MAT2402: Networks
Vers 3.30

Josef Lauri©
University of Malta

Contents

1 Set books

1. Dossey, Otto, Spence and Vanden Eynden *Discrete Mathematics. 3rd Edition*

2. Dolan and Aldous *Networks and Algorithms*. These two books are the closest to the course.

3. Biggs *Discrete Mathematics*. Some parts are a very close match to the course.

4. Wilson *Introduction to Graph Theory*. This or any other introduction to graph theory. Not really necessary because the others contain their own introduction to GT.

5. W. L. Winston *Operations Research: Applications & Algorithms*. Not essential to the course, but a good introduction to OR.

6. Cormen, Leiserson and Rivest *Algorithms*. Not essential to the course, but parts of it are excellent presentations of graph theory applied to data structures and OR (the latter being what we shall mostly be covering).

2 Introduction

2.1 Combinatorial optimisation

This course will deal with optimisation problems. By optimisation we do not mean the classical type involving calculus, but what is sometimes referred to as 'Combinatorial Optimisation'. In fact, we shall only be touching briefly on one aspect of Combinatorial Optimisation, namely, that aspect which involved graphs (networks).

Perhaps the best way to explain what we shall be doing is by giving a few examples.

- Connecting sites with cables using the shortest possible length of cable.
- Finding the shortest route between any two cities given distances between any two pairs.
- Also, problems which do not obviously involve networks, such as, given a list of jobs and a list of candidates to match candidates to the jobs they do best.
- Or, given the subtasks involved in a project, with their durations and which subtask should precede others, to find the minimum time for finishing the project.

In theory, in all these problems we can try all possible solutions (since there is a finite number of them) and choose the best one (shortest, etc). But this way, the number of cases to deal with will be astronomically high (see Table 1). Algorithms with complexity (number of operations) that are not polynomial, but are exponential or factorial are considered as 'bad' algorithms. If the complexity is polynomial then the algorithm is considered to be 'good' or 'efficient'.

Operations	Size of problem		
	$n = 20$	$n = 40$	$n = 60$
n	$2 \times 10^{-5} sec$	$4 \times 10^{-5} sec$	$6 \times 10^{-5} sec$
n^2	$4 \times 10^{-4} sec$	$1.6 \times 10^{-3} sec$	$3.6 \times 10^{-2} sec$
n^3	$8 \times 10^{-3} sec$	$6.4 \times 10^{-2} sec$	$2.2 \times 10^{-1} sec$
2^n	$1.0 sec$	$12.7 days$	$366 centuries$
$n!$		WORSE!!	

Table 1: Time taken at rate of 10^6 operations per second

In this course we shall be considering the worst case complexity of algorithms (the average case complexity might be more useful, but it is outside the scope of this first course in combinatorial optimisation).

2.2 Big O notation

To help us analyse the complexities of algorithms we employ a notation which emphasises the "leading" term of a function—the philosophy being that we are interested in problems where the input size n is large and in this case the leading term is the most important.

For example, let $f(n) = 3n^3 + 2n - 5$, $g(n) = 7n^3 - n$ and $h(n) = 23n^2 + 99n$. Then f and g are considered to be in the same 'family' as n^3 but h is in the 'family' of n^2, although, for small n, f and g could be smaller than h.

This idea can be made more exact by the big O notation. Thus, if f and g are two function we write $f(n) = Og(n)$ to mean that there exists a constant K such that, for all sufficiently large n, $|F(n)| \leq K|g(n)|$.

Thus, in the above examples $f(n) = O(n^3)$, $g(n) = O(n^3)$ and $h(n) = O(n^2)$.

Our aim would be to find algorithms whose worst case complexity is O(polynomial), that is, 'good' algorithms. The troublesome problems will be those for which the only algorithms known till now have complexity like $O(c^n)$ or $O(n!)$.

More can be said about all this which is essentially asymptotic analysis of functions, but the explanantion given in Gibbons or Biggs is quite sufficient for our purpose.

2.3 Mathematical techniques required

For this course you will require very little or no calculus. You will require some combinatorial sense, knowledge of some simple enumeration and even simpler probability. Knowledge of some graph theory, which we shall cover below. And the ability to describe algorithms in a pseudo-Pascal language which we shall cover in the next section.

The task facing someone who is trying to solve an optimisation problem involves three main components:

1. Finding an algorithm and *proving* that it works, that is, that it does find the optimal solution one is looking for.

2. Finding its time-complexity.

3. Implementing the algorithm as a computer language.

We shall mostly be concerned with (1) and (2) where possible, but we shall also have to go a little way along (3) in order to be able to carry out (1) and (2).

In the next section we shall describe the pseudo-Pascal language by means of which we shall be describing our algorithms.

2.4 A pseudo-Pascal language

As in Gibbons or Biggs we shall describe our algorithms using these constructs. To make the nesting and scope of these constructs more clear we number each line and a nested numbering will correspond to a nested construct (this will be clearer when we see actual examples).

```
 1. for i:= 1 to n do
        1.1 . . .
        1.2 . . .
        1.3 . . .
            . . .


1. while (condition) do
        1.1 . . .
        1.2 . . .
        1.3 . . .
            . . .


1. if (condition) then
        1.1 . . .
        1.2 . . .
        1.3 . . .
            . . .
2. else
        2.1 . . .
        2.2 . . .
        2.3 . . .
            . . .
```

'Conditions' are Boolean conditions (value = Yes or No), usually the result of a simple mathematical operation, for example:

```
  if (i<3) do
    . . .
```

The dots are often more of the constructs nested within each other or, ultimately, instructions written in plain English; we shall not attempt to refine the algorithm to the point where it is actual computer code but we would need to refine the description to the stage where the proof of correctness of the algorithm and the time-complexity analysis (how many loops? etc) can be carried out.

2.5 Some basic graph theory

We shall here give only a brief treatment of the definitions from graph theory required for this course. More details are given in the recommended text-books. A *graph* G consists of two finite sets $(V(G), E(G))$ where $E(G)$ consists of unordered pairs of elements of $V(G)$—the elements of $V(G)$ are called *vertices* while the elements of $E(G)$ are called *edges*. In a *directed graph*, or *digraph*, for short, the pairs are ordered pairs—we call these *arcs*. We shall sometimes also meet mixed graphs, containing both edges and arcs.

Graphs or digraphs are usually represented by a diagram where the vertices are denoted by dots and the edges are denoted by lines (with an appropriate arrowhead if it is an arc).

Loops (edges joining a vertex to itself) will not be permitted but *multiple* (sometimes called 'parallel') edges or arcs (joining the same pair of vertices more than once) can occur.

The *degree* $\deg(u)$ of a vertex u is the number of edges incident to u. In digraphs, the *in-degree* $\deg_{\text{in}}(u)$ and the *out-degree* $\deg_{\text{out}}(u)$ of the vertex u are, respectively, the number of arcs incident from and the number of arcs incident to u.

Usually, n will denote the number of vertices of a graph while m will denote the number of edges or arcs.

We can now see that even with the very rudimentary definitions we have listed till now, we can still state and prove a theorem. As we said earlier, in order to find the proof all that is required is a bit of combinatorial sense.

Theorem 2.1 *Let G be a graph. Then*

$$\sum_{u \in V(G)} \deg(u) = 2|E(G)|.$$

Proof Exercise.

Corollary 2.2 *A graph cannot have an odd number of vertices with odd degree*

Proof Exercise.

For example, there can be no graph with degrees 2,2,3,3,5,6,7,7,8.

A *path* is a sequence of distinct edges, $e_1, e_2, \ldots, e_k$, such that if $e_i = \{u_i, u_{i+1}\}$ then $e_{i+1} = \{u_{i+1}, u_{i+2}\}$. The *length* of such a path is k and we say that u_1 and u_k are *joined* by the path.

A *directed path* or *dipath* is similarly defined but the edges are now arcs (u_i, u_{i+1}).

The *underlying graph* of a digraph is obtained by 'forgetting' directions, that is, changing all arcs into edges.

A graph is *connected* if any two vertices are joined by at least one path. If a graph is *disconnected* then the maximal connected subgraphs are called components

A *cycle* is a path but where we allow $u_1 = u_k$. Similarly for *directed cycles* or *dicycles*.

2.6 Special types of graphs

The *complete graph* K_n which consists of n vertices and all possible edges between them.

Easy exercise: K_n has $n(n-1)/2$ edges,

A *bipartite graph* is a graph such that its vertex set can be partitioned into V_1 and V_2 and such that any edge joins a vertex from V_1 to a vertex from V_2. Such graphs are very useful to model, say, situations such as the jobs vs. candidates one we mentioned above.

Easy exercise: If a bipartite graph contains all possible edges then the number of edges is $|V_1| \cdot |V_2|$. Such a graph is denotes by $K_{p,q}$ where p and q are the sizes of V_1 and V_2, respectively.

An *acyclic* digraph is one which has no directed cycles. Such graphs are important in situations where there is a hierarchy, for example, a list of tasks and rules that determine which tasks should precede any given task.

A *tree* is a connected graph which has no cycles. This is the single most important type of graph, appears when modelling data structures, as a basis for sorting or structuring databases, etc. In this course which is about finding 'optimal' types of graphs trees are very important because they represent the most 'economical' way of connecting up the vertices of a graph. We shall have more to say about this in later sections.

2.7 Ways of representing a graph

Representing a graph as a drawing with dots and lines is fine for small graphs. But for large graphs, or, if we want to store a graph in a computer, we need some more formal ways of representing a graph.

The *adjacency matrix* $A = A(G)$ of a graph with n vertices labelled $v_1, v_2, \ldots, v_n$ is an n by n matrix such $A_{ij} = 1$ if v_i and v_j are adjacent, and 0 otherwise.

If a graph has n vertices as above and m edges $b_1, b_2, \ldots, b_m$ then the *incidence matrix* $B = B(G)$ of G is an m by n matrix such that $B_{ij} = 1$ if edge b_i is incident to vertex v_j and 0 otherwise.

The *adjacency list* of a graph is made up of two columns: the first column contains the vertices $v_1, v_2, \ldots$ in order, while in the ith line of the second column are listed the neighbours of v_i.

As a taste of what is involved in working with a graph stored as a data structure in a computer, try writing a programme to determine if a graph (given in any of the above ways) is connected, or has cycles, or has a path of a certain length.

2.8 Networks

Although the course is called "Networks" we shall not here define the term. This is because the meaning of 'network' will change with the situation we are modelling. Basically, a network will be a graph/digraph with labels on the vertices or edges and the instruction to 'optimise' something pertaining to this structure. The labels could be costs, distances, capacities for carrying flows, etc. We shall therefore be meeting activity networks, road networks, flow networks, etc. These will be defined as we proceed.

3 Sheet 1: Elementary Graph Theory

1. Show that the vertices of a graph without loops or multiple edges cannot have vertices all with distinct degrees. [Use the pigeonhole principle.]

2. * Let the maximum degree, minimum degree, number of vertices and number of edges of a graph be Δ, δ, n, m, respectively. Show that

$$\delta \leq \frac{2m}{n} \leq \Delta.$$

3. * A graph all of whose vertices have the same degree r is said to be *regular* or *r-regular*. How many edges does an r-regular graph on n vertices have?

   How many edges does the complete graph K_n have? [Do this in two different ways. The answer gives the maximum number of edges which a graph on n vertices without loops or multiple edges can have.]

4. Let T be a tree such that all vertices which are not endvertices have degree 3. Let i be the number of such (internal) vertices and let l be the number of endvertices (leaves). Show that

$$l = i + 2.$$

5. Suppose an algorithm consists of n stages and that the ith stage requires i operation. Show that the time-complexity of the algorithm is $O(n^3)$.

6. * Let F be a forest on n vertices and k edges. Show that the forest contains exactly $n - k$ components (which are, therefore, trees).

4 Complex networks: search engines

One of the types of graph whose importance in applications has literally seen an explosion in the last few years is that of complex networks. In this context, by a complex network we mean a network with a very large number of vertices and edges, and which has been built up in a random, or at least not completely deterministic, fashion. The possible random processes which can be used to model the growth of a random graph are several, and this topic is an area of very active current research. We shall not go into this here, but we note two things. Firstly, that we are considering networks which have grown without some central agency specifying an overall, global direction for the process. It is as if the network has grown (or is still growing) on its own, and one of the intriguing aspects of the study of such networks is that the local probabilistic rules for growth (meaning generally, those occurring at individual vertices) give a structure with some clearly defined global properties, such as the overall degree distribution and short distance between most pairs of vertices, in spite of the large size of the network (the "small world phenomenon").

Secondly, one must point out that we are surrounded with an abundance of such networks, the WWW being the most obvious example (vertices are pages and arcs are the links between pages). But there are several other examples which are the objects of active research: networks describing protein-protein interaction, modelling of food webs, modelling the spread of viruses, neural networks (the real ones inside our skulls) as well as artificial ones, networks of

metabolic reactions, power grids, the Word Web of human language, and models of the vascular structure of body organs, to mention only a few.

The WWW is probably the complex network which has, most of all, spurred on the development of this discipline. This is mainly because, by its very nature, it is relatively easier to devise methods to study it using fast electronic look-up and storage techniques. It might seem paradoxical at first sight, but this subject is fast becoming a branch of physics (statistical mechanics, in particular) and some prominent physicists have described such networks as "one of the few fundamental objects of the Universe." [1]

In this short section we shall only have time to study one little aspect of such networks—in fact, one little aspect of how search engines over the WWW rank the pages retrieved. There are many factors which we will not talk about: the design of web crawlers (or spiders, or robots) and retrieval of information by the crawlers; the semantic lookup of the retrieved web-pages, that is, building a table which gives, for each web-page, the number of times which particular keywords appear in each (there is also some interesting linear algebra at work here); compression of this data in a way which allows quick look-up while at the same time ensuring that storage requirements scale well with the growth of the Web. I shall not expect you to study these issues in any detail, but I would expect you to read the two papers I shall suggest below where these issues are, at least, mentioned—just skip over the hard mathematical parts, but try to understand the issues.

In this section we shall consider a search engine would rank those pages which satisfy a particular query, that is, those pages which contain the keywords requested. It is here that the network structure of the Web is exploited. Until a few years ago, the order of pages presented in response to a query depended primarily on how many times the query string appeared in the relative pages. But it was becoming apparent that this was not giving good results. For example, in answer to the query string "search engine", few search engines listed themselves amongst the top ten pages, because few search engines actually used the term "search engine" often enough on their web-sites. More examples of this problem can be found in the papers listed below.

But first we shall consider a seemingly simpler and more familiar situation.

4.1 Ranking participants in a tournament

How do we rank players or teams in a tournament? Generally, we give each participant a certain number of points for each game won and a certain number for each draw, add the total for each player, and then rank the players in descending order of points gained. This is certainly a mathematical way of ranking participants, although a very simple mathematical method. Is it completely satisfactory? What about ties? And what about situations over which sports followers often argue? For example, a team could win a tournament by drawing against the top teams and consistently beat the weaker ones. The runner-up could have a better record amongst the top teams but a less consistent performance against the weaker ones. The question of which is really the best team of the tournament could give rise to endless discussions amongst sports commentators. Could mathematics give alternate ways of ranking the teams?

[1] S.N. Dorogovtsev & J.F.F. Mendes, *Evolution of Networks: From Biological Nets to the Internet and WWW*. Oxf. U. Press

Let us consider the following example. Five players, 1, 2, 3, 4, 5, play a tennis tournament. The results of the tournament are given below, where an entry in row i and column j means that player i has beaten player j.

$$A = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}.$$

In terms of the scores of the players (that is, the number of games won), the ranking of the players is player 2 in first place, with a score of 3, players 1, 3 and 4 joint second to fourth with 2 points, and player 5 in fifth place with a score of 1. However, one notes that the only win obtained by player 5 is against the top player who has beaten all the other players in the tournament. Should this win earn more than just one point?

One way to go about treating this question is to award each player p the sum of the scores obtained by the players which p has beaten. Let us call this score the SOS of each player, short for Sum of Opponents' Scores. If we calculate the SOS of each player in our example we get end up with player 2 having an SOS of 6, and all the other players getting an SOS of 3. So player 5 has moved up to joint second to fifth! But if the SOS is really a better indication of the relative strengths of the players than the raw scores, should we not try the sum of opponents' SOS? This gives scores of 9 for the top player and 6 for each of the others. But we can do one better, and calculate the sum of the sum of the SOS's and iterate this on and on. There are two problems with this method.

One is mathematical. These iteration do not give results which settles down, that is, converges to some limit. The scores keep increasing and the relative positions of the players might change from iteration to iteration. We shall see in the following sections that this problem can often be solved to give a result to which iterations of the type discussed do converge.

The second problem is a subjective one. Even if the iterations do converge, will the result be acceptable to, say, sports enthusiasts. It seems very difficult to imagine any sport in which player 5 in the tournament above would be ranked equal to or better than any of the other players. But in one area—ranking of web pages by importance—there has been such a general consensus that these methods give the best results that in a few years Google became the leading search engine thanks to PageRank which uses these methods. So we shall now return to the question of ranking web pages in order of "importance".

4.2 Hubs & authorities: Kleinberg's Algorithm

Kleinberg suggested a method to improve the ranking of retrieved sites by exploiting how they are linked to other sites, that is, by exploiting the network structure of the web.

His idea was the following. A site is probably a useful site if it is pointed to by many other sites—in graph theory terms, if it has high in-degree. This would mean that the site is a "popular site"—popular sites are called *authorities*. But this is only half the story. Some sites are important because they point to many sites—that is, they have high out-degree. Such sites are called *hubs*. A popular

site is considered to be an authority not just in view of its popularity (the in-degree) but, more importantly, if many of these arcs pointing to it come from good hubs. And a hub is considered to be a good hub not merely because of its high out-degree but mainly if these arcs point to good authorities. So the definitions of good hubs and good authorities are circular: they depend on each other. How are we to measure how good a hub or authority a vertex is?

Kleinberg's algorithm does this iteratively. We shall describe it without going into many details. One first gives an initial hub value and authority value to each vertex of the sub-network of the Web with which one is working (usually this sub-network consists of those pages satisfying the query, together with their links). These initial values could be out-degrees and in-degrees, possibly scaled by some constant value. Then the hub values are updated by increasing the value of those hubs which point to high-ranking authorities, and decreasing the value of those which do not. Based on these new hub values, the authority values are updated by increasing the values of those vertices pointed to by high-ranking hubs and decreasing the others. This process is repeated either up to a pre-determined number of iterates, or until the top hub and authority values stabilise. Then the search engine lists the retrieved pages twice: in decreasing order of authority value and decreasing order of hub value.

4.3 A modification: The method of Brin & Page

Kleinberg's method, in simulation studies, immediately showed a marked improvement on previous methods used by search-engines to rank the pages retrieved. Brin and Page, who founded Google while they were graduate students at Stanford University, suggested variations which modified Kleinberg's method. We shall consider in some detail an even more simplified version of their algorithm, and we shall show that, surprisingly, it leads to important results from linear algebra.

So, suppose for simplicity that the network we are considering is a graph (that is, it has undirected edges), and suppose that we shall give vertices only one label; let us call it the "authority" label and let us denote it by $a(v)$ for any vertex v. Now, we want to assign these labels such that a vertex adjacent to vertices with high authority will inherit high authority itself. Again, we have a seemingly circular definition: the authority of a vertex v depends on that of its neighbours, but the authority of each of its neighbours also depends on that of v. How can we find these authority values algorithmically?

The answer is a simplified version of the iterative algorithm of Kleinberg. We start by giving all vertices an initial label, say equal to the degree. Then, we go through the list of vertices in some arbitrary but specified order, and we update the label $a(v)$ by considering, the sum of the labels $a(w)$ for all w neighbours of v (recall the sum of opponents' scores!). And we repeat this either up to a pre-determined number of iterates or until the labels converge to steady values. Of course, for these values to converge we must multiply the sum of authority values by some scaling factor, otherwise the numbers just keep increasing without limit. What we are ultimately looking for is the following. Let the vertices be $v_1, v_2, \ldots, v_n$. Then we require labels $a(v_i)$ such that the

label of any vertex v_i is given by

$$\lambda \cdot a(v_i) = \sum_{v_j \text{ n'bours of } v_i} a(v_j) \tag{1}$$

where λ is some constant scaling factor. That is, *the authority of a vertex is the (scaled) sum of the authorities of its neighbours.*

If, for short, we write a_i for $a(v_i)$ then this equation can be written as

$$\lambda \cdot a_i = \sum_{v_j \text{ n'bours of } v_i} a_j \tag{2}$$

Now let us look in more detail at Equations 1 and 2. We shall do this by considering the graph in Figure 1. (This graph, with eight vertices, is certainly not a complex network, but the numerical results which we shall obtain will illustrate the method well.) Let us write Equation 2 for vertex v_1

$$\lambda a_1 = (a_2 + a_3)$$

and for vertex v_2

$$\lambda a_2 = (a_1 + a_3 + a_4 + a_5)$$

and so on for the other vertices.

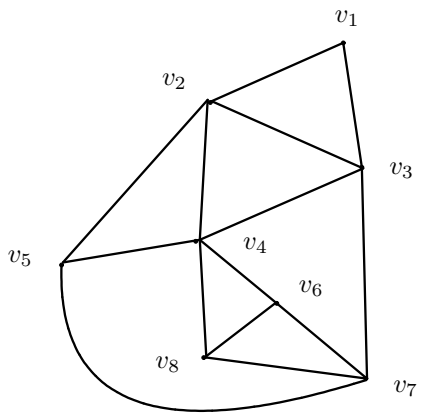


Figure 1: Determining the authority value of vertices

Now, there is a very elegant algebraic way of writing these equations using the adjacency matrix of the graph which, in this case, is

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix}.$$

Let us call this matrix A. Let x be the column vector which is the transpose of the vector

$$[a_1, a_2, \ldots, a_8],$$

that is, x is the column vector listing the authority values.

Then Equation 2 for all the vertices v_1 to v_8 can be written as one matrix equation

$$\lambda x = Ax \tag{3}$$

Therefore given A (that is, the network) we are required to find λ and x which satisfy Equation 3.

Now such vectors and scalars for a given matrix are very important and well known in mathematics. The vector x is called an *eigenvector* of A and the scaling factor λ is the corresponding *eigenvalue*. Moreover, A is a matrix with no negative entries and which represents a connected graph. Now, there is a little advanced linear algebra which says that for such matrices there is exactly one eigenvector with all its entries not negative—this eigenvector (or any of its multiples) is called the *principal eigenvector* or the *Perron eigenvector* of A. The method of Brin and Page (which is at the heart of PageRank, Google's ranking algorithm) finds in fact the principal eigenvector of the adjacency matrix A.

Continuing with our example, this eigenvector can be found with, say, *Mathematica*, to be the transpose of the vector

$$(0.746493, 1.34326, 1.30695, 1.61874, 1.09669, 1, 0.931478, 1),$$

and these are the authority labels for the vertices v_1 to v_8, respectively. This means that, ranking the vertices of the network in decreasing order of authority gives: v_4, v_2, v_3, followed jointly by v_6 and v_8 and then v_7 and v_1.

Does this make sense if we analyse the network in Figure 1 "by sight"? (This is something we can do here since this is not really a complex network, and in fact it has been constructed in order to make this visual analysis feasible.) It seems clear that v_4 should be the most important vertex (according to the way we are defining "importance") because it has the highest degree. The next candidates, because of their degree, would then be v_2 and v_3. Can we break the tie between them? What about the importance of their neighbours from which they inherit authority? This factor might even override mere degree considerations. Note that v_2 and v_3 are both adjacent to each other and to v_1 and v_4 so, as far as these vertices go, it makes no difference to the authority which v_2 and v_3 inherit. But v_2 is also adjacent to v_5 which is adjacent to the most important vertex v_4, whereas v_3 is adjacent to v_7 which is joined to v_4 via the intermediary v_6. This suggests that v_2 should have a higher authority value than v_3, as in fact the principal eigenvector tells us. One can continue to analyse the small network of Figure 1 in this way, but we have already seen enough to indicate that the result given by the principal eigenvector agrees well with our intuition in this small example.

4.4 Directed graphs

Let us now look at the possibility of directed edges. This is a more realistic view since it corresponds to how links really are on the WWW. This not only brings

the discussion closer to how a ranking algorithm actually works, but it is also a good way of introducing a type of matrix which is very important, especially in applications.

The sub-network of the WWW which we want to rank is now represented by a directed graph whose arcs correspond to the links connected the pages. Therefore, there is an arc from vertex a to vertex b if there is a link from the page corresponding to a to the page corresponding to b. The idea here is similar to what we have seen above. But now a page (vertex) is considered to be more important if it has several arcs incident *to* it—this corresponds to saying that a web-page is important or has high authority value if there are several other pages pointing to it.

But not all links are of equal importance. A link to vertex a coming from an important vertex contributes to a's importance more than a link coming from a less important vertex. Again we have this seemingly circular definition of importance. Consider the simple digraph shown in Figure 2.

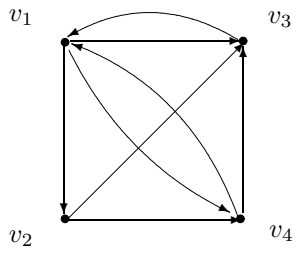


Figure 2: A small directed network of web pages

The authority value of v_1, for example, would be given by

$$a(v_1) = a(v_3) + a(v_4)$$

while the authority value of v_4 will be given by

$$a(v_4) = a(v_1) + a(v_2).$$

How are we going to resolve this apparent circularity.

We let A be the matrix corresponding to the digraph in Figure 1, where

$$A = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix}.$$

Note that now, in A, the i-th row records the incoming arcs into vertex v_i; that is, if the ij entry is 1, then this means that there is an arc directed from vertex v_j to vertex v_i. If again we denote $a(v_i)$ by a_i and we let x be the column vector which is the transpose of the vector

$$[a_1, a_2, a_3, a_4],$$

then the above equations can be written as

$$Ax = x$$

which is again an eigenvalue problem. As the equation stands now, we are seeking an eigenvector of A for the eigenvalue 1—but the matrix A might not have an eigenvalue equal to 1. There is another difficulty: we do not want a vertex to gain importance simply by creating many links. So we divide each column of A by a factor such that the sum of the numbers in that column is 1. This gives the modified matrix

$$S = \begin{bmatrix} 0 & 0 & 1 & \frac{1}{2} \\ \frac{1}{3} & 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{2} & 0 & \frac{1}{2} \\ \frac{1}{3} & \frac{1}{2} & 0 & 0 \end{bmatrix}$$

and the eigenvalue problem now becomes

$$Sx = x.$$

It turns out that the matrix S does in fact have an eigenvalue 1, and the corresponding eigenvector x (that is, the solution to $Sx = x$) is

$$(12, 4, 9, 5).$$

Therefore the vertices of Figure 2 are ranked, in decreasing order of authority, as

$$v_1, v_3, v_4, v_2.$$

Note that this order is not the same as the order corresponding to the number of incoming arcs into each vertex. (Can you explain, in terms of the digraph, why this happens?)

The matrix S with all column entries summing to 1 is called a *column stochastic matrix*. Stochastic matrices form an extremely important class of matrices, and you might very well meet them in some other courses. [2] It turns out that if a matrix S is a stochastic matrix corresponding to a digraph without *sinks* (a sink is a vertex without outgoing arcs, that is, with out-degree equal to zero) then S always has 1 as an eigenvalue. Therefore the above method will always work, that is, we can always find the eigenvector corresponding to the eigenvalue 1 and use it to rank the vertices.

4.5 Computational issues

So Google calculates eigenvectors when ranking pages in response to a query! But how are the eigenvectors actually computed? For a 2×2 or 3×3 matrix one can solve what is called the *characteristic polynomial* of the matrix, which in these cases would be a quadratic or a cubic equation, respectively. For larger matrices one can use packages like *Mathematica* as we did above. But what should be done with massively large matrices which are typically the ones

[2] For example, in operations research courses, or in courses on stochastic processes, Markov chains, simulation, Monte Carlo methods, etc.

encountered by search engines? In this case one has to use more efficient computational methods coming from numerical analysis. Some of the best methods for finding eigenvectors are iterative. That is, one performs simple algebraic manipulations starting from the matrix and some appropriate initial vector until the result converges to the principal eigenvector.

And these iterative methods actually correspond to the ones which we described above when discussing Kleinberg's algorithm (and even sum of opponents' scores) and before introducing the concept of eigenvectors.

So, although a ranking algorithm might be computing eigenvectors, it does not use the same algebraic manipulations that we might employ on small examples. It is essentially programmed to carry out the iterative methods of Kleinberg or Brin & Page which we briefly described above.

Moreover, the structure of the network on which the search is being carried out might be such that the adjacency matrix does not have an eigenvector of the type discussed above, so the iterative methods might not converge. This requires modifications to the network before the iterations start.

Several other modifications have been employed to make the process converge faster, and what we have presented here is a very idealised picture of the situation. Suffice it to say that several papers and at least one book have been written about Google's PageRank and related methods. The suggested readings below can give you a better understanding of the situation.

4.6 Background reading

This discussion is necessarily an over-simplification of the way programmes like PageRank work. To learn a little bit more, here are the two papers (and their availability) which I mentioned earlier. You are expected to read them.

J.M. Kleinberg. Authoritative sources in a hyperlinked environment.
http://www.cs.cornell.edu/home/kleinber/auth.pdf

S. Brin & L. Page. The anatomy of a large-scale hypertextual web search engine.
http://www-db.stanford.edu/~backrub/google.html

The following papers provide more technical information about the workings of PageRank.

C. Moler. The World's latgest matrix computation.
http://www.mathworks.com/company/newsletters/news_notes/clevescorner/oct02_cleve.html

I. Rogers. The Google PageRank algorithm and how it works.
http://www.iprcom.com/papers/pagerank/

5 Sheet 0: Ranking of vertices in digraphs

1. * Five players, 1, 2, 3, 4, 5, play a tennis tournament. The results of the tournament are given below, where an entry in row i and column j means that player i has beaten player j.

$$A = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}.$$

What is the ranking of the players in terms of the number of games each has won? Devise an eigenvector method for ranking the players (hint: obtain a column stochastic matrix corresponding to the matrix A), and find the ranking of the players using this method. (Use an appropriate software package to find the eigenvector.)

6 Trees

As we said, trees are a very important type of graph, and they merit a section all on their own. We recap the definition.

A *tree* is a connected graph without cycles.

Lemma 6.1 *A tree must have at least one vertex of degree 1.*

Proof Exercise.

Note. Vertices of degree one are called endvertices or leaves. It can be shown that, in fact, a tree must have at least two endvertices, the minimum attained when the tree is a path.

Theorem 6.1 *A tree on n vertices has $n-1$ edges.*

Proof By induction on n. Exercise.

The next theorem provides us with six different ways of defining a tree. You are not expected to learn the proof, but the statements should convince you that a tree is a connected graph without 'redundancies'.

Theorem 6.2 *The following statements are equivalent.*

1. *T is a connected graph without cycles.*
2. *T is connected and if n, m are, respectively the number of vertices and edges, then $m = n - 1$.*
3. *T has no cycles and $m = n - 1$.*
4. *T is connected but removing any edge gives a disconneceted graph.*
5. *T has no cycles but adding any new edge creates one cycle.*
6. *Any pair of vertices in T are joined by one and only one path.*

One more definition: A disconnected graph all of whose components are trees is called a *forest*.

6.1 Finding a minimum weight spanning tree MWST

Let G be a graph. A *spanning tree* of G is a subgraph of G which includes all vertices. (Note that G contains a spanning tree if and only if it is connected.)

Suppose that the edges of G are labelled with numbers called costs, or weights or distances. The weight of any subgraph of G is the sum of the weights of its edges. Our problem is to find a spanning tree of G which has minimum weight.

6.1.1 Kruskal's Algorithm

Suppose a graph is given as a list of edges with corresponding weights. Table 2 gives an example of such a representation and a figure showing the graph is given in Figure 3. The numberings in the first row give the order in which the edges would appear if they were sorted in increasing weight.

2	4	1	5	3	5
a	a	b	b	c	b
b	d	c	d	d	e
2	3	1	7	2	8

Table 2: Edges with weights

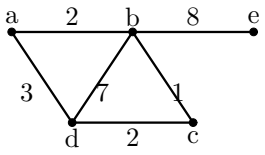


Figure 3: The graph with weights on edges

Kruskal's algorithm is quite simple to describe. It is called a greedy algorithm because at every stage it simply takes the shortest edge available, as long as no cycle is created.

```
KRUSKAL'S ALGORITHM
     1.  Sort the edges e_1,...,e_m in order of increasing weight;
     2.  T = ∅;
     3.  i := 1;
     4.  While |T| < n − 1 do
               4.1 if T ∪ e_i contains no cycles
                      4.1.1 then T := T ∪ e_i;
               4.2 i=i+1
```

The tree (more likely a forest, while it is being built) can be stored in a $1 \times (n-1)$ array, where the j entry will contain i if e_i is the jth edge chosen. Thus, in the above example, the final array would have the entries 1,2,3,6 in that order.

To refine (and analyse) the above algorithm we need to implement the condition "If $T \cup \{e_i\}$ contains no cycles". How do we determine efficiently that a new edge does not create a cycle? We do it this way.

- If the endvertices of e_i are in the same component of T grown so far, then discard e_i.
- If the endvertices of e_i are in different components, then add e_i to T.

To be able to determine whether the new edge is joining vertices in the same or in different components, let the vertices be denoted by $1, 2, \ldots, 6$, and define an array `component`$[i]$, $1 \leq i \leq n$; `component`$[i]$ gives the component number of vertex i during the current stage of the algorithm.

Initially, `component`$[i] := i$ for all i since all the vertices are in different components. Subsequently, if edge ij is added, and, say, `component`$[i] <$ `component`$[j]$, then change the value of all `component` entries equal to `component`$[j]$ into `component`$[i]$. At the end of the algorithm `component`$[i] = 1$ for all i since all vertices would be in the same component. The table below shows how the `component` labels change in the above example.

a	b	c	d	e
1	2	3	4	5
1	2	2	4	5
1	1	1	4	5
1	1	1	1	5
1	1	1	1	1

We can now analyse the worst case time-complexity of Kruskal's algorithm. We do this by counting the number of comparisons.

1. Sorting m edges: this can be done in $O(m \log m)$ comparisons.
2. At worst, the tree is not completed until the mth edge is added.
   (a) For each edge, one comparison to check if its ends are in the same component. Therefore a total of m comparisons.
   (b) For each of the $n-1$ edges actually chosen, must scan all the other vertices and re-label all those whose `component` label is equal to that of the largest endvertex of the edge chosen. This gives a total of $n \times (n-1)$ comparisons.

Therefore the total number of comparisons is at worst

$$O(m \log m + m + n(n-1))$$

and since $m \leq n(n-1)/2$ this is, at worst

$$O(n^2 \log n).$$

Therefore the algorithm has polynomial complexity.

In the next section we shall develop the theory which enables us to prove that Kruskal's algorithm does, in fact, find a MWST.

7 Correctness of Kruskal's Algorithm: Matroids

We shall prove more than the correctness of Kruskal's Algorithm. We shall present a general setting in which the greedy algorithm works. (For a direct proof that Kruskal's Algorithm is correct see Biggs Th. 9.3, pp. 191–192. But note similarities between Biggs' proof and our more general one.)

7.1 Finding a heaviest subset

We shall try and understand why Kruskal's method works in a way which is applicable to other optimisation problems for which the greedy algorithm works. What is the essence of the minimum spanning tree problems? We are faced with the edges of a graph — let us call this set S. Each edge is given a weight. We are interested in a special type of subsets of S, those subsets which form a spanning tree. In particular, we want to find a spanning tree with minimum wights. We can do this by checking the weight of all possible spanning trees and then pick one which has minimum weight, but this would be inefficient, since there are too many spanning trees. So we apply the greedy algorithm. In the greedy algorithm we are only allowed to work with particular subsets of S, precisely those subsets which form subforests of the graph. We shall call this family of subsets $\mathcal{I}$. (Note that for every subset of edges it was easy to determine if it was in $\mathcal{I}$ (that is, if it was a subforest) without having to go through all the subsets in $\mathcal{I}$, and we shall assume that is always the case.)

Now let us generalise this to a situation in which the sets S and its subsets need not be edges of a graph. We shall illustrate with a toy example. Let the set S consist of the elements $\{a_1, \ldots, a_6\}$ with weights as shown in the following table.

Element	a_1	a_2	a_3	a_4	a_5	a_6
Weight	9	9	8	8	7	6

Let the family of subsets $\mathcal{I}$ consist of the following subsets of S:

$$\left\{ \begin{array}{c} \{a_1, a_3, a_6\}, \{a_2, a_4, a_5\}, \\ \{a_1, a_3\}, \{a_1, a_6\}, \{a_3, a_6\}, \\ \{a_2, a_4\}, \{a_2, a_5\}, \{a_4, a_5\}, \\ \{a_1\}, \{a_2\}, \{a_3\}, \{a_4\}, \{a_5\}, \{a_6\}, \\ \emptyset \end{array} \right\}.$$

To make the transition to the theory below easier, we shall assume that we are required to find a *heaviest* subset in I — as we shall see below, this will have no effect on the applicability to Kruskal's algorithm since the minimum spanning tree problem can be easily converted to a maximum spanning tree problem. It is clear in this simple illustration that the heaviest subset in $\mathcal{I}$ is $\{a_2, a_4, a_5\}$, but let us try to find this using the greedy algorithm.

The greedy algorithm in this case goes as follows. Start with T equal to the empty set. Consider the elements $a_1, a_2, \ldots$ in that order (they have already been sorted in *decreasing* order by weight). At each stage, put a_i in T only if the set T would still be in $\mathcal{I}$.

If you carry out this easy routine you will end with the set $\{a_1, a_3, a_6\}$ which is clearly not the heaviest subset in $\mathcal{I}$. Why is this so? Can the same thing happen with Kruskal's algorithm or, in fact, other situations in optimisation where the greedy algorithm is used? Mathematicians have discovered that the reason for the failure of the greedy algorithm in such a case is because the subsets in $\mathcal{I}$ do not satisfy two key conditions:

1. If B is in $\mathcal{I}$ then all subsets of B must be in $\mathcal{I}$.
2. If A and B are in $\mathcal{I}$ and the size of $|A|$ is less than the size of $|B|$ then there must be some element $x \in B - A$ such that x added to A gives a set in $\mathcal{I}$.

If you check carefully you will see that the subsets in $\mathcal{I}$ do satisfy the first condition but not the third because let $A = \{a_1, a_3\}$ and $B = \{a_2, a_4, a_5\}$. Then it is clear that there is no element of B which added to A gives a set in $\mathcal{I}$.

The fact that the above two conditions guarantee that the greedy algorithm works not only proves that Kruskal's algorithm is correct (because the subsets of edges forming subforests do satisfy these conditions, as we shall see below) but it is an important mathematical result in combinatorial optimisation because it applies in most cases where the greedy algorithm is used. We shall therefore proceed to proof this result formally.

7.2 Matroids

A matroid M consists of two things: a finite non-empty set S and a non-empty collection $\mathcal{I}$ of subsets of S such that,

1. *The hereditary property.* If $B \in \mathcal{I}$ and $A \subset B$ then $A \in \mathcal{I}$.
2. *The exchange property.* If $A, B \in \mathcal{I}$ and $|A| < |B|$ then there is some element $x \in B - A$ such that $A \cup \{x\} \in \mathcal{I}$.

The subsets in $\mathcal{I}$ are called the *independent subsets* of S.

The prototype for all matroids is the following (which is the reason why matroids are sometimes called *independent structures.*): Let S be a finite set of vectors from a vector space and let $\mathcal{I}$ be the collection of all linearly independent subsets of S. The hereditary property is clear while the exchange property is an exercise in linear algebra.

Our main interest here is in *graphic matroids.* Let G be a graph and let S be the set of edges of G, $E(G)$. Let the independent subsets of S be all those sets of edges which do not contain a cycle—in other words, those that form sub-forests of G. Then we have the following important result.

Theorem 7.1 *The pair $(S, \mathcal{I})$ defined above for a graph G is a matroid.*

Proof That $\mathcal{I}$ is hereditary is easy since a subset of a forest is again a forest (no cycles). Let $A, B \in \mathcal{I}$. Suppose $|B| > |A|$ (that is, B has more edges). Now, the number of components of forest B is $|V| - |B|$ (exercise in Sheet 1) and the number of components of forest A is $|V| - |A|$.

Therefore B has fewer trees (connected components) than A. Therefore B has a tree T which contains vertices in diferent components of A. Hence T contains an edge uv such that u, v are in diferent components of A.

But this means that adding the edge uv to A does not create any cycle, that is, $A \cup \{uv\}$ is still in $\mathcal{I}$, as required. □

A *basis* for a matroid is an independent set which is not contained in any other independent set.

A *weighted matroid* is a matroid $(S, \mathcal{I})$ such that each element x of S is given a positive wieght $w(x)$.

The *weight of a subset* of S is the sum of the wieghts of its elements.

An *optimal subset* of the weighted matroid $(S, \mathcal{I})$ is an independent set with maximum weight.

Note: We can easily convert the minimum-spanning tree problem into a maximum-spanning tree problem by replacing each weight $w(e)$ of an edge e by

$W_0 - w(e)$, for W_0 larger than all the weights. Then, finding a maximum tree here is equivalent to finding a minimum tree in the original problem.

The proofs of the following two corollaries follow from the definitions.

Corollary 7.2 *Any two bases of a matroid have the same number of elements (by the Exchange Property).*

Corollary 7.3 *Any optimal set is a basis (since weights are positive).*

Theorem 7.4 *Consider the following greedy algorithm for finding an optimal (heaviest) set in a weighted matroid $(S, \mathcal{I})$.*

```
1.  A := ∅;
2.  Sort the elements of S in non-increasing order by weight:
        s_1, s_2, ..., s_m;
3.  For i := 1 to m do
        3.1 if A ∪ {s_i} is independent
                3.1.1 then A := A ∪ {s_i};
```

Then the resulting A is an optimal subset of the matroid.

Proof Let the elements in A be $\{a_1, a_2, \ldots, a_r\}$ in the order in which they were added (therefore $w(a_1) \geq w(a_2) \geq \ldots$). Suppose there is a basis B which is heavier than A. Let $B = \{b_1, b_2, \ldots, b_r\}$ with $w(b_1) \geq w(b_2) \geq \ldots$. Then there must exist a j such that $w(a_j) < w(b_j)$.

Consider $\{a_1, \ldots, a_{j-1}\}$ and $\{b_1, \ldots, b_j\}$ which are both in $\mathcal{I}$ (by the Hereditary Property).

By the Exchange Property, there exists $1 \leq k \leq j$ such that

$$\{a_1, \ldots, a_{j-1}, b_k\} \in \mathcal{I}.$$

But $w(b_k) \geq w(b_j) > w(a_j)$, and $b_k \notin \{a_1, \ldots, a_j\}$. Therefore in the jth stage of building up A (when $A = \{a_1, \ldots, a_{j-1}\}$), the element $b_k$would have been added to A instead of a_j (since it is heavier (therefore comes before) and $A \cup \{b_k\}$ is independent). But this is a contradiction. □

Corollary 7.5 *Kruskal's Algorithm gives a minimum-weight -spanning tree.*

8 Queues and Stacks

Sometimes an algorithm requires that a graph be traversed along its edges such that all vertices are visited. By 'visiting' a vertex we could mean simply outputting its label; or checking whether the particular vertex has a property we are seeking. The key point is that all vertices are visited. We therefore need a systematic way of traversing a graph which ensures that no vertex is left out but also that this traversal is done economically and that we do not fall into the opposite trap of re-visiting the same vertices over and over again, possibly getting stuck in a loop.

If a graph were a linear structure, that is a path, then we could simply visit the first vertex, then the second, and so on. But this is not possible for a general graph. Two main ways have been developed for traversing a graph: Breadth First Search (BFS) and Depth First Search (DFS)—the reason for these names will soon become apparent. It will turn out that in some algorithms which we shall study later, sometimes one of the two searches is more efficient and sometimes the other.

These two searches depend on two important data structures, queues and stacks.

8.1 Queues

A *queue* or a *first-in-first-out (FIFO)* list is an ordered list q of elements (e.g., stored in an array) with the element occupying the first position said to be in the *front* of the queue, and the one in the last position said to be in the *back* of the queue, and on which the following operations can be carried out:

`isempty`(q)
Boolean function returning YES or NO depending on whether q is or is not empty; q remains unchanged.

`add`(q, v)
Adds v to the end of the queue; v becomes the new element in the back of q.

`remove`(q)
Removes the front element from q; the next element becomes the new front element.

`front`(q)
Returns the value of the front element of q; q remains unchanged.

8.2 Stacks

A *stack* or a *last-in-first-out (LIFO)*list is an ordered list s, with the last element called the *top* of the stack, and on which the following operations are defined:

`isempty`(s)
Boolean function returning YES or NO depending on whether s is or is not empty; s remains unchanged.

`push`(s, v)
Adds v to the top of the stack; v becomes the new top element of s.

pop(s)
Removes the top element from s; the resulting last element in the list becomes the new top element.

top(s)
Returns the value of the top element of s; s remains unchanged.

[Those familiar with recursive programming will recognise that such a programme depends on some underlying stack. Alternatively, if one were required to transform a recursive programme into an iterative one, then one would need to implement an appropriate stack.]

8.3 BFS and DFS

We can now present the algorithm for BFS and DFS. Notice that the 'search route' created in both cases is a tree, because we only traverse each vertex once—hence the search contains no cycles and is as 'economical' as possible.

```
Breadth First Search (BFS)
     1.  Visit vertex v; { v is the designated start vertex for the search}
     2.  add(q, v);
     3.  While(not isempty(q)) do
               3.1 x := front(q);
               3.2 if(x is adjacent to an unvisited vertex y) then
                     3.2.1 visit y;
                     3.2.1 add(q, y)
               3.3 else
                     3.3.1 remove(q); { the front vertex of q has been 'served'};

Depth First Search (DFS)
     1.  Visit vertex v; { v is the designated start vertex for the search}
     2.  push(s, v);
     3.  While (not isempty(s)) do
                3.1 x := top(s);
                3.2 if(x is adjacent to an unvisited vertex y) then
                      3.2.1 visit y;
                      3.2.2 push(s, y)
                3.3 else
                      3.3.1 pop(q); { top item has been 'served'};
```

8.4 An example

The examples below show the graph in Figure 4 traversed using the two methods. In both cases we give the vertex being currently visited at each step, the composition of the stack or queue at the corresponding stage and the search tree which is grown as a subgraph of the graph. Note that when more than one vertex can be visited at some stage, the algorithms do not specify which one to choose—it can be taken at random or, say, as we do here in alphabetical order.
Breadth-First-Search

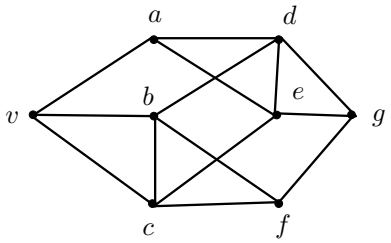


Figure 4: Graph to be searched

Visiting	Queue
v	v
a	va
b	vab
c	$vabc$
	abc
d	$abcd$
e	$abcde$
	$bcde$
f	$bcdef$
	$cdef$
	def
g	$defg$
	efg
	fg
	g
	$\emptyset$

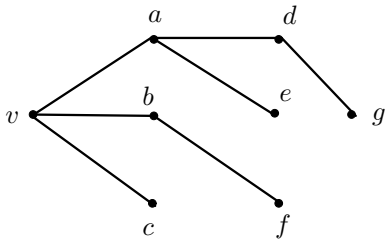


Figure 5: Search tree grown by BFS

Depth-First-Search

Visiting	Stack
v	v
a	av
e	eav
g	$geav$
f	$fgeav$
c	$cfgeav$
	$fgeav$
b	$bfgeav$
d	$dbfgeav$
	$bfgeav$
	$fgeav$
	$geav$
	eav
	av
	v
	$\emptyset$

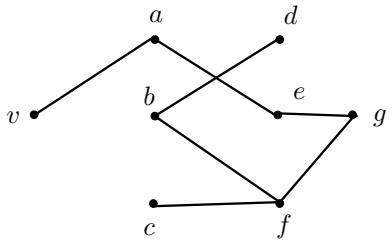


Figure 6: Search tree grown by DFS

9 Shortest Path Problem

We shall describe Dijkstra's Algorithm which is a variant of BFS. We are given a connected graph G with positive weights (lengths) on every edge. A start vertex v is designated. The algorithm finds minimum distances between v and each other vertex. Some points to note regarding this algorithm:

- Pairs of vertices which are not adjacent are assumed to be joined by an edge of weight ∞, where ∞ is a large number (say, larger than the sum of all the weights of the other edges).
- At the end of the algorithm, each vertex u will get a label $l(u)$ which will be the minimum distance from v to u. During the course of the algorithm $l(u)$ could be a temporary label which will be successively refined. At the end of the algorithm, all labels $l(u)$ will be permanent.

- The algorithm can be modified by including a backtracking from u to v to find an *actual* path of minimum length, not only the value of this minimum length.
- The algorithm works equally well for digraphs but *not* for graphs with negative weight (see exercise at the end of the section).

In a following subsection we shall describe a tabular method which is very useful for implementing the algorithm by hand for relatively small graphs or digraphs. We shall also analyse the computational complexity of Dijkstra's Algorithm

9.1 Dijkstra's Algorithm

For more details on this algorithm you can see Biggs Sect 9.6. Gibbons describes the algorithm in Fig 1.10 and proves that it works (that is, actually gives the minimum distances) in Th 1.3.

```
1.  S := v; {the designated start vertex}
2.  l(v) := 0; {distance from v to v is zero}
3.  S̄ := V − S; { S is the set of permanently marked vertices; S̄ the others}
4.  T := 0; {edges in the partial tree which will eventually contain shortest
        paths from v to all other vertices;
        T will grow in BFS fashion}
5.  latest:= v;
6.  for every x ∈ S̄ do {initialisation}
        6.1 l(x) := ∞; pred(x) := nil;
7.  while S̄ ≠ 0 do
         7.1 for every x ∈ S̄ do
                7.1.1 l(x) := min(l(x), l(latest) + w(x, latest);
                7.1.2 if l(x) has been updated then
                          7.1.2.1 pred(x) :=latest;
         7.2 find a vertex x ∈ S such that l(x) is minimal;
         7.3 let y :=pred(x);
         7.4 let latest:= x;
         7.5 S := S ∪ {x};
         7.6 S̄ := S̄ − x;
         7.7 T := T ∪ {xy};
```

Note: The lines 1–4, 7, and 7.2–7.7 grow a tree in BFS mode, and the algorithm is grafted over this BFS..

9.2 An example: The tabular method

We shall illustrate the method by an example. The graph in question is the one shown in Figure 7. The method actually gives slightly more than the algorithm as presented above. The last column of the table (headed "new edge") gives the edge along which the most recent improvement to the l value of the "next" vertex has occurred (this can easily be read off from the rest of the table). These new edges between them give a (BFS) subtree of the graph such that the path from v to any vertex x along this tree is one of shortest length.

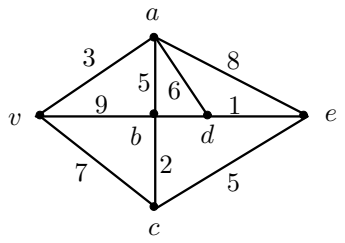


Figure 7: Finding minimum distances from v

	Labels							
latest	v	a	b	c	d	e	next	new edge
v	$\underline{0}$	∞	∞	∞	∞	∞		
v		$\underline{3}$	9	7	∞	∞	a	va
a			8	$\underline{7}$	9	11	c	vc
c			$\underline{8}$		9	11	b	ab
b					$\underline{9}$	11	d	ad
d						$\underline{10}$	e	de

A suitable value for ∞ would be, say $\sum$(weights) $= 50$. The final spanning tree shown below contains a shortest path from v to any vertex in the graph.

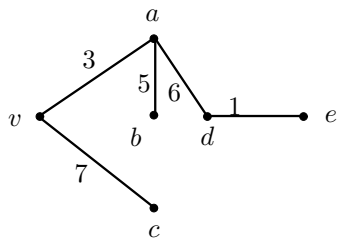


Figure 8: The spanning tree containing minimum length paths from v

9.3 Computational complexity of Dijkstra's Algorithm

At the ith stage of the algorithm, $|S| = i$ and $|\overline{S}| = n - i$. Therefore

- to update the "l" values of the vertices in $\overline{S}$ and to check if the latest addition to S can replace any of the "predecessors" of the vertices in $\overline{S}$ requires $(n - i)$ comparisons and $(n - i)$ additions;
- to find a vertex with smallest "l" value in $\overline{S}$ requires $(n-i-1)$ comparisons.

Therefore the total number of operations required equals

$$\sum_{i=1}^{n-1} 2(n-i) + (n-i-1) = 2(n-1)(n-2) + (n-2)(n-3) = O(n^2).$$

10 Sheet 2: Minimum trees & shortest paths

1. * Find a shortest route from A to F, and its length, in the graph with vertices A, B, ..., F and distances given by the table below:

	A	B	C	D	E	F
A	-	5	8	3	4	9
B		-	6	1	5	4
C			-	3	9	2
D				-	4	6
E					-	5
F						-

2. * Find a minimum spanning tree and its weight in the graph with vertex set $\{x, a, b, c, d, e, f\}$ and edges and weights as given in the table below:

xa	xb	xc	xd	xe	xf	ab	bc	cd	de	ef	fa
6	3	2	4	3	7	6	2	3	1	8	6

3. * Consider the graph below. Using Kruskal's Algorithm, find a minimum weight spanning tree (MWST) T_1 for this graph. Using Dijkstra's Algorithm, find the tree T_2 which gives all the shortest paths from a to every other vertex.

   Verify that in T_1 the path from vertex a to vertex x is not a shortest path. Verify also that T_2 is not a MWST.

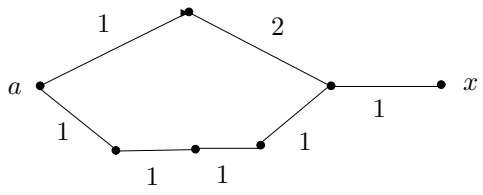


Figure 9: Kruskal and Dijkstra give different spanning trees

4. Use Dijkstra's Algorithm to find a shortest path from vertex 1 to vertex 4 in the following digraph, and hence deduce that the algorithm fails in this case.

5. Determine whether the graph described by the following adjacency matrix is connected.

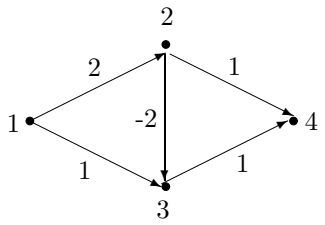


Figure 10: Failure of Dijkstra's Algorithm with negative weights

	1	2	3	4	5	6	7	8	9
1	0	1	0	0	0	1	0	1	1
2		0	0	0	0	1	0	0	1
3			0	1	1	0	0	0	0
4				0	1	0	1	0	0
5					0	0	1	0	0
6						0	0	1	0
7							0	0	0
8								0	1
9									0

6. Suppose that the vertices of a graph G are denoted by $v_1, \ldots, v_n$ and its edges by $e_1, \ldots, e_m$. Let B be the *incidence matrix* of G, that is, $B_{ij} = 1$ if v_i is incident to e_j and $B_{ij} = 0$ if v_i and e_j are not incident. Let the weight of each edge e_j be given by w_j. Consider the problem of finding a shortest path from v_1 to v_n. Let $x_1, \ldots, x_m$ be the variables which can take values 0 or 1 where $x_j = 1$ means that the edge e_j is in the shortest path and $x_j = 0$ means that it is not. Formulate the problem as a LP in the variables $x_1, \ldots, x_n$.

   If G is a digraph then the definition of the matrix B is modified so that now $B_{ij} = 1$ if e_j is incident from v_i and $B_{ij} = -1$ if e_j is incident to v_i. Reformulate the problem as a LP in this case.

7. * A car has just been purchased for Lm1200. The cost of maintaining a car during the year depends on the age of the car at the beginning of the year, as given in the table below. At the end of any year, the car may be traded in and a new one purchased. The price received on a trade-in depends on the age of the car at the time of the trade-in (see table). Assume that the cost of a new car remains Lm1200, find an optimal policy which will minimise the net cost (purchase + maintainance - trade-in) incurred during the next five years. (Assume that at the end of the five years the car will be written off no matter what its age is.)

Age of car (yrs)	Maintenance cost for a year (Lm)	Trade-in price (Lm)
0 (new)	200	-
1	400	700
2	500	600
3	900	200
4	1200	100
5	-	0

   [*Hint:* Draw a network with vertices $v_1, \ldots, v_6$ such that the cost of an arc $v_i v_j (i < j)$ denotes the total cost of purchasing a new car at the beginning of

year i and trading it in at the beginning of year j. (Note that the cost of $v_i v_j$ depends only on $j - i$.)]

8. * A library must build shelving to store 200 4-inch high books, 100 8-inch high books and 80 12-inch high books. Each book is 0.5 inches thick. The library can order 4-inch or 8-inch or 12-inch high shelves, and it can, for instance, store all books of height ≤ 8 inches in a shelf of height 8 inches and all 12-inch books in a 12-inch shelf; or else, it could store all books in one 12-inch high shelf.

   A shelf costs Lm230 to build and a cost of 50c per sq. in. is incurred for book storage. (Assume that the area required by a book is given by the height of the shelf times the book's thickness. Assume also that all books of the same height are to be stored on the same shelf.)

   Formulate and solve a shortest path problem that could be used to help the library determine how to shelve the books at minimum cost.

   [*Hint:* Have vertices 0,4,8 and 12 and an arc joining vertex i to vertex j for all $i < j$; the cost c_{ij} of this arc would be the minimum of shelving all books of height $> i$ and $\leq j$ on a single shelf.]

9. * Consider the network shown which represents the road system connecting the seven towns on a small island.

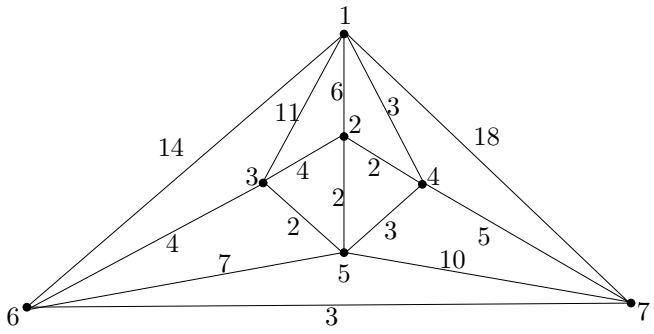


Figure 11: Seven towns and their distances

(a) The following table gives the shortest distances between all pairs of vertices of the network. Verify a few of them using Dijkstra's Algorithm.

	1	2	3	4	5	6	7
1	-	5	8	3	6	11	8
2		-	4	2	2	8	7
3			-	5	2	4	7
4				-	3	8	5
5					-	6	8
6						-	3
7							-

(b) The populations of the seven towns is as shown in the following table. A hospital is to be built in one of the towns. Assuming that each member of the population will pay one visit per year on average to the hospital, where should it be located to minimise the total distance travelled by the population?

Town	1	2	3	4	5	6	7
Popl. ×1000	41	28	30	26	27	40	38

(c) Two schools are to be built and towns 1, 3, 4, 6 and 7 offer suitable sites. Assuming that the size of the schools can be adjusted so that any child can attend its nearest school, where should the schools be built in order to minimise the maximum distance any child must travel to school?

11 Critical Path Analysis

Suppose that a project consists of n tasks or jobs and that the duration of each task is given. Suppose we are also the precedence relations (that is, task B cannot start before task A finishes, etc) amongst the tasks are known.

Let us also make a few other assumptions in order to simplify the problem. We suppose that, at any instant, there is a sufficient number of workers in order to start all tasks which can be carried out at that moment subject to the precedence rules (we shall see later the consequence of having a limited number of workers). Suppose also that if job A must precede job B then all of A must be completed before work on B can start. Finally, we shall assume that a worker can work on one job at a time and if a job is started then it must be carried out to termination.

Under these conditions, can we determine how long the whole project will take?

In order to solve this problem we shall represent the situation by an *activity network*. (Our form of activity network is also called an *activity on node (AON) network*; there is also what is called the *activity on arc (AOA) network* which we shall not be using.) This will be a connected digraph D *without directed cycles* (that is, acyclic) and no multiple arcs, on $n+2$ vertices, such that:

1. There is one vertex s (the *start vertex*) with indegree zero and one vertex t (the *finsih vertex*) with outdegree zero;

2. The vertices are labelled $0, 1, 2, \ldots, n, n+1$ such that s is labelled 0, t is labelled $n+1$, and such that, for any arc uv, label(u) $<$ label(v. This labelling, called a *topological sort* of the network, is possible because the network is acyclic.)

3. Each arc uv is given a positive label $w(u, v)$ such that all arcs incident from the same vertex u have the same label. This label is called the *duration* of u. Vertex s always has duration 0, while the duration of t is not defined since it has no arcs incident from it. The length of a directed path in the network is defined to be the sum of the weights of the arcs on the path.

The answer to our question can now be translated into a question about activity networks. For a given project, draw the corresponding activity network where each job is represented by a vertex and such that, if job A must precede job B, then there is an arc from the vertex corresponding to A to the vertex corresponding to B. Then the minimum time-span over which the project can be finished is equal to the maximum length of a directed path from s to t. (Think a few minutes about this to convince yourself that the required minimum is equal to a maximum length.) Paths of longest length in the activity network

are called *critical paths*, their length is called the *critical time* of the project and jobs corresponding to vertices on critical paths are called *critical jobs* because delaying any one of them will delay the overall project.

In the next section we shall see how such a maximum path is determined.

11.1 An example

We shall illustrate the method by means of an example. Note that determining maximum paths is equivalent to determining minimum paths. Therefore our algorithm will essentially be Dijkstra's Algorithm but, however, a speeded up version of it—this is possible because the digraphs we are working with are acyclic. The topological sort algorithm ensures that the labels $1, 2, \ldots, n$ given to the jobs is consistent with the precedence relations, as described above. It is essentially this numbering that will help exploit the acyclic nature of the digraph

PROJECT

Activity	A	B	C	D	E	F	G	H	I	J
Duration	2	7	15	8	10	2	5	8	2	3

PRECEDENCE RULES

D	must follow	E and B
E	must follow	A and B
F	must follow	D and G
G	must follow	E
H	must follow	G
I	must follow	C, F, G

`TOPOLOGICAL SORT` { `Labelling the activities` $1, 2, \ldots, n$ `& detecting any cycles`}

1. $N := \{1, 2, \ldots, n\}$;
2. $L(S) := 0$;
3. `while` $N \neq \emptyset$
   - 3.1 `remove from the network the last group of vertices labelled;`
   - 3.2 `label vertices with in-deg=` 0 `with first available labels from` N;
   - 3.3 `remove these labels from` N;

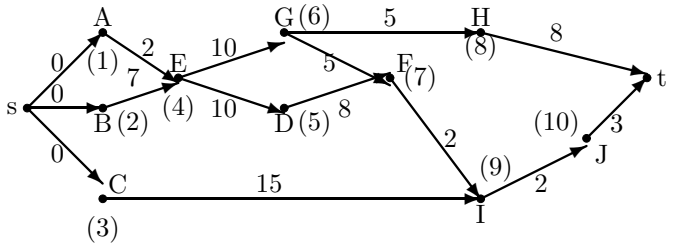


Figure 12: The above project represented graphically

The next algorithm, finding a maximum path, is a speeded up version of Dijkstra's Algorithm (exploiting the above acyclic labels). $E(v)$ is analogous to $l(v)$ in Dijkstra, and gives the length of a longest path from s to v, that is, the *earliest starting time* of v.

EARLIEST STARTING TIME $E(v)$
$E(v)$ is obtained recursively as follows:-
$E(s) := 0$;
$E(v) := \max_u\{E(u) + w(u, v)\}$;
where the maximum is taken over all vertices u such that uv is anarc and $E(u)$ has already bee

After the above "forward recursion" we carry out a "backward recursion" to find the *latest starting times*. (The latest starting time of an activity is the time beyond which the activity cannot be delayed without delaying the whole project.)

LATEST STARTING TIME $L(v)$
$L(v)$ is obtained recursively as follows:-
$L(t) := E(t)$;
$L(v) := \min_x\{L(x) - w(v, x)\}$;
where the minimum is taken over all vertices x such that vx is anarc and $L(x)$ has already bee

THE ABOVE EXAMPLE CONTINUED

vertex v	s	A	B	C	E	D	G	F	H	I	J	t
$E(v)$	0	0	0	0	7	17	17	25	22	27	29	32
$L(v)$	0	5	0	12	7	17	17	25	24	27	29	32
slack time or *float time* $= L(v) - E(v)$	0	5	0	12	0	0	0	0	2	0	0	0

Activities with 0 slack time must be started as soon as allowed by the precedence rules (that is, by time $E(v)$). These are called *critical activities*.

To introduce us to the next section, suppose we continue this example but this time we have only two workers (or teams of workers or, more generally, processors) available. How do we schedule the jobs, that is, if at an instant there are more jobs ready to be started than workers available, which jobs should we start first. We shall discuss this situation in some more detail in the next section, but for this example we illustrate a useful and popular way of representing the scheduling in what is called a *time diagram* or *Gantt chart*.

Can you see why the above scheduling with two processors is optimal?

11.2 Limited number of processors

As we have remarked, the above assumes that we have an unlimited number of processors available. In that case the minimum possible duration of the project is equal to the critical time. Consider the other extreme, when only one processor is available. Clearly, the only way to do the jobs is one by one, respecting the precedence rules, and the total project duration will equal the sum of the durations of every subtask.

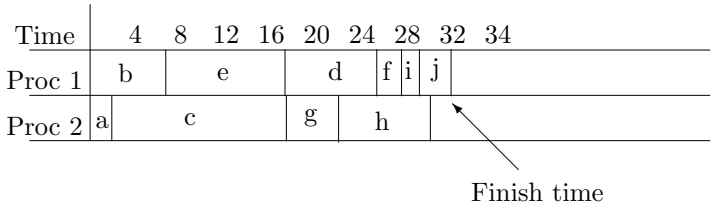


Figure 13: *Time diagram* or *Gantt chart* for the above example with two processors

We now consider the more realistic intermediate problem, that is, a fixed number of processors (such as in the example at the end of the previous section. The main aim of this section is not to give complete solutions, but to introduce you to the idea of combinatorial optimisation problems for which, unlike all the problems we have seen up to now, we do not know if they have a solution. By this we mean that *there is no known algorithm which gives the solution in polynomial time but nobody has yet shown that such an algorithm exists.* And scheduling problems are a very good example of such problems, because they are very important in practice, and because some of the questions which arise are amongst the most difficult when dealing with such types of problem.

Very often, when dealing with scheduling situations, the following "busy rules" are applied:

1. No processor is idle if it can be doing some activity. This is sometimes described by saying that the scheduling contains no *intentional idle periods.*
2. Once a processor starts an activity it must finish it.
3. The project must be finished in optimal time.

Not only is 3) an unsolved problem in the sense described above, but rules 1) and 2), which seem very natural and efficient, are sometimes incompatible with 3). Take the following theorem, for example. [No attempt to prove any of these theorems will be made, since their proofs are very long and difficult.]

Theorem 11.1 *For a given set of tasks, let w denote the total elapsed time when tasks are scheduled, amongst r processors, without intentional idle periods, and let w_o be the optimal time. Then*

$$\frac{w}{w_o} \leq 2 - \frac{1}{r}.$$

Moreover, it is possible to construct projects where actually w/w_o is equal to $2 - 1/r$.

Corollary 11.2 *The time w taken by a scheduling without intentional idle periods is never worse than $[(1 - 1/r) \times 100]\%$ of the optimal time.*

Proof From theorem it follows that

$$\frac{w - w_o}{w_o} \leq (1 - \frac{1}{r}.$$

□

The best way to illustrate the above result is by a simple example. Figure 14 shows a project which is to be scheduled amongst two processors.

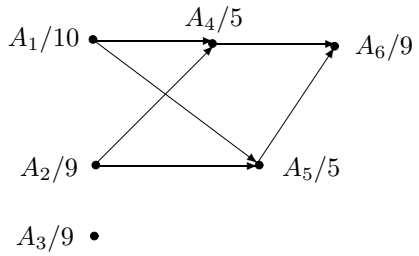


Figure 14: A simple project

By simple trial and error you can convince yourself that the scheduling given in Figure 15 is optimal and that, moreover, any scheduling which avoids intentional idle periods must give a worse than optimal result.

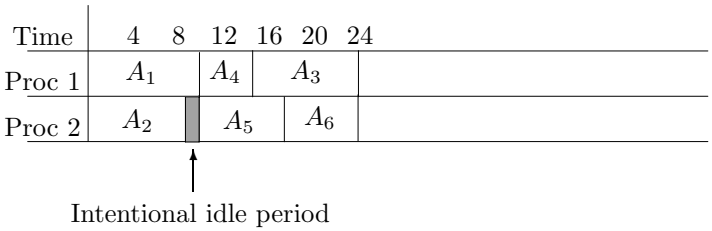


Figure 15: Optimal time can only be achieved with intentional idle periods

A word of note is important here. Many students, seeing a result such as the previous theorem and others which will follow soon wonder how we can obtain estimates in terms of the optimal time w_o which we do not know. To show you how this is done one nedds to go through the proof, which, as already remarked, is very difficult. But in a later chapter we shall encounter an easier result of this type, with full proof, and the fact that one can obtain estimates depending on unknown optimal values should become clear. The important thing is that we can obtain results such as the corollary, namely, a guarantee that the solution obtained is not worse than the optimal by some known factor. This consideration is of utmost important in the next section.

11.2.1 Heuristic criteria

So we now know that there are problems for which a polynomial time solution is not in general available. What do we do in such cases? One of the strategies applied to tackle such problems is to use *heuristic* algorithms. "Heuristic" means approaching a problem by trial and error, getting an approximate result without full knowledge of how to solve it completely. What a heuristic algorithm is designed to do is to obtain an approximate solution to the problem, but in polynomial time. The analysis of heuristic algorithms differs markedly from the other algorithms we have studied so far. Showing that the algorithm has polynomial time complexity is usually not difficult. Implementing the heuristic algorithm as computer code is generally not more or less difficult than for algorithms which give an optimal solution. The main difference is that we must here show that there is an upper bound which limits how far the solution obtained is from the optimal. Such results are similar to Theorem 11.1, and very often obtaining such results is very difficult. We shall give here and in the next sections (without proof!) some results of this type, but in the later section on the travelling salesman problem we shall encounter a result of this type whose proof is quite easy and which we shall give in full. At least, the student would have seen one instance of how such results are obtained.

Returning to our problem, that is, scheduling with a limited number of processors, what heuristic can one propose. One popular heurisitc is called *critical path scheduling*. Here, if a number of jobs can be executed but the number of processors is less, we always assign first the job *with the smallest (i.e. earliest) latest-starting-time*. That is, if job A has latest starting time 10 days, and job B has latest starting time 7 days, then we schedule job B first. This simple heuristic is really an extension of the idea of giving preference to critical jobs—here we are giving preference to the "more critical" jobs. In spite of its being quite reasonable, this heuristic, like all heurisitcs for this problem does not guarantee an optimal solution, as examples can show. And although the idea is simple, finding an upper bound on the error which can arise is very difficult indeed. In fact, the following theorem does that but with the added simplifying assumption that there are no precedence relations between the jobs. The proof still remains difficult.

Theorem 11.3 *Let W_{cp} be the time obtained by critical path scheduling and assume that there are no precedence relations between jobs (that is, the scheduling consists in doing the longest jobs first). Let there be r processors and let w_o be the time taken by an optimal scheduling. Then*

$$\frac{w_{cp}}{w_o} \leq \frac{4}{3} - \frac{1}{3r}$$

and the upper bound can be attained.

11.2.2 Bin packing

If you have tried your hand at a few examples of scheduling jobs amongst a fixed number of processors you might very likely have realised that this is a sort of packing problem. Think of the rows of a Gantt chart to be tubes or drawers (one for each processor) and the jobs to be blocks of different length (durations) which have to be packed into the drawers (subject to precedence rules) with the

aim of minimising the length of blocks contained in the drawer with the longest sequence of blocks.

In fact, scheduling and packing are often different ways to view the same problem, and both are, in general, very difficult problems. To bring out more the analogy and to mention, albeit briefly, another well-known problem we shall consider the bin packing problem. Here, the number r of "drawers" is unlimited, but the maximum length of "blocks" which can be fitted in each drawer is fixed (and the same for each drawer. The aim now is to minimise r, the number of drawers used. The reason for the name "bin packing" should be clear: the drawers are bins of equal capacity, and a number of objects is to be packed in the least number of bins while not exceeding the In terms of job scheduling, the problem can be worded this way: a given finish time (not more less the critical time of the project, of course) is set as a target, and we are required to find a scheduling which completes the projects with this specified time but employing the least number of processors.

There are two obvious heuristics to apply here (again, we assume, for simplicity, that there are no precedence rules). One is called *first fit*, that is, we schedule the jobs in the order in which they happen to arrive (sometimes the only way to carry out the task, for example, in an "on-line" situation. The second is called *first fit decreasing*, that is, we order the jobs in decreasing order of length (largest first), and schedule them accordingly.

Theorem 11.4 *Let r_{ff} and r_{ffd} be the number of processors required by first fit and first fit decreasing scheduling, respectively. Let r_o be the number of processors required by an optimal scheduling. Then,*

$$r_{ff} \leq \frac{17}{10} r_o + 2$$

and

$$r_{ffd} \leq \frac{11}{9} r_o + 4,$$

and moreover these bounds are both attainable.

The bin packing problem appears in various guises: determining the minimum number of advert slots (of given length) within which to place the adverts which need to be aired; finding the minimum number of pipes of standard length from which to cut an order of pipes of varying lengths; etc.

A simple exercise in the problem sheet below illustrates how counterintuitive solutions to this problem can be. An excellent and very readable reference which presents concrete examples illustrating the seemingly paradoxical nature of scheduling and packing problems is the chapter written by R.L. Graham, "Combinatorial Scheduling Theory", found in the book *Mathematics Today* edited by Lynn Arthur Steen (available in the UoM Library).

11.2.3 A simple scheduling problem and matroids

We shall now present a scheduling problem which, although somewhat artificially simplified, illustrates two points: that not all scheduling problems must be "unsolvable" and that the concept of matroids which we introduced earlier has wider applications in combinatorial optimisation than just the application

to Kruskal's Algorithm. We shall not give all the details which can be found in Cormen, Leiserson and Rivest (CLR) in Chapter 17.

Let S be a given set of tasks $1, 2, \ldots, n$. There is only one processor so tasks have to be carried out sequentially, one by one. There are no precedence rules and each duration is one unit (day, second, week, etc). For every task i there is given its deadline d_i and the penalty w_i if the deadline is exceeded (the amount to be paid as penalty does not depend on *how much* later than the deadline the task is carried out. What is required is to find a schedule (that is, a sequencing of the tasks) which minimises the total penalty.

Let us note the following points about this problem.

1. A schedule is simply an ordering of the tasks.
2. Any schedule can include *early tasks*—those carried out before their deadline—and *late tasks*—those carried out after their deadline.
3. Each schedule can be re-arranged so that early tasks come first. This does not change which tasks are early and which are late, that is, does not affect the overall penalty.
4. The early tasks can be re-arranged in order of non-decreasing deadlines, again without changing early or late tasks. This is usually called the *canonical form* of the scheduling.
5. The late tasks can be carried out in any order without changing the overall penalty.
6. The problem of minimising the sum of the penalties of late tasks is equivalent to the problem of maximising the sum of the penalties of the early tasks.

The above points are useful as an exercise in understanding the nature of the problem, but more importantly in the proof of the next theorem. We omit this proof which can be found in CLR as Theorem 17.12 (p.352).

First one definition. A subset A of the set of tasks S is said to be *independent* if all the tasks in A can be scheduled such that none are late. Let $\mathcal{I}$ be the set of independent subsets of S.

Theorem 11.5 *The system $(S, \mathcal{I})$ is a matroid.*

This then easily implies that the following greedy algorithm finds an optimal schedule.

1. Order the tasks in decreasing order of penalties those with heavier penalties first;
2. Let $A := \emptyset$;
3. While $i \leq n$ do
   3.1 if $A \cup \{i\}$ is independent then $A := A \cup \{i\}$;
4. The set A which is returned is a set of early tasks in an optimal schedule;

This is best illustrated by an example. Here the tasks are already sorted in decreasing order of penalties.

Task	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10
Choose (Set A)	1	2	3	4			7
Reject					5	6	

Therefore an optimal scheduling is given if we schedule the tasks $1, 2, 3, 4, 7$ as early tasks. This gives the schedule $\langle 1, 2, 4, 3, 7, 5, 6\rangle$. An equivalent schedule (that is, the same set A of early tasks and hence the same overall penalty) would be given, say, by $\langle 2, 4, 1, 3, 7, 6, 5\rangle$.

11.3 Variations on the Critical path theme

We now leave the problem of scheduling with a limited number of processors and continue with our treatment of critical path analysis, but we consider two variants.

11.3.1 Crashing the project

Suppose we are given not only the duration of each activity but also the amount by which this duration can be shortened (this is called *crashing* the activity and, for each activity, the amount per unit time reduced that crashing will cost. Suppose that there is a bonus to be gained by shortening the duration of the overall activity, or else, there is a penalty to be paid in delaying it. Then there are trade-offs to be considered in whether, which and by how much activities are to be crashed so that the overall cost (or profit) is minimised (maximised).

The best way to deal with such problems is to reduce them to a linear programming problem, and then solve them by, say, the simplex method. We shall illustrate all this by an example, presenting along the way the main assumptions involved in the simplest version of crashing. We do not attempt to solve the linear programme; in this course we content ourselves in *formulating* the problem as an LP.

Example 11.1 *We first present a strightforward critical path problem without crashing, and show how this can be formulated as an LP. This will then be the basis of the formulation as an LP of the same problem with crashing.*

Activity	*1*	*2*	*3*	*4*	*5*	*6*
Predecessors	-	-	*1,2*	*1,2*	*4*	*3,5*
Duration	*6*	*9*	*8*	*7*	*10*	*12*

Formulate as an LP the problem of finding the critical time.

Solution Let x_i be the earliest starting time of activity i. Let activity 7 denote the finish.

Problem: Minimise $z = x_7 - x_1$ subject to the constraints:

$$
\begin{aligned}
x_3 &\geq x_1 + 6 \\
x_3 &\geq x_2 + 9 \\
x_4 &\geq x_1 + 6 \\
x_4 &\geq x_2 + 9 \\
x_5 &\geq x_4 + 7 \\
x_6 &\geq x_3 + 8 \\
x_6 &\geq x_5 + 10 \\
x_7 &\geq x_6 + 12 \\
x_7 &\geq x_5 + 12
\end{aligned}
$$

□

Example 11.2 *Now suppose that each activity can be "crashed". The table below gives the crash time, the cost for the crash time, the cost for the normal time (that above durations) and the income gained per day reduced in overall project time is Lm250.*

Activity	*1*	*2*	*3*	*4*	*5*	*6*
Crash time	*6*	*5*	*5*	*4*	*8*	*7*
Crash cost (× Lm100)	-	*7*	*2*	*2*	*3*	*4*
Normal cost	-	*2*	*1*	*1*	*1*	*1*

Obtain an LP formulation for finding the optimal overall cost obtained by crashing activities appropriately.

Solution *Assumptions:*

1. Cost of crashing is linear, that is cost per unit time crashed equals (crash cost - normal cost) divided (normal time - crash time).
2. Similarly for overall project cost.

Costs per unit time crashed:

Activity	1	2	3	4	5	6
Crash cost/time (Lm100)	-	5/4	1/3	1/3	1	3/5

Overall savings/unit time = 2.5 (× Lm100).
Let y_i be the amount by which activity i is crashed.
Problem: Minimimise

$$\frac{5}{4}y_2 + \frac{1}{3}y_3 + \frac{1}{3}y_4 + y_5 + \frac{3}{5}y_6 - 2.5y_7$$

subject to the constraints

$$
\begin{aligned}
0 \leq\ & y_2 \leq 4 \\
0 \leq\ & y_3 \leq 3 \\
0 \leq\ & y_4 \leq 3 \\
0 \leq\ & y_5 \leq 2 \\
0 \leq\ & y_6 \leq 5 \\
0 \leq\ & y_7
\end{aligned}
$$

and

$$\begin{aligned}
x_3 &\geq x_1 + 6 \\
x_3 &\geq x_2 + 9 - y_2 \\
x_4 &\geq x_1 + 6 \\
x_4 &\geq x_2 + 9 - y_2 \\
x_5 &\geq x_4 + 7 - y_4 \\
x_6 &\geq x_3 + 8 - y_3 \\
x_6 &\geq x_5 + 10 - y_5 \\
x_7 &\geq x_6 + 12 - y_6 \\
x_7 &\geq x_5 + 12 - y_5
\end{aligned}$$

and $x_7 - x_1 \leq$ critical time found from first part (or else, any other target date; or unspecified). □

11.3.2 PERT

In this variant durations are given probabilistically; in fact, the expected duration is given for each activity. Usual assumptions are:

1. Durations are independent random variables.
2. Often, each duration is assumed to follow a beta distribution. That is, for a given duration d_i we are given a_i, esatimate of the most favourable duration; b_i, estimate of the least favourable duration; m_i, estimate of the most likely duration. Then, the expected value of d_i is
$$\frac{a_i + 4m_i + b_i}{6}$$
and its variance is
$$(b_i - a_i)^2/36.$$
3. Critical path contains enough activities to justify the use of the Central Limit Theorem, that is, the critical time is normally distributed with mean
$$\sum_{i \text{ on path}} E(d_i)$$
and variance
$$\sum_{i \text{ on path}} Var(d_i).$$
4. The expected duration of the project depends on the expected duration of the critical path.

One can then attempt to answer questions concerning the probability that, say, the project will be completed by a certain date.

Example 11.3 *Suppose that the critical path has expected duration 38 days with variance 7.22 days. What is the probability that the project will be completed within 35 days?*

Solution Assuming normal distribution, the standard z-value corresponding to 35 is

$$\frac{35-38}{\sqrt{7.22}} = -1.12$$

(this is equal to the number of standard deviations away from the mean).

Therefore, from tables,

$$P(z \leq -1.12) = 0.13,$$

that is, there is a 13% chance that the project will finish within 35 days. □

One can easily criticise some of the main assumptions of PERT. That durations are independent random variables could be a quite reasonable assumption, as also could be the assumption that there are enough activities on the critical path to justify the use of the Central Limit Theorem (although certainly not in these small examples which are designed to be worked out "by hand").

However, the most serious shortcoming of PERT is that it assumes that the critical path, being the longest path, is the only one that affects the probability of the overall duration of the project. This is certainly not the case in general. One can easily realise that if the critical path has expected duration 30 days, while another path has expected duration of 28 days, then *probability* that the overall duration is so and so could just as much be affected by the second path as by the first, especially if the second path has a much larger variance.

It is very difficult to solve this problem analytically, since here we certainly cannot assume that the expected durations of the *paths* are independent. The main tool to solve these problems is often simulation.

12 Sheet 3: Critical path and scheduling

1. A project is made up of the following activities:

Activity	Duration	Must follow
A	1	-
B	2	-
C	3	-
D	5	-
E	3	B, C
F	2	A
G	5	C
H	2	E, F, I
I	3	B, C, E, G

   (a) Construct an activity network for this project.

   (b) Find the earliest and latest starting times for each activity.

   (c) Schedule the jobs using the CP-scheduling algorithm if (i) two workers are available; (ii) three workers are available. In each case determine (by trial and error) whether your solutions are optimal schedules and, if not, find an optimal schedule.

2. * The table below lists six activities which constitute a project, together with the sequencing requirements and estimated durations for each activity.

Find the earliest and latest starting times for each activity and obtain the least number of days within which the project can be completed.

Activity	A	B	C	D	E	F
Duration	2	3	4	6	2	8
Pre-requisites	-	A	A	B, C	-	E

Suppose that two teams of workers are available and that each team can work on only one activity at a time and also both teams cannot work on the same activity. Draw up the CP-schedule of the activities in the project for the two teams of workers. Is this schedule optimal?

3. *

   (a) A pipe-works factory has an order for ten pipes of the following lengths:

   $$760, 395, 395, 379, 379, 241, 200, 105, 105, 40.$$

   These lengths have to be cut from pipes of standard length 1000. It is decided that the pipes will be produced on a first-fit decreasing (FFD) basis, that is, in decreasing order of lengths, and in the following fashion: the first pipe used will be called Pipe 1, the second Pipe 2, etc; whenever a new pipe is to be cut, Pipe 1 will be used if possible, otherwise Pipe 2, etc, and if none of the already used pipes is long enough, then a new pipe is used.

   Show that three standard pipes are required to meet the order and that, if each of the above lengths were reduced by 1, then four standard pipes would be required!

   (b) Suppose that the pipes ordered are of lengths

   $$7, 9, 7, 1, 6, 2, 4, 3$$

   and standard pipes are of length 13. Suppose that pipes will be cut on a first-fit (FF) basis, that is, in the order in which they are presented above, but otherwise using the same method as before.

   Show that three standard pipes are required but that if the pipe of length 1 is omitted then four standard pipes would be required!

4. * A toy company wants to introduce a new product for Christmas. The tasks required before the product can be made available in the shops, together with estimates for their durations, are:

Activity	Predecessors	a	b	m
A=obtain raw materials	-	2	10	6
B=train workers	-	5	13	9
C=publicity campaign	A, B	3	13	8
D=set up prod line	A, B	1	13	7
E=produce product	D	8	12	10
F=obtain orders	C, E	9	15	12

[*Note:* a=estimate under best conditions; b=estimate under worst conditions; m=most likely estimate. Expected value equals $(a + 4m + b)/6$ and variance equals $(b - a)^2/36$.]

When should work start on the project so that there is a 99% chance that the new product is in the stores by 1 December?

Comment on the assumptions made in obtaining this result.

5. * Consider the simplified list of activities that are involved in erecting a new building.

Activity	Predecessors	Duration
A=foundations	-	5
B=walls and ceilings	A	8
C=roof	B	10
D=electrical wiring	B	5
E=windows	B	4
F=siding	E	6
G=paintwork	C, F	3

By hiring additional workers, the duration of each activity can be reduced. The costs per day of reducing the duration of the activities are given in the next table.

Activity	Cost/time unit of reducing duration	Max possible reduction
A	30	2
B	15	3
C	20	1
D	40	2
E	20	2
F	30	3
G	40	1

Write down an LP that can be solved to minimise the total cost of completing the project with 20 time units.

6. * The table below lists tasks each with a deadline d_i and a penalty w_i if the deadline is exceeded. There is one processor to carry out the tasks. Find a schedule to carry out the tasks sequentially so that the total penalty incurred is minimised.

Task	1	2	3	4	5	6	7
d_i	4	2	4	2	4	4	8
w_i	90	75	60	50	35	25	15

[Each task has duration 1 time unit.]

13 Flows in transport networks

We shall now consider a classic problem in combinatorial optimisation, and one which can also appear in different guises, as the problems set below and the next section show. For more details on this problem you should read Biggs pp. 234–245.

A *basic transport network* is defined to be a digraph D such that every arc e has a positive label $c(e)$ called the *capacity of the arc*. Also, D has exactly one vertex s called the *source* such that $\deg_{\text{in}}(s) = 0$ and exactly one vertex t called the *sink* such that $\deg_{\text{out}}(t) = 0$.

A flow in a transport network D is an assignment of non-negative numbers $f(e)$ to all the arcs e of D such that

1. $0 \leq f(e) \leq c(e)$ (Feasibility Condition); and
2. for every vertex $v \neq s$ or t,

$$\sum_{e \text{ inc to } v} f(e) = \sum_{e \text{ inc from } v} f(e),$$

(Flow Conservation Condition).

The *value* of the flow f, denoted by $\mathrm{val}(f)$, is defined by

$$\sum_{e \text{ inc from } s} f(e).$$

The problem here is to find, for a given basic network, a flow with maximum value.

We first show an elementary fact, namely that the sum of flows coming out of s (that is, the value of f) is equal to the sum of flows going into t. (We shall do this rather quickly, since this is a minor result upon which you should not spend too much time. Think a bit about the following lines, otherwise just accept the result which is intuitively what one would expect. Or else, read Biggs' alternative explanation.)

Let B be the incidence matrix of the digraph D underlying the transport network. Clearly $\sum_{i=1}^{n} B_{ij} = 0$ for all $j = 1, \ldots, m$ (recall, m is the number of edges and n the number of vertices.) Now let $f(e_i)$ be denoted by f_i. Then, by the conservation law,

$$Bf = B \begin{bmatrix} f_i \\ \vdots \\ f_m \end{bmatrix} = \begin{bmatrix} \alpha \\ \vdots \\ \beta \end{bmatrix}.$$

Summing vertically and horizontally gives $0 = \alpha + \beta$, that is, $\alpha = -\beta$. Therefore the value of f is also equal to

$$\sum_{e \text{ incident to } t} f(e),$$

as claimed.

We shall now define a few concepts before presenting the algorithm to find a maximum flow. Suppose that Figure 16(a) represents a path from s to t in a transport network with an existing flow (a label like 10/2 near an edge means that it has capacity 10 and flow 2). Note that we are only showing a sub-path in the network—the flow conservation condition holds because of flows in other arcs not shown.

Observe that no arc in this path is *saturated* (that is, flow equals capacity). Therefore the flow can be increased by the *minimum* of $c(e) - f(e)$ over all arcs in the path, giving the path in Figure 16(b). Note that the flow has increased by 3 and that the conservation condition still holds (since it held with the first path. Such a path is called a *flow-augmenting path.*

But this is not the only type of flow-augmenting path possible. Consider the undirected path from s to t shown in Figure 16(c). Here we see *forward arcs* (that is, arcs in the "correct" direction s to t) which are unsaturated and *backward arcs* (that is pointing from t to s) with a non-zero flow. By increasing flows on the forward arcs and decreasing flows on the backward arcs we can increase the net value of the flow by an amount which equals the minimum amongst $c(e) - f(e)$ for forward arcs e and $f(e)$ for backward arcs e. In this case we get the path shown in Figure 16(d), and we see that the flow has

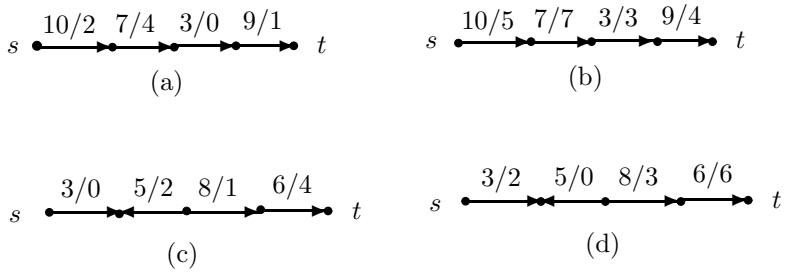


Figure 16: Flow augmenting paths

increased by 2 and the conservation condition still holds again because it held with the original path.

It turns out that the above are the only two types of flow augmenting paths possible. This will become clear when we show that the algorithm in the next section truly gives a maximum flow.

We therefore define a *flow augmenting path* to be a (not necessarily directed) path from s to t consisting of forward unsaturated arcs or backward arcs with non-zero flow.

13.1 The labelling algorithm

We shall present an algorithm which works by taking a basic network with an existing feasible flow (it could initially be the all-zero flow), finding a flow augmenting path, augmenting the value of the flow accordingly, and repeating the process over and over again until no flow augmenting path can be found, in which case the flow is maximum—this will be proved in the next section.

Before proceeding we need a few more definitions. When a flow augmenting path is found, this is called *breakthrough*. If the arc (x, y) is an unsaturated forward arc, then we denote this for short by $F(x, y)$; if $x, y)$ is a backward arc with a non-zero flow we write $B(x, y)$.

The algorithm below is based on that presented in Biggs. For better understanding we give the algorithm in two versions, the second version being a refinement of the first.

Note that the algorithm searches for a flow augmenting path using a depth first search (via a queue)—the importance of this will be explained below.

THE LABELLING ALGORITHM FOR FINDING MAXIMUM FLOWS (1ST VERSION)

1. $f(v, w) := 0$ for all arcs vw; $val(f) := 0$; {initialisation}
2. $Q := (s)$; $l_2(s) := \infty$; {initialise queue}
3. while $Q \neq 0$ do {while there are still unlabelled vertices}
   - 3.1 $x := front(Q)$;
   - 3.2 (A) if x is adjacent to a new vertex y then
     - 3.2.1 check if y is a useful new vertex, and if yes, then label y accordingly;
     - 3.2.2 check if the new vertex y is t (breakthrough), and if yes

3.2.3 then
3.2.3.1 augment f accordingly;
3.2.3.2 re-initialise Q;
3.3 (B) if x is not adjacent to a new vertex then remove it from the front of the queue
{ x has been scanned}
4. Resulting flow is maximal

THE LABELLING ALGORITHM FOR FINDING MAXIMUM FLOWS (2ND REFINED VERSION)

1. $f(v, w) := 0$ for all arcs vw; $val(f) := 0$;
2. $Q := (s)$; $l_2(s) := \infty$;
3. while $Q \neq 0$ do
3.1 $x := front(Q)$;
3.2 (A) if x is adjacent to a new vertex y then
3.2.1 if $F(x, y)$ or $B(x, y)$ then
3.2.1.1 label y; $Q := add(Q, y)$;
3.2.1.2 if $y = t$ then
3.2.1.2.1 augment f: $val(f) := val(f) + l_2(t)$
3.2.1.2.2 $Q := (s)$; $l_2(s) := \infty$; $l_1(v) :=$ nil $\forall v$;
3.3 (B) else $Q := remove(Q, x)$
4. Resulting flow is maximal

In the next section we shall show that this algorithm really finds a maximum flow. We shall do this as a by-line of proving a very important theorem in combinatorial optimisation.

We shall not show that this algorithm is a polynomial time algorithm (which it is), but we just observe here that if a depth first search were used instead of a breadth first search, then the algorithm could become inefficient, as a simple problem in the exercise sheet below shows. (In this respect, there is an error in Dolan and Aldous.)

13.2 The Maximum Flow Minimum Cut Theorem

A *cut* in a basic transport network is a set C of arcs whose removal disconnects the network into two components such that the vertices s and t are in different components. The *capacity* of the cut, denoted by $\text{cap}(C)$ is equal to the sum of the capacities of all those arc leading from the component containing s to the component containing t. A cut with minimum capacity in the network is called a *minimu8m cut*.

Now suppose one is given a basic transport network with a flow f and one finds in the network a cut C. A moment's thought will confirm that the cut forms an bottleneck which forces any flow not to exceed its capacity. In other words

$$\text{val}(f) \leq \text{cap}(C).$$

In fact, this is true for all cuts and all possible flows in the network. Therefore we have the following easy but basic rule, namely that

1. The value of any maximum flow in a basic network is less then or equal to the capacity of any minimum cut.

So now suppose we find a flow f and a cut C in the network such that $\text{val}(f) = \text{cap}(C)$. This would be a lucky break for we would be able to declare the following. First that the flow is maximum, since no flow can have value more than $\text{cap}(C)$; secondly that the cut is minimum, since no cut can have capacity less than $\text{val}(f)$; thirdly that, for this network, the value of a maximum flow coincides with the capacity of a minimum cut. We can summarise this by saying that

> *2. Whenever we find, in a basic network, a flow f and a cut C such that $val(f) = cap(C)$, then f is maximum and C is minimum.*

What is surprising is that this is not a lucky break, but it is what always happens, that is, for any maximum flow in a basic transport network there corresponds a minimum cut, and this happens precisely when the labelling algorithm terminates without breakthrough. We shall prove this result in the next theorem and the proof also shows that the labelling algorithm does work, that is, when it terminates without achieving breakthrough then the flow is really maximum and, in fact, the algorithm not only finds a maximum flow but also identifies a minimum cut [3]

Theorem 13.1 (Max-Flow-Min-Cut Theorem) *In any basic network the value of a maximum flow is equal to the capacity of a minimum cut.*

Proof We shall basically exploit Fact 2 above. Suppose that the we apply the Labelling Algorithm to the basic network D, and suppose that the algorithm eventually stops (as it must) without achieving breakthrough. Let the flow be f. Consider the search for flow augmenting paths which has stopped short of breakthrough. Let A be the set of vertices reached by the search and B the set of vertices not reached. Then s must be in A, since the search starts from s, and t must be in B, since breakthrough has not been achieved. Consider the set C of arcs joining vertices in A to and from vertices in B (note that C is a cut). Since the algorithm has stopped, then all forward arcs (leading from A to B) in C must be saturated and all backward arcs (leading from B to A) must have zero flow. Therefore the net value of the flow from s to t must be

$$\begin{aligned} \sum_{\text{arcs } e \text{ from } A \text{ to } B} f(e) - \sum_{\text{arcs } e \text{ from } B \text{ to } A} f(e) &= \\ \sum_{\text{arcs } e \text{ from } A \text{ to } B} c(e) &= \\ \text{cap}(C) & \end{aligned}$$

Therefore by Fact 2 above we conclude that f is maximum and C is minimum, which proves the theorem. □

[3] This is a very important theorem in combinatorial optimisation. OR students will appreciate that max-min theorems are very fundamental in the subject—recall, for example, the Duality Theorem of Linear Programming.

14 Sheet 4: Network Flows

1. Consider the following transport network which is not a basic transport network. It contains more than one source and more than one sink, undirected edges and capacities on the vertices(these are shown encircled; a capacity on a vertex means that the amount flowing into (or out from) the vertex cannot exceed the capacity).

   Show how such a network can be transformed into a basic network such that the solution to the latter gives a solution to the former.

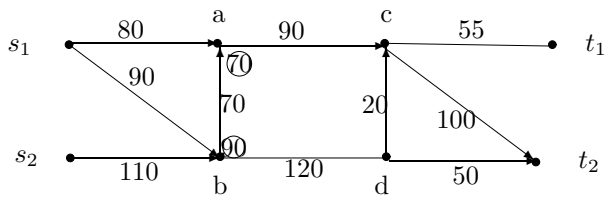


Figure 17: A non-basic transport network

2. * Seven types of packages are to be delivered by five trucks. There are three packages of each type, and the capacities of the five trucks are 6, 4, 5, 4 and 3 packages respectively. Set up a maximum flow problem that can be used to determine whether the packages can be loaded so that no truch carries two packages of the same type.

3. * A traffic engineer is tudying the traffic flow between city A and city B during the morning peak hours, 7.00am to 9.00am. During that period, commuters can travel via two possible routes: A to B directly or A to city C then to B. The engineer wishes to find the maximum number of cars which can travel from A to B during these two hours. Data are available for maximum traffic capacity over 30-minute intervals for each link: A to C has 2000 units capacity; A to B has 5000 capacity; and C to B has 3000 capacity.

   The average trip times are: A to C 60 mins; A to B 60 mins; C to B 30 mins. Vehicles that cannot enter a link because its capacity is used up will queue. For simplicity it is assumed that any number of cars can queue up at city A or city C.

   Set up a maximum flow problem that can be used to determine how many cars can travel from A to B in these two hours.

   [*Hint:* Have portions of the network represent $t = 7.00, 7.30, 8.00, 8.30, 9.00$.]

4. * During the next four months a construction firm must complete three projects. Project 1 must be completed within three months and requires eight man-months of labour. Project 2 must be completed within four months and requires 10 man-months labour. Project 3 must be completed at the end of two months and requires 12 man-months labour. Each month 8 workers are available. During a given month no more than six workers can work on a single job.

Formulate a maximum flow problem that could be used to determine whether all three projects can be completed on time.

[*Hint:* Set up a network in which the commodity flowing is man-months. If the maximum flow in the network is at least 30 $(= 8 + 10 + 12)$ then all projects can be completed on time.]

5. Formulate the maximum flow problem as a LP problem.

6. * The folowing simple example shows that the efficiency of the labelling algorithm depends on the fact that the search for flow augmenting paths is a BFS: Show that a DFS for flow augmenting paths on the network shown in Figure 18 (starting from the zero flow) could take up to $2k$ augmentation cycles (where $2k$ can be arbitrarily large compared with the number of vertices and arcs) whereas BFS takes only two cycles.

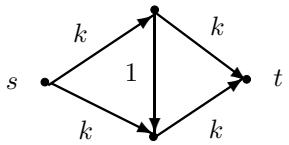


Figure 18: A basic transport network with capacities as shown

15 Matchings in Bipartite Graphs

We shall not go into too much detail regarding this problem, not that it is necessarily more difficult than some others we have considered, but because in a short one-credit course (2 ECTS) only so much can be covered. The aim of this section is to introduce to you this very important problem in combinatorial optimisation, point out connections with some other problems we have studied, and prepare the way for a heuristic algorithm which we shall consider at the end of the course.

Consider this problem. Five persons, A, B, C, D, E apply for five jobs P, Q, R, S, T. After interviews it was determined that A could do jobs P and Q, B could do P, R and T, C could do Q and S, D could do R, S and T, and E could do Q and T. The problem is: how can we assign jobs to the candidates such that every candidate is doing a job he or she is able to do, each candidate does exactly one job, and each job is done by exactly one candidate?

First we shall see how we can represent the above information succinctly by means of a graph. This is done in Figure 19(a).

Note that is is an example of a *bipartite graph*, that is a graph whose vertices can be partitioned into two parts (here $\{A, B, C, D, E\}$ and $\{P, Q, R, S, T\}$) such that any edge joins two vertices from different parts.

The problem we have is therefore one of finding a set of edges in this graph which join candidates to jobs such that *no two edges have an endvertex in common* (that is, the condition that one candidate does only one job and one job is done by only one candidate); such a set of edges is called a *matching*, and an example of a matching in this graph is shown in Figure 19(b).

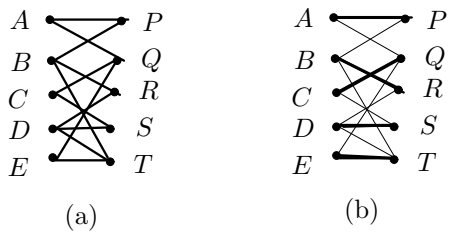


Figure 19: A bipartite graph and a perfect matching

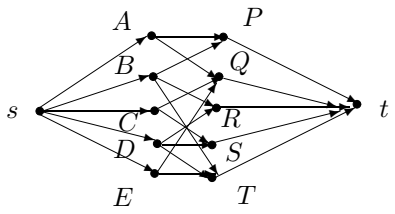


Figure 20: Transforming the matching problem into the flow problem

One way of finding a matching is to transform the problem into a network flow problem. This is done in Figure 20, where the capacity (not shown) of all arcs is 1. Finding a matching which assigns as many jobs as possible to candidates in the above problem is equivalent to finding a maximum flow in this basic network *with the proviso that all flows must be either 0 or 1* since a candidate cannot do 3/4 of one job and 1/4 of another. So, in principle, we have a way of finding the required matching, although the 0/1 condition might pose problems of efficiency (as those of you who have studied, say, integer linear programming might know).

Notice that the matching in Figure 19 covers all candidates, and it is therefore called a *perfect matching*. It could also happen that a bipartite graph does not have a perfect matching because, say, there are more candidates than jobs, or, some group of candidates between them are able to do less jobs than their number. In such a case we try to look for a *maximum matching*, that is, one which assigns jobs to the largest possible number of candidates.

So how do we find a maximum (possibly perfect) matching in a bipartite graph? When is a perfect matching not possible? Practical (that is, algorithmic) and theoretical answers to these questions exist and are given in the two theorems below. It is often the case that when there is a complete theoretical understanding of some problem there is also a corresponding efficient algorithmic solution—we shall see another example of this in th two problems which we shall consider in the next section.

We shall not give proofs to these theorems (proofs can be found in Biggs or Gibbons), not because they are more difficult than some other things which we

have covered, but for lack of time in this two-credit course. For the first theorem (the existence of an efficient algorithm for finding a matching), let us note that the most popular algorithm for doing this is one called the Hungarian Algorithm. It is basically a refinement or adaptation of the labelling algorithm for maximum flows (as the above transformation would suggest) where flow-augmenting paths are replaced by "matching-augmenting paths" which turn out to be paths in which the edges are alternately in and not in the matching (think a bit why this would be so). Therefore such paths are called *M-alternating paths*, where M is the existing matching which the algorithm is trying to augment.

The second theorem is a very important theorem closely related to min-max theorems in OR. It can be proved in a variety of ways. One as a corollary of the Duality Theorem of Linear Programming; another way is by consideration of when the Hungarian algorithm terminates, in much the same way that the Max-Flow-Min-Cut Theorem is proved by consideration of when the Labelling Algorithm terminates.

Theorem 15.1 (The Hungarian Algorithm) *There exists a polynomial time algorithm for finding a maximum matching in a bipartite graph. Moreover, if the edges of the graph are given positive weights then there is a polynomial time algorithm for finding a maximum matching with a minimum weight (where the weight of a matching is the sum of the weights of the edges in the matching).*

Theorem 15.2 (Hall's Theorem) *Let G be a bipartite graph with vertex bipartition $V_1 \cup V_2$. For any subset A of V_1 let $N(A)$ denote the set of neighbours in V_2 of all the vertices in A. Then G contains a perfect matching if and only if, for any subset $A \subseteq V_1$,*

$$|A| \leq |N(A)|.$$

In the nomenclature of candidates and jobs, this condition can be re-worded as say that any set A of workers between them know as many jobs as the number of workers in the set A.

Finally, what about matchings in non-bipartite graphs. Here the problem becomes much more difficult, but it has also been solved.

Theorem 15.3 (Edmonds' Algorithm) *Let G be a graph which is not necessarily bipartite. There exists a polynomial time algorithm for finding a maximum matching in G. Moreover, if the edges of G are given positive weights then there is a polynomial time algorithm for finding a maximum matching with a minimum weight.*

16 Eulerian trails and Hamiltonian cycles

There are two problems in graph theory which seem very similar yet are quite different. One problem asks whether the edges of a given graph can be traversed such that each edge is traversed once and only once and the journey terminates at the starting vertex. Such a journey is called an *Eulerian trail* in honour of Euler who first treated this problem mathematically and thereby obtaining the first ever result in graph theory. Note that in a trail vertices can be repeated but edges cannot. Therefore an Eulerian trail is like a sequence of vertices and edges, much like a path, which, however, can repeat vertices but not edges, and

which must cover all the edges of the graph and, moreover, end up at the initial vertex. This problem is a familiar and popular puzzle: one is given a diagram made up of lines meeting at points (the graph) and the puzzle is to determine if and how the diagram can be drawn without taking the pencil off the paper.

A graph which has an Eulerian trail is called an *Eulerian graph*.

The theoretical reason for the existence or otherwise of Eulerian trails in a graph is quite well understood and is given by the following, which is the first theorem proved in graph theory.

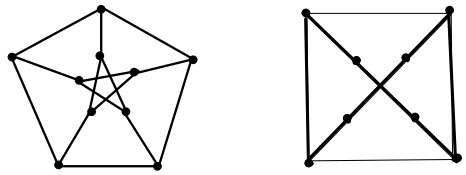
Theorem 16.1 (Euler) *A graph is Eulerian if and only if the degrees of all of its vertices are even.*

The truth of this result is quite intuitively acceptable, since every time an Eulerian trail "enters" a vertex it must "exit" again (including the initial vertex) and since this covers all the edges of the graph, then every degree must be even. This, however, is not a mathematical proof of Euler's Theorem which needs more care to write down correctly.

As often happens, when there is a simple complete theoretical result for the existence of some structure, there is also an efficient algorithm for finding the structure. Although we shall not attempt to describe the algorithm (given in Gibbons), we shall record this as a result which we shall quote below.

Theorem 16.2 *Given an Eulerian graph G there is a polynomial time algorithm for finding an Eulerian trail in G.*

We now turn to the second problem, apparently similar to the Euler trail problem, but radically different. Suppose we have a graph G and we would like to find a spanning cycle of G. The word "spanning" refers to any subgraph which includes all the vertices of G. Since we want a cycle, no vertices or edges can be repeated, therefore we are looking for a cycle which passes through every vertex once and only once. Note here that we do not intend to cover all edges of G but all vertices. A spanning cycle is called a *Hamiltonian cycle* (after the Irish mathematician Hamilton) and a graph which has a Hamiltonian cycle is called a *Hamiltonian graph*. The graph in Figure 21(a) is not Hamiltonian (try finding a Hamiltonian cycle!), while a Hamiltonian cycles is shown in the graph in Figure 21(b).



(a) (b)

Figure 21: A non-Hamiltonian and a Hamiltonian graph

Unlike the Eulerian problem, there are no known necessary and sufficient conditions for a graph to be Hamiltonian (conditons which are only necessary, as well as others which are only sufficient, are known) and this theoretical gap is matched by an algorithmic one: no polynomial time algorithm for finding Hamiltonian cycles is known, and it is not known whether such an algorithm does exist.

We are therefore faced by a problem similar in nature to the scheduling problems we mentioned above, and we therefore have to look for heuristics to tackle this problem. We shall do this when we tackle the Travelling Salesman Problem in a section below, which is an optimisation problem extending the Hamiltonian cycle problem.

16.1 An application of Eulerian trails

A strand of DNA can, for the purposes of this section, be considered to be a very long word from the alphabet $\{A, C, G, T\}$. To study the structure of such a long string a technique called Sequencing by Hybridisation (SBH) is sometimes used. Here, the string is cut up into smaller substrings which can be handled in the laboratory (typically of length 3 or 4). Then, knowing the structure of all the shorter strings, the task is to obtain the structure of the long strand with which we started.

We shall illustrate this with a small example. Suppose that the strand to be determined is

$$W = ACACGCAACTTAAA.$$

but that after SBH we all know all its subwords of length 3:

$$AAA, AAC, ACA, ACG, ACT, CAA$$

$$CAC, CGC, CTT, GCA, TAA, TTT.$$

How can we determine the unknown string from these fragments? Draw a digraph with all possible $4^{3-1} = 4^2 = 16$ vertices representing all possible strings of length $2 = 3 - 1$ (since the fragments are 3-words) from the 4-alphabet $\{A, C, G, T\}$. Now, every sub-word obtained by SBH gives rise to an arc as follows: for example, the subword ACG gives the arc from AC to CG. Let the resulting digraph be D. Any Eulerian semi-trail in D corresponds to a possible word with the given subwords, and conversely, any possible word (that is, any potential solution to the SBH problem) has a corresponding Eulerian semi-trail in D. This is illustrated for our example by the digraph shown in Figure 22 (isolated vertices, like TG, do not affect the solution and are therefore not shown in the figure). Note that the original strand we started with is not the only solution possible here. For example, the word

$$ACAACGCAAACTTAA$$

is another solution to this SBH problem.

16.2 The Travelling Salesman Problem and some heuristics

The Hamiltonian cycle problem can be easily extended to an optimisation problem. Suppose the graph G has weights on its edges, and suppose that the

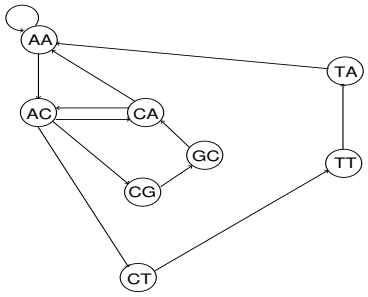


Figure 22: The digraph for the SBH example

problem now is to find a Hamiltonian cycle *with the minimum weight*, where the weight of a cycle is defined to be the sum of the weights on its edges. Clearly this problem is at least as difficult as simply finding a Hamiltonian path. It is called the Travelling Salesman Problem (TSP) because it can be described as follows. The vertices of the graph are cities, the edges are roadways connecting the cities, and the weights are distances. The travelling salesman requires an itinerary which enables him to visit each city once and only once, returning to his starting city, in such a way the the total distance travelled is minimised.

We shall henceforth assume that we are working with a complete graph G with positive lengths on the edges. This does not involve any loss of generality because if some edges (roadways) are not really there we can give them a sufficient large weight (relative to the other weights) that no decent algorithm will choose that edge (which does not really exist) for a minimum weight Hamiltonian cycle.

Note now the following points. (C_o will denote a solution to the TSP—an "optimal cycle".)

1. The length $l(C)$ of any cycle is greater than the weight of a MST T. Because $l(C) > l(C - e) \geq w(T)$, since $C - e$ (C less an edge) is a tree. In particular, $l(C_o) > w(T)$.

2. *Another lower bound.* Remove any vertex u from G. Let a be the sum of the lengths of the two shortest edges incident to u, and let b be the weight of a MST in $G - u$. Then C_o is made up of two edges incident to u plus a path joining the ends of these two edges. Let the lengths of these two parts be A and B, respectively. (Remember that a path is a special type of tree therefore its length is at least as great as b.) Therefore

$$l(C_o) = A + B \geq a + b.$$

   Hence $a + b$ is a lower bound for $l(C_o)$. This can be repeated for other vertices u and the maximum of all the lower bounds obtained can be taken.

3. *An upper bound.* The length of any cycle C is, in fact, an upper bound. All heuristics try to obtain a C such that $l(C)$ is guaranteed to be not larger than $l(C_o)$ by some known factor. We shall see this below.

We shall now describe three heuristics for the TSP.

16.2.1 Nearest neighbour heuristic

This heuristic is very simple: Start from any vertex and, at any stage, travel to the nearest neighbour which has not yet been visited.

It can be proved that any cycle C obtained by this heuristic satisfies

$$l(C_o) \leq l(C) \leq l(C_o) \times \frac{1}{2}(\lceil \log_2 n \rceil + 1)$$

where n is the number of vertices in the graph.

We would prefer a heuristic where the factor bounding $l(C)/l(C_o)$ is a constant—here it depends on n which can therefore become large for large networks.

We describe two such heuristics below.

16.2.2 Twice-around-the-MST

Here and in the next section we assume that the weights on the edges satisfy the *triangular inequality*. The steps of the algorithm are as follows:

1. Find a minimum weight spanning tree T of G.
2. Construct a DFS of T giving each vertex v a label $c(v)$ denoting the order in which vertices were visited. Let v_i denote the vertex with label $c(v) = i$, that is, the ith vertex visited, $1 \leq i \leq n$.
3. Output the cycle $v_1 v_2 \dots v_n$, using edges not in T to bypass repetitions.

Theorem 16.3 *Let C be the cycle output by the above algorithm. Then,*

$$l(C_o) \leq l(C) \leq 2l(C_o).$$

Proof Clearly $l(C_o) \leq l(C)$ since C_o is optimal. Let l' be the weight of T. We claim that $l' < l(C_o)$. Because deleting any edge from the optimal cycle gives a path with length less than $l(C_o)$; also, this path is a spanning tree, therefore its weight must be at least $w(T) = l'$.

In other words,

$$w(T) = l' \leq l(C_o - e) < l(C_o).$$

Therefore $l' < l(C_o)$.

Now, if we traverse the cycle $v_1 v_2 \dots v_n v_1$ in that order but along the edges of T, the result would be a walk W which uses each edge of T twice, that is, whose length is $2l'$.

The actual cycle C is obtained going directly to the next vertex and, if necessary, not repeating edges of T. This bypass is a shorter way of travel, by the triangular inequality. Therefore,

$$l(C) \leq 2l' \leq 2l(C_o),$$

by the above. □

16.2.3 The minimum weight matching algorithm

This heuristic uses algorithms which we have referred to earlier but not described.

1. Find a minimum weight spanning tree T of G.
2. Let V' be the set of vertices which have odd degree *in* T. Find a minimum weight perfect matching M for V' using, if necessary, edges in G not in T.
3. Add the edges of M to T. (This might create double edges if some edge in M is already in T.) Call the resulting graph G'. Clearly G' is Eulerian.
4. Find an Eulerian trail R in G'.
5. Give each vertex v in G' a label $c(v)$ denoting the order in which vertices are first visited by the Eulerian trail R. Let v_i denote the vertex with label $c(v) = i$, that is, the ith vertex visited, $1 \leq i \leq n$.
6. Output the cycle $v_1 v_2 \dots v_n$, using, if necessary, edges not in G' to bypass repetitions.

Theorem 16.4 *Let C be the cycle output by the above algorithm. Then,*

$$l(C) \leq \frac{3}{2} l(C_o).$$

Proof Gibbons Theorem 6.14. □

The minimum weight matching algorithm is the heuristic with the best known (to date) approximate guarantee for the TSP.

17 Sheet 5: The travelling salesman problem

1. The following table gives the airline distances in hundreds of miles between six cities, London, Mexico City, New York, Paris, Beijing and Tokyo.

	L	MC	BY	P	B	T
L	-	56	35	2	51	60
MC		-	21	57	78	70
NY			-	36	68	68
P				-	52	61
B					-	13

   Find a shortest spanning tree connecting these six cities.

   A traveller wishes to visit each of these cities once, returning to his starting point. Use the spanning tree obtained above to obtain an itinerary. If l_o is the length of a shortest solution to the traveller's requirements, deduce from your result the least value which l_o can have.

2. * There are four pins on a printed circuit. The distance between each pair of pins is given in the table below. We want to place three wirse between the pins in a way that connects all of them and uses the minimum amount of wire. However, if two wirse meet at the same pin a short circuit occurs. Set up a TSP problem to solve this problem.

	1	2	3	4
1	-	1	2	2
2		-	3	2.9
3			-	3
4				-

[*Hint:* Add vertex 5 with distance 0 to each of the other four pins.]

3. * A long roll of wallpaper repeats its pattern every metre. Four sheets of wallpaper must be cut from the roll. With reference to the beginning (call it point 0) of the wallpaper, the beginning and the end of each sheet are located as shown in the table below.

	Beginning (metres)	End (metres)
Sheet 1	0.3	0.7
Sheet 2	0.4	0.8
Sheet 3	0.2	0.5
Sheet 4	0.7	0.9

(Thus, for example, sheet 1 begins 0.3m from the beginning and ends 0.7m from the beginning.)

Assume we are at the beginning of the roll. Formulate a TSP to solve the problem of determining the order in which the sheets should be cut to minimise the total amount of wasted paper. Assume that a final cut is made to bring the roll back to the beginning of the pattern.

4. * A company produces four types of gasoline every day. Because of cleaning and resetting of machines, the time required to produce a batch depends on the type of gasoline last produced. The times (minutes) required to manufacture each type is shown in the table below. Set up a TSP to solve the problem of determining the order in which the batches should be produced each day (carrying on to the next day).

	Type to be next produced			
Last type produced	A	B	C	D
A	-	50	120	140
B	60	-	140	110
C	90	130	-	60
D	130	120	80	-

18 A few words on computational complexity in general

Although this is not a course in computational complexity, this issue has arisen often and it is worthwhile to say a few words in on this general problem (more details can be found in Chapter 8 of Gibbons and a less advanced treatment in Chapter 20 of Dolan and Aldous—it is recommended that the student reads this last chapter).

As we said, in general, one considers that an efficient algorithm exists for finding a solution to a problem (for example, finding a minimum weight spanning tree in a given graph) if there is a general algorithm such that the number of operations that it takes to solve the problem is a polynomial function of the size of the input (say, the number of vertices in the graph); one says that the algorithm solves the problem in polynomial time.

Of course, there are several terms in the previous sentence that need exact definitions, but we shall here take an intuitive approach and refer the student to the above texts for more exact details.

Those problems for which an efficient (polynomial-time) algorithm exists form the class denoted by P (which stands for "polynomial"). However, there are several problems for which it is not known whether there does exist an efficient algorithm. In order to tackle this question of computational intractibility, two important ideas have been developed.

Firstly, the class NP (which stands for "nondeterministic polynomial") is defined. Roughly (again we refer the reader to the above textbooks for more details) this class contains all of those problems for which, given a candidate solution, one can verify in polynomial time that it is in fact a correct solution. For example, the problem of determining whether a graph has a Hamiltonian cycle is in NP, since, given a cycle, it is easy to determine in polynomial time that it is Hamiltonian.

Now the main question in computational complexity is whether P=NP (clearly P$\subseteq$NP), and to tackle this question another important idea is introduced. Given two problems A and B, one says that A is (polynomially) reducible to B if, given an algorithm for solving B, it can be transformed in polynomial time into an algorithm for solving A. Reducibility therefore introduces a hierarchy between problems for, if A is reducible to B, then, in a sense, A cannot be more difficult to solve (computationally) than B. In particular, if there is an efficient algorithm for solving B, then there is also an efficient algorithm for solving A.

Now, the question of reducibility took on special significance by the discovery that in the class NP there are problems, called NP-complete, to which any other problem in NP is reducible (that is, the NP-complete problems are the most difficult problems in NP.

In other words, if an efficient algorithm can be found for any NP-complete problem, then all problems in NP would have an efficient algorithm to solve them, and P would be equal to NP. An example of such a problem is the TSP, which therefore gives it a special status and importance.