

Combinatorial Optimisation

Notes to accompany lectures in
Combinatorial Optimisation
Hilary Term 2007

Colin McDiarmid

1 Introduction

In order to minimise $f(x)$ over $x \in [0, 1]$ we are used to setting $f'(x) = 0$. This approach will not work for us here. Typically, we will be given a large but finite domain S , for example the set of all spanning trees in a graph, and we are to find an $x \in S$ minimising $f(x)$. The set S may be huge but will be described in some compact way, and we need to understand its structure in order to find good ways to solve the problem.

The plan for the course is that the lectures will be broken into the following 10 chapters, given with an indication of the corresponding lectures.

chap 1 Introduction (L 1)

chap 2 Minimum spanning trees (L 1-2)

chap 3 Shortest paths (L 3-4)

chap 4 Dynamic programming (L 5-6)

chap 5 Scheduling (L 7-9)

chap 6 Matchings in bipartite graphs (L 9-11)

chap 7 Assignment problem (L 11-12)

chap 8 Maximum matching in general graphs - not on course

chap 9 Maximum flows in networks (L 12-14)

chap 10 Minimum cost flows (L 14-15)

chap 11 Matroids and the greedy algorithm (L 16?)

Lectures are on Mondays 5-6pm and Fridays 9-10am in L2 in the Mathematical Institute. For part A, there will be 6 problem sheets and six classes, which will be on Thursday afternoons at 2-2.55pm in weeks 3,4,5,6,7,8. For the MSc in Applied Statistics, there will be 2 problem sheets and 2 classes, which will be on Thursday afternoons at 3-3.55pm in weeks 5 and 8. These notes, some slides and the problem sets are available from:

<http://www.stats.ox.ac.uk/~cmcd/combopt.htm>

The notes will eventually cover all of the material (apart from missing figures). They are adapted from lecture notes I wrote some time ago, still available in some libraries.

2 Minimum spanning trees

What is the cheapest way to connect up a set of cities or electrical terminals or computers using roads or wires or telephone lines?

Consider a graph $G = (V, E)$. For nodes (or vertices) u and v , let $u \sim v$ if there is a path in G between u and v . Clearly this is an equivalence relation on V . The equivalence classes are the vertex sets of the *connected components* of G . We say that G is *connected* if there is just one component, that is, if there is a path between each pair of nodes.

Let $G = (V, E)$ be a connected graph with $|V| = n$ nodes and $|E| = m$ edges, such that each edge e has a positive cost or weight or length $c(e)$. We wish to find a set E' of edges of least total cost such that the corresponding spanning subgraph $G' = (V, E')$ of G is connected. (Here *spanning* means that the subgraph must contain all the vertices of the original graph.)

A *tree* T is an acyclic connected graph. Thus we seek a minimal cost spanning tree T of G . We shall also talk of T as a set of edges.

Example

Kruskal's greedy algorithm

insert figure

```
input the connected graph  $G$ 
initialise  $T$  to be  $\emptyset$ 
while edges remain in  $G$ 
    delete a cheapest edge  $e$ 
    if  $T \cup \{e\}$  is acyclic then add  $e$  to  $T$ 
return  $T$ 
```

(Note that if two edges have the same cost they may be considered in either order.) Does this method always yield a minimal cost spanning tree?

Lemma 2.1 (see problem sheet 1)

(a) A graph with n nodes is a tree if and only if it is acyclic and has exactly $n - 1$ edges.

(b) Let T be a spanning tree in a graph $G = (V, E)$ and let $e \in E$ be an edge not in T . Then $T \cup \{e\}$ contains a unique cycle C ; and if f is any edge in C then $(T \cup \{e\}) \setminus \{f\}$ is a spanning tree.

Theorem 2.2 *Let the graph G be connected. Then Kruskal's algorithm yields a minimal cost spanning tree T .*

Proof It is easy to see that T is a spanning tree. For we never allow a cycle to be formed so T is acyclic; and if it were not connected then some edge $e \in E$ would join two connected components of T , and e should have been added to T .

We must show that T is optimal. Let $e_1, e_2, \dots, e_{n-1}$ be the ordered list of edges chosen to add to T . Let T^* be an optimal tree with $|T \cap T^*|$ as large as possible. Suppose that $T^* \neq T$. We shall deduce a contradiction, thus completing the proof.

Since $T^* \neq T$, for some j with $0 \leq j < n - 1$ each of $e_1, \dots, e_j$ is in T^* but e_{j+1} is not. By the lemma above, since e_{j+1} is not in the tree T^* , there is a unique cycle C in $T^* \cup \{e_{j+1}\}$. Let f be an edge in the cycle C not in the tree T . (Observe that f is in T^* since $f \in C \setminus \{e_{j+1}\} \subset T^*$.) The edges $e_1, \dots, e_j, f$ are all in T^* , and so they do not contain a cycle. But after choosing $e_1, \dots, e_j$ the algorithm chose e_{j+1} not f , and so $c(e_{j+1}) \leq c(f)$.

By the lemma above, $\hat{T} = (T^* \cup \{e_{j+1}\}) \setminus \{f\}$ is a spanning tree. Then $\hat{T}$ has length at most that of the optimal spanning tree T^* and so $\hat{T}$ is also optimal. But $|T \cap \hat{T}| = |T \cap T^*| + 1$, contradicting our choice of T^* . $\square$

Time? We can sort the m edges in time $O(m \log m)$, and indeed we can implement the method so that the total time is also $O(m \log m)$.

We can do this roughly as follows. Observe that the main ‘while loop’ is repeated m times. The trick is to test quickly if $T \cup \{e\}$ is acyclic. To do this we may maintain a data structure which tells us for each node which component of the forest T so far chosen contains the node. If the end nodes of a possible new edge e are in different components then the edge e is added to T and the components merged.

There are other ways of organising the basic greedy strategy. For example, in the method of Jarník (or Prim), the edge set T chosen always forms a subtree and we look for a cheapest edge to add to that tree. Thus we start from an arbitrary single node v_1 , add a cheapest edge incident with v_1 , say

the edge $\{v_1, v_2\}$, add a cheapest edge between v_1 or v_2 and the rest of the graph, and so on.

For further details on these algorithms (and others, for example that of Boruvka which is well suited to parallel computation), see for example Brassard and Bratley *Algorithmics* or Ahuja, Magnanti and Orlin, *Network Flows*.

Here is a general approach that covers both Kruskal's and Jarník's methods and others. A *cut* is the set of edges between B and $V \setminus B$ for some non-empty set $B \subset V$ of nodes. For a proof of the following lemma see the first problem set.

Lemma 2.3 *A cut and a cycle cannot meet in a single edge*

On input the connected graph G with costs on the edges, colour the edges green (in) or red (out) by applying the following rules, in any order. The 'green rule' picks edges, and the 'red rule' discards edges.

Green rule: find a cut containing no green edge, and a minimum cost uncoloured edge in the cut, and colour the edge green.

Red rule: find a cycle containing no red edge, and a maximum cost uncoloured edge in the cycle, and colour the edge red.

Prim's method consists of repeatedly applying the green rule, with the cut being the set of edges between the current tree and the rest of the graph. What about Kruskal's method?

Let the edge $e = \{u, v\}$ be uncoloured and be a cheapest such edge. If there is a path of green edges between u and v then we can colour e red; and if not, then we can colour e green – let B be the set of all nodes joined to u by a path of green edges, and let X be the corresponding cut.

This shows also that, however the rules have been applied so far, if some edge is still uncoloured then we can colour another edge; and thus any partial colouring can be completed following the rules.

Theorem 2.4 (The red-green theorem) *However the rules are applied, there must be a minimum spanning tree which contains all the green edges and none of the red ones.*

Proof Let $e_1, e_2, \dots, e_k$ be the ordered list of edges coloured green or red. Let us say that a spanning tree T 'gets it right on e_i ' if T contains e_i if it

is green and T avoids e_i if it is red. Suppose that no minimum spanning tree gets it right on all of these edges. Then for some $0 \leq j < k$, there is a minimum spanning tree T^* which gets it right on all of $e_1, \dots, e_j$, but there is no minimum spanning tree which gets it right on all of $e_1, \dots, e_{j+1}$. We shall deduce a contradiction. There are two cases, depending on the colour of e_{j+1} . The first case resembles the proof of Theorem 2.2, and the second case ‘mirrors’ it.

(a) Suppose that e_{j+1} is coloured green, but T^* does not contain it. Consider the stage at which e_{j+1} was given the colour green, and the corresponding cut X which then had no green edges. There is a unique cycle C in $T^* \cup \{e_{j+1}\}$. Note that e_{j+1} is in the cut X and the cycle C . Let $e \neq e_{j+1}$ be another such edge (there must be one by the last lemma). Observe that f is in T^* since $f \in C \setminus \{e_{j+1}\} \subset T^*$.

Could e already be coloured at this stage, that is could we have $e \in \{e_1, \dots, e_j\}$? No! For e is not a red edge in $\{e_1, \dots, e_j\}$ since $e \in T^*$; and e is not a green edge in $\{e_1, \dots, e_j\}$ since $e \in X$ (and X has no green edges at this stage). Hence $c(e_{j+1}) \leq c(e)$ by the green rule, and $\hat{T} = (T^* \cup \{e_{j+1}\}) \setminus \{e\}$ is a minimum spanning tree, which contradicts the choice of j .

(b) Suppose that e_{j+1} is coloured red, but T^* contains it. Consider the stage at which e_{j+1} was given the colour red, and the corresponding cycle C which then had no red edges. The graph $T^* \setminus \{e_{j+1}\}$ falls into two components (see problem set 1): let X be the cut consisting of the edges between them. Then e_{j+1} is in the cut X and the cycle C . Let $e \neq e_{j+1}$ be another such edge (as before, the last lemma shows that there is such an edge). Note that $e \notin T^*$ (again see problem set 1).

Could e already be coloured at this stage, that is, could we have $e \in \{e_1, \dots, e_j\}$? No! For e is not a red edge in $\{e_1, \dots, e_j\}$ since $e \in C$ (and C has no red edges at this stage); and e is not a green edge in $\{e_1, \dots, e_j\}$ since T^* contains all these and $e \notin T^*$. Hence $c(e_{j+1}) \geq c(e)$ by the red rule, and $\hat{T} = (T^* \setminus \{e_{j+1}\}) \cup \{e\}$ is a minimum spanning tree which contradicts the choice of j . (To see that $\hat{T}$ is a spanning tree, see problem set 1). $\square$

3 Shortest paths

We consider how to find a shortest path from one node to another in a network: in fact, all our methods will yield shortest paths from say node 1 to *each* other node in a network, so let us make that our task. When the arc lengths correspond to distances between cities then these lengths will naturally be non-negative, but in some applications this will not be the case (see for example section 3.3 below). Let $D = (V, A)$ be a directed graph with set $V = \{1, \dots, n\}$ of nodes, where each arc ij has length a_{ij} (which could be < 0 or ∞). If there is no arc ij then it is convenient to take a_{ij} to be ∞ . It is convenient also to assume that each $a_{ii} = 0$. We consider the three cases; when D is acyclic, when each $a_{ij} \geq 0$, and when there are no negative cycles.

3.1 Acyclic networks

An important special case is when the directed graph D has no cycles – see in particular the brief discussion at the end of this section on project scheduling. There is a natural simple method to find shortest distances in this case, and this is a good place to start.

The first step is to label the n nodes in a convenient way. We shall see later that it is possible to give the nodes distinct labels $l(1), \dots, l(n)$ from $\{1, \dots, n\}$ such that if there is an arc ij then $l(i) < l(j)$; and indeed we shall see how to do this efficiently. In the meantime let us assume that this has already been done, and each node i has label i .

Example

$$(a_{ij}) = \begin{pmatrix} 0 & 50 & 20 & & & \\ & 0 & & 40 & 10 & \\ & & 0 & 60 & 30 & \\ & & & 0 & & 30 \\ & & & & 0 & 70 \\ & & & & & 0 \end{pmatrix}$$

insert figure

In this example, we can determine the shortest distances from node 1 to node $1, 2, \dots, 6$ one after another by inspection. We find $u_1^* = 0$, $u_2^* = 50$, $u_3^* = 20$, then

$$u_4^* = \min\{u_2^* + a_{24}, u_3^* + a_{34}\} = \min\{50 + 40, 20 + 60\} = 80,$$

and so on.

In general we have the following algorithm, which has input an acyclic network in which the nodes are labelled so that if ij is an arc then $i < j$, and which outputs the shortest distances from node 1 to nodes $1, 2, \dots, n$. We need a sensible way of handling ∞ . For a real number x we let $x < \infty$ and $x + \infty = \infty$. Also, the minimum over an empty set is ∞ .

Acyclic shortest distances algorithm

```

set  $u_1 = 0$ 
for  $k = 2$  to  $n$ 
    set  $u_k = \min_{1 \leq i < k} \{u_i + a_{ik}\}$ 
return  $u_1, \dots, u_n$ 

```

In the example above we find

k	1	2	3	4	5	6
u_k	0	50	20	80	50	110

It is straightforward to see that the method is correct. Formally, we may prove by induction on k that for each $j = 1, \dots, k$ the calculated quantity u_j equals the minimum length u_j^* of a $1-j$ path, that is, a path from node 1 to node j . Clearly this is true for $k = 1$, since $u_1 = u_1^* = 0$. Now let $2 \leq k \leq n$ and suppose that the statement is true for $k - 1$. Then

$$u_k = \min_{1 \leq i < k} \{u_i + a_{ik}\} = \min_{1 \leq i < k} \{u_i^* + a_{ik}\} = u_k^*.$$

The first above equality is immediate, and the second equality follows from the induction hypothesis. Consider the third equality: we have ' $\geq$ ' since there must be a path of length $u_i^* + a_{ik}$ if this is finite; and we have ' $\leq$ ' since if u_k^* is finite and we pick a $1-k$ path with length u_k^* , it has a penultimate node i_0 where $i_0 < k$, and then $u_{i_0}^* + a_{i_0k} \leq u_k^*$. Thus the statement is true for k . It follows that it holds for the value n , as required. Also, observe that each iteration of the 'for loop' requires time $O(n)$, so the algorithm works in time $O(n^2)$.

If we wish to find all the shortest **paths** from node 1 rather than just the shortest distances, we simply keep a record of where the various minima were obtained. We do this using a 'predecessor' array $P(1), P(2), \dots, P(n)$. The extended algorithm is now as follows. It takes as input an acyclic network in which the nodes are labelled so that if ij is an arc then $i < j$ (as before),

and outputs the shortest distances from node 1 to nodes $1, 2, \dots, n$ together with the predecessor array.

Acyclic shortest paths algorithm

```

set  $u_1 = 0$  and  $P(1) = 0$ ; and for  $k = 2, \dots, n$  set  $u_k = \infty$  and  $P(k) = 0$ 
for  $k = 2$  to  $n$ 
    if  $u_i + a_{ik}$  is finite for some  $i$  in  $\{1, \dots, k-1\}$  then
        let  $j$  be a node  $i$  in  $\{1, \dots, k-1\}$  minimising  $u_i + a_{ik}$ 
        set  $u_k = u_j + a_{jk}$  and  $P[k] = j$ 
return  $u_1, \dots, u_n$  and  $P[1], \dots, P[n]$ 

```

In the example, when we determined u_4 above, we set $u_4 = u_3 + a_{34}$, and so we now also set $P[4] = 3$. In full we find

k	1	2	3	4	5	6
u_k	0	50	20	80	50	110
$P[k]$	-	1	1	3	3	4

We may find a shortest 1-6 path by tracing backwards from node 6 using the predecessor array. We find the nodes 6, $P[6] = 4$, $P[4] = 3$, $P[3] = 1$, and so a shortest 1-6 path has nodes 1,3,4,6. Indeed, we may find a ‘tree’ of shortest paths from node 1.

To determine longest paths we may proceed similarly with min replaced by max. (How do we find all shortest paths from the node labelled j if $j \neq 1$?) Now let us return to the problem of finding an appropriate labelling of the nodes.

Example continued The same acyclic digraph D with an unhelpful initial numbering might look like:

$$(a_{ij}) = \begin{pmatrix} 0 & 60 & & 30 & \\ & 0 & 30 & & \\ & & 0 & & \\ 20 & & & 0 & 30 \\ & 70 & 0 & & \\ & 40 & 10 & 0 & \end{pmatrix}$$

insert figure

How do we find a labelling $l(1), \dots, l(n)$ as required, that is such that if there is an arc ij then $l(i) < l(j)$? The *indegree* of node i is the number of arcs ji directed into i ; and a *source* is a node with indegree 0. Since node 4 is the

only source we must set $l(4) = 1$. In the digraph obtained by deleting node 4, both nodes 1 and 6 are sources, so we may for example set $l(6) = 2$ and $l(1) = 3$, and so on to complete the labelling as required. To see that we can always succeed we use:

Lemma 3.1 *In an acyclic digraph there must be at least one source.*

Proof Suppose that each node has indegree at least 1. Start at some node i_0 , pick an arc i_1i_0 , pick an arc i_2i_1 , and so on. Eventually some node must be visited twice: suppose that this happens for the first time at node i_t , and that $i_t = i_s$ where $s < t$. Then $i_t, i_{t-1}, \dots, i_{s+1}, i_s$ forms a cycle. $\square$

Let d_i denote the indegree of node i . Thus for example, we see that $d_3 = 2$ by looking down the third column of the lengths matrix (a_{ij}) . We can calculate all the indegrees in $O(n^2)$ steps.

Here we have $d_1, \dots, d_6 = 1, 2, 2, 0, 2, 1$. Observe that $d_4 = 0$ and so we can set $l(4) = 1$. Let D' denote the digraph obtained by deleting node 4 (which we have just labelled). We can calculate the indegrees in D' from $d_1, \dots, d_6$ by ignoring d_4 and subtracting 1 from d_1 and d_6 , since there are arcs from node 4 to nodes 1 and 6 (and to no other nodes). The indegrees are now $d'_1, d'_2, d'_3, d'_5, d'_6 = 0, 2, 2, 2, 0$. Both nodes 1 and 6 have value 0 (some value had to be 0 since D' is acyclic) and we may for example set $l(6) = 2$, and continue. We are led to our labelling algorithm, which takes as input an acyclic digraph and outputs a labelling $l(1), \dots, l(n)$ such that if ij is an arc then $l(i) < l(j)$.

Acyclic labelling algorithm

```

set  $U = \{1, \dots, n\}$ 
calculate the indegrees  $d_1, \dots, d_n$ 
for  $k = 1$  to  $n$ 
    let  $j$  be a node in  $U$  with  $d_j = 0$ 
    set  $l(j) = k$ , and delete  $j$  from  $U$ 
    for each node  $k \in U$  such that there is an arc  $jk$ , decrease  $d_k$  by 1
return  $l(1), \dots, l(n)$ 
```

From the discussion above, the algorithm does indeed output a labelling as required. It runs in $O(n^2)$ steps, and this shows that the entire method for finding shortest paths in an acyclic network runs in $O(n^2)$ steps. [If the digraph is given by its adjacency lists, then the entire procedure can be

implemented to run in $O(m+n)$ steps, where m is the number of arcs. Here the *adjacency lists* give, for each node j , a list $out(j)$ of the nodes k such that jk is an arc, and a list $in(j)$ of the nodes i such that ij is an arc.]

CPM/PERT

Suppose that a complicated project can be divided into a number n of activities. Each activity will take a certain known time, and cannot be started until certain other activities are completed (for example we cannot build the walls of a house until we have laid the foundations).

activity	a	b	c	d	e	f	g	h
immediate predecessors	—	—	a	a	b	b	c, e	d, f
duration (days)	4	3	2	3	4	2	4	2

We may represent this 8-activity project by the network below.

insert figure

To determine a longest 1-6 path, let $u_1 = 0$ and for $j = 2, \dots, n$ let $u_j = \max\{u_k + a_{kj}\}$, where the maximum is over all k such that $1 \leq k < j$ and (k, j) is an arc.

The numbers u_j are shown in the figure, together with arcs where the maxima are attained. We thus find that the minimum project duration is 11 days, since each node may be reached at time u_j after the start but no earlier. Also, the ‘critical path’ is b, e, g , so that if any of these activities overruns then the whole project is delayed. Ideas such as these have been developed into the critical path method (CPM) or the project evaluation and review technique (PERT) much used in the planning and control of large projects.

3.2 All arc lengths non-negative

Now we consider networks (like most road networks!) that may have cycles but where all arc lengths are non-negative. Dijkstra's shortest path algorithm proceeds as follows. It partitions the nodes into two sets F (fixed) and T (temporary). Initially $F = \{1\}$ and T contains all the other nodes, and at each stage a nearest node in T is moved into F . It also maintains a value u_k for each node k . We shall see that

- (i) for each node $k \in F$, u_k is the minimum length of a 1- k path, and
- (ii) for each node $k \in T$,

$$u_k = \min_{i \in F} \{u_i + a_{ik}\};$$

that is, given (i), that u_k is the minimum length of a 1- k path with penultimate node in F .

Example Recall that $a_{jk} = \infty$ if there is no arc jk . We indicate this in the matrix by $-$.

$$(a_{ij}) = \begin{pmatrix} 0 & 90 & 60 & 20 & - \\ - & 0 & - & 20 & 40 \\ - & 10 & 0 & - & - \\ - & 50 & 30 & 0 & 90 \\ - & - & - & - & 0 \end{pmatrix}$$

insert figure

In the example, initially T will consist of nodes 2, 3, 4, 5; and $u_2 = 90$, $u_3 = 60$, $u_4 = 20$ and $u_5 = \infty$, since these are the lengths of the corresponding arcs from node 1. At the first iteration, node 4 will be moved out of T since u_4 is the smallest of these values, and the other values u_j will be updated.

Dijkstra's shortest path algorithm

```

set  $u_1 = 0$ ,  $P[1] = 0$  and  $T = \{2, \dots, n\}$ 
for  $k = 2$  to  $n$ 
    set  $u_k = a_{1k}$  and set  $P[k] = 0$ 
    if  $a_{1k} < \infty$  then set  $P[k] = 1$ 
while  $T \neq \emptyset$ 
    let  $j$  be a node  $k$  in  $T$  minimising  $u_k$ 
    delete  $j$  from  $T$ 
    for each  $k \in T$  such that there is an arc  $jk$ 
        if  $u_j + a_{jk} < u_k$  then set  $u_k = u_j + a_{jk}$  and set  $P[k] = j$ 
return  $u_1, \dots, u_n$  and  $P[1], \dots, P[n]$ 

```

Example continued

Step	j	T	u	P
0	—	{2, 3, 4, 5}	(0, 90, 60, 20, ∞)	(0, 1, 1, 1, 0)
1	4	{2, 3, 5}	(, 70, 50, , 110)	(0, 4, 4, 1, 4)
2	3	{2, 5}	(, 60, , , 110)	(0, 3, 4, 1, 4)
3	2	{5}	(, , , , 100)	(0, 3, 4, 1, 2)

Thus the minimum length of a 1–5 path is 100; and since $P[5] = 2$, $P[2] = 3$, $P[3] = 4$ and $P[4] = 1$ we find that a shortest 1–5 path is 1,4,3,2,5.

Correct? Let us ignore the predecessor array P and show that the basic algorithm works correctly. It may then be shown that the predecessor array P allows us to trace back to find a tree of shortest paths.

Now, statements (i) and (ii) hold after initialisation, before we start the ‘repeat loop’. Assume that they hold at the start of a certain pass through the repeat loop. We claim that u_j is the minimum length of a 1– j path. It will then follow easily that the two statements hold at the end of that pass, since the step updating the values u_k for $k \in T$ correctly allows for the new node j added to F . We can then deduce that they hold throughout, and so the algorithm indeed returns the correct values.

To establish the claim, note first that, if u_j is finite, there is a 1– j path of length u_j (with penultimate node in F , by (ii)). Now consider any 1– j path Q . We must show that the length of Q is at least u_j . Let x be the first node of Q in T (perhaps $x = j$), and let w be its predecessor on Q (so w is in F , perhaps $w = 1$). Then

$$\begin{aligned}
\text{length of } Q &\geq \text{length up to } x && \text{since all } a_{ij} \geq 0 \\
&\geq u_w + a_{wx} && \text{by (i) for node } w \\
&\geq u_x && \text{by (ii) for node } x \\
&\geq u_j && \text{by choice of } j.
\end{aligned}$$

Thus every 1– j path has length at least u_j , and the claim follows. Finally note that if u_j is infinite then there can be no path from node 1 to any node in T , and again the claim is correct. $\square$

Comment A similar induction shows that $u_j \leq u_k$ for all $j \in F$ and $k \in T$, and so the nodes are indeed added to F in non-decreasing order of distance from node 1.

Time? In each pass through the repeat loop, it takes $O(n)$ time to find j and $O(n)$ time to update u_k and $P[k]$ for $k \in T$. It follows easily that the total time is $O(n^2)$. [We are often interested in sparse networks, where the number m of arcs is much less than n^2 : we can use heaps to implement the above method in time $O((m + n) \log n)$ – see for example [AMO].]

3.3 No negative cycles

Now we shall discuss the case when arcs may have negative length (or cost), though we shall insist that there are no negative cycles, that is cycles with strictly negative total length. Consider for example a problem where a ship is to get to a distant port as cheaply as possible, perhaps via other ports: the net cost of going from port A to port B may be negative if the ship can pick up a cargo at A and deliver it at B. (To what does a negative cycle correspond?) More importantly, we shall want to be able to handle negative cost arcs in order to solve minimum cost flow problems. If we do not impose some restriction such as having no negative cycles, then the problem changes nature and becomes hard, indeed NP-hard – see section 5.3 below.

add arbitrage
example?

We need a new (possibly slower) method to handle this more general case. Let $u_k^{(r)}$ be the minimum length of a $1-k$ path with at most r arcs. Thus $u_k^{(1)} = a_{1k}$ and $u_k^{(n-1)}$ equals the minimum length u_k^* of a $1-k$ path. Also, for each $r \geq 1$ we have

$$u_k^{(r+1)} = \min_j \{u_j^{(r)} + a_{jk}\} \quad (\text{for each } k = 2, \dots, n).$$

To see why this is true, let us distinguish between a *path* or *simple path* where no nodes are revisited, and a *walk* where we may do so. Since there are no negative cycles, the minimum length of a $1-k$ path is the same as for a $1-k$ walk. Observe that the LHS is at most the RHS, since each term $u_j^{(r)} + a_{jk}$ is the length of a $1-k$ walk with at most $r + 1$ arcs, and thus it is at least the length of such a path, and so it is at least the LHS. For the opposite inequality, consider a shortest $1-k$ path Q with at most $r + 1$ arcs: now Q has some penultimate node j , and so it has length $\geq u_j^{(r)} + a_{jk} \geq \text{RHS}$.

We are led to the following algorithm for finding shortest distances, due to Bellman and Ford. (For simplicity we avoid mentioning predecessor arrays here.)

Shortest paths algorithm (no negative cycles)

```
for  $k = 1$  to  $n$  set  $x_k = a_{1k}$ 
repeat  $(n - 2)$  times
  for  $k = 2$  to  $n$  set  $x_k = \min_j \{x_j + a_{jk}\}$ 
return  $x_1, \dots, x_n$ 
```

Correct? After the initialisation, for each k the value x_k is the length of some $1-k$ walk if it is finite, and this value x_k never increases (since $a_{kk} = 0$). Let $x_k^{(r)}$ denote the value of x_k just before the r th pass through the repeat loop. We **claim** that $x_k^{(r)} \leq u_k^{(r)}$. It will follow that at the end

$$u_k^* \leq x_k \leq u_k^{(n-1)} = u_k^*,$$

so that the algorithm returns the correct values $x_k = u_k^*$.

We shall prove the claim by induction on r . The result is true for $r = 1$, since $x_k^{(1)} = a_{1k} = u_k^{(1)}$. Let $r \geq 1$ and suppose that $x_k^{(r)} \leq u_k^{(r)}$ for each node $k \neq 1$. Then for each node $k \neq 1$,

$$\begin{aligned} x_k^{(r+1)} &\leq \min_j \{x_j^{(r)} + a_{jk}\} \\ &\leq \min_j \{u_j^{(r)} + a_{jk}\} \\ &= u_k^{(r+1)} \end{aligned}$$

where we used the induction hypothesis in the second inequality above. So we have $x_k^{(r+1)} \leq u_k^{(r+1)}$ for each node $k \neq 1$, as required, and our proof by induction on r is complete.

Time? Each pass through the inner ‘for loop’ takes time $O(n^2)$ so the total time is $O(n^3)$. The method may be implemented to run in time $O(nm)$, where m is the number of arcs.

3.4 All shortest paths

Suppose that we wish to find the shortest distance between each pair of nodes. We are still assuming that there are no negative cycles. Let $u_{ij}^{(k)}$ be the minimum length of an $i-j$ path such that any intermediate nodes must be contained in $\{1, \dots, k\}$. Then the minimum length of an $i-j$ (simple)

path with any intermediate nodes contained in $\{1, \dots, k\}$ and which passes through the node k is equal to $u_{ik}^{(k-1)} + u_{kj}^{(k-1)}$. Hence

$$u_{ij}^{(k)} = \min \{u_{ij}^{(k-1)}, u_{ik}^{(k-1)} + u_{kj}^{(k-1)}\}.$$

We are led to Floyd's elegant algorithm, where again we focus on distances rather than paths.

All pairs shortest paths algorithm

```

for  $i = 1$  to  $n$ 
  for  $j = 1$  to  $n$ 
    set  $x_{ij} = a_{ij}$ 
  for  $k = 1$  to  $n$ 
    for  $i = 1$  to  $n$ 
      for  $j = 1$  to  $n$ 
        set  $x_{ij} = \min \{x_{ij}, x_{ik} + x_{kj}\}$ 
return the matrix  $(x_{ij})$ 

```

The algorithm may be proved correct just as for the preceding algorithm. Let u_{ij}^* be the minimum length of an $i-j$ path, so that $u_{ij}^{(n)} = u_{ij}^*$. We initialise x_{ij} to the value $x_{ij}^{(0)} = a_{ij} = u_{ij}^{(0)}$. From then on, x_{ij} is always the length of an $i-j$ walk, so $x_{ij} \geq u_{ij}^*$; and the value x_{ij} never increases. Let $x_{ij}^{(r)}$ denote the value of x_{ij} at the end of the loop with $k = r$. We **claim** that $x_{ij}^{(r)} \leq u_{ij}^{(r)}$ (in fact $=$). It will follow that

$$u_{ij}^* \leq x_{ij}^{(n)} \leq u_{ij}^{(n)} = u_{ij}^*,$$

and thus the algorithm returns the correct values $x_{ij} = u_{ij}^*$.

We shall prove the claim by induction on r . We have seen that it is true for $r = 0$. Let $1 \leq k \leq n$, and suppose that it holds for $r = k - 1$. Then, by the induction hypothesis,

$$x_{ij}^{(k)} \leq \min \{x_{ij}^{(k-1)}, x_{ik}^{(k-1)} + x_{kj}^{(k-1)}\} \leq \min \{u_{ij}^{(k-1)}, u_{ik}^{(k-1)} + u_{kj}^{(k-1)}\} = u_{ij}^{(k)}.$$

Thus the claim holds also for $r = k$, and we are done. The algorithm takes time $O(n^3)$ and space $O(n^2)$.

We have been assuming that there are no negative cycles, but suppose now that there is a negative cycle. Then the above algorithm will detect that

there is one, and indeed we can find one from the appropriate predecessor arrays. For suppose that there is a negative cycle C with highest and second highest indexed nodes k_1 and k_2 respectively: then after the iteration with $k = k_2$ we will find $x_{k_1, k_1} < 0$. To see this, note that at the start of that iteration we will have $x_{k_1 k_2} + x_{k_2 k_1}$ at most the length of C .

3.5 A recurrence relation

Assume again that there are no negative cycles. Recall that u_k^* is the minimum distance from node 1 to node k . Thus $u_1^* = 0$ and for each $k \neq 1$ we have the recurrence

$$u_k^* = \min_{j \neq k} \{u_j^* + a_{jk}\}.$$

To prove this, consider the penultimate node j on a $1 - k$ path (where j could be 1). We see that for $k \neq 1$, u_k^* is the minimum over all $j \neq k$ of the minimum length of a $1 - k$ path with penultimate node j . Here we are using the observation that, since there are no negative cycles, the minimum length of an $i - j$ path (with no repeated nodes) equals the minimum length of an $i - j$ walk (where we allow repeated nodes).

We have implicitly used these results before, and indeed our methods could be considered to derive from the recurrence relation.

4 Dynamic Programming

The *optimality principle* states that in a sequential decision problem, an optimal policy must be optimal from any intermediate stage onwards. This often yields a recurrence equation and leads to an iterative solution method, as in our discussion of shortest paths.

4.1 Knapsack problem

There are n items, where the j th item has value p_j and weight w_j , and we have a knapsack which can take weight up to a limit b . [Here the p_j , a_j , and b are given positive integers.] What is the maximum value we can carry? The problem is to choose $x_1, \dots, x_n$ in order to

$$\max \sum_{j=1}^n p_j x_j \text{ subject to } \sum_{j=1}^n a_j x_j \leq b, \text{ each } x_j = 0 \text{ or } 1.$$

We solve the problem by breaking it into *stages*, where at stage m we consider only items $1, \dots, m$. Call the maximum value above $F_n(b)$. For each $m = 1, \dots, n$ and $c = 0, 1, \dots, b$ we define $F_m(c)$ in the natural way; that is, we let

$$F_m(c) = \max \sum_{j=1}^m p_j x_j \text{ subject to } \sum_{j=1}^m a_j x_j \leq c, \text{ each } x_j = 0 \text{ or } 1.$$

Also let $F_0(c) = 0$ for each $c = 0, \dots, b$. Then we know $F_0(c)$ for each c ; we want $F_n(b)$; and for $m = 1, \dots, n$,

$$F_m(c) = \begin{cases} F_{m-1}(c) & \text{if } c < a_m \\ \max\{F_{m-1}(c), p_m + F_{m-1}(c - a_m)\} & \text{if } c \geq a_m. \end{cases}$$

Thus we may calculate in turn $F_1(c)$ (for each c), $F_2(c), \dots, F_n(c)$. There are about nb values $F_m(c)$ to calculate, which takes $O(nb)$ time. We need only $O(b)$ space, since when calculating these values for a given m we need remember only values for $m - 1$ and m .

4.2 Resource allocation

(a) One resource type

A single ‘discrete’ resource (perhaps people) is to be allocated to n different projects with different rates of return. We are to choose $x_1, \dots, x_n$ in order to

$$\max \sum_{i=1}^n r_i(x_i) \text{ subject to } \sum_{i=1}^n x_i = \alpha, \text{ each } x_i \geq 0 \text{ and integer.}$$

Here α is a given positive integer and we assume that we have a table of values $r_i(x)$ (see for example problem set 2). Denote the maximum value above by $F_n(\alpha)$. Again we break the problem into stages: at the m th stage we consider only projects $1, \dots, m$. We calculate the values $F_m(a)$ for $m = 1, \dots, n$ and $a = 0, 1, \dots, \alpha$, where (naturally)

$$F_m(a) = \max \sum_{i=1}^m r_i(x_i) \text{ subject to } \sum_{i=1}^m x_i = a, \text{ each } x_i \geq 0 \text{ and integer.}$$

Also let $F_0(a) = 0$ for each $a = 0, 1, \dots, \alpha$. Then we know $F_0(a)$ for each a ; we want $F_n(\alpha)$; and for $m = 1, \dots, n$

$$F_m(a) = \max_{x=0, \dots, a} \{r_m(x) + F_{m-1}(a-x)\}.$$

For, if we decide to set $x_m = x$ then the immediate return is $r_m(x)$ and the best we can obtain by allocating the remaining $a-x$ to projects $1, \dots, m-1$ is $F_{m-1}(a-x)$. Thus we may calculate in turn $F_1(a)$ (for each a), $F_2(a)$, $\dots$, $F_n(a)$. There are about $n\alpha$ values to calculate, which takes time $O(n\alpha^2)$ and space $O(\alpha)$.

(b) Two resource types

Suppose now that there are two discrete resources (perhaps labour and capital, or chiefs and indians) to be allocated to n projects. We are to choose x_i, y_i to solve the following problem P

$$\max \sum_{i=1}^n r_i(x_i, y_i) \text{ subject to } \sum_{i=1}^n x_i = \alpha, \sum_{i=1}^n y_i = \beta, \text{ each } x_i, y_i \geq 0 \text{ and integer.}$$

Here α and β are given positive integers and we assume that we have a table of values $r_i(x, y)$. Denote the maximum value here by $F_n(\alpha, \beta)$.

Method 1 ‘Two-dimensional’ dynamic programming

We have the recurrence that for $m = 1, \dots, n$

$$F_m(a, b) = \max\{r_m(x, y) + F_{m-1}(a-x, b-y) : x = 0, \dots, a \text{ and } y = 0, \dots, b\}.$$

This is a ‘two-dimensional’ recurrence. To solve the problem P will take time $O(n\alpha^2\beta^2)$ (to determine about $n\alpha\beta$ values $F_m(a, b)$, where each computation takes $O(\alpha\beta)$ steps) and space $O(\alpha\beta)$. If time or space here is excessive, we may try a heuristic approach, as follows.

Method 2 Lagrangean relaxation with one-dimensional dp

We ‘move one constraint into the objective function’. Given a real number λ we consider the *Lagrangian relaxation* P^λ , which is to find non-negative integers x_i, y_i ($i = 1, \dots, n$) in order to

$$\begin{aligned} \max \quad & \sum_{i=1}^n r_i(x_i, y_i) - \lambda \sum_{i=1}^n y_i \\ \text{subject to} \quad & \sum_{i=1}^n x_i = \alpha, \text{ each } y_i \leq \beta. \end{aligned}$$

We can solve P^λ with a one-dimensional recurrence (problem set 2). The idea is then to vary λ , solving P^λ each time (quickly) to find a corresponding optimal solution x_i^λ, y_i^λ ($i = 1, \dots, n$), until $\sum_i y_i^\lambda$ equals β (or is sufficiently close to β).

Lemma 4.1 Suppose that $\sum y_i^\lambda = \beta$. Then x_i^λ, y_i^λ ($i = 1, \dots, n$) solves P .

Proof Note first that x_i^λ, y_i^λ is feasible for P . Let $\bar{x}_i, \bar{y}_i$ be *any* feasible solution for P . We must show that

$$\sum_i r_i(x_i^\lambda, y_i^\lambda) \geq \sum_i r_i(\bar{x}_i, \bar{y}_i).$$

But x_i^λ, y_i^λ is optimal for P^λ , and $\bar{x}_i, \bar{y}_i$ is feasible for P^λ , and so

$$\sum_i r_i(x_i^\lambda, y_i^\lambda) - \lambda \sum_i y_i^\lambda \geq \sum_i r_i(\bar{x}_i, \bar{y}_i) - \lambda \sum_i \bar{y}_i;$$

and the desired inequality follows. □

We may think of λ as a penalty for overusing the second resource.

Lemma 4.2 *If $\lambda_1 < \lambda_2$ then $\sum_i y_i^{\lambda_1} \geq \sum_i y_i^{\lambda_2}$.*

Proof Since $x_i^{\lambda_1}, y_i^{\lambda_1}$ is optimal for P^{λ_1} and $x_i^{\lambda_2}, y_i^{\lambda_2}$ is feasible for P^{λ_1} we have

$$\sum_i r_i(x_i^{\lambda_1}, y_i^{\lambda_1}) - \lambda_1 \sum_i y_i^{\lambda_1} \geq \sum_i r_i(x_i^{\lambda_2}, y_i^{\lambda_2}) - \lambda_1 \sum_i y_i^{\lambda_2}.$$

Similarly,

$$\sum_i r_i(x_i^{\lambda_2}, y_i^{\lambda_2}) - \lambda_2 \sum_i y_i^{\lambda_2} \geq \sum_i r_i(x_i^{\lambda_1}, y_i^{\lambda_1}) - \lambda_2 \sum_i y_i^{\lambda_1}.$$

Adding, we obtain

$$(\lambda_2 - \lambda_1) \sum_i y_i^{\lambda_1} \geq (\lambda_2 - \lambda_1) \sum_i y_i^{\lambda_2},$$

which yields the lemma. □

We are led to the following heuristic approach. Choose λ , and find a feasible solution x_i^λ, y_i^λ to P^λ .

$$\mathbf{case} \begin{cases} \sum y_i^\lambda = \beta & \text{stop, current solution is optimal for } P \\ \sum y_i^\lambda > \beta & \text{increase } \lambda, \text{ try again} \\ \sum y_i^\lambda < \beta & \text{decrease } \lambda, \text{ try again} \end{cases}$$

The time taken is $O(n\alpha(\alpha + \beta))$ per value of λ , with space $O(\alpha)$ (see the problem sets); and we hope not to have to try too many values of λ .

5 Scheduling

We shall talk of assigning jobs to machines, though we could for example be patients to doctors. Suppose that we have $m = 1$ machines for the present; n jobs J_i , $i = 1, \dots, n$; and for each job J_i a *ready time* r_i , a *processing time* p_i , and a *due date* d_i . We shall always assume here that all jobs are ready at time $r_i = 0$.

For a single machine, a *schedule* assigns to each job J_i an interval $[C_i - p_i, C_i)$ of length p_i , between its start time and its *completion time* C_i , such that the intervals for different jobs are disjoint. Often we can assume that there is no *idle time*, so that a job starts as soon as the preceding job has completed.

Given a schedule S , each job has a *flowtime* $F_i = C_i - r_i$, *lateness* $L_i = C_i - d_i$, and *tardiness* $T_i = L_i^+ (= \max\{L_i, 0\})$. We may wish to minimise the maximum flowtime $F_{\max}$; the mean flowtime $\bar{F} = \frac{1}{n} \sum_i F_i$; the maximum lateness $L_{\max}$; the number n_T of tardy (late) jobs, etc. All these are *regular* measures, that is non-decreasing functions of $(C_1, \dots, C_n)$. A good text for background reading is S. French, *Sequencing and Scheduling*.

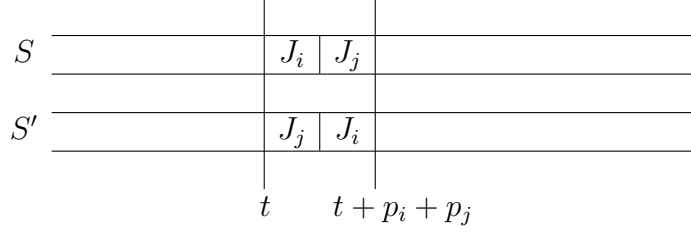
5.1 Single machine scheduling

Let us first consider how to minimise **maximum lateness**. This problem is denoted by $n/1//L_{\max}$ – n jobs, 1 machine, no extra constraints, minimise $L_{\max}$. If the jobs J_i are ordered by non-decreasing due date d_i , and all jobs are started as soon as possible, this is called an *earliest due date* (EDD) schedule.

Theorem 5.1 *For an $n/1//L_{\max}$ problem, any EDD schedule $\hat{S}$ is optimal.*

Proof We use an *interchange argument*. Recall that, given two linear orders on a set, an *inversion* is an unordered pair x, y of elements such that x precedes y in one order and y precedes x in the other.

Choose an optimal schedule S with as few inversions as possible relative to $\hat{S}$. If there are no inversions then $S = \hat{S}$ and we are done, so suppose that S has at least one inversion: we shall deduce a contradiction. Now S must have successive jobs J_i, J_j with J_j before J_i in $\hat{S}$, not necessarily immediately (exercise). Since J_j is before J_i in $\hat{S}$ we have $d_j \leq d_i$. Form S' from S by interchanging J_i, J_j , as in the following *Gantt chart* (where time increases from left to right).



```

function Moore(I)
  if each job  $J_i$  for  $i \in I$  is early
    then return I
  else
     $f \leftarrow \min\{i \in I : \text{job } J_i \text{ late}\}$ 
     $F \leftarrow \{i \in I : i \leq f\}$ 
     $m \leftarrow$  an index  $i \in F$  which maximises  $p_i$ 
    return Moore( $I \setminus \{m\}$ )

```

Note that $\text{Moore}(\emptyset) = \emptyset$.

Example

Jobs	1	2	3	4	5	6	f	m
due date	6	9	15	20	23	30		
processing time	3	4	10	10	8	6		
completion times	3	7	17				3	3
	3	7	*	17	25		5	4
	3	7	*	*	15	21		

Thus $n_T = 4$ and an optimal schedule has jobs in the order 1, 2, 5, 6, 3, 4.

Clearly each pass through the ‘if loop’ takes $O(n)$ time, so to compute $\text{Moore}(\{1, \dots, n\})$ takes $O(n^2)$ time. We now show that the method is correct.

Theorem 5.3 *For an $n/1//n_T$ problem, $\text{Moore}(\{1, \dots, n\})$ returns the indices of the early set in an optimal schedule.*

Proof We shall use induction on $|I|$ to show that $\text{Moore}(I)$ returns a corresponding ‘optimal early set’ for each $I \subseteq \{1, \dots, n\}$. Let $P(k)$ be the proposition that this holds for each such set I with $|I| = k$. Clearly $P(0)$ holds. Suppose that $0 \leq k < n$ and $P(k)$ holds. Let $I \subseteq \{1, \dots, n\}$ with $|I| = k + 1$. We must show that $\text{Moore}(I)$ returns an optimal early set for I . This is clearly true if each job in I is early, so we may suppose that this is not the case. To prove the theorem, it now suffices to prove

Claim *There is an optimal schedule for I with job J_m late.*

For then the maximum number of early jobs for I equals the maximum number of early jobs for $I \setminus \{m\}$, and by the induction hypothesis $P(k)$ we know that $\text{Moore}(I \setminus \{m\})$ gives a schedule with this number of jobs early.

Proof of claim By the last lemma, there is an optimal schedule S with all late jobs after all early jobs, and with the list E of early jobs in the original EDD order. We may assume that job J_m is early, since otherwise we are done. At least one job J_i for $i \in F$ is late under S , say job J_ℓ , where $\ell \leq f$. Let S' be the schedule obtained from S by putting J_m to the end and putting job J_ℓ in EDD order amongst $E \setminus \{J_m\}$.

	E		early		late	
S		J_m		J_ℓ		
S'			J_ℓ			J_m

Let $E' = (E \setminus \{J_m\}) \cup \{J_\ell\}$. It suffices to show that all jobs in E' are early under S' , for then S' is also optimal and has job J_m late.

Firstly, note that by the definition of f , each job J_k in E' with $k < f$ is still early (as it is no later than with the EDD order on all the jobs in I). Secondly, each job J_k in E' with $k > f$ is at least as early with S' as with S (indeed, $C'_k \leq C_k - p_m + p_\ell$, and $p_\ell \leq p_m$) and so it is still early. It remains only to consider job J_f . If $m = f$ we are done, since then $J_f \notin E'$; so suppose that $m \neq f$. Let job $J_{\hat{f}}$ be the immediate predecessor of job J_f in F (note that $|F| > 1$ since $m \neq f$, so there is a job $J_{\hat{f}}$). Then

$$C'_f \leq \sum_{i \in F \setminus \{m\}} p_i \leq \sum_{i \in F \setminus \{f\}} p_i \leq d_{\hat{f}} \leq d_f.$$

Thus job J_f is early under S' , which completes the proof. $\square$

5.2 Flowshop

Assume that we have m machines $M_1, \dots, M_m$; and each of the n jobs has a given processing time on each machine and must go through all the machines in order. Let C_i denote the completion time of job J_i on the last machine M_m . Call a criterion B *regular* if it is a non-decreasing function of $(C_1, \dots, C_n)$.

Lemma 5.4 *Given an $n/m/F/B$ problem where the criterion B is regular (and F is for flowshop), there is an optimal schedule S with the same job sequence on machines M_1 and M_2 .*

Proof Let S^* be an optimal schedule, with no gaps on M_1 . Fix the M_i order for $i \geq 2$; and let $\hat{S}$ be an optimal schedule, which minimises the number of inversions in the M_1 -order relative to the fixed M_2 -order. If this number is not zero, then $\hat{S}$ has jobs J_i, J_j successive on M_1 and reversed on M_2 (not necessarily successive).

$\hat{S}$		J_i	J_j					M_1	$\rightarrow$ time
				J_j	$\cdots$	J_i		M_2	

Interchange J_i, J_j on M_1 to find a contradiction, since no job on M_2 need be moved later. $\square$

Our main interest in this section is the problem $n/2/F/F_{\max}$, to minimise $F_{\max}$ in a two-machine flowshop. [When each ready time $r_j = 0$ as here, the maximum flowtime $F_{\max}$ is the same as the maximum completion time, also called the makespan.] By the last lemma we can restrict our attention to permutation schedules (that is, with the same order on each machine), with all jobs starting as early as possible. For each job J_i let a_i, b_i be the processing time on machine M_1, M_2 respectively.

Johnson's Algorithm

List the jobs in non-decreasing order of $\min\{a_i, b_i\}$
 set $k = 1$ and $\ell = n$
for $j = 1$ to n
 let J_i be next in the list
 if $a_i \leq b_i$ **then** put J_i in position k in the schedule (the first available slot), and add 1 to k
 else put J_i in position ℓ in the schedule (the last available slot), and subtract 1 from ℓ .

Example

i	1	2	3	4	5	6	7
a_i	6	2	4	1	7	4	7
b_i	3	9	3	8	1	5	6
$\min\{a_i, b_i\}$	3	2	3	1	1	4	6

We list the jobs in the order 4, 5, 2, 3, 1, 6, 7 and obtain the schedule

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

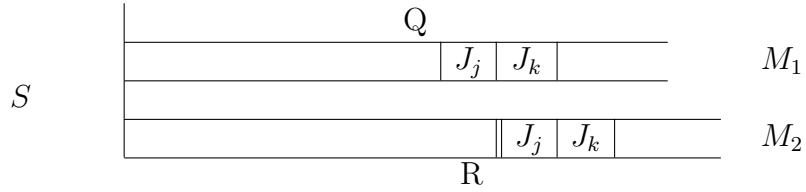
Time? $O(n \log n)$ to sort, so $O(n \log n)$ overall.

Correct? We shall use one lemma.

Lemma 5.5 (i) Consider a schedule S in which job J_j immediately precedes job J_k on both machines, where $a_k \leq b_k, a_j$. Then the schedule S' obtained by interchanging jobs J_j and J_k on both machines, and ‘closing up’ to remove idle time, has $F_{\max}(S') \leq F_{\max}(S)$.

(ii) If instead $b_j \leq a_j, b_k$ then the same conclusion holds.

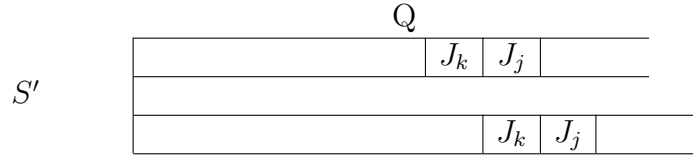
Proof Part (ii) will follow from part (i) with time reversed, so let us consider part (i).



We may assume that S has no unnecessary idle time. Let Q be the time at which M_1 starts job J_j under S , and let R be the time at which M_2 is ready to start job J_j .

Under S , M_2 starts job J_j at $\max\{Q + a_j, R\}$ and so finishes J_k at

$$C_k \geq \max\{Q + a_j, R\} + b_j + b_k.$$



Under S' , M_2 starts J_k at $\max\{Q + a_k, R\}$; so M_2 starts J_j at

$$\max\{\max\{Q + a_k, R\} + b_k, Q + a_k + a_j\};$$

and so

$$C'_j = \max\{\max\{Q + a_k, R\} + b_k + b_j, Q + a_k + a_j + b_j\}.$$

The former term here is at most C_k , since $a_k \leq a_j$; and the latter term also is at most C_k since $a_k \leq b_k$. So $C'_j \leq C_k$, and hence $F_{\max}(S') \leq F_{\max}(S)$. $\square$

Theorem 5.6 *Johnson's algorithm yields an optimal schedule.*

Proof Let A be the set of jobs J_i such that $a_i \leq b_i$, and let B be the set of other jobs. Let S be any permutation schedule in which

- the jobs in A precede the jobs in B ;
- the jobs in A are ordered by non-decreasing a_j ; and
- the jobs in B are ordered by non-increasing b_j .

Clearly Johnson's rule yields such a schedule. We shall prove that S is optimal.

Let $\hat{S}$ be an optimal permutation schedule with as few as possible inversions relative to the order S . If some job $J_j \in B$ precedes some job $J_k \in A$ in $\hat{S}$, then there is a successive pair of such jobs. Now $a_k \leq b_k$ and $a_j > b_j$. If $a_k \leq a_j$ then $a_k \leq b_k, a_j$; and if not then $b_j \leq a_j, b_k$. In either case, we see from the last lemma that interchanging jobs J_j and J_k leads to an optimal schedule with fewer inversions than $\hat{S}$, contradicting our choice of $\hat{S}$. Thus in $\hat{S}$ each job in A precedes each job in B . Now we may use the same approach (that is, the last lemma and an interchange argument) to show that $\hat{S} = S$, and so S is optimal, as required. $\square$

5.3 A hard problem

Let us return to single machine scheduling, and consider the problem $n/1//\bar{T}$. Here we have n jobs J_j , each with a processing time p_j and due date d_j ; and we are to minimise the average tardiness $\bar{T}$ (see problem set 3 for an example).

We have crossed the divide: no good method is known for this problem, and indeed none is expected since the problem is *NP-hard*. (An efficient algorithm to solve it would yield efficient algorithms for each problem in a wide class *NP* of problems. This class includes such notoriously difficult problems as the travelling salesman problem and graph colouring.) The best we can do is perhaps to use dynamic programming.

Let us consider the tardiness sum $\sum T_j$ rather than $\bar{T}$. For $Q \subseteq \{1, \dots, n\}$ let $F(Q)$ be the minimum tardiness sum for scheduling the jobs J_i for $i \in Q$ (and ignoring any other jobs). Then $F(\emptyset) = 0$, and for $Q \neq \emptyset$ we have

$$F(Q) = \min_{i \in Q} \{F(Q \setminus \{i\}) + (\sum_{j \in Q} p_j - d_i)^+\},$$

since some job J_i for $i \in Q$ must go last, thus completing at time $\sum_{j \in Q} p_j$. We may compute $F(Q)$ for each set Q of size 1 then for each of size 2 and so on, until we reach $F(\{1, \dots, n\})$.

Time? In particular, is this faster than running through all $n!$ orders for the jobs? Yes! For, there are 2^n values $F(Q)$ to compute, and each takes $O(n)$ steps, so the total time is $O(n2^n)$. Is there a way to solve this problem in $o(2^n)$ time?

6 Matchings

6.1 Introduction

A *matching* in a graph G is a subset M of the edges such that no two edges in M share the same node.

insert figure

A node of G is *covered* if some edge in M is incident with it and otherwise is *exposed*. A matching is *maximum* if it is of maximal size. It is *perfect* (or *complete*) if no nodes are exposed. A path (without repeated vertices) is *M -alternating* if the edges are alternately in M and out of M eg 2,4,5,6 above. An M -alternating path is *M -augmenting* if both end vertices are exposed eg 1,2,3,5,4,6. We may identify a path with its set of edges.

Lemma 6.1 *Let M be a matching and P an M -augmenting path. Then $M' = M \triangle P$ is a matching of cardinality $|M| + 1$.*

Proof Let e, e' be distinct edges in M' . We must show that they have no node in common. There are three cases (i) $e, e' \in M \setminus P$, trivial; (ii) $e, e' \in P \setminus M$, ok since P has no repeated vertices; and (iii) $e \in M \setminus P, e' \in P \setminus M$, ok since e has no nodes in common with P . $\square$

Theorem 6.2 (*Berge 1957*) *A matching M in a graph G is maximum if and only if there is no M -augmenting path.*

Proof ($\Rightarrow$) done by Lemma 6.1.

($\Leftarrow$) Suppose M is not maximum. Let M' be a matching with $|M'| > |M|$. Consider the edges in $M \triangle M'$. In the corresponding subgraph H of G each vertex can be incident with at most one edge of M and one of M' . Hence

each component of H is either an even cycle or a path, with edges alternately in M and M' . But H has more M' edges than M edges. Thus some path component of H has more M' edges than M edges, and it then forms an M -augmenting path. $\square$

So to find a maximum matching we start with say $M = \phi$ and repeatedly search for augmenting paths. But how? Before we find out how to go let us note a way to stop. A *cover* in G is a set K of nodes such that every edge of G has at least one end in K .

Lemma 6.3 *Let M be a matching and K a cover in G . Then $|M| \leq |K|$.*

Proof Each edge in M must contain at least one node in K . $\square$

If above we have $|M| = |K|$ then of course M is a maximum matching (and K is a minimum cover, and K consists of exactly one end node of each edge in M). Note that in the 5-cycle C_5 we cannot have equality. However, for bipartite graphs (see below) we shall have equality and we can base an algorithm for maximum matchings on Lemmas 6.1 and 6.3.

A graph $G = (V, E)$ is *bipartite* if its nodes may be partitioned as $V = S \cup T$ such that each edge $e \in E$ has one end node in S and one in T . It may be shown that a graph is bipartite if and only if it contains no odd cycle (exercise). Further, given a graph G on n nodes, in $O(n^2)$ time we may test if it is bipartite and if so output a corresponding partition $S \cup T$. We shall often use the notation $G = (S, T, E)$.

insert figure
of C_5 ?

6.2 Bipartite maximum cardinality matching algorithm

The ‘Hungarian algorithm’ may be described as follows.

Start

We start with a bipartite graph $G = (S, T, E)$, and a matching M , perhaps empty. No nodes are labelled or scanned.

Tree building

give the label 0 to each exposed node in S

while there is no breakthrough and there is a labelled unscanned node

 find a labelled unscanned node i

if the node $i \in S$ **then** scan it as follows:

 for each edge $\{i, j\} \notin M$ incident to node i ,

 give node j the label ‘ i ’ if node j is not already labelled,

 and declare ‘breakthrough’ if node j is exposed.

if the node $i \in T$ **then** scan it as follows:

 find the unique edge $\{i, j\} \in M$ incident to node i ,

 and give node j the label ‘ i ’.

Augment

if there has just been a breakthrough **then**

 start at the breakthrough node j and backtrack using the labels to find an M -augmenting path, augment M , remove all labels, and return to the step *Tree building*.

Hungarian labelling

return the matching M [a maximum matching]

 and the set K consisting of the unlabelled nodes in S

 together with the labelled nodes in T [a minimum cover].

Example

Theorem 6.4 *The algorithm returns a matching M and a cover K with $|M| = |K|$ in $O(m^2n)$ steps, where $|S| = m$, $|T| = n$, $m \leq n$.*

Proof Let us say that we enter a new *stage* each time we enter the ‘tree-building’ step. During each stage, we scan nodes $i \in S$ at most m times, and each performance takes $O(n)$ steps. This then takes $O(mn)$ steps. Considering scanning a node $i \in T$ similarly, we see that during each stage scanning takes $O(mn)$ steps. Also, by lemma 6.1 the augmentation step correctly makes the matching one larger, within time $O(mn)$.

Now there are $O(m)$ stages, each takes $O(mn)$ steps, and throughout we have a proper matching. Hence after $O(m^2n)$ steps we must stop, with a matching M . The set K is a cover (of edges by nodes) for there can be no edge between a labelled node $i \in S$ and an unlabelled node $j \in T$ (there is no such edge in M , since then i would have received its label from j ; and no such edge out of M since then i would give a label to j). Further K consists of exactly one node from each edge in M , and so $|M| = |K|$.

Now note that K contains only covered nodes, since all exposed nodes in S are labelled and we have not labelled any exposed node in T . Also, for each edge ij in M where $i \in S$ and $j \in T$, either both or neither of its end nodes i, j are labelled, since if j is labelled then it gives a label to i and if j is not labelled then i cannot be labelled. If both i and j are labelled then i goes in K , and if neither is labelled then j goes in K . $\square$

6.3 König’s theorem and Hall’s theorem

Let G be a bipartite graph. We observed that $|M| \leq |K|$ for any matching M and cover K . We described an algorithm which yields a matching M and cover K with $|M| = |K|$. We have thus also proved

Theorem 6.5 (*König’s theorem*) *In a bipartite graph, $\max |M| = \min |K|$.*

This result was originally phrased as follows. In a 0–1 matrix the maximum number of 1’s no two on the same line (that is, row or column) equals the minimum number of lines to cover all 1’s. It is easy to deduce Hall’s ‘marriage theorem’ from König’s theorem (or vice versa). We suppose that some pairs of boys and girls know each other.

Theorem 6.6 (*Hall’s theorem*) *We can marry all the boys to girls they know iff for each k , each set of k boys between them know at least k girls.*

Proof The condition is clearly necessary. Suppose now that it holds. Form the bipartite graph $G = (B, G, E)$ where B, G are the sets of boys, girls respectively, and $\{b, g\}$ is an edge if boy b and girl g know each other. We must show that there is a matching in G of size $|B|$.

Let K be a cover. By König's theorem it suffices to show that $|K| \geq |B|$.

insert figure

All girls known to boys in $B \setminus K$ must be in $G \cap K$. So by the condition we have $|B \setminus K| \leq |G \cap K|$. Hence

$$|K| = |B \cap K| + |G \cap K| \geq |B \cap K| + |B \setminus K| = |B|.$$

□

7 Assignment Problem

Suppose that we are to assign n jobs to n machines, one job per machine (or say n pilots to n planes). Job i assigned to machine j incurs cost $c_{ij} \geq 0$, and we seek an assignment with least total cost. In other words, we seek a minimum cost perfect matching in the corresponding complete bipartite graph, with nodes $R_1, \dots, R_n$ and $C_1, \dots, C_n$ and for each i, j an edge $R_i C_j$ joining R_i and C_j with cost c_{ij} .

Our algorithm for the assignment problem is best understood in the framework of LP (linear programming) duality. However, let us ignore LP for now and return to that later.

Call real numbers $\mathbf{u} = (u_1, \dots, u_n)$ and $\mathbf{v} = (v_1, \dots, v_n)$ *dual-feasible* if the reduced cost $\bar{c}_{ij} = c_{ij} - u_i - v_j \geq 0$ for each $i, j = 1, \dots, n$. Given dual feasible $\mathbf{u}, \mathbf{v}$ the *equality graph* $G_{\mathbf{u}, \mathbf{v}}$ is the bipartite graph with nodes $R_1, \dots, R_n$ and $C_1, \dots, C_n$ and an edge joining R_i and C_j whenever the reduced cost $\bar{c}_{ij} = 0$.

Lemma 7.1 *Let $\mathbf{u} = (u_1, \dots, u_n)$ and $\mathbf{v} = (v_1, \dots, v_n)$ be dual-feasible, and let M^* be a perfect matching in the equality graph $G_{\mathbf{u}, \mathbf{v}}$. Then M^* is an optimal assignment.*

Proof For any perfect matching M in the complete bipartite graph,

$$\text{cost of } M = \sum_{R_i C_j \in M} c_{ij} \geq \sum_{R_i C_j \in M} (u_i + v_j) = \sum_i u_i + \sum_j v_j$$

with equality for M^* , so that

$$\text{cost of } M \geq \sum_i u_i + \sum_j v_j = \text{cost of } M^*.$$

□

Our approach to solving the assignment problem is to maintain throughout a dual feasible solution and a matching in the corresponding equality graph, and to seek a perfect matching. The algorithm may be described as follows. The input is an $n \times n$ matrix of costs $c_{ij} \geq 0$. Our main tool is the bipartite maximum cardinality matching algorithm, with the step ‘Hungarian labelling’ replaced by a new step ‘Update dual solution’.

Hungarian method for the assignment problem

find dual feasible u_i, v_j (perhaps all 0), construct the equality graph $G_{u,v}$, and find a matching M in $G_{u,v}$ (perhaps the empty matching).

while M is not perfect

tree-building with M in equality graph

while not breakthrough

update dual solution

continue tree building

augment M

return M [an optimal matching] and u_i, v_j [which are dual optimal].

The step *update dual solution* is as follows.

Compute $\delta = \min\{\bar{c}_{ij} : i \in S \cap L, j \in T \cap \bar{L}\}$, where $\bar{c}_{ij}$ is the reduced cost $c_{ij} - u_i - v_j$, L is the set of labelled vertices, and $\bar{L}$ is the rest. For each $i \in S \cap L$ increase u_i by δ ; and for each $j \in T \cap L$ decrease v_j by δ . Construct the new equality graph $G_{\mathbf{u}, \mathbf{v}}$.

Example

		$v_1 = 0$	$v_2 = 0$	$v_3 = 0$	$v_4 = 1$
	$u_1 = 2$	5	2	3	4
(c_{ij})	$u_2 = 4$	7	8	4	5
	$u_3 = 3$	6	3	5	6
	$u_4 = 2$	2	2	3	5

$(\bar{c}_{ij})$	$\begin{pmatrix} 3 & 0 & 1 & 1 \\ 3 & 4 & 0 & 0 \\ 3 & 0 & 2 & 2 \\ 0 & 0 & 1 & 2 \end{pmatrix}$
------------------	--

Observe that $K = (S \cap \bar{L}) \cup (T \cap L) = \{R_2, R_4, C_2\}$ is a cover. Also,

$$\begin{aligned} \delta &= \min\{\bar{c}_{ij} : i \in S \cap L, j \in T \cap \bar{L}\} \\ &= \min\{\bar{c}_{ij} \text{ over uncovered elements of matrix} \\ &= 1. \end{aligned}$$

Increase u_1 and u_3 by δ , and decrease v_2 by δ .

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

The boldface above entries correspond to the edges in the new equality graph $G_{u,v}$. Observe that each old matching edge and each old tree edge is still in $G_{u,v}$.

We now obtain the optimal assignment $R_1, C_4; R_2, C_3; R_3, C_2; R_4, C_1$ with reduced cost 0. The original cost is $2 + 3 + 4 + 4 = 13$. Note that we have also an optimal dual solution ($\sum u_i + \sum v_j = 3 + 4 + 4 + 2 + 0 + (-1) + 0 + 1 = 13$).

Theorem 7.2 *The Hungarian method yields an optimal assignment $\mathbf{x}^*$ in $O(n^4)$ steps (together with an optimal solution to (D) which verifies that $\mathbf{x}^*$ is an optimal solution to (P)).*

Proof Let us say that we are *happy* at some point if we have dual feasible $\mathbf{u}, \mathbf{v}$ and a matching M in the equality graph $G_{\mathbf{u}, \mathbf{v}}$. Certainly we start happy. We want to show that we stay happy throughout, and clearly we need only check that the dual update step keeps us happy.

Suppose then that we enter this step happy. Note that M is not perfect. Certainly δ is well defined (that is, $S \cap L \neq \emptyset$ and $T \cap \bar{L} \neq \emptyset$) and $\delta > 0$

(since the set K was a cover, that is since there are no edges in the equality graph between $S \cap L$ and $T \cap \bar{L}$). Our changes to u_i, v_j change $\bar{c}_{ij} = c_{ij} - u_i - v_j$ as follows:

$$\begin{array}{ll}
\text{decrease } \bar{c}_{ij} \text{ by } \delta & \text{if } i \in S \cap L, j \in T \cap \bar{L} \\
\text{fix } \bar{c}_{ij} & \text{if } i \in S \cap L, j \in T \cap L \\
& \text{or if } i \in S \cap \bar{L}, j \in T \cap \bar{L} \\
\text{increase } \bar{c}_{ij} \text{ by } \delta & \text{if } i \in S \cap \bar{L}, j \in T \cap L.
\end{array}$$

Hence still each $\bar{c}_{ij} \geq 0$. Also, for each edge $\{i, j\}$ in M either each end is labelled or each end is unlabelled, and so $\bar{c}_{ij}$ stays fixed at 0. Hence we shall leave the dual update step happy, as required. Further each old tree edge (along which the last labelling took place) is still in the new equality graph, and we shall be able to label at least one new node in T .

We now know that we stay happy throughout. Also between augmentations we can go round the loop including the dual update step at most n times. Further each loop takes $O(n^2)$ steps. It follows easily that between augmentations we take $O(n^3)$ steps. But there can be at most n augmentations, and so after $O(n^4)$ steps we must stop. Now we are still happy but we have a perfect matching, which corresponds to a feasible solution $\mathbf{x}^*$ to (P). Now Lemma 7.1 completes the proof. $\square$

With more care we can organise matters so that between augmentations we take $O(n^2)$ steps, and we thus obtain an implementation taking $O(n^3)$ steps (see for example [AMO] or [PS]).

LP duality

Now let us return to set the method above explicitly in the framework of LP duality. Consider the LP

$$\min \sum_{i,j} c_{ij} x_{ij}$$

subject to

$$(P) \quad \begin{aligned} \sum_j x_{ij} &= 1 & (i = 1, \dots, n) \\ \sum_i x_{ij} &= 1 & (j = 1, \dots, n) \\ x_{ij} &\geq 0 & (i, j = 1, \dots, n). \end{aligned}$$

The integral feasible solutions are the possible assignments or the perfect matchings in the corresponding bipartite graph with costs on the edges. [The LP is highly degenerate, which can lead to awkwardness with simplex-type methods.]

Now suppose more generally that we are given vectors $\mathbf{c} \in \mathbf{R}^n$ and $\mathbf{b} \in \mathbf{R}^m$ and an $m \times n$ real matrix A . Consider a primal linear programme (P) in the following convenient form. Choose $\mathbf{x} \in \mathbf{R}^n$ in order to

$$(P) \quad \begin{aligned} \min \quad & \mathbf{c}'\mathbf{x} \\ \text{subject to} \quad & A\mathbf{x} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}. \end{aligned}$$

The associated dual programme (D) is then to choose $\mathbf{y} \in \mathbf{R}^m$ in order to

$$(D) \quad \begin{aligned} \max \quad & \mathbf{y}'\mathbf{b} \\ \text{subject to} \quad & \mathbf{y}'A \leq \mathbf{c}'. \end{aligned}$$

The fundamental duality theorem then states that if both (P) and (D) have feasible solutions then both have optimal solutions $\mathbf{x}^*, \mathbf{y}^*$ and the two values are equal (i.e. $\mathbf{c}'\mathbf{x}^* = (\mathbf{y}^*)'\mathbf{b}$). From this follows the

Theorem 7.3 *(Complementary Slackness Theorem)*

Let $\mathbf{x}$ be feasible for (P) and $\mathbf{y}$ for (D). Then both these feasible solutions are optimal if and only if

$$(x_j > 0 \Rightarrow (\mathbf{y}'A)_j = \mathbf{c}_j \quad \text{for each } j = 1, \dots, n).$$

To form the dual of the assignment LP introduce dual variables $u_1, \dots, u_n$ and $v_1, \dots, v_n$. Observe that the column of the A matrix corresponding to

the primal variable x_{ij} is $\begin{pmatrix} \mathbf{e}_i \\ \mathbf{e}_j \end{pmatrix}$ where $\mathbf{e}_k$ denotes the k th unit vector in $\mathbf{R}^n$. Thus we see that the dual is

$$(D) \quad \begin{array}{ll} \max & \sum_i u_i + \sum_j v_j \\ \text{subject to} & u_i + v_j \leq c_{ij} \quad \text{for each } i, j = 1, \dots, n. \end{array}$$

This explains the earlier use of the words ‘dual-feasible’. The complementary slackness theorem now gives the following extension of Lemma 7.1.

Lemma 7.4 *Let x_{ij} be feasible for (P) and u_i, v_j for (D). Then both are optimal if and only if*

$$(x_{ij} > 0 \Rightarrow u_i + v_j = c_{ij} \quad \text{for each } i, j = 1, \dots, n).$$

We may see that our approach to solving the assignment problem is to maintain throughout both **dual** feasibility and the complementary slackness (CS) conditions, and to seek primal feasibility. (In the simplex method and the transportation algorithm, we maintain primal feasibility and CS and seek dual feasibility.) We end up with an primal optimal solution which is integral, which is thus an optimal assignment.

8 Maximum matching in general graphs

Given a graph $G = (V, E)$, let $\kappa_o(G)$ denote the number of odd components (that is, components with an odd number of nodes). Also, for $S \subseteq V$ let $G \setminus S$ denote the graph obtained from G by deleting the nodes in S and any edges incident with them.

Lemma 8.1 *Let M be a matching in a graph $G = (V, E)$ and let $S \subset V$. Then*

$$|M| \leq \frac{1}{2}(|V| + |S| - \kappa_o(G \setminus S)).$$

Proof Let the components of $G \setminus S$ be $G_i = (V_i, E_i)$ for $i = 1, \dots, k$. The number of edges in M with at least one end node in S is at most $|S|$. For each component G_i , $|M \cap E_i| \leq \lfloor |V_i|/2 \rfloor$. Hence

$$\begin{aligned} |M| &\leq |S| + \sum_{i=1}^k \lfloor |V_i|/2 \rfloor \\ &= |S| + \frac{1}{2}|V \setminus S| - \frac{1}{2}\kappa_o(G \setminus S) \\ &= \frac{1}{2}(|V| + |S| - \kappa_o(G \setminus S)). \end{aligned}$$

□

Theorem 8.2 (*Tutte, Berge*) *For any graph $G = (V, E)$, the maximum cardinality of a matching M is given by*

$$\max_M |M| = \min_{S \subseteq V} \frac{1}{2}(|V| + |S| - \kappa_o(G \setminus S)).$$

We shall sketch an algorithm that will prove this theorem. Given a matching M , the algorithm will either find an augmenting path, or find a set S that yields equality in the theorem. We need two preliminary notions, shrinking and blossoms.

To ‘shrink’ a set S of nodes in a graph $G = (V, E)$ means to contract the nodes in the set down to a single node and then remove any loops and identify any multiple edges. Formally, to *shrink* or *contract* a set $S \subseteq V$ in G means to start with $G \setminus S$, take a new node v_S , and join v_S to each node in $V \setminus S$ which was joined in G to a node in S . We denote the graph formed by G/S .

Given a matching M in G , there is a natural corresponding set M/S of edges in G/S . If at most one edge in M has an end in S and an end not in S , then M/S is a matching in G/S . Also given a matching M in a graph G , a *blossom* consists of an odd number $2k + 1$ of nodes forming a cycle which contains k edges of M . The *base* is the node which is not covered by one of these edges. Thus the nodes a,b,c,d,e above form a blossom, with base b.

Lemma 8.3 *Let S be a blossom in G with respect to the matching M . Any augmenting path p in the contracted graph G/S with respect to the matching M/S may be ‘extended’ to an augmenting path in G with respect to M .*

Proof If the path p does not go through v_S , then it is already an augmenting path in G with respect to M .

Suppose that v_S is an internal node in p . Then we may write p as p_1, u, v_S, w, p_2 where the edge uv_S is in M/S and the path p_2 may be trivial.

Now $ub \in M$ where b is the base of S , and $vw \in E \setminus M$ for some node $v \in S$ (where perhaps $v = b$).

There is a path p_3 within the blossom S from b to v which is alternating and ends with an edge in M (or is the trivial path at b if $v = b$). Then p_1, u, p_3, w, p_2 is an augmenting path in G with respect to M .

Finally, if v_s is an end node of the path p , then the base node b in G must be exposed, and we may again use the above argument (check!). $\square$

In Edmonds’ matching algorithm we start with a matching M , colour some nodes blue (B) or red (R), assign predecessors to coloured nodes, and shrink blossoms. When we stop, we either use the predecessors to find an augmenting path, or we find a set S giving equality in Theorem 8.2.

Edmonds' matching algorithm

Start

Start with a matching M in the graph G .

Root

If there is an uncoloured exposed node, colour one blue, call it the root, and go to the step *Breakthrough test*. If there is no such node then stop.

Breakthrough test

If a blue node is adjacent to an uncoloured exposed node v then stop.

Grow

If a blue node u is adjacent to an uncoloured node v which is matched with the uncoloured node w , then colour v red and direct the edge from v to u , and colour w blue and direct the edge from w to v ; and return to the step *Breakthrough test*. Otherwise continue to the step *Blossom*.

Observe that for each edge in M , either both ends are uncoloured or one is blue and one is red.

Blossom

If no two blue nodes are adjacent then return to the step *Root*. Otherwise find two adjacent blue nodes. Trace back along the predecessors from each to reach the (same) root. Let b be the first common node on these paths. The segments of the paths up to b form a blossom S .

Shrink the blossom S and colour the shrunken node v_S blue, keeping all predecessor relations (and record the shrinking so that it can be

undone later). (Observe that the difference between the numbers of blue and red nodes does not change.) Return to the step *Breakthrough test*.

This completes the description of the algorithm.

If the procedure stops in the step *Breakthrough test*, then we have a path of the form $U, B, R, B, \dots, R, B$ from an uncoloured (U) exposed node to the root (B), and where some of the blue nodes may be shrunken sets of nodes from G . This is an augmenting path in the final shrunken graph $\tilde{G}$ with respect to the matching in that graph. Now expand the shrunken nodes one at a time in reverse order of shrinking. By the last lemma, our path may be ‘extended’ to an augmenting path in the original graph G . So we may augment the matching and start again.

Now suppose that the procedure stops in the step *Root*, with a final shrunken graph $\tilde{G}$. Note that all the red nodes are from the original graph G (that is, not shrunken). Look at the difference between the numbers of blue and red nodes. The only operation that can change this is in the step *Root*, when we pick an exposed node as a root and colour it blue. Thus

$$\# \text{blue nodes} - \# \text{red nodes} = \# \text{roots} = \# \text{exposed nodes in } G = |V| - 2|M|.$$

In $\tilde{G}$, if we delete the set S of red nodes, the blue nodes which remain are isolated (not adjacent to any node), and each will unshrink to an odd component of $G \setminus S$. Any uncoloured nodes must form even components

of $G \setminus S$, since the edges in M yield perfect matchings for them. Thus $\# \text{blue nodes} = \kappa_o(G \setminus S)$. Hence

$$\kappa_o(G \setminus S) - |S| = |V| - 2|M|.$$

The procedure sketched above yields an algorithm which finds a maximum matching M (and a set S which proves that M is maximum) in time $O(n^4)$, where $n = |V|$. To see this, let us say that we enter a new *stage* at the start or when we complete an augmentation. During a stage, we can shrink blossoms at most n times. Each shrinking takes time $O(n^2)$, and the colouring steps between shrinkings also take time $O(n^2)$. Thus during a stage, the colouring and shrinking take time $O(n^3)$. But there are at most n stages, and so our algorithm will in total take time $O(n^4)$. The method and analysis can be refined so that we obtain a time bound of $O(n^{2.5})$ – for references see [AMO].

9 Maximum flows in networks

Many Operational Research problems can be cast in the form of finding a maximum flow in a network. These are ‘structured’ LP’s that can be solved very quickly. We shall describe the classical augmenting path method.

Travellers’ example A group of travellers wants to fly from San Francisco to New York City at short notice. The seat availabilities on possible flights are as shown.

For example we may send 4 along the ‘middle’ route, 1 along the north route and 3 along the south route.

So 8 people can make it, but can we do better?

By a *network* we mean a directed graph $G = (V, E)$ with n nodes in which each arc ij has a capacity $u_{ij} \geq 0$, and in which we have a specified *source* node s and *sink* node t . The remaining $n - 2$ nodes are called *intermediate*. It is convenient to assume that no arc enters the source s and no arc leaves the sink t .

A *flow* $\mathbf{x}$ through the network is an assignment of numbers x_{ij} to the arcs ij satisfying the *conservation equations*

$$\sum_{i \in V} x_{ij} = \sum_{k \in V} x_{jk} \quad (1)$$

for each intermediate node j . In the first sum it is to be understood that we sum over all nodes i such that ij is an arc; and similarly for other sums here.

Sum the equations (1) over all the intermediate nodes to obtain

$$\sum_{j \in V} x_{sj} = \sum_{j \in V} x_{jt}. \quad (2)$$

The common value here is called the *volume* $v(\mathbf{x})$ of the flow $\mathbf{x}$. A flow $\mathbf{x}$ is *feasible* if $0 \leq x_a \leq u_a$ for each arc a . A feasible flow of maximal volume is called a *maximum flow* through the network.

A *cut* is any set C of nodes with $s \in C$ and $t \notin C$. Suppose that C is a cut and $\mathbf{x}$ is a flow. Sum the equations (1) over all $j \in C' = C \setminus \{s\}$ to obtain

$$\sum_{i \in V} \sum_{j \in C'} x_{ij} = \sum_{j \in C'} \sum_{k \in V} x_{jk}.$$

Now add the equation $v(\mathbf{x}) = \sum_k x_{sk}$, and we find

$$v(\mathbf{x}) + \sum_{i \in V} \sum_{j \in C} x_{ij} = \sum_{j \in C} \sum_{k \in V} x_{jk},$$

so that

$$v(\mathbf{x}) = \sum_{j \in C} \sum_{k \notin C} x_{jk} - \sum_{i \notin C} \sum_{j \in C} x_{ij}. \quad (3)$$

This says that the volume $v(\mathbf{x})$ equals the net export out of C .

The *capacity* $\text{cap}(C)$ is defined to be $\sum_{j \in C} \sum_{k \notin C} u_{jk}$. From (3) we must have

$$v(\mathbf{x}) \leq \text{cap}(C), \quad (4)$$

for any feasible flow $\mathbf{x}$ and cut C . A *minimum cut* is a cut of minimal capacity.

Example again

We can send one extra passenger along the south route as far as A, who could then continue to NYC if we route one less passenger $D \rightarrow A$; and we could send this passenger along $D \rightarrow C \rightarrow \text{NYC}$. The modifications are represented by

leading to the flow indicated of volume 9

The shaded vertices give a cut of capacity 9 so we have found a maximum flow (and a minimum cut).

Consider a feasible flow $\mathbf{x}$ in the network. Let us allow a *path* now to ignore the direction of arcs. An $\mathbf{x}$ -alterable path is a path such that $x_a < u_a$ for each forward arc a and $x_a > 0$ for each reverse arc a . An $\mathbf{x}$ -alterable path from s to t is an $\mathbf{x}$ -augmenting path.

Given an $\mathbf{x}$ -augmenting path let δ be the minimum value of $c_a - x_a$ over all forward arcs a and of x_a over all reverse arcs a . Thus $\delta > 0$. (In the example $\delta = 1$.) Increase x_a by δ for each forward arc a and decrease x_a by δ for each reverse arc. It is easy to see that we have created a new feasible flow $\hat{\mathbf{x}}$ with volume $v(\hat{\mathbf{x}}) = v(\mathbf{x}) + \delta$. This is the key observation. We are now ready for the famous theorem of Ford and Fulkerson (1956) - though see also the discussion in Chvátal *Linear Programming*.

Theorem 9.1 (*The max-flow min-cut theorem*)

For any network, the maximum value of a flow equals the minimum capacity of a cut.

Proof By the inequality (4) it suffices to show that there is a feasible flow $\mathbf{x}$ and a cut C with $v(\mathbf{x}) = \text{cap}(C)$.

Note that the set of feasible flows is a closed and bounded set in R^E , and so it is compact. Also $v(\mathbf{x}) = \sum_j x_{sj}$ is a continuous function of $\mathbf{x}$. Hence there exists a maximum flow $\hat{\mathbf{x}}$.

Let C be the set of nodes i such that there is an $\hat{\mathbf{x}}$ -alterable path from s to i . (Include s in C .) By our discussion above, there cannot be an $\hat{\mathbf{x}}$ -augmenting path, and so $t \notin C$. Also for each arc jk with $j \in C$, $k \notin C$ we must have $\hat{x}_{jk} = u_{jk}$ (for if $j \in C$ and $\hat{x}_{jk} < u_{jk}$ then also $k \in C$); and similarly for each arc ij with $i \notin C$, $j \in C$ we have $\hat{x}_{ij} = 0$. Hence by (3)

$$v(\hat{\mathbf{x}}) = \sum_{j \in C} \sum_{k \notin C} \hat{x}_{jk} - \sum_{i \notin C} \sum_{j \in C} \hat{x}_{ij} = \sum_{j \in C} \sum_{k \notin C} u_{jk} = \text{cap}(C).$$

□

In order to design an algorithm to find a maximum flow we want an efficient way to look for augmenting paths. Here is one such method.

Maximum flow algorithm

Start

We start with a feasible flow in the network, perhaps the zero flow. No nodes are labelled or scanned.

Search

give the source s the label 0.

while there is no breakthrough and there is a labelled unscanned node

 find a labelled unscanned node i

 scan node i as follows:

 for each arc ij such that $x_{ij} < u_{ij}$ and node j is not labelled,
 give node j the label i .

 for each arc ki such that $x_{ki} > 0$ and node k is not labelled,
 give node k the label i .

 if the sink is labelled then declare ‘breakthrough’.

Augment

if there has just been a breakthrough **then**

 use the labels to find the corresponding $\mathbf{x}$ -augmenting path,

 augment the flow by the corresponding $\delta > 0$,

 remove all labels and go to the step *Search*.

Finish

return the current flow $\mathbf{x}$ [which is a maximum flow]

 and the set C of labelled nodes [which is a minimum cut].

Example again

It is easily seen from the proof of the max-flow min-cut theorem that the method yields a sequence of feasible flows of strictly increasing volume, and that if it stops then indeed the current flow is a maximum flow and the set C is a minimum cut. Also, if all capacities are integral then clearly each δ is integral. So we obtain

Theorem 9.2 *Suppose that each capacity u_a is integral. Then the above method yields a maximum flow $\mathbf{x}^*$ and a minimum cut C in at most $v(\mathbf{x}^*) (= \text{cap } C)$ iterations, and further the flow $\mathbf{x}^*$ is integral.*

Corollary 9.3 *(Integrality theorem) If each capacity is integral then there is a maximum flow which is integral.*

It is not hard to see that we may implement the method so that each iteration takes $O(n^2)$ time: in fact we can ensure that the time taken is $O(m)$ where $m = |E|$ (see for example Chvátal). This is good, but the bound $v(\mathbf{x}^*)$ on the number of iterations is not good practically or theoretically (we want a bound which at most a polynomial in the number of bits in the input).

Example

Here each arc capacity is M (big) except that one arc has capacity 1 as shown. With a bad choice of next node to scan we may repeatedly have $\delta = 1$ and take M iterations (although the input has only $O(\log M)$ bits).

Also what happens if the capacities are not all integers? If all are rationals with least common denominator d then we must obtain a maximum flow in at most $dv(\mathbf{x}^*)$ iterations (just clear fractions to see this). But, it is possible to construct examples with some irrational capacities such that bad choices of augmenting paths lead to flows with values not converging to the maximum value (see Chvátal).

In a major paper in 1972 Edmonds and Karp showed that if we organise the scanning using ‘first-labelled first-scanned’ then we require at most $mn/2$ iterations, yielding a good bound of $O(m^2n)$ steps overall. There are now better methods based on “blocking flows” that run in $O(n^3)$ steps.

Infinite capacities It is convenient sometimes to allow some capacities u_{ij} to be infinite. This presents no difficulties. Observe first that if each cut has infinite capacity then there is an $s - t$ directed path in the subnetwork containing only the infinite capacity arcs, and so there are (integral) flows of arbitrarily large volume. (To see that there is an $s - t$ directed path as claimed, consider the set C of nodes reachable from s just using infinite capacity arcs: if $t \notin C$ we would have a cut with finite capacity.) Of more interest is the case when some cut has finite capacity.

Theorem 9.4 *Assume that some cut has finite capacity.*

(a) There is a maximum flow and its volume equals the minimum capacity of a cut.

(b) If all finite capacities are integral then there is an integral maximum flow.

To deduce this from our earlier results just replace each infinite capacity by a suitably large integer M , say M greater than the sum of the finite capacities. Similarly, our maximum flow algorithm is easily adapted to allow for infinite capacities. We end this section with a model involving infinite capacities.

A strip mining model

Given a reliable survey of ore deposits, we are to design the open pit to be created. The survey typically divides the volume under consideration into blocks with sides of about 30 feet (depending for example on the distance between bore holes) and there may be several thousands of these.

With each block i we associate the marginal profit w_i resulting when i is added to the pit: to obtain w_i we subtract the cost of excavating i (not the accumulated cost of excavating i along with blocks above it) from the profit from the ore in it. Thus the total profit from a pit P equals $\sum_{i \in P} w_i$.

Not every set P of blocks is a feasible pit. If we want block i in a pit P then we must also include some other blocks j in P , usually forming a cone above

i : let us write $i \rightarrow j$ for each of these. Now a set P of blocks constitutes a feasible pit if and only if

$$i \in P, i \rightarrow j \Rightarrow j \in P.$$

We are led to the following problem.

Problem Let $G = (V, E)$ be a directed graph, with a value w_i for each node $i \in V$. Call a set C of nodes *closed* if $i \in C, ij \in E \Rightarrow j \in C$. The objective is to find a closed set C maximising $\sum_{i \in C} w_i$.

We may convert this into a network flow problem as follows. Let $A = \{i \in V : w_i \geq 0\}$, $B = V \setminus A$. Extend G by adding a source s and sink t , with arcs si for each $i \in A$ and jt for each $j \in B$.

Define non-negative upper bounds on the new arcs by setting $u_{si} = w_i$ for each $i \in A$ and $u_{jt} = -w_j$ for each $j \in B$. Let the upper bounds on the original arcs of G be infinite. We now have a network N .

Observe that a set C of nodes in G is closed if and only if the cut $C \cup \{s\}$ in N has finite capacity; and then

$$\begin{aligned} \text{cap}(C \cup \{s\}) &= \sum_{i \in A \setminus C} u_{si} + \sum_{j \in B \cap C} u_{jt} \\ &= \sum_{i \in A \setminus C} w_i - \sum_{j \in B \cap C} w_j \\ &= \sum_{i \in A} w_i - \sum_{j \in C} w_j. \end{aligned}$$

Since $\sum_{i \in A} w_i$ is a constant, minimising $\text{cap}(C \cup \{s\})$ is equivalent to maximising $\sum_{i \in C} w_i$. So a minimum cut in N is an optimal closed set in the original problem. Thus we can solve our strip mining problem by using the maximum flow algorithm to obtain a minimum cut.

10 Minimum cost flows in networks

Suppose that, as in the section on maximum flows, we have a network N consisting of a directed graph with a specified source node s and sink node t , in which each arc a has a capacity $u_a \geq 0$. Suppose now that each arc a also has a cost $c_a \geq 0$. Given a positive value ν , our task is to find a feasible flow of volume ν which has minimum cost (if there is a flow of this volume). Here the *cost* of a flow $\mathbf{x}$ is $\sum_a c_a x_a$, where the sum is over all the arcs a .

We shall describe two algorithms to solve such problems, both pleasingly straightforward. For many applications and for further solution methods see [AMO]. Our algorithms use the work we have done on finding shortest paths in networks which may have negative cost arcs.

Let us assume for simplicity that the original network never contains both the arcs ij and ji (note that we could subdivide arcs to achieve this). It is convenient to introduce the ‘residual network’ for a given feasible flow $\mathbf{x}$. We replace each arc ij of the original network by two arcs ij and ji : the *forward* arc ij has (residual) capacity $u_{ij} - x_{ij}$ and cost c_{ij} , and the *reverse* arc ji has (residual) capacity x_{ij} and cost $-c_{ij}$. The *residual network* $N(\mathbf{x})$ contains just the arcs with strictly positive residual capacity.

If we ignore the costs, we may see that in the maximum flow problem, an $\mathbf{x}$ -augmenting path is just a source-sink path in the residual network $N(\mathbf{x})$, and the maximum flow algorithm repeatedly searches for such paths in the residual network. It is convenient now to work explicitly with residual networks. The following result is crucial to our methods.

Theorem 10.1 (*Negative cycle optimality condition*) *A feasible flow $\mathbf{x}$ is optimal if and only if there is no negative cycle in the residual network $N(\mathbf{x})$.*

To prove this result we use one lemma. Recall that a *circulation* is a flow $\mathbf{x}$ such that there is conservation at **all** nodes. (The values x_a can be any positive or negative numbers.)

Lemma 10.2 *Any non-negative integral circulation can be decomposed as a sum of unit flows around directed cycles.*

Proof - see problem sets. □

Proof of Theorem 10.1 If there is a negative cost (directed) cycle in $N(\mathbf{x})$, then $\mathbf{x}$ is not optimal, since we can augment flow around the cycle and thus

strictly decrease the cost, without changing the volume of the flow. Now suppose that there is a feasible flow $\hat{\mathbf{x}}$ (in the original network N) with cost strictly less than that of $\mathbf{x}$. We must show that there is a negative cycle in $N(\mathbf{x})$. That will complete the proof of the theorem.

Since $\hat{\mathbf{x}}$ and $\mathbf{x}$ have the same volume, the difference $\hat{\mathbf{x}} - \mathbf{x}$ is a circulation, with strictly negative cost. We shall replace each negative flow in an arc by a positive flow in the reversed arc, to obtain a non-negative circulation $\mathbf{y}$ in $N(\mathbf{x})$. If $\hat{x}_{ij} - x_{ij} > 0$ then $x_{ij} < u_{ij}$ and so the arc ij is in $N(\mathbf{x})$: simply set $y_{ij} = \hat{x}_{ij} - x_{ij}$. If $\hat{x}_{ij} - x_{ij} < 0$ then $x_{ij} > 0$ and so the arc ji is in $N(\mathbf{x})$: set $y_{ji} = -(\hat{x}_{ij} - x_{ij}) > 0$. Note that these changes do not upset flow conservation or change cost. Thus $\mathbf{y}$ is a non-negative integral circulation in $N(\mathbf{x})$, with cost equal to that of $\hat{\mathbf{x}} - \mathbf{x}$, which is strictly negative.

By the last lemma, $\mathbf{y}$ can be decomposed as a sum of unit flows around directed cycles, and the cost of the circulation will be the sum of the costs of these flows. Hence there must be a negative cycle in $N(\mathbf{x})$. $\square$

We now describe our first method for solving the minimum cost flow problem. Let us assume that ν and the capacities are integral, and there exists a feasible flow of volume ν (and thus an integral such flow).

Build-up algorithm

start with the zero flow

while the flow volume $< \nu$

find a minimum cost source-sink path P in the residual network

[which has no negative cycles], and

augment the flow along the path P until either the flow

volume is ν or we reach the residual capacity of an arc in P

return the final flow $\mathbf{x}$

Theorem 10.3 *The algorithm build-up returns an integral optimal flow (that is, an optimal flow which also is integral). Each flow constructed is an integral optimal flow of that volume, and so the residual networks have no negative cycles.*

Proof Suppose that $\mathbf{x}$ is an integral minimum cost feasible flow of some volume $\alpha < \nu$. Let P be a minimum cost source-sink path in the residual network $N(\mathbf{x})$ (there must be one), and let the positive integer δ be at most the residual capacity of any arc in P . Let $\hat{\mathbf{x}}$ be the flow obtained by augmenting along P by δ . We claim that $\hat{\mathbf{x}}$ is an integral minimum cost feasible

flow of volume $\alpha + \delta$. This claim together with Theorem 10.1 will prove the theorem, since the flow volume will strictly increase at each iteration, and so we must terminate, with an integral optimal flow.

It remains then to prove the claim. By Theorem 10.1 we must show that there is no negative cycle in the residual network $N(\hat{\mathbf{x}})$. Suppose to the contrary that there is such a cycle C . Form a list Q of the arcs in P or C , where we list an arc twice if it appears in both P and C . If an arc ij and its reverse ji both appear in the list then delete both.

All the arcs in the list Q are in $N(\mathbf{x})$, since if an arc ij of C is in $N(\hat{\mathbf{x}})$ and not in $N(\mathbf{x})$ then the reverse arc ji must be in P , and so both would be deleted from Q . Note that deleting the arcs ij and ji from Q does not change its cost (that is, the sum of the costs of the arcs in Q with multiplicities). Thus the cost of Q equals the cost of P plus the cost of C , which is strictly less than the cost of P . But the arcs in Q (with multiplicities) can be partitioned into a source-sink path P' and possibly some cycles C' in $N(\mathbf{x})$: to see this, consider adding the arc ts and use the last lemma. Since any such cycle C' has non-negative cost (by Theorem 10.1), the path P' must have cost strictly less than that of P , a contradiction. $\square$

When we discussed finding shortest paths in networks, we described efficient methods to find shortest paths if there are no negative cycles. We can use any such method in the build-up algorithm above. [It is in fact possible to maintain the arc costs non-negative throughout – see [AMO] – and then we

can use for example Dijkstra's method.]

We also saw how to determine if there is a negative cycle and if so to find one. We now describe a second method for solving the minimum cost flow problem, which would be useful in particular if arc costs change. Let us assume again that ν and the capacities are integral, and there is a feasible (integral) flow of volume ν .

Cycle-cancelling algorithm

find a feasible integral flow $\mathbf{x}$ of volume ν

while there is a negative cycle in the residual network

 find such a cycle and augment the flow around it until we

 reach the residual capacity of an arc in the cycle

return the final value of $\mathbf{x}$

Theorem 10.4 *The cycle-cancelling algorithm returns an integral optimal flow.*

Proof The algorithm deals only with feasible integral flows of volume ν , and the cost of the flow strictly decreases at each iteration. Hence the algorithm must stop, and by Theorem 10.1 it will return an optimal solution. $\square$

11 Matroids and the greedy algorithm

An *hereditary system* is a pair $(E, \mathcal{I})$, where E is a finite set and $\mathcal{I}$ is a non-empty collection of subsets of E , called *independent sets*, such that if $A \in \mathcal{I}$ and $B \subseteq A$ then $B \in \mathcal{I}$. Thus such a system always contains the empty set. For example, let G be a graph with edge set E . Let $\mathcal{F}$ be the collection of edge-sets of forests, and let $\mathcal{M}$ be the collection of matchings. Then $(E, \mathcal{F})$ and $(E, \mathcal{M})$ are hereditary systems of interest to us. Given non-negative weights $w(e)$ for $e \in E$, we may seek an independent set I of maximum total weight $w(I) = \sum_{i \in I} w(i)$.

Greedy algorithm

start with $I = \emptyset$

while $E \neq \emptyset$

 find an element $e \in E$ of maximum weight

 remove e from E

 if $I \cup \{e\} \in \mathcal{I}$ then add e to I

return I

For example, the greedy algorithm yields a maximum weight independent set when applied to the forests $(E, \mathcal{F})$, but not necessarily when applied to the matchings $(E, \mathcal{M})$. A *matroid* is an hereditary system $\mathbf{M} = (E, \mathcal{I})$ such that

- (1) if $I, J \in \mathcal{I}$ and $|I| < |J|$ then there exists $j \in J \setminus I$ such that $I \cup \{j\} \in \mathcal{I}$,
or equivalently
- (2) for each $A \subseteq E$, all maximal independent subsets of A have the same cardinality (called the *rank* of A).

Let us prove that these two conditions are equivalent. Suppose first that $\mathbf{M}$ satisfies the ‘extension’ property (1). If B and C are independent subsets of A with B maximal, then by (1) we must have $|B| \geq |C|$. It follows that $\mathbf{M}$ must satisfy the ‘rank’ property (2). Conversely, suppose that $\mathbf{M}$ satisfies (2). If I and J are independent and $|I| < |J|$, then I is not a maximal independent subset of $A = I \cup J$. It follows that $\mathbf{M}$ satisfies (1).

Some examples

Uniform matroids. Let $\mathcal{I}$ consist of all subsets of E of cardinality at most k .

Cycle matroids. Let G be a graph with edge set E , and let $\mathcal{F}$ be the collection of edge-sets of forests. Then $(E, \mathcal{F})$ is a matroid, the *cycle* matroid of G , a *graphic* matroid.

Matching matroids. Let $G = (V, E)$ be a graph. Then the hereditary system $(E, \mathcal{M})$ is not a matroid in general. But let $\mathcal{I}$ be the collection of sets of nodes covered by matchings. Then $(V, \mathcal{I})$ is a matroid, the *matching* matroid of G .

Vector matroids. Let E be the set of columns of a matrix, with elements in some field, and let $\mathcal{I}$ consist of the linearly independent sets of columns. Then $(E, \mathcal{I})$ is a matroid. It is called a *vector* (or *matric* or *representable*) matroid.

For many further examples of matroids, and for an excellent introduction to matroid theory, see [O].

Theorem 11.1 *Let $\mathbf{M} = (E, \mathcal{I})$ be an hereditary system. Then the greedy algorithm GA returns an optimal solution for all choices of non-negative weights $w(e)$ if and only if $\mathbf{M}$ is a matroid.*

Proof Suppose first that $\mathbf{M}$ is a matroid, and that we are given non-negative weights $w(e)$ for $e \in E$. We shall show that GA returns a maximum weight independent set I , that is an optimal solution. Note first that $|I| = r$, the rank of E . Suppose then that GA returns the set $I = \{e_1, e_2, \dots, e_r\}$, where the elements were added to I in the order $e_1, e_2, \dots, e_r$. Let $J = \{f_1, f_2, \dots, f_r\}$ be any maximal independent set, where $w(f_1) \geq w(f_2) \geq \dots \geq w(f_r)$. We shall show that $w(e_k) \geq w(f_k)$ for each $k = 1, \dots, r$. This certainly will imply that $w(I) \geq w(J)$, and show that GA does indeed yield an optimal solution.

Let $1 \leq k \leq r$ and let $I_{k-1} = \{e_1, \dots, e_{k-1}\}$ and $J_k = \{f_1, \dots, f_k\}$. Then $I_{k-1} \cup \{f\} \in \mathcal{I}$ for some $f \in J_k \setminus I_{k-1}$. Thus when GA has just formed I_{k-1} the element f is still acceptable, and hence the next element e_k added by GA must satisfy $w(e_k) \geq w(f)$. But $w(f) \geq w(f_k)$, and so $w(e_k) \geq w(f_k)$, as required.

Conversely, suppose now that GA always returns an optimal solution. In order to show that $\mathbf{M}$ is a matroid, let $I, J \in \mathcal{I}$ with $k = |I| < |J|$. Consider weights given as follows: $w(e) = k + 2$ for $e \in I$, $w(e) = k + 1$ for $e \in J \setminus I$, and $w(e) = 0$ otherwise. Then GA first picks the set I . But

$$w(I) = k(k + 2) < (k + 1)^2 \leq w(J).$$

Hence GA must continue and add to I at least one element $j \in J \setminus I$, and then we have $I \cup \{j\} \in \mathcal{I}$. Thus $\mathbf{M}$ is indeed a matroid. $\square$

References

- [AMO] R.K. Ahuja, T.L. Magnanti and J.B. Orlin, *Network Flows*, Prentice Hall, 1993.
- [BB] G. Brassard and P. Bratley, *Algorithmics*, Prentice-Hall, 1988.
- [C] V. Chvátal, *Linear Programming*, Freeman, 1983.
- [CLR] T.H. Cormen, C.E. Leiserson and R.L. Rivest, *Introduction to Algorithms*, MIT Press, 1990.
- [CCPS] W.J. Cook, W.H. Cunningham, W.R. Pulleyblank and A. Schrijver, *Combinatorial Optimization*, Wiley, 1998.
- [GLS] M. Grötschel, L. Lovász and A. Schrijver, *Geometric Algorithms and Combinatorial Optimization*, Springer-Verlag, 1988.
- [NW] G. Nemhauser and L.A. Wolsey, *Integer and Combinatorial Optimization*, Wiley, 1988.
- [O] J.G. Oxley, *Matroid Theory*, Oxford University Press, 1992.
- [PS] C.H. Papadimitriou and K. Steiglitz, *Combinatorial Optimisation: Algorithms and Complexity*, Prentice Hall, 1982.
- [W] R.J. Wilson, *Introduction to Graph Theory*, 4th ed., Longman, 1996.