Designing, Analysing and Applying Algorithms
Course Notes for COMS2003

Notes by
Scott Hazelhurst

Course Presented by
Pravesh Ranchod

Preface

This is the set of course notes for COMS2003. The notes are intended to complement the lectures and other course material, and are by no means intended to be complete. Students should consult the various references that have been given to find additional material, and different views on the subject matter.

This material is still under development. Please would you report any errors or problems (no matter how big or small). Any suggestions would be gratefully received.

My thanks to Winnie Hlatshwayo for help with typesetting of this material and to Tal Even-Tov who pointed out a number of errors in a previous versions of these notes.

School of Electrical and Information
University of the Witwatersrand,
Johannesburg
Private Bag 3
2050 Wits
South Africa

Contents

Chapter 1

Introduction

Computer scientists are problem solvers. Given a problem, a computer scientist analyses it, and comes up with a solution. The heart of the solution is an *algorithm*, and so the bread and butter of a computer scientist is the ability to design and analyse algorithms.

We shall be looking at techniques for designing and analysing algorithms. Given, a problem we want to be able to answer questions like:

- How do we go about solving the problem?
- Which algorithmic techniques should be used?
- Given an algorithm, can we estimate how efficient it will be? And, given a number of algorithms to choose from, how do we select the best?

There is very important and useful theory that will guide us in answering these questions. Since it is very important for you to know and be able to use this theory, we shall spend some time on it.

But the value of the theory is not just that there is some interesting mathematics behind it, but that it is a powerful theory with lots of application. This course is entitled *Applications of Algorithms* and much of what we will be looking at is application and practical use. We will see in the practical sections how the theory can be used.

Part of being a computer scientist is having a good repertoire of algorithms to choose from. New problems can often be solved by simple modifications to old ones.

Most of this course will be taken up with the study of the very important class of graph and network algorithms.

- It is one of the most important classes of algorithms, both theoretically and practically.
- They have applications is a wide range of problem domains.
- The techniques we use in design and analysis can be applied to other types of algorithms too.

- Similarly, some of the practical programming ideas we'll use can be adopted elsewhere.

In summary, we shall be studying graph and network algorithms because they are an interesting and important class of algorithms in their own right, and as a vehicle for the study of algorithms more generally.

Role of programming and programming language

This course is a practical one. We won't just be coming up with paper algorithms, but implementing them as Java programs and running them.

- Practical programming skills are very important and by the end of this course you must be able to design and write *good* Java programs up to a few hundred lines in length.
- *However*, it is the 'coming up with the paper algorithm' that precedes the programming and is a more difficult task.

  Almost everyone can pick up a smattering of Java syntax and semantics so as to get a program up and running in some fashion. But, to solve difficult problems properly requires more and this is your task.

We'll be using Java as our programming language: Java is an attractive language for several reasons, is very popular at the moment, and can run a wide range of platforms. However, it is important to realise that while you will have to learn the Java language and ideas of object-orientation, central messages of this course are language independent. They apply to any other language you care to use, and will apply as much in twenty years time as they do now.

1.1 The design and analysis of algorithms

The classical phases of the software design process are shown below. In real-life, most systems are cyclical in nature. For example, problems found in design or implementation may be a result of errors or omissions in the requirements phase. Thus, the process tends to be cyclical rather than linear in nature.

1.1.1 Coming up with the user requirements

A customer comes to you with a problem to solve, what are their actual requirements? This may sound trivial, but in fact is often the most difficult part of the whole problem. People usually don't know what they want, and won't be happy until they get it. And it's one thing to know at a very general, high-level what problem you want to solve; it's another thing to know exactly what functionality a program should have.

Usually, the requirements are specified in a way that the user can understand. They may form a legal contract between a customer and software supplier.

1.1.2 Specification

The specification is the translation of the requirements into a form suitable for the designers and implementors. The requirements are aimed at a client, whereas the specification is designed for the people who will actually be building the system.

The specification can often be given in different formats and often mathematical notations can be used.

1.1.3 Design

Once the specification is done, we can start building the system. Just as an architect's plans precede the laying of bricks for a building, the software architect's plans precede the writing of code.

The design presents the overall framework of the solution to the problem.

1.1.4 Implementation

The design can be presented at different levels of details, just as a building architect's can be. Initial plans may only describe a building's gross structure: how many floors, what the outside looks like, etc. But the final plans could describe where network connections are, the colour of the curtains etc.

The most detailed level of design of a solution to a problem is the algorithm that solves it, in a form that we can directly convert into a program. One way of looking at the process of coming up with an algorithm or program is seeing it as translating of high-level designs into lower-level of designs.

1.1.5 Testing, debugging and integration

After the implementation comes putting the pieces together, making sure that it works and fixing problems. Different techniques can be used for ensuring correctness:

- Testing: is the most common approach. Given an algorithm we select a set of test case studies and check whether the algorithm works correctly. This is a very useful approach since it gives us quick, meaningful feedback. But there are many limitations. For most real systems, it is only possible to test a very small proportion of test cases. The construction of good test cases is also expensive.

- Verification: the use of mathematical techniques for proving the correctness of systems. This can either be done after the algorithm is constructed, or during program construction.

  The great attraction of verification is that a verification gives a very high confidence of correctness since it proves that the algorithm correctly implements the specification always (unlike testing, where we cannot be sure that the next case tested will fail). Of course, the question of whether the specification is correct is another problem!

This approach is not commonly used, because it requires a high-level of expertise, which many practitioners do not have. However, the theory and practice of verification has seen tremendous leaps over the last ten years. In hardware, verification is now used widely by the leading semi-conductor manufacturers, and we are likely to see an increase of the use of these techniques in software.

Comments

In most systems, the cost of testing and integration is the most expensive part of the whole process, but the causes of the problems are earlier on in the process. For this reason, it is important to have a scientific and methodical approach to problem solution. It is also important to realise that different phases are interlinked.

In this course, we'll be looking at specification, design, and implementation. We'll not only look at lots of real examples, but try to develop an approach to problem solving.

Of course, we won't be looking any many related issues (for example, requirements and verification). This is not because they are unimportant — on the contrary. However, this is only one course and we have to find some focus. But I do want to realise where what we are doing fits in, and that it is by no means the whole story.

1.2 Approaches to algorithm design

The central difficulty with algorithm design is the complexity of problems that need to be solved. This difficulty shows itself in many ways:

- Solution size: solutions of real problems are often big, sometimes huge.
- Level of detail. Real problems usually have many different levels of detail.

  To use our architect analogy: we have to worry about the overall structure of the building; we also have to worry about the colour of the paint. We have to worry about what power supply the building needs (a heavy current problem), we have to worry about where to put power points (an ergonomics issue).

So problem solving involved *managing complexity*. However powerful our brains, we can't deal with a problem as a monolithic thing. Our basic tools are: *divide-and-conquer* and *abstraction*.

1.2.1 Divide and conquer

This approach is often used in politics or in war. If you have a strong opposition, try to divide them up in parts and then tackle each part separately.

Exactly the same thing works with problem solving, except we have a constructive goal, rather than a destructive one. So, when we look at a problem, our first step is often to divide the

problem up into parts. And, then tackle each problem separately. The sub-problems themselves may in turn be too difficult solve, so we may have to break the sub-problems up into sub-sub-problems and so on ...

The reason this works is that the complexity of a solution is not linearly related to problem size. Informally, this means that a problem of size $2x$ is going to be much more than twice as difficult to solve than a problem of size x. For this reason, it is worth spending effort in dividing up the problem, solving each sub-problem separately, and then putting the solutions together again.

Example 1.1. *Suppose we wish to develop a student registration system. The first step is to break down the problem to discover what we might need. For example, we might need modules that:*

- *Read in and write out student records to the student database system.*
- *A module that checks that a student's registration rules like prerequisites and corequisites, minimum and maximum course loads.*
- *A user interface module that interacts with a user.*
- *A query module that performs tasks like producing class lists, sorting lists etc.*

A. Separation of concerns

One of the reasons this works is that at each stage we focus on the problem at hand, which simplifies things considerably. In our example above, when we worry about how to implement the sorting routines we don't have to worry about user interface issues or security. When we implement the user interface module, we don't have to worry about how the sorting algorithm is implemented.

B. Interfaces

Separation of concerns is not absolute as the different modules of our system do have to work together after all, and so our modules have to know *appropriate* information about the other modules. Here are some examples:

- When we implement our sorting algorithm, we need to know something about the data that will be used in practice. There are many sorting algorithms that can be used, and which one is best will depend on the situation: for example, the number of items to be sorted, the type of data, whether the items to be sorted are likely to be partially sorted already and so on ...

  So, when implementing one module we may need to know something of what other modules that use this module might need in terms of functional or non-functional requirements.

- One module may rely on another module: module *A* may use a function, procedure, method, capability of module *B*. *A* therefore needs to know what functionality module *B* provides. Each function should be clearly specified saying *what* is does, how it should be *called*, how it should be *used*, as well as any limitations. **But we don't have to know how it is implemented!**

  Example: our user interface module might call a sort function implemented in the query module. The user interface module only has to know the name of the function and what parameters it takes. It doesn't have to know how the sort algorithm is implemented.

The key point here is that each module and part of a module should have a specification which precisely describes what it does. Suppose we are building a module *A*. Module *A*'s specification has two purposes:

- For a module that intends to use module *A*, it says *what* the module *A* does and how it should be used. This is what is important for the user module. The user module does not want to know how *A* is implemented.

- For the person implementing module *A* it says what *A* must do. The specification captures or summarises what the implementor must know about the way in which module *A* will be used. It is the implementor's interface to the outside world. The implementor must make sure that the specification is implemented by her/his code, but anything that is not in the specification is irrelevant for the implementor and need not be considered.

1.2.2 Abstraction

This brings us to *abstraction*, one of the most powerful techniques for managing complexity. The Concise Oxford Dictionary defines the word *abstract* as follows:

- *adj.* denoting a quality or condition ... rather than a concrete object.

- *v.* consider abstractly or separately from something else.

- *n.* a statement or summary of a book etc.

All these definitions give a flavour of what we mean by the term. Essentially we mean: consider the problem, algorithm or program at the right level of detail. By doing so we are able to handle the complexity.

Example 1.2. Suppose our system has a routine that checks that the student's registration meets the faculty's requirements. We could consider the routine at the following two levels:

First, a programmer who just wanted to use the routine would be given its header, something like:

```
boolean check_reg(studentNum, database, propreg, rules)
```

together with some specification, formal or informal, of what it does. Here the routine might have the following specification:

> Given the student's number, a reference to the database which stores the student's file, the proposed registration, and the rules which apply to the student, `check_reg` returns true if the registration meets all pre- and co-requisites as well as the faculty's minimum and maximum credit requirements.

This information is all someone who uses the routine needs to know. We could also view the routine at a more detailed level, showing how it is actually implemented.

```
boolean check_reg(studentNum,database,propreg,rules) {
  if (ValidstudentNumber)  {
       history = retrieve_record(studentNumber,database);
       ok1 = check_minmax_credits(propreg,rules);
       ok2 = check_prereqs(history,propreg);
       ok3 = check_coreqs(history,propreg);
       return ok1 && ok2 && ok3;
  } else {
    /* Deal with error */
  }
};
```

Clearly, someone has to see the routine at this level, but most programmers don't. Programmers who just want to use the `check_reg` routine don't need to and don't want to know the details of the implementation. In fact knowing the details of the implementation may just cloud the issues.

This lesson is even more important when designing algorithms. Getting the wrong level of detail at a too early stage will complicate matters.

The interface is one way of building an abstraction, since an interface is a way of viewing something that leaves out inessential details:

- For something that does not already exist, the interface acts as a specification;
- For something that does exist, we can build different interfaces for different actors.

Object orientation

In many modern languages, object orientation is the tool used for abstraction. The basic philosophy of object orientation is that the things that we wish to model are represented as objects. Objects are divided up into classes; all objects belonging to the same class have similar characteristics.

In our running example, we might have the following classes:

- Student class
- Course class
- Rule class

Each student we keep will be represented as an object of the Student class; similarly each set of rules will be represented as an object of the rule class; and so on.

We can also have hierarchy of classes. For example, we could define a sub-class of the Student class, called the Science student class. Objects of this class would share the characteristics of the Student class, but in addition would have characteristics appropriate for science students. We say that the Science student class *inherits* from the Student super-class.

A class definition will give the following:

- The data that each object must store (e.g. student name, number etc.);
- The operations that can be performed on objects. These are called *methods* or *messages*. So to change the state of an object, or to retrieve some internal data, one *invokes* the appropriate method.

Usually, we consider the data that each object stores as an implementation detail (it is part of how something is implemented rather than how we should use it). The set of method declarations (or a sub-set of the methods) often form the interface. The idea is that in order to use a class we need to know: the names of the methods, the arguments they take, and their specification. We do not need to know how they are implemented.

On the other side, the person who has to implement the class uses the method names and specification as the instructions that she should follow: as long as she meets these requirements she will have done her job.

Designing an object-oriented program

Designing a good object-oriented program requires skill, experience and a good understanding of the problem domain. There is no quick cook-book method of using object-oriented design, and in these notes, we don't have time to explore this in any sort of depth.

The first step is (as with any methodology) to understand the problem domain and to decide how to model the universe of the problem domain. In object-oriented design the most important modelling question is what classes of objects the universe will be divided into (e.g. see the example above). For each class we need to decide (a) what the public interface of the class is (and hence how it will interact with other classes), and (b) the type of data that objects of the class will store.

I cannot emphasise enough that the starting point is understanding the problem domain and then deciding how to model it: the coding of it in Java should only come after the ideas have been thoroughly tested. Before coding starts, therefore, a designer should test out the concepts.

- Is the public interface correct? One way of testing this is to consider what objects of the other classes need to know about the class. So consider each of the other classes in turn and think about how those classes need to interact with our class.
- Make up a small example, and graphically show how the data for that example would be represented (into classes and objects).
- Then, work through a solution for your small example to see how you would solve the problem. This will give you an idea of whether your public interface is good enough, or whether it could be simplified.
- When you design a new class, consider whether it has to be a brand new class, or whether you could make it as a sub-class of an existing class. If you can, you can save a lot of work and help future re-use. On other hand, don't be artificial ...

1.3 Applications of Algorithm to Graphs & Networks

The first part of this course will look at some general theory of algorithm design and analysis, and we shall spend significant time here. This part is the most theoretical, but it is important for you to see where we are going and some of the practical applications of the theory.

Why are we looking at graph theory and networks? Partly, because they are a good vehicle for studying algorithm design and analysis. But there's a very practical reason too. Graph theory algorithms have very many practical applications. Throughout this book, we'll look at practical examples, but let's take look at a few examples now:

Airline reservations Consider an air-line which flies between a number of cities. Some of the questions which the airline might want answered are: Can we fly direct between town A and B? Can we fly someone (perhaps with intermediate stops) from A to B? What is the shortest (or cheapest) route between A and B? These, similar problems, and generalisations of these questions can easily represented as graphs and solved.

Telephone system The international telephone system is a very complicated system. If you make an international call, your call will probably go through a number of intermediate exchanges. If you make the same call at different times, your call could well be connected via different routes. Some of the questions you might ask if you were running the system are: if an exchange goes down, how can calls be rerouted so as to minimise disruption? if a telephone line goes down, how can calls be rerouted? how many calls can be made between two points?

Road system If you are building a road system, besides the obvious questions about whether you can get from one place to another, there are a number of other things you will need to find out when planning. For example, what is the maximum traffic flow which your road can handle? which roads should be laid so as it minimise the cost of road construction while still having a reasonable degree of connectivity?

(i) – Computational biology Graph theoretical algorithms can be used to answer questions computational biology. For example, some DNA sequencing algorithms can be posed as graph problems, and then solved using efficient algorithms.

Scheduling Given n jobs to perform, and m processors to run the jobs, assign the jobs to processors so as to minimise execution time. Variations of this problem are very important in computer science. In fact, this is a very hard question in general. Graph theoretic techniques provide techniques for getting good results.

Theoretical results One of the areas of theoretical computer science is proving that some problems are NP-complete . Informally what this means is that any algorithm that solves the problem must be exponential in the size of the input. There are a number of graph problems which are known to be NP-complete. The power of this is that if we can transform (in some well-defined way) any given problem into one of these well-known graph problems, then we can tell that the given problem is also NP-complete.

Some other examples Other examples of questions which graph theory can help us solve are:

- Given a map, how many colours do we need to use so that no two adjacent countries have the same colour?
- Given an electric circuit (or a plumbing system or something similar) do we have some wires that must intersect each other?
- Given $2n$ people who rank each other according to some preferences, assign these people to n rooms, each with two people so that the assignment is a "good" one?
- Given n cities and a set of roads between them, can we visit all the cities, using no road more than once?

1.4 Mathematics and Notation

Part of what this course tries to show is the link between maths and computer science, and so mathematics notation is used throughout the book. Many of the high-level algorithms can be very elegantly and compactly expressed using mathematical notation. One of the skills that students need to acquire is the ability to read mathematics fluently and to be comfortable thinking in mathematics. Students must also be able to translate between mathematics notation and natural language and mathematics notation and code. Some of the symbols that will be used are shown in the table below.

Symbol	Meaning	Example
$\emptyset$	empty set	
$\in$	set membership	$s \in S$
$\exists$	exists	$\exists x \in S$
$\ni$	such that	$\exists x \in S \ni x > 10$
$\vee$	boolean or/disjunction	$x > 10 \vee S = \emptyset$
$\wedge$	boolean and/conjunction	$x > 10 \wedge S = \emptyset$
$\cup$	set union	
$\cap$	set intersection	
max	maximum	$\max S$ (biggest element in S)
$\arg_i \max$	arg-max	$\arg_i \max\{x_1, \ldots, x_n\}$ (for which i is x_i biggest)

In the case of $\arg_i \max$ there may be many i which gives the biggest: in this case an arbitrary choice of one of all the i is made. We can give similar definitions for $\arg_i \min$.

- $\min\{d(y) : \exists (x,y) \in E \ni x \in W, y \notin W\}$

  Break this down:

  - d is a function that takes an element and returns a number.
  - E is a set that contains pairs of elements. We'll see later that it contains the edges in a graph.
  - W is a set of elements (we'll see later they are the vertices in graph).
  - $\{(x,y) \in E : x \in W, y \notin W\}$
    All the elements (x,y) in E where x is an element of W and y isn't.
  - $\{y : \exists (x,y) \in E \ni x \in W, y \notin W\}$
    This is the set of all vertices in the graph that are not in W but are next to vertices which are in W

    The set of all y such that there is an element (x,y) an element of E such that x is an element of W and y is not.

    In the previous example, we wanted all the *pairs* in E that met the condition. In this example, we only want the second element of each pair.

    An equivalent way of expressing this would be in two steps.

    $F \stackrel{\text{def}}{=} \{(x,y) \in E : x \in W, y \notin W\}$
    $\{y : (x,y) \in F\}$
  - $\{d(y) : \exists (x,y) \in E \ni x \in W, y \notin W\}$
    Similar to the previous example, but now instead of the elements y we want the list of corresponding values $d(y)$.
  - $\min\{d(y) : \exists (x,y) \in E \ni x \in W, y \notin W\}$
    The smallest value $d(y)$ found in the set.
  - $\arg_y \min\{d(y) : \exists (x,y) \in E \ni x \in W, y \notin W\}$
    The y that gives the smallest $d(y)$ value.

1.5 Structure of this book

This is the provisional structure of the book — the order and depth of presentation may change as the course progresses.

- Chapter 2 motivates the need for algorithm analysis, gives the basic definitions and techniques for performing algorithm analysis, which enables us to predict the resource requirements of a program before it runs; this enables us to determine the feasibility of proposed solutions and to select between possible approaches.
- Chapter 3 gives the basic definitions of graphs and network theory. It also presents a number of example applications of graph theory in networks and other domains.
- Chapter 4 illustrates the ideas and theory covered so far by taking two example applications to show how the theory can be put into practice.
- Chapter 5 deals with an important class of graphs — *trees*. Many real applications have a tree-structure. But in addition, given any graph we can impose a tree-structure that helps us understand and manipulate graphs. For this reason, tree algorithms form the basis of many other graph algorithms.
- Chapter 6 looks at one of the central class of graph algorithms: how to find a path from one place to another. These algorithms are also used by other graph algorithms as partial steps.
- We then look at flows in Chapter 8. Here, we are interested in questions like: What is the capacity of a network? How much/many liquid/calls/packets can be pumped/sent from one place to another? Flow algorithms can also be used in *matching algorithms*, where we need to match objects with each other.
- Chapter 9 deals with planarity. Here we are particularly interested in graphical representations of graphs. Can we draw a graph so that no edges intersect. Again, this is of theoretical and practical interest.
- Our final chapter deals with finding circuits, cycles and tours in graphs, and we can consider this as an extension of Chapter 6.

Chapter 2

Algorithm Analysis

Efficiency is one of the key criteria by which algorithms and programs are judged by. The usability of a system – and even its feasibility – are determined by how many resources a program needs to run. The two most obvious resources in which we are interested are time and space:

- How long will the program take to run?
- How much memory does the program need to run?

With this information we are able to tell whether a particular algorithm will be usable, and if there are a number of possible algorithms to use, we can determine which will give better performance. Ideally, we would like an algorithm that minimises both these measures, though in general there is often a trade-off between the two.

There are many factors which can influence the performance of an algorithm. The most important factor is the data on which the program executes. For example, how long an algorithm that finds a shortest path in the graph takes to run depends on the size of the graph. It would probably also depend on the shape of the graph (are the two vertices which we are trying to find a path between close to each other).

2.1 Case analysis

The most common approach is to try to measure the performance of an algorithm with respect to the size of the inputs. For example:

- For an algorithm that sorts an array, we would measure the algorithm in terms of the size of the array;
- For an algorithm that finds a path in a graph, we would measure the algorithm in terms of the number of vertices and the number of edges in the graph.

However, in general, an algorithm may have different performance on different inputs of the same size. For example, some sorting algorithms have very good performance when the data they run on is already almost in order, but poor performance if the data is in reverse order. Therefore we distinguish three cases;

Best case: $B(n)$ measures the best case performance of an algorithm, if for each value of n, over all data sets of size n, the best performance of the algorithm is $B(n)$.

Worst case: $W(n)$ measures the worst case performance of an algorithm, if for each value of n, over all data sets of size n, the worst performance of the algorithm is $W(n)$.

Average case: $A(n)$ measures the average case performance of an algorithm, if for each value of n, over all data sets of size n, the average performance of the algorithm is $A(n)$. The average is the most difficult analysis for a number of reasons:

- We need to decide what 'average' means. Does it mean the average over all possible inputs of size n? Or, does it mean the average of the inputs that the program is likely to get? Usually, average case analysis requires probabilistic information and analysis.
- The mathematics is more complex.

2.2 Bounds

Sometimes the mathematics of doing analysis becomes complex, and we don't have all the information needed to do the analysis. In these cases, it may not be possible to get an exact measure of an algorithm's performance, but possible to get bounds on the algorithm's performance:

- An upper bound tells us that the algorithm never takes more time (or more space) than the upper bound;
- A lower bound tells us that the algorithm never takes less time (or memory) than the lower bound.

Note that the upper bound/lower bound analysis is not the same thing as best case/worst case. We often use both in the same analysis, e.g. we could say that

- $g(n)$ is an upper bound of the performance of an algorithm in the best case.

This would mean that for input of size n for which an algorithm performs *best*, the cost of the algorithm is less than or equal to $g(n)$.

2.3 Theoretical and Experimental Analysis

There are two main ways to determine the complexity of algorithm: one is theoretical, the other experimental. The theoretical analysis is done using the program text; it is the more powerful approach because we can have a very high degree of confidence on the correctness of the answer. Unfortunately, its also the more difficult.

The alternative approach is experimental. Here, we choose data sets of different sizes, run the program on these sample sets and then plot the time versus the size of the input. Of course, this approach depends entirely on how representative the data sets are, and also on how easy it is to generalise the result. Nevertheless, this can be a useful analysis technique, especially when used in conjunction with theoretical analysis.

Space versus time complexity: For the next while, we concentrate on time complexity. Much of what is said applies to space analysis too.

2.3.1 Theoretical analysis – the basic approach

The principle of theoretical analysis is straight-forward. Look at your program and do the following steps:

1. For each instruction in the program, determine the cost of performing the instruction. This could be done experimentally, or by using information available from the compiler and the data book of the processor. Obviously in doing this we need to take into account overhead, like the cost of doing function or method calls.

2. For each instruction, determine how many times the instruction is executed.

3. Suppose that there are n instructions that make up the program, and that instruction i takes t_i seconds to execute, and is executed m_i times. Then the cost of executing the program is

   $$\Sigma_{i=1}^{n} m_i t_i.$$

   Note, that usually the t_i are independent of the size of the input, while the m_i depend on the size of the input.

Determining the cost of the individual instructions: This might sound rather tedious, and indeed it would be if we had to do it. However, in practice it usually isn't necessary.

Typically, certain instructions are much more expensive than others and we just count them since they dominate the cost of the algorithm. Making sure we count the right operation needs a little thought, but isn't too difficult. Thus, we take a more abstract view of the program and we just count some things rather than everything. This simplifies our analysis, and we still get a fairly accurate measure of program execution.

In addition, we are particularly interested in the sensitivity of the program to the size of the input — is it linear, is it quadratic etc? Of course, the constant factors are important too, but often some simple experimental analysis will tell us these.

Determining how many times an instruction is executed: This is the difficult part of doing analysis since we need to take into account the effect of loops and selection statements. For loops, we need to know how many times the loop will be executed. Sometimes this is easy to see, but

often it requires work. For selection statements (like the Java **if** and **switch** statements), we need to know the probabilities of the different cases being taken. It is particularly with loops and selection statements that we make simplifications of doing case analysis or deriving bounds rather than exact results.

Example 2.1. Consider the program segment below.

```
a = 1;
b = 10;
i = 0;
```

Three assignments are made. If an assignment requires 10 ns to execute[1] the program will complete in approximately 30ns.

Example 2.2. Consider the program segment below.

```
a = 2 * i * b + 10;
b = (b + a)/3 + 1;
i = i + 1;
```

Suppose assignments take 10ns, multiplications and integer divisions 30ns, and additions 15ns to execute.

- There are 3 multiplications and integer divisions (90ns);
- There are 4 additions (60ns);
- There are 3 assignments (30ns);
- So in total the program takes 180ns to execute.

Example 2.3. Consider the program fragment below:

```
1 a = 1;
2 b = 10;
3 i = 0;
4 while (i < 10) {
5         a = 2 * i * b + 10;
6         b = (b + a)/3 + 1;
7         i = i + 1;
8 };
```

Suppose that an integer comparison takes 15ns.

- The first three lines take 30ns;

[1] A nanosecond – abbreviated *ns* is 10^{-9} of a second.

- Line 4 takes approximately 15ns to execute. It is executed 11 times, so the total cost is 165ns;
- Lines 5 to 7 take 180ns to execute; but they are executed 10 times. Therefore the total cost of these lines is 1800ns;
- Therefore the total cost of executing the code is 1995ns.

Example 2.4. Consider the program fragment below:

```
1 a = 1;
2 b = 10;
3 i = 0;
4 while (i < q) {
5         a = 2 * i * b + 10;
6         b = (b + a)/3 + 1;
7         i = i + 1;
8 };
```

- The first three lines take 30ns;
- Line 4 takes approximately 15ns to execute. It is executed $q+1$ times, so the total cost is $15q+15$ ns;
- Lines 5 to 7 take 180ns to execute; but they are executed q times. Therefore the total cost of these lines is $180q$ ns;
- Therefore the total cost of executing the code is $(195q+45)$ ns.

Example 2.5. Take the same program as the previous example. A simpler approach would be to say this: the most expensive operations are multiplication and integer division. These only occur in the loop. There are 4 such operations in each iteration of the loop; the loop executes q times, so there are $4q$ such operations in total. Since (1) these operations are the most expensive, and (2) no other operations get executed more than the multiplication and division operations, the overall cost is approximately cq for some constant c. This is useful information because it tells us that there is a linear relationship between the cost of the algorithm and q. If we really need to know what the constant c is, we can run some simple experiments to determine it.

2.3.2 Methodology of analysis

This section describes the basic approach to algorithm analysis.

Hierarchical decomposition: The first step is to look at your algorithm hierarchically. This is in the best traditions of the use of abstraction. Just as you shouldn't try to write your program all at once, you shouldn't try to analyse your algorithm all at once either. Do things step-by-step, to keep things simple.

The reason we are doing this is that it is easier to analyse smaller and simpler code than more complex code. So, we break the program down into its component pieces, analyse the component pieces and then put the results together.

To view the program hierarchically, do a hierarchical decomposition by breaking the program up into blocks of code. (It is hierarchical because blocks can contain blocks.) Essentially a block of code is: the body of a method, a sequence of statements, a loop, a body of a loop, a selection statement, or a body of a component of a selection statement.

Consider the program in Figure 2.1. The program doesn't do anything useful: it shows that we can mechanically decompose a programe and perform an analysis on it without deep insight into what the program does. However, we shall see later that to do good analyses one *must* understand what the program does: as in other mathematics we use a combination of insight and rules to answer a problem. The program can be broken into blocks as follows:

```
1  i = 10;
2  j = 0;
3  z = 4;
4  while (i > j) {
5          z = z + i − j;
6          if (num[i] < num[j])   {
7             num[i] = num[j] + z;
8             z--;
9          };
10         for (k = 0; k < i; k++) {
11              num[k]++;
12         };
13         i--;
14 };
```

Figure 2.1: Using hierarchical decomposition

- Lines 1–14
  - Lines 1–3
  - Lines 4–14
    - Lines 5–14
      - Line 5
      - Lines 6–9
        - Line 7–8

· Lines 10–12
  −− Line 11
· Line 13

Cost of the basic components: Working out the cost of the basic components like assignments is easy. If we are working out the cost in detail, there is quite a lot of detailed work to do, but it's not difficult. If we are just counting certain operations, then the task is even easier.

Putting the costs together: We use the following rules to work out the overall costs.

- *Sequence:* if the block is just a sequence of other blocks, the cost is just the sum of the costs the blocks;

  In the simple example below we have a sequence of three statements. Each statement has one assignment; therefore the total cost is 3 assignments.

```
x = 1;
y = 2;
z = 3;
```

- *Loop:* If the block is a loop, then (1) if the loop executes n times, and (2) the cost of executing the *body* of the loop on its i-th iteration is t_i, and (3) the cost of computing the loop condition and control c_i, *then* the cost of executing the loop is $\Sigma_1^n(t_i+c_i)+c_{n+1}$. Note, the control is executed a final time!

  In many cases, the cost of executing the body of the loop depends on which iteration of the loop is being executed. In some cases, each iteration costs the same. In these, simpler, cases then if the body of the loop costs t and performing the loop condition costs c, the cost of executing the loop is $n(c+t)+t$.

  It is not always easy to work out how many times the loop will be executed. For this reason we often have to do case analysis and treat best, worst, and average case separately.

  Another complication that can arise is the use of **`exit`** or **`continue`** statements which will mean that on some iterations of the loop only parts of the loop body will be executed. Then, the formula has to be adjusted above.

  Consider this loop.

```
i = 0;
while (i < 10) {
        j = j*10;
        i++;
}
```

If we count multiplications only then we see that there is one multiplication in the body of the loop and no other multiplications. Since the loop is executed 10 times, and on each iteration the cost is the same, the total amount of work is $10 \times 1 = 10$.

- *Selection:* As with loops, we often have to do case analysis, because the cost of a selection (either a switch or an if) depends on the conditions of the selection. Take this simple example:

```
if (cond) {SeqA; } else {SeqB; }
```

If the condition is true, one sequence is executed; if it is false then another sequence will be executed. If the cost of executing *SeqA* is c_a and the cost of executing *SeqB* is c_b, then in the best case the cost of executing the if statement will be $\min\{c_a, c_b\}$, while in the worst case it will be $\max\{c_a, c_b\}$.

For the average case, we need some probability of the condition being true. If the probability of the condition being true is p, then the cost of executing the if-statement is $pc_a + (1-p)c_b$.

Consider this example:

```
if (x < 100) {
   y = 2 * y + z;
   z = 3 * z;
} else
   y = y * z
```

If we are counting multiplications then in the best case there is 1 multiplication, and in the worst case 2 multiplications. Average case is more difficult. Suppose after studying the application we work out that the probability of x being less than 100 is 0.4. Then, in the average case, the amount of work is $0.4 \times 2 + 0.6 \times 1 = 1.4$.

The approach being suggested here is to try to use local knowledge as often as possible so that you can keep parts of the program separate when you do the analysis. However, there are times when you do need global knowledge of how the program works. Here is a simple example.

```
do (true) {
     num = SimpleInput.consoleReadInt();
     if (num == -1) { break; }
        ...
        ...
     };
```

In this program, the user enters a sequence of numbers, and the loop terminates when the user enters a -1. So if the user enters n numbers before entering a -1, the loop executes n times in total. Looking at the code we can easily see that the condition of the if statement is true exactly once! So using the global knowledge (that if the user enters a -1 the loop terminates and so the **`if`** statement won't be executed again), we can simplify our analysis considerably.

Let's try to apply the technique to our example on page 18. For simplicity, let's count the number of additions and subtractions in the *worst case*. For each block, we first work out the cost of all the sub-blocks, and the depending on whether the block is a sequence, a loop or an if, we work out the cost of the block.

- Lines 1–14: 0+160=160
  - Lines 1–3: 0
  - Lines 4–14

    This is a loop. The cost of executing the body of the loop depends on which iteration it is, in particular on the value of i. We see below from the analysis below that on each iteration of the loop, there are $2i+5$ operations. The loop iterates 10 times (i varies from 10 to 1, when i gets to 0, the loop terminates).

    So the cost is: $\Sigma_{i=1}^{10}(2i+5) = 160$.

    This is how we got $2i+5$ for each iteration:
    - Lines 5–14: $2+2+2i+1 = 2i+5$
      - Line 5: 2
      - Lines 6–9: 2 (worst-case)

        −− Line 7–8: 2
      - Lines 10–12: $i \times 2 = 2i$ (need to take into account the cost of the addition in the loop control.

        −− Line 11: 1
      - Line 13: 1

The best case is not when $n = 1$

A common misconception is that the best case for an algorithm is when $n = 1$ or $n = 0$ or 'when the list is empty' (etc.). This indicates a misunderstanding of what analysis is about. The purpose of an analysis is to determine the cost of the algorithm with respect to the size of the input. That is, we want a formula that, given the size of the input n, determines long the algorithm will run in the best case **for that size input**.

Let me give an analogy. I live about 10kms from Wits, and it usually takes me about 15 minutes to get from my front door to my office door (the average case). In the worst case, it takes a lot longer: I have to go back to check that I switched off the stove; one of my neighbours buttonholes me and yacks endlessly; the traffic lights are all against me; there's a traffic jam on the highway; someone's parked in my parking place etc. I travel the same distance, but the conditions are different and it takes me 25 minutes. On the other hand, if I come in at 5am, and

the traffic lights are all synchronised, I might do the trip in 11 minutes. Same distance, different conditions, different times. So my worst case is 25 minutes, my best case is 11 minutes.

The best case is not if I live 0km away (I am happy where I live at the moment, and I don't want to move onto campus). The best case is: given the distance I do live, what is the least amount of time I can expect to do the trip?

More generally if I were working in the Department of Traffic Management, I might try to have a formula that estimates the time my trip takes as a function of distance. We might have different formulas: this is the time it takes in normal traffic, this is the time it takes in heavy traffic, etc. I certainly wouldn't be able to sell advice which says *if you live where you work it doesn't take much time to get to work*. I might be able to sell advice that says: *here are a set of formulas: to work out how long it takes to get to work, choose the formula appropriate to the traffic conditions, and then plug in how far you live from work.*

2.3.3 Experimental analysis

We'll be using experimental analysis to complement and validate our theoretical analysis. Suppose we do an analysis and get that the algorithm runs in time $an+b$ seconds, where n is the size of the input and a and b are constants. As discussed earlier, we could work out the constants a and b given enough perseverance. But, often it is easier to work it out experimentally.

We run the program on different size data, and tabulate the actual run time versus the size of the input. Then we *curve-fit*: try to match the experimental date against the predicted curve. Suppose we get:

Size of input (n)	10	100	200	300	400
Time (in seconds)	120	570	1070	1570	2070

Then, with a little bit of algebra, you can get that $a = 5$ and $b = 70$. It's pretty easy in this example, because the figures were made up to be exact. For various reasons, in the real world, even with simple examples, one never gets the exact answer. There are various curve-fitting techniques, that we'll see later.

Of course, if you got this as your experimental results the following table:

Size of input (n)	10	100	200	300	400
Time (in seconds)	120	5070	20070	45070	80070

then you know that something is very wrong. There is absolutely no way in which you could fit a linear curve to this data. Either your analysis or your program (or both!) are wrong.

2.4 Asymptotic analysis

2.4.1 Motivation

We are particularly interested in analysis algorithms as the data size grows very large. This is what is meant by asymptotic analysis.

The major reason for concentrating on asymptotic analysis is that this is a much more interesting question. For example, if you are sorting 10 numbers, then almost any algorithm you use will do the job perfectly adequately. It's when you are trying to sort 1000000 numbers, that it really matters. We are also interested in how the algorithm will scale. What happens when our customer wants to double the capacity of his plant? Will the algorithm cope? (One advantage of doing asymptotic analysis is that some of the maths becomes simpler in parts.)

There are times when analysis algorithms for small data sets is critical, and considerable care needs to be taken in the design of such algorithms. However, the real challenge comes with dealing with large data.

A focus of asymptotic analysis is on the 'shape' of the curve that describes the algorithm's performance. For example, is it linear, is it quadratic, etc? It may seem that we aren't interested in constant factors at all: i.e. we particularly want to know that an algorithm is quadratic, but are less interested in knowing that it takes $10n^2 - 3n + 25$ seconds. Of course, the constant factors are important: given a choice between algorithm that takes $10n$ units of time, and one that takes n units of time, we would all choose the latter. But the 'shape' or class of the curve is the first question to be answered.

To see this, let's look at this table taken from [5]. Suppose we have a program that runs in time $f(n)$ time. This is how long it will take, depending on what f is. You can see why the *class* of f is so important. (In table s stands for seconds, m for minutes, d for days, y for years, and C for centuries.)

$f(n)$	10	20	30	40	50	60
n	$0.00001s$	$0.00002s$	$0.00003s$	$0.00004s$	$0.00005s$	$0.00006s$
n^2	$0.0001s$	$0.0004s$	$0.0009s$	$0.0016s$	$0.0025s$	$0.0036s$
n^3	$0.001s$	$0.008s$	$0.027s$	$0.064s$	$0.125s$	$0.216s$
n^5	$0.1s$	$3.2s$	$24.3s$	$1.7m$	$5.2m$	$13.0m$
2^n	$0.001s$	$1s$	$17.9m$	$12.7d$	$35.7y$	$366C$
3^n	$0.059s$	$58m$	$6.5y$	$3855C$	$2 \times 10^8 C$	$>>$

More instructive is the table that shows what effect getting a faster computer has. Suppose we have three models of computer A, B and C, where B is 100 times faster than A and C is 1000 times faster than A. In this example, suppose we have an hour to complete a task. Under column A, we see how big a problem we can solve in that time if we use machine A, and similarly for columns B and C.

$f(n)$	A	B	C
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31.6N_2$
n^3	N_3	$4.64N_3$	$10N_3$
n^5	N_4	$2.5N_4$	$3.98N_4$
2^n	N_5	$N_5 + 6.6$	$N_5 + 10$
3^n	N_6	$N_6 + 4.2$	$N_6 + 6.3$

So if using machine A we can solve a problem of size 100 in an hour, then with machine C we can solve a problem of size

- 100000, if the cost is linear;
- 3160, if the cost is quadratic;
- 1000, if the cost cubic;
- 110, if the cost is 2^n

These table shows that as the problem size grows it's not long before an algorithm that takes an seconds runs quicker than an algorithm that runs in bn^2 seconds, even if $a > b$.

2.4.2 Complexity classes

In performing algorithm analysis, we derive some function that given the data input size, determines the cost of the algorithm or bounds on the cost of the algorithm. We saw in the previous section that the shape or class of the function was very important. Here, we see the formal definitions of 'shape'.

First we look at upper bounds.

A. Upperbounds: Big-O Notation

Definition 2.1. *$O(f(n))$ is the set of functions $g : \mathbb{N} \to \mathbb{R}^+$ such that for some $c \in \mathbb{R}^+$ and for some $n_0 \in \mathbb{N}$, $g(n) \leq cf(n)$, for all $n \geq n_0$.*

Each $O(f(n))$ is called a complexity class. Suppose $g \in O(f(n))$ (this is often informally pronounced as 'suppose g is $O(f(n))$'). Informally, this captures the fact that g is dominated by $f(n)$ (perhaps multiplied by a constant factor). Also, since we are only interested in the behaviour of functions as n gets large, we don't have to show that g is dominated by f for all values of n, only that f dominates g from some point.

- $O(n^2)$ is the set of functions that are quadratic or smaller than quadratic.
- $n^2 \in O(n^2)$: just choose $n_0 = 1, c = 1$ and we get $n^2 \leq 1 \times n^2, \forall n > n_0$.
- $n \in O(n^2)$: just choose $n_0 = 1, c = 1$ and we get $n \leq 1 \times n^2, \forall n > n_0$.
- $n + 100 \in O(n^2)$. Note here that for small values of n, $n + 100 > n^2$. For example, for $n = 5, n + 100 = 105 > 25 = n^2$, but as n gets larger, n^2 soon catches up, because the rate of growth on n^2 is much larger than the rate of growth of n. We can find out exactly where they are equal by solving the equation $n^2 = n + 100$, with the constraint that $n > 0$. By doing this, we see that for all integers greater than or equal to 11, $n + 100 \leq n^2$. So the definition above is met, with $c = 1$, $n_0 = 11$.
- $100n + 100 \in O(n^2)$. Note here that for small values of n, $100n + 100 > n^2$. But by a similar process to the above argument, we can show that n^2 dominates $100n + 100$. For example, choose $n_0 = 101, c = 1$, we have that $\forall n \geq n_0, 100n + 100 \leq cn^2$.

- Usually, there are many n_0, c pairs that work. In the above example, we could choose $n_0 = 2, c = 100$. Note that $100 \times 2 + 100 = 300 < 100 \times 2^2$. While there is some aesthetic value in finding the smallest values of c and n_0 for which the relationship holds, what's important is finding some pair for which the relationship holds.

- $100n^2 + 100 \in O(n^2)$. We wish to find a c and $n_0 > 0$ such that the relationship holds. One way of doing this is by experimenting to find the values and then prove that the relationship holds. This is usually the easiest way, but it can also be done more formally using the following approach:

  1. Recall that the derivative of a function gives its rate of growth. The derivative (with respect to n) of $100n^2 + 100$ is $200n$. The derivative of cn^2 is $2cn$. So this implies that we should choose $2cn \geq 200n$. Since $n > 0$, this means that we should choose some $c \geq 100$.
  2. In this case, we can see that $100n^2 + 100$ has a 'head start' over $100n^2$, so if we choose $c = 100$, cn^2 will never be able to catch up with $100n^2 + 100$. So, choose $c > 100$. For simplicity, we choose $c = 150$ (a smaller c would do).
  3. Now choose n_0 such that $100n_0^2 + 100 \leq 150n_0^2$. Doing the algebra here, says choose n_0 such that $n_0 \geq 2$.

  The final step is to verify that these values of c and n_0 are correct; usually this can be done algebraically, or by induction.

- $10000n + 98897823678630663763 \in O(n^2)$

- $n^2 + 200n + 100 \in O(n^2)$: we could work through the analysis formally to show this. Intuitively, why this holds is that as n grows large, the term $200n + 100$ gets very small compared to n^2 (in fact, $\lim_{n\to\infty} \frac{200n+100}{n^2} = 0$).

Theorem 2.1. *$O(f(n)) = O(cf(n))$ for all constants $c > 0$.*

Theorem 2.2. *If $r, s \in \mathbb{R}^+, r < s$, then $O(n^r) \subset O(n^s)$. For all values of $i \in \mathbb{N}$, $O(n^i) \subset O(n^i \log n) \subset O(n^{i+1})$.*

Theorem 2.3. *For all values of k, $O(n^k) \subset O(2^n)$.*

Finding an upper bound on the performance of an algorithm is important because it gives us an assurance on how bad things can be. If this upper-bound performance is acceptable, then that is very useful information. So, in doing this analysis, we try to derive a function that is an upper-bound on the performance of the algorithm, and then decide which complexity class it belongs to.

We can see from the above theorems that in general a function belongs to many complexity classes. Ideally, we would like to find the smallest complexity class that the function belongs to. So, if we do an analysis of a sorting algorithm and we get that its performance is $2n^2 + 12n + 5$, we say that it is in $O(n^2)$. While it is true that it is also in $O(n^3)$ and $O(2^n)$, this isn't as useful information.

Summary: To say that an algorithm is $O(f(n))$ means that its performance as n gets larger is bound by above (no worse) than $f(n)$ multiplied by some constant.

B. $\Omega(n)$: Lower bounds

We use the Ω notation to describe lower bounds.

Definition 2.2. *$\Omega(f(n))$ is the set of functions $g : \mathbb{N} \to \mathbb{R}^+$ such that for some $c \in \mathbb{R}^+$ and for some $n_1 \in \mathbb{N}$, $g(n) \geq cf(n)$, for all $n \geq n_1$.*

The definition is symmetrical to the one above, but here we are talking about lower bounds. So, if we are measuring time complexity, to say that an algorithm is $\Omega(f(n))$ means that it will take longer than time $f(n)$ (multiplied by some constant).

Here are some examples:

- $n^2 \in \Omega(n^2)$, $0.0001n^2 \in \Omega(n^2)$;
- $n^2 - 100 \in \Omega(n^2)$;
- $n^3 \in \Omega(n^2)$;
- $n \notin \Omega(n^2)$.

Theorem 2.4. *$\Omega(f(n)) = \Omega(cf(n))$ for all constants $c > 0$.*

Theorem 2.5. *If $r, s \in \mathbb{R}^+, r < s$, then $\Omega(n^r) \supset \Omega(n^s)$. For all values of $i \in \mathbb{N}$, $\Omega(n^i) \supset \Omega(n^i \log n) \supset \Omega(n^{i+1})$.*

Theorem 2.6. *For all values of k, $\Omega(n^k) \supset \Omega(2^n)$.*

C. Order notation: Θ

We can put the upper bounds and lower bounds together.

Definition 2.3. $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$

If $g \in \Theta(f(n))$ (we say 'g is order $f(n)$'), then

- $\exists c_0, n_0$, such that $\forall n \geq n_0$, $g(n) \leq c_0 f(n)$;
- $\exists c_1, n_1$, such that $\forall n \geq n_1$, $g(n) \geq c_1 f(n)$.

Our ideal is to derive a function $g(n)$ that describes the complexity of an algorithm and then determine which Θ class it belongs to. Why then do we bother with O and Ω notation? The reason is very pragmatic: in the analysis of an algorithm we often have to make certain simplifications to make the mathematics tractable or because we lack certain information (like probability distributions).

One simplification is to turn equations into inequations. For example, in doing an analysis of a loop the number of times the loop executes is critical. However, it may be difficult to determine the exact number of times that the loop will iterate. However, it may be easier to determine an upper bound (or a lower bound) on the number of times the loop will iterate. This will turn our analysis in to finding upper or lower bounds on the cost of the algorithm. It is quite common to have results like:

- the algorithm is $O(n^2)$;
- the algorithm is $\Omega(n^{1.5})$;

and not have any more precise information.

2.5 Summary

This chapter introduced the analysis of algorithms, how we try to measure the performance of an algorithm with respect to the size of the input. But, it is not only the size of the data input that matters, but also the data itself. For this reason we often have to do case analysis (see Section 2.1, which discusses best case, worst case, and average case analysis).

Section 2.2 showed that to simplify analysis, we sometimes try to work out bounds rather than exact results.

Section 2.3 presented a basic methodology for performing analysis, and Section 2.4 introduced the asymptotic notation and showed how this notation is a useful way of capturing the essence of the performance of an algorithm.

References

Additional references for this section: Baase [1]; Cormen *et al.* [4].

Chapter 3

Basics of Graphs and Networks

Graph Theory is a central tool in computer science. First, it plays a very important part in many results of theoretical computer science. Second, many real problems have natural representation as graphs, and familiarity with the properties of and algorithms on graphs is extremely useful in solving problems. This section introduces the basic definitions of graph theory

3.1 Definitions and notations

Unfortunately each book on graph theory tends to use slightly different notation and terminology. Here's the set we'll use.

3.1.1 Undirected graphs

Graphically, we represent a graph on a piece of paper as a set of points, with arcs joining some of the points. The points are called the vertices (singular: vertex) and the arcs between them are called the edges. As an example, if a graph represents a road system, the vertices would be the cities and the edges would be the roads between the cities.

Definition 3.1. A graph $G = (V, E)$ is a pair, V of vertices and E of edges. V is a non-empty set, and E consists of a subset of the two element subsets of V. Each edge is denoted by two vertices (which the edge joins).

Example 3.1. *Suppose we want to construct at computer network joining the universities in South Africa. We will represent this computer network as a graph. The universities will be the vertices and the connections between the universities will be the edges. Let* $V = \{$*Wits, UCT, NW, FH, UKZN, Tuk, Rhodes*$\}$*, and let* $E = \{\{$*Tuk, Wits*$\}, \{$*Wits, NW*$\}, \{$*Wits, Rhodes*$\}, \{$*NW, FH*$\}, \{$*UKZN, FH*$\}, \{$*FH, Rhodes*$\}, \{$*FH, UCT*$\}, \{$*UCT, Rhodes*$\}\}$*. This can be represented graphically as shown in Figure 3.1.*

Definition 3.2. *Basic Graph Definitions*

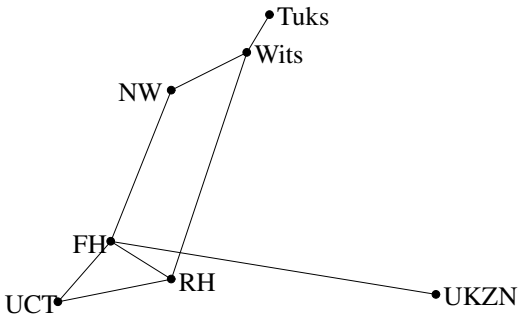


Figure 3.1: A graph representing a simple computer network

1. A *subgraph* G' of G is a graph (V', E') where $V' \subseteq V$ and $E' \subseteq E$
2. A *spanning subgraph* G' of G is a graph (V, E') where $E' \subseteq E$.
3. The number of vertices in V is known as the *order* of G and the number of edges is the *size* of G.
4. Two vertices joined by an edge are said to be *adjacent*.
5. The set of all vertices adjacent to a vertex v are the *neighbourhood* of v.
6. The number of edges which a vertex has is called the *degree* of a vertex.
7. If the degree of each vertex of a graph is the same (say r) then it is called an *r-regular graph*
8. A *walk* between vertices v and w is a set of edges $(v, v_1), (v_1, v_2), \ldots, (v_n, w)$.
9. A *path* between vertices v and w is a set of edges $(v, v_1), (v_1, v_2), \ldots, (v_n, w)$, where all the v_i are distinct.
10. A *cycle* is a path consisting of at least two edges from a vertex to its self.
11. A graph is *connected* if for each v and w in V, there is a path between v and w.
12. A graph is *complete* if there is an edge between every two vertices.

Exercise 3.1. *Look at the graph in example 3.1. Find the neighbourhood set for each vertex. Draw a subgraph of the graph which is not a spanning subgraph. Draw two spanning subgraphs of the graph, one of which should be connected, and one should not be.*

3.1.2 Directed graphs

The graphs we have seen so far have been *undirected*. Intuitively, this means that we can follow edges either way. Remember, the definition of E was as a set of subsets of V of size 2. Thus, the edge (u,v) is exactly the same as the edge (v,u). Technically, we should denote an undirected edge $\{u,v\}$ rather than as (u,v), but it is more common to use the latter and just remember that this is just convenient notation.

Undirected graphs are very useful in many circumstances, but sometimes, we cannot represent problems this way. Sometimes, a link is a one-way link only. For example, if we have a road system with no one-way roads, then we could use an undirected graph quite easily to represent this. But, the moment we had to include some one-way streets, we would have a problem. A directed graph can be used here. The fundamental distinction between a directed and undirected graph is that the edges in a direct graph are one-way only, and so the edges (u,v) and edges (v,u) are quite different. Usually, when we just talk of a 'graph' we mean an undirected graph, but it's best to distinguish between the two if there's any danger of ambiguity. We will now look at analogous definitions for directed graphs (we call directed graphs *digraphs*).

Definition 3.3. A directed graph $G = (V,E)$ is an ordered pair, V of vertices and E of edges. V is a non-empty set, and E consists of a subset of $V \times V$. Each edge is denoted by two vertices which the edge joins.

Graphically, we represent a directed graph in the same way as an undirected graph, except that we place an arrow on the edges to show the direction.

Exercise 3.2. *The directed graph below shows an irrigation system. Water can flow in one direction in a pipe. The direction is shown by an arrow.*

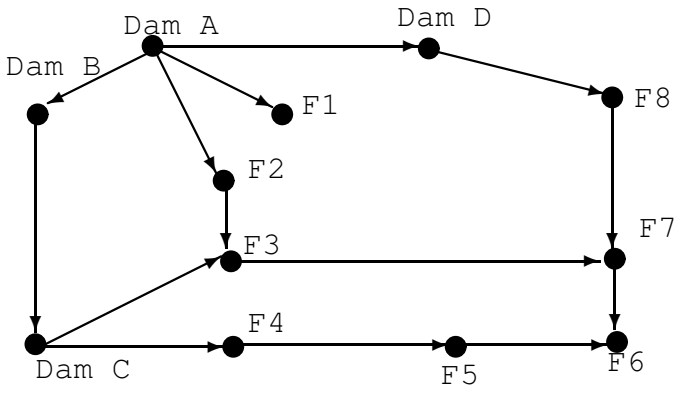


Figure 3.2: Irrigation scheme example

- *Give the set theoretic definition of the above graph.*

- *Suppose that the pumping system is very reliable and that all farmers are allocated a fair share of water. Which farms would you prefer to live on? Why?*
- *Under the same assumption, on which farms would you least like to live?*
- *Suppose that the pumping system is very* un*reliable. Which farm would you prefer to live on?*
- *Under the same assumption, which farms would you least like to live on?*
- *Which are the three most critical links in the irrigation system?*

Definition 3.4. 1. A subgraph G' of a directed graph G is a graph (V', E') where $V' \subseteq V$ and $E' \subseteq E$.

2. A spanning subgraph G' of a directed graph G is a graph (V, E) where $E' \subseteq E$.
3. The number of vertices in V is known as the order of G and the number of edges is the *size* of$G$.
4. If (u, v) is an edge then we say that v is *adjacent* to u;

   u is adjacent to v only if (v, u) is an edge too.
5. The set of all vertices adjacent to a vertex v are the *neighbourhood* of v.
6. The number of vertices adjacent to v is called the *out-degree* of v, the number of vertices to which v is adjacent is called the *in-degree*. Informally, this means that the out-degree of a vertex is the number of edges going from the vertex, and the in-degree is the number of edges coming into the vertex.
7. A *directed walk* between vertices v and w is a set of directed edges $(v, v_1), (v_1, v_2)$, ..., (v_n, w).
8. A directed path between vertices v and w is a set of directed edges $(v, v_1), (v_1, v_2), \ldots, (v_n, w)$, where all the v_i are distinct.
9. A *cycle* is a path consisting of at least two edges from a vertex to itself.
10. A directed graph is *strongly connected* if for each v and w in V, there is a directed path between v and w.
11. A directed graph is *weakly connected* if for each v and w in V, there is an undirected path between v and w.
12. A directed graph is complete if there is an edge between every two vertices.

3.1.3 More definitions!

The definitions below apply to both undirected and directed graphs.

Definition 3.5. A *component* of a graph is a maximal subgraph which is connected.

Definition 3.6. A *bipartite graph* $G = (V, E)$ is a graph such that V can be partitioned into two disjoint sets[1], V_1 and V_2 so that each edge in E joins one vertex in V_1 with a vertex in V_2.

Definition 3.7. A *tree* is a connected, acyclic graph.

Definition 3.8. A *forest* is an acyclic graph, each of its components being a tree.

Definition 3.9. A directed, acyclic graph (DAG) is a directed graph, with no directed cycles. A DAG is only a tree if no vertex has in-degree greater than one.

Definition 3.10. A *bridge* is an edge of connected graph, which, if removed, disconnects the graph. An *articulation point* is a vertex of a connected graph, which if removed with all edges incident to it disconnects the graph. The *cutset* of a connected graph is the set of all articulation points.

Often, we can draw the same graph in different ways by labelling the vertices differently: the structure of the graph could be the same. To capture this essential similarity we use the notion of isomorphism.

Definition 3.11. Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are *isomorphic* if there exists a function $f : V_1 \to V_2$ which is both one-to-one and onto such that$(x, y) \in E_1$ if, and only if, $(f(x), f(y)) \in E_2$.

Definition 3.12. The *complement* of the graph $G = (V, E)$ is the graph $\bar{G} = (V, \overline{E})$ where $(x, y) \in \overline{E}$ iff $(x, y) \notin E$.

3.1.4 Weighted graphs

Often, we are not only interested in the fact that two vertices are adjacent, but wish to quantify the relationship between them. For example, if we are representing a road system as a graph, with towns as vertices and roads as edges, then we would also want to specify the distance between the towns by associating a distance with the edges. This is exactly what a weighted graph is. Each edge is given a weight which specifies the cost or reward of traversing the edge. More formally, a weighted graph G is a graph (V, E) together with a cost function $c : E \to \mathbb{R}$. For each edge in the graph, the cost function returns a number indicating the weight of the edge. Very often, we restrict the cost function so that its range is only the positive numbers or positive integers. From now on, unless otherwise specified, we assume that the cost function's range is the set of positive real numbers.

[1]Two sets are disjoint if their intersection is empty.

Exercise 3.3. *Take the graph in exercise 3.2, and turn it into a weighted graph.*

- *first let the weights be maximum possible flow that a pipe can take;*
- *then let the weights indicate the distance between the vertices. You can see from this that in some cases the larger the weight of an edge the "better" and in some cases, the smaller the "better".*

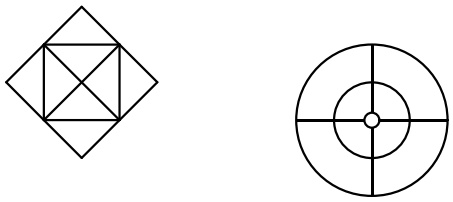
3.2 Simple examples

3.2.1 Work — PERT charts

Performance evaluation and review technique (PERT) charts are useful tools in many management problems, including the planning and management of large computer programs. A simplified description of the chart works follows. Divide the job which has to be completed into a number of tasks. Some of these tasks will depend on the successful completion of other tasks, while some tasks will be able to be done in parallel. Represent each task by a vertex in the graph. The directed, weighted edges between vertices indicate how long after one task has completed the other one can be. Questions that you might want to ask include: which jobs should be done in parallel? what is the critical path (the longest path through the graph) which indicates how long the job will take? This information can tell us things like which jobs have slack (i.e. are not time critical) and which ones which if they take longer than they should will hold up the project considerably. With these simple examples, one can see that management planning of any task can be improved.

3.2.2 Play

You've probably all seen puzzles of this form (and these aren't just for recreation, the solutions to these problems also have practical applications). Draw the graph on the left without lifting a pen. How many continuous pen-strokes are needed to draw the graph on the right without repeating any edges.



Graph theory gives you the tools to help solve the problem or show that it can't be done. Don't say that you don't learn anything useful in computer science.

3.2.3 Science

Graph theory has many practical applications in a variety of scientific fields. One place where it has helped is in chemistry in the enumeration of certain molecules. By example, the alkanes or paraffins are those molecules with the general formula C_nH_{2n+2}. Due to the valence of carbon atoms and hydrogen atoms, this problem boils down to the question of finding the arrangements of the carbon atoms. For example, butane and isobutane both have the same chemical formula C_4H_{10}, but have different arrangements of carbon and hydrogen atoms. This reduces to finding all the trees with n elements where each vertex has degree four or less. Graph theoretic techniques helped to solve this and other problems.

3.2.4 Finite state machines

Finite state machines (FSMs) and other automata have many uses for representing and specifying systems in computer science. Graphs are natural representations for finite state machines (the vertices are the states, the edges the transitions). Many operations on FSMs and questions about FSMs can be solved using graph algorithms.

3.2.5 Theory

Of course there doesn't have to be a point to studying graph theory. Some people study graph theory as a branch of mathematics because it's there.

3.3 Computer representations of graphs

Mathematicians can get away with using set notation and drawing pictures in order to depict graphs. Computer scientists need to do better because we use graphs to represent problems that we need to solve. There are two very common ways of representing graphs, which are explained here. A third method that is very useful in many applications is also discussed briefly. However, you should not think that these are the only ways in which graphs can be represented.

3.3.1 Adjacency matrix

One way to represent a graph is to use an adjacency matrix. If we have a graph $G = (V,E)$, of degree n, with $V = \{v_1 \ldots, v_n\}$, we can represent the graph by an n $\times$ n matrix A, where A is defined by $A_{ij} = \mathbf{t}$ if $(v_i, v_j) \in E$ and $A_{ij} = \mathbf{f}$ otherwise.

For a weighted graph, this is changed slightly to: A is defined by $A_{ij} = c((v_i, v_j))$ if $(v_i, v_j) \in E$ and $A_{ij} = \hat{c}$ otherwise, where $\hat{c}$ is a constant which would depend on the nature of the problem. If we think of the weights as costs, such as the distance between two towns, then we let $\hat{c} = \infty$; if we think of the weight as a capacity (such as the rate a water pipe can pump), then we let $\hat{c} = 0$. In an undirected graph, the adjacency matrix is symmetric $A_{ij} = A_{ji}$, but this is not the case in a directed graph.

Exercise 3.4. *Take one of the directed graphs which you found in exercise 3.2, and draw the adjacency matrix for it.*

Adjacency matrices are a natural representation of graphs, and can be used in many algorithms. They do have two drawbacks. Both drawbacks stem from the fact that for a graph with n vertices and m edges, the adjacency matrix has to have n^2 entries. This means that for a fairly large graph, even if $m \ll n$, the adjacency matrix gets big very quickly. Obviously, this means that the space taken to represent the graph can be considerable. More than that though, many algorithms need to process all the edges. This will imply that if an adjacency matrix is used, the algorithms will run in $\Omega(n^2)$ time, when in fact they could run in $\Theta(m)$ time.

Java representation of adjacency matrix: We can just represent the adjacency matrix directly in Java as an array. For unweighted graphs, this would be:

```
bool adj_matrix [][];
```

while for weighted graphs this would be:

```
int adj_matrix [][];
```

3.3.2 Adjacency list

The second general way of storing a graph is to use an adjacency list. For every vertex, we keep a linked list of the vertices adjacent to it. This representation uses much less space — $\Theta(n+m)$ — and also allows many (but not all) algorithms to run in $\Theta(n+m)$ time. There are times, however, when it is not the most convenient representation. And, in the worst case, where $m \approx n^2/2$, the adjacency list representation is not better than the adjacency matrix representation.

Java representation of adjacency list There are more elegant ways of doing this, but the essence of the adjacency list is shown below:

- We use the class *Edge* to represent an edge. This just contains two vertex labels. Note, here we have a relatively rare example of a class with non-private data members and no methods. (For those familiar with C or Pascal, this is effectively what a record is.)
- *EdgeList* can be used to represent an edge. It is based on the idea that a list of edges, is just an edge, followed by a list of edges.

```
class Edge {
    int x, y;
    int weight; // omit if not a weighted graph
}
```

```
class EdgeList {
    public Edge e;
    public EdgeList next;
}
```

One disadvantage of this is that it is slightly more complicated than the adjacency matrix.

Exercise 3.5. *Write a Java program which can read in a file from disk, and the print out the adjacency matrix on to the screen. Assume that the graph is weighted and undirected.*

3.3.3 Logical expression representation

Another representation of a graph is to use boolean expressions. For example, the directed graph in (V,E) defined by

- $V = \{a,b,c,d\}$;
- $E = \{(a,b),(b,c),(c,d),(d,b),(c,a)\}$.

could be represented by the boolean function:

$$\begin{aligned} e(x,y) \stackrel{\text{def}}{=} & (x = a \wedge y = b) \vee (x = b \wedge y = c) \vee \\ & (x = c \wedge y = d) \vee (x = d \wedge y = b) \vee \\ & (x = c \wedge y = a) \end{aligned}$$

We can use the function e to determine whether there is an edge between two vertices: it returns true when there is an edge between vertices and false otherwise.

This may seem to be an inconvenient representation, but there are many efficient data structures for representing boolean data [2], and for some graph applications, this is an excellent mechanism for graph representation and manipulation.

3.4 Summary

This chapter gave the basic definitions of graph theory, illustrated these definitions with examples, and gave some further motivation for the study of graph theory. Armed with these definitions, we can start looking at graph theory algorithms.

The references used for this section were: [1] for some of the motivation, definitions and some of the discussion of representation; [3] for one of the examples; and [6] for some of the definitions and examples.

Exercise 3.6. ***Graph class hierarchy:*** *Design a Java class hierarchy for graphs. Graphs can be directed or undirected. Edges can weighted or unweighted. What criteria should you use for the design of your hierarchy? When should you have a derived class?*

Chapter 4

Putting the Ideas into Practice

4.1 Introduction

In this chapter, we explore two case studies. These case studies serve a number of purposes:

- They illustrate the range of applications of graph algorithms.
- They show us the process of going from an understanding of a problem to a high-level design, to a more detailed design, to a program.
- They show us common algorithmic idioms.
- We see how algorithm analysis is done in practice, and the how the choice of data structure affects the efficiency of an algorithm.

4.2 Graph Colouring

Graph colouring is an important area in graph theory because it is a rich source of theoretical and practical interest. In this section, we shall have a quick introduction to the problem of graph colouring and applications of graph colouring, before examining a graph colouring algorithm in some detail.

To colour vertices means to assign colours to vertices (have a function mapping from vertices to a set of colours). Colours are abstract concepts: they could be sets of what we think of as colours, or just an arbitrary labels like integers (*Roses are 0, violets are 1, etc.*).

Consider the simple graph in Figure 4.1, and two possible sets of colours:

- Let $C_1 = \{\text{red}, \text{blue}, \text{green}\}$.
- Let $C_2 = \{1, 2, 3\}$.

One way of assigning colours to vertices is $f(A) = f(B) = \text{blue}$; $f(C) = f(D) = \text{red}$ and $f(E) = \text{green}$. Another way is $f(A) = f(E) = 1$; $f(B) = f(C) = 2$; $f(D) = 3$

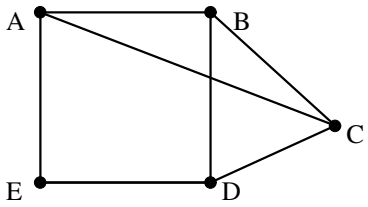


Figure 4.1: Simple graph

A *colouring* of a graph is the colouring of the vertices so that no two adjacent vertices have the same colour. A colouring that uses n colours is called an *n-colouring*. The minimum number of colours in a colouring of G is called the chromatic number of G and is denoted $\chi(G)$. Usually there are many ways to colour a graph.

Definition 4.1.
- *Denote the complete graph of order n (with n vertices) as K_n. Note that each vertex in K_n has degree $n-1$*
- *A bipartite graph $G=(V,E)$ is a graph such that V can be partitioned into two disjoint sets, V_1 and V_2 so that each edge in E joins one vertex in V_1 with a vertex in V_2. A graph is a* complete bipartite graph *if every vertex in V_1 is adjacent to every vertex in V_2: if the number of vertices in V_1 is r, and the number of vertices in V_2 is s,then this is denoted $K_{r,s}$*

Exercise 4.1.
1. *Colour the graph shown above.*
2. *What is the chromatic number of K_5?*
3. *What is the chromatic number of $K_{3,3}$?*

The results of the exercise can generalised: $\chi(K_n)=n$, and a graph G is bipartite if, and only if, $\chi(G)=2$.

We say that a graph G is *k-colourable* if $\chi(G)\leq k$.

Theorem 4.1. *Let $G=(V,E)$ be a graph. If the vertex with greatest degree has degree r, then the graph is $(r+1)$-colourable.*

Proof: The proof is by induction on the number of vertices. Let G be a graph

- *Base case: n*=0. Maximum degree is 0, so result follows directly
- *Induction hypothesis*: If the number of vertices in G is n, then the graph is $r+1$ colourable.
- *Induction step*: Suppose G has $n+1$ vertices, and let v be a vertex of degree r (i.e. a vertex of maximum degree).
  - Delete v with the edges incident to it from G to get the graph G'.

- There are n vertices left in G'; so by the induction hypothesis G' is $(r+1)$-colourable.
- This gives each vertex in G adjacent to v a colour.
- But, v has at most r neighbours of v in G, so $\exists$ a spare colour for v.

Theorem 4.2. *Every planar graph is* 4*-colourable.*

This last theorem is one of the most famous theorems in mathematics and computer science. Since this theorem was first posed a century ago, much work was done into trying to prove it true or false. Although no counter-example could be found, the proof was elusive. Finally in the 1970s, two mathematicians showed that all planar graphs were represented by about twelve hundred special cases, and that if each of these special cases could be 4-coloured, then so could all planar graphs. Then, by using a computer and doing exhaustive searches a colouring was found for each special graph, thereby showing that all planar graphs are 4-colourable. A consequence of this theorem is that all maps can be drawn with four colours so that no two adjacent countries have the same colour. For a more detailed discussion of this see [8].

4.2.1 Vertex Colouring and complexity

Although apparently a simple problem, vertex colouring is very difficult.

> **The Three Colour Problem**: Given an arbitrary graph G, determine whether it is possible to colour it with three colours.

Theorem 4.3. *The Three Colour problem is NP-complete.*

Proof: `Do honours.`

Conclusion: there is no general, efficient way of finding the chromatic number of a graph.

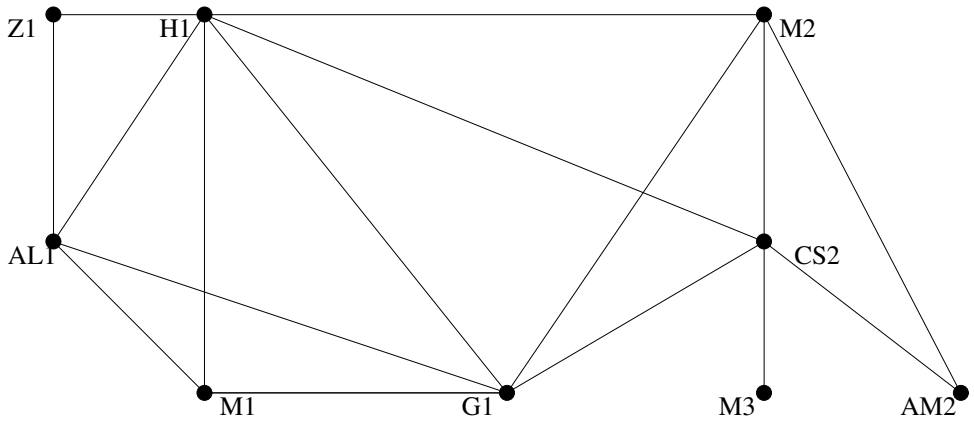
4.2.2 Applications of colouring

Colouring can be used for all sorts of scheduling: courses to class rooms, variables to registers etc. Here's an example. Suppose we have six students: A, B, C, D, E and F, each writing a number of exams. Schedule the exams so that there are no clashes.

- A writes Computer Science II, Maths II and History I
- B writes Computer Science II, Maths II and Applied Maths II
- C writes Computer Science II and Maths III
- D writes History I, Zulu I and African Literature I
- E writes Geography I, History I, Maths I and African Literature I
- F writes Computer Science II, Maths II and Geography I

Solution – use colouring:

- Create a graph with a vertex for each exam.
- There is an edge between two vertices if they have a common student.



Applying a colouring algorithm to the graph, we find a colouring. Each colour stands for a slot in the exam time-table, and so the colour of the vertex representing a course tells us what slot the exam should be scheduled in. This works because if two vertices/courses have the same colour, they cannot be adjacent, and so by the way the graph is built they cannot have any common students. Therefore, their exams may be scheduled in the same slot. The algorithm presented below finds the following colouring.

Slot	0	1	2	3	4
	History I	African Lit I	CSII	Maths II	Geog I
	Maths III		Zulu I		Applied Maths II
			Maths I		

In this very simple example, we could easily have found a time-table manually (and maybe even a better time table) more quickly than using the algorithm we did here. But, consider an institution like the University with thousands of courses and well over 10 000 undergraduate students. Manual preparation of a time-table would be impossible. We might need to extend the algorithm to take into account problems like not wanting students to have exams in successive slots. It might also be important to have a good colouring algorithm as venues and slots available may be a problem.

4.2.3 An approximate colouring algorithm

It's not possible to find an optimal colouring algorithm that is efficient, but much work has gone into finding good heuristic algorithms. These algorithms are approximate in that although

```
ApproxColour(V,E) {
  Sort the vertices so that deg(v_i) ≥ deg(v_{i+1});
  colour(v_0) = 0
  for (i = 1; i < n; i++) {
      colour(v_i) = Smallest colour so no clash;
  };
};
```

Figure 4.2: Approximate colouring algorithm

they find colourings, they number of colours used may be more than is necessary. A simple approximate colouring algorithm is given in Figure 4.2. We use this algorithm as an example. This algorithm is a fairly high-level algorithm as major steps are left without detail. This is a good way to start, in line with our approach of using abstraction and divide and conquer.

Exercise 4.2. *Apply this algorithm to the example given previously.*

There are two major parts of the algorithm that need refinement. We need to write code that does sorting, and then we need to show to implement the line that says to colour a vertex with the smallest colour that does not clash the vertex's neighbours.

A. Sorting

Sorting is one of the most useful and common algorithms. Sorting has been studied for many years and there are lots of different sorting algorithms with different advantages and disadvantages. In this section, we shall look at Insertion Sort — this is a very simple sort, but it is not suitable for many applications because is not an efficient algorithm.

The basic philosophy of the Insertion Sort is to keep the list to be sorted into two parts: the sorted part and the unsorted part. Initially the sorted part contains only the first element of the list, and the unsorted part contains the whole list; at the end, the sorted part contains the entire list and the unsorted part is empty.

The algorithm contains a main loop: at each iteration of this loop we take the first element in the unsorted part and *insert* it into the sorted part at the right place.

From this description we can see why the algorithm works and that it finishes (as long as we implement the algorithm correctly):

- The *invariant*: The part we call sorted *is* always sorted. Initially this is true trivially because the sorted part contains only one element. We must make sure that after each iteration the invariant continues to be true by inserting each element in the correct position. Then, at the end if the sorted part contains the entire list then we have sorted the entire list.

- The *variant*: The unsorted part shrinks after each iteration of the algorithm, and eventually will become empty. We make sure the algorithm terminates when the unsorted part becomes empty.

A(i) – Specification of the code: Here is the specification of the insertion sort code.

```
void insertion_sort(int numbers[],int sindex[]);
```

- Initially: *numbers* contains a set of integers
- Finally: *sindex* gives the permutation of the index into the array to make *numbers* sorted. That is,

  sindex[0] is the *index* of the largest number in *numbers* and *sindex*$[n-1]$ is the *index* of the smallest number in *numbers* (assuming that it contains n numbers). More generally,

  For $0 \leq i \leq n-2$, *numbers*[*sindex*[i]] $\geq$ *numbers*[*sindex*[$i+1$]].

A(ii) – Outline of code: Here is an outline of the code:

```
void insertion_sort(int numbers[],int sindex[]) {
      int next;
      sindex[0] = 0;
      for (next = 1; next < numbers.length; next++){
          insert(numbers[next]);
      }
}
```

A(iii) – Full implementation of code: Now we can refine this even further to a complete Java program, found in Figure 4.3.

Analysis of the algorithm: Suppose there are n elements in the array.

Look at the inner loop: in the worst case it iterates `next` times (when does this happen?), and in the best case it never iterates. Thus the cost in the worst case is `next` $\times c_1$ and in the best case is c_2 for some constants c_1 and c_2.

The value of the variable `next` is determined by which iteration of the outer loop we are in: it varies from 1 to $n-1$. For convenience, let's say the value of `next` is j. Thus the cost of iterating the outer loop in the worst case is $jc_1 + c_3$ taking into account the other parts of the outer loop, and in the best case it is $c_2 + c_3$.

This leads us to the following analysis.

- Worst case: Cost is $\sum_{j=1}^{n-1} jc_1 + c_3 = n(n-1)c_1/2 + nc_3 \approx n^2c_1/2 + nc_3 \in O(n^2)$.
- Best case: Cost is $\sum_{j=1}^{n-1} c_2 + c_3 = n(c_2 + c_3) \in O(n)$.

Thus in the worst case this algorithm is $O(n^2)$ and in the best case it is $O(n)$. A little more analysis can show that on average (assuming randomly distributed data) the insertion sort is $O(n^2)$.

```
static void insertion_sort(int nums [], int sindex[] ) {
    int next, curr, i;

    sindex[0]=0;
    // Initially the sorted part is the first elt, the unsorted
    // part the rest

    // Invariant: nums[sindex[j-1]] >= nums[sindex[j]] 0 <= j < next
    for (next=1; next < nums.length; next++) {
        curr = nums[next];
        i = next;
        while (i > 0 && nums[sindex[i-1]] < curr) {
            sindex[i] = sindex[i-1];
            i--;
        }
        sindex[i]= next;
    }
}
```

Figure 4.3: Insertion sort code

B. Finding the smallest colour that doesn't clash

Now we have finished that task, we can turn our minds to the other part that needs attention. At each step we look at the next vertex in the sequence that needs colouring and then look at all the neighbours of that vertex. We choose as a colour for our vertex the smallest colour that is not used by the neighbours. The table below shows some examples:

Colours of neighbours	Colour to use
0, 1, 2, 3, 4	5
0, 1, 2, 3	4
0, 1, 2, 4	3
1, 2, 5, 6	0

My solution works as follows. An array `used` keeps track of which colours are used by neighbours: `used[j]` is true if colour *j* is used, and is false if colour *j* is not used. Each time we call `FindSmallestFreeColour`, each entry of the array `used` is set to false (this is the default of the allocation of an array). We then check the colour of each neighbour that has been coloured and mark off the colour as used. Once we have processed all the neighbours, we search the `used` array to find the first unmarked entry. Here's an outline of the solution.

```
void FindSmallestFreeColour(v) {
   for each neighbour w of v that is coloured
       mark colour of w used

   find first free entry in used.
}
```

The next step is to implement the code. A few notes, first:

- In the code above, we haven't said how the graph is represented. In fact, the algorithm that we develop is independent of the representation. We assume there is a *Graph* class that has methods *degree* and *isedge*.
- We shall use an array `colour` that stores for each vertex the colour that has been used. Initially, each entry in the array is set to -1 to indicate that it has not yet been coloured.
- In the code above we see a common example of a statement in a high level algorithm:

  `for each neighbour w of v that is coloured`

  This statement implies repeating over elements as well as checking conditions. There are different ways in which this can be translated into a programming language. A common one is to have a loop that enables us to iterate over all elements that could be of interest and to have an **`if`** statement to check the conditions.
- When we move from pseudo-code to program we have to take care of book-keeping (ensuring parameters are passed, variables declared and initialised) that may be under-specified in the pseudo-code.

Exercise 4.3. *Consider the code in Figure 4.4.*

1. *The last loop of* FindSmallestFreeColour *reads*

```
while (used[w])
    w++;
```

   The correctness of this relies on the fact that in the range $0,\ldots,N-1$*, there is at least one* `used`$[j]$ *that is false. Why can we make this assumption?*

2. FindSmallestFreeColour *uses an auxiliary array* `used` *to keep track of which colours are used. We need not use any extra space or variables and still find the smallest colour not used. However, the code will be longer and more expensive. Give the alternative solution.*

C. Analysis of the algorithm

We have already seen that insertion sort is $O(n^2)$ in the worst and average case. *FindSmallestFreeColour* runs in $O(n)$ time each time it is called. The main loop of *ApproxColour* is executed n times, hence this entire algorithm is $O(n^2)$ in the worst case.

Exercise 4.4. *What is the performance of this algorithm in the best case?*

Note on approximate colouring: This algorithm is efficient, though it does not always produce good colourings. Other approximate algorithms exist and produce better answers but more costly. Actually approximate colouring of graphs is very difficult. It can be shown that unless P=NP, no polynomial algorithm to colour graphs can always use fewer than $2\chi(G)$ colours [5]!

```
static int FindSmallestFreeColour(Graph g, int colour [], int v) {
    int w, N;
    boolean used [];

    N = g.getNumVertices();
    used = new boolean [N];

    for (w=0; w<N; w++) {
        if (g.isedge(v,w)  && colour[w] != -1) {
            used[colour[w]] = true;
        }
    }

    w=0;
    while (used[w])
        w++;
    return w;
}

static int []  ApproxColour(Graph g) {
    int N = g.getNumVertices();

    int v, i, new_colour;
    int colour [], degree [],  sindex[];

    colour = new int [N];
    degree = new int [N];
    sindex = new int [N];
    for (v=0; v<N; v++) {
        colour[v]=-1;
        degree[v] = g.degree(v);
    }

    Sort.insertion_sort(degree, sindex);
    colour[sindex[0]]=0;
    for(i=1; i<N; i++) {
        v = sindex[i];
        new_colour = FindSmallestFreeColour(g, colour, v);
        colour[v] = new_colour;
    }
    return colour;
}
```

Figure 4.4: Program for Approximate Colouring

4.2.4 Comment on method

This worked example shows how we can solve a problem, starting with a theoretical understanding of the problem, starting with a high-level design and then refining the solution to more a more detailed algorithm and program. We also saw the use of divide-and-conquer. When we were working out a sorting algorithm, we just focussed on that, and didn't worry about other aspects of the program or on graph colouring. When we worked out the *FindSmallestFreeColour* method we didn't worry about the detail of the sorting algorithm.

4.2.5 Alternative implementation

In the sorting algorithm I wanted to show you that sometimes it is useful to introduce a level of indirection, and we sorted the index array. However, there are alternatives, especially in object-oriented languages like Java or languages which allow functions to be passed as parameters (such as C, Pascal and functional languages).

In an object-oriented language, when we define a class, we describe what objects look like and their behaviour: in Java this means describing the instance variables and the method for objects. We'll define a vertex class. Here's the first part of it:

```
class EdgeList {
   Vertex y;
   EdgeList next;
}

class Vertex implements Comparable {
    int index;
    int colour;
    int degree;
    EdgeList adj;

    public Vertex(int v) {
      colour = -1;
      adjlist = null;
      index = v;
    }
```

With each vertex we'll store its colour, its degree, as well as its adjacency list. We also store the index of the vertex with it – this is slightly redundant, but it will make our work easier. We also have a simple constructor. I'll explain what *implements* means in a minute.

The main data structure in the Graph class in an array of vertices. We no longer have a separate adjacency list, since each vertex stores its own adjacency list.

```
class Graph {

    int numvertices;
    Vertex vertex [];
```

```
..
..
}
```

In the *Vertex* class we define methods that describe how to compare one vertex with another. In Java, there is a recommended way of doing this (you don't have to follow it, but it makes sense and there are language facilities to help support the process).

- Define a method `compareTo` in the vertex class.

  `x.compareTo(y)` will return a negative number if `x` should come before `y`, 0 if they are 'equal' in the ordering, and a positive number if `x` should come after `y`.

We have the freedom to decide how to implement this method depending on what we want to capture. We'll compare vertices based upon their degree. `x.compareTo(y)` $== -1$ will mean that `x` has greater degree than `y`, and so should come before `y` in the ordering.

```
public int compareTo(Object o) {
  Vertex v;
  v = (Vertex) o;
  return degree - v.degree;
}
```

The *implements* instruction indicates that this method honours the *Comparable* interface. A discussion of interfaces is beyond the scope of these notes, but essentially it is a promise that the class has the `compareTo` method. The requirement that the parameter be an *Object* which we have to cast, comes from the promise in the interface. The sort method now becomes:

```
static void insertion_sort(Object o []) {
    int next, curr, i;

    for (next=1; next < o.length; next++) {
        curr = o[next];
        i = next;
        while (i > 0 && curr.compareTo(o[i-1])==-1) {
            o[i] = [i-1];
            i--;
        }
        o[i]=curr;
    }
}
```

We'll move the *ApproxColour* routine into the Graph class:

```
void ApproxColour() {
  int i, v;
  Vertex vcopy [] = new Vertex[numvertices];
  for (i=0; i<N; i++) vcopy[i]= vertex[i];
```

```
    Sort.insertion_sort(vcopy);

    v = vcopy[0].index;  // vertex of highest degree
    vertex[v].colour=0;
    for(i=1; i<N; i++) {
        v = vcopy[i].index;
        new_colour = FindSmallestFreeColour(g, colour, v);
        vertex[v].colour = new_colour;
    }
}
```

Some notes:

- We use `vcopy` as a copy of the vertex array because the elements in the array are going to be moved around and we don't want to mov around the elements in the `vertex` instance variable.
- Although we are making a copy of the array, note we are not making copies of the vertices:
- Rather than our defining our own sort routine, it would be easier and more efficient to use a sorting routine that comes in the `Arrays` class, which can be imported from `java.util.arrays`.

  We would say `Arrays.sort(vcopy`. Note this relies upon the fact that there is a `comparesTo` method defined on the *Vertex* class.

  The advantage of doing this is that we no longer have to implement a sorting routine. And it's not as if we are particularly wedded to the $O(n^2)$ insertion sort either – Sun's implementation of `sort` is a quicksort, which while is $O(n^2)$ in the worst case, is $O(n \log n)$ on most real data sets.

4.3 Stable Marriage Problem

Our second example comes from a different application of graph theory – namely matching. The basic idea of matching is that we have two sets A and B and we wish to pair elements in A with elements in B according to some criteria.

Matching is often conveniently expressed as a bipartite graph. All elements of A and B are vertices in the graph, and we have edges between $x \in A$ and $y \in B$ if x and y can be paired off. The problem of matching is to select a subset of the edges.

We'll study the matching algorithm in Chapter 8 once we have done some more graph theory. Here, we'll study a special case of matching – the stable marriage problem. Strictly, this doesn't have to be viewed as a graph algorithm, but if fits nicely into matching and is a nice example for our use now.

We'll again see how we can design a high-level algorithm to solve a problem once we have analysed the problem. We'll see how to turn the high-level algorithm into a more detailed algorithm and then into Java code. We'll also see the choice of data structure can have a big impact on the performance of the algorithm.

In many cases we want to match the elements in such a way as to optimise some measure of 'happiness' or cost. For example, in allocating lecturers to courses, we could just do it on the basis of who can teach what. However, as anyone who has had to do this work allocation in practice knows, preferences are very important in ensuring a happy department.

Even if we are cold-hearted about this and ignore 'happiness', from a system perspective it is critical to take preferences into account. This was illustrated in one of the earliest large-scale examples of this problem: the hospital/interns problem which was faced in the US about fifty years ago. After graduating, doctors were (and are) required to spend a one year internship in a hospital. At the time, there was no centralised system of allocating interns to hospitals. The result was that interns would apply separately to a number of hospitals, which then individually decided whom they wanted.

This led to a situation of tremendous instability. Hospital A would make an offer to a candidate who would accept. On that basis they would turn down their other applicants. Then their candidate would get made a better offer shortly before starting. Result: a whole lot of unhappy hospitals without interns and interns either with no jobs or undesirable ones.

4.3.1 Formalisation of the problem

We look at the following abstract version of the problem. We are given:

- B, a set of n boys;
- G, a set of n girls;
- Each $b \in B$ has a strictly ordered list of preferences of the gs in G.
- Each $g \in G$ has a strictly ordered list of preferences of the bs in B.

We say that g prefers b_1 to b_2 (written $b_1 <_g b_1$) if b_1 appears before b_2 on g's preference list (and similarly for boys).

A matching M is a bijection (one-to-one, onto function) between B and G (or from G to B). If M matches b and g then b and g are partners in M. This can be written as:

- $p_M(b) = g$, $p_M(g) = b$; or
- $(b,g) \in M$.

A b and a g *block* a matching M if b and g are not matched in M and they prefer each other to their partners in M. This condition can be written formally as:

$$(b,g) \notin M \wedge g <_b p_M(b) \wedge b <_g p_M(g).$$

A matching that has a blocking pair like this is called *unstable* (since at the first opportunity, they'll desert their partners). If M has no blocking pair then it is a *stable* matching. A stable matching will always exist — usually many stable matchings exist.

A note on notation: We use the terminology *boys* and *girls* for convenience, but they should only be considered as notational conveniences. The problem (and algorithm to follow) generalises in desired or desirable ways.

4.3.2 Data structures for the problem

We need to have data structures to represent the preferences of the boys and girls. A simple approach is to use $n \times n$ matrices (one for the boys' preferences, one for the girls').

There are two ways in which we might want to look at the preferences: (1) who is girl g's i-th preference; and (2) where does girl g rank boy b.

Preference arrays: The array `gp` records the girls' preferences with $\texttt{gp}(g,i)$ recording girl g's i-th preference. So, $\texttt{gp}(g,0)$ is girl g's favourite boy, and $\texttt{gp}(g,n-1)$ her least favoured. Similarly, we'll use an array `bp` to represent the boys' preferences.

Ranking arrays: We can also use an array `gr` to represent the rankings. $\texttt{gr}(g,b)$ is girl g's ranking of boy b. So if g prefers b_1 to b_2, then $\texttt{gr}(g,b_1) < \texttt{gr}(g,b_2)$. Similarly, we'll use an array `br` to represent the boys' rankings.

Note that arrays `gp` and `gr` contain the same information, and the one can be constructed from the other. The difference between the two is how quickly we can find information. To discover whether girl g prefers b_1 to b_2 can be done in $O(1)$ time using the ranking matrix `gr`, but in $O(n)$ time using the matrix `gp`. And to discover who g's favourite is takes $O(1)$ time using `gp`, but $O(n)$ time using `gr`.

Which is used depends on the types of queries done more often. Of course, we could use both at the cost of extra memory – this might be an appropriate price in many situations.

4.3.3 Checking stability

Before, looking at how to find a stable matching, let's look at how to check whether a given matching is stable. The algorithm checks whether there are any blocking pairs.

For each girl we consider the boy to whom she is partnered, and *all* the boys she prefers to this partner. If one of the boys she prefers to her partner likes her better than *his* partner then we have found a blocking pair and so the matching is unstable. If we find no blocking pairs then the matching is stable. Here is the abstract algorithm.

```
boolean StabilityCheck(M)
  foreach g ∈ G {
     foreach b ∈ B ∋ g prefers b to p_M(g) {
        if (b prefers g to p_M(b)) {
          return false ;
        };
     };
  };
  return true ;
};
```

This is a fairly abstract algorithm. Let's try to make it more concrete. In making it more concrete, we need to look at how to represent the boys and girls as well as efficiently extract the information we need.

- The simplest way to represent the boys and girls is to number them each from 0 to $n-1$;
- This implies that the line **foreach** $g \in G$ can be implemented as a simple **for** loop;
- How about finding all the boys a girl g prefers to her partner $p_M(g)$?

  A high-level algorithm often says something like *for each x such that* .... To turn this into a program we commonly use a loop, coupled with a selection (typically using an **if** statement) .

  For convenience let's denote $p_M(g)$ as $\widehat{b}$ (i.e. $\widehat{b}$ is g's matching in M).

  1. We could use the ranking array `gr`. Find the rank of boy $\widehat{b}$, and then loop over all the boys to find which ones have a better (lower) rank.
  2. We could use the preference array `gp`. Suppose g ranks $\widehat{b}$ as number j. (Thus $\widehat{b} = \texttt{gp}(g, j)$.)

     Then g prefers boys $\texttt{gp}(g,0), \ldots \texttt{gp}(g, j-1)$ to $\widehat{b}$.

     So, we could just loop over the g's preferences until we find $\widehat{b}$.

  Both of these are $O(n)$ in the worst case, but the latter is more efficient because we only need to search part of g's preferences rather than all.

- How do we check to see whether a boy prefers one girl to another? We could use either the preference array or the ranking array – but the former takes $O(n)$ work, whereas the latter takes $O(1)$ work. Here we'll use the ranking arrays.

```
1  boolean StabilityCheck(M)
2    for (g = 0; g < n; g++) {
3         for (p = 0; gp(g,p) !=p_M(g); p++) {
4              b = gp(g,p);
5              if (br(b,g) < br(b,p_M(b)))
6                return false ;
7         };
8    };
9    return true ;
10 }
```

Clearly, this algorithm could be presented symmetrically from the boys' perspective.

Note how the instruction in the abstract algorithm

foreach $b \in B \ni g$ prefers b to $p_M(g)$

was translated into

for $(p = 0;\ \mathtt{gp}(g,p)\ !=p_M(g);\ p{+}{+})\{$

It is obvious that the **foreach** implies a loop. The 'such that ...' implies some selection must take place. By the nature of the preference arrays we do not have to explicitly perform the selection: all the boys who fall into the first part of girl g's preference array are preferred to her current match. However, if we use a ranking array, then selection must be done explicitly.

Exercise 4.5. *Rewrite the code using ranking arrays only.*

A. Invariants and variants

This is a fairly advanced section, and should be skipped at first reading. To ensure that the loops above are correct, variants and invariants need to be found. Let's look at the inner loop first (lines 3–7).

The variant here is almost straight-forward because we have a for loop. The value of p is set to 0 and then incremented. There is an upper-bound on the value of p, and hence the loop must terminate because the condition guarding the inner loop must become false (i.e. $\mathtt{gp}(g,p) == p_M(g)$ before p has the value n). It must become false because of the requirement that every girl ranks every boy and hence boy $p_M(g)$ must appear somewhere in g's preference list. Formally, the variant is the expression $n - \mathtt{gr}(g, \mathtt{gp}(g,p))$. After each iteration, the expression decreases by 1. And since it can never be less than 0, we know the loop must terminate.

The invariant is that either:

- for the girl g, that she and every boy on the preference list ranked better than p do not form a blocking pair; or
- girl g and boy $\mathtt{gp}(g,p)$ forms a blocking pair.

This is true trivially when $p = 0$, since there is no boy that girl g prefers to boy $\mathtt{gp}(g,p)$. We know that girl g prefers boy $\mathtt{gp}(g,p)$ to her current fiancé, and so in each iteration we check to see how $\mathtt{gp}(g,p)$ feels about g. If he prefers her to his current partner, then we exit the loop and return false (and so hence the second condition of the invariant becomes true); if he doesn't then they don't form a blocking pair and so if we increase p by 1, then the first condition of the invariant is true.

Exercise 4.6. *Find the variant and invariant of the outer loop.*

B. Analysis of algorithm:

The condition on line 5 executes in each iteration of the inner loop. The inner loop iterates 0 times in the best case (when the girl is given her favourite choice) and n times in the worst case (when the girl is given her least favourite). The outer loop executes n times. Thus the algorithm is $O(n^2)$ in the worst case (which occurs whenever there *is* a stable marriage, emphasising that *worst* means most expensive in computing, not undesirable in result).

4.3.4 Finding stable marriages: The Gale-Shapley algorithm

The stable marriage problem has a fairly simple solution – the Gale-Shapley algorithm — , which is presented below. The version below is the girl-oriented version.

The way it works is that the girls successively make proposals to boys (who may or may not be engaged). If a girl proposes to a boy who is free (not yet paired), the boy 'accepts' the proposal.

If the girl proposes to a boy who is already paired, then if this proposal is a better proposal from the boy's perspective, then the boy breaks his previous engagement and accepts the new one. However, if the proposal is not a better one, the boy rejects the proposal.

Here is an outline of the algorithm. In the spirit of the anthropomorphising we are doing, we call the boy (girl) to whom a girl (boy) is currently partnered a fiancé(e).

```
assign each person free;
while some girl g is free do
    let b be the first boy on g's list to whom g has not proposed;
    if b is free
       pair together b and g;
    else
       if b prefers  g to his fiancee g'
           make g' free;
           pair together b and g;
       else
           b rejects g;
```

Let's try to formalise the algorithm, and start to think how to implement the algorithm more concretely. The following data structures will prove useful:

- An array p to keep the fiancés and fiancées currently partnered. When the algorithm terminates this will store the matching.
- Sets `FreeBoys` and `FreeGirls` to record those boys and girls currently without partners.
- An array `nextproposal` with an entry for each girl. This array keeps, for each girl, g, the index in g's preference list of the first boy to whom she has not made a proposal. Initially, `nextproposal` is initialised to all zeroes.

The algorithm can be found in Figure 4.5. It is a fairly concrete representation of the algorithm, though there are still some open questions — the most important of which is how to represent the sets of boys and girls. We'll come back to later, but first we'll look at a proof of correctness and an analysis of the algorithm.

A. Discussion of the Gale-Shapley algorithm

Although the steps in the algorithm seem sensible, there is no proof that the algorithm will terminate, let alone give the right answer. Here is some reassurance.

```
StableMatching
  FreeBoys = B;
  FreeGirls = G;
  nextproposal = {0,0,...,0};
  while FreeGirls ≠ ∅ {
        choose g ∈ FreeGirls;
        b = gp[g][nextproposal[g]];
        nextproposal[g]++;
        if (b ∈ FreeBoys) {
           p[g] = b; FreeGirls = FreeGirls − {g};
           p[b] = g;
           FreeBoys = FreeBoys − {b}; }
        else
         if (br[b][g] < br[b][p[b]]) {
            FreeGirls = FreeGirls ∪ {p[b]} − {g};
            p[g] = b;
            p[b] = g;
          };
  }
```

Figure 4.5: The Gale-Shapley algorithm

Theorem 4.4. *For any given instance of the stable marriage problem, the Gale-Shapley algorithm terminates, and on termination, the engaged pairs form a stable matching.*

Proof: We start with the high-level outline of the proof:

1. The number of iterations of the loop cannot exceed n^2
2. Every girl is accepted by some boy.
3. Therefore, the algorithm terminates, with a matching.
4. The matching is a stable matching.

Here is a detailed proof:

1. A girl proposes to a particular boy at most once. An upper-bound on the number of proposals is n^2 (n girls $\times$ n proposals per girl).
2. No girl can be rejected by all the boys.

   Once engaged, a boy never becomes free again. So, a rejection of a girl by the last boy on her list would imply that all the boys were already engaged. This is impossible since the number of boys and girls are the same and the algorithm ensures that no girl has two fiancés.

3. Therefore the algorithm terminates as eventually the set of free girls becomes empty. When it terminates, there is a matching, with each girl paired with one boy (and vice-versa).

4. The matching is stable. Let the matching found be M.

   Suppose $(g,b) \in M$ (i.e. $p_M(g) = b$).

   Suppose g prefers b' to b. Then b' must have rejected g at some stage in the algorithm (since g asks all the boys in turn), and have paired with some g' whom he prefers to g. Thus (g,b') cannot block the matching. Since this is true for all boys that g prefers to b there can be no blocking pair containing g. And since the choice of g is arbitrary, there are no blocking pairs at all.

Note that there is some choice in the algorithm about the order the girls make proposals. However, in all cases the same matching is found. Moreover, the ordering is *optimal* for the girls.

Exercise 4.7. *Based on this discussion, work out the invariant and variant of the loop.*

B. Optimality of the solution

Theorem 4.5. *For any instance of the stable marriage problem, all possible executions of the Gale-Shapley algorithm with girls as proposers yield the same stable matching, and in this stable matching, each girl has her most favoured partner possible in any stable matching.*

Proof: Suppose during an execution, E, of the algorithm, the stable matching M is found. The proof will be by contradiction.

Let M' be a stable matching where there are girls that prefer their partners in M' to their partners in M. Let the set of such girls be denoted $\widehat{G}$.

1. During the execution of E, each girl in $\widehat{G}$ must have been rejected by the boy in M' that she prefers to her partner in M.

   Why? Girls propose in their order of preference: so each girl in $\widehat{G}$ would have proposed to their partner in M' before their partner in M. Since they didn't get their choice, those boys must have rejected them.

2. Let $g \in \widehat{G}$ be the first girl who was rejected during the execution of E by their stable partner in M'.

   Let $b = p_M(g)$, $b' = p_{M'}(g)$.

3. Suppose this rejection happened because b' prefers g' to g.

4. g' cannot have any other stable partner preferred to b' (since this is the first time that a boy rejected a stable partner, and g' makes choices in order of preference).

5. So g' prefers b' to her partner in M'.

6. Thus (b',g') blocks M'.

7. Hence M' cannot be stable.

8. Hence, as E was arbitrary, M gives each girl the most favourable partner possible among all stable matchings, and all executions of the algorithm must yield the same matching.

The version of the Gale-Shapley algorithm we have seen looks at things from the girls' perspective, so is girl-oriented. It is girl-optimal in the sense that the matching it comes up with is optimal for the girls.

A slightly stronger result is known as principle of Weak Pareto Optimality.

Theorem 4.6 (Weak Pareto Optimality)**.** *For a given stable marriage instance, there is no matching, stable or otherwise, in which every girl has a partner whom she strictly prefers to her partner in the girl-optimal stable matching produced by the Gale-Shapley algorithm.*

Proof: See [7]

Some notes:

- At first, it might seem that the algorithm favours boys because they can break engagements. However, the ability to make the proposals in the first place is what makes the algorithm favour the girls.

- The result is interesting because it is possible for the girls – although each wanting to optimise her own position – to agree on what is best collectively.

- The optimality result doesn't mean that each girl gets her favourite boy. But, any boy whom she prefers to her stable partner will desert her at the first opportunity ...

- Of course, the role of the girls and the boys can be swapped. We can then find the matching which optimises the positions for the boys and finds the boy-optimal matching.

  The – perhaps depressing – result below shows the relationship between the boy-optimal and girl-optimal stable matchings.

Theorem 4.7. *In the boy-optimal stable matching, each girl has the worst partner she can have in any stable matching.*

Proof: See [7]

C. Computational complexity

We can now finish this section with an analysis of the algorithm for its complexity. We saw in Theorem 4.4 that the main loop executes $O(n^2)$ times. How much work is done in each iteration? This depends on the choices we make for data structures. In fact, the analysis guides us to which data structures should be used. These are the most complex operations needed to be performed:

- Checking whether a given boy is free.

- Finding a free girl.
- Testing whether the set of free girls is empty.
- Adding and removing elements from sets.

The key issue then is how to represent sets. We have seen one way to represent a set already: have a boolean array, with one element in the array for each element in the set. This works very well for checking to whether an element is in the set or not. So, we could represent the set of free boys as a boolean array;

```
boolean []freeboys;
freeboys[] = new boolean[n];
```

Then to test whether boy j is free or not, we can just check whether `freeboys`$[j]$ is true or not. This means we can test membership of the set in $O(1)$ time. However, this representation is not efficient for finding an element in the set, or checking whether the set is empty. This takes $\Theta(n)$ in the worst case since we need to do a linear scan. Adding and deleting elements can be done in constant time since all we need to do is to change one boolean value in the array.

An alternative approach is to use a linked list. All the elements that are in the set are kept in the linked list. This is bad for testing whether an element is in the set or not (we may need to search the entire list – $O(n)$). But, if any element of the set will do when we want one, then this is a good approach – just take the front element off the list, which can be done in constant time. Adding to the front also takes constant time (provided we can be sure that the element is not already in the list). This implies that a linked list – in fact a stack – will be a good way to represent the set of girls.

In Java, stacks are provided in the standard java.util package as a class *Stack*. You can have a stack of any objects, and the standard operations like push, pop, checking whether the stack is empty are provided.

So, if we use a boolean array to represent the set of boys and a stack to represent the set of girls then each iteration of the loop takes constant time, and hence the entire execution of the algorithm takes $O(n^2)$ operations.

Exercise 4.8. *If we use a list representation for sets, then in general adding an element to the set is* $\Theta(n)$. *Explain why. Explain why in this application we can do it in* $\Theta(1)$.

D. Java program

Let's now turn our algorithm into Java code (there are still one or two minor syntax fixes that would be necessary before this would compile).

```
static private void
       GaleShapley(int n, int girlprefs [][], int boyrank [][],
                   int partG[], int partB[]) {
           int i, g, b;
```

```
        Stack   freegirls       = new Stack();
        boolean freeboys []     = new boolean [n];
        int     nextproposal [] = new int [n];

        for (i=0; i<n; i++) {
                freeboys[i] = true;
                freegirls.push(i);
                nextproposal[i]=0;
        }
        while (! freegirls.empty()) {
          g = freegirls.peek(); //get top of stack
          b = girlprefs[g][nextproposal[g]];
          nextproposal[g]++;
          if (freeboys[b]) {
             partG[g] = b;
             partB[b] = g;
             freeboys[b] = false;
             freegirls.pop();
          } else // check b's current engagement
             if (boyrank[b][g] < boyrank[b][prefB[b]]) {
                freegirls.pop();
                freegirls.push(prefB[b]);
                partG[g] = b;
                part[b] = g;
             }
        }
}
```

4.4 Summary

In this section we have seen real examples of algorithm development for problems with real applications. We have seen how refinement of high-level algorithms into programs is done and how decisions are made at each step. Often, a refinement step can be done in different ways, and algorithm analysis is very useful in making the choices.

Other points identified are:–

- The role of data structures is critical in the efficiency of an algorithm. Often we can represent data in different ways, and which we choose will help determine performance since the different ways often have different advantages and disadvantages.
- The use of theory in driving algorithm development is important: in understanding the problem, understanding correctness, and algorithm analysis.

In the subsequent chapters, the techniques seen in this chapter will be used many times.

Chapter 5

Trees

Trees are an important class of graphs. To recap, a tree is an acyclic, connected graph. Two things in particular make trees useful to study. First they are natural representations for many problems as their hierarchical structure models many real-life situations. Therefore, to be able to represent and operate upon trees is very useful. Second, trees have many properties which can be used in other situations. Finding a subgraph of a graph which is a tree with certain properties is the first step in solving many problems. In the first section of this chapter, we look in more detail at the properties of trees. Then, we look at spanning trees, subgraphs of a graph with useful properties. Finally, we look at search trees. By the end of this section you should be able to define what a tree is, prove properties of trees, and apply important tree algorithms to graphs.

5.1 Properties of trees

This section looks in more detail at the theory of trees. This will help later , when we examine some practical applications.

Theorem 5.1. Let T be a graph with n vertices. Then the following statements are equivalent.

1. T is a tree;
2. T is connected, and has $n-1$ edges;
3. T contains no cycles and has $n-1$ edges;
4. T is connected, and every edge is a bridge;
5. Any two vertices of T are connected by exactly one path;
6. T contains no circuits, but the addition of any new edge creates exactly one circuit.

Aside: The structure of this theorem is fairly common. At first sight it looks like this is going to be a terrible thing to prove, because we have to show that each statement is equivalent to all the others. So in the above example, we would have 30 proofs.

The way to prove something like this is to show that $(1) \Rightarrow (2)$; $(2) \Rightarrow (3)$; $(3) \Rightarrow (4)$; $(4) \Rightarrow (5)$; $(5) \Rightarrow (6)$; $(6) \Rightarrow (1)$.

In this way, we can use transitivity to show that if one of the statements is true, then all must be true; if one is false all must be false.

Proof: In a graph with only one vertex, the results are trivial, so we have to consider $n > 1$. We shall just look at $(1) \Rightarrow (2)$. The rest is left as an exercise. The proof is by induction.

1. The base case $n = 1$ has already been proved.
2. *Induction hypothesis:* Suppose that for all trees with fewer than $n > 1$ vertices $(1) \Rightarrow (2)$.
3. *Induction step:* Consider T, a tree with n vertices.
   - By definition of trees, T is connected.
   - There must be at least one vertex, v, which has only one edge incident to it otherwise there would be a cycle.
     Let the edge incident to v be e.
   - Construct T' from T by deleting v and e.
     - T' is a tree since the removal of v and e can neither introduce a cycle nor disconnect the graph.
     - T' has $n-2$ edges by the induction hypothesis.
   - Therefore T has $n-1$ edges.

Definition 5.1. Vertex a is an *ancestor* of b in the tree $T = (V, E')$ if a lies in the unique path in E' from the root of T to b. Vertex b is a *descendant* of a in the tree $T = (V, E')$ if a lies in the unique path in E' from the root of T to b.

Exercise 5.1. Simple properties of trees

1. Consider the tree in Figure 5.1. Show that each of the above statements is true for the graph.

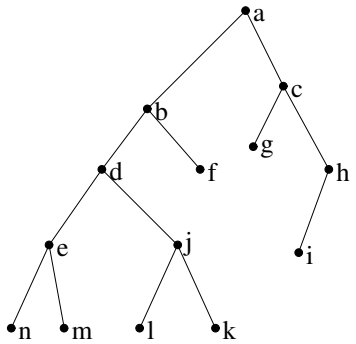


Figure 5.1: Simple tree

2. Give an example of a tree which describes the structure: universities, faculties, departments.
3. Draw a tree which is complete. Draw a tree which is a path.
4. Find some ancestor, descendant pairs.

5.2 Spanning trees

A *spanning tree* of a graph is a subgraph of the graph which contains all the vertices of the graph and is also a tree. A graph can only have a spanning tree if the graph is connected. In general, a connected graph can have more than one spanning tree. An unconnected graph has a spanning forest: a collection of disjoint spanning trees.

A *minimum weighted spanning tree* of a weighted graph is a spanning tree of a graph such that the sum of the cost of the edges of the spanning tree is less than or equal to the sum of the cost of the edges of any other spanning tree of that graph.

Exercise 5.2. Spanning trees:

- Draw a spanning tree for the graph in Figure 5.2.

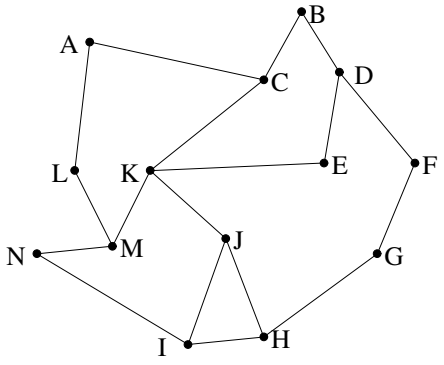


Figure 5.2: Finding a spanning tree

- Draw a spanning forest for the graph in Figure 5.2.

5.2.1 Justification

Spanning trees of graphs allow us to find spanning, connected subgraphs of a graph with no redundant edges. Practical applications are variations of problems of the form of how to lay network cables between computers so that all the computers are connected, and the cost of the cable is minimised. Finding minimum-weighted spanning trees can also be subproblems of other graph problems.

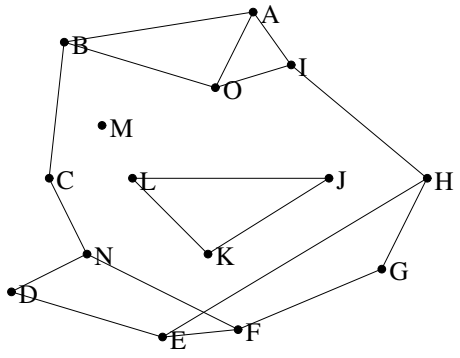


Figure 5.3: An unconnected graph

5.2.2 Algorithm

The algorithm to find the minimum-weighted spanning tree is an example of what is known as a *greedy algorithm*. Greedy algorithms work by making the decision at each step which seems best in the short-term, and not worrying about the long-term consequences. As in life such a strategy doesn't always work out, but it does here. The correctness of the algorithm relies on the following fact:

- If we take any sub-tree of a minimum weighted spanning tree, then if we add the cheapest edge leading from the sub-tree, then the new sub-tree is a sub-tree of the minimum weighed spanning tree.

How does this lead to an algorithm? We build up the minimum weighted spanning tree in stages.

- Initially, we start with some arbitrary vertex and no edges. This is trivially a sub-tree of a minimum weighted spanning tree. We don't know what this MWST is, but we know its existence. And to use the theorem, all we need is its existence.

- So, repeatedly add the cheapest edge that leads from a vertex into the sub-tree built so far, to a vertex not in the sub-tree. By the fact above, we get a sub-tree of a minimum weighted spanning tree.

- Eventually we add all the vertices in the graph to the sub-tree. Since it contains all the vertices it is a spanning tree of the graph, and by the fact it is a minimum weighted spanning tree.

We now prove the fact above.

Theorem 5.2. *Let $G = (V,E)$ be a weighted graph with cost function c. Let $T = (V,E_T)$ be a minimum weighted spanning tree of G, and let $E' \subseteq E_T$, and V' be the vertices incident to E'.*

Of all the edges in E with one vertex in V' and one vertex not in V', choose an edge (x,y) which has a weight no greater than any other such edge, then $E' \cup \{(x,y)\}$ is a subset of a minimum weighted spanning tree.

Proof: If $(x,y) \in E_T$, the proof follows immediately since $E' \cup (x,y)$ is a subset of the minimum weight spanning tree E_T.

We must now show that the result holds if $(x,y) \notin E_T$, Remember that as T is a tree, there is a unique path between x and y in T. Let (v,w) be the first edge in that path with exactly one vertex in V' (and we shall assume that v is that vertex). Now, let $E'_T = E_T - \{(v,w)\} \cup \{(x,y)\}$ and let $T' = (V,E'_T)$. Then

1. T' is a spanning tree.

   Justification: By definition it contains all the vertices, the only question is whether the edges make a tree?

   - T is a tree and so contains $n-1$ edges, where n is the number of vertices in V.
   - T' has the same number of edges as T by construction, so T' also has $n-1$ edges.
   - T' is also connected: the only possible cause of disconnection is the removal of (v,w) . But, there is a path from v to x to y to w in T' so v and w are still connected.
   - As T' is connected and has $n-1$ edges by Theorem 5.1 T' is a tree.

2. $c((x,y)) \leq c((v,w))$ (by the choice of x and y, if $c((v,w)) < c((x,y))$, then we wouldn't have chosen x and y).

3. This implies that the total weight of T' is less than or equal to the total weight of T, since all the other edges of T' and T are common.

4. From (1) and (3), T' is a minimum weighted spanning tree, and since $E' \cup \{(x,y)\}$ is a subset of T', the result follows.

This theorem leads directly to the algorithm. In the algorithm, keep the vertices partitioned into three groups. Those vertices which are part of the spanning tree found so far (call these vertices V_T), those vertices which are adjacent to the spanning tree found so far (call these F – for fringe), and the rest (call them R). To start off with, set V_T to an arbitrary vertex. When we finish, F and R are empty and V_T comprises the entire tree. At the first step, where there is only one vertex in the part of the spanning tree found so far, there are no edges yet found, and so trivially, the edges found so far are a subset of the edges of a minimum spanning tree since the empty set is a subset of all sets. At each step in the algorithm, examine all the edges between vertices in V_T and vertices in F, and choose the edge with the smallest weight. Add the vertex in F to which that edge is incident to the set V_T and remove it from F. By the above theorem since we start off with a subset of a minimum weighted spanning tree, we get after each step a subset of a minimum

spanning tree. Since all vertices must finally be a member of V_T, finally we will get a minimum spanning tree. The algorithm to find a minimum spanning tree follows below.

Algorithm MWST(V,E)
  choose $x \in V$;
  $V_T = \{x\}$;
  $E_T = \emptyset$;
  $F = \{(x,y) \in E : x \in V_T, y \notin V_T\}$;
  while $(V_T \neq V)$ {
        choose $(u,v) \in F \ni c((u,v)) = \min\{c(e) : e \in F\}$
        update(F)
        $V_T = V_T \cup \{v\}$
        $E_T = E_T \cup \{(u,v)\}$
  };

Note the invariant of the loop:

- It is always the case that at the start of every iteration of the loop, T is a minimum-weighted spanning tree of the marked vertices.

Exercise 5.3. *Implement this algorithm, and analyse your algorithm. The key question will be how you store the list F. Will you build it anew in each iteration of the loop? Or will you build it incrementally, adding and deleting as you go? How can you do this efficiently? Think about it and come up with two different solutions.*

The algorithm can be implemented in $O(m \lg n)$, and is known in as Prim's algorithm. We haven't done all the algorithms and data structures to justify this claim, but we can justify it informally:

- Finding the element (u,v) in F with the required property can be done in $\lg n$ time.
- Doing the updating requires $\deg(v) \lg n$ work.
- Hence the total work is $O(n \lg n + m \lg m) = O(m \lg m)$

5.3 Search trees

Another form of spanning tree (or more generally a spanning forest) which is very useful in many applications is a search tree. A search tree is just a way of visiting each vertex in the graph in a systematic way. The way in which we do this creates a useful structure for some algorithms. There are two types of search trees: depth-first search trees and breadth-first search trees.

5.3.1 Depth-first search tree

A depth-first search tree is perhaps the more useful. The general idea is to start off with an arbitrary vertex, visit a neighbour of the vertex (as long as it hasn't been visited before), visit a neighbour of the neighbour, and so on. Each time we visit a vertex, we put the edge followed into the search tree. If we ever get to a vertex where we have visited all the neighbours of that vertex, then we backtrack to the vertex from which we arrived, and continue there. The idea is outlined below.

```
boolean marked[n];
EdgeSet T;

Algorithm DFS(x) {
  marked = { false , false ,..., false };
  T = ∅;
  DepthFirstSearch(x);
};

Algorithm DepthFirstSearch(int v) {
  marked[v] = true ;
  while ∃w ∋ ¬marked[w] ∧ isedge(v,w) {
        T = T ∪ {(v,w)};
        DepthFirstSearch(w);
  };
};
```

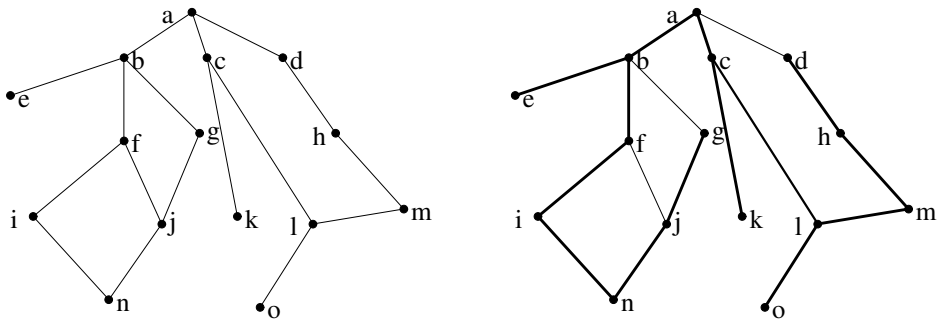
Applying the algorithm to the graph shown in Figure 5.4(a), yields the depth-first search tree found in Figure 5.4(b).

The algorithm can be generalised so that from an unconnected graph, a set of depth-first search trees can be obtained. A depth-first numbering of the graph associates with each vertex a unique number depending on the order in which the vertices were searched in a depth-first search. Note that in general, there is more than one depth-first search tree for a graph.

Alternative view of algorithm

There is another approach to representing the tree, rather than explicitly storing the edges in the tree. What we do is keep, for each vertex in the graph its *parent* in the tree (the root of the tree is numbered 0). This is an efficient means of representing the tree, and though it is not the most convenient in many situations, it has its uses. This is what the DFS algorithm using this representation looks like.

```
boolean marked[n];
int  parent[n];
```



(a) Graph (b) Depth-first search tree

Figure 5.4: Graph and one of its depth-first search trees

```
Algorithm DFS(x) {
  marked = { false , false ,..., false };
  parent[x] = −1;
  DepthFirstSearch(x);
};

Algorithm DepthFirstSearch(int  v) {
  marked[v] =  true ;
  while ∃w ∋ ¬marked[w] ∧ isedge(v,w) {
        parent[w] = v;
        DepthFirstSearch(w);
  };
};
```

A. Analysis of algorithm

The cost of executing the algorithm depends on the what data structure is used to represent the graph. The algorithm as presented above is fairly abstract and doesn't impose any conditions on the representation technique. Nor does it talk about how we implement

```
while ∃w ∋ ¬marked[w] ∧ isedge(v,w)
```

After all, *exists* and *such that* are not defined primitively in most languages. If we were implementing this in Java using an adjacency matrix, our code might look something like:

```
boolean marked[n];
int   parent[n];

void DFS(x) {
      marked = { false , false ,..., false };
      parent[x] = -1;
      DepthFirstSearch(x);
};

void DepthFirstSearch(int   v) {
      int w;
      marked[v] =  true ;
      for (w = 0; w < n; w++) {
           if (¬marked[w] ∧ isedge(v,w))   {
             parent[w] = v;
             DepthFirstSearch(w);
           };
      };
};
```

The key questions are:

- How many times is *DepthFirstSearch* executed?

  n times – since the algorithm creates a spanning tree, it has to visit each vertex (i.e., call *DepthFirstSearch*) at least once. But, since a vertex is only visited if it is unmarked, and it is marked immediately the algorithm decides to visit it, it cannot be visited more than once.

  This means that the if-condition in *DepthFirstSearch* can only evaluate to *true* n times, and so the body of the if-statement can only execute n times.

- How many times is the key condition in the loop executed?

  n^2: *DepthFirstSearch* is called n times, and for each call, we check each vertex.

In summary, if an adjacency matrix is used, the algorithm will be $\Theta(n^2)$; if an adjacency list is used, the algorithm will be $\Theta(n+m)$.

Exercise 5.4. *Do a proper algorithm analysis, showing the effect of the choice of data structure.*

```
Algorithm BFS(V,E){
  label = {−1,−1,...,−1};
  i = 0;
  T = ∅;
  Choose v ∈ V;
  label[v] = i;
  parent[v] = −1;
  do {
      more = false;
      foreach w ∈ V ∋ label[w] == i∧
                      ∃(w,x) ∈ E ∋ label[x] == −1 {
              T = T ∪ {(w,x)};
              more = true;
              label[x] = i+1;
              parent[x] = w
      };
      i = i+1;
  } while (more)
```

Figure 5.5: Breadth-first search algorithm

5.3.2 Breadth-first search

The depth-first search works by carrying on exploring from the most recently visited vertex. Breadth-first searching works by searching in 'levels' from the root. First, we add edges incident to the root then those edges that one edge away from the root, and so on... The algorithm is shown in Figure 5.5. Note that the algorithm explicitly stores the *parent*s and the tree edges, but only one is strictly necessary.

This version of the BFS is not the most convenient to program. It does show how we search the graph in a breadth-first way, but we can do things more simply. The point of labelling the vertices is that we know how far the vertices are from the source and to make sure that we then explore them in order of distance. In the approach below we implement this idea directly. As we encounter vertices, we add them to a list to be explored. We keep the invariant that (1) the vertices are always kept on the list in ascending order of distance from the source; and (2) the distance of the first vertex in the list is either the same as the distance of the last vertex on the list or the distance of the last vertex minus one.

Initially the list consists of only the source vertex. At each iteration of the loop, we remove the vertex from the front of the list (the closest vertex not yet processed) and add the all its neighbours that we have not yet seen to the list. These new vertices are one step further from the source than the original vertex, so we add them to the back of the list (which maintains the invariant). The first time we encounter a vertex we set its `parent` array appropriately.

We carry on until we have completed the entire graph. This algorithm is shown in Figure 5.6.

```
Algorithm BFS(V,E){
  parent = {0,0,...,0};
  marked = {false,...,false};
  Choose v ∈ V;
  list = {v};
  marked[v] = true;
  while (list ≠ ∅) {
        w = list.first();
        list.removeFirst();
        foreach x ∈ neighbourhood(w)∧!marked[x] {
                marked[x] = true;
                parent[x] = w;
                list.addLast(x);
        };
  }
```

Figure 5.6: A Better Breadth-first search algorithm

The tree in Figure 5.7 is a breadth-first search tree made from the graph in Figure 5.4(a). As with depth- first search trees, breadth-first search trees are not necessarily unique.

5.3.3 Classification of edges in search trees

If we have found a search tree for a graph, then all the edges in the graph which are in the tree are called the *tree edges*. In an undirected graph, those edges which are not in the tree are known as the back edges.

In a directed graph, those edges which are not in the tree and go from a vertex

- to a descendant of it in the tree is known a *forward edge*
- to an ancestor in the tree is known as a *back edge*
- to another vertex which is neither a descendant nor an ancestor is a *cross edge*.

5.4 Summary

This section of the notes examined trees, subgraphs which have a number of useful properties. These properties are used by many graph algorithms. In the first part of the section we looked at some of the properties of trees in more details. The second part looked at spanning trees which are in themselves very useful. The last part looked at two types of search trees – depth-first and breadth-first – which are constructed as a step in many graph algorithms. We shall encounter these in algorithms in subsequent sections.

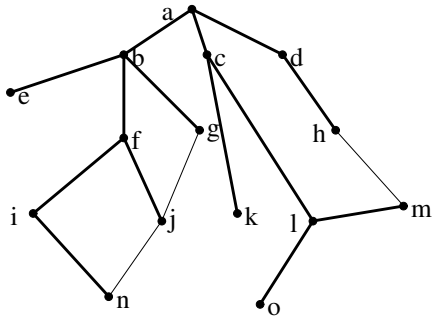


Figure 5.7: Breadth-first search tree

References: [1], [9], [10]

Chapter 6

Paths, searching and connectivity

In this chapter we look at examples of algorithms which use search trees to solve problems. The shortest path problem, which we encounter first, traverses the tree using a search tree algorithm as it finds the shortest path. We also look at finding the longest path in a restricted class of graphs.

The next three algorithms, finding connected components, bicomponents and strongly connected components of graphs use the depth-first search tree algorithm as part of the solution. These new algorithms are also fundamental graph algorithms, with practical application. The objectives of the chapter are straight-forward. By the end of the chapter, you should be able to apply the appropriate algorithms to find:—

- shortest path in a graph
- critical paths in graphs and construct PERT charts.
- connected and strongly components of a graph.
- bicomponents of a graph and
- prove correctness of the algorithms, and analyse their complexity
- compare and contrast different implementation techniques

6.1 Shortest path

Finding the shortest path in a graph is one of the 'standard' algorithms. It has many obvious applications, and can be used in a variety of situations. Formally, the cost of a path is the sum of the weight of the edges which make up the path. The shortest path problem is to find that path which has cost no greater than any other path. In general, there may be more than one shortest path between two vertices. Most shortest path algorithms do more work than strictly necessary in that finding the shortest path between a and b also finds the shortest path between a and all other vertices in the graph. The explanation below is framed as determining the shortest path from

a vertex to all other vertices. To just find the shortest path to a particular vertex, the stopping condition just needs to be changed.

At first sight, it might seem that a variation of the algorithm for the MWST (which we looked at in the previous chapter) would work. In fact, it can be seen very easily that that will not work looking at Figure 6.1.

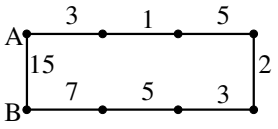


Figure 6.1: Why greed doesn't pay off...

Looking at the figure, the MWST algorithm to find the shortest path from A to B would find a path rightward and around, greedily taking the shortest edge at each stage. But the shortest path is direct; here it doesn't pay to be greedy. The shortest path algorithm is not a greedy algorithm – at each step the algorithm uses all the information found so far in the graph (rather than just one piece of information). So after step j, it may seem that the best path is along a certain edge — a few iterations later it may become clear that this would be the wrong edge to use.

6.1.1 The algorithm

Let $G = (V, E)$ be a weighted graph, with a cost function $c : E \to \mathbb{R}^+$. The basic approach is to do a breadth-first search tree, modified by considering the weights of edges. We call the vertex at the start of the path the source vertex and the vertex at the end of the path the target vertex (t) . The algorithm maintains three lists of vertices:

- the tree vertices, which are the vertices in the tree constructed so far (in the algorithm below, W is used to store the vertices in the tree constructed so far, and T the edges in the tree);
- the fringe vertices, which are the vertices which are not already in the tree, but are immediately adjacent to some vertex in the tree;
- unseen vertices — all the others.

The principle underlying the algorithm (and the invariant of the main loop) is that at each step, the algorithms knows the shortest path from the source vertex to all the vertices currently in the tree. So at the start, when the tree is empty, we only know the shortest path from the source to itself! At completion, when all the vertices in the graph are in the tree, then we know the shortest path to all the vertices.

We also use the notation that when considering an edge e which goes from a tree vertex to a fringe vertex, the vertex in the tree is called the *tail* and is denoted *tail(e)*, and the vertex in the

fringe is called the head and is denoted *head(e)*. This notation is also used more generally when referring to edges in directed graphs.

The algorithm maintains an array *dist* where dist[v] is the cost of the shortest *known* path from the source to v. Note the invariant of the loop in the algorithm: at the start of each iteration, dist[v] *is* the cost of a shortest path from the source to v for all $v \in W$. For $v \notin W$, dist[v] is just an estimate of the cost of a shortest path.

Initially, dist[s] $= 0$, while dist[v] $= c(s,v)$ for all other vertices. We'll use the convention that if there isn't an edge between two vertices a and b that $c(a,b) = \infty$.

A brief sketch of the algorithm is given below. Note that the algorithm is still abstract: we haven't yet decided whether to use an adjacency matrix or list. For simplicity, we use the *parent* array representation of the tree edges.

```
1  W = {s};
2  dist[s] = 0;
3  parent[s] = −1;
4  foreach v ∈ V − {s}
5    dist[v] = c((s,v));
7  while (W ≠ V) {
8        choose (u,v) ∈ E ∋
9              u ∈ W, v ∉ W ∧
10             dist(v) = min{dist(y) : ∃(x,y) ∋ x ∈ W, y ∉ W};
11       W = W ∪ {v};
12       foreach  w ∈ neighbourood(v) ∩ (V − W) {
13         if (dist[v] + c((v,w)) < dist[w])  {
14           dist[w] = dist[v] + c(v,w));
15           parent[w] = v;
16         };
17       };
18 };
```

6.1.2 Example

First, let's look at an example to get some intuition for how algorithm actually works. Consider how the algorithm works when applied to the graph shown in Figure 6.2. The working of the algorithm is shown in Table 6.1.

6.1.3 Proof of correctness

Second, let's look at some theory so as to be able to *prove* that the algorithm is correct. The proof of correctness rests on the following theorem.

Theorem 6.1. Suppose $G = (V,E)$ is a weighted graph with cost function $c\colon E \to \mathbb{R}^+$.

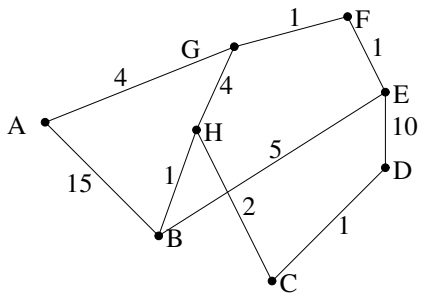


Figure 6.2: Sample graph to find shortest path

			dist					Edge	Tree
A	B	C	D	E	F	G	H	**chosen next**	**vertices**
0	**15**	∞	∞	∞	∞	**4**	∞	(A,G)	A
0	**15**	∞	∞	∞	**5**	4	**8**	(G,F)	AG
0	**15**	∞	∞	**6**	5	4	**8**	(F,E)	AFG
0	**11**	∞	**16**	6	5	4	**8**	(G,H)	$AEFG$
0	**9**	**10**	**16**	6	5	4	8	(B,H)	$AEFGH$
0	9	**10**	**16**	6	5	4	8	(C,H)	$ABEFGH$
0	9	10	**11**	6	5	4	8	(C,D)	$ABCEFGH$
0	9	10	11	6	5	4	8		$ABCDEFGH$

Table 6.1: Output of algorithm in Figure 6.2. At each stage, the fringe vertices are shown in bold.

- Let $W \subset V$ and let $v \in W$.
- Let $d(x)$ be the distance of the shortest path going from v to x.
- Suppose that the shortest path from v to any other vertex in W only goes through vertices in W.

Choose $(y,z) \in E$, with $y \in W, z \notin W$ to minimise $d(y)+c(y,z)$ over all edges with one vertex in W and one vertex not in W.

Then the path consisting of the shortest path from v to y followed by the edge (y,z) is a shortest path from v to z, and the path only goes through $W \cup \{z\}$.

By proving this theorem, we show that the algorithm presented above is correct.

Proof:

1. Choose $(y,z) \in E$ so that
   - $y \in W, z \notin W$;
   - $d(y)+c(y,z) = \min\{d(u)+c(u,w) : (u,w) \in E, u \in W, w \notin W\}$.

2. Let the shortest path from v to y be $v, x_1, \ldots, x_r, y$.

   By assumption, all the vertices in the path go through W.

3. Let $P = v, x_1, \ldots, y, z$.

   $c(P) = d(y) + c(y,z)$.

4. Let $P' = v, z_1, \ldots z_{l-1}, z_l, \ldots, z$ be any path from v to z.

   We must show that the total cost of P is less than or equal to the cost of P'.

5. Let z_l be the first vertex in P' which is not in W.

   z_{l-1} is in W, and z_l is not.

6. By the choice of (y,z), $d(y) + c(y,z) \leq d(z_{l-1}) + c(z_{l-1}, z_l)$

7. So, $c(P) = d(y) + c(y,z) \leq d(z_{l-1}) + c(z_{l-1}, z_l) \leq c(P')$.

8. By construction, P only goes through $W \cup \{z\}$

6.1.4 Complexity of algorithm

Let's look at a worst case analysis of the algorithm, concentrating on the main *while* loop as it is where most of the work is going to be done. The complexity of the algorithm does depend on the concrete implementation – for example whether an adjacency list or matrix is used, and so we cannot be entirely precise without refining the algorithm. For example, how many times does the inner loop execute s on each iteration of the outer loop: depending on the actual implementation it could be n times or $n-k-1$ times (where k is the number of elements of W).

Let's count the number of time we take the minimum of two numbers. (Why is this the right thing to count?)

- In the inner loop, there is one min operation. How many times the inner loop executes depends on the concrete implementation. If we use an adjacency matrix, we shall need to perform $n-1$ operations to find which vertices are adjacenct to a vertex.
- In each execution of the outer loop there will be: $2n-2$ operations.
  - $n-1$ operations due to the inner loop;
  - As well as operations due to line 10. In the worst case there will be $n-1$ times we take the minimum of two numbers (if an adjacency matrix is used).
- The outer loop executes with W varying from the set containing the source until T contains one less element than V. Hence k varies from 1 to $n-1$.

  Hence $T(n) = \sum_{k=1}^{n-1} 2n-2 = \in \Theta(n^2)$ in the worst case.

Exercise 6.1. *Convert the algorithm given in Figure 19 into an efficient Java program.*

6.1.5 Alternative approaches

There are a number of other shortest path algorithms that exist. These can be found by consulting text books on algorithms. Many provide better performance than this one in the average case, or for special sub-classes of graphs.

6.2 PERT charts

This section looks at finding longest paths in graph. While the shortest path problem can be solved efficiently, the longest path problem cannot in general (it is an example of an NP-complete problem). In some special cases, however, efficient algorithms do exist.

PERT charts – *PERT* stands for *performance, evaluation and review technique* – are a useful tool in project management. The activities of the project are identified, analysed, and related, and a PERT chart helps determine how activities should be scheduled and which are more critical than others.

Most projects comprise a number of sub-activities; some of these activities can be carried out in parallel, while some activities may rely on others to be finished before they can start. Organising the sub-activities appropriately is important in accomplishing the tasks efficiently.

The steps to be carried out are:

1. Identifying the tasks to be carried out, the time it will take to complete the tasks, and the relationships between the tasks.

2. Representing these tasks, costs and relationships as a directed, acyclic graph called a PERT chart.

3. Analysing the PERT chart to find the *critical path*, that sequence of tasks which determines how long the whole project will take to complete.

6.2.1 Example – Moving an office

Someone needs to move to another office. This is a very disruptive task; how can we plan this so as minimise disruption and get the move over with as soon as possible.

The tasks that must be completed (with the length of time it takes to complete the tasks in days):

T1 Arrange for telephone to be moved (5)

T2 Get network connection installed (9)

T3 Get clean power supply (24)

T4 Have office painted (19)

T5 Paint dries (3)

T6 Arrange for movers to come (23)

T7 Pack boxes in old office (3)

T8 Boxes moved (1)
T9 Unpack boxes in new office (1)
T10 New furniture moved in (1)
T11 Move computer [it's big] (4)
T12 Party (2)
T13 Move nameplate (0.1)

How these tasks should be related would depend on the requirements of the organisation and the person moving. Obviously T4 *must* happen before T5; but it may be possible (even desirable) for the person to move office even if their computer hasn't been moved or their new phone installed. Note that the time doesn't necessarily mean that the task takes that long – for example, it doesn't actually take 19 days to paint the office, but it takes 19 days from when we place the booking for the painters until they finish the job.

There are some modelling decisions to be made here. We could, instead, model the painting as two tasks: one of 17 days, the lead time for the painters to start; one of 2 days, the time it actually takes to paint the office. Which is the best way of modelling requires understanding of the world being modelled and how meaningful or appropriate it is to split up a task.

Here is one possible set of relationships between the tasks. For each task, the tasks which depend on it are listed. Note, the question whether there is a relationship between tasks is also a modelling question and would depend on the application domain. You might disagree with some of the relationships, or think there should be additional ones.

T1 : T12
T2 : T4, T11
T3 : T11
T4 : T5, T8, T10, T11
T5 : T12
T6 : T8
T7 : T8
T8 : T9
T9 : T12, T13
T10 : T12
T11 : T12
T12 : —
T13 : —

For example, task T4 must have completed before tasks T5, T8, T10, and T11 can start. This can be shown graphically as shown in Figure 6.3. Two dummy nodes, T0 and T14 are introduced to mark the start and end.

This is a directed graph and we use the convention that the direction is left to right unless shown otherwise. If a task will take x units of time, then all edges leaving the corresponding

vertex are weighted with x. If no weight is shown, a weight of 0 is taken. Also, we often do not represent redundant edges (so the edge from T6 to T8 is omitted, since that constraint is included in the constraints T6-T7 and T7-T8) since this makes the analysis algorithm a little more efficient (and our drawings less cluttered).

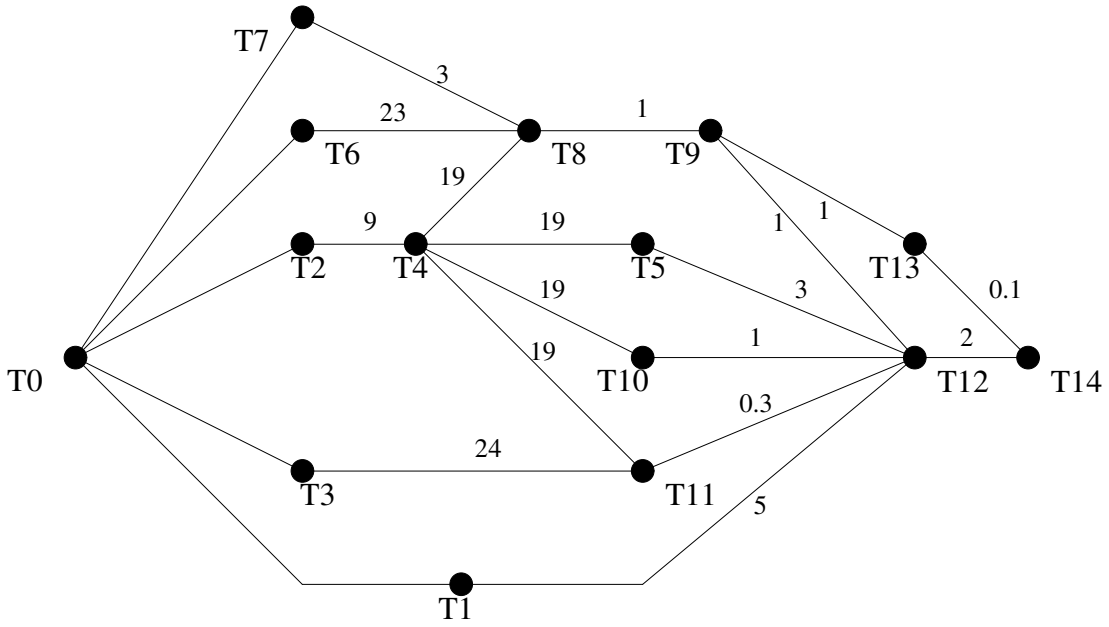


Figure 6.3: PERT Chart Example

6.2.2 Analysis of PERT chart example

The PERT chart can be analysed to help planning. The longest path from source to final nodes is known as the *critical path*. This tells us the minimum time in which the job can be completed. In this case, it is the path T0, T2, T4, T5, T12, T14. The path is critical in the sense that any delay in these will definitely lead to a delay in moving. However, with some of the other tasks there is a fair amount of *slack*. For example, T7 (packing boxes) can start immediately, but provided it is completed four days before before the planned move no delays will be casued. This gives significant freedom in scheduling the task.

Using the information like slack and critical path, a planner can allocate resources appropriately. If there are many people working on a job, she can try to minimise idle time. Jobs with lots of slack can be scheduled for more convenient times, whereas jobs on the critical path have to be done on time, or delay will be caused.

6.2.3 Critical paths and slack

Now, let's look at how PERT charts should be analysed in general. For each vertex (task), the following is computed:

- *earliest start time* (*est*): the earliest time at which a task can be started given the constraints of the precedent tasks.
- *latest finish time* (*lft*): the latest time at which a task can finish without delaying the finish time of the entire project.

The *slack* of a vertex/task v is: $lft(v) - est(v) - length(v)$. A task v can start from any time from $est(v)$ up to $est(v) + slack(v)$ without delaying the completion of the project. All the vertices on the critical path have a slack of zero.

6.2.4 Algorithm for computing slacks etc.

The algorithm for computing the *lft*, *est* and *slack* is a two phase algorithm:

- The first phase, shown in Figure 6.4, works forward in the graph, using a modified version of the breadth-first algorithm to compute the earliest start time.
  - The earliest start time of a task is the maximum of all the earliest start times of the predecessors of the task plus the costs of those predecessor tasks. Put formally:
    $est(x) = \max\{est(u) + c(u,x) : (u,x) \in E\}$ or
    $est(x) = \max\{est(u) + length(u) : (u,x) \in E\}$

  Initially, we estimate the start time of each task as 0. As we work through from start to finish, the algorithm refines the estimate of the earliest start time. Once this phase has finished, the algorithm has computed not only the earliest start time of each task, but the shortest possible time in which the whole project can be computed. The critical path can be found from the *parent* array, working backwards from the destination.
- The second phase, shown in Figure 6.5 works backwards from the finish node to start node (using a similar algorithm) to compute the latest finish time. Essentially the latest finish time of a task x is the minimum of the latest finish time of all successors of x less the cost of those successors. Formally,
  $lft(x) = \min\{lft(u) - length(u) : (x,u) \in E\}$.

Exercise 6.2. *Apply the PERT algorithm to the graph given in Figure 6.3.*

PERT charts can be generalised and extended in many ways. For example, often we cannot know exactly how long a task will take, but will have reasonable upper and lower bound estimates for the costs of tasks. These estimates can be represented in the PERT chart so that even in the absence of exact information, reasonable analysis can be done. A full working Java program can be found in Section A.1 on page 131.

```
Algorithm EST(V,E,s){
  label = {−1,−1,...,−1};
  est = {0,0,...,0};
  parent = {0,0,...,0};
  i = 0;
  label[s] = i;
  do {
      change = false;
      foreach w ∋ label[w] == i {
              foreach (w,x) ∈ E {
                change = true;
                label[x] = i+1;
                if (est(w)+c(w,x) > est(x))   {
                  est(x) = est(w)+c(w,x);
                  parent[x] = w };
              };
      }
  };
  i = i+1;
} while (change)
```

Figure 6.4: Forward phase of PERT algorithm: finding *est*

Alternative implementation

There is a much more elegant (and still efficient) recursive solution. Consider the definition above:

$$est(x) = \max\{est(u) + length(u) : (u,x) \in E\}$$

We could define implement this directly using a simple recursive algorithm. Assume that vertex 0 is the dummy start vertex, then we can say:

```
Algorithm EST(v)
  estpred = 0;
  if (v == 0)
    return 0;
  foreach u ∈ V ∋ (u,v) ∈ E
    if (EST(u)+length(u) > estpred)
      estpred = EST(u)+length(u);
  est[v] = estpred;
  return estpred;
```

```
Algorithm LFT(V,E,f){
  label = {−1,−1,...,−1};
  lft = {est(f),...,est(f)};
  slack = {0,0,...,0};
  i = 0;
  label[f] = i;
  do {
      change = false;
      foreach w ∋ label[w] == i {
        change = true;
        label[x] = i+1;
        foreach (x,w) ∈ E {
          if (lft(w) − length(w) < lft(x))  {
            lft(x) = lft(w) − length(w);
          };
        }
      };
      i = i+1;
  } while (change);
  foreach v ∈ V
    slack(v) = lft(v) − est(v) − length(v);
```

Figure 6.5: Backward phase of PERT algorithm: finding *lft*

and then calling `EST(n)`, where *n* is the dummy finish node, will compute the EST values of all the nodes. This is not efficient though because it will result `EST` being called many many times on the same vertex. So, we use the idea of caching the results.

First, we initialise the array `est` with a 0 entry for the dummy start and -1 for all the other vertices. Whenever we compute the EST of a vertex, we record it in the `est` array. The big difference is that the first thing we do when we call `EST` is check whether we have already computed the EST value: we can do this by checking the `est` array, and if it has a non-negative value return that value. The Java code is below. Comparing with the code above shows it much more elegant and probably more efficient. (It's not such an easy one for a human to trace because

you need to record the program stack!)

```
float ComputeEST(int v) {
    int u;
    float estpred;
    if (v==0) return 0;
    if (est[v]>=0)  return est[v];
    for(u=0; u<numvertices; u++) {
       if (edge[u][v]) {
         estpred = ComputeEST(u);
         if (estpred + length[u]> est[v]) {
            est[v] = estpred + length[u];
            parent[v]=u;
         }
       }
     }
     return est[v];
}
```

LFT can be computed in a similar way.

Exercise 6.3. *The Pert chart uses a modified breadth-first search tree. One of the complications of the implementation we have used is that it is inefficient in two respects: (1) at each stage we have to find the vertices that are labelled with the current label (2) we may reset a label of a vertex several times. Re-implement the code using a queue data structure and show how that simplifies the code.*

Exercise 6.4. *Another approach is to use the following depth-first search algorithm.*

```
void DFSest(curr) {
   for(w=adjlist[curr]; w!= null; w=w.next) {
       if est[w.y] < est[curr]+len[curr] {
            est[w.y] = est[curr]+len[curr]
            parent[w.y] = curr;
            DFSest(w.y)
       }
   }
}
```

(1) Discuss the correctness or otherwise of this algorithm. (2) How expensive is it? (3) Compare its performance to the dynamic programming algorithm given previously.

6.3 Connectivity

The goal of this section is to find an algorithm which determines whether a graph is connected, and if it is not connected an algorithm which finds the connected components. The practical application for finding whether a graph is connected should be clear. Questions of the form 'can we go from x to y?' are very important to answer.

One algorithm, based on the adjacency matrix representation of the graph, computes the connectivity matrix from the adjacency matrix. In the adjacency matrix, A, of a graph of n vertices let the i, j-th entry be 1 if there is an edge between vertex i and vertex j, and 0 otherwise. Now compute the connectivity matrix A^n, where matrix multiplication is done in the standard way. A non-zero entry in the i, j-th entry indicates that there is a path between i j. Once we have computed the connectivity matrix, it only requires n^2 operations to determine whether the graph is connected, and another n^2 at most to determine the connected components. So that's not too bad. Unfortunately, computing the connectivity matrix is expensive. The straight-forward approach requires $\Theta(n)$ matrix multiplications, and each matrix multiplication takes $\Theta(n^3)$ operations making, in total, $\Theta(n^4)$ operations. Using the best known matrix multiplication algorithm (Strassen's) reduces the cost to $O(n^{2.376})$ operations per matrix multiplication, and a cleverer approach to computing A^n takes $\Theta(\log n)$ operations. This means we can probably get away with $O(n^{2.376}\text{logn})$ operations, which is still quite expensive.

Warshall's algorithm — much simpler than what's been suggested here — can find the connectivity matrix in $\Theta(n^3)$ operations.

For finite state machine analysis the logical representation of graphs using binary decision diagrams has been extremely successful. While in the worst case, performance can be terrible, the approach has been used with great success in many examples of well over 10^{100} states!

Here we'll look at adjacency list representations. In the previous chapter, we saw we could use the depth-first search tree of a graph to find a spanning tree for a connected graph. To find whether a graph is connected, we choose an arbitrary vertex, and find a depth-first search tree from that algorithm. If the resulting tree is a spanning tree, then we know that the graph is connected. If it is not a spanning tree, then we have found one of the connected components. The algorithm, based on the depth-first search tree is outlined in Figure 6.6. The algorithm uses the adjacency list, and also an array called *mark* which records whether the vertex has been visited. Initially, all entries in *mark* are set to false to indicate that they have not yet been traversed by the depth-first search tree. Then, an arbitrary vertex is chosen (for convenience, we choose the first vertex, but any would do), and a depth-first search is commenced. We mark each vertex we find with true, printing out any vertices found in the connected component.

When we can go no further, we leave the *DFS* procedure, and look for another vertex which we have not yet marked, and start a depth-first search tree from there, marking any vertices found. We carry on this approach until all the vertices have been marked.

Exercise 6.5. *The algorithm is presented just prints out the vertices in the components.*

1. *Show how to modify it so that the edges of each component are printed out.*

2. *Rather than printing out the components, modify the algorithm so that it records for each*

```
Algorithm DFS(int v,boolean marked[]) {
   // Perform DFS from vertex v
   // -- just prints out vertices
  EdgeList w;
  marked[v] = true ;
  print(v);
  for (w = adjlist[v]; w! = null; w = w.next) {
      if (!marked[w])  {
        DFS(w,marked);
      }
  };


Algorithm ConnectedComponents() {
  boolean marked[] = new boolean[numvertices];
  int i,component = 0;
  for (i = 0; i < numvertices; i++) marked[i] = false ;
  for (i = 0; i < numvertices; i++) {
      if (!marked[i])  {
        println("Component" + component + "contains:");
        DFS(i,marked);
        component++;
      }
  }
}
```

Figure 6.6: Algorithm for finding connected components

data structure which component it is in. Each of the components should be numbered, and your algorithm should record for each vertex the number of the component.

One way of doing so, is to change marked *from a boolean array to an int array – rather than just saying whether a vertex is marked or not, it should indicate whether it is not marked (0), or if it is marked, which component it is in.*

Performance of algorithm: Note that *DFS* is called exactly once for each vertex exactly once. On each call of *DFS*, the algorithm goes through the vertex's adjacency list. In the case of an undirected graph, this means that each edge will be examined twice, and in the case of a directed graph each edge will be examined once. This means that the algorithm is $\Theta(n+m)$. How would this chance if an adjacency matrix were used?

6.4 Bicomponents

Finding whether a graph is connected, and finding the connected components is clearly very useful. There are, however, different degrees of connectivity. Suppose that you were responsible for planning how the telephone exchanges were connected. Two possible choices would be to connect the exchanges in the form of a tree, and in the form of a complete graph (obviously, there would be other choices too). The complete graph form would be much more expensive than the tree form. But, the tree graph also has a significant disadvantage. If one of the telephone exchanges goes down, then the telephone system becomes disconnected. The graphs in Figure 6.7 show the problem.

Figure 6.7: Different levels of connectivity

In the plan on the left, if exchange C goes down, then exchanges A and E can no longer communicate. In the plan on the right, C going down doesn't affect anyone else (except that they can no longer communicate with C!).

This concept can be formalised.

Definition 6.1. *A graph is* k-connected*, if the removal of fewer than any* k *vertices and the edges incident to them leaves the remaining graph connected.*

A 1-connected graph is just the ordinary connectedness we have seen already. A 2-connected graph (or a *biconnected* graph) is graph which the removal of any one vertex still leaves connected. In other words, a biconnected graph is a connected graph without any articulation points. In this section, we will focus on biconnectivity. The algorithm that we develop can be generalised to deal with other forms of connectivity.

Definition 6.2. *A biconnected component (*bicomponent*) of a graph is maximal subgraph of a graph which is biconnected.*

Finding whether a graph is biconnected or not has many applications. If a graph is not biconnected, then we want to be able to find the articulation points of the graph. For example, if we do have a telephone system which has articulation points, then we know where problems can occur, and where we need to spend money to make sure that the equipment is reliable. There is an analogous problem: does the removal of an edge (or more generally k edges) leave the graph connected. We don't deal with this problem here, but similar techniques to the one we see here

can be used to solve this problem. Before trying to develop an algorithm to solve this problem, let's try to find out more about the properties of depth-first search trees and their relationship to articulation points.

Theorem 6.2. *Let $G = (V,E)$ be an undirected graph with a depth-first search tree $T = (V,E_T)$. Let v be a vertex of the graph which is not the root of the tree T.*

v is articulation point if and only if (1) it is not a leaf and (2) some subtree of v has no back edge incident with some proper ancestor of v.

Proof:

$\Longrightarrow$ Suppose that v is an articulation point.

1. Then, there are vertices x and y (with x, y and v all distinct) with v in every path between x and y .

2. If neither x nor y were descendants of v in the tree, then there would be a path going from x to y in the tree without going through v. This contradicts the supposition that v is an articulation point:

   (a) At least one of x and y is a descendant of v;

   (b) Therefore v is not a leaf in the tree;

   (c) Without loss of generality, let x be the descendant.

   We will assume that all subtrees of v have an edge going to a proper ancestor of v and show that this contradicts the assumption that v is an articulation point.

3. Use proof by contradiction: assume that there are back edges from all sub-trees of v to proper ancestors of v. We'll show that this assumption leads to the conclusion that v is *not* an articulation point.

   We construct a path from x to y that does not contain v. Let the back edge that connects the sub-tree containing x to a proper ancestor of v be (x_t, x_v). There are three cases to consider:

   (a) If y is an ancestor of v, then we can go from x to x_t, from x_t to x_v and from x_v to y, without going through v. .

   (b) If y is a descendant of v then, as we are for the meantime assuming than there is a back edge going from *each* subtree of v to a proper ancestor of v, there is a back edge going from the subtree of v containing y to a proper ancestor of v, Let this back edge be (y_t, y_v). Then we can go from x to x_t from x_t to x_v, from x_v to y_v, from y_v to y_t, and from y_t to y.

   (c) The case of y being neither an ancestor nor descendant of v is similar to 3a.

So, assuming that all subtrees of v have an edge going to a proper ancestor of v implies that v is not an articulation point as we have been able to find paths from x to y which don't contain v. This is a contradiction of the hypothesis that v was an articulation point, and so assuming that all subtrees of v have an edge going to a proper ancestor of v is wrong. Therefore there is at *least* one subtree of v which does not have an edge going to a proper ancestor of v.

$\Longrightarrow$ Now, suppose that v is not a leaf and that there is some subtree of v which does not contain a back edge going from it to a proper ancestor of v. Let x be the root of this subtree (so x is a child of v) and r be the root of the depth-first search tree.

1. Let $x, x_1, \ldots, x_m, r$ be a path from x to r. Each edge must either be a back edge or a tree edge.

2. Let x_i be the first vertex in the path which is a proper ancestor of v (it may be r itself). Then, the edge (v, x_i) must be in the path since there is no other tree edge which has this property, and there are no back edges going from the tree containing x to a proper ancestor of v.

3. Therefore v is in all paths connecting x and r. This implies that v is an articulation point.

Theorem 6.3. *If v the root of a depth-first search tree, then it is an articulation point if and only if it has more than one child.*

Well, how does this help? Well, consider what happens during a depth-first search. Suppose, when at vertex v and the algorithm recursively does a depth-first search from w, an unmarked neighbour of v. If by the time that we have finished exploring the sub-tree rooted at w, we have found no back-edge which goes to a proper ancestor of v, then we know that v is an articulation point.

This can be detected in the following way. During the depth-first search, the algorithm computes the depth-first numbering of each vertex in the graph. In addition, it keeps track, for each vertex, of how far *back up* the tree we can go by following tree edges *down* the tree, and back edges *up* the tree. The algorithm accomplishes this by keeping two arrays:

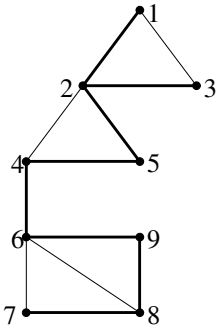
1. dfsnum: the depth-first number of each vertex

2. back: the lowest depth-first number of a vertex reachable going down the tree using tree edges, and up using back edges.

Suppose the algorithm is at v and then does the recursive depth-first search from a neighbour w. Once the depth-first search from w completes, If `back`$[w]$ is less than `dfsnum`$[v]$, then there is a back edge from the subtree rooted at w to a proper ancestor of v. If not, then there is no back edge going from the subtree rooted at w to a proper ancestor of v and hence (by the theorem) we can conclude that the vertex v is an articulation point. The algorithm below follows this technique closely; the only significant difference is that as we process edges we place them on the stack so that when we find an articulation point we can find the corresponding bicomponent.

Algorithm

The algorithm does the dfs-search, computing the dfs-numbers and the back numbers. At each vertex, the children are dfs-searched one by one. If after dfs-searching from a child, the child's back number is not less than the vertex's dfs-number, an articulation point has been found. The algorithm is given in Figure 6.8.



Example 6.1.

v	*dfs num*	$back[v]$	*Bicomponents*
1	0	0	$0:(1,2),(2,3),(3,1)$
2	1	0	$1:(2,4),(4,5),(5,2)$
3	2	0	
4	4	1	$2:(4,6)$
5	3	1	
6	5	5	
7	8	5	$3:(6,9),(9,8),(8,6),(8,7),(7,6)$
8	7	5	
9	6	5	

Articulation points **emboldened**

Some points about the algorithm in Figure 6.8.

- In line 125, we have just completed the recursive call to BDFS for our child. If we have found a better back number, then up update our back value.
- In line 128, the neighbour is a is already marked and so we don't recursively call BDFS, we just update our back number. There are three possibilities:
  - If the neighbour is an ancestor other than our parent, we have found a back edge up the tree and so we can just update our back value to the dfsnumber of this neighbour.
  - If the neighbour is a child then in fact we have already completed the BDFS call for the sub-tree that contains the child. Strictly it is not necessary to update the back value since we have already explored this sub-tree. However, checking that it is a child and not an ancestor is as much work as doing the redundant work of of again comparing the back number of this vertex with our dfsnumber.

```
Algorithm BDFS(int v,int  parent,int  curr_dfsnum) {
  marked[v] =  true ;
  dfsnum[v] = curr_dfsnum;
  back[v] = curr_dfsnum;
  for (w = adjlist[v]; w! = null; w = w.next) {
       if (!marked[w])
         curr_dfsnum = BDFS(w,v,curr_dfsnum+1);
         if (back[w] >= dfsnum[v])
           print(v+`` is an articulation point'');
         else
           back[v] = Math.min(back[v],back[w]);;
       else
       if (parent! = w)
          back[v] = min(dfsnum[w],back[v]);
       };
       return curr_dfsnum;
  }


Algorithm  BicomponentDFS() {
  boolean morechildren =  false ;
  for (i = 1; i < numvertices; i++) marked[i] =  false ;
  marked[0] =  true ;
  curr_dfsnum = 0;

  for (w = adjlist[0]; w! = null; w = w.next) {
       if (!marked[w])  {
         curr_dfsnum = BDFS(w,0,curr_dfsnum+1);
         if (morechildren)  {
           put(``Root is an articulation point''); };
         morechildren =  true ;
       }
};
```

Figure 6.8: Algorithm for finding bicomponents: A working Java implementation of this algorithm can be found in Section A.2 on page 140

– If the neighbour is our parent then strictly we should not update our back number. We could explicitly check whether the neighbour is our parent before doing the checking. However, doing this check creates extra work since we need to do the check for every neighbour. Omitting the check does not cause problems. If there is back edge that goes higher up the tree, line 128 has no effect.

: if we do set our back number to the dfsnumber of our parent, at the stage when we backtrack we shall discover that the we have smae

Analysis of the algorithm

The initialisation takes $\Theta(n)$ steps. The depth-first search tree takes $\Theta(m)$ operations since we traverse each edge twice. Therefore the performance is $\Theta(n+m)$ operations in total.

6.5 Strongly connected components

We have already seen how to find the weakly connected components of a graph. We cannot use the same algorithm to find the strongly connected components of a directed graph *(digraph)* because we need to know that for any two vertices x and y, there is a path from x to y and from y to x (i.e., the vertices must be on a *cycle*). However, there is a strong analogy between the algorithm to find the bicomponents of a graph and the strongly connected components of a digraph.

Again, we use the depth-first search as the basis of the algorithm. By the definition of a tree, we know that we can go from an ancestor to a descendant. We want to know more. Can we go from a descendant to an ancestor? As we build the depth-first search tree, we record for each vertex how far up the tree we can go using back edges and cross edges. If we cannot go to any ancestor of a vertex going *down* the tree using tree edges, back edges and cross edges, then the vertex forms a boundary between two strongly connected components. The vertex's parent is in a different strong component to it. First we look at some properties of depth-first search trees in a digraph, and then we see how we can apply these properties to find the strong components. Let $G = (V,E)$ be a digraph, and $T = (V,E_T)$ be a depth-first search tree of the digraph then if we define:

- v to be left of w or vLw if v is encountered before w in the depth-first search tree;
- `oldest`$[v]$ is the ancestor of v closest to the root which can be reached by using tree edges, cross edges or back edges. Remember in digraphs that edges can only be traversed in one direction.

Theorem 6.4. *Let $G = (V,E)$ be a digraph, and $T = (V,E_T)$ be a depth-first search tree. Then*

1. *If $(v,w) \in E$ then wLv or w is a* descendant *of v.*
2. *x and y are in the same strong component iff* oldest[x] = oldest[y].

Proof:

1. This is the same things as saying that edges are tree edges, back edges, forward edges or cross edges, and if an edge is a cross edge then it goes from right to left. If(v,w) is a tree edge or a forward edge, then w is a descendant of v. If (v,w) is a back edge then wLv, since w must have been encountered first. If (v,w) is a cross edge then w must have also been encountered first. (Why? Suppose v had been encountered first in the depth-first search — what would the relationship in the tree be between v and w?)

2. We prove the only if part here and relegate the if part to the tut.

   (a) Suppose that x and y are in the same strong component. Assume that xLy, the converse being symmetric.

   (b) Consider the path P which lies entirely within the strong component that goes from x to y. Let z be the vertex on the path which has the smallest dfs-number (i.e. for all other vertices w in P, zLw).

   (c) From z onwards, all vertices in the path must be descendants of z since any edge going to an ancestor or unrelated vertex would go to a vertex with a smaller dfs-number. Thus, y is a descendant of z, with zLy.

   (d) If the $z = x$, then we have that y is a descendant of x.

   Otherwise by the choice of z, zLx and by assumption, xLy, with y a descendant of z. But this implies that x is a descendant of z (by the way in which the dfs-numbering is done: since x is 'between' z and y in the numbering sequence, and y is a descendant of z, x must be too).

   (e) Thus, we have found a common ancestor of x and y which lies within the strong component containing them both. There may be many such ancestors: choose the one closest to the root, call it u.

   (f) u must be `oldest`$[x]$, since if there were an ancestor of x closer to the root which were also a member of the strong component, it would be an ancestor of u and hence y, and so a common ancestor of both x and y, which is a result contrary to the choice of u.

   Similarly, u must be `oldest`$[y]$.

This theorem tells us how to go about finding the strong components. As the depth-first search tree is constructed, the *oldest* of each vertex is computed. Once the search tree is constructed, scanning these values indicates which vertices are in the same strong components. Computing *oldest* though can be quite expensive. So, better than using *oldest* is to use something analogous to the array *back* which was used in the finding on bicomponents. The array *low* contains for each vertex v the lowest numbered vertex of V which can be reached by tree edges and *one* back or cross edge, and is in the same strong component of v.

```
1  Algorithm  StrSrch(v) {
2    marked[v] =  true ;
3    dfsnum[v] = curr_dfs;
4    curr_dfs++;
5    low[v] = dfsnum[v];
6    vstack.push(v);
7    foreach  w ∋ ∃(v,w) ∈ E
8       if (!marked[w])  {
9         StrSrch(w);
10        low[v] = min(low[v],low[w]); }
11      else
12        if (onstack(w))
13          low[v] = min(dfsnum[w],low[v]);
14   if (low[v] == dfsnum[v])
15     print(``Found strong component'');
16     while (!vstack.empty && dfsnum[vstack.top] >= dfsnum[v]) {
17           x = vstack.pop;
18           print(x);
19     };
20   };
21
22   Algorithm StrongSearch() {
23     curr_dfs = 0;
24     for (i = 0; i < numvertices; i++) marked[i] =  false ;
25     vstack.init();
26     for (i = 0; i < numvertices; i++)
27         if (!marked(i))  StrSrch(i);
28   };
```

Figure 6.9: Algorithm for Strong Components: The corresponding Java code with some commentary can be found in Section A.3 on page 142

The algorithm

How the algorithm shown in Figure 6.9 works:-

- When v is first encountered, $\texttt{low}[v]$ is assigned v's depth-first numbering.
- As the algorithm encounters vertices, the vertices are pushed on a stack, which keeps track of the strong components. (How the stack keeps track of this is explained shortly.)
- When exploring from w, if a back edge or cross edge is encountered, say (w,y), then $\texttt{low}[w]$ is updated if appropriate.
  - if a back edge is encountered, $\texttt{low}[w]$ is updated (if *dfsnum[y]* is less than current value of $\texttt{low}[w]$);
  - if a cross edge is encountered then we have to know whether y is in the same strong component before trying to update $\texttt{low}[w]$. This can easily be tested by checking to see whether y is in the stack:
    - If y is on the stack then y is in the same strong component and $\texttt{low}[w]$ can be updated.
    - If it's not then y is not in the same strong component and $\texttt{low}[w]$ should not be updated.
- When backing up from w to v, if $\texttt{dfsnum}[w] = \texttt{low}[w]$, then w marks the end of a strong component, and the entire contents of the stack is printed out as being the strong component.

Why is the algorithm correct? Consider the dfs-tree that is constructed.

If when the algorithm backtracks from a vertex v, $\texttt{low}[v] = \texttt{dfsnum}[v]$, then there is no back edge or cross edge from the tree rooted at v to an ancestor of v in the tree.

- In particular, there is no path going from v to its parent and the two must be in different strong components.

On the other hand, suppose that during the traversal, an edge (v,w) is encountered. If the dfs number of w is greater than that of v, the edge is a forward edge (remember cross edges go to the left), and so can be ignored as we already have a path from v to w.

If w is not on the stack, then at an earlier phase in the algorithm, the strong component containing w was found, and so v cannot be in the same strong component as w.

If w is on the stack, then either it is an ancestor of v (i.e. the edge is a back edge) or a cross edge. In either case w is in the same strong component as v. Why?

- If w is an ancestor, then there is already a path going from w to v consisting entirely of tree edges. We now have found a path from v to w containing only one edge, which is what we want.

- So what if w is not an ancestor. We wish to show that there must be a path from w to v.

  Recall that `low`$[x]$ is the lowest numbered vertex known to be in the same strong component as x which can be reached by tree edges and one cross edge or back edge.

  Let $w_1 = w$; $w_{i+1} = \text{low}[w_i]$ for each i, until some k, where $\text{low}[w_k] = \text{dfsnum}[w_k]$. w_k must be an ancestor of v. Why?

  Suppose it weren't an ancestor. Consider the sequence $v = w_0, w = w_1, w_2, \ldots, w_k$. Choose the largest j, such that w_k is not an ancestor of w_j — by our assumption that w_k is not an ancestor of v such a j must exist.

  But by the definition of low and the choice of the w_i, $w_k L w_j$. And since w_k is not an ancestor of w_j, the dfs-search backed up from w_k before encountering w_j.

  Now as $\text{low}[w_k] = \text{dfsnum}[w_k]$, when the search backed up from w_k, w_k and all its descendants would be removed from the stack.

  This is a contradiction as w_{j+1} is a descendant of w_k and so would be removed from the stack at the time that $\text{low}[w_j]$ was set, and so $\text{low}[w_j]$ could not be given the value w_{j+1}. So w_k is an ancestor of v. This gives us the desired path from w_k to v. Using the paths from w_1 to w_2 to $\ldots w_k$ together with the path in the tree from w_k to v.

How expensive is the algorithm? The initialisation is $\Theta(n)$. In the execution of the algorithm, we consider each edge a fixed number of times. In the process of doing this, we will also investigate each vertex. Thus this will take $\Theta(\max(n,m))$, the overall complexity of the algorithm. The only thing which might look tricky is checking to see whether the vertex is in the stack. Instead of actually checking the stack, we can just use an auxiliary boolean array, one entry per vertex indicating whether it is 'on the stack. Updating this correctly allows the check to be made in constant time.

6.6 Summary

This chapter looked at the use of search trees in some of the standard graph algorithms. In each case we looked at some theory (and saw that search trees were used in each case), and then at the algorithms. The graph algorithms examined were:-

- shortest path
  - greedy algorithm not good enough
  - finds shortest path from vertex to all other vertices
  - A tree is built up with source as root
  - At each stage in algorithm know shortest path from source to all tree vertices; have estimates for fringe vertices
  - At each step in algorithm add the fringe vertex which is closest to the root; then update estimate for all fringe vertices

- connected components
  - use a dfs to find the connected components
- bicomponents
  - biconnected graph is a graph with no articulation points
  - bicomponent of a graph is a maximal biconnected subgraph
  - use a dfs tree as skeleton
  - algorithm uses fact that if a vertex has a sub-tree which does not have an edge going to a proper ancestor of the vertex, then it is an articulation point
- strong components
  - strongly connected graph is graph so that for any pair of vertices there are paths going both ways from one to the other
  - strong component is a maximal strongly connected subgraph
  - approach analogous to bicomponents - use a dfs tree as skeleton
  - can use the concept of *oldest* (but expensive)
  - rather find *low* for each vertex; this is how far *up* the tree we can go, going down using tree edges, and up using *one* back edge or cross edge
  - if $\texttt{low}[v] == \texttt{dfs}[v]$ then v is in a different strong component to its parent

Chapter 7

Circuit problems

Finding circuits and cycles in graphs — especially circuits and cycles with certain properties — has some practical application. In this chapter we investigate particular circuits and cycles.

Objectives

By the end of this chapter you should be able to:

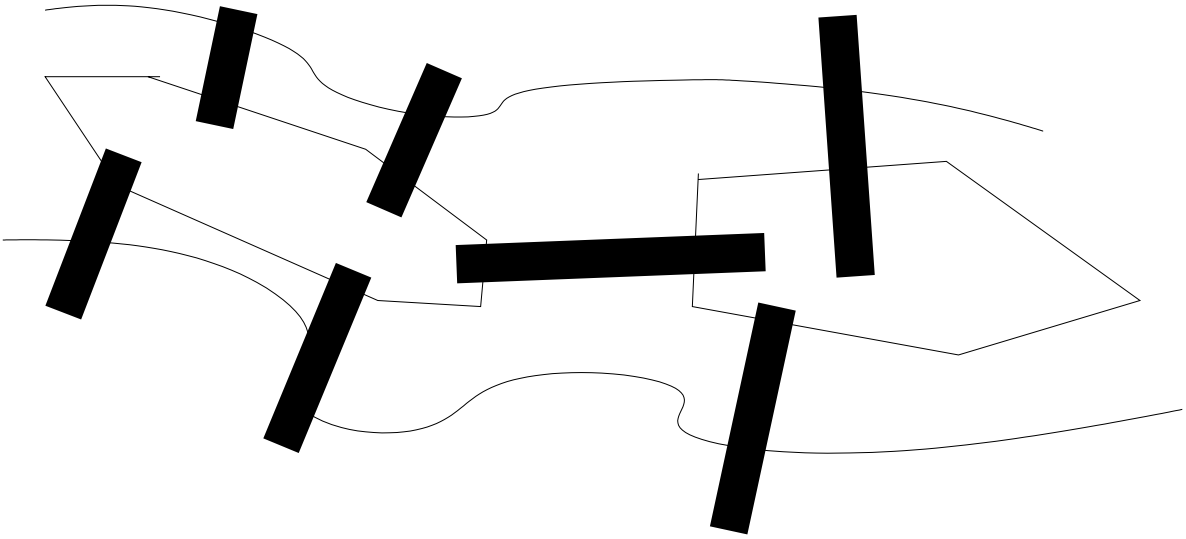
- define Pert charts, explain their applications, implement the related algorithm and solve Pert problems.
- define, Eulerian and Hamiltonian circuits, and the travelling salesperson problem;
- prove important properties of these problems;
- design and analyse algorithms which solve them

Definition 7.1. *A cycle is a path consisting of at least two edges from a vertex to itself. Definitions are non-standard, but we will take a circuit to be a walk from a vertex to itself.*

7.1 Eulerian circuits

Euler proposed the following problem (or was asked to solve it or solved it, depending on what book you read). Below is a map of the city of Königsberg – there is a river separating two parts of the town, and two islands in the middle of the river. Is it possible to walk over each bridge once and exactly once (no swimming, flying or extra-long jumping allowed). A range of similar problems exists. This general problem can be solved in some circumstances and not in others. In this section we will look at some theory which tells us when the problem can be solved and when it can't.

Eulerian circuits have a number of real applications — for example a number of problems in computational biology, including the DNA layout problem are solved using the Eulerian circuit algorithm.



Theorem 7.1. *If G is a graph in which every vertex has degree of at least two then the graph has a circuit.*

Proof: Choose v arbitrarily. Let v_1 be any neighbour of v. For $i > 1$, let v_{i+1} be any neighbour of v_i, which is not v_{i-1}. Such a vertex must exist since the degree of each vertex is at least two. This allows us to construct a walk: $v, v_1, v_2, \ldots, v_{n-1}, v_n, v_{n+1}, \ldots$ But there are only n vertices in the graph. This means that some vertex must appear more than once. Therefore there is a circuit.

Definition 7.2. *: A graph is* Eulerian *if there is a circuit in the graph which contains every edge in the graph once and only once. A graph is* semi-Eulerian *if and only if it has a walk which contains each edge once and only once.*

Theorem 7.2. *: If $G = (V, E)$ is a connected graph, then G is Eulerian if, and only if, every vertex in G has even degree.*

Proof:

$\Rightarrow$ If a graph is Eulerian. Each vertex must appear at least once in the circuit. Each time it appears, there is an edge "coming into" the vertex and an edge "going out from" the vertex. All edges are accounted for in this way. Therefore the degree of each vertex is even.

$\Leftarrow$ Suppose that every vertex in G has even degree. The proof is by induction on the number of edges in the graph.

- *Basis step*

  If there are zero edges in the graph, then the graph has a trivial Eulerian circuit, since there is only one vertex.

- *Induction hypothesis*

  Any graph with m edges is Eulerian if each vertex has even degree.

– Induction step
Let $G = (V, E)$ be a connected graph with $m + 1$ edges. As each vertex must have degree greater than 2 by theorem 7.1, the graph has a circuit. Let this circuit be $v_1, v_2, \ldots, v_k$. Remove from the graph all the edges in this circuit to obtain the graph H. This may be disconnected, but all vertices have even degree. Consider each component of the graph. Each component is connected, and has m or fewer edges. Thus by the induction hypothesis each component has an Eulerian circuit. The circuit we first found intersects these sub-circuits in at least one vertex. These circuits combined form an Eulerian circuit, since each edge appears exactly once .

Exercise: Use this algorithm to find an Eulerian circuit in the graph in Figure 7.1.

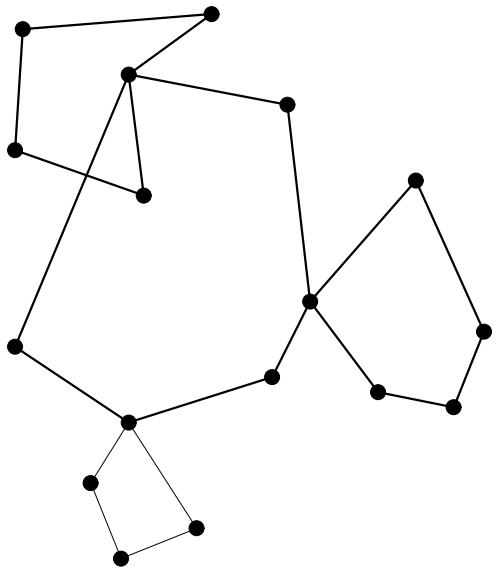


Figure 7.1: Graph for exercise

Theorem 7.3. . *The algorithm in Figure 7.2 constructs an Eulerian circuit where possible.*

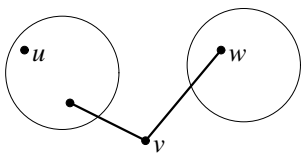
Proof: First recall that the sum of the degrees of all the vertices is an even number. This implies that no graph can have exactly one vertex of odd degree. Suppose the graph is Eulerian. Now suppose that we start off at vertex u and reach vertex $v, v \neq u$. First we show that we can continue the construction. Each vertex originally has even degree. To get to v, we traversed one edge, and deleted it. Therefore v must now be of odd degree, which means that we can continue. Second, the construction does not disconnect the graph. Both u and v will be of odd degree.

```
EulerConstruct(V,E)
  Choose u ∈ V;
  Print``start at u'';
  while (E ≠ ∅ ∧ Neighbourhood(u) ≠ ∅) {
        if (∃(u,v) ∈ E ∋ (u,v) is not a bridge)
          w = v;
        else
          Choose w ∈ Neighbourhood(u);
        E = E − {(u,w)};
        if (deg(u) == 0) V = V − {u};
        if (deg(w) == 0) V = V − {w};
        u = w;
  }
  return (E == ∅);
```

Figure 7.2: Fleury's algorithm

- Suppose that there is only one bridge incident to v. Let this bridge be (v,w). If this edge is traversed and removed then v will be isolated (of degree 0) since we only traverse bridges if there is no other choice. As v is of degree 0, it will be removed from the graph. As there are no other edges incident to v, the removal of v cannot disconnect the remainder of the graph.
- So suppose that there are at least two bridges incident to v (see the drawing below). This means that there is a bridge (v,w) so that u will not be in the same component as w once the edge (v,w) is deleted. Now w will of odd degree since we have now deleted one of its edges. But all the other edges in its component will still have even degree. This component containing w is a graph in its own right. But it is impossible for a graph to contain only one vertex of odd degree. Hence there can be no such graph. Hence there can be not more than one bridge. Hence the construction does not disconnect the graph.

If $v = w$, and there are still edges to be deleted then since the graph is connected still, the construction can continue. If $v = w$ and there are no more edges to be deleted, then we have found an Eulerian circuit: we have started at off at a vertex, gotten back to the vertex an traversed every edge once and only once. This algorithm is called Fleury's algorithm.



Theorem 7.4. *A connected graph is semi-Eulerian if and only if either (i) there is no vertex of odd degree, or (ii) exactly two vertices have odd degree.*

The proof is left as an exercise.

A better algorithm

A key part of the algorithm is to determine whether edges are bridges. On fairly small graphs, it is easy for humans to do do this at a glance, and so the algorithm is a convenient one to use. But, it is not the best to use for computer algorithms (nor by humans on real examples!). The problem is that as the algorithm 'deletes' edges from the graph as it proceeds, whether an edge is a bridge or not will change over time: so we can't just do an initial pre-processing step, we have to contintually update the status of edges.

The reason that we want to choose edges that are not bridges is so that we don't disconnect the graph. This allows us to find the Eulerian circuit in one shot. The algorithm below takes a different approach. We build up the the circuit in a number of iterations.

First, we have an auxiliary routine that finds a path in a graph. It's the same basic idea as Fleury's algorithm, except we don't worry about whether an edge is a bridge or not.

```
FindCircuit(V,E,u)
  P ← ⟨⟩
  while (∃ unmarked (u,v) ∈ E)
        mark(u,v)
        P.append((u,v))
        u ← v
  }
  return P;
```

If a graph is Eulerian, and we apply the `FindCircuit` algorithm to an arbitrary vertex then we may find the Eulerian circuit. However, since we don't worry about whether the edges traversed are bridges or not, we might paint ourself into a corner: although we have found a circuit, . it may not contain all the edges.

Suppose that we have found the circuit $v_0,\ldots,v_k,v_0$. If the circuit doesn't contain all the edges, then at least one of the vertices in the circuit must still have an edge that is not yet marked. Let this vertex be v_j. We now repeat the `FindCircuit` from, v_j to get another circuit, say $v_j,w_0,\ldots,w_r,v_j$. We can now *insert* this circuit into the first circuit to get a new one: $v_0,\ldots,v_j,w_0,\ldots,w_r,v_j,v_{j+1},\ldots,v_k$. If this new circuit still doesn't contain the entire graph, we repeat the process. So our algorithm is

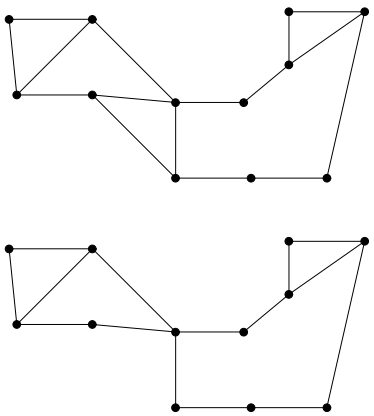
```
FindEulerian(V,E)
  Choose u
  EPath = FindCircuit(V,E,u)
```

```
while (len(EPath) < len(E))
        Find (x,y) ∈ EPath ∋ ∃ unmarked (x,w)
        NPath = FindCircuit(V,E,x)
        Insert NPath into EPath
}
```

7.2 Hamiltonian Circuits

At first sight, the Hamiltonian Circuit problem looks very similar to the Eulerian Circuit problem: is there a circuit which contains each vertex exactly once. However, it is a much more difficult problem. With the Eulerian circuit problem, given a graph, there is a simple property of the graph which says whether the graph has an Eulerian circuit or not. This is not the case with the Hamiltonian Circuit. There is no simple property with which tells us this, and the search for solutions is an ongoing one. We will explore this in more detail at the end of this section.

Exercise 7.1. *Do the following graphs have a Hamiltonian circuit?*



Definition 7.3. *A Hamiltonian cycle is a walk which contains every vertex of a graph exactly once, except for the first and last vertices which are the same. A Hamiltonian path is a path which contains every vertex in the path exactly once. G is Hamiltonian if it has a Hamiltonian circuit.*

Although there are no general categorisations of the Hamiltonian problem, there are some interesting results.

Theorem 7.5. *If G is graph of order $p \geq 3$ such that for all pairs of distinct nonadjacent vertices x and y, $deg(x) + deg(y) \geq p$, then G is Hamiltonian.*

Proof: We assume that it is not true. This will lead to a contradiction of something that we know to be true, and so if the assumption that the theorem is false is absurd, then the theorem must be true. If the theorem is false, then there must be at least one graph with order $p \geq 3$ such that all pairs of distinct nonadjacent vertices have $deg(x) + deg(y) \geq p$, and the graph does not have a Hamiltonian circuit. Now, such a graph cannot be complete, since all complete graphs have a Hamiltonian circuit. Why?

Therefore, there must be some graph $G = (V,E)$, with order with order $p \geq 3$ such that all pairs of distinct nonadjacent vertices have $deg(x) + deg(y) \geq p$, and G does not have a Hamiltonian circuit, but there are two vertices u and v such that $H = (V, E \cup \{(u,v)\})$ is Hamiltonian. Why?

So, consider H which is Hamiltonian. Every Hamiltonian circuit must contain (u,v) (why?).

This means that there is a Hamiltonian path in G between u and v. Let this path be P: $u = w_1, w_2, w_3, \ldots, w_p = v$. If w_i is adjacent to u and w_{i-1} is adjacent to v, then we could construct the following circuit: $u = w_1, w_i, w_{i+1}, \ldots, w_p, w_{i-1}, \ldots, w_1$. This is a Hamiltonian circuit in G which we know does not exist. So if w_i is adjacent to u, then w_{i-1} is not adjacent to v. So for every vertex that is adjacent to u, there is a vertex that is not v nor adjacent to v. This means that the number of vertices adjacent to u is less than or equal to the number of vertices in the graph less one (for v itself) less the number of vertices adjacent to v. Therefore $deg(u) \leq p - 1 - deg(v)$. Therefore $deg(u) + deg(v) \leq p - 1$. But, this is a contradiction of the assumption that all pairs of distinct nonadjacent vertices x and y have $deg(x) + deg(y) \geq p$. Thus there cannot exist a graph with order $p \geq 3$ such that all pairs of distinct nonadjacent vertices have $deg(x) + deg(y) \geq p$, and the graph does not have a Hamiltonian circuit.

Corollary 7.1. *If G is a graph of order $p \geq 3$ such that the minimum degree of a vertex in the graph is no less than $p/2$, then G is Hamiltonian.*

The Hamiltonian Circuit problem (HC) has a special place in Computer Science. It was one of the first problems to be discovered that was provably NP-complete. Being NP- complete means that just discovering whether a general graph is NP-complete or not can (probably) only be answered in time exponential to the number of vertices in the graph. More than that, if an algorithm of polynomial order does exist which solves the general case, then all NP-complete problems can also be solved in polynomial time. Another interesting aspect of this problem is how close the problem seems to be to the Eulerian circuit problem, which we can solve in linear time. How illusory this similarity is! Another example of such a pair of graph theory problems is the shortest path/longest path in a graph problems. The shortest path problem can easily be solved in polynomial time as we have already seen. Finding the longest path in a graph is an NP-complete problem - essentially, the only way to find the longest path is to try all possibilities.

A practical application of the Hamiltonian problem

In many planning activities, the solution to the Hamiltonian problem is a natural way to find an answer. Example: an architect may need to design a factory plant. This plant will consist of a number of different sub-parts or rooms. The needs of the factory will determine a number of

relationships between the rooms in the fac tory. Some rooms must be close together, while some must be apart. Some should be near the entrance to the building, some perhaps should not be. One approach to solving the problem is to represent the factory as a graph. Finding a suitable plan for the factory reduces to finding a Hamiltonian circuit for the graph.

7.3 Travelling salesperson problem

This problem is very similar to the HC problem. A travelling salesperson needs to visit n cities. In which order should s/he visit the cities so as to minimise the distance travelled? It can be very easily seen that this problem is also NP-complete, as it is harder than the HC problem. The HC problem is a special case of the travelling salesperson problem where the weight of each edge is one. If could solve the general case of travelling salesperson problem in polynomial time, then we could certainly solve the special case in polynomial time. There are number of approaches to finding sub-optimal solutions to this problem. The solution to this problem has many practical applications, for example, in industry, and so being able to find some solutions would be very useful. (Think of a robot arm which has to place spot welds on a piece of metal). The Monte Carlo method starts off at an arbitrary vertex, and then makes random choices about where to go next. Often, the walk produced is not too bad. If we make the assumption that the shortest path between two vertices is the edge between them, that is $c((u,v)) \leq c((u,x)) + c((x,v))$, then there are better heuristics to use. This is a reasonable assumption in many real-life situations. One solution would be to construct a minimum-weighted spanning tree, and traverse each edge twice. If we make the assumption that the graph is also complete, we can do better, and so "cut" corners to ensure that we don't have to traverse edges twice. This assumption is also reasonable in many situations. If you are ever confronted by a problem like this, you should be aware that there has been a lot of work done on this problem, both from a heuristic and theoretical approach, and you should ensure that you consult the literature.

7.4 Summary

In this chapter, we examined important types of circuits: Eulerian circuits, Hamiltonian circuits, and the travelling salesperson problem. For the Euler circuit, there is theorem which easily determines whether a graph has an Eulerian circuit. For a graph which is Eulerian, the Eulerian circuit can be found easily using Fleury's algorithm. Both the Hamiltonian and travelling salesperson problems are NP-complete. This means there is no easy general solution. There are, however, solutions for special cases and heuristic solutions. All the problems seen in this chapter are important both from a theoretical and practical point of view. Although we have not dealt in detail with the latter two problems, you should be able to recognise the problem in future, and be aware that much work has been done in this respect.

Chapter 8

Flows

The next class of graph algorithms we look at has many obvious applications, and some non-obvious ones too. Here are some examples.

Example 8.1. The local brewery company has three vats *A*, *B* and *C*. There is an elaborate pumping system to get from the vats to the outlet, as shown in Figure 8.1. Each pump has a capacity indicated by the weight of the edges. What is the maximum amount of beer that can be pumped to the outlet per hour?

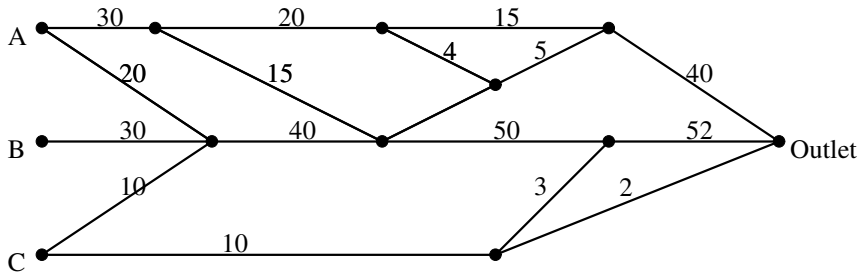


Figure 8.1: Simple Flow Example

Realistic examples of this sort are plumbing, sewage and irrigation schemes.

There are many other application areas:–

- A new housing scheme is being developed. Part of the development includes the building of roads. How many roads and of what capacity need to be built?
- A computer network is being constructed. What capacity do the different links in the system need to have.

- A new telephone exchange system is being built for a country. What should the capacity of the exchanges and the lines be to ensure that the system can cater for peak-time trunk calls?

Flow algorithms also have some non-obvious applications. They can be used to solve many resource allocation or matching problems. Given such a matching problem, it can often be represented as a graph, in a form where applying a flow algorithm can solve the original problem. This is an example of a very powerful technique called *reduction*: problem A reduces to problem B if an instance of problem A can be transformed into an instance of problem B such that the solution of the instance of problem B gives the answer to the original problem

Objectives:

By the end of the chapter you should be able to:-

- define the flow problem;
- prove properties of flows and cuts;
- analyse flow algorithms for their complexity;
- solve the flow problem for particular graphs;
- define various forms of the matching problem; and
- solve matching problems using various techniques.

8.1 Preliminaries

Given a weighted digraph $G = (V, E)$ with cost function c, a source vertex s and a target vertex t.

- $in(v)$ is the set of edges coming into v;
- $out(v)$ is the set of edges going out from v;
- $in(S) = \{(x, y) : x \in V - S, y \in S\}$; the edges going into S;
- $out(S) = \{(x, y) : x \in S, y \in V - S\}$; the edges leaving S;
- A set $C \subseteq E$ is a *cut* if every path between s and t contains at least one edge in C.
- A set $S \subseteq V$ determines a cut (the cut being $out(S)$) if $out(S)$ is a cut.

Exercise 8.1. *A, B and C determine a cut of the graph in Figure 8.2. If $S = A, B, C$ what is $in(S)$ and $out(S)$? Find another cut of the graph.*

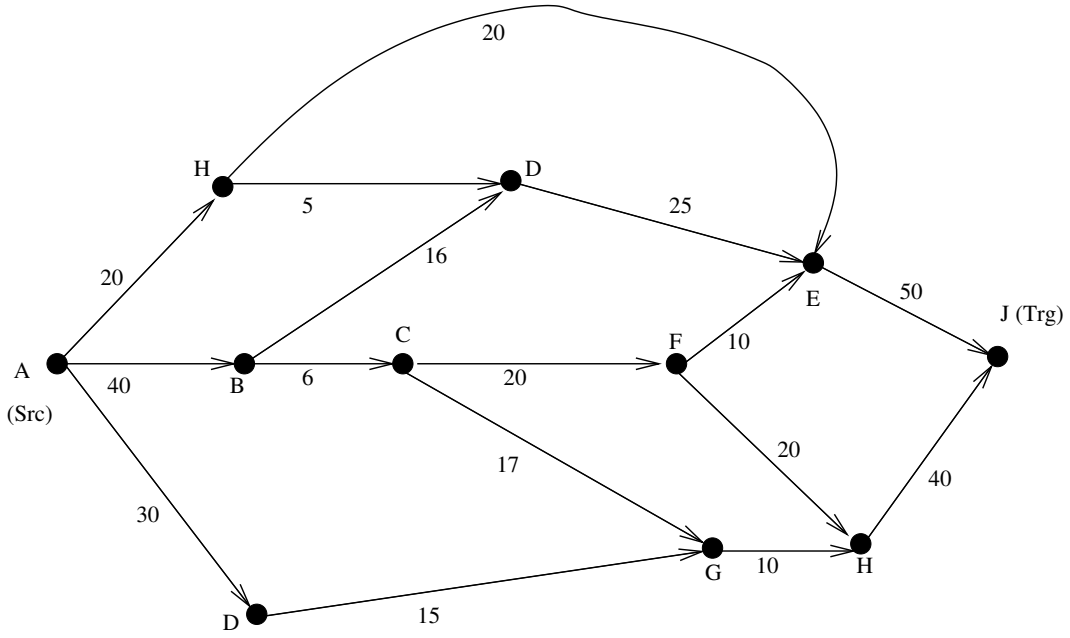


Figure 8.2: Example of cuts in flow graphs

Definition 8.1. The *flow* of the graph is a mapping, f, from E to the non-negative real numbers such that:-

- $f(e) \leq c(e)$ for every $e \in E$.
- for each $v \in V - \{s,t\}$, $\sum_{e \in in(v)} f(e) - \sum_{e \in out(v)} f(e) = 0$

The total flow of the graph (the net flow into t) is

$$F(G) \stackrel{\text{def}}{=} \sum_{e \in in(t)} f(e) - \sum_{e \in out(t)} f(e)$$

Theorem 8.1. *For every $S \subseteq V$ such that $s \in S$ and $t \in V - S$,*

$$F(G) = \sum_{e \in out(S)} f(e) - \sum_{e \in in(S)} f(e)$$

Proof:

1. $\sum_{e \in in(v)} f(e) - \sum_{e \in out(v)} f(e) = 0, \ \forall v \in V - S - t$

2. So, $\sum_{v \in V-S-\{t\}} (\sum_{e \in in(v)} f(e) - \sum_{e \in out(v)} f(e)) = 0$

3. $F(G) = F(G) + \sum_{v \in V-S-\{t\}} (\sum_{e \in in(v)} f(e) - \sum_{e \in out(v)} f(e))$
$= \sum_{v \in V-S} (\sum_{e \in in(v)} f(e) - \sum_{e \in out(v)} f(e))$

In the summation each edge in G will appear:-

- zero times in this sum if both its endpoints are in S
  - the edge contributes nothing to the sum
- once if its tail is in S and its head is in $V - S$
  - the edge contributes $f(e)$ to the sum since $e \in in(v)$ for some $v \in V - S$
- once if its tail is in $V - S$ and its head is in S
  - the edge contributes $-f(e)$ to the sum since $e \in out(v)$ for some $v \in V - S$
- twice if both its endpoints are in $V - S$
  - the edge contributes $f(e) - f(e) = 0$ since $e \in in(v)$ for some $v \in V - S$ and $e \in out(w)$ for some $w \in V - S$.

Therefore the only edges that contribute to the sum are those edges that leave S and do not end in S, and those edges which enter S but do not start in S.

$$\Longrightarrow F(G) = \sum_{e \in out(S)} f(e) - \sum_{e \in in(S)} f(e)$$

Definition: Gven a set $S \subseteq V$, the capacity of the cut determined by S is $c(S) = \sum_{e \in \text{out}(S)} c(e)$.

Exercise 8.2. *Consider the graph in Figure 8.2 and the set S defined in Exercise 8.1. Compute $c(S)$ for that graph.*

Theorem 8.2. *Given a weighted digraph G with cost function c and flow f, and a set S such that $s \in S$ and $t \in V - S, F(G) \leq c(S)$.*

Proof:

$$\begin{aligned} F(G) &= \sum_{e \in out(S)} f(e) - \sum_{e \in in(S)} f(e) \\ &\leq \sum_{e \in out(S)} f(e) \\ &\leq \sum_{e \in out(S)} c(e) \\ &= c(S) \end{aligned}$$

Theorem 8.3. Given a weighted digraph G with cost function c and flow f, and a set S such that $s \in S$ and $t \in V - S$, if $F(G) = c(S)$ then f is a maximum and the cut determined by S is of minimum capacity.

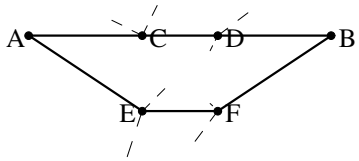
Proof: Let F^* be a total flow greater than F and K be a cut of minimum capacity. Then $F(G) \leq F^* \leq c(K) \leq c(S) = F(G)$.

This implies that F is of maximum flow and S is of minimum capacity.

These results are used in the Ford/Fulkerson algorithm.

8.2 Ford/Fulkerson Algorithm

To illustrate how this algorithm works, consider the simple example below.



Think of this as a partial extract from a larger graph which represents an irrigation system where the goal is to pump water from A to B. There are two paths drawn between A and B: $ACDB$ and $AEFB$. Consider the edges CD and EF. There are two ways in which the flow along the graph could be increased:-

- If the flow along CD is less than the weight of the edge (i.e. if $f(CD) \leq c(CD)$) then we can increase the flow to B. The difference between the two, $\lambda = c(CD) - f(CD)$ is the slack. If $\lambda = 0$, the we say that the edge is *saturated*. If $\lambda > 0$, then the flow can be increased by λ.

- If the water is flowing the 'wrong' way along EF (for some reason), then if this flow could be reduced somehow, we could also increase the flow to B.

The Ford/Fulkerson approach uses both these heuristics to find the maximum flow. The paths from the source vertex to the target vertex are examined. For each path, the algorithm tries to increase the flow to the target using the above two techniques. An *augmenting path* from s to t

is a path (which may be an undirected path) from s to t which can be used to increase the flow from s to t. Once we find the augmenting path, we change the flow along the augmenting path, increasing the flow to the target. We then repeat this process until no more augmenting paths have been found. This is the outline of the algorithm.

```
∀e ∈ E, f(e) = 0;
while ∃augmenting paths{
      find augmenting path;
      update flow
      }
```

A. Augmenting paths and residual networks

To find the augmenting path efficiently, the concept of a residual network is used.

For each pair of vertices in the graph, we keep its *residual capacity*, $c_f(u,v)$, defined by

$$c_f(u,v) \stackrel{\text{def}}{=} c(u,v) - f(u,v).$$

An edge is *saturated* if its residual capacity is 0.

For an edge e, $c_f(e)$ gives the extra flow that e can take. But the residual capacity is also defined backwards along an edge. Suppose we have an edge (u,v) in our original graph; $c_f(u,v) = c(u,v) - f(u,v)$ is the residual capacity along (u,v). But in the residual graph, there is not only an edge going from u to v but also an edge going from v to u. Why? Because sometimes we might be able to increase the overall flow to the target by reducing the flow along a particular edge. It is convenient algorithmically to represent reducing the flow along the edge as increasing the flow backwards along the edge.

Formally, the residual graph $G_f \stackrel{\text{def}}{=} (V, E_f)$, where

$$E_f = \{(u,v) \in V \times V : c_f(u,v) > 0\}.$$

Example 8.2. *Suppose we have an edge (u,v) in the graph such that the capacity of (u,v) is 20 and the current flow along (u,v) is 15. Then $c_f(u,v) = 5$ (we can pump up to 5 more units along the edge).*

On the other hand in this scenario, $c(v,u) = 0$, and $f(v,u) = -15$. Thus $c_f(v,u) = 15$ (we can pup up to 15 more units back *along the edge).*

Finding an augmenting path from the source to the target in this framework is easy. Construct the residual network, find a path using a standard algorithm from the source to the target. Compute the minimum residual capacity along this path, and update all the edges along the path.

Where do cuts come in? How does this relate to cuts and the theory we've just seen? Eventually, we shan't be able to find any more augmenting paths since there will be no path from source to the target that goes through any edge with a positive residual capacity. All the edges with a

zero residual capacity together then must form a cut. In particular, if we let S be the source vertex and all the vertices reachable from the source, then $out(S)$ forms a cut and by Theorem 8.3 we have found the maximum flow.

Which path algorithm to use? The shortest path algorithm gives the best results.

8.2.1 The algorithm

```
foreach (u,v) ∈ E {
  f(u,v) = 0;
  c_f(u,v) = c(u,v);
  c_f(v,u) = 0;
};
while ( true ){
      path = BreadthFirst(G_f,s,t);
      if (path does not exist)  break;
      λ = min{c_f(u,v) : (u,v) ∈ path};
      foreach (u,v) ∈ path {
        c_f(u,v) = c_f(u,v) − λ;
        c_f(v,u) = c_f(v,u) + λ;
        if ((u,v) ∈ E)
           f(u,v) = f(u,v) + λ;
        else
           f(v,u) = f(v,u) − λ;
      }
  }
```

In summary, the algorithm finds a path from s to t in the *residual network*. The minimum weight in this path is found — this corresponds to the minimum 'slack' along any edge in this path. The flow along this augmenting path is then updated, which increases the overall flow to the target. This process is repeated until no more paths can be found.

Exercise 8.3. *: Find the maximum flow from A to the outlet in the Figure 8.3*

8.2.2 The proofs

Theorem 8.4. *The flow in a graph is a maximum if, and only if, there are no augmenting paths.*

Proof: Let the source vertex be s, and the target vertex t, the flow in the graph be f and the total flow in the graph be $F(G)$.

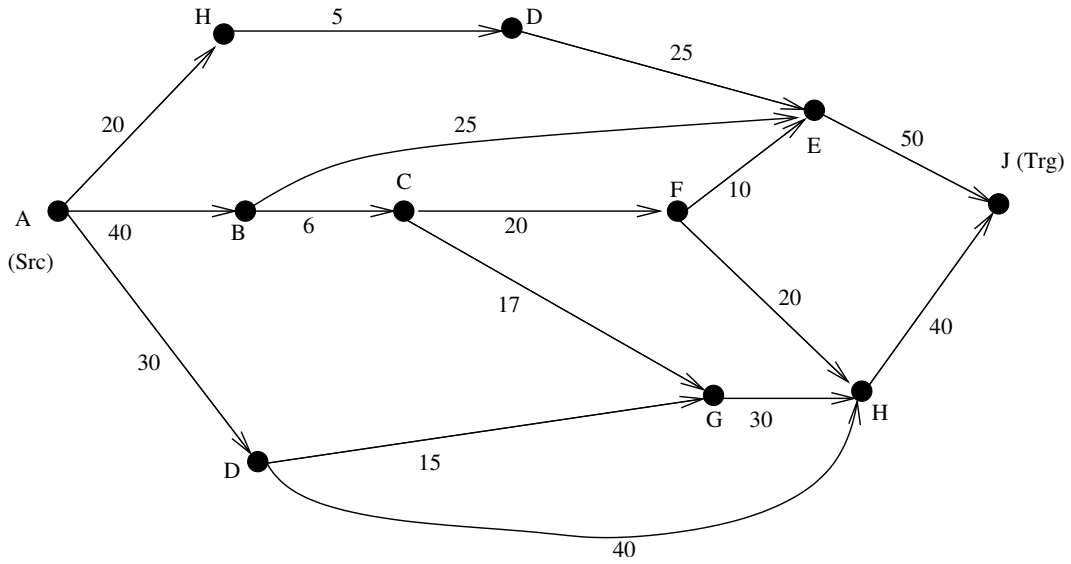


Figure 8.3: Flow example

1. Suppose that there is an augmenting path, with minimum slack λ. The flow of each edge in the path can be incremented by λ, and so the total flow will increase. Therefore, the original flow could not be a maximum.

2. On the other hand, suppose that there is no augmenting path between the s and t.

3. This implies that t is not reachable from s.

4. Let S be those vertices reachable from s

   - $s \in S$,
   - $t \in V - S$.

5. Each edge in $out(S)$ must be saturated, and each edge in $in(S)$ must have no flow. Thus by Theorem 8.2, $F(G) = c(S)$. By Theorem 8.3 f must be such that $F(G)$ is at a maximum.

This algorithm was the first solution to the problem - there are problems with it from an algorithmic point of view. First, if all the weights are irrational, then the algorithm may not terminate. Second, the algorithm is not efficient. In the next section we see a more efficient algorithm. A corollary to the theorem above is known as the max flow-min cut theorem and is one of the classic results of graph theory.

Corollary 8.1. *In a weighted digraph $G = (V,E)$ with cost function c and source and target vertices s,t, the maximum total flow of G is the minimum capacity of a cut.*

Proof:

1. If C is a cut of minimum capacity, and F a maximum total flow,Theorem 8.3 says that $F \leq c(C)$. We need to show that $c(C) \leq F$.

2. Consider the construction of S in the theorem above; $c(C) \leq c(S))$ by the definition of C. But $c(S) = F$ (as argued above), and the result follows.

8.2.3 Analysis of algorithms

When all the capacities are integral the Ford-Fulkerson algorithm is guaranteed to run in time $O(f^*m)$, where m is the number of edges in the graph and f^* is the maximum flow:

- The cost of finding a path and updating the flows is $O(m)$;

- In the worst case, we can only be guaranteed of improving the flow by 1 each time, so we might need f^* iterations.

While usually the worst case does not happen, it is not hard to find examples where it does. Hence if f^* is very big, this algorithm may be bad. As alluded to earlier if the weights are not integral the performance of the algorithm may be very bad.

A more sophisticated analysis tells us that if a breadth-first search is used to find the augmenting path, then the algorithm runs in $O(nm^2)$ time where n is number of vertices. See [4] for details.

8.3 Generalisations and added constraints

In this section we see how we can find flows with more general conditions on the graph. There are a number of other generalisations and restrictions which have practical application.

8.3.1 Vertex constraints

In many problems, there are also constraints on the vertex. For example, in traffic systems, the vertices may represent intersections. An intersection will generally not be able to carry all the traffic which the roads going into the intersection bring in. This can also be easily managed in the framework we have built. Suppose each vertex v, has constraint c_v on the flow going through it, where the flow going through the vertex is

$$\sum_{e \in out(v)} f(e).$$

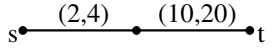
Replace v with two vertices i_v and o_v. Each edge going into v now becomes incident to i_v and each edge coming from v now becomes adjacent to o_v. We also add an edge e_v which connects i_v and o_v; this edge is given the weight c_v. The maximum flow problem can be solved now using the techniques developed earlier.

8.3.2 Multiple sources and multiple targets

Multiple sources and targets can easily be dealt with. Suppose that there are r sources, $s_1, \ldots, s_r$. Just add a dummy node s, with an edge from s to each s_i. Each edge is given an appropriate weight, perhaps ∞. Multiple targets can be dealt with in the same way. All the theorems and the algorithms we have done will apply. This means that what we have done can apply to general systems, like a traffic system, where there are multiple sources and multiple destinations.

8.3.3 Minimum constraints on edges

Up to now we have only considered the case where there is an upper bound on the flow of an edge, namely the weight of the edge. In many real problems, there may also be lower bound constraints on the flow. Thus we may have to find an f which has the added constraint that $l(e) \leq f(e)$, where l is a lower bound on the flow of an edge. (An example where this may arise is in a pumping system across large distances. It may only be economically feasible for a particular pipeline to operate if it has a minimum flow). In a case like this, there will be some networks which do not have feasible flows. The upper bounds and lower bounds may be contradictory. The simple graph below gives an example. Each edge is labelled with (x, y), where x represents the lower bound on the flow and y represents the upper bound on the flow.



For graphs which do have feasible flows, the flow problem can be solved fairly easily using the techniques above. Let $G = (V, E)$ be a flow graph with minimum and maximum flow constraints, with source s and target t. What we do is add two new vertices s' and t'. For each vertex x, we add an edge (s', x) with weight $\text{in}(x)$ and an edge (x, t') with weight $\text{out}(x)$. For each edge e with upper-bound c and lower bound l, change the weight to $c(e) - l(e)$. Edges should be added both ways between the original source and the original target, with weight ∞. Once this process has been completed for each such edge, we compute the maximum total flow from s' to t'. If this total flow is equal to the sum of the weight of the edges of s (or equivalently the sum of the weight of the edges of t), then a feasible flow exists. Note that this process finds a feasible flow, it does not produce the best feasible flow.

8.4 Matching

Matching – pairing elements with each other – has many real applications. Many resource allocation problems can be phrased as a matching problem. Here are some examples:

- Given a set of class rooms, each with certain properties (e.g. size, shape of room), and a set of courses each with certain requirements for rooms, allocate each course a class room.
- Suppose we are given a set of lecturers and a set of courses. Each lecturer is capable of teaching a certain subset of courses. Allocate lecturers to each of the courses.

We look at the following form of the matching problem. Let A and B be two sets of n_a and n_b elements, respectively. Let $P \subseteq A \times B$ be given. Find a biggest $M \subseteq P$ such that each element of A and each element of B appears at most once. This is knows as a *maximum matching*. (We assume here that the sets A and B are disjoint, though it is possible to relax this assumption.)

- We want to match as many elements of A as possible with exactly one element of B (and vice-versa).
- P is a set of pairs, each pair containing one element of A and one of B.

  If $(a,b) \in P$ it means that we can match a with b.

  We can think of P as the set of potential matches.

  If both (a,b) and (a,c) are elements of P we could not choose both pairs to be in our matching.
- M is a biggest subset of P which partners an elements of A with elements of B so that each element is partnered at most once. There could be many such Ms, or there could be exactly one.

  What P is determines whether it is possible to match all the As and all the Bs.

  It is only possible to partner every element in A and every element in B if $n_a = n_b$ (and again, this still depends on P).

There are a number of variations and generalisations of the matching problem, and we shall explore some of them later.

8.4.1 Examples

Example 8.3. *Let:*

- $A = \{a_0, a_1, a_2, a_3\}$;
- $B = \{b_0, b_1, b_2\}$;
- $P = \{(a_0, b_2), (a_1, b_0), (a_1, b_1), (a_1, b_2), (a_2, b_0), (a_3, b_1)\}$.

Then a possible matching is: $M = \{(a_0, b_2), (a_1, b_0), (a_3, b_1)\}$. *Can you find another matching?*

Example 8.4. *The list below shows for each lecturer, the courses they can teach:*

- *Lecturer A: COMS100, COMS101, COMS203, COMS305, COMS307*
- *Lecturer B: COMS102, COMS203, COMS308*
- *Lecturer C: COMS101, COMS203*
- *Lecturer D: COMS101, COMS203, COMS307*

- *Lecturer E: COMS100, COMS305, COMS308*
- *Lecturer F: COMS203, COMS305*
- *Lecturer G: COMS102, COMS307*

Can you allocate lecturers to courses so that each lecturer has one course, and each course has one lecturer?

8.4.2 The matching problem as a graph problem

We first look at how a matching problem can be viewed as a graph problem that can be solved using a flow algorithm.

It is easy to see how a matching problem can be viewed as a bipartite graph. The two sets are the partitions and vertices have edges between them if they are in the pair. For example, the matching problem of Example 8.3 can be depicted as the graph in Figure 8.4. The edges show the associations given by P, and of those, the thick lines show the matching.

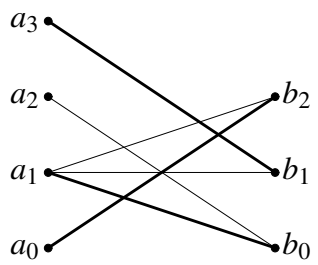


Figure 8.4: Matching problem viewed as a bipartite graph

An instance of the matching problem, i.e. sets A and B and a $P \subseteq A \times B$, can be seen as a bipartite graph. The vertices come from A and B and there is an edge between an a and a b if $(a,b) \in P$.

8.4.3 Algorithm

Given a matching problem phrased as a bipartite graph, modify it slightly to a flow problem. Let $G = (A \cup B, E)$ be a bipartite graph (all edges join a vertex in A with a vertex in B). The corresponding flow problem is a directed graph $G = (V', E')$, where

- $V' = \{s,t\} \cup V'$;
- $E' = \{(a,b) \in E : a \in A, b \in B\} \cup \{(s,a) : a \in A\} \cup \{(b,t) : b \in B\}$

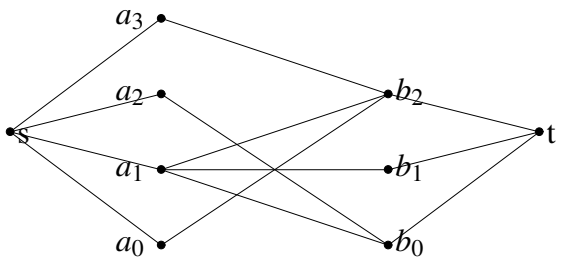


Figure 8.5: Matching problem viewed as a flow problem: edges go from left to right.

All the edges are weighted 1. This is shown in Figure 8.5. Now, we can use a flow algorithm to find a maximum flow. This maximum flow gives us a maximum matching: those edges between A and B that have a flow of 1 are those edges in the matching. We can be sure that each vertex $a \in A$ can be involved in one match at most since it is only supplied with one unit of flow (and symmetrically, each b can only be in one matching because it can only supply one unit of flow to the dummy target). Moreover, each flow in the graph corresponds to a matching and vice-versa: so the maximum flow defines the maximum matching. This is formalised in the following theory.

Theorem 8.5. *Let $G = (V,E)$ be a bipartite graph with V being partitioned between A and B. Let $G' = (V',E')$ be the corresponding flow network. If M is a matching in G then there is an integer valued f in G' with value $|f| = |M|$. Conversely, if f is an integer-valued flow in G', then there is a matching M in G with size $|M| = |f|$.*

Theorem 8.6. *Given an instance of a flow problem, if the capacity of all edges in the graph is integral, the maximum flow found by the Ford-Fulkerson algorithm is integral.*

Theorem 8.7. *The cardinality of a maximum matching in a bipartite graph G is the value of a maximum flow in its corresponding flow network G'.*

8.5 Summary

In this section, we defined what a flow in a graph was and gave some examples of where flows were useful. Then, using the definition we proved properties of flow; the most important of which is that if the total flow of a graph is equal to the capacity of a cut, then the flow is a maximum. From this result we can prove that the Ford-Fulkerson algorithm is correct. We then looked at the analysis of the algorithm and how to improve the performance.

Chapter 9

Draft: Planarity

The topics covered are interesting from a theoretical point of view. They are also useful in many practical applications. By the end of this chapter you should be able to:-

- define the concepts of planarity;
- identify planar graphs, and properties of planar graphs
- prove and use characterisation of planar graphs
- define the thickness of a graph
- define the chromatic number of a graph
- use an approximate colouring algorithm

A graph is an abstract mathematical object. However, it's often useful to represent the graph by drawing it graphically. Or, a graph may represent some real object or objects and so a natural representation is found by drawing the graph. Usually, we depict vertices in a graph as blobs, and the edges as lines connecting the blobs. Where do we draw the graph? It seems the most natural (and it's certainly the easiest) way of drawing the graph is on a two-dimensional plane (like a piece of paper, or blackboard). But, there's no reason, in general, that we shouldn't draw the graph on a sphere, or in three-dimensions. The nature of many graph problems makes, however, the two dimensional representation of graphs the most natural thing to use, and this is what we explore here.

Definitions

: An *embedding* of a graph in the plane is the pictorial representation of a graph on the plane. An embedding of a graph in the plane is called a *plane graph* if no edges intersect. A graph is said to be *planar* is it has a plane embedding. Note that a graph will have an infinite number of embeddings, and that a planar graph will have some plane and some non-plane embeddings. Not

all graphs are planar, and a plane embedding of the graph may not be obvious. This is what we explore here.

Note that if we embed a graph in three dimensional space ,then there is always an embedding of the graph which is a plane- a way of representing the graph so that no edges intersect.

Recall the following definitions:

- Denote the complete graph of order n (with n vertices) as K_n. Note that each vertex in K_n has degree $n-1$
- A bipartite graph $G=(V,E)$ is a graph such that V can be partitioned into two disjoint *sets*, V_1 and V_2 so that each edge in E joins one vertex in V_1 with a vertex in V_2. A graph is a *complete bipartite graph* if every vertex in V_1 is adjacent to every vertex in V_2: if the number of vertices in V_1 is r, and the number of vertices in V_2 is s,then this is denoted $K_{r,s}$.
- If a graph G is embedded in the plane then a *region* of the graph is a connected section of the plane which is bounded by some set of edges in the graph. Note that in a finite graph, there will always be one exterior or infinite region.

Exercise 9.1. *1. Draw K_4, K_5, and K_6.*

2. Draw a bipartite graph which is not a complete bipartite graph.

3. Draw $K_{4,2}$ and $K_{3,3}$.

Theorem 9.1. *If G is a connected plane graph with p vertices, q edges, and r regions, then $p-q+r=2$.*

Proof: The proof is by induction on the number of edges.

- *Basis step*

  If $q=0$, since the graph is connected, $p=1$ and $r=1$. Therefore, $p-q+r=2$.

- *Induction hypothesis*

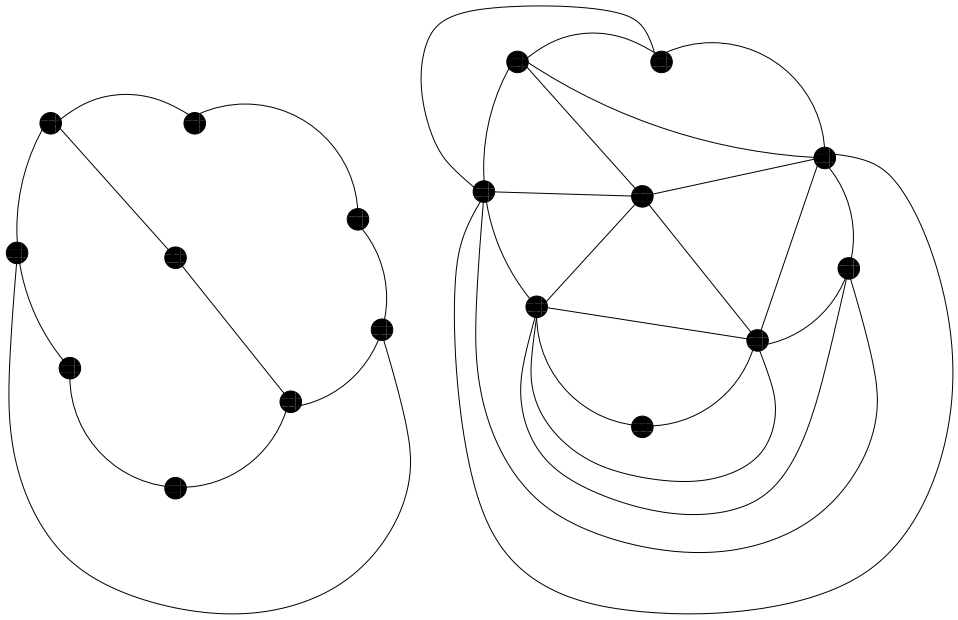
  For any connected graph with q edges, p vertices and r regions, $p-q+r=2$.

- *Induction step*

  1. Let G be a plane graph with $q+1$edges, p vertices and r regions. We must show that $p-(q+1)+r=2$.
  2. If G is a tree, then $p=q+2$ and $r=1$. So $p-(q+1)+r=q+2-(q+1)+1=2$, and the theorem is true.
  3. If G is not a tree, then it has a cycle. Let e be an edge in the cycle,.
  4. Remove this edge from G to get the graph H. H has $r-1$ regions, p vertices and q edges.
  5. Since it has q edges, the induction hypothesis holds and so we get $p-q+(r-1)=2$,
  6. Rearranging we get $p-(q+1)+r=2$, which is what we wished to show. The theorem therefore holds. This is called Euler's formula.

9.1 Some characterisations of planar graphs

Suppose we have a plane graph. Can we add an edge into the graph, and keep it plane? As long as there is a region which is bounded by a cycle of length four or more, it is possible to add edges. Consider the graph below on the left. It has a region bounded by a cycle of length greater than three. The graph on the right has one edge added. There are no regions bound by a cycle of length greater than three. A planar graph to which no edge can be added without making a crossing is called a *maximal* planar graph. In a maximal planar graph, every region is bounded by a triangle: the number of edges bounding each region is exactly three.



(a) A non-maximal planar graph (b) A maximal planar graph

Lemma 9.1. *If $G = (V,E)$ is a maximal planar graph with p vertices, q edges and r regions, then $3r = 2q$.*

Proof:

1. Let the regions in the graph be $R = \{f_1, \ldots, f_r\}$, and the number of edges bounding region i be $n(i)$.
2. By the above discussion $n(i) = 3$ for all i, as the graph is a maximal planar graph.
3. So, $\sum_{f_i \in R} n(i) = 3r$
4. In this enumeration, every edge is counted twice: once for each region which it bounds.
5. Therefore $3r = 2q$.

Theorem 9.2. *If $G = (V,E)$ is a maximal planar graph with p vertices, q edges and r regions, then $q = 3p - 6$.*

Proof:

1. Euler's formula says $p - q + r = 2$
2. This implies$3p - 3q + 3r = 6$
3. Substituting in $3r = 2q$ from lemma 5.2 yields
4. $3p - 3q + 2q = 6$ which, simplified, gives
5. $q = 3p - 6$, which is what we wished to show.

These two results allow us to characterise planar graphs.

Theorem 9.3. *If $G = (V,E)$ is a planar graph with p vertices and q edges then $q \leq 3p - 6$.*

Proof:

1. Add edges to G until we get a maximal planar graph H.H has p vertices and $q*$ edges
2. By theorem 9.3, q^* = $3p - 6$.
3. But $q \leq q^*$.
4. Therefore $q \leq 3p - 6$.

This is not a complete characterisation. It does *not* say that if $q \leq 3p - 6$ then the graph is planar. It does say that if $q > 3p - 6$, then the graph is not planar.

Another incomplete characterisation of planar graphs is given by the next theorem.

Theorem 9.4. *If $G = (V,E)$ is a planar graph, then there exists a vertex which has degree no greater than five.*

Proof:

1. Let $V = v_1, \ldots, v_p$. If $p \leq 6$, then the result must be true since no vertex can have degree greater than five.
2. Let there be q edges.
3. $\sum_{i=1}^{p} deg(v_i) = 2q$.
4. We know from theorem 9.3, that $q \leq 3p - 6$
5. So $2q \leq 6p - 12$ which implies that $2q < 6p$
6. Assume that all vertices in the graph have degree greater than 5, then $\sum_{i=1}^{p} deg(v_i) \geq 6p$
7. So $2q \geq 6p$
8. This is a contradiction. Therefore, all the vertices in the graph cannot have degree greater 5.

9.2 The two basic non-planar graphs

All non-planar graphs - those graphs which do not have a plane embedding - are related to two basic graphs $K_{3,3}$ and K_5. In this section, we will prove that they are both non-planar. In subsequent sections, we will see how we can use this to discover whether more general graphs are planar or not.

Theorem 9.5. *If $K_{3,3}$ is not planar.*

Proof: Assume that $K_{3,3}$ is planar.

1. It has 6 vertices and 9 edges.
2. Substituting into Euler's equation $(p-q+r=2)$, gives $r=5$.
3. There are no cycles of length three in the graph. (This must be the case since $K_{3,3}$ is a bipartite graph).
4. Therefore, all cycles are of at least length four.
5. All regions in the graph, therefore, are bounded by at least four edges.
6. Using the same notation as lemma 5.2, we have $\sum_{fi \in R} n(i) = 2q$ and $\sum_{fi \in R} n(i) \geq 4r$
7. Therefore $2q \geq 4r$, or $2r \leq 9$.
8. But we know that $r=5$ and so $2r=10$, and so this is a contradiction. Therefore $K_{3,3}$ is not planar.

Theorem 9.6. *K_5 is not planar.*

Proof:

1. The sum of the degree of all the vertices is twice the number of edges. Each vertex in K_5 is of degree four, so the sum of the degree of all the vertices is twenty, and so there are ten edges in the graph.
2. But theorem 9.3 says that if the graph is planar $q \leq 3p-6$.
3. Therefore, $q \leq 9$, which is impossible.
4. Therefore, the graph cannot be planar.

Corollary 9.1. *Any graph which contains a non-planar subgraph is non-planar. Any graph which contains K_5 or $K_{3,3}$ is non-planar.*

Proof: If we can't draw G without crossing edges, we certainly won't be able to manage if we have even more vertices and edges to draw. As K_5 and $K_{3,3}$ are non-planar, any graph containing either of them is also non-planar.

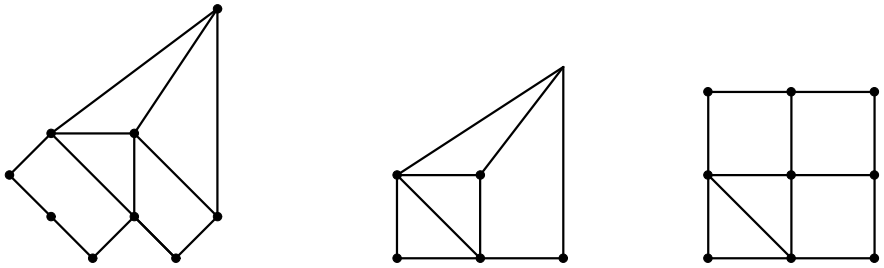
9.3 More tools to find non-planar graphs

Corollary 9.1 can be extended to be more useful. We will see that if a graph contains a sub-graph which "looks like" K_5 or like $K_{3,3}$ then it is not planar and if it doesn't then it is planar. Before showing this "looks like" must be carefully explained. This is done by defining the following operation on graphs.

Definition 9.1.

- *A* subdivision *of an edge (x,y) is the removal of the edge from the graph, and the adding to the graph of a new vertex w, and new edges (x,w), and (w,y). We can think of subdividing an edge as inserting a new vertex in the middle of it.*
- *A graph is* homeomorphic *from G if it is isomorphic to G or it can be obtained from G by some sequence of subdividing edges.*
- *A graph H is* homeomorphic *with G if they are both homeomorphic from some other graph F.*

Exercise 9.2. *: Show that the graph on the left and the graph on the right are homeomorphic with each other by showing that they are both homeomorphic from the graph in the middle.*



(c) Homeomorphic graph 1 (d) Base graph (e) Homeomorphic graph 2

Suppose a graph G is non-planar. Then by sub-dividing an edge, we aren't going to be able to make it planar. This leads to the next lemma.

Lemma 9.2. *If a graph H contains a graph G' which is homeomorphic from a non-planar graph G, then H is non-planar.*

Proof:

1. G' is non-planar since it is homeomorphic from G. If there is a problem drawing G in the plane since some edge (x,y) crosses some other edge then there will still be a problem drawing it if (x,y) is sub-divided.
2. By corollary 9.1, H is non-planar since it has a non-planar sub-graph.

The mathematician Kuratowski show that all non-planar graphs have a sub-graph homeomorphic to K_5 or $K_{3,3}$. This leads to the main result of this section.

Theorem 9.7. *A graph G is planar if, and only if, it contains no subgraph homeomorphic with K_5 or $K_{3,3}$.*

Proof:

- We have shown most of the one way of the proof in the previous lemma. The rest of the proof is not difficult, but it is long as there are a number of cases to consider. This theorem is proven in many graph theory text books.

9.4 Algorithms to find planarity

There are a number of algorithms to determine planarity. Some simple tests follows.
If $G = (V, E)$ is a graph then:-

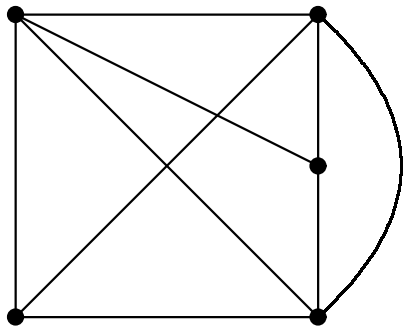
1. If $\mathrm{E} > 3p - 6$, then the graph is not planar
2. If the graph is disconnected, consider each component separately
3. If G is a tree, then it is planar
4. If the graph has an articulation point then, consider each bicomponent separately.
5. If there is a vertex of degree 2, we can replace it by joining its neighbours. This is called contracting the graph. A graph is planar if and only a *contraction* of the graph is planar.

Algorithms to determine whether a graph is planar all use tests like this. The basic idea is to try break the graph down in to smaller subgraphs, and then test the subgraphs. The algorithm of Hopcroft and Tarjan works by finding a cycle in the graph (using a depth- first search), and using this as the framework for the rest of the graph. Drawing the cycle in the plane divides the plane in two - that part of the plane within the cycle and that part of the plane outside the cycle. The rest of the graph consists of segments, "hanging off" the cycle. The algorithm then tries to embed each segment, either inside the cycle, or outside the cycle. If the algorithm succeeds then, clearly the graph is planar. And, if it does not succeed (then not so clearly) it is not planar. Again, although the proof and the algorithm are not difficult, they are too long to study in this course. You should, however, be aware that there are solutions to the planarity problem, and that the best algorithms run in $\oslash(n)$ time, where n is the number of vertices in the graph.

9.5 Thickness of graphs

In many graphs which represent real problems, the graph represents some form of network. The edges may represent wires, or circuits on a printed circuit board, or a plumbing system, or a

road system. In all these situations planar graphs are desirable. For example on one wafer of a silicon chip, "wires" or edges can't cross. Thus, knowing that the graph is not planar enables us to see whether we can resdesign the chip so as to have a non-planar graph. Unfortunately, many problems defy this sort of redesign, and we are still left with a non- planar situation. Consider the graph below, which represents five cities. We wish to have air routes between each pair of cities. The problem with this is that some routes will have to cross each other. This will make flight scheduling very difficult if all planes fly at the same altitude. Not having routes between each pair of cities may not be economically or socially desirable, and so we have to think of another solution.



One solution is to make the different routes fly at different altitudes. This will make flight scheduling much easier. This concept can be formalised.

Definition 9.2. *The thickness of a graph G, denoted $t(G)$, is the smallest number of planar graphs which can be superimposed to make up G. Clearly the thickness of a planar graph is 1.*

In the example above, we can split the routes in two, each group of routes flying at different altitudes. This stops the possibility of collisions (well reduces the possibility any way).
If we superimpose these two graphs, we get what we started off with. The graph is $K_{5,5}$, which is non-planar. Therefore its thickness is greater than one. We have managed to find two planar graphs which when superimposed give us the graph. Therefore its thickness is two.

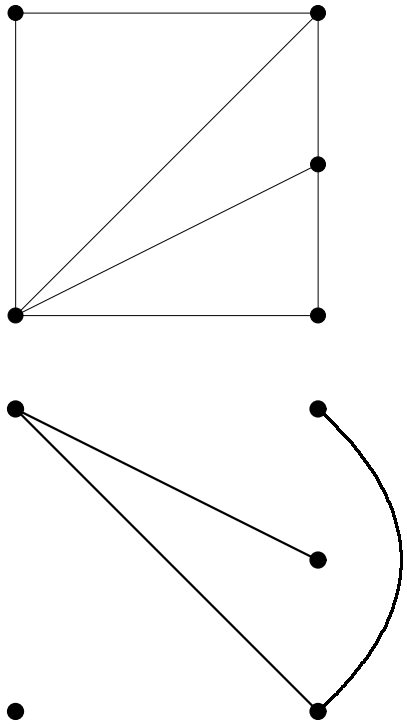
Theorem 9.8. *If $G = (V,E)$ is a graph with p vertices and q edges, then the thickness of the graph is constrained by the following inequality:-*

- $t(G) \geq \lceil \frac{q}{3p-6} \rceil$

Proof:

1. If the graph is decomposed into a set of planar subgraphs, then each subgraph must obey theorem 9.3 the number of edges is less than or equal to three times the number of vertices minus 6.

2. Suppose the thickness of the graph is t. This means that the graph can be decomposed into t subgraphs, each with p vertices and q_i edges, where $\sum_{i=1}^{t} q_i = q$



3. In each planar subgraph, we have $q_i \leq 3p-6$
4. Summing for each planar subgraph gives $q \leq t(3p-6)$
5. Hence $t \geq \frac{q}{3p-6}$, (assuming at least three vertices)
6. From which the result of the theorem follows since t must be a whole number.

Appendix A

Program code

This appendix contains working Java code for some of the algorithms presented earlier.

A.1 PERT code

A.1.1 Program notes

1. We represent tasks as a directed graph. We can make some savings in costs and representation since for each vertex all outdoing edges have the same weight. Therefore, we just store the length of the task separate and we need only keep track of whether there is an edge from x to y; the cost of the edge from x to y is the length of the task x.

2. Edges to/from start and end dummy node

   Any node which has no predecessor nodes given in the input data file should be considered as having the start dummy node as their predecessor. Similarly, any nodes not having any successor nodes given in the data file should be considered as having the end dummy node as their successor.

   To determine this, for each vertex we scan through the entries in the adjacency matrix to determine whether they have predecessors or successors. If not, we add the appropriate edges.

   Note as an example of defensive programming, if we find a node that has no predecessor or successor nodes, we print out a warning. This situation could be – but not necessarily – an indication of an error in the data file and so it is worth indicating.

3. GetTasks reads in the first part of the data file. It reads in the list of the tasks. Each task is given on a line. Each line contains the number of the task, the length of the task, and the name of the task.

   The tasks must be consecutively numbered from 1. Note, that it may appear redundant to have the task numbers in data file at this point since the tasks are consecutively ordered from 1. The redundancy is useful as a way of reducing errors in the data file. The second

part of the data file contains the relationships – an ordered list of pair of task numbers. For the human creating the data file, it is useful to have the association of number and name of task explicitly in the data file.

Note how the GetTasks method checks the consistency of the data file to ensure that tasks are consecutively numbered and there are a correct number of tasks. While not necessary, we can check for a number of errors in the data file.

- note how the Java exceptions method is used to do the error handling. Basically the "throw" keyword triggers the reporting of an error or exception. We could provide code that "catches" the error, but in this case we don't and so the program will halt printing out the given error message as well as giving an indication of where the error occured. Don't worry about the details of the syntax.

4. The algorithm as presented borrows from the shortest path algorithm and keeps a list of the "fringe" vertices added at each step. However, here we don't actually need to know what the "fringe" vertices are, we just need to know *whether* for each level we add new vertices. When we do a round in which we don't add any fringe vertices, we can just stop. Note that at the beginning of each round we set new_fringe to true, and that whenever we find a vertex that is a successor to a vertex in the current round, we set new_fringe to true.

5. In the concrete program we have to specify a lot more than in the algorithm. The algorithm just says

```
foreach w such that label[w]==i
   foreach (w,x) in E
```

Our program implements this as follows

```
for (w=0; t<numvertices; t++) {
    if (label[w] == i) {
        for (x=0; x<numvertices; x++) {
            if (edge[w][x]) {
```

The outer for loop goes through each vertex; the if statement thenn checks the "such that" clause to find which vertices are labelled with *i*.

The inner loop with its associated if statement then checks the vertices one by one to find which edges leave the vertex *w*.

6. This algorithm looks very similar to the bfs-tree algorithm. There are two key differences. The first is that in the bfs-tree algorithm we are more choosy in the code shown in point 5 above. In the bfs-algorithm, not only must there be an edge between (w,x), but x must be unlabelled. In the est-algorithm we don't have this latter requirement. This is how we implement the the definition of est

   ```
   est(x) = max{est(w)+length(w): (x,x)  in E}
   ```

   The complementary difference is that in the bfs-algorithm, once a vertex is labelled with a number i, the label remains unchanged. In the est-algorithm, we update the label of a vertex, v, whenever we find a predecessor in the graph with a label greater than or equal to v. This enables us to propagate all the dependencies up the graph.

   Note that this means that the est-algorithm will be more expensive than the bfs-one.

7. Notes 4-6 apply to the computation of LFT as well.

8. ShowCriticalPath presents the critical path as shown in the parent array. The parent array allows us easily to print out the critical path backwards, but we really want to print it out forwards.

   A simple recursive algorithm allows us to do this. Essentially, we reverse the list as given by parent. If we think of a list as a first element followed by the rest, the reverse of the list is just the
   (reverse of the rest of the list) followed by the first element

A.1.2 The code

```
import java.io.*;
import java.text.*;

// See the last part of note 3
class FileFormatException extends RuntimeException{
    public FileFormatException(String s) {
        super(s);}
}



class Pert {


    // Heart of PERT chart is directed graph
    private int numvertices;   // number of tasks
    private boolean edge[][];  // records precedence reln note 1
```

```
// information for tasks
private String names []; // names of tasks
private float length []; // length of tasks;
private float est [], lft [], slack [];
private int   label [], parent [];



//------------------- Auxiliary routines

//  put & putln are just synonyms for System.out.println
//  -- takes up less space on a line
private void putln(String s) { System.out.println(s); }
private void putln(int s) { System.out.println(s); }
private void put(String s) { System.out.print(s); }
private void put(int s) { System.out.print(s); }




private String setR(int num, int spacing) {
   // returns a string of "spacing" length containing the number
   // num, right justified.
   String snum;
   String spaces = "
   int  pad;

   snum = Integer.toString(num);
   pad = spacing-snum.length();
   if (pad >= 1)
        snum=spaces.substring(0,Math.min(pad,spaces.length()))+snum;
   return snum;
}

private String setR(float num, int spacing) {
   // returns a string of "spacing" length containing the number
   // num, right justified.
   String snum;
   String spaces = "
   int  pad;

   snum = Float.toString(num);
   pad = spacing-snum.length();
   if (pad >= 1)
        snum=spaces.substring(0,Math.min(pad,spaces.length()))+snum;
   return snum;
}
```

```
private String setR(String str, int spacing) {
   // returns a string of "spacing" length containing the string
   // str right justified.
   String spaces = "
   int  pad;
   pad = spacing-str.length();
   if (pad >= 1)
        str=spaces.substring(0,Math.min(pad,spaces.length()))+str;
   return str;
}


private String setL(String str, int spacing) {
   // returns a string of "spacing" length containing the string
   // str left justified.
   String spaces = "
   int  pad;
   pad = spacing-str.length();
   if (pad >= 1)
        str=str+spaces.substring(0,Math.min(pad,spaces.length()));
   return str;
}



//----------------- Constructor code starts here
// NB constructor calls helper methods

private void SetUpDummies(int n) {
    names[0] = "Start";
    length[0] = 0;
    names[n+1] = "End";
    length[n+1] = 0;
    // Now check to see which tasks depend only on the
    // dummy start node, and which tasks have no successors
    boolean nopred, nosucc;
    for (int t=1; t<=n; t++) {
        nopred = true;
        nosucc = true;
        for (int v=1; v<=n; v++) {
            nopred = nopred && !edge[v][t];
            nosucc = nosucc && !edge[t][v];
        }
        if (nopred && nosucc)
           putln("Warning: task "+t+" has no preds or succs");
        edge[0][t] = nopred;
        edge[t][n+1] = nosucc;
    }
}
```

```
private void  GetTasks(SimpleInput data, int n) throws IOException {
    // Read the  tasks in one by one, checking format
    String  line, temp, nametask;
    int     split, split2, taskid;
    float   tasklen;

    for (int task=1; task <= n; task++)  {
        // get next line of input --
        //   contains number followed by string
        // see Note 3
            line = data.readLine();
            if (line == null) {// Should not happen
                temp = "Reached end of file at task  "+ task+
                       "; expecting "+n+ " tasks";
                throw new FileFormatException(temp);
            }
            split = line.indexOf(" ");
            // extract out the numbers and convert to integer
            temp = line.substring(0,split);
            taskid = Integer.parseInt(temp);
            if (taskid != task) { // Should not happen
                temp="Error in input : "+line+"-expected task "+task;
                throw new FileFormatException(temp);
            }
            split2   = line.indexOf(" ", split+1);
            temp     = line.substring(split+1,split2);
            nametask = line.substring(split2+1);
            tasklen  = Float.parseFloat(temp);
            names[task] = nametask;
            length[task] = tasklen;
    }

}


private void GetRelationships(SimpleInput data, int n)
    throws IOException {
    // Now read the relationships between the tasks
    String line, first, second;
    int x, y, split;

    while (true){
              // get next line of input -- containts two ints split by a " "
        line = data.readLine();
        if (line == null) {break;  }  // Exit here
        // find where the split is
        split = line.indexOf(" ");
        // extract out the numbers and convert to integer
        first = line.substring(0,split);
        second= line.substring(split+1);
```

```
            x = Integer.parseInt(first);
            y = Integer.parseInt(second);
            edge[x][y] = true;
        }
    }

    // --- the constructor proper
    public Pert(String fname) throws IOException{
        int n,i,j;

        SimpleInput inp = new SimpleInput(fname);
        // get number of tasks; add 2 for dummy start and end
        // tasks
        n = inp.readInt();
        numvertices = n+2;

        edge   = new boolean [numvertices][numvertices];
        names  = new String[numvertices];
        length = new float [numvertices];
        est    = new float [numvertices];
        lft    = new float [numvertices];
        slack  = new float [numvertices];
        label  = new int   [numvertices];
        parent = new int   [numvertices];

        for (i=0; i<numvertices; i++)
            for(j=0; j<numvertices; j++)
                edge[i][j]=false;
        GetTasks(inp, n);
        GetRelationships(inp,n);
        SetUpDummies(n);

    }



    //---------------------- Critical path routines


    private void ComputeEST() {
        int t, i, w, x;
        boolean new_fringe; // see note 4
        for (t=0; t < numvertices; t++) {
            label[t] = -1;
            est[t]   = 0;
            parent[t]= 0;
        }
        i=0;
        label[0] = i;
        do {
            new_fringe = false;  //see note 4
```

```
        for (w=0; w<numvertices; w++) { // see note 5
            if (label[w] == i) {
                for (x=0; x<numvertices; x++) { // see note 6
                    if (edge[w][x]) {
                        new_fringe = true;
                        label[x] = i+1;
                        if (est[w]+length[w] > est[x]) {
                            est[x] = est[w]+length[w];
                            parent[x] = w;}}
                }
            }
        }
        i  = i+1;
    } while (new_fringe);
}



private void ComputeLFT() {
    int t, i, w, x;
    boolean new_fringe; // see note 7
    for (t=0; t < numvertices; t++) {
        label[t] = -1;
        lft[t]   = est[numvertices-1];
    }
    i=0;
    label[numvertices-1] = i;
    do {
        new_fringe = false;
        // check for all vertices labelled i
        for (w=0; w<numvertices; w++) {
            if (label[w] == i) {
                // find all the neigbours
                for (x=0; x<numvertices; x++) {
                    if (edge[x][w]) {
                        new_fringe = true;
                        label[x] = i+1;
                        if (lft[w]-length[w] < lft[x])
                            lft[x] = lft[w]-length[w];
                    }
                }
            }
        }
        i  = i+1;
    } while (new_fringe);
    for(t=0; t<numvertices; t++)
        slack[t] = lft[t]-est[t]-length[t];

}
```

```
public void ComputeCriticalData() {
    ComputeEST();
    ComputeLFT();
}


//---------------------  Print routines



private void ShowCriticalPath(int last) {
    // see note 8
    if (last != 0)
        ShowCriticalPath(parent[last]);
    put(last+"; ");
}



public void PrintResults() {
    //Prints out the EST, LFT etc...
    int t;

    putln("\n\nSCHEDULE FOR THIS PROJECT\n\n");
    putln("Task#       Label         EST       LFT        Slack   Parent");
    for(t=0; t<numvertices; t++) {
        putln(setR(t,6)+"  "+setR(label[t],11)+
              setR(est[t],11)+setR(lft[t],11)+
              setR(slack[t],13)+setR(parent[t],11));
    }

    //Now show critical path
    putln("\n\nAny task with slack 0 is on a critical path");
    putln("One critical path is");
    ShowCriticalPath(numvertices-1);
    putln("");
}



public void PrintTasks() {
    // prints out the list of tasks, their names, and
    // the tasks that depend on them.
    int t,u;
```

```
        putln("Task #   Task description                Length    Successors");
        for(t=0; t<numvertices; t++) {
            put(setR(t,6)+"  "+setL(names[t],31)+
                setR(length[t],7)+"      ");
            for(u=1; u<numvertices; u++)
               {if (edge[t][u]) put(u+"; ");}
            putln("");
        }
    }


}
```

A.2 Bicomponents code

This code follows the algorithm presented in Figure 6.8 on 91.

- There are some minor differences as we need to take care of 'details' like where variables are declared, how they are passed as parameters etc.
- We also make the code more concrete. So the line in the algorithm that says this:

  foreach $w \ni \exists(v,w) \in E$

  is replaced by:

  ```
  for (w=adjlist[v]; w != null; w = w.next) {
  ```

  since we are using an adjacency list.

  How would we translate the line if we used an adjacency matrix?

The code

```
    private int BDFS(int v, int parent,
                     boolean marked[], int back [], int curr_dfsnum) {
        // Perform DFS from vertex v
        // returns the back
        EdgeList w;
        int this_back;

        marked[v]=true;
        dfsnum[v]=curr_dfsnum;
        this_back = curr_dfsnum;
```

```
        for (w=adjlist[v]; w != null; w=w.next) {
            if (!marked[w.y]) {
                curr_dfsnum = BDFS(w.y, v, marked, back, curr_dfsnum+1);
                if (back[w.y] >= dfsnum[v])
                    put("V"+v+" is an articulation point");
                else
                    this_back = Math.min(this_back,back[w.y]);;
            }
            if (w.y != parent)
                {this_back = Math.min(this_back,back[w.y]);};
        }
        back[v] = this_back;
        return curr_dfsnum;
    }


    public void  BicomponentDFS() {
        boolean marked [] = new boolean [numvertices];
        int     back   [] = new  int [numvertices];
        int curr_dfsnum, i;
        EdgeList w;
        boolean morechildren=false;

        puts("\n\n Finding bicomponents....\n\n");
        for (i=0; i<numvertices; i++)  marked[i]=false;
        marked[0] = true;
        curr_dfsnum = 0;

        for (w=adjlist[0]; w != null; w=w.next) {
            if (!marked[w.y]) {
                curr_dfsnum = BDFS(w.y, -1, marked, back, curr_dfsnum+1);
                if (morechildren) {
                    put("Root is an articulation point");};
                morechildren = true;
            }

        }
        PrintBiComponents(back, marked);
    }



    public void PrintBiComponents(int back [], boolean marked[]) {
        int v, w;
        put(" Vertex     dfsnum    back");
        put("----------------------------");
        for(v=0; v < numvertices; v++) {
            if (marked[v]) {
               puts(set(v,8)+"  "+set(dfsnum[v],8)+"   "+
                    set(back[v],6)+" |   ");
```

```
            }
            put("");
        }
    }
```

A.3 Strong components

This code follows the algorithm presented in Figure 6.9 on 94.

- There are some minor differences as we need to take care of 'details' like where variables are declared, how they are passed as parameters etc.
- We also make the code more concrete. So the line in the algorithm that says this:

  foreach $w \ni \exists(v,w) \in E$

  is replaced by:

  ```
  for (w=adjlist[v]; w != null; w = w.next) {
  ```

  since we are using an adjacency list.

  How would we translate the line if we used an adjacency matrix?

- We also consider how the stack is implemented. Naturally, we just use the standard Java stack package, which works well. But in addition to the normal operations like push and pop, we also need to test whether an element is on the stack. Now, this isn't a standard stack operation, but it is implemented in the Java stack class (as a method called *search*). However, this would take linear time. Instead we buy time for memory and introduce an auxiliary boolean array called `onstack` which records whether a vertex is currently on the stack. Not only do we need to pay memory for this, but we also have to update the array correctly. The cost is worth it, however, since we can check for membership of the stack in constant time.

The code

```
public void PrintLowDfs() {
    int v, w;
    put(" Vertex     dfsnum    low");
    put("----------------------------");
    for(v=0; v < numvertices; v++)
           put(set(v,8)+"  "+set(dfsnum[v],8)+"   "+
                set(low[v],6)+" |   ");
```

```
        put("");
    }



    void StrSrch(int v, IntegerStack vstack,
                 boolean marked [], boolean onstack []){
        EdgeList w;
        int x;

        marked[v] = true;
        dfsnum[v] = curr_dfs;
        curr_dfs++;
        low[v] = dfsnum[v];
        vstack.push(v);
        onstack[v] = true;
        for (w=adjlist[v]; w != null; w = w.next) {
            if (!marked[w.y]) {
                StrSrch(w.y, vstack, marked, onstack);
                low[v] = Math.min(low[v], low[w.y]);
            } else
                if (onstack[w.y]) {
                    low[v] = Math.min(dfsnum[w.y], low[v]);
                }
        }
        if(low[v] == dfsnum[v]) {
            put("Found strong component");
            while (!vstack.empty() && dfsnum[vstack.top()]>=dfsnum[v]) {
                x = vstack.popi();
                onstack[x]=false;
                puts(x+";");
            }
            puts("\n\n");
        }
    }

    void StrongSearch() {
        int i;
        boolean  marked  [] = new boolean [numvertices];
        boolean  onstack [] = new boolean [numvertices];
        IntegerStack  vstack = new IntegerStack() ;

        curr_dfs= 0;
        for (i=0; i<numvertices; i++) {
            onstack[i] = false;
            marked[i]  = false;
        }
        for (i=0; i<numvertices; i++)
            if(!marked[i])
                StrSrch(i, vstack, marked, onstack);
        PrintLowDfs();
```

```
}
```

Bibliography

[1] S. Baase. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley, 1988.

[2] R.E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[3] V. Chachra, P.M. Ghare, and J.M. Moore. *Applications of Graph Theory Algorithms*. Elsevier North Holland, New York, 1979.

[4] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, Massachusetts, 1990.

[5] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, 1979.

[6] R. Gould. *Graph Theory*. Benjamin/Cummings, Menlo Park, California, 1988.

[7] D. Gusfield and R.W. Irving. *The Stable Marriage Problem*. MIT Press, Cambridge, MA, 1989.

[8] D. Harel. *Algorithmics: The spirit of computing*. Addison-Wesley, Wokingham, 1987.

[9] J.A. McHugh. *Algorithmic Graph Theory*. Prentice-Hall, 1988.

[10] R. Wilson. *Introduction to Graph Theory*. Harlow, 1985.

Index