Discrete Mathematics II : Yet More Mathematics used in Computer Science

by
Bruce Elenbogen
John Baugh

0.1 Dedication

I would like to dedicate this book to my family whose support and love make living and writing books worthwhile. None of the mistakes in this book are the fault of my cats {Panther and Prince }. They suggested many changes, but I only followed their advice to sleep early and often.

0.2 Preface

This book contains some of the mathematics used in computer science. This is by no means intended to be complete overview but is a broad introduction along with the applications that show the power of mathematics . Each chapter is centered on an application which will serve to motivate the mathematical structure constructed. The authors intensely believe in the power of examples in learning and each chapter has at least three examples for each topic as well as plenty of homework problems with solutions. The authors also have observed that traditional textbooks are not widely used by current students and hope their light tone and non-traditional exposition might tempt students to more readily utilize the text.

The book is intended to be used for a one semester undergraduate class and assumes that the students have taken a traditional discrete mathematics class covering : Boolean algebra, proof techniques, logic, combinatorics, graphs and trees as well as one semester of calculus. Although these topics are only referred to tangentially, it is assumed that the students are able to reliably do complex computations as well as understand and write simple proofs.

Contents

List of Figures

List of Tables

Chapter 1

RSA Encryption using Number Theory

No matter how hard you try to keep your secret, it's a universal law that sooner or later it will be discovered. Mettrie L.

1.1 Introduction

The question: What do sending a secret military message in the 1st Century B.C., performing a transaction with your credit card online, and getting money from an ATM all have in common?

The answer: cryptography.

When Gaius Julius Caesar sent secret military messages, he encrypted, or in layman's terms, scrambled, the messages so that they were not intelligible to anyone without the key information for unlocking the encrypted message.

When you perform a transaction on–line (at least with any reputable company), your private information (such as your credit card number) should be encrypted so that eavesdropping third parties should not be able to steal it.

When you withdrawal money from your checking account using an ATM, your account information should only be sent in an encrypted fashion over the network.

Cryptography is one of the most powerful practical mathematical tools in human history, and we begin our adventure into this topic with symmetric encryption and will then move on to asymmetric cryptosystems, also known as public key cryptosystems. Specifically, this chapter will explore the Rivest, Shamir, Adleman (RSA) public key cryptosystem as well as the mathematics on which it is based.

1.2 Symmetric Cryptography

Before we delve into more complicated topics, we need to get some terminology under our belts. We will define and explain several concepts so that we are all on the same page, so to speak, in our understanding of cryptographic primitives.

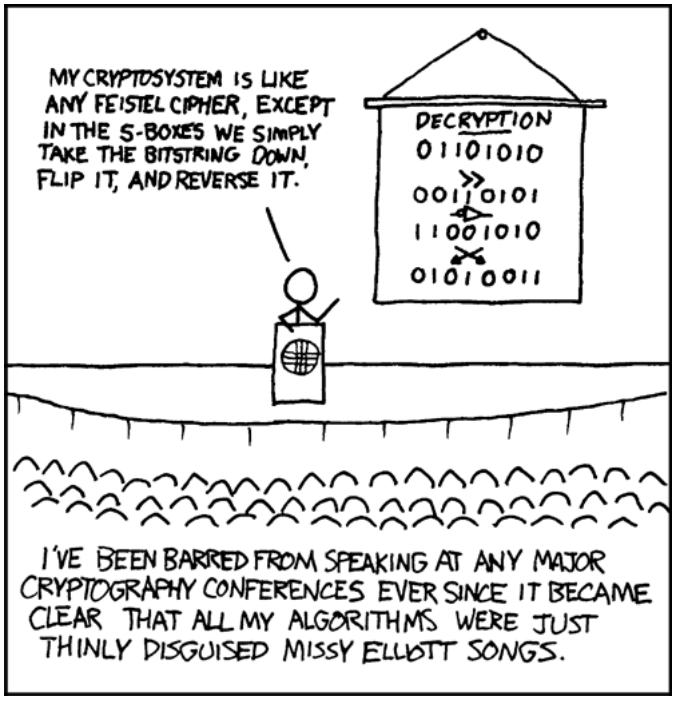


Figure 1.1: Cryptography , XKCD : A webcomic of romance, sarcasm, math and langauge.

In a general sense, cryptography is the study and practical application of techniques used to render information unintelligible to anyone but the intended recipients of that information. In other words, cryptography is one of the techniques (and by far, the most widely used technique) used to conceal information from everyone except the individual(s) who is supposed to get the information.

Another term used often in cryptography is the word key. Keys are used in both symmetric (secret key) and asymmetric (public key) cryptosystems. A key is simply some information that is used as input, along with the message to either encrypt or decrypt the message.

When an original message (called the plaintext) is encrypted, it is transformed using the key via the encryption algorithm. The resulting "scrambled" message is called the *ciphertext*. The sender will send this encrypted message, which appears as a bunch of garbled information, to the intended recipient to be decrypted, that is, to be transformed from the ciphertext back into the original plaintext message.

We observe the processes of encryption and decryption in the example in Figure 1.2. As you can see, the original word, *hello* is input into the encryption algorithm, along with the key. The ciphertext result is *ifmmp*. When the recipient receives the ciphertext, he runs the ciphertext, along with the key through the decryption algorithm, and the result is the original plaintext message. The above is an example of a symmetric cryptosystem. If you spent a few minutes looking at it, you could probably figure out both the encryption and decryption algorithms, along with the key.

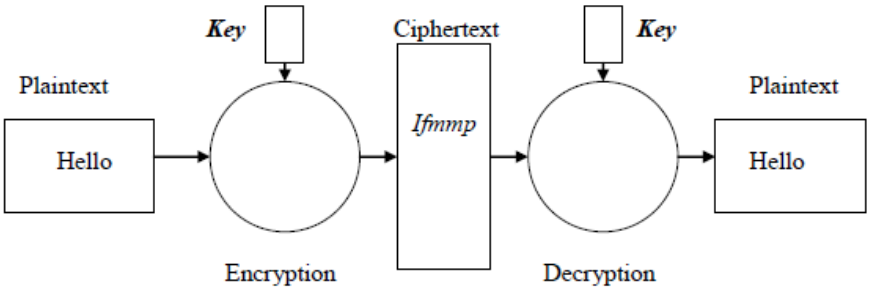


Figure 1.2: Data flow of encryption and decryption of *Hello*

The first type of cryptography we will examine is symmetric cryptography. In symmetric cryptography, as the name implies, the decryption key and the encryption key are identical and shared by both parties (the sender and receiver) .

For a simple example, let's consider the Caesar cipher alluded to in the introduction. When Caesar wanted to send a message to one of his generals on the battlefield, he would perform a left-shift of the ciphertext alphabet thusly:

Plaintext	a	b	c	d	e	f	g	h	...	z
Ciphertext	*d*	*e*	*f*	*g*	*h*	*i*	*j*	*k*	...	*c*

The Caesar cipher is one of the simplest examples of a substitution cipher, so-called because a character in plaintext is simply substituted with a different character.

If we want to encrypt the message, hello, we would simply use the table to perform the encryption:

Plaintext	h	e	l	l	o
Ciphertext	*k*	*h*	*o*	*o*	*r*

The reverse is performed when the recipient receives the ciphertext, *khoor*. He simply looks at each of the characters in the ciphertext row, and finds the plaintext equivalent in the plaintext row.

Example 1.2.1. Caesar cipher example:

For a thorough example of what is done when producing a Caesar cipher, consider that we want to send the message, "We are at war with Greece" using a Caesar cipher with a left shift of 4.

We first line up two rows of the alphabet: a plaintext row and a ciphertext row, thusly

Plaintext	a	b	c	d	e	f	g	h	...	z
Ciphertext	*a*	*b*	*c*	*d*	*e*	*f*	*g*	*h*	...	*z*

At this point, there are no shifts, and thus the plaintext is equal to the ciphertext. Next, let us observe what happens when we shift the entire ciphertext alphabet to the left by 1.

Plaintext	a	b	c	d	e	f	g	h	...	z
Ciphertext	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	...	*a*

Left Shift of 1

We now see that the b was shifted to the left, where the a was in the original alphabet, the c shifted into the b position, etc. At the end, we see that a z in plaintext is an a in ciphertext. This is due to the wraparound that occurred.

Now, let us see the next shift tables:

Plaintext	a	b	c	d	e	f	g	h	...	z
Ciphertext	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*	...	*b*

Left Shift of 2

Plaintext	a	b	c	d	e	f	g	h	...	z
Ciphertext	*d*	*e*	*f*	*g*	*h*	*i*	*j*	*k*	...	*c*

Left Shift of 3

Plaintext	a	b	c	d	e	f	g	h	...	z
Ciphertext	*e*	*f*	*g*	*h*	*i*	*j*	*k*	*l*	...	*d*

Left Shift of 4

Now, we want to translate our message, “We are at war with Greece” into the ciphertext alphabet.

Plaintext	***Ciphertext***
w	*a*
e	*i*
a	*e*
r	*v*
e	*i*
a	*e*
t	*x*
w	*a*
a	*e*
r	*v*
w	*a*
i	*m*
t	*c*
h	*l*
g	*k*
r	*v*
e	*i*
e	*i*
c	*g*
e	*i*

The plaintext has been translated into: *ai evi ex aev amxl kviigi*

And typically, this would be sent without spaces as: *aieviexaevamxlkviigi*

Thus, for this Caesar cipher, the key value is "left shift 4." When the receiver of the message, knowing the key value, wishes to decrypt the message, he would simply reconstruct the table and work from the ciphertext to the plaintext. ■

Example 1.2.2. To encode the message " there is no spoon" with a right shift of 4, the shift table would be :

Plaintext	a	b	c	d	e	f	g	h	...	z
Ciphertext	*w*	*x*	*y*	*z*	*a*	*b*	*c*	*d*	...	*v*

Right Shift of 4

So the substitution table would become :

Plaintext	***Ciphertext***
t	p
h	d
e	a
r	n
e	a
i	e
s	o
n	j
o	k
s	o
p	l
o	k
o	k
n	j

The encrypted message would then become : *pdanaeojkolkkj* ■

In general, the letter substitution of any type is not an effective means of encryption as the use of frequency tables can be used to decode the message. As anyone who has studied televisions game of Wheel of Fortune can tell you, the English letter relative frequency sequence is approximately "ETAONRISHDLFCMUGYPWBVKXJQZ". So in any sufficiently long message a count of the frequency of letter usage would help identify the letters. If the single letter frequency is insufficient to decrypt the message, then a study of common pairs of letters (letters that appear in English words adjacent to each other) can further help decode a message. The most frequent letter pairs are "TH HE AN RE ER IN ON AT ND ST ES EN OF TE ED OR TI HI AS TO" It is therefore imperative to further scramble the message.

Another technique used in symmetric cryptography is a transposition. As the name suggests, characters within a message are transposed within the message. In other words, the characters within the message are simply exchanged with one another.

The simplest of these techniques is the *rail fence technique*, which is demonstrated as follows:

Plaintext: meet me in the lobby

If we wanted to encrypt this with a 2 rail cipher, we simply put the message in a table with 2 rows (rails) in column major order (we fill in one column at a time as opposed to row major (filling in one row at a time)) :

m	e	m	i	t	e	o	b
e	t	e	n	h	l	b	y

The message is then read off in row major order to produce the ciphertext : *memiteobetenhlby.*

Decrypting this simple transposition cipher is quite easy. The number of rows in a rail fence cipher serves as the cryptographic key to encrypting and decrypting messages. To decrypt, we simply apply the algorithm in reverse.

The receiver of the message, knowing that the number of rows (the key) is 2, would create a table with 2 rows, and divide the number of characters in the ciphertext message (16) by the number of rows (2), to obtain the number of columns, 8. They would then construct a 2 x 8 table and begin filling characters from the ciphertext into the table. The result would be the original table used to create the ciphertext.

Alternatively, had we used a 4 rail cipher, then the table would become :

m	m	t	o
e	e	h	b
e	i	e	b
t	n	l	y

This time the encrypted message becomes : mmtoeehbeiebtnly.

The number of columns is typically found as follows:

m = # characters in the message
k = # of rails
c = number of columns, then
c = ceiling(m /k)

If m is not evenly divisible by k, then we add padding to fill out the remainder of the columns. In fact, in practice, padding is added anyway, creating additional columns.

Example 1.2.3. Encode the message "the rebels base is on corellia" using a Caesar cipher shifting left 3 followed by a 3 rail cipher.

Plaintext	a	b	c	d	e	f	g	h	...	z
Ciphertext	*d*	*e*	*f*	*g*	*h*	*i*	*j*	*k*	...	*c*

Left Shift of 3

After the Ceaser cipher the message is : *wkh uheho edvh lv rq fruhoold*. Placing this in a 3 rail table yields:

w	*u*	*h*	*e*	*r*	*r*	*o*
k	*h*	*h*	*d*	*q*	*h*	*l*
h	*e*	*o*	*v*	*f*	*o*	*d*

So the encoded message is *wuherrokhhdqhlheovfod*. ■

Example 1.2.4. Decode the message " *hlbzikxhfpqfksqkdoxd* " which was encoded with a Ceaser cipher shifting right 3 followed by a 5 rail cipher. First we decode the rail cipher by placing the encoded message into a 5 x 4 maxtrix in row major order.

h	l	b	z
i	k	x	h
f	p	q	f
k	x	q	k
d	o	x	d

Now read the message out in column major order. When we read the message in column major order, we see that the encrypted message is *hifkdlkpxobxqqxzhfkd*. Finally, we decode the message by shifting each letter 3 places to the left, which generates, *klingonsareattacking* or, "Klingons are attacking." ■

Although the aforementioned techniques are elementary, they illustrate principles used in encryption today. The Advanced Encryption Standard (AES) is a symmetric key encryption adopted by the US in 2002. The encryption breaks the message up into blocks, and uses the key to permute the bits of the message. It does this in a reversible method, using exclusive OR on the bits of the message along with the key. In addition, the algorithm mixes row shifts and column swapping. The AES has the advantage of being fast to implement while remaining (as of this writing) secure. The key used in AES is usually transmitted using a secure key distribution mechanism. Typically, this mechanism would be a public key (asymmetric) system.

1.3 Asymmetric Cryptography

Up until the 1970s, symmetric cryptography was the primary means by which encryption and decryption were accomplished. In 1976, Whitfield Diffie and Martin Hellman at Stanford University introduced the concept of public key cryptography.

The development of public key cryptography was pursued primarily to answer two major questions:

1. For symmetric encryption (shared key encryption, see section 1.2), how do we *securely* distribute the keys to the parties?

2. How can we be sure that a particular message came from the individual that claims to have sent it?

To explain why the first question is important, consider what happens if the shared secret key is sent to the intended recipients in a fashion that lacks security. If we send the cryptographic keys via an unsecured medium, such as e–mail, then we might as well just forget encrypting and decrypting messages entirely. This is because the e–mail could be eavesdropped upon and the key copied. Thus, this would be the same as if we never even encrypted the message. All confidentiality would be lost.

For the second question, Diffie and Hellman introduced the idea of a *digital signature*. If cryptography were widely deployed, then there should be a way to ensure that a message satisfied *origin integrity*, that is, that it indeed came from the stated sender. So, there should be some sort of digital equivalent of a signature on a legal document, to prove that the origin of the message.

To address these issues, *public key* cryptography introduces a radically new concept. As the name suggests, there is a public key involved with encryption, decryption, and as we shall see, the generation of digital signatures.

As with symmetric key cryptography, public key cryptography involves keeping a private key. However, in public key systems, two keys are generated. One is used for encryption, while the other is used for decryption. One of the keys is kept private (the private key), and the other is made publicly known (the public key).

A public key system utilizes six elements in order to function. They are as follows:

1) Plaintext: As before, this is the message before it is encrypted.

2) Ciphertext: This is the message after it has been encrypted.

3) Public Key: This is the publicly displayed key.

4) Private Key: This is the key that an individual keeps private, but is related to their public key.

5) Encryption Cipher: This is the algorithm that performs the encryption on the plaintext, converting it to ciphertext.

6) Decryption Cipher: This is the algorithm that performs the decryption on the ciphertext, converting it back to the plaintext.

Let us consider two individuals, Alice and Bob (A and B). When Bob wishes to send a confidential message to Alice, he uses Alice′s public key to encrypt the message. Bob knows that only Alice will have the matching private key to decrypt the message.

To provide authentication, that is, to prove to Alice that the message came from Bob, Bob can use his private key with the encryption algorithm, and then the message can be decrypted using his public key. The thought process behind authentication is as follows: If a message can be decrypted using Bob's public key, then it must have been encrypted using his private key. The entire encrypted message serves as a digital signature, because Bob has essentially put his verifiable signature on the message by saying"See, only I, Bob, could have encrypted this message, because only my private key pairs with my public key."

So, in a case where Bob may wish to broadcast a message, he would simply encrypt it using his private key, and send the message out. In the case of broadcasting messages, confidentiality is not a major concern. This is obvious since the message is sent out for everyone to read. The only concern is from the recipients perspective. Their question is, "How do we know this came from Bob?" All of the recipients can verify that it came from Bob by using his public key. If it decrypts the message, then the message obviously came from Bob.

In a situation where Bob intends on only sending a message to Alice, and he wants Alice to be able to verify that he is the one that sent the message, he performs encryption twice. Bob would first use his private key to encrypt the message, providing a digital signature, just as if he

were going to broadcast the message. Next, he would take the encrypted message and encrypt it again using Alice's public key. This would provide confidentiality so that only Alice could read the message.

Let us describe this situation in a more mathematical fashion. Note that since both public and private begin with P, we use the second letter, U for Public and R for Private. K stands for "key" since we are describing different keys used in this process.

KU_a = Alice's Public Key
KU_b = Bob's Public Key
KR_a =Alice's Private Key
KR_b =Bob's Private Key

E with any subscript refers to the encryption cipher, using the key described in the subscript Likewise, D refers to the decryption cipher, using the key described in the subscript.
M = original message
N = encrypted message
Bob will first apply his private key and the encryption algorithm (to produce a digital signature), then he will use Alices public key to encrypt it (to provide confidentiality), as follows:

$$N = E_{KU_a}[E_{KR_b}(M)]$$

To recap, we see that in the brackets, Bob used his private key

$$KR_b$$

with the encryption algorithm on the original message M. Then, the result was encrypted using Alices public key, KU_a.

When Alice receives the message, she will perform the following to obtain the original message:

$$M = D_{KU_b}[D_{KR_a}(N)]$$

It should be noted that in practice, the entire message is not used to produce a digital signature, but only a portion of the message is used. Digital signatures also provide a *non–repudiation* mechanism which means that Alice can't send a signed message to Bob and then claim later she didn't send it. We now examine the mathematics behind one type of public key encryption.

1.4 Number Theory for RSA Cryptography

In order to understand the mathematics behind RSA cryptography, we need to understand some fundamental number theory. Number theory is a branch of mathematics concerned with the properties of numbers, with a specific focus on the properties of integers. This section is divided into subsections so that we can organize our discussion better. We will begin with a basic introduction to prime numbers, then introduce the concept of congruence relations, then move on into Fermat's Little Theorem, and then the Euclidean algorithm.

1.4.1 Prime Numbers

One of the major topics of discussion within number theory is the subject of prime numbers. Prime numbers have many interesting properties that are of interest in various fields, and specifically within cryptography.

Given an integer p, with $p > 1$, we say that p is prime if and only if it is divisible by p and 1. That is, it is only divisible by itself (or its negative counterpart) and 1 (or -1).

The integers 3, 5, 7, 11 and 101 are all prime numbers.

Another concept that will be important to our discussion is that of *co–primality*. Two integers are said to be *co–prime,* or *relatively prime* if 1 is the only common factor they share. In order to be co–prime, the integers need not be prime.

For example, 9 and 34 are co–prime, despite the fact that neither is prime. The numbers 9 and 34 are co–prime due to the fact that they have no common factors other than 1 .

1.4.2 Congruence Relations

The concept of congruence exists in different subfields of mathematics. In elementary geometry, for example, we might say that two angles are congruent if their measures are the same. Geometry instructors typically take great care to distinguish congruence from equality. Congruence indicates equivalence of magnitude, or size.

In number theory, similarly, congruence indicates equivalence of magnitude. Specifically, we are concerned with the equivalence of magnitude of *residue*, that is, the *remainder* after division.

Definition 1.4.1. Given a number $c \neq 0$, we say $a \equiv b \bmod c$ which is read, "a is congruent to b modulo c" iff $(a - b)/c$ is an integer. In other words, if c evenly divides both a and b, then a is congruent to b modulo c. Equivalently, a and b produce the same remainder when divided by c.

Example 1.4.1. Is $9 \equiv 6 \bmod 2$?

If we use the property that $(a - b)/c$ is an integer.

(9 - 6)/2 = (3)/2 = 1.5

So, 9 is not congruent to 6 modulo 2.

Another way of determining the veracity of the above is to verify that 9 and 6 have the same remainder when divided by 2. 9 mod 2 = 1, since 9/2 = 4 with a remainder of 1

6 mod 2 = 0, since 6/2 = 3, with a remainder of 0.

Since their remainders are not equal, we can say that 9 is not congruent to 6 modulo 2. ■

Example 1.4.2. Is $9 \equiv 6 \bmod 3$?

Using the first technique, we calculate the following: (9 - 6) / 3 = (3) / 3 = 1

Since 1 is an integer (that is, 9-6 evenly divides 3), then 9 is congruent to 6 modulo 3.

Using our second technique, we calculate their remainders independently, modulo 3.
9 mod 3 = 0, since 9/3 = 3, with a remainder of 0
6 mod 3 = 0, since 6/3 = 2, with a remainder of 0
Since their remainders are equal, we can say that 9 is congruent to 6 modulo 3. ■

1.4.3 Modular Arithmetic

One of the most useful properties of modular arithmetic can be used to simplify modular multiplication.

Theorem 1.4.1.

$$(a \times b)(\text{mod} c) = [(a(\text{mod} c) \times b(\text{mod} c))]\,(\text{mod} c)$$

Proof: Let $r_a = a(\text{ mod } c)$ and let $r_b = b(\text{ mod } c)$. Then by the definition of mod, we know that :

$r_a = a - n_a c$ and $r_b = b - n_b c$
for some integers n_a and n_b.
Then

$$[a(\text{mod} c) \times b(\text{mod} c)]\,(\text{mod} c) =$$

$$[(a - n_a c) \times (b - n_b c)]\,(\text{mod} c) =$$

$$\left[(ab - bn_a c - an_b c + n_a n_b c^2)\right]\,(\text{mod} c)$$

Now the last 3 terms are all multiples of c, so they do not affect the total.
Hence the above is equal to $(ab)(\text{mod } c)$. ■

Example 1.4.3. Compute [64 × 9 × 10 × 5] mod 7
[64 × 9 × 10 × 5] mod 7 =
[8 mod 7 × 8 mod 7 ×9 mod 7 × 10 mod 7 × 5 mod 7] mod 7 =
[1×1× 2× 3× 5] mod 7 = 2 ■

Example 1.4.4. (7× 7) (mod 6) = 49 (mod 6) = 1
while
(7 (mod 6) × 7 (mod 6)) (mod 6)=1 × 1 (mod 6) = 1 ■

Example 1.4.5. In a more complicated example we will compute 2^{50} (mod 7). In this example we want to simplify the computation by finding an exponent e, so that 2^e mod 7 is 1. Here we see that 3 works well. Now 50/3 = 16 with 2 remaining. Thus:

$$2^{50}(\text{mod} 7) = (2^3(\text{mod} 7))^{16} \times (2^2(\text{mod} 7))(\text{mod} 7) = 1^{16} \times (4(\text{mod} 7))(\text{mod} 7) = 4$$

■

Example 1.4.6. In a manner similar to the above we want to simplify the computation by rewriting 8^{1000} mod 5. Since computing powers of 8 soon becomes complex it is easier to begin by factoring 8 into 2^3 so that $8^{1000} = (2^3)^{1000} = 2^{3000}$. Now we can use that 2^4 mod 5 = 1.

8^{1000} (mod 5) = (2^{3000}) (mod 5) = $[(2^4(\text{ mod } 5))^{750}\ ]$ (mod 5) = $[(16\ (\text{mod } 5))^{750}\ ]$ (mod 5) = 1. ■

1.4.4 Fermat's Little Theorem

Pierre de Fermat is not only famous for his Last Theorem, but for his many other contributions to number theory. Fermat's Little theorem is the key to understanding RSA encryption.

Theorem 1.4.2. *Fermat's Little Theorem*
Given a prime number p, and any integer a, then $(a^p - a)$ is evenly divisible by p. In congruence notation, that is: $a^p \equiv a \mod p$

Proof: Consider the first $p-1$ positive multiples of a : $a, 2a, 3a, ..., (p-1)a$. Note that if $ra \mod p \equiv sa \mod p$, then $r = s \mod p$. Now if we take mod p of each term of our sequence and p is prime, we will end up with a value in the range from 1 to $p-1$. Since each of the terms on the left are distinct multiples of a and p is prime then the result (left hand side) is $p-1$ distinct values in the range from 1 to $p-1$. Thus multiplying all the terms of the sequence we find that

$$[a \times 2a \times 3a...(p-1)a] \mod p = [1 \times 2 \times 3...p-1] \mod p.$$

Dividing both sides by $(p-1)!$ yields:

$$a^{(p-1)} \mod p = 1 \mod p.$$

Corollary. *If p is a prime number, and a is an integer and is co–prime to p then $(a^{p-1} - 1)$ is evenly divisible by p. In congruence notation, that is: $a^{p-1} \equiv 1 \mod p$*

Let us select two numbers that are relatively prime to one another so that we may demonstrate both variations of this theorem. In this first example, we will explore the first variation of the theorem, namely, $a^p \equiv a \mod p$. In the second example, we will explore the second variation of the theorem, which applies to values of a and p that are co–prime.

To apply the theorem, p must be a prime number. To demonstrate the corollary of Fermat's Little Theorem, it is necessary for one of the numbers, p, to be prime, with the additional stipulation that the other integer, a, be co–prime (relatively prime) to p.

Example 1.4.7. We select $p = 3$ and $a = 2$;
3 and 2 are co–prime, since neither share a common factor except 1.
To illustrate Fermat's little theorem we compute:
$a^p \equiv a \mod p$
$2^3 \equiv 2 \mod 3$
$8 \equiv 2 \mod 3$
At this point, if you recall, we can perform the calculation in two different ways. We check if (8 - 2) is evenly divisible by 3.
(8 - 2) / 3 = (6) / 3 = 2
Since the result is 2, an integer, the congruence, holds. ■

For our second example, we utilize the corollary of Fermat's Little Theorem, namely, $a^{p-1} \equiv 1 \mod p$, in which a must be relatively prime to p. We will use the same values for a and p that we used in the first example.

Example 1.4.8. Consider $a = 3$ and $p = 7$.

$a^{p-1} \equiv 1 \bmod p$

$3^{(7-1)} \bmod 7 =$

$3^6 \bmod 7 =$

$(3^2)^3 \bmod 7 =$

$(3^2 \bmod 7)^3 \bmod 7 =$

$(9 \bmod 7)^3 \bmod 7 =$

$2^3 \bmod 7 = 8 \bmod 7 = 1$

Thus, the corollary holds. ■

Example 1.4.9. Let $a = 20$, $n = 23$. Then $a^{n-1} \pmod n = 20^{22} \bmod 23) =$

$[(20^2 \pmod{23})^{11}] \pmod{23} =$

$[9^{11}\ ] \pmod{23} =$

$[(9^2 \pmod{23})^5 \times 9] \pmod{23} =$

$[(12)^5 \times 9] \pmod{23} =$

$[(12 \pmod{23})^2 \times (12 \times 9) \pmod{23}] \pmod{23} =$

$[62 \times 16] \pmod{23} =$

$[576] \pmod{23} = 1$.

■

1.5 RSA Public Key Cryptosystem

While Diffie and Hellman developed the concepts and requirements of a public key cryptosystem, they did not specify algorithms that met the requirements of a public key cryptosystem. Shortly after the publication of their paper, the RSA algorithm was developed in 1977 by Ron Rivest, Adi Shamir, and Len Adleman of the Massachusetts Institute of Technology at a Passover Seder. Their widely cited paper on the topic was published in 1978 [Rivest78].

RSA is a block cipher, meaning that encryption is performed on blocks of plaintext. RSA, like most public key cryptosystems consists of three distinct steps during its usage:

1) Key Generation

2) Encryption

3) Decryption

1.5.1 Overview of RSA

For generation of keys, Alice (the sender of the secure message) first selects two prime numbers, p and q, with $p \neq q$. In practice, these will be very large prime numbers, but for our purposes we will keep them relatively small. Second, Alice calculates: $n = p \times q$ Third, she calculates: $t = \text{lcm}[(p-1), (q-1)]$ Fourth, Alice selects a prime $e < t$ such that e is relatively prime to t. In practice e is usually chosen as another prime number. Finally, Alice calculates d such that: $demodt \equiv 1.$

The public key and private key are constructed as follows:

To encode the message M, into the ciphertext C we compute $C = M^e \bmod n$ while to decode the C back to the original message M, we merely compute $M = C^d \bmod n$.

Thus we can think of the public key (KU) and the private key (KR) as $KU = \{e, n\}$ and $KR = \{d, n\}$.

Example 1.5.1. First choose two prime numbers. Lets choose $p = 5$ and $q = 11$.

Part of the key is n where $n = p \times q$. In this example, $n = 5 \times 11 = 55$

Next, we compute $t = \text{lcm}\ [(p-1)(q\text{ - }1)]$: $t = \text{lcm}[(5\text{-}1),(11\text{-}1)] = \text{lcm}[4,\ 10] = 20$

Now we select an integer, e, such that e is co–prime to t. Arbitrarily choose 7, since 7 is co–prime to 20.

Finally we calculate d: $d \equiv e^{-1} \bmod 20$.

Here d = 3 since $7 \times 3 = 21$ and $21 \bmod 20 = 1$.

So if the message M is 12, then using the encryption key {7 and 55 }, the encrypted message C would be

$C = 12^7 \bmod 55 = 35831808 \bmod 55 = 23$.

To recreate the original message M we use the decryption key {3 and 55},

$M = 23^3 \bmod 55 = 12167 \bmod 55 = 12$. ■

1.5.2 Extended Euclidean Algorithm

The method for computing the keys requires calculating the (LCM) to compute t, as well as the inverse of e mod t. Both of these tasks can be accomplished with the Euclidean Algorithm. Recall that the Euclidean Algorithm is an efficient method to compute the greatest common divisor GCD.

Theorem 1.5.1. *The Euclidean Algorithm*

The GCD(m, n) for positive integers m and n where $0 < m < n$, is equal to the GCD($nmodm, m$). When $nmodm$ reaches 0, the algorithm ends and the result is m.

Example 1.5.2. The GCD(20,42) = GCD(42 % 20, 20) = GCD(20 % 2, 2) = GCD(0, 2) = 2 ■

The gcd is related to the LCM through the relation : $\text{LCM}(m, n) = \frac{mn}{\text{GCD}(m,n)}$

This can be seen by factoring m and n. The LCM(m,n) is the product of the unique factors in either m or n. The GCD(m,n) is the product of the factors in common to m and n, while mn is the product of every factor in either m or n .

Example 1.5.3. Consider $m = 12$ and $n = 42$. Here the factors of m are 2, 2 and 3 while the factors of n are 2, 3, and 7. Then $mn = 504$ The GCD(m, n) is 6 , while the LCM (m, n) is $2 \times 2 \times 3 \times 7 = 84$. So the above equation implies that $84 = 504/6$. ■

The Euclidean algorithm is just a procedure that computes the GCD (m, n) for positive integers m and n by exploiting the property that $\text{GCD}(m, n) = \text{GCD}(n, m \bmod n)$.

```
int gcd(int m, int n)
// assumes m, n > 0
{
  if (m < n)
    return gcd(n, m);
  else if (n == 0)
    return m;
  else
    return gcd (n, m%n);
}
```

Example 1.5.4. Hence to compute the GCD(60, 42) . I will use the C++ notation % to represent the mod function.

GCD(60,42) = GCD(42, 60 % 42) = GCD(42, 18) = GCD(18, 42%18) = GCD(18,6) = GCD(6, 6%18) = GCD(6,0) = 6. ■

Example 1.5.5. Hence to compute the LCM(144, 64) we find LCM(144, 64) = 144× 64 / GCD(144, 64) Using the Euclidean Algorithm : GCD(144,64) = GCD(64, 16) = GCD(16, 0) = 16.

Hence the LCM(144, 64) = 9216/16 = 576. ■

The Euclidean algorithm, with only slight modifications, can be used to find inverse of m mod n by keeping track of the all the intermediate calculations . This is called the *Extended Euclidean Algorithm.* The inputs to the Extended Euclidean Algorithm are two positive integers m and n, while the output is $k = \text{GCD}(m, n)$ along with the integers u and v such that $um + vn = k$. If m^{-1} mod n exists then the GCD(m,n) = 1. So if we have that

$$um + vn = 1,$$

then if we take the mod n of both sides, and using vn mod $n = 0$, while 1 mod n is 1 for all positive n, um mod $n = 1$ mod $n = 1$. Hence u is the inverse of m mod n. In a similar fashion if we take the mod m of both sides of $um + vn = 1$ we find vn mod $m = 1$, and we find v is the inverse of n mod m.

The best way to perform the extended Euclidean algorithm by hand is to just use the Euclidean algorithm, and use back substitution. This is most easily explained with an example.

Example 1.5.6. Find 11^{-1} mod 27 and 27^{-1} mod 11. The steps of the Euclidean algorithm are:

The gcd(27, 11) = gcd(11, 27 % 11) = gcd(11, 5) .

gcd(11,5) = gcd(5, 11 % 5) = gcd(5, 1) = 1.

Now if we rewrite the steps solving for the remainders and keeping track of the factors, we find :

$$5 = 27 - 2(11) \tag{1.1}$$

$$1 = 11 - 2(5) \tag{1.2}$$

Now all that remains to convert all the colored factors on the right hand side to terms using the original two factors, 11 and 27. The first equation is fine. In the second equation we can replace 5

by equation 1.1.

$$1 = 11 - 2[27 - 2(11)]$$

combining the coefficients of 11 and the coefficients of 27 we find

$$1 = -2(27) + 5(11) \tag{1.3}$$

Hence the inverse of 11 mod 27 is 5 while the inverse of 27 mod 11 is -2 mod 11 = 9.

We can easily check this 5(11) mod 27 = 55 mod 27 = 1 while 9(27) mod 11 = 243 mod 11 = 1 ■

Example 1.5.7. Here is a more complex example. Find 15^{-1} mod 26 and 26^{-1} mod 15. The steps of the Euclidean algorithm are:

The gcd(26, 15) = gcd(15, 26 % 15) = gcd(15, 11) .

gcd(15,11) = gcd(11, 15 % 11) = gcd(11, 4).

gcd(11, 4) = gcd(4, 11 % 4) = gcd (4, 3).

gcd(4, 3) = gcd(3, 4 % 3) = gcd(3, 1) = 1.

Now if we rewrite the steps solving for the remainders and keeping track of the factors, we find :

$$11 = 26 - 1(15) \tag{1.4}$$

$$4 = 15 - 1(11) \tag{1.5}$$

$$3 = 11 - 2(4) \tag{1.6}$$

$$1 = 4 - 1(3) \tag{1.7}$$

Now all that remains to convert all the colored factors on the right hand side to terms using the original two factors, 15 and 26. The first equation is fine. In the second equation we can replace 11 by equation 1.4.

$$4 = 15 - 1[26 - 1(15)]$$

combining the coefficients of 15 and the coefficients of 26 we find

$$4 = -1(26) + 2(15) \tag{1.8}$$

We can then use equation 1.8 and equation 1.4 to convert the equation for 3.

3 = [26 - 1(15)] - 2 [(-1(26) + 2(15))]

combining the coefficients of 15 and the coefficiets of 26 we find

$$3 = 3(26) - 5(15). \tag{1.9}$$

Finally we can substitute equation 1.9 and 1.8 into 1.7.

$$1 = [-1(26) + 2(15)] - 1[3(26) - 5(15)]$$

$$1 = -4(26) + 7(15)$$

Hence the inverse of 15 mod 26 is 7 while the inverse of 26 mod 15 is -4 mod 15 = 11.

We can easily check this 15(7) mod 26 = 1 while 11(26) mod 15 = 1 ■

The substitution method however is not the best way to compute this on a computer. The following code is similar to the substitution method used in the non-recursive euclidean algorithm

```
void ExtendedEuclidean(int a, int b, int & d, int & u, int & v)
{ // assumes a, b > 0  returns u, v, d such that ua +vb = d
   int c, x, y, tmp, q;
   c = min(a,b) ;     d = max(a,b);
   x = 1;   y = 0;   u = 0;   v = 1;
   while (d > 0 && c > 0) {
    q = d/c;
    tmp = d;
    d = c;
    c = tmp % c;
    tmp = x;
    x = u - q*x;
    u = tmp;
    tmp = y;
    y = v - q*y;
    v = tmp;
   }
}
```

The next example puts all the mathematics together to show how we can encode and decode a small message 101.

Example 1.5.8. Let $p = 19$ and $q = 37$. Then $n = 703$, $t = (18 \times 36)/\gcd(18,36) = 36$. If we choose $e = 5$, then the extended Euclidean algorithm can be used to compute $d = 29$. So if the message $m = 101$, then the encoded message C is $(101)^5 \pmod{703}$

$C = [((101)^2 \pmod{703})^2 \times 101] \pmod{703} = [359^2 \times 101] \pmod{703} = 233$

We should be able to recover the message as follows:

$M = (233)^{29} \pmod{703} = [((233)^2 \pmod{703})^{14} \times 233] \pmod{703}$

$= (158^{14} \times 233) \pmod{703} = [((158^2) \pmod{703})^7 \times 233] \pmod{703}$

$= (359^{29} \times 233) \pmod{703} = [((359^2) \pmod{703})^3 \times 359 \times 233] \pmod{703}$

$= (232^3 \times 359 \times 233) \pmod{703} = [482 \times 359 \times 233] \pmod{703} = 101$ ■

1.5.3 Chinese Remainder Theorem, and Garner's Formula

With the addition of the Chinese Remainder Theorem, we can now show why the RSA method works. The Chinese Remainder Theorem is credited the Chinese mathematician Sun Tsu, who lived about 300 AC. The theorem implies that the given an integer $x < pq$ where p and q are co-prime, then x can be recovered from its remainders with respect to p and q , respectively.

Example 1.5.9. Let p be 10 and let q be 9. Note that p and q are co–prime. Assume that you are given that x mod $p = 2$ while x mod $q = 7$. Then we can determine x uniquely. The remainder mod 10, implies that $x \in \{2, 12, 22, 32, 42, 52, 62, 72, 82\}$ while the remainder mod 9 implies that $x \in \{7, 16, 25, 34, 43, 52, 61, 70, 79, 88\}$. Hence, the unique value of x is 52. ■

In fact, there is a formula for computing the value of x given x mod p and x mod q attributed to Garner.

Theorem 1.5.2. *Given x mod $p = a$ and x mod $q = b$, where p and q are coprime and $x < pq$, then $x = (((\ ab)\ (q-1 \bmod p)) \bmod p)q + b$*

Although, the Chinese Remainder Theorem is quite powerful, the proof of RSA requires only the special case of when $a = b = 1$. It is clear then from Garner's formula that here x is also 1. We can now show that RSA will decode what it encodes.

Theorem 1.5.3. *Given p and q are co–prime and $M < pq$, then if $ed \bmod (lcm\ (p-1, q-1)) = 1$ and $C = M^e \bmod (pq)$ then $M = C^d \bmod (pq)$*

Proof:

$$\begin{aligned} C^d(\text{mod } n) &= [(M^e)^d](\text{mod } n) \\ &= [M^{kt+1}](\text{mod } n), \text{for some integer } k \\ &= [(M^t)^k M](\text{mod } n) \\ &= [(M^t \text{mod } n)^k M](\text{mod } n) \\ &= [1^k M](\text{mod } n) \\ &= M(\text{mod } n). \end{aligned} \tag{1.10}$$

Therefore, the original message is recovered as long as it is shorter than n. Equation 1.10 needs a bit of explanation and is most easily shown using the Chinese Remainder Theorem It is easy to see from Fermat's Little theorem and the definition of t that M^t mod $p = M^{r(p-1)}$ mod p =1 and that M^t mod $q = M^{s(q-1)}$ mod q =1. Since the answer is 1 for both mod p and mod q then the Chinese Remainder Theorem implies that M^t mod pq must be 1.

1.6 Summary

Theoretical Concepts

This chapter covered many concepts from number theory. These included Modular Arithmetic, Fermat's Little Theorem, Euclidean Algorithm, Extended Euclidean Algorithm, Chinese Remainder Theorem, and Garners Theorem. These can be used in many areas of computer science not limited to cryptography but also compression, computer networks and digital signatures.

Applications

This chapter briefly discussed symmetric encryption and RSA encryption. The RSA method as presented above has several pitfalls, some of which are addressed by the implementation(s) used in practice. One such pitfall is if the message is so small that mod n operation does not change the original message. The algorithm used in practice pads such messages for greater security.

If you want to implement RSA, you will need very large (hundreds of digits) prime numbers as well as the ability to do arithmetic on these very large numbers. There is an excellent overview of implementing a library of large number routines in [Knuth81]. There are many free libraries of such routines on the Internet for you to download as well as languages such as Python, which have built in large integer support. One such library is the GNU Multiple Precision Arithmetic Library, the "Bignum Library" at http://gmplib.org/ .

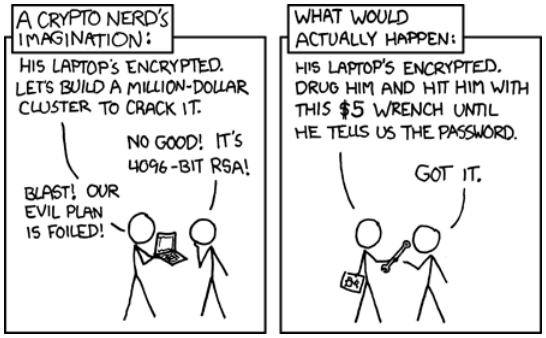


Figure 1.3: Security: XKCD A webcomic of romance, sarcasm, math, and language.

1.6.1 Linear Diophantine Equations

A Diophantine equation is one where the variables and coefficients can only take on integer values. A linear Diophantine equation is one where the variables only appear in linear fashion.

Definition 1.6.1. A ***linear Diophantine equation in two variables*** is an equation of the form

$$ax + by = c$$

where a, b, c, x, y are integers.

First a small theorem to characterize solutions.

Theorem 1.6.1. *The linear Diophantine equation $ax + by = c$ has a solution if and only if the $gcd(a, b)$ divides c.*

Proof Assume (x_0, y_0) is a solution to $ax + by = c$, and let $d = \gcd(a, b)$. Then we know that d is a factor of both a and b and hence d is a factor of $ax_0 + by_0$. Since this is (x_0, y_0) is a solution , we know that d is a factor of c.

Now if d is a factor of c, we have that $c = dc_1$ for some integer c_1. The fundamental theorem of algebra implies that

$$d = ua + vb$$

for some integers u and v. Multiplying this equation by c_1 yields

$$uac_1 + vbc_1 = dc_1 = c$$

Thus the solution to $ax + by = c$ is (uc_1, vc_1). ■

Now that we can determine when we have solutions, it is is time to find the solution.

Theorem 1.6.2. *If $gcd(a, b) = d$ is a factor of c then the Diophantine equation $ax + by = c$ has an infinite number of solutions, $x = x_0 + (b/d)t$ and $y_0 - (a/d)t$ for all integer t and where $x_0 = uc/d$ and $x_0 = vc/d$ and where $d = ua + vb$.*

The u and v are just the inverses found by the extended Euclidean algorithm. Hence the extended Euclidean algorithm can be used to solve linear Diophantine equations which have solutions.

Proof The proof is left to the reader. ■

Example 1.6.1. Find all the solutions to

$$77x + 42y = 35$$

The gcd(77,42) is 7 and 7 is a factor of 35 so there are an infinite number of solutions. The extended Euclidean algorithm yields :

$$7 = 42 - 35$$

$$7 = 42 - (77 - 42)$$

$$7 = (-1) \times 77 + 2 \times 42$$

■

1.7 Exercises

1) Encrypt the message "battle is at dawn" using a Caesar cipher with a left shift of 5.

2) Encrypt the same message as in (1) using a Caesar cipher with a left shift of 9.

3) What is an example of a non–zero left shift that would cause the plaintext alphabet and the ciphertext alphabet to be equal?

4) Encrypt the message "We are attacking from the north" using a rail fence cipher with a row depth of 4.

5) Encrypt the message "We will destroy their forces in three days" using a rail fence cipher with a row depth of 6.

6) Use the properties of mod to compute $2^{1,000,001} \bmod 5$.

7) Use the properties of mod to compute $3^{100,001} \bmod 5$.

8) Use the properties of mod to compute $2^{1,000,001} \bmod 7$.

9) Find the GCD (144, 128)

10) Find the GCD (252, 72)

11) Find the LCM (144, 128)

12) Find the LCM (252, 72)

13) Find the inverse of 5 mod 26 using the Extended Euclidean Algorithm by hand.

14) Find the inverse of 19 mod 26 using the Extended Euclidean Algorithm by hand.

15) Find the inverse of 13 mod 22 using the Extended Euclidean Algorithm by hand.

16) Encode the message 57 using $p = 17$, $q = 11$ and $e = 5$

17) Encode the message 101 using $p = 17$, $q = 11$ and $e = 5$

18) Decode the message 57 using $p = 17$, $q = 11$ and $d = 7$

19) Decode the message 101 using $p = 17$, $q = 11$ and $d = 7$

20) Use the Chinese Remainder Theorem to compute $(101)^{15} \bmod (11 \times 17)$

21) Use the Chinese Remainder Theorem to compute $(57)^{15} \bmod (11 \times 17)$

22) Use Garner's Formula to compute find x where $x \bmod 11 = 7$ and $x \bmod 17 = 5$.

23) Use Garner's Formula to compute find x where $x \bmod 11 = 5$ and $x \bmod 17 = 7$.

1.7.1 Answers to Exercises

1) gfyyqj nx fy ifbs

2) kjccun rb jc mjfw

3) a shift of n*26 for any integer n

4) wrtk mereetif t anrtnhaacgohov

5) wlrernee oic dyretdwe shaistf rylthoies

6) 2

7) 3

8) 4

9) 16

10) 36

11) 1152

12) 504

13) 21

14) 11

15) 17

16) 57

17) 21

18) 73

19) 141

20) 186

21) 54

22) 73

23) 126

1.7.2 Computer Activities

1) Prompt the user for a Linear Diophantine Equation (one with only integer solutions) in two variables of the form $ax + by = c$. Use the extended Euclidean algorithm to find all solutions or report none exist.

2) Prompt the user for two large (up to 100 digits) numbers a and b. Then compute a mod b.

3) Write a function whose input is k the number of digits of n which will generate the numbers p, q, n, e, and d used in RSA encryption.

4) Write a function to encode messages given m, n and e as well as a function to decode messages given c, n and d.

5) Write a function that encrypts and decrypts a message using a Caesar cipher, given two inputs:

   - The message in plaintext

   - The key (i.e. the amount to shift the plaintext alphabet left to produce the ciphertext alphabet)

Bibliography

[1] Diffie, W. and Hellman, M.E. , "New directions in cryptography," *IEEE Trans. Information Theory*,1976 , **IT–22**, 644–654.

[2] Rivest, R., Shamir, A., and Adleman, L., "A Method for Obtaining Digital Signatures and Public–Key Cryptosystems," *Communications of the ACM*, 1978, **Vol. 21 2**, 120–126.

[3] Ferguson,N., and Schneier., *Practical Cryptography*", John Wiley, 2003 .

[4] Knuth,D. , *Semi–numerical Algorithms : The Art of Computer Programming Vol. 2* , Addison Wesley , 1981.

Chapter 2

Decision Theory

Informed decision-making comes from a long tradition of guessing and then blaming others for inadequate results. Scott Adams (creator of Dilbert)

In this chapter we introduce some of the mathematics behind rational decision making. We first consider a branch of mathematics known as game theory. A host of mathematicians are credited with developing game theory, including Emile Borel in 1938 as well as John von Neumann and Oskar Morenstern in 1944. There have been at least eight Nobel prizes awarded for work in game theory including one to John Nash for his work that inspired the Oscar winning movie "A Beautiful Mind". Game theory guides decisions of a rational agent who is competing or cooperating with other rational agents. Other types of mathematics can be used to optimize the design and manufacturing of a product, but it is game theory that helps one choose which product to produce in the face of competing products.

Game theory has many applications in computer science. Besides helping determine which software product to produce for the marketplace, game theory is used to model interactive computations such as those made in a multi-agent system. Additionally, game theoretic analysis can be applied to creating software (such as operating systems) that strives to stay functioning in the face of users who may be (intentionally or unintentionally) trying to compromise them.

We will begin, as always, with terminology and classification of games. Next we will discuss the simplest games (two person, zero sum games) and then move on to more complex and interesting situations.

2.1 Concepts and Terminology

Although, there are many classifications of games, with new designations still being defined, we will start by discussing the traditional classifications which are based on actual games.

1) *Random* and *Non–random* games. Random games involve some unpredictable aspect such as the order of a deck of cards, the result of a dice throw, or the result of a spinner and include : Poker, Monopoly , and Candy Land. Non-random games have pure strategy such as chess, checkers and Stratego.

2) *Perfect knowledge* and *Non–perfect knowledge* games. Perfect knowledge games are games where all parts of the game are known to all the players, such as chess, checkers, and Monopoly. Non-perfect knowledge games involve some hidden aspect such as Stratego, Poker, and Battleship.

3) *n–Player* games. The number of players in a game is usually self–evident. Klondike, Angry Birds, mazes and puzzles have 1 player. Chess, checkers, and Stratego have exactly two players. Cribbage and Bridge are played with four players while Monopoly and poker can have an arbitrary (in theory) number of players. It is clear there cannot be more than 10 poker players with a single deck of 52 cards.

4) *Zero sum* and *non–Zero sum* games. In a zero sum game the value of the game never increases. In poker the total amount of money at the start of the game is equal to the total at the end of the game. In chess the number of pieces only decreases in the course of the game. Monopoly is a non-zero sum game since every time a player passes GO, they collect an additional $200, which is not taken from another player.

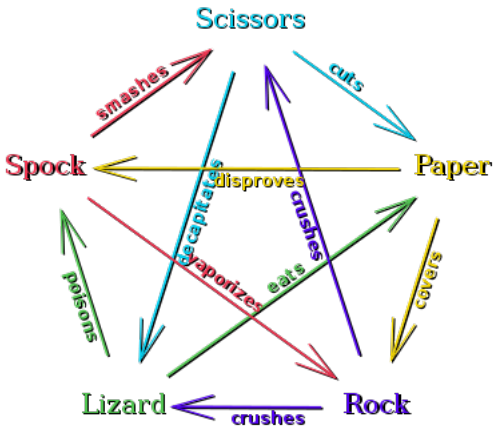
2.2 Two Person, Zero Sum Games with Pure Strategies

Our discussion of games begins with the simplest such games, two person, zero sum games. These are the simplest games we will study. Consider a game where each player makes a single decision to determine the outcome, and that decision is made simultaneously by both players, (such as the classic rock-paper-scissors, pictured above). Consider such a game where $1 is the prize for winning a round. We can represent that game as a matrix with the columns indicating the choices for your opponent r(ock), p(aper) and s(cissors) respectively, while the rows indicate your options. The values in the matrix indicate the amount of money you win, in each situation. Note that the matrix is relative to you only and the amount you win is the the amount the opponent loses. Such a matrix is pictured below. The matrix indicates that no money changes hands if the two players both make the same choice. Additionally it indicates that rock beats scissors, scissors beats, (cuts) paper and that paper beats (covers) rock.

$$\begin{array}{cc} & \textit{Opponent} \\ & \begin{array}{ccc} r & p & s \end{array} \\ \textit{You} \begin{array}{c} r \\ p \\ s \end{array} & \left[\begin{array}{ccc} 0 & -1 & 1 \\ 1 & 0 & -1 \\ -1 & 1 & 0 \end{array}\right] \end{array}$$

Note that the payoffs (values in the matrix) are relative to you alone and that if you win one

dollar, your opponent will lose exactly \$1 and vice – versa. This game is not very interesting since it is intuitively obvious that all choices have the same chance of winning. Studies have shown that if you play the game with people you know multiple times, the game will usually result in a tie. The variant of the game with five choices, (rock, paper, scissors, lizard, Spock) was invented to combat this short coming. (see the figure below courtesy of Wikipedia)



Let us now consider another game where there is clearly a best strategy.

$$\begin{array}{ccc} & & \textit{Opponent} \\ & & \begin{array}{cc} x & y \end{array} \\ \textit{You} & \begin{array}{c} a \\ b \end{array} & \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \end{array}$$

This is a good game for you to play since regardless of the choices, you make money. Since you want to make as much as possible, it is clear that you should play strategy b. It is equally clear that if your opponent wants to lose as little as possible, he/she should play x every time. This pair of strategies (b, x) is called *stable* since if either player changes from the equilibrium, the result will be worse for the player who made the change. If both players have a stable strategy, the game is said to have an *equilibrium strategy* and the *value* of the game matrix at the equilibrium strategy is called the *equilibrium point*.

Example 2.2.1. Consider the following game that arises from politics. This example is from a book on Game Theory by Davis () . Assume in an election year there is a dispute between state X and state Y regarding water rights. Each of the two political parties must decide to either favor X , favor Y or evade the issue entirely. Assume that the parties will announce their decisions on water rights simultaneously. A poll was run before the announcement with the following results, where the values are percentages of the total vote, and where X indicates favor X, Y indicates favor Y and E represents evading the issue entirely.

$$\begin{array}{cc} & \text{Republicans} \\ & \begin{array}{ccc} X & Y & E \end{array} \\ \text{Democrats} \begin{array}{c} X \\ Y \\ E \end{array} & \left[\begin{array}{ccc} 45 & 50 & 40 \\ 60 & 55 & 50 \\ 45 & 55 & 40 \end{array}\right] \end{array}$$

In this case it is clear that the Democrats should favor Y while the Republicans should evade the issue to minimize the effect of the Democrats choice. ■

Example 2.2.2. Consider the following game matrix.

$$\begin{array}{cc} & \textit{Opponent} \\ & \begin{array}{ccc} x & y & z \end{array} \\ \textit{You} \begin{array}{c} a \\ b \\ c \end{array} & \left[\begin{array}{ccc} 1 & 2 & 3 \\ 8 & 9 & 5 \\ 3 & 2 & 1 \end{array}\right] \end{array}$$

In this game you have an optimal strategy choosing b, but your opponent's choice is not so obvious. No column is clearly better for the opponent to minimize their loss. However if the opponent assumes you are going to make a rational choice, then z is clearly best. So in this case the equilibrium is (b, z) and the value of the game is 5. If your opponent makes any other choice you will only take more of his/her money. ■

Example 2.2.3. Here is another example .

$$\begin{array}{cc} & \textit{Opponent} \\ & \begin{array}{ccc} x & y & z \end{array} \\ \textit{You} \begin{array}{c} a \\ b \\ c \end{array} & \left[\begin{array}{ccc} 1 & 2 & 1 \\ 7 & 5 & 4 \\ 8 & 4 & 3 \end{array}\right] \end{array}$$

There is no best strategy for you, however, your opponent will choose z. With the assumption that your opponent will act in their own self interest, it is clear you should choose b and the equilibrium is (b, z) with a value of 4. ■

Example 2.2.4. Consider the following game.

$$\begin{array}{cc} & \textit{Opponent} \\ & \begin{array}{ccc} x & y & z \end{array} \\ \textit{You} \begin{array}{c} a \\ b \\ c \end{array} & \left[\begin{array}{ccc} 8 & 8 & 5 \\ 9 & 2 & 4 \\ 7 & 2 & 1 \end{array}\right] \end{array}$$

In this game, it is not clear what strategy is best for either you or the opponent. However it is quickly apparent that you do not want to play c since it is worse in all cases, and your opponent does not want to play strategy x. With that in mind, the game is reduced to only two possible options for each player.

$$
\begin{array}{cc} & \textit{Opponent} \\ & \begin{array}{ccc} x & y & z \end{array} \\ \textit{You} \quad \begin{array}{c} a \\ b \\ c \end{array} & \left[\begin{array}{c|cc} 8 & 8 & 5 \\ 9 & 2 & 4 \\ \hline 7 & 2 & 1 \end{array}\right] \end{array}
$$

In the reduced game, it is best for you to play a which results in your opponent playing z. So the equilibrium strategy is (a, z) with a value of 5. Note that all this analysis is predicated on the view that both players are rational. If both players make irrational choices , you could do better. However, the equilibrium choice guarantees that you will do no worse than the 5. ■

There can be more than one equilibrium point in two–person, zero–sum games. When this occurs however, the value of the equilibrium points is equal. The method we used above to determine the equilibrium strategies is called *dominance*. A row (or column) of matrix **A** is said to dominate another row (or column) **B** if all values of **A** are greater (less) than all values of **B**. It is intuitively clear that if a row is dominated it is poor choice for you, while it is equally true that if a column is dominated, then it is a poor choice for your opponent. We found dominate strategies and chose them or eliminated strategies (rows) that were dominated for ourselves and eliminated dominate strategies (columns) or choose strategies that were dominated for our opponent. If we can continue this process, simplifying the matrix until we reach an equilibrium strategy.

However we have already seen that not all games have a strategy that can be determined by domination. The game rock-paper-scissors is one such example. Another example is :

$$
\begin{array}{cc} & \textit{Opponent} \\ & \begin{array}{cc} x & y \end{array} \\ \textit{You} \quad \begin{array}{c} a \\ b \end{array} & \left[\begin{array}{cc} 1 & -1 \\ -1 & 1 \end{array}\right] \end{array}
$$

No row or column dominates another. If your choice is known by your opponent, the opponent can always win. If you played this game over and over again, then it would be best to randomly choose either A or B. If you choose the same strategy over and over, your opponent would quickly choose the opposite strategy and would always win. If you choose either A or B with 50% probability, it is clear that you would win half the time and lose half the time. This is called a mixed strategy.

However it was shown by Nash that all games have mixed strategy equilibrium points. A mixed strategy is when a random method is used to select a strategy. We will discuss mixed strategies in the next section.

2.3 Mixed Strategies and the Mini–Max Theorem

When we calculated the equilibrium strategies we tried to maximize our payoff while our opponent tried to minimize the payoff. This means that equilibrium strategies can be characterized as mini–max points. John von Neumann proved that in all zero–sum two–person games, there is a mixed strategy that produces an equilibrium point.

Theorem 2.3.1. *Let X and Y be column vector mixed strategies for two players in a game defined by the matrix* $\mathbf{A}$*. Then the value of the game ν can be determined by:*

$$\max_{\mathbf{X}} \min_{\mathbf{Y}} \mathbf{X}^{\mathrm{T}}\mathbf{A}\mathbf{Y} = \min_{\mathbf{Y}} \max_{\mathbf{X}} \mathbf{X}^{\mathrm{T}}\mathbf{A}\mathbf{Y} = \nu$$

Note that since $\mathbf{X}$ and $\mathbf{Y}$ are mixed strategies, their entries are probabilities of making that choice and the sum of their entries is 1.

A mixed strategy payoff is one that is expected on average when the game is played a large number of times. Sadly it says nothing about the short run. For example consider a game similar to the television show Deal or No Deal. In this game, there are 2 cases, one case contains \$1 and the other contains \$1,000,000 dollars, where the amounts in each case are hidden from you. If you were offered to either pick a case and keep that amount or \$300,000 dollars to walk way, most people would take the guaranteed three hundred thousand dollars even though the expected average payoff is half a million dollars. Now if the game was played many times, and the contestant could keep the average amount won, then choosing a single case at random every time would earn the contestant to an extra 200,000 dollars. So the existence of an mini-max strategy does not always lead to the expected payoff unless the game is repeated.

Some games are played repeatedly. Consider the situation where two television networks each need to choose what time to schedule their hit television programs in order to get the most viewers. In this game the payoff would be the number of viewers for each show. Since the shows are broadcast every week the game is essentially repeated each week of the television season.

The trait of some people to choose a smaller guaranteed amount over a chance at a larger amount is called *risk adverse*. In computations concerning human behavior, the tendency to avoid risk can often be taken into account by adjusting the payoffs.

Lets begin by computing the payoff when using a mixed strategy.

$$\begin{array}{cc} & \textit{Opponent} \\ & \begin{array}{ccc} x & y & z \end{array} \\ \textit{You} \begin{array}{c} a \\ b \\ c \end{array} & \left[\begin{array}{ccc} 0 & 7 & 10 \\ 5 & 0 & 5 \\ 10 & 7 & 0 \end{array}\right] \end{array}$$

It is easily seen that no row or column dominates or is dominated so the game cannot be reduced. If the mixed strategy for the opponent is to choose x half the time and z half the time, then the value of the game is

$$\begin{bmatrix} 0 \\ 5 \\ 10 \end{bmatrix} (0.5) + \begin{bmatrix} 10 \\ 5 \\ 0 \end{bmatrix} (0.5) = \begin{bmatrix} 5 \\ 5 \\ 5 \end{bmatrix}$$

It is intuitively clear that computing mixed strategies will be considerably harder than computing pure strategies. If we are solving the problem by hand we begin by simplifying the problem using domination. We remove all dominated rows,and columns. Once that is done we are left with assigning probabilities to each row and column which results in a linear algebra problem. We will illustrate the procedure with the following example.

Example 2.3.1. Assume you are making a simple artificial intelligence, for a computer football game. When the user is on offense, they have the options to run or pass. The computer can either setup a run or pass defense. The payoff will be the expected number of yards gained or lost.

$$\begin{array}{ccc} & & \text{defense} \\ & & \begin{array}{cc} run & pass \end{array} \\ \text{offense} & \begin{array}{c} run \\ pass \end{array} & \left[\begin{array}{cc} -2 & 10 \\ 20 & -10 \end{array}\right] \end{array}$$

It is clear that the offense cannot always run or pass , since any predictable strategy will result in the the defense anticipating the play and choosing a strategy that results in losses. Using the mini-max theorem we see

$$\max_p \min_q \begin{bmatrix} p & 1-p \end{bmatrix} \begin{bmatrix} -2 & 10 \\ 20 & -10 \end{bmatrix} \begin{bmatrix} q \\ 1-q \end{bmatrix} = \min_q \max_p \begin{bmatrix} p & 1-p \end{bmatrix} \begin{bmatrix} -2 & 10 \\ 20 & -10 \end{bmatrix} \begin{bmatrix} q \\ 1-q \end{bmatrix} = \nu$$

First consider the offensive strategy. Then we want to find the p to maximize the minimum column of

$$\begin{bmatrix} p & 1-p \end{bmatrix} \begin{bmatrix} -2 & 10 \\ 20 & -10 \end{bmatrix} =$$

$$\begin{bmatrix} -2p+20(1-p) & 10p-10(1-p) \end{bmatrix} =$$

$$\begin{bmatrix} -22p+20 & 20p-10 \end{bmatrix}$$

Since we want to maximize this in the face of our opponent's defense by choosing the minimum column, we need to set the values equal to each other.

$$-22p+20 = 20p-10 \Rightarrow p = \frac{15}{21} \Rightarrow p = \frac{5}{7}$$

So we found the offense should pick *run* with a probability of $\frac{5}{7}$ and choose *pass* with a probability of $\frac{2}{7}$. The value of the game is found by plugging in the value for p in either column. Here $\nu = -22(\frac{5}{7}) + 20 = 4\frac{2}{7}$

In an analogous manner we can solve for the defense's mixed strategy.

$$\begin{bmatrix} -2 & 10 \\ 20 & -10 \end{bmatrix} \begin{bmatrix} q \\ 1-q \end{bmatrix} = \begin{bmatrix} -2q+10(1-q) \\ 20q-10(1-q) \end{bmatrix} = \begin{bmatrix} 10-12q \\ 30q-10 \end{bmatrix}$$

Here to minimize the value against all choice of rows by the offense, the defense sets them equal. So $10-12q = 30q-10$ or $42q = 20$ and $q = \frac{10}{21}$. So the mini–max strategy for the defense is choose *run* with the probability $\frac{10}{21}$ and choose *pass* with probability $\frac{11}{21}$.

As a check we can recompute ν using $q = \frac{10}{21}$ and see if it agrees with our previous value.

$\nu = 10 - 12(\frac{10}{21}) = 10 - (\frac{40}{7}) = 4\frac{2}{7}$. ■

Example 2.3.2. Consider the following payoff matrix:

$$\begin{array}{ccc} & & \textit{Opponent} \\ & & \begin{array}{cc} x & y \end{array} \\ \textit{You} & \begin{array}{c} a \\ b \end{array} & \begin{bmatrix} 5 & 2 \\ 4 & 5 \end{bmatrix} \end{array}$$

Applying the Mini-Max Theorem, we find

$$\max_p \min_q \begin{bmatrix} p & 1-p \end{bmatrix} \begin{bmatrix} 5 & 2 \\ 4 & 5 \end{bmatrix} \begin{bmatrix} q \\ 1-q \end{bmatrix} = \min_q \max_p \begin{bmatrix} p & 1-p \end{bmatrix} \begin{bmatrix} 5 & 2 \\ 4 & 5 \end{bmatrix} \begin{bmatrix} q \\ 1-q \end{bmatrix} = \nu$$

Now let us just concern ourselves with the strategies for you. Then we want to find the p to maximize

$$\begin{bmatrix} p & 1-p \end{bmatrix} \begin{bmatrix} 5 & 2 \\ 4 & 5 \end{bmatrix} =$$

$$\begin{bmatrix} 5p+4(1-p) & 2p+5(1-p) \end{bmatrix} =$$

$$\begin{bmatrix} p+4 & 5-3p \end{bmatrix}$$

Since we want to maximize this in the face of our opponent choosing the minimum column, we need to set the values equal to each other.

$$p+4 = 5-3p \Rightarrow 4p = 1 \Rightarrow p = \frac{1}{4}$$

So we found we should pick a with a probability of $\frac{1}{4}$ and choose b with a probability of $\frac{3}{4}$. The value of the game is found by plugging in the value for p in either column. Here $\nu = p+4 = 4\frac{1}{4} = 5-3(\frac{1}{4})$

In an analogous manner we can solve for the opponents mixed strategy.

$$\begin{bmatrix} 5 & 2 \\ 4 & 5 \end{bmatrix} \begin{bmatrix} q \\ 1-q \end{bmatrix} = \begin{bmatrix} 5q+2(1-q) \\ 4q+5(1-q) \end{bmatrix} = \begin{bmatrix} 2+3q \\ 5-q \end{bmatrix}$$

Here to minimize the value against all choice of rows by you, the opponent sets them equal. So $2+3q = 5-q$ or $4q = 3$ and $q = \frac{3}{4}$. So the mini–max strategy for your opponent is choose x with the probability $\frac{3}{4}$ and choose y with probability $\frac{1}{4}$.

As a check we can recompute ν using $q = \frac{3}{4}$ and see if it agrees with our previous value.

$\nu = 2+3q = 2+3(\frac{3}{4}) = 4\frac{1}{4}$. ■

The following is another example illustrating complex procedure.

Example 2.3.3. Consider the game defined by the payoff matrix :

$$\begin{array}{ccc} & & \textit{Opponent} \\ & & \begin{array}{cc} x & y \end{array} \\ \textit{You} & \begin{array}{c} a \\ b \end{array} & \begin{bmatrix} -1 & 5 \\ 3 & -5 \end{bmatrix} \end{array}$$

It is clear that there is no non–mixed strategy in this situation. So to determine the mixed–strategy for you we maximize :

$$\begin{bmatrix} p & 1-p \end{bmatrix} \begin{bmatrix} -1 & 5 \\ 3 & -5 \end{bmatrix} = \begin{bmatrix} -p+3(1-p) & 5p-5(1-p) \end{bmatrix} = \begin{bmatrix} 3-4p & 10p-5 \end{bmatrix}$$

Setting the 2 columns equal to each other yields: $3-4p = 10p-5 \Rightarrow 8 = 14p \Rightarrow p = \frac{4}{7}$
Thus your mini–max strategy is $a(\frac{4}{7}) + b(\frac{3}{7})$ and the value ν of the game is $3-4(\frac{4}{7}) = 3-\frac{16}{7} = \frac{5}{7}$
To determine the mixed–strategy for your opponent we minimize :

$$\begin{bmatrix} -1 & 5 \\ 3 & -5 \end{bmatrix} \begin{bmatrix} q \\ 1-q \end{bmatrix} = \begin{bmatrix} -q+5(1-q) \\ 3q-5(1-q) \end{bmatrix} = \begin{bmatrix} 5-6q \\ -5+8q \end{bmatrix}$$

Setting the two rows to each other yields : $5-6q = -5+8q \Rightarrow 10 = 14q \Rightarrow q = \frac{5}{7}$ Thus the opponent's mini–max strategy is $x(\frac{5}{7}) + y(\frac{2}{7})$ and the value ν of the game is 5 -6$(\frac{5}{7})$ = 5 -$(\frac{30}{7})$ = $\frac{5}{7}$.
■

Although finding the mixed strategy in a 2 person game with only 2 possible strategies, the complexity of the procedure grows exponentially as the number of choices increases. The following example illustrates the procedure where you have 3 choices and your opponent has 2 choices.

Example 2.3.4. Consider the following payoff matrix:

$$\begin{array}{cc} & \textit{Opponent} \\ & \begin{array}{cc} x & y \end{array} \\ \textit{You} \begin{array}{c} a \\ b \\ c \end{array} & \begin{bmatrix} 4 & 6 \\ 3 & 8 \\ 6 & 3 \end{bmatrix} \end{array}$$

Note that no row or column can be eliminated through dominance reductions. To solve for your strategy, we try to maximize the minimum value of all rows of

$$\begin{bmatrix} p & q & 1-p-q \end{bmatrix} \begin{bmatrix} 4 & 6 \\ 3 & 8 \\ 6 & 3 \end{bmatrix}$$

This leads to trying to maximize the minimum column of

$$\begin{bmatrix} 4p+3q+6(1-p-q) & 6p+8q+3(1-p-q) \end{bmatrix} =$$

$$\begin{bmatrix} 6-2p-3q & 3+3p+5q \end{bmatrix}$$

Setting both columns equal to each other yields :

$$3 = 5p+8q$$

To find your opponent's mixed strategy we try to minimize the maximum of

$$\begin{bmatrix} 4 & 6 \\ 3 & 8 \\ 6 & 3 \end{bmatrix} \begin{bmatrix} r \\ 1-r \end{bmatrix} = \begin{bmatrix} 4r+6(1-r) \\ 3r+8(1-r) \\ 6r+3(1-r) \end{bmatrix} = \begin{bmatrix} 6-2r \\ 8-5r \\ 3+3r \end{bmatrix}$$

We are now faced with a choice of what rows to set equal. Setting the first 2 rows equal we find, $r = \frac{2}{3}$ With this value for r, the values of the rows are :

$$\begin{bmatrix} 4\frac{2}{3} \\ 4\frac{2}{3} \\ 5 \end{bmatrix}$$

If we set the last 2 rows equal we find, $r = \frac{5}{8}$ With this value for r, the values of the rows are :

$$\begin{bmatrix} 4\frac{2}{8} \\ 4\frac{7}{8} \\ 4\frac{7}{8} \end{bmatrix}$$

Finally if we set the first and last rows equal we find, $r = \frac{3}{5}$ With this value for r, the values of the rows are :

$$\begin{bmatrix} 4\frac{4}{5} \\ 5 \\ 4\frac{4}{5} \end{bmatrix}$$

In the first case and last cases, the maximum rows is 5. So the opponent will pick the middle case where the maximum is only $4\ \frac{7}{8}$.

Using the value of the game, we can now solve for p and q. $6-2p-3q = 4\frac{7}{8}$. Combining this equation with our previous equation for p and q we find $p = 0$ and $q = \frac{3}{8}$ ■

The above example illustrates the complexity in computing a mini-max equilibrium. This type of problem is in the class of *linear programming* and is typically solved using the *simplex method* . Although, the simplex method and its variants are used in practice, it cannot be shown that some cases will result in unreasonably long times (exponential) for the computation. In 1979, Khachian (1979) published a new algorithm for linear programming , called the *ellipsoid method*, which *is* provably polynomial but seems to be slower for most problems. Typically at the time of this writing, equilibriums in games with dozens of options for each player are quickly computed, but for games with hundreds of choices for each player the computation may be intractable.

2.4 Two–Person Non–zero Sum Games

In many economic contexts the "games" are not zero-sum. Goods and services can be created, and destroyed resulting in net gains or losses. These games are more difficult to analyze, result in no clear cut optimal strategy and the results are subject to interpretation as the players have both competitive and cooperative traits. To represent such a game, there now needs to be two values in each square of the matrix. The first value will represent the profit for row player, while the second value will represent the profit for the column player. Again we will be looking for equilibria, which

are strategies where if either player changes their strategy unilaterally from the equilibrium, that player will do worse.

To find the equilibria in a non-zero sum game we look at each row and mark the greatest value for the column player. Similarly, we then examine each column and mark the greatest value for the row player. If we have marked both values in a strategy pair, that pair of strategies is an equilibrium. Once we have identified the equibria, we will need to consider if it is likely to occur. This is often a matter of cooperation and trust. These ideas are illustrated in the following example.

Example 2.4.1. Suppose a game played by Pepsi and Coke. Their choices are to advertise or not, and the payoffs will be their profits. We will assume that each company earns \$100 million and if they advertise they earn an additional \$30 million from their competitor and that advertising costs \$20 million. This game is then represented by the following payoff matrix.

$$\begin{array}{cc} & \\ Pepsi & \begin{array}{c} \\ \text{noAd} \\ \text{Ad} \end{array} \end{array} \begin{array}{c} \begin{array}{cc} \multicolumn{2}{c}{Coke} \\ \text{noAd} & \text{Ad} \end{array} \\ \left[\begin{array}{cc} (100,100) & (70,110) \\ (110,70) & (80,80) \end{array}\right] \end{array}$$

We now mark the largest payoff columns for each row in red, and the largest payoff rows in each column in blue.

$$\begin{array}{cc} & \\ Pepsi & \begin{array}{c} \\ \text{noAd} \\ \text{Ad} \end{array} \end{array} \begin{array}{c} \begin{array}{cc} \multicolumn{2}{c}{Coke} \\ \text{noAd} & \text{Ad} \end{array} \\ \left[\begin{array}{cc} (100,100) & (70,110) \\ (110,70) & (80,80) \end{array}\right] \end{array}$$

So in this case, logic tells both companies to advertise and each earns \$80 million. However, the matrix also suggests that if they both trust each other not to advertise, they will both earn more profit. It is interesting to note that this is exactly what happened to the cigarette industry when television advertising was banned. The cigarette companies all made more profits. ■

The above game is an example of the prisoner's dilemma. If the prisoners trust each other, they both profit the most. However trust among thieves or in corporate America might be hard to find. Note that the equilibrium strategy is not optimal. We now consider a more complex example of finding Nash equilibriums in a non–zero sum game.

Example 2.4.2.

$$\begin{array}{cc} & \\ \text{You} & \begin{array}{c} a \\ b \\ c \\ d \\ e \end{array} \end{array} \begin{array}{c} \begin{array}{ccccc} \multicolumn{5}{c}{\text{Opponent}} \\ u & v & x & y & z \end{array} \\ \left[\begin{array}{ccccc} (9,8) & (7,1) & (5,6) & (3,4) & (1,2) \\ (7,6) & (5,2) & (3,6) & (1,4) & (9,4) \\ (5,4) & (3,4) & (1,8) & (9,7) & (7,1) \\ (3,2) & (1,6) & (9,4) & (7,8) & (5,8) \\ (1,5) & (8,8) & (7,6) & (5,2) & (3,7) \end{array}\right] \end{array}$$

We color the maximum row value in each column blue and color the maximum column value in each row red, we find that there are two equilibriums $\{(a,u),(e,v)\}$. Since they have different

values, the best strategy is not clear. If *you* trust your opponent, you might be tempted to choose a. However if the *opponent* is out to eliminate the competition, they may choose a non–equilibrium strategy in the hopes of a short term loss will eventually run *you* out of business and result in a long term gain. ■

If a game is repeated (*iterated*), then there is a possibility to build trust. Let us consider some well–known strategies for a the classic Prisoner's Dilemma game.

Example 2.4.3. In this game there are 2 prisoners . They are given the choice to plead guilty or innocent. If they both plead guilty, they are given 3 years each. If they both plead innocent, then they are set free. But if one pleads guilty and one pleads innocence, then the prisoner pleading guilty is given 1 year while the prisoner pleading innocence receives 5 years. Since long prison sentences are undesirable, the payoff matrix contains the negative of the length of the prison sentences.

$$\begin{array}{ccc} & & P_2 \\ & & \begin{array}{cc} I & G \end{array} \\ P_1 \begin{array}{c} I \\ G \end{array} & \left[\begin{array}{cc} 0,0 & -5,-1 \\ -1,-5 & -3,-3 \end{array}\right] \end{array}$$

Here the Nash Equilibrium is (G, G) but the optimal strategy for both players is (I, I). We will explore using iteration to come to move to the optimal strategy. We will play the game repeatedly and sum the payoffs. Some well known strategies are :

1) Tit–for–Tat (TFT) In this strategy the prisoner chooses innocent in the first step after that, chooses exactly what the other prisoner choose in the last iteration.

2) Tit–for–Two–Tats (TFTT) In this strategy the prisoner chooses innocent unless the other prisoner chooses guilty in the last two iterations.

3) Never Again (NA) In this strategy the prisoner chooses innocent unless the other prisoner chooses guilty, ever.

4) Always Innocent (AI)

5) Always Guilty (AG)

Axelrod in his book on cooperation, examined these strategies where a computer followed the above strategies against a human opponent. He found that Tit–for–Tat worked the best. ■

The above game was approximated during World War I . This was one of the first wars where trench warfare was a major component of the ground war. In trench warfare, groups of soldiers on either side hid in trenches a short distance apart. If one side attacks the other sides retaliates and both sides are hurt. However if neither side attacks, then no one gets hurt. So even though the soldiers were enemies, they avoided attacking each other and this is one of the reasons that World War I lasted so long.

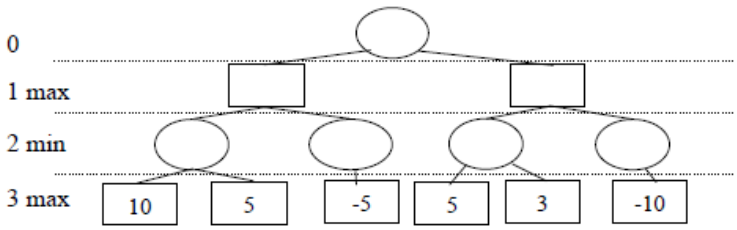
Trench Warfare from WWI

In the non-zero sum games, the game can be simplified by looking for dominated and dominating strategies. In the situation where there is no equilibrium strategies, we again turn to a mixed strategy. These strategies are computed in a manner similar to the computation of mixed strategies in zero–sum games using the min-max theorem.

2.5 Zero-sum sequential games

In games like chess, checkers and Tic-Tac-Toe, the players alternate making moves. To figure out your best strategy, you work backwards from the end of the game knowing that each step the *opponent* will attempt to minimize the payoff while at each step you attempt to maximize the payoff. In Figure 2.1, we have drawn a portion of the *game tree* for the game Tic–Tac–Toe. At the top (root), the board is empty. The first row shows player X's possible moves. The second row contains all of players O's possible moves based on the previous board. Due to space considerations, we have only drawn the boards resulting from X choosing the upper right hand corner first. Each row alternates X moves and O moves until no more legal moves are possible. We then give a value to each of these terminal boards. Here we assigned 1 for X winning, -1 for O winning and 0 for a tie (cat's game).

Now to solve the game we will work backwards. We will look at all the terminal nodes and assign values to each of the interior boards. The values are assigned by assuming that the O player will make best move for them (minimizing the payoff) while the X player will make the choice for himself to maximize the payoff. For example consider the following small game tree. In this tree we have the oval player who is trying to minimize the payoffs while the rectangle player is trying to maximize the payoff.



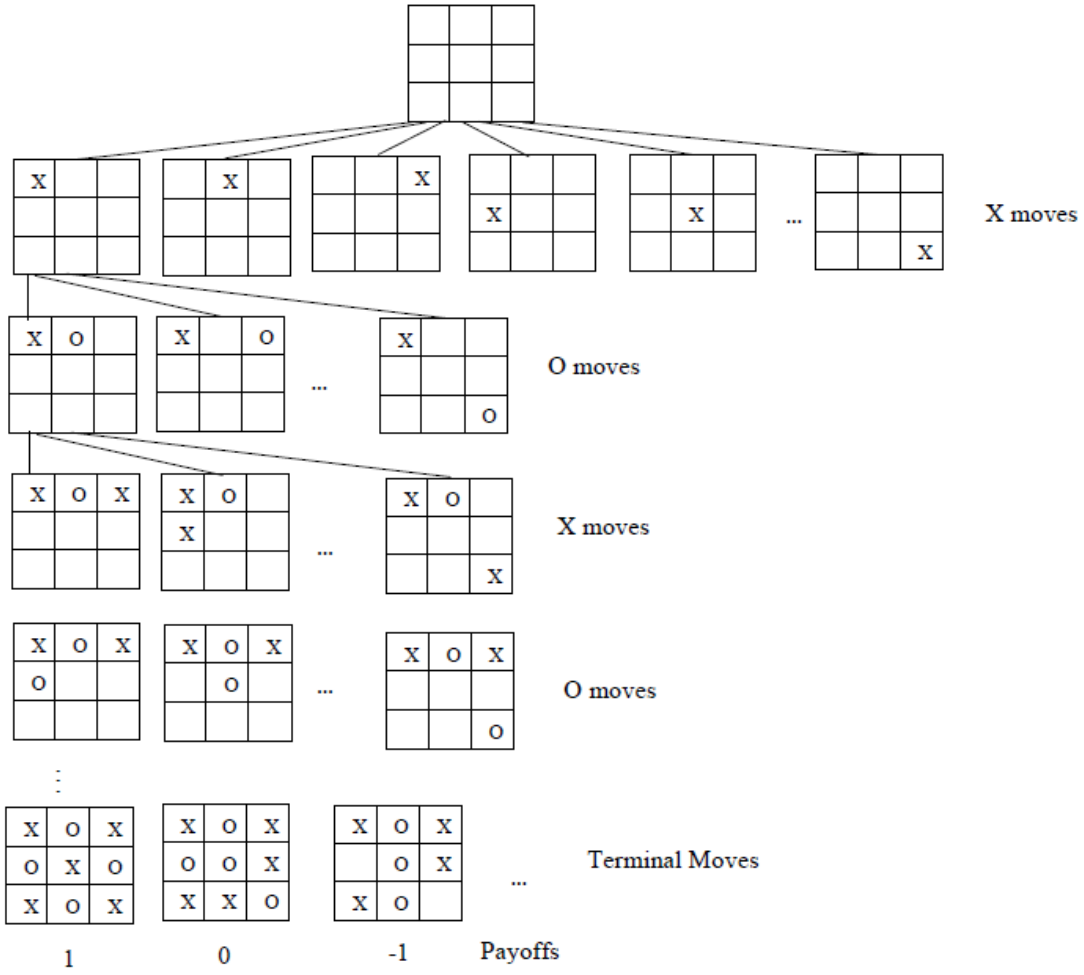
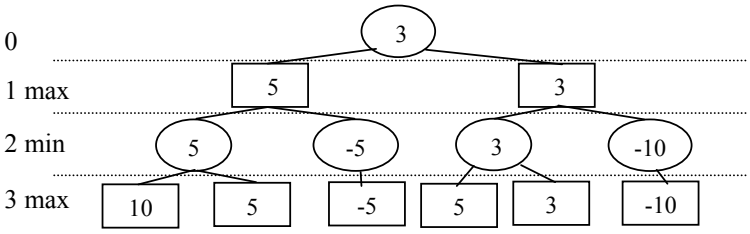


Figure 2.1: Game Tree for Tic–Tac–Toe.

Payoffs only in a Game Tree for a small game.

Example 2.5.1. To compute the values for row 2 then, we use the minimum of the nodes in row 3. To compute the values for row 1 then, we maximize the values in row 2. To find the value of the root we use the minimum of the elements in row 1, which is 3. This is shown in figure 2.5.1. So the oval player will choose the right branch, the rectangle player will choose the left branch and then the oval player will chose the right branch resulting in the payoff of 3.


0
1 max
2 min
3 max
3
5
3
5
-5
3
-10
10
5
-5
5
3
-10

Minimax Interior for a Game Tree for a small game.

■

Example 2.5.2. Figure 2.4 illustrates another example of finding the minimax solution to a sequential game.

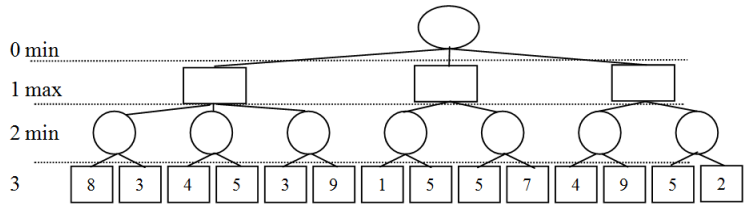


Figure 2.4: Payoffs for the Game Tree in example 2.5.2

By working backwards from the third row we find: Figure 2.5. ■

The game Tic–Tac–Toe has only a small game tree, with less than 300 possible boards. In a larger game such as chess (where there are about 10^{135} possible boards, only part of the game tree can be generated and an evaluation function is introduced to give value to the boards at the bottom of the tree. This choice of function determines how accurate the strategy will play. For example in checkers the function could be the number of your checkers + the number of your kings* 10 - the number of your opponents checkers - the number of your opponents kings*10. This function will attempt to maximize the number of your pieces and minimize your opponents pieces. Computer programs that look only a fixed number of moves, are susceptible to the *horizon effect*. This effect is the possibility that the program will make a move which is detrimental, but the detrimental effect is not visible because the program does not search to the depth of the error (i.e. beyond its horizon). The horizon effect can be mitigated by extending the search algorithm with a *quiescence search*. This gives the search algorithm ability to look beyond its horizon for a certain class of moves of major importance to the game state, such as the capture of pieces in chess or checkers.

2.6 Summary

Theoretical Concepts

This chapter briefly discussed the following concepts encountered when choosing an optimal strategy: two–person, zero–sum, non–zero–sum, payoff matrices, value of the game, iterated games, sequential games, equilibrium strategy, equilibrium value, dominate strategy, mixed strategy and game trees. In all of the games discussed, we attempted to find a minimax strategy, one that maximizes your payoff in the face of opposition.

Applications

The techniques here can be used in building artificial intelligence in games as well as in cooperating agents. In some real world situations these techniques can also be used to make decisions.

2.7 Exercises

1) Find the best strategy for both you and your opponent and value for the following payoff matrix.

$$\begin{array}{cc} & Opponent \\ & \begin{array}{cc} x & y \end{array} \\ You \begin{array}{c} a \\ b \end{array} & \left[\begin{array}{cc} 1 & 2 \\ 4 & 5 \end{array}\right] \end{array}$$

2) Find the best strategy for both you and your opponent and value for the following payoff matrix.

$$\begin{array}{cc} & Opponent \\ & \begin{array}{cc} x & y \end{array} \\ You \begin{array}{c} a \\ b \end{array} & \left[\begin{array}{cc} 3 & 2 \\ 4 & 1 \end{array}\right] \end{array}$$

3) Find the best strategy for both you and your opponent and value for the following payoff matrix.

$$\begin{array}{cc} & Opponent \\ & \begin{array}{ccc} x & y & z \end{array} \\ \begin{array}{c} \\ You \\ \\ \end{array} \begin{array}{c} a \\ b \\ c \end{array} & \left[\begin{array}{ccc} 1 & 2 & 1 \\ 7 & 2 & 1 \\ 6 & 5 & 5 \end{array}\right] \end{array}$$

4) Find the best strategy for both you and your opponent and value for the following payoff matrix.

$$\begin{array}{cc} & Opponent \\ & \begin{array}{ccc} x & y & z \end{array} \\ \begin{array}{c} \\ You \\ \\ \end{array} \begin{array}{c} a \\ b \\ c \end{array} & \left[\begin{array}{ccc} 5 & 3 & 4 \\ 7 & 2 & 2 \\ 3 & 1 & 2 \end{array}\right] \end{array}$$

5) Find the best strategy for both you and your opponent and value for the following payoff matrix.

$$\begin{array}{cc} & Opponent \\ & \begin{array}{ccc} x & y & z \end{array} \\ \begin{array}{c} \\ You \\ \\ \end{array} \begin{array}{c} a \\ b \\ c \end{array} & \left[\begin{array}{ccc} 35 & 10 & 60 \\ 45 & 55 & 55 \\ 40 & 10 & 65 \end{array}\right] \end{array}$$

6) Find the best strategy for both you and your opponent and value for the following payoff matrix.

$$\begin{array}{cc} & Opponent \\ & \begin{array}{cc} x & y \end{array} \\ You \begin{array}{c} a \\ b \end{array} & \left[\begin{array}{cc} 5 & 2 \\ 4 & 5 \end{array}\right] \end{array}$$

7) Find the best strategy for both you and your opponent and value for the following payoff

matrix.

$$\begin{array}{ccc} & & Opponent \\ & & \begin{array}{cc} x & y \end{array} \\ You & \begin{array}{c} a \\ b \end{array} & \left[\begin{array}{cc} -1 & 2 \\ 3 & -4 \end{array}\right] \end{array}$$

8) Identify all the pure strategy Nash Equilibria.

$$\begin{array}{ccc} & & Opponent \\ & & \begin{array}{ccc} x & y & z \end{array} \\ You & \begin{array}{c} a \\ b \\ c \end{array} & \left[\begin{array}{ccc} 15,15 & 0,20 & 0,10 \\ 20,0 & 10,10 & 5,10 \\ 10,0 & 10,0 & 5,5 \end{array}\right] \end{array}$$

9) Identify all the pure strategy Nash Equilibria.

$$\begin{array}{ccc} & & Opponent \\ & & \begin{array}{ccc} x & y & z \end{array} \\ You & \begin{array}{c} a \\ b \\ c \end{array} & \left[\begin{array}{ccc} -5,-5 & 10,0 & 3,-3 \\ 0,10 & 2,2 & 1,6 \\ -3,3 & 6,1 & 5,5 \end{array}\right] \end{array}$$

10) Find the value of the following sequential game where the max player goes first.

2.7.1 Answers to Exercises

1) (You, $Opponent$) =(b,x) , $\nu = 4$

2) (You, $Opponent$) =(a,y) , $\nu = 2$

3) (You, $Opponent$) =(c,z) , $\nu = 5$

4) (You, $Opponent$) =(a,y) , $\nu = 3$

5) (You, $Opponent$) =(b,x) , $\nu = 45$

6) (You, $Opponent$) =$(\frac{a}{4} + \frac{3b}{4}, \frac{3x}{4} + \frac{y}{4}), \nu = 5\frac{1}{4}$

7) (You, $Opponent$) =$(\frac{a}{2} + \frac{b}{2}, \frac{4x}{5} + \frac{y}{5}), \nu = 1$

8) (You, $Opponent$) = $\{(b, y), (b, z), (c, z)\}$

9) (You, $Opponent$) = $\{(a, y), (b, x), (c, z)\}$

2.7.2 Computer Activities

UVa indicates that the problem comes from the Universidad de Valladolid on line problem archive. A link to the UVa is given below:

`http://uva.onlinejudge.org/index.php`

1) Prompt the user for a payoff matrix in a zero sum game and identify all pure Nash Equalibria.
2) Prompt the user for a non zero sum payoff matrix and identify all Nash Equalibria.
3) Implement a solution to the programming problem (UVa 162) 10097 - The Color Game
4) Implement a solution to the programming problem (Uva 11863) - Prime Game

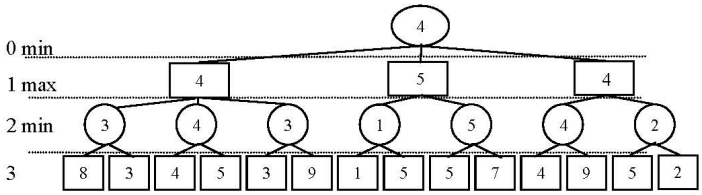


Figure 2.5: Game Tree Interior for example 2.5.2.

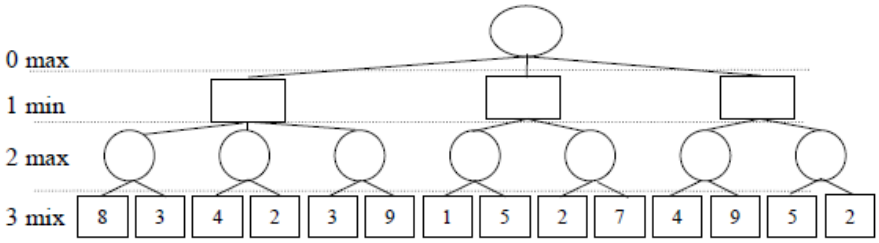


Figure 2.2: Game Tree Homework 10

Bibliography

[1] Axelrod, R. , *The Evolution of Cooperation*, 1984.

[2] Borel, E. , *Applications aux Jeux de Hasard*, 1938.

[3] Dantzig, G. , *Linear Programming and extensions*, 1963.

[4] Nash, J, , "Equilibrium points in n-person games", *Proceedings of the National Academy of Sciences of the United States of America*, **36 (1)**, 48–49, 1950.

[5] Shor, N.m Zhurbenko, N., G. , "The minimization method using space dilatation in direction of difference of two sequential gradients "*Kibermetika*, **3**, 51–59, 1971.

[6] von Neumann, J. and Morgenstern, O. , *Theory of Games and Economic Behavior*, 1944.

Chapter 3

Regular Languages

By and large, language is a tool for concealing the truth. George Carlin

3.1 Introduction and History

In the next three chapters we will consider theoretical models of computer programs. The models will be simpler than real programs and this simplicity will allow us to analyze them more fully and answers such questions as :

1) Are there problems a computer cannot solve?

2) Are there problems a computer cannot solve in a reasonable amount of time?

3) Why are programming languages in the form they are now?

Additionally these models will deepen our understanding of programming and introduce new paradigms which can be utilized in writing computer programs.

This study of what can and cannot be computed can be traced back to George Cantor (1845–1918). Cantor did a great deal of work in set theory and in the process came up with some mathematical paradoxes. Some of these paradoxes could be tolerated such as various sizes of infinities. However some of these paradoxes confounded the mathematicians of his time. For example if the universal set contains everything, how can there be sets with more elements than the universal set ?

Some of the great men in the history of the theory of computation include: David Hilbert (1862–1942) was a great mathematician who among his many works tried to eliminate all the contradictions brought on by Cantor's set theory. He also wondered if every truth involving mathematics could be proven true. This question was eventually answered by Kurt Gödel (1906 – 1978) .

Gödel's incompleteness theorems are two theorems of mathematical logic that establish inherent limitations of all but the most trivial axiomatic systems capable of doing arithmetic. The theorems, proven by Kurt Gdel in 1931, are important both in mathematical logic and in the philosophy of mathematics. The two results are widely, but not universally, interpreted as showing that Hilbert's program to find a complete and consistent set of axioms for all mathematics is impossible, giving a negative answer to Hilbert's second problem.

Alan Turing (1912- 1954) was the one of the pioneers of the modern computer. He helped invent one of the first electronic computers to help decode the secret codes of the Nazis produced by their famous *Enigma machine*. Further he is credited with proving that there are some fundamental problems no computer can solve.

The easiest problems that models can solve are *yes* or *no* problems. A subset of these problems are language recognition problems. Given a set of "words" we want to determine if our computation model can determine if an input word is in or is not in the set. We will begin by formalizing the notion of a language.

3.2 Mathematical Prerequisites

We will begin with formal definitions of alphabet, string, and language.

Definition 3.2.1. An *alphabet* denoted by Σ is a set of letters.

Example 3.2.1. The following are examples of alphabets.
In English the alphabet is $\Sigma = \{a, b, c, ..., z\}$.
In binary the alphabet is $\Sigma = \{0, 1\}$.
To make decimal numbers the alphabet is $\Sigma = \{+, -, ., 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. ■

Definition 3.2.2. An *string* or *word* is a collection of letters concatenated together.

Definition 3.2.3. The *null string* or *empty string* denoted by λ . This string has no letters and takes no space. It is not a part of any alphabet.

Definition 3.2.4. A *language* is a set of strings.

Example 3.2.2. The following are examples of strings and languages.
For example in English some strings are : cat, dog, fish, frog.
In binary some strings are 0110, 10100, 000.
If $\Sigma = \{x\}$, then one language is $L = \{x, xx, xxx, ...\} = \{x^n, n = 1, 2, 3...\}$
In C the language of reserved words is {auto, if, break, int, case, long, char, register, continue, return, default, short, do, sizeof, double, static, else, struct, entry, switch, extern, typedef, float, union, for, unsigned, goto, while} ■

We now consider some functions on strings.

Definition 3.2.5. A *cardinality* of a string w denoted $|w|$ is the number of letters in the string. It is also referred to as the *size* or the *length*

Example 3.2.3. The following are examples of computing cardinality.
Consider the alphabet $\Sigma = \{a, b, c, ..., z\}$. Using this alphabet $|\text{cat}| = 3$, and $|\text{mississippi}| = 11$.
However if $\Sigma =$ {m, i, ss, pp}, then $|\text{mississippi}| = 8$. This is because now pp is considered a single letter as is ss. ■

Definition 3.2.6. The *reverse* of a string w is the string created by writing the letters of w in reverse order.

Example 3.2.4. The following are examples of computing reverse.

- reverse(123) = 321
- reverse(cat) = tac
- reverse(xxx) = xxx

■

Definition 3.2.7. A string w is a *palindrome* if and only if reverse$(w) = w$.

Palindromes are fun and are used to illustrate some interesting properties of languages.

Example 3.2.5. The following are examples of palindromes (usually with spaces and capitalization ignored).

- radar
- dad
- racecar
- Madam I'm Adam
- Dammit I'm mad
- Able was I ere I saw Elba. (Elba is an island off the coast of France where Napoleon was exiled)
- A man, a plan, a canal Panama
- Rats live on no evil star. (note that this palindrome has the spaces appropriately placed).

■

Definition 3.2.8. The *Kleene star* of a set of strings S, denoted S^* is the union of all finite concatenations (including 0 concatenations) of the elements of S.

Example 3.2.6. The following are examples of the use of Kleene star.
If $S = \{x\}$, then $S^* = \{\lambda, x, xx, xxx, xxxx, ...\}$
If $S = \{a, b, c\}$, then $S^* = \{\lambda, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, aab, ...\}$
If $S = \{1, 01\}$, then $S^* = \{\lambda, 1, 01, 11, 111, 101, 011, 1111, 1101, 1011, 0111, 0101, ...\}$ ■

To show a string $w \in S^*$, we need to show that $w = w_1w_2w_3...w_n$ such that each of the $w_i \in S$

Theorem 3.2.1. *For any set of strings S, $S^* = (S^*)^*$.*

Proof

We must show both that $(S^*)^* \subseteq S^*$ as well as $S^* \subseteq (S^*)^*$. As usual with this type of proof, one direction is easy. Since for any set A, A^* contains all elements of A, then $(S^*)^*$ contains all strings in S^*, hence $S^* \subseteq (S^*)^*$. Now consider $w \in (S^*)^*$. Then $w = w_1 w_2 w_3 w_n$ where each w_i is an element of S^*, by the definition of Kleene star. Now since w_i is an element of S^*, $w_i = w_{i1} w_{i2} w_{im_i}$. where $w_{ij} \in S$. So w is exactly a concatenation of elements of S, so w is an element of S^*. Thus $(S^*)^* \subseteq S^*$ and we conclude that $S^* = (S^*)^*$.■

Example 3.2.7. Let T and S be two languages. If $S^* = T^*$ what can you conclude?

Certainly if $T = S$ then $S^* = T^*$ but can they be the equal if $T \neq S$?

Consider if $T = \{a, b\}$ and $S = \{a, b, aa\}$ Now T^* will contain all strings made up of a's and b's so T^* will certainly contain aa. Thus again $S^* = T^*$.

So we can conjecture that if $T \subseteq S$ and $S - T \in T^*$ or vice versa then $S^* = T^*$. ■

3.3 Recursive Definitions and Regular Expressions

Recursive definitions are another way (besides a listing , and a rule) to define a language. They are very useful when defining an infinite language and most closely resemble an induction proof. A recursive definition consists of three parts :

1) **Base Case** This is a finite list of strings that are members of the language.
2) **Recursive Step** This is a rule or a set of rules for generating new strings in the language by referring other strings in the language.
3) **Nothing Else** A statement that nothing else is in the language.

Example 3.3.1. The following is a definition of language PEI of positive even integers = {2, 4, 6, 8, 10 ... }.

1) **Base Case** $2 \in$ PEI.
2) **Recursive Step** If x is in PEI so is $x + 2$.
3) **Nothing Else** Nothing not described in the base case or the recursive step are in PEI.

Example 3.3.2. The following is a definition of language EvenLengthEvenA of the strings using only a and b which have even length and where every even letter is a. A listing of this set is {λ, aa, ba, baba, aaaa, baaa, aaba ... }.

1) **Base Case** λ is an element of EvenLengthEvenA.
2) **Recursive Step** If w is in EvenLengthEvenA so is $w$ba and $w$aa.
3) **Nothing Else** Nothing not described in the base case or the recursive step are in EvenLengthEvenA.

Example 3.3.3. The following is a definition of language Pal of the strings using only a and b and which are Palindromes. A listing of this set is {λ, a, b, aa, bb, aaa, aba, bab, bbb, aaaa, abba, baab, bbbb ... }.

1) **Base Case** λ, a, b are elements of Pal.

2) **Recursive Step** If w is in Pal so is awa and bwb.

3) **Nothing Else** Nothing not described in the base case or the recursive step are in Pal.

A special type of recursive definition is a *regular expression.* Any language that can be described by a regular expression is called a *regular language.*

Definition 3.3.1. Regular Expression of Language L using alphabet Σ

1) **Base Case** λ and any element of Σ is a regular expression .

2) **Recursive Step** If r_1 and r_2 are regular expressions, then so is

   (r_1)

   $r_1 r_2$

   $r_1 + r_2$

   r_1^*

3) **Nothing Else** Nothing not described in the base case or the recursive step are regular expressions.

Regular Expressions are very useful in pattern matching. They are implemented in the perl programming language and are used for parsing strings. For example input can be searched for email addresses, and phone numbers . Below is a comic from XKCD illustrating this fact.

Example 3.3.4. These examples show how to use regular expressions.

The + sign indicates the *or* operation. So if the expression is $s_1 + s_2$ this means use either s_1 or s_2.

Hence, abba + ab + bb describes the set {abba, ab, bb}

(a+b)(a+b) = aa+ab+ba+bb, since we choose either a or b, from the first factor and a or b from the second factor.

To generate any even length string then we use : ((a+b)(a+b))*

(a+b)* is the arbitrary concatenations of a or b . In English the above expression describes all strings made up of only a's or b's and including λ. ■

Example 3.3.5. Find a regular expression for strings made up of only a's and b's which of the the form of an even number of a's is even followed by an odd number of b's. $\{b, aab, aabbbb, ...\}$

(aa)*b(bb)* ■

Example 3.3.6. Find a regular expression for strings composed of only a's and b's such that the total number of 's is 3. Examples of these strings include : $\{bbb, abbb, babb, ...\}$

a*ba*ba*ba* ■

Example 3.3.7. Find a regular expression for strings composed of only a's and b's such that all the letters alternate. λ+ a(ba)*(b+λ) + b(ba)*(b+λ)

The first term is of course the empty string. The second term is for alternating letters which start with a. After the Kleene star, the factor (b+λ) is to account for when the alternations end with b. The last term is when the alternating letters start with b. ■

Figure 3.1: From XKCD, A web comic of romance, sarcasm, math, and language.

Example 3.3.8. Describe the regular expression ((a+b)a)* in English.

Here we see the first letter is a or b, while the second letter is a. We then concatenate this with itself an arbitrary number of times. Hence this describes all strings of even length where a appears in the even numbered positions. Here I classify λ as an even length string. ■

Example 3.3.9. Describe the regular expression (a(a+bb)*)*in English.

Often it helps to list the first few strings described by the expression and look for patterns. Here the first strings in order of size are {λ, a, aa, bb, aaa, abb, aaaa, aabb, abba ... }

Here we see the non –zero length strings start with an a.

Also we see b‘s only occur in pairs. Hence the number of b‘s in any group of b‘s is even.

The above description describes the language. ■

Before we leave our introduction to regular expressions it should be noted that the perl implementation of regular expressions uses a different notation and perl contains more constructs for matching than those discussed above. However an understanding of the theoretical regular expressions will help with the understanding and usage of perl.

3.4 Finite Automata

The next model of a language recognition system we will discuss is *finite automata*. One can think of a finite automata as a machine constructed for each language.

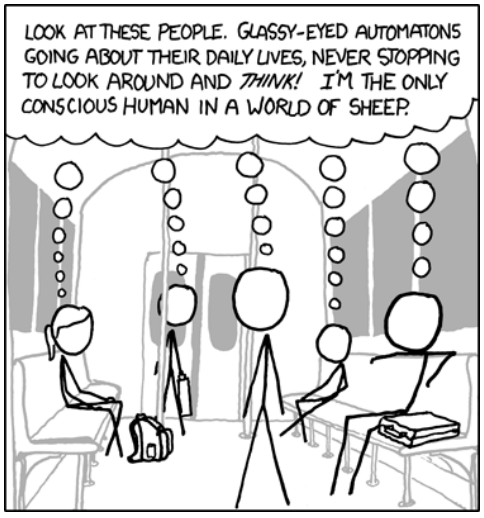


Figure 3.2: (Sheeple Automatons)XKCD : A webcomic of romance, sarcasm, math, and language.

The machine's input is a string and whose output is "yes" or "no" which indicate whether the input string is part of the language or not.

Definition 3.4.1. A *finite automata* is a collection of three things:

1) A finite number of *states* (denoted σ) one of which is denoted the *start state* and some (or none) of which are designated as *final* or *accept* states.

2) A finite *alphabet* (denoted Σ) of possible input letters. For the most part this text will use $\Sigma = \{a,b\}$.

3) A finite set of rules or *transitions* T that uniquely map every state and every letter into another state, in other words $T : \sigma \times \Sigma \to \sigma$.

To draw these finite automaton we will use the Java Formal Language and Automata Program (JFLAP). This software was developed by Susan Rodgers at Duke University. Copies can be obtained at no cost from http://www.jflap.org/. Here a state is denoted by a circle. A transition is an edge between two states. The label near the arrow indicates on what letter that transition is activated. The start state has a white triangle next to it, while an accepting state is a bullseye. When the Finite Automata tests a word, the machine begins in the start state. As each letter is read in turn, the Finite Automata changes state based on the state it is currently in and the letter read. It is only when the end of the word is reached that the finite automaton decides if the string is in the language or not. If the machine is in an accepting state, then the string is accepted, otherwise it is rejected.

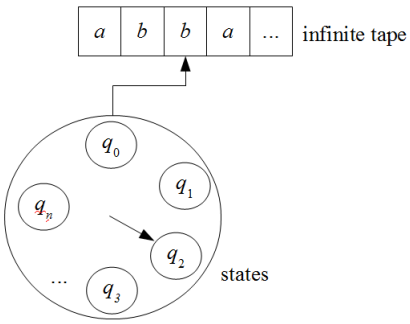


Figure 3.3: Artist's Conception of a Finite Automaton

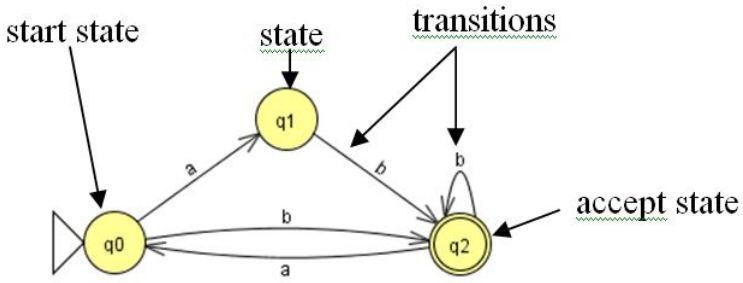


Figure 3.4: JFLAP's Finite Automaton

Note that every possible path through the finite automaton must be defined. In the case of the two letter alphabet $\{a, b\}$, this means that every state must have exactly 2 transitions leaving the state, one for when the next letter is an a and the other for when the next letter is b.

Example 3.4.1. Draw a finite automata over $\Sigma = \{a, b\}$, whose first letter is a. I have added labels to the states to make my reasoning more clear. Note that once you read the first letter the finite automaton can decide if it is in the language or not. States, such as 1 and 2 that you enter and never leave are sometimes termed "black holes" since once you enter them, you never leave. ■

Example 3.4.2. Draw a finite automaton over $\Sigma = \{a, b\}$,which contains the substring "aba". To create this finite automaton, we try to create the string "aba". This leads to the creation of states 1, 2, and 3, with the appropriate transitions (a from 0 to 1, b from 1 to 2 and a from 2 to 3), and label them appropriately. The labels refer to the suffix of the string that is usable in creating "aba". We make state 3 an accepting state and add more transitions all states so that every possible state, letter combination is defined. ■

Example 3.4.3. Draw a finite automaton over $\Sigma = \{a, b\}$,which ends in "aba". To create this finite automaton, we again made the three states 1, 2, 3 to recognize the string "aba", and made state 3 an accept state. However in this case, once we entered state 3 there is no guarantee more letters will not follow. We have to add appropriate transitions from state 3 complete the finite automaton. ■

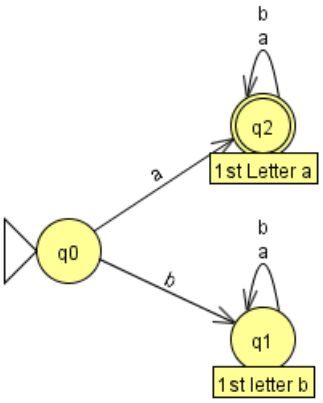


Figure 3.5: First letter is an "a" finite automaton

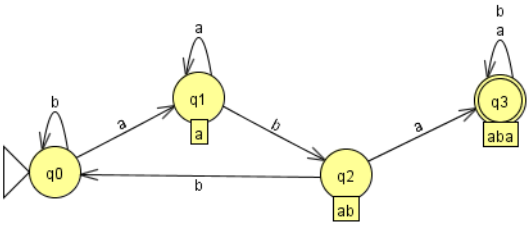


Figure 3.6: Accepts strings that contain "aba"

Example 3.4.4. In our final example we will construct a finite automaton which does not refer to a substring of the input. Draw a finite automaton over $\Sigma = \{a, b\}$,which contains an odd number of a's and an odd number of b's. In this finite automaton the states truly represent states of the input read so far. ■

Finite automaton can be used to visually represent any system that has a finite number of states. They are part of UML (universal modeling language) and used by software engineers. For example the case of writing a program to place an on–line purchase is easily finite automaton. States could include type of delivery, gift wrapping, and type of credit card. Each state would engender its own unique behaviors.

A generalization of finite automata is discussed in the next section. This model introduces non–determinism and is useful in proving the relationships between finite automata and regular languages.

3.5 Transition Graphs

Transitions graphs are a generalization of finite automata where the rules are relaxed. They introduce the notion of non–determinism and are very useful in proving Kleene's theorem.

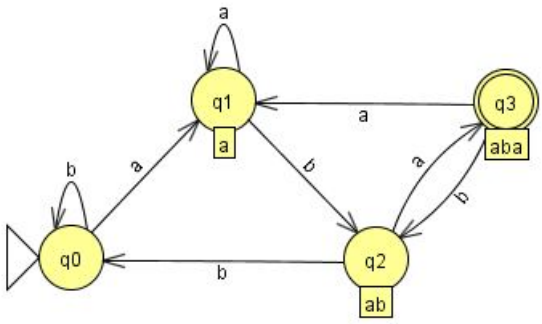


Figure 3.7: Accepts strings that end in "*aba*"

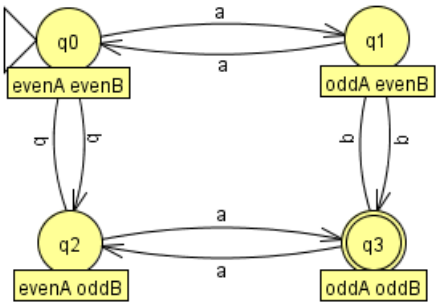


Figure 3.8: Accepts strings containing an odd # a's - odd # b's

Definition 3.5.1. A *transition graph* is is a collection of three things:

1) A finite number of *states* (denoted σ) one or **more** of which are denoted the *start states* and some (or none) of which are designated as *final* or *accept* states.

2) A finite *alphabet* (denoted Σ) of possible input letters. For the most part this text will use Σ = {a,b}.

3) A finite set of rules or *transitions* T that maps some states and some strings from Σ^* into another state, in other words $T : \sigma \times \Sigma^* \rightarrow \sigma$.

The differences between a transition graph and a finite automata is that the transition graph may have multiple start states and allow strings or λ as an edge label. A transition graph can also present you with two or more alternative destination states when reading a letter in a given state or no destination state at all. When no alternative exists, the transition graph will crash and the input string will be rejected. We say that the transition graph accepts an input string if there exists a set of choices that lead to an accepting state when the entire string has been processed.

Example 3.5.1. Consider the following transition graph. Consider the following strings and their path(s) through the transition graph:

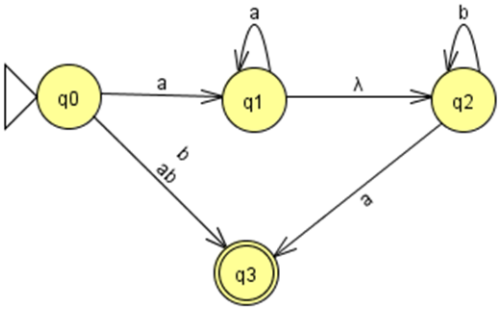


Figure 3.9: Transition Graph for Example 3.5.1

string	**paths**	**accept**
λ	0	no
a	0,1,2	no
b	0,3	yes
aa	0,1,2,3	yes
ab	0,3 or 0,1,2	yes
aba	0,3, crash or 0,1,2,3	yes
abba	0,3, crash or 0,1,2,2,3	yes
bab	0,3, crash	no

When λ is encountered on an edge, you have a choice to move to the next state, since there is a λ string between every pair of letters in a string. Note that with "aba" you had the choice of going directly to state 3 with the string "ab" and then crashing or moving to 1 after you read "a", moving to 2 immediately reading λ, looping in state 2 reading" b" and finally moving to state 3 reading "a". ■

Example 3.5.2. Consider the following transition graph.

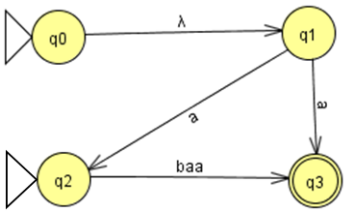


Figure 3.10: Transition Graph for Example 3.5.2

Consider the following strings from the previous example and their path(s) through the transition graph:

string	paths	accept
λ	0 or 0,1 or 2	no
a	0, 1, 2 or 2, crash or 0, 1, 3	yes
b	0,0 or 0,1 or 2, crash	no
aa	0, 1, 3, crash or 0, 1, 2, crash or 2, crash	no
ab	0, 1, crash or 0,1, 2, crash or 2, crash	no
baa	2,3 or 0, 0,1,2, crash or 0,1,3, crash	yes
abaa	0,1,3 or 0,1,2,3, crash	yes
bba	0,0,1,3 or 2, crash	yes

Here every string can start in either state 0 or state 2. So there are even more paths that need to be considered. Again, if ANY path ends in an accept state, then the string is accepted. ■

It is clear that every finite automata is a transition graph. Hence every language recognized by a finite automata is recognized by a transition graph. The question arises, can a transition graph recognize a language not recognized by a transition graph. The question of what languages are recognized by finite automata and transition graphs is explained in Kleene's theorem in the next section.

3.6 Kleene's Theorem

In this section we characterize the languages recognized and by transition graphs and finite automata.

Theorem 3.6.1. *Kleene's Theorem*
For any language L then the following are equivalent:

1) L can be described by a regular expression.

2) L can be accepted by a finite automaton.

3) L can be accepted by a transition graph.

Proof (parts 1 and 2) The proof can be divided into four parts. It is more constructive and easier to understand than many other important proofs. First, it is trivial to find a transition graph that accepts the same language as a given finite automata (FA ⇒ TG) . Given a transition graph, we will show how to find an equivalent finite automata that accepts the same language (TG ⇒ FA) . Given any transition graph, we will construct a process to find a regular expression that describes the same language (TG ⇒ RE). Finally, given any regular expression we will show how to find an equivalent transition graph that accepts the language described by the regular expression. (RE ⇒ TG) .

Part 1 (FA ⇒ TG)

Since every finite automaton is a transition graph, doing nothing to the finite automaton results in a transition graph that accepts the same language.

Part 2 (TG ⇒ FA) Consider any transition graph with non–determinism. We can simulate the non–determinism of the transition graph by creating new group states that represent the choices

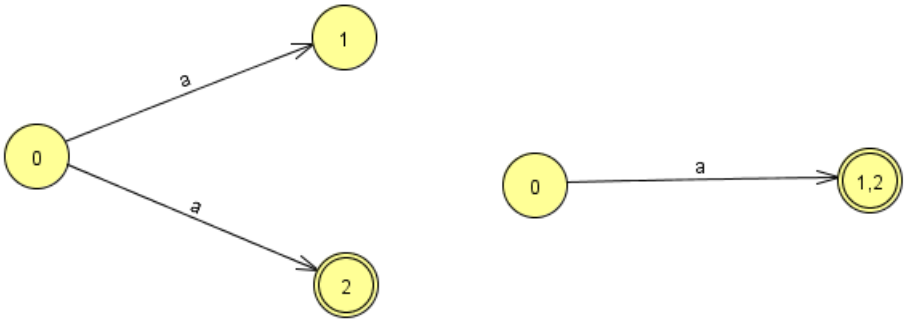


Figure 3.11: Non–Determinism Simulation with Accepting States

the transition graph has available. If at the end of the process, the simulation is in a group state that contains an accepting state, we will accept the string.

We can simulate strings on an edge by introducing new states along the edge.

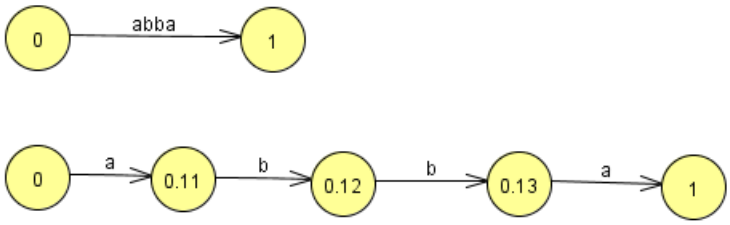


Figure 3.12: Strings on Edge Simulation

The names of the new intermediate states is arbitrary. However the convention used above can reduce confusion especially in long examples. The name of the states are of the form startState dot endState letterPostion So the name 3.53 is the third letter on the transition from state 3 to state 5. This convention can only handle 9 letter strings on an edge.

The problem of dealing with multiple start states is easily dealt with using the group state concept. One merely starts in a group starting state. The problem of dealing with λ on an edge is dealt with by considering it choice to move on to the next state. This is illustrated in the following example.

Finally if there is no edge appropriate for a given input letter, we send it to a black hole state. To avoid confusion the author will use a table of transitions to define the finite automatons. These concepts are used in the following examples.

■

Example 3.6.1. Convert the following transition graph into an equivalent finite automaton.

We will create a transition table for the equivalent finite automaton. The table will have 3

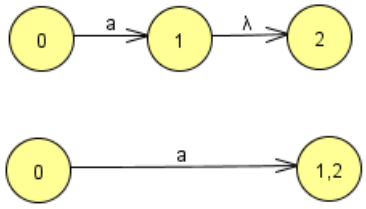


Figure 3.13: λ on an Edge Simulation

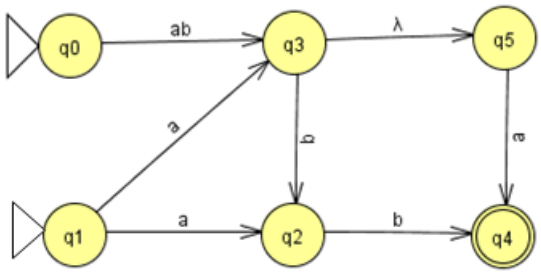


Figure 3.14: Transition Graph for Example 3.6.1

columns for the starting state, the destination state when the letter "a " is read and the destination state when letter "b" is read. BH stands for black hole state.

The simulated finite automaton will start in both states 0 and 1 initially.

If it encounters an a from state 0, it moves part way to state 3. (0.31)

If it encounters an a from state 1, it can move to state 3, move to 5 (via λ) or move to state 2.

Hence the new group state (0.31, 2, 3, 5).

If it encounters a b from state 0 or 1, it crashes, so the simulation sends it to the black hole state.

We now expand the state(0.31, 2, 3, 5), when reading an a 0.31 $\rightarrow$ crash, 2 $\rightarrow$ crash, and 5 $\rightarrow$ 4. The union of these states is 4.

When reading an b, 0.31 $\rightarrow$ 3 and $\rightarrow$ 5, 2 $\rightarrow$ 4, 3 $\rightarrow$ 2,and 5 $\rightarrow$ crash. So the union of these is 3, 4, 5.

There are no transitions leaving state 4, so the destinations are the black hole.

Expanding (2, 3, 4, 5), from a we find :

2$\rightarrow$ crash, 3 $\rightarrow$ crash, 4 $\rightarrow$ crash, 5 $\rightarrow$ 4, while under b we find:

2 $\rightarrow$ 4, 3 $\rightarrow$ 2, 4 $\rightarrow$ crash, and 5 $\rightarrow$ 4.

Finally we expand 2, 4 and the results are summarized in the table below.

start	**destination** a	**destination** b
0, 1	0.31, 2, 3, 5	BH
0.31, 2, 3, 5	4	2, 3, 4, 5
+4	BH	BH
+2, 3, 4, 5	4	2, 4
+2, 4	BH	4
BH	BH	BH

Note we start in the start state(s) and need only expand group states that appear later in the table. A + sign indicates that the group of states is an accepting state. The resulting finite automaton is pictured in Figure 3.15. ■

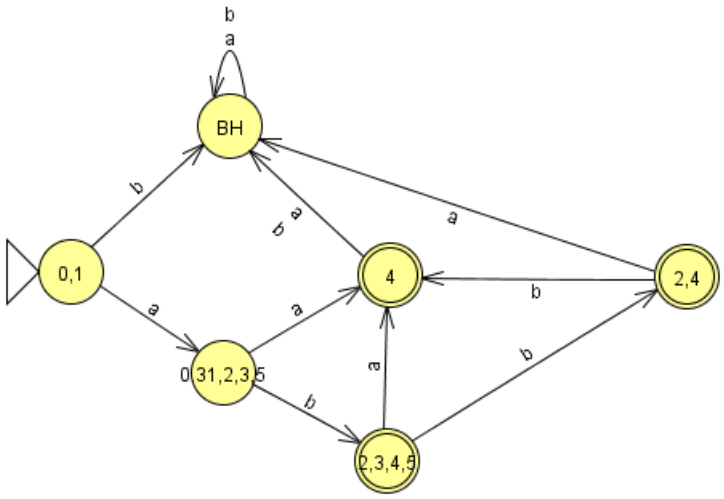


Figure 3.15: Finite Automaton for Example 3.6.1

Example 3.6.2. Convert the following transition graph to an finite automaton which accepts the same language.

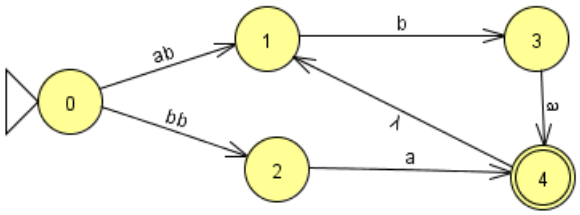


Figure 3.16: Transition Graph for Example 3.6.2

start	**destination** a	**destination** b
0	0.11	0.21
0.11	BH	1
0.21	BH	2
1	BH	3
2	1,4	BH
3	1,4	BH
+4, 1	BH	3
BH	BH	BH

In this example we have only 1 starting state, but can only traverse part way to states 1 and 2 under a single letter. From those intermediate states we move to states 1 and 2 respectively and to the black hole otherwise. The λ only plays a role when we enter state 4, and have the option to move to state 1 immediately. The finite automaton corresponding to above table, is shown below. ■

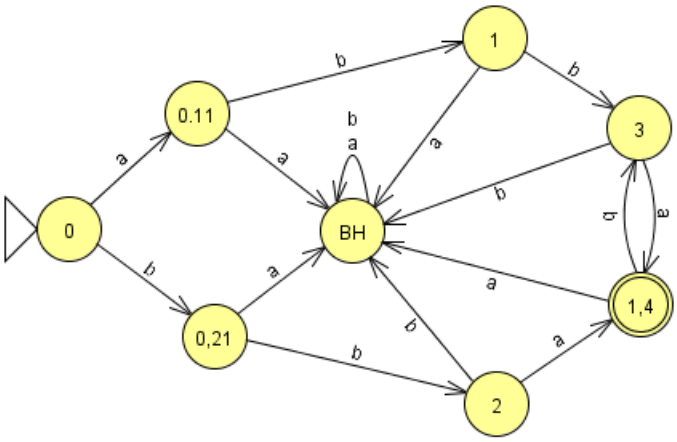


Figure 3.17: Finite Automaton for Example 3.6.2

It should be noted that the tables we generate with this method, can have 2^n rows where n is the number of states in the original transition graph. This is because every subset of the original set of states can appear as a group state in the finite automaton.

Proof (part 3 TG ⇒ RE)

We will now discuss how to find the regular expression which describes the language accepted by a given transition graph. The method will attempt to reduce any transition graph, into the following form

Note the above is not a transition graph, since a regular expression is not a legal edge label for a transition graph.

We can eliminate states in a transition graph if they appear in series. We replace the edge label between first and last state, with the concatenation of the two edges labels surrounding the

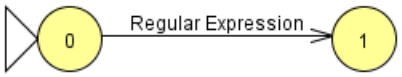


Figure 3.18: Target Form for Conversion from Transition Graph

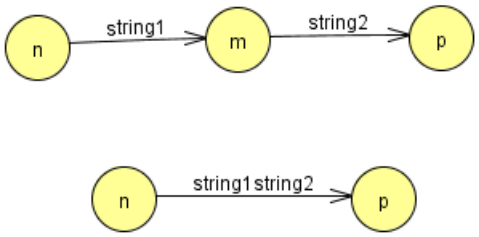


Figure 3.19: Eliminating States in a Transition Graph

middle state. (see Figure 3.19 below) We can eliminate edges in a transition graph if they appear in parallel. We replace the edge label with the union (+) of the two edge labels. (see Figure 3.20) We can eliminate loops in a transition graph, appending the edge label with a Kleene star to

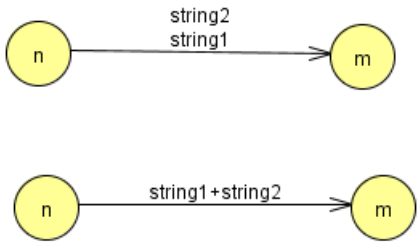


Figure 3.20: Eliminating Edges in a Transition Graph

any outgoing edge labels. (see Figure 3.21 below) We can eliminate cycles in the transition graph by converting them into loops. (see Figure 3.22 below) The above strategy of cycle elimination depends on having a sense of direction in the transition graph. In other words, knowing which state to start at, and which to end. This can be confusing especially where there are multiple start states and multiple accepting states. We can often make this clearer by creating a single start state and a single accept states which are connected to the old start and accept states with λ transitions. This is illustrated in Figure 3.23 below. We can replace individual states by replacing them with edges that account for all paths that utilize that state. The next examples put to work the entire method.

Example 3.6.3. Find the regular expression that describes the following transition graph. We can reduce the upper path to state 5 with the string ab*abab. We can reduce the cycle to a loop

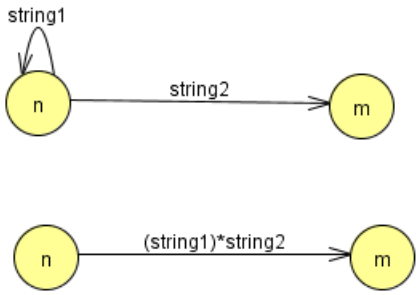


Figure 3.21: Eliminating Loops in a Transition Graph

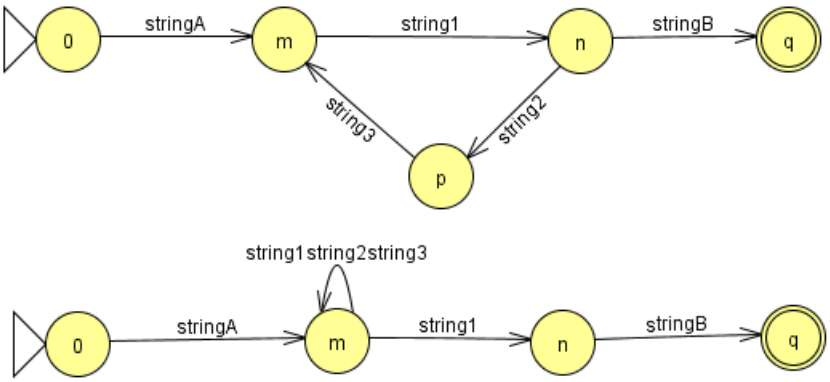


Figure 3.22: Eliminating Cycles in a Transition Graph

and the result is the following graph. By reducing the states in the lower path and combining the two regular expressions, we obtain the following graph. Hence the equivalent regular expression is ab*abab+b(aba)*abb. ■

Example 3.6.4. Find the regular expression that describes the following transition graph. We collapse the $(q3, q4, q6)$ cycle into a loop, and eliminate state $q1$. Hence the equivalent regular expression is aa*(b(aba*b)*aa + abab). ■

The following example illustrates that complexity that is possible with this method even with a simple three state transition graph.

Example 3.6.5. Find the regular expression that describes the following transition graph. In this example we need to add a new start and end state before we attempt reduction. A quick examination shows that only direct path from the start state a to the accept state z is a, 0 , 2, z. So we will attempt to eliminate state 1. We will then need to introduce new edges. We begin with an enumeration of all edges that start or leave state 1.

start	destination	label
0	0	aa
2	0	aa

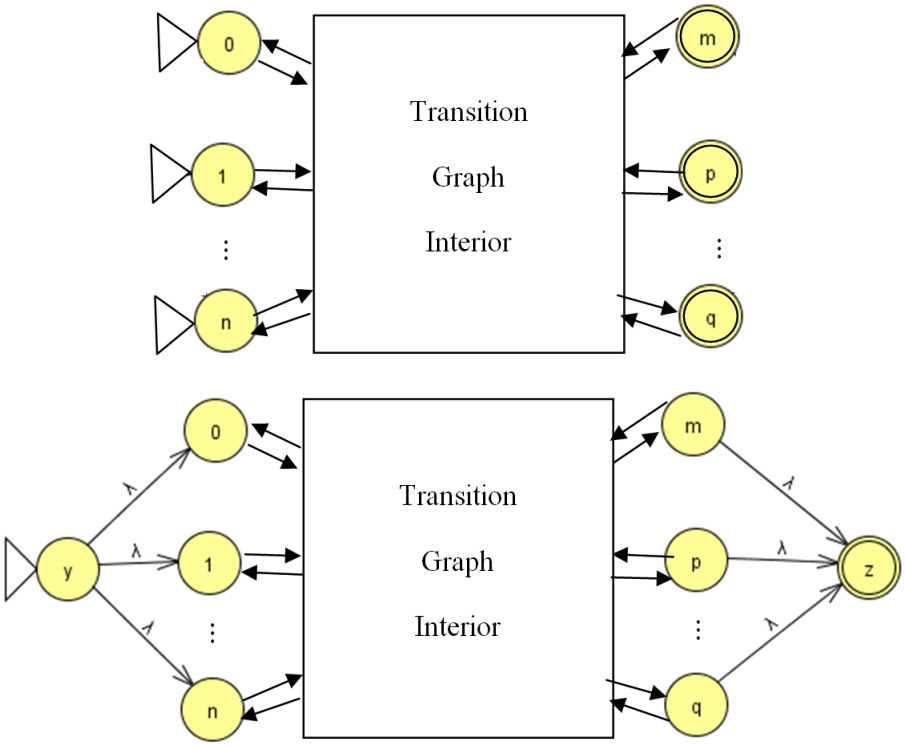


Figure 3.23: Single Start and Accept States in a Transition Graph

So the equivalent regular expression is (baa+aa)*b. ■

Example 3.6.6. Find the regular expression that describes the following transition graph. In this example we need to add a new start and end state before we attempt reduction. We will attempt to eliminate state 1. Enumerating the edges utilizing state 1 (including the loop) we produce the following table.

start	destination	label
0	0	ab*b
0	2	ab*a
3	0	bb*b
3	2	bb*a

Enumerating the edges using state 2 (including the loop) we produce the following table.

start	destination	label
0	3	ab*a + b
3	3	bb*aa

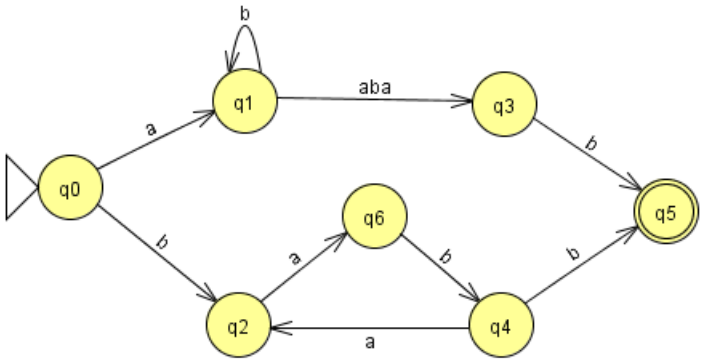


Figure 3.24: Transition Graph for Example 3.6.3

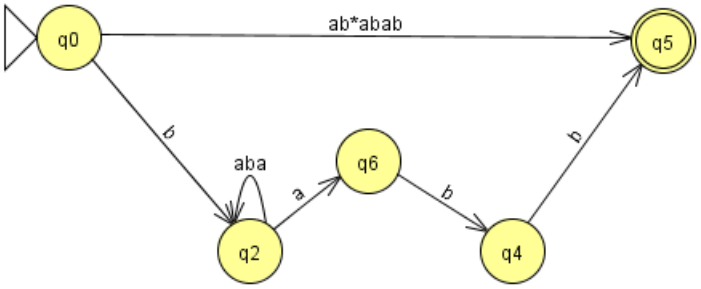


Figure 3.25: Reduced Transition Graph for Example 3.6.3

Finally we eliminate state 3.

start	destination	label
0	0	(ab*a + b) (bb*aa)*(bb*b)
0	z	(ab*a + b) (bb*aa)*

So the equivalent regular expression is ((ab*a+b)(bb*aa)(bb*aa)*(bb*b) + ab*b)*(ab*a+b)(bb*aa)
■

Proof (part 4 RE ⇒ TG) The last part of the proof will appeal to the recursive definition of regular expression. We will show that transition graphs can mimic each part of the recursive definition of regular expressions.

Recall that the definition of regular expression states that :

1) **Base Case** λ and any element of Σ is a regular expression .

2) **Recursive Step** If r_1 and r_2 are regular expressions, then so is

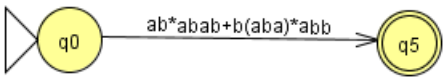


Figure 3.26: Reduced Transition Graph for Example 3.6.3

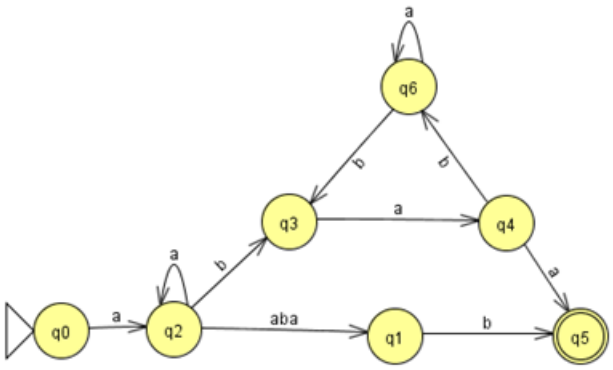


Figure 3.27: Transition Graph for Example 3.6.4

(r_1)

$r_1 r_2$

$r_1 + r_2$

r_1^*

3) **Nothing Else** Nothing not described in the base case or the recursive step are regular expressions.

Figure 3.38 displays a transition graph to accept λ or any element of Σ, where *letter* is replaced by λ or any element of Σ. Now assume transition graph T_1 accepts the language described by the regular expression r_1 and transition graph T_2 accepts the language described by the regular expression r_2 . These two transition graphs can be used to create transition graphs to accept the langauges described by $r_1 r_2$, $r_1 + r_2$ and r_1^*.

To create the concatenation transition graph for $r_1 r_2$ we make the start states of T_2 ordinary states, and draw λ transitions from the accept states of T_1 to the formerly start states of T_2 (see Figures 3.39 and 3.40). To create the union (+) transition graph for $r_1 + r_2$, we need only combine the transition graphs for r_1 and r_2 without any added edges. This disconnected graph, allows the string to start in either graph if either graph accepts it , then it is accepted, which is the definition of the union.

Finally to create a transition graph for r_1^* we modify the transition graph for r_1 by adding a new state that is both a start state and an accept state. This state will then accept λ. We then draw λ edges from the new state to the old accept states, and draw λ edges from the old accept states to our new state. This is illustrated in Figure 3.41 below. This completes the proof of Kleene‘s

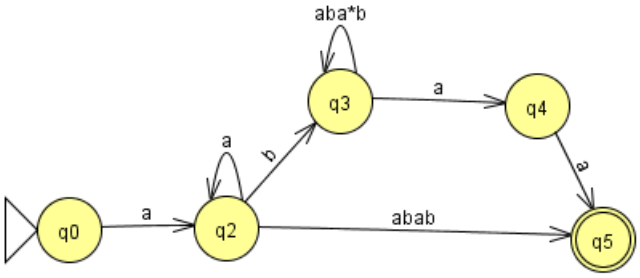


Figure 3.28: Reduced Graph for Example 3.6.4

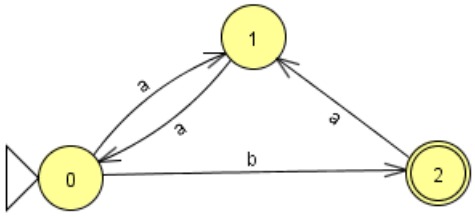


Figure 3.29: Transition Graph for Example 3.6.5

theorem. ■

3.7 Non-Regular Languages

Regular Languages (those that can be described by a regular expression or accepted by a finite automaton) are but the simplest class of languages to study. Every finite language is regular and the simple proof is given below. However there are many (infinite) number of languages that are not regular. We will present two methods for proving a language is not regular and will use these methods to introduce some non– regular languages.

Theorem 3.7.1. *All finite languages are regular.*

Proof Let the finite language $L = \{s_1, s_2 ... s_n\}$ then L can be described by the regular expression $s_1 + s_2 + ... + s_n$.

Now we will consider two methods of proving a language is not regular. The first method utilizes the pumping lemma which was proven by Yehoshua Bar-Hillel, Micha A. Perles and Eliahu Shamir in 1961.

Lemma 3.7.2. *The Pumping Lemma*

Let L be an infinite regular language. Then there exists 3 strings x, y and z, where y is not λ such that $xyz^n \in L$ for all positive integers n.

Proof Kleene's Theorem implies that there is a finite automaton A, that accepts L. Now consider a word w in L such that w has more letters than A has states. It is clear such a word exists since

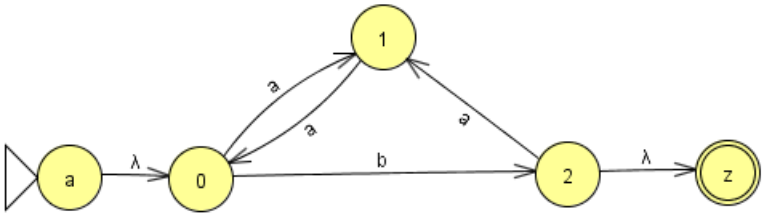


Figure 3.30: Expanded Graph for Example 3.6.5

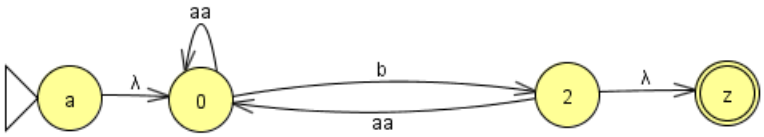


Figure 3.31: Removal of state 1 for Example 3.6.5

L is infinite and must have arbitrarily long strings. Since w has more letters than A has states, one or more of the states of A must be used more than once. This means that A has a loop that produces an accepted string. Now w can be divided into 3 parts xyz. We will let x represent the part of the string from the start state of A until the string enters the loop. We will let y represent the part of the string processed by the loop. Finally z is the part the string of from the loop until the accept state. This is illustrated in Figure 3.42 below.

Now if the A can go through the loop once, then it can traverse the it, twice. Thus $xyyz \in L$, $xyyyz \in L$, $xyyyyz \in L$ and so on. ■.

Note that to force a string into the loop, we only need to ensure that the number of letters of the string, is greater than the number of states of A. Further we remark that $|x| + |y| <$ the number of states of A. We can use these two observations to strengthen the lemma into a form that is easier to use.

Corollary. *Let L be a regular language accepted by a finite automaton A , with N states, and let $w \in L$ such that $|w| > N$. Then w can be divided into 3 substrings such that:*

$$w = xyz,\ |x| + |y| < N, |y| > 0\ ,\ \textit{and}\ xyyz \in L.$$

We will use this to prove that some languages are *NOT* regular. All these proofs will be proofs by contradiction. They have the same general pattern. We assume the language is regular and accepted by a finite automaton A with N states. We then choose a string with more that N letters and apply the corollary to generate a string ($xyyz$) that is not in L. This is a contradiction, so the language L cannot be regular. The next four examples illustrate this process.

Example 3.7.1. The string $w = a^N b^N \in L$ where N are the number of states in the A. Then the corollary to the pumping lemma implies that $w = xyz$, where $|x| + |y| < N$. Hence we know that both x and y are just a group of a's. So it is known that $x = a^r, y = a^s$ for some integers r, s where

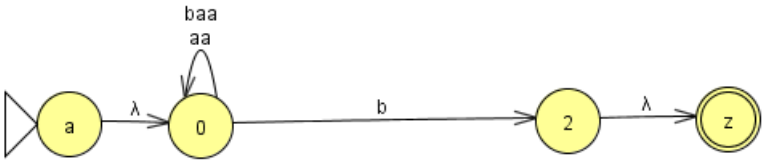


Figure 3.32: Simplified Graph for Example 3.6.5

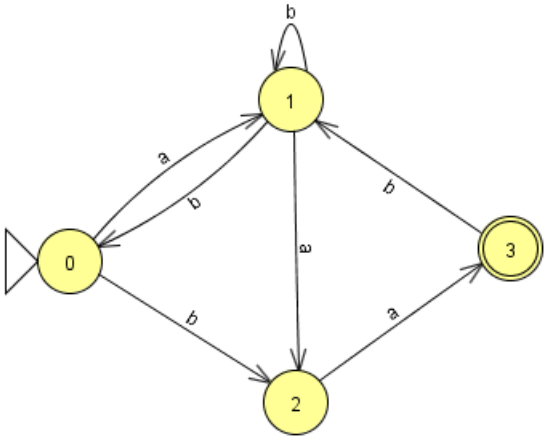


Figure 3.33: Transition Graph for Example 3.6.6

$0 \leq r < N$ and $0 < s < N$. This implies that $z = a^{N-r-s}b^N$. Since the corollary also implies that $xyyz \in L$ we see that $xyyz = a^r a^s a^s a^{N-r-s} b^N = a^{N+s} b^N$. Now we know that $s > 0$ so we know this string cannot be in the language $a^n b^n$ since the number of a's is $>$ the number of b's. This is a contradiction so L cannot be regular. ■

Example 3.7.2. Prove that the language $L = \{a^n b^n a^n\}$ for $n = 0, 1, 2, \ldots$ is not regular.

Proof Contrariwise, assume L is regular and accepted by a finite automaton A with N states. Now consider

Contrariwise, assume L is regular and accepted by a finite automaton A with N states. Now consider the string $w = a^N b^N a^N \in L$ where N are the number of states in the A. Then the corollary to the pumping lemma implies that $w = xyz$, where $|x| + |y| < N$. Hence we know that both x and y are just a group of a's. So it is known that $x = a^r, y = a^s$ for some integers r, s where $0 \leq r < N$ and $0 < s < N$. This implies that $z = a^{N-r-s} b^N a^N$. Since the corollary also implies that $xyyz \in L$ we see that $xyyz = a^r a^s a^s a^{N-r-s} b^N a^N = a^{N+s} b^N a^N$. Now we know that $s > 0$ so we know this string cannot be in the language $a^n b^n a^n$ since the number of a's is $>$ the number of b's. This is a contradiction so L cannot be regular. ■

The above example was almost identical to the previous one. Let's consider a language that looks a bit different.

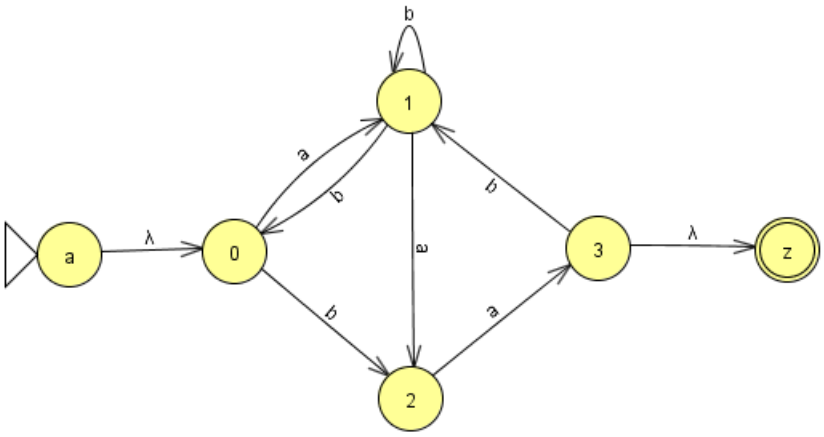


Figure 3.34: Expanded Graph for Example 3.6.6

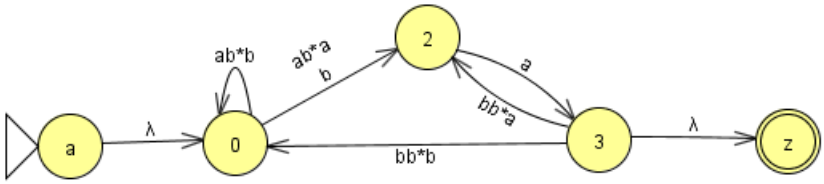


Figure 3.35: Removal of state 1 for Example 3.6.6

Example 3.7.3. Prove that the language L = set of palindromes using only $\{a, b\}$ is not regular.

Proof

Contrariwise, assume L is regular and accepted by a finite automaton A with N states. Now consider the string $w = a^N b a^N \in L$ where N are the number of states in the A. Then the corollary to the pumping lemma implies that $w = xyz$, where $|x| + |y| < N$. Hence we know that both x and y are just a group of a's. So it is known that $x = a^r, y = a^s$ for some integers r, s where $0 \leq r < N$ and $0 < s < N$. This implies that $z = a^{N-r-s} b a^N$. Since the corollary also implies that $xyyz \in L$ we see that $xyyz = a^r a^s a^s a^{N-r-s} b a^N = a^{N+s} b a^N$. Now we know that $s > 0$ so we this string cannot be in the language of palindromes since the number of a's in the front is $>$ the number of a's at the back. This is a contradiction so L cannot be regular. ■

The previous example shows that the critical part of the proof is choosing the appropriate string w. In all the examples we choose the first letter to be N letters long. Note in the previous example if we had chosen $w = a^N$, the proof would fall apart since a^{N+s} is a palindrome. It was critical that the b was chosen to be in the center of the string. In the final pumping lemma example we will need to pump multiple times.

Example 3.7.4. Prove that the language $L = \{a^n\}$ where n is a perfect square, i.e., $n = 1, 4, 9,$

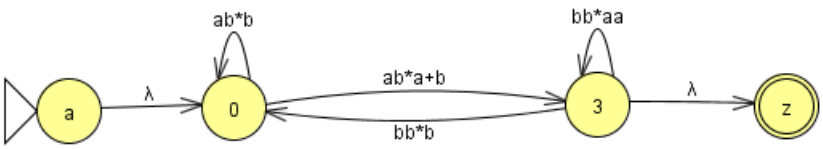


Figure 3.36: Eliminating State 2 for Example 3.6.6

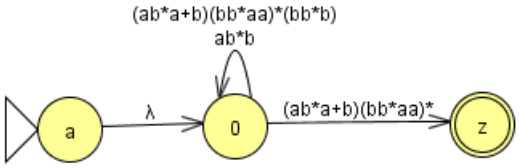


Figure 3.37: Eliminating State 3 for Example 3.6.6

... is not regular.

Proof

Contrariwise, assume L is regular and accepted by a finite automaton A with N states. Now consider the string $w = a^{N*N} \in L$ where N are the number of states in the A. Then the corollary to the pumping lemma implies that $w = xyz$, where $|x| + |y| < N$. Hence we know that both x and y are just a group of a's. So it is known that $x = a^r, y = a^s$ for some integers r, s where $0 \leq r < N$ and $0 < s < N$. This implies that $z = a^{N*N-r-s}$. Since the corollary also implies that $xyyz \in L$ we see that $xyyz = a^r a^s a^s a^{N*N-r-s} = a^{N*N+s} \in L$. So far there is no contradiction. If we pump it again we find that $a^{N*N+2s} \in L$. Now we know that $s > 0$ so we have that $N * N + s$ is a perfect square , as well as $N * N + 2s$ is a perfect square. We also know from the corollary that $|y| < N$, so $N^2 < N^2 + s < N^2 + N < N^2 + 2N + 1 = (N + 1)^2$ But this implies that $N^2 + s$ cannot be a perfect square. This is a contradiction so L cannot be regular. ■

Another way to prove languages are not regular is to use the Myhill-Nerode theorem. This theorem is more powerful than the pumping lemma since it can also be used to prove a language is regular, as well as proving a language is not regular.

Theorem 3.7.3. *Myhill-Nerode* *We will say two strings x and y are in the same suffix class for the langauge L if for all strings in $w \in \Sigma^*$, xw and yw are either both in L or both not in L.*

Then L is regular if and only if the number of suffix equivalence classes for L is finite.

We will denote the classes by square brackets surrounding a representative element of the class such as [representative element]. The following examples illustrate the idea of suffix equivalence classes.

Example 3.7.5. Let $L = a^*$. Then there are 2 suffix equivalence classes, $[\lambda] = \{\lambda, a, aa, ...\} = L$ and $[b] = L'$ (where $'$ denotes the compliment). Now for any suffix a^n for any n , strings in the first class are in L while all the strings in the second equivalence class are not in L. For any suffix not in L all strings in both class are not in L. Since there are two suffix equivalence classes, L is regular. ■

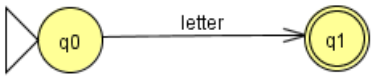


Figure 3.38: Transition Graph for the base case of the Proof

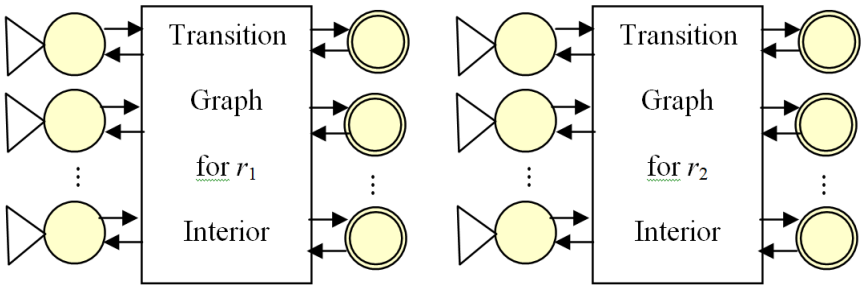


Figure 3.39: Transition Graphs for r_1 and r_2

Example 3.7.6. Let $L = a^n b^n$. Here there are an infinite number of suffix equivalence classes, $[\lambda]$, $[a]$, $[aa]$, $[aaa]$, ... $[a^i]$,.... Note that each of the above classes contain only a single string. Now for any suffix b^i, only the class containing $[a^i]$ generates a string in L . Thus there are an infinite number of equivalence classes, so L is not regular. ■

Example 3.7.7. Consider the language L = Double Word = $\{ww | w \in \{a, b\}^*\}$. Double Word is the language of all even length strings using only a or b where the first half matches the last half. Again there is an infinite number of suffix equivalence classes, $[\lambda]$, $[ab]$, $[aab]$, $[aaab]$, ... $[a^i b]$,.... Now for any suffix $a^i b$, only the class containing $[a^i b]$ generates a string in L . Thus there are an infinite number of equivalence classes, so L is not regular. ■

Although we will not give a formal proof of the theorem, we will motivate why it is true.

Informal Proof :

Let L be regular. Then from Kleene′s theorem, we know that there is a finite automaton A that accepts L. Then we know if strings x and y are in different classes then there is a string w such that exactly one of xw and yw is an element of L. So we know that when A processes strings x and y, they will end up in distinct states of the finite automaton . If there is an infinite number of equivalence classes, then A must have an infinite number of states. This contradicts the fact that A is a *finite* automaton. So if there are an infinite number of suffix equivalence classes, L must not be regular.

If there is a finite number of suffix equivalence classes, $\{[x_0], [x_1], [x_2], ..., [x_n]\}$ we will show that these suffix equivalence classes can be used to create states of a finite automaton that accepts L. We will create a state for each of the $[x_i]$. Make the state that contains λ the start state and we will set the states $[x_i]$ to accept states if $x_i \in L$.

Now create an a transition from the start state $[x_i]$ to the state that represents the class that contains $x_i a$ and we will create a b transition from the start state $[x_i]$ to the state that represents the class that contains $x_i b$. We continue this process for each of the states. It is clear that this

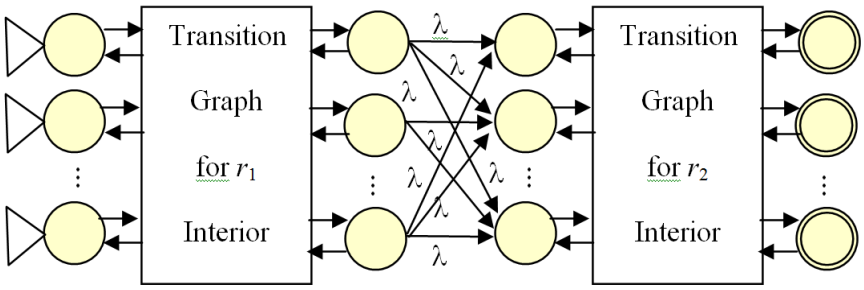


Figure 3.40: Transition Graphs for $r_1 r_2$

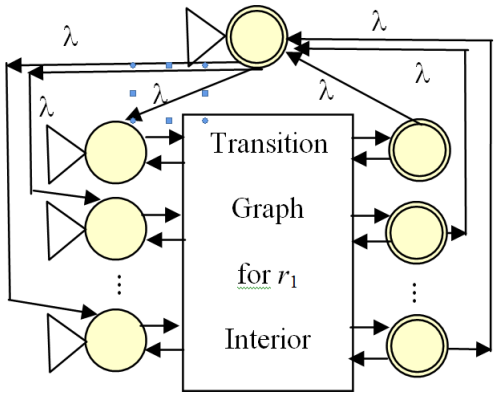


Figure 3.41: Transition Graph for r_1^*

automaton accepts L since we can create any string in the accepting states, letter by letter, starting from the state representing the λ equivalence class. The automaton we constructed is such that processing the string w results in ending up in the state represented by $[w]$. If $w \notin L$ then clearly it cannot be in the equivalence class of an accepting state, since the λ suffix distinguishes w from a string in L.

3.8 Decidability

To complete our study of regular languages, we will consider some *decision problems* . A decision problem is one that has a yes or no answer. If the problem can be solved with a finite algorithm, then that problem is called *decidable*. We will discuss the following questions:

1) Can we determine if two finite automata accept the same language?

2) Can we determine if two regular expressions describe the same language?

3) Given a finite automaton, can we determine if accepts a non–empty, finite, or infinite language?

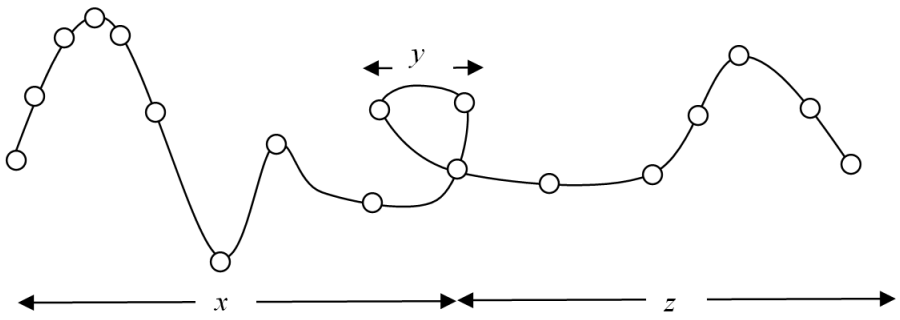


Figure 3.42: Representation of the path string w follows in the A

To determine if two automata A_1 and A_2 accept the same language, we will create an intersection, union and compliment finite automata. We can do all three at once using a process similar to the one we used to in the proof of Kleene's theorem to find a finite automaton equivalent to a transition graph. To simplify our discussion, let L_1 denote the language accepted by A_1 and let L_2 denote the language accepted by A_2. Again, we will make use group states, in this case each group state will represent one state from A_1 and one state from A_2. The idea will be to traverse both machines simultaneously. If both machines end in an accept state then the string is in $L_1 \cap L_2$. If only one machine accepts the string, then the string is in $L_1 \cup L_2$. If A_1 accepts the string while A_2 rejects the string, then the string is in $L_1 \cap L_2'$.

So if we can form the machine to accept $(L_1 \cap L_2') \cup (L_2 \cap L_1')$ and it accepts no words then we know that two finite automata accept the same language.

Example 3.8.1. Determine if the following two finite automatons accept the same language.

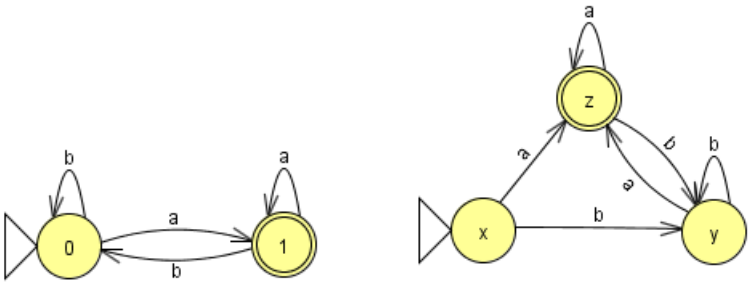


Figure 3.43: Two Finite Automatons for Example 3.8.1

We now create the table for the finite automata that accept $(L_1 \cap L_2')$ and $(L_2 \cap L_1')$. Note that the second finite automaton is labeled with letters so that in the composite machine each state will contain exactly one number and one letter.

start	**destination** a	**destination** b
0, x	1, z	0, y
1, z	1, z	0, y
0, y	1, z	0, y

Now the accepts states for $(L_1 \cap L_2')$ are $\{(1, x), (1, y)\}$, while the accepts states for $(L_2 \cap L_1')$ are $\{(0, z)\}$. Since none of these states appear in our composite machine, we know that$(L_1 \cap L_2')$ and $(L_2 \cap L_1')$ are both empty, so their union must be empty and hence two machines accept the same language. ■

The next example illustrates the procedure when the two finite automata accept different languages.

Example 3.8.2. Determine if the following two finite automatons accept the same language.

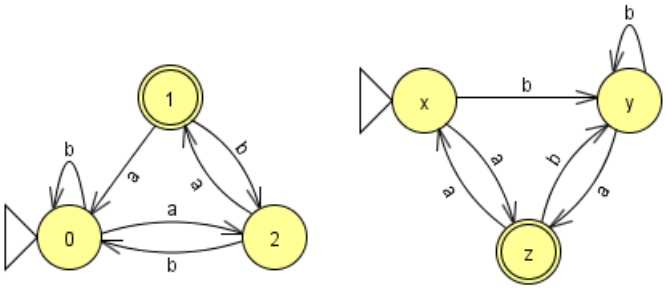


Figure 3.44: Two Finite Automatons for Example 3.8.2

Below is the table for the finite automata that accepts $(L_1 \cap L_2')$ and $(L_2 \cap L_1')$.

start	**destination** a	**destination** b
0, x	2, z	0, y
2, z	1, x	0, y
0, y	2, z	0, y
1, x	0, z	2, y
0, z	2, x	0, y
2, y	1, z	0, y
2, x	1, z	0, y
1, z	1, x	2, y

Now the accepts states for $(L_1 \cap L_2')$ are $\{(0, z), (2, z)\}$. Since $(2, z)$ can be reached from the start state under the string a, we conclude a is accepted by A_1 but not A_2. Hence two machines do *not* accept the same language. ■

To decide the second question, (determine if 2 regular expressions describe the same language), we can use the constructions we made in the proof of Kleene's theorem. We will convert each regular expression into a transition graph, convert the transition graphs into finite automata, and

finally use the decision procedure we used to answer question 1. This procedure is laborious but is certainly finite and well defined.

To decide the emptiness question (determine if a given finite automaton, accepts an empty language) we will use a "blue paint" procedure. A blue paint procedure is one where items are marked blue and then removed from consideration. Historically blue paint refers to the mark given to preprocessing tokens by the C preprocessor that temporarily disables expansion of those tokens. A token is said to be painted blue when it has been disabled in this way. While the original author of the term is disputed, the 1972 C Standard [ ] states that it came about as a reference to blue ink used by the C committee.

In this use of blue paint, we paint the start state of a finite automaton blue. Then repeatedly paint any state blue state, if there is an edge from a blue state to a non–blue state. You can think of this as the blue paint flowing along outgoing edges. In a finite number of steps, (less or equal to than the total number of states), no more states can be painted blue. If an accept state is blue, then we conclude that the machine accepts some words, while if no accept state is blue, then the machine accepts the empty language.

Example 3.8.3. Determine if the finite automaton, accepts any words. First we color the state 0

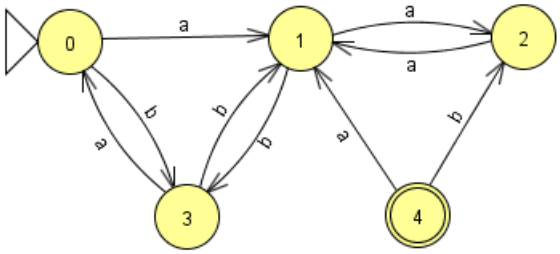


Figure 3.45: Finite Automaton for Example 3.8.3

blue. Then we color states 1 and 3 blue. Finally we color state 2. However no more states can be

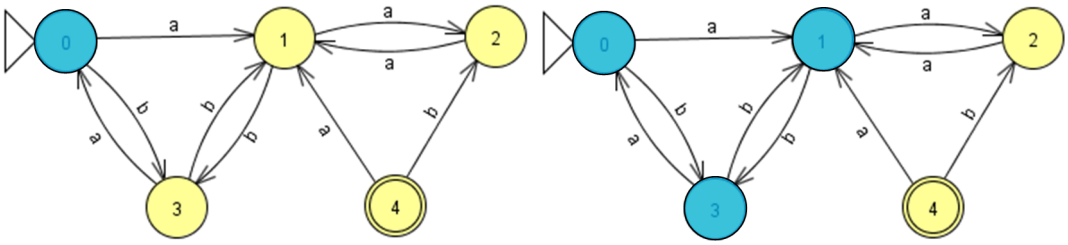


Figure 3.46: Blue Paint steps 1 and 2 for Example 3.8.3

colored blue (see Figure 3.47) . Since no accept state is painted blue, we conclude that this finite automaton accepts the empty set. ■

The final decision problem we will consider is that of *finiteness* . From the discussion of the pumping lemma and the Myhill–Nerode theorem, we know that a finite automaton accepts an

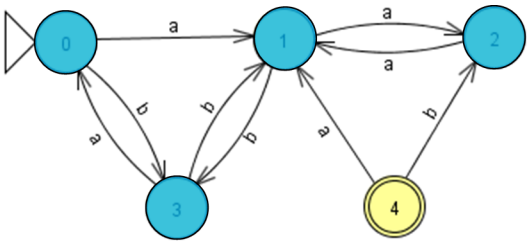


Figure 3.47: Final Painted Finite Automaton for Example 3.8.3

infinite language if and only if the machine contains a loop that is reachable from the state state and leads to a final state. So if the machine accepts any string longer than than the number of states, we know that the string has used a non–zero loop to be accepted, and the automaton accepts an infinite language.

So if we just test strings of length greater than or equal to the number of states, we can decide if it accepts an infinite language or not. As stated however the above process is not finite. We must also note that the once we have tested all strings less than twice the number of states, we can give up since no loop can be longer than twice the number of states. In summation, then if the machine accepts any string w such that $N \leq |w| < 2N$ where N is the number of states in the automaton, then the automaton accepts an infinite language .

Example 3.8.4. Determine if the following finite automaton accepts a finite or infinite set of words. Since this finite automaton accepts baa, we know that the finite automaton accepts an

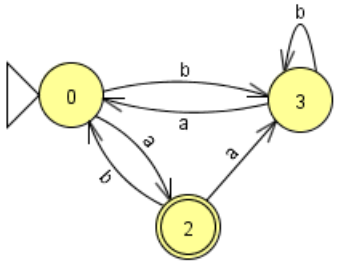


Figure 3.48: Finite Automaton for Example 3.8.4

infinite language. ■

Example 3.8.5. Determine if the following finite automaton accepts a finite or infinite set of words. Testing $\{aaaa, aaab, aaba, aabb, abaa, abab, abba, abbb, baaa, ..., aaaaa, aaaab, ..., b^7\}$ we find they are all rejected and hence the language of this machine is finite. ■

It should also be noted that this process grows exponentially with the number of states in the machine. To prove a machine with N states, that uses alphabet Σ accepts a finite language we would need to test $\Sigma^N + \Sigma^{N+1} + ... + \Sigma^{2N-1}$ strings.

Regular languages are the simplest class of the artificial languages. Regular languages are closed under intersection, unions and Kleene star. A deeper study will show that regular languages

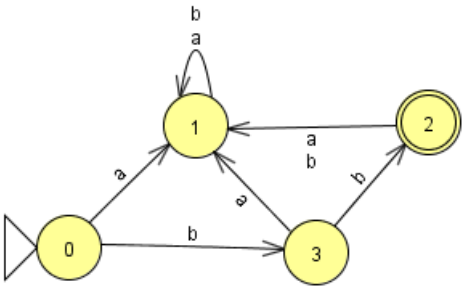


Figure 3.49: Finite Automaton for Example 3.8.5

are closed under a wide variety of operations. Also we found many decision problems concerning regular languages are decidable.

The study of regular languages sets the stage of more complex languages. Each class can be associated with a machine that will accept it as well as closure properties and decisions properties that may or may not be decidable.

3.9 Exercises

1) List the 5 shortest elements of S^* where $S = \{a, ba\}$.

2) List all words $w \in S^*$, where $S = \{a, ba, abb\}$ where $|w| < 4$

3) Write a recursive definition for odd positive integers $\{1, 3, 5, ...54321, ...86423, ...\}$.

4) Write a recursive definition for the set of positive integer powers of 3, $\{3, 9, 27, 81...\}$.

5) Write a recursive definition for the set L of all strings of the form $\{a^n b^{2n} : n \geq 0\} = \{\lambda, abb, aabbbb, aaabbbbbb, aaaabbbbbbbb, \}$.

6) Write a recursive definition for strings for odd length strings over $\{a, b\}$ which contain more a's than b's which we will call $OddMoreA = \{a, aaa, aab, aba, baa, aaaaa, aaaab.\}$.

7) Write a recursive definition for the language EndB. EndB contains all strings over the alphabet $\{a, b\}$ which end in b. EndB = $\{b, ab, bb, aab, abb, \}$.

8) Write a recursive definition for the language NoAA. NoAA is the language made up of only $a's$ and $b's$ of all strings which do not contain the substring aa.

9) Write a regular expression for the language EndB (as described above) union the language of all strings over the alphabet $\{a, b\}$ which contain bab.

10) Write a regular expression that describes the set of all strings where the total number of a's mod 3 = 0 union the set of all words that contain the substring aba.

11) Write a regular expression for all strings over the alphabet $\{a, b\}$ in which a only appears as tripled (in other words in clumps of 3) = $\{aaa, baaa, baaab, bbaaabaaa, \}$ but not $ababa$.

12) Write a regular expression for all strings over the alphabet $\{a, b\}$ in which the total number of a's is divisible by 3 no matter how they are distributed such as :

    $\{\lambda, b, bb, bbb, aaa, baaa, aaba, aaab, bbbb, abaa, aaba, aaab, baaa, abbaba, \ldots\}$.

13) Write the regular expression that describes the set of all words made up of only a's and b's which contain aa or have even length .

14) Write the regular expression that describes the set of all words made up of only a's and b's which *do not* contain aa.

15) Draw a finite automaton that accepts EXACTLY the language of all words over the alphabet $\{a, b\}$ which end in baa.

16) Draw a finite automaton that accepts EXACTLY the language of all words over the alphabet $\{a, b\}$ which end in ab or ba.

17) Draw a finite automaton that ends in aba or contains aa.

18) Which of the following strings is accepted by the transition graph pictured in Figure 3.50?

a) *bbb*

b) *abbb*

c) *a*

d) *ab*

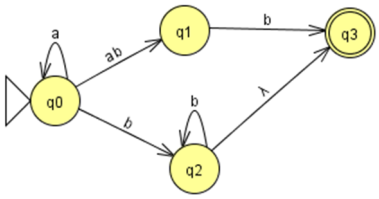


Figure 3.50: Transition Graph for Homework 18, 20, 22

19) Which of the following strings is accepted by the transition graph pictured in figure 3.9?

a) *bbb*

b) *abbba*

c) *a*

d) *ab*

20) Convert the transition graph shown in Figure 3.50 into an finite automaton by drawing a table. Be sure to label start and final states.

21) Convert the transition graph shown in Figure 3.51 into an finite automaton Be sure to label start and final states.

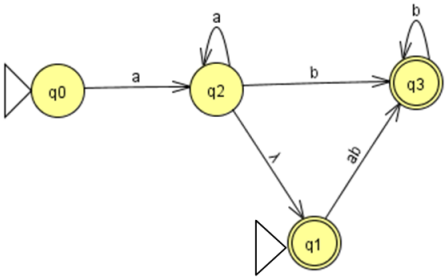


Figure 3.51: Transition Graph for Homework 21

22) Find the regular expression that describes the language of the transition graph pictured in Figure 3.50.

23) Convert the transition graph in Figure 3.52 to the equivalent regular expression

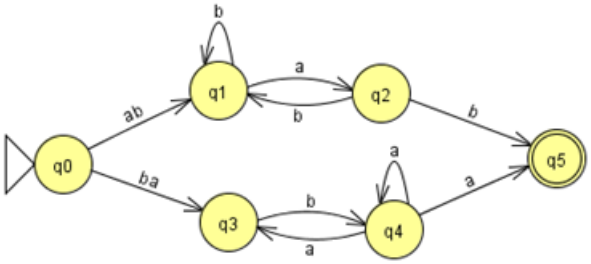


Figure 3.52: Transition Graph for Homework 23

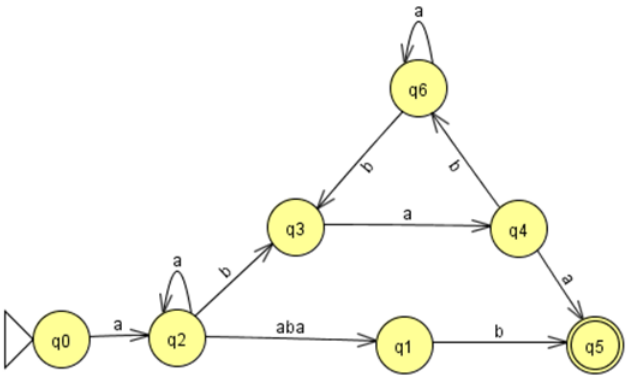


Figure 3.53: Finite Automata for Exercise 24

24) Convert the transition graph in Figure 3.53 into the equivalent regular expression.

25) Determine if the following finite automata pictured in Figure 3.54 accept the same language by making a table showing the accept states of the composite machine.

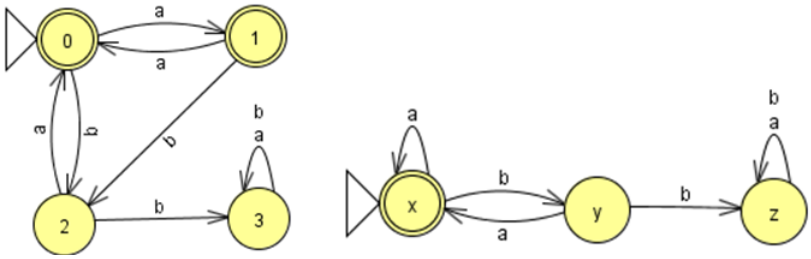


Figure 3.54: Finite Automata for Exercise 25

26) How many strings would you have to test to determine if a 5 state finite automaton (using only a and b) accepted an infinite or finite language?

27) Use the decision processes we discussed to determine if the machine in Figure 3.55 accepts an infinite(non empty), finite (non empty) or empty language.

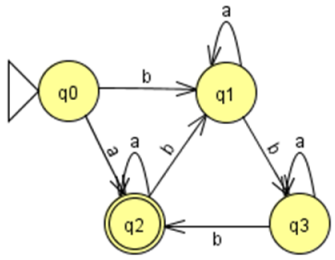


Figure 3.55: Finite Automata for Exercise 27

3.9.1 Answers to Exercises

1) $\{\lambda, a, aa, ba, aaa\}$

2) $\{\lambda, a, aa, ba, aaa, aba, aba, baa\}$

3) a. 1 is an odd positive integer.
   b. If x is an odd positive integer, so is $x + 1$.

4) a. 3 is an power of 3.
   b. If x is an power of 3, so is x*3 .

5) a. $\lambda \in L$.
   b. If $x \in L$, then so is $axbb$.

6) a. $a \in OddMoreA$.
   b. If $x \in OddMoreA$, then so is axa, axb, and bxa .

7) a. $b \in EndB$.
   b. If $x \in EndB$, then so is ax and bx .

8) a. λ and a are members of NoAA.
   b. If $x \in$ NoAA , then so is bx, xb, abx and xba.

9) $(a + b)^*b + (a + b)^*bab(a + b)^*$

10) $b^*ab^*ab^*ab^* + (a + b)^*aba(a + b)^*$

11) $(b + aaa)^*$

12) $b^*ab^*ab^*ab^*a$

13) $(a + b)^*aa(a + b)^* + ((a + b)(a + b))^*$

14) $(b + ab)^*(a + \lambda)$

15)

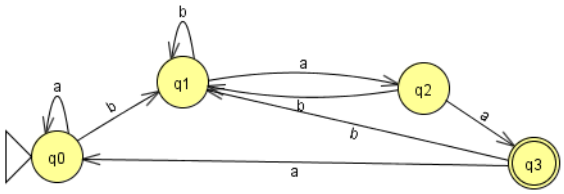


Figure 3.56: Finite Automata for Solution to Exercise 15

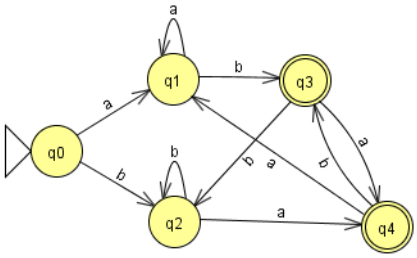


Figure 3.57: Finite Automata for Solution to Exercise 16

16)

17)

18) $bbb, abbb$, and ab

19) $abbba$ and ab

20)

states	**under** a	**under** b
0 start	0, 0.1	2, 3
0, 0.1	0, 0.1	1, 2, 3
2, 3 accept	BH	2, 3
1, 2, 3 accept	BH	2, 3
BH	BH	BH

21)

states	**under** a	**under** b
0, 1 start	0, 1, 1.3, 2	BH
0, 1, 1.3, 2	0, 1, 1.3, 2	3
3 accept	BH	3
BH	BH	BH

22) $a*(abb+bb*)$

23) $ab(b^*+ab)^*ab+ba(ba^*a)^*ba^*a$

24) $aa^*(b(abb)^*aa+abab)$

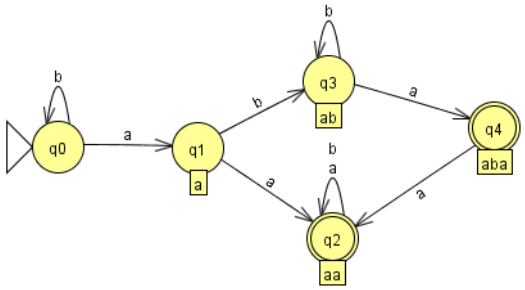


Figure 3.58: Finite Automata for Solution to Exercise 17

25)

states	**under** a	**under** b
$0x$ start	$1x$	$2y$
$1x$	$0x$	$2y$
$2y$	$0x$	$3z$
$3z$	$3z$	$3z$

The accept states for M1 $\cap$ M2' are $\{0y, 0z, 1y, 1z\}$

The accept states for M2' $\cap$ M2 are {2x, 3x}

26) $2^5 + 2^6 + 2^7 + 2^8 + 2^9$

27) This accepts an infinite language since it accepts $aaaa$.

3.9.2 Computer Activities

1) Write a simulator whose input is a finite automata in table form and a string. The simulator should print out the list of states as well as if the string is accepted or not.

2) Write a program whose input is a finite automata in table form and whose output is the equivalent regular expression.

3) Write a program whose input is a transition graph and whose output is an equivalent finite automata.

Bibliography

[1] Cohen, Daniel I.A. , "Introduction to Computer Theory, 2nd Edition," John Wiley & Sons, Inc., , (1991).

[2] McCulloch, W. S. and Pitts, W., "A Logical Calculus of the Ideas Imminent in Nervous Activity," *Bulletin of Mathematical Biophysics*, **5**, pg 115 –133, (1943).

[3] Kleene, S. C., "Representation of Events in Nerve Nets and Finite Automata," in Shannon, C. E., and McCarthy, J. (eds) , *Automata Studies*, Princeton Univ. Press, Princeton, NJ, pg 3 –42, (1956).

[4] Myhill, J., "Finite Automata and the Representation of Events," Wright Air Development Center Technical Report **57–642**, Wright Patterson Air Force Base, OH, pp. 112- 137, (1957).

[5] Bar–Hillel, Y., Perles, M. and Shamir E. , " On Formal Properties of Simple Phrase Structure Grammars, " Y. Bar–Hillel (ed.), *Language and Information*, Addison–Wesley, Reading, MA pp. 116 - 150, (1964).

[6] Jones, Derek M. . The New C Standard (Excerpted material): An Economic and Cultural Commentary, (2008).

Chapter 4

Grammars and Context Free Languages

Grammar, which knows how to control even kings. Moliere
Grammar is the logic of speech, even as logic is the grammar of reason. Richard Chenevix Trench

4.1 Grammars and Trees

Before we explore the next class of languages, we introduce grammars and trees. Grammars are the primary tool in the construction of artificial languages, such as those in programming languages, web page construction or job control scripts. They are even used in artificial languages between people such as Esperanto (the international language project) and Lojban the logical language. (see Figure 4.1) The idea for the loglan/lojban project was to create a language which is both highly expressive as well as culturally neutral as possible. It is an experiment to use the power of language to affect the way we think.

Figure 4.1: XKCD: A webcomic of romance, sarcasm, math, and language.

For example consider a subset of English where sentences are defined as a *noun* followed by

a *verb* followed by a period. Then given a list of nouns { dog, cat, Bob, pan} and a list of verbs {walks, talks, barks, throws } We can make several sentences such as:

dog barks.
Bob talks.
pan talks.

The grammar gives the form or *syntax* of the sentences but does not deal with the meaning of legal sentences. In this case many meaningless sentences can be constructed. The meaning of the sentences is called the *semantics*. This is equivalent to the errors found by a compiler. These compiler errors indicate the form of the program does *match* the rules of the programming language's grammar (syntax). When a program has a run–time error, then the semantics is wrong.

More formally we can think of this grammar as a collection of three parts. The set of terminal words { dog, cat, Bob, pan, walks, talks, barks, . } which can be used to replace the non–terminals $\{noun, verb, S\}$ and the rules:

$$\begin{aligned} S &\rightarrow noun\ verb\ . \\ noun &\rightarrow \text{dog} \\ noun &\rightarrow \text{cat} \\ noun &\rightarrow \text{Bob} \\ noun &\rightarrow \text{pan} \\ verb &\rightarrow \text{walks} \\ verb &\rightarrow \text{talks} \\ verb &\rightarrow \text{barks} \end{aligned}$$

One uses the rules for replacing the left hand side of the arrow with the right hand side of the arrow. One can apply any rule in any order starting with the S symbol, until only the terminal strings are left. These rules and the set of terminals and non–terminals define all legal sentences. Rules such as those given above may be familair to computer science students who have looked at the grammar of a programming language. An example of a legal if–statement in c, c++ or java might be :

$$< \text{if\_statement} > \rightarrow \text{if } (< \text{boolean\_expression} >) < \text{statement} > ;$$

In the above the non–terminals are { < if_statement >, < boolean_expression>, < statement > } and the terminals are { if, (,), ; }.

We now formalize these concepts and apply them to strings, not sentences. In this way we will use the grammar to generate a language.

Definition 4.1.1. A **grammar** for a language is a collection of three things.

1) A finite set of terminal characters, Σ (Sigma).

2) A finite set of non–terminal characters, Γ (Gamma), one of which is denoted S for the starting non–terminal. When it is possible we will capitalize the non–terminals to distinguish them from terminals.

3) A finite set of rules also termed **productions** for replacing expressions containing non-terminals.

A **context–free grammar** is a grammar where all the productions are of the form non–terminal $\rightarrow$ finite string of terminals and non–terminals

Definition 4.1.2. A **context–free language** is the language generated by a context–free grammar.

Definition 4.1.3. The process of using productions to convert the starting non–terminal S into a string of terminals is called a **derivation**. We will use the $\Rightarrow$ to denote that a production has been used.

Example 4.1.1. Let $\Sigma = \{a, \lambda\}$ and let $\Gamma = \{S\}$ and let the two productions be

$S \rightarrow Sa$

$S \rightarrow \lambda$

Using the above, we can derive that :

$S \Rightarrow \lambda$

$S \Rightarrow Sa \Rightarrow \lambda a = a$

$S \Rightarrow Sa \Rightarrow Saa \Rightarrow \lambda aa = aa$

We can conclude that the language generated by this grammar is $\{\lambda, a, aa, aa, ...\} = a^*$. ■

Example 4.1.2. Let $\Sigma = \{a, b\}$ and let $\Gamma = \{S\}$ and let the three productions be

$S \rightarrow aS$

$S \rightarrow bS$

$S \rightarrow abba$

Using the above, we can derive that :

$S \Rightarrow abba$

$S \Rightarrow aS \Rightarrow aabba$

$S \Rightarrow bS \Rightarrow babba$

$S \Rightarrow aS \Rightarrow aaS \Rightarrow aaabba$

$S \Rightarrow bS \Rightarrow baS \Rightarrow baabba$

We can quickly see that the language generated by this grammar is $\{abba, aabba, babba, ...\} = (a + b)^*abba$. In other words this is the language of any string (using only a and b that ends in $abba$. ■

It is past time to introduce some **BNF** (**Backus Normal Form** or **Backus Naur Form**). BNF is just a slight variation in notation for defining a grammar that saves time and space when defining a grammar. In BNF we denote the two productions

$N \rightarrow$ stuff1

$N \rightarrow$ stuff2

with $N \rightarrow$ stuff1 | stuff2

The | sign can be considered an "or" between productions. It can be used as many times a needed. For example in the grammar in Example 4.1.2 can be denoted:

$S \rightarrow aS|bS|abba.$

Example 4.1.3. Let $\Sigma = \{a, b, \lambda\}$ and let $\Gamma = \{S\}$ and let the two productions be

$S \rightarrow aSb|\lambda$

Using the above, we can derive that :

$S \Rightarrow \lambda$

$S \Rightarrow aSb \Rightarrow a\lambda b = ab$

$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aa\lambda bb = aabb$

$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb = aaa\lambda bbb = aaabbb$

Then the language generated by this grammar is $\{ab, aabb, aaabbb, ...\} = a^n b^n$. This language proves that context–free grammars can generate non–regular languages. ■

4.1.1 Trees

Another important representation of a derivation of a word from a grammar uses a tree. These trees are used in a variety of applications and are denoted **syntax trees**, **parse trees**, **generation trees**, **production trees** and **derivation trees**. For computer scientists one of the most important of these is the parse tree, which is used by compilers to give meaning to a computer program.

The tree starts with the starting S non–terminal and illustrates the productions used to form a single string. Consider a derivation $S \Rightarrow Sa \Rightarrow ba$. This would be drawn as the following tree.

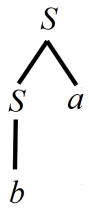


Figure 4.2: Parse Tree for $S \Rightarrow Sa \Rightarrow ba$

The leaves of the tree concatenated together from left to right form the generated string ba.

Example 4.1.4. Consider a more complex derivation. The non–terminals in this example are { S, X, Y} and the terminals are $\{a, b\}$. In this derivation the left most non–terminal will be replaced next. This is called a **left–most derivation**.

$S \Rightarrow aXYb \Rightarrow aaYb \Rightarrow aabXb \Rightarrow aabab$.

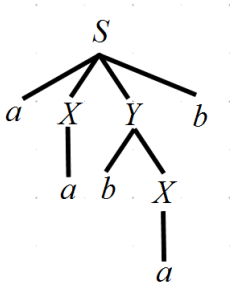


Figure 4.3: Parse Tree for derivation of $aabab$

■

Another type of tree can also be used to show all the words generated by a grammar. This is called a **total language tree**. The interior vertices contain **working strings**, (strings that contain both terminal and non–terminal characters), while the leaves (terminal vertices) contain only terminal characters. In the following example the leaves are in bold face.

Example 4.1.5. Draw the portion of the total language tree for a following grammar.

$S \to AB|aA$,

$A \to a$,

$B \to ba$.

The total language tree is pictured in Figure 4.4 where the terminal strings are in boldface.

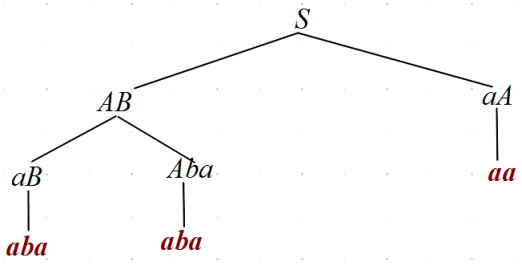


Figure 4.4: Total Language Tree

■

Example 4.1.6. Draw the portion of the total language tree including all strings of less than 5 letters for a following grammar.

$S \to aaS|Sb|b$

The relevant portion of the total language tree is pictured in Figure 4.5 where the terminal strings are in boldface.

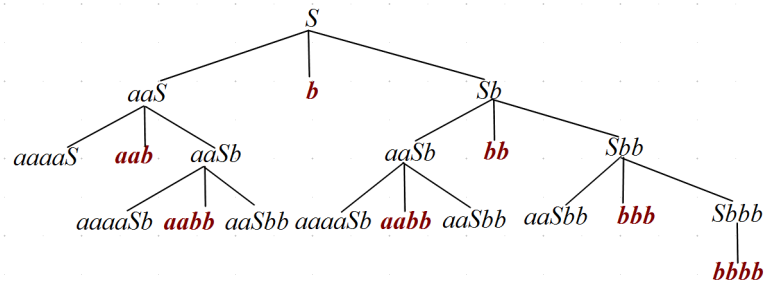


Figure 4.5: Total Language Tree of strings $w : |w| < 5$

■

Given a parse tree one can determine the effect of the productions which were used but not the order in which the productions were used. Sometimes the effect of the productions is not unique

in that a different sequence of productions can be used to generate the same string. This property of unique sequence of productions for each string is studied in the following subsection.

4.1.2 Ambiguity

To understand the meaning of a set of code, a compiler generates a parse tree and assigns values to the leaves. However if a string has two distinct parses, then the string can have two distinct meanings. Clearly this is an undesirable property of grammars. We formalize this concept below.

Definition 4.1.4. A grammar is called **ambiguous** if there exists a string w which can be generated with two distinct left–most derivations.

Example 4.1.7. Consider the grammar defined by $\Gamma = \{S\}$, $\Sigma = \{a\}$ and with the productions
$S \to aS$
$S \to Sa$
$S \to a$
Then the string aaa can be derived in four distinct left–most derivations:
$S \Rightarrow aS \Rightarrow aaS \Rightarrow aaa$
$S \Rightarrow aS \Rightarrow aSa \Rightarrow aaa$
$S \Rightarrow Sa \Rightarrow aSa \Rightarrow aaa$
$S \Rightarrow Sa \Rightarrow Saa \Rightarrow aaa$
The parse trees for these derivations are given below:

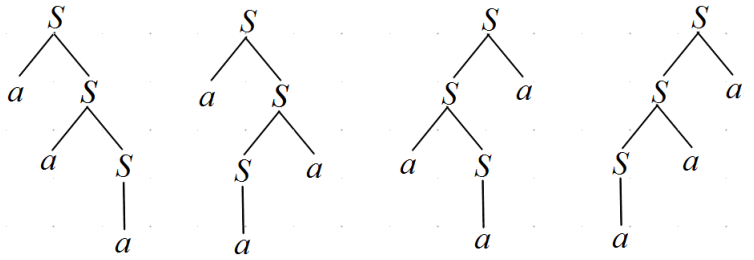


Figure 4.6: Parse Trees for the derivation of aaa

Since there are multiple derivations aaa, we can conclude that this grammar is ambiguous. ■

Example 4.1.8. Prove the following grammar with the following productions is ambiguous.
$S \to AB|aaB$,
$A \to aA|a$,
$B \to b$.
In order to prove a grammar is ambiguous, we need find a word with two distinct left–most derivations. In this case it is clear that repeated use of $A \to aA$ can be used with $S \to AB$ to mimic the production $S \to aaB$.
Consider the word aab. We will show two left–most derivations of this word.
$S \Rightarrow aaB \Rightarrow aab$ and
$S \Rightarrow AB \Rightarrow aAB \Rightarrow aaB \Rightarrow aab$. ■

Example 4.1.9. Prove the following grammar with the following productions is ambiguous.
$S \to aSb|Sb|Sa|a$,
In this case we can use of $S \to Sb$ and $S \to Sa$ to mimic the production $S \to aSb$.
Consider the word aab. We will show two left–most derivations of this word.
$S \Rightarrow aSb \Rightarrow aab$ and
$S \Rightarrow Sb \Rightarrow Sab \Rightarrow aab$. ■

Example 4.1.10. Prove the following grammar with the following productions is ambiguous.
$S \to bSS|SSb|SbS|a$,
In this case we can use first two productions and just mix the order.
Consider the word $baaab$. We will show two left–most derivations of this word.
$S \Rightarrow bSS \Rightarrow baS \Rightarrow baSSb \Rightarrow baaSb \Rightarrow baaab$
$S \Rightarrow Sb \Rightarrow Sab \Rightarrow baaab$. ■

4.2 Context Free Languages

In this section we explore the nature of context free languages. We have seen that several regular languages can be generated with a context free grammar. The first question to consider is whether all regular languages can be described by a context free grammar.

Theorem 4.2.1. *All regular languages are context free.*

Informal Proof:
Consider a regular language L. By Kleene's theorem, we know that there exists a finite automaton A which accepts L. We now present an algorithm for building a context free grammar to generate L based on the finite automaton A. Each state will represent a non–terminal. Label the start state S and for the remainder of the states label the i^{th} state X_i. Now if there is an edge from state X_i to state X_j under letter σ we add the production:
$X_i \to \sigma X_j$
If X_i is an accept state then add the production:
$X_i \to \lambda$
Now the productions mimic the finite automaton. Every time we transition from state X_i to state X_j reading the letter σ, we add letter σ to the front of the working string . We terminate the working strings when we reach a final state. ■

The context free grammars generated by the above proof have a special form in that all productions are of the form : $N_1 \to \sigma N_2$ or $N \to \lambda$, where $N, N_1, and N_2$ are non–terminals.

Grammars of this form are called **regular grammars** since they generate regular languages.

Example 4.2.1. Find a context free grammar for language of strings over $\{a, b\}$ which end in aba.
The finite automaton for this language is given in Figure 4.7 , where we have labeled the states with capital letters.
Using the algorithm outlined above, we can quickly find the grammar from this automaton as:

$$S \to bS|aX \ ;$$

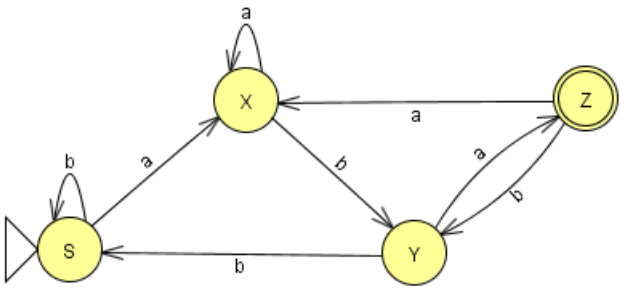


Figure 4.7: Finite Automaton for ends in aba

$$X \to aX|bY \ ;$$
$$Y \to bS|aZ \ ;$$
$$Z \to aX|bY|\lambda.$$

Recall $Z \to \lambda$ since Z is an accepting state. Before we leave, it is well worth noting that this method does not always yield the simplest grammar. In this case it is clear that the following grammar which uses only 1 nonterminal will also generate the langauge of words that end in aba.

$S \to aS|bS|aba$ ■

Example 4.2.2. Devise a context free grammar for the regular language over the alphabet $\{a, b\}$ which contain an even number of a's and an odd number of b's. In this case, a grammar is not as intuitive as in the last case. Many students find it easier to begin with the finite automaton pictured below in Figure 4.8.

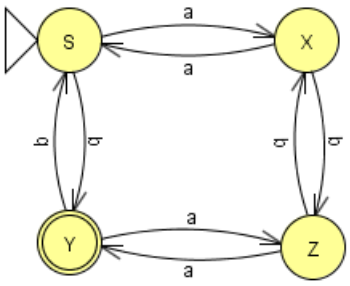


Figure 4.8: Finite Automaton for even a's, odd b's

with the corresponding grammar.

$$S \to aX|bY \ ;$$
$$X \to aS|bZ \ ;$$
$$Y \to aZ|bS|\lambda \ ;$$

$$Z \to aY|bX.$$

■

We have seen that context free grammars can also describe non–regular languages. It is not a simple task to generate grammars for these languages but studying some simple ones can help. In languages where we are matching the number of one string with another, we need to link them with a production of the type

$N \to string1Nstring2$;

Example 4.2.3. Write a context free grammar for the language $a^n b^* a^n$, $n = 1, 2, ... = \{aa, aba, abba, aabaa, aabbaa, abbbba, aaaabbaaaa, ... \}$

We will use a production involving the non–terminal S to make the matching a's and another nonterminal X to make the unmatched b's.

$$S \to aSa|aXa;$$
$$X \to bX|\lambda$$

■

Another tip that can help in the creation of context–free grammars is to use concatenation of non–terminals to represent different parts of the words in the grammar.

Example 4.2.4. Devise a context free grammar for the language $a^n b^n a^m b^m, n, m = 1, 2, 3, ...$

Here we recall that the grammar for $a^n b^n$ is given by

$$S \to aSb|ab$$

The language of this example is essentially two of the aforementioned grammars concatenated together.

$$S \to XY$$
$$X \to aXb|ab$$
$$Y \to aYb|ab$$

■

4.2.1 Chomsky Normal Form

There are several "normal" forms for context free grammars. These are restrictions on the type of productions that can be included in the grammar. They are normal in the sense that every context free language can be generated by a grammar in normal form. By using Chomsky normal form we will be able to determine the size of the string based on the number of productions used in its creation. Chomsky normal form (named for Noam Chomsky []) is especially useful in that it can be used to simplify some important proofs about context free languages. Dr. Chomsky was a linguistics professor at MIT and is credited with the theory of generative grammars and the establishment of the Chomsky hierarchy, which is a classification of formal languages. He was the most cited author in the years 1980 – 1992 and he was also known for his political activism. His quick wit and wide expertise made him a frequent guest of the political comedian Bill Maher.

Definition 4.2.1. A grammar is in **Chomsky normal form** if all the productions are of the form :

$$N \to N_1 N_2$$
or
$$N \to t$$
or
$$S \to \lambda$$

for non–terminals N, N_1, N_2 and terminal character t.

We will now outline a 3 step method for converting any context free grammar into Chomsky normal form.

Step 1. Remove all null productions

In this step remove all productions of the form $N \to \lambda$. We do this by identifying which non–terminals can be replaced with λ. We call these non–terminals as nullable. We will remove all such productions and add in new productions where the nullable non–terminal is replaced with λ. If S is nullable we will create a new start non–terminal S^* and add the production $S^* \to S|\lambda$ at the end.

Example 4.2.5. Remove the λ productions from the grammar

$S \to aX|bX, X \to a|b|\lambda$

It is clear that X here is nullable, since $X \to \lambda$. Removing the λ production and replacing each of the nullables with λ results in the following :

$S \to aX|bX|a|b, X \to a|b$ ■

Example 4.2.6. Remove the λ productions from the grammar

$S \to XaY|bX, Y \to XX|aY, X \to aX|\lambda$

It is clear that X here is nullable, since $X \to \lambda$, however Y is also nullable since $Y \to XX$ and each of the X's can be replaced by λ. Hence when we remove the λ production from X, we must add the resulting non–unit productions to both X and Y. Removing the λ production and replacing each of the nullables with λ results in the following :

$S \to XaY|bX|aY|Xa|a, Y \to XX|X|aY|a, X \to aX|a$ ■

Step 2. Remove all unit productions

Unit productions are productions of the form $N1 \to N2$. These productions are just using one non–terminal as an alias for another. This will confuse our calculations and need to be removed. The process is straightforward in that we just replace $N1 \to N2$ with $N1 \to non-unit\ productions\ of\ N2$. However it can be complicated if there is a "chain" of units such as $N1 \to N2$ and $N2 \to N3$. In this case we remove the unit productions and add the productions for $N1 \to$ all the non–unit productions of both $N2$ and $N3$.

Example 4.2.7. Remove the unit productions from the grammar :
$S \to aX|bY; X \to S; Y \to bY|b;$

non–terminal	unit	non–unit
S		$aX\|bY$
X	S	
Y		$bY\|b$

Replacing the unit production $X \to S$ with $X \to$ to the non–units of S we convert the grammar to the form :

$S \to aX|bY; X \to aX|bY; Y \to bY|b;$ ■

Example 4.2.8. Remove the unit productions from the grammar :

$S \to aX|bY|Y; X \to S; Y \to Yb|b;$

non–terminal	unit	non–unit
S	Y	$aX\|bY$
X	S	
Y		$Yb\|b$

Here there are three unit productions. $S \to Y$, $X \to S$, and $X \to S \to Y$.
So the grammar becomes :

$S \to aX|bY|Yb|b; X \to aX|bY|Yb|b; Y \to aX|bY|Yb|b;$ ■

Step 3. Convert inappropriate right hand sides
We remove productions of the form $N \to$ *stuff* where *stuff* is not a 2 nonterminals or not a single terminal with the creation of new non-terminals that partially reconstruct the production. For example if the production is $N \to abba$ then it can be replaced with :

$N \to XY$, $X \to AB$, $Y \to BA$, $A \to a$, $B \to b$

where X, Y, A and B are non–terminals not already used in the grammar.

Example 4.2.9. Convert the following grammar without unit productions and without λ productions into Chomsky normal form.

$S \to aY|bX; X \to aY|a; Y \to bX|b.$

Here we need to convert aY into two non–terminals by changing a to A and adding the production $A \to a$. We follow a similar procedure for bX and the result follows.

$S \to AY|BX; X \to AY|a; Y \to BX|b; A \to a; B \to b.$ ■

The next example illustrates how to convert right hand sides that are too long.

Example 4.2.10. Convert the following grammar without unit productions and without λ productions into Chomsky normal form.

$S \to XYS|XY|a; X \to aX|b; Y \to bY|a$

Here we need to convert aX and bY along with XYS. The first two right hands are converted as in the previous example, but the last one requires a new non–terminal (say N) to be replaced by either XY or YS. The result is :

$S \to NS|XY|a; X \to AX|b; Y \to BY|a; A \to a; B \to b; N \to XY.$
■

The next example incorporates all three steps of the process.

Example 4.2.11. Convert the context free grammar $S \to XSX|aY; X \to Y|S; Y \to b|\lambda$
into Chomsky normal form.

Step 1. Remove all null productions.

Y is nullable, since $Y \to \lambda$ but since $X \to Y \to \lambda$, X is nullable also. Replacing X and Y with λ individually where ever they occur and removing $Y \to \lambda$ results in the following equivalent grammar .

$S \to XSX|SX|XS|aY|a; X \to Y|S; Y \to b$

Step 2. Remove all unit productions.

We will make a table of non–terminals and identify the unit productions and the non–unit productions.

non–terminal	**unit**	**non–unit**
S		$XSX\|SX\|XS\|aY\|a$
X	$Y\|S$	
Y		b

So we will replace $X \to Y$ with the $X \to$ non–unit productions of Y, and do a the same for $X \to S$. The result is :

$S \to XSX|SX|XS|aY|a$; $X \to b|XSX|SX|XS|aY|a$; $Y \to b$.

Step 3. Convert inappropriate right hand sides.

We will need to introduce new non–terminals to replace XSX and aY. We will let $N \to XS$ and $A \to a$.

The final equivalent grammar in Chomsky normal form is :

$S \to NX|SX|XS|AY|a$; $X \to b|NX|SX|XS|AY|a$; $Y \to b$; $N \to XS$; $A \to a$. ■

4.3 Pushdown Automata

Just as finite automata are the machines that accept regular languages, there exists a machine to accept context free languages. These are called **pushdown automata**. A pushdown automaton is essentially just a finite automaton with a stack added. The automaton has the option to read or write to the stack and to make decisions (change states) based on the information it has stored on the stack. We will also relax the rules that transitions must form a function, (i.e. we will all the machine to reject a string by crashing). Again if there is a path through the machine that ends in an accept state, the string is accepted. There are actually several variations of pushdown automata, and we will use the one implemented in JFLAP. A formal definition of pushdown automata is given below.

Definition 4.3.1. A **pushdown automaton** is a collection of four things :

1) A finite alphabet Σ of input characters.

2) An finite set of states Ω at least one of which is denoted as a start state.

3) A finite set of stack characters Π (one of which is Z).

4) A finite set of transitions between states of the form read, pop; push.

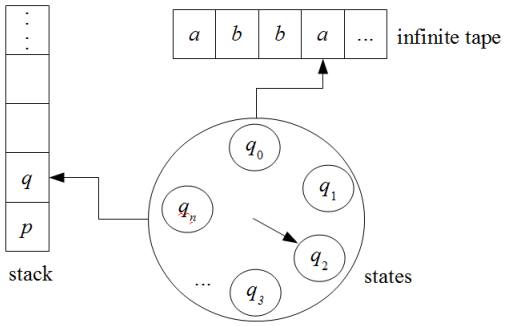


Figure 4.9: Artists Conception of a push down automaton

It should be noted that pushdown automata are non–deterministic. This matches the non–determinism inherent in grammars. Analogous to transition graphs, if the machine has any path that enters an accept state on a given input string, we will determine that the machine accepts that string. We will explore pushdown automata with a few examples.

Example 4.3.1. Build a pushdown automaton that accepts the language of all words over $\{a, b\}$ that end in aba. This is a regular language will not need to use the stack.

This pushdown automaton is essentially the same as the finite automaton for the language but has to run out of letters before it enters the accept state. It does not change the stack at all. ■

Example 4.3.2. Build a pushdown automaton that accepts the language of $a^n b^n$. Since this language is not regular we must make use of the stack. In this case the stack will hold the number of a's read - the number of b's read.

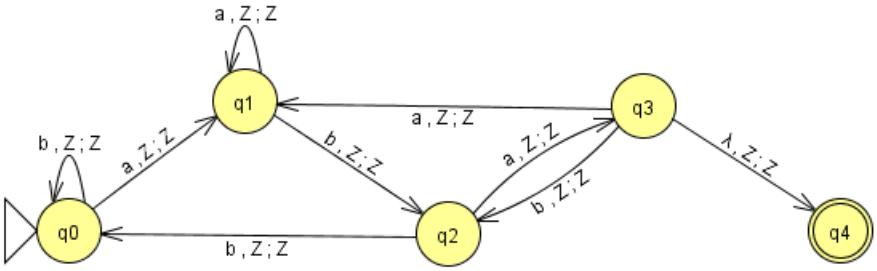


Figure 4.10: Pushdown automaton for ends in *aba*

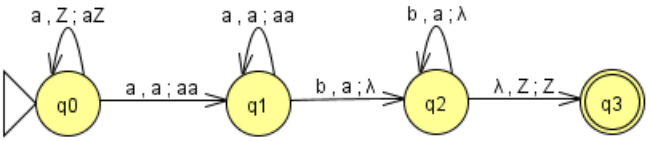


Figure 4.11: Pushdown automaton for $a^n b^n$

■

In our final example we will create a pushdown automaton that works non–deterministically.

Example 4.3.3. Build a pushdown automaton that accepts the language of even palindromes over the alphabet $\{a, b\}$. In this case the stack will hold the first half of the input string. Since a stack is a last–in/ first–out data structure, it will nicely match the letters surrounding the middle.

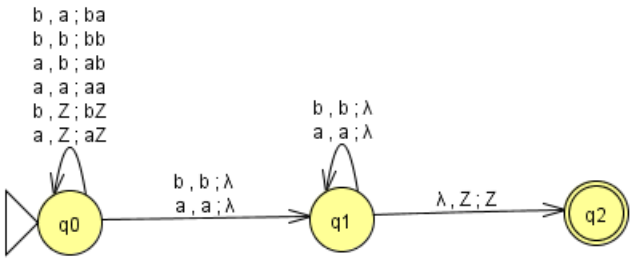


Figure 4.12: Pushdown automaton for Even Palindrome

Many students find it confusing that the pushdown automaton knows when to stop pushing (reading the first half of the string) and start popping (reading the last half of the string. The machines does not really know, but does both. Below is the JFLAP simulation after reading the

last letter of the input *abba*. Note that there are two distinct final positions that the machine can end up in. In the first situation (state= q_0 , stack = $abbaZ$), the machine never stopped pushing letters onto the stack. In the second situation (state= q_2 , stack = Z) it stops half way and pops the last half of the input string. (Note that it can only stop when the letter it reads matches the letter on the top of the stack).

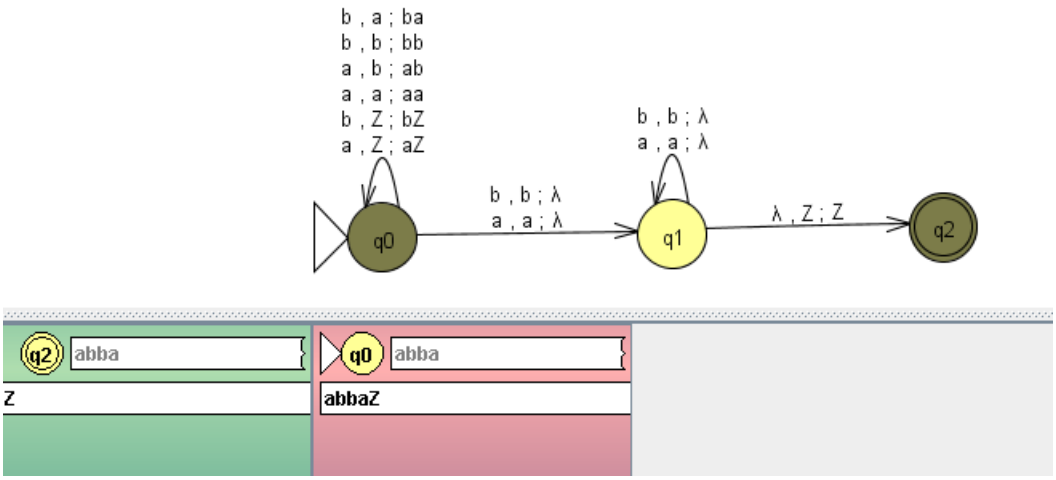


Figure 4.13: Pushdown Simulation for Even Palindrome on input *abba*

■

It is now shown that there is a non–deterministic pushdown automaton for every context free language.

Theorem 4.3.1. *There exists a pushdown automaton that accepts every context free language.*

Informal Proof Let L be a context free language. Then there exists a context free grammar in Chomsky normal form that generates L. The pushdown automaton that simulates the grammar will have just three states. A start state, a middle state and an accepting state. The stack will hold the leftmost non–terminal to be processed.

The start state will be used to push the start state S onto the stack. The middle state will handle implementing all the productions, while the final state will terminate the process when all non–terminals have been replaced. The start state will transition to the middle state by reading nothing, popping nothing and pushing S (the start state) onto the stack. The middle state will process all the productions. Recall that in Chomsky normal form, there are just two types of productions: a non-terminal can be replaced with two non–terminals, or a terminal can be replaced with a terminal.

If the production is of the form $N1 \rightarrow N2N3$, (where $N1, N2$, and $N3$ are non–terminals), the transition from the middle state to itself will be read λ , pop $N1$, push $N2N3$. If the production is of the form $N \rightarrow t$, (where N is a non–terminal and t is a terminal), then transition from the middle state to itself is read t, pop N and push λ.

Finally there is a transition from the middle state to the accepting state that reads λ, pops Z (the symbol on the stack indicating end of stack), and pushing Z. ■

The next couple of examples will illustrate this procedure.

Example 4.3.4. Find a non–deterministic pushdown automaton that accepts the language generated by the following grammar in Chomsky normal form:

$$S \to AS|a; A \to a;$$

Since there are three productions there will be three transitions from the middle state to itself. The two productions that read an a will result in transitions that read a. The production $S \to AS$ will pop A and push on AS.

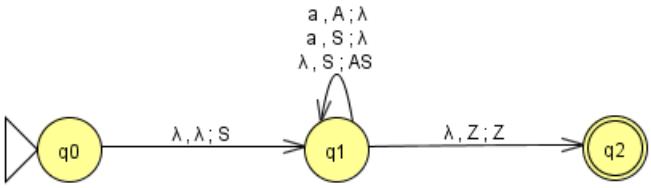


Figure 4.14: PDA for the above grammar.

■

4.4 Closure Properties

A closure property is an operation which when applied to arbitrary elements of the set, produce another element of the set. For example, the integers are closed under addition, multiplication, and subtraction but not division. We will explore some of the closure properties of context free grammars.

We will quickly prove some closure properties of regular languages and then discuss the properties of context free languages. The closure proofs of regular languages will make use of either finite automata, transitions graphs or regular expressions. We will build a description of the desired language based on the descriptions of the underlying languages. Regular languages are simple languages and are closed under a large variety of operations.

Theorem 4.4.1. *Regular languages are closed under unions.*

Proof

Let L_1 and L_2 be regular languages described by the regular expressions r_1 and r_2, respectively. Then $L_1 \cup L_2$ is described by the regular expression $r_1 + r_2$.

■

Theorem 4.4.2. *Regular languages are closed under complements.*

Proof The complement of a language L is the set of all strings composed of letters in the alphabet of that language, that are *NOT* in L and is denoted L'.

Let L be a regular language accepted by the finite automaton A. (We know that A exists by Kleene′s theorem.) Now we can modify A to form the finite automaton A' which accepts L' by making all the accept states non–accept states and vice–versa.

■

Theorem 4.4.3. *Regular languages are closed under intersections.*

Proof

Let L_1 and L_2 be regular languages. Then De Morgan′s laws tell us that $L_1 \cap L_2 = (L_1' \cup L_2')'$. Since regular languages are closed under unions and complements, they are also closed under intersections.

■

Theorem 4.4.4. *Regular languages are closed under concatenations.*

Proof

Let L_1 and L_2 be regular languages and let R_1 and R_2 be the regular expressions that describe L_1 and L_2 respectively. Then $(R_1)(R_2)$ is a regular expression describes the concatenation of strings in L_1 with strings in L_2 . Hence regular languages are closed under concatenations.

■

Theorem 4.4.5. *Regular languages are closed under Kleene star.*

Proof

Let L be a regular language and let R be the regular expression that describes L. Then $(R)^*$ describes the language in L^* Hence regular languages are closed under Kleene star.

■

Unlike regular languages which are closed under intersection, union , concatenations, and complements, context free languages are only closed under unions and concatenation. In these proofs we will make use of either the underlying context free grammars or the push down automata.

Theorem 4.4.6. *Context free languages are closed under unions.*

Proof

Let L_1 and L_2 be context free languages generated by the context free grammars
$S_1 \rightarrow$ stuff1 ; ... and $S_2 \rightarrow$ stuff2 ; ...,

where S_i is the starting non–terminal for L_i. Then the $L_1 \cup L_2$ is generated by the context free grammar with the starting non–terminal S with the productions :

$$S \rightarrow S_1 | S_2;$$

$$S_1 \rightarrow \text{stuff1 ;... and } S_2 \rightarrow \text{ stuff2 ; ...}$$

■

Theorem 4.4.7. *Context free languages are closed under concatenations.*

Proof
Let L_1 and L_2 be context free languages generated by the context free grammars

$S_1 \rightarrow$ stuff1 ; ... and $S_2 \rightarrow$ stuff2 ; ...,

where S_i is the starting non–terminal for L_i. Then the L_1L_2 is generated by the context free grammar with the starting non–terminal S with the productions :

$S \rightarrow S_1S_2$; $S_1 \rightarrow$stuff1 ;... and $S_2 \rightarrow$ stuff2 ; ...

■

To show that context free languages are not closed under intersections, we will need to prove some languages are not context free. We will prove some languages are not context free with the aid of pumping lemma in the next section.

4.5 Non-Context-Free Languages

To prove a language is not context free we will use a pumping lemma for context free languages. Again the idea is that if we have a string in a context free language that is sufficiently large, then we can create more strings in the language by "pumping" the string. In this situation we will use the grammar representation of the language. We know that when language is not context free, it will be infinite. (All finite languages are regular and hence context free.) For a grammar to produce an infinite language there must be a derivation where some nonterminal is self–embedded. In other words a derivation of the form

$$S \rightarrow ... \rightarrow \text{stuff } N \text{ stuff } ... \rightarrow \text{newStuff } N \text{ newStuff } ... \rightarrow \textit{terminals}$$

Since the grammar is finite, then the only way to produce an infinite number of words is for at least non-terminal to be self–embedded. Our "pump" in this situation will be embed a non–terminal more than once. So the strategy will be find a large enough word that guarantees self–embedding and then introduce another self–embedding to generate a longer word that is also in the language.

To better understand the growth of strings, we will assume that the grammar is in Chomsky normal form. Recall that in Chomsky normal form the productions are restricted to two types, namely :

$N_1 \rightarrow N_2N_3$; one non–terminal being replaced by two non–nonterminals

or

$N \rightarrow t$; one non-terminal being replaced by a single terminal.

We will call the first type of production a "live" production. If we call the string of terminals and non–terminals a "working string", then the use of a live production will grow the working string by exactly 1 letter. The use of the other type of productions will not increase the length of the working string at all. If the terminal is λ the working string will actually shrink. We are now ready to state and prove the pumping lemma for context free languages.

Theorem 4.5.1. *Let L be an infinite context free language generated by the context free grammar in Chomsky normal form with m non–terminals. Let $s \in L$ such that $|s| > m$. Then s can be divided into 5 parts u, v, x, y, z such that*

1) $s = uvxyz$

2) $|vxy| \leq m$

3) $|vy| > 0$

4) $uv^n xy^n z \in L$ *for every integer* $n > 0$

Informal Proof Consider an infinite context free language L that is generated by a context free grammar in Chomsky normal form with m live productions. Consider any string $s \in L$ whose length is greater than m. First we claim that some non–terminal has been used twice in the production of s. For each letter of s we use at least one new non–terminal. So since there are only m non–terminals and $s > m$ one non–terminal has been used more than once. Hence the derivation tree for s looks like the tree in Figure 4.15.

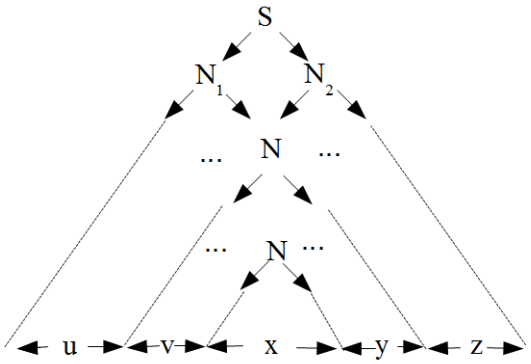


Figure 4.15: Derivation Tree for s.

The substring u represents the part of s before the self–embedded non–terminal has been used. The substring v represents the part of s after the self–embedded non–terminal has been used for the first time but before it has been used for the second time. The substring x represents the part of s generated by the use of the self–embedded non–terminal. The substring y represents the part of s after the self–embedded non–terminal has been used for the second time but not generated by the first use of the non–terminal. . The substring z represents the part of s after the self–embedded non–terminal has been used for the first time.

If N is the only self–embedded non–terminal then we know that $|vxy|$ must be less than the number of non–terminals. Now if we can self–embed N once, we can do it again generating another vxy in the middle of our string so that uv^2xy^2z is also part of L. Repeating this argument n times gives the desired result.

■

We will now use this lemma in a proof by contradiction to show $L = a^n b^n a^n$ is not context free.

Example 4.5.1. $L = a^n b^n a^n$ is not context free. Assume that $L = a^n b^n a^n$ *is* context free and generated by a context free grammar in Chomsky normal form with N non–terminals. Consider the word $s = a^{N+1}b^{N+1}a^{N+1}$ The pumping lemma tells us that s can be written in the form $s = uvxyz$, where u, v, x, y, and z are substrings, such that ,$|vxy| \leq N$, $|vy| > 0$ and $uv^i xy^i z$ is $\in L$ for every integer i . By our choice of s and the fact that $|vxy| \leq N$, it is easily seen that the substring vxy can contain no more than two distinct groups of letters. That is, we have one of five possibilities for vxy:

1) $vxy = a^j$ for some $j \leq N$.

2) $vxy = a^j b^k$ for some j and k with $j + k \leq N$.

3) $vxy = b^j$ for some $j \leq N$.

4) $vxy = b^j a^k$ for some j and k with $j + k \leq N$.

5) $vxy = a^j$ for some $j \leq N$.

For each case, it is easily verified that $uv^i xy^i z$ does not contain equal numbers of each letter for any . Thus, uv^2xy^2z does not have the form $a^i b^i a^i$. This contradicts the definition of L. Therefore, our initial assumption that L is context free must be false. ■

It should be noted that like the pumping lemma for regular expressions, this pumping lemma can only prove languages are not context free. It cannot be used to prove a language is context free. However we can now prove that context free languages are not closed under intersection.

Theorem 4.5.2. *Context free languages are not closed under intersection.*

Proof
Consider the langauge $L_1 = a^i b^i a^j$ and $L_2 = a^i b^j a^j$, where $i, j > 0$.
Both these langages are context free since they can be generated by the context free grammars

$$S_1 \to XA \ ; \ X \to aXb|ab \ ; \ A \to aA|a$$
$$S_2 \to AX \ ; \ X \to bXa|ba \ ; \ A \to aA|a$$

where S_1 is the starting symbol for the context free grammar for language L_1 and S_2 is the starting symbol for the context free grammar for language L_2.

Now $L_1 \cap L_2 = a^i b^i a^i, i > 0$ which was shown above not to be context free.
■

4.6 Decidability

There were a multitude of properties which are decidable for regular languages. The same is not true for context free languages. However we can decide, given a context free grammar if it generates any words, an infinite number of words and if a given word is generated.

To decide if a grammar generates any words, we can use a procedure akin to blue paint. However we start at the terminals. For any production whose right hand side is all terminals we replace that

non–terminal with its right hand side. We then replace that non-terminal where ever it occurs with the terminal string that it can be replace it. We repeat this process until either the start state is replaced with all terminal characters, (indicating that the grammar produces at least one string) , or until no more replacements are possible and we conclude the grammar produces no words.

Example 4.6.1. Decide if the following context free grammar generates any words or not.

$$S \to XY; X \to AX|AA; Y \to BY|BB; A \to a; B \to b$$

Here B can be replaced with b and A can be replaced with a. Making these replacements yields:

$$S \to XY; X \to aX|aa; Y \to bY|bb; A \to a; B \to b$$

Now X can be replaced with aa and Y can be replaced with bb. Applying these replacements we find :

$$S \to aabb; X \to abb|aa; Y \to baa|bb; A \to a; B \to b$$

Hence we know that aabb is one word generated by this grammar. ■

The next example illustrates a grammar which produces no words.

Example 4.6.2. Decide if the following context free grammar generates any words or not.

$$S \to XY; X \to AX; Y \to BY; A \to a; B \to b$$

Here B can be replaced with b and A can be replaced with a. Making these replacements yields:

$$S \to XY; X \to aX; Y \to bY; A \to a; B \to b$$

However in this case the process cannot be continued, so no words are generated by this grammar. ■

To determine if a grammar produces an infinite number of words we will need to find an embedded non–terminal that we can use effectively. There are two conditions that must be met for a non-terminal to be effective. It must be able to be replaced by a string of terminals, and it must be reachable from the start symbol. The replacement condition can proceed analogously to the procedure for determining if a language produces any words. One can consider the procedure for testing if a grammar generates any words to testing (in part) the effectiveness of S, the start symbol. We will remove any non–terminals that cannot be replaced by all terminals from the grammar as being ineffective.

The second part of effectiveness deals with reachability from the start non–terminal. We can test for reachability by painting the start state blue, and then if any non–terminal is blue we paint the right hand side non-terminals blue. We repeat this until no more non–terminals can be colored blue. We will remove all non–painted non–terminals.

Example 4.6.3. Reduce the following grammar to only effective non–terminals.

$$S \to Aba|BAZ|b; A \to Xb|bZa; B \to bAA; X \to aZa|aaa; Z \to Zaba$$

First we will see what non–terminals can be replaced by all terminals.

$$S \to b$$
$$X \to aaa$$

$$A \to Xb \to aaab$$
$$B \to bAA \to baaabaaab$$

Hence S, X, A and B can all be terminated, while Z cannot. Hence Z is removed. Now we remove all unreachable non–terminals. The resulting reduced grammar is :

$$S \to Aba|b; A \to Xb; B \to bAA; X \to aaa;$$

S is always reachable. Since $S \to Aba$, A is reachable. Since $A \to Xb$, X is reachable. B is not reachalbe so the grammar can be reduced to :

$$S \to Aba|b; A \to Xb; X \to aaa$$

■

The following is another example of reducing a grammar to only it′s effective productions.

Example 4.6.4. Reduce the following grammar to only effective non–terminals.

$$S \to XS|b; X \to YZ|ab; Y \to ab; Z \to XY$$

Again we begin by removing non–terminals that do not terminate.

$$S \to b$$
$$X \to ab$$
$$Y \to ab$$

Z cannot reduce to a string of terminals so it can be removed. The resulting grammar is :

$$S \to XS|b; X \to ab; Y \to ab;$$

It is clear from inspection that Y is no longer reachable so the grammar reduces to :

$$S \to XS|b; X \to ab;$$

■

Once the grammar is reduced,we will look for any set of productions that can lead to a non–terminal being self–embedded. The authors can offer no clever algorithm for this process. The process is best seen as an example.

Example 4.6.5. Determine if there is any self–embedding in the following reduced grammar.

$$S \to XY|b; X \to YZ|ab; Z \to XY; Y \to ab$$

By inspection we see that $X \to YZ \to YXY$.
So X is self–embedded, and the language is infinite. ■

Example 4.6.6. Determine if there is any self–embedding in the following reduced grammar.

$$S \to ABa|b; X \to bA|aaa; A \to Xb; B \to bAA$$

Again by inspection we see that $A \to Xb \to bAb$.
So A is self–embedded and the language is infinite. ■

The final decision problem we will consider is that of membership. Given a string and a context free grammar determine if that string is produced by the grammar. The method presented below is basically intelligent brute force. We will enumerate all strings produced by the grammar in a table. We will use the table to more quickly generate new and longer strings. The method presented below is a variation of the CYK - algorithm. This algorithm uses "*dynamic programming*", in other words it uses the solution of smaller instances of the problem to help solve larger instances of the problem.

First convert the language to Chomsky normal form. Then make a table with the number of rows equal to the length of the string whose membership is in question. The number of columns of the table is the number of non–terminals in the grammar. The algorithm will generate strings of increasing length. Since the grammar is in Chomsky normal form the algorithm needs only consider strings from 2 columns (2 non–terminals) to produce new strings.

Example 4.6.7. Determine if the string $abbaa$ is generated from the language described by the following context free grammar in Chomsky normal form :

$$S \to XY|b; X \to SY|bb|a; Y \to SS|aa$$

We begin by setting up the table and filling in any terminal string that that are produced directly from a non–terminal.

# letters	S	X	Y
1		a	
2		bb	aa
3			
4			
5			

Now using the 1 letter strings, we create 2 letter strings. (In this case no new strings are created.) Now using the 2 and 1 letter strings we create the 3 letter strings.

# letters	S	X	Y
1		a	
2		bb	aa
3	aaa		
4			
5			

Now we fill in row 4 with either pairs of 2 letter strings or 1 letter strings with 3 letter strings.

# letters	S	X	Y
1		a	
2		bb	aa
3	aaa		
4	$bbaa$		
5			

Finally we need to fill in just the elements in the 5^{th} row and S^{th} column. This will contain all the 5 letter strings that can be made using this grammar. This would be made with 4 letter strings of X with 1 letter strings of Y and 3 letter strings of X with 2 letter strings of Y and 2 letter strings of X with 3 letter strings of Y and 1 letter strings of X with 4 letter strings of Y. Since none of these composite strings exists we conclude that abaaa is not generated by this language. ■

Now we consider a more complex example.

Example 4.6.8. Determine if the string $abbab$ is generated from the language described by the following context free grammar in Chomsky normal form :

$$S \to AB; A \to BB|a; B \to BA|b$$

We begin by setting up the table and filling in any terminal string that that are produced directly from a non–terminal.

# letters	S	A	B
1		a	b
2			
3			
4			
5			

Now we fill in the 2^{nd} row.

# letters	S	A	B
1		a	b
2	ab	bb	ba
3			
4			
5			

Filling in the 3^{rd} row results in the following table, by concatenating 1 letter strings with 2 letter strings.

# letters	S	A	B
1		a	b
2	ab	bb	ba
3	$aba,\ bbb$	bab, bba	baa, bbb
4			
5			

Next we fill in the 4^{th} row using 3 pairs of strings. (1 letters with 3 letters or 2 letters with 2 letters and 3 letters with 1 letter strings). I have suppressed duplicates in the resulting table below.

# letters	S	A	B
1		a	b
2	ab	bb	ba
3	aba, bbb	bab, bba	baa, bbb
4	$abaa, abbb, bbba, babb, bbab$	$bbaa, bbbb, baba, baab$	$baaa, bbba, babb, bbab$
5			

Finally in the 5^{th} row, we need only complete the S column.

# letters	S	A	B
1		a	b
2	ab	bb	ba
3	aba, bbb	bab, bba	baa, bbb
4	$abaa, abbb, bbba, babb, bbab$	$bbaa, bbbb, baba, baab$	$baaa, bbba, babb, bbab$
5	$b^2a^2b, b^5, babab, baabb, babba, bbaba,$ $babba, b^3a^2, aba^3, ab^3a, ababb, abbab$		

Since $abbab$ is found in the table, the string **is** generated by the grammar. ■

Although we have shown three key questions are decidable for context free grammars. A number of problems concerning context free languages *NOT* decidable. Although it is beyond the scope of this text, it can be shown that the following are undecidable :

1) Given two context free grammars, can one decide if they generate the same language?

2) Given a context free grammar, can one decide if it is ambiguous?

3) Given two context free grammars, can one decide if their intersection is empty?

As languages and machines become more sophisticated, the number of questions we can answer about them gets fewer. This trend continues in the next chapter where we discuss Turing Machines.

4.7 Exercises

1) Find a context free grammar for the language where the total number of as mod $3 = 0$.

2) Find a context free grammar for the language $\{a^{2n}b^n\}$

3) Draw the parse tree for the string $aabb$ for the grammar $S \to aaS|Sb|b$

4) Prove that $S \to AB|aaB; A \to aA|a; B \to b$ is ambiguous.

5) Prove the following grammar is ambiguous. Prove that $S \to aSb|Sb|Sa|a$ is ambiguous.

6) Find a context free grammar for $L = \{a^n w b^n | w \in (a+b)^*\} = \{ab, aab, abb, aaabb, \}$

7) Convert the following grammar to Chomsky normal form $S \to BAS|a; A \to BB|a; B \to AB|bB|\lambda$

8) Convert the following grammar to Chomsky normal form $S \to ASA|aB; A \to B|S; B \to b|\lambda$

9) Draw a deterministic push down automaton for the language $L = \{a^n b^m c^{n-m}$ where $n \geq m\}$

10) Draw a deterministic push down automaton for the language $L = \{a^n b^{n+m} a^m$ where $m, n > 0\}$

11) Use the algorithm from the text to find the equivalent non–deterministic push down automaton for the context free grammar $S \to XY|b, X \to b, Y \to a$

12) Use the algorithm from the text to find the equivalent non–deterministic push down automaton for the context free grammar $S \to AB|a; A \to BA|a; B \to BB|b$

13) Determine if the following grammar generates any words

$$S \to AB; A \to BSB|CC|a; B \to AAS; C \to CC|b$$

14) Determine if the following grammar generates any words

$$S \to XY|SZ; Z \to Za|a; X \to aX|XY; Y \to bY|b$$

15) Determine if the following grammar generates an infinite language

$$S \to ZS|ZA|XY|bb; X \to YY; Y \to XY|SS; Z \to ZZ|ZS; A \to SA|a$$

16) Determine if the following grammar generates an infinite language

$$S \to XY|SZ|aA; Z \to aZ|ZW; W \to XW|aZ|YW; X \to aX|a; Y \to aY|WZ; A \to aA|a$$

4.7.1 Solutions to Exercises

1) $S \to bS|aX; X \to bX|aY; Y \to bY|aZ; Z \to bZ|aS|\lambda$

2) $S \to aaSb|\lambda$

3)

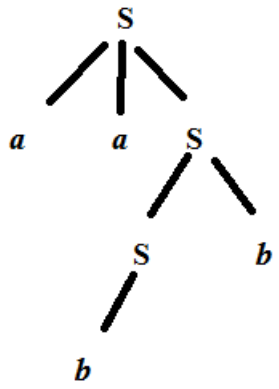


Figure 4.16: Tree for Exercise 3

4) It has two leftmost derivations. $S \to AB \to Ab \to aAb \to aab$ and $S \to AB \to aAB \to aab$

5) It has more than two leftmost derivations. $S \to aSb \to aab$ and
$S \to aSb \to aSbb \to aabb$ and $S \to aSb \to aSab \to aaab$ etc.

6) $S \to aSb|X, X \to aX|bX|\lambda$

7) $S \to CS|AS|a; C \to BA; D \to BB; E \to a; F \to b$
$A \to DE|BE|a; B \to AB|DE|BE|FB|a|b$

8) $S \to DA|AS|SA|CB|a;$
$B \to b$;
$A \to DA|AS|SA|CB|a|b;$
$C \to a$;
$D \to AS;$

9) Idea:
Read an a, push a *smurf*
Read a b, pop a *smurf*
Read a c, pop a *smurf*

smurf is any arbitrary letter (you just need something to push and pop).
See Figure 4.17

10) See Figure 4.18

11) See Figure 4.19

12)

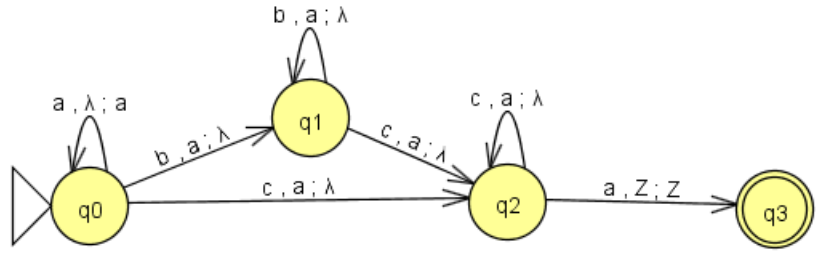


Figure 4.17: Pushdown Automata for Exercise 9

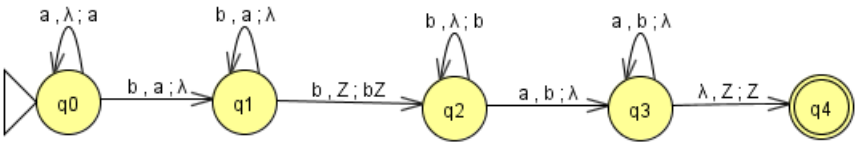


Figure 4.18: Pushdown Automata for Exercise 10

13) Start by replacing any non-terminals that have their right-side all terminals:
Replace A with a and C with b
That leaves you with:
$S \to aB$; $A \to BSB|bb|a$; $B \to aaS$; $C \to bb|b$
This where you have to stop and you can see that no words are generated.

14) Start by replacing any non-terminals that have their right-side all terminals:
Replace Z with a and Y with b
That leaves you with:
$S \to Xb|Sa$; $Z \to aa|a$; $X \to aX|Xb$; $Y \to bb|b$
This where you have to stop and you can see that no words are generated.

15) First you must reduce the grammar to only effective non-terminals.
Replace S with bb and A with a
Now you have:
$S \to Zbb|Za|XY|bb$; $X \to YY \to bbbbbbbb$; $Y \to XY \to bbbb$; $Z \to ZZ|Zbb$; $A \to bba$
You can see above that Z cannot be terminated. So you remove it.
Which leaves you with:
$S \to XY|bb$; $X \to YY$; $Y \to XY|SS$; $A \to SA|a$
A is unreachable so it is removed, leaving you with:
$S \to XY|bb$; $X \to YY$; $Y \to XY|SS$ so this grammar generates an infinite language.

16) First you must reduce the grammar to only effective non-terminals.
Replace X with a and A with a, which leaves you with:

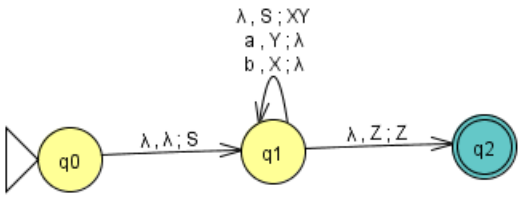


Figure 4.19: Pushdown Automata for Exercise 11

$S \to aY|SZ|aa$; $Z \to aZ|ZW$; $W \to aW|aZ|YW$; $X \to aa|a$; $Y \to aY|WZ$; $A \to aa|a$
Z, W, and Y are non-terminating so they are removed, leaving:
$S \to aa$; $X \to aX|a$; $A \to aA|a$ and X and A are unreachable.
$S \to aa$ so this grammar generates an infinite language.

4.7.2 Computer Activities

1)

Bibliography

[1] Cohen, Daniel I.A. , "Introduction to Computer Theory, 2nd Edition," John Wiley & Sons, Inc., , (1991).

Chapter 5

Turing Machine Languages

A computation is a process that obeys finitely describable rules. Rudy Rucker

5.1 Introduction

In this chapter we learn about the ultimate model for computation in the 20^{th} century, Turing Machines. Conceived and named after the genius Alan Mathison Turing (1912 - 1954) , who greatly influenced the modern ideas on algorithms, computation, cryptography and artificial intelligence. The test for artificial intelligence (the Turing test) is credited to Alan Turing. The test is simply that if a person communicates via the keyboard to another entity and cannot tell if he is talking to computer or a human, then the computer must be considered intelligent.

Alan Turing was part of the team responsible for breaking a number of German ciphers during World War II and helped win the war for the western powers. Turing's short life was very troubled due to his homosexuality, which was a crime in England in the 1950s. Turing routinely took drugs as an alternative to prison and died tragically as a result of a drug overdose (which was officially determined to be suicide, but is doubted as accidental by others).

Figure 5.1: XKCD: A webcomic of romance, sarcasm, math, and language.

With our discussion of Turing machines, we will complete our discussion of models of computa-

tion. Turing machines, although discussed prior to the other models of computation, will be used to model what computers can compute. It is interesting to compare the different models and their properties. The following table summarizes some of these results.

Table 5.1: Languages by Grammar, Accepter and Decidability

Grammar	**Accepter**	**Closure**	**Decidable**	**Application**
Regular	Finite automaton	$\cup, \cap, ^*$, suffix, prefix , concatenation, reverse, ...	emptiness finiteness membership equivalence	text editors
Context Free	Pushdown automaton	$\cup, ^*$, concatenation	emptiness finiteness membership	compiler
Context Sensitive	Turing machine	$\cup, \cap, ^*$, concatenation	not much	computers

From the above table it appears that as the languages become more complex, less can be decided about them, or as the model of computation gets more powerful, the less can be decided about them. This is indeed the case. In fact, Gödel's first incompleteness theorem shows that any consistent formal system that includes a sufficient amount of the theory of numbers is incomplete. In other words if the system is complex enough, there are true statements expressible in its language that are unprovable. The implication is that there are some problems (even yes or no problems) that no computer program can answer.

Below is an illustration of the sets of languages we have covered so far. Context sensitive languages have grammars with no restrictions. It is therefore clear that context sensitive languages must contain context free languages. Each set is listed with an example language that belongs to that set.

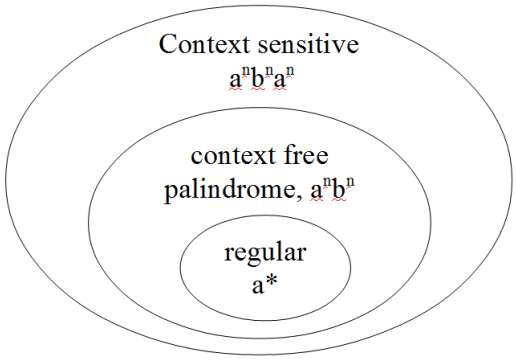


Figure 5.2: Language Universe (so far)

5.2 Turing Machines

We will begin with a definition of a turning machine.

Definition 5.2.1. A **Turing machine** is a collection of the following three items:

1) A finite set of input and output letters Σ. The input is written on an initially blank infinite tape, with the read/write head of the machine on the leftmost letter of the input.

2) A finite set of states Ω, one of which is denoted as the start state and one of which may be denoted as the halt (accepting) state.

3) A finite set of transitions between states of the form read, write ; move . The tape head can move 1 square either right or left or stay.

Superficially, a Turing machine differs from the earlier machines in that the halt state, once entered, is never left. Additionally, Turing machines have the ability to leave output on the tape. Since our previous examples concerned language recognition, the lack of output was not a limitation. Having the machine stop computation when it reached the accept state, could have been accomplished with the both finite automata and push down automata by adding a new accept state connected to the old accept states with λ–transitions. The new state will ignore whatever is left on the input tape or the stack and and remain in that state. Fundamentally the Turing machine differs from earlier machines by being able to store anything on the tape. A Turing machine can even store a "program" to run later, such as modern computer.

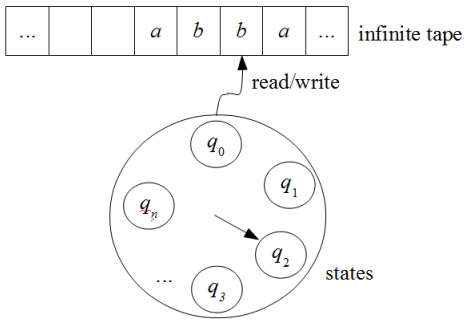


Figure 5.3: Artists Conception of a Turing Machine

Again we will use JFLAP to actually construct Turing machines. The three parts of a transition will be read; write ; move. (The punctuation is that of the jflap simulator.) The read/write head will be over a cell and the machine will be in a given state. Then depending on what is written on the current cell of the input tape, the Turing machine will write a new symbol on the tape (at the current position) and then move either right or left 1 square (or stay).

Example 5.2.1. The first example will be a Turing Machine to accept the context free language $a^n b^n$ The algorithm we use will be recursive. We first check if the read/write head is over a blank. If so we halt. If not, we should be over an a which we will erase and move to the end of the string and erase the last b. We will then move back to the start of the string and repeat.

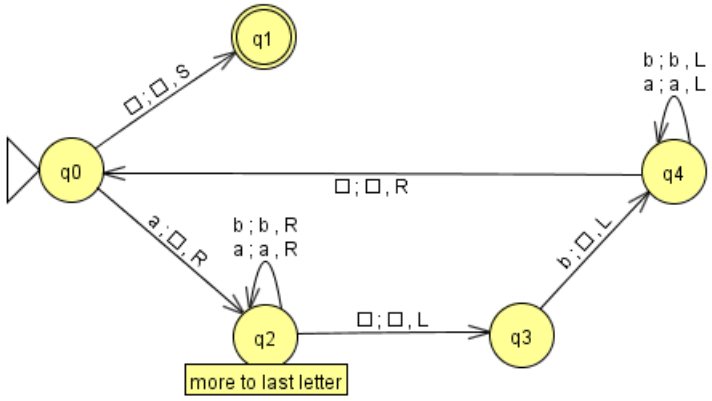


Figure 5.4: Turing Machine for $a^n b^n$

■

In our next example, we consider a language which is not context free.

Example 5.2.2. This example will recognize the language $a^n b^n a^n$. Again the algorithm will be recursive. We first check if the read/write head is over a blank. If so we halt. If not, we should be over an a which we will erase and move through all the a's and all the b's, and then write a a on top of the last b. Then we move through the last group of a's and erase the last 2 a's. Finally we move back to the start of the string and repeat. ■

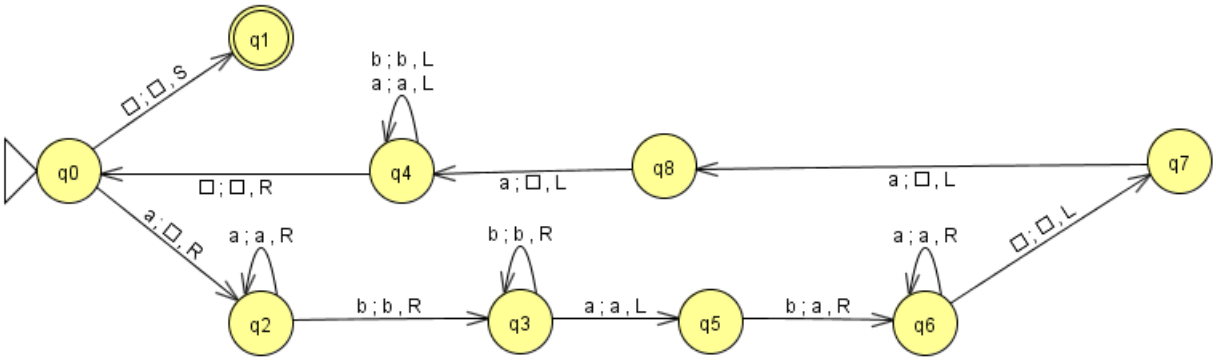


Figure 5.5: Turing Machine for $a^n b^n a^n$

We will now show the Turing Machines power by doing computation. The simplest method is to encode numbers in unary. 1 represents 1, 11 represents 2, 111 represents 3 and so on. Our first example will be addition.

Example 5.2.3. Find the sum of 2 numbers written in unary. The two numbers will be separated by a space. In this example we merely remove the space by overwriting it with a 1 and removing the last 1. The machine then returns the tape head to the beginning of the number. Leaving the

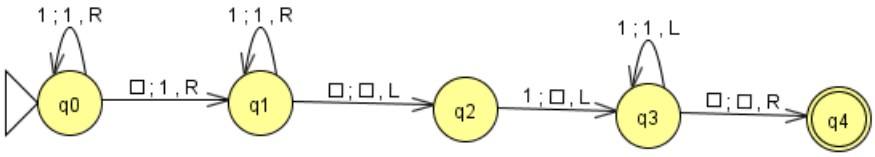


Figure 5.6: Turing Machine for Addition in Unary

machine in this state will allow us to makes these little functions, subroutines and then call them one after another, to accomplish much more complicated procedures. ■

The final example is a slightly more complicated computation, division by 2.

Example 5.2.4. Construct a Turing machine which starts with 1^n on the tape and ends with $\lceil 1^{n/2} \rceil$ on the tape.

We will again do something recursive. The idea will be to read a 1 from the front and lop off a 1 from the back. A problem immediately comes to mind that the front 1's will eventually turn into back 1's. So we will mark each front 1 by changing it to the letter a. When the entire string is a's or all but 1 of the symbols is a, we will go back and change all the a's back to 1's. ■

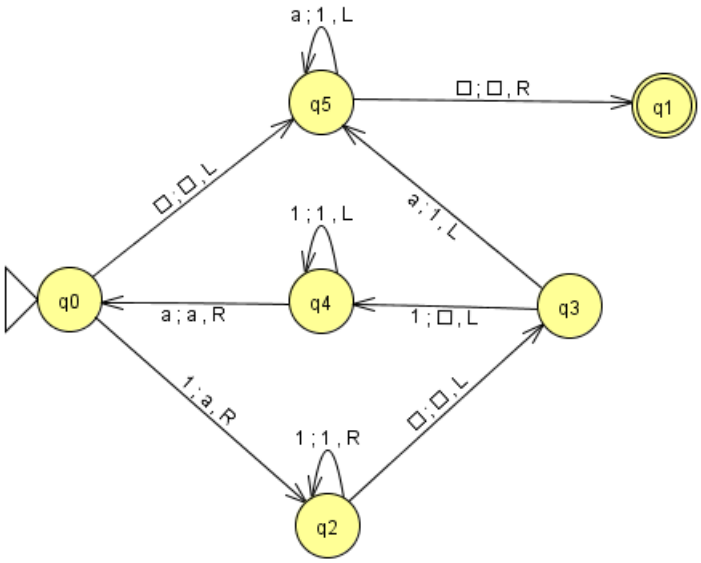


Figure 5.7: Turing Machine for Division by 2 in Unary

By now you should be convinced that Turing Machines can do lots of computations. By using the addition machine as a subroutine, we can easily do multiplication. By using multiplication as a subroutine, we can make a Turing machine for exponentiation. A Turning machine although slow, is actually more powerful than your computer. A Turing machine has an infinite memory which is larger than whatever mega gigabytes your computer has on board.

5.2.1 Languages associated with a Turing Machine

In this section we will consider the three languages associated with a Turing machine. In contrast to a finite automaton, and just like computer programs, a Turing machine can halt, crash or enter an infinite loop. Given a Turing machine T, we will consider what input will result in a halt (Accept(T)) , a crash (Reject(T)) or an infinite loop (Loop(T)). If we limit our discussion to languages using only a and b then we can conclude that :

$$\text{Accept}(T) \cup \text{Reject}(T) \cup \text{Loop}(T) = (a+b)^*$$
and
$$\text{Accept}(T) \cap \text{Reject}(T) = \text{Accept}(T) \cap \text{Loop}(T) = \text{Reject }(T) \cap \text{Loop}(T) = \emptyset$$

As anyone who has tried to debug someone else's code will attest, this is a daunting task. It is essentially looking at the code and deciding what the program will do. We will limit our examples to very simple machines.

Example 5.2.5. Find Accept(T), Reject(T) and Loop(T) for the following Turing machine.

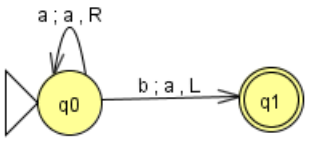


Figure 5.8: Turing Machine for Example 1.2.5

To find Accept(T), we start at the halt state and work backwards to the start state. Here we see we need to read an b, and preceding that we could encounter many a's. Hence

$$\text{Accept}(T) = a^*b(a+b)^*$$

Note that $(a+b)^*$ at the end of the regular expression allows for any input after the machine has entered the halt state . Since the machine halts after reading the first b, whatever follows that first b, is ignored. The only loop here is the one on the start state reading a's, hence there is no possibility for an infinite loop. Hence

$$\text{Loop}(T) = \emptyset$$

Now Reject(T) must be the compliment of Accept(T) or any string that does not contain a b. Hence

$$\text{Reject}(T) = a^*$$

■

The next example includes a non–empty Loop() set.

Example 5.2.6. Find Accept(T), Reject(T) and Loop(T) for the following Turing machine.

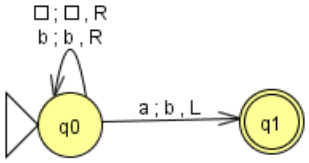


Figure 5.9: Turing Machine for Example 1.2.6

To reach the halt state we need to read an a, and proceeding that we could encounter many b's. Hence

$$\text{Accept}(T) = b^*a(a+b)^*$$

The difference with the previous example, is that we are also looping on blanks. Since there an infinite number of blanks on the tape, we can enter an infinite loop. Hence

$$\text{Loop}(T) = b^*$$

Since the Loop(T) is the compliment of Accept(T) we have that :

$$\text{Reject}(T) = \emptyset$$

■

Example 5.2.7. Find Accept(T), Reject(T) and Loop(T) for the following Turing machine.

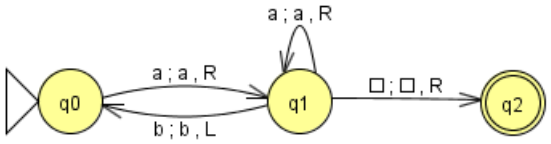


Figure 5.10: Turing Machine for Example 1.2.7

In this case it is easier to begin at the start. We must read an a to start. If we ever read a b then the machine backs up a square (which was an a) and will go into an infinite loop. Everything else will be rejected. Hence

$$\text{Accept}(T) = aa^*$$
$$\text{Loop}(T) = aa^*b(a+b)^*$$
$$\text{Reject}(T) = \lambda + b(a+b)*$$

■

5.3 Non–determinism

Finite automata and transition graphs accept the same set of languages. It was stated but not proven that deterministic and non–deterministic push down automata accept different sets of languages. The question naturally arises "Do non–deterministic Turing machines accept a larger set

of languages than deterministic Turing machines ?". The answer is no, and can be shown using a deterministic simulation of the non–deterministic Turing machine. The proof is instructive and a simplified informal version is presented below.

First we need to show that adding more tracks (tapes) does not change the set of languages that Turing machines accept.

Theorem 5.3.1. *A k-track Turing machine can be stimulated with 1-track Turing machine.*

Informal Proof Consider a k–track pictured below.

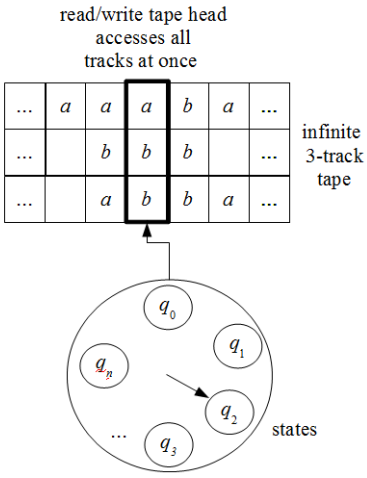


Figure 5.11: Artists conception of a 3–track Turing machine

We can simulate a n–track Turing machine with a standard Turing machine by changing the input alphabet. If the n track machine uses the alphabet Σ, the simulation will use an n–tuple, Σ^n. In the picture the tape head is currently reading a, b, and b on each of the three tapes respectively, then the simulation will read (a, b, b). The instructions will then have to be minimally adjusted appropriately, but then we can construct a standard Turing machine to accept the same language as any n–track Turing machine. ■.

We can now deal with the issue of non–determinism. Analogous to the non–determinism we have discussed concerning finite automata and push down automata, a non–deterministic Turing machine will accept a string (or halt) if there is *any* path that leads to a halt. We discovered that adding non–determinism to finite automata did not change the set of languages they accepted, however adding non–determinism to push down automata did change the set of acceptable languages. We will now address this question for Turing machines.

Theorem 5.3.2. *A non–deterministic Turing machine can be simulated with a standard Turing machine.*

Informal Proof We will simulate a non–deterministic Turing machine with a 3 track machine. On the top track we will keep a pristine copy of the input. This track will not be changed in our simulation. On the middle track we will generate every possible sequence of states in the non–deterministic Turing machine. For example if the non-deterministic Turing machine had 3 states,

$\{q_0, q_1, q_2\}$, then all possible sequences would be first all the sequences of length 1. Then all the sequences of length 2, followed by all the sequences of length 3.

$$\{q_0, q_1, q_2, q_0q_0, q_0q_1, q_0q_2, q_1q_0, q_1q_1, ..., q_2q_2, q_0q_0q_0, q_0q_0q_1, q_0q_0q_2, q_0q_1q_0, q_0q_1q_2, ..., q_2q_2q_2\}$$

It does not matter if the sequence can be followed in the non–deterministic Turing machine. Finally on the bottom track we will have our working string. The simulation will follow the following cycle:

1) Copy the input from the top track to the bottom track.

2) Generate the next possible sequence on the middle track.

3) Attempt to run that sequence of states on the input on the bottom track.

   if the sequence is illegal (starts in a non–start state, or makes an illegal transition of the non–deterministic Turing machine on the input, or just terminates in an non–halt state) then

   erase the bottom track and go to 1.

   if the sequence enters a halt state of the non-deterministic Turing machine then halt.

The simulation works since the list of all possible sequences is countable. In this way our simulation will find a path through the non–deterministic Turing machine, if one exists. Therefore the simulation will either halt (if the non–deterministic Turing machine halts), or will enter an infinite loop. It will never crash, but just attempt the next sequence of states. ■

5.3.1 Recursive and Recursively Enumerable

Practically as well as theoretically a distinction must be made between Turing machines that sometimes enter an infinite loop to reject a string. Obviously infinite loops are undesirable to ever occur. We will distinguish between these two sets of languages.

Definition 5.3.1. A language L is called **recursive** if there exists a Turing machine T such that $\text{Accept}(T) = L$, $\text{Reject}(T) = L'$, and $\text{Loop}(T) = \emptyset$.

A language L is called **recursively enumerable** if there exists a Turing machine T such that $\text{Accept}(T) = L$, $\text{Reject}(T) \cup \text{Loop}(T) = L'$.

Theorem 5.3.3. *If both L and L' are recursively enumerable, then L is recursive*

Proof Assume L and L' are recursively enumerable. Then there exists Turing machines T and T' such that :

$$\text{Accept}(T) = L \text{ and } \text{Reject}(T) \cup \text{Reject}(T) = L' \text{ and}$$

$$\text{Accept}(T') = L' \text{ and } \text{Reject}(T') \cup \text{Reject}(T') = L.$$

Now consider a 2 track Turing machine S. We will write the input on both tracks. We will run the T on the first track and T' on the second track, alternating moves. First one move of T then one move of T'. Since we know that both L and L' are recursively enumerable, and the input is in either L or L', then either T or T' will halt. If T halts, then we know that the input is in L . If T'

halts, we know that the input is in L'. We are guaranteed to have one of the Turing machines halt. Hence S can determine if a string is in L or not without ever going into an infinite loop. Thus L is recursive.

■

Example 5.3.1. Certainly all regular and context free languages are recursive. We have also seen that $\{a^n b^n a^n\}$ is also recursive. The question arises "Are there any non–recursive languages?". We will show now that this is indeed the case in the next section. ■

5.4 Universal Turing Machines

So far all of our Turing machines have had a single purpose, like the computer programs one runs. Indeed, in the early days of computer science, the computers were rewired every time they were used to solve different problems. However now , (thanks to van Neumann) the programs are equivalent to data and are input in our compiler–operating system–computer hardware system. The compiler–operating system–computer hardware has the power to run any program input into it. We will now discuss a Universal Turing machine which is similar to a modern computer. The Universal Turing machine will take as part of its input an encoding of any Turing machine simulate its function, just like a modern computer will input a program and follow its instructions.

We will now discuss construction of a Universal Turing machine. This machine will treat other machines as input and then run them. The input to the Universal Turing machine will be the encoding of a Turing machine. Each transition in the Turing machine will be encoded as 5 tuple holding the information about what state the transition starts in, what state the transition ends in, what it reads, what it writes and which way it moves. In other words it is in the form (from, to, read, write, move). The key is to determine an encoding for a Turing machine. We will use unary to keep things simple. State 1 will be a, state 2 will be aa , and state n will be denoted a^n. We can assume without loss of generality that state 1 will be the start state and state 2 will be the halt state. We will use a b to denote the end of a state name. Table 5.2 will given the encodings for the input output and movement.

Object	**Encoding**
a	aa
b	ab
blank	ba
#	bb
Right	aa
Left	bb
Stay	ab

Table 5.2: Encoding of a Turing Machine

The table allows for four input characters: a, b, blank, and # as well as three movements: right, left, and stay. The restricted alphabet will make creation of some complex Turing machines more

cumbersome, it can be shown that any Turing machine, can be simulated with a Turing machine using only the above restricted alphabet.

Once we have encoded each of the transitions we will order them in lexicographical (dictionary) order. Given two strings $a = a_1a_2...a_n$ and $b = b_1b_2...b_m$ then in lexicographic order $a < b$ if in the first position i where $a_i \neq b_i$, $a_i < b_i$. Blanks are considered letters that come before any other letter so that if one string matches all the characters of another until it runs out of characters, the shorter string is first. In this way, we have unique encodings for each Turing Machine. However, it could be the case that several Turing machines behave the same way. The language made up of encodings is easy to recognize. It is a regular language described by aa*baa*b(a+b)(a+b)(a+b)(a+b)(a+b)(a+b). Not all the machines described by this language do anything useful but all can be built. We will illustrate some encodings in the following examples.

Example 5.4.1. Encode the following Turing Machine using table 5.2.

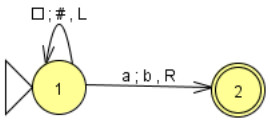


Figure 5.12: Turing machine to be Encoded for Example 5.4.1

The loop is from 1 to 1 reading blank, writing # and moving left. It is encoded as
ababbabbab
The transition to the halt is from 1 to 2 reading a, writing b and moving right. It is encoded as
abaabaabaaa
The second transition is first lexicographically so the machine is encoded as :
abaabaaabaaa ababbabbab
The space above is just included to enhance readability. It is not part of the encoding. ■

Below is a slightly more complex example.

Example 5.4.2. Encode the following Turing Machine using table 5.2.

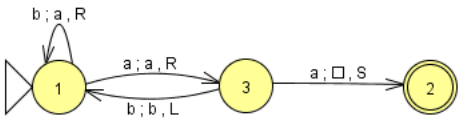


Figure 5.13: Turing machine to be Encoded for Example 5.4.2

The loop is from 1 to 1 reading b, writing a and moving right. It is encoded as
ababbaaaa

The transition to the middle state is from 1 to 3 reading a, writing a and moving right. It is encoded as

abaaabaaaaaa

The transition back to 1 is from 3 to 1 reading b, writing b and moving left :

$aaababababbb$

Lastly, the transition to the halt state is from 3 to 2 reading a, writing blank and moving stay :

$aaabaabaabaab$

Putting it all together in lexicographical order (with spaces for readability) yields :

aaabaabaabaab aaababababab abaaabaaaaaa abababaaaa

■

5.4.1 Languages concerning Turing machines

Now that we can encode Turing Machines, we can define languages concerning them. For example we consider the language L_1 made up of all encoding of Turing machines that take an even number of moves when run on the empty tape. Or we can consider the language L_2 of all Turing machines that halt on the empty tape within 100 moves.

The brute force Turing machine to accept this languages would just simulate each input running on the empty tape. The language L_2 is clearly recursive. If the simulator keep a move counter (say on another track) then it could stop after 100 moves and make a decision to accept or reject. However, this solution to accept L_1 will sometimes run into an infinite loop.

Note this does not prove that the this language is recursively enumerable. There might be an algorithm to determine if the machine makes an even number of moves without actually simulating it. You have seen many examples of algorithms that find answers without trying every possibility. Recall that we can search a sorted list in $O(\lg(n))$ time for a list of length n. Dijkstra's algorithm for finding the shortest path is much faster than attempting all possible paths.

We can also determine if every finite automaton will make an even or odd number of moves on empty input without ever going into an infinite loop. So it is not beyond the realm of possibility that a through analysis of the inputted Turing machine might yield an answer without ever attempting to simulate it.

We will now create a language that is *not* recursively enumerable. The proof will rely heavily on the fact that since all Turing machines are encodable, we can order them and count them. In other words there is a systematic method of running through every possible Turing machine. We can now refer to M_i as the i^{th} Turing machine. The proof will also use a diagonal argument. This type of argument occurs often in the field of language theory. Consider the set of strings $\{aa^*\}$ $= \{a, aa, aaa, ...\}$. Now each Turing machine M_i accepts or rejects the string a^j. Consider the language $L^\ddagger$ which contains the string a^i if M_i rejects it and does not contain a^i if M_i accepts it. We claim that $L^\ddagger$ is not recursively enumerable.

Theorem 5.4.1. *There exists non–recursively enumerable languages.*

Proof Consider the language $L^\ddagger$ described above. We claim there is no Turing machine that will accept it. This because it differs from every Turing machine. For example if the M_i accepts a^i, then a^i is *not* in $L^\ddagger$ and vice versa. Hence if a Turing machine existed for ‡, then it would have to

differ from the M_i when run on input a^i. This must not be one of the enumerated Turing machines. ■

We are now ready to display a language which is recursively enumerable but not recursive.

Theorem 5.4.2. *There exists recursively enumerable languages which are not recursive.*

Proof Consider the language $L^\dagger$ which contains the string a^i if the i^{th} Turing machine M_i accepts it and does not contain a^i if M_i rejects it. We claim $L^\dagger$ is recursively enumerable. To construct a Turing machine to accept $L^\dagger$, we simply read the the input string a^i, and check that it contains only a's and compute i. We then construct the i^{th} Turing machine M_i to run on the input. If M_i accepts the input we accept, if not, we reject. Note that sometimes the M_i may go into an infinite loop. Hence $L^\dagger$ is recursively enumerable.

To show that $L^\dagger$ is not recursive, we need only point out that $L^\dagger$ is the complement of $L^\ddagger$ which was proven to be not recursively enumerable above. Since Theorem 5.3.3 says that if a language is recursive, it's complement is also recursive, we can conclude that $L^\dagger$ is not recursive since it's complement is not recursively enumerable. ■

Both proofs use a diagonal argument. In the table below we have enumerated the Turing machines and the strings of a^i. A zero indicates that the string is not accepted by the Turing machine while a 1 indicates that it is accepted. The table below is a possible representation for the language $L^\dagger$ since each M_i accepts a^i.

TM vs string	a	aa	aaa	$aaaa$	...
M_1	**1**	1	0	0	...
M_2	0	**1**	1	1	...
M_3	1	0	**1**	0	...
:	:	:	:	:	...

Table 5.3: Possible table of TMs vs a^i

The diagonal of the table allows us to make statements about every Turing machine. In this case we created a language that was not accepted by any Turing machine and its complement to illustrate recursively enumerable and a language that no Turing machine can accept.

Here is another example (From Daniel Cohen's text) of a non–recursively enumerable language and it's complement which is recursively enumerable but not recursive. Both languages are named after Alan Mathison Turing.

Example 5.4.3. Define the language *Alan* to all elements of aa*baa*b(a+b)(a+b)(a+b)(a+b)(a+b)(a+b) which either represent invalid Turing machines or which represent Turing machines, which reject or loop when run on its own encoding. We claim that Alan is not recursively enumerable. **Proof** We will use a proof by contradiction. Assume that Alan is recursively enumerable and accepted by the Turing machine T. Then we know that Accept(T) = Alan. The question arises as to whether $T \in$ Alan or not.

If $T \in$ Alan, then T's encoding is accepted by T.
However that contradicts the definition of Alan.

If $T \notin$ Alan, then T's encoding is rejected by T or T goes into an infinite loop.
However that indicates that T should be in Alan. However T was constructed to accept all words in Alan. Again we have a contradiction.
Hence T cannot exist, so Alan cannot be recursively enumerable.
■

Define the complement of Alan in aa*baa*b(a+b)(a+b)(a+b)(a+b)(a+b)(a+b) to be Mathison. This is the set of Turing machines that accept their own encoding. We can build a Turing machines, that which constructs the input Turing machine, and runs it on its input. If it accepts the input it is in Mathison. ■

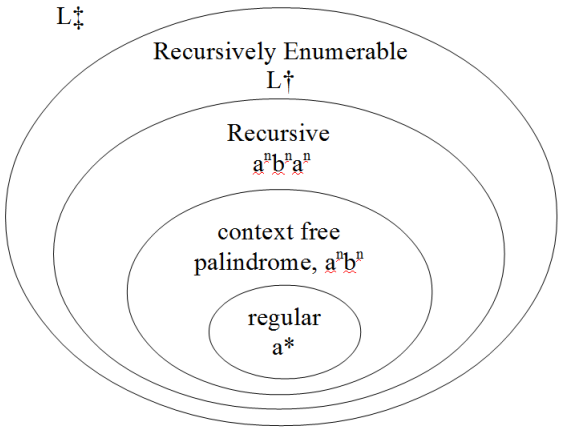


Figure 5.14: Expanded Classification of Languages

Our classification of languages has now expanded further to following diagram shown in Figure 5.14. The classification has grown in recent years with new classes of languages being introduced all the time. Most notable additions include languages that be recognized in a given time based on the size of the input, or languages that be be recognized using a limited space. As our understanding of computation becomes more complete our classification system also becomes more complete.

5.4.2 Chomsky Hierarchy and Church's Thesis

We now come to the grammar equivalent to a Turing machine. In this grammar the production rules are almost unrestricted. It simply requires at least one non–terminal to replace. The replacement rule can now depend on the context (the surround substrings). For example we can have a rule of the form

$$aaN \rightarrow \text{stuff}$$

The above rule indicates that ONLY when the non–terminal N is proceeded by two a's can we replace it, and when we do, we can change the a's also. So the meaning of the non–terminal depends on it's context. This is true of English too. For example consider the word "shot" in the following two sentences.

The man shot the bird with his gun.

The man shot the bird with his camera.

Definition 5.4.1. A **context sensitive** grammar is one where the only restriction on the productions, is that the left hand side contains at least one non–terminal.

With a context sensitive grammar almost anything goes. It is intuitively clear that context sensitive grammars are the most powerful (can describe the largest set of languages) of the grammars. Any regular or context free grammar is also context sensitive. We will state without proof that context sensitive grammars are equivalent to Turing machines.

Theorem 5.4.3. *Context sensitive grammars generate the same set of languages accepted by Turing machines.*

The "ultimate power" of Turing machines is captured in the follow conjecture attributed to Alonzo Church, know famously as "Church's Thesis".

Turing machines are the ultimate model of computation.

Indeed for many years the most powerful algorithmic tool was considered a Turing machine. In recent years however, it has been shown that more languages can be recognized if the Turing machine is augmented, such as with a feed back mechanism.

5.5 The Halting Problem and Decidability

It is now time to discuss some decidability problems. In the past sections we introduced algorithms to solve decidability problems. Now we will be attempting something much more difficult ; proving a problem is undecidable. This is a statement that no algorithm exists to decide the problem. However once one problem has been proven to be undecidable, it will be much easier to prove other problems are undecidable using a technique called reduction. We begin with the classic undecidable problem "the Halting problem".

Theorem 5.5.1. *The problem of determining whether a given Turing machine will halt on a given input is undecidable.*

Proof This will be a proof by contradiction. Assume that deciding whether a given Turing machine will halt on a given input string, by the Turing machine H (pictured below).

Figure 5.15 shows the Halting machine with two inputs (one for an encoded Turing Machine T) and another for the input string S. If it halts, then that indicates that T halts on input S. Now if we can build the Halting machine, we can build what we call the Mö bius Halting machine.

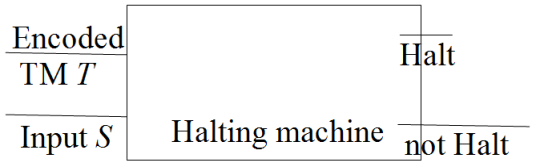


Figure 5.15: Artist's conception of the Halting Machine

Recall a Möbius strip, is a piece of paper which is twisted and then the ends are taped together. The Möbius Halting machine contains the Halting machine, and is twisted in that if the Halting machine indicates T halts on it's own encoding then the Möbius Halting machine enters an infinite loop. While if the Halting machine indicates the T does not halt on it's own encoding, then the Möbius Halting machine halts.

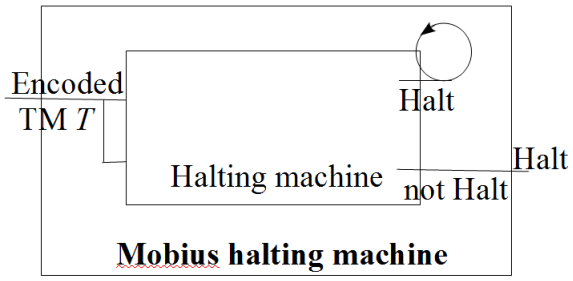


Figure 5.16: Artists conception of the Möbius Halting Machine

Now consider what happens if we feed the encoding of the Möbius Halting machine into the input of the Möbius Halting machine. If the Möbius Halting machines halts on it's own input, then it goes into an infinite loop. On the other hand if the Möbius Halting machine does not halt on it's own input, then it halts. Either way we have a contradiction. Hence the Möbius Halting machine cannot exist. Hence the Halting machine cannot exist.

■

Once we have our first undecidable problem, we can generate many more using the same type of proof.

Theorem 5.5.2. *The problem of determining whether a given Turing machine will halt on a the empty tape is undecidable.*

Proof Assume we can determine if a given Turing machine T will halt on the empty tape this is decided by the Empty Turing machine illustrated below.

Now using the Empty machine, we will build a Turing machine to solve the halting problem. We will write a subroutine whose input is the code for a Turing machine T and a string S and

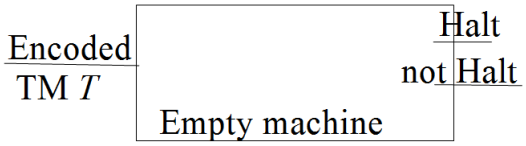


Figure 5.17: Artist's conception of the Empty Machine

which modifies the code for T to first write S to the tape and then run T. We will call this resultant machine Ts.

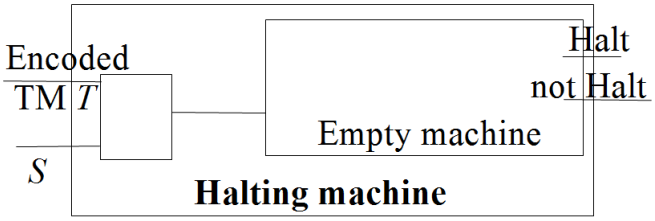


Figure 5.18: Artists conception of the Halting Machine using the Empty Machine

The empty machine decides if Ts halts on the empty tape. But Ts halts on the empty tape if and only if T halts on input S. Thus the resultant machine decides the halting problem. Since the halting problem cannot be decided, the Empty machine cannot exist.

■

Theorem 5.5.3. *The problem of determining whether a given Turing machine will accept every possible input is undecidable.*

Proof Assume that the question of whether a Turing machine T will accept every string is decidable by the Every string machine pictured below.

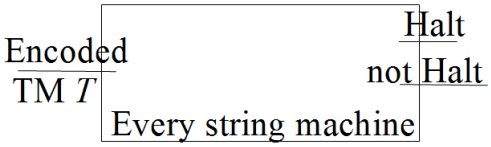


Figure 5.19: Artists conception of the Every string machine

We will use the Every string machine to determine if a Turing machine halts on the empty tape. Again we will make a subroutine preprocessor which will first erase the input and and then run the encoded T on the result. We call this modified machine T'. Shown in Figure 5.20

Now T' accepts everything if and only if T halts on the empty tape. Since halting on the empty

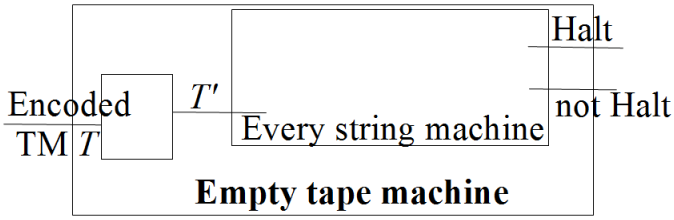


Figure 5.20: Artists conception of the Empty machine using the Every string machine

tape is undecidable, then the Every string machine cannot exist and deciding if a machine accepts every string is undecidable.

■

The method we used to prove things are undecidable based on other things that are undecidable is called reduction. Reduction is powerful technique which basically converts one problem into another problem, such that solving one can solve both. Basically if have trouble solving one problem, convert into a problem you can solve. It can be used to create a partial order of problems. Students may have seen this used in solving linear homogeneous constant coefficient recurrences, which are reduced to quadratic equations. The same reduction is used in differential equations.

It is important for computer scientists to be able to recognize undecidable problems when they are encountered. Typically these problems are concerned with programs that make decisions about programs. For example the halting problem is equivalent to writing a compiler that finds all infinite loops. Although many infinite loops can be detected, it is fruitless to attempt to write a compiler that will find all infinite loops. Similarily we cannot write a compiler which determines if a program will run on all possible inputs.

This chapter ended with identifying some problems no computer can solve. In the next chapter we discuss problems which can be solved in a reasonable amount of time. We will again be using reduction to compute the computational difficulty in solving problems. The magic of this technique that we will not solve ANY of the problems, we will just be able to order their solutions without knowing any of the solutions.

5.6 Exercises

1) Build a Turing machine that accepts the language of all strings composed only of a's and b's where the number of a's is equal to the number of b's.

2) Build a Turing machine that accepts the language of all strings composed only of a's and b's of the form $wa^{|w|}|w \in (a+b)^*$

3) Build a Turing machine that takes a number in binary (least significant digit first) and increments it by 1.

4) Build a Turing machine whose input is a string of a's and b's and which halts with the input string reversed.

5) Find the languages Accept(T), Reject(T), and Loop(T) for the Turing machine pictured in Figure 5.21.

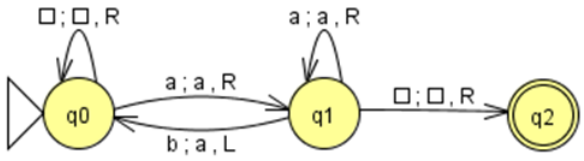


Figure 5.21: Turing Machine T

6) Find the languages Accept(T), Reject(T), and Loop(T) for the Turing machine pictured in Figure 5.22.

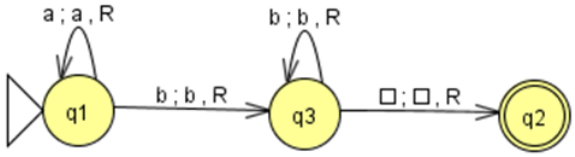


Figure 5.22: Turing Machine T

7) Encode the Turing machine in Figure 5.23 . Then decide if it is in Alan or Mathison

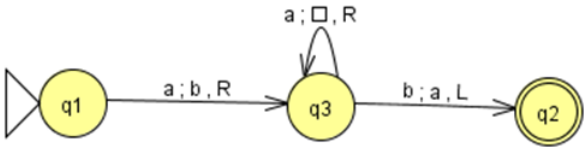


Figure 5.23: Turing Machine T

8) Encode the Turing machine pictured Figure 5.23 and decide if it is in Alan or Mathison.

9) Prove that one cannot decide if a two Turing machines accept the same language. (*Hint: fix one of the machines and solve the every string problem.*)

10) Prove that one cannot decide if a Turing machine will accept any string. (*Hint: use a preprocessor subroutine and reduce it to the halts on the empty tape problem.*)

11) Are Turing machines more powerful than your computer?

5.6.1 Solutions to Exercises

1) See Figure 5.24

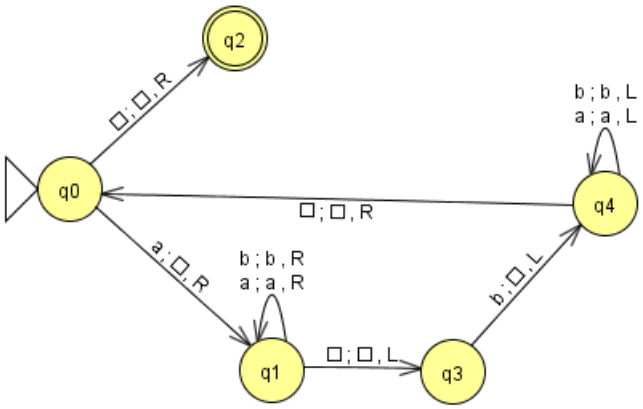


Figure 5.24: Turing Machine for Exercise 1

2) See Figure 5.25

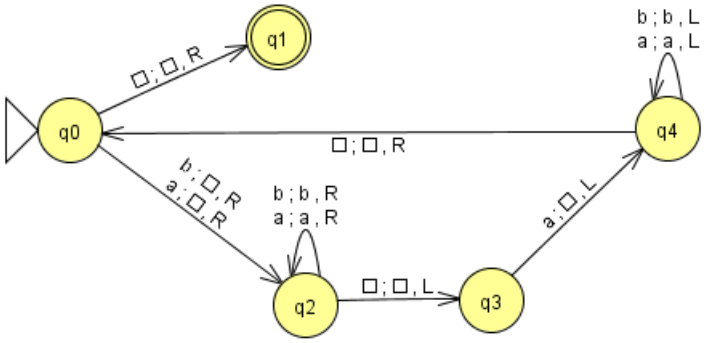


Figure 5.25: Turing Machine for Exercise 2

3) See Figure 5.26

4) See Figure 5.27

5) Accept(T)= aa^*
Reject(T)= $b(a+b)^*$
Loop(T) = $\lambda + aa^*b(a+b)^*$

6) Accept(T)= a^*bb^*
Reject(T)= $a^*ba(a+b)^*$
Loop(T)= $\emptyset$

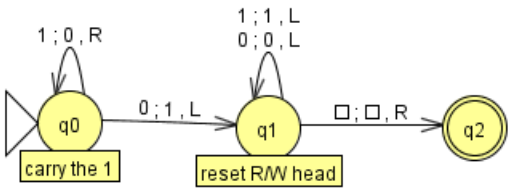


Figure 5.26: Turing Machine for Exercise 3

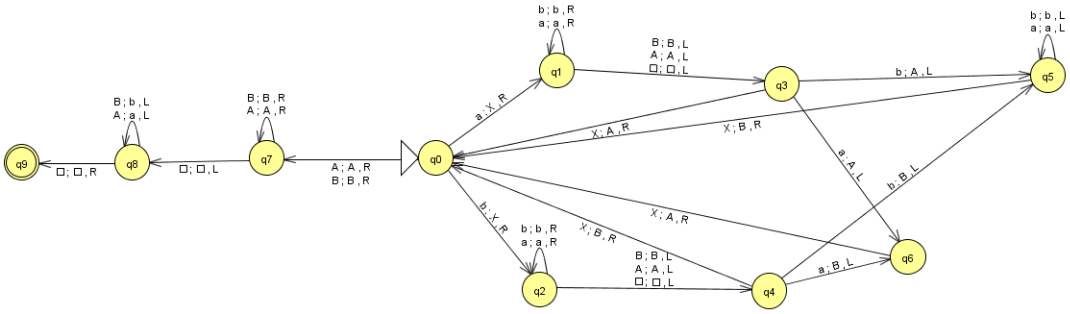


Figure 5.27: Turing Machine for Exercise 4

7) In Lexicographical Order:
aaabaaabaabaaa|*aaabaababaabb*|*abaaabaaabaa*
Mathison

8) In Lexicographical Order:
aaabaaabaabaaa|*aaabaababaabb*|*abaaabaaabaa*
Alan

9) Assume one can decide if two Turing machines accept the same language, and this is decided by a Turing machine Same. We will use this to decide if one Turing machine accepts every string, which was previously shown to be undecidable. We will decide the every string question, by a Turing machine named EveryString. The EveryString machine will just feed its input into the Same machine along with the a Turing machine that immediately halts (and hence halts on all input). Now the Same machine will halt if and only if its input halts on all input. See Figure 5.28

10) Assume we can decide if an encoded Turing machine will accept any string (in other words does there exist any string which the input encoded Turing machine will halt.) Assume we can build the Any Turing machine which decides this question. We will use this machine now to build the EveryString machine which we have already shown is undecidable. We will put a prepocessor on the which erases the input and then feed the result into the Any machine. The Any machine then will halt if and only if the input machine halts on the empty tape. Since deciding if a Turing machine halts on the empty tape is

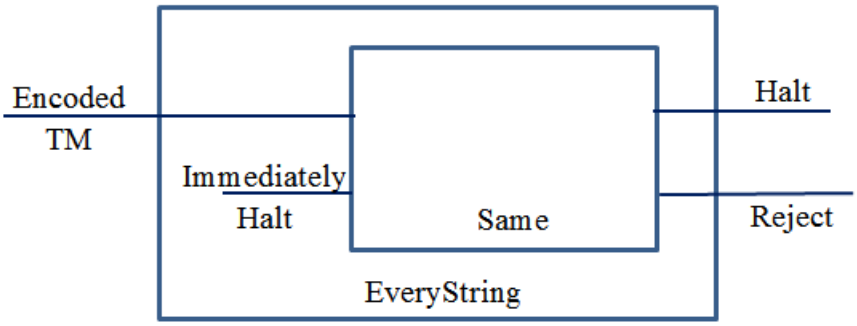


Figure 5.28: EveryString built out of Same Exercise 10

11) In some ways, Turing machine is more powerful. It has an infinite memory so unlike your home computer, it will never run out of memory. However it is much slower than your home computer, as it must move 1 memory cell at a time and the program and data are both stored on the same linear tape.

5.6.2 Computer Activities

1) Write a simulator for a Turing machine whose input is a list of transitions and a finite tape (say 20 characters long). The output should be the the tape after the computation.

2) Write a program whose input is a Turing machine which decides if it is in Alan or Mathison. (Note it may go into an infinite loop, reject it if it takes more than 1,000 steps).

Bibliography

[1] Cohen, Daniel I.A. , "Introduction to Computer Theory, 2nd Edition," John Wiley & Sons, Inc., , (1991).

[2] McCulloch, W. S. and Pitts, W., "A Logical Calculus of the Ideas Imminent in Nervous Activity," *Bulletin of Mathematical Biophysics*, **5**, pg 115 –133, (1943).

[3] Kleene, S. C., "Representation of Events in Nerve Nets and Finite Automata," in Shannon, C. E., and McCarthy, J. (eds) , *Automata Studies*, Princeton Univ. Press, Princeton, NJ, pg 3 –42, (1956).

[4] Myhill, J., "Finite automata and the Represention of Events," Wright Air Development Center Technical Report **57–642**, Wright Patterson Air Force Base, OH, pp. 112- 137, (1957).

[5] Bar–Hillel, Y., Perles, M. and Shamir E. , " On Formal Properties of Simple Phrase Structure Grammars, " Y. Bar–Hillel (ed.), *Language and Information*, Addison–Wesley, Reading, MA pp. 116 - 150, (1964).

[6] Jones, Derek M. . The New C Standard (Excerpted material): An Economic and Cultural Commentary, (2008).

Chapter 6

Complexity Theory

Fools ignore complexity. Pragmatists suffer it. Some can avoid it. Geniuses remove it. Alan Perlis

6.1 Introduction, Reductions and Decisions

Often students ask their instructors, "Will this be hard to solve?" Of course that question cannot be answered with accuracy. What is hard for one rdeems easy, another person finds difficult. The comic in Figure 6.1 is one persons classification of fruit. Many others might disagree.

Mathematicians have attempted to answer the questions of relative difficulty by making an hierarchy of problems which they call complexity classes. Problems are in the same complexity class if the fastest algorithms to solve the problems have similar aymptotic growth rates. Since finding the best possible algorithm is difficult to prove, we can determine complexity classes using a concept called *reduction.*

Definition 6.1.1. If problem A can be solved using an algorithm that solves problem B we say A can be **reduced** to B.

Intuitively, this implies that B is at least as hard as A.

Example 6.1.1. For example if one problem is (**Largest**) : Given a list of integers, find the largest integer, and another problem is (**Sorting**) : Given a list of integers, sort them into descending order. We can say that **Largest** reduces to **Sorting** since by sorting the list and returning the first element, we have found the largest element. Further we can conclude that sorting is at least as hard as finding the largest value. ■

In this way they can definitely compare the difficulty in solving various problems. However classifying a problem can itself be very difficult. Many problems are unrelated and so reduction might not be found. Further since the classification is a statement not just on the speed of a single algorthm but the speed of ***all*** algorithms that solve the problem the analysis can be very complex.

We will limit our discussion to asymptotic growth rates, such as $O()$, and $\Theta()$). By limiting our discussion to asymptotic growth rates the choice of machine will not affect the results. This means what we are able to prove about Turing machines, should be true about most computers. We will

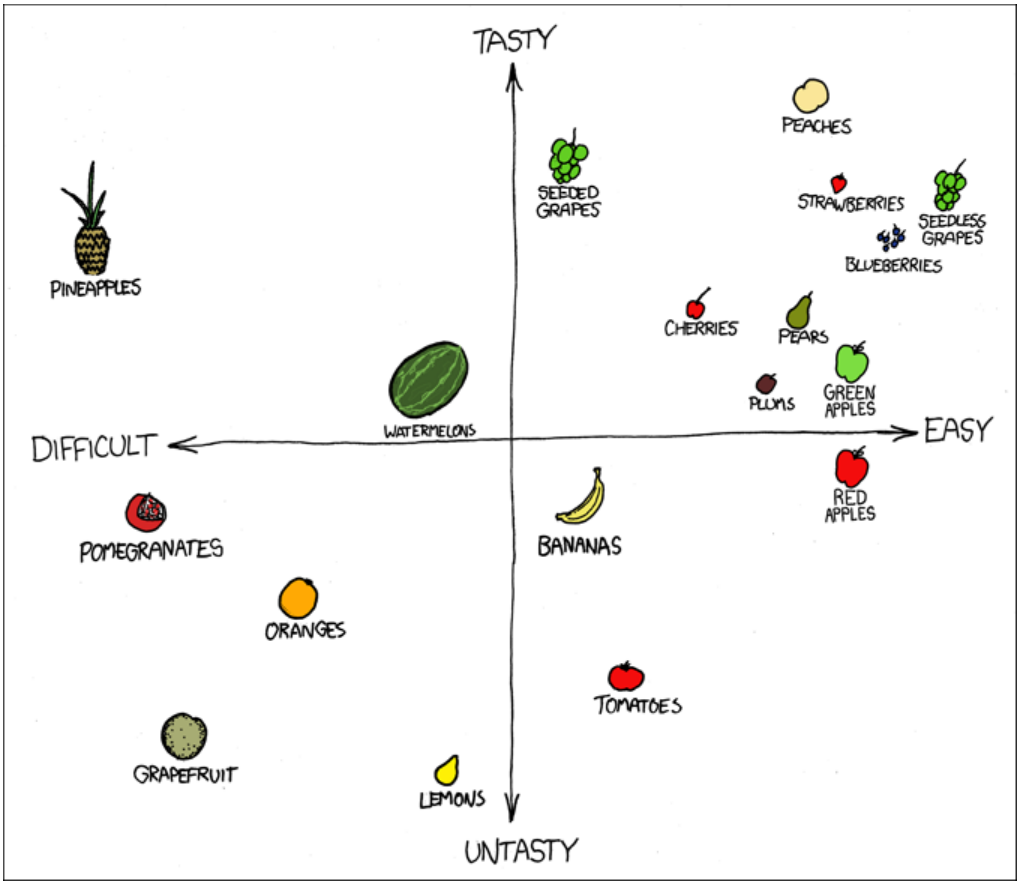


Figure 6.1: XKCD: A webcomic of romance, sarcasm, math, and language.

begin our discussion considering decision problems. Recall that decision problems are ones where there is only a yes or no answer. These should be the easiest problems to analyze. Many of the examples will be language recognition problems since they are all decision problems. The following are examples of decision problems and algorithms to solve them.

Example 6.1.2. Consider the problem of recognizing the language of strings whose last letter is b. One algorithm for this problem would be to examine the last letter of input. If the last letter is b, we will accept, otherwise we will reject the string. On a Turing machine, we would have to walk down the string and the solution would take $O(n)$. On a modern computer we need only look at the n^{th} position and we can solve it in $O(1)$.

Example 6.1.3. Consider the problem of recognizing the language of strings which are palindromes. Our Turing machine solution (see previous chapter) read the first letter, and then walked to the end of the string to determine if it matched. This will take n moves for the first letter. The machine then returns to the front of the string n and determines if the 2^{nd} letter matches the 2^{nd} from the last letter of the input. This traversal will only require $n-1$ moves. Repeating this procedure, we find the total number of moves is given by :

$$\begin{aligned} n+n+n-1+n-1+...1+1 &= 2(n+n-1+n-2+...+2+1) \\ &= n(n+1) = O(n^2) \end{aligned} \tag{6.1}$$

On a modern computer we would have to make $\frac{n}{2}$ comparisons so the time required is $O(\frac{n}{2})$.

The above two examples illustrate finding the big Oh of a *specific* algorithm to solve a decision problem. Many problems have a decision problem counterpart.

Example 6.1.4. Consider the **traveling salesperson problem** . Given a weighted graph $G(V, E, W)$ find the path the lowest edge weight total that traverses all the verticies.
This problem is related to the following decision problem. Given a weighted graph $G(V, E, W)$ and a maximum weight M determine if there exists a path that traverses all the verticies and whose total edge weight is less than or equal to M .
It is clear that the first problem is at least as difficult as the second and so we can use the decision problem to find a lower bound on the complexity of the **traveling salesperson problem**.

However we want to make conclusions about problems over **all possible** algorithms. We begin by defining which problems are feasible to solve on large data sets.

6.2 The classes P and NP

We will first divide up problems into those which are tractable (can be solved for large instances of the input) and those that are intractable. The Cobham–Edmonds thesis (named after Alan Cobham [2] and Jack Edmonds) asserts that computational problems can be scaled up (i.e. *tractable*) and solved if there exists a worst case time algorithm to solve them takes $O(n^\alpha)$ time where n is the size input and α is a constant. In other words if the asymptotic time is a polynomial of the size of the input.

Definition 6.2.1. The class **P** is the set of decision problems where there is an algorithm which can solve the problem in time $p(n)$ worst case time, where p is a polynomial.

Some problems in the class **P** which you have probably studied are : sorting a list of numbers, searching a list of numbers, finding the shortest path in a graph and matrix multiplication.

Conversely, we can now also define "intractable" as those problems whose growth rate is so large that it is impractical to solve them for large instances of input.

Definition 6.2.2. The set of problems which cannot be solved in polynomial time are called **intractable**.

Example 6.2.1. Consider the problem of generating the power set from a given set. Recall that if a set has n elements, then the power set contains 2^n elements. Hence any algorithm to solve this problem will require at least $O(2^n)$ time just to write the output. So this problem is inherently intractable.

Other famous "intractable" problems include: finding a Hamiltonian cycle in a graph, factoring integers, and deciding if there is an input for a boolean circuit that will result in an output of true.

It should be noted that for some specific cases some "tractable" problems are impractical to solve, while some "intractable problems" are solvable. For example if the algorithm requires 2^n time but $n < 20$ or if the algorithm requires $2^{0.0000001n}$ time then large instances can indeed be solved. On the other hand if the problem grows as n^{10} only very small problems can be solved. In general if a solution is grows faster than $O(n^3)$ it is usually too slow to solve instances much greater than one thousand. Even if computers double in speed every 18 months, it is difficult to overcome the exponential time increases to solve larger problems.

Below is a small result concerning the class P and language recognition.

Theorem 6.2.1. *The set of languages in the class P is closed under unions.*

Proof Let L_1 and L_2 be 2 languages in P. Then we know that there exists Turing machines T_1 and T_2 which decide L_1 and L_2 respectively, and such that T_1 works in $O(n^\alpha)$ time and T_2 works in $O(n^\beta)$ time where n is the size of the input. We can construct a 2 track Turing machine to accept $L_1 \cup L_2$ by putting the input on both tracks. The machine alternates 1 step of T_1 on track 1 followed by 1 step of T_2 on track 2. The composite machine will finish in time $O(n^{max(\alpha,\beta)})$ time. Since this is a polynomial, we have that P is closed under unions. ■

Another important class problems we need to discuss is the set **NP**. Contrary to many students intuition, **NP** does *NOT* mean not polynomial. Instead, the class name refers to problems that can be solved with a non–deterministic algorithm in polynomial time. However it much easier for students to consider the following equivalent definition of **NP**.

Definition 6.2.3. The class **NP** is the set of decision problems where there is a algorithm, which when given an input and a "proposed solution" which generates a "yes" decision, we can verify the solution in polynomial time of the size of the input. Note that we do not have to have an polynomial time verification algorithm for "no" decisions.

It must be noted that $\mathbf{P} \subseteq \mathbf{NP}$ since all problems that can be solved in polynomial time can be verified in polynomial time. The following is an example of a **NP** problem. In this case it is much easier to not consider a Turing machine but an actual computer to verify that is in the class **NP**.

Example 6.2.2. Membership in **NP**
Consider the problem of determining whether a given graph has a Hamiltonian cycle in **NP**. Recall that a Hamiltonian cycle is a simple cycle which visits every interior vertex exactly once. A cycle must start and end on the same vertex.

Now if the input was a graph and sequence of vertexes (said to compose a Hamiltonian cycle) once could easily verify the Hamiltonian cycle by verifying that adjacent vertexes in the proposed solution are adjacent in the graph, not repeated and contain all the vertexes in the graph. The time it would take for such a verification depends on the data structure used to hold the graph. For example if an adjacency matrix is used, one can test if 2 vertexes are adjacent in $O(1)$ time. On the other hand if an adjacency list is used, the test for 2 vertexes being adjacent is $O(n)$ time, where n is the number of vertexes in the graph. Regardless however, to check the correctness of a proposed solution to the Hamiltonian cycle problem will require no more than $O(n^2)$ time. So the problem is in **NP**. ■

It is imperative to note that we classified this problem without ANY reference to the a solution. Complexity classifications depend only the problem, and NOT on any proposed the solution.

Example 6.2.3. Membership in **NP**
Consider the problem of given a set of integers, $A = \{a_1, a_2, ..., a_n\}$, and a target integer t to determine if there exists a subset S of A whose elements total t. We will show this problem is in **NP**. Here a proposed solution would be the subset A. We need only sum the elements of A and see if the total is t. This would require only $O(n)$ time so the problem is in **NP**. ■

The class **NP** represents the problems which are the considered the most difficult of the potentially tractable problems. Clearly if we cannot verify a solution in polynomial time, then we it is unlikely that we could solve it in polynomial time. We will now define the most general problems of this class with the property that if we can solve these most difficult of the difficult problems in polynomial time we can solve all the problems in the class in polynomial time.

6.3 Polynomial Time Reductions

Reductions are sometimes used in mathematics is to convert one problem into one that that we already know how to solve.

Example 6.3.1.
Consider an example from linear algebra when we attempt to solve the matrix equation

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \tag{6.2}$$

for the $n \times n$ matrix A and the $1 \times n$ vectors x and b. Many methods (such as Gaussian elimination) attempt to transform the problem to an equivalent problem to n scaler equations of the form:

$$\begin{aligned} d_1 x_1 &= c_1 \\ d_2 x_2 &= c_2 \\ &\vdots \\ d_n x_n &= c_n \end{aligned} \tag{6.3}$$

for $i = 1, \ldots, n$. Since scaler problems are easy to solve. The solution to the system of scalar equations solves the matrix equation. In the case the transformation into a system of linear equations requires *MORE* work than the solving the system. Such transformations are not helpful in forming complexity classes.

To distinguish **P** from **NP** we will need reductions that can be done in polynomial time. We will formalize this notion which considers the time of the reduction for problems in the following definition.

Definition 6.3.1. Polynomial Time Reductions for Decision Problems
Consider the following decision problems A and B. We say that problem A **reduces** to problem B if we can find a mapping f of instances of $a \in A$ into instances of $b \in B$ such that if the solution to a is yes, then $f(a)$ is yes, and if the solution to a is no, then $f(a)$ is no. This is denoted by $A < B$. If the transformation f can be computed in polynomial time, then we call f a **polynomial time reduction** and denote this by $A <_p B$.

Similarly, we can define polynomial time reductions for general problems.

Definition 6.3.2. Polynomial Time Reductions for General Problems
Consider the following problems A and B. We say that problem A **reduces** to problem B if we can find a mapping f of instances of $a \in A$ into instances of $f(a) \in B$ such that f can be computed in polynomial time and the solution to a is found using at most a polynomial number of applications of the algorithms uses to solve $f(a)$ plus some polynomial time inverse transformation.

The following are examples of polynomial time reductions.

Example 6.3.2. Consider **problem1** : Given n logical variables $\{x_1, x_2, ...x_n\}$, does at least one of them have the value of true ?

and consider **problem2** : Given n integers $\{y_1, y_2, ...y_n\}$is the largest one positive ?

Consider the transformation f which maps x_i into 1, if x_i is true and into 0 if x_i is false. Again, it is essential here to note that we are NOT solving either problem. We are just mapping problems of one type into problems of the other type.
It is clear that f maps instances of the **problem1** where the answer is yes, into instances of **problem2** where the answer is yes and maps instances of the **problem1** where the answer is no, into instances of **problem2** where the answer is no. Since f can be computed in O(n) time we can say that **problem1** $<_p$ **problem2**. ■

Below is a slightly more complex example using languages.

Example 6.3.3. Consider the language of even palindrome (**EvenPal**) over the alphabet $\{a, b\} = \{aa, bb, aaaa, abba, baab, bbbb, ...\}$
and the language of double words (**Double**) over the alphabet $\{a, b\ \} = \{ww | w \in (a + b)^*\ \} = \{aa, bb, aaaa, abab, baba, bbbb, ...\}$

We will construct a polynomial time reduction from **EvenPal** to **Double**. The idea of the transformation is easy : reverse the last half the word. Since a word w of length n where $w = w_0w_1...w_{n-1}$ is $\in$ **EvenPal** if and only if n is even and $w_i = w_{n-i-1}$ for $i = 0, 1, ...\frac{n}{2} - 1$. While the word w is in **Double** if and only if $w_i = w_{\frac{n}{2}+i}$ for $i = 0, 1, ...\frac{n}{2} - 1$.

Thus our mapping $f(w) = w_0w_1...w_{\frac{n}{2}-1}w_{n-1}w_{n-1}w_{n-2}...w_{\frac{n}{2}}$

Now to implement this on a Turing machine we first have to determine the center of the word. We can do this by capitalizing the first letter. Move to the end of the word ($O(n)$ moves) and capitalize the last letter. Then move back to the first uncapitalized letter $O(n-2)$ moves. We then repeat the process. So the total number of moves is

$$O(n+n-2+n-2+n-4+n-4+...2+2) = O(n^2).$$

Once the center of the word has been found we than move back and forth swapping letters the in the last half of the word. This process is similar to that of finding the center but is performed only on the last half of the word and now we use uncapitalized letters to to show that we have processed the letters. So the total time for this process is $O((\frac{n}{2})^2)$. Finally we walk through the first half of the word converting the the remaining capital letters to small letters in time $O(n)$.

Thus the total time is $O(n^2)$. So if problem 1 is : Is the input string a word in EvenPal ?, and problem 2 is : Is the input string a word in Double ? Then $f()$ maps instances of problem 1 which are yes, into instances of problem 2 which are yes maps words in language 1 and it maps instances of no for problem 1 into instances of no for problem 2. Since f can be computed in $O(n^2)$ time we can say that problem 1 $<_p$ problem 2, which implies that problem 2 is at least as difficult to solve as problem 1. ■

We will now consider some general polynomial time reductions to sorting.

Example 6.3.4. Our reduction of from " finding the largest value of a list of numbers (**largest**)" into "sorting a list of numbers into descending order (**sorting**)" is a polynomial time reduction. This is denoted **largest** $<_p$ **sorting**. The transformation takes $O(1)$ time since we do nothing and once the list is sorted the remaining required time is $O(1)$ since we just read off the first value. ■

Example 6.3.5. Our reduction of from " determine the unique values in a list of numbers (**unique**)" into "sorting a list of numbers into descending order (**sorting**)" is a polynomial time reduction. The transformation takes $O(1)$ time since we do nothing. Once the list is sorted we walk through list and count the number of times adjacent elements differ. This takes $O(n)$ time. ■

6.4 NP-Complete

Now that we have a method for comparing the complexities between different decision problems, we can identify the hardest of the **NP** problems which are called **NP-Complete** or **NPC**.

Definition 6.4.1. A problem Q is called ***NP-complete*** if $Q \in$ **NP** and for all problems $P \in$ **NP**, $P <_p Q$.

What this means is that an **NP-complete** problem is at least as hard as any other **NP** problem and that if any **NP-complete** problem can be solved in polynomial time of it's input, then *every* **NP** problem can be solved in polynomial time. Since it is clear that **P** $\subseteq$ **NP** then if we solve one **NP** problem in polynomial time, then **P** = **NP**. Indeed this is still an open question after more 50 years of research and even millions of dollars of prize money being offered to settle the question.

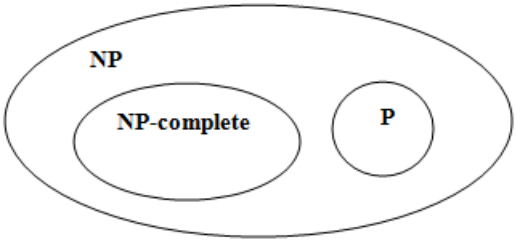


Figure 6.2: Artists conception of complexity classes

Although it would seem that **NP–complete** problems would be few and far between researchers have identified hundreds of them [3]. One such problem is related to the "mine sweeper" game found on many personal computers [4]. The problem is : Given a minesweeper grid, is it consistent? Other well known **NP–complete** problems include : The traveling salesperson problem, Linear Programming, and Circuit Satisfiability.

Finding the first **NP–complete** problem is difficult. However once one is found, finding others is much easier. Given the P is an **NP–complete** problem we can prove that Q is an **NP–complete** problem in two steps:

1) Prove that $Q \in$ **NP** and

2) that $P <_p Q$.

The next examples will illustrate the process of proving a problem is **NP-complete**

Example 6.4.1. In this example we will show that if the **Independent Set** problem is **NP–complete** then so is the **Vertex Cover** problem. The **Independent Set** problem can be stated as : Given a graph $G(V, E)$ and an integer k, determine if the graph have a subset of vertices $S \subseteq V$ with the property that no two vertices in S are adjacent and where $|S| \geq k$.

The **Vertex Cover** problem can be stated as : Given a graph $G(V, E)$ and an integer k , determine if the graph have a subset of vertices $T \subseteq V$ with the property that every edge $e \in E$ has an endpoint in T.

We first show that **Vertex Cover** $\in$ **NP**. Consider a proposed solution to **Vertex Cover** is the set of vertices T. Now we can verify that T is a vertex cover by testing the end points of all edges. Hence the time required for the verification is O($|E|$).

The mapping in this case will be almost trivial. The graph in the instance of **Independent Set** will be the same graph as in the target instance of **Vertex Cover**. However, the k in **Independent Set** is transformed into $|V| - k$ in the instance of **Vertex Cover**. It is clear then the transformation would take time O($|V|$).

The transformation works because S is an independent set if and only if $V - S$ is a vertex cover. Suppose that S is an independent set. Consider an edge $e \in E$ where $e = (u, v)$. Since

S is independent one or both of u and v are not in S. Hence each edge has an endpoint in the complement of V , $V - S$ and we can conclude that $V - S$ is a vertex cover.

Now assume that $V - S$ is a vertex cover and consider any two vertices in u, $v \in S$. If there is an edge connecting u with v, then this contradicts that $V - S$ is a vertex cover. Hence no two vertices in S are adjacent and S is an independent set.

Thus **Independent Set** $<_p$ **Vertex Cover** and if **Independent Set** is **NP–complete** so is **Vertex Cover**. ■

The following example will make use of the **NP–completeness** of **Vertex Cover** to prove another problem is also **NP–complete**. This illustrates how we can generate a chain of **NP–complete** problems and also how complexity can make statements about the relative difficulty to a vast set of problems. However it must be noted that not every pair of problems can be compared. The complexity classes form what is known as a *partial order* among problems.

Example 6.4.2. In this example we will show that the **Set Cover** problem is **NP–complete**. This decision problem can be stated as given a set U of n elements and a set of $S_1, S_2, ..., S_m$ of subsets of U, and an integer k, does there exist a set of k subsets whose union is U?

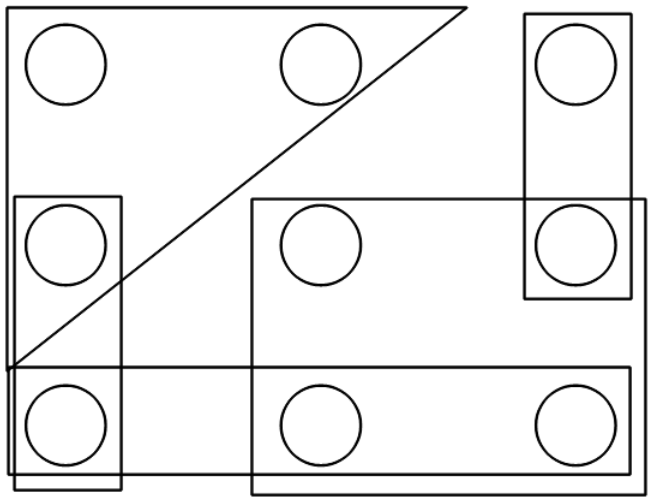


Figure 6.3: An instance of a **Set Cover** Problem

Figure 6.3 depicts an instance of a **Set Cover** problem where the nine circles represent the elements and the rectangles and triangles represent the subsets. In this instance the set can be covered with 4 subsets but not with 3 subsets.

Consider for example you want to learn nine software packages (represented by the circles). These software packages are taught in several courses (each represented by a polygon). You want to seek a few number of courses to learn all the software packages. In the example above one would need to take four classes to gain all the required knowledge.

First we will show that **Set Cover** is in **NP**. Consider a proposed solution to a **Set Cover** problem. It would be the sets $\{ S_i\}$whose union is U. To verify this solution on would check each of the elements in U against each of the elements in $\{S_i\}$. If we assume a primitive data structure each check would take O($|U|$) time so the verification would take O($|U^2|$) time, and hence **Set Cover** is in **NP**.

To prove that **Set Cover** is **NP–complete** we will solve **Vertex Cover** by transforming it into an instance of **Set Cover**. In **Vertex Cover** we "cover" the edges of a the graph with verticies. So the transformation should map edges onto the elements of U to be covered, while each vertex v_i will map into a subset S_i. In our transformation S_i will contain all the edges which have v_i as an end point.

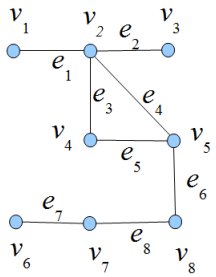


Figure 6.4: An instance of a **Vertex Cover** Problem

The **Set Cover** instance associated with the instance of **Vertex Cover** pictured in Figure 6.4 is : $U = \{e_1, e_2, ..., e_8\}$, $S_1 = \{e_1\}$, $S_2 = \{e_1, e_2, e_3, e_4\}$, $S_3 = \{e_2\}$, $S_4 = \{e_3, e_5\}$, $S_5 = \{e_4, e_5, e_6\}$, $S_6 = \{e_7\}$, $S_7 = \{e_7, e_8\}$, $S_8 = \{e_6, e_8\}$.

It is clear from our construction that the graph G in the instance of **Vertex Cover** problem has a cover of size k if and only if the resulting **Set Cover** problem has a cover of size k. The cover for G is adjacent to all the edges in G. Hence the resulting subsets contain all the elements of U. Conversely a set cover must contain all the edges, so the corresponding vertices also touch all the edges. Hence **Vertex Cover** $<_p$ **Set Cover**.

Our final example relates to boolean expressions . These are very general formulations and can describe problems in decision making, artificial intelligence, and circuits. The reduction is the most difficult of the examples and illustrates how a reduction can be between two very different realms.

Example 6.4.3. Consider the **CNF–SAT** problem. This decision problem asks if given a logical expression (given in conjugative normal form) whether one can consistently assign the variables **true** or **false** values so that logical expression is true. Recall that a logical expression in conjugative normal form is a a conjunction (**AND**s)of clauses, where a clause is a disjunction (**OR**s)of literals. And where a literal is logical variable or it's complement (which is denoted by a bar over the variable).

For example consider the logical expressions in **CNF**

$$(x_1 \vee x_2) \wedge (\bar{x}_2 \vee x_3) \wedge (x_3) \tag{6.4}$$

$$(x_2) \wedge (\bar{x}_2 \vee x_3) \wedge (\bar{x}_3) \tag{6.5}$$

Equation 6.4 can be satisfied with the assignment (**true**, **true**, **true**) for (x_1, x_2, x_3) but no choice of values can make equation 6.5 evaluate to **true**.

The **CNF–SAT** problem is **NP–complete**. This will be stated without proof. The proof is fairly long and is specific to the **CNF–SAT** problem. We will only make use of the fact that is **NP–complete** to illustrate the two–step method given above to prove that **Clique** is also **NP–complete**.
The **Clique** problem is : Given a graph $G(V, E)$ and an integer k, determine if G has a complete subgraph (or clique) of size at least k. Recall that a complete graph is a graph where every vertex is connected to every other vertex. The complete graph containing i vertexes is denoted K_i and shown in Figure **??** below.

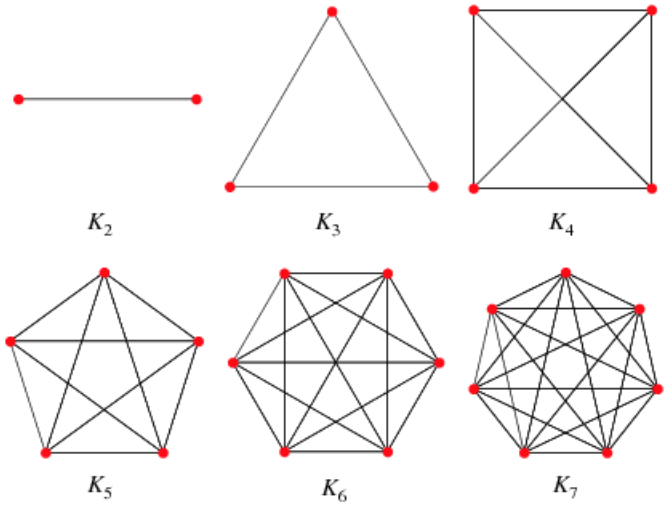


Figure 6.5: Some Complete Graphs [5]

First we will show that **Clique** is in **NP**. Given an instance of **Clique** (which is a graph $G(V, E)$ and an integer k), as well as a list of vertexes which form the clique. We can verify that the vertexes form a complete graph in time $O(k^2)$ time by checking if each vertex is indeed connected to the other $k - 1$ vertexes in the graph.

Now we need only find a polynomial time transform from a **CNF–SAT** problem into a **Clique** problem that maps instances of "yes" into "yes" and "no" into "no". Now to satisfy an expression in **CNF** we need to find one literal in each clause that will evaluate to **true**. We will make these **true** literals the vertexes that form the complete subgraph. If we label each vertex by the literal it represents and the clause that it is in, then $V = (x, i)$ for each literal x and clause i.

We will connect each vertex with the vertexes in the other other clauses which do not contradict it. Hence $E = ((x,i),(y,j))$ if $i \neq j$ and $\bar{x} \neq y$..
Consider the following clause

$$(x_1) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3) \tag{6.6}$$

There will be seven vertexes (one for each of the seven literals). Vertex (x_1,1) will be connected every vertex except ($\bar{x}_1$,2) and ($\bar{x}_1$,3). Vertex ($\bar{x}_2, 2$) will be connected to every vertex except those in its clause ($\bar{x}_1$,2) and (rx_3,2) and it will not be connected to ($\bar{x}_2, 3$) since x_2 and $\bar{x}_2$ cannot both be **true** at the same time. In a similar manner the following graph is constructed.

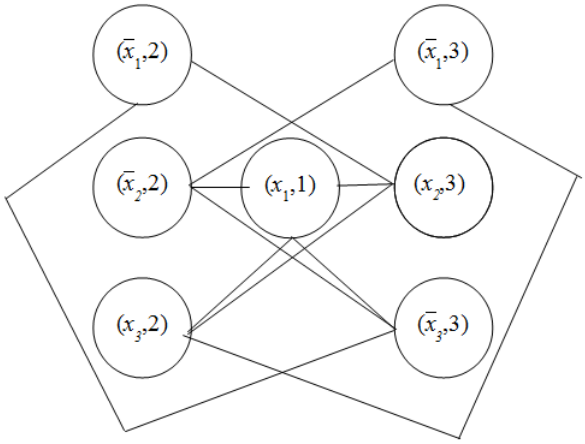


Figure 6.6: Graph from expression 6.6

So there are two cliques of size 3 in the resulting graph : {(x_1,1), (x_2,2),(x_3,3) } and {(x_1,1), ($\bar{x}_2$,3),($\bar{x}_3$,3) } which correspond to the two assignments of **true** which make expression 6.6 **true**.

We now proceed with the proof. The mapping of **CNF** expressions into graphs is well defined above. The k in the instance of a **Clique** is the number of clauses in the expression. The mapping will take O(n^2) where n is the number of literals in the expression. First we create the vertices which require O(n) time. For each literal, we need to examine the rest of the expression to determine if an edge is to be drawn between the literal and the rest of the literals. So the total time required a polynomial of the size of the input. We know show that instances of satisfiable expressions map into instances of graphs with cliques of size at least size k, and unsatisfiable expressions map into instances of graphs with cliques of size less than k.

If the expression is satisfiable, then there are truth assignments such that x_1, x_2, , x_n which make the expression **true**. This means there is at least one literal in each clause that is **true**. Picking that vertex from each clause forms a clique of size k.

If the expression is not satisfiable, then we cannot select a vertex in each clause to connect, so any cliques formed must be of size less than k.

Hence **SAT** $<_p$ **Clique** and **Clique** is **NP–complete**. ■

6.5 Lower Bound Theory

6.6 Conclusion

Determining the complexity of problems is *HARD*. It is often easier to solve a problem than to determine it's complexity. However, knowing the difficulty of a problem can help determine how close you are to an optimal solution. Further, being able to recognize **NP–complete** problems is a skill that can save much time and effort wasted in attempting to come up with a polynomial time solution for a problem that has perplexed the greatest minds of the last half century.

Although as of this writing, no one has found a polynomial time solution to an **NP–complete** problem, there are plenty of polynomial time approximation algorithms that can often be " good enough" and "fast enough" to solve specific needs. Studying the "approximation algorithms" is a worthy study but beyond the scope of this class.

6.7 Exercises

1) Prove that the set of languages in the class **P** is closed under intersections.

2) Prove that the set of languages in the class **P** is closed under unions.

3) Prove that the following problem is in **P**:

   Given an a list of integers and an integer k determine the k^{th} largest value.

4) Prove that the following problem is in **P**:

   Given an a list of integers determine if the list is sorted.

5) Prove that the following problem is in **NP**:

   Given an a list of integers determine if the list is sorted.

6) Prove that the following problem is in **NP**:

   Given an integer determine if the it is prime.

7) Prove that the following problem is in **NP**:

   Given a graph $G(V, E)$, and integer k determine if the verticies of G can be colored with k so that adjacent verticies of G have different colors.

8) Prove that the **Independent Set** problem described in Example 6.4.1 is in **NP**.

9) Prove that the **CNF-Satisfiability** problem described in Example 6.4.3 is in **NP**.

10) Consider **integer factorization** problem. Given integers n and k does there exist another integer f such that f divides n evenly and $1 < f < k$. Argue that the integer factorization problem is in **NP**.

11) Find a polynomial time reduction from a^n where n is a perfect square, to b^n where n is a perfect square.

12) Find a polynomial time reduction from $(ba)^n b^n$ to $a^n b^n$ where n is prime number.

13) Find a polynomial time reduction from the language (over the alphabet $(a+b)^*$) of a prime number of a's to the language (over the alphabet $(a+b)^*$), a prime number of b's.

14) Find a polynomial time reduction from Double Word, (over the alphabet $(a+b)^*$), to the language Trailing count which is $\{wa^{|w|} | w \in (a+b)^* \}$.

15) Find a polynomial time reduction from the problem of finding the median of a list of integers, into the problem of finding the k^{th} largest value of a list of integers.

16) Describe a Polynomial Time reduction from Equal = set of words with equal #s of as and bs and the $a^n b^n$. Argue that it takes polynomial time to evaluate your transformation.

17) Let **problem1** be " Given a list of integers, pair that is closest in value," and let **problem2** be " Given a list of integers, sort them". Find a polynomial time reduction from **problem1** to **problem2**.

18) Let **problem1** be " Given a list of integers, pair that is farthest (in value)," and let **problem2** be " Given a list of integers, find the largest value". Find a polynomial time reduction from **problem1** to **problem2**. *Hint: use the solution to two instances of* **problem2**

19) Given that the **Hamiltonian Cycle** problem is **NP–complete**, prove that the **Traveling Salesperson** is also **NP–complete**.

    The **Hamiltonian Cycle** problem is : Given a graph $G(V, E)$ determine if the graph has a cycle that visits each vertex exactly once.

    The **Traveling Salesperson** problem is : Given a weighted graph $G(V, E, W)$ and maximum weight M, determine if the graph has a cycle that visits exactly once and whose total weight of the edges in the cycle is less than or equal to M.

6.7.1 Answers to Exercises

1) Consider two languages $p \in \mathbf{P}$ and $q \in \mathbf{P}$. Since they are members of $\mathbf{P}$ then there exists Turing machines M_p and M_q which accept p and q respectively. We also know that both machines will run in polynomial time. So without loss of generality we can say that M_p runs in $O(n^a)$ time and M_q runs in $O(n^b)$ for some constants a and b. To show that $\mathbf{P}$ is closed under intersections, we need to construct a Turing machine $M_{p \cap q}$ which accepts the language $p \cap q$ and runs in polynomial time. Since we have shown that a multi–tape Turing machine is equivalent to a regular Turing machine, it is sufficient to construct $M_{p \cap q}$ as a two–tape Turing machine. We will place the input on both tapes, and run the M_p on the upper tape and M_q on the lower tape. If both machines finish on their respective tapes we accept. The time for $M_{p \cap q}$ to run is $O(n^{max(a,b)})$ and hence $p \cap q \in \mathbf{P}$.

2) Consider two languages $p \in \mathbf{P}$ and $q \in \mathbf{P}$ accepted by M_p and M_q respectively, as describe in the previous problem. To show that $\mathbf{P}$ is closed under unions, we need to construct a Turing machine $M_{p \cup q}$ which accepts the language $p \cup q$ and runs in polynomial time. Again we will build a two–tape Turing machine running M_p on the top tape and M_q on the bottom tape. However, this time we alternate steps, making one move for M_p followed by one move for

M_q. Now if either machine halts we accept and the composite machine again runs in time $O(n^{max(a,b)})$. Note that we had to alternate steps in this example since there is no guarantee that the first Turing machine will ever finish. Hence **P** is closed under unions.

3) If we sort the list of integers, ($O(n\lg(n))$ time), and then select the k^{th} largest item, we have solved the problem in **P** time.

4) Assume the list is the elements x_i, $i = 1, 3, \ldots, n$. Now for each $i < n$, test if $x_i < x_{i+1}$ If this is true for all i, then list is sorted, otherwise the list is not sorted. The time for this algorithm is $O(n)$ is the problem is in **P**.

5) To show the problem is in **NP** we need to verify a"yes" solution in **P** time. The algorithm in the previous problem can be used for this purpose, in $O(n)$ time.

6) To verify an integer n is prime, (a "yes" solution), we need only test if n mod i is 0 for each value of $i \leq \sqrt{n}$. Since a solution can be verified in polynomial time, the problem is in **NP**.

7) A "yes" solution to this problem would be a list of vertices and the color associated with each vertex. To verify the solution, one would need to ensure that for each vertex, the adjacent vertices are not of the same color, as well as the total number of colors is less than or equal to k. If the graph is stored as an adjacency matrix, the check would require $O(|V| + |E|)$ time. Since the input to this problem requires $O(|V| + |E|)$ space, the verification can be done in polynomial time of the size of the input and the problem is in **NP**.

8) A "yes" solution to the **Independent Set** problem would be a set of vertices S. To verify this proposed solution, one would have to verify that the cardinality of $S \geq k$ as well as ensuring that the elements of S are not adjacent. The first condition would require O(k) time to verify while the second condition would require $O(|V| + |E|)$ time if the graph where stored as an adjacency matrix. Since the input to this problem requires $O(|V| + |E|)$ space, the verification can be done in polynomial time of the size of the input and the problem is in **NP**.

9) A "yes" solution to the **CNF–Satisfiability** problem would a set of truth assignments for each logical variable which reportedly make the CNF expression true. This can be verified by making those assignments and evaluating the logical expression. Since the time to evaluate the expression is proportional to the length of the expression, the verification can be done in linear time. Hence the **CNF–Satisfiability** problem is in **NP**.

10) A "yes" solution to the **integer factorization** problem would be an integer f. We need only verify that f ¡ k and that n mod $f = 0$, which can be done in constant time. Since a"yes" solution can be verified in polynomial time the **integer factorization** problem is in **NP**.

11) Given an word w placed on the tape of a Turing machine. We need only move to the right changing each a we encounter into a b and each b we encounter into an a. Now the original word is in the language a^n if and only if the output is in the language b^n. Since the reduction took $O(|w|)$ time. We have the the language $\{a^n\} <_p thelanguage \{b^n\}$.

12) Given an word w placed on the tape of a Turing machine. We will move to the right changing each ba we encounter into a single a. As we make each change, we will have to shift all the

letters to the right of the change 1 square over. Each such shift requires $|w|$ time the entire transformation will require $O(|w|^2)$ time on a Turing machine. Now the original word is in the language $(ba)^n b^n$, where n is a prime number, if and only if the output is in the language $a^n b^n$ where$n$ is a prime number. Since the reduction took $O(|w|^2)$ time. We have the the language $\{(ba)^n b^n\}$ where n is a prime number $<_p$ the language $\{a^n b^n\}$ where n is a prime number.

13) Given an word w placed on the tape of a Turing machine. We will move to the right changing each a to a b and each b to an a. The time for this transformation is $O(|w|)$. The input word is in the language $\{a^n\}$ where n is prime if and only if the output word is in the language $\{b^n\}$ where n is prime. Thus we have shown the language $\{a^n\}$ where n is prime $<_p$ the language $\{b^n\}$ where n is prime.

14) Given an word w placed on the tape of a Turing machine. We will find the middle of the word and then see if the i^{th} letter matches the $(\frac{|w|}{2} + i)^{th}$ letter and if so, then replace the the $(\frac{|w|}{2} + i)^{th}$ letter with a if they do not match, we will overwrite the $(\frac{|w|}{2} + i)^{th}$ letter with b . One way to find the middle of the word on a Turing machine requires one to capitalize the first letter and the last letter of the word and repeat on all the non–capital letters. When no small letters are left, then the read/write head is in the center of the input word. The time for this $O(|w|^2)$. Then to change all the last half letters to a will also require $O(|w|^2)$ time. Finally, we would have to make all the capital letters small letters in $O(|w|)$ time. Since the input is in Double Word if and only if the output is in Trailing count, and the transformation takes $O(|w|^2)$ time, we have that Double Word $<_p$ Trailing count.

15) Consider a list of integers $\{x_i\}$, $i = 1, 2, ..., n$. To reduce the median problem into the k^{th} largest problem we need only set k to $\frac{n}{2}$. This reduction takes $O(1)$ time, so median $<_p$ k^{th} largest.

16) Consider a word w placed on the tape of a Turing machine. Start at the front of the word and find the first b and change it to an X. Go to the end of the word and move left to the last a in the word. Overwrite the last a with B and then find the X and over write it with an A. This requires $O(2|w|)$ time. Repeat this until all the letters are capital letters. The total time required to preform this rearrangement is $O(|w|^2)$. The original word is in Equal if and only if the output is in $\{a^n b^n\}$. Since the transform took polynomial time we have that Equal $<_p$ $\{a^n b^n\}$.

17) To solve **Problem1**, we need only sort the list (**Problem2**) and then check adjacent elements to find the closest pair. The reduction takes $O(1)$ time but to solve **Problem1** we needed an extra $O(n)$ time where n is the length of the original list. Hence **Problem1** $<_p$ **Problem2**.

18) To solve **Problem1**, we need find the largest element (**Problem2**) which will be one of the numbers in the pair. We then multiply the numbers by -1 and again solve **Problem2**. The negative of this result will the other number in the pair. The reduction takes $O(n)$ time to multiply the list by negative 1 . Hence **Problem1** $<_p$ **Problem2**.

19) To reduce a **Hamiltonian Cycle** problem to the **Traveling Salesperson**, we need to change the graph $G(V, E)$ in the **Hamiltonian Cycle** instance, into a weighted graph $G(V, E, W)$.

We do this by making each edge weight 1. The bound M is chosen to be to be $|V|+1$. Now if the original graph has a Hamiltonian cycle if and only if the instance of traveling salesperson has a solution.

6.7.2 Computer Activities

1) Test the complexity of sorting by sorting a random input list of 100, 200, 400, 800, 1600 and 3200 integers using quicksort, and bucketsort 1,000 times each. Use the system clock to measure the elapsed time.

2) Implement a solution to double word and then solve even palendrome by using reduction.

3) Implement Exercise 16.

Bibliography

[1] Cook, Stephen, "The complexity of theorem proving procedures". *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pp. 151158, (1971).

[2] Cobham Alan, "The intrinsic computational difficulty of functions", *Proc. Logic, Methodology, and Philosophy of Science II*, North Holland(1965).

[3] Garey, M., and Johnson D., *Computers and Intractability : A Guide to the Theory of NP-Completeness*, W.H. Freedman, San Francisco, 1979.

[4] Kay, R.W., "Minesweeper is NP-complete," *The Mathematical Intelligencer*, vol 22, **2**, pg 9 –15, (2000).

[5] http://mathworld.wolfram.com/CompleteGraph.html

Chapter 7

Computational Geometry

Equations are just the boring part of mathematics. I attempt to see things in terms of geometry. Stephen Hawking

7.1 Introduction

Computational geometry is a collection of algorithms used to efficiently solve geometric problems. The field of computational geometry grew out of applications in the of computer graphics, chemistry, statistical analysis, and pattern recognition. As computers and their applications have become more pervasive, computational geometry has developed even more applications in robotics, mechanical engineering and biology. This chapter serves only as an introduction to a few of the best known algorithms which will prove useful in a wide variety of settings. and will concentrate on polygons, segment intersection, convex hulls, and triangulations. Although there are a wide variety of applications where geometry can be used, the comic below illustrates that there some problems that even geometry cannot solve.

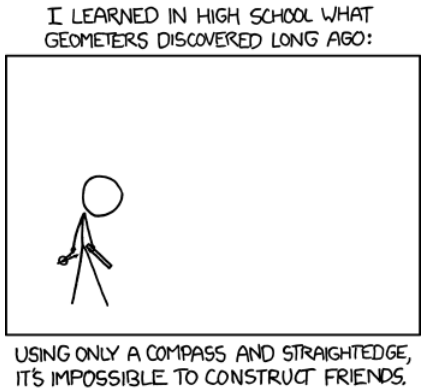


Figure 7.1: XKCD: A webcomic of romance, sarcasm, math, and language.

7.2 Vectors and Inner and Cross products

We begin with a review of some concepts from linear algebra. Recall that another way to describe objects in the plane or in space is to use vectors. Ordered pairs are represent points in the plane or in space as ordered pairs or triples of coordinates such as (x,y) or (x,y,z). Vectors are also described with an ordered n-tuple of coordinates by refer to a line segment whose tail is at the origin and which points to the coordinates given.

Definition 7.2.1. *Vectors* are geometric quantities with both a direction and a magnitude. They are often visually represented as a arrow whose initial point is a the origin of the coordinate system. The vector representing the line segment between the points A and B is denoted $\overrightarrow{AB}$.

Example 7.2.1. The line segment from (1,3) to (5,1) can be represented by the vector (4,-1). It is computed by forming the difference of the x coordinates and the difference of the y coordinates. Both the line segment and the vector (4,-1) have the same magnitude (length) and direction. The magnitude of the two dimensional vector $A = (x, y)$ is denoted $|A|$ and has the value $\sqrt{x^2 + y^2}$ while the magnitude of a three dimensional vector (x, y, z) has the value $\sqrt{x^2 + y^2 + z^2}$

Vectors can be added together to form a new vector.

Definition 7.2.2. The ***sum*** of two vectors is the vector whose coordinates is the sum of the corresponding coordinates. If $\boldsymbol{A} = (x_1, y_1)$ and $\boldsymbol{B} = (x_2, y_2)$ then $\boldsymbol{A} + \boldsymbol{B} = (x_1 + x_2, y_1 + y_2)$, while if $\boldsymbol{A} = (x_1, y_1, z_1)$ and $\boldsymbol{B} = (x_2, y_2, z_2)$ then $\boldsymbol{A} + \boldsymbol{B} = (x_1 + x_2, y_1 + y_2, z_1 + z_2)$

Just as in scaler addition, vector addition is both associative and communative. In other words the order in which the addition is performed does not matter. The addition of vectors ***A, B, C*** and ***D*** in various orders is pictured below.

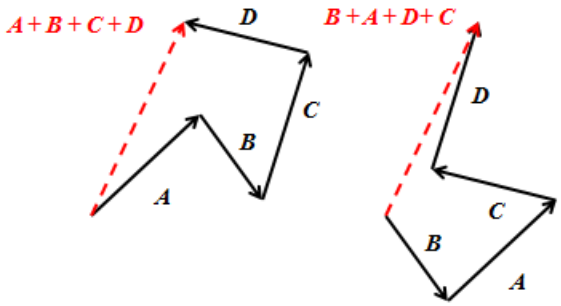


Figure 7.2: Addition of vectors ***A, B, C*** and ***D*** in various orders.

Another binary operator on vectors is the **dot product**. The dot product of two vectors $\boldsymbol{A}$ and $\boldsymbol{B}$ is a scaler (a single number) and the has the value of $|\boldsymbol{A}|\ |\boldsymbol{B}|\ \cos(\theta)$, where θ is the angle between the vectors.

Definition 7.2.3. The ***dot product*** between the vectors $\boldsymbol{A}$ and $\boldsymbol{B}$ is denoted $\boldsymbol{A} \bullet \boldsymbol{B}$ and has the value of the sum of the product of the corresponding coordinates.

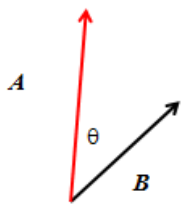


Figure 7.3: The angle θ formed between vectors $\boldsymbol{A}$ and $\boldsymbol{B}$.

We can therefore, calculate $\cos(\theta) = (\boldsymbol{A} \bullet \boldsymbol{B})/(|\boldsymbol{A}| \times |\boldsymbol{B}|)$. By using the arc-cosine function, we can then compute θ. It is useful to recall that $\cos(90) = 0$ and $\cos(0) = 1$. Thus a dot product of 0 indicates two perpendicular lines, and that the dot product is greatest when the lines are parallel. A final note about dot products is that they are not limited to two dimensional geometry. We can take dot products of vectors with any number of elements, and the above equality still holds.

The need frequently arises to compute the dot product of the line segments $\overrightarrow{\mathbf{AB}}$ and $\overrightarrow{\mathbf{AC}}$. The following C++ code efficiently does the job.

```
//Compute the dot product AB  BC
int dot(int[] A, int[] B, int[] C)
{ AB = new int[2];
  BC = new int[2];
  AB[0] = B[0]-A[0];
  AB[1] = B[1]-A[1];
  BC[0] = C[0]-B[0];
  BC[1] = C[1]-B[1];
  int dot = AB[0] * BC[0] + AB[1] * BC[1];
return dot;
}
```

Definition 7.2.4. The ***cross product*** of vector $\overrightarrow{\mathbf{A}}$ with $\overrightarrow{\mathbf{B}}$ is denoted $\overrightarrow{\mathbf{A}} \times \overrightarrow{\mathbf{B}}$, is the vector perpendicular to both $\overrightarrow{\mathbf{A}}$ and $\overrightarrow{\mathbf{B}}$. If $\overrightarrow{\mathbf{A}} = (x_1, y_1, z_1)$ and $\overrightarrow{\mathbf{B}} = (x_2, y_2, z_2)$ then $\overrightarrow{\mathbf{A}} \times \overrightarrow{\mathbf{B}} = (y_1 z_2 - z_1 y_2, z_1 x_2 - x_1 z_2, x_1 y_2 - y_1 x_2)$. The magnitude of the cross product is equal to the area of the parallelogram show in Figure 7.4 and is equal to $|\overrightarrow{\mathbf{A}} \times \overrightarrow{\mathbf{B}}| = |\overrightarrow{\mathbf{A}}||\overrightarrow{\mathbf{B}}| \sin(\theta)$. In two dimensions $|\overrightarrow{\mathbf{A}} \times \overrightarrow{\mathbf{B}}| = x_1 \times y_2 - y_1 \times x_2$.

The following code finds the cross product in two dimensions between the vectors $\overrightarrow{\mathbf{AB}}$ and the vectors $\overrightarrow{\mathbf{BC}}$.

```
//Compute the cross product AB  BC
int cross(int[] A, int[] B, int[] C)
```

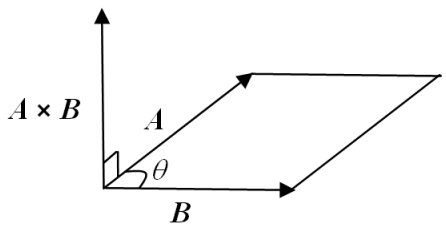


Figure 7.4: The cross product between vectors $\boldsymbol{A}$ and $\boldsymbol{B}$.

```
{ AB = new int[2];
  AC = new int[2];
  AB[0] = B[0]-A[0];
  AB[1] = B[1]-A[1];
  AC[0] = C[0]-A[0];
  AC[1] = C[1]-A[1];
  int cross = AB[0]*AC[1] - AB[1]*AC[0];
 return cross;
}
```

7.3 Lines and Line Segments in a Plane

The next section has a few important algorithms concerning lines in two and three dimensions.

7.3.1 Distance from a line to a point

Now that we have reviewed the basic tools of computational geometry, (vectors, magnitudes, dot products and cross products), we are ready to cover some basic algorithms. The first problem we will consider is computing the distance from a point to a line. This is can certainly be done with algebra, but the computation can lead to numerical instability such as division by very small numbers or lines with infinite or near infinite slope. The vector approach avoids all mention of slope and has only one division which can be adjusted to be far from zero. The method is based on computing the area of the parallelogram property of cross products.

Consider a line containing the points $\mathbf{A}$ and $\mathbf{B}$ and a point $\mathbf{C}$ not on the line containing $\mathbf{A}$ and $\mathbf{B}$ (pictured in Figure 7.5). Consider the area of the triangle formed by the vectors $\overrightarrow{\mathbf{AB}}$ and $\overrightarrow{\mathbf{AC}}$. The area of the parallelogram is given by the cross product and is equal to $|\overrightarrow{\mathbf{AB}}| \times h$, where h is the height of the parallelogram. However h is also the distance from $\mathbf{C}$ to the line containing the points $\mathbf{A}$ and $\mathbf{B}$. So computing the area of the parallelogram and dividing by $|\overrightarrow{\mathbf{AB}}|$ gives the h the distance from the point to the line.

The preceding discussion gives the following result :

Theorem 7.3.1. *The distance from a line containing the points* $\mathbf{A}$ *and* $\mathbf{B}$ *and a point* $\mathbf{C}$ *not on the line is given by :*

$$|\overrightarrow{\mathbf{AB}} \times \overrightarrow{\mathbf{AC}}|/|\overrightarrow{\mathbf{AB}}| \tag{7.1}$$

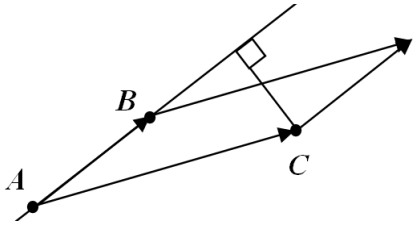


Figure 7.5: Finding the distance from the line containing **A** and **B** to **C**.

Note that the formula has only one division in it and that one is dividing by the magnitude of $\overrightarrow{\mathbf{AB}}$. This quantity is only small when **A** and **B** are near coincident and one can always choose other points to represent the line to avoid the numerical instability. Indeed given the two points **A** and **B**, then $\mathbf{C} = t \times (\mathbf{B} - \mathbf{A}) + \mathbf{A}$ is also a point on the line for all t.

Example 7.3.1. Write an expression using cross product and dot product for the distance between the line that passes through the two points (0,0), and (2,3) and the point (5,5). Here **A** = (0,0), **B** = (2,3) and **C** = (5,5)

First we compute the vectors :

$$\overrightarrow{\mathbf{AB}} = (2 - 0, 3 - 0) = (2, 3)$$

and

$$\overrightarrow{\mathbf{AC}} = (5 - 0, 5 - 0).$$

Now compute the cross product and the magnitude of $\overrightarrow{\mathbf{AB}}$ as

$$|\overrightarrow{\mathbf{AB}} \times \overrightarrow{\mathbf{AC}}| = |2 \times 5 - 5 \times 3| = |10 - 15| = |-5| = 5$$

$$|\overrightarrow{\mathbf{AB}}| = \sqrt{2^2 + 3^2} = \sqrt{13}$$

So the distance is $5/\sqrt{13}$. ■

7.3.2 Intersection between two Lines in a plane

We now consider the intersection between a two lines in a plane. Ideally the lines will be in the form $Ax + By = C$, but in practice that is rarely the case. However given two points on the line, it is easy to find the appropriate A, B and C. If the two points are (x_1, y_1) and (x_2, y_2) then :

$$A = y_2 - y_1, B = x_1 - x_2, C = A(x_1) + B(y_1)$$

.

So after conversion, consider two lines, given by the equations:

$$\mathbf{A_1}x + \mathbf{B_1}y = \mathbf{C_1}$$

and

$$\mathbf{A_2}x + \mathbf{B_2}y = \mathbf{C_2}$$

To find the point (x, y) at which the two lines intersect, we simply need to solve the two equations for the two unknowns, x and y. These equations can be solved most easily by the computer using Cramer's rule [1]. The solution is expressed in terms of the determinants which is also the cross product of appropriate vectors. Indeed the correctness of Cramer's rule can be given by using the geometric interpretation of cross products.

Theorem 7.3.2. *The intersection of the lines* $\mathbf{A_1}x + \mathbf{B_1}y = \mathbf{C_1}$ *and* $\mathbf{A_2}x + \mathbf{B_2}y = \mathbf{C_2}$ *is given* $((\mathbf{C} \times \mathbf{B})/(\mathbf{A} \times \mathbf{B}), (\mathbf{A} \times \mathbf{C})/(\mathbf{A} \times \mathbf{B}))$.

A C++ implementation of the theorem for computing the intersection is given below.

```
// code to determine the intersection of
// A1x + B1y = C1 with A2x + B2y = C2
//
double det = A1*B2 - A2*B1 ; //A x B
if(det == 0)
  { //Lines are parallel }
else
  {
  double x = (B2*C1 - B1*C2)/det;//CxB/det
  double y = (A1*C2 - A2*C1)/det;//AxC/det
  }
```

Example 7.3.2. Find the intersection of $x + 2y = 3$, with $3x + y = 2$.
Here we find that : $A = (1, 3), B = (2, 1)$ and $C = (3, 2)$
Computing the determinates results in :
$\mathbf{A} \times \mathbf{B}$ = (1)(1) -2(3) = -5
$\mathbf{A} \times \mathbf{C}$ =1(2) (3)(3) = -7
$\mathbf{C} \times \mathbf{B}$ = - (2(2) 1(3)) = -1
Hence the point of intersection is (1/5, 7/5) . ■

Example 7.3.3. Find the intersection of $x + 3y = 4$, with $2x + y = 2$.
Here we find that : $A = (1, 2), B = (3, 1)$ and $C = (4, 2)$
Computing the determinates results in :
$\mathbf{A} \times \mathbf{B}$ = (1)(1) - 3(2) = -5
$\mathbf{C} \times \mathbf{B}$ =4(1) (2)(4) = -2
$\mathbf{A} \times \mathbf{C}$=1(2) (2)(4) = -6
Hence the point of intersection is (2/5, 6/5) . ■

7.3.3 Intersection of Line Segments

We conclude this section with an intersection problem that is the easy for people to solve visually, yet the most difficult of the ones we have considered so far : Determining if two line segments intersect. This problem arises often in navigation and in computer games where one object hitting another object is important. The chapter will present two different algorithms for this test, one now and one after the angle discusssion.

Line segments are completely determined by their end points. Figure 7.6 illustrates three pairs of line segments. One line segment is determined by $\{p_1, p_2\}$ and the other is determined by $\{q_1, q_2\}$/

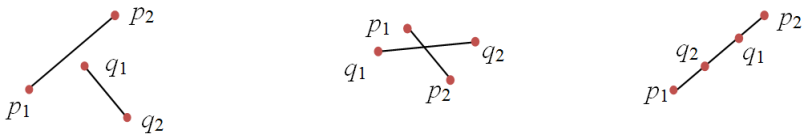


Figure 7.6: Three examples of two line segments $\{p_1, p_2\}$ and $\{q_1, q_2\}$

In the following discussion we will denote the x and y coordinates of each point p as $p.x$ and $p.y$ respectively. Numerically a person might attempt to solve this considering the lines which contain each line segment and attempt to compute the point of intersection and then see if that point lies on one of the line segments. This method is perfectly valid but has difficulties if one of the line segments is vertical (has an infinite slope) or if the line segments are collinear.

A simpler method that avoids the aforementioned problems is to first find bounding boxes for each line segment, check if the boxes overlap and then check if the line segments straddle each other. If the bounding box test fails, then we can conclude that the two line segments are disjoint, so we only perform the straddle test if the data passes the bounding box test.

Definition 7.3.1. The **bounding box** of a line segment is the smallest rectangle that surrounds the segment and has sides that are parallel to the x-axis and y-axis.

The bounding box rectangle can be defined by the coordinates of either pair of non–adjacent verticies which are on the ends of a diagonal. One such pair has for its lower left point the minimum x-coordinate and the minimum y-coordinate, while the upper right point has the maximum x-coordinate and the maximum y-coordinate. For a line segment with endpoints p_1 and p_2 where $p_1 = (p_1.x, p_2.y)$ and $p_2 = (p_2.x, p_2.y)$, then the bottom left point is $(\min(p_1.x, p_2.x), \min(p_1.y, p_2.y))$ and whose upper right point is $(\max(p_1.x, p_2.x), \max(p_1.y, p_2.y))$.

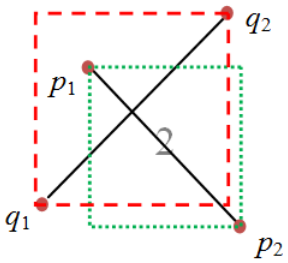


Figure 7.7: The bounding boxes of two line segments.

Due to the rectangular shape of a bounding box, it is easy to determine if they are overlapping. When the boxes are disjoint then either one box is the left of the other (the maximum x coordinate of one is $<$ the minimum x coordinate of the other) or one box is on top of the other (the maximum

y coordinate of one is $<$ the minimum y coordinate of the other). This is resolved by the following test for overlapping bounding boxes :

$\max(p1.x, p2.x) \leq \min(q1.x, q2.x)$ && //max x coord of seg1 $\leq$ min x coord of seg2
$\max(q1.x, q2.x) \leq \min(p1.x, p2.x)$ && //max x coord of seg2 $\leq$ min x coord of seg1
$\max(p1.y, p2.y) \leq \min(q1.y, q2.y)$ && //max y coord of seg1 $\leq$ min y coord of seg2
$\max(q1.y, q2.y) \leq \min(p1.y, p2.y)$ //max y coord of seg2 $\leq$ min y coord of seg1

Unfortunately overlapping bounding boxes are insufficient to determine if two line segments intersect as shown in the example below.

Figure 7.8: Overlapping Bounding Boxes without Intersection.

If the bounding boxes of the line segments intersect, we proceed with the straddle test. The straddle test determines if the endpoints of the q line segment are clockwise or counter clockwise to one of the endpoints of the p line segment (p_1) with respect to the other end point of the same segment (p_2). If one point of q is clockwise and other point of q is counter– clockwise then the q segment straddles the p segment. Specifically we compute the difference in slopes between the line segments $\overline{p_1q_1}$ and $\overline{p_1p_2}$ and compare it to the difference in slopes between$\overline{p_1q_2}$ and $\overline{p_1p_2}$. When comparing slopes, we can avoid the problem of possibly dividing by zero by multiplying both slopes by the difference in the y values. This yields:

$$z_1 = (q_1.x - p_1.x)(p_2.y - p_1.y) - (q_1.y - p_1.y)(p_2.x - p_1.x)$$

$$z_2 = (q_2.x - p_1.x)(p_2.y - p_1.y) - (q_2.y - p_1.y)(p_2.x - p_1.x)$$

If the signs of z_1 andz_2 are different, or if either is zero, the line segments straddle each other. When one of the zs is zero, one of the points of the second line segment is contained on the line containing the second line segment. Since if we perform this test, we have already shown that the bounding boxes intersect, the line segments intersect as well.

```
struct Point
{
   double x;
   double y;
};
```

```
int lint(Point p1, Point p2, Point q1, Point q2) {
double             z1,
                   z2;
/*****************************************************************
*  Overlapping Box test.
*****************************************************************/
if (!(max(p1.x, p2.x) >= min(q1.x, q2.x) &&
      max(q1.x, q2.x) >= min(p1.x, p2.x) &&
      max(p1.y, p2.y) >= min(q1.y, q2.y) &&
      max(q1.y, q2.y) >= min(p1.y, p2.y))) {
   return 0;
}
/*****************************************************************
*  Straddle Test
*****************************************************************/
z1 = (q1.x - p1.x)*(p2.y - p1.y)) - ((q1.y - p1.y)*(p2.x - p1.x)) ;
z2 = (q2.x - p1.x)*(p2.y - p1.y)) - ((q2.y - p1.y)*(p2.x - p1.x)) ;

return (z1*z2 <= 0);
}
```

The code was much easier to write than the analysis and best of all it runs in O(1) time. The code does ***NOT*** consider the degenerate cases where one endpoint lies on the ***other*** line segment. These cases must be dealt with by other code.

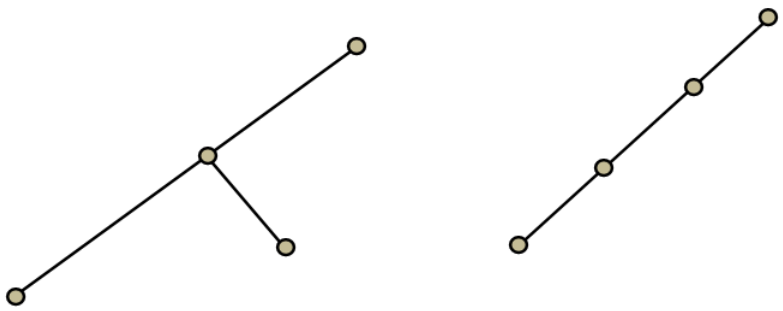


Figure 7.9: Degenerate Line Segments

7.4 Polygons

Definition 7.4.1. A **polygon** is any 2 dimensional closed figure whose boundary is composed of non-intersecting line segments. Polygon literally means many sided. A **convex polygon** is a

polygon with the property that any line segment connecting 2 interior points, lies entirely inside the polygon. A **simple polygon** is a closed polygonal chain of line segments that do not cross each other.

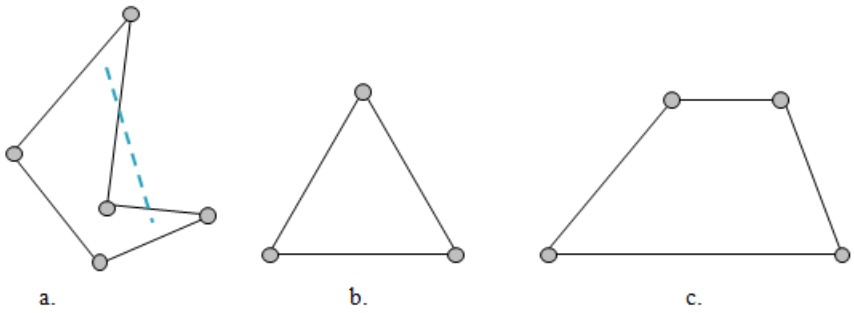


Figure 7.10: Three Polygons where polygons b. and c. are convex

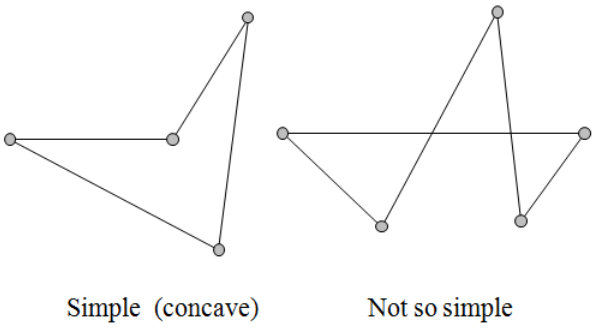


Figure 7.11: A simple and non-simple polygon

There are formulas that students memorize for the area of simple polygons. These include triangles, rectangles, trapezoids, parallelograms, and regular figures (squares, hexagons, octagons, etc.) However there exists a simple formula for the area of ***ANY*** polygon, which is easy to implement.

Theorem 7.4.1. *Given a polygon with n vertices (x_i, y_i), $i = 0, \ldots, n-1$ arranged counterclockwise with $(x_0, y_0) = (x_n, y_n)$. (see Figure 7.12) Then the area is given by the Surveyor's Formula*

$$Area = \frac{1}{2}\sum_{i=0}^{n-1}(x_i, y_i)\times(x_{i+1}, y_{i+1})$$

Proof idea

The formula is based on the fact that the cross product of adjacent vectors is the area of a parallelogram whose sides are the adjacent vectors. The Surveyor's Formula computes the area of

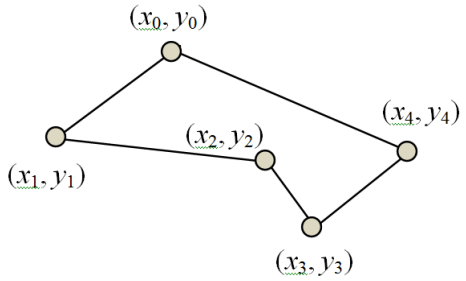


Figure 7.12: A General Polygon

triangles (half the parallelograms) whose total area is the area of the polygon. Some of the areas are negative which cancel out overlapping areas and areas that fall outside the polygon. ■

To use the formula is often essential to put the verticies in counter–clockwise order. Such is not the case with our first example concerning the triangle. The points are always either clockwise or counter–clockwise so as long as we take the absolute value of the answer the order does not matter.

Example 7.4.1. Write a mathematical expression for the area of polygon whose verticies are at (1,1), (1, 5) and (5, 3). Computing the cross products in the order the points were given yields :

$$\frac{1}{2}(1(5) - 1(1) + 1(3) - 5(5) + 5(1) - 3(1)) = -8$$

Since the area must be a positive quantity, the area of the triangle is 8. ■

In the next example we will first have to order the verticies. Note that in most computer applications this problem does not arise, as the points are given in either clockwise or counter-clockwise order.

Example 7.4.2. Write an expression use the Surveyors Formula that computes the area of the polygon bounded by (2,6), (1,1) , (10, 10), (6, 3). We first plot the points to order the verticies.

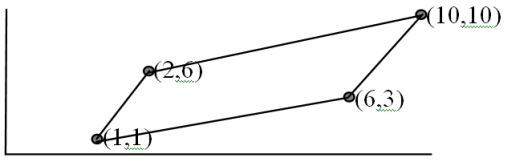


Figure 7.13: Polygon for Example 7.4.2

So we can order the vertices (1,1), (6,3), (10,10) and (2,6) and the area is

$$\tfrac{1}{2}\left[1(3) - 1(6) + 6(10) - 3(10) + 10(6) - 10(2) + 2(1) - 6(1)\right] = 31\tfrac{1}{2}$$

■

7.4.1 Determining if a Point is Interior to a Polygon

A problem often encountered in computer graphics is to determine if a given point (x,y) is interior or exterior to a given polygon.

Given a point p and consider a ray starting a p and directed parallel to the x axis. (Of course the direction of the ray is arbitrary, and the same result holds for any direction.) It is easy to see that if the ray intersects the line segments making up the polygon an even number of times then the point is outside the polygon. While if the number of intersections is odd then the point lies inside the polygon.

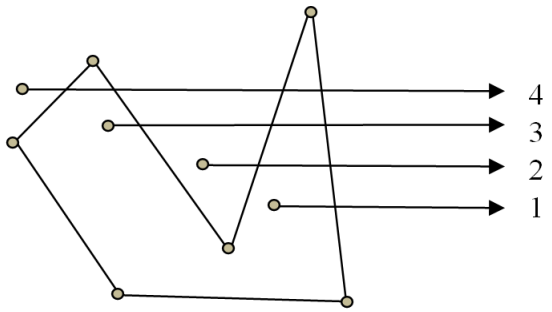


Figure 7.14: A polygon intersected by rays from several points.

```
/*
   pnpoly returns the number of intersections of a ray
   starting at (x,y) with the sides of the polygon defined
   by the points (xp[i],yp[i]), i = 0,1,2, ..., npol-1
*/
int pnpoly(int npol, float *xp, float *yp, float x, float y)
{
  int i, j, c = 0;
  for (i = 0, j = npol-1; i < npol; j = i++)
  { if ((((yp[i] <= y) && (y < yp[j])) || ((yp[j] <= y) &&
      (y < yp[i]))) && (x < (xp[j] - xp[i]) * (y - yp[i]) / (yp[j] - yp[i]) + xp[i]))
    c = !c; }
  return c;
}
```

For the algorithm above there is a pathological case if the point being queries lies exactly on a vertex or the ray being exactly along an edge. The easiest way to cope with this is to test that as a separate process and make your own decision as to whether you want to consider them inside or outside.

7.5 Angles

In this section we briefly discuss angles in the plane. The word *angle* comes from the Latin word *angulus* which means *corner*.

Definition 7.5.1. An **angle** is formed by two line segments which share a common end point. This common endpoint is called the **vertex** of the angle. The magnitude of the angle is the "amount of rotation" needed to make the line segments collinear.

We already can find the measure of any angle by using the definition of cross or dot product. However what is not so obvious is given three ordered points (the end points of the line segments that form the angle) is the angle clockwise or counter–clockwise in the given order.

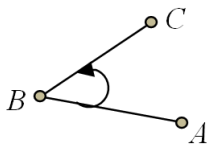


Figure 7.15: Angle with vertex B.

Consider an angle defined by the three points $A = (a.x, a.y)$, $B = (b.x, b.y)$ and $C = (c.x, c.y)$ with vertex at B. The three points are in counterclockwise order if and only if the slope of the line $\overrightarrow{BA}$ is less than the slope of the line $\overrightarrow{BC}$.

$$counterclockwise \Leftrightarrow \frac{b.y - a.y}{b.x - a.x} < \frac{c.y - b.y}{c.x - b.x}$$

To avoid any problems with divisions by numbers near zero, the condition for counterclockwise can be written as

$$counterclockwise \Leftrightarrow (b.y - a.y)(c.x - b.x) < (b.y - a.y)(b.x - a.x)$$

Below is an implementation called ccw() which tests if a sequnce of three points, is clockwise (returns -1) counter-clockwise (returns 1) or collinear (returns 0).

```
struct point
{
  double x;
  double y;
};

int ccw(point p, point q, point r)
/*
   determines if the sequence p, q, r is counter-clockwise
*/
{
  int result = (r.x - q.x)*(p.y - q.y) - (r.y - q.y)*(p.x - q.x);
```

```
  if (result < 0) return -1; //clockwise
  if (result > 0) return  1; //counter-clockwise
  return 0;                  // p, q, r are collinear
}
```

7.5.1 Another Line Segment Intersection Algorithm

The ccw() function is very useful in computational geometry. Our first application of the function will be an alternative method of determining if two line segments intersect.

Theorem 7.5.1. *Two line segments $\overline{p_1p_2}$ and $\overline{q_1q_2}$ intersect if and only if*

1) p_1 and p_2 are on opposite sides of the line $\overleftrightarrow{q_1q_2}$, and

2) q_1 and q_2 are on opposite sides of the line $\overleftrightarrow{p_1p_2}$

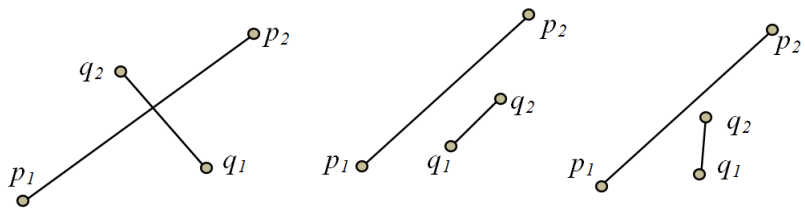


Figure 7.16: Three pairs of line segments.

Note that the first condition is true only if the angle formed by the sequence of points (p_1, q_1, q_2) is clockwise and the angle formed by (p_2, q_1, q_2) is counter-clockwise (or vice versa). A similar condition is necessary for the second condition of the theorem to hold.

This leads to the following code.

```
//    Determines if the line segment (p1,p2) intersects the line segment (q1, q2)

bool intersect(point p1, point p2, point q1, point q2)
{
  if (ccw(p1, q1, q2) == ccw(p2, q1, q2)
     return false;
  if (ccw(q1, p1, p2) == ccw(q2, p1, p2)
     return false;
  else
    return true;
}
```

7.6 Convexity

We end the chapter with a brief discussion of convex hulls.

Definition 7.6.1. The **convex hull** of a set of points in the plane is the smallest polygon containing all of the points. The convex hull can also be thought of as the shape a rubber band would make if it was stretched around all the points and then released.

Figure 7.17: Forming a convex hull of a set of points.

The convex hull for a set of points is unique. For a small number of points, the convex hull is easy to find. For example if the number of points totals 1 or 2, the convex hull is the polynomial containing all the points. If there are 3 points which are not collinear, then again the hull contains all the points. However for four or more points algorithms are needed to find the hull efficiently.

There are several algorithms for finding the convex hull including: Jarvis's march (gift–wrapping algorithm), divide–and–conquer, Graham's scan, and Chan's algorithm (shattering). The convex hull algorithms resemble sorting algorithms. Jarvis's march is the simplest, resembling selection sort, but also the slowest and requires $O(n^2)$ time. Divide–and–conquer resembles quicksort and works in $O(n \lg n)$ time on average, but is only $O(n^2)$ in the worst case. Graham's scan is the algorithm that we will present here and requires only $O(n \lg n)$ worst case time. Chan's algorithm is never slower than either Jarvis's march or Graham's scan and is a combination of of divide–and–conquer and gift–wrapping.

Graham's scan computes the convex hull in three phases. In phase 1, it locates the lowest point (sometimes called the anchor point). In phase 2, it sorts the points in a counter–clockwise fashion relative to the anchor point. In phase 3 each of the points is examined and points not on the convex hull are discarded. The three phases are pictured below in Figure 7.18.

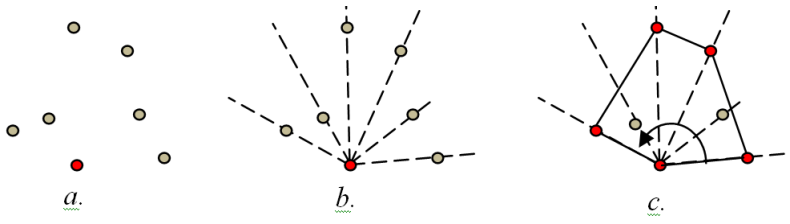


Figure 7.18: a. Find the lowest point, b. Order the points by angle, c. Compute the Hull.

Phase 1 is easily accomplished by finding the point with the lowest y coordinate. Phase 2 is also uses any $O(n \lg n)$ sort but instead of computing the slopes and comparing them, we instead

make use of the ccw() function. If the lowest point is denoted l, then to compare points p and p we need only compute ccw(l, p, q). If ccw$(l, p, q) > 0$, then we swap the order of p and q.

We now have the points ordered as $a_0 = l, a_1, a_2, ...a_{n-1}, a_n = l$ in a counter–clockwise order. For phase 3, we will maintain a stack S of candidate hull points. The points a_0 and a_1 are always part of the hull and are pushed on to the stack. Each subsequent point a_i is compared to the top two points on the stack using ccw(). If the value is positive (indicating a concave turn), the point is discarded otherwise is pushed on top of the stack. This is a $O(n)$ time process. A C++ implementation, adapted from one in [4] is given below:

```
struct point
{
  double x;
  double y;
};

point lowPoint; // global point low

int ccw(const point  p, const point & q, const point & r)
{
  int result;
  result = (int)((r.x - q.x)*(p.y - q.y) - (r.y - q.y)*(p.x - q.x));
  cout<<"result is "<<result<<endl;
  if (result < 0) return -1; //clockwise
  if (result > 0) return  1; //counter-clockwise
  return 0;                  // p, q, r are collinear
}

bool angleCompare(const point & a, const point & b)
{
     return (ccw(lowPoint, a, b) > 0) ;
}

void pointPrint(point p)
{
  cout<<"("<<p.x<<","<<p.y<<") ";
}

vector<point> convexHull(vector<point> points)
/*
  Assumes at least 3 points
*/
{
   //Phase 1 : Find the point with the lowest y value.
```

```
 //If tie use the rightmost point
int  i, low = 0, N = points.size();
for (i = 1; i < N; i++)
  if (points[i].y  < points[low].y ||
     (points[i].y == points[low].y &&
      points[i].x > points[low].x))
   low = i;

point temp = points[0];//swap selected vertex with points[0]
points[0] = points[low];
points[low] = temp;
lowPoint = points[0];

//Phase 2 : sort the points with respect to points[low]
sort(++points.begin(), points.end(), angleCompare);

//Phase 3 : compute the stack
stack <point> S;
point prev, curr;
S.push(points[N-1]);
S.push(points[0]);
i = 1;
while (i < N && !S.empty())
  {
   curr = S.top();
   S.pop();
   prev = S.top();
   S.push(curr);
   if (ccw(prev, curr, points[i]) >0)
     {
      S.push(points[i]);
      i++;
     }
   else
     S.pop();
  }
vector<point> hull;
while (!S.empty())
  {
   hull.push_back(S.top());
   S.pop();
  }
hull.pop_back();
return hull;
}
```

The above code will work with any set of unique points which contain 3 or more points.

Computational geometry contains a wealth of interesting problems with some power algorithms. This chapter has only only scratched the surface of the many problems it considers. Most notably are the algorithms associated with computer graphics which make computer aided design and manufacturing fast and accurate, as well as animations lifelike. Students interested in learning more about Computational geometry are invited to read the several texts devoted to the subject found in the bibliography.

7.7 Exercises

1) Find the dot product of (1,2) and (3,4).

2) Find the magnitude of the cross product of (1,2) and (3,4).

3) Use the formula given in the text to find distance from the point (4,1) to the line $y = 2x + 4$

4) Use the formula given in the text to find distance from the point (3,4) to the line $y = x + 2$

5) Use the formula from the text to find intersection of the lines $y = 2x + 5$, and $2y + x = 5$

6) Use the formula from the text to find intersection of the lines $y = 2x + 1$, and $2y = x + 4$

7) Use the formula from the text to find intersection of the lines $y = 3x - 1$, and $2y = 2x + 6$.

8) Write an mathematical expression that uses the Surveyor's Formula to compute the area of the polygon bounded by (7,10), (2,7), (5, 15), (4, 2).

9) Write an expression to use the Surveyors Formula to find the area of the polygon bounded by (0,10), (10,10) , (0, 0), (10, 0).

10) Write an expression use the Surveyor's Formula that computes the area of the polygon bounded by (0,5), (10,10) , (0, 0), (4, 0).

11) Sketch the following polygon and indicate the convex hull given from the five points $\{a_1, a_2, a_3, a_4, a_5, a6\}$ where a_1 has the minimum y value and the other points are ordered in a counterclockwise manner to a_1. ccw(a_1, a_2, a_3) >0 ccw$(a_2, a_3, a_4) = 0$, ccw(a_3, a_4, a_5)<0, ccw(a_4, a_5, a_6)>0.

12) Sketch the following polygon and indicate the convex hull given from the five points $\{a_1, a_2, a_3, a_4, a_5, a_6\}$ where a_1 has the minimum y value and the other points are ordered in a counterclockwise manner to a_1. ccw(a_1, a_2, a_3) >0 ccw$(a_2, a_3, a_4) > 0$, ccw(a_3, a_4, a_5)=0, ccw(a_4, a_5, a_6)<0.

13) Sketch the following polygon and indicate the convex hull given from the five points $\{a_1, a_2, a_3, a_4, a_5, a_6\}$ where a1 has the minimum y value and the other points are ordered in a counterclockwise manner to a_1. ccw$(a_1, a2, a3)$ >0 ccw(a_2, a_3, a_4) <0, ccw(a_3, a_4, a_5)>0, ccw(a_4, a_5, a_6)=0

7.7.1 Solutions to Exercises

1) $1(2) + 3(4) = 14$

2) $1(4) - 3(2) = -2$

3) Let the point (4, 1) be C
In the line $y = 2x + 4$, if $x = 0$, then $y = 4$. This point, (0, 4) will be A.
When $x = 1$, $y = 6$. This point, (1, 6) will be B.
$\overrightarrow{AB} = (1 - 0, 6 - 4) = (1, 2)$
$\overrightarrow{AC} = (4 - 0, 1 - 4) = (4, -3)$
$|\overrightarrow{AB} \times \overrightarrow{AC}| = |1(-3) - 2(4)| = |-11| = 11$
$||\overrightarrow{AB}|| = \sqrt{1^2 + 2^2} = \sqrt{5}$
$\frac{|\overrightarrow{AB} \times \overrightarrow{AC}|}{||\overrightarrow{AB}||} = \frac{11}{\sqrt{5}}$

4) Let the point (3, 4) be C
In the line $y = x + 2$, if $x = 0$, then $y = 2$. This point, (0, 2) will be A.
When $x = 1$, $y = 3$. This point, (1, 3) will be B.
$\overrightarrow{AB} = (1 - 0, 3 - 2) = (1, 1)$
$\overrightarrow{AC} = (3 - 0, 4 - 2) = (3, 2)$
$|\overrightarrow{AB} \times \overrightarrow{AC}| = |1(2) - 1(3)| = |-1| = 1$
$||\overrightarrow{AB}|| = \sqrt{1^2 + 1^2} = \sqrt{2}$
$\frac{|\overrightarrow{AB} \times \overrightarrow{AC}|}{||\overrightarrow{AB}||} = \frac{1}{\sqrt{2}}$ or $\frac{\sqrt{2}}{2}$

5) $y = 2x + 5 \Rightarrow 2x - y = -5$
$A = (2, 1)$, $B = (-1, 2)$, $C = (-5, 5)$
$A \times B = 2(2) - 1(-1) = 5$
$A \times C = 2(5) - 1(-5) = 15$
$C \times B = -5(2) - 5(-1) = -5$
$\frac{C \times B}{A \times B} = \frac{-5}{5} = -1$ and $\frac{A \times C}{A \times B} = \frac{15}{5} = 3$
Point of Intersection is: $(-1, 3)$

6) $y = 2x + 1 \Rightarrow 2x - y = -1$
$2y = x + 4 \Rightarrow x - 2y = -4$
$A = (2, 1)$, $B = (-1, -2)$, $C = (-1, -4)$
$A \times B = 2(-2) - 1(-1) = -3$
$A \times C = 2(-4) - 1(-1) = -7$
$C \times B = -1(-2) - (-4)(-1) = -2$
$\frac{C \times B}{A \times B} = \frac{-2}{-3} = \frac{2}{3}$ and $\frac{A \times C}{A \times B} = \frac{-7}{-3} = \frac{7}{3}$
Point of Intersection is: $(\frac{2}{3}, \frac{7}{3})$

7) $y = 3x - 1 \Rightarrow 3x - y = 1$
$2y = 2x + 6 \Rightarrow 2x - 2y = -6$
$A = (3, 2)$, $B = (-1, -2)$, $C = (1, -6)$
$A \times B = 3(-2) - 2(1) = -4$

$A \times C = 3(-6) - 2(1) = -20$
$C \times B = 1(-2) - (-6)(-1) = -8$
$\frac{C \times B}{A \times B} = \frac{-8}{-4} = 2$ and $\frac{A \times C}{A \times B} = \frac{-20}{-4} = 5$
Point of Intersection is: $(2, 5)$

8) Points in Order: (2, 7), (5, 15), (7, 10), (4, 2).
$= \frac{1}{2}[(2(15)-7(5))+(5(10)-15(7))+(7(2)-10(4))+(4(7)-2(2))] = \frac{1}{2}[-5-55-26+24] = -31$
Area must be positive so it is 31.

9) Points in Order: (0, 0), (0, 10), (10, 10), (10, 0).
$= \frac{1}{2}[(0) + (0) - 10(10)) + (0 - 10(10)) + (0)] = \frac{1}{2}[0 - 100 - 100 + 0] = -100$ Area must be positive so it is 100.
The points form a 10×10 square so the answer can easily be checked. $A = b \times h$.

10) Points in Order: (0, 0), (0, 5), (10, 10), (4, 0).
$= \frac{1}{2}[(0)+(0-5(10))+(0-10(4))+(0-0)] = \frac{1}{2}[0-50-40+0] = -45$ Area must be positive so it is 45.

11) See Figure 7.19

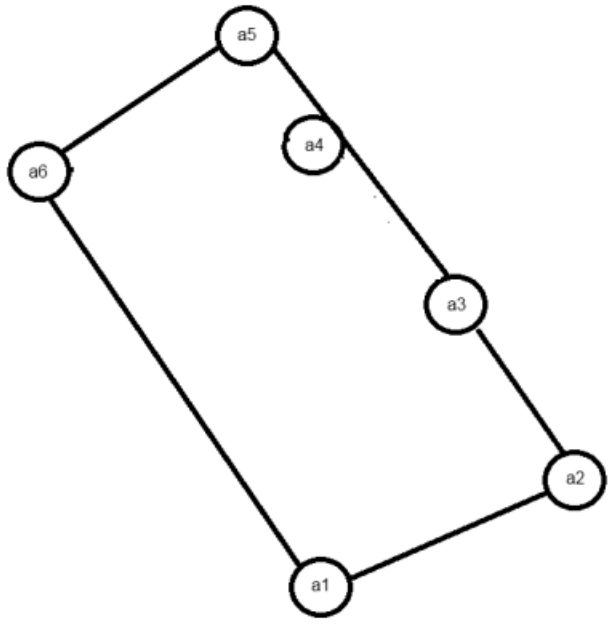


Figure 7.19: Convex Hull Diagram for Exercise 11

12) See Figure 7.20

13) See Figure 7.21

7.7.2 Computer Exercises

UVa indicates that the problem comes from the Universidad de Valladolid on line problem archive. The problems classifications came from [4]. A link to the UVa is given below:

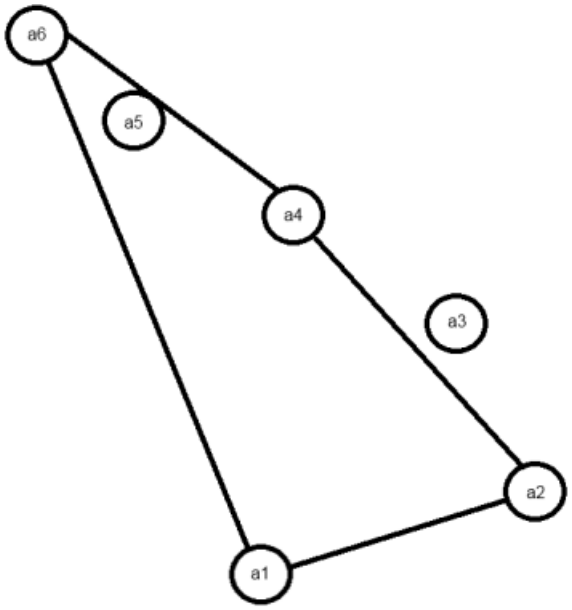


Figure 7.20: Convex Hull Diagram for Exercise 12

`http://uva.onlinejudge.org/index.php`

1) (Uva 190) Circle through 3 points
2) (Uva 10180) Rope Crisis In Ropeland (closest point on a line segment)
3) (Uva 11068) An Easy Task ($ax + by = c$)
4) (Uva 866) Intersecting Line segments
5) (Uva 378) Intersecting Lines
6) (UVa 478) Determine if a point is inside multiple polygons.
7) (UVa 10078) Art Gallery (testing if a polygon is convex)
8) (UVa 10112) Myacm Triangles
9) (UVa 11447) Reservoir Logs
10) (Uva 109) Scud Busters
11) (Uva 218) Moth Eradication

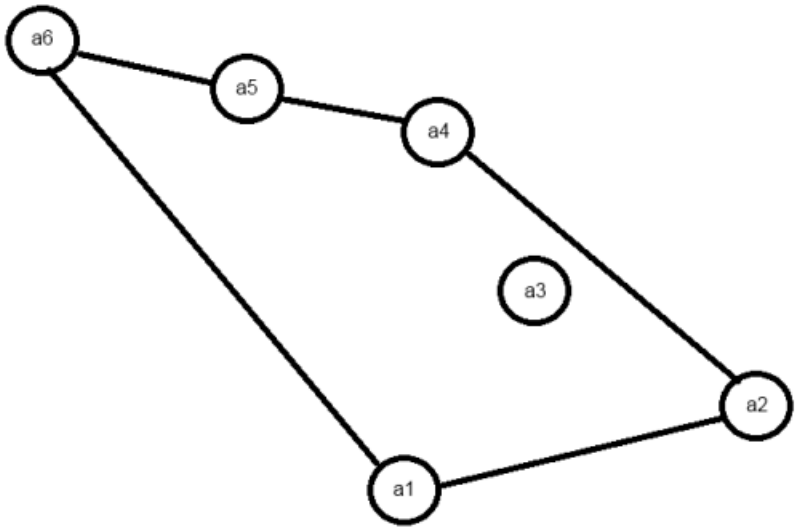


Figure 7.21: Convex Hull Diagram for Exercise 13

Bibliography

[1] Cramer Gabriel , "Introduction a la l'analyse des lignes courbes algbriques (Introduction to the analysis of algebraic curves), (1750).

[2] de Berg, M., van Kreveld, M., Overmars, M., and Schwarzkoph, O., *Computational Geometry: Algorithms and Applications* ,Springer, 2nd edition, 2000 .

[3] Erickson, J. *Algorithms*, http://www.uiuc.edu/ jeffe/teaching/algorithms/, 2009 .

[4] Halim, S., Halim F.,*Competitive Programming : Increasing the Lower Bound of Programming Contests*, Hulu, 2010.

[5] O'Rourke, J., ,*Computational Geometry in C*, Cambridge University Press, 2nd edition, 1998.

[6] Skiena, S., Revilla, M., *Programming Challenges*, Springer, 2003.

Chapter 8

Quantum Computing

Quantum computation is... a distinctively new way of harnessing nature... It will be the first technology that allows useful tasks to be performed in collaboration between parallel universes. David Deutsch

8.1 Introduction

Quantum computers are computers that take advantage of quantum mechanical effects (such as superposition and entanglement) in order to speed up computations. Quantum computing is still in it's infancy and as of this writing, physical quantum computers exist only in research laboratories. However algorithms for quantum computers have already been written and these algorithms can work exponentially faster than their classical computer counterparts. So although you cannot buy a quantum computer for your home, they have the potential to change the world in the upcoming decades.

One quantum algorithm is Shor's algorithm [1] which can factor numbers so fast that RSA encryption will no longer be secure. In this chapter we will briefly discuss a few basics of quantum computing. This discussion is not intended to equip the reader for writing their own quantum computer programs, however, it should give the reader a sense of how quantum programs differs from classical computing.

8.2 Qubits

We will start the discussion with a consideration of the memory of a quantum computer. The basic memory unit of a classical computer is the bit. Recall a bit can be in either one of two states, usually referred to as 0 or 1. This is usually implemented with an electronic circuit which will remain in one of two voltage states until the circuit loses power. A set of bits can be used to store any integer (in base 2 of course) or a finite set of floating point numbers, letters, or symbols. Anything that only takes on a finite set of values can be stored with a finite number of bits.

Now consider the basic unit of memory in a quantum computer, a **qubit** (or quantum bit). A qubit can be in state 0, state 1 or a **superposition** of states 1 and 0. By this we mean that the state can be in a linear combination of the 0 and 1 states. Specifically, if we adopt the Dirac

notation used by physicists, then we denote the states as $|0>$ and $|1>$ and a superposition of the states is $\alpha_0|0> + \alpha_1|1>$ where we restrict the coefficients to satisfy $|\alpha_0|^2 + |\alpha_1|^2 = 1$ for any complex numbers a_i. Note that the coefficients are not probabilities. The coefficients can be negative or even imaginary numbers. A qubit represents both the state and the entanglement of the states.

There are multiple ways proposed to implement a qubits including super conductors, quantum dots, nuclear magnetic resonance, and optical lattices. The most practical method for implementing qubits has not yet be determined so there is a variety of research using the aforementioned methods. Consider an electron orbiting a nucleus where there are two stable orbits. However the laws of quantum mechanics imply that the position of the electron is given a by a complex function. Hence the electron can be used to store a single qubit. (see Figure 8.1)

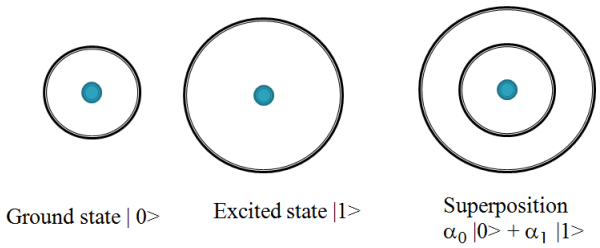


Figure 8.1: Illustration of a qubit.

Example 8.2.1. For example the state of the electron could be in the state :

$$1/\sqrt{2}|0> + 1/\sqrt{2}|1>$$

Alternatively the electron could be in the state :

$$1/\sqrt{5}|0> - 2i/\sqrt{5}|1>$$

■

It should be noted that the α_i's are *not* probabilities. They can be negative or even imaginary numbers. However quantum mechanics also implies that when a measurement of the position of the electron is made, the electron is forced to a single state either 0 or 1. (see Figure 8.2) The probability of the qubit being in state α_i s after a measurement, is given by $|\alpha_i|^2$.

This does not hold for just two state systems. In reality the electron can be many different states. If the electron could be found in k states, then the electron would represent a k state system with the states $|0>, |1>, ..., |k>$. The equation describing the electrons position would be

$$\alpha_0|0> + \alpha_1|1> + ... + \alpha_k - 1|k-1> \text{ , where}$$
$$\sum_{j=0}^{k-1} |\alpha_j|^2 = 1$$

Measurement of this system forces it into one of the k states.

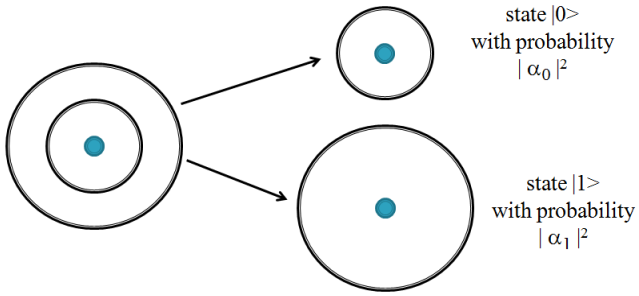


Figure 8.2: Measurement forces the qubit into a single state.

8.2.1 Encoding with qubits

Classically we use the states of the electron to encode information. It would 2 states to represent a bit, one state for 0 and one state for 1. We would need one state for each value to store. Thus it would require 2^n states to store n a value represented by n bits. A better solution is to use n qubits. If there are just two qubits, there are are four possible states namely 00, 01, 10, and 11. This is described by the $|\alpha>=|\alpha_{00}|00>+|\alpha_{01}|01>+|\alpha_{10}|10>+|\alpha_{11}|11>$, where $\sum_{j=0,k=0}^{1,1}|\alpha_{jk}|^2=1$.

Again, measuring this system will force it in of the states 00, 01, 10 or 11 with probability $|\alpha_{ij}|^2$. Now it is possible to measure a single qubit. If that happens then the state of the system will change in one that is consistent with the measurement. For example if we measure the first qubit and find the it is in state 0 then the new state is given by :

$$|\alpha_{new}>=\frac{\alpha_{00}|00>}{\sqrt{|\alpha_{00}|^2+|\alpha_{01}|^2}}+\frac{\alpha_{01}|01>}{\sqrt{|\alpha_{00}|^2+|\alpha_{01}|^2}}$$

Example 8.2.2. If the state of a system is given by : $|\psi>=\frac{|00>}{\sqrt{2}}-\frac{|11>}{\sqrt{2}}$ what would be the state after measuring the first qubit and determining that it is 0 ?

Clearly the state would be forced into $|\psi>=|00>$. What is remarkable about this is that measurement of one qubit affects the the other qubit.

Similarly if the measure of the second qubit was 1 then the state would be forced into $|\psi>=|11>$. ■

Consider a system of 100 hydrogen atoms. If each atom has two states for its electron, then this system can store 100 bits of information. However if we can make the system a superposition of the states of the atoms, then we can store 2^{100} bits of information. So this illustrates the opportunity of qubits. We me set or measure the state of this 100 atom system, we can only set or measure 100 bits. However, if the atoms are coupled so that superposition takes place, we can effectively manipulate 2^{100} bits of information.

The challenge of quantum programming is that can set n bits and our answer can be n bits but due to quantum effects, we can use 2^n bits in the middle of our calculation. The drawbacks are that the output will have only a probability of being in the desired state and that we cannot observe our calculation midway through. We will have to manipulate the system without ever observing the system. The usual debugging technique of setting break points and looking at watch windows, will

not work, since any observation of even part of the system will change the entire system. As long as we can guarantee a high enough probability of computing the correct answer, we can increase our confidence in the answer by rerunning the computation several times. improve our confidence in the result.

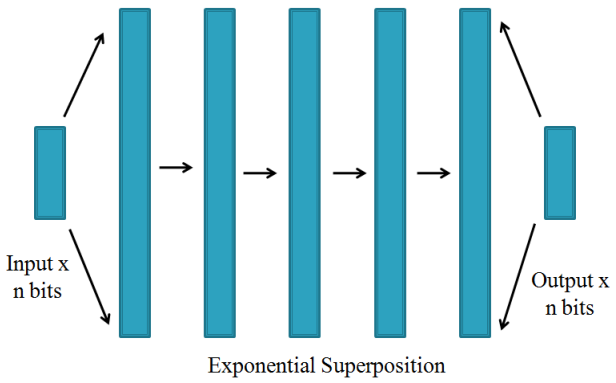


Figure 8.3: Overview of a quantum program.

8.3 Quantum Gates

The classical method to implement the logic of a computer is through the use of gates. These gates, operate on one or two bits and impose the logic of the computer. For example, including NOT, AND, OR and NAND. In similar manner several gates have been proposed for quantum computers that allow the qubits to be manipulated without affecting the entire system. These gates also operate on one or two qubits and result in a new state for the system. The first quantum gate we will discuss is the Hadamard gate. This is a linear gate denoted by H, and effect of the gate is

$$H(\alpha_0|0> +\alpha_1|1>) = \frac{\alpha_0 + \alpha_1}{\sqrt{2}}|0> +\frac{\alpha_0 - \alpha_1}{\sqrt{2}}|1>$$

Using this definition we can see that

$$H(|0>) = \frac{|0>}{\sqrt{2}} + \frac{|1>}{\sqrt{2}}$$

$$H(|1>) = \frac{|0>}{\sqrt{2}} - \frac{|1>}{\sqrt{2}}$$

and

$$H(H(\alpha_0|0> +\alpha_1|1>)) = H(\frac{\alpha_0 + \alpha_1}{\sqrt{2}}|0> +\frac{\alpha_0 - \alpha_1}{\sqrt{2}}|1>) = \alpha_0|0> +\alpha_1|1>$$

The circuit element for the Hadamard gate is pictured below.

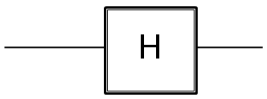


Figure 8.4: Symbol for the Hadamard Gate.

8.4 Grover's Algorithm

In 1996 Lov Grover[] published a quantum algorithm to search a space of size n in time $O(\sqrt{(n)})$ time. We will attempt to explain this algorithm using linear algebra.

Assume are given a function $f(x)$ the maps the integers in $\{0, 1, \ldots, N-1\}$ into either true or false. Additionally assume $f(x_0) = 1$ for some x_0 while $f(x) = 0$ for the rest of values. We want to find x_0. Classically, it would require $O(N)$ evaluations of f to find bfx_0, however Grover's algorithm can solve the problem in$O(\sqrt{N})$ time.

Consider for example that $N = 16$, let $x_0 = 10$. Then a classical search might follow the path $0 \to 1 \to 2 \to 3 \to \ldots 10$. The number of elements to search is $2^4 = 16$ and the average time for a search would be $\frac{16}{2} = 8$.

However quantum computer can use superposition so the search might look like $|Super_1 > \to |Super_2 > \to \ldots 10$, where $Super_i$ indicates a superposition (linear combination) of states. The computation will start with $|Super_1 >= \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x >$This is the superposition where all the N states are equally likely.

Now we want to transform this superposition into one where the x_0^{th} coefficient is 1 and the rest are 0. We will call this solution superposition r. At any time our possible solution q can be represented as a linear combination of the s and r as : $q = as + br$, where $|a|^2 + |b|^2 = 1$.
Initially we have that $a = 1$ and $b = 0$ and we want to get closer to a vector where $a = 0$ and $b = 1$. At each step of Grover's algorithm, operate on our super position with quantum gates. This is analogous to multiplying the vector q by an $N \times N$ unitary matrix. A unitary matrix U has the property that $|Uq| = |q|$, or it does not change the length of the vectors when the matrix is multiplied by the vector.

The first unitary matrix we will use is A where A is the diagonal matrix that has $1's$ on the diagonal except for the x_0^{th} row where it has -1. Basically we will be applying $1 - 2f(x)$ to each element of our superposition.

The next unitary matrix we use is B where B is defined as :

$$B = \begin{bmatrix} \frac{2}{N} - 1 & \frac{2}{N} & \frac{2}{N} & \frac{2}{N} \\ \frac{2}{N} & \frac{2}{N} - 1 & \frac{2}{N} & \frac{2}{N} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{2}{N} & \frac{2}{N} & \frac{2}{N} & \frac{2}{N} - 1 \end{bmatrix}$$

The algorithm is just to compute the vector $q = (BA)^m s$, where $m \approx \sqrt{(N)}$.

The action of A and B on q is a rotation though an angle of $\arcsin(2\sqrt{\frac{1}{N}})$. Taylor series, or the small angle approximation to sin() implies that this is approximately $2\sqrt{\frac{1}{N}}$ for large N.

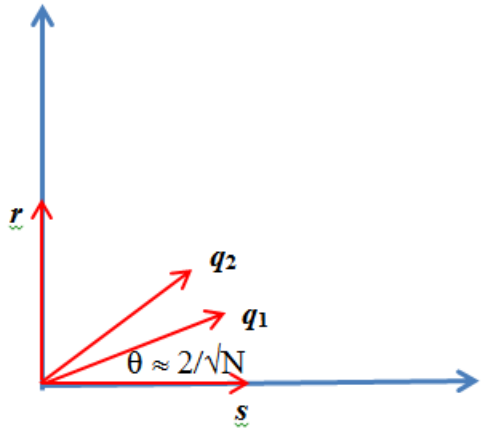


Figure 8.5: Geometric Interpretation of Grover's Algorithm

The operator A is a reflection though the hyperplane orthogonal to r while the operator B is a reflection though q. It can be shown that the exact probability of measuring the correct answer after m iterations is $\sin^2((m+\frac{1}{2})2\arcsin\frac{1}{\sqrt{N}})$.

8.5 Deutsch's Algorithm

8.6 Exercises

1) What is the result of the quantum circuit pictured in Figure 8.6 on the the qubit $|\psi>=|01>$

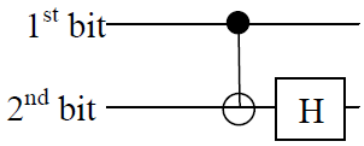


Figure 8.6: Quantum Circuit for Exercise 1

2) What is the result of the quantum circuit pictured in Figure 8.7 on the the qubit $|\psi>=|01>$

8.6.1 Solutions to exercises

1) $\frac{1}{\sqrt{2}}(|00>+|01>)$

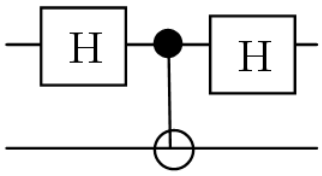


Figure 8.7: Quantum Circuit for Exercise 2

2) $\frac{1}{\sqrt{2}}(|00> + |01>)$

Bibliography

[1] Shor, Peter W. (1997), "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer", SIAM J. Comput. 26 (5): 14841509.

[2] Nielsen, M. and Chuang,I. ,*Quantum Computation and Quantum Information*, Cambridge University Press, 2000.

[3] Deutsch, David (September 8, 1989), "Quantum computational networks", Proc. R. Soc. Lond. A 425 (1868): 7390.

[4] Grover L.K.: A fast quantum mechanical algorithm for database search, Proceedings, 28th Annual ACM Symposium on the Theory of Computing, (May 1996) p. 212

[5] Grover L.K.: From Schrödinger's equation to quantum search algorithm, American Journal of Physics, 69(7): 769-777, 2001. Pedagogical review of the algorithm and its history.

[6] http://rjlipton.wordpress.com/2010/08/25/quantum-algorithms-a-different-view-again/