Theory of Computing 2006

Philip Sterne
Department of Computer Science
Rhodes University

May 1, 2006

Contents

Chapter 1

Introduction

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

1.1 Example Problems

The course will also build on computational complexity which was covered briefly in Advanced Programming. Some example problems which will be tackled are given in the next few sub-sections. Can you spot the problems which are efficiently solvable? Are there problems which might not work in all cases?

1.1.1 The weary student

After finishing exams, the weary student needs to go home to recuperate for the next semester. Unfortunately he's not sure what the quickest way to his home town is. (Roads don't go directly from Grahamstown to every other town.) Given a description of the roads which connect different towns (and their lengths) as shown in figure 1.1 can you find the shortest distance solution?

1.1.2 Cable-laying

After deciding that the science faculty make 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 soon discovers 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 will be of the form shown in figure 1.2, but you'd better hurry, there might be riots soon.

1.1.3 The Traveling Salesman

A salesman has just launched a new range of household cleaning products. Named the Whizzo$^{\mathrm{TM}}$range they could potentially change household cleaning as we know it. To promote this range the salesman has to go on a tour of all

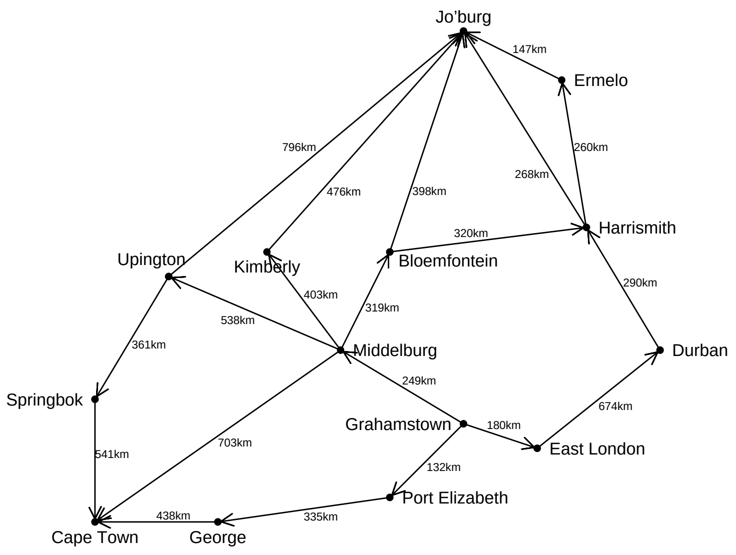


Figure 1.1: Traveling home from Grahamstown. With petrol so expensive it's important to find the shortest distance!

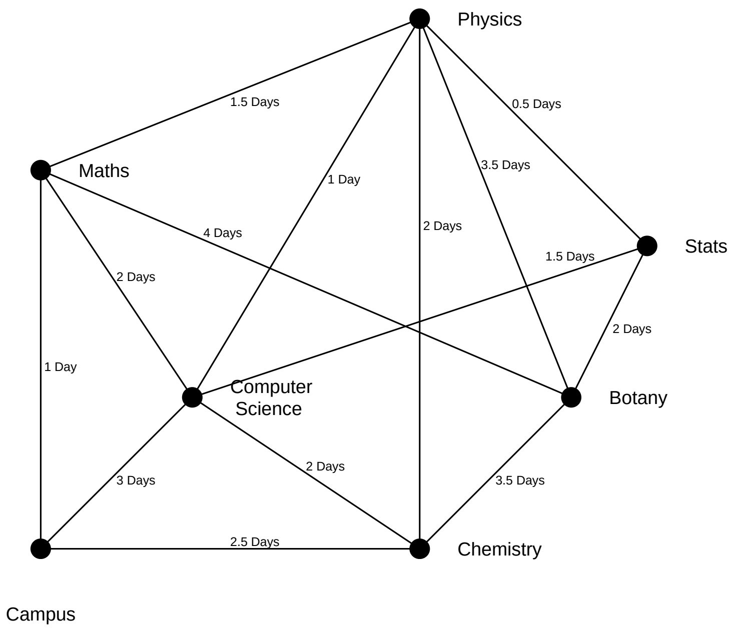


Figure 1.2: How best to connect the new departments to campus? This figure shows the time it would take to lay a cable connecting two departments. (If Department A is connected to campus and we connect Department B to Department A then it is also considered connected to campus.)

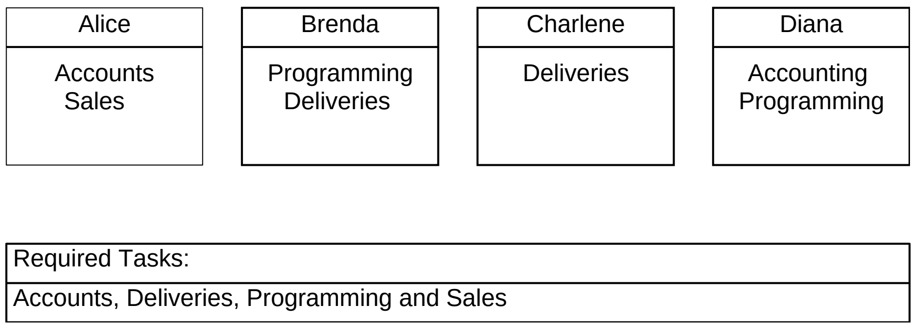


Figure 1.3: Given many employees, each with different abilities, and a set of tasks find the best way to assign the employees to these tasks.

the major cities. Ideally he'd like the tour to be as short as possible and include every city, without visiting it twice. Given a road map showing which cities are connected by roads and the lengths of the roads, can you find this ideal path? (Obviously in some cases there won't be a solution since the roads might force him to visit the same city twice.)

1.1.4 The New Manager

Congratulations! You've just been hired as a manager for an ailing company. After some investigation you realise that many employees are performing tasks which they are not suited to. You decide that the best way to turn things around and make the company profitable again is to reassign employees to tasks which better suit them. Given a description (see figure 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?

1.1.5 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'll instead write a program which will analyse 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 then MicroNaff can rest assured that the customers are obviously imagining the bugs. Can you write a program which will test other programs for infinite loops?

1.2 Conclusion

It isn't entirely obvious as to which of the above problems can be efficiently solved. In fact some problems in the list above can be proven not to work for certain input. Can you spot which? Since the limits of computation aren't entirely obvious this course will first consider simpler computation systems and their limitations in the next few chapters. Using these simpler systems makes it possible to find their limits of computation much more easily. These limitations then suggest what changes should be made to turn them into more powerful systems, and slowly build up to the conceptual equivalent of modern computers.

Chapter 2

Finite State Automata

In this chapter we consider the simplest form of computation - the Finite-State Automaton (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'll look at another way of deriving an FSA's language using a grammar. Finite State Automata's are useful for lexical analysis (which is covered in the compiler's course) as well as string matching.

After defining all the components of an FSA, several examples will be shown to make the concepts more concrete. An extension of finite-state automata is also considered, giving rise to non-deterministic automata. We will also consider what languages FSA's cannot recognise which will suggest how to turn them into more powerful machines.

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 while the machine is in an accept state then the whole string is accepted. An informal representation of an FSA is shown in figure 2.1, and the basic processing is shown in figure 2.2.

A finite-state automaton consists of several entities which need to be defined first(from [Brookshear, 1989]):

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 the calculation of 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 then it will move to the new state determined by $\delta(12, a)$.
4. **Start state** This is the initial state in which the automata starts.

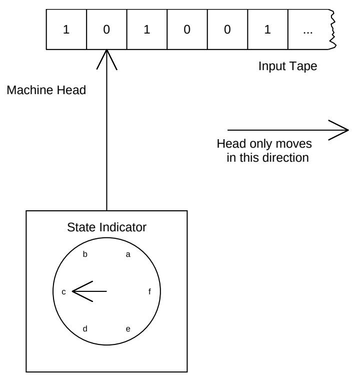


Figure 2.1: A conceptual model of an FSA.

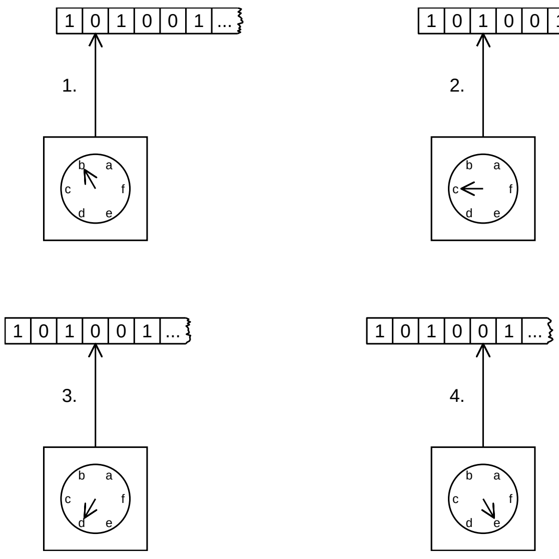


Figure 2.2: As the input string is processed the internal state of the machine changes.

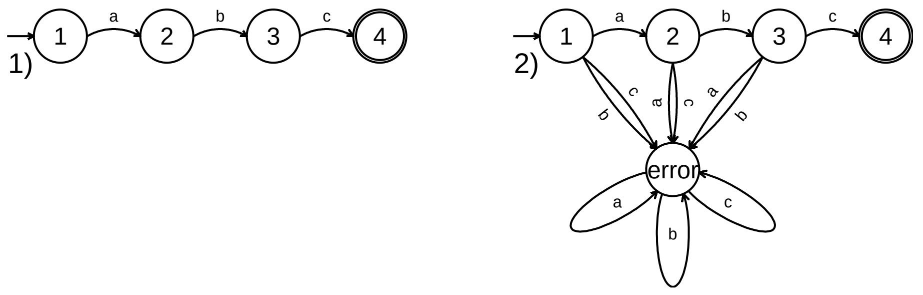


Figure 2.3: If a possible transition is not shown then it is assumed to lead to an implicit error state. This makes the above two FSA's equivalent.

5. **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.

These five items are the only things allowed as part of an FSA and together define it completely. Since it is quite hard to visualise the transition function we normally depict FSA's 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 the accepting states.

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 which leave the FSA in a state from which it might still accept the string. Implicit in our diagrams will be an error state. Any undefined characters for each state will transition to this error state. The error state will not be 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 figure 2.3 for a comparison.

2.1.1 Example FSA's

A Vending machine

In figure 2.4 we see a simple cool 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 cool 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 5c coins? How about if the price of a can was raised to (a more realistic) R4,50?

Recognising numbers

In figure 2.5 we have an FSA which is able to recognise some of the valid `float`'s for the Java 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 which recognises all valid `long`'s? How about `double`'s?

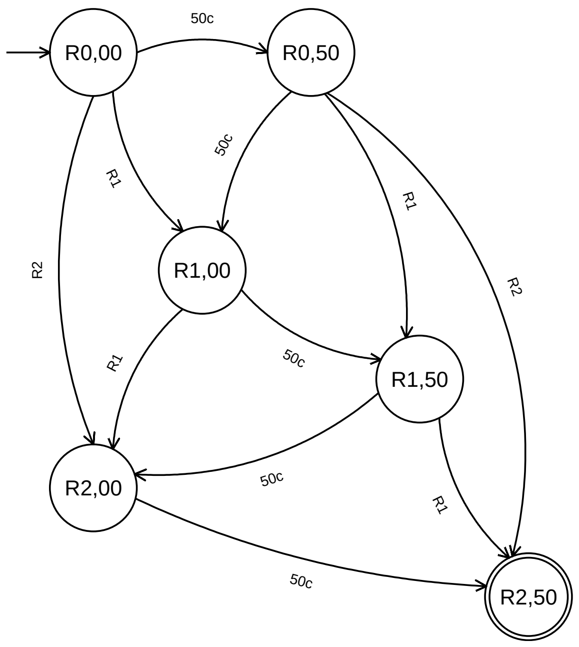


Figure 2.4: A simple vending machine FSA.

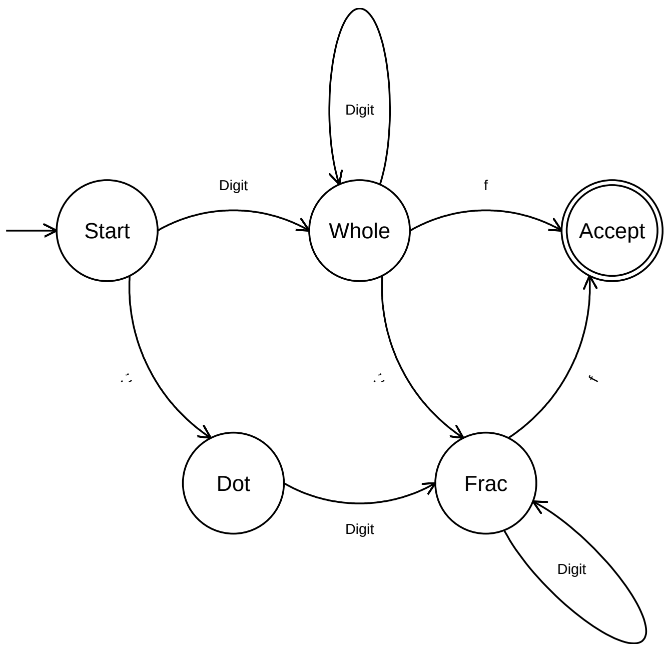


Figure 2.5: An FSA which recognises valid floating point numbers.

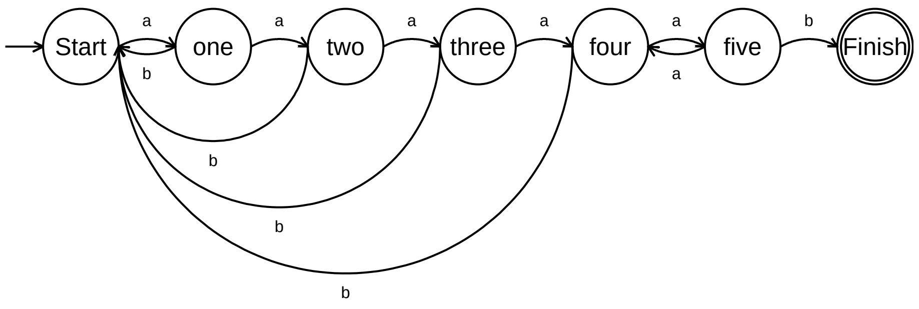


Figure 2.6: An FSA which recognises the string ‘aaaaab’.

2.2 Efficient String Recognition

In this section we’ll consider a powerful application of these automata. If, given a string to search for in a long text most computer programmers will come up with a solution similar to:

```
    public static int find(String f, String longString){
        for (int a=0; a<longString.length(); a++)
        {
            for (int b=0; b<f.length(); b++)
            {
                if (a+b >= longString.length())
                    return -1;
                if (longString.charAt(a+b) != f.charAt(b))
                    break;
                if (b+1 == f.length())
                    return a;
            }
        }
        return -1;
    }
```

If we analyse 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 = "aaaaaaaaaaaaaaaaaaaaaaab"`. Hopefully this will convince you that the worst-case order of this algorithm is $O(nm)$.

Fortunately there is an algorithm [Knuth et al., 1977] which has $O(n + m)$. It relies on creating a finite state automata which can recognise the string in a fast manner. See figure 2.6 which 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 procedure is clearly linear in the length of the text to search (i.e. $O(n)$).

However this method is useless if there isn’t 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 doesn't 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 which 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 didn't read a '`c`') but we could have already started reading '`aba`' of the string we're 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 down to finding the smallest initial string which 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.

2.3 Non-deterministic FSA's

Let's try make our FSA more powerful. To do this we will introduce the concept of non-determinism. This means our Finite State Automaton 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.

How would a non-deterministic automaton (NFA) accept strings? Since we don't 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. If when the string terminates 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 figure 2.7, this NFA will recognise only integers, longs, floats and doubles which are valid in Java. The first transition in the NFA is not deterministic since there are several transitions which expect a number (and two which expect '.'). This means our NFA will start by either magically guessing the correct type of the string or running all the possibilities in parallel.

If the NFSA in figure 2.7 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.

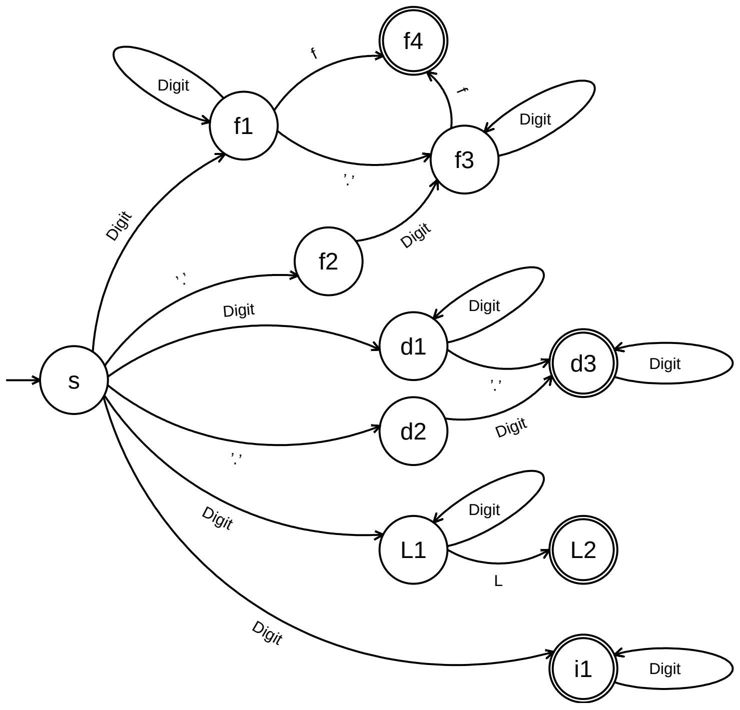


Figure 2.7: A non-deterministic FSA which can recognise all doubles, longs, ints, and floats. Notice how simple deterministic FSA's have been added together to create this NFSA.

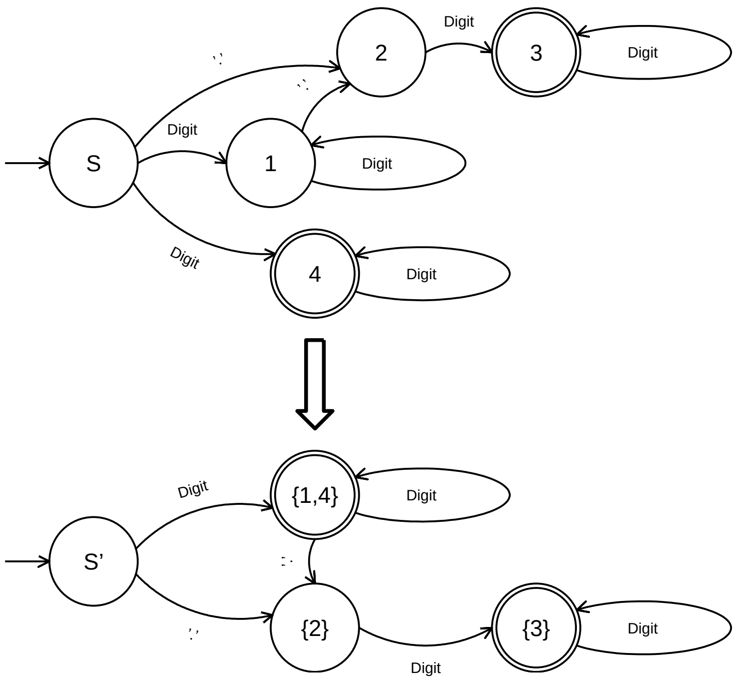


Figure 2.8: Converting a nondeterministic automata into a deterministic one. Since the start state has two transitions for a digit (states 1 and 4), we create a new state which represents having states 1 and 4 as both possibly enabled. As state 4 is an accept state then the new state is also an accept 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 then 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.

Unfortunately it can be shown that for every NFA there is an FSA which 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, it is itself in a state. Since the number of states is finite then the number of possible subsets (or power-set) is also finite. We can represent the different subsets as different states, and the resulting automata is deterministic. This process is shown in figure 2.8.

2.4 Links to a Grammar

It is interesting to examine the set of all strings accepted by a Finite-State automaton. Let's define this set to be the **Language** of the FSA. It turns out that this set can be constructed in a manner entirely different to an automaton. This brings us to the idea of a regular grammar. Hopefully in high school some grammar rules were taught to you. These generally took the form of:

```
<Sentence> ↦ <Noun> <Predicate>

<Noun> ↦ <Article> <Adjective> <Common Noun>
<Noun> ↦ <Proper Noun>

<Predicate> ↦ <Verb> <Noun>

<Article> ↦ a
<Article> ↦ the

<Adjective> ↦ hard-working
<Adjective> ↦ nasty

<Common Noun> ↦ prac
<Common Noun> ↦ student
<Proper Noun> ↦ philip

<Verb> ↦ solves
<Verb> ↦ helps
```

This grammar allows us to derive very informative strings such as "`philip helps the hard-working student`" and "`a hard-working student solves the nasty prac`". In the above grammar we can see that there are symbols such as "`<Noun>`" which never appear in the final string. We will call these symbols **non-terminals** and the symbols which do appear in the final string as **terminals**.

As it turns out there is a simple grammar which will generate all possible strings accepted by an FSA. The grammar has a very restricted form. The rules must be of one of two forms:

```
<Non-terminal> ↦ Terminal <Non-terminal>
<Non-terminal> ↦ Terminal
```

As an example let's consider creating the grammar for the FSA we considered in figure 2.5 which could recognise floating point numbers:

```
<Start> ↦ digit <Whole>
<Start> ↦ ‘.’  <Dot>

<Whole> ↦ digit <Whole>
<Whole> ↦ ‘f’
<Whole> ↦ ‘.’  <Frac>

<Dot> ↦ digit <Frac>

<Frac> ↦ digit <Frac>
<Frac> ↦ ‘f’
```

Where `digit` represents the characters ‘0’-‘9’. It should be evident from studying figure 2.5 that there is a nice one-to-one mapping which turns our FSA into this grammar. 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> ↦ c <State'>`. If the state is an accept state then we add the rule containing a single terminal: `<State> ↦ c`.

2.4.1 Recognising strings or performing computations?

The approach we have taken 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 which are not inherent in general computability. Doesn't this mean we should rather forget about boring grammars and rather consider the limits of the latest and greatest Whizzomatic^{TM}computer which 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 which would recognise strings of the form: ‘12+13=25’, and ‘237+1=238’, then in effect we can say that the grammar is able to perform addition.

2.5 Limitations of FSA's

Such a simple machine cannot hope to do everything, and we run into the FSA's main limitation if we try to design one which can recognise pairs of matching brackets. For my latest computer language I want to read in expressions such as `"((()()))())"` and determine whether or not 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 figure 2.9. 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

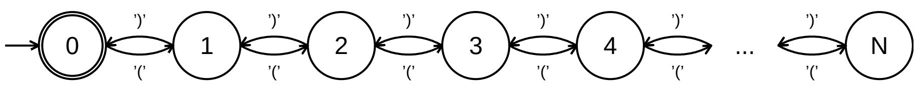


Figure 2.9: An attempt at writing an FSA which can recognise strings with balanced brackets.

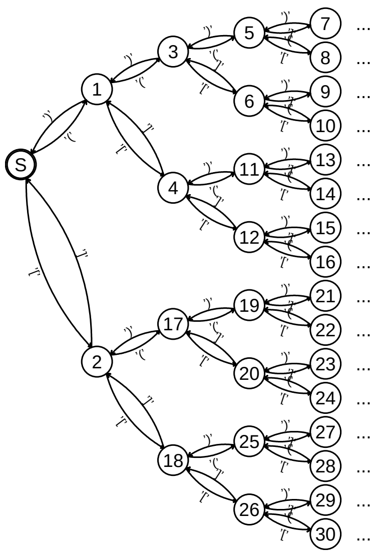


Figure 2.10: Things go badly wrong when trying to match different brackets!

impossible. For any FSA designed it is possible to create a string which is incorrectly handled. If the FSA has 1,000,000 states and then discards any strings which 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 brackets. Consider trying to match the following string "[([[]])()[()]]", this problem would suggest a solution of the form given in figure 2.10. 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 which we can use to remember which brackets we've seen. This leads us into the next chapter on Pushdown Automata's.

Exercise 2.1

What is the purpose of the `Dot` state in figure 2.5 (on page 12)? 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 figure 2.7 and show that the string '`1.2L`' has no possible states, which means the string is rejected.

Exercise 2.5

Turn the NFA in 2.7 into a deterministic FSA. 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 which creates the set of all strings accepted by your automata.

Chapter 3

Pushdown Automata

In this chapter we create a Pushdown Automaton 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 limitations of an FSA's finiteness, and recognise more languages. We will also show that a Pushdown Automaton (PDA) is equivalent to a context-free grammar.

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

3.1 Definition

Conceptually a pushdown automaton can be viewed as a finite-state automaton augmented with a stack. This is shown in figure 3.1. For a more formal definition a pushdown automaton consists of six things:

1. **Input Alphabet** The set of all characters which can appear on the input tape.

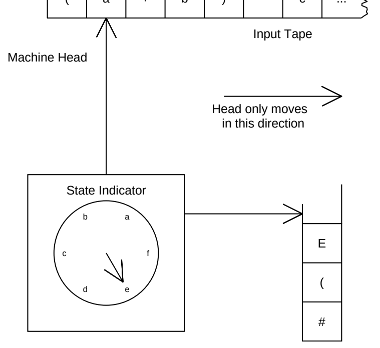


Figure 3.1: A conceptual model of a PDA.

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 the same as an FSA, a pushdown automaton must consist of a finite number of states.

4. **Transition Function** The transition function is now a function which considers the current state, the input character just read, and the character on the top of the stack. It then outputs a new state which the PDA will transition to. It also decides whether to pop a symbol off the stack and/or 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 states which, if the end of the string is reached *and* the stack is empty, then the entire string is accepted.

In contrast with FSA's, pushdown automata have increased power when they are made non-deterministic. As a simple example consider a PDA which detects palindromes (figure 3.3(b)). 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 then 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.

The languages generated by deterministic PDA's 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[1]. 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].

3.1.1 Examples of PDA's

Before we cover the examples we will first explain the notation used when describing a transition. Every arc will be 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 will consist of *any number* of symbols which can be pushed onto the stack. In all these cases if a 'λ' is shown then that action is not performed for that transition.

Recognising Matching Brackets

In figure 3.3(a), is a pushdown automaton capable of matching different brackets. 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, then the PDA goes into the

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

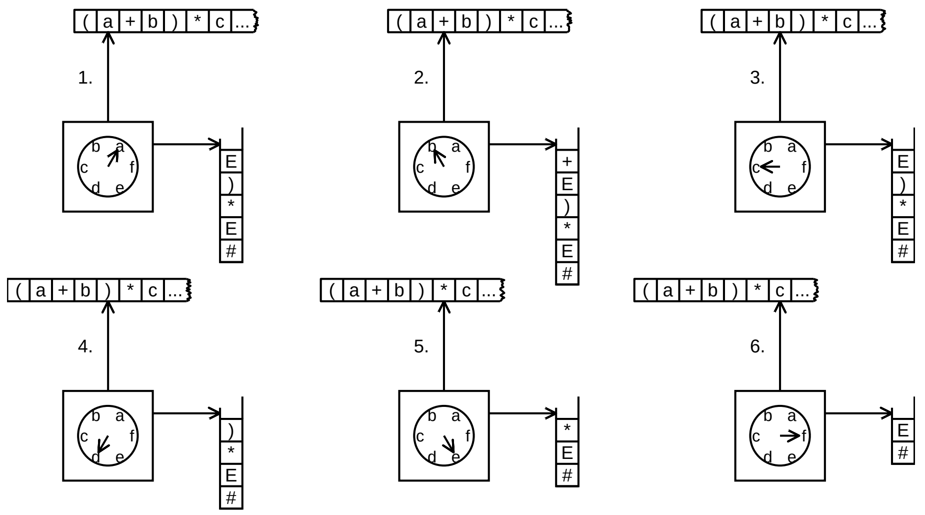


Figure 3.2: As the input string is processed subexpressions are pushed onto the stack and popped off as characters are read.

implicit error state, otherwise the recognition process continues. Notice that in this example it is not necessary to use non-determinism.

Palindromes

Another task which FSA's cannot perform is palindrome recognition (3.3(b)). A palindrome is a string which reads the same when reversed as it does normally. Simple examples include '`mom`', '`hannah`'. Can you tell why there are two transitions from the 'pushing' state to the 'popping' state?

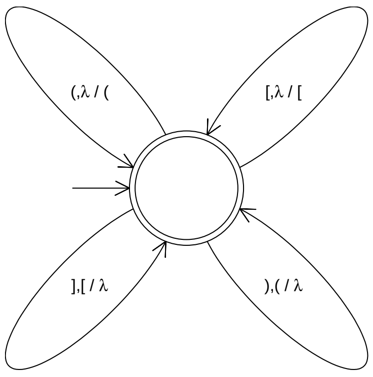
Soap-Opera recogniser

Figure 3.3(c) shows a pushdown-automaton which is capable of recognising soap-opera plots of the following form: '*Lee's father's cousin's sister kidnapped Cherel'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. We use several shorthands:

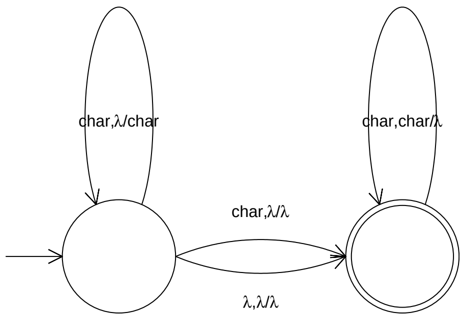
$$\begin{aligned}
act &= \{framed, kidnapped, blackmailed, drugged, loves, killed\}.\\
nam &= \{Barker, Lee, Maggie, Glen, Steve, Agnes, Nandipha, Vusi\}.\\
rel &= \{father, mother, sister, brother, cousin\}.
\end{aligned}$$

Recognising arithmetic expressions

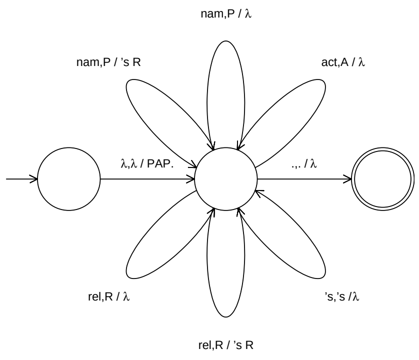
Since PDA's are capable of matching brackets they can also recognise arithmetic expressions. Figure 3.3(d) shows a PDA capable of recognising expressions such



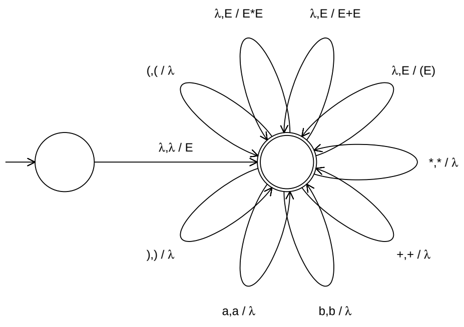
(a) Matching brackets.



(b) Recognising palindromes.



(c) Recognising soap-opera plots.



(d) Recognising arithmetic expressions.

Figure 3.3: Several example Pushdown Automata.

as: '*a+a*', '*b*(a+a+a)*' and '*(b+b)*(a+a)*'. Similar to the Soap opera PDA we use the trick of pushing a symbol onto the stack to determine which transitions are applicable. When several transitions are applicable then we rely very heavily on the non-determinism of the machine to choose the correct transition[2].

3.2 Context-Free Grammar

A context-free grammar is less restricted than a regular grammar, the only restriction is that all the rules are only allowed a single nonterminal on the left-hand side. The right-hand side can now contain any number of terminals, and nonterminals. By only allowing a single nonterminal on the left-hand side of the rules, we are able to expand a nonterminal without having to consider any symbols next to it (i.e. we don't consider the context of the nonterminal).

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 non-deterministic PDA. To back this claim up, consider the PDA for this grammar. In the beginning an `E` is pushed onto the stack. This is directly equivalent to the start rule and determines which transitions are applicable. In general the stack contains nonterminals (Stack Symbols) and terminals (Input Alphabet) which have yet to be processed.

1. For every rule which maps a non-terminal to some other combination of symbols we have a transition which doesn't 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 the non-determinism to choose the correct option.

In this manner the pushdown automaton 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. However this corresponds to a correct derivation from the grammar.

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

[2]It is possible to parse expressions of this form without using non-determinism, but that is left for a course in compilers, here we just want to show that it is possible for pushdown automata to recognise these expressions which FSA's weren't able to.

3.3 Limitations of PDA's

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

In the next chapter we examine simple computational systems which 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.

Exercise 3.1

Extend the Palindrome pushdown automaton 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

Convert the PDA which recognises soap-operas into a context-free grammar which can generate all the soap-opera plots. Implement this grammar in your favorite programming language and randomly generate a few sentences. How many have actually happened on a soap-opera? (Consult someone 'knowledgeable' if you don't watch soap-operas.)

Exercise 3.3

Actually the soap-opera recogniser is even simpler than a pushdown automaton, it is possible to recognise soap-operas with only a finite-state automaton.

- 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 which 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 which 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?

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

Chapter 4

Turing Machines

This chapter defines the Turing machine 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 Phrase-Structure 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 Definition

A Turing Machine 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 figure 4.1, it 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 throughout these notes by '#').

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

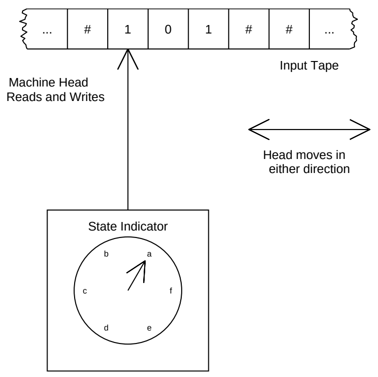


Figure 4.1: The conceptual model of a Turing machine.

3. **Tape symbols** An extra set of symbols which 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 the Turing machine is initially started in.

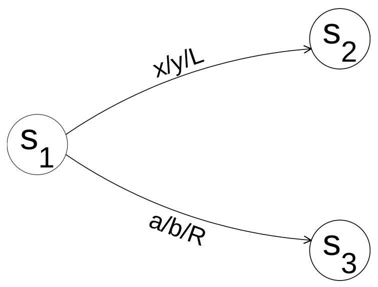
5. **Halt states** Typically we define two states which signal that the machine has stopped processing. The accept state indicates that the input has been accepted, while the reject state indicates that the input was rejected. In our diagrams we will assume that the reject state is implicitly the error 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 figure 4.2.

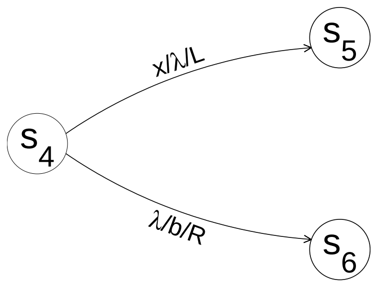
4.1.1 Misbehaving Turing Machines

In the previous chapters on FSA's and PDA's 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. Since Turing machines 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.

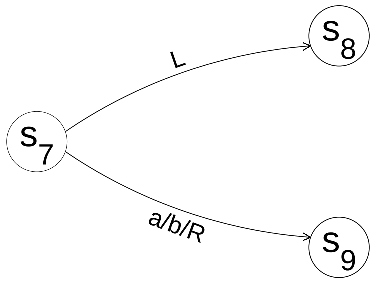
If it can be proved that a Turing machine will always terminate then the Turing machine is said to **decide** the language. If no proof is found then a Turing machine can only **accept** the language. (This doesn't mean there isn't a proof, it might just be that the proof hasn't been found yet.)



(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 doesn't write anything. All other symbols get overwritten with a `b`, and the head moves right.



(c) For all characters other than an `a` the machine will not write anything, and move one step left.

Figure 4.2: Notation for depicting transitions in a Turing Machine.

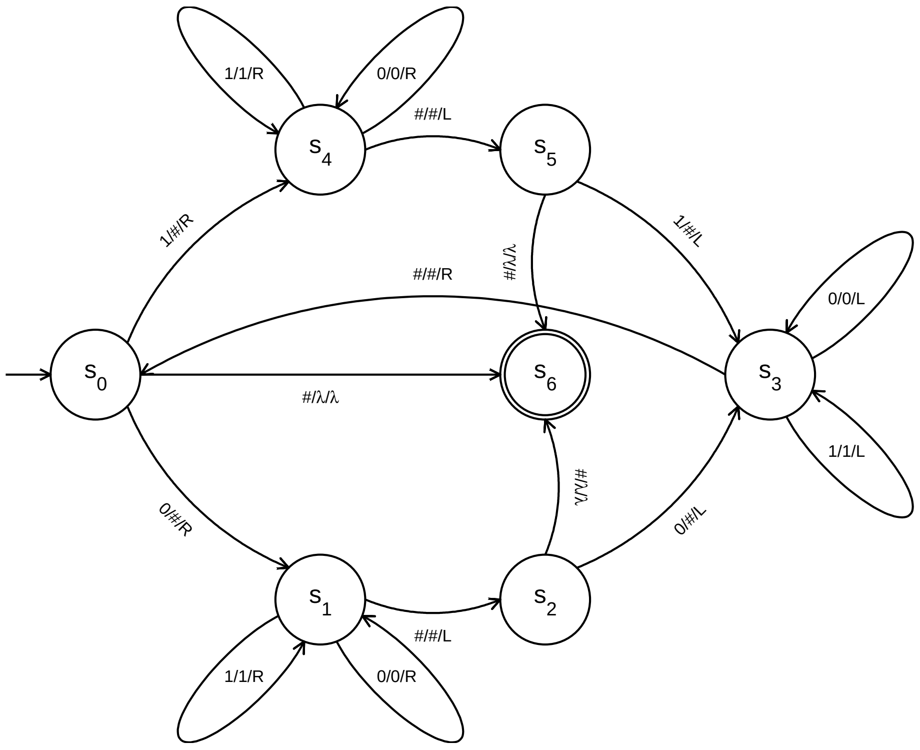


Figure 4.3: The palindrome Turing machine.

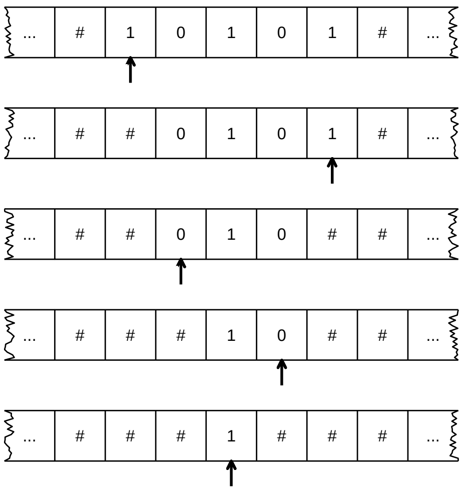


Figure 4.4: Crossing out letters to decide the palindrome.

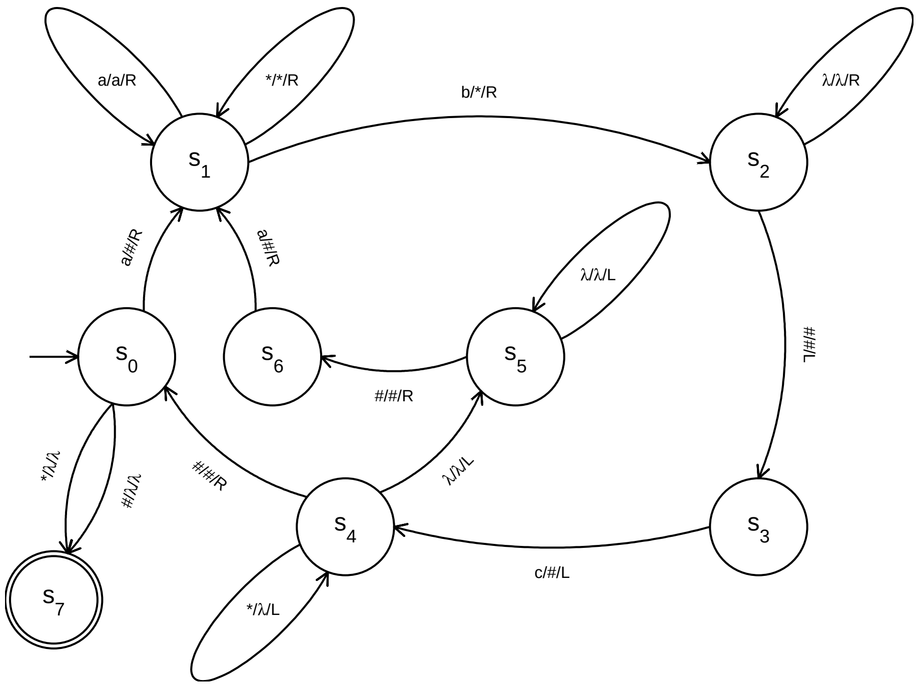


Figure 4.5: The $a^n b^n c^n$ Turing machine.

4.1.2 Examples of Turing Machines

Detecting Palindromes

The Turing machine in figure 4.3 is capable of *deterministically* deciding palindromes. This is in contrast to the pushdown automaton which was non-deterministic. The machine starts with its head at the leftmost character. It then crosses out the character and searches for the rightmost character. If these two characters don't match (as determined by the state of the machine) then the machine rejects the input. Otherwise the machine crosses out the letter and returns to the new leftmost letter. If this process is repeated until there are no more letters left in the string then the string is accepted as a palindrome.

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

Figure 4.5 shows a Turing machine which can recognise the language which a PDA could not. This shows that the Turing machine has different capabilities from a PDA (and since it is possible to simulate a PDA on a Turing Machine, this means that a Turing Machine is strictly more powerful than a PDA). Some of the steps the machine takes are shown in figure 4.6, notice how the letters are crossed out with different symbols to aid us when detecting the new left-most character. Also note that after the machine has left S_0 the only possible way for it to get back is if there are no input symbols encountered in the whole string. This signals that the processing is complete.

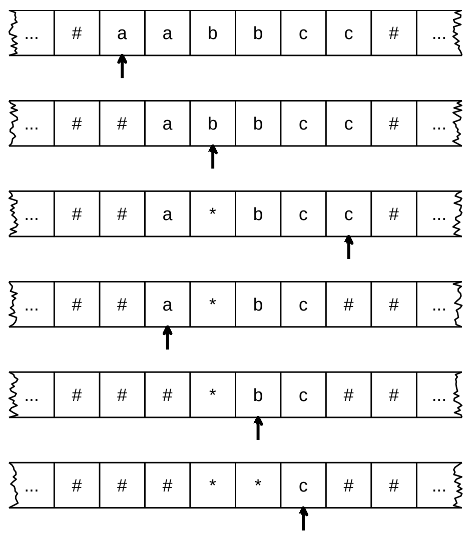


Figure 4.6: Crossing out letters to decide membership of the set $\{a^n b^n c^n\}$.

XOR'ing two numbers

Turing machines are capable of XOR'ing two numbers in binary (see figure 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 first number is reversed on the tape. The second is placed normally on the tape, with an extra zero for padding. The state of the machine's tape is shown in figure 4.8. As one can see the machine crosses out the least significant bits and places the result in place of the left-hand number. To read off the answer the final output string must be reversed.

Multiplication

Here we describe the possible design of a Turing machine 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 isn't.

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

3. Cross off an a and scan to the right until a b occurs. Shuttle between the b's and c's crossing one of each until all the b's are gone.

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

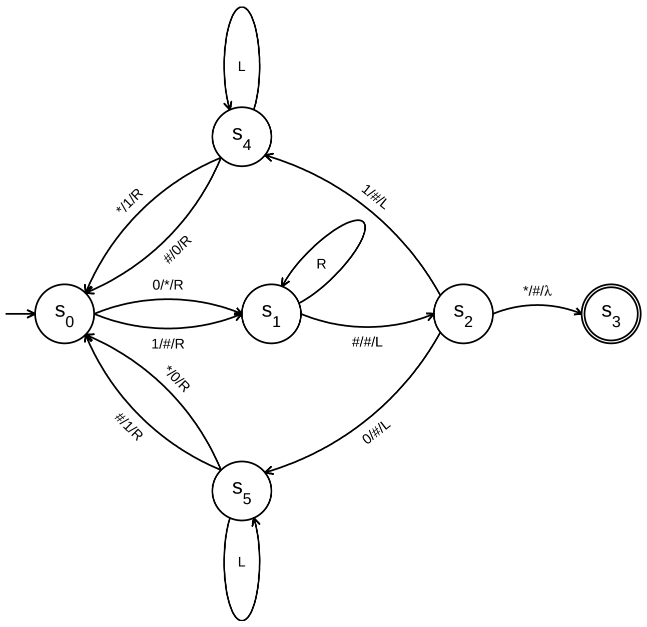


Figure 4.7: This Turing machine is capable of XOR'ing two numbers. The first number must be reversed and there must be a zero padding the two numbers.

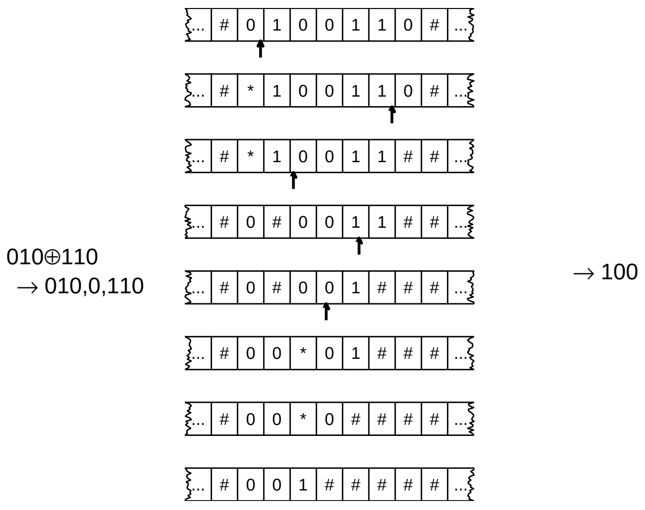


Figure 4.8: XOR'ing two numbers.

4.1.3 Improving the Turing Machine?

How should we improve our Turing machine to arrive at a more powerful computational model? It is not at all obvious that we can. 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 Turing machine[Martin, 2003].

If we allow the Turing machine 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 Turing machine which in turn is equivalent to a single tape Turing machine[Martin, 2003]).

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

Remember that these other possible Turing machines 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 there is nothing new that they can compute. These features then don't add to the computational power, which is what we are looking for.

Instead let's compare other computational systems and hopefully draw inspiration from them as to the next feature which will improve the power of a Turing machine.

4.2 Phrase-Structure languages

Phrase-structure grammars (also known as context-sensitive) 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 Turing Machines 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 Turing Machine can :

$$\boxed{\begin{array}{l} S \mapsto \mathtt{ab}NS\mathtt{c} \\ S \mapsto \epsilon \\ \mathtt{b}N\mathtt{a} \mapsto \mathtt{ab}N \\ \mathtt{b}N\mathtt{c} \mapsto \mathtt{bc} \\ \mathtt{b}N\mathtt{b} \mapsto \mathtt{bb}N \end{array}}$$

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 Turing machine). In fact it is capable of generating all bitstrings which 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 The 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[1]. 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 favorite constructs from other programming languages. To set the value of a variable to zero:

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

As a shorthand we will refer to the above code as `clear`, but remember, it is not a procedure, just shorthand. 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--
}
```

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

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

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

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

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

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 Turing machine. The proof is beyond the scope of these notes and won't be covered here. It seems as if all these different approaches to computation are equivalent.

4.4 The Church-Turing Thesis

In all of the above three sections on Turing machines, Phrase-structure languages, and the Impoverished programming language there doesn't seem to be enough mechanisms to solve complex problems. Yet Turing conjectured in the 1930's 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 Turing Machine.

It is known as the Church-Turing thesis since a similar theory by Alonzo Church which 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 incredibly long 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 Turing machine can solve the problem, but if it takes more than 10 billion years to halt, it probably isn't 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 Turing machine. (Hint: efficiency is not important here.)

Exercise 4.2

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

Exercise 4.3

Find a corresponding phrase-sensitive grammar which 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 which are easy to read.

Exercise 4.4

Construct a phrase-sensitive grammar which 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 4.5

Write short code snippets which 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 4.6

Implement the factorial function in the impoverished programming language.

Exercise 4.7

Many interesting computational systems have been shown to be equivalent to a Turing machine. 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 4.8

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 which 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 and Complexity

Having seemingly reached the limitations of computation we now seek a more comprehensive classification. To this end 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. We define the complexity classes P and NP, and discuss whether P=NP, which is an open issue in Computer Science today.

5.1 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 $O(n^k)$ for some constant value k. In some cases k might be very big (say $k = 10$), then the algorithm will be unusable for all but the smallest n. However this order is still not as bad as problems covered in the next section which are thought to have exponential complexity.

Before we cover the algorithms in more depth though we first cover some graph terminology so that we can discuss the solutions with clarity and exactness.

5.1.1 Some graph 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 figure 1.1) and buildings (in figure 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.

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.

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).

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 which returns to the starting vertex without visiting any vertex more than once.

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 edges connect vertices from the one set with vertices chosen from the other.

Hamiltonian Path

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

5.1.2 The weary student

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 recursive in nature; to calculate the shortest path from Grahamstown to Johannesburg we first calculate the shortest paths from:

- Port Elizabeth 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 Grahamstown; 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. The pseudo-code below specifies this algorithm more succinctly.

```
$sd(v_i, d)$ – Find the shortest distance from vertex $v_i$ to destination $d$
  1. if ($v_i == d$) return 0
  2. if $dist[i]$ is known return $dist[i]$
  3. $ans = \infty$
  4. for each of the vertices directly connected to $v_i$
     (a) $temp = sd(v_k, d)$
     (b) $temp = temp +$ (edge weight)
     (c) if ($temp < ans$) $ans = temp$
  5. $dist[i] = ans$
  6. return $ans$
```

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

$$sd(GT, Jo) = \min(132 + sd(PE, Jo), 180 + sd(EL, Jo), 249 + sd(Mi, Jo))$$

$$\begin{aligned}
sd(PE, Jo) &= 335 + sd(Ge, Jo) \\
&= 335 + 438 + sd(CT, Jo) \\
&= 132 + 335 + 438 + \infty \\
&= \infty \\
sd(EL, Jo) &= 674 + sd(Du, Jo) \\
&= 674 + 290 + sd(Ha, Jo)
\end{aligned}$$

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

$$\begin{aligned}
sd(EL, Jo) &= 674 + 290 + 268 \\
&= 1232 \\
sd(Mi, Jo) &= \min(538 + sd(Up, Jo), 403 + sd(Ki, Jo), 319 + sd(Bl, Jo))
\end{aligned}$$

$$\begin{aligned}
sd(Up, Jo) &= \min(796, 361 + sd(Sp, Jo)) \\
&= \min(796, 361 + 541 + sd(CT, Jo) \\
&= \min(796, \infty) \\
&= 796 \\
sd(Ki, Jo) &= 476 \\
sd(Bl, Jo) &= \min(398, 320 + sd(Ha, Jo)) \\
&= \min(398, 320 + 268) \\
&= 398
\end{aligned}$$

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

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

Notice how saving the result for $sd(Ha, Jo)$ saved us having to recompute 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 and hence tractable.

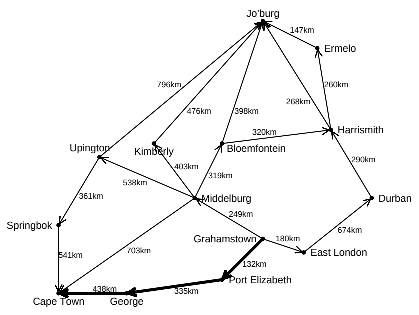
5.1.3 Cable-laying

To solve the cable-laying problem we need to find what is known in graph terminology as a minimal 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.

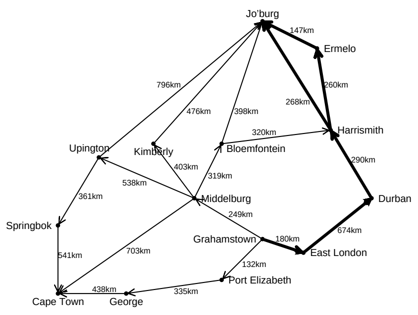
To start the algorithm we note that the shortest edge of any vertex will always be part of the minimal spanning tree. As an informal proof imagine that the algorithm is nearly complete and only has to connect one more vertex. This means that all the other vertices are already connected and we must choose 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's 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



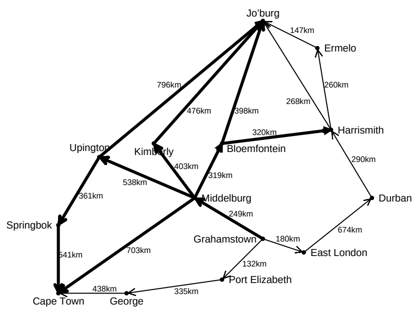
BL=	CT=∞	DU=	EL=	ER=	GE=∞
GHT=	HA=	MI=	PE=∞	SP=	UP=

(a) Going via Port Elizabeth



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 5.1: To calculate the shortest distance from Grahamstown to Johannesburg, we first calculate the shortest distance from Grahamstown's neighbours.

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.

The algorithm more formally specified:

```
$mst(edges)$ – minimum spanning tree, returns list of used edges

1. Initialise boolean array $used$ to all false.
2. Initialise list $ans$ to $\{\}$.
3. Pick a random vertex.
4. Add the vertex's shortest edge to $ans$, set $used$ for both vertices to $true$.
5. While there are unused vertices:
    (a) Find smallest edge between a used($v_i$) and an unused vertex($v_j$).
    (b) Add this edge to $ans$ and set $used[j] = true$
6. return $ans$
```

The solution for the cable-laying problem is shown in figure 5.2. 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.

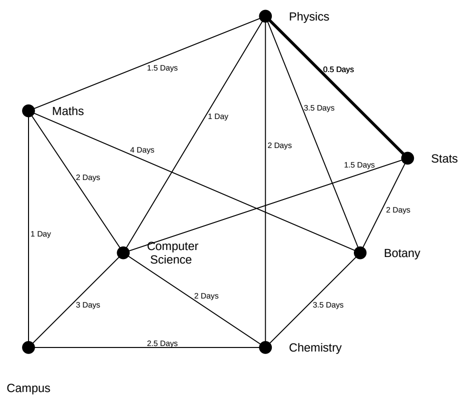
To analyse the complexity of this algorithm we note that we have to add $(n-2)$ vertices to the tree ($O(n)$). For each addition though we might be forced to search through the entire list of edges ($O(m)$). This means the order is again roughly $O(mn)$. This is again polynomial and hence considered 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.

5.1.4 The New Manager

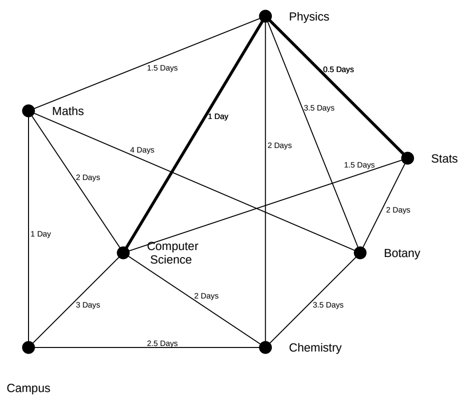
Assigning employees to tasks can also be shown to have a polynomial solution. The trick is to turn the problem into a graph. In figure 5.3 we create a directed bipartite graph, connecting people to the jobs they are able to perform. A source node, and a sink node are also added, as they simplify the algorithm. The source node connects to all people, while all the jobs are connected to the sink node.

To perform a matching[1] 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 figure 5.4 where bad assignments occur in the first two matchings, and are then reassigned in the last two matchings. We obtain

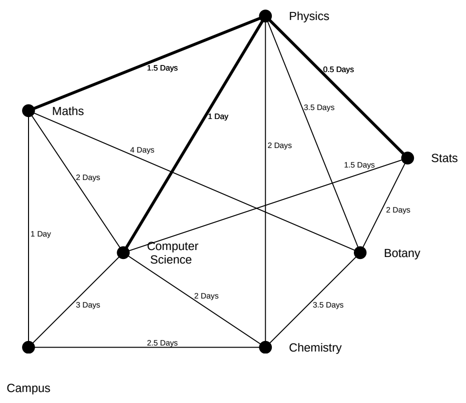
[1]Maximum matching is actually a specialisation of the network flow algorithm. Imagine a network of roads which many cars want to use to get from point A to point B. All the roads can handle different amounts of traffic as some of the roads are highways and some are single-lane country roads. The network-flow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



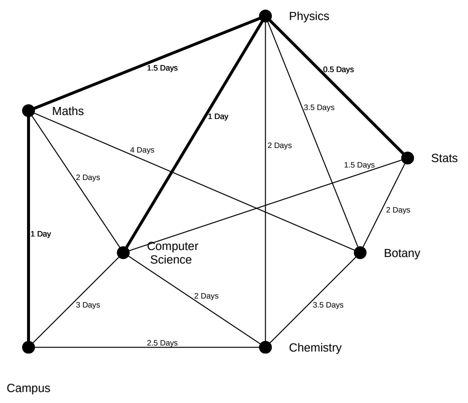
(a) Choose Random vertex (Stats), and find shortest edge = Physics-Stats (0.5 days)



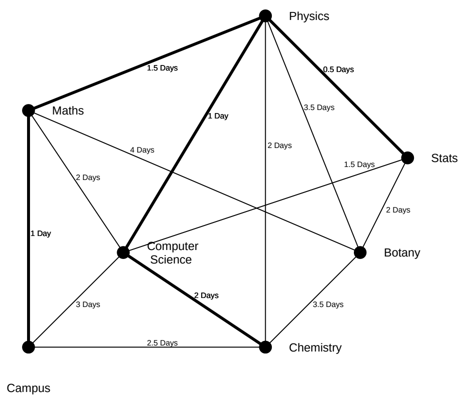
(b) Shortest edge between used and unused vertices = CS-Physics (1 day).



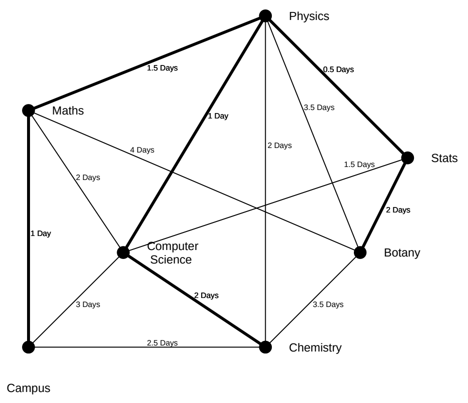
(c) Shortest edge between used and unused vertices = Maths-Physics (1.5 days)



(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 5.2: Finding the minimal spanning tree for the science departments.

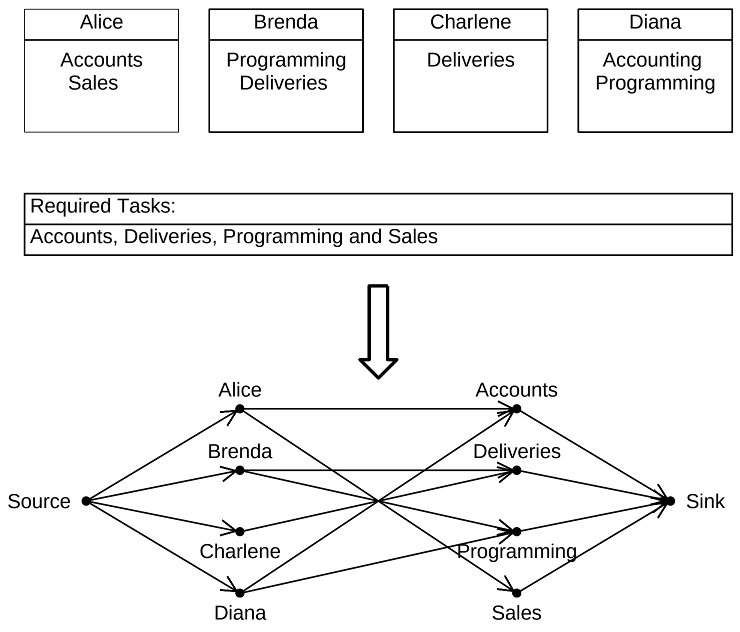


Figure 5.3: Transforming the problem into a graph problem.

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 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 which 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 which a matched person could switch to that is not already matched. This is the definition of an optimal matching.

More formally the algorithm can be described as:

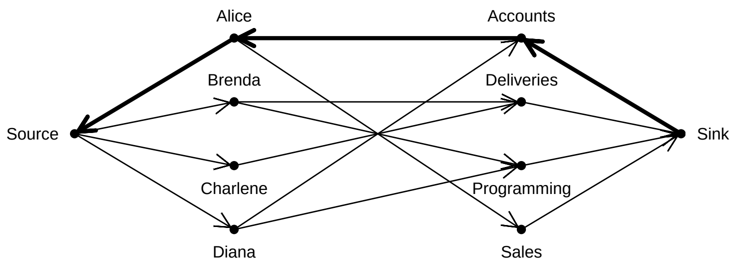
```
$findpath(s, d)$
  1. if $s == d$ return $true$
  2. if $visited[s]$ return $false$
  3. $visited[s] = true$
  4. for each vertex($v_i$) which $s$ connects to
       • if $findpath(v_i, d)$ return $true$
  5. return $false$

$mm$ – perform a maximal matching
  1. Create appropriate graph with sink and source vertices.
  2. $n = 0$
  3. Initialise boolean array $visited$ to all false.
  4. While $findpath(source, sink)$
     (a) $n = n + 1$
     (b) Reverse all edges which make up the path.
     (c) Reset $visited$ to all false.
  5. $n$ represents the maximum number of assignments possible; the individ-
     ual assignments are given by the reversed edges, which are not connected
     to the source or sink.
```

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.

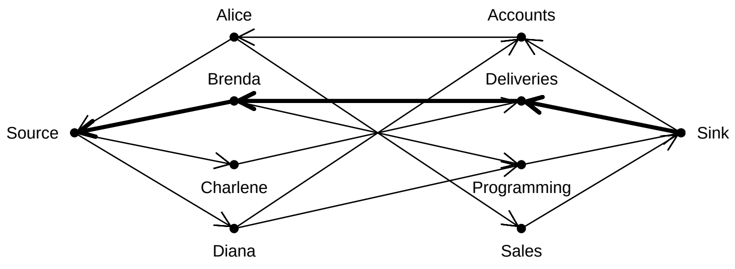
5.2 NP-Complete problems

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

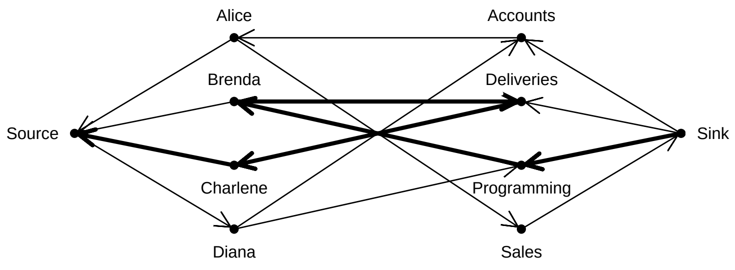
An interesting aspect of these problems is that they have all been proven equivalent to each other. This means that it is possible to transform one problem into another using an algorithm of polynomial order. If we find an efficient (i.e.



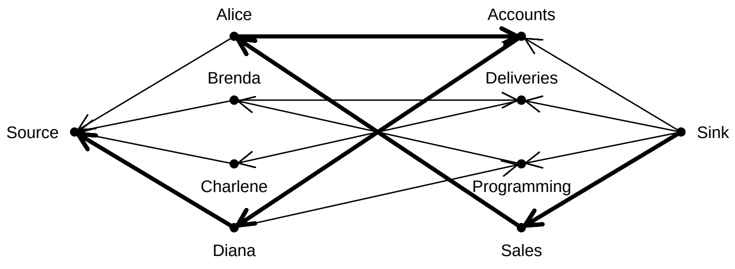
(a) Assigning Alice to Accounts.



(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 5.4: Using a graph to solve the matching problem.

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 (P) is equal to the class of Nondeterministic Polynomial problems (NP) is the largest outstanding issue in theoretical computer science today. It has also motivated a large amount of research behind quantum computing which would be able to solve NP problems in polynomial time.

5.2.1 The Traveling Salesman

The Traveling Salesman problem described in the introductory chapter is trying to find a Hamiltonian path in a graph representing a road map. Unfortunately the Traveling Salesman problem is a problem which 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 haven't 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.

5.2.2 3-SAT

This problem has historical significance as it was the first problem to be proven NP-complete. 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 $\{v_1 = true, v_2 = false, \ldots, v_k = false\}$ which satisfy the input expression. This problem has a naive 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 P or NP.

5.3 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.

5.3.1 The Halting Problem

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 which correctly identifies the programs that halt for all types of input and always terminates. Call this program '`Halts`'. Now construct a program '`S`' of the following form:

```
Program Halts(C,I)
//Accepts code C and input I and returns true or false
//Representing whether program C will terminate with input I
//Note that it always terminates.

Program S(W)
If Halts(W,W)
   While true
   {
     //Infinite Loop!
   }
Else
   Return false
```

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 which doesn't 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 which 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 which *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 which 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.

5.3.2 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 which produce identical strings for both the upper

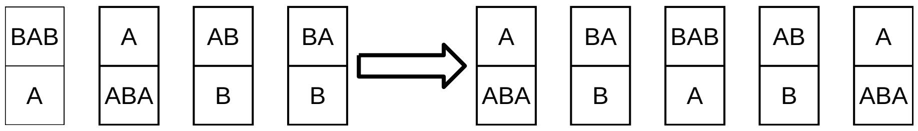


Figure 5.5: A solution to Post's Correspondence Problem: Given a set of dominoes (on the left) is it possible to find a configuration (with possibly repeated dominoes) where the string formed by the top row is the same as the string formed in the bottom row (as shown on the right).

and lower halves. An example correspondence problem and a solution is shown in figure 5.5. 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.

Exercise 5.1

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 5.2

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 5.3

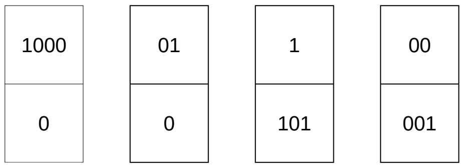
Write a program which solves the Travelling Salesman problem. To simplify the problem just search for any Hamiltonian path, rather than the shortest. Write a method which generates random graphs, with roughly half of the vertices connected to any given vertex. Test your solution on these random graphs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

Exercise 5.4

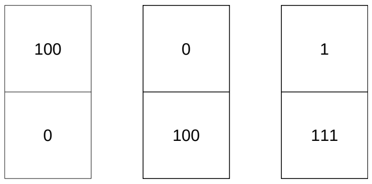
Read up on modifications to the minimum spanning tree algorithm which change the order to $O(m \log(m))$.

Exercise 5.5

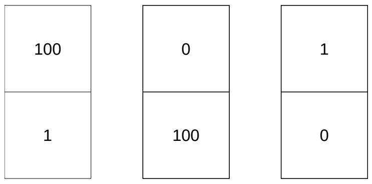
An interesting function related to the Halting Problem is the Busy Beaver function. Define $BB(n)$ to be the maximum number of ones which can be marked on a halting Turing machine of n states. $BB(n)$ is extremely difficult to calculate, even for very small values of n. Part of this difficulty is due to the number of possible Turing Machines being exponential in n. It is made worse by the fact that some of the machine's don't 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.



(a) Problem A



(b) Problem B



(c) Problem C

Figure 5.6: Three instances of Post's Correspondence Problem. Two are solvable but hard, and the other can be proven to have no solution.

Exercise 5.6

In figure 5.6 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. Can you find the problem with no solutions?
2. (Much harder) Solve the other two problems!

Chapter 6

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 were their main limitation and as such they couldn't remember an arbitrary number of previous symbols.

To overcome this limitation a pushdown automaton was introduced. This machine had a memory which could remember as many previously seen symbols as was necessary. While there are differences between deterministic and non-deterministic pushdown automata, these differences were not covered as they are more appropriately covered in a compiler course. Instead the limitations of non-deterministic pushdown automata were emphasized. Since pushdown automata are equivalent to context-free grammars, pushdown machines are unable to recognise any language which 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 wouldn't 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 automata or a finite-state machine then it is tractable. If the problem requires a Turing machine and doesn't seem to have an efficient solution there are now several options. Proving the problem equivalent to a known 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 which works for all possible inputs.

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

Bibliography

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

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

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

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

[Kinber and Smith, 2001] Kinber, E. and Smith, C. (2001). *Theory of Computing - A gentle introduction.* Prentice-Hall.

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

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

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

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

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