

Combinatorial Optimisation

Notes to accompany lectures in
Combinatorial Optimisation
MSc in Applied Statistics

Hilary Term 2010

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 chapters, given with an indication of the corresponding lectures.

chap 0 What is Combinatorial Optimisation?

chap 1 Minimum spanning trees (L 1-2)

chap 2 Shortest paths (L 3-4)

chap 3 Dynamic programming (L 5-6)

chap 4 Scheduling (L 6-7)

chap 5 Matchings in bipartite graphs (L 8-9)

chap 6 Assignment problem (L 9-10)

chap 7 Flows in networks (L 11-12)

Lectures are on Mondays and Wednesdays at 10am in weeks 1-2 and 4-7. The aim is to have problem classes in weeks 5 and 8. These notes, some slides and the problem sets will be available from my departmental webpage

http://www.stats.ox.ac.uk/people/academic_staff/colin_mcdiarmid

under ‘Teaching’. The notes will eventually cover all of the material (apart from missing figures - come to lectures!). They include some material (for example on maximum matchings in general graphs and on matroids, and some non-examinable proofs) not on the course: the examinable course corresponds to the lectures.

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 a connected graph which is *acyclic* – that is, contains no cycles. Thus we seek a minimum cost spanning tree T of G . We shall also talk of T as a set of edges. We write $T \cup e$ and $T - e$ for $T \cup \{e\}$ and $T \setminus \{e\}$ respectively.

Example

Kruskal's greedy algorithm

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

(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.

insert figure

(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 $(T \cup e) - f$ is a spanning tree.

insert figure

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 $f \in C - e_{j+1} \subseteq T^*$.) 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, $\hat{T} = (T^* \cup e_{j+1}) - f$ 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; that is, if the end nodes of e are in different components of the current forest T . 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.

example

For further details on these algorithms (and others, for example that of Boruvka which is well suited to parallel computation), see for example 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 X of edges between B and $V \setminus B$ for some non-empty set $B \subset V$ of nodes. A cut X and a cycle C cannot meet in a single edge, that is $|X \cap C| \neq 1$ (exercise).

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 2.3 (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.*

The proof of this theorem is not on the course this year and so the rest of this section may be ignored.

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 $f \neq e_{j+1}$ be another such edge (there must be one since a cut and a cycle cannot meet in a single edge). Observe that f is in T^* since $f \in C - e_{j+1} \subseteq T^*$.

Could f already be coloured at this stage, that is could we have $f \in \{e_1, \dots, e_j\}$? No! For f is not a red edge in $\{e_1, \dots, e_j\}$ since $f \in T^*$; and f is not a green edge in $\{e_1, \dots, e_j\}$ since $f \in X$ (and X has no green edges at this stage). Hence $c(e_{j+1}) \leq c(f)$ by the green rule, and $\hat{T} = (T^* \cup e_{j+1}) - f$ 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 $f \neq e_{j+1}$ be another such edge (as before, there must be such an edge). Note that $f \notin T^*$.

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

3 Shortest paths

How can we find a shortest path from one node to another in a network? All our methods will in fact 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

We can determine the shortest distances u_k^* from node 1 to node k for $k = 1, 2, \dots, 6$ one after another. 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 ∞ . (Alternatively, we could assume that there is a 1- k path for each node k , for example by adding a ‘long’ arc $1k$ if it is not there already.)

Acyclic shortest distances algorithm

```

set  $u_1 = 0$  and  $u_k = \infty$  for  $k = 2, \dots, n$ 
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: let us accept this. 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. (How do we find all shortest paths from a node $j \neq 1$?)

To determine longest paths we may proceed similarly with min replaced by max. 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 relevant 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  $i \in U$  such that there is an arc  $ji$ , decrease  $d_i$  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 $u_6 = 11$ 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. For example, let z_j be the maximum length of a path from node j to the end node, node 6. For each arc ij let $s_{ij} = u_6 - u_i - a_{ij} - z_j$. Then each s_{ij} is ≥ 0 (since the longest 1-6 path is at least as long as the longest such path through the arc ij). What does the value of s_{ij} tell us?

3.2 All arc lengths non-negative

Now 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 , such 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)) 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

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.

Clearly statements (i) and (ii) hold after initialisation, before we start the ‘while loop’. Assume that they hold at the start of a certain pass through the while loop. We claim that u_j is the minimum length of a 1– j path – this is the crucial point. 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

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 [?].]

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.

Sometimes negative cycles are of interest so we shall return to this. 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?)

As a second example, suppose that we are given exchange rates r_{uv} , where r_{uv} is the number of units of currency v that can be purchased with one unit of currency u . Observe that if we convert one unit of currency 1 into currency 2 and then convert into currency 3, then we obtain $r_{12}r_{23}$ units of currency 3. Thus we see that we could make money if we could find a cycle of currencies $v_0, v_1, \dots, v_k = v_0$ such that $\prod_{i=1}^k r_{v_{i-1}v_i} > 1$. Form a directed graph with nodes the currencies, and for each pair u, v the arc uv with cost $-\log r_{uv}$. Then such a cycle of currencies makes money precisely when it has negative cost, since its cost is

$$-\sum_{i=1}^k \log r_{v_{i-1}v_i} = -\log \prod_{i=1}^k r_{v_{i-1}v_i}.$$

We shall also want to be able to handle negative cost arcs later 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 ?? below.

Suppose that there are no negative cycles, and we wish to find the shortest distance between each pair of nodes. Let $u_{ij}^{(k)}$ be the minimum length of an $i-j$ path such that any intermediate nodes must be contained in $\{1, \dots, k\}$. Since there are no negative cycles, it would not matter here if we replaced ‘path’ by ‘walk’ (that is, we allowed nodes or edges to be repeated). Now the minimum length of an $i-j$ walk 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 distances 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 seen to be correct as follows. 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 .

Example Consider two very similar 4-node networks with cost matrices

$$A = \begin{pmatrix} 0 & -2 & \infty & \infty \\ \infty & 0 & 5 & 2 \\ \infty & \infty & 0 & \infty \\ 1 & \infty & \infty & 0 \end{pmatrix} \text{ and } B = \begin{pmatrix} 0 & -4 & \infty & \infty \\ \infty & 0 & 5 & 2 \\ \infty & \infty & 0 & \infty \\ 1 & \infty & \infty & 0 \end{pmatrix}.$$

With the costs A , where there are no negative cost cycles, after the iteration

with $k = 1$ we have $\begin{pmatrix} 0 & -2 & \infty & \infty \\ \infty & 0 & 5 & 2 \\ \infty & \infty & 0 & \infty \\ 1 & -1 & \infty & 0 \end{pmatrix}$ (where x_{42} has been replaced by

$x_{41} + x_{12}$). After the iteration with $k = 2$ we have $\begin{pmatrix} 0 & -2 & 3 & 0 \\ \infty & 0 & 5 & 2 \\ \infty & \infty & 0 & \infty \\ 1 & -1 & 4 & 0 \end{pmatrix}$. The

iteration with $k = 3$ changes nothing, and then the iteration with $k = 4$

yields the final output $\begin{pmatrix} 0 & -2 & 3 & 0 \\ 3 & 0 & 5 & 2 \\ \infty & \infty & 0 & \infty \\ 1 & -1 & 4 & 0 \end{pmatrix}$.

With the costs B , after the iteration with $k = 1$ we have $\begin{pmatrix} 0 & -4 & \infty & \infty \\ \infty & 0 & 5 & 2 \\ \infty & \infty & 0 & \infty \\ 1 & -3 & \infty & 0 \end{pmatrix}$,

after the iteration with $k = 2$ we have $\begin{pmatrix} 0 & -4 & 1 & -2 \\ \infty & 0 & 5 & 2 \\ \infty & \infty & 0 & \infty \\ 1 & -3 & 2 & -1 \end{pmatrix}$. The fact that $x_{44} < 0$ detects a negative cycle. The value came for replacing x_{44} by $x_{42} + x_{24}$, and the value of x_{42} came from $x_{41} + x_{12}$, so we find the negative cycle 1, 2, 4.