \to 2^A.$$

Let

$$B = \{x \in A \mid x \notin f(x)\}$$

(this is the normal way of *complementing the diagonal*). Then $B \subseteq A$. Since, f is onto, there must exist a $y \in A$ such that $f(y) = B$. Now ask whether $y \in f(y)$ and discover a contradiction:

$$\begin{aligned} y \in f(y) &\iff y \in B && \text{since } B = f(y) \\ &\iff y \notin f(y) && \text{definition of } B. \end{aligned}$$

Thus no such f can exist.

6.3.3 Undecidability of the Halting Problem

We have discussed how to encode descriptions of Turing machines as strings in $\{0,1\}^*$ so that these descriptions can be read and simulated by a universal Turing machine U. The machine U takes as input an encoding of a Turing machine M and a string x and simulates M on input x, and

- halts and accepts if M halts and accepts x,
- halts and rejects if M halts and rejects x, and
- loops if M loops on x.

The machine U does not do any fancy analysis on the machine M to try to determine whether or not it will halt. It just blindly simulates M step by step. If M does not halt on x, then U will just go on happily simulating M forever.

It is natural to ask whether we can do better than just a blind simulation. Might there be some way to analyse M to determine in advance, before doing the simulation, whether M would eventually halt on x? If U could say for sure in advance that M would not halt on x, then it could skip the simulation and save itself a lot of useless work. On the other hand, if U could ascertain that M *would* eventually halt on x, then it could go ahead with the simulation to determine whether M accepts or rejects. We could then build a machine U' that takes as input an encoding of a Turing machine M and a string x, and

- halts and accepts if M halts and accepts x,
- halts and rejects if M halts and rejects x,
- halts and rejects if M loops on x.

This would say that $L(U') = L(U) = MP$ is a recursive set.

Unfortunately, this is not possible in general. There are certainly machines for which it is possible to determine halting by some heuristic or other: machines for which the start state is the accept state, for example. However, there is no general method that gives the right answer for all machines.

We can prove this using Cantor's diagonalisation technique. For $x \in \{0,1\}^*$, let M_x be the Turing machine with input alphabet $\{0,1\}$ whose encoding over $\{0,1\}^*$ is x. (If x is not a legal description of a TM with input alphabet $\{0,1\}^*$ according to our encoding scheme, we take M_x to be some arbitrary but fixed TM with input alphabet $\{0,1\}$, say a trivial TM with one state that immediately halts.) In this way we get a list

$$M_\epsilon, M_0, M_1, M_{00}, M_{01}, M_{10}, M_{11}, M_{000}, M_{001}, M_{010}, \ldots \tag{6.3}$$

containing all possible Turing machines with input alphabet $\{0,1\}$ index by strings in $\{0,1\}^*$. We make sure that the encoding scheme is simple enough that a universal machine can determine M_x from x for the purpose of simulation.

Now consider an infinite two-dimensional matrix indexed along the top by strings in $\{0,1\}^*$ and down the left by the TMs in the list (6.3). The matrix contains an H in position x, y if M_x halts on input y and an L if M_x loops on input y.

	ϵ	0	1	00	01	10	11	000	001	010	$\cdots$
M_ϵ	H	L	L	H	H	H	L	L	H	H	
M_0	L	H	L	H	H	H	H	L	L	H	
M_1	L	L	L	L	H	H	H	H	L	H	
M_{00}	H	H	H	H	H	H	H	H	L	H	
M_{01}	L	H	H	L	L	H	H	H	L	H	$\ldots$
M_{10}	L	L	H	L	L	H	H	H	L	L	
M_{11}	L	L	H	L	L	H	L	L	H	L	
M_{000}	L	H	L	H	L	L	H	L	H	L	
M_{001}	H	H	H	L	L	L	H	L	L	H	
M_{010}	H	H	L	L	L	L	H	L	H	L	
$\ldots$				$\vdots$						$\ddots$	

Here, M_ϵ halts on inputs $\epsilon, 00, 01, 10, 001, 010, \ldots$ and does not halt on inputs $0, 1, 11, 000, \ldots$.

Suppose (for a contradiction) that there existed a *total* machine K accepting the set *HP*; that is, a machine that for any given x and y could determine the (x, y)-th entry of the above table in finite time. Thus on input $M\#x$,

- K halts and accepts if M halts on x, and
- K halts and rejects if M loops on x.

Consider a machine N that on input $x \in \{0,1\}^*$

1. constructs M_x from x and writes $M_x\#x$ on its tape;
2. runs K on input $M_x\#x$, accepting if K rejects and going into a trivial loop if K accepts.

Note that N is essentially complementing the diagonal of the above matrix. Then for any $x \in \{0,1\}^*$,

$$\begin{array}{lcll} N \text{ halts on } x & \iff & K \text{ rejects } M_x\#x & \text{definition of } N \\ & \iff & M_x \text{ loops on } x & \text{assumption about } K. \end{array}$$

This says that N's behaviour is different from every M_x on at least one string, namely x. But the list (6.3) was supposed to contain all Turing machines over the alphabet $\{0,1\}$, including N. This is a contradiction.

The fallacious assumption that lead to the contradiction was that it is possible to determine the entries of the matrix effectively; in other words, that there existed a Turing machine K that given M and x could determine in a finite time whether or not M halts on x.

One can always simulate a given machine on a given input. If the machine ever halts, then we will know this eventually, and we can stop the simulation and say that it halted; but if not, there is no way in general to stop after a finite time and say for certain that it will never halt.

6.3.4 Undecidability of the Membership Problem

The membership problem is also undecidable. We can show this by *reducing* the halting problem to it. In other words, we show that if there is a way to decide membership in general, we could use this as a

subroutine to decide halting in general. But we just showed that halting is undecidable, so membership must be undecidable too.

Here is how we would use a total TM that decides membership as a subroutine to decide halting. Given a machine M and an input x, suppose we wanted to find out whether M halts on x. Build a new machine N that is exactly like M, except that it accepts whenever M would either accept or reject. The machine N can be constructed from M simply by adding a new accept state and making the old accept and reject states transfer to this new accept state. Then for all x, N accepts x iff M halts on x. The membership problem for M and x (asking whether $x \in L(N)$) is therefore the same as the halting problem for M and x (asking whether M halts on x). If the membership problem were decidable, then we could decide whether M halts on x by constructing N and asking whether $x \in L(N)$. But we have shown that the halting problem is undecidable, therefore the membership problem must also be undecidable.

6.4 Decidable and Undecidable Problems

Here are some examples of decision problems involving Turing machines. Is it decidable whether a given Turing machine

1. has at least 481 states?
2. takes more than 481 steps on input ϵ?
3. takes more than 481 steps on *some* input?
4. takes more than 481 steps on *all* inputs?
5. ever moves its head more than 481 tape cells away from the left endmarker on input ϵ?
6. accepts the null string ϵ?
7. accepts any string at all?
8. accepts every string?
9. accepts a finite set?
10. accepts a regular set?
11. accepts a context-free language?
12. accepts a recursive set?
13. is equivalent to a Turing machine with a shorter description?

Problems 1–5 are decidable and problems 6–13 are undecidable (proofs below). We will show that problems 6–12 are undecidable by showing that a decision procedure for one of these problems could be used to construct a decision procedure for the halting problem, which we know is impossible. Problem 13 is a little more difficult, and will be left as an exercise (Exercise 6.4.1). Translated into modern terms, Problem 13 is the same as determining whether there exists a shorter Java program equivalent to a given one.

The best way to show that a problem is decidable is to give a total Turing machine that accepts exactly the "yes" instances. Because it must be total, it must also reject the "no" instances; in other words, it must not loop on any input.

Problem 1 is easily decidable, since the number of states of M can be read off from the encoding of M. We can build a Turing machine that, given the encoding of M written on its input tape, counts the number of states of M and accepts or rejects depending on whether that number is at least 481.

Problem 2 is decidable, since we can simulate M on input ϵ with a universal machine for 481 steps (counting up to 481 on a separate track) and accept or reject depending on whether M has halted by that time.

Problem 3 is decidable: we can just simulate M on all inputs of length at most 481 for 481 steps. If M takes more than 481 steps on some input, then it will take more than 481 steps on some input of length at most 481, since in 481 steps in can read at most the first 481 symbols of the input.

The argument for 4 is similar. If M takes more than 481 steps on all inputs of length at most 481, then it will take more than 481 steps on all inputs.

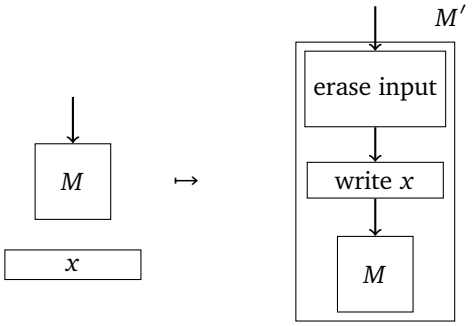
For problem 5, if M never moves more than 481 tape cells away from the left endmarker, then it will either halt or loop in such a way that we can detect the looping after a finite time. This is because if M has k states and m tape symbols, and never moves more than 481 tape cells away from the left endmarker, then there are only $482km^{481}$ configurations it could possibly ever be in, one for each choice of head position, state, and tape contents that fit within 481 tape cells. If it runs for any longer than that without moving more than 481 tape cells away from the end marker, then in must be in a loop, because it must have repeated a configuration. This can be detected by a machine that simulates M, counting the number of steps M takes on a separate track and declaring M to be in a loop if the bound of $482km^{481}$ is ever exceeded.

Problems 6–12 are undecidable. To show this, we show that the ability to decide any one of these problems could be used to decide the halting problem. Since we know that the halting problem is undecidable, these problems must be undecidable too. This is called a *reduction*.

Let's consider 6 first (although the construction will take care of 7 and 9 as well). We will show that it is undecidable whether a given machine accepts ϵ, because the ability to decide this question would give the ability to decide the halting problem, which we know is impossible.

Suppose we could decide whether a given machine accepts ϵ. We could then decide the halting problem as follows. Say we are given a Turing machine M and a string x, and we wish to determine whether M halts on x. Construct from M a new machine M' that does the following on input y:

1. erases its input y;
2. writes x on its tape (M' has x hard-wired in its finite control);
3. runs M on input x (M' also has a description of M hard-wired in its finite control);
4. accepts if M halts on x.



Note that M' does the same thing on all inputs y; if M halts on x, then M' accepts its input y; and if M does not halt on x, then M' does not halt on y, therefore does not accept y. Moreover, this is true for every y. Thus

$$L(M') = \begin{cases} \Sigma^* & \text{if } M \text{ halts on } x, \\ \emptyset & \text{if } M \text{ does not halt on } x. \end{cases}$$

Now if we could decide whether a given machine accepts the null string ϵ, we could apply this decision procedure to the M' just constructed, and this would tell whether M halts on x. In other words, we could obtain a decision procedure for halting as follows: given M and x, construct M' and then ask whether M' accepts ϵ. The answer to the latter question is "yes" iff M halts on x. Since we know the halting problem is undecidable, it must also be undecidable whether a given machine accepts ϵ.

Similarly, if we could decide whether a given machine accepts any string at all, or whether it accepts every string, or whether the set of strings it accepts is finite, we could apply any of these decision procedures to M' and this would tell whether M halts on x. Since we know that the halting problem is undecidable, all of these problems are undecidable too.

To show that 9–12 are undecidable, pick your favourite r.e. but non-recursive set A (*HP* or *MP* will do) and modify the above construction as follows. Given M and x, build a new machine M'' that does the following on input y:

1. saves y on a separate track of its tape;
2. writes x on a different track (x is hard-wired in the finite control of M'');
3. runs M on input x (M is also hard-wired in the finite control of M'');
4. if M halts on x, then M'' runs a machine accepting A on its original input y, and accepts if that machine accepts.

Either M does not halt on x, in which case the simulation in step 3 never halts and M'' never accepts any string; or M does halt on x, in which case M'' accepts its input y iff $y \in A$. Thus

$$L(M'') = \begin{cases} A & \text{if } M \text{ halts on } x, \\ \emptyset & \text{if } M \text{ does not halt on } x. \end{cases}$$

Since A is neither recursive, context-free, nor regular, and $\emptyset$ is all three of these things, if one could decide whether a given TM accepts a recursive, context-free language, or regular set, then one could apply this decision procedure to M'' and this would tell whether M halts on x.

Exercise 6.4.1 *A TM is* minimal *if it has the fewest states among all TMs that accept the same set. Prove that there does not exist an infinite r.e. set of minimal TMs.*

6.4.1 Reduction

There are two main techniques for showing that problems are undecidable: *diagonalisation* and *reduction*. We have seen examples of both types.

Once we have established that a problem such as *HP* is undecidable, we can show that another problem B is undecidable by reducing *HP* to B. Intuitively, this means we can manipulate instances of *HP* to make them look like instances of the problem B in such a way that "yes" instances of *HP* become "yes" instances of B and "no" instances of *HP* becomes "no" instances of B. Although we cannot tell effectively whether a given instance of *HP* is a "yes" instance, the manipulation preserves "yes"-ness and "no"-ness. If there existed a decision procedure for B, then we could apply it to the disguised instances of *HP* to decide membership in *HP*. In other words, combining a decision procedure for B with the manipulation procedure would give a decision procedure for HP. Since we have already shown that no such decision procedure for *HP* can exist, we can conclude that no such decision procedure for B can exist.

We can give an abstract definition of reduction and prove a general theorem that will save us a lot of work in undecidability proofs from now on.

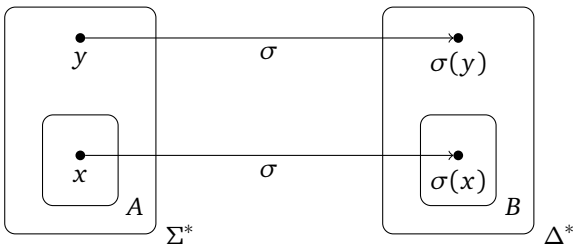
Given sets $A \subseteq \Sigma^*$ and $B \subseteq \Delta^*$, a (many-one) *reduction* of A to B is a computable function

$$\sigma : \Sigma^* \to \Delta^*$$

such that for all $x \in \Sigma^*$,

$$x \in A \iff \sigma(x) \in B \tag{6.4}$$

In other words, strings in A must go to strings in B under σ, and strings not in A must go to strings not in B under σ.



The function σ need not be one-to-one or onto—indeed, it may even map all elements of A to a single element of B and all elements not in A to a single element not in B. It must, however, be *total* and *effectively computable*. This means that σ must be computable by a total Turing machine that on any input x halts with $\sigma(x)$ written on its tape. When such a reduction exists, we say that A is reducible to B via the map σ, and we write $A \leq_m B$. The subscript m, which stands for "many-one," is used to distinguish this relation from other types of reducibility relations.

The relation $\leq_m$ of reducibility between languages is transitive: if $A \leq_m B$ and $B \leq_m C$, then $A \leq_m C$. This is because if σ reduces A to B and τ reduces B to C, then $\tau \circ \sigma$, the composition of σ and τ, is computable and reduces A to C.

Although we have not mentioned it explicitly, we have used reductions in the previous material to show that various problems are undecidable.

Remark 6.4.2 *Even though $\leq_m$ is called a many-one reduction, its key property is not that it is many-to-one, but that it is total and and effectively computable.*

Example 6.4.3 *In showing that it is undecidable whether a given TM accepts the null string, we constructed from a given TM M and string x a TM M' that accepted the null string iff M halts on x. In this example,*

$$\begin{aligned} A &= \{M\#x \mid M \text{ halts on } x\} = HP, \\ B &= \{M \mid \epsilon \in L(M)\}, \end{aligned}$$

and σ is the computable map $M\#x \mapsto M'$.

Example 6.4.4 *In showing that it is undecidable whether a given TM accepts a regular set, we constructed from a given TM M and string x a TM M'' such that $L(M'')$ is a nonregular set if M halts on x and $\emptyset$ otherwise. In this example,*

$$\begin{aligned} A &= \{M\#x \mid M \text{ halts on } x\} = HP, \\ B &= \{M \mid L(M) \text{ is regular}\}, \end{aligned}$$

and σ is the computable map $M\#x \mapsto M''$.

Here is a general theorem that will save us some work.

Theorem 6.4.5

1. *If $A \leq_m B$ and B is r.e., then so is A. Equivalently, if $A \leq_m B$ and A is not r.e., then neither is B.*
2. *If $A \leq_m B$ and B is recursive, then so is A. Equivalently, if $A \leq_m B$ and A is not recursive, then neither is B.*

Proof. 1) Suppose $A \leq_m B$ via the map σ and B is r.e. Let M be a TM such that $B = L(M)$. Build a machine N for A as follows: on input x, first compute $\sigma(x)$, then run M on input $\sigma(x)$, accepting if M accepts. Then

$$\begin{aligned} N \text{ accepts } x &\iff M \text{ accepts } \sigma(x) && \text{definition of } N \\ &\iff \sigma(x) \in B && \text{definition of } M \\ &\iff x \in A && \text{by (6.4).} \end{aligned}$$

2) Recall that a set is recursive iff both it and its complement are r.e. Suppose $A \leq_m B$ via the map σ and B is recursive. Note that $\sim A \leq_m \sim B$ via the same σ (Check the definition!). If B is recursive, then both B and $\sim B$ are r.e. By 1), both A and $\sim A$ are r.e., thus A is recursive. □

We can use Theorem 6.4.5(1) to show that certain sets are not r.e. and Theorem 6.4.5(2) to show that certain sets are not recursive. To show that a set B is not r.e., we need only give a reduction from a set A we already know is not r.e. (such as $\sim HP$) to B. By Theorem 6.4.5(1), B cannot be r.e.

Example 6.4.6 *Let's illustrate by showing that neither the set*

$$FIN = \{M \mid L(M) \text{ is finite}\}$$

nor its complement is r.e. We show that neither of these sets is r.e. by reducing $\sim HP$ to each of them,

$$\sim HP = \{M\#x \mid M \text{ does not halt on } x\}:$$

(a) $\sim HP \leq_m FIN$,

(b) $\sim HP \leq_m \sim FIN$

Since we already know that $\sim HP$ is not r.e., it follows from Theorem 6.4.5(1) that neither FIN nor $\sim FIN$ is r.e.

For (a), we want to give a computable map σ such that

$$M\#x \in \sim HP \iff \sigma(M\#x) \in FIN$$

In other words, from $M\#x$ we want to construct a Turing machine $M' = \sigma(M\#x)$ such that

$$M \text{ does not halt on } x \quad \iff \quad L(M') \text{ is finite.} \tag{6.5}$$

Note that the description of M' can depend on M and x. In particular, M' can have a description of M and the string x hard-wired in its finite control if desired.

We have actually already given a construction satisfying (6.5). Given $M\#x$, construct M' such that on all inputs y, M' takes the following actions:

1. *erases its input y;*
2. *writes x on its tape (M' has x hard-wired in its finite control);*
3. *runs M on input x (M' also has a description of M hard-wired in its finite control);*
4. *accepts if M halts on x.*

If M does not halt on input x, then the simulation in step 3 never halts, and M' never reaches step 4. In this case M' does not accept its input y. This happens the same way for all inputs y, therefore in this case, $L(M) = \emptyset$. On the other hand, if M does halt on x, then the simulation step 3 halts, and y is accepted in step 4. Moreover, this is true for all y. In this case, $L(M) = \Sigma^*$. Thus

$$\begin{array}{lcll} M \text{ halts on } x & \Rightarrow & L(M') = \Sigma^* & \Rightarrow \quad L(M') \text{ is infinite,} \\ M \text{ does not halt on } x & \Rightarrow & L(M') = \emptyset & \Rightarrow \quad L(M') \text{ is finite.} \end{array}$$

Thus (6.5) is satisfied. Note that this is all we have to do to show that FIN is not r.e.: we have given the reduction (a), so by Theorem 6.4.5(1) we are done.

There is a common pitfall here that we should be careful to avoid. It is important to observe that the computable map σ that produces a description of M' from M and x does not need to execute the program 1–4. It only produces the description of a machine M' that does so. The computation of σ is quite simple—it does not involve the simulation of any other machines or anything complicated at all. It merely takes a description of a Turing machine M and a string x and plugs them into a general description of a machine that executes 1–4. This can be done quite easily by a total TM, so σ is total and effectively computable.

Now (b). By definition of reduction, a map reducing $\sim HP$ to $\sim FIN$ also reduces HP to FIN, so it suffices to give a computable map τ such that

$$M\#x \in HP \iff \tau(M\#x) \in FIN$$

In other words, from M and x we want to construct a Turing machine $M'' = \tau(M\#x)$ such that

$$M \text{ halts on } x \quad \iff \quad L(M'') \text{ is finite.} \tag{6.6}$$

Given $M\#x$, construct a machine M'' that on input

1. *saves y on a separate track;*
2. *writes x on the tape;*
3. *simulates M on x for $|y|$ steps (it erases one symbol of y for each step of M on x that it simulates);*
4. *accepts if M has* not *halted within that time, otherwise rejects.*

Now if M never halts on x, then M'' halts and accepts y in step 4 after $|y|$ steps of the simulation, and this is true for all y. In this case, $L(M'') = \Sigma^*$. On the other hand, if M does halt of x, then it does so after some finite number of steps, say n. Then M'' accepts y in 4 if $|y| < n$ (since the simulation in 3 has not finished by $|y|$ steps) and rejects y in 4 if $|y| > n$ (since the simulation in 3 does not have time to complete). In this case M'' accepts all strings of length less than n and rejects all strings of length n or greater, so $L(M'')$ is a finite set. Thus,

$$\begin{aligned} M \text{ halts on } x \quad &\Rightarrow \quad \{y \mid |y| < \text{ running time of } M \text{ on } x\} \\ &\Rightarrow \quad L(M'') \text{ is finite,} \end{aligned}$$

$$\begin{aligned} M \text{ does not halt on } x \quad &\Rightarrow \quad L(M'') = \Sigma^* \\ &\Rightarrow \quad L(M'') \text{ is infinite.} \end{aligned}$$

Then (6.6) is satisfied.

It is important that the functions σ and τ in these two reductions can be computed by Turing machines that always halt.

Part IV

Graph Theory

Chapter 7

Graph Theory

7.1 Motivational problems

Graph theory has many practical applications in various disciplines, including, to name a few, biology, computer science, economics, engineering, informatics, linguistics, mathematics, medicine and social science. Graphs are excellent modelling tools.

Here are a few classic problems graph theory has been used to address.

7.1.1 The Bridges of Königsberg

This was the first problem to be stated then solved using graph theory.

Figure 7.1 shows the layout of the bridges of Königsberg. The Pregal River in Königsberg has two banks, labelled B_1 and B_2, and it splits to form two islands, labelled I_1 and I_2. These islands were connected to each other with seven bridges, as show in the figure. The problem was to make a round trip through downtown Königsberg, traversing each bridge exactly once.

This problem can be represented using the graph show in Figure 7.2.

The problem now is to start at some vertex (black dot), go along each line exactly one, and end up at the starting dot. We shall see that Euler gave an exact characterisation of when such problems have solutions, in terms of each vertex having an even number of edges going into it, thereby showing in particular that the bridges of Königsberg has no solution.

Exercise 7.1.1 *How many additional bridges need to be built in Königsberg to make the problem solvable? Where?*

7.1.2 World Wide Web Communities

This problem is about discovering communities on the World Wide Web (WWW).

The World Wide Web can be modelled as a graph, where the web pages are represented as vertices and the hyperlinks between them are represented by edges in the graph. One can discover interesting information by examining this graph. For example, the graph in Figure 7.3 is called a *Web community*, because the vertices represent two different classes of objects, and each vertex representing one type of object is connected to every vertex representing the other kind of objects. In graph theory, such a

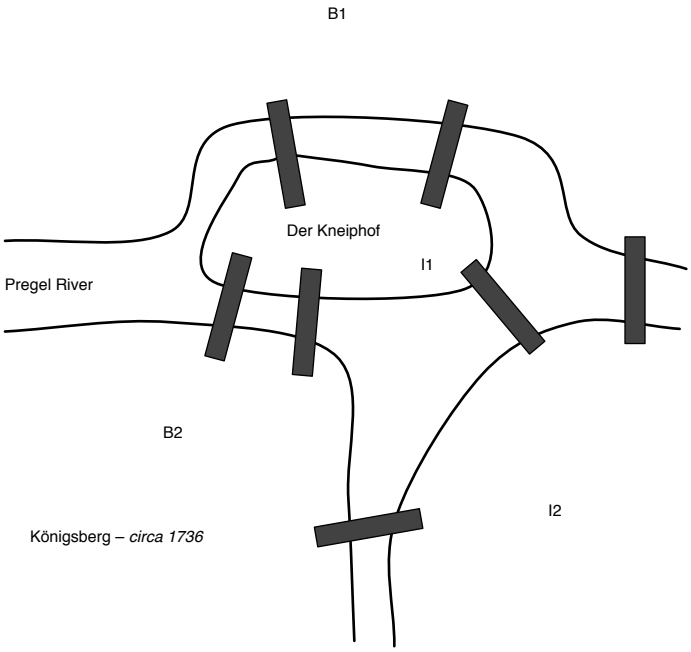


Figure 7.1: The bridges of Königsberg (not to scale)

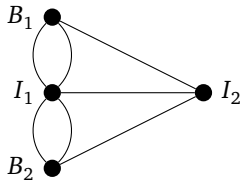


Figure 7.2: The graph of the bridges of Königsberg problem

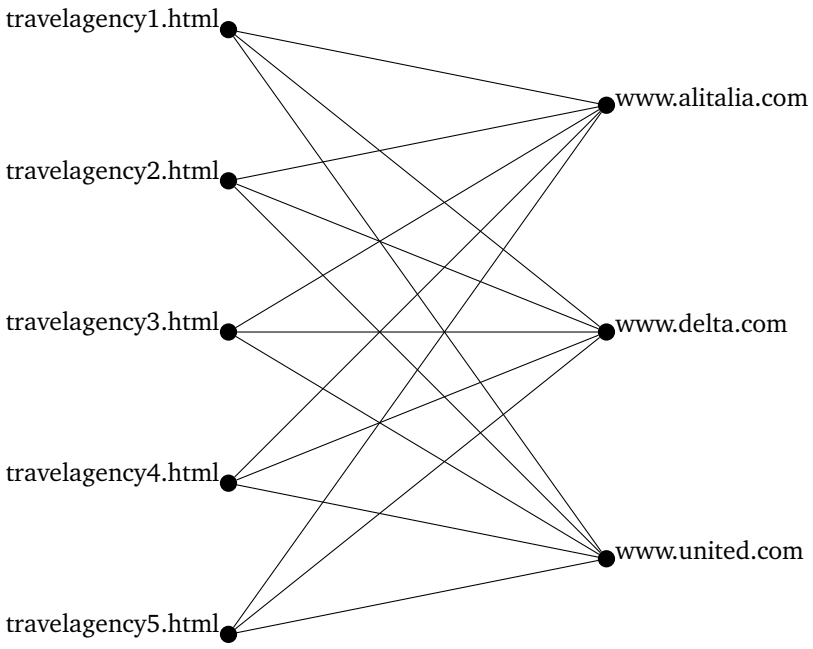


Figure 7.3: World Wide Web Community

graph is called a *complete bipartite graph*. Such a graph exists because the vertices in each group are competitors, and hence do not have links to each others websites.

Such Web communities can be discovered by finding *complete bipartite subgraphs* in the Web graph. Web community information can be used for marketing purposes or for examining the relationships among companies in a given industry.

7.1.3 Job Assignments

This problem deals with job assignments.

A certain company has five different jobs labelled by J_i, where $i \in \{1..5\}$, and wants to hire people for these positions. The seven applicants, labelled by A_j for $j \in \{1..7\}$, are qualified for some of the positions but not for others. An edge between an applicant A_j and a job J_i means that applicant A_j is qualified for job J_i. Figure 7.4 provides an example. This graph is an example of a *bipartite graph*, where the vertices represent two kinds of objects: the applicants and the jobs. The question in this problem is: Can the company fill all of its open positions with qualified applicants?

Exercise 7.1.2 *Design an algorithm to find suitable job assignments.*

Exercise 7.1.3 *Assume that the edges of the job graph are weighted with an indication of how suitable the applicant is for a given job. Design an algorithm which selects the most suitable job assignment.*

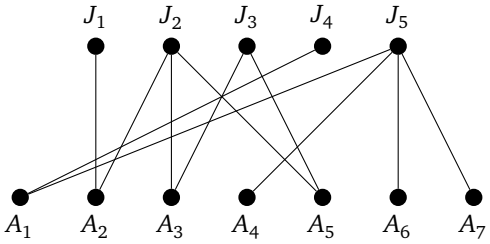


Figure 7.4: Bipartite graph describing job qualifications

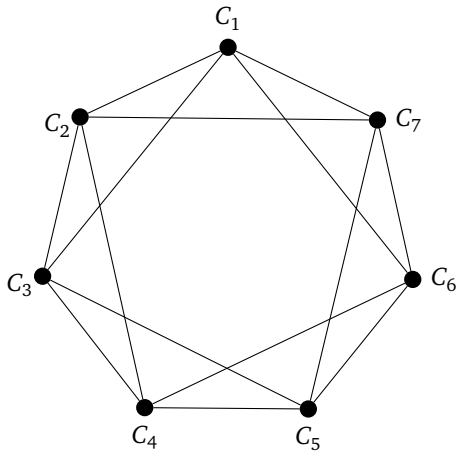


Figure 7.5: Connected chemicals should not be stored togethers

7.1.4 Storing Volatile Chemicals

The next problem illustrates the use of graph theory when determining the minimum number of warehouses needed to store volatile chemicals.

A particular chemical factory produces seven different kinds of chemicals, labelled as $C_1, \ldots, C_7$. For safety reasons some of the chemicals should not be stored in the same warehouse. The graph depicted in Figure 7.5 show the situation of volatility between the chemicals, where an edge between chemicals C_i and C_j indicates a grave danger in storing these chemicals in the same warehouse, whereas the absence of an edge indicates that it is safe to store the chemicals in the same warehouse. Here is the question: What is the minimum number of warehouses the factory needs in order to store its chemical products safely.

If we think of each warehouse as a specific *colour* assigned to each vertex, the problem boils down to using the least number of colours to assign to the vertices of the graph such that no two vertices with an edge between them receive the same colour. The least number of colours needed to colour the vertices of a graph in this manner is called the *chromatic number* of the graph. We are able to assign four colours to the vertices the graph in Figure 7.5 such that no edge is between two vertices of the

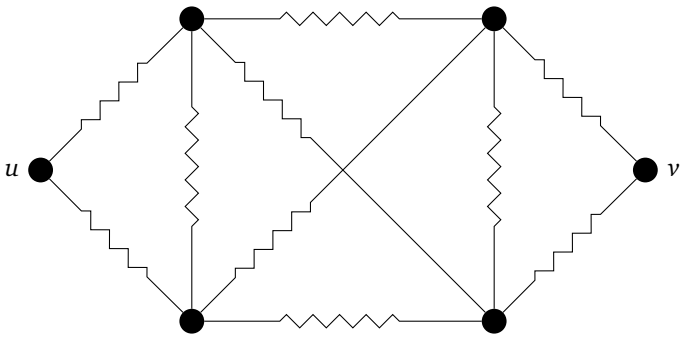


Figure 7.6: Electrical resistance network

same colour. We can, for example, colour vertex C_1 with colour 1, C_2 with colour 2, C_3 with colour 3, C_4 with colour 1, C_5 with colour 2, C_6 with colour 3, and C_7 with colour 4. The graph cannot be vertex coloured using three colours, and hence the chromatic number of the graph is four; that is, the minimum number of warehouses needed for our problem is four.

In general, vertex colouring of graphs is a hard problem from an algorithmic point of view. There is no easy solution that can be applied to all such problems.

7.1.5 Electrical Networks

In Figure 7.6 each edge represents a fixed electrical resistance of 1Ω (*ohm*). How can we compute the resistance between the vertices u and v? This problem is easy to solve for certain kinds of networks (consisting of resisters in serial and/or parallel), but harder for other networks, where a significant physical insight is needed.

7.2 Basic definitions

Definition 7.2.1 (Graph) *A* graph *is an ordered triple $G = (V, E, \phi)$, where*

1. $V \neq \emptyset$
2. $V \cap E = \emptyset$
3. $\phi : E \to \mathscr{P}(V)$ *is a map such that* $|\phi(e)| \in \{1, 2\}$ *for each* $e \in E$.

The elements of V are the *vertices* of G, and the elements of E are the *edges* of G. The map ϕ is called an *edgemap*, and the vertices in $\phi(e)$ are called the *endvertices* of the edge e. When the ordered triple defining a graph G is not given, then the set $V(G)$ will always denote the set of vertices of G, and $E(G)$ will always denote the set of edges of G.

Example 7.2.2 *Consider the diagram shown in Figure 7.7. Here we see that $G = (V, E, \phi)$, where $V = \{u_1, u_2, u_3, u_4, u_5\}$, $E = \{e_1, e_2, e_3, e_4, e_5, e_6\}$, and the edgemap ϕ given by:*

$$\phi(e_1) \quad = \quad \{u_1, u_2\},$$

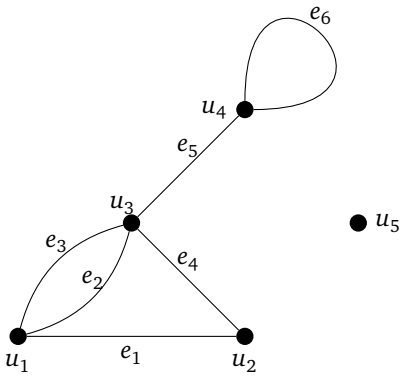


Figure 7.7: A graph with five vertices and six edges

$$\begin{aligned}
\phi(e_2) &= \phi(e_3) = \{u_1, u_3\},\\
\phi(e_4) &= \{u_2, u_3\},\\
\phi(e_5) &= \{u_3, u_4\},\\
\phi(e_6) &= \{u_4\}.
\end{aligned}$$

Generally, we will use the diagrams to represent graphs.

Definition 7.2.3 *Let $G = (V, E, \phi)$ be a graph.*

- *Vertices u and v in V are* adjacent *or* neighbours*, if they are the endvertices of some edge $e \in E$. That is, they are adjacent if there is an $e \in E$ such that $\phi(e) = \{u, v\}$.*
- *Two distinct edges e and f are* adjacent *to each other, if they have a common endvertex. That is, if $\phi(e) \cap \phi(f) \neq \emptyset$.*
- *A vertex u and an edge e are* incident *if $u \in \phi(e)$—that is, if u is an endvertex of e.*
- *A* loop *is an edge e, whose endvertices are equal. That is, $|\phi(e)| = 1$.*
- *We say that $E' \subseteq E$ is a set of* multiple edge *or* parallel edge*, if $|E'| \geq 2$ and all $e' \in E'$ have the same set of endvertices. That is, $\phi(e') = \phi(f')$ for all $e', f' \in E'$.*
- *A vertex u is called* isolated *if it is not an endvertex of any edge. That is, $u \notin \phi(e)$ for all $e \in E$.*

Example 7.2.4 *Regarding to the graph given in Figure 7.7, we see the following: The endvertices of e_2 are u_1 and u_3. The vertices u_1 and u_2 are adjacent, but u_1 and u_4 are not; so u_1 and u_2 are neighbours, but u_2 and u_4 are not. The vertex u_4 is adjacent to itself. The edges e_1 and e_2 are adjacent, but e_1 and e_5 are not. The edge e_6 is adjacent to both itself and to e_5. The edge e_6 is the only loop in the graph. The edges e_2 and e_3 are parallel. The vertex u_5 is isolated.*

When tackling a problem that can be phrased in graph-theoretic terms, often it can be reduced to a problem involving a graph having no multiple edges or any loops. Such graphs constitute an important class called *simple graphs*. Since there is at most one edge between a pair of vertices in a simple graph, the edges are in one-to-one correspondence with their distinct endvertices. Therefore, a simple graph can be defined without the endmap ϕ from Definition 7.2.1. Because of the importance of simple graphs, we state their formal definition separately.

Definition 7.2.5 (Simple Graph) *A* simple graph *is an ordered pair $G = (V, E)$, where V is a nonempty set of vertices and E is a set of 2-elements subsets of V, that is,*

$$\begin{aligned} E &\subseteq \{X \mid X \subseteq V, |X| = 2\} \\ &= \{\{u, v\} \mid u, v \in V, u \neq v\}. \end{aligned}$$

Note, because the edges E are a set, this means that no edge can be repeated.

Definition 7.2.6 (Null Graph) *The* null graph *on n vertices is the graph N_n, where*

$$\begin{aligned} V(N_n) &= \{u_1, u_2, \ldots, u_n\}, \\ E(N_n) &= \emptyset. \end{aligned}$$

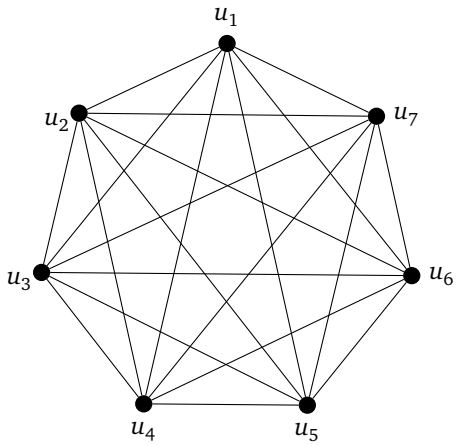
That is, the null graph has no edges.

Example 7.2.7 (Complete Graph) *The* complete graph *on n vertices is the simple graph K_n, where*

$$\begin{aligned} V(K_n) &= \{u_1, u_2, \ldots, u_n\}, \\ E(K_n) &= \{\{u_i, u_j\} \mid 1 \leq i < j \leq n\}. \end{aligned}$$

That is, every pair of distinct vertices is connected by an edge.

Here is the complete graph K_7.

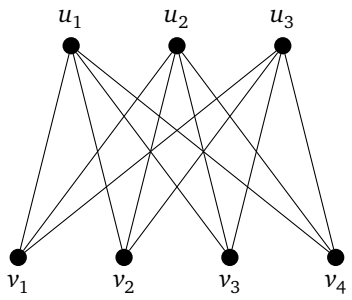


Definition 7.2.8 (Bipartite graph) *A* bipartite graph *(or bigraph) is a graph whose vertices can be divided into two disjoint sets U and V such that every edge connects a vertex in U to one in V.*

Example 7.2.9 (Complete bipartite graph) *The* complete bipartite (m,n)-graph on $m+n$ vertices *is the simple graph $K_{m,n}$, where*

$$\begin{aligned} V(K_{m,n}) &= \{u_1,\ldots,u_m\} \cup \{v_1,\ldots,v_n\}, \\ E(K_{m,n}) &= \{\{u_i,v_j\} \mid 1 \le i \le m, 1 \le j \le m\}. \end{aligned}$$

The following is the complete bipartite graph $K_{3,4}$



Many properties of graphs must be stated in terms of numerical values associated with the graph or some of its components. The first attribute we define is the *degree* of a vertex in a graph.

Definition 7.2.10 (Degree) *Let $G = (V,E,\phi)$ be a graph, an $u \in V$ a vertex of G. The* degree *of u, denoted, $d_G(u)$, or $d(u)$ when there is no danger of ambiguity, is defined by*

$$d_G(u) = \left|\{e \in E \mid e \in \phi(e), |\phi(e)| = 2\}\right| + 2\left|\{e \in E \mid e \in \phi(e), |\phi(e)| = 1\}\right|.$$

Informally speaking, $d_G(u)$ just tallies the number of 'edge ends' connected to u. Thus it makes intuitive sense to count the loops twice, since both of a loop's ends go into its endvertex.

Theorem 7.2.11 (Hand-Shaking Theorem) *For a graph G we have:*

$$\sum_{u \in V(G)} d_G(u) = 2|E(G)|.$$

Proof. Exercise. □

A consequence of the Hank-Shaking Theorem is that there is always an even number of vertices in a graph having odd degree.

When studying a specific graph, many times our attention is focussed solely on a special part of the graph, perhaps on a smaller graph lying inside the larger graph. This motivates the following definition of a *subgraph*.

Definition 7.2.12 (Subgraph) *For graphs $G' = (V',E',\phi')$ and $G = (V,E,\phi)$, we say that G' is a* subgraph *of G if*

1. *$V' \subseteq V$,*
2. *$E' \subseteq E$,*

3. $\phi'(e) = \phi(e)$ *for all* $e \in E'$.

We denote the subgraph relation on graphs using the standard subset notion $\subseteq$.

The following are properties of the subgraph relation:

- $G \subseteq G$, every graph is a subgraph of itself.
- If $G \subseteq G'$ and $G' \subseteq G$, then $G = G'$.
- If $G \subseteq G''$ and $G'' \subseteq G'$, then $G \subseteq G'$.

Subgraphs are often obtained from a specific set of vertices or edges, as described in the following:

Definition 7.2.13 *For a nonempty subset* $W \subseteq V(G)$ *of vertices, the subgraph of* G induced by W*, denoted* $G[W]$*, is the graph with the vertex set* W *whose edge set consists of all edges of* G *having their endvertices in* W.

For a nonempty subset $F \subseteq E(G)$ *of edges, the subgraph of* G induced by F*, or simply* formed by F*, denoted* $G[F]$*, is the graph with edge set* F *whose vertex set consists of all the endvertices of edges of* F.

Definition 7.2.14 (Clique) *A* clique *in a graph* $G = (V, E, \phi)$ *is a subset of the vertex set* $C \subseteq V$ *such that for every two vertices in* C *there is an edge connecting the two. This is equivalent to saying that the subgraph induced by* C *is complete.*

The *maximum clique problem* arises in the following real-world setting. Consider a social network, where the graph's vertices represent people, and the graph's edges represent mutual acquaintance. To find a largest subset of people who all know each other, one can systematically inspect all subsets, a process that is too time-consuming to be practical for social networks comprising more than a few dozen people. Although this brute-force search can be improved by more efficient algorithms, all of these algorithms take exponential time to solve the problem. Much of the theory about the clique problem is devoted to identifying special types of graph that admit more efficient algorithms, or to establishing the computational difficulty of the general problem in various models of computation. Along with its applications in social networks, the clique problem also has many applications in bioinformatics and computational chemistry.

7.2.1 Directed Graphs

In some situations we need to keep track of the direction between vertices, not unlike knowing which streets in a big city are one-way streets and which are not.

Definition 7.2.15 (Directed Graph) *A* directed graph *or* digraph *is an ordered triple* $\vec{G} = (V, E, \eta)$*, where*

- $V \neq \emptyset$.
- $V \cap E = \emptyset$.
- $\eta : E \to V \times V$ *is a map.*

The set V is the set of vertices, and the set E the directed *edges. If $\eta(e) = (u, v)$, then u is called the* source *of e and v is called the* target *of e. Often, v is called a* successor *of u and u is called a* predecessor *of v.*

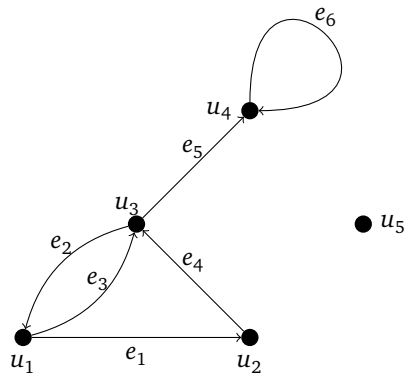
If $\eta(e) = (u, u)$, then e is called a directed loop*. If $\vec{G}$ is a digraph, then $V(\vec{G})$ and $E(\vec{G})$ will always denote the set of vertices and directed edges of $\vec{G}$, respectively.*

Two directed edges e and e' are said to be parallel edges *if $\eta(e) = \eta(e')$. That is, the edges are mapped into the same ordered pair of vertices.*

The following is an example digraph with five vertices and six directed edges. Here $\vec{G} = (V, E, \eta)$, where $V = \{u_1, u_2, u_3, u_4, u_5\}$, $E = \{e_1, e_2, e_3, e_4, e_5, e_6\}$, and the edgemap η is given by

$$\begin{aligned} \eta(e_1) &= (u_1, u_2), \\ \eta(e_2) &= (u_3, u_1), \\ \eta(e_3) &= (u_1, u_3), \\ \eta(e_4) &= (u_2, u_3), \\ \eta(e_5) &= (u_3, u_4), \\ \eta(e_6) &= (u_4, u_4). \end{aligned}$$

In particular, the directed edge e_6 is a directed loop.



7.3 Basic Concepts

The essence of a graph is whether two given vertices in it are connected or not. In many situations, though, we do not need a *direct* connection with an edge, but rather a route to be able to 'go' from one vertex to another in some number of steps. Although two vertices in a given graph may not be adjacent, they might be adjacent to a common neighbour, or more generally they might be connected by a sequence of edges. The following definition captures this idea and some varieties of it.

Definition 7.3.1

1. *A* walk *in a graph $G = (V, E, \phi)$ is an alternating sequence*

$$(u_0, e_1, u_1, e_2, \ldots, e_k, u_k)$$

of vertices and edges that begins and ends with a vertex. For each $i \in \{1, \ldots, k\}$, the endvertices of e_i are u_{i-1} and u_i. That is, $\phi(e_i) = \{u_{i-1}, u_i\}$.

The vertex u_0 is the initial vertex *of the walk. The vertex u_k is the* final vertex *of the walk. The number k is the* length *of the walk.*

2. *A* trail *in G is a walk with all of its edges $e_1, e_2, \ldots, e_k$ distinct.*

3. *A* path *in G is a walk with all of its nodes $u_0, e_1, \ldots, u_k$ distinct.*

4. *A walk or trail of length at least one is* closed *if its initial vertex and final vertex are the same. A closed trail is also called a* circuit.

5. *A* cycle *is a closed walk with distinct vertices except for the initial and final vertices, which are the same.*

Write *x, y-path* (walk etc) for a path (walk etc) from vertex x to vertex y.

The notion of a *composite* walk (trail or path) can be formed, by joining two walks (trails or paths) together. For paths p and q such that the final vertex of p is the initial vertex of q, the composite walk is denoted pq. The *inverse* walk p^{-1} (trail or path) is a path in the reverse direction.

The next result shows that if there is a walk between two vertices in a graph, then there must also be a path between them. Intuitively, the idea is simply to go directly between the two vertices without taking any 'detours'.

Theorem 7.3.2 *Let G be graph having distinct vertices u and v. If G contains a walk from u to v, then G contains a path from u to v.*

7.3.1 Connectivity

It is intuitively clear what we mean by a graph being connected, namely, that we can start at any vertex and walk to any other vertex along the edges of the graph.

Definition 7.3.3 (Connected) *A graph G is* connected *if for every pair of distinct vertices $u, v \in V(G)$, there is a path from u to v. Otherwise we say that the graph is* disconnected.

The connected subgraphs of a graph are called *components* or *connected components* of the graph.

Definition 7.3.4 (Connector components) *Let G be a graph. Let $H_1, \ldots, H_k$ be connected subgraphs of G whose vertex sets and edge sets are pairwise disjoint and such that they* cover *all the vertices and edges of G. That is,*

$$\begin{aligned} V(G) &= V(H_1) \cup \cdots \cup V(H_k), \\ E(G) &= E(H_1) \cup \cdots \cup E(H_k), \end{aligned}$$

where $V(H_i) \cap V(H_j) = \emptyset = E(H_i) \cap E(H_j)$, for each distinct i and j.

Each of the subgraphs H_i is called a component *or* connected component *of G.*

Theorem 7.3.5 *Every graph G has a unique collection of connected components $H_1, \ldots, H_k$. In particular, the number of connected components of G is uniquely determined by G.*

We delay the proof of this theorem, and first introduce some convenient mathematical tools.

The following result shows that an equivalence relation on a finite nonempty set S is really the same as a *partition* of that set; that is, we can write $S = S_1 \cup \cdots \cup S_k$ for some $k \in \mathbb{N}$, where

1. $S_i \neq \emptyset$ for $i \in \{1, \ldots, k\}$,
2. $S_i \cap S_j = \emptyset$ for $1 \leq i < j \leq k$.

Theorem 7.3.6 *Let S be a finite set and $\sim$ an equivalence relation. Then for some k we have a partition $S = S_1 \cup \cdots \cup S_k$ of S, which is uniquely determined by $\sim$. The partition satisfies*

$$S_i \quad = \quad \{t \in S \mid t \sim s_i\} \quad \textit{for each } s_i \in S_i.$$

Each of the subsets S_i is called the equivalence class corresponding to $\sim$.

Conversely, the equivalence relation $\sim$ on S yielding this partition is unique and can be retrieved by

$$s \sim t \quad \Longleftrightarrow \quad s, t \in S_i \textit{ for some } i \in \{1, \ldots, k\}.$$

Theorem 7.3.7 *For a graph G let $\sim$ be the* path relation *on $V(G)$ given by*

$$u \sim v \quad \Longleftrightarrow \quad G \textit{ has a path from } u \textit{ to } v.$$

Then $\sim$ is an equivalence relation on $V(G)$.

Proof. For any vertex u the path of length zero is in G, hence $u \sim u$.

Suppose $u \sim v$ and that p is a path from u to v in G. Then the inverse path p^{-1} is a path from v to u in G, hence $u \sim v$.

Finally, if p is a path from u to v in G and q is a path from v to w in G, then pq is a walk from u to w in G. By Theorem 7.3.2 there is a path from u to w in G, which completes the proof. □

We are now ready to present the proof of Theorem 7.3.5.

Proof. (Proof of Theorem 7.3.5.) By Theorem 7.3.7 using the path relation $\sim$, we get a partition of the vertices of G

$$V(G) = V_1 \cup \cdots \cup V_k.$$

where $V_i \cap V_j = \emptyset$ for $1 \leq i < j \leq k$. Note that every pair of vertices u and v in each V_i is connected by a path in G.

Let $H_i = G[V_i]$ be the subgraph of G induced by V_i. Let E_i be the edge set of H_i for $1 \leq i \leq k$. Since no two of the H_i's have a vertex in common, they have no edges in common either, implying that $E_i \cap E_j = \emptyset$ for $1 \leq i < j \leq k$.

Finally, if $e \in E$ is an edge of G, then the endvertices of e, say u and v, are connected by a path (u, e, v). So $u, v \in V_i$ for some i, and therefore $e \in E_i$. Thus we have $E(G) = E_1 \cup \cdots \cup E_k$, which completes the proof of the theorem. □

7.3.2 Digraph Connectivity

The concept of connectivity for digraphs requires a little more care. One approach is to simply adapt the definitions for graphs, saying that a digraph $\vec{G}$ is connected if and only if its underlying graph G is connected (that is, by forgetting the directions on the edges). In this case, we say that $\vec{G}$ is *weakly connected*.

There is, however, a more useful and common way of describing connectivity in digraphs. For that we need directed notions of walk and so forth.

Definition 7.3.8 *A* directed walk $\vec{w}$ *in a digraph* $\vec{G}$ *is an alternating sequence*

$$\vec{w} = (u_0, e_1, u_1, e_2, \ldots, e_k, u_k)$$

of vertices and directed edges, where for each $i \in \{1, \ldots, k\}$ *the source and target of* e_i *are* u_{i-1} *and* u_i*, respectively. That is,* $\eta(e_i) = (u_{i-1}, u_i)$*.*

The notions of initial *and* final vertex*,* length*,* directed trail*, and* directed path *etc. extend in a natural fashion.*

Note that as each directed edge has a unique source and target, there is no ambiguity in writing a directed walk as

$$\vec{w} = (e_1, e_2, \ldots, e_k),$$

where it is understood that the initial vertex of $\vec{w}$ is the tail of e_1 and the final vertex is the head of e_k.

Definition 7.3.9 (Strong Connectivity) *We say that a graph* $\vec{G}$ *is* strongly connected *if for every pair* $u, v \in V(\vec{G})$ *of distinct vertices, there is a directed walk from* u *to* v *in* $\vec{G}$*.*

The strong components *of* $\vec{G}$ *are the maximal strongly connected subdigraphs of* $\vec{G}$*.*

Later, we will look at algorithms for computing the strongly connected components of a graph.

7.4 Trees

Trees and *forests* are some of the most frequently occurring graphs. They are especially common when organising certain data structures and for developing search algorithms. Trees are important because they minimally connect the vertices that represent some form of data.

Definition 7.4.1 (Tree, Forest) *A* tree *is a connected graph that has no cycle as a subgraph. A* forest *is a graph in which every component is a tree.*

Definition 7.4.2 (Leaf) *A vertex* u *of a simple graph* G *is called a* leaf *if* $d_G(u) = 1$*. A vertex that is not a leaf is called an* internal vertex*.*

The following give some useful properties about trees.

Theorem 7.4.3 *Every tree with at least two vertices has at least two leaves.*

This theorem provides a useful tool for proving many properties about trees. In many cases, all that matters is that every tree of two or more vertices has at least *one* leaf. Let T be a tree with a leaf u. Notice that the subgraph $T' = T - u$ has no cycles, since T has none. Also, T' is connected, hence T' is a tree and it has one fewer vertices than T. Many proofs about T can be done inductively by assuming the argument is true for T'.

The following lemma is also useful.

Lemma 7.4.4 *A simple graph G on n vertices with k components has at least $n-k$ edges.*

The following theorem provides alternative characterisations of trees.

Theorem 7.4.5 *If T is a simple graph on n vertices, then the following statements are equivalent:*

1. *T is a tree.*
2. *T has $n-1$ edges and no cycles.*
3. *T has $n-1$ edges and is connected.*
4. *T is connected and each edge is a bridge (i.e., removal of the edge increases the connected components).*
5. *Between every pair of distinct vertices in T there is exactly one path.*
6. *T has no cycles, but when adding an edge to T between a pair of nonadjacent vertices, exactly one simple cycle is formed.*

Proof. We prove the theorem by showing the following cycle of implications:

$$1 \Rightarrow 2 \Rightarrow 3 \Rightarrow 4 \Rightarrow 5 \Rightarrow 6 \Rightarrow 1.$$

$1 \Rightarrow 2$. We argue by induction on the number of vertices n. We need to show that T has $n-1$ edges. The statement is clearly true if $n = 1$. Let T be a tree on $n \geq 2$ vertices. By Theorem 7.4.3 T has a leaf u. Hence, $T' = T - u$ is a tree on $n-1$ vertices. Therefore, by the induction hypothesis, T' has $n-2$ edges. Since u is a leaf, T' has one fewer edges than T. Hence T has $n-1$ vertices.

$2 \Rightarrow 3$. We need to show that T is connected. By definition we have that T is a forest. Let $k \geq 1$ be the number of components in T. Every component is a tree and therefore has one fewer edges than vertices, as we say in proving $1 \Rightarrow 2$. Hence the number of edges in T is $n-k$. Thus $n-k = n-1$, hence $k = 1$, so T is connected.

$3 \Rightarrow 4$. We show that each edge is a bridge. By Lemma 7.4.4 every connected graph *must* have at least $n-1$ edges. Therefore, the removal of any edge from T will disconnect it. Therefore each edge is a bridge.

$4 \Rightarrow 5$. We need to show that there is a unique path between each pair of vertices in T. By connectivity of T there is at least one path between any two vertices of T. In there are two vertices u and v having two distinct paths between them, then we can take p_1 of the shortest possible length, and argue that the other p_2 must contain an edge that is not contained in p_1.

Since p_1 is of the shortest possible length, the sequence of edges in p_2 cannot be a rearrangement of those in p_1, since if that were the case, we could use these common edges of p_1 and p_2 to form a common path from u to v which is strictly shorter than p_1, by starting at u and always traversing the edge which is closer to v on either p_1 or p_2. Therefore there must be an edge e in p_2 which is not contained in p_1. Since there is a walk in $T - e$ connecting the endvertices of e, formed by the edges of p_1 and the remaining edges of p_2, the edge e is not a bridge of T, which is a contradiction.

$5 \Rightarrow 6$. Clearly, T has no cycles, since that would imply two distinct paths between a pair of vertices. We must show that adding any edge to T adds exactly one simple cycle. Let u and v be two vertices and p a path from u to v in T. If we add an edge between u and v, then $c = p(v, e, u)$ is a simple cycle in T. If there is another such cycle c', then c' would have to contain e, and by removing e from c', one would get yet another path from u to v.

$6 \Rightarrow 1$. We must show that T is indeed connected. Let u and v be vertices in T. By adding an edge e between u and v, we obtain a single cycle. This cycle must contain the edge e. The removal of e yields a simple path from u to v. Hence, T is connected and therefore a tree. □

Depending on the situation, one of the characterisations given in this theorem may be preferable to the others.

7.5 Spanning Trees

As we saw, trees are minimally connected in the sense that each edge is a bridge; the removal of any one of them will disconnect the graph. They are the graphs that connect a given collection of vertices with the fewest possible edges, namely $n-1$ of them. If G is a connected graph with $V(G) = V$ as its set of vertices, an obvious question is whether we can find a subgraph of G that connects all the vertices of G with the fewest possible edges. Such a subtree must have at least $n-1$, by virtue of Lemma 7.4.4, otherwise it is not connected. But is it always possible to find such a subgraph of G with exactly $n-1$ edges that connects all the vertices of G? We shall see that this is always possible.

Suppose G has n vertices; then a connected subgraph containing $n-1$ edges an all n vertices of G is, by Theorem 7.4.5, necessarily a subtree of G.

Definition 7.5.1 *Let G be a graph.*

1. *A subtree T of G is called a* spanning tree *of G if $V(T) = V(G)$.*
2. *A subforest F of G is called a* spanning forest *of G if for each component H of G, the subgraph $F \cap H$ is a spanning tree of H.*

Note that if T is a spanning tree of G and G' is the graph obtained by removing all the loops from G, then T is a spanning tree of G'.

7.5.1 Minimal Cost Spanning Trees

So far spanning trees were described for graphs that did not have any costs associated with the edges. In this section, we take into account graphs with *weights* on the edges. We will describe how to find the *minimum cost spanning tree* on a graph G, which is simply the spanning tree with the least cost.

Definition 7.5.2 *Let G be a graph and $W : E(G) \to \mathbb{R}$ a function.*

1. *The ordered tuple (G, W) is called a* weighted graph. *The function W is called a* weight *or* weight function *of (G, W).*
2. *If G' is a subgraph of G, then*
$$W(G') = \sum_{e \in E(G')} W(e)$$
is called the weight *of G' in G.*

When there is no possibility of ambiguity, we talk about the weighted graph G instead of the tuple (G, W). In most applications we also assume that $W(e) \geq 0$ for all $e \in E(G)$.

For a weighted graph G, when we ask for a minimum cost subgraph $G' \subseteq G$ that covers all the vertices, $V(G') = V(G)$, we can assume that G' has no cycle, since the connectedness of G' is not altered by removing one edge from the cycle of G' when $V(e) \geq 0$. Repeating this process, removing one each from each cycle of G', we will end up with a connected subtree of G.

Definition 7.5.3 *For a connected weighted graph (G, W), as spanning tree T with the minimum weight $W(T)$ is called a* minimum cost spanning tree.

Example 7.5.4 *Suppose we have a collection of n cities $c_1, \ldots, c_n$ in a certain state, where the cost of building roads between the cities varies. Assume we have the task of building a road system of the lowest possible cost, such that between any two cities c_i and c_j, there is always a road (not necessarily a direct road) from c_i to c_j. This road might go through some other cities on the way. Denote the cost, in millions of euros, of building the road between c_i and c_j by w_{ij}. Evidently for some $i \neq j$ the cost w_{ij} could be extremely high, due perhaps to high mountains or wet swamp land between the cities. In some cases, due to governmental regulations regarding national parks and whatnot, it might in fact be impossible to build a direct road between c_i and c_j, in which case we set $w_{ij} = \infty$.*

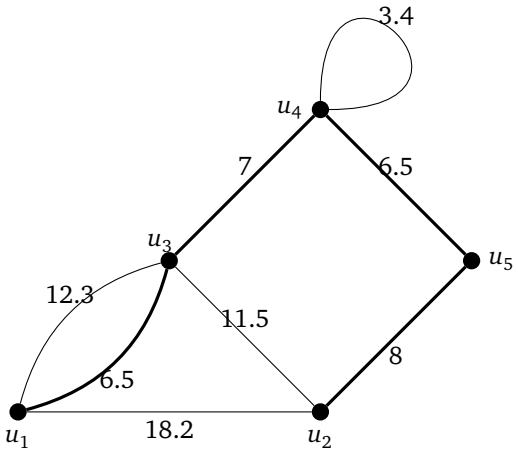
Here, the cheapest road system would be given by a minimum cost spanning tree for the weighted complete graph (K_n, W), where the vertices and the weight function W are given by

$$\begin{aligned} V(K_n) &= \{c_1, \ldots, c_n\}, \\ W(\{c_i, c_j\}) &= w_{ij}. \end{aligned}$$

Th following algorithm, called *Kruskal's Algorithm*, computes a minimum cost spanning tree in any connected weighted graph.

Kruskal's Algorithm
INPUT: A connected weighted graph (G, W) on n vertices.
OUTPUT: A minimum cost spanning tree T on G.
begin
  $T_1 = \emptyset$.
  for $i = 1$ **to** $n - 1$ **do** {
    let $e_i \in E(G) \setminus E(T_i)$ be a minimum weight edge such that $T_i \cup \{e_i\}$ is a forest;
    $T_{i+1} = T_i \cup \{e_i\}$; // that is, e_i along with its other endpoint added
  }
  output $T = T_n$.
end

Example 7.5.5 *Consider the following weighted graph G with five vertices.*



Note that some pairs of edges are not connected, in which case we let the weight of the corresponding 'imaginary' edges be infinity ∞. The weights are given in the symmetric 5×5 table:

	u_1	u_2	u_3	u_4	u_5
u_1	$-$	18.2	6.5	∞	∞
u_2	18.2	$-$	11.5	∞	8
u_3	6.5	11.5	$-$	7	∞
u_4	∞	∞	7	$-$	6.5
u_5	∞	8	∞	6.5	$-$

Here, the (i,j)-th entry denotes the weight $W(\{u_i,u_j\})$. Note that in the case of multiple edges between a pair of vertices, we list only one edge with the minimum cost among all edges between these vertices. Also, the weight of a loop contributes nothing and is therefore not listed.

Kruskal's algorithm begins by selected a minimum cost edge to include in the forest it is constructing. We start with the empty tree $T_1 = \emptyset$.

In what follows we represent a tree by a set of edges. The algorithm first picks a minimum cost edge, either $\{u_1,u_3\}$ or $\{u_4,u_5\}$, say the first one, so $T_2 = \{\{u_1,u_3\}\}$.

The algorithm then picks a minimum cost edge from the remaining edges that do not form a cycle when added to T_2. In this case, the edge is $\{u_4,u_5\}$, so $T_3 = \{\{u_1,u_3\},\{u_4,u_5\}\}$.

Similarly, edge $\{u_3,u_4\}$ is selected in the next round to give $T_4 = \{\{u_1,u_3\},\{u_4,u_5\},\{u_3,u_4\}\}$.

In the final round, $\{u_2,u_5\}$ is selected, thus $T_5 = \{\{u_1,u_3\},\{u_4,u_5\},\{u_3,u_4\},\{u_2,u_5\}\}$. This is the graph marked in bold above.

Note that in each step of Kruskal's algorithm we have a forest, but not necessarily a connected tree. For a large weighted graph G, Kruskal's algorithm can have T_i consisting of many little scattered trees from G at each phase, but after the final step we can rest assured that we end up with a *connected* forest, namely the minimum cost spanning tree of G.

The cost here of the minimum cost spanning tree T is given by $W(T) = 6.5 + 6.5 + 7 + 8 = 28$.

The following theorem shows that Kruskal's algorithm always computes a minimum cost spanning tree.

Theorem 7.5.6 *Let (G, W) be a connected weighted graph. Kruskal's Algorithm computes a minimum cost spanning tree T of G.*

Proof. It is clear that Kruskal's algorithm builds a spanning tree T of the connected graph G on n vertices, because it constructs a simple graph having $n-1$ edges. We need to show that if T_1 is another spanning tree of G, then $W(T) \leq W(T_1)$.

Assume we have labelling $e_1, \ldots, e_{n-1}$ (respectively, $e'_1, \ldots, e'_{n-1}$) of the edges of T (respectively, T_1) such that $W(e_1) \leq \cdots \leq W(e_{n-1})$ (respectively, $W(e'_1) \leq \cdots \leq W(e'_{n-1})$). Since T_1 and T are not equal, there is a least number k such that $e_k \notin \{e'_1, \ldots, e'_{n-1}\}$ but $e_{k-1} \in \{e'_1, \ldots, e'_{n-1}\}$.

Notice that $T_1 \cup e_k$ has a unique cycle C that is not completely contained in T. Hence, there is an edge e' in C that is not in T. In this case we have that

$$T_2 = (T_1 \cup e_k) \setminus e'$$

is also a spanning tree of G. In particular, by the minimality of k, the edges $e_1, \ldots, e_{k-1}$ and e' are all in T, and hence e' does not form a cycle together with the edges $e_1, \ldots, e_{k-1}$. Hence, by the algorithm's choice of e_k, we have $W(e_k) \leq W(e')$ and hence $W(T_2) \leq W(T_1)$.

Note now that T_2 has one more edge in common with T than T_1 does. If T_2 and T are not equal, we continue in the same fashion and obtain a spanning tree T_3 with $W(T_3) \leq W(T_2)$ and where T_3 has one more edge in common with T than T_2 does.

In this way we obtain a sequence of trees $T_1, \ldots, T_r$ with $W(T_r) \leq \cdots \leq W(T_1)$ and where $T_r = T$. In particular, we have $W(T) \leq W(T_1)$ and we have the theorem. □

The next algorithm called *Prim's Algorithm*, also yields a minimal cost spanning tree in any connected weighted graph, but it constructs the the tree in a different way.

Prim's Algorithm
INPUT: A connected weighted graph (G, W) on n vertices.
OUTPUT: A minimum cost spanning tree T on G.
begin
  $T_1 = (\{u_1\}, \emptyset)$. // u_i arbitrary initial vertex
  for $i = 1$ **to** n **do** {
    let $e_i \in E(G) \setminus E(T_i)$ be a minimum weight edge such that $|V(T_i) \cap e_i| = 1$;
    $T_{i+1} = T_i \cup \{e_i\}$; // e_i with its other endpoint added
  }
  output $T = T_n$.
end

We now informally argue the correctness of Prim's Algorithm as follows:

At each phase of the algorithm, the edges obtained thus far form a spanning tree of the subgraph of G induced by these edges. This subtree is a minimum cost spanning tree in the corresponding weighted subgraph of G. Therefore, *Prim's algorithm* computes a minimum cost spanning tree of G.

Exercise 7.5.7 *Apply Prim's algorithm to the example graph above.*

Exercise 7.5.8 *Find a graph for which Prim's algorithm and Kruskal's algorithms result in different trees.*

Both Kruskal's and Prim's algorithms are examples of *Greedy algorithms*. Such algorithms select the cheapest next edge each step. Both algorithms are reasonably efficient in terms of n, the number of vertices in G.

7.6 Fundamental Properties of Graphs and Digraphs

The main goal of this section is to study cycles and circuits in graphs, or the complete lack thereof. We start by demonstrating how cycles are used for describing the structure of bipartite graphs.

7.6.1 Bipartite Graphs

Definition 7.6.1 *A graph G is a* bipartite graph *if there are $X, Y \subseteq V(G)$ meeting the following conditions:*

1. $V(G) = X \cup Y$,
2. $X \cap Y = \emptyset$,
3. $G[X]$ *and* $G[Y]$ *are both null graphs.*

(Recall that $G[X]$ is the graph G restricted to just vertices in X, and a null graph has no edges.)

The word "bipartite" means "two parts." This is the way to remember the definition.

Bipartite graphs are often used to model some common real-world situations, where the vertices in one part X represent agents and the vertices in the other part Y represent projects which are assigned to the agents by means of edges between X and Y.

Several points are worth noting:

- A bipartite graph G with partition $V(G) = X \cup Y$ implies that each edge of G has one endvertex in X and the other in Y.
- Every null graph N_n is bipartite; any partition $V(N_n) = X \cup Y$ demonstrates this fact.
- A bipartite graph cannot have any loops.
- A graph is bipartite if and only if each of its components is bipartite.
- For all $m, n \in \mathbb{N}$, the complete bipartite graph $K_{m,n}$ is a simple bipartite graph. (Simple = no loops and no multiple edges.)

It is not hard to see that every cycle in a bipartite graph must have an even length. This property is in fact a characterising property of bipartite graphs.

Theorem 7.6.2 *For a graph G the following statements are equivalent:*

1. *G is bipartite.*
2. *Every cycle in G has an even length.*

Moreover, if G is connected and satisfies either 1 *or* 2*, then the partition $V(G) = X \cup Y$ is unique.*

Before proving Theorem 7.6.2, we need the following observation.

Lemma 7.6.3 *If G contains a closed walk of odd length, then it contains a cycle of odd length.*

Proof. Assume that $p = (u_0, e_1, u_1, \ldots, e_n, u_n)$ is a closed walk of shortest possible length n. We show that p is actually a cycle. If p is not a cycle, then there are two indices j and k with $0 \leq j < k \leq n$, $(j,k) \neq (0,n)$, and $u_j = u_k$. In this case we have two closed walks $p_1 = (u_0, e_1, u_1, \ldots, u_j, e_k, u_{k+1}, \ldots, u_n)$ and $p_1 = (u_j, e_j, \ldots, e_{k-1}, u_{k-1})$. The lengths of p_1 and p_2 are both positive and sum up to n. Hence, one of these lengths must be odd. Thus we have a closed walk of odd length less than p, which contradicts our initial assumption. Therefore, p must be a cycle. □

Note that there are graphs containing closed walks or circuits of even lengths that do not contain any cycles of even length.

We now use Lemma 7.6.3 to prove Theorem 7.6.2.

Proof. (Theorem 7.6.2) Without loss of generality, we can assume that G is connected.

Assume first that $V(G) = X \cup Y$ is a partition of the vertices such that both $G[X]$ and $G[Y]$ are null graphs. Let $p = (u_0, e_1, u_1, \ldots, e_n, u_n)$ be a cycle. Since each edge in G has one endvertex in X and the other in Y, we see that for each i the vertices u_i and u_{i+1} of the cycle p are always in opposite parts, that is, one is in X and the other is in Y. Since p is a closed walk, $u_0 = u_n$ must be in the same part X or Y, so n must be even.

For the second half of the theorem, assume that each cycle of G has even length. We show that G is bipartite. Chose a fixed vertex $v \in V(G)$. Consider an arbitrary vertex $y \in V(G)$. If p and q are distinct paths from x to y of respective lengths n and m, then pq^{-1} is a closed walk in G of length $n+m$. Combining our assumption that each cycle of G has even length with the contrapositive of Lemma 7.6.3, we can conclude that $n+m$ is even. Hence, either n and m are both even or they are both odd. Therefore we can form a partition of $V(G)$ by letting

$$\begin{aligned} X &= \{y \in V(G) \mid \text{the length of each path from } x \text{ to } y \text{ is even}\}, \\ Y &= \{y \in V(G) \mid \text{the length of each path from } x \text{ to } y \text{ is odd}\}. \end{aligned}$$

When G is connected, we have $V(G) = X \cup Y$ and $X \cap Y = \emptyset$. Lastly, consider an edge of e of G with endvertices u and v. We need to show that they are not contained in the same part. Suppose p is a x,u-path of length l and q is an x,v-path of length l'. In this case, $(u,e,v)q^{-1}p$ is a closed walk of length $l+l'+1$, which by Lemma 7.6.3 must be even. Hence l and l' must be of opposite parity modulo 2. Hence, one of u and v is contained in X, and the other in Y. Hence, $G[X]$ and $G[Y]$ are both null graphs.

Since G is connected, every two vertices are joined by a simple path. Hence, the partition $V(G) = X \cup Y$ is uniquely determined. □

We can easily extend the definition of a bipartite graph: A graph G for which $V(G)$ can be partitioned into $k \geq 2$ sets

$$V(G) = X_1 \cup \cdots \cup X_k,$$

where $X_i \cap X_j = \emptyset$ for each $i \neq j$ and each of the induced subgraphs $G[X_i]$ is a null graph, is called a *k-partite graph*. If G is k-partite for some k, then G is called a *multipartite graph*. However, the nice characterisation for bipartite graphs given in Theorem 7.6.2 cannot be extended for $k \geq 3$.

7.6.2 Eulerian Graphs

In this section we are interested in the following 'spanning' property: Is there a circuit or trail in a graph that spans or covers all of the edges in the graph? Note that unlike for spanning trees, we cannot

answer this question positively for every connected graph or digraph.

This question is related to the Königsberg Bridge problem (stated above) and was first solved by Euler in 1736, in the very first paper on graph theory. The more general question Euler solved was: what type of a connected graph has a circuit that contains all of the edges of the graph?

Definition 7.6.4 *Let G be a graph. A trail of G that contains each edge of G is called an* Eulerian trail *of G. A circuit of G that contains each edge of G is called an* Eulerian circuit *of G. If G has an Eulerian circuit, then G is called an* Eulerian graph*.*

Note that an Eulerian graph, with no isolated vertices, is necessarily a connected graph, since a circuit going through each edge of G must connect all the non-isolated vertices of G.

The following result was from the first paper on graph theory by Euler from 1736.

Theorem 7.6.5 *A connected graph G is Eulerian if and only if each vertex in G has even degree.*

Proof. We start with the easier part first and assume that G has an Eulerian circuit. Starting at a given vertex in c and traversing along c, we leave each vertex $u \in V(G)$ for each time we enter it. Since we enter and leave e each time along different edges, the number of edges in c that have u as an endvertex must be even.

Assume now that G is connected where each vertex has an even positive degree. We show by induction on $|E(G)|$ that G has an Eulerian circuit. We start at an arbitrary vertex $u_0 \in V(G)$.

1. Assume that we have constructed a simple path $p_i = (u_0, e_1, u_1, \ldots, e_i, u_i)$ in G of length $i \geq 0$. Since $d_G(u_i) > 0$ is even, there is an edge $e_{i+1} \notin \{e_1, \ldots, e_i\}$ with u_i as one endvertex and u_{i+1} as the other endvertex.
   - If $u_{i+1} \notin \{u_0, \ldots, u_i\}$, then $p_{i+1} = p_i(u_i, e_{i+1}, u_{i+1})$ is a simple path of length $i+1$. Change i to $i+1$, and then go to 1.
   - If $u_{i+1} = u_j \in \{u_0, \ldots, u_i\}$, then $c' = (u_j, e_{j+1}, \ldots, e_{i+1}, u_{i+1})$ is a simple cycle of positive length, so let $E_{c'} = \{e_{j+1}, \ldots, e_{i+1}\}$, and go to 2.
2. Let $G' = G - E_{c'}$.

Note that G' has strictly fewer edges than G and each vertex in G' has an even degree. If the components of G' are $H_1, \ldots, H_k$, then by the induction hypothesis, each H_i has an Eulerian circuit c_i that starts and ends at a vertex v_i on the cycle c'. Since we can write $c' = p_1 p_2 \cdots p_{k-1} p_k$, where p_i is a trail on c' from v_i to v_{i+1} if $1 \leq i \leq k-1$, and p_k is a trail from v_k to v_1, then c given by

$$c = c_1 p_1 c_2 p_2 \ldots p_{k-1} c_k p_k$$

is an Eulerian circuit of G, which completes the proof. □

Clearly the vertices in the graph shown in Figure 7.8 do not all have even degree (in fact, they all have *odd* degree!!). Hence, the graph does not have an Eulerian circuit; and therefore the original question of the Königsberg Bridge problem, whether one can start on one bank, and walk over every bridge exactly once and end up in the same spot, is by Theorem 7.6.5 answered in the negative.

There are several corollaries we can deduce from Theorem 7.6.5.

Corollary 7.6.6 *A connected graph G has an Eulerian trail if and only if all except two vertices in G have an even degree.*

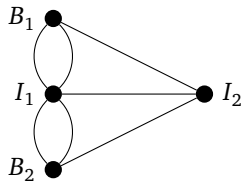


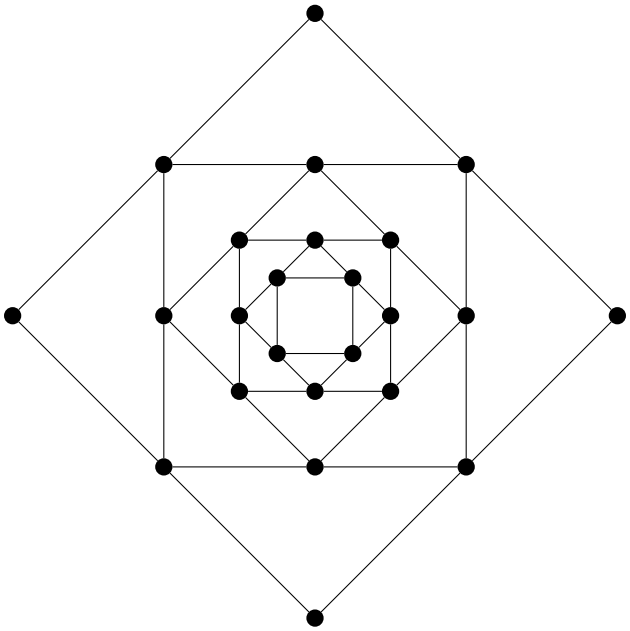
Figure 7.8: The graph of the bridges of Königsberg problem

Corollary 7.6.7 *For a graph G the following statements are equivalent:*

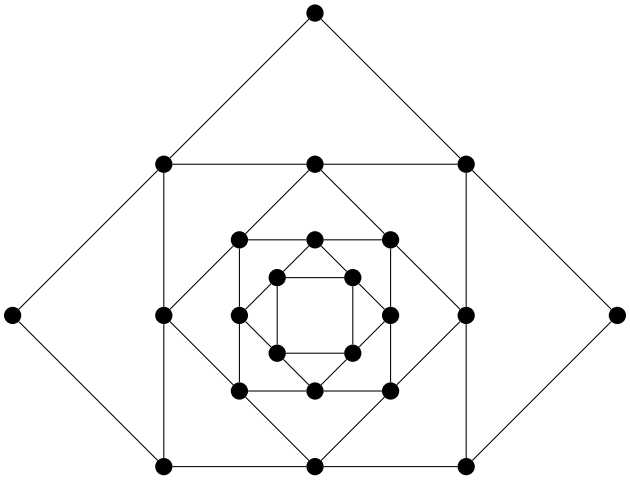
1. *All of the vertices have a positive even degree.*
2. *G is a disjoint union of edge-disjoint cycles in G.*

In particular, a connected graph G is Eulerian if and only if it is the disjoint union of cycles in G.

In many drawing games from puzzle magazines, we find that Theorem 7.6.5 and Corollary 7.6.6 can save us a sleepless night or two. Consider the following two graphs. The first has an Eulerian circuit because all vertices have an even degree. One can in fact inductively construct the Eulerian circuit by removing one cycle at a time from the graph—though a better method exists (find it!).



In the second graph, there are two vertices that do not have even degree. By Theorem 7.6.5 it is impossible to draw this graph without lifting one's pencil, no matter how hard one tries. However since there are only two vertices of odd degree, the graph has by Corollary 7.6.6 an Euler trail. This it is possible to draw with graph without lifting one's pencil, if one can start at one point and end at another.

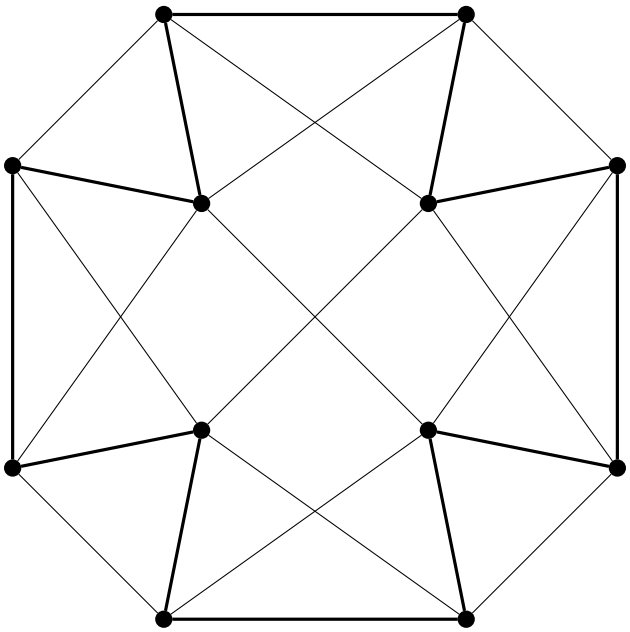


7.6.3 Hamiltonian Graphs

In this section we ask if there is a 'round trip' through a given graph G such that every *vertex* is visited exactly once. As we shall see, this is a much harder question than the ones for edges just addressed.

Definition 7.6.8 *Let G be a graph. A path in G that includes every vertex of G is called a* Hamiltonian path *of G. A cycle that includes every vertex in G is called a* Hamiltonian cycle *of G. If G contains a Hamiltonian cycle, then G is called a* Hamiltonian graph*.*

The following graph is a Hamiltonian graph, since it contains a Hamiltonian cycle shown by thick edges.



Note that every Hamiltonian cycle of a Hamiltonian graph G on n vertices has exactly n vertices and n edges. Hence, most likely, not all edges of G are included in a Hamiltonian cycle of G. This is fundamentally different from an Eulerian graph, where the Eulerian circuit covers all of the edges and therefore all of the vertices as well! From this point alone, one can see that much less can be said about the structure of a graph when we know that it is Hamiltonian than when we know it is Eulerian.

We make the following observations:

- Every Hamiltonian graph must be connected.
- No tree is Hamiltonian.
- For each $n \geq 3$, the cycle graph C_n is Hamiltonian.
- For each $n \geq 3$, the complete graph K_n is Hamiltonian.
- For each $n \geq 2$, the complete bipartite graph $K_{n,n}$ is Hamiltonian.

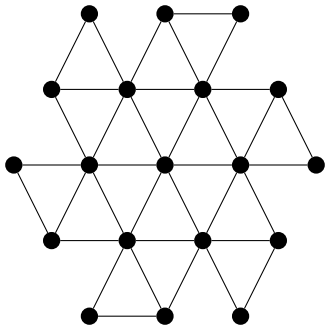
As for Eulerian graphs, it is natural to ask: what are the necessary and sufficient conditions a graph G must satisfy in order to be Hamiltonian? As we just saw, G must be connected, but that is not sufficient, since trees are connected by clearly not Hamiltonian. In addition, a graph G is Hamiltonian if and only the graph G', where all loops and multiple edges of G have been removed, is Hamiltonian. Hence when determining whether or not a graph G is Hamiltonian, we can assume G is simple and connected.

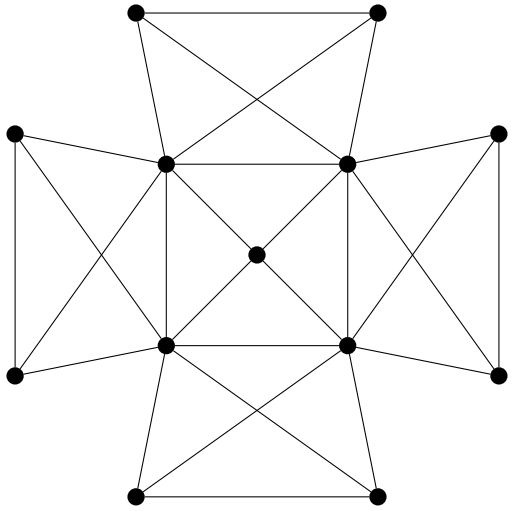
The following theorem shows us a condition that is necessary for a graph to be Hamiltonian. We will discuss some sufficient conditions later.

Theorem 7.6.9 *If G is a simple Hamiltonian graph, then for each $S \subseteq V(G)$, the number of components of $G - S$ is at most $|S|$.*

Proof. Let C be a Hamiltonian cycle of G, viewed as a subgraph of G. Because C is a subgraph of G that includes all of the vertices of G, the number of components of $G - S$ is at most the number of components of $C - S$. Since C is a cycle, the number of components of $C - S$ is at most $|S|$, which completes the proof. □

Theorem 7.6.9 can be used to prove that neither of the following two graphs is Hamiltonian.





Theorem 7.6.9 provides an easy necessary condition for a graph to be Hamiltonian—easy in the sense that if we can find a subset $S \subseteq V(G)$ with more than $|S|$ components of $G - S$, then G is not Hamiltonian. It can be hard, however, for large graphs, to find such a subset S, even if it exists.

Example 7.6.10 *Let the complete bipartite graph $K_{2,3}$ be presented on the vertices*

$$V(K_{2,3}) = \{u_1, u_2, u_3\} \cup \{v_1, v_2\}.$$

where each u_i is connected to each v_j. If we let $S = \{v_1, v_2\}$, then $|S| = 2$ but $G - S$ is a graph consisting of three isolated vertices u_1, u_2 and u_3, and hence $G - S$ has three components, one more than the elements of S. By Theorem 7.6.9 the graph is not Hamiltonian.

Example 7.6.11 *For $n \in \mathbb{N}$, let K_{n+1} be the complete graph with vertices $\{u, u_1, \ldots, u_n\}$. For each $i \in \{1, \ldots, n\}$ connect an additional vertex v_i to u and u_i. Let G be the resulting graph on $2n + 1$ vertices $\{u, u_1, \ldots, u_n, v_1, \ldots, v_n\}$. Note that each vertex v_i has degree 2 in G. Since a potential Hamiltonian cycle c of G must contain each vertex v_i, we see that c must contain edges $\{u_i, v_i\}$ and $\{v_i, u\}$ for each i. So c cannot be a cycle and G is not Hamiltonian.*

On the other hand, one can see that for each $S \subseteq V(G)$ the number of components of $G - S$ is at most $|S|$.

Therefore, there are infinitely many graphs satisfying the necessary conditions of Theorem 7.6.9 that are not Hamiltonian.

7.6.4 The Travelling Salesman Problem

In this section we briefly discuss Hamiltonian cycles in weighted graphs (G, W) on n vertices. Recall that if C is a cycle in G, then

$$W(C) = \sum_{e \in E(C)} W(e)$$

is the weight of the cycle C in G. When seeking the minimum cost Hamiltonian cycle in G, we can assume G is a complete graph on n vertices—that is, $G = K_n$—simply by putting $W(\{u, v\}) = \infty$ whenever

u and v are not adjacent in G. In this way, the weight of the minimum cost Hamiltonian cycle of G is infinite if and only if G has no Hamiltonian cycle.

The algorithmic problem of finding a minimum cost Hamiltonian cycle in a weighted graph G is a well-known and very hard problem, which usually goes by the name of the *travelling salesman problem*.

> A travelling salesman wants to make a round trip through n cities, $c_1, \ldots, c_n$ to sell his goods. He starts in his home city c_1, visits each remaining city c_i exactly once, and ends in his home city c_1, where he started the trip. If he knows the distances between each pair of cities c_i and c_j, how should he plan his round trip to make the total round-trip distance as short as possible?

Here we can represent the cities as vertices in the complete graph K_n and let the weight between two distinct cities be the distance in kilometres:

$$\begin{aligned} V(K_n) &= \{c_1, \ldots, c_n\}, \\ W(\{c_i, c_j\}) &= \text{the distance between } c_i \text{ and } c_j \text{ in kilometres.} \end{aligned}$$

In this way, the problem of finding the shortest route is precisely that of finding a minimum weight Hamiltonian cycle of the weighted complete graph K_n.

In theory, we could just check all of the possible round trips and pick the one that has the minimum weight among them. This is, however, not computationally feasible, since the total number of rounds trips for n cities is given by $(n-1)!/2$, which quickly becomes too large for exhaustively checking as n grows.

To this day, a large amount of research focuses on developing efficient algorithms to find the minimum cost round trip to solve the travelling salesman problem. Approximations of the minimum cost round trip have been studied extensively—that is, how to find a round trip that we know for sure is at most, say, 30% longer than the optimal minimum round trip.

TSP is in NP.

7.7 Graph Traversal

We now present two algorithms for traversing a graph. These form the basis for other graph algorithms.

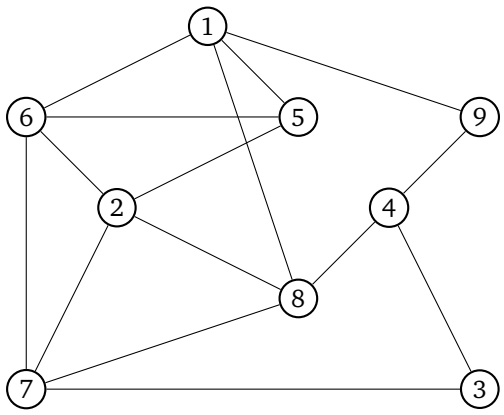
7.7.1 Breadth First Search

This algorithm is useful for exploring all the vertices of a graph. The idea is to start at a designated source vertex and then explore all neighbours of the original vertex. Once all of these have been examined, their neighbours are explored. This process is repeated until all vertices of the entire graph have been visited. In this way a spanning forest of the input graph can be constructed. The output of the algorithm for this variant of breadth-first search is a breadth-first search tree. For simplicity, we assume that the input graph is connected.

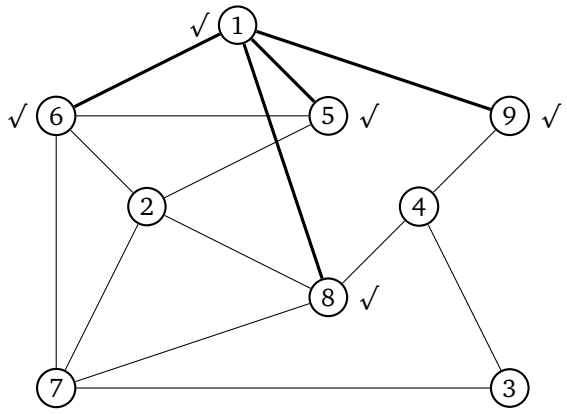
The algorithm is based on a *queue* data structure that is useful for expressing the algorithm. In a queue elements are removed from the front and added to the rear. Three operations are performed on the queue: ENQUEUE, DEQUEUE and EMPTY, which correspond, respectively, to adding an item to the tail of the queue, removing an item from the head of the queue, and checking to see if there is anything in the queue.

Breadth-First Search Algorithm
INPUT: A simple connected graph $G = (V, E)$ on n vertices and a designated source vertex s.
OUTPUT: A breadth-first search tree T on G.
begin
  $T_1 = (\{s\}, \emptyset)$;
  $i = 1$;
  mark s;
  initialise Q to the empty queue;
  ENQUEUE(Q, s);
  while not EMPTY(Q) **do** {
    $x =$ DEQUEUE(Q);
    for all edges $\{x, y\}$ where y is not marked **do** {
      mark y;
      $i = i + 1$;
      $T_i = T_{i-1} \cup \{x, y\}$;
      ENQUEUE(Q, y);
    }
  }
  output T_n;
end

Before analysing the running time of the algorithm, we present an example. We will use the following graph to illustrate the algorithm, as well as the depth-first algorithm which follows in the next subsection.



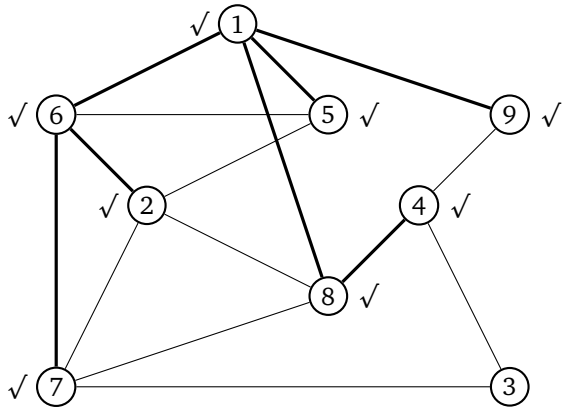
For illustrating the breadth-first search algorithm, the source vertex s is vertex 1. The following figure illustrates the search as it progresses from vertex 1. In the graph, vertices 6, 8, 5 and 9 were explored in this order (arbitrarily chosen as are other orders throughout this example), and the edges between these vertices and vertex 1 were added to the breadth-first search tree being constructed. Edges added to the tree are show in thick lines. Furthermore, vertices 6, 8, 5, and 9 have been queued up, in this order. These vertices plus vertex 1 have been marked (denoted by a $\surd$).



Queue: (head) 6 8 5 9 (tail)

From this point vertex 6 is dequeued and its neighbours 1, 5, 2 and 7 are explored. Since vertices 1 and 5 are already marked, no new edges are added to the tree based on them. On the other hand, vertices 2 and 7 are unmarked. The edges $\{2,6\}$ and $\{6,7\}$ are therefore added to the breadth-first search tree. Vertex 2 and next vertex 7 are marked and added to the queue.

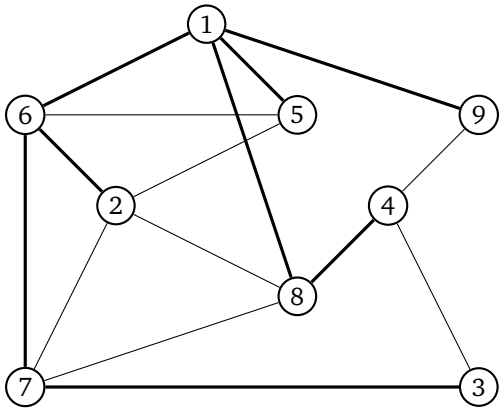
The search from vertex 8 results in the edge $\{4,8\}$ being added to the breadth-first search tree and the vertex 4 being marked and queued. Since all of the edges from vertex 5 lead to marked vertices, no new edges are added during its exploration. The following figure shows the snapshot just before vertex 9 is dequeued.



Queue: (head) 9 2 4 7 (tail)

Vertices 9 and 2 are then dequeued, and since all of their neighbours are marked, no new edges are added during their exploration. When vertex 7 is explored, we add edge $\{3,7\}$ to the tree, and mark and queue vertex 3. No additional edges are added to the tree while the queue is being emptied.

The resulting breadth-first search tree is show below.



We now turn our discussion to determining the worst-case running time of the Breadth First Search Algorithm. Consider the algorithm on input $G = (V, E)$ and a vertex s, where $|V| = n$ and $|E| = m$. Since the graph is connected, it has at least $n - 1$ edges, and so $n = O(m)$. Suppose $V = \{u_1, \ldots, u_n\}$. The algorithm begins with several straightforward initialisations. These require constant time $O(1)$. The time-consuming parts of the algorithm are the **while** loop and the **for** loops contained inside it. Note that every vertex will at some point be queued, and each operation within the **for** loop can be done in constant time. When a vertex is dequeued, all edges emanating from it are explored. Thus, each edge is explored twice, once from each end vertex. So, the worst-case running time of the algorithm is given by

$$t(n) = O\left(\sum_{i=1}^{n} d_G(u_i)\right).$$

Therefore, $t(n) = O(m)$, by the Hand-shaking theorem (Theorem 7.2.11). It is intuitively clear that a breadth-first search of a graph must explore all edges of the graph. Thus, up to constant factors, our Breadth-First Search Algorithm is optimal—that is, it runs as efficiently as possible.

7.7.2 Depth First Search

Next we describe the Depth-First Search Algorithm. In contrast to the Breadth-First Search Algorithm, which explored all the neighbours of a given vertex at a time, the Depth-First Search Algorithm explores only one neighbour at a time, and then tries to go "deeper" into the graph. The idea is to start at a given vertex and then explore a path in the graph as far as possible. When it is no longer possible to go "forward", the algorithm backtracks one level and then tries again to go deeper. This process repeats until all vertices of the entire graph have been visited. In this way a spanning forest of the input graph can be constructed. If the original graph is connected, the exploration forms a structure called a *depth-first search tree*. Note that the algorithm is recursive.

Depth-First Search Algorithm
INPUT: A simple connected graph $G = (V, E)$ on n vertices and a designated source vertex s.
OUTPUT: A depth-first search tree T on G.

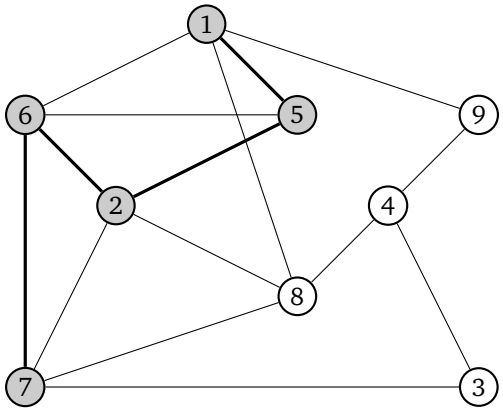
```
procedure DFSSEARCH(H)
begin
  for all v ∈ V do
    colour[v]=white;
  for all v ∈ V do
    if colour[v]=white then DFS(v);
end

procedure DFS(H)
// vertex v has not yet been visited
begin
  colour[v]=grey;
  for all w ∈ V such that {v,w} ∈ E(G) do
    if colour[w]=white then {
      add edge {v,w} to T;
      DFS(w);
    }
end

main
begin
  DFSSEARCH(G);
end
```

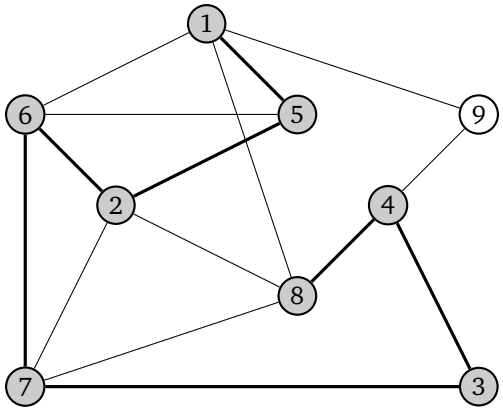
We use the graph above to illustrate the Depth-First Search Algorithm, again starting from vertex 1. When a choice exists, by convention we assume here that lower-numbered vertices get visited first.

The initial call to DFSSEARCH(G) results in all vertices being coloured white. This marking signifies that none of the vertices have yet been visited by the search. Suppose vertex 1 is selected initially. This choice results in the call DFS(1). Vertex 1 is coloured grey. Vertex 1's lowest-numbered neighbour, vertex 5, is then selected. The edge $\{1,5\}$ is added to the depth-first search tree, and the call DFS(5) is made. Vertex 5 is coloured grey. Vertex 5's lowest-numbered neighbour is vertex 1, which has already been coloured grey. Vertex 5's next lowest-numbered neighbour is vertex 2. The edge $\{2,5\}$ is added to the depth-first search tree, and the call DFS(2) is made. Vertex 2 is then coloured grey. Vertex 2's lowest-numbered neighbour is vertex 5. As vertex 5 has already been coloured grey, the algorithm explore the next lowest-numbered neighbour, namely vertex 6. The edge $\{2,6\}$ is added to the depth-first search tree, and the call DFS(6) is made. Vertex 6 is coloured grey. Vertex 6's two lowest-numbered neighbours, vertices 1 and 2, are already coloured grey, so vertex 7 is explored. The edge $\{6,7\}$ is added to the depth-first search tree, and the call DFS(7) is made. Vertex 7 is coloured grey. The following figure depicts the search up to this point.



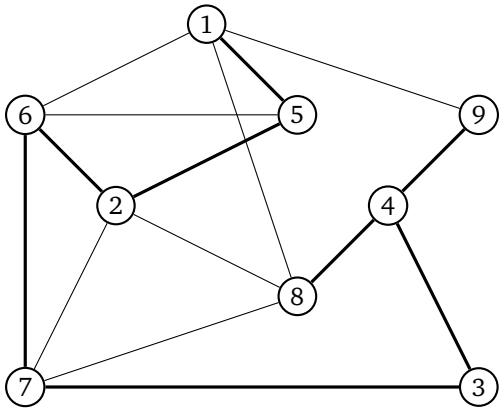
Note that in the figure for depth first search we have shown in thick lines the edges "along which" new vertices are visited. These edges comprise the depth-first search tree. For example, vertex 7 was explored as a neighbour of vertex 6, not from vertex 2.

The search continues from vertex 7, resulting in vertices 3, 4, and 8 being visited and coloured grey. The following illustrates this progression of the search, and the additional edges that are added to the depth-first search tree.



From vertex 8 the search attempts to go deeper by trying vertices 1, 2 and 4. However, they are all already coloured grey. Now the search backtracks to vertex 4. Earlier the search had tried unsuccessfully to visit vertices 3 and 5 from vertex 4, and actually did successfully visit vertex 8. So now the search proceeds to vertex 9. The edge $\{4, 9\}$ is added to the depth-first search tree. Vertex 9 is coloured grey. At this point all vertices have been coloured grey, although the search does not "know" this yet. So, the search continues trying to explore and backtracks, eventually backtracking all the way to vertex 1, where vertices 6, 8 and 9 are tried. At this point the search discovers that it is finished, since there are no more vertices to try. If there were another connected component, the search would jump into it by choosing an arbitrary vertex to explore. Thus, the algorithm actually works for general graphs and constructs a *depth-first search forest*.

The resulting depth-first search tree for the complete search is shown below.



We now discuss determining the worst-case running time of the Depth-First Search Algorithm. Consider the algorithm on input $G = (V, E)$, where $|V| = n$ and $|E| = m$. Since each vertex is visited exactly once, the procedure DFS$(\cdot)$ is called n times. Assuming that a data structure such as an adjacency list is used to represent the graph, successive neighbours on the adjacency list can be accessed in one step. Since from each vertex all of its neighbours are checked, the total number of checks is simply twice the number of edges of G, or the value $2m$. Thus, the algorithm's worst-case writing time is $O(\max(n, m))$. Note that if G is connected, then G has at least $n - 1$ edges, and hence the running time is $O(\max(n, m)) = O(m)$, as was the case for the Breadth-First Search Algorithm. This running time is optimal, because it is no larger than the actual encoding of the graph itself. That is, any algorithm which performs a depth-first search must at least read its entire input.

Exercise 7.7.1 *Use the algorithm for depth-first search to compute the connected components of a graph.*

Part V

Complexity Theory

Chapter 8

Complexity Theory

Computational complexity theory is a branch of the theory of computation in theoretical computer science and mathematics that focuses on classifying computational problems according to their inherent difficulty. In this context, a computational problem is understood to be a task that is in principle amenable to being solved by a computer. Informally, a computational problem consists of problem instances and solutions to these problem instances. For example, primality testing is the problem of determining whether a given number is prime or not. The instances of this problem are natural numbers, and the solution to an instance is yes or no based on whether the number is prime or not. A problem is regarded as inherently difficult if solving the problem requires a large amount of resources, whatever the algorithm used for solving it. The theory formalises this intuition, by introducing mathematical models of computation to study these problems and quantifying the amount of resources needed to solve them, such as time and storage.

8.1 Computational Complexity

How does complexity theory look on for a Turing machine. Here's a definition.

Definition 8.1.1 *Let M be a deterministic Turing machine that halt on all inputs. The* running time *or* time complexity *of M is the function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the maximum number of steps that M uses on any input of length n.*

The Big-O, Ω, and Θ notations defined in Chapter 2 still apply.
Let's set up some notation for classifying languages according to their time requirements.

Definition 8.1.2 *Let $t : \mathbb{N} \to \mathbb{R}^+$ be a function. Define the* time complexity class *$TIME(t(n))$ to be the collection of all languages that are decidable by an $O(t(n))$ time Turing machine.*

The language $A = \{0^k 1^k \mid k \geq 0\}$ takes time $O(n^2)$ to recognise using the following Turing machine, thus $A \in TIME(n^2)$.
M_1 is the machine which on input string w does the following:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.

2. Repeat if both 0s and 1s remain on the tape:
   Scan across the tape, crossing of a single 0 and a single 1.

3. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain on the tape, *accept*.

There are, however, better algorithms. The following machine for A shows that $A \in TIME(n \log n)$.
M_2 is the machine which on input string w does the following:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.

2. Repeat if both 0s and 1s remain on the tape:
   Scan across the tape, checking whether the total number of 0s and 1s is even or odd. If it is odd, *reject*.
   Scan across the tape, crossing off every other 0 starting with the first 0 and every other 1 starting with the first 1.

3. If no 0s and no 1s remain on the tape, *accept*. Otherwise, *reject*.

Note that this result cannot be further improved on a single-tape Turing machine. (In fact, it can be show than any language that can be decided in $O(n \log n)$ is regular.)

We can actually decide the language A in $O(n)$ time (*linear time*) if the Turing machine has a second tape. The following two-tape TM M_3 decides A in linear time. It simply copies the 0s to its second tape and then matches them against the 1s.

M_3 is the machine which on input string w does the following:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.

2. Scan across the 0s on tape 1 until the first 1, copying each 0 onto tape 2.

3. Scan across the 1s on tape 1 until the end of input. For each 1 read on tape 1, cross of a 0 on tape 2. If all 0s are crossed off before all the 1s are read, *reject*.

4. If all the 0s have been crossed of, *accept*. If any 0s remain, *reject*.

What the above discussion suggests is that the complexity of A depends upon the model of computation selected. This discussion highlights an important difference between complexity theory and computability theory. In computability theory, the Church-Turing thesis implies that all reasonable models of computation are equivalent—that is, they all decide the same class of languages. In complexity theory, the choice of model affects the time complexity of languages. Languages that are decidable in, say, linear time on one model aren't necessarily decidable in linear time on another.

In complexity theory, we classify computational problems according to their time complexity. But with which model do we measure time? The same language may have different time requirements on different models.

Fortunately, time requirements don't differ greatly for typical deterministic models. So, if our classification system isn't very sensitive to relatively small differences in complexity, the choice of deterministic model isn't crucial. We discuss this idea further in the next few sections.

8.1.1 Complexity Relationships Among Models

Here we examine how the choice of computational model can affect the time complexity of languages. We consider three models: the single-tape Turing machine, the multi-tape Turing machine, and the non-deterministic Turing machine.

Theorem 8.1.3 *Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time multi-tape Turing machine has an equivalent $O(t^2(n))$ time single-tape Turing machine.*

Proof Idea. The idea behind this proof is quite simple. Recall that we showed how to convert any multi-tape TM into a single-tape TM that simulates it. Now we analyse the simulation to determine how much additional time it requires. We show that simulating each step of the multi-tape machine uses at most $O(t(n))$ steps on the single-tape machine. Hence the total time used is $O(t^2(n))$. □

Next, we consider the analogous theorem for nondeterministic single-tape Turing machines (which are the obvious generalisation of deterministic TMs). We show that any language that is decidable on such a machine is decidable on a deterministic single-tape Turing machine that requires significantly more time. Before doing so, we must define the running time of a nondeterministic Turing machine. Call a nondeterministic Turing machine a *decider* if all its computation branches halt on all inputs.

Definition 8.1.4 *Let N be a nondeterministic Turing machine that is a decider. The* running time *of N is the function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the maximum number of steps that N uses on any branch of its computation of length n.*

The definition of the running time of a nondeterministic Turing machine is not intended to correspond to any real-world computing device. Rather, it is a useful mathematical definition that assists in characterising the complexity of and important class of computational problems.

Theorem 8.1.5 *Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time nondeterministic single-tape Turing machine has an equivalent $2^{O(t(n))}$ time deterministic single-tape Turing machine.*

Proof. Let N be a nondeterministic TM running in $t(n)$ time.

We construct a deterministic TM D that simulates N by searching N's nondeterministic computation tree. In this construction, D has 3 tapes, one for storing the input string, one for simulating a run of the individual branches of the N's computation, and one for recording D's location in N's non-deterministic computation tree. The third tape is a sequence of numbers indicating which choice was taken whenever a choice was possible in N.

D proceeds as follows:

1. Initially tape 1 contains the input w, and tapes 2 and 3 are empty.
2. Copy tape 1 to tape 2.
3. Use tape 2 to simulate N with input w on one branch of its non-deterministic computation. Before each step of N consult the next symbol on tape 3 to determine which choice to make among those allowed by N's transition function. If no more symbols remain on tape 3 or if this non-deterministic choice is invalid, abort this branch by going to stage 4. Also go to state 4 if a rejecting configuration is encountered. If an accepting configuration is encountered, *accept* the input.

4. Replace the string on tape 3 with the lexicographically next string. Simulate the next branch of $N's$ computation by going to stage 2.

Now we analyse that simulation. On an input if length n, every branch of N's nondeterministic computation tree has length of at most $t(n)$. Every node in the tree can have at most b children, where b is the maximum number of legal choices given by N's transition function. Thus the total number of leaves in the tree is at most $b^{t(n)}$.

The simulation proceeds by exploring this tree breadth first. In other words, it visits all nodes at depth d before going on to any of the nodes at depth $d+1$. The total number of nodes in the tree is at least twice the maximum number of leaves, so we bound it by $O(b^{t(n)})$. The time for running from the root and travelling down to a node is $O(t(n))$. Therefore the running time of D is $O(t(n)b^{t(n)}) = 2^{O(t(n))}$.

As D has three tapes, converting it to a single-tape TM at most squares the running time. Thus the running time of the single-tape simulator is $(2^{O(t(n))})^2 = 2^{O(2t(n))} = 2^{O(t(n))}$, and the theorem is proved. □

8.2 The Class P

In the previous section we discovered two significant differences between different simulations. On one hand, we demonstrated that at most a square or *polynomial* difference between the time complexity of problems measured on deterministic single-tape and multi-tape Turing machines. On the other hand, we showed at most an *exponential* difference between the time complexity of problems on deterministic and non-deterministic Turing machines.

8.2.1 Polynomial Time

For our purposes, polynomial differences in running time are considered small, whereas exponential differences are considered to be large. This is because, comparatively, exponentials grow extraordinarily faster compared to polynomials.

Exponential time algorithms typically arise when we solve by exhaustively searching through a space of solutions, called *brute force search*. Sometimes brute-force search may be avoided through a deeper understanding of a problem, which may reveal a polynomial time algorithm of greater utility.

All reasonable deterministic computational models are *polynomially equivalent*. That is, any one of them can simulate another with only a polynomial increase in running time.

From here on we focus on aspects of time complexity theory that are unaffected by polynomial differences in running time. We consider such differences to be insignificant and ignore them. Doing so allows us to develop the theory in a way that does not depend on the selection of a particular model of computation. Our aim is to present the fundamental properties of computation, rather than properties of Turing machines or any other special model.

Based on this discussion, we come to the following important definition in complexity theory.

Definition 8.2.1 (The Class P) *P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,*

$$P = \bigcup_k TIME(n^k).$$

The class P places a central role in complexity theory because

1. P is invariant for all models of computation that are polynomially equivalent to the deterministic single-tape Turing machine, and
2. P roughly corresponds to the class of problems that are realistically solvable on a computer.

Item 1 indicates that P is a mathematically robust class. Item 2 indicates that P is relevant from a practical standpoint.

8.2.2 Example Problems of P

$$PATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph that has a directed path from } s \text{ to } t\}.$$

A polynomial algorithm for *PATH* operates as follows.
M takes input $\langle G, s, t\rangle$ where G is a directed graph with nodes s and t.

1. Place a mark on node s.
2. Repeat the following until no additional edges are marked:
   Scan all the edges of G. If an edge (a, b) is going from a marked node a to an unmarked node b, mark node b.
3. If t is marked, *accept*. Otherwise, *reject*.

Hence

$$PATH \in P.$$

Say that two numbers are *relatively prime* if 1 is the largest integer that evenly divides them both. For example, 10 and 21 are relatively prime, whereas 10 and 22 are not because both are divisible by 2.

$$RELPRIME = \{\langle x, y\rangle \mid x \text{ and } y \text{ are relatively prime}\}.$$

The algorithm for computing this is the well known *Euclidean Algorithm*, based on the following algorithm for computing the *greatest common divisor*.
$gcd(x, y)$ takes two natural numbers:

$$\begin{aligned} gcd(x, 0) &= x \\ gcd(x, y) &= gcd(y, x \bmod y) \qquad y > 0 \end{aligned}$$

Now the algorithm for computing *RELPRIME* is

1. Compute $gcd(x, y)$.
2. If result is 1, *accept*. Otherwise, *reject*.

This is clearly polynomial as the step $x \bmod y$ reduces the value of x by at least half.
Thus

$$RELPRIME \in P.$$

The final example of a polynomial time algorithm is to show that every context-free language is decidable in polynomial time.

Theorem 8.2.2 *Every context-free language is a member of P.*

If L is a context free language, then it is generated by some context-free grammar G. This can be readily placed in Chomsky normal form (Every rule is of the form $A \to BC$ or $A \to a$, where B and C cannot be the start symbol, plus possibly $S \to \epsilon$). Any derivation of string w has $2n-1$ steps, where n is the length of w, because G is in Chomsky normal form. The decider for L works by trying all possible derivations with $2n-1$ steps when the input is of length n. If any of these is a derivation of w, then the decider accepts. If not it rejects.

A quick analysis of this algorithm shows that it does not run in polynomial time, as the number of derivations with k steps may be exponential in k.

To get a polynomial time algorithm we use a powerful technique known as *dynamic programming*, which uses the accumulation of information about smaller subproblems to solve larger problems. We do so by making a table of all subproblems and entering their solutions systematically as we find them.

In this case, we consider the subproblems of determining whether each nonterminal in G generates each substring of w. The algorithm enters the solution to this subproblem in an $n \times n$ table. For $i \leq j$ the (i,j)th entry of the table contains the collection of non-terminals that generate the substring $w_i w_{i+1} \cdots w_j$.

The algorithm fills in table entries for each substring of w. First it fills in the entries for the substrings of length 1, then those of length 2, and so on. It uses the entries for the shorter lengths to assist in determining the entries for longer lengths.

Some effort shows that this algorithm can run in $O(n^3)$.

Thus,

$$CFL \in P.$$

8.3 The Class NP

Although it has been possible to avoid brute force algorithms, which are typically exponential, and find more clever polynomial time algorithms in many cases, for many interesting and useful problems this search has not been successful, and polynomial time algorithms that solve them are not known to exist.

Why have we been unsuccessful in finding polynomial time algorithms for these problems? The answer to this question is not known. But one remarkable discovery concerning this question is that the complexities of many problems are linked. Finding a polynomial time algorithm for one such problem can be used to solve an entire class of problems.

Consider the following example.

A *Hamiltonian path* in a directed graph G is a directed path that goes through each node exactly once. Consider the following problem:

$$HAMPATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph with a Hamiltonian path from } s \text{ to } t.\}$$

We can easily obtain an exponential time algorithm for *HAMPATH*, but no one knows whether *HAMPATH* is solvable in polynomial time.

The *HAMPATH* problem does have a feature called *polynomial verifiability* that is important for understanding its complexity. Even though we don't know a fast way to determine whether a graph contains a Hamiltonian path, if such a path were discovered somehow (perhaps using an exponential time algorithm), we could easily (in polynomial time) verify that the path was indeed Hamiltonian.

Another polynomially verifiable problem is compositeness. Recall that a natural number is *composite* if it is the product of two integers greater than 1 (that is, it is not a prime number).

$$COMPOSITES = \{x \mid x = pq, \text{ for integers } p, q > 1\}.$$

It is easy to check given a divisor of a number whether the number is composite. A polynomial time algorithm for testing primeness/compositeness has recently been found, but it is considerably more complicated that the method for checking that a number is composite, given one of its divisors.

Some problems may not be polynomially verifiable. $\overline{HAMPATH}$, the complement of the *HAMPATH* problem, is one such one. For even if we know that a graph does not have a Hamiltonian path, we do not know any way of verifying this fact without using some exponential algorithm for making the determination in the first place.

Definition 8.3.1 (Verifier) *A* verifier *for a language A is an algorithm V, where*

$$A = \{w \mid V \text{ accepts } \langle w, c\rangle \text{ for some string } c\}.$$

We measure the time of a verifier only in the length of w, so a polynomial time verifier *runs in polynomial time in the length of w. A language A is* polynomially verifiable *if it has a polynomial time verifier.*

The idea is that the verifier uses the information encoded in string c, often known as a certificate, to verify that the string w is a member of A.

Definition 8.3.2 *NP is the class of languages that have polynomial verifiers.*

The term NP comes from *non-deterministic polynomial time* and is derived from an alternative characterisation by using non-deterministic polynomial time Turing machines. Problems in NP are often called NP-problems.

Theorem 8.3.3 *A language is in NP if and only it is decided by some non-deterministic polynomial time Turing machine.*

Proof. For the forward direction of this theorem, let $A \in NP$ and show that A is decided by a polynomial time NTM. Let V be the polynomial time verifier for A that exists by the definition of NP. Assume that V is a TM that runs in time n^k and construct N as follows.

N takes input w of length n and

1. Nondeterministically selects a string c of length at most n^k.
2. Runs V on the input $\langle w, c\rangle$.
3. If V accepts, *accept*; otherwise, *reject*.

To prove the other direction of the theorem, assume that A is decided by a polynomial time NTM N and construct a polynomial time verifier as follows.

V takes input $\langle w, c\rangle$ where w and c are strings and

1. Simulate N on input w, treating each symbol c as a description of the non-deterministic choice to make at each step.
2. If the branch of N's computation accepts, *accept*; otherwise, *reject*.

□

An alternative characterisation is as follows. Define the non-deterministic time complexity class $NTIME(t(n))$ as follows:

$$NTIME(t(n)) = \{L \mid L \text{ is a language decidable by a } O(t(n)) \text{ time non-deterministic Turing machine}\}.$$

Corollary 8.3.4

$$NP = \bigcup_k NTIME(n^k).$$

When analysing an algorithm to determine whether it is in NP, each step of the algorithm must be reasonably computed in non-deterministic polynomial time on some reasonable non-deterministic computation model. Typically, we analyse the algorithm to show that every branch uses at most polynomially many stages.

8.3.1 Example Problems in NP

A *clique* of an undirected graph is a subgraph wherein every two nodes are connected by an edge. A k-clique is a clique that contains k nodes. The clique problem is to determine whether a graph contains a clique of a specified size. Let

$$CLIQUE = \{\langle G, k\rangle \mid G \text{ is an undirected graph with a } k\text{-clique}\}$$

Theorem 8.3.5 *CLIQUE is in NP.*

Proof. The clique is the certificate.
The following is a verifier V for *CLIQUE*.
V takes input $\langle\langle G, k\rangle, c\rangle$ and:

1. Test whether c is a set of nodes in G.
2. Test whether G contains all edged connecting nodes in c.
3. If both pass, *accept*; otherwise, *reject*.

□

The *SUBSET-SUM* problem concerns integer arithmetic. In this problem, we have a collection of numbers $x_1, \ldots, x_k$ (possibly with duplicates) and a target number t. We want to determine whether the collection contains a subcollection that adds up to t.

$$SUBSET\text{-}SUM = \left\{\langle S, t\rangle \mid S = \{x_1, \ldots, x_k\} \text{ and for some } Y \subseteq S, \text{ we have } \sum Y = t\right\}.$$

For example, $\langle\{4, 11, 16, 21, 27\}, 25\rangle \in SUBSET\text{-}SUM$ because $4 + 21 = 25$.

Theorem 8.3.6 *SUBSET-SUM is in NP.*

Proof. The subset is the certificate.

The following is a verifier for *SUBSET-SUM*.

V takes input $\langle\langle S, t\rangle, c\rangle$ and

1. Test whether c is a collection of numbers that sum to t.
2. Test whether S contains all the numbers in c.
3. If both pass, *accept*; otherwise, *reject*.

□

Observe that the complements of these sets, $\overline{CLIQUE}$ and $\overline{SUBSET\text{-}SUM}$ are not obviously members of NP. Verifying that something is *not* present seems to be more difficult that verifying that it is present. We make a separate complexity class, called coNP which contains languages that are the complements of languages in NP.

It is unknown whether coNP is different from NP.

8.3.2 The P vs NP Question

Recall:

- P = the class of languages for which membership can be *decided* quickly.
- NP = the class of languages for which membership can be *verified* quickly.

It may be hard to imagine, but P and NP could be equal. We are unable to *prove* the existence of a single language in NP that is not in P.

The question of whether P=NP is one of the greatest unsolved problems in theoretical computer science and contemporary mathematics. If these classes were equal, any polynomially verifiable problem would be polynomially decidable. Most people believe that the classes are not equal, because a considerable amount of effort has been invested in finding algorithms for certain problems in NP without success.

In any case, we have either P⊂NP or P=NP.

The best know method for solving languages in NP deterministically uses exponential time. In other words we can prove that

$$NP \subseteq EXPTIME = \bigcup_k TIME(2^{n^k}),$$

but we do not know whether NP is contained in a smaller deterministic complexity class.

8.4 NP-Completeness

One important advance on the P vs NP question came in the early 1970s with the work of Stephen Cook and Leonid Levin. They discovered certain problems in NP whose individual complexity is related to that of the entire class. If a polynomial time algorithm exists for any of these problems, *all* problems in NP would be polynomial time solvable. These problems are called *NP-complete*. The phenomenon of NP-completeness is important for both theoretical and practical reasons.

On the theoretical side, a researcher trying to show that P is unequal to NP may focus on an NP-complete problem. If any problem in NP requires more than polynomial time, an NP-complete one does. Furthermore, a researcher attempting to prove that P equals NP only needs to find a polynomial time algorithm for an NP-complete problem to achieve this goal.

On the practical side, the phenomenon of NP-completeness may prevent wasting time searching for a nonexistent polynomial time algorithm to solve a particular problem. Even though we may not have the necessary mathematics to prove that the problem is unsolvable in polynomial time, it is generally believed that P is unequal to NP, so proving that a problem is NP-complete is strong evidence of its non-polynomiality.

The first NP-complete problem that we present is called the *satisfiability problem*. Recall that *Boolean variables* take on the values TRUE and FALSE (represented by 1 and 0). Recall also the usual *Boolean operators* $\wedge$, $\vee$, and $\neg$. A *Boolean formula* is an expression involving Boolean variables and operators, for example: $\phi = (\neg x \wedge y) \vee (x \wedge \neg z)$. A Boolean formula is satisfiable if some assignment of 0s and 1s to the variables makes the formula evaluate to 1. The preceding formula is satisfiable because the assignment $x = 0$, $y = 1$, and $z = 0$ make ϕ evaluate to 1. We say that the assignment *satisfies* ϕ. The *satisfiability problem* is to test whether a Boolean formula is satisfiable. Let

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

We now state the Cook-Levin theorem, which links the complexity of the *SAT* problem to the complexities of all problems in NP.

Theorem 8.4.1 (Cook-Levin) *SAT* $\in P$ *iff* $P = NP$.

Next we develop the method that is central to the proof of the Cook-Levin Theorem.

8.5 Polynomial-time Reducibility

We expand upon the notion of reducibility between problems. When a problem A is *efficiently* reducible to problem B, an efficient solution to B can be used to solve A efficiently.

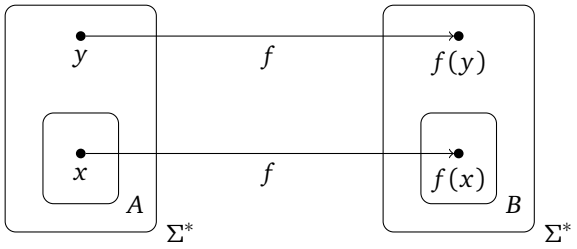
Definition 8.5.1 *A function* $f : \Sigma^* \to \Sigma^*$ *is a* polynomial time computable function *if some polynomial time Turing machine exists that halts with just* $f(w)$ *on its tape, when started on any input* w.

Definition 8.5.2 *Language* A *is* polynomial time reducible *to language* B*, written* $A \leq_P B$*, if a polynomial time computable function* $f : \Sigma^* \to \Sigma^*$ *exists, where for every* w*,*

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

The function f *is called the* polynomial time reduction *of* A *to* B.

This is illustrated by the following figure.



As with an ordinary mapping reduction, a polynomial time reduction of A to B provides a way to convert membership testing in A to membership testing in B, but now the conversion is done efficiently. To test whether $w \in A$, we use the reduction f to map w to $f(w)$ and test whether $f(w) \in B$.

If one language is polynomial time reducible to a language already known to have a polynomial time solution, we obtain a polynomial time solution to the original language, as in the following theorem.

Theorem 8.5.3 *If $A \leq_P B$ and $B \in P$, then $A \in P$.*

Proof. Let M be the polynomial time algorithm deciding B and f be the polynomial time reduction from A to B. We describe a polynomial time algorithm N deciding A as follows.

N is does the following on input w:

1. Compute $f(w)$
2. Run M on input $f(w)$ and output whatever M outputs.

We have $w \in A$ whenever $f(w) \in B$, because f is a reduction from A to B. Thus M accepts $f(w)$ whenever $w \in A$. Moreover, N runs in polynomial time because each of its two steps runs in polynomial time. □

Before demonstrating a polynomial time reduction we introduce *3SAT*, a special case of the satisfiability problem whereby all formulas are in a special form. A *literal* is a Boolean variable or a negated Boolean variable, as in x or $\overline{x}$ (denoting $\neg x$). A *clause* is several literals connected with $\vee$s, as in $x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4$. A Boolean formula is in *conjunctive normal form*, called a *cnf-formula*, if it comprises several clauses connected with $\wedge$s, as in

$$(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6}).$$

It is a *3cnf-formula* if all the clauses have three literals, as in

$$(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6} \vee x_4).$$

Let

$$3SAT = \{\langle \phi \rangle \mid \phi \text{ is a satisfiable 3cnf-formula}\}.$$

In a satisfiable cnf-formula, each clause must contain at least one literal that is assigned 1.

The following theorem presents a polymonial time reduction from the *3SAT* problem to the *CLIQUE* problem.

Theorem 8.5.4 3SAT *is polynomial time reducible to* CLIQUE.

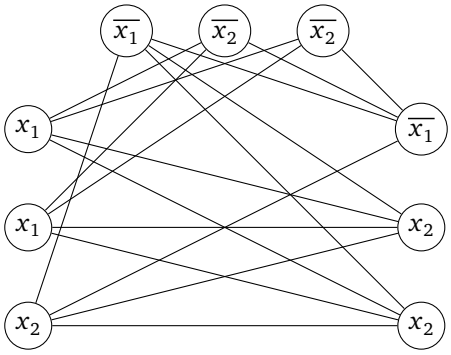
Proof. Let ϕ be a formula with k clauses such as

$$\phi = (a_1 \vee b_1 \vee c_1) \wedge \cdots \wedge (a_k \vee b_k \vee c_k).$$

The reduction f generates the string $\langle G, k \rangle$, where G is an undirected graph defined as follows.

The nodes in G are organised into k groups of three nodes each called the triples $t_1, \ldots, t_k$. Each triple corresponds to one of the clauses in ϕ, and each node is a triple corresponding to a literal in the associated clause. Label each node of G with its corresponding literal in ϕ.

The edges of G connect all but two types of pairs of nodes in G. No edge is present between nodes in the same triple and no edge is present between two nodes with contradictory labels, as in x_2 and $\overline{x_2}$. The following figure illustrates this construction when $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$.



Now we demonstrate why this construction works. We show that ϕ is satisfiable iff G has a k-clique.

Suppose that ϕ has a satisfying assignment. In each satisfying assignment, at least one literal is true in every clause. In each triple of G, we select one node corresponding to a true literal in the satisfying assignment. If more than one literal is true, we choose one of the true literals arbitrarily. The nodes just selected form a k-clique. The number of nodes selected is k, because we chose one from each of the k triples. Each pair of selected nodes is joined by an edge because no pair fits one of the exceptions described previously. They could not be from the same triple because we selected only one node per triple. They could not be contradictory labels because the associated literals were both true in the satisfying assignment. Therefore G contains a k-clique.

Suppose that G has a k-clique. No two of the clique's nodes occur in the same triple because nodes in the same triple aren't connected by edges. Therefore each of the k triples contains exactly one of the k clique nodes. We assign truth values to the variables of ϕ so that each literal labelling a clique is made true. Doing so is always possible because two nodes labelled in a contradictory way are not connected by an edge and hence cannot be in the clique. This assignment to the variables satisfies ϕ because each triple contains a clique node and hence each clause contains a literal assigned to TRUE. Therefore ϕ is satisfiable. □

Theorems 8.5.3 and 8.5.4 tell us that, if *CLIQUE* is solvable in polynomial time, so is *3SAT*. At first glance, this connection between these two problems appears quite remarkable because, superficially, they are rather different. But polynomial time reducibility allows us to link their complexities. Now we turn to a definition that will allow us similarly to link the complexities of entire classes of problems.

Definition 8.5.5 (NP-complete) *A language B is* NP-complete *if it satisfies two conditions:*

1. *B is in NP, and*
2. *every A in NP is polynomial reducible to B.*

Theorem 8.5.6 *If B is NP-complete and $B \in P$, then $P = NP$.*

Proof. This theorem follows directly from the definition of polynomial time reducibility. □

Theorem 8.5.7 *If B is NP-complete and $B \leq_P C$ for C in NP, then C is NP-complete.*

Proof. We already know that C is in NP, so we must show that every A in NP is polynomial time reducible to C. Because B is NP-complete, every language in NP is polynomial time reducible to B, and B is in turn polynomial-time reducible to C. Polynomial time reductions compose; thus, A is polynomial-time reducible to C. Hence every language in NP is polynomial-time reducible to C. □

8.5.1 The Cook-Levin Theorem

Once we have one NP-complete problem, we may obtain others by polynomial time reduction from it. However, establishing the first NP-complete problem is more difficult. We state here that *SAT* is NP-complete and give the proof idea.

Theorem 8.5.8 (Cook-Levin) SAT *is NP-complete.*

Proof Idea. Showing that *SAT* is in NP is easy. The hard part is showing that any language in NP is polynomial reducible to *SAT*.

To do so we construct a polynomial time reduction for each language A in NP to SAT. The reduction for A takes a string w and produces a Boolean formula ϕ that simulates the NP machine for A on w. If the machine accepts, ϕ has a satisfying assignment that corresponds to the accepting computation. If the machine doesn't accept, no assignment satisfies ϕ. Therefore w is in A if and only if ϕ is satisfiable.

Actually constructing the reduction to work in this way is conceptually a simple task, though we must cope with many details. A Boolean formula may contain the Boolean operators AND, OR, and NOT, and these operations form the basis for the circuitry used in electronic computers. Hence the fact that we can design a Boolean formula to simulate a Turing machine isn't surprising. The details are in the implementation of this idea.

You are encouraged too look this construction up in one of the books mentioned at the start of these notes. □

The proof is quite lengthy. But now that we have one NP-complete problem, showing that other problems are NP-complete is much simpler: one need only provide a polynomial time reduction from a language that is already known to be NP-complete. We can use *SAT* for this purpose, but using *3SAT* is usually easier. That *3SAT* is NP-complete is a corollary to Theorem 8.5.8.

Corollary 8.5.9 3SAT *is NP-complete.*

8.5.2 Additional NP-Complete Problems

Corollary 8.5.10 CLIQUE *is NP-complete.*

Theorem 8.5.11 HAMPATH *is NP-complete.*

Let *UHAMPATH* correspond to the problem of finding an undirected Hamiltonian path.

Theorem 8.5.12 UHAMPATH *is NP-complete.*

We show the following in detail.

Theorem 8.5.13 SUBSET-SUM *is NP-complete.*

Proof. We already know that *SUBSET-SUM* $\in$ *NP*, so we now show that *3SAT* $\leq_P$ *SUBSET-SUM*.

Let ϕ be a Boolean formula with variables $x_1, \ldots, x_l$ and clauses $c_1, \ldots, c_k$. The reduction converts ϕ to an instance of the *SUBSET-SUM* problem $\langle S, t \rangle$, wherein the elements of S and the number t are the rows in the table below, expressed in ordinary decimal notation.

	1	2	3	4	$\cdots$	l	c_1	c_2	$\cdots$	c_k
y_1	1	0	0	0	$\cdots$	0	1	0	$\cdots$	0
z_1	1	0	0	0	$\cdots$	0	0	0	$\cdots$	0
y_2		1	0	0	$\cdots$	0	0	1	$\cdots$	0
z_2		1	0	0	$\cdots$	0	1	0	$\cdots$	0
y_3			1	0	$\cdots$	0	1	1	$\cdots$	0
z_3			1	0	$\cdots$	0	0	0	$\cdots$	1
$\vdots$					$\ddots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$
y_l						1	0	0	$\cdots$	0
z_l						1	0	0	$\cdots$	0
g_1							1	0	$\cdots$	0
h_1							1	0	$\cdots$	0
g_2								1	$\cdots$	0
h_2								1	$\cdots$	0
$\vdots$									$\ddots$	$\vdots$
g_k										1
h_k										1
t	1	1	1	1	$\cdots$	1	3	3	$\cdots$	3

The rows above the double line are labelled

$$y_1, z_1, y_2, z_2, \ldots, y_l, z_l \quad \text{and} \quad g_1, h_1, g_2, h_2, \ldots, g_k, h_k$$

and comprise the elements of S. The row below the double line is t.

Thus S contains one pair of numbers y_i, z_i for each variable x_i in ϕ. The decimal representation of these numbers comes in two parts, as indicated in the table. The left-hand part comprises a 1 followed by $l - i$ 0s. The right-hand part contains one digit for each clause, where the jth digit of y_i is 1 if clause c_j contains literal x_i and the jth digit of z_i is 1 if clause c_j contains literal $\overline{x_j}$. Digits not specified to be 1 are 0.

The table is partially filled in to illustrate sample clauses, c_1, c_2, and c_k:

$$(x_1 \vee \overline{x_2} \vee x_3) \wedge (x_2 \vee x_3 \vee \cdots) \wedge \cdots \wedge (\overline{x_3} \vee \cdots \vee \cdots)$$

Additionally, S contains one pair of numbers, g_j, h_j, for each clause c_j. These two numbers are equal and consist of a 1 followed by $k - j$ 0s.

Finally, the target number t, the bottom row of the table, consists of l 1s followed by k 3s.

Now we show why this construction works. We demonstrate that ϕ is satisfiable iff some subset S sums to t.

Suppose that ϕ is satisfiable. We construct the subset of S as follows. We select y_i if x_i is assigned TRUE in the satisfying statement and z_i if x_i is assigned FALSE. If we add up what we have selected so far, we obtain a 1 in each of the first l digits because we have selected either y_i and z_i for each i.

Furthermore, each of the last k digits is a number between 1 and 3 because each clause is satisfied and contains between 1 and 3 true literals. Now we further select enough of the g and h numbers to bring each of the last k digits up to 3, thus hitting the target.

Suppose that a subset of S sums to t. We construct a satisfying assignment to ϕ after making several observations. First, all the digits in members of S are either 0 or 1. Furthermore, each column in the table describing S contains at most five 1s. Hence a "carry" into the next column never occurs when a subset of S is added. To get a 1 in each of the first l columns, the subset must have either y_i or z_i for each i, but no both.

Now we make the satisfying assignment. If the subset contains y_i, we assign x_i TRUE; otherwise we assign it false. This assignment must satisfy ϕ because in each of the final k columns the sum is always 3. In column c_j, at most 2 can come from g_j and h_j, so at least 1 in this column must come from some y_i or z_i in the subset. If it is y_i, then x_i appears in c_j and is assigned TRUE, so c_j is satisfied. If it is z_i, then $\overline{x_i}$ appears in c_j and x_i is assigned FALSE, so c_j is satisfied. Therefore ϕ is satisfied.

Finally, we must be sure that the reduction can be carried out in polynomial time. The table has a size of roughly $(k+l)^2$, and each entry can be easily calculated for any ϕ. So the total time is $O(n^2)$ easy stages. □

8.6 Space Complexity

We now briefly cover the topic of space complexity, mainly to capture the notion of *PSPACE*.

Space complexity is defined in much the same way that time complexity is, but instead of talking about the number of steps required to perform the computation, it talks about the maximum number of tape cells required.

Definition 8.6.1 (Space Complexity) *Let M be a deterministic Turing machine that halts on all inputs. The* space complexity *of M is the function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the maximum number of tape cells that M scans on any input of length n.*

If M is a non-deterministic Turing machine wherein all branches halt on all inputs, we define its space complexity $f(n)$ to be the maximum number of tape cells that M scans on any branch of its computation for any input of length n.

We typically estimate the space complexity of a Turing machine using the usual asymptotic notation.

Definition 8.6.2 *Let $f : \mathbb{N} \to \mathbb{R}^+$ be a function. The* space complexity classes *are defined as follows:*

$$\begin{aligned} SPACE(f(n)) &= \{L \mid L \text{ is a language decided by an } O(f(n)) \text{ space deterministic Turing machine}\} \\ NSPACE(f(n)) &= \{L \mid L \text{ is a language decided by an } O(f(n)) \text{ space non-deterministic Turing machine}\} \end{aligned}$$

SAT can be solved by a linear space algorithm. One simply generates each truth assignment (which requires n loop variables) and evaluates the truth of the formula. The machine runs in linear space because it can reuse the same tape portion for each different assignment to variables, and the testing of a single assignment can be done in $O(n)$ space.

Let

$$ALL_{NFA} = \{\langle A \rangle \mid A \text{ is an NFA and } L(A) = \Sigma^*\}.$$

There is a non-deterministic linear space algorithm that decides the complement of this language, $\overline{ALL_{NFA}}$. It simply uses non-determinism to guess a string that is rejected by the NFA and uses linear space to keep track of which states the NFA could be in at a particular time. Note that this language is not known to be in NP or coNP.

The following theorem relates deterministic and non-deterministic space complexity.

Theorem 8.6.3 (Savitch's Theorem) *For any function $f : \mathbb{N} \to \mathbb{R}^+$, where $f(n) \geq n$,*

$$NSPACE(f(n)) \subseteq SPACE(f^2(n)).$$

The proof involves analysing a particular approach to encoding the computation of a non-deterministic TM in terms of a deterministic one.

8.6.1 The Class PSPACE

By analogy with the class P, we define the class *PSPACE* for space complexity.

Definition 8.6.4 *PSPACE is the class of languages that are decidable in polynomial space on a deterministic Turing machine. In other words,*

$$PSPACE = \bigcup SPACE(n^k).$$

We define *NPSPACE* as the non-deterministic counterpart to *PSPACE*. However, $NPSPACE = PSPACE$ by virtue of Savitch's theorem.

We conclude by describing what is known about some of the various complexity classes we have seen:

$$P \subseteq NP \subseteq PSPACE = NPSPACE \subseteq EXPTIME.$$

It is only known that $P \neq EXPTIME$. Most researchers believe that these containments are proper, but no proofs exist.

Part VI

Exercises

Chapter 9

Exercises

9.1 Exercises #1

1. Find DFAs for the following languages:

   (a) The set of strings $\{4, 8, 1\}^*$ containing the substring 481.

   (b) The set of strings in $\{a\}^*$ whose length is divisible by either 2 or 7.

   (c) The set of strings $x \in \{0, 1\}^*$ such that $\#_0(x)$ is even and $\#_1(x)$ is a multiple of 3.

   (d) The set of strings over the alphabet a, b containing at least three occurrences of three consecutive b's, overlapping permitted (e.g., the string $bbbbb$ should be accepted).

2. Find NFAs for the following languages:

   (a) $\{x \in \{0, 1\}^* \mid x$ contains an equal number of occurrences of 01 and 10$\}$

   (b) $\{x \in \{a, b\}^* \mid x$ contains an even number of a's *or* an odd number of b's$\}$

3. Construct a nondeterministic finite automaton and an equivalent deterministic one that accept those sequences over the alphabet $\{a, b, c, d\}$ such that at least one symbol appears precisely twice in the sequence.

4. Describe the following languages using regular expressions ($\Sigma = \{a, b\}$):

   (a) $\{x \mid x$ contains an even number of a's$\}$.

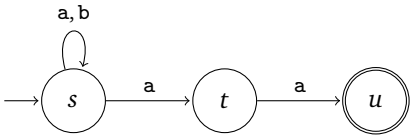
   (b) $\{x \mid x$ contains an odd number of b's$\}$.

   (c) $\{x \mid x$ contains an even number of a's *or* an odd number of b's$\}$.

   (d) $\{x \mid x$ contains an even number of a's *and* an odd number of b's$\}$.

   Try to simplify the expressions are much as possible (justify each simplification).

5. Convert the following NFA into a DFA using the subset construction:



Clearly show which subset of $\{s, t, u\}$ corresponds to each state of the deterministic automaton. Omit inaccessible states.

6. Minimise the following DFA using the quotient construction:

		a	b
→	1	1	4
	2	3	1
	3F	4	2
	4F	3	5
	5	4	6
	6	6	3
	7	2	4
	8	3	1

7. Convert the DFA from the previous question into a regular expression (use either original DFA or quotient automaton).

8. Convert the following regular expressions into DFAs:

   (a) $(000^* \cup 111^*)^*$.

   (b) $(01 \cup 10)(01 \cup 10)(01 \cup 10)$.

   (c) $(0 \cup 1(01^*0)^*1)^*$.

   Try to simplify as much as possible.

9. Use the pumping lemma to demonstrate that the following languages are not regular:

   (a) $\{a^n b^m \mid n = 2m\}$.

   (b) $\{x \in \{a, b, c\}^* \mid x \text{ is a palindrome; i.e., } x = x^R\}$.

   (c) The set *PAREN* of balanced parentheses ($\Sigma = \{(,)\}$).

10. Give a context-free grammar for the following languages:

    (a) $PAREN_2$ of balanced strings of parentheses of two types ( ) and [ ]. How would you prove that your grammar is correct?

    (b) The set of non-null strings over $\{a, b\}$ with equally many a's as b's.

11. Recall that the *reverse* of a string x, denoted x^R, is x written backwards. Formally,

$$\epsilon^R = \epsilon \qquad (xa)^R = ax^R.$$

    For language $L \subseteq \Sigma^*$, define

$$L^R = \{x^R \mid x \in L\}.$$

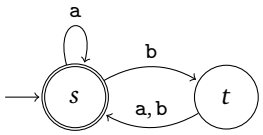
(a) Given $u, v \in \Sigma^*$. Show that $(uv)^R = v^R u^R$.

(b) Given $L_1, L_2 \subseteq \Sigma^*$. Show that $(L_1 L_2)^R = L_2^R L_1^R$.

(c) Show that for any $L \subseteq \Sigma^*$, if L is regular, then so is L^R.

9.2 Exercises #2

1. Give a context-free grammar that generates the language
$$L = \{a^i b^j c^k \mid i = j \text{ or } j = k \text{ where } i, j, k \geq 0\}.$$
2. Give a pushdown automaton for the following languages:
   (a) $PAREN_2$ of balanced strings of parentheses of two types ( ) and [ ].
   (b) $\{a^i b^j c^k \mid i, j, k \geq 0 \text{ and } i = j \text{ or } i = k\}$.
   (c) The complement of the language $\{a^n b^n \mid n \geq 0\}$.
   (d) $\{w \in \{0,1\}^* \mid w = w^R, \text{ that is, } w \text{ is a palindrome.}\}$
3. Use the pumping lemma for context free languages to show that the following are not context free
   (a) $\{a^n b^n c^n \mid n \geq 0\}$.
   (b) $A = \{ww \mid w \in \{0,1\}^*\}$. Hint consider $A \cap L(a^* b^* a^* b^*)$.
4. Prove that every regular language is context free, by showing how to convert a regular expression directly into an equivalent context-free grammar.
5. Use derivatives to show that $abbb$ is an element of $(a \cup b)b^*a^*$, and that $abbab$ is not.
6. Use derivatives to generate a DFA for $(a \cup b)b^*a^*$.
7. Use the axioms of Kleene algebra to show the following. Use only the axioms (A.1) to (A.15) from Table 9.1.
   (a) $a^*a^* = a^*$
   (b) $a^*a = aa^*$
   (c) $(a^*b)^*a^* = (a+b)^*$.
8. Given the following automata.



   (a) Express the language accepted by the automata as a series of linear equations.
   (b) Solve the equations to produce a regular expression describing the accepted language.
9. Give a Turing machine (in full detail) to perform
   (a) addition
   (b) multiplication

   Assume that the input string is of the form $a^n \# a^m$. In the first case, the result, namely the string on the TM when it halts, should be a^{n+m}. In the second case, the result should be a^{nm}.

   The TM should reject if the input string is not of the correct form.

Table 9.1: Axioms of Kleene Algebra

(A.1) $a + (b + c) = (a + b) + c$

(A.2) $a + b = b + a$

(A.3) $a + a = a$

(A.4) $a + 0 = a$

(A.5) $a(bc) = (ab)c$

(A.6) $a1 = 1a = a$

(A.7) $a0 = 0a = 0$

(A.8) $a(b + c) = ab + ac$

(A.9) $(a + b)c = ac + bc$

(A.10) $1 + aa^* = a^*$

(A.11) $1 + a^*a = a^*$

(A.12) $b + ac \leq c \Rightarrow a^*b \leq c$

(A.13) $b + ca \leq c \Rightarrow ba^* \leq c$

(A.14) $ac \leq c \Rightarrow a^*c \leq c$

(A.15) $ca \leq c \Rightarrow ca^* \leq c$

9.3 Exercises #3

1. Give Turing machines for the following:

   (a) Decide $L = \{w \in \{0,1\}^* \mid L \text{ does not contain twice as many 0s as 1s}\}$.

   (b) Given input a^m, the TM halts with a^{m^2} on its tape.

2. Design a Turing machine to compute the function $\max(x, y) =$ the larger of x and y.

3. Consider a Turing machine model that uses a 2-dimensional tape, corresponding to the upper right quadrant of the plane. The head of such a Turing machine can move to the right, left, up or down.

   Sketch a proof that such a model does not add extra computing power; that is, the class of languages recognised by such Turing machines is the same as the class recognised by basic Turing machines.

   Be careful to define what the language of the 2-d TM is.

   Discuss how to formalise this model.

4. Is is decidable for TM M whether $L(M) = (L(M))^R$, that is, is the language of M equal to its reverse?

5. Show that $\sim HP = \{M\#x \mid M \text{ does not halt on } x\}$ is not decidable.

6. Determine whether the following problems are decidable or undecidable. Give proof.

   (a) Given a TM M and a string y, does M write the symbol # on its tape on input y?

   (b) Given a context-free grammar G, does G generate all strings except ϵ?

   (c) Given a TM M and a string y, does M ever write a non-blank symbol on its tape on input y?

   (d) Given a TM M and a string y, does the machine ever attempt to move its head left at any point during the computation on y?

7. Determine whether the following languages are decidable or not. If so, give a Turing machine for deciding it. If not, give a proof. Assume that the input M is a description of a Turing machine.

   (a) $\{\langle M \rangle \mid M \text{ accepts at least two strings of different length.}\}$

   (b) $\{\langle M, w \rangle \mid M \text{ accepts } w \text{ and rejects } w^R\}$.

9.4 Exercises #4

1. Express the minimum and maximum degrees vertices in the graphs K_n (complete graphs) and $K_{n,m}$ (complete bi-partite graphs) in terms of n and m.

2. Specify a minimum value for n in terms of l and m such that $K_{l,m}$ is a subgraph of K_n.

3. A *regular graph* is a graph where each vertex has the same number of neighbours, i.e. every vertex has the same degree. A regular graph with vertices of degree k is called a *k-regular graph*.

   For every k-regular graph is there a $(k+1)$-regular graph that contains the k-regular graph as a subgraph? Proof or counterexample.

4. Is there a *simple* graph on n vertices such that all vertices have distinct degrees? Is there a (general) graph with this property?

5. Is it possible to draw a graph that has a trail of length seven but no paths of length seven? If so, draw such an example. Is it possible to draw a graph that has a path of length seven but no trail of length seven? If so, draw such an example.

6. Show that for $n \in \mathbb{N}$, the graph $K_{n,n}$ has a subgraph isomorphic to C_k, a cycle of length k, for even $k \in \{4, \ldots, 2n\}$.

7. Prove that if a graph G has exactly two vertices u and v of odd degree, then G contains a path from u to v.

8. Define a relation on graphs such that G and G' are related if and only if the maximum degree of G is less than or equivalent to the maximum degree of G'. Does the relation satisfy the reflexive, transitive, and antisymmetric properties? Is it a partial order? Does it satisfy the symmetric property? Is the relation an equivalence relation?[1]

9. It can be shown that there are exactly 11 trees on seven vertices. Draw these eleven trees, making sure that no two are isomorphic.

10. Let G be a k-regular bipartite graph with $k \geq 2$. Show that G has no cut edge (or bridge).

11. Show that if a digraph $\vec{G}$ contains a directed circuit of positive length, then $\vec{G}$ must contain a cycle.

12. For $k \in \mathbb{N}$, let G be a connected graph that contains $2k$ vertices of odd degree. Show that there exist k edge-disjoint subgraphs $G_1, \ldots, G_k$, such that

[1]

reflexivity a relation $R \subseteq A \times A$ is reflexive if and only if for all $a \in A$ we have $(a, a) \in R$.

transitivity a relation $R \subseteq A \times A$ is transitive if and only if for all $a, b, c \in A$ if $(a, b) \in R$ and $(b, c) \in R$ then $(a, c) \in R$.

antisymmetry a relation $R \subseteq A \times A$ is antisymmetric if and only if for all $a, b \in A$ if $(a, b) \in R$ and $(b, a) \in R$, then $a = b$.

partial order a relation is a partial order if and only if it is reflexive, transitive and antisymmetric.

symmetry a relation $R \subseteq A \times A$ is symmetric if and only if for all $a, b \in A$ if $(a, b) \in R$ then $(b, a) \in R$.

equivalence relation a relation is equivalence relation if and only if it is reflexive, symmetric and transitive.

- $E(G) = E(G_1) \cup \cdots E(G_k)$, and
- each G_i has an Eulerian trail.

[*Hint*: Consider the graph when you add k edges to form an Eulerian graph.]

Note that two subgraphs G_1 and G_2 of G are edge-disjoint whenever $E(G_1) \cap E(G_2) = \emptyset$.

13. Let G be a k-regular bipartite graph with $k \geq 2$. Show that G has no cut edge (or bridge).

14. For $n \geq 2$ show that the complete bipartite graph $K_{n,n}$ is a Hamiltonian graph.

15. Determine all $m, n \in \mathbb{N}$ such that the complete bipartite graph $K_{m,n}$ is Hamiltonian.

16. Let G be a connected graph, let T_1 and T_2 be (the edge sets of) two spanning trees of G, and let $e \in T_1 \setminus T_2$. Show that:

    (a) there exists $f \in T_2 \setminus T_1$ such that $(T_1 \setminus \{e\}) \cup \{f\}$ is a spanning tree of G, and

    (b) there exists $f \in T_2 \setminus T_1$ such that $(T_2 \setminus \{f\}) \cup \{e\}$ is a spanning tree of G.

    (Each of these two properties is called the *Tree Exchange Property*.)

17. A graph in which every vertex has even degree is called an *even graph*. The complement $E \setminus T$ of a spanning tree T is called a *cotree*.

    Show that every cotree of a connected graph G is contained in a unique even subgraph of the graph.

9.5 Exercises #5

1. A *permutation* on a set $\{1, \ldots, k\}$ is a one-to-one, onto function on this set. When p is a permutation, p^t means the composition of p with itself t times. Let

$$\textit{PERM-POWER} = \{\langle p, q, t\rangle \mid p = q^t \text{ where } p \text{ and } q \text{ are permutations on } \{1, \ldots, k\} \text{ and } t \text{ is a binary integer}\}$$

   Show that *PERM-POWER* $\in$ P. (Note that the most obvious algorithm does not run within polynomial time. Hint: First try it where t is a power of 2.)

2. Show that NP is closed under the Kleene star operation. That is, if $L \in$ NP, then $L^* \in$ NP.

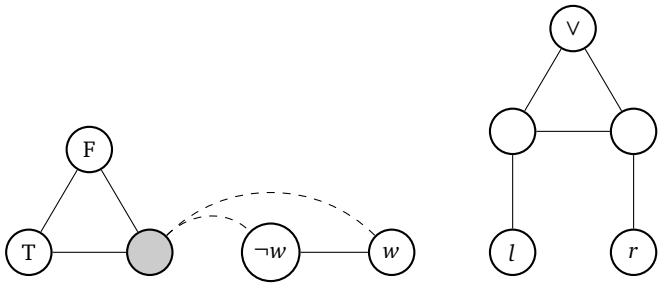
3. Let *DOUBLE-SAT* $= \{\langle \phi \rangle \mid \phi$ has at least two satisfying assignments$\}$. Show that *DOUBLE-SAT* $\in$ NP-complete.

4. A *colouring* of a graph is an assignment of colours to its vertices so that no two adjacent vertices are assigned the same colour. Let

$$\textit{3COLOUR} = \{\langle G\rangle \mid \text{the vertices of } G \text{ can be coloured with three colours such that no two vertices joined by an edge have the same colour.}\}$$

   Show that *3COLOUR* is NP-complete.

   Substantial Hint: The reduction will be of the form *3SAT* $\leq_p$ *3COLOUR*. Use the following subgraphs:



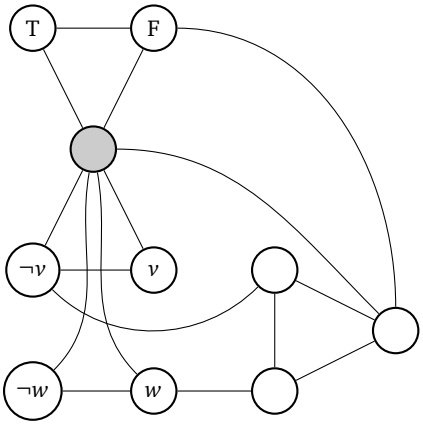
   The three colours correspond to true (T), false (F), and a third one (grey) to help enforce that other nodes are coloured T or F.

   Each variable is modelled using the two nodes marked w and $\neg w$.

   The first graph is used to enforce that w and $\neg w$ are assigned values T and F or F and T.

   The second graph is used to denote binary logical OR between the two legs l and r, which may be either variables or other the roots ($\vee$) of other logical ORs. To ensure that an OR is coloured T, every root $\vee$-node is connected to F and N.

   For example, the following graph corresponds to the encoding of $w \vee \neg v$:



5. Let $EQ_{\mathsf{REX}} = \{\langle R, S\rangle \mid R \text{ and } S \text{ are equivalent regular expressions}\}$.
   Show that $EQ_{\mathsf{REX}} \in$ PSPACE.

Bibliography

[1] Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.

[2] Scott Owens, John Reppy, and Aaron Turon. Regular-expression derivatives re-examined. *Journal of Functional Programming*, 19(02):173–190, 2009.