THEORY OF COMPUTATION
CSc202 2022

Philip Sterne
revised and extended by Karen Bradshaw and Alan Herbert
with additions by Philip Machanick

Department of Computer Science
Rhodes University

Contents

List of Figures

Listings

Chapter 1

Introduction

This course examines some central issues in Computer Science such as: "What kind of problems can we expect to solve with a computer? Are there problems that we cannot solve efficiently? Can we find algorithms for all problems?"

1.1 Historical Overview

We can trace the history of computability and computation back to its origin with the first ideas about theoretical limits to mathematical computation in the early 1900s. The first machines we would recognise as computers were code-breaking machines in World War II; this lead to the development of general-purpose computers in the 1950s. In the early days of development of electronic computers, these ideas were further developed as computation moved away from paper to electronic machines. Up to that time, the word "computer" meant a person who performed computation. In one of the last major projects before elecronic computers replaced humans, a group of African American women did a lot of the computation needed for the early space race [Shetterly 2017]; that women did this work was not unusual and women were also among the pioneers of computer programming and computer science [Howell 2017; Gürer 1995].

- 1930s, 1940s: Researchers like Alonzo Church, Emil Post, Alan Turing, A.A. Markov and Stephen Kleene contributed proofs, models, and unsolved problems forming the basis of the theory of computing still relevant today. For example, Church's thesis is still considered the most general concept of automated computation.
- 1950s: Noam Chomsky defined a set of formal languages and grammars, on which modern computing languages are based, while Alan Turing introduced the Turing Machine, which still forms the basis of all modern computer architectures. (The latter will only change when quantum computing becomes a reality.)
- 1960s, 1970s: Much work was done on determining minimum resources required for computation; for example, how many operations and/or memory cells are needed. Nowadays, not many computer users are even aware of these limitations, owing to the vast (relatively speaking!) resources available in modern computers.

1.2 Example Problems

Some of the example problems we will consider in this course are described in the next few subsections. Can you spot the problems that are efficiently solvable? Are there problems that may not be solvable for all inputs?

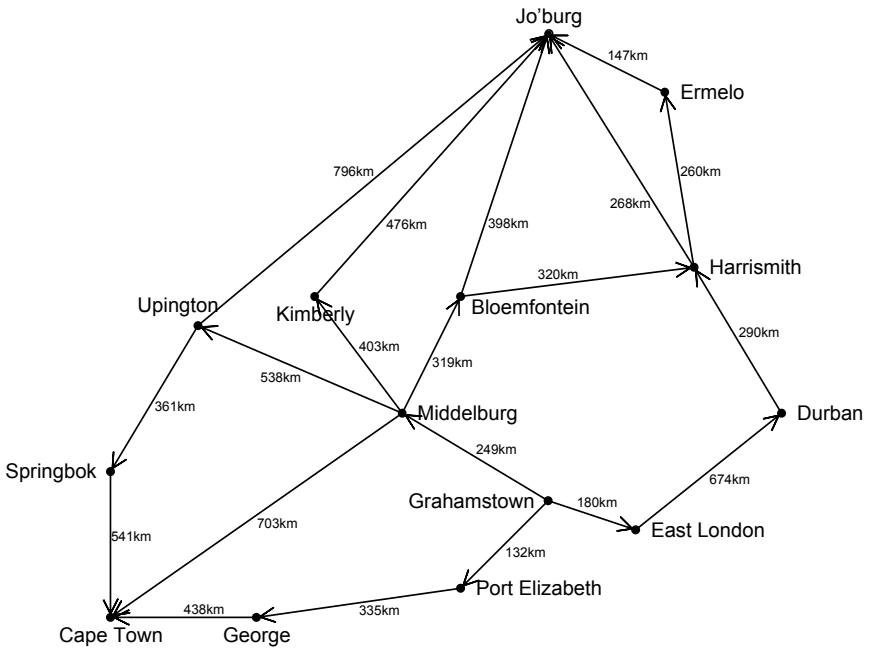


Figure 1.1: Travelling home from Makhanda. With petrol so expensive it is important to find the shortest distance!

1.2.1 Weary student

After finishing exams, the weary student needs to return home to recuperate for the next semester. Unfortunately, they are not sure what the quickest route to their home town is. (Roads do not go directly from Makhanda to every other town.) Given a description of the roads that connect different towns (and their lengths) as shown in Fig. 1.1 can you find the shortest distance solution?

1.2.2 Cable laying

After deciding that the science faculty makes too many geeky jokes, the humanities faculty successfully petitions Rhodes University to segregate the scientists from the rest of campus. Each department is set up in its own new building, but the scientists soon discover that there is no Internet access. An emergency of this magnitude must be dealt with immediately, but the staff are undecided on the quickest way to solve this problem. Given a single team of workers, what is the quickest way to connect all the buildings? The information including the time needed to trench fibre between each building is given in the form shown in Fig. 1.2.

1.2.3 Traveling salesperson

A company has just launched a new range of household cleaning products. Named the Whizzo™ range they could potentially change household cleaning as we know it. To promote this range, a salesperson must tour all the major cities. Ideally they would like the tour to be as short as possible and include every city, without visiting any city twice. Given a road map showing which cities are connected by roads and the lengths of the roads, can you find this ideal route? (Obviously, in some cases there is no exact solution since the roads may force the salesperson to visit the same city twice.)

1.2.4 New manager

Congratulations! You have just been hired as a manager by an ailing company. After some investigation you realise that many employees are performing tasks to which they are not suited. You decide that the

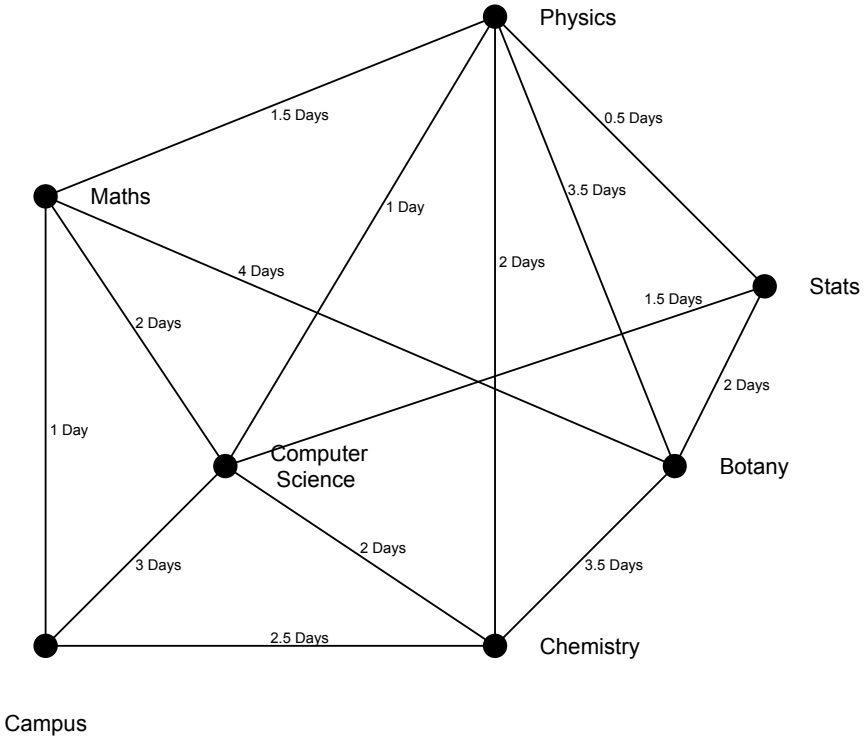


Figure 1.2: How best to connect new departmental buildings to campus? This figure shows the time it would take to lay a cable connecting two departments. (Note: If Department A is connected to campus and we connect Department B to Department A then it is also considered connected to campus.)

best way to turn things around and make the company profitable again is to reassign employees to tasks that better suit them. Given a description (see Fig. 1.3) of the tasks the employees can perform and the tasks needed to be completed, can you find the best assignment of tasks to employees?

1.2.5 Program analysis

MicroNaff has several buggy programs bundled together in an office suite. Most of the bugs result in infinite loops and their customers are getting rather upset with them. They decide that rather than find the bugs, they will instead write a program analysing their office suite and decide whether the program will terminate for all possible input. If their new program gives the OK for their office suite, MicroNaff can rest assured that the customers are obviously imagining the bugs. Can you write a program that will test another program for an infinite loop?

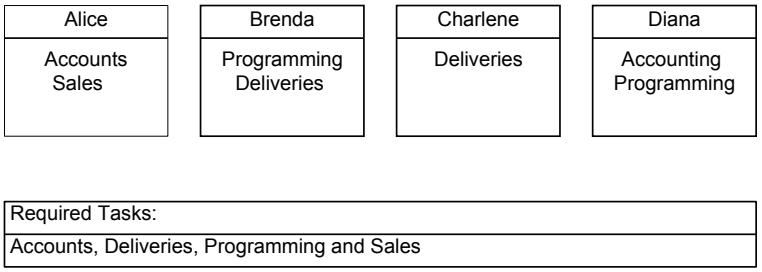


Figure 1.3: Given many employees each with different abilities, and a set of tasks, find the best way to assign the employees to these tasks.

1.3 What Next?

It is not entirely obvious which of the above problems can be solved efficiently. In fact, some problems listed above can be proven not to be solvable in general. Can you spot which these are? Since the limits of computation are not entirely obvious, this course will first consider simpler computation systems and their limitations in the next few chapters. Using simpler systems makes it possible to find their limits of computation more easily. These limitations then suggest what changes should be made to turn them into more powerful systems, thereby slowly building up to the conceptual equivalent of modern computers.

Chapter 2

Finite State Automata

In this chapter we first discuss regular expressions, which are used to define patterns for matching strings. Thereafter, we consider our simplest modest of computation – the Finite State Automaton[1] (FSA). This machine has the ability to distinguish between valid and invalid strings. The set of valid strings for an FSA is known as its *language*. We will look at another way of deriving an FSA's language using a grammar. FSA are useful for lexical analysis (which is covered in the third year compiler course) as well as string matching.

After defining all the components of an FSA, several examples are shown to make the concepts more concrete. An extension of the FSA is also considered, giving rise to non-deterministic automata. Finally, we discuss what languages FSA cannot recognise, which will suggest how to turn them into more powerful machines.

2.1 Regular Expressions

2.1.1 What is a regular expression?

A *regular expression* (or regex for short) is a text string that describes a search pattern. You may already have encountered wildcard notations when working at the command line, e.g. using `ls *.c` in Linux to list all C program files in a directory, or `dir *.c` to do the same in a Windows environment. You can think of regular expressions as "wildcards on steroids".

A common notation for patterns is described as follows. Let Σ represent the finite set of symbols belonging to the language, called the *alphabet*.

For example, $\Sigma_i = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ is an alphabet that we could refer to as *digits*. Note however, that $N = \{0, 1, 2, 3, 4, 5, \ldots\}$ is not a valid alphabet as it is not finite.

Using a given alphabet, we can construct strings of length n ($n \geq 0$), where each string is an ordered n-tuple of elements of Σ written without punctuation. So for example, consider a small subset of the English language with $\{a, \ldots, z, A, \ldots, Z\}$ as its alphabet. Strings in this language could include `dog`, `Rhodes`, and `student`[2]. For a simple language with $\Sigma = \{a, b\}$, possible strings might include `aa`, `ababab`, and `b` amongst others. These strings have length 2, 6, and 1, respectively.

The set of all strings (of any finite length) over Σ is denoted by Σ^*. The *empty string* is a unique string for all alphabets; it is denoted by Greek letter ϵ (pronounced epsilon or sometimes λ –lambda), and has zero length.

[1] A note on Latin plurals: singular is *automaton*; plural is *automata*.
[2] Random examples; they are not related.

2.1.2 Creating and matching regular expressions

When defining a language, having an alphabet is not enough; we also need rules for creating strings over that alphabet. There are several rules for creating regular expressions as given below:

1. Single symbol: Any symbol $a \in \Sigma$ is a regular expression. In addition, ϵ is a regular expression.
2. Concatenation: Any two strings can be placed adjacent to each other to form a new string. For example, let $u =$ `abb` and $v =$ `abab` where $\Sigma = \{a, b\}$; then $uv =$ `abbabab`. This rule can be applied to two or more strings.
3. Union: This rule involves using the union operator, denoted by $+$. Here, $u+v(u, v \in \Sigma^*)$ means either u or v. This rule can also be applied to two or more strings.
4. Repetition: The star operator can be used to create repeated substrings, where u^* gives $u_1u_2 \ldots u_k$, where $k \geq 0$. Thus, if $u =$ `abb`, u^* gives ϵ or `abb` or `abbabb` or `abbabbabb`, etc. Note that the star operator only applies to its immediate antecedent; therefore, you are should use parentheses if the star operator applies to a complex regular expression. For example, $a + b^*$ denotes strings ϵ or `a` or `b` or `bb` or `bbb`, etc. Regular expression $(a + b)^*$, however, denotes any combination of `a`'s or `b`'s as well as ϵ.

Now that we have rules by which to create regular expressions, how do we use these to determine whether a given string is valid (i.e. it matches the pattern given by the regular expression)? Using the rules given above for creating regular expressions, we can state that:

- u matches $a \in \Sigma$, iff $u = a$
- u matches ϵ, iff $u = \epsilon$
- u matches $(r + s)$, iff u matches r or u matches s
- u matches rs, iff $u = vw$ and v matches r and w matches s
- u matches r^*, iff u either matches r or u is a concatenation of strings, each of which matches r – or u is ϵ

Some examples of regular expressions for $\Sigma = \{0, 1\}$ are given below, together with an explanation of what kinds of strings would be valid for each of these.

- `0+1` means only strings `0` and `1` are valid
- `(0+1)`* means ϵ or any combination of 0's and 1's in Σ^* is valid
- `0(0+1)`* means any combination of 0's and 1's starting with a `0` is valid
- `((0+1)(0+1))`* means any string of even length in Σ^* is valid[3]
- `(`ϵ`+0)(`ϵ`+1)+11` means only strings ϵ, `0`, `1`, `01`, `11` are valid

2.2 What is an FSA?

2.2.1 Definition

A Finite State Automaton is a machine suited to string recognition. As it reads a string the FSA changes its internal state based on the string's characters. Some of these states are accept states so that if the string ends when the machine reaches in an accept state then the whole string is accepted. An informal representation of an FSA is shown in Fig. 2.1, and the basic processing is shown in Fig. 2.2.

An FSA consists of several entities, which need to be defined (taken from Brookshear [1989]):

[3]Noting that zero is an even number.

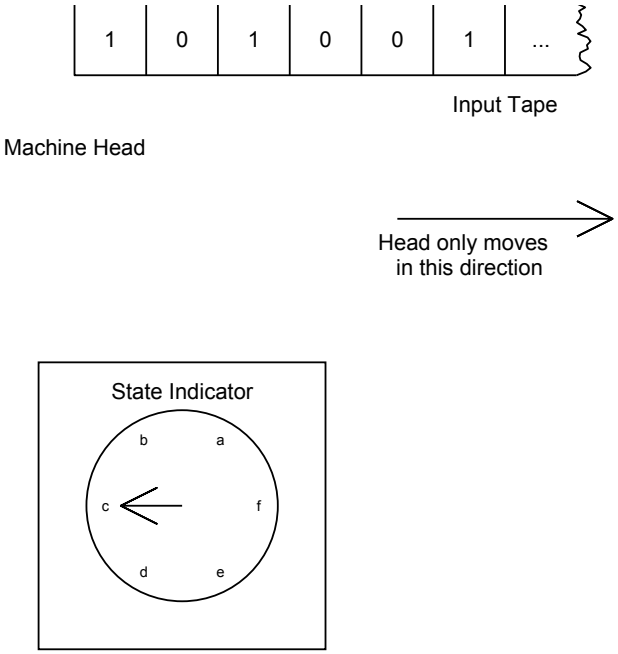


Figure 2.1: Conceptual model of an FSA.

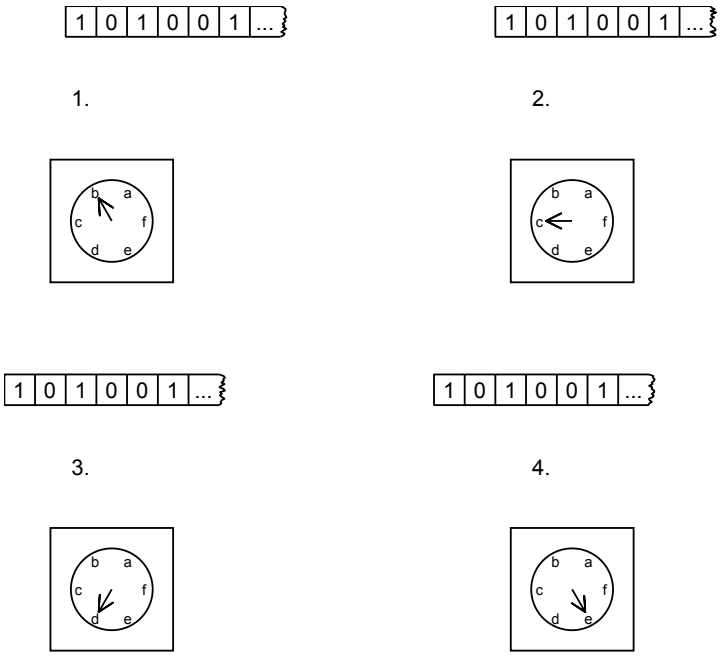


Figure 2.2: As the input string is processed the internal state of the machine changes.

state	a	b	c	***accept?***
q0	q1	error	error	no
q1	error	q2	error	no
q2	error	error	q3	no
q3				yes

Figure 2.3: FSA as a table. Every row is a state and has a transition for each letter of the alphabet (here, $\{a, b, c\}$). If you label states with numbers (so they index the next state in the table) rather than names, you can use an invalid number like -1 to indicate an error. The last row has no entries for the alphabet since there are no transitions out of it but an accept state can have transitions out of it.

1. **Alphabet:** The alphabet of an FSA is the set of all characters from which the strings to be recognised are constructed.

2. **States:** An FSA consists of a set of states. These represent intermediate or final steps in determining whether the string is acceptable or not.

3. **Transition Function:** The transition function (usually denoted δ) is the heart of the FSA. It is a mapping from states and characters to the next state. This function therefore determines the behaviour of the FSA. If a machine encounters symbol a while in state 12, it will move to the new state determined by $\delta(12, a)$.

4. **Accept States:** A subset of the FSA's states. If the machine finishes the string and is currently in an accept state then the entire string is accepted, otherwise the entire string is rejected.

A way to represent the transition function is to use a table as in Fig. 2.3 in which there is a column for each symbol in the alphabet. The start and accept states have to be separately noted. Each entry in the table contains the number of the next state or an error indicator. We can use -1 to indicate an error; we can't indicate an accept state as easily as an accept state could also allow a transition out of it.

These five items are the only things allowed as part of an FSA and together define it completely. Since it is hard to visualise the transition function we normally depict FSAs by graphical means, using circles to denote states and arcs between these circles represent the transitions given by the transition function. A single arrow points out the start state. Double circles are used to indicate accepting states (of which there must be at least one). Each arc is labelled by the symbol (or ϵ) that is matched to trigger that transition.

While a function must be defined for all possible inputs this can result in a cluttered graph. As a result we will only show transitions that leave the FSA in a state from which it might still accept the string. Implicit in our diagrams is an error state. Any undefined characters for each state will transition to this error state. The error state is not an accept state and it is not possible to leave this error state. By adopting these conventions our diagrams are far easier to read. See Fig. 2.4 for a comparison.

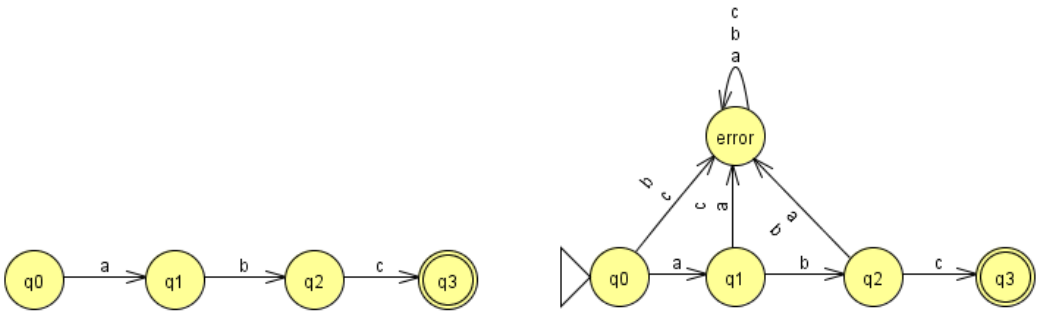


Figure 2.4: If a possible transition is not shown then it is assumed to lead to an implicit error state with no transition out of it. This makes the two FSAs shown here equivalent.

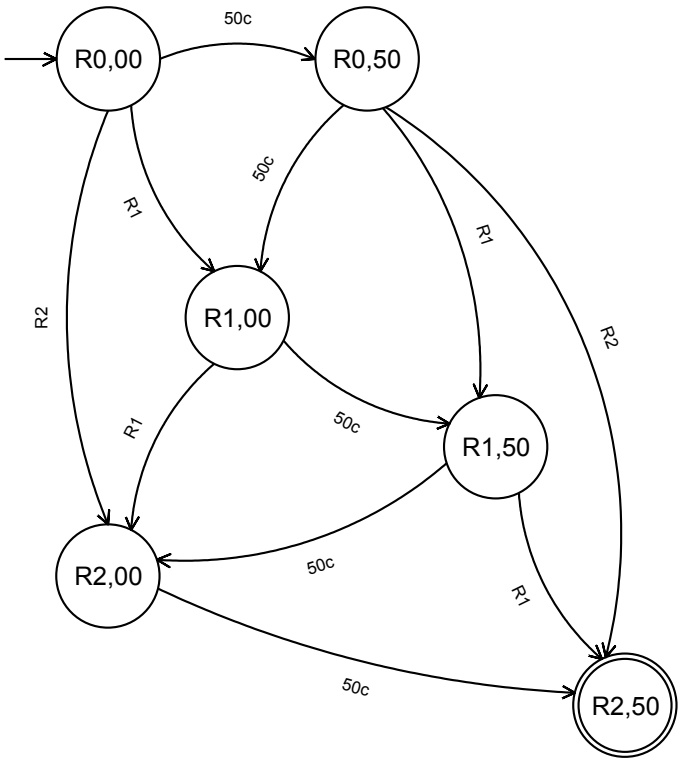


Figure 2.5: Simple vending machine FSA.

2.2.2 Example FSA

Vending machine

In Fig. 2.5 we see a simple soft drink vending machine. It accepts 50c, R1, and R2 coins. When exactly R2.50 is reached the machine moves to an accept state and dispenses the soft drink. Spend a few minutes getting used to this FSA. What is the alphabet of this machine? What changes would we need to make if we allowed 20c coins? What if the price of a can was a more realistic R19.50?

Recognising numbers

In Fig. 2.6 we have an FSA that is able to recognise some of the valid `floats` for Java or generally a C-like programming language. In this case the alphabet consists of the set $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ., f\}$. To make the diagram more readable we use the shorthand $digit$ to represent a numeric character. Can you create an FSA that recognises all valid `longs`? How about `doubles`?

2.3 Efficient String Recognition

In this section we'll consider a powerful application of these automata: string search. This differs slightly from the previous examples, in that we are happy if the string we are searching for matches part of a longer string, rather than a test whether a whole string matches a given pattern.

If asked to search for a given string in a long text, most computer programmers will come up with a solution similar to listing 2.1, without too much effort. This is called *naïve string search* – the obvious algorithm. Take one position at a time in the string you are searching for and compare against the longer string your are searching in. If there is a mismatch, slide the search string along one position and start again. Stop when you reach the end of the search string and report where the match was in the long string, or report -1 if you fall off the end of the long string without a match.

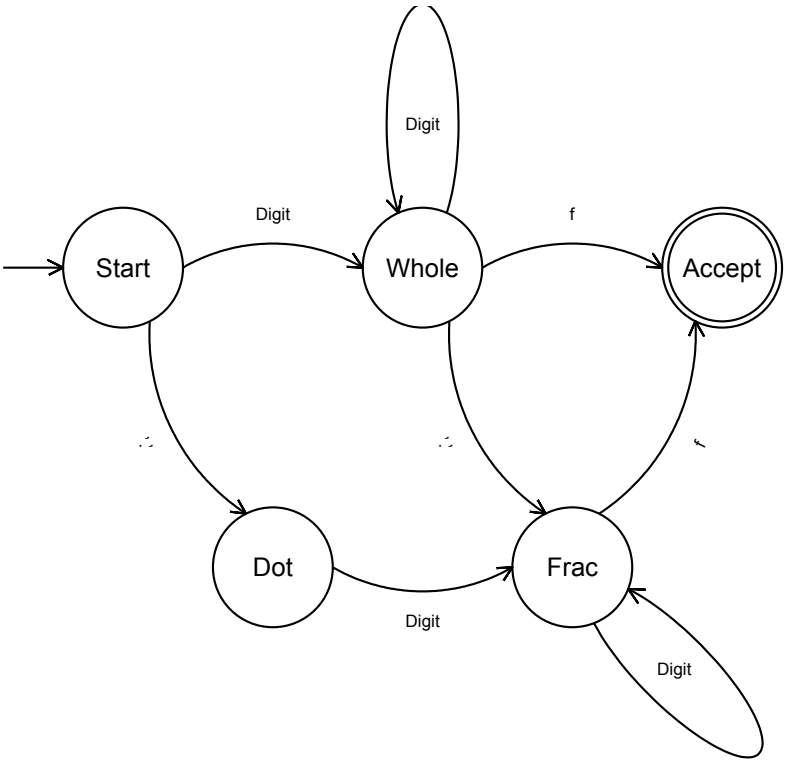


Figure 2.6: FSA that recognises valid floating point numbers.

```
1     int find(char *f, char *longString){
2         int llen = strlen(longString);
3         int flen = strlen(f);
4         for (int a=0; a<llen; a++)
5         {
6             for (int b=0; b<flen; b++)
7             {
8                 if (a+b >= llen)
9                     return -1;
10                if (longString[a+b] != f[b])
11                    break;
12                if (b+1 == flen)
13                    return a;
14            }
15        }
16        return -1;
17    }
```

Listing 2.1: Naïve string search

Analysing the complexity of this algorithm, we see that it is a function of the length of the search string (m) and the length of the text in which to search (n). The worst case complexity for this code occurs when searching for strings with lots of repeated characters. Trace through the code with `f="aaaaab"` and `longString = "aaaaaaaaaaaaaaaaaaaaaaaaab"`. This should convince you that the worst-case order of this algorithm is $O(nm)$.

Fortunately there is an algorithm called Knuth-Morris-Pratt (KMP) after its inventors [Knuth et al. 1977] that has $O(n + m)$ complexity. It is based on the idea of creating an FSA that can recognise the string in a fast manner, with a number of tricks to avoid repeating work. The FSA in Fig. 2.7 reaches the accept state if and only if the string `"aaaaab"` has just been read. Using this FSA makes string recognition very easy. We simply feed in the string, a character at a time to the FSA, if it ever reaches the accept state then we know we have just finished reading the search string and can stop. This

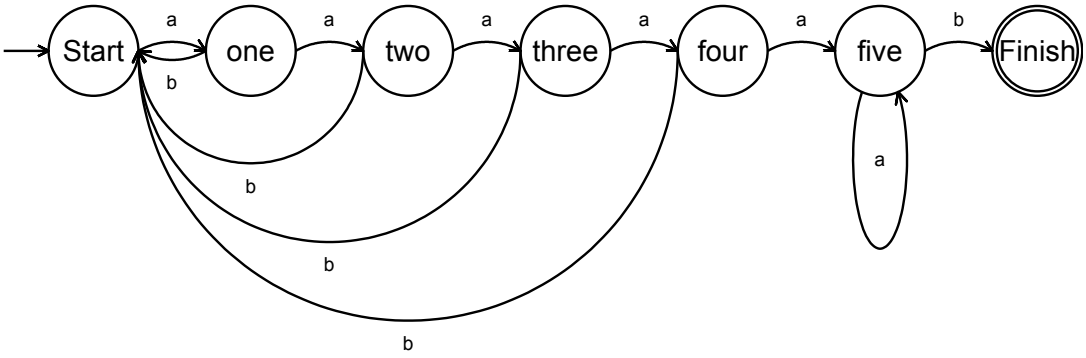


Figure 2.7: FSA that recognises the string `"aaaaab"`.

procedure is clearly linear in the length of the text to search (i.e. $O(n)$). KMP differs from an FSA in adding links to handle mismatches, which can result in skipping far ahead in the string; we will stick with FSAs here, since we are working through models of computation, not methods of string search[4].

An FSA-based method is useless if there is not an efficient means of constructing the FSA. To construct the FSA we work exclusively with the search string. To represent the FSA we note that there are only transitions one step forward (if the string matches) or a single jump backwards (if the string does not match). We use an array to store the indices of these jumps backwards.

When determining how far back to jump the crucial insight is to see that one must jump to the previous state that could start the search string. As an example consider the search string '`ababcd`'. In this case if '`abab`' has already been matched and the next letter is an '`a`' then there is a failure (since we did not read a '`c`') but we could have already started reading '`aba`' of the string we are looking for. To check this we jump back to the matched state '`ab`' and try to continue matching.

Given a matched string of length n, the problem is reduced to finding the smallest initial string that can be thrown away while still leaving a string that forms an initial part of the search string. Fortunately knowing how much of the string to throw away for the matched string of length $(n-1)$ makes the task considerably easier. If the current character matches the next character then the position to jump back is increased by one, otherwise the jump is back to the beginning of the string. In most cases there is no such initial string and the matching process must start from the beginning again.

2.4 Non-deterministic FSA

Let's try make our FSA more powerful. To do this, we introduce the concept of *non-determinism*. This means our FSA can have *several transitions for the same character in the same state*. We assume that our FSA can either 'magically' pick the correct decision, or it has the ability to explore all possible options in parallel.

If a state has more than one transition out and one is labelled with the empty string, by ϵ, that also makes the FSA nondeterministic as there is no way to determine whether the transition labelled ϵ should be taken in preference to a non-empty transition without looking ahead.

How would a non-deterministic automaton (NFA) accept strings? Since we do not have the ability to 'magically' pick the correct transition let us deal with performing the operations in parallel. The NFA would need to maintain a set of all possible states that it could be in. If one of those states encounters the implicit error state then it ceases to be a possible state and the machine can discard that possibility. When the string terminates, if at least one of the possible states is an accept state then the entire string is accepted, otherwise it is rejected.

As an example of an NFA consider Fig. 2.8. This NFA will recognise only integers, longs, floats and doubles, which are valid in Java, C etc. The first transition in the NFA is not deterministic since

[4]Here is a detailed description of KMP: `https://www.scaler.com/topics/data-structures/kmp-algorithm/`

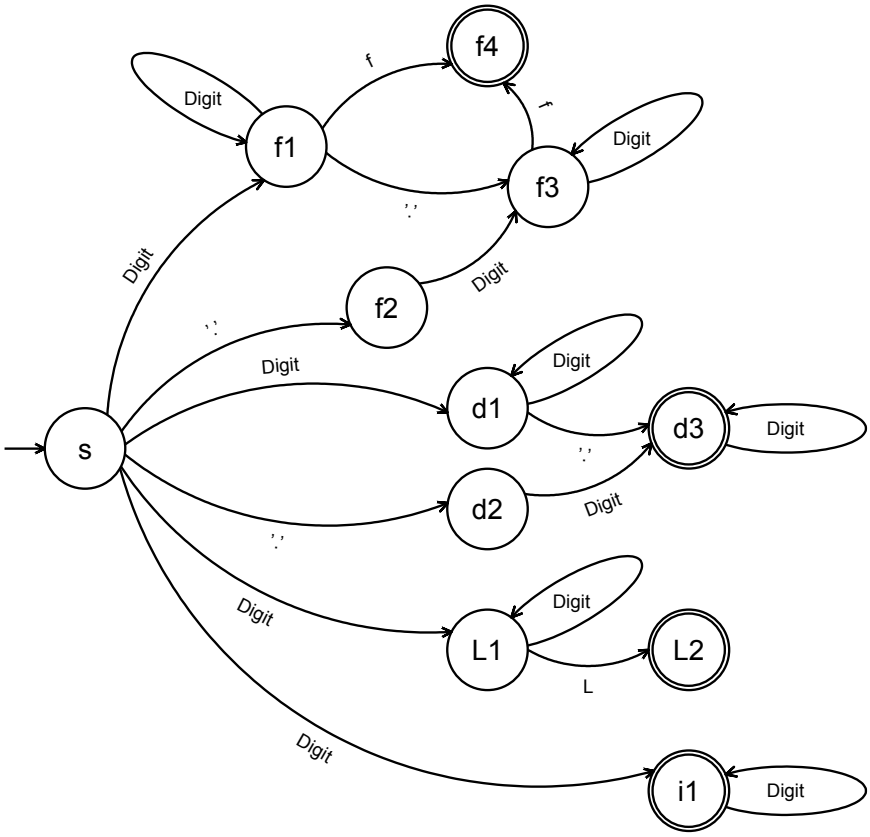


Figure 2.8: A non-deterministic FSA that recognises `double`, `long`, `int` and `float` types. Notice how simple deterministic FSAs have been combined to create this NFA.

there are several transitions that expect a digit (and two that expect '.' – in quotes to make the dot easier to see). This means that our NFA will start by either magically guessing the correct type of the string or running all the possibilities in parallel.

If the NFA in Fig. 2.8 was to process the string '`1.3f`'. The sequence would be as follows:

1. After the '`1`' is encountered then the possible states would be $\{f1, d1, l1, i1\}$.

2. Encountering a '`.`' reduces the set of possible states to$\{f3, d3\}$.

3. The '`3`' leaves the set of possible states unchanged.

4. '`f`' leaves but a single state left $\{f4\}$. Since this is the end of the string and $f4$ is an accept state the entire string is accepted.

It can be shown that for every NFA there is an FSA that is able to recognise the same language! This is a fairly unexpected result since it seems that we are endowing our machine with a powerful ability, yet there is no additional power. The key insight is to realise that while the NFA is maintaining a set of possible states, that set of states is itself a state. Since the number of states is finite, the number of possible subsets (or power-sets) is also finite. We can represent the different subsets as different states, and the resulting automaton is deterministic. This process is shown in Fig. 2.9 where each set of states is merged into a single state in a new deterministic finite automaton (DFA) that recognises the same strings as the original NFA.

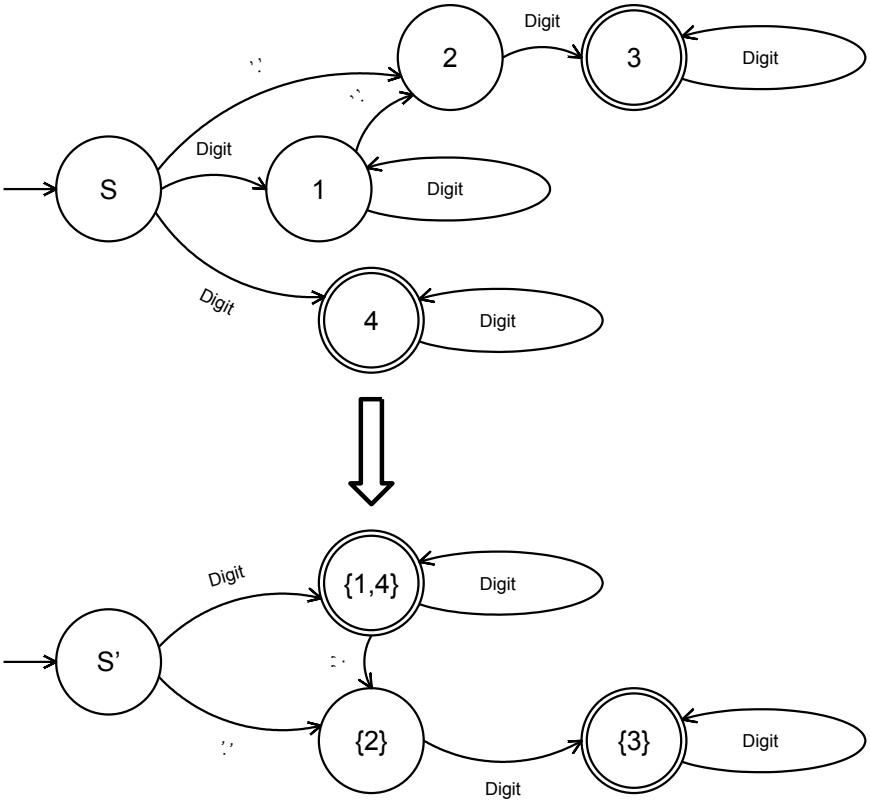


Figure 2.9: Converting a nondeterministic automaton into a deterministic one. Since the start state has two transitions for a digit (states 1 and 4), we create a new state that is a set of states 1 and 4, both of which are reached from state S by a digit. As state 4 is an accept state, the new state $\{1, 4\}$ is also an accept state. If in this new state we encounter a digit then state 1 transitions to itself and state 4 transitions to itself as well. This means in the state $\{1, 4\}$ a digit also transitions to itself. However if a '.' is encountered, state 4 moves to the error state and state 1 moves to state 2. Thus, if state $\{1, 4\}$ encounters a '.' it moves into state $\{2\}$. The rest of the transitions are similarly obtained.

2.5 Links to a Grammar

2.5.1 What is a grammar?

Thinking back to your school days, you are likely to have been taught some grammar rules for a spoken language. For the English language, these generally take the form of:

```
Sentence ↦ NounPhrase Predicate

NounPhrase ↦ Article Adjective CommonNoun
NounPhrase ↦ ProperNoun

Predicate ↦ Verb NounPhrase

Article ↦ a
Article ↦ the

Adjective ↦ hard-working
Adjective ↦ nasty

CommonNoun ↦ prac
CommonNoun ↦ student
ProperNoun ↦ Philip

Verb ↦ solves
Verb ↦ helps
```

This grammar allows us to derive very informative strings such as "`Philip helps the hard-working student`"[5] and "`a hard-working student solves the nasty prac`". In the above grammar we can see that there are symbols such as "`<NounPhrase>`" that never appear in the final string. We will call these symbols **non-terminals** and the symbols that do appear in the final string, **terminals**. Note that the non-terminal symbols all start with an uppercase letter, while the terminal symbols start with a lowercase letter. This convention will be used throughout these notes.

It is interesting to characterise the set of all strings accepted by an FSA. We call this set the *language* of the FSA. It turns out that this set can be defined in a different manner to an automaton. This brings us to the idea of a *regular grammar*.

2.5.2 Regular grammar

There is a simple grammar called a *regular grammar* that can generate all possible strings accepted by an FSA. The grammar has a very restricted form with the rules taking one of the following two forms:

```
NonTerminal ↦ terminal NonTerminal
NonTerminal ↦ terminal
```

As an example let us consider creating the grammar for the FSA, considered previously, to recognise floating point numbers (see Fig. 2.6):

[5] Similarity of the names of the first and third author is a coincidence.

```
Start ↦ 0 Whole
Start ↦ 1 Whole
Start ↦ 2 Whole
Start ↦ 3 Whole
Start ↦ 4 Whole
Start ↦ 5 Whole
Start ↦ 6 Whole
Start ↦ 7 Whole
Start ↦ 8 Whole
Start ↦ 9 Whole
Start ↦ .  Dot

Whole ↦ 0 Whole
Whole ↦ 1 Whole
Whole ↦ 2 Whole
Whole ↦ 3 Whole
Whole ↦ 4 Whole
Whole ↦ 5 Whole
Whole ↦ 6 Whole
Whole ↦ 7 Whole
Whole ↦ 8 Whole
Whole ↦ 9 Whole
Whole ↦ f
Whole ↦ .  Frac

Dot ↦ 0 Frac
Dot ↦ 1 Frac
Dot ↦ 2 Frac
Dot ↦ 3 Frac
Dot ↦ 4 Frac
Dot ↦ 5 Frac
Dot ↦ 6 Frac
Dot ↦ 7 Frac
Dot ↦ 8 Frac
Dot ↦ 0 Frac

Frac ↦ 0 Frac
Frac ↦ 1 Frac
Frac ↦ 2 Frac
Frac ↦ 3 Frac
Frac ↦ 4 Frac
Frac ↦ 5 Frac
Frac ↦ 6 Frac
Frac ↦ 7 Frac
Frac ↦ 8 Frac
Frac ↦ 9 Frac
Frac ↦ f
```

It should be evident from studying Fig. 2.6 that there is a one-to-one mapping that turns our FSA into this grammar. The astute reader will note that for the grammar rules we have given an explicit rule

Table 2.1: Chomsky's Hierarchy

Language class	Grammar	Automaton
0	Unrestricted	Turing machine
1	Context-sensitive	Linear-bounded automaton
2	Context-free	Push-down automaton
3	Regular	NFA or DFA

for each digit (which is the way it should be), whereas in the figure, we have represented multiple transitions by a single path to make the diagram more readable. When we do practical work, we use a tool, JFLAP, to implement automata, and JFLAP requires that you create a specific transition for each input symbol.

To perform the conversion we turn our states into the non-terminals and for every transition of the form $\delta(State, c) = State'$ we add a rule to our grammar of the form `State` $\mapsto$ `c State'`. If the state is an accept state then we add the rule containing a single terminal: `State` $\mapsto$ `c`.

It is possible to prove that an FsA can be represented by a regular grammar and vice-versa and that regular expressions are also equivalent in expressive power. I other words, all three notations are equivalent (define the same class of language).

2.5.3 Chomsky's hierarchy

Not all computer languages can be described by a regular grammar. Noam Chomsky categorised regular and other languages as in Tab. 2.1.

Since this is a hierarchy, every language of class 3 is also a language of classes 0, 1 and 2; every language of class 1 is also a language of class 0; and so on. In terms of automata, a machine capable of recognising a class 0 language is a powerful enough machine to recognize classes 1, 2 and 3. But not vice-versa: an FSA for example cannot be constructed to recognize a class 0, 1 or 2 language. So why not just stick with the most general class of automaton? Because the more general the class, the less efficient the automaton is at solving problems that a less efficient atomaton can solve.

The distinction between languages can be seen by examining the structure of the production rules of the corresponding grammar and the nature of the automata that can be used to identify them. In this chapter, we discussed regular grammars and showed the format of the production rules defining such grammars. In the next two chapters, we look at the remaining language classes defined by Chomsky.

2.5.4 Recognising strings or performing computations?

The approach we have taken thus far, might feel unnatural to some. We started out with the intention of exploring the limits of computability and have now wandered into defining our own grammar. Surely there are limits inherent in a grammar that are not inherent in general computability. Does this mean we should rather forget about boring grammars and rather consider the limits of the latest and greatest Whizzomatic™ computer that has just been released?

It turns out that an unrestricted grammar can be far more powerful than most people realise and can perform any computation that the latest computer can. In fact since the grammar does not have any space limitations it is more powerful since it will never run out of memory.

As an informal argument consider that under certain circumstances *recognising* that a string matches certain requirements is the same as *performing* a calculation. If we found a grammar that could recognise strings of the form: '12+13=25', and '237+1=238', then in effect we could say that the grammar is able to perform addition.

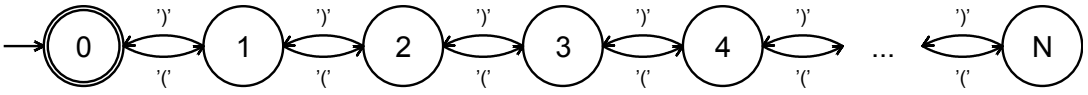


Figure 2.10: An attempt at writing an FSA that can recognise strings with balanced brackets.

2.6 Limitations of an FSA

Such a simple machine cannot hope to do everything, and we run into the FSA's main limitation if we try to design one that can recognise pairs of matching brackets. For my latest computer language I want to read in expressions such as "((((()())))())" and determine whether the brackets are properly balanced. This means that the entire string should contain an equal number of left and right brackets and no prefix of the string should contain more right brackets than left brackets. This suggests an architecture as shown in Fig. 2.10. Each time we encounter an opening bracket we increase the state by one and every time we encounter a closing bracket we decrease the state by one. However to match all possible strings with only a finite number of states is impossible. For any FSA designed it is possible to create a string that is incorrectly handled. If the FSA has 1,000,000 states and then discards any strings that exceed this limit then the string with 1,000,001 opening brackets followed by 1,000,001 closing brackets is incorrectly rejected.

The problem gets even worse if we wish to match pairs of different kinds of brackets. Consider trying to match the following string "[([[]])()[()]]", this problem would suggest a solution of the form given in Fig. 2.11. It should be clear that we cannot use a finite number of states to match arbitrary strings of these forms. Ideally we should have some form of memory that can be used to remember which brackets we have seen. This leads us into the next chapter on Pushdown Automata.

2.7 Pumping Lemma for Regular Languages

But first, let us be clear on why some languages cannot be regular. An FSM has a finite number of states and a finite number of transitions. This leads to the *Pumping Lemma for Regular Languages*.

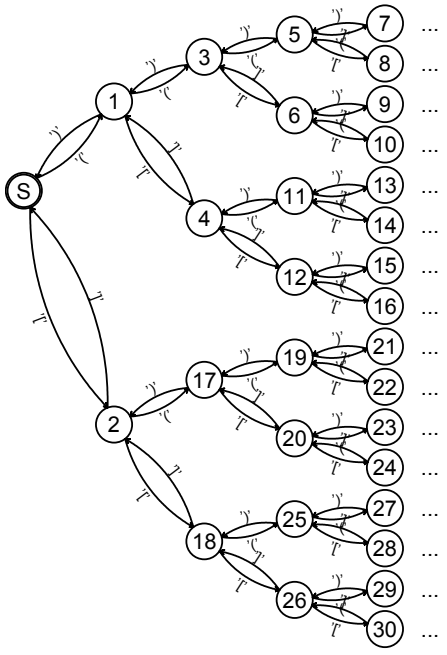


Figure 2.11: Things go *even more* badly wrong when trying to match different kinds of brackets!

For any regular language L, $\exists$ an integer p, such that $\forall x \in L$ with $|x| \geq p$, $\exists u, v, w \in \Sigma*$, such that $x = uvw$, and

1. $|uv| \leq p$
2. $|v| \geq 1$
3. $\forall i \geq 0 : uv^i w \in L$

Let's see if we can translate this into English.

A regular language can always be converted to an FSM and such an FSM must have a finite number of transitions. Any language that allows a string longer than the longest string of transitions with no repetition can only be recognised by an FSM with a loop. So the pumping lemma divides any such string into three parts: a part that may or may not be of a fixed length with no loops (u), a part of non-zero length that definitely has a loop (v) and another part that may or may not have a loop, w. If the language is regular then repeating the v part any number of times should be possible as that corresponds to the way the FSM would be constructed. Repeating this loop is "pumping" a string. If doing so results in a string that is *not* part of the language, we have proved that the language is not regular.

A proof using the pumping lemma for regular languages (there is another one for context-free languages) starts by assuming a language is regular, identifying a substring that could be pumped then showing that repeating that substring takes you out of the language.

For example, you can prove that a language of the form $a^n b^n$ is not regular. Assume that it is regular. Now set $n = p$, as defined in the lemma, and $u = a^n$, with $v = b^n$ and $w = \epsilon$. Then if we "pump" v, we get the string $a^n b^{n+1}$, which is not in the language, so we have proved that the language $a^n b^n$ isn't regular.

Exercise 2.1

What is the purpose of the `Dot` state in Fig. 2.6 (on page 10)? What illegal string(s) would be accepted without it?

Exercise 2.2

Extend the float-recognising FSA so that it also accepts floats of the form : `"3e7f"`, `"3.1415e-1f"` and `".3e01f"`.

Exercise 2.3

Compare the efficiency of the Knuth-Morris-Pratt string matching algorithm with the naive method. How do the methods compare for normal English text? How much of a difference is there on degenerate text where there is lots of repeated characters?

Exercise 2.4

It is always good practise to ensure that illegal strings are correctly rejected. Simulate the NFA from Fig. 2.8 and show that the string '`1.2L`' has no route to a final state, which means the string is rejected.

Exercise 2.5

Turn the NFA in Fig. 2.8 into a deterministic FSA (DFA). You may try the power-set method but it may be easier to try and solve the problem directly. How many states does your solution have? Construct a regular grammar that creates the set of all strings accepted by your automaton.

Exercise 2.6

Show using the pumping lemma for regular languages that checking for matching parentheses cannot be done with an FSM (i.e., a language with this requirement is not regular). *Hint*: assume a string of length p of one of the types of brackets exists in a string.

Exercise 2.7

The pumping lemma for regular languages can prove that a language is not regular. Can it prove that a language *is* regular? Explain.

Chapter 3

Pushdown Automata

In this chapter we move on to a Pushdown Automaton (PDA) by augmenting a finite-state automaton with a stack, which allows it to remember an arbitrary amount of information regarding the previously seen symbols. This allows us to overcome the limitation of an FSA's finiteness, and recognise more languages. We will also show that a PDA is equivalent to a *context-free grammar* (CFG).

Most computer languages are designed to be parsed by a PDA as this simplifies writing the compiler. Of course a machine as simple as a PDA must have some limitations. We highlight these limitations and again seek a more powerful machine in the next chapter.

3.1 What is a PDA?

3.1.1 Definition

Conceptually, a PDA can be viewed as a finite-state automaton augmented with a stack. This is shown in Fig. 3.1. For a more formal definition, a PDA comprises six components:

1. **Input Alphabet:** The set of all characters that can appear on the input tape.
2. **Stack Symbols:** Another set of characters (distinct from the input alphabet). This extra set of characters can be pushed onto the stack to help the automaton remember intermediate calculations.
3. **States:** Exactly as with an FSA, a PDA must consist of a finite number of states.
4. **Transition Function:** The transition function is now a function that considers the current state, the input character just read, and the character on the top of the stack. It then outputs a new state, to which the PDA will transition. It also decides whether to pop a symbol off the stack as well as whether to push any number of symbols onto the stack.
5. **Start State:** A single state must be identified as the state from which to start.
6. **Accept States:** A subset of the states at which, if the end of the string is reached *and* the stack is empty, then the entire string is accepted.

By contrast with FSAs, PDAs have increased power when they are made non-deterministic. As a simple example consider a PDA that detects palindromes (Fig. 3.3(b)[1]). At the beginning of the string characters are pushed onto the stack. For the last half of the string, characters are popped off the stack and compared. If they all match, the string is accepted as a palindrome. Unfortunately it is impossible to tell deterministically when the middle of the string has been reached. To solve this problem we assume that the PDA is able to non-deterministically choose when to start popping the characters off.

[1]We use here the notation of JFLAP, which uses λ for the empty string, which means the same thing as ϵ.

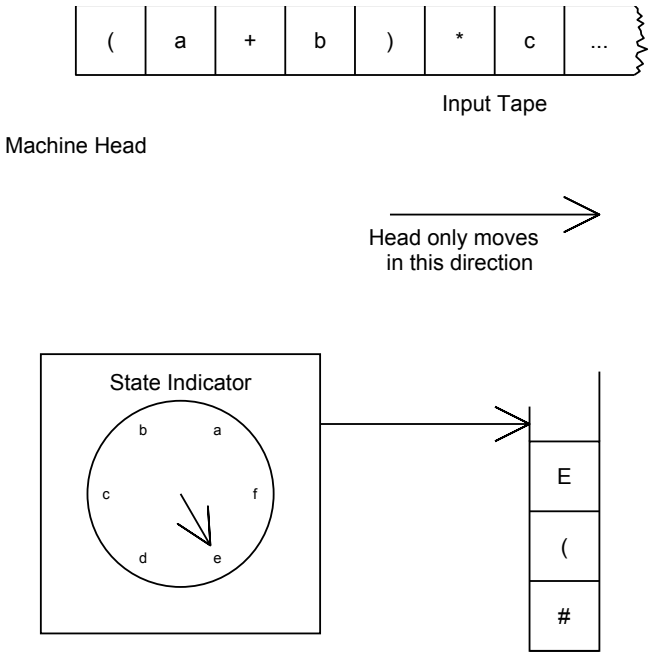


Figure 3.1: Conceptual model of a PDA.

The languages recognised by deterministic PDA are also interesting, and lead naturally to computer programming language design since the languages are sufficiently complex to express one's thoughts, yet still reasonably simple to recognise[2]. To cover them adequately would require an entire course by itself, and is beyond the scope of these notes. The interested reader is advised to consult Aho et al. [1986] or Terry [2004].

[2]It is possible to parse languages from non-deterministic PDA, but the complexity of parsing changes from $O(n)$ to $O(n^3)$. This is done using the CYK algorithm [Cohen 1997].

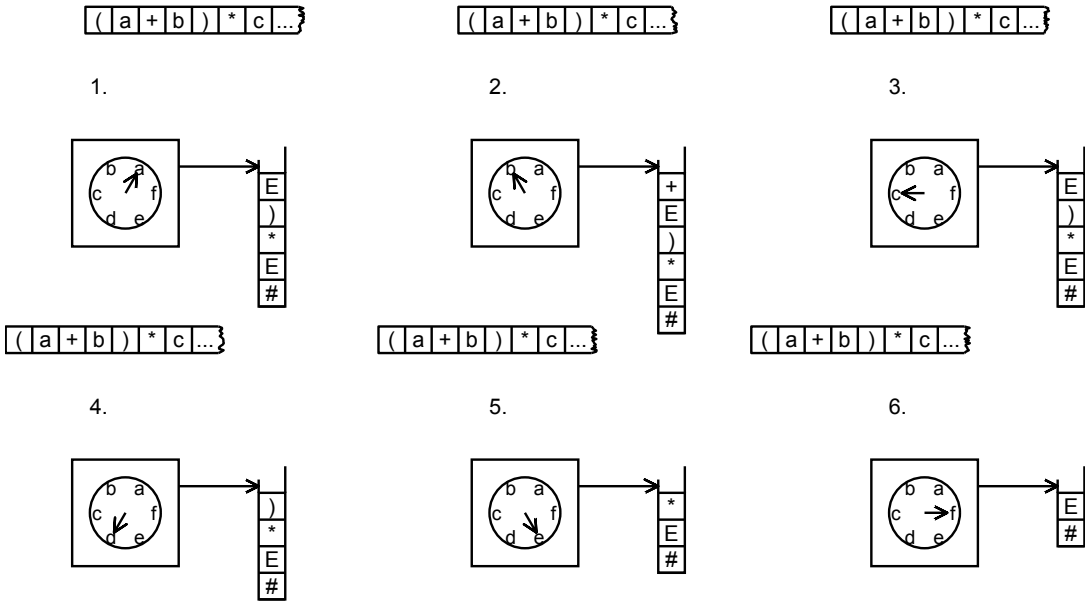


Figure 3.2: As the input string is processed, subexpressions are pushed onto the stack and popped off as characters are read.

3.1.2 Examples of PDAs

Before we cover the examples we first explain the notation used when describing a transition. Every arc is annotated with a string containing three components of the form: '$Read, Pop; Push$'. The first represents the *single* character that will be read, the second represents the *single* symbol that will be popped off the stack. The final component consists of *any number* of symbols that can be pushed onto the stack. In all these cases, where 'λ' (empty string, the same as ϵ) is shown, that action is in effect not performed for that transition.

If the same symbol occurs in more than one of the three components, the transition should only happen if it matches in all cases (e.g., if you have a symbol `X` in the input and also as the symbol to pop, the transition should only happen if the input is `X` and the top item on the stack is also `X` – you can't pop something off the stack unless it is at the top).

Recognising matching brackets

Fig. 3.3(a) depicts a PDA capable of matching different kinds of bracket. Every time we encounter an opening bracket we push it onto the stack. If a closing bracket is encountered, the top symbol is popped off the stack and compared. If the brackets are not of the same type, the PDA goes into the implicit error state, otherwise the recognition process continues. Notice that in this example it is not necessary to use non-determinism.

Look at the transitions for an open bracket of either type. For instance, for a square bracket, the $Read, Pop; Push$ sequence is `[, λ ; [`. So if the input open square bracket is read, nothing (λ) is popped off the stack and an open square bracket is pushed onto the stack.

Compare with the transitions for a close bracket of either type. Staying with square brackets, if the next input is a close square bracket, the $Read, Pop; Push$ sequence is `], [ ; λ`. This means: if there is a close square bracket on the input, there must be an open square bracket on the top of the stack, so you can pop it and push nothing, i.e., λ. This means that any close bracket of either type – compare the rules for `(` and `)` – can only be processed if an open bracket *of the same type* is at the top of the stack.

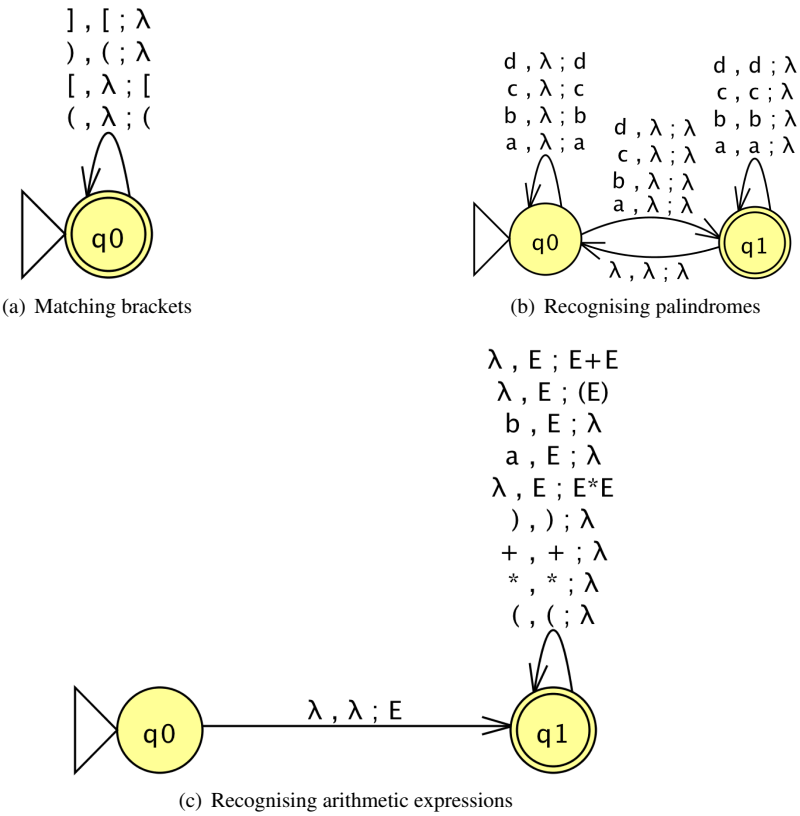
Palindromes

Another task that an FSA cannot perform is palindrome recognition. A palindrome is a string that reads the same when reversed. Simple examples include `mom`, `hannah`. The PDA in 3.3(b) has two states: the first pushes new input onto the stack; the second pops the top of the stack if it matches the input. Here we simplify the alphabet to {`a`, `b`, `c`, `d`} to avoid writing transitions for every character in the alphabet Σ. Why does this work? In the first state, for every symbol we read, we do not pop anything (λ in the middle position of the label on the arc), but we do push the same character as we just read. In the second state, for every character on the input, we pop the matching character off the top of the stack and push nothing (our friend λ).

Can you explain why there is a transitions back to the 'pushing' state from the 'popping' state? Why is this PDA nondeterministic?

Soap-opera recogniser

Not illustrated here, since the JFLAP simulator we use doesn't easily represent big alphabets and whole words: you could create a PDA that is capable of recognising soap-opera plots of the following form: '*Lee's father's cousin's sister kidnapped Cheryl's mother's cousin.*', '*Glen's mother's sister's cousin's father loves Maggie's brother.*'. To start the PDA we push several symbols onto the stack. These symbols determine which transitions are applicable in the processing state. Some of the transitions push other symbols onto the stack, others simply pop the symbols off. The last symbol on the stack is the full stop, which marks the end of the sentence. Popping this symbol off leads to the accept state.



(a) Matching brackets

(b) Recognising palindromes

(c) Recognising arithmetic expressions

Figure 3.3: Several examples of PDAs.

We use several shorthand notations:

$$\begin{aligned} act &= \{framed, kidnapped, blackmailed, drugged, loves, killed\}. \\ nam &= \{Barker, Lee, Maggie, Glen, Steve, Agnes, Nandipha, Vusi\}. \\ rel &= \{father, mother, sister, brother, cousin\}. \end{aligned}$$

Recognising arithmetic expressions

Since PDAs are capable of matching brackets they can also recognise arithmetic expressions. Fig. 3.3(c) shows a PDA capable of recognising expressions such as: `a+a`, `b*(a+a+a)` and `(b+b)*(a+a)`. We use the trick of pushing a symbol that is not in the language onto the stack to determine which transitions are applicable. When several transitions are applicable, we rely very heavily on the non-determinism of the machine to choose the correct transition[3].

The last example is not so easy to understand straight from examining the PDA; more complex examples are better described by a grammar. As with FSAs, a particular class of grammar can be shown to have equivalent expressive power to a PDA: a *context-free grammar*.

3.2 Context-Free Grammar

A context-free grammar is less restricted than a regular grammar, with the only restriction being that only a single non-terminal is allowed on the left-hand side of all rules. The right-hand side can contain

[3]It is possible to parse expressions of this form without using non-determinism, which matters for reasons of efficiency in compilers; here we just want to show that it is possible for a PDA to recognise these expressions, which FSAs were not able to.

any combination of terminals and non-terminals. By only allowing a single non-terminal on the left-hand side of a rule, we are able to expand a non-terminal without having to consider any symbols next to it (i.e. we do not consider the context of the non-terminal).

In the example below, we consider a context-free grammar for the simple algebraic expressions considered in the previous section:

```
Start ↦ Expression
Expression ↦ ( Expression )
Expression ↦ Expression + Expression
Expression ↦ Expression * Expression
Expression ↦ a
Expression ↦ b
```

We make the claim that any context-free grammar can be recognised by a PDA; nondeterminism is a property of the grammar as well as of a PDA so if the grammar is deterministic, so is a PDA that recognizes it. Focus for now on how the PDA works; we will return to nondeterminism.

In the beginning, an `E` is pushed onto the stack. This is directly equivalent to the start rule (assuming that `E` is short for `Expression`) and determines which transitions are applicable. In general the stack contains non-terminals (*Stack Symbols*) and terminals (*Input Alphabet*), which have yet to be processed.

1. For every rule that maps a non-terminal to some other combination of symbols we have a transition, which does not read any characters, pops off the applicable non-terminal, and pushes the combination of other symbols.

2. If there is a terminal symbol on top of the stack then it is popped off; provided it matches the symbol on the input tape.

If there are several possibilities we use non-determinism to choose the correct option. If going either way has the same result, the grammar and hence the PDA is unambiguous.

In this manner the PDA will slowly work through the input string, using only rules from the grammar. If the end of the input is reached and there are no symbols on the stack then the string is accepted. This corresponds to a correct derivation from the grammar.

Because there are straightforward transformation rules to convert a context-free grammar to a PDA, this means that a CFG is no more powerful that a PDA.

A similar argument shows that the language of any PDA can be represented by a context-free grammar. This is more technical in nature, however, and will not be covered in the course. The interested reader is advised to consult Brookshear [1989] for more details.

Putting these two arguments together shows that a PDA and a CFG represent the same class of languages.

For programming languages, it is important that a grammar be *unambiguous* – i.e., there is only one way a particular string can match the grammar. Deterministic PDAs are always unambiguous; it is possible that a nondeterministic grammar (and hence the matching PDA) is also unambiguous. All of this was well understood by the 1970s [Hopcroft and Ullman 1979] – as we said at the outset, this course is about the good stuff that doesn't change.

This `Expression` grammar above is ambiguous – it can either interpret an expression like `a + b * a` the way you would expect with multiplication first or the "wrong" way with addition first, depending how the nondeterminism is resolved. A real programming language should define such constructs unambiguously. One way of doing so is to define higher-precedence operations with another layer of rules, such as in this revised grammar.

```
Start ↦ Expression
Expression ↦ Term + Term
Expression ↦ Term
Term ↦ Term * Factor
Term ↦ Factor
Factor ↦ a
Factor ↦ b
Factor ↦ ( Expression )
```

With this version of the grammar, if you match the `a + b` part of `a + b * a` first using the rule

`Expression ↦ Term + Term`

then there is no further expansion that allows the `* a` to match. If however you use the rightmost `Term` in the following

`Term ↦ Term * Factor`

we have grouped the sub-expression containing the multiply as `Term * Factor` so it will be treated as a unit and hence prioritised over the addition.

3.3 Limitations of a PDA

Since the possible languages of PDAs are the same as the possible languages of context-free grammars, we can find their limitations by looking for languages that do depend on the context. The language consisting of the strings: $\{abc, aabbcc, aaabbbccc, \ldots, a^n b^n c^n\}$ is context-sensitive since we can only add another ab to our string if we add a c several characters away. This means that a PDA cannot recognise this language.

In the next chapter we examine simple computational systems that are able to recognise this language. This is achieved by modifying the machine so that it is able to write on the tape, which negates the need for a stack.

There is a pumping lemma for context-free languages. As with regular languages, a string that is in the language can be split but instead of splitting three ways with one part that can be pumped, the equivalent lemma for context-free languages splits a string from the language five ways with two independent parts of the form x^n and y^n that must both be "pumped" at the same time. As with the regular language version, you can show a that a language is not in this class (context free) using proof by contradiction. But you cannot use this lemma to show that a language *is* context-free. If you look at the above example, you will see it has three parts that are repeated n times – this hints that the pumping lemma for context-free languages, starting with such a string, would "pump" it to an example that no longer fits the language definition.

We will not look into this further – but if you want to venture into the theory of languages, this is something you will need to know.

Exercise 3.1

Extend the palindrome PDA so that it is able to discard punctuation such as spaces, colons and apostrophes. This should make your new PDA able to recognise much longer palindromes such as: '`a man, a plan, a canal:panama!`'

Exercise 3.2

Create a PDA that recognises soap operas then convert it into a context-free grammar that can generate all the soap-opera plots. Simplify the problem by using short symbols to represent words. Implement this grammar in your favourite programming language and randomly generate a few sentences. How

many have actually happened on a soap opera? (Consult someone 'knowledgeable' if you do not watch soap operas.)

Exercise 3.3

Actually the soap-opera recogniser is even simpler than a PDA, it is possible to recognise soap operas with only an FSA.

- Simplify your context-free grammar from the above exercise into a regular grammar.
- Now convert your simplified grammar into an FSA.

Exercise 3.4

Convert the non-deterministic PDA that recognises simple arithmetic expressions into a deterministic PDA. (This means that at no point should more than one transition be applicable.)

Exercise 3.5

Write a context-free grammar that generates *all* strings with twice as many '`a`'s as `b`'s.

Exercise 3.6

A grammar is said to be ambiguous if there are two ways of deriving the same string. In the ambiguous grammar below the following string has two possible derivations:

```
if a then if b then c else d
```

Find both derivations. What implications does this have for most computer languages? Can you rewrite the grammar to remove the ambiguity as we did for the expressions example?

```
Statement ↦ if Statement then Statement else Statement
Statement ↦ if Statement then Statement
Statement ↦ a
Statement ↦ b
Statement ↦ c
Statement ↦ d
```

Chapter 4

Turing Machine

This chapter defines the Turing Machine (TM) as designed by Alan Turing in 1936. It is a simple machine, yet a surprising amount can be done with it. We explore the equivalent grammar (known as a *context-sensitive grammar*). A simple programming language is also considered, which turns out to be surprisingly powerful as well. These results lead us to state the Church-Turing thesis.

4.1 What is a TM?

4.1.1 Definition

A TM was initially proposed as a model of human computation by Alan Turing. In the original model Turing envisaged [Brookshear 1989]:

> "...that the human could only concentrate on a restricted portion of the paper at any time and, in turn, the collection of marks found on this portion of paper could be considered collectively as a single symbol...Turing argued that when considering a particular section of the paper, the human mind could either alter that section or choose to move to another section. Which action would be taken and the details of that action would depend on the symbol currently in that section and the human's state of mind. As with the number of symbols, Turing reasoned that the human mind was capable of only a finite number of distinguishable states of mind...To keep the availability of paper from restricting the power of the model, Turing proposed that the amount of paper available for the computation be unlimited.

Using this model as a skeleton for designing our automaton we arrive at the model depicted in Fig. 4.1, which consists of:

1. **Input Tape:** We imagine the input tape as unlimited in either direction. At any step the automaton can choose to move one step left, move one step right, or stay where it is. At the beginning of the computation the input tape is marked with any necessary input, and blanks are assumed in unused cells (depicted in these notes by #[1]).

2. **Input Alphabet:** A set of symbols from which the input will be constructed.

3. **Tape Symbols:** An extra set of symbols that the machine can use to help process the data. This helps separate inputs or mark a position to return to later.

4. **Initial State:** The state in which the TM is initially started.

[1] Except when we reproduce examples from JFLAP, which depicts a blank cell as a small square, in which case we are free to use hashes as tape symbols.

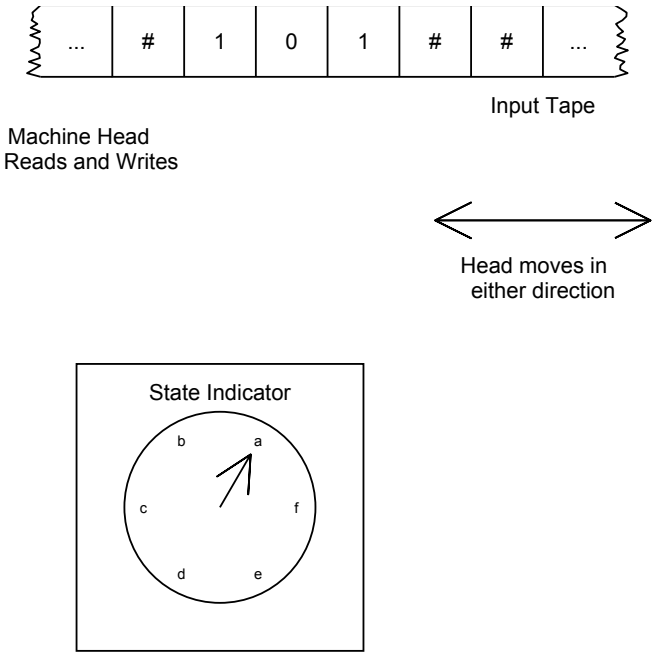


Figure 4.1: Conceptual model of a TM

5. **Halt States:** Typically we define two states which signal that the machine has stopped processing. The accept state indicates that the input has been accepted, while the reject state indicates that the input was rejected. In our diagrams we will assume that the reject state is implicitly the error state and therefore not shown.

6. **Transition Function** For every state and every possible symbol read there must be a clear action to be performed. This action consists of two choices: write a new symbol drawn from either the input alphabet or tape symbols and move either left or right. To concisely specify the transitions we will adopt the notation given in Fig. 4.2.

4.1.2 Misbehaving TMs

In the previous chapters on FSA and PDA we were guaranteed that for any finite input, these machines would terminate. This was because at each step the machine's head would always advance by one step[2]. Since TMs can move backwards and forwards, it is easy to create machines which get stuck in an infinite loop. A simple example of such a machine would move one step right regardless of the symbol read. Since the input tape is infinite, this machine never terminates.

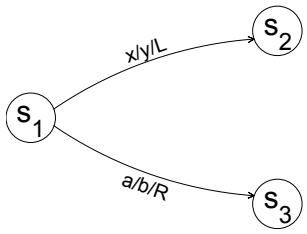
If it can be proved that a TM will always terminate then the TM is said to **decide** the language. If no proof is found then a TM can only **accept** the language. (This does not mean that there is no proof, it might just be that the proof has not yet been found.)

4.1.3 Example TMs

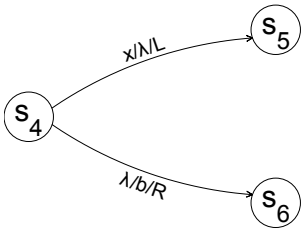
Detecting Palindromes

The TM in Fig. 4.3 is capable of *deterministically* deciding palindromes. This is in contrast to the pushdown automaton for the same language, which was non-deterministic. To keep it simple, we use a two-character alphabet, $\Sigma = \{0, 1\}$. The machine starts with its head at the leftmost character. It then deletes the character (replaces it by a blank) and searches for the rightmost character. If these two

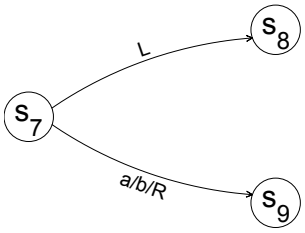
[2] A nondeterministic PDA may have to explore a lot of variations but the number is finite.



(a) Reading an `x` in s_1 will write a `y`, and the head moves left. Reading an `a` will write a `b`, and move right. Reading any other symbol will transition to the error $Halt$ state.



(b) Reading an `x` moves the head left and it does not write anything. All other symbols get overwritten with a `b`, and the head moves right.



(c) For all characters other than an `a` (which will be replaced by a `b`, with a move to the right) the machine will not write anything, and move one step left.

Figure 4.2: Notation for depicting transitions in a TM: *symbol read/symbol written/direction to move*.

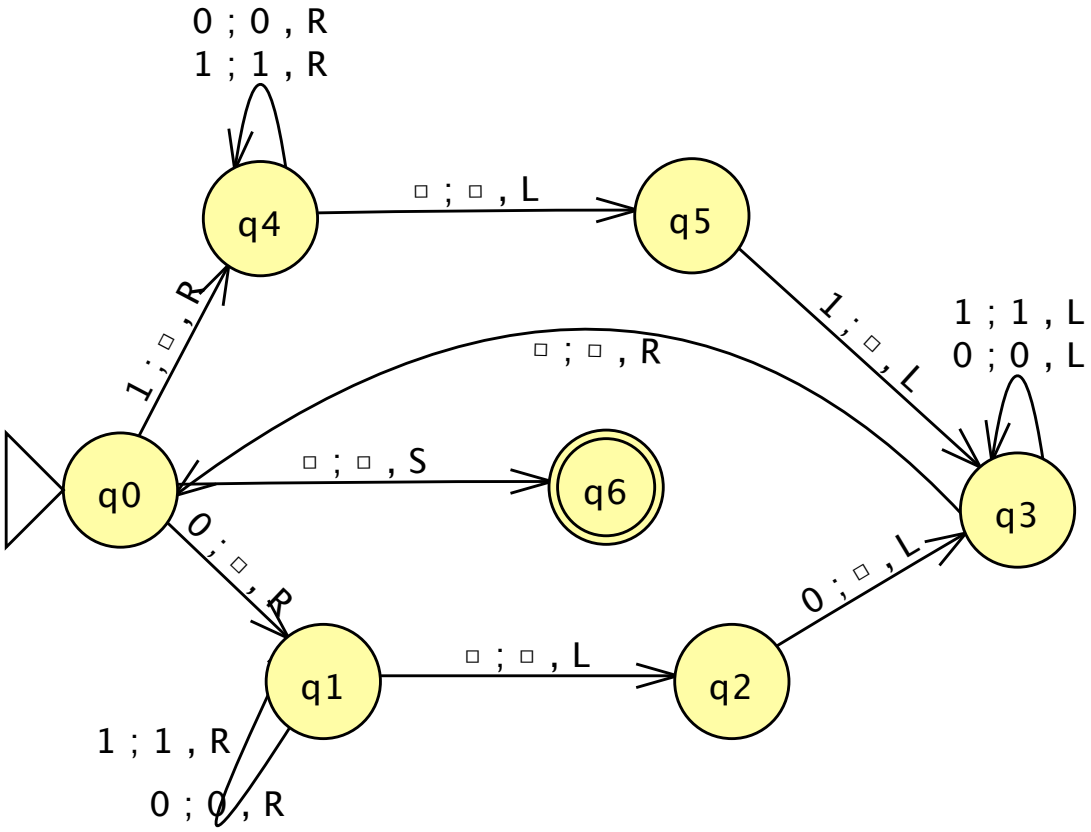


Figure 4.3: Palindrome TM. This one is created in JFLAP so blanks are depicted as small squares, not as #.

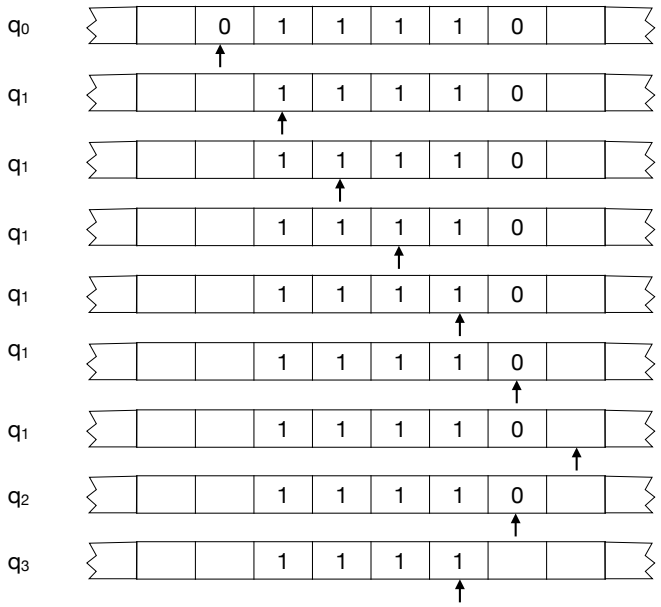


Figure 4.4: Deleting letters to decide a palindrome ($\Sigma = \{0, 1\}$).

characters do not match (as determined by the state of the machine) then the machine rejects the input. Otherwise the machine deletes the letter and returns to the new leftmost letter. This process is repeated until there are no more letters left in the string when the string is accepted as a palindrome. Note that the assumption is there is an even number of letters; a palindrome with an odd number of letters should end when trying to find a match at the end of the string finds a null string.

The top part of the diagram illustrates the case where the left position being compared with the right position is a `1`; the bottom part is the same logic for matching a `0` at the start with a `0` at the end. In both cases, after finding the matching rightmost symbol, in state q_3 they skip over all alphabet symbols to the left until they hit an empty cell then move right, back to q_0. If the TM finds no more alphabet symbols before hitting a blank in q_0, that means a palindrome (length zero is allowed) has been found.

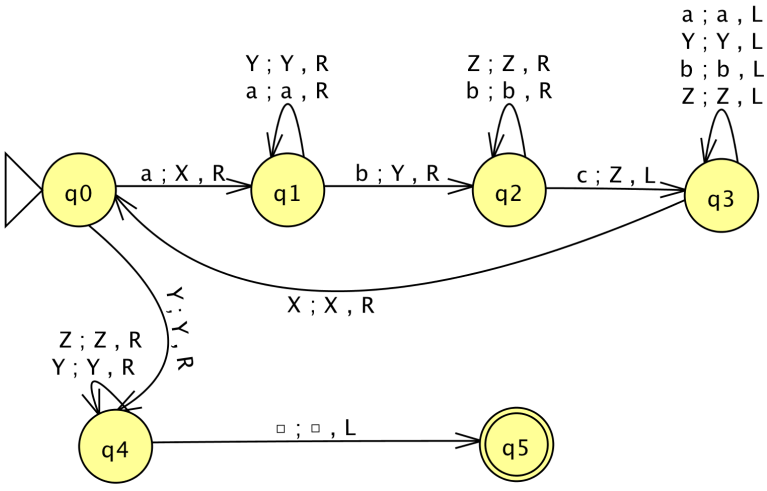


Figure 4.5: The $a^n b^n c^n$ TM. Note that this one is created in JFLAP, which represents an empty space on the tape as a small square, rather than a #.

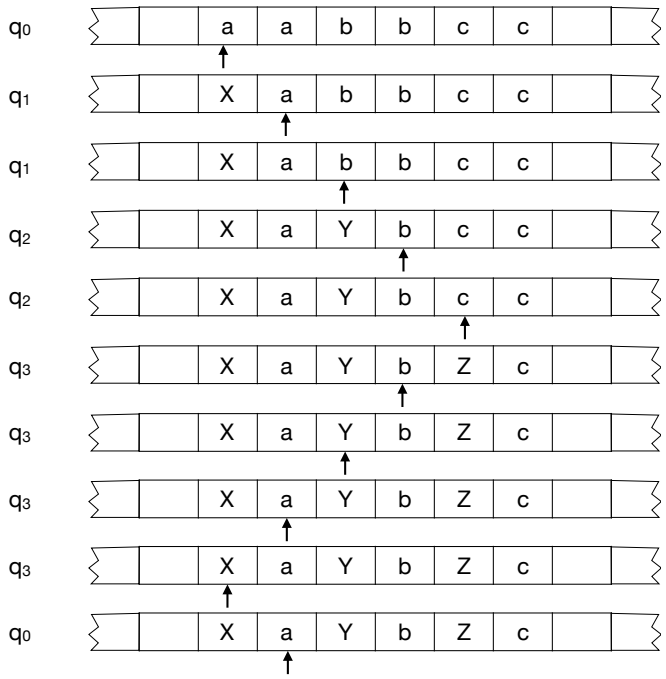


Figure 4.6: Counting letters to decide membership of the set $\{a^n b^n c^n\}$.

Deciding the language $\{a^n b^n c^n\}$

Fig. 4.5 shows a TM that can recognise the language $a^n b^n c^n$, which a PDA could not[3]. This shows that the TM has different capabilities than a PDA, and since it is possible to simulate a PDA on a TM, this means that a TM is strictly more powerful than a PDA.

How does it work? It moves to the right when it sees an `a`, overwriting it with `X`, then skipping ahead to find a `b`, which it overwrites with a `Y`, and ends by replacing the first `c` it finds by `Z`, then moves back to the first position where and `a` was replaced by an `X`.

In a bit more detail, state q_1 skips over any `a` after the one replaced by `X` until it hits a `b`; q_2 skips over any more `b` after the one replaced by `Y` until it hits a `c`, then it backtracks first through any of `a`, `b`, `Y` or `Z` until it hits an `X`, representing the last `a` that was overwritten. This takes it back to the start state. As the transition from q_3 to q_0 has moved to the right, it should find the next `a` – unless we have run out of `a`s. Note the start state q_0 has another transition to q_4 that allows consuming all `Y` and `Z` to the right of the current position and eventually goes to the final state on a blank (a small square in JFLAP notation).

States q_1, q_2 and q_3 need to skip the new tape symbols (`X`, `Y`, `Z`) that have overwritten the alphabet symbols; these transitions only apply after the first pass, when alphabet symbols have been replaced on the tape.

Each pass should replace exactly one of each of the original letters by `X`, `Y` or `Z`. Thus, when the last `a` is converted and we reach the rightmost end of the string, backtracking should only find `X`, `Y` or `Z` and there will be no match on `a` at the start state. We confirm that the absence of an `a` at the start means all other letters have been counted off by scanning for `Y` or `Z` after we fail to find an `a` to the right of the last `X` in the start state (q_0).

This TM should fail to recognize a string if any of the `a`s, `b`s or `c`s were not the same number (n).

Some of the steps the machine takes are shown in Fig. 4.6; notice how the letters are rewritten as we go. Once all `c`s are found, it should only be possible to get to the right end of the string (marked by a blank cell to the right) if all of the original letters have been replaced by their new values.

[3]Source: `https://scanftree.com/automata/turing-machine-for-a-to-power-n-b-to-power-n-c-to-power-n`.

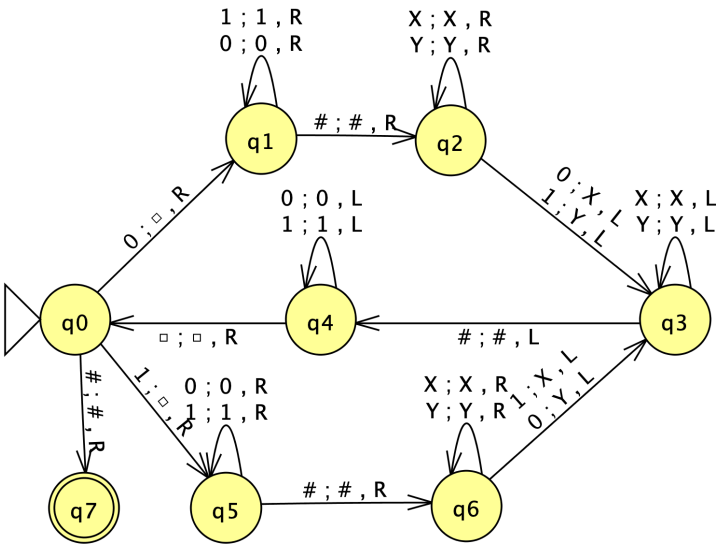


Figure 4.7: This TM is capable of XOR-ing two numbers. The second number is overwritten with the answer represented as 0=X and 1=Y and there is a hash sign separating the two numbers.

The strategy is slightly different here from the palindrome TM: instead of deleting each alphabet symbol, it is replaced by a tape symbol (not in the alphabet). Why? This way, we could recover the original data if we wanted to. The approach is only slightly more complicated: instead of skipping blank cells, we skip cells containing the tape symbols (X, Y, Z). It is a simple exercise to extend the given TM so that, once it has checked for the required language $a^n b^n c^n$, it can scan over the tape again and replace the tape symbols by the original letters from the alphabet.

XOR-ing two numbers

Turing machines are capable of XOR-ing two numbers in binary (see Fig. 4.7). Two numbers in binary are placed on the machine's input. For simplicity both numbers are assumed to have equal length, and the two numbers are separated by a hash sign (#). The second is replaced by the answer on the tape, but with the answer represented as 0=X and 1=Y.

The machine deletes the least significant bits and writes the result in the place of the right-hand number. To read off the answer the final output string must be converted to 0s and 1s (a simple exercise to add to the given TM).

The state of the machine's tape is shown in Fig. 4.8 up to the point where the second digit is about to be processed.

Multiplication

Here we describe the possible design of a TM which accepts strings of the following form: $\{a^i b^j c^k | i \times j = k \text{ and } i, j, k > 0\}$. Once the input string has been received [Sipser 1997]:

1. Scan the input from left to right to ensure that it is a member of $a^* b^* c^*$ and reject if it is not.

2. Return the head to the left-hand side of the tape.

3. Delete an a and scan to the right until a b occurs. Shuttle between the b's and c's deleting of each until all the b's are gone.

4. Restore the deleted b's and repeat stage 3 if there is another a to delete. If all a's are deleted, check on whether all c's are also deleted. If yes *accept*, otherwise *reject*.

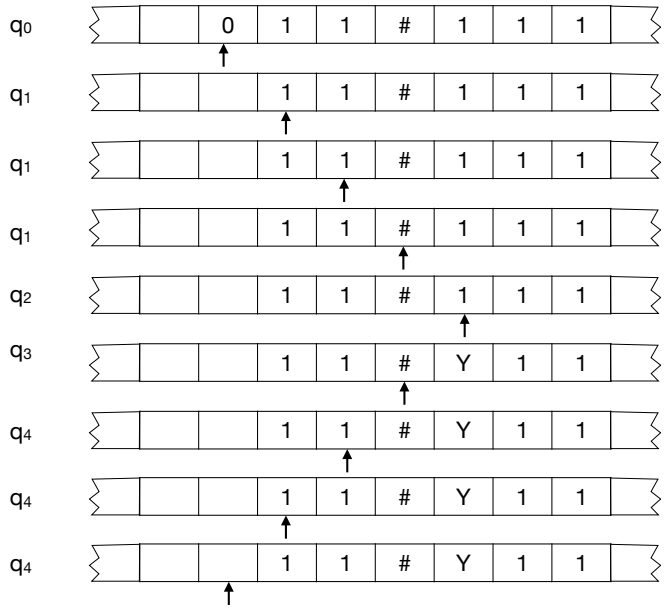


Figure 4.8: XOR-ing two numbers. Note: the final result must be converted from `X` and `Y` to 0 and 1, respectively.

4.1.4 Improving the TM?

How should we improve our TM concept to arrive at a more powerful computational model? By this we mean a model that can solve problems that a TM cannot solve (just as a PDA could not recognize $a^n b^n c^n$ whereas a TM can). Efficiency is not the concern here – clearly it is easy to design a machine that is more efficient than a TM.

It is not at all obvious that we can design a more powerful computational model.

Should we give the machine several input tapes and let it choose which tape to read from next? It turns out it is possible to emulate such a machine on a single-tape TM [Martin 2003].

If we allow the TM to non-deterministically pick its actions, we can still simulate this machine using a deterministic machine, which carefully remembers which decisions it has made and slowly works through all possible alternatives (it is easiest to show this using a three-tape TM which in turn is equivalent to a single tape TM [Martin 2003]).

Allowing random access of the tape (i.e., the TM can now jump to any location it desires) also does not improve its power. If a TM is given k registers storing locations to jump to, then it can be simulated on a $k + 3$ tape TM [Kinber and Smith 2001]. Again this multi-tape machine can in turn be simulated by a single-tape TM.

Remember that these other possible TMs would in all likelihood be much more efficient, just as fancy computers nowadays with pipelining and predictive branching are much more efficient than old computers. However, since there is nothing new that they can compute, hese features do not add to the computational power, which is what we are looking for.

Instead let us compare other computational systems and possibly draw inspiration from them as to the next feature which will improve the power of a TM.

4.2 Context-sensitive Language

Context-sensitive grammars have no restrictions on the form that their rules can take. Any number of terminals and non-terminals are allowed on both sides of the transition. This lack of restrictions makes them very powerful. In fact they are equivalent to TMs in their capabilities (the proof is beyond the scope of these notes, but see Brookshear [1989] for more details).

As an example the following grammar recognises the language of the form $\{a^n b^n c^n\}$. This is a grammar, which a PDA is unable to recognise, yet a TM can:

$S \mapsto \texttt{ab}N S\texttt{c}$
$S \mapsto \epsilon$
$\texttt{b}N\texttt{a} \mapsto \texttt{ab}N$
$\texttt{b}N\texttt{c} \mapsto \texttt{bc}$
$\texttt{b}N\texttt{b} \mapsto \texttt{bb}N$

As further proof that recognising a language is equivalent to performing a computation consider the grammar presented below. It is capable of generating strings such as: '`R1R`$\oplus$`0=1`', '`R0101R`$\oplus$`0011=1001`'. The '`R`'s surrounding the first number represent that it has been reversed (as it was in our TM). In fact it is capable of generating all bitstrings that satisfy the `XOR` operation.

$S \mapsto \texttt{R}\ M\texttt{=}$

$M \mapsto \texttt{R}\ \oplus$
$M \mapsto \texttt{0}M\texttt{0}\ P_0$
$M \mapsto \texttt{0}M\texttt{1}\ P_1$
$M \mapsto \texttt{1}M\texttt{0}\ P_1$
$M \mapsto \texttt{1}M\texttt{1}\ P_0$

$P_0\texttt{0} \mapsto \texttt{0}P_0$
$P_0\texttt{1} \mapsto \texttt{1}P_0$
$P_1\texttt{0} \mapsto \texttt{0}P_1$
$P_1\texttt{1} \mapsto \texttt{1}P_1$

$P_0\texttt{=} \mapsto \texttt{=0}$
$P_1\texttt{=} \mapsto \texttt{=1}$

4.3 Impoverished Programming Language

In this section a very simple programming language is created. So simple that there are only four types of statements: create a new variable, increment it, decrement it, and a while-loop which tests for zero. The variables are also very simple, and cannot represent negative numbers[4]. Their syntax is as follows:

```
int a - Declaration
a++ - Increment
a--- Decrement
while (a!=0){ - While-loop
  //do something
}
```

This is a very basic language, yet we can copy a few of our favourite constructs from other programming languages. To set the value of a variable to zero:

```
while (a!=0){
  a--
}
```

As shorthand we will refer to the above code as `clear`, but remember, it is not a function (method), just

[4]Trying to decrement a variable whose value is already zero, returns zero.

shorthand – it stands for the same code. To copy a value to another variable we can use the following shorthand `b<-a` which represents:

```
//Copy a's value to b
clear temp
clear b
while (a!=0){
  a--
  temp++
}
while (temp!=0){
  a++
  b++
  temp--
}
```

When we need an `if (a!=0) then ..1.. else ..2..` let us use the following code:

```
temp<-a
clear aux
aux++

while (temp!=0){
  ..1..
  clear temp
  aux--
}

while (aux!=0){
  ..2..
clear aux
}
```

That may not look super clear but work through it carefully. IF a is not zero, the first while loop will run exactly once since temp is initialised to a and clearede after the code in the position `..1..` is processed, If you get into the first while loop, `aux` (set to 1 outside the first loop) is decremented so the second loop won't be executed. If it is, it gets executed once, as `aux` is cleared (set to zero) inside the second loop.

Initially it seems as if this computer language will be useless, yet we have been able to define some essential programming constructs from this basic definition. In fact this language has been shown to be equivalent in power to the TM. The proof is beyond the scope of these notes and is not covered here. All of these different approaches to computation are equivalent (in terms of computing power – not efficiency, which is not what concerns us for now).

Knowing that all these models are equivalent means that we can choose any of them, depending which works best in that scenario, to prove theoretical results such as whether it is possible at all to computer a particular function.

4.4 Church-Turing Thesis

In the previous three sections on TMs, context-sensitive languages, and the impoverished programming language, there do not seem to be enough mechanisms for solving complex problems. Yet Turing conjectured in the 1930s that these systems have the same computational power as any possible computational system. So far no-one has been able to prove otherwise, since all proposed models of

computation so far can be emulated on a TM.

It is known as the Church-Turing thesis since a similar theory by Alonzo Church that viewed computation as recursively applying functions to other functions independently arrived at an equivalent conclusion. Church's theory has led to a field of programming known as functional programming.

This does not mean that all attempts to advance programming languages are futile. For practical purposes there is a vast difference between using the impoverished programming language, and a high-level language. Humans are fallible and known to make lots of careless little mistakes. If a programming language helps avoid such mistakes then it makes sense to use it. The efficiency of the impoverished language will also be terrible; there are no arithmetic operations beyond counting. If one wanted to implement 128 bit cryptography in this language, it would take an impractically long time to count up to numbers this large.

Since we appear to have reached the theoretical bounds of a computational system, let us instead focus now on more practical issues. We might be able to prove that our TM can solve the problem, but if it takes more than 10 billion years to halt, it is probably not a practical system. In the next chapter we will turn from analysing the system to analysing the performance of individual problems.

Exercise 4.1

In the previous chapter we made the claim that being able to write on the input tape meant that there was no need for a stack. Give details of how a stack could be implemented on a TM. (Hint: efficiency is not important here.)

Exercise 4.2

Design a TM that can reverse a string. This would allow the `XOR` machine to accept two ordinary numbers and reverse the number itself.

Exercise 4.3

Alter the palindrome of Fig. 4.3 so that it will not accept a zero-length string.

Exercise 4.4

For the TM of Fig. 4.5, enumerate all ways it could fail and check that the TM will actually fail in those cases. Will it work for an empty tape, i.e., where $N = 0$ in $a^N b^n c^N$?

Exercise 4.5

For the TM of Fig. 4.5, add extra states starting from the original TM's final state to restore the letters of the alphabet `{a, b, c}` to the tape.

Exercise 4.6

Add the missing part of the TM in Fig. 4.7 to convert the answer from `X` and `Y` to `0` and `1`, respectively. Now add in an extra check that the second number is not longer than the first (hint: do this before getting rid of `X` and `Y` – if the two numbers are the same length, there should be no left over 0s or 1s on the right. What would happen if the first number was longer than the second number?

Exercise 4.7

Find a corresponding context-sensitive grammar that is able to reverse a string of non-terminals (ensuring that they can only become terminal symbols when they have been properly reversed). This will allow the `XOR` grammar to generate correct strings that are easy to read.

Exercise 4.8

Construct a context-sensitive grammar that can generate the correct addition of any two binary numbers of equal length. This means one should be able to derive strings such as: '`1011+0001=1100`'.

Exercise 4.9

Write short code snippets to perform: addition, subtraction, multiplication and division in the impoverished programming language. Let your code accept the values from variables `a` and `b` and store the answer in `ans`.

Exercise 4.10

Implement the factorial function in the impoverished programming language.

Exercise 4.11

Many interesting computational systems have been shown to be equivalent to a TM. One of the most surprising is Conway's Game of Life. This is a simple two-dimensional world of finite state automata, each only has two possible states 'dead' or 'alive'. The states are updated according to very basic rules:

- Live cells with less than two living neighbours die from loneliness.
- Live cells with two or three living neighbours carry on living.
- Live cells with more than three living neighbours die from over-crowding.
- Dead cells with exactly three living neighbours come back to life.

These simple rules give rise to many patterns, and many different behaviours have been observed. By combining some of these behaviours correctly it is theoretically possible to create a computer capable of performing any computation. Find out how such a game could be turned into a computer. (The Internet provides many implementations of the Game of life, as well as examples of interesting patterns found.)

Exercise 4.12

Initially recursive function theory (on which Church's view of computation was based) conjectured that all computable functions could be composed from simple functions composed in simple ways. However, in 1928 Ackermann found a function that cannot be constructed in such a manner, yet is computable. The definition is as follows:

$$\begin{aligned} A(0,y) &= y+1 \\ A(x,0) &= A(x-1,1) \\ A(x,y) &= A(x-1,A(x,y-1)) \end{aligned}$$

Implement this function in your favorite programming language. What is the biggest value of x for which you can compute $A(x,1)$?

Chapter 5

Computability Theory

Theory of computing deals with problems, and two important questions about those problems: Can the problem be solved using an algorithm and if so, what resources will be required? The first question is dealt with by computability theory and the second is handled by complexity theory.

In an attempt to answer the former question, we classify problems as either tractable (guaranteeing a solution in polynomial time), intractable (solutions to these problems appear to take an exponential amount of time) and undecidable (these problems might never complete). While many of the problems we classify were presented in the introductory chapter we also introduce a few others.

It is important to understand the difference between studying the properties of *problems* and the properties of *algorithms*.

A problem has inherent properties that define whether it is solvable or not and, if so, how efficient a solution could be found. Knowing that does not define a solution; an algorithm does. Knowing an algorithm tells us a problem is solvable and also tells us, if we know how to do algorithm analysis, how efficient it is. But even if we know an algorithm, is it the best we can find? For example, we know that the best sorting algorithms (excluding special cases, like the data is already sorted), take time $O(n \log n)$. But is there a theoretical limit that says we can't do better than this, or should we still look for better algorithms?

We study a range of problems and their properties in this chapter and turn to complexity theory in the next chapter, which takes us a bit closer to algorithm analysis though still mostly in the problem domain.

5.1 Computability

Computability theory deals with whether a problem can be solved at all, regardless of the resources required. In the first part of this course we discussed various computational models as shown in Fig. 5.1. Now we need to deal with the possibility that a problem may not be solveable.

A problem is computable (the term decidable means the same thing) if there exists an algorithm that solves the problem. For an algorithm to be a solution to a problem, it must always terminate (otherwise it is not an algorithm) and must always produce the correct output for all possible inputs to the program. If no such algorithm exists, the problem is uncomputable (or undecidable).

5.2 Undecidable Problems

A problem is **undecidable** if it can be proven that not all inputs will terminate (regardless of the algorithm used). This is disconcerting since it means for some problems we cannot tell if we are making progress towards an answer, or are stuck trying to solve a problem with no solution.

All undecidable problems share the following properties:

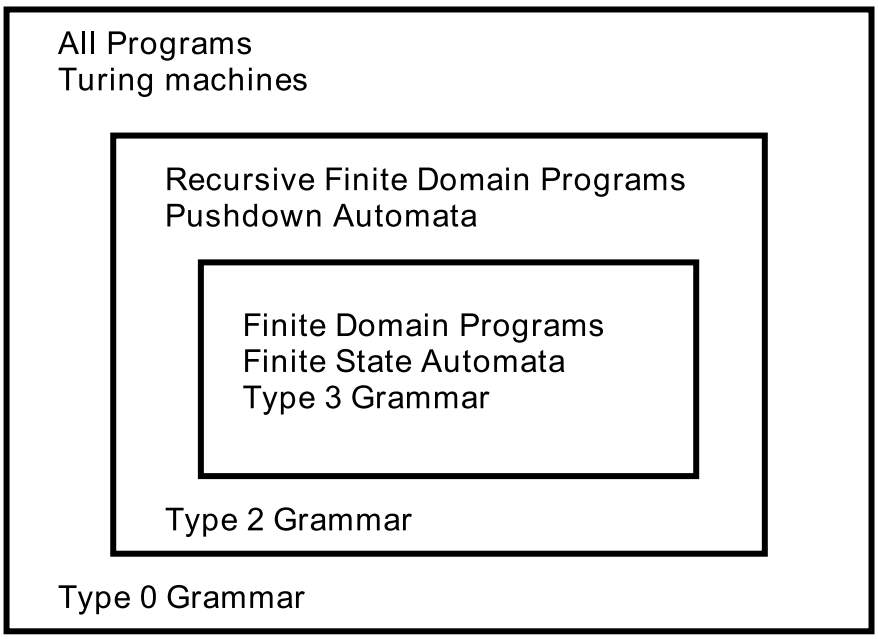


Figure 5.1: Categories of Computability

- They have a possibly infinite search space.
- The worst case bound on complexity cannot be proved.
- As n increases, there are always new cases that are not handled by previous heuristics.

Thus, any algorithm for these problems is infinite in either time or space and the assertion that logic can solve any problem is instrinsically false.

5.2.1 Halting problem

The implication of this problem is that no compiler can test whether your program has an infinite loop. Once the undecidability of the halting problem has been proved, the result can be used to prove that other problems are also undecidable by showing that they can be reduced to the problem in question.

The halting problem was introduced in Chapter 1. MicroNaff is going to have a hard time writing their code verifier since it is impossible to determine whether all programs will halt for a given input. This can be proven by contradiction. Assume that there exists a program that correctly identifies programs that halt for all types of input and always terminates. Call this program '`Halts`'. Now construct a program '`S`' of form given in listing 5.1.

```
1  Program Halts(C,I)\\
2  //Accepts code C and input I and returns true or false\\
3  //Representing whether program C will terminate with input I\\
4  //Note that it \emph{always} terminates.
5  \begin{verbatim}
6  Program S(W)
7   If Halts(W,W)
8     While true
9     {
10      //Infinite Loop!
11    }
12  Else
```

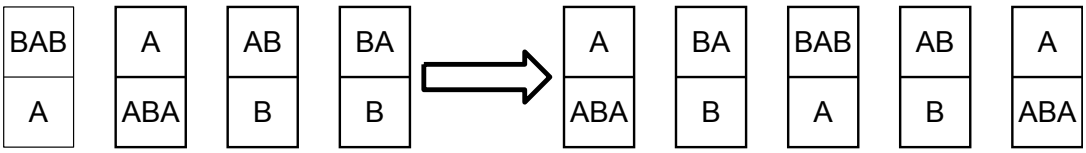


Figure 5.2: A solution to Post's Correspondence Problem: Given a set of dominoes (on the left) is it possible to find a configuration (with possibly repeated dominoes) where the string formed by the top row is the same as the string formed in the bottom row (as shown on the right).

```
13      Return false
```

Listing 5.1: The Halting Problem

Consider what happens when '`S(S)`' is called. This in turn calls `Halts(S,S)`, which *must* return an answer.

- If it returns false (i.e. `Halts` deems `S` to be a program that does not halt when run with an input of `S`) then `S(S)` returns immediately. This clearly contradicts the prediction made by `Halts`.

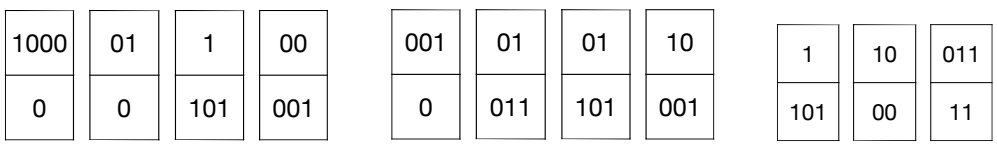
- If it returns true (i.e. `Halts` predicts that `S` is a program that halts when run with an input of `S`) then `S(S)` goes into an infinite loop. Again this clearly contradicts the prediction made by `Halts`.

Since both possibilities lead to contradiction this means one of our assumptions must have been inconsistent. This means our original assumption of `Halts` being a program that *always* halts and *always* returns the correct answer is incorrect. There is *no* such program.

Remember that this is just to prove the existence of a single problem that is undecidable. However, many problems can be shown to be equivalent to solving the Halting problem, which means they are also undecidable. Another well-known example of an undecidable problem is Post's Correspondence Problem.

5.2.2 Post's correspondence problem

In Post's Correspondence problem several dominoes are given [Linz 2001]. Each domino has writing on the top half and the lower half. A sequence of these dominoes can generate two strings, by concatenating the strings of the top halves and doing likewise for the lower halves. The task is to find whether there is a sequence of dominoes that produce identical strings from both the upper and lower halves. An example correspondence problem and a solution is shown in Fig. 5.2. It has been proven undecidable with 7 or more dominoes. This means that in *some* cases, given a set of seven dominoes, it is impossible to tell whether or not the matching process will terminate.



(a) Problem A (b) Problem B (c) Problem C

Figure 5.3: Three instances of Post's Correspondence Problem. Two are solvable; one hard, the other easy and the other has no solution.

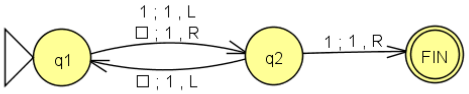


Figure 5.4: BB(2)

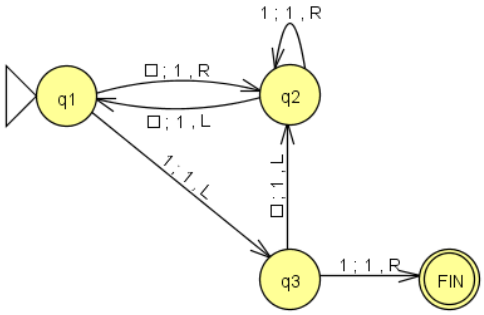


Figure 5.5: BB(3)

Exercise 5.1

In Fig. 5.3 three examples of Post Correspondence Problems are shown. Two of these are solvable, while the third can be shown not to have any solutions[1].

1. Can you find the problem with no solutions?
2. Much harder for one of them at least. . . solve the other two problems!

5.2.3 Busy beaver

An interesting function related to the Halting Problem is the Busy Beaver function. A Busy Beaver is a TM (with n states) that starts with a blank tape and writes the maximum number of ones before halting.

Although the problem is well-defined, $BB(n)$ is extremely difficult to calculate, even for very small values of n. Part of this difficulty is due to the number of possible TMs being exponential in n. It is made worse by the fact that some of the machines do not halt, while others just run for a really long time. Since telling the difference in all cases would be equivalent to solving the halting problem we have to run all possible candidates for a long time.

[1]If you give up, try the solver here `https://www.arnevogel.com/post-correspondence-problem`.

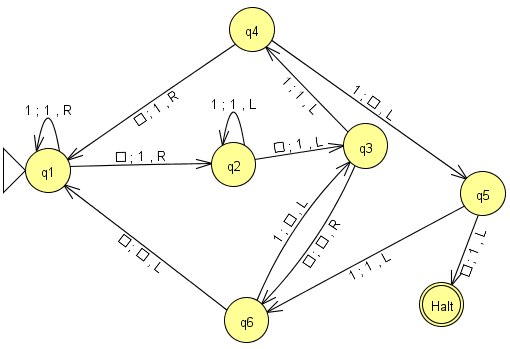


Figure 5.6: BB(6)

Let $BB(n)$ be the maximum number of ones written for an n state machine before halting (note that not all TMs have been tested for $BB(5)$ and $BB(6)$, $T(n)$ be the number of transitions made when writing the maximum number of ones, and $S(n)$ be the maximum number of transitions for any n state machine. Then, we have:

$BB(2) = 4$	$T(2) = 6$	$S(2) = 6$
$BB(3) = 6$	$T(3) = 14$	$S(3) = 21$
$BB(4) = 13$	$T(4) = 107$	$S(4) = 107$
$BB(5) \geq 4098$	$T(5) = 47,176,870$	$S(5) \geq 47,176,870$
$BB(6) \geq 3.515^{18267}$	$T(6) = 7.412^{36534}$	$S(6) \geq 7.412^{36534}$

5.2.4 Wang tiles

This problem is stated as follows. Given a set of tiles, can all $k \times n$ rectangles be tiled using the given set? A valid tiling is one where the colour of adjacent edges match.

It is possible to translate any Turing machine into a set of Wang tiles, such that the Wang tiles can tile the plane, if and only if the Turing machine never halts. But as the halting problem is undecidable, the Wang tiling problem is also undecidable.

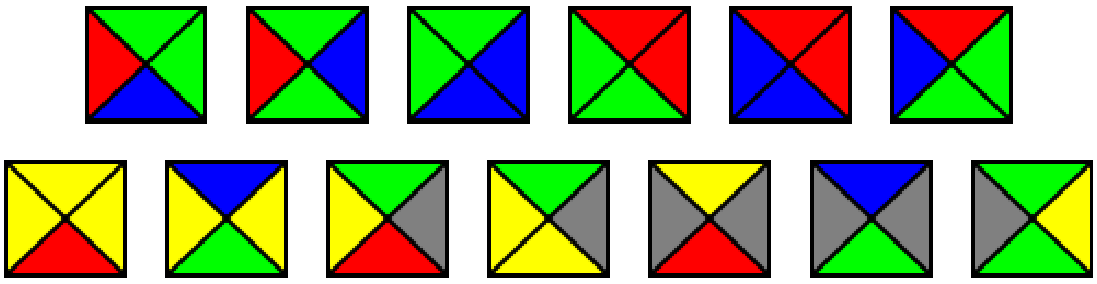


Figure 5.7: Set of Wang Tiles

Chapter 6

Complexity Theory

Having seen some examples in the previous chapter of problems that are undecidable, we know seek to classify tractable and intractable problems. While many of the problems we classify were presented in the introductory chapter we also introduce a few others. We define the complexity classes $\mathcal{P}$ and $\mathcal{NP}$ (see Fig. 6.1), and discuss whether $\mathcal{P} = \mathcal{NP}$, which is an open issue in Computer Science today.

Complexity theory deals with the relative computational difficulty of computing functions and solving other problems.

As noted in the previous chapter, we are primarily concerned here with analysing problems, not algorithms: if a theoretical analysis shows that it is not possible to find an efficient algorithm, we can save ourselves the bother of trying to find one. Alternatively: we can change our focus to finding an *approximate solution* – one that breaks the definition of a true algorithm by finding solutions that are not 100% correct.

As with computability theory, an important part of the theory of complexity is showing that different problems fall in the same class, so a result needs only to be proved once and then it applies to a wide range of problems. Since this is an introductory course, we do not cover such proofs but it is important to be aware that this is possible as a problem can be described as being in a particular class like $\mathcal{NP}$-Complete, and knowing this tells you whether an efficient solution is possible.

- Time complexity is the measure of how long a computation takes to execute. On a Turing machine this is counted as the number of moves. On a digital computer, it is the number of instructions executed – or more accurately, elapsed time.
- Space complexity is the measure of how much storage is required over and above the input data. On a Turing machine this is the number of additional tape squares used. On a digital computer it is the number of bytes used for intermediate results.

Both measures are functions of the size of input, n.

6.1 Big-O Notation

You have already used this notation in your Advanced Programming course. Here we give a brief recap of the relevant definitions.

Big-O describes the asymptotic upper bound for the magnitude of a function in terms of another function, usually a simpler function. A more formal definition states that a function $f(n)$ is $O(g(n))$ if given a constant C, $f(n) \leq Cg(n)$, for sufficiently large values of n. Thus, function $g(n)$ is the upper bound of function f. Note, that since we are interested in the worst case performance, complexity is a guarantee that the algorithm will finish in the time given by the Big-O value.

We can also define a lower bound on complexity, written as Big-Omega (Ω), where the inequality is reversed ($f(n) \geq Cg(n)$), and a tight bound on complexity, written as Big-Theta (Θ), where the

functions f and g differ asymptotically by a constant factor ($f(n) = Cg(n)$). In all cases, "Big" means it is a capital letter. We only use Big-O here but it is useful to know the other variants for more advanced studies of algorithms. Often, when we say big-O, we really mean $\Theta(g(n))$ – but even when we genuinely mean $O(g(n)$, we want a reasonably tight bound, i.e., one that is informative, So it is not useful to describe an algorithm that takes time that is a function of $k(n \log n)$ plus lower terms as being in $O(n^2$ even if it is technically true.

The important thing to understand is that all these notations are "asymptotic", i.e., they apply for sufficiently large n. We are not concerned with smaller values of n unless the complexity is a very fast-growing function, like a^n so asymptotic behaviour is useful to characterise complexity.

Since we are interested in the rate of growth as n increases, only the most significant factors are retained and any constant multiplicative factors are removed. For example, $O(2^n + n^{64})$ is given as $O(2^n)$ and $O(32n^2)$ is given as $O(n^2)$.

Exercise 6.1

To test your understanding of the rate of growth, arrange the following in increasing order of complexity: $n, \sqrt{n}, n^2, n!, n^{-1}, n^5, 2^n, log(n), n^{\frac{2}{3}}$

Exercise 6.2

Consider a program that takes n seconds to run where n is the number of items processed. Complete the following table:

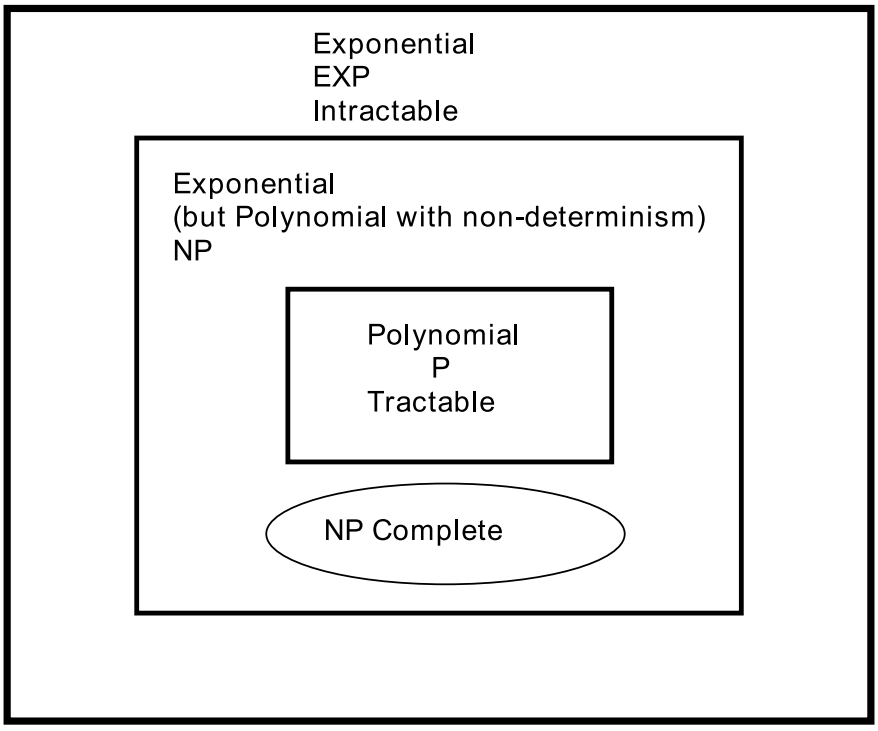


Figure 6.1: Categories of Complexity

	Time for 10 items	**Time for 20 items**	**Time for 40 items**	**Time for 80 Items**
n	10	20	40	80
$2n$				
n^2				
2^n				

Exercise 6.3

Find the complexity of the following factorial program:

```
results = 1
for i = 1 to n
    result = result * i
return result
```

6.2 Graph Theory

Since most of the examples we look at in this chapter deal with graphs, we first cover some graph theory so that we can discuss the solutions with clarity and exactness.

6.2.1 Terminology

Vertex

Often also called a node, a vertex is an abstraction of some item. In the introductory chapter vertices were used as abstractions of cities (in Fig. 1.1) and buildings (in Fig. 1.2). This abstraction is useful since it generally does not matter if we are talking about buildings or cities; what is important are the relations between them.

Two vertices are **adjacent** if they are joined directly by an edge (see below).

Edge

Edges connect two vertices. Edges can be directed, or undirected. If an edge is undirected and connects vertex A with vertex B then it also connects vertex B with vertex A. (Think of this as a two-way street, if you travel from X to Y using only two-way streets then you are guaranteed to be able to retrace your steps.)

If the edge is directed then a connection from vertex A to vertex B does *not* imply that vertex B is connected to vertex A (although this does not rule out another edge connecting them).

In some of the problems we consider, edges are also weighted. This means there is some cost associated with traversing the edge. For our purposes we will only consider nonnegative weightings.

The number of edges attached to a vertex is the **degree** of that vertex.

Graph

A graph consists of a set of vertices and a set of edges connecting them. The edges can be directed (giving a directed graph), or undirected (giving an undirected graph).

A **connected** graph has a path from every vertex to every other vertex.

Tree

A Tree is simply a graph with no cycles in it. This means that for any starting vertex it is impossible to find a path that returns to the starting vertex (**root**) without visiting any vertex more than once. A vertex with no successors is called a **leaf**.

A **connected** tree has links between a node and its parent and children nodes.

The **height** of a tree is the longest path (number of edges) from the root to a leaf. A tree is **perfectly balanced** if all subtrees starting at the same level have the same height; it is **balanced** if no subtree at the same level differs in height by more than 1. It can be shown that a perfectly balanced tree with L leaves has height $\log_2 L$; for an ordinary balanced tree, its height $\geq \log_2 L$.

Bipartite Graph

A Bipartite graph is a graph whose nodes can naturally be split into two subsets, with none of the graph's edges joining vertices in the same subset. This means that all vertices from the one subset only have edges connecting to vertices from the other subset.

Hamiltonian Path

A Hamiltonian path is a path in an undirected graph that visits all vertices exactly once. A graph with a Hamiltonian path is called a traceable graph.

Hamiltonian Cycle

A Hamiltonian cycle is a path that visits all vertices exactly once and at the end of the path is able to return to the initial vertex.

Euler Path

A Euler path is a path that traverses each edge exactly once. How does this differ from a Hamiltonian Path?

Spanning Tree

A spanning tree for a graph is a connected tree that contains all vertices of the graph. Every connected graph contains a spanning tree as a subgraph.

Graph Colouring

Colouring of a graph with k colours assigns one of the colours to each vertex, so that no edge joins vertices of the same colour. A bipartite graph is 2-colourable.The **chromatic number** of a graph is the smallest number of colours that can be used in colouring.

6.2.2 Representing graphs

A typical representation makes use of an adjacency matrix. For a graph with n vertices, we need an $n \times n$ matrix. The matrix contains a 1 in position $[i][j]$ if there is an edge between vertices *i* and *j* (see Fig. 6.2).

6.2.3 Abstract Decision-Tree Proof of Bounds of Sorting

If you study sorting algorithms, you will know that in the general case (no special knowledge of the data, like it is already sorted), the best algorithms seem to be able to do no better than $O(n \log n)$. Is this a hard limit or could a clever programmer come up with something better?

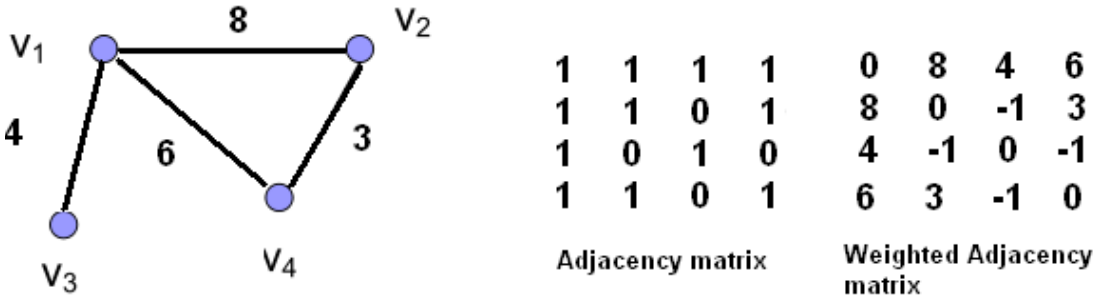


Figure 6.2: Adjacency matrix representations of a graph.

We can study the problem of sorting in the abstract without considering any algorithm by constructing a tree representing every move *any* algorithm that compares pairs of values to sort would make. The first move compares two values and, depending on the outcome, does some other comparison. It keeps doing this until the data is reordered in such a way that is sorted. If the original data is in a different order, the sequence of moves would differ. We can draw a tree that represents all such moves, without being specific about *what* they are. All we know is that each leaf of the tree represents a different reordering of the original data. Why? Because each different ordering of the original data will be a different path through the tree and hence must end at a different leaf.

How many leaves L will this *decision tree* have? We can reorder n values $n!$ different ways so $n!$ is the number of leaves in the tree. The shortest possible path through the tree is $n-1$ since you have to do $n-1$ comparisons at least with a method like this; the height of the tree gives you the worst case. What is the lowest height the tree can have? That would be if it is balanced, so the height of the tree $\geq log_2 L$ and $L = n!$ so we can use Stirling's approximation to factorials to simplify the height to $\approx n \log_n - n \log e$, hence $O(n \log n)$[1].

Does this establish $O(n \log n)$ as the best a sort can do? No. If we are sorting by comparison, this establishes its worst case is $O(n \log n)$ and we can also show that the average path through the tree is $O(n \log n)$. However: if you can find a sorting method that does not involve pairwise comparisons, you can do better – for example, a bucket sort places values directly in a "bucket" representing a value of range of values. A version of this is a *lexicographic* sort, where each bucket represents a single letter of the alphabet. If you sort words k letters long and recursively sort each bucket the same way, the sort does $k \times n$ moves, which makes it $O(n)$.

What we have done here is argue *in general* about sorting algorithms. Just as with the Halting Problem, we did not need to look at a specific algorithm. This is why this analysis is part of complexity theory, not analysis of algorithms, as would be the case e.g. if we anaylzed quicksort or mergesort.

Exercise 6.4

Write out the decision tree proof that sorting by comparison algorithms can't have a worst case better than $O(n \log n)$ in more detail. Explain why a lexicographic sort, despite this proof, is $O(n)$.

Is this really the best we can do? Unless we can make assumptions about the data, yes. Here is an example. Let's say we know that we can arrange the data in order by sorting on it one bit at a time. This works for integers; for 1s and 2s complement, we would need to sort the sign bit differently. Otherwise, if the sorting algorithm splits b-bit numbers based on whether each bit is 0 or 1, the algorithm looks a lot like quicksort, except it runs in time $O(n)$. Why? Because for each bit position, a number is looked at a constant number of times, k and there are b bits. So there are kb comparisons and at most kb swaps

[1]Explained more formally in these Stanford University notes: `https://web.stanford.edu/class/archive/cs/cs161/cs161.1168/lecture7.pdf`

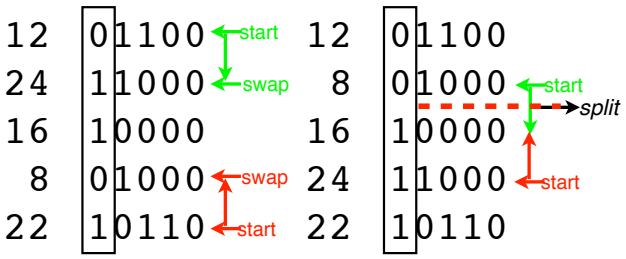


Figure 6.3: Bitsort first pass.

for each number and kbn operations are in $O(n)$.

Is this really better than $O(n \log n)$? After all, to represent n unique numbers, you need at least $\log_2 n$ bits. However, the overheads are very low compared with quicksort and an implementation of *bitsort* can be very fast, at the expense of only working for very specific data types.

Listing 6.1 is C code for bitsort; see if you can work out what it does.

```
1  void bitsort (int data [], int N, unsigned bit) {
2      if (N < 2) return;
3      int start = 0, end = N-1;
4      while (start < end) {
5          if (! (data[start] & bit)) {
6              start++;
7              continue;
8          }
9          if (data[end] & bit) {
10             end--;
11             continue;
12         }
13         SWAP(int,data[start],data[end])
14         start++;
15         end--;
16     }
17     if (data[start] & bit) start--;
18     bit >>= 1;
19    if (bit) {
20         if (start > 0 && start < N)
21             bitsort(data, start+1, bit);
22         if (start < N-1)
23             bitsort(&data[start+1], N-start-1, bit);
24     }
25 }
```

Listing 6.1: Bitsort

The basic idea (so far only considering unsigned numbers) is you start with the high bit, and partition the data between those with the bit set and those where the bit is not set. Those with the high bit set are higher values. Figure 6.3 illustrates the first pass through the data. We start with one pointer or index to the first element, and scan towards the end of the array until it hits a 1 in its current bit position. We then scan the opposite way until we hit a 0. As long as the high and low pointers do not meet, whenever the lower hits a 1 and the higher hits a 0, the numbers are in the wrong order, and so we can swap them.

Once we have sorted on one bit, we can partition the data where that bit switches from 0 to 1, and sort recursively on the next bit, until we run out of bits to sort.

For signed numbers, we need to reverse the order for the sign bit, then treat each partition the same way from then on assuming 1s or 2s complement as bitwise ordering of positives and negatives works the same way, except the sign bit puts numbers in sorted order with a 0 ahead of a 1.

The detail of the method isn't important – just the question of whether we have invalidated the tree-based proof of the $O(n \log n)$ bound on the worst case of sorting by pairwise comparisons. The critical

words are *by pairwise comparisons*. We have not actually compared pairs of values. We have compared bits. This may seem to be a small distinction but it makes a difference, and this insight leads to the broader field of *radix* (another word for base), lexicographic (from the same root as *lexicon*, another word for dictionary) and *bucket* sorts, where we use a known property of the data to place elements directly rather than having to do pairwise comparisons.

6.3 Polynomial Problems

In this section we present the good news. Problems here are considered **tractable** since they are guaranteed to finish in polynomial time. This means that the time it takes is $O(n^p)$ for some constant value p. In some cases p might be very big (say $p = 10$); then the algorithm will be unusable for all but the smallest n. In practical terms, even $p = 2$ grows very fast, and we look for better algorithms. For example, sorting methods with $O(n^2)$ time do not scale well, though they can be faster than more complicated $O(n \log n)$ methods for small n. However $O(n^p)$ is still not as bad as problems covered in the next section, which are thought to have exponential complexity.

Polynomial-time problems are in class $\mathcal{P}$.

6.3.1 Weary student (Shortest path)

In this section we show that the weary student problem has a polynomial-order solution, which should be good news for all students who will be traveling in the holidays. In the example considered here we will assume the student comes from Johannesburg and needs to return. In this solution we will make the simplifying assumption that there are no cycles in our graph. This simplifies our solution since there is no need to maintain a list of previously visited cities (vertices).

The solution presented here uses dynamic programming. Dynamic programming is a bit like recursion in that we solve sub-problems to solve a bigger problem. Unlike recursion, we solve sub-problems bottom-up, which creates the possibility of reusing partial solutions as we go along. To calculate the shortest path from Makhanda to Johannesburg we first calculate the shortest paths from:

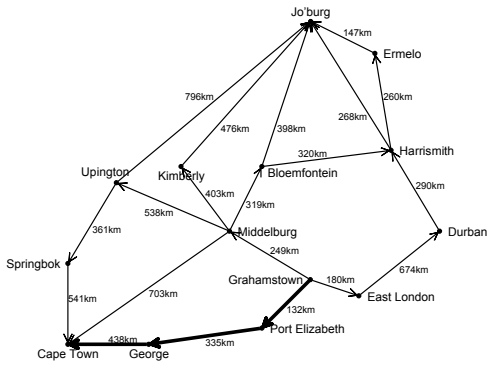
- Gqeberha to Johannesburg,
- East London to Johannesburg,
- and Middelburg to Johannesburg.

Once all of these shortest paths are known then it is trivial to find the shortest path from Makhanda; add the distance to get to each of the cities to the shortest distance from those cities. The city with the smallest sum represents the best route to go, and the sum represents the distance you will have to travel.

In more formal terms, given a weighted directed graph, containing no cycles, we wish to find the shortest path from vertex S to D. Let:

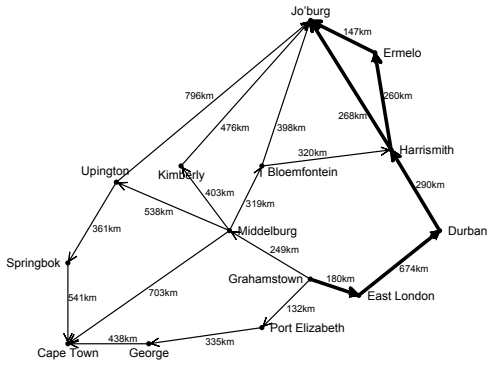
- the set of vertices be $\{v_i\}$ where i is numbered from $1..n$.
- the set of edges form the set$\{e_j\}$ where j is numbered from $1..m$.
- $from(e_j)$ be the function that returns the vertex which edge e_j starts from.
- $to(e_j)$ be the function that returns the vertex that edge e_j goes to.
- For efficiency purposes we store intermediate results for other vertices in an array as this ensures they are not recomputed. This array is called $dist$.

Storing intermediate results is typical of a dynamic programming algorithm; if we used recursion, we would repeat a lot of calculations. The pseudocode in listing 6.2 specifies this algorithm more succinctly.



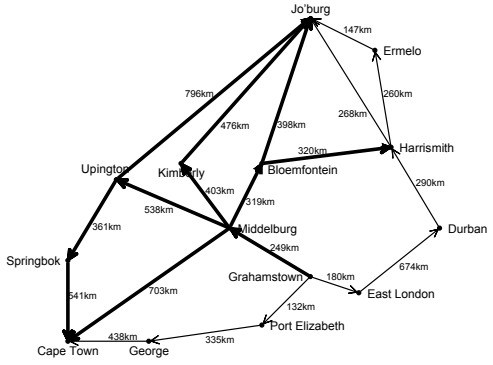
BL=	CT=∞	DU=	EL=	ER=	GE=∞
GHT=	HA=	MI=	PE=∞	SP=	UP=

(a) Going via Gqeberha



BL=	CT=∞	DU=558	EL=1232	ER=147	GE=∞
GHT=	HA=268	MI=	PE=∞	SP=	UP=

(b) Going via East London



BL=398	CT=∞	DU=558	EL=1232	ER=147	GE=∞
KI=476	HA=268	MI=717	PE=∞	SP=∞	UP=796

(c) Going via Middelburg

Figure 6.4: To calculate the shortest distance from Makhanda to Johannesburg, we first calculate the shortest distance from Makhanda's neighbours. Note: we use an old map showing names as Port Elizabeth (Gqeberha) and Grahamstown (Makhanda).

```
1       if (v_i==d) return 0
2       if dist[i] is known return dist[i]
3       ans = ∞
4       for each of the vertices directly connected to v_i
5
6           temp = sd(v_k,d)
7           temp = temp + (edge weight)
8           if (temp < ans) ans = temp
9
10    dist[i] = ans
11    return ans
```

Listing 6.2: Find the shortest distance from vertex v_i to destination d

Now that we have given a formal description of the algorithm, let us trace through it to ensure we understand it fully.

$$\begin{aligned} sd(GT, Jo) &= \min(132 + sd(PE, Jo), 180 + sd(EL, Jo), 249 + sd(Mi, Jo)) \end{aligned}$$

$$\begin{aligned} sd(PE, Jo) &= 335 + sd(Ge, Jo) \\ &= 335 + 438 + sd(CT, Jo) \\ &= 132 + 335 + 438 + \infty \\ &= \infty \end{aligned}$$

$$\begin{aligned} sd(EL, Jo) &= 674 + sd(Du, Jo) \\ &= 674 + 290 + sd(Ha, Jo) \end{aligned}$$

$$\begin{aligned} sd(Ha, Jo) &= \min(268, 260 + sd(Er, Jo)) \\ &= \min(268, 260 + 147) \\ &= 268 \end{aligned}$$

$$\begin{aligned} sd(EL, Jo) &= 674 + 290 + 268 \\ &= 1232 \end{aligned}$$

$$\begin{aligned} sd(Mi, Jo) &= \min(538 + sd(Up, Jo), 403 + sd(Ki, Jo), 319 + sd(Bl, Jo)) \end{aligned}$$

$$\begin{aligned} sd(Up, Jo) &= \min(796, 361 + sd(Sp, Jo)) \\ &= \min(796, 361 + 541 + sd(CT, Jo) \\ &= \min(796, \infty) \\ &= 796 \end{aligned}$$

$$\begin{aligned} sd(Ki, Jo) &= 476 \end{aligned}$$

$$\begin{aligned} sd(Bl, Jo) &= \min(398, 320 + sd(Ha, Jo)) \\ &= \min(398, 320 + 268) \\ &= 398 \end{aligned}$$

$$\begin{aligned} sd(Mi, Jo) &= \min(538 + 796, 403 + 476, 319 + 398) \\ &= 717 \end{aligned}$$

$$\begin{aligned} sd(GT, Jo) &= \min(132 + \infty, 180 + 1232, 249 + 717) \\ &= 966 \end{aligned}$$

Notice how saving the result for $sd(Ha, Jo)$ saved us having to recompute it when we calculated $sd(Bl, Jo)$. In graphs with more edges we can expect this to save us even more effort. Since the array ensures that we never visit a vertex more than once we know that our traversal is linear in the number of vertices ($O(n)$). At each vertex we do work proportional to the number of edges ($O(m)$), this means a rough upper bound on this algorithm is $O(mn)$, which is polynomial[2] and hence tractable.

6.3.2 Cable laying (minimum spanning tree)

To solve the cable-laying problem we need to find what is known in graph terminology as a minimum spanning tree. The 'minimal' refers to the fact that the sum of all the edges found is the minimum possible. 'Spanning' refers to the fact that every vertex is reachable from every other. The 'tree' refers to the fact that there must be no cycles in the solution. If there was a cycle it would be possible to drop one of the edges and still reach all other vertices.

In this solution we present Prim's algorithm. There are other well-known algorithms (such as Kruskal or Borůvka). Prim's algorithm works by picking an initial edge and then growing the tree from the already connected vertices.

To start the algorithm we note that the shortest edge of any vertex will always be part of the minimal spanning tree. As an informal proof: imagine that the algorithm is nearly complete and only has to connect one more vertex. This means that all the other vertices are already connected and we must choose which edge to use to connect this last vertex. Our choice is simple. We pick the shortest edge since there is no better choice.

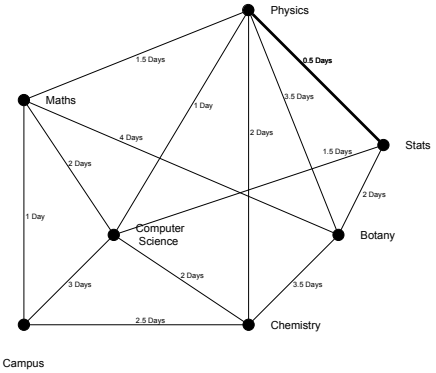
This gives us our starting step; now let us imagine we have constructed some of our tree: how should we pick the next vertex to include? Again it helps if we imagine that all the unconnected nodes have been connected together in another tree and we now seek the best place to connect these two trees. This is simply the shortest edge between the two trees. This suggests that we must find the shortest edge that connects a connected vertex with an unconnected vertex.

A more formal algorithm is given in listing 6.3.

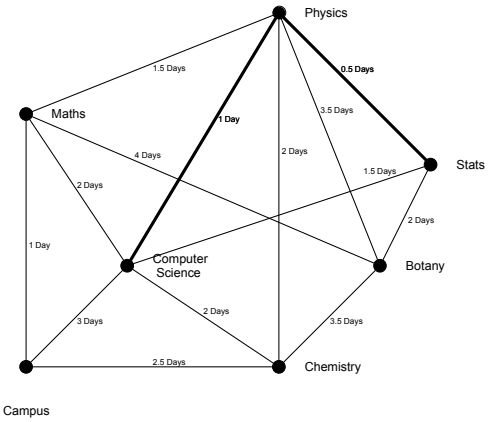
```
1  mst(edges) -- minimum spanning tree, returns list of used edges
2   Initialise boolean array used to all false.
3   Initialise list ans to {}.
4   Pick a random vertex.
5   Add the vertex's shortest edge to ans, set used for both vertices to true.
6   While there are unused vertices:
7    Find smallest edge between a used(v_i) and an unused vertex(v_j).
8    Add this edge to ans and set used[j]=true
9   Return ans
```

Listing 6.3: $mst(edges)$ – minimum spanning tree, returns list of used edges

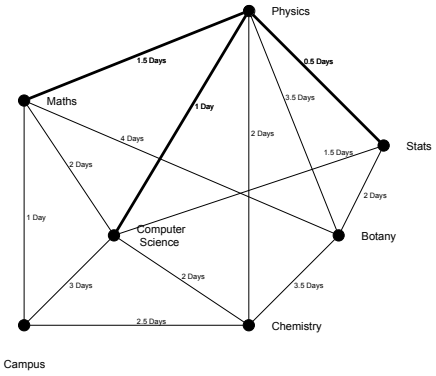
[2]We don't know the exact relationship of m and n but they are related by a constant so $O(mn)$ is in effect $O(n^2)$.



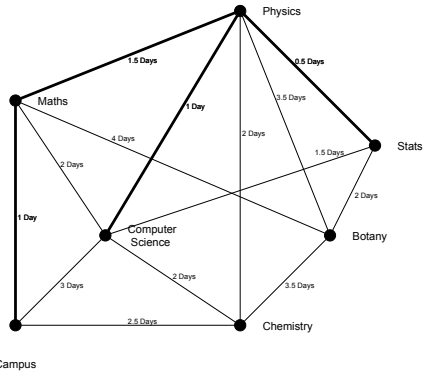
(a) Choose Random vertex (Stats), and find shortest edge = Physics-Stats (0.5 days)



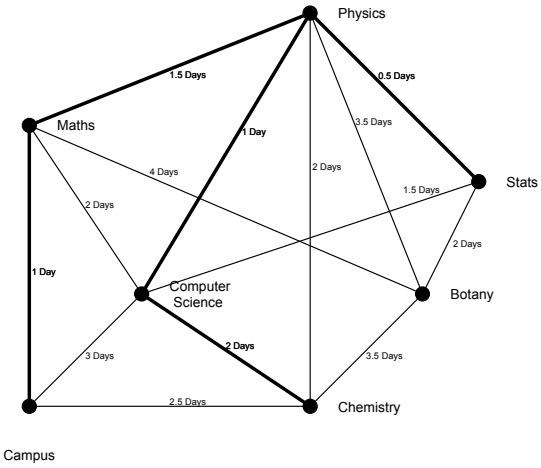
(b) Shortest edge between used and unused vertices = CS-Physics (1 day).



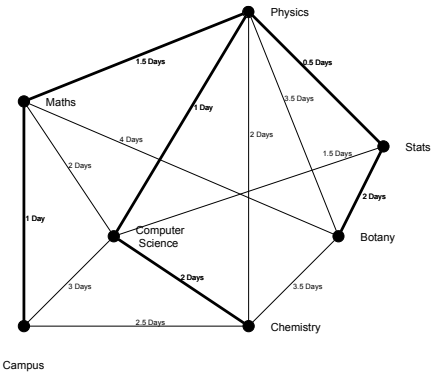
(c) Shortest edge between used and unused vertices = Maths-Physics (1.5 days)



(d) Shortest edge between used and unused vertices = Maths-Campus (1 Day)



(e) Shortest edge between used and unused vertices = CS-Chemistry (2 days)



(f) The finished tree

Figure 6.5: Finding the minimal spanning tree for the science departments.

The solution for the cable-laying problem is shown in Fig. 6.5. After the tree has been constructed it is easy to determine the shortest time in which all departments will have their internet connection restored. Assuming a single team of workers laying the cable, the time taken will be 8 days, which is the sum of all the used edges.

To analyse the complexity of this algorithm we note that we have to add $n-)$ vertices to the tree ($O(n)$). For each addition though we might be forced to search through the entire list of edges ($O(m)$). This means order is again roughly $O(mn)$. This is again polynomial and so considered tractable. Be aware that it is possible to improve the order of this algorithm using more sophisticated data structures, however for our purposes we just need to show that it is possible to find a polynomial algorithm.

6.3.3 New manager

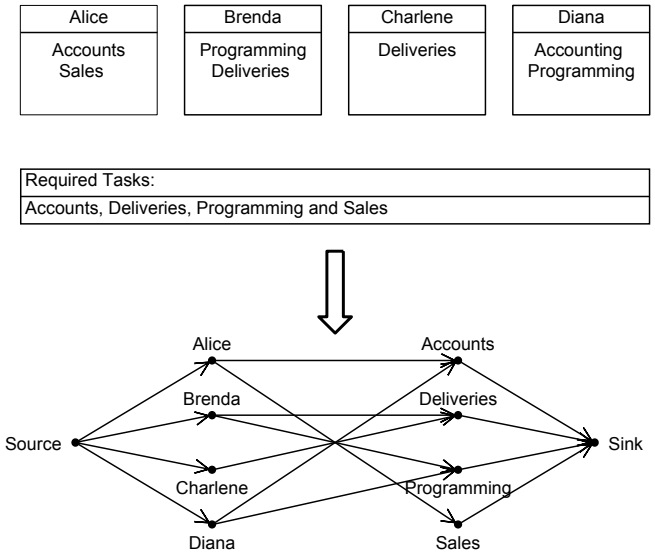


Figure 6.6: Transforming the problem into a graph problem.

Assigning employees to tasks can also be shown to have a polynomial solution. The trick is to turn the problem into a graph. In Fig. 6.6 we create a directed bipartite graph, connecting people to the jobs they are able to perform. A source node, and a sink node are also added, as they simplify the algorithm. The source node connects to all people, while all the jobs are connected to the sink node.

To perform a matching[3] we look for a path from the source node to the sink node. If there is no path then the matching process is over and as many people as possible have been assigned jobs. If we find a path to the sink node, we indicate that the path has been used by reversing all the edges in that path. If a matching is bad, then the reversed direction of the edge allows us to reassign jobs. This can be seen in Fig. 6.7 where bad assignments occur in the first two matchings, and are then reassigned in the last two matchings. We obtain the final assignment by examining the reversed edges, which will point from a job to a person, indicating which person should be assigned that job.

This algorithm is guaranteed to terminate since for every path we find we reverse one more edge from the source to a person. Since there are only a finite number of people, we will eventually run out of possible edges from which to leave the source vertex. This will ensure there are no more paths and the algorithm will terminate.

As an informal argument that this procedure will always result in the largest number of assigned jobs, consider the graph found at the end of this algorithm. There will be no more paths, meaning that

[3]Maximum matching is actually a specialisation of the network flow algorithm. Imagine a network of roads that is used by many cars to get from point A to point B. All the roads can handle different amounts of traffic as some of the roads are highways and some are single-lane country roads. The network-flow algorithm is capable of calculating the maximum number of cars that can use this system of roads.

every edge from the source to a person (i.e. an unmatched person) has no path. This means that every job that an unmatched person could perform has already been assigned. Moreover since there are no paths it also means that one cannot travel from an already assigned skill, to a matched person and find another job that has not been matched. This means that every unmatched person's set of jobs is already performed by someone else and there is no job, to which a matched person could switch, that is not already matched. This is the definition of an optimal matching.

More formally the algorithm can be described as in listing 6.4.

```
1  findpath(s,d)
2   if s==d return true
3   if visited[s] return false
4   $visited[s] = true
5   for each vertex(v_i) that s connects to
6      if findpath(v_i,d) return true
7   return false
8
9  mm
10  Create appropriate graph with sink and source vertices.
11  n=0
12  Initialise boolean array visited to all false.
13  While findpath(source,sink)
14   n=n+1
15   Reverse all edges that make up the path.
16   Reset visited to all false.
17  n represents the maximum number of assignments possible; the individual assignments
       are given by the reversed edges, which are not connected to the source or sink.
```

Listing 6.4: mm – perform a maximal matching

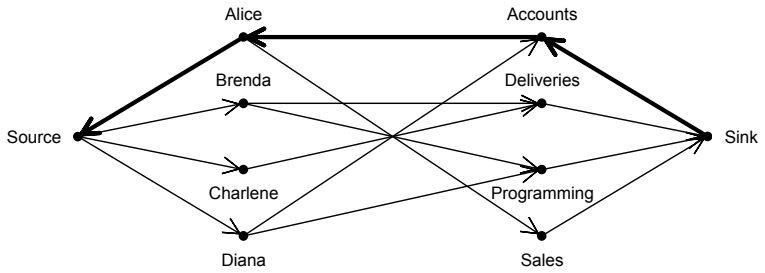
Let us give a rough approximation of the order of this technique, by considering the worst case. Here if k matchings have been made then in the worst case the available path will cover $2(k+1)$ edges. This path corresponds to the previous k matchings all being reassigned (each requiring 2 edges) and traversing the source and sink edge. Since we would have to do this for all of the n nodes, this makes the algorithm at worst an $O(n^2)$ algorithm. This is still considered efficient when compared to the problems presented in the next section.

6.4 $\mathcal{NP}$-Complete Problems

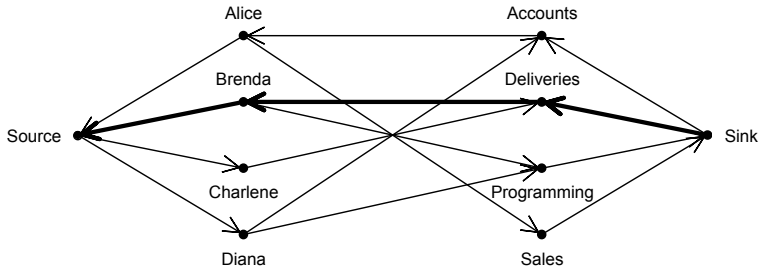
These are problems that are conjectured to have no solution in polynomial time. So far researchers have only been able to find solutions that are exponential in time. However these problems do have solutions in non-deterministic polynomial time ($\mathcal{NP}$). This means that if we had a computer that was capable of non-deterministically choosing the correct decision at every point then these problems could be solved in polynomial time. The problem class known as $\mathcal{NP}$-Complete (NPC) are the hardest problems in $\mathcal{NP}$. If a proof is found that NPC problems can be solved in polynomial time then it will show that all problems in $\mathcal{NP}$ are also in $\mathcal{P}$.

An interesting aspect of these problems is that they have all been proven equivalent to each other. This means that it is possible to transform one problem into another using an algorithm of polynomial order. If we find an efficient (i.e. tractable) solution for one of these problems then it will be possible to solve all the problems by transforming them into the solvable problem, solving them and then transforming them back.

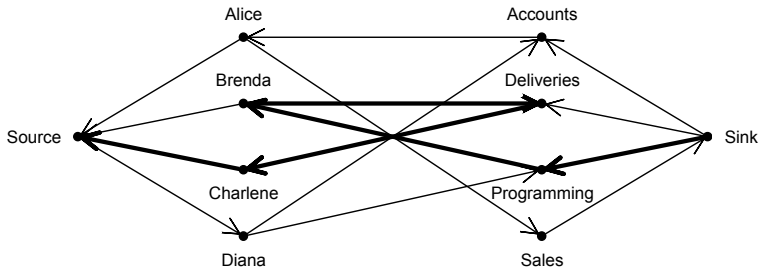
It must be emphasized that the question of proving or disproving whether the class of Polynomial problems ($\mathcal{P}$) is equal to the class of Nondeterministic Polynomial problems ($\mathcal{NP}$) is the largest outstanding issue in theoretical computer science today. It has also motivated a great deal of research on quantum computing, which would be able to solve $\mathcal{NP}$ problems in polynomial time.



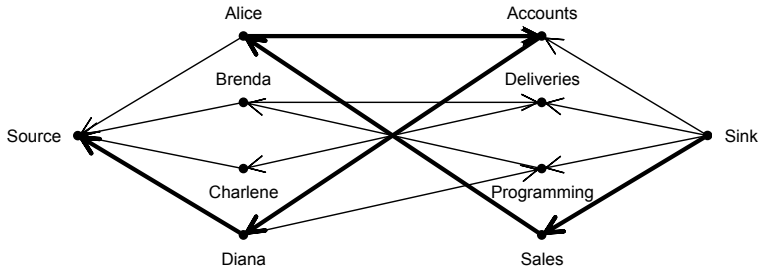
(a) Assigning Alice to Accounts.



(b) Assigning Brenda to Deliveries.



(c) Charlene's only path is through Deliveries and Brenda. This reassigns Brenda to Programming and Charlene is assigned Deliveries.



(d) Diana's only path is through Alice and Accounts. This reassigns Alice to Sales and Diana is assigned Accounts.

Figure 6.7: Using a graph to solve the matching problem.

6.4.1 Traveling salesperson (Hamiltonian cycle)

The traveling salesperson problem described in the introductory chapter is trying to find a Hamiltonian path in a graph representing a road map. Unfortunately the traveling salesperson problem is a problem that arises frequently in real life in such application as: the design of telephone networks, integrated circuits, the programming of industrial robots etc. [Harel 1989].

An exponential algorithm for this problem is easy to find. Just generate all paths and remember the minimum. The order of generating all paths if there are n vertices and roughly k edges at every vertex is $O(k^n)$. Unfortunately researchers have not been able to significantly improve that bound and still guarantee optimality. In some cases **heuristics** (or rules-of-thumb), which seem to work, can achieve acceptable results.

Finding a better guaranteed-optimal algorithm appears difficult as the solution is heavily influenced by the global structure of the graph, yet there appears no simple way of using this global structure when deciding on the next vertex to include in the path.

6.4.2 Satisfiability problem

The Satisfiability problem (SAT) uses Boolean expressions in Conjunctive Normal Form (CNF). A *satisfiable* Boolean expression is one that can be true.

$e = t_1 \cap t_2 \cap t_3 \ldots \cap t_n$

Each t_i is formed by or-ing Boolean variables or their negation ($\sim$):

$t_i = s_0 \cup s_m \cup \ldots \cup s_k$

where the s_i are called literals, and the t_i are called clauses. Or: CNF is ands of ors of variables (including negations of variables). Any logical expression can be converted to CNF form using standard rules of logic like De Morgan's Law, negation of negation, distributive laws, etc.

Problem: Given a satisfiable expression (one where it can be true) e in CNF, find an assignment of values to the variables that will make the value of e true – i.e., show that it is in fact satisfiable.

Examples:

$$\begin{array}{lcl} e = (x \cup \sim x) \cap (x \cup y) & \Rightarrow & x = T; y = F \\ e = (x \cup y) \cap \sim x \cap \sim y & \Rightarrow & \text{No Solution} \end{array}$$

Solution:

- Deterministic algorithm easy – exhaustive approach
  - All possibilities of n variable values = $O(2^n)$

A non-deterministic algorithm can be found that is in $O(n)$.

- An oracle provides the correct value of each variable [Blass and Gurevich 1982]

2-SAT

Given an expression in CNF where every clause contains at most 2 literals, can you find a satisfying assignment? Yes, in polynomial time.

Can draw an implication graph and check it for cycles. Replace each disjunction by a pair of implications: *If node u is true then node v is also*, if there exists an edge from u to v.

The implication graph for the 2-SAT clause given below is shown in Fig. 6.8.

$$(x_0 \vee x_2) \wedge (x_0 \vee \neg x_3) \wedge (x_1 \vee \neg x_3) \wedge (x_1 \vee \neg x_4) \wedge (x_2 \vee \neg x_4) \wedge$$

$$(x_0 \vee x_5) \wedge (x_1 \vee x_5) \wedge (x_2 \vee x_5) \wedge (x_3 \vee x_6) \wedge (x_4 \vee x_6) \wedge (x_5 \vee x_6)$$

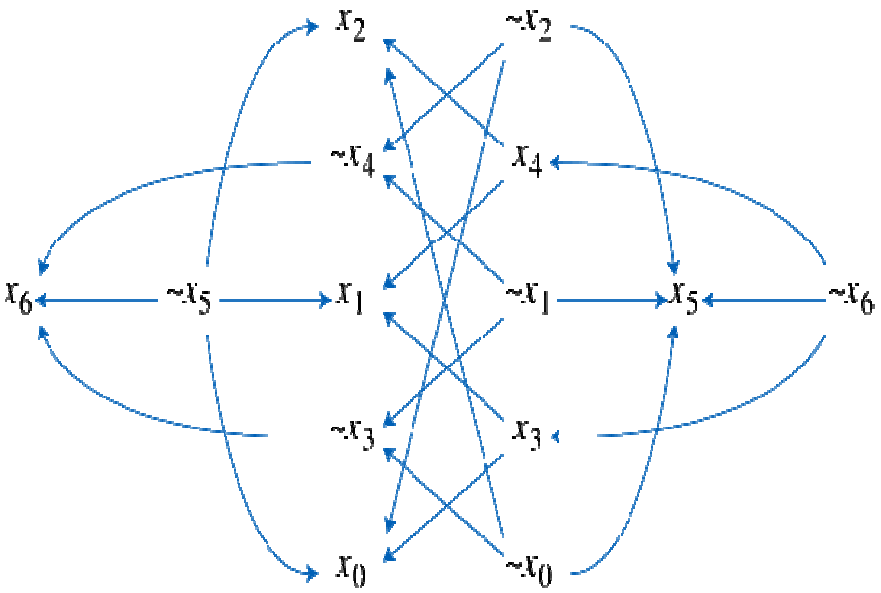


Figure 6.8: Implication graph for a 2-SAT clause.

3-SAT

This problem has historical significance as it was the first problem to be proven $\mathcal{NP}$-complete.The proof uses the Cook-Levin Theorem, which provides a way of converting Turing machines into Boolean formulas if we know a bound on the run time.

Input for the problem consists of a long Boolean expression of the form:

$$(v_1 \vee \neg v_2 \vee v_3) \wedge (\neg v_1 \vee \neg v_2 \vee \neg v_3) \wedge \ldots \wedge (v_{15} \vee \neg v_{17} \vee v_k)$$

One must then find a set of assignments $\{v_1 = true, v_2 = false, \ldots, v_k = false\}$ that satisfy the input expression. This problem has a naïve solution of testing all possible assignments. Unfortunately the number of possible assignments is $O(2^k)$.

One can see that it is easy to verify a given solution, one can simply substitute the values in the expression and evaluate it. This has linear order, and is a lower bound on the complexity of the solution. Unfortunately researchers have only been able to prove an upper bound, which is exponential. By tightening the bounds of our proof we may yet find out if this problem is contained in $\mathcal{P}$ or $\mathcal{NP}$.

6.4.3 3-Colour problem

Problem: Colour a graph with R,G,B so that no adjacent vertices end up with the same colour.

This can be proved to be $\mathcal{NP}$-complete by a reduction from SAT (convert it to a known problem). *Can we make a graph that is 3-colourable iff the Boolean formula is satisfiable?*

6.4.4 Other $\mathcal{NP}$ problems

15-puzzle: Finding any answer is in $\mathcal{P}$. Finding the optimal answer is $\mathcal{NP}$-complete.

Knapsack problem or bin packing: This is a maximization problem based on items with a cost attached. The aim is to fit the maximum number of items into the knapsack, subject to a given constraint, or to use the minimum number of bins.

6.4.5 Sample reductions

1. Traveling Salesperson (TS): given a weighted graph G and integer k, is there a cycle with cost $\geq k$ that visits each vertex exactly once?

   Hamiltonian cycle (HC): given a graph G, is there a cycle that visits each vertex exactly once?

   It is possible to reduce HC to TS: Given input graph G to HP, set the cost of each edge to 1, and consider TS with $k =$ the number of vertices.

2. HC: Given an undirected graph, does it have an HC?

   Direct Hamiltonian cycle (DHC): Given a directed graph, does it have a HC?

   Reducing HC to DHC: Replace each undirected edge (u,v) by a directed edge from u to v and from v to a.

   Reducing DHC to HC: very complicated reduction

Exercise 6.5

Show that exponential order will *always* be greater than polynomial order for large enough n. Find the smallest integer n for which $1.0001^n > n^{10,000}$.

Exercise 6.6

Modify the dynamic programming algorithm so that it can handle graphs with cycles. Analyse the order of your algorithm in the worst case. Experimentally create some graphs and try to approximate the average order too.

Exercise 6.7

Write a program that solves the Travelling Salesperson problem. To simplify the problem just search for any Hamiltonian path, rather than the shortest. Write a method that generates random graphs, with roughly half of the vertices connected to any given vertex. Test your solution on these random graphs for various sizes. Try graphs with 5, 10, and 15 vertices. How long might it take your program for 30 or 40 vertices?

Exercise 6.8

Read up on modifications to the minimum spanning tree algorithm that change the order to $O(m \log(m))$.

Chapter 7

Conclusion

This brief course is logically split into two main themes; discovering the limitations implicit in different models of computation, and differentiating between practical and impractical solutions to problems.

The simplest model of computation – the finite-state automaton – is able to recognise simple numbers and variable names. Trying to add non-determinism to a finite-state automaton did not increase its power as it was possible to use a deterministic FSA with a larger number of states to simulate the non-deterministic FSA. The finiteness of these state machines is their main limitation; as a result they cannot remember an arbitrary number of previous symbols.

To overcome this limitation, we introduced the pushdown automaton. This machine had a memory that can remember as many previously seen symbols as is necessary, using a stack. While there are differences between deterministic and non-deterministic pushdown automata, these differences are not covered as they are more appropriately covered in a compiler course. Instead the limitations of pushdown automata are emphasized. Since pushdown automata are equivalent to context-free grammars, pushdown machines are unable to recognise any language that depends on context.

Thus the next logical improvement was to be able to test for the context of a symbol. This is done by allowing the machine to move both backwards and forwards along the input tape. If the machine is allowed to write on the tape as well then a separate memory is unnecessary since the symbols can be stored on the tape itself. This is a Turing machine, and is thought to have the same computational power as any possible computational system.

The second theme of the course looked at individual problems, and tried to classify their complexity. Several definitions were used; a problem could be classified as *tractable*, which meant that the best solutions were guaranteed to solve the problem in polynomial time. If a problem was *intractable*, this meant that we have only be been able to find solutions with exponential (or worse) complexity. Problems with exponential complexity are impractical, since it could take billions of years to solve reasonably small instances.

Showing that a problem was *undecidable* however meant that there were some inputs for the problem that either would not terminate or would result in an incorrect answer. This is very disturbing since it means that these problems are unsolvable for all cases, regardless of new developments in computers and algorithms.

This gives a much clearer view of the computability of certain problems. If a problem can be solved on a pushdown automaton or a finite-state machine then it is tractable. If the problem requires a Turing machine and does not seem to have an efficient solution there are now several options. Proving the problem equivalent to a known $\mathcal{NP}$-complete problem, will show that currently there is no known polynomial solution and the problem is intractable. If the problem can be shown equivalent to the halting problem, then the problem is undecidable and has no solution that works for all possible inputs.

In the last two cases an optimal solution appears infeasible, and one should instead find heuristics that can produce reasonable solutions. In this sense the theory of computing can be an immensely useful and practical tool for any computer scientist.

References

Aho, A., Sethi, R., and Ullman, J. (1986). *Compilers: Principles, Techniques, Tools*. Addison Wesley.

Blass, A. and Gurevich, Y. (1982). On the unique satisfiability problem. *Information and Control*, 55(1-3):80–88.

Brookshear, J. G. (1989). *Theory of Computation: Formal languages, Automata and Complexity*. The Benjamin/Cummings Publishing Company.

Cohen, D. I. A. (1997). *Introduction to Computer Theory*. John Wiley and Sons Inc.

Gürer, D. W. (1995). Pioneering women in computer science. *Communications of the ACM*, 38(1):45–54.

Harel, D. (1989). *The Science of Computing*. Addison-Wesley Publishing Company.

Hopcroft, J. and Ullman, J. (1979). *Introduction to automata theory, languages, and computation*. Addison-Wesly.

Howell, E. (2017). The story of NASA's real "Hidden Figures". *Scientific American*.

Kinber, E. and Smith, C. (2001). *Theory of Computing – A Gentle Introduction*. Prentice-Hall.

Knuth, D. E., Morris, J. H., and Pratt, V. (1977). Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350.

Linz, P. (2001). *An Introduction to Formal Languages and Automata*. Jones and Bartlett Publishers.

Martin, J. (2003). *Introduction to Languages and the Theory of Computation*. McGraw-Hill Publishers.

Shetterly, M. L. (2017). *Hidden figures*. HarperCollins Nordic.

Sipser, M. (1997). *Introduction to the Theory of Computation*. PWS Publishing Company.

Terry, P. (2004). *Compiling with C# and Java*. Pearson Education.