The Directed Chinese Postman Problem

HAROLD THIMBLEBY

School of Computing Science, Middlesex University, Bounds Green Road, LONDON, N11 2NQ.
Email: harold@mdx.ac.uk

The Chinese Postman Problem on weighted directed multigraphs has many applications, including analysing interactive system and web site usability. Clear algorithms for the problem are not available. This paper derives an algorithm, which is presented as a complete, executable Java class, and it shows that no significantly more efficient algorithm is possible. The paper also solves a generalisation, the directed Rural Chinese Postman.

Keywords: Directed Chinese Postman Problem, Usability Evaluation.

Received April 1999; revised October 1999, June 2000 & October 2000.

1. INTRODUCTION

Finding the shortest route for a travelling salesman, who wishes to visit every city, is a well known problem. Less well known is the Chinese Postman who wishes to travel along every road to deliver letters. The Chinese Postman Problem has numerous applications, but has not so far received a satisfactory treatment in the computing science literature.

This paper:

- Reviews motivating applications of the Chinese Postman Problem.
- Provides a derivation of the mathematical formulation of the Chinese Postman Problem for directed multigraphs.
- Provides and explains executable code to solve the problem. The code is written in Java [6] (a well-defined programming language that has a multi-platform executable environment) — provided as an Appendix, and also available on the World Wide Web. (The web site also provides a *Mathematica* [57] version of the code.)
- Shows that the problem is equivalent to a non-trivial linear programming problem (not merely that it can be solved *by* linear programming): specifically, there is essentially no simpler algorithm for the Chinese Postman Problem.

1.1. Background to the problem

The *Chinese Postman Problem* (CPP) is interesting because it is simple problem, with wide application, but for which there is no simple algorithm. The code for it, given at the end of this paper, is about ten times longer than that for the Travelling Salesman Problem (TSP) (as given in [50]) — and, as we argue below, there is no way to make it significantly shorter.

The simple algorithm mentioned above for the TSP is so slow that it has little practical use, but since the TSP has practical applications of economic importance, there are numerous specialised algorithms for it that exploit assumptions in the structure of particular classes of problem [41]. Likewise, the CPP for undirected and mixed graphs is NP-complete and has many algorithms [25], but in contrast the CPP for directed graphs — the topic of this paper — is polynomial and efficient and has basically one algorithm. This algorithm has never been published in full before now.

Unfortunately the CPP is caught up in the tensions between three clear points of view.

From the point of view of operations research, the CPP is a standard solved problem (e.g., [23]), yet Skiena's comprehensive *Algorithm Design Manual* [51] says no suitable algorithm (e.g., in a code library) has been identified for it, and that you have to do it yourself.

Some references, such as [14], merely assert "there is an efficient algorithm" but provide no details; reference [4] says, "[t]he details of the algorithm are too complicated to give here"; many mention CPP but do not give any solution because (they say) of the complexity of the accepted methods (e.g., [56]). The undergraduate reference work [18] mentions CPP as an exercise but gives no answer. Some describe the algorithm in a mixture of English and mathematical steps: for instance, the algorithm sketched in [39] cannot be made to work without very close reading of the rest of the paper, plus expertise in network flow algorithms. Some, such as [15, 51], describe the algorithm entirely in prose, so at least there are no unexpected sources of confusion, but it is unhelpful for people who want correct, executable algorithms.

Some, such as [30], just sketch the main steps of algorithms; indeed the presentation of the CPP in [30] is, in its own words, "rather informal" (unlike almost all other algorithms in the same book). Some, such as [2], provide a mixture of mathematics and English, and provide code for most of the relevant routines, but the code is presented in semi-formal notations, which are difficult to translate into complete and correct algorithms (e.g., because what may look like a simple variable is in fact a non-trivial dynamically bound expression).

In short, the literature on the CPP is not an effective starting point for getting a reliable algorithm. The published algorithms might better be called *quasi-algorithms*, for they clearly fail the test of definiteness required in computing science [33]. The paper [39], mentioned above, is a case in point. (To be more charitable, when the CPP was defined in the early 1960s [38] algorithms were not presented formally — the computer science literature has only discussed algorithms rigourously since the mid-1970s, and even today rigour is surprisingly patchy.)

Secondly, the CPP has real-world applications (see Section 2), but real-world applications generally require more sophistication than the basic problem covers. Operations research has generalised the CPP to more and more sophisticated applications, and the basic algorithm seems no longer an issue. This, combined with the previous point, means that the published quasi-algorithms for the CPP are at the same time trivial to professional operations researchers, yet fundamentally flawed to computer scientists.

Finally, from a professional point of view, specific cases of the CPP can be solved using efficient, though expensive, packages. The University of Karlsruhe has a web site listing numerous commercial packages for vehicle routing (of which the CPP is a special case) [48]; various reviews of commercial products are also available [29] — which says that vendors are "tight lipped" about their algorithms. From the professional point of view, then, there is no interest in (perhaps a resistance to) a clear exposition of the algorithm.

. . .

It is interesting to compare this situation with the otherwise wide accessibility of algorithms in the computing science literature. There are very few modern discussions of sorting, say, that do not express their contribution with an emphasis on correctness and by using a well-defined programming notation with no hidden surprises.

Computer scientists use the term algorithm to mean a definite procedure, but more generally good science is distinguished by being reproducible or, as Ziman puts it, *consensible* [58]: its claims should be expressed clearly enough so that others can either give assent or find well-founded objections. Either way, others should be able to build directly on published work, with a minimum of guesswork. This paper derives and provides a basic, executable algorithm for the CPP, expressed in Java (see Appendix A), a well-defined programming language. Indeed the algorithm is directly executable *exactly* as presented in this paper (see Appendix C for the justification of this claim).

Since nothing is missing from the program (Java systems accept and run it as it stands), our presentation provides as much as possible — both explanation and code — for anyone to precisely formulate their objections or improvements. More practically, readers of this paper may find the executable algorithm directly useful in their own work. It should be emphasised that our aim is to contextualise and present a correct, clear and precise algorithm, rather than an efficient algorithm which would be more complex. The paper, however, provides many pointers to the relevant literature, including a discussion (in Appendix D) of some alternative approaches to a core part of the algorithm.

Even if there were adequate published algorithms for the CPP, our particular implementation is elegant and interesting in its own right: the preliminary phase combines checking (to confirm it is given a valid problem) with initialising a data structure, which is then exploited in two different ways in the subsequent phases of the implementation; we also use a single shortest paths routine for three different purposes.

2. THE CHINESE POSTMAN PROBLEM AND ITS APPLICATIONS

A postman delivering letters in a village may wish to know a circuit that traverses each street (in the appropriate direction if one-way streets), starting and returning to the office. The postman may have to visit some streets more than once. This is obviously a graph theoretic problem: roads are edges, and road junctions are vertices. The postman requires a *Chinese Postman Tour*, which we abbreviate CPT [38].

There are important variations on the Chinese Postman Problem. Primarily, the graph may be undirected, directed or mixed. Computationally, the undirected and directed cases are polynomial, whereas the mixed is NP-hard [44]. This paper considers the directed case on multigraphs (graphs with parallel edges), in particular because this case arises naturally in analysing links in the World Wide Web.

Figure 1 shows a simple village, with four intersections and six one-way streets. The least cost CPT in this case requires travelling down streets ten times. In this case, some streets have to be used more than once, but this is not always the case.

The postman probably wants a shortest such circuit, with few repeated street visits. The *cost* of a CPT is defined as the total edge weight, summed along the circuit (e.g., the total distance walked). An *optimal CPT* is a CPT of minimal cost. If some weights are negative an optimal CPT may not be defined: if there is any circuit with an overall negative weight, the postman

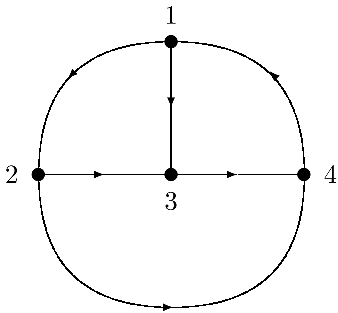

FIGURE 1. A simple directed graph that is not Eulerian. Taking each edge to be of equal weight, an optimal Chinese Postman Tour of this graph is $1, 2, 4, 1, 2, 3, 4, 1, 3, 4, 1$, a total of 10 edges. (There are other optimal tours of this graph; an optimal tour need not be unique.)

could arbitrarily repeat it and get a total cost lower and lower without bound.

Imagine trying to understand your mobile phone. Pressing buttons takes the phone to new states, and corresponds to travelling down one-way streets. After some considerable work, one might obtain a map of how the phone works. The question then is, is this map correct? Unfortunately, the map may be complex and difficult to test systematically. Given the map, a CPT will provide a systematic test sequence that will exercise each transition in every state. The optimal CPT will give the shortest test sequence possible; you could then step through the CPT 'instructions' and note any unusual behaviour of the mobile. (On the first visit to a state, all buttons supposedly unused in that state should be checked.) The length of the optimal CPT is therefore a measure of a machine's complexity [53].

As a concrete example, the Nokia 2110 mobile phone has a menu subsystem of 88 menu-items and 273 actions [42]; the optimal CPT of this takes 515 button presses, plus 79 presses (done at appropriate points) to check presses that do nothing. (Some states have fewer options than there are buttons; each unused button in a state corresponds to a self-loop in the graph.) In comparison, the shortest trip that visits each vertex at least once (a solution to the TSP), for instance to check that each menu-item function corresponds correctly to its name, is only 98 button presses. Thus it is much easier to check the functionality than check the user interface. It is clear that usability tests involving real users — even for such a "simple" device — are unlikely to be exhaustive, even if users make no mistakes. Indeed the Nokia 2110's user interface is quirky, perhaps a corroboration of the difficulty of performing effective user tests (cf. Figure 8 in [54]).

Another example is checking links in a web site. Since links are often descriptive, and require an understanding of their purpose, they must be checked manually to see whether they link to appropriate pages. However, on following a link, the human checker is now on another page. The CPT gives a route around a web site that exercises every link, and does so with minimal clicking. The web site for Benjamin Franklin's House [40] at the time of writing had 66 pages and 1191 links. The optimal CPT is 2248 links, which is hard for a human to do unaided. Either the human tester needs tool support, or an alternative method must be used. In fact, this site was generated by a web site compiler, which compiled 78 pages with only 201 links, most of which could be checked by the compiler automatically. The source site has a CPT of only 241 links, including the links the compiler checks itself.

Ideally a hypertext development system will provide mechanisms to help check sites (for instance, so that the human checker can take a break and not lose track of where they had got to). However a user of a web site, browsing it, may want to find the quickest route around the entire site. The longer the CPT, necessarily the harder it will be for someone to find their way around a site. It is clear, then, that the CPT gives a measure of the complexity of a web site. It would be interesting to research whether a user's sense of being "lost in hyperspace" [52] depends on a normalised length of the CPT — say, the CPT length divided by the total number of pages, which would be a bound on the rate of progress any user could make through a site.

To find the map of a mobile phone, video recorder or a web site in the first place is not easy if it has to be done by reverse engineering. This problem is equivalent to the *mobile robot exploration problem*, where a robot has to explore (by physically moving around in) a network when it does not know, to start with, what the network is. The robot has to explore every edge and vertex of the network (obeying any one-way restrictions), and it has to do so travelling a minimum distance. For a network of m edges, an algorithm has been found that takes at most $m\phi^{O(\log \phi)}$ steps [3], where ϕ is the *deficiency* of the graph, a term we define below. We will see that the algorithm for the CPT also determines the deficiency. Deficiency is a measure of the ease of use of a system: it relates to how hard a device is to fully understand without benefit of a map, or how hard a network (e.g., a building or interactive device) is to learn.

There are many variations of the CPP. The *hierarchical CPP* [26] partitions the graph into clusters, so that all edges in each cluster must be visited before the next stage starts: this variation has applications in construction. The *selecting CPP* (otherwise known as the *rural CPP*) is a variation where the postman must visit certain roads (the ones for which there are letters to be delivered) but not necessarily all of them [47]. In machine test terms, the selecting CPP arises, for instance, if we wish to find the shortest test sequence that tests each button on a machine at least once (but not necessarily to test all state transitions) [16, 49]. The selecting

CPP can be used to solve the *incremental CPP*, where an impatient postman sets off before finding the optimal CPT solution. The selecting CPP can also be used for solving the problem of checking a web site where some links have been mechanically constructed (e.g., from templates and are therefore known to be correct) but others, the ones to be checked, were created by hand. The *piecemeal* problem [8] is to perform a tour, while occasionally returning to the starting point (for instance, to refuel a robot, or to reset the mobile phone to standby — say, if it has timeouts that force a reset).

If web pages cannot be accessed from the home page of a site, then they are hardly part of the site, but there is no need for every page to be accessible from everywhere else. (For example, the site may be a directed tree rooted at the home page.) In this case, a tour of a site checking links will get stuck. However, browsers provide bookmarks, and provided the bookmarks have been initialised with pages from the site, it is possible to continue the tour of the site by occasionally 'jumping' to other pages using the browser features rather than intrinsic links in the site. This problem is a special case of the rural CPP. In Section 6 we show how a simple generalisation of the CPT solves this problem.

Because we are using multigraphs, a CPP algorithm can also solve postman problems where multiple visits to a street have different costs. Using the rural postman problem, any street visits necessary only to "get back" can have different costs again. For example, the first visit to a street may be slow because letters are delivered, the second visit (if necessary) faster because a few large letters are delivered, and if the postman uses a street to get back (without any deliveries), it can be done faster still. In testing web sites, to give a different example, it might be that the first visit to a page takes longer because it is being read more carefully.

We do not consider here undirected or mixed graphs (graphs containing both directed and undirected edges); but see [17, 20, 45]. In particular, we do not consider the *windy CPP* [46], which assumes the costs are different depending on the direction — as if the postman had to walk against the wind one way (but, as the edges are undirected, only needs to go down a street once in either direction).

Other applications of the CPP include routing snow ploughs so that streets are not wastefully re-cleared, garbage collection, milk delivery, and so on. The CPP can be used for solving the problem facing an art connoisseur who wishes to see every exhibit in a museum but without strenuous walking. For further references, see also [7, 9, 10, 13, 21, 22, 23] — references [21, 22] together form a substantial two-part review.

Finally Orloff [43] originally introduced the idea of the *General Routing Problem*: to visit some edges and some vertices in a graph — if all edges are to be visited, the problem is the CPP; if some edges are to be visited, the problem is the rural CPP; if all vertices are to be visited the problem is the TSP.

3. DEFINITIONS

Discussion of graph-theoretic terms may be found in standard graph theory references: e.g., any of [2, 4, 12, 27, 30, 56] cover everything required here; [18, 50] are more oriented to computing science, and in particular [50] has a good bibliography and provides many algorithms in *Mathematica* [57].

In this paper we consider weighted directed multigraphs (graphs that can have more than one edge between vertices). This suits the problem of checking web sites: a vertex is a page, an edge is a link, and the weight represents a cost, perhaps estimated in seconds, of the user checking the link. One might give different costs to within-page links than to links to other pages, frames, images, and so on.

Although our algorithm can handle applications other that link checking, it is worth considering a small point. Browsers typically provide 'back' buttons so a user can, after following a link from page A to page B, get back to page A. But this does not mean that the link AB is undirected. First, it is not possible to use the back button to get from B to A unless the previous action was to get from A to B. Secondly, we want to check the correctness of links, so regardless of the short-cuts the browser provides, if there is a link from A to B, we want to check it, and we must do so by going in that direction.

The *degree* of a vertex is the number of edges incident to the vertex. More specifically, the *in-degree* of a vertex v (the number of edges going into it) is $d^-(v)$, and the *out-degree* (the number of edges pointing out of it) is $d^+(v)$. Let $\delta(v) = d^+(v) - d^-(v)$. If $\delta(v) = 0$, we say the vertex v is *balanced*.

In general graphs may have isolated vertices, but we only consider graphs as collections of edges. It follows that every vertex must have at least one edge: the minimum degree of the graph is 1. Such a graph G is a collection of edges $\langle label, (i,j), c\rangle$, where *label* is an identifier for the edge (e.g., a URL and file offset, possibly along with the hot text), (i,j) is a directed edge (from vertex i to j), and c the cost associated with the edge. The goal is to determine a list of labels that, in order, constitute an optimal Chinese Postman Tour.

Theorem 3.1. *A graph of minimum degree at least 1 has a CPT (a circuit that traverses each edge at least once) if and only if it is strongly connected.*

(A graph that is weakly connected must have a minimum degree of at least 1, but the converse is not necessarily true.)

Proof.

(i) If a graph has a CPT then there is a circuit that traverses each edge (at least once). In a graph of minimum degree 1 every vertex has an incident edge, so all vertices must be visited, and since a CPT is closed, there is a directed path between every pair of vertices. Therefore the graph is strongly connected. (Note that a graph of minimum degree 0 may have a CPT yet with-

out being strongly connected.)

(*ii*) The proof of "only if" constructs a CPT. At any stage in the tour, identify any as-yet unvisited edge. Since the graph is strongly connected, there is a path from the current vertex to the vertex that originates the unvisited edge. Thus the tour may be extended with this path and to include the edge. The process is repeated until no unvisited edge remains. Finally a path is found from the last vertex visited to the initial vertex: such a path exists since the graph is strongly connected. □

As noted above, whether a graph with a CPT has an optimal CPT further requires there are no cycles of negative weight.

3.1. Eulerian graphs

An Eulerian graph is defined as one that has a circuit traversing each edge *exactly* once, and which returns to the start vertex. Finding an Euler circuit has standard, simple algorithms (see below, §5.4). Certain classes of Eulerian graph have trivial solutions, notably *randomly Eulerian* graphs [12], where circuits, possibly starting from a restricted subset of vertices, can be constructed by traversing any unused edge as each vertex is visited.

An Euler circuit of a graph is obviously an optimal CPT, since each edge is traversed exactly once. Unfortunately not all graphs are Eulerian and admit Euler circuits:

THEOREM 3.2. *A directed graph is Eulerian if and only if it is connected and every vertex is balanced.*

The proof is straightforward but (for the directed case) too lengthy to include here. (See [11] or [12] for an interesting historical perspective.)

4. AN OPTIMAL CPT ALGORITHM

The derivation of the algorithm in this section is quite detailed; the algorithm itself is discussed more concretely from Section 5 onwards. This section establishes what subproblems a CPT algorithm must solve, and it shows they are both necessary and sufficient (though this does not preclude alternative approaches that are equivalent). Moreover, to find a least cost tour, we show under what terms optimality can also be achieved.

By Theorem 3.1 we know the problem is soluble, but a CPT constructed in the manner of the proof is unlikely to be optimal. We must derive a better algorithm!

Every edge must be traversed at least once in a CPT. The total cost of traversing each edge exactly once (whether or not this is possible on a single path) is fixed, and is the sum of all edge costs.

Let f_{ij} be the *extra* number of times an edge (i, j) is traversed in a CPT; since every edge has to be traversed at least once, $f_{ij} + 1 \geq 1$ is the number of times an edge (i, j) is traversed in the CPT.

It is convenient to define c_{ij} as the least cost of any parallel edge (i, j).

Since a CPT has to traverse all edges at least once, finding the optimal tour amounts to finding how to minimise the excess cost, ϕ, the additional cost of traversing the repeated edges: $\phi = \sum c_{ij} f_{ij}$.

If each edge weight is 1, the smallest value of ϕ is the least number of edges that need to be added to make the graph Eulerian (proved below); this is the *deficiency* of the graph, as required for the robot exploration problem, mentioned above.

If G is Eulerian, all f would be zero, and ϕ would be zero and thus minimised for that graph.[2] If G is not Eulerian, however, ϕ clearly cannot be zero, so there must be constraints on the values f can take. What are these constraints?

Define two subsets of vertices:

$$\begin{aligned} D^+ &= \{v \mid \delta(v) > 0\} \\ D^- &= \{v \mid \delta(v) < 0\} \end{aligned}$$

So D^+ is the set of unbalanced vertices with an excess of out-going edges, and D^- the set of unbalanced vertices with an excess of in-going edges.

Three observations are required: that the algorithm's approach is necessary (i.e., that it does not do unnecessary work), sufficient (i.e., that it determines a valid CPT), and that it finds an optimal CPT:

1. *Necessity.* By Theorem 3.2, a directed graph is Eulerian if it is connected and every vertex is balanced. By definition, vertices in D^- and D^+ are not balanced: a CPT therefore requires edges adjoined out of D^- and edges (not necessarily the same edges) into D^+. These edges will form (part of) extra, adjoined, paths in the graph, and are a way of representing the fact that the CPT will require some edges in the original graph to be traversed more than once.

In order for all vertices to become balanced, the number of adjoined edges into a vertex $v \in D^+$ must equal the $\delta(v)$ of that vertex; and the number of adjoined edges out of a vertex $v \in D^-$ must equal the $-\delta(v)$ of that vertex.

THEOREM 4.1. *Exactly k adjoined paths are required to balance all vertices, where*

$$k = \sum_{v \in D^+} \delta(v).$$

PROOF.

(*i*) If edges are duplicated along any path with initial vertex in D^- to a final vertex in D^+, this does not affect the balance of any intermediate vertices (since the path goes both in and out of each intermediate vertex).

[2] True when the costs are positive; otherwise the minimal ϕ can be negative. Consider the graph $\{\langle a, (1, 2), -1\rangle, \langle b, (2, 1), 2\rangle, \langle c, (2, 1), 2\rangle\}$ (which has no negative cycles and therefore has a valid CPT).

(ii) If there were no edges in a graph, then $d^+(v)$ and $d^-(v)$ would be zero for every vertex. Now each edge (i, j) in a directed graph contributes 1 to $d^+(i)$ and 1 to $d^-(j)$ so necessarily, over all vertices v, $\sum_v d^+(v) - d^-(v) = 0$. By definition, then, $\sum_v \delta(v) = 0$. Since $\delta(v) > 0$ if and only if $v \in D^+$ and $\delta(v) < 0$ if and only if $v \in D^-$, we have

$$k = \sum_{v \in D^+} \delta(v) = -\sum_{v \in D^-} \delta(v)$$

(iii) Hence by choosing unbalanced vertices as the start and end of paths, adjoining k paths from D^- to D^+ makes all vertices balanced. □

2. *Sufficiency.* By Theorem 4.1 all vertices can be balanced by adjoining k paths from vertices in D^- to vertices in D^+. Since the paths are least cost paths, they are sufficiently determined by their end vertices.

In fact Observation 2 shows we can also find 'solutions' for arbitrary graphs, and therefore for graphs that are not strongly connected, which (by Theorem 3.1) cannot have CPTs. Thus the algorithm requires a guard that the graph is strongly connected.

Theorem 4.2. *If any adjoined edge is part of an optimal CPT, then that edge must occur as part of a minimal cost path from D^- to D^+.*

Proof. Since the total cost of the CPT is simply a sum of costs, then if any path is not a minimal cost path, we can reduce the cost of the CPT by replacing the path with a lower cost path. Any adjoined edge not on a minimal cost path must be suboptimal. □

Note that least cost paths may go through some vertices in D^- or D^+ because of restrictions in the connectivity of the graph.

3. *Optimality.* By Theorem 4.2, paths that are least cost paths from D^- to D^+ provide an optimal solution.

Conceptually our approach is therefore to adjoin paths to create a new graph $G^\star$ that is Eulerian, and where $G^\star$ is a smallest such graph (least total edge weight) that contains G as a subgraph. (The algorithm given in Appendix A does not represent $G^\star$ explicitly, instead it constructs the graph $G^\star - G$.) Thus $G^\star$ is constructed by finding least cost paths $(i \rightsquigarrow j)$ connecting the unbalanced vertices in G and adjoining corresponding edges (i, j) in $G^\star$. Adjoined edges $(i, j) \in G^\star$ have weight $c^\star_{ij}$, the weight of the least cost paths $(i \rightsquigarrow j) \in G$; note that $c^\star_{ij} \le c_{ij}$. Similarly the generalisation of f_{ij} of G in $G^\star$ is $f^\star_{ij}$.

Once we have the Eulerian graph $G^\star$, any Euler circuit of it (all Euler circuits have the same cost) represents an optimal CPT of the original graph G.

Linear programming often provides a concise formulation for an optimisation problem, as now. Recalling that each required path $(i \rightsquigarrow j)$ contributes 1 to $f^\star_{ij}$ we have an integer linear programming problem: minimise ϕ subject to certain constraints, as follows.

$$\begin{array}{rl}
\textbf{given}: & \left\{\begin{array}{rcl} \delta(i) & < & 0 \\ \delta(j) & > & 0 \\ \sum \delta(j) & = & -\sum \delta(i) \end{array}\right. \\[2ex]
\textbf{minimise}: & \quad \phi \;=\; \sum c^\star_{ij} f^\star_{ij} \\[2ex]
\textbf{subject to}: & \left\{\begin{array}{rcl} f^\star_{ij} & \ge & 0 \\ \sum_j f^\star_{ij} & = & -\delta(i) \\ \sum_i f^\star_{ij} & = & \delta(j) \end{array}\right. \\[2ex]
\multicolumn{2}{c}{\textbf{where } i \in D^-; j \in D^+; f^\star, \delta \in \mathbf{Z}; c^\star, \phi \in \mathbf{R}}
\end{array} \qquad (1)$$

This form of linear programming represents a *transportation problem* [36]. If we have appropriate library routines available, the problem is of course as good as solved (Appendix D), but if we do not, we need to work out an approach. First we show that the CPT yields a non-trivial transportation problem, then (in Section 5.3 below) we derive an algorithm appropriate for it. (The fully coded algorithm is given in Appendix A.)

The problem is that, while any of the k (cf. Theorem 4.1) adjoined least cost paths are easy enough to find *given* the choice of their start and end points in D^- and D^+, there are $k!$ possible pairings of end points to choose from. Thus, unless k is 0 or 1 (when $k! = k$), the optimal solution will be some least cost subset of the possible paths. We need to find a solution to this subsidiary problem more efficiently than blind search.

It is worthwhile briefly showing that this problem is non-trivial. An obvious way to try to 'solve' the problem would be to use the greedy method — to pick off paths, cheapest first.

Now, consider the following set of potential paths and costs:

$$\begin{array}{c} \delta(u_1) = -1,\ \delta(v_1) = 1 \\ \delta(u_2) = -1,\ \delta(v_2) = 1 \end{array}$$

$$\begin{array}{ll} ((u_1 \rightsquigarrow v_1),\ 1), & ((u_1 \rightsquigarrow v_2),\ 10), \\ ((u_2 \rightsquigarrow v_1), 10), & ((u_2 \rightsquigarrow v_2), 100) \end{array}$$

(We leave it as a simple exercise to show that there are graphs G that result in these conditions.)

Here $k = \delta(v_1) + \delta(v_2) = 2$ (in other words, $k = 2$ paths, chosen out of the $k! = 2$ possible subsets of paths must be found). Here the cheapest path is $(u_1 \rightsquigarrow v_1)$, but if it is selected, then the other path will have to be $(u_2 \rightsquigarrow v_2)$ and the total cost will be 101. But as can be seen by inspection this is not the best solution, which uses the paths $(u_1 \rightsquigarrow v_2)$ and $(u_2 \rightsquigarrow v_1)$, achieving a total cost of 20.

Incidentally, the greedy algorithm in this example happens to have found a *stable marriage* (neither u_1 nor v_1 would agree to swap their partners — so the pairings are stable): since it is stable but wrong, a stable marriage algorithm [34] is inadequate to solve the problem.

At the other extreme, another 'obvious' way to try to solve the problem would be to try all $k!$ cases systematically. A systematic search can be speeded up by using a branch and bound algorithm (once a solution has been found, no partial solution that costs more need be fully expanded). But without a guarantee that paths are explored in a suitable order, it could take a long time, since the total number of subsets of paths to examine grows prohibitively.

5. ALGORITHMIC DETAILS

The phases of the algorithm are to:

1. Check the graph is strongly connected (the algorithm otherwise "runs," but incorrectly). Check no cycle in the graph has a negative weight.
2. Identify unbalanced vertices that are the start and end of paths to be adjoined, and determine the costs of all possible paths between them. If the paths themselves are recorded at this stage, some of them will be used in Phase 7.
3. Of the set of possible paths, determine the subset of least cost paths so that all start and end vertices are 'used.' This problem can be solved by various methods. We describe one explicit method in Section 5.3, and review some alternatives in Appendix D: the simplex method, matching and network flow, etc.

If only the cost of the CPT is required, the following three phases, which determine the tour itself, are not required.

4. Construct an Eulerian graph $G^\star$, using the solution from Phase 3, where $E(G^\star) = E(G) \uplus \{f^\star_{ij} \times (i,j)\}$.
5. Determine the Euler circuit of this graph.
6. Convert the Euler circuit into a CPT of the original graph (this subproblem requires the least cost paths from Phase 2). There are variations on this phase, depending on whether a complete circuit is required (i.e., a return to the start vertex).

In the literature, phases 1–3 are typically assumed or glossed; phase 4 is so familiar to the operations research literature that it is dismissed — whereas in fact it conceals some problems; and phases 5 and 6 are both straight forward and well explained in the literature. (Phase 6 can be combined elegantly with 1 and 3.) Finally, phase 7 is straight forward. Appendix A provides full details of all phases.

5.1. Worked example

For the example in Figure 1, and taking all edges to have weight 1, we have the excess cost $\phi = 2f^\star_{31} + 3f^\star_{32} + f^\star_{41} + 2f^\star_{42}$. The cost of the least cost path $(4 \rightsquigarrow 1)$ is 1, $(3 \rightsquigarrow 1)$ and $(4 \rightsquigarrow 2)$ is 2, $(3 \rightsquigarrow 2)$ is 3, so these are the coefficients of f_{ij}. The constraints are $f^\star_{31} + f^\star_{41} = 1, f^\star_{32} + f^\star_{42} = 1, f^\star_{31} + f^\star_{32} = 1, f^\star_{41} + f^\star_{42} = 1$ — one equation for each unbalanced vertex. Notice that this problem only has four variables, not one for each of the sixteen possible paths, but just one for each path from $\{3, 4\}$ (those in D^-) to each of $\{1, 2\}$ (those in D^+). An optimal solution is $f^\star_{31} = 0, f^\star_{32} = 1, f^\star_{41} = 1, f^\star_{42} = 0$, and the deficiency of the graph is therefore 4. The graph has 6 edges, so the optimal CPT takes $6+4 = 10$ edges. Figure 2 summarises.

Although the example is not complex (e.g., edges have equal weight), it does have the property that ϕ does not depend on $f^\star$. This is relevant because it provides a counter-example to the idea that a transportation problem (such as the CPP) necessarily *only* has integer solutions. There is a solution $f^\star_{31} = f^\star_{32} = f^\star_{41} = f^\star_{42} = \frac{1}{2}$ where ϕ has the optimal value of 4 and the constraints are still satisfied except the $f^\star_{ij}$ are not integral. This is a problem, since non-integral $f^\star$ are not acceptable: a valid CPT solution requires $f^\star$ be integral. That $f^\star$ *can be* integral at the optimal ϕ follows from the form of the constraints: the polyhedron of constraints has integer vertices, and a minimal value for ϕ can always be found at a vertex.

5.2. Phases 1 & 2: Preparation

First it must be established that G represents a strongly connected graph with no negative weight cycles. This may be done by finding the all-pairs shortest path costs, $c^\star$ (for example, by using the Floyd-Warshall algorithm [1, 51]). If some element of $c^\star$ is unbounded, the graph is not connected. For any vertex v, $c^\star_{vv}$ is the cost of a least cost cycle including that vertex; a graph with negative weight cycles will have $c^\star_{vv} < 0$ for some v.

The following notes show that this stage of the algorithm can be exploited further in preparation for later stages:

5.2.1. Finding shortest paths

Later the algorithm for the CPP will require certain shortest paths and their costs. The Floyd-Warshall algorithm can efficiently find and record *all* shortest paths at the same time as establishing the costs, by building a matrix P (called `path` in Appendix A) such that the first edge of the least cost path $(u \rightsquigarrow v)$ from vertex u to vertex v is P_{uv}. (Subsequent edges along the path are found similarly; if P_{uv} is an edge (u, u') to $u' \neq v$ then $P_{u'v}$ is the next edge to take towards v, and so on.)

Original graph G (as list of edges)	$(1,2),(1,3),(2,3),(2,4),(3,4),(4,1)$
Vertex δs	$\delta(1)=1,\ \delta(2)=1,\ \delta(3)=-1,\ \delta(4)=-1$
D^-	$\{3,4\}$
D^+	$\{1,2\}$
Variables	$f^\star_{31}, f^\star_{32}, f^\star_{41}, f^\star_{42}$
ϕ	$2f^\star_{31}+3f^\star_{32}+f^\star_{41}+2f^\star_{42}$
Constraints	$f^\star_{31}+f^\star_{41}=1,\ f^\star_{32}+f^\star_{42}=1,\ f^\star_{31}+f^\star_{32}=1,\ f^\star_{41}+f^\star_{42}=1$
Integer solution minimising ϕ	$f^\star_{31}=0,\ f^\star_{32}=1,\ f^\star_{41}=1,\ f^\star_{42}=0$
Extra paths to adjoin	$f^\star_{ij}\times(i\rightsquigarrow j)=(3\rightsquigarrow 2),(4\rightsquigarrow 1)$
Adjoined to make $G^\star$	$(1,2),(1,3),(2,4),(2,3),(3,4),(4,1),(3,2),(4,1)$
Euler circuit for $G^\star$	$3,2,4,1,3,4,1,2,3$
Least cost path for edge not in the graph G	$(3\rightsquigarrow 2)\mapsto 3,4,1,2$
... splice in to make CPT of G	$3,4,1,2,4,1,3,4,1,2,3$
Most expensive adjoined path	$(3\rightsquigarrow 2)$
... hence least cost unrestricted CPT	$2,4,1,3,4,1,2,3$

FIGURE 2. Sample data solving the CPT for the graph from Figure 1.

5.2.2. Finding a spanning tree

The last phase of the CPP requires an Eulerian circuit. One algorithm for finding an Eulerian circuit uses a spanning tree (see §5.4) — note that the matrix P gives the edges of a spanning tree T_v of G rooted at v by $T_v=\{P_{uv}: u\in V, u\neq v\}$. Since, by construction, G is a spanning subgraph of $G^\star$, any spanning tree of G is a spanning tree of $G^\star$, hence an Eulerian circuit of $G^\star$ can be found using T_v.

The algorithm presented in Appendix A uses both ideas. However if it is known *a priori* that the graph is connected and has no negative cycles (many classes of graphs encountered in practice are known to have positive weights) then — depending on trade-offs with other algorithms used — it may be more efficient to compute the vertex degrees, and then find the fewer shortest paths (and their costs) only between the unbalanced vertices.

5.3. Phase 3: Solving the optimisation problem

Although the core of the algorithm (Phase 3) can be expressed as a linear programming problem, using a general purpose method (e.g., the simplex method), while possibly convenient if it is available as a subroutine (as it is in, say, *Mathematica*), is not the only or most efficient way of attacking it. Here we develop a *cycle canceling algorithm* [31]; we review some alternatives in Appendix D.

For notational clarity, in this section it is convenient to use a plain c and f instead of $c^\star$ and $f^\star$. A set of values $F=\{f_{ij}\}$ that satisfies the constraints (1), but which may or may not yield an optimal value for ϕ, is termed a *feasible solution*. Thus the algorithm must find a feasible solution that is also optimal.

Any feasible solution F' that is an improvement over another feasible solution F must adjust some values: $f'_{ij}=f_{ij}-a_{ij}$, which we can write as $F'=F-A$, A being the graph of non-zero a-edges. The overall improvement in ϕ will be $\sum c_{ij}a_{ij}$.

An algorithm therefore suggests itself:

- Establish an initial feasible solution, F (e.g., using the greedy method);
- Iterate $F:=F'$ while better solutions $F'=F-A$ can be found.

If $F'=F-A$ is a feasible solution, what can we say about the adjustment A? Both $\sum_j f_{ij}$ (for all $i\in D^-$) and $\sum_i f_{ij}$ (for all $j\in D^+$) are fixed for all feasible solutions (of a particular problem), so necessarily both $\sum_i a_{ij}$ and $\sum_j a_{ij}$ are zero. We call an adjustment satisfying these zero sum constraints and with no $a_{ij}>f_{ij}$ (since all $f'\geq 0$) a *valid adjustment*.

The problem, then, is to find valid adjustments. We will show that any valid adjustment $F'=F-A$ can be broken down into steps, each of which uses a valid adjustment which is a cycle (we want to be sure this is the case when A is an adjustment that finds an optimal solution). Finding such cycles is straight forward, as we shall see.

Now, a cycle of adjustment values $W(k)=\langle w_{12}=k, w_{23}=-k, w_{34}=k\ldots, w_{n1}=-k\rangle$ (all others zero) satisfies the zero sum constraints, provided it alternates between D^- and D^+. If no $w_{ij}>f_{ij}$, then all F' will be non-negative, and $W(k)$ will be a valid adjustment.

Theorem 5.1. *Any valid adjustment A is a sum of valid cycles.*

Proof. If all edges in A are zero, the adjustment is trivially a sum of zero cycles. Otherwise, for any valid adjustment A with some non-zero edges, let k be the least absolute value of a non-zero edge. We can clearly choose a cycle $W(k)$ that traces a non-zero cycle in A and that "cancels" at least one edge, so it is zero in $A-W(k)$. Moreover, since k is the least cost of a valid adjustment, $A-W(k)$ must also be valid, as is $W(k)$. Since subtracting $W(k)$ decreases the number of non-zero edges, there is a finite series of cycles that will cancel all non-zero edges. Thus, if A is any valid

adjustment, it is a sum of valid cycles. □

By Theorem 5.1, a valid adjustment A can always be expanded as a series of valid cycles. If F' is an improved solution over F, then at least one of these cycles must have a positive cost. Conversely if there is no such cycle, then there is no F' that can be an improved solution (and the algorithm terminates with F as an optimal solution).

Conveniently the cost of a cycle $W(k)$ is k times the cost of $W(1)$. Hence to identify a cycle of positive cost, find a cycle W that has a positive total $\sum \pm c_{ij}$ summed with alternating signs along W; to find a valid cycle we must exclude from consideration any edges where f_{ij} would be made negative. Since $k \geq 1$, the edges that are excluded from consideration are those with $f = 0$.

Fortunately this subproblem is easier than it appears! Create a weighted directed graph R (a so-called *residual graph*) with pairs of edges

$$\begin{array}{ll} (i,j) \text{ with cost } +c_{ij} & \text{if } i \in D^-,\ j \in D^+ \\ (j,i) \text{ with cost } -c_{ij} & \text{if } f_{ij} \neq 0 \end{array}$$

This graph has cycles of edges with costs of alternating signs (strictly, of alternating ± 1 multipliers), though for reasons that will become apparent shortly we have reversed all signs. To identify W, then, find a *negative* cost cycle in R.

One way to do this is to find the all-pairs shortest paths; if there is any negative cycle in R there will in particular be a path $(i \rightsquigarrow i)$ with negative cost for some i. Choose one, fix k as the largest that would make no f' negative along that cycle, and hence generate the solution $F' = F - W(k)$.

Unfortunately the Floyd-Warshall algorithm can find spurious negative cycles: for once a negative cycle has been found, any other cycle $(j \rightsquigarrow j)$ can be diverted through the negative cycle, thus reducing its cost, and possibly creating what appears to be another negative cycle. Thus we modify the Floyd-Warshall algorithm to terminate as soon as the first negative cycle is found. Since this also ensures that no vertex is recorded as being on more than one cycle, recovering the path of the cycle is easy, using the method of Section 5.2.1.

Now we see why we did not use $-R$ and look for a cycle of positive cost. The modified Floyd-Warshall algorithm finds (if any) at most one negative cycle, and otherwise all shortest paths. Since elsewhere in the CPT a negative cycle (i.e., in G) is an error, finding at most one is sufficient, and the same routine can be used in all contexts correctly.

Three important questions remain.

Does the algorithm find integer solutions? We know from Section 5.1 that there may be non-integral optimal solutions, which would be inappropriate for the CPT, so it is essential to argue that the algorithm does find integral solutions. Provided the greedy solution to the initialisation finds integral values for F, the k for canceling any negative cycle must be integral. Hence each iteration changes f by integers, and solutions found will be integral. (Since Phase 1 checked G is strongly connected, by Theorem 3.1, we know there is a feasible solution. We leave it as an exercise to show that the greedy approach (see Appendix A) finds a feasible integer solution.)

Does the algorithm terminate? Since the f values are integral, not less than zero, and bounded from above (from the constraints), there are only a finite number of feasible solutions. Since each iteration either decreases ϕ or terminates, the changes to the f values cannot cycle. Therefore the algorithm terminates in a finite number of steps.

Does the algorithm find an optimal solution? If there is an improved solution F' there must be a valid adjustment cycle. Therefore every iteration decreases ϕ, and the algorithm does not terminate until there is no possible improvement.

Thus the algorithm is correct; whether it is efficient is another matter; cf. Appendix B, which provides some suggestions. We have developed one that is simple and easy to show correct in preference for a sophisticated algorithm that might be usefully faster for very large problems.

5.4. Phases 4, 5, & 6: Finding the tour

Construct the graph $G^\star$: for each $f^\star_{ij} > 0$ of the optimal solution to the transportation problem, adjoin $f^\star_{ij}$ edges (i,j) to G. The optimal CPT of G then corresponds to any Eulerian circuit of $G^\star$, where each *adjoined* edge (i,j) in $G^\star$ represents a least cost *path* $(i \rightsquigarrow j)$ in the CPT of G. Notice the proviso '*adjoined*,' since if we simply chose least cost paths in G for any edge in $G^\star$, then of a parallel set of original edges in G, possibly of different cost, we would always take the least cost, and therefore not find a complete tour.

An Eulerian circuit of $G^\star$ can be found using a standard algorithm. Find a spanning tree of G (this also spans $G^\star$). To read off the Eulerian circuit, edges are chosen in order such that: no edge in $G^\star$ is traversed more than once and no edge is chosen from the tree if another edge is still available [27]. Equivalently, the CPT can be read off without explicitly representing $G^\star$: for any vertex on the tour, take a path from $f^\star_{ij} \times (i,j)$ if possible, else take an edge from G other than an edge in the spanning tree if possible, else take an edge from the spanning tree.

Finding a spanning tree is trivial assuming we have already worked out the shortest path matrix P (e.g., having used the Floyd-Warshall algorithm to check connectivity and find shortest paths), as described above in §5.2.

6. THE DIRECTED RURAL CHINESE POSTMAN AND OTHER VARIATIONS

The CPT requires paths from D^- to D^+. In a strongly connected graph G, these paths can be found in G. If

the graph is not strongly connected, no CPT is possible. If for some graph F, $F \cup G$ is strongly connected, there is a CPT of the union. The *Rural Chinese Postman Tour* is a minimal cost tour of a graph (here, the directed multigraph $F \cup G$) such that every edge of a specified subgraph (here, G) is traversed at least once.

As described earlier, a practical application of this is web site link checking. The graph F corresponds to the bookmark list of the browser, which allows the user to 'jump' to any page of the site. The browser bookmarks, however, do not need checking by hand (they can be generated automatically). In this case, F may be a complete graph.

- To solve the problem, set the costs $c^\star_{ij}$ as the minimal cost of paths $(i \rightsquigarrow j) \in F \cup G$, and keep δ unchanged as the balance of G. Clearly, to find the tour (rather than just its cost) a separate structure is required to track whether the minimum cost paths are to be taken from F or G.
- If costs of edges in F are less than those in G, it is clear that no repeated edges of G will be required in a CPT.
- Since the only adjoined edges are in D^- to D^+, the bookmark list need only give D^+. (To make checking easier for a human, bookmarks may be repeated δ times and put in an order corresponding to the chosen CPT.)

6.1. Other variations on the Postman Problem

A solution following a complete Eulerian circuit (of $G^\star$) starts and finishes at the same vertex, which is appropriate for some problems (e.g., for snow ploughs, which need to start and return to base) but may be an unnecessary restriction for other applications.

- For problems where one only wishes to visit every edge without restriction on where one starts or finishes (e.g., as in checking a web site), first determine the most expensive adjoined path $(u \rightsquigarrow v)$ and start the tour at v and stop at the last visit to u.
- If there is a restriction that the tour must start at a nominated vertex (e.g., a home page) v, but no restriction on where it finishes, then determine the most expensive adjoined path $(u \rightsquigarrow v)$ into v. Start the tour at v and stop at the last visit to u.

The algorithm above (§5.4) is easily modified: having found an Eulerian circuit of $G^\star$, mark $(u \rightsquigarrow v)$ as having been traversed once (the complete CPT may have required it to be traversed more than once), then start the tour, as before, starting at v. The tour then finishes at u when there are no untraversed edges or paths to take.

7. THE DIFFICULTY OF CPP

Section 4 showed that the CPP can be solved by reducing it to a transport integer linear programming problem. As pointed out, this result in itself is not surprising, since linear programming is a powerful problem solving paradigm (for example, linear programming can solve the TSP [5] — see, too, the seminal paper by Dantzig *et al.* [19]). We now show that this form of linear programming can be reduced (in linear time) to finding an optimal CPT, and that the CPT represents the solution. It follows that the two problems are equivalent (up to the cost of the reduction), and therefore no significantly better algorithm can be found for the CPP.

Matching can be done efficiently (in polynomial time), though no particularly simple algorithm is known — Knuth's C code for matching ([32] and Appendix D.1) runs to about 9 book pages of code, 16 including commentary. Though it may still be possible to combine the various stages of CPP together into a different algorithm not explicitly requiring a matching stage, since the reduction of matching to CPT is trivial it follows that no simple algorithm for CPP is available. This is in contrast to the TSP, for which there are very simple algorithms (e.g., depth first search takes a few lines of code) but for which there are no efficient algorithms — see [41] for an excellent review of the TSP.

Any optimisation problem of the form (1) can be converted to a suitable graph. Let $n = |D^-|$ and $p = |D^+|$, $n, p > 0$. Construct a graph with vertices $\{b,\ u_1, \ldots, u_n,\ v_1, \ldots, v_p\}$ as follows.

Add all $n \times p$ edges (u_i, v_j), and assign them weights $c^\star_{ij}$. For each vertex u_i add p parallel edges (b, u_i) from b, and for each vertex v_j add n parallel edges (v_j, b) to b.

This graph is Eulerian.

Vertices u_i should, however, have balance $-\delta(u_i)$; this is achieved by adding this number of edges from b to u_i. Similarly, vertices v_j should have balance $\delta(v_j)$; this is achieved by adding this number of edges to b from v_j.

To ensure the shortest paths from u_i to v_j are the edges (u_i, v_j), no indirect path of the form (u_i, v', b, u', v_j) can have a cost less than $c^\star_{ij}$. This is readily achieved by assigning the edges into and from b weights at least $\max(c^\star_{ij})$.

This completes the construction; see Figure 3. Of course, this graph is 'just another' representation of a graph matching problem (§D.1), but with the desired property that it has a meaningful CPT.

We next need to show that the vertex b is balanced, so that the CPT of the graph will only involve repeating the edges (u_i, v_j).

Each vertex u_i has an overall balance of $-\delta(u_i)$ by construction. Hence the total contribution of the u vertices to the out-degree of b is

$$np + \sum_{i=1..n} -\delta(u_i) = np - \sum_{u_i \in D^-} \delta(u_i)$$

Similarly the contribution of the v vertices to the total in-degree of b is

$$np + \sum_{j=1..p} \delta(v_j) = np + \sum_{v_j \in D^+} \delta(v_j)$$

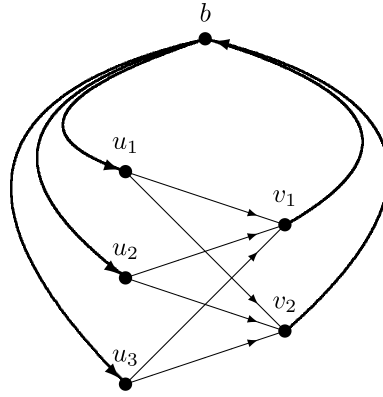

FIGURE 3. Example construction of a graph corresponding to the linear programming formulation of the CPP. Here $n = 3 = |D^-|$ and $p = 2 = |D^+|$. The thicker lines represent multiple parallel edges; the thin lines single edges (for all vertices, the in-degrees and out-degrees are equal).

These sums are equal by the constraints of (1), as consistent with Observation 2. Therefore b is balanced, and the CPT for the graph only repeats the edges (u_i, v_j), as required.

The solution to the optimisation problem can be derived from the optimal CPT of that graph. Given a CPT of such a graph, identify the edges that are traversed more than once (these are not incident to b). This immediately identifies the vertices in D^- and D^+ and the optimal paths between them. The values of $f^\star$ can be read off as one less than the number of times each path $(u_i \rightsquigarrow v_j)$ is traversed, and the value of ϕ is the total of $f^\star$ times the weight of each path (i.e., $\phi = \sum c^\star_{u_i v_j} f^\star_{u_i v_j}$).

The reduction is efficient. If the reduction of the linear programming problem to the CPP, and obtaining the solution from the CPT were sufficiently costly operations, then we could not claim the problems were equivalent. We must show the reduction is efficient — specifically it must, in the limit, be more efficient than the polynomial cost of performing the linear programming (Knuth's algorithm is cubic).

Unfortunately the graph constructed has a number of edges, $3np + \sum |\delta|$, dependent on the *values* of the constraints, not just on the *number* of constraints. However if the graph is represented as a set of triples, $\langle (v_1, v_2), c, n \rangle$ — as an edge, a weight, and a repetition — then the number of triples is $n + p + np$, and independent of the constraint values. The graph can therefore be constructed in linear time on the number of constraints.

Reading off the solution from the CPT requires counting repeated edges. The naïve cost of doing this is proportional to the number of edges in the CPT, $3np + \frac{3}{2} \sum |\delta|$, rather than proportional to the number of constraints. However, a tour is a sequence of the form $b, (u, v)_1, b, (u, v)_2, b \ldots$. In this sequence, the order of traversal of the edges $(u, v)_i$ is immaterial — since all paths cross over through the vertex b, there is no 'memory' of the previous edges. Therefore the CPT can be permuted so that repeated (u, v) edge traversals are consecutive. Without loss of generality, then, the CPT can be represented canonically as a set of np pairs, $\langle (u, v), n \rangle$ — as an edge and a repetition. This representation of the CPT is a direct representation of the solution, *viz* $\langle (u, v), f^\star_{uv} + 1 \rangle$.

Finally, because of the necessity and optimality conditions (Section 4) an efficient CPT algorithm cannot omit the shortest paths computation (though one might use more efficient algorithms than we do here) and the construction of the Euler tour. Thus we conclude that there is no substantially better algorithm possible.

8. CONCLUSIONS

This paper has shown that the Chinese Postman Problem may be solved using standard algorithms. The tour itself is found from an Euler circuit of a derived graph, which is found as a solution to a linear programming problem, by a minimum cost bipartite matching problem, or in other ways; in turn, the Euler circuit can be found from spanning trees, and the paths required are found from least cost path algorithms.

The Chinese Postman Problem does not get the recognition it deserves because it has seemed too complex for elementary treatment, yet it is in a sense trivial — it is equivalent to, and therefore as hard as, a collection of well-known problems.

Appendix A provides complete Java code. The World Wide Web site

`http://www.cs.mdx.ac.uk/harold/cpp`

provides downloadable code in both Java and *Mathematica* (along with test data, including the Nokia mobile phone example) as a complete *Mathematica* notebook that can be directly executed or viewed (e.g., for copy-and-paste) in a *Mathematica* 'reader.'

ACKNOWLEDGEMENTS

The author is grateful for constructive comments from Paul Cairns, Paul Curzon, Matt Jones, Peter Ladkin, James Orlin and Peter Rowlinson, and from the referees. This work was supported by EPSRC Grants No. GR/J43110 and No. GR/K79376.

REFERENCES

[1] R. K. AHUJA, K. MEHLHORN, J. B. ORLIN & R. E. TARJAN, "Faster Algorithms for the Shortest Path Problem," *Journal of the ACM*, **37**(2):213–223, 1990.

[2] R. K. AHUJA, T. L. MAGNANTI & J. B. ORLIN, *Network Flows: Theory, Algorithms, and Applications*, Prentice-Hall (Simon and Schuster), 1993.

[3] S. ALBERS & M. R. HENZINGER, *Exploring Unknown Environments*, Digital Systems Research Center, SRC Technical Note 1997-014, 1997.

[4] J. M. ALDOUS & R. J. WILSON, *Graphs and Applications*, The Open University, Springer Verlag, 2000.

[5] D. APPLEGATE, R. BIXBY, V. CHVÁTAL & W. COOK, "On the Solution of Traveling Salesman Problems," *Documenta Mathematica*, Extra Volume **ICM III**:645–656, 1998.

[6] K. ARNOLD & J. GOSLING, *The Java™ Programming Language Second Edition*, 2nd. ed., Addison-Wesley, 1998.

[7] A. A. ASSAD & B. L. GOLDEN, "Arc Routing Methods and Applications," in *Network Routing*, M. O. BALL, T. L. MAGNANTI, C. L. MONMA & G. L. NEMHAUSER, eds., 375–483, North Holland, 1995.

[8] B. AWERBUCH, M. BETKE, R. L. RIVEST & M. SINGH, "Piecemeal Graph Exploration by a Mobile Robot," *Information and Computation*, **152**(2):155–172, 1999.

[9] F. BARAHONA, "On Some Applications of the Chinese Postman Problem," in *Paths, Flows and VLSI-layout*, B. KORTE, L. LOVÁSZ, H. PRÖMEL & A. SCHRIJEVER, eds., 1–16, Springer, 1990.

[10] E. BELTRAMI & L. BODIN, "Networks and Vehicle Routing for Municipal Waste Collection," *Networks*, **4**(1):65–94, 1974.

[11] N. L. BIGGS, E. K. LLOYD & R. J. WILSON, *Graph Theory, 1736–1936*, Oxford University Press, 1976.

[12] B. BOLLOBÁS, *Graph Theory*, Springer-Verlag, 1979.

[13] P. BRUCKER, "The Chinese Postman Problem for Mixed Networks," Proceedings of the International Workshop on Graphtheoretic Concepts in Computer Science, *Lecture Notes in Computer Science*, **100**, Springer-Verlag, 1980.

[14] P. J. CAMERON, *Combinatorics: Topics, Techniques, Algorithms*, Cambridge University Press, 1994.

[15] G. CHARTRAND & O. R. OELLERMANN, *Applied and Algorithmic Graph Theory*, McGraw-Hill, 1993.

[16] W-H. CHEN, "Test Sequence Generation from the Protocol Data Portion Based on the Selecting Chinese Postman Problem," *Information Processing Letters*, **65**(5):261–268, 1998.

[17] A. CORBERÁN, R. MARTÌ & A. ROMERO, "Heuristics for the Mixed Rural Postman Problem," *Computers & Operations Research*, **27**(2):183–203, 2000.

[18] T. H. CORMAN, C. E. LEISERSON & R. L. RIVEST, *Introduction to Algorithms*, MIT Press, 1992.

[19] G. B. DANTZIG, R. FULKERSON & S. M. JOHNSON, "Solution of a Large-scale Traveling Salesman Problem," *Operations Research*, **2**:393–410, 1954.

[20] J. EDMONDS & E. L. JOHNSON, "Matching, Euler Tours and the Chinese Postman," *Mathematical Programming*, **5**:88–124, 1973.

[21] H. A. EISELT, M. GENDREAU & G. LAPORTE, "Arc-routing Problems, Part 1: The Chinese Postman Problem," *Operations Research*, **43**:231–242, 1995.

[22] H. A. EISELT, M. GENDREAU & G. LAPORTE, "Arc-routing Problems, Part 2: The Rural Postman Problem," *Operations Research*, **43**:399–414, 1995.

[23] H. FLEISCHNER, *Eulerian Graphs and Related Topics*, **2**(1), *Annals of Discrete Mathematics*, **50**, North-Holland, 1991.

[24] L. R. FORD & D. R. FULKERSON, *Flows in Networks*, Princeton University Press, 1962.

[25] G. N. FREDERICKSON, "Approximation Algorithms for Some Postman Problems," *Journal of the Association of Computing Machinery*, **26**(3):538–554, 1979.

[26] G. GHIAN, G. IMPROTA, "An Algorithm for the Hierarchical Chinese Postman Problem," *Operations Research Letters*, **26**(1):27–32, 2000.

[27] A. GIBBONS, *Algorithmic Graph Theory*, Cambridge University Press, 1985.

[28] A. V. GOLDBERG & R. E. TARJAN, "Finding Minimum-Cost Circulations by Canceling Negative Cycles," *Journal of the ACM*, **36**(4):873–886, 1989.

[29] R. W. HALL & J. G. PARTYKA, "On The Road to Efficiency," *OR/MS Today*, **24**(3):38–47, `http://lionhrtpub.com/orms/orms-6-97/Vehicle-Routing.html`, 1997.

[30] D. JUNGNICKEL, *Graphs, Networks and Algorithms*, Algorithms and Computation in Mathematics, **5**, Springer, 1999.

[31] M. KLEIN, "A Primal Method for Minimum Cost Flows with Applications to Assignment and Transportation Problems," *Management Science*, **14**(3):205–220, 1967.

[32] D. E. KNUTH, *The Stanford GraphBase*, Addison-Wesley, 1993.

[33] D. E. KNUTH, *The Art of Computer Programming*, **1**, 3rd. ed., Addison-Wesley, 1997.

[34] D. E. KNUTH, *Stable Marriage and Its Relation to Other Combinatorial Problems: An Introduction to the Mathematical Analysis of Algorithms*, tr. M. GOLDSTEIN, CRM [Centre de Recherches Mathématiques Université de Montreal] Proceedings & Lecture Notes, **10**, American Mathematical Society, 1997.

[35] D. E. KNUTH & S. LEVY, *The CWEB System of Structured Documentation, Version 3.0*, Addison-Wesley, 1994.

[36] B. KOLMAN & R. E. BECK, *Elementary Linear Programming with Applications*, Academic Press, 2nd. edition, 1995.

[37] R. J. KORSAN, "Mixed-Integer Linear Programming," *Mathematica Journal*, **3**(2):48–51, 1993.

[38] Kuan (Kwan or Guǎn) Mei-Ko, "Graphic Programming Using Odd or Even Points," *Chinese Mathematics*, **1**:273–277, 1962.

[39] Y. Lin & Y. Zhao, "A New Algorithm for the Directed Chinese Postman Problem," *Computers and Operations Research*, **15**(6):577–584, 1988.

[40] G. Marsden & H. Thimbleby, *Benjamin Franklin Centre*, `http://www.rsa.org.uk/franklin/`, 1998.

[41] Z. Michalewicz & D. B. Fogel, *How to Solve It: Modern Heuristics*, Springer, 2000.

[42] Nokia Mobile Phones, *Nokia 2110 User's Guide*, Issue 5, 1996.

[43] C. S. Orloff, "A Fundamental Problem in Vehicle Routing," *Networks*, **4**(1):35–64, 1974.

[44] C. H. Papadimitriou, "On the Complexity of Edge Traversing," *Journal of the ACM*, **23**:544–554, 1976.

[45] W. L. Pearn & J. B. Chou, "Improved Solutions for the Chinese Postman Problem on Mixed Networks," *Computers & Operations Research*, **26**(8):819–827, 1999.

[46] W. L. Pearn & M. L. Li, "Algorithms for the Windy Postman Problem," *Computers & Operations Research*, **21**(6):641-651, 1994.

[47] W. L. Pearn & C. M. Liu, "Algorithms for the Rural Postman Problem." *Computers & Operations Research*, **22**(8):819–828, 1995.

[48] F. Rösch & C. Schwindt, `http://www.wior.uni-karlsruhe.de/Biliothek`, Faculty of Business Administration and Industrial Engineering, University of Karlsruhe (TH), 1999.

[49] Y. N. Shen & F. Lombardi, "Graph Algorithms for Conformance Testing using the Rural Chinese Postman Tour," *SIAM Journal on Discrete Mathematics*, **9**(4):511–528, 1996.

[50] S. Skiena, *Implementing Discrete Mathematics*, Addison-Wesley, 1990.

[51] S. Skiena, *The Algorithm Design Manual*, Springer Verlag, 1998.

[52] Y.-L. Theng, M. Jones & H. Thimbleby, "'Lost in Hyperspace': Psychological Problem or Bad Design," in *First Asia Pacific Conference on Human Computer Interaction*, L. K. Yong, L. Herman, Y. K. Leung & J. Moyes, eds., 387–396, 1996.

[53] H. W. Thimbleby & I. H. Witten, "User Modelling as Machine Identification: New Methods for HCI," *Advances in Human-Computer Interaction*, H. R. Hartson & D. Hix, eds., **IV**:58–86, 1993.

[54] H. W. Thimbleby, "Visualising the Potential of Interactive Systems," *The 10th. IEEE International Conference on Image Analysis and Processing*, ICIAP'99, 670–677, 1999.

[55] H. S. Wilf, *Algorithms and Complexity*, Internet Edition, `http://www.math.upenn.edu/~wilf/AlgComp.pdf`, 1994.

[56] R. J. Wilson & J. J. Watkins, *Graphs: An Introductory Approach*, John Wiley & Sons, 1990.

[57] S. Wolfram, *The Mathematica Book*, 4th. ed., Addison-Wesley, 1999.

[58] J. Ziman, *Reliable Knowledge*, Canto ed., Cambridge University Press, 1991.

A. JAVA CODE

The main phases of the algorithm are called as methods, discussed in sections in the body of the paper: §5.2 (least cost paths and checking the graph); §5.3 (finding a feasible solution and iterating); §5.4 (printing the tour).

```
public void cpp()
{ checkInitialised();
  leastCostPaths();
  checkValid();
  findFeasible();
  while( improvements() );
  printCPT(0); // a tour starting from 0
}
```

Here is how the code would be used to find a CPT for the example used in the main body of the paper (the method `addedge` is described later). Note how vertices are numbered from zero, as is conventional in Java.

```
Graph G = new Graph(4);
G.addedge("a", 0, 1, 1).addedge("b", 0, 2, 1)
 .addedge("c", 1, 2, 1).addedge("d", 1, 3, 1)
 .addedge("e", 2, 3, 1).addedge("f", 3, 0, 1);
G.cpp();
```

. . . and the output from running the example is as follows:

```
Take edge b from 0 to 2
  Take path from 2 to 0:
    Take edge e from 2 to 3
    Take edge f from 3 to 0
Take edge a from 0 to 1
Take edge c from 1 to 2
Take edge e from 2 to 3
  Take path from 3 to 1:
    Take edge f from 3 to 0
    Take edge a from 0 to 1
Take edge d from 1 to 3
Take edge f from 3 to 0
```

We check the graph is strongly connected and has no negative cycles: this ensures there is an optimal CPT.

```
private void checkValid()
{ for( int i = 0; i < n; i++ )
    if( c[i][i] < 0 )
      throw new Error("Negative cycles");
    else for( int j = 0; j < n; j++ )
      if( !defined[i][j] )
        throw new Error("Not strongly connected");
}
```

A modified Floyd-Warshall algorithm is used, with a check to terminate it early if any negative cycle is found.

```
private void leastCostPaths()
{ for( int k = 0; k < n; k++ )
    for( int i = 0; i < n; i++ )
      if( defined[i][k] )
        for( int j = 0; j < n; j++ )
```

```
        if( defined[k][j]
            && (!defined[i][j]
                || c[i][j] > c[i][k]+c[k][j]) )
        { defined[i][j] = true;
          path[i][j] = path[i][k];
          c[i][j] = c[i][k]+c[k][j];
          // return on negative cycle
          if( i == j && c[i][j] < 0 )
            return;
        }
}
```

The greedy method is used to establish an initial integral feasible solution. The arrays `neg` and `pos` represent the sets D^- and D^+ of the derivation in the main part of the paper in §4.

```
private void findFeasible()
{ int nn = 0, np = 0;
  for( int i = 0; i < n; i++ )
    if( degree[i] < 0 ) nn++;
    else if( degree[i] > 0 ) np++;

  neg = new int[nn];
  pos = new int[np];
  nn = np = 0;
  for( int i = 0; i < n; i++ )
    if( degree[i] < 0 ) neg[nn++] = i;
    else if( degree[i] > 0 ) pos[np++] = i;

  for( int u = 0; u < nn; u++ )
  { int i = neg[u];
    for( int v = 0; v < np; v++ )
    { int j = pos[v];
      f[i][j] = -degree[i] < degree[j]?
                -degree[i]: degree[j];
      degree[i] += f[i][j];
      degree[j] -= f[i][j];
    }
  }
}
```

If a negative cycle can be found in the residual graph, cancel it — and return `true` so that the main routine (`cpp`) repeats the iteration. We create a residual graph `R` of size `n`, which will create an adjacency matrix of size $\mathtt{n}^2$, but in fact a matrix of size $(\mathtt{neg.length} + \mathtt{pos.length})^2$, usually smaller, would be sufficient — except we would have to map subscripts to the ranges of the smaller matrix.

```
private boolean improvements()
{ Graph R = new Graph(n);
  for( int u = 0; u < neg.length; u++ )
  { int i = neg[u];
    for( int v = 0; v < pos.length; v++ )
    { int j = pos[v];
      if( edges[i][j] > 0 )
        R.addedge(null, i, j, c[i][j]);
      if( f[i][j] != 0 )
        R.addedge(null, j, i, -c[i][j]);
    }
  }
  // find a negative cycle
  R.leastCostPaths();
  // cancel the cycle (if any)
  for( int i = 0; i < n; i++ )
    if( R.c[i][i] < 0 )
    { int k = 0, u, v;
      boolean kunset = true;
      u = i; do // find k to cancel
      { v = R.path[u][i];
        if( R.c[u][v] < 0
            && (kunset || k > f[v][u]) )
        { k = f[v][u];
          kunset = false;
        }
      } while( (u = v) != i );
      u = i; do // cancel k along the cycle
      { v = R.path[u][i];
        if( R.c[u][v] < 0 ) f[v][u] -= k;
        else f[u][v] += k;
      } while( (u = v) != i );
      return true; // have another go
    }
  return false; // no improvements found
}
```

Print an optimal Chinese Postman Tour.

```
private void printCPT(int start)
{ printPath: for( int v = start;;)
  { int u = v;
    // find an adjoined path, if any
    for( int i = 0; i < n; i++ )
      if( f[u][i] > 0 )
      { v = i;
        f[u][v]--;
        System.out.println(
          "  Take path from "+u+" to "+v+": ");
        for( int p; u != v; u = p )
        { p = path[u][v];
          System.out.println("    Take edge "
            +label[u][p]
             .elementAt(cheapestEdge[u][p])
            +" from "+u+" to "+p);
        }
        continue printPath;
      }
    // find an existing edge, using bridge last
    v = -1;
    for( int i = 0; i < n; i++ )
      if( edges[u][i] > 0 )
        if( v == -1 || i != path[u][start] )
          v = i;
    // finished when no more bridges
    if( v == -1 )
      return;
    // note how this uses each edge in turn
    System.out.println("Take edge "
      +label[u][v].elementAt(edges[u][v]-1)
      +" from "+u+" to "+v);
    // remove edge that's been used
    edges[u][v]--;
  }
}
```

Graphs are most easily constructed by adding edges. (Some tedious complexity in the code arises from keeping track of multiple labels for parallel edges; otherwise we only need to know, for any pair of vertices, the number of edges and the cheapest cost.)

```
Graph addedge(String lab, int u, int v, float cost)
{ if( !defined[u][v] )
    label[u][v] = new Vector();
  label[u][v].addElement(lab);
  if( !defined[u][v] || c[u][v] > cost )
  { c[u][v] = cost;
    cheapestEdge[u][v] = edges[u][v];
    defined[u][v] = true;
    path[u][v] = v;
  }
  edges[u][v]++;
  degree[u]++;
  degree[v]--;
  return this;
}
```

Having defined all the relevant methods, now we give the rest of the Java file, the class definition including the class fields and initialisers. All the methods described above are defined within this class.

Note that initialisation in Java sets booleans to `false` and integers to zero. This is relied on: all edges are initially undefined (i.e., `defined` for all edges is `false`), `edges`, `neg` and `pos` are zero, and so on.

```
import java.io.*;
import java.awt.*;
import java.util.*;

public class Graph
{ private int n; // number of vertices
  private int degree[]; // degrees of vertices
  private int neg[], pos[]; // unbalanced vertices
  private int path[][], edges[][],
    cheapestEdge[][], f[][];
  private boolean defined[][];
  private Vector label[][]; // names of edges
  private float c[][];

  Graph(int vertices)
  { n = vertices;
    degree = new int[n];
    defined = new boolean[n][n];
    label = new Vector[n][n];
    c = new float[n][n];
    f = new int[n][n];
    edges = new int[n][n];
    cheapestEdge = new int[n][n];
    path = new int[n][n];
    initialised = true;
  }

   ⋮
  // Other methods are described
  // elsewhere in this Appendix
   ⋮
}
```

Finally note that, for simplicity, as currently defined the code has two minor restrictions:

- To avoid the complexity of changing the size of the vectors and matrices dynamically, a new `Graph` is created with a fixed size. This means that, although a graph is defined by its edges, it has to be specified by first saying how many vertices it has — this might be tedious if we wanted to create a graph while walking a web site of initially unknown size. (A dynamic array can easily be subclassed from Java's `Vector`, as one solution, but doing so would take our code into programming Java rather than providing a clear CPP algorithm.)
- The code corrupts the degree and edge counts *in situ* and therefore it cannot be repeatedly invoked without reinitialising the graph. Thus the method below checks the graph is initialised. The corruption is easy to fix at the expense of adding more code: solutions include working on copies of the arrays (which could be made in the `checkInitialised()` routine), or re-establishing their correct values at termination, *etc.*

```
private boolean initialised;

private void checkInitialised()
{ if( !initialised )
    throw new Error("Graph not initialised");
  // the algorithm destroys some fields
  initialised = false;
}
```

B. POSSIBLE OPTIMISATIONS, ETC.

The Java code is deliberately simple, and provides many opportunities to be optimised; however many optimisations would make the code longer and more obscure, and are therefore beyond the scope and interest of this paper.

The least cost paths method itself detects negative cycles, but both places where it is used there is a secondary check to find negative cycles. A simple optimisation would be to assume that all costs are less than some huge value: the field `defined` could then be disposed of — by using costs equal to the huge value to denote undefined edges.

The residual graph is constructed conveniently the same size as the original graph, but it could be smaller — this issue is discussed above.

A common inefficiency in the code arises from the use of simple linear searches throughout, for example in searching for edges when the tour is constructed. There are many ways to improve the speed, for instance by

constructing lists of out-edges by vertex when the cost matrix is initialised.

The most sophisticated inefficiency is caused by the choice of negative cycle to cancel. As it happens, the code searches for a negative cycle sequentially and stops at the first one it finds, but there are better schemes for making the choice [28]. Note that such algorithms are applied to our (smaller) derived graph $G^\star$, not to the initial (larger) problem G, and therefore their improved orders of magnitude of time complexity are less significant than might be expected — and may be overcome by higher overheads.

C. HOW THE JAVA CODE WAS FORMATTED

Presenting accurate code has been a major concern of this paper. A systematic approach was taken to ensure the integrity of the typeset code and examples used.

Java source code was marked up using comments of the form

```
// $save$ filename
```

which was processed (by a simple Java filter) to copy text on subsequent lines to the specified file, adhering to LaTeX typesetting conventions (e.g., converting `&` into `\&`). Using several such comments thus split the Java class file into several LaTeX files.

The LaTeX file that is this paper then included the processed Java code directly from the generated files. Thus the sections of Java code in Appendix A are *exactly* the code that was debugged and used for running the example. In fact the example output was also processed automatically: it was prepended with "`// $save$ example.tex`" and the resulting output file was processed just like the Java source files (and therefore conveniently converted to LaTeX).

This approach is a simplification of *literate programming* [35] with the advantage that the source file is conventional, pure Java rather than a literate programming document, which would require the availability of other tools to process to extract compilable Java.

The approach taken has benefitted from the advantages claimed for literate programming: a minor typo in the Java was corrected as a result of the ease of proofreading the LaTeX formatting juxtaposed with the English descriptions.

D. ALTERNATIVE CORE ALGORITHMS

We showed how to reduce the core of CPP to integer linear programming in Section 4. With a suitable library routine available, this will be sufficient to proceed for this part of the problem. For example, the simplex algorithm provides integer solutions in this case [36]. However, if a library routine is used, we may not know exactly what algorithm is used, so we cannot guarantee obtaining integer solutions (see the example used in Section 5.1, which shows integer solutions are not guaranteed), though simple iterative search techniques can be used [37] for converging to integer solutions.

If we were only interested in the deficiency (ϕ) nonintegral solutions for $f^\star$ would not matter, but for finding the CPT integers are required and unfortunately not all linear programming routines necessarily find the integer solutions. Fortunately it is easy to show that the simplex method guarantees integer solutions in this case [36], as would using appropriate network algorithms [18] (in general, network flow need not provide integer solutions).

Note that the linear programming problem is simpler than it looks. Because of the given relation of the δs, that the degrees of the unbalanced vertices sum to zero, one constraint on the sums of $f^\star$ is redundant. Furthermore the $f^\star_{ij}$ variable notation makes it look like there are always $|V|^2$ variables to deal with, since both i and j range over all the vertices of G. There are indeed $|V|$ vertices, so apparently $|V|^2$ variables, but in general some vertices will be balanced (their $\delta(v)$ will be zero), so the sums of $f^\star$ in or out of these vertices will be zero; thus, since $f^\star \geq 0$ by definition, some of the constraints may trivially make some $f^\star$ zero. In fact, as we only consider paths from D^- to D^+ no paths are added just between vertices from the same set. Specifically, there are at most $|D^-| \times |D^+|$ non-zero variables, and there will always be no more than $|V|^2/4$, since $|D^-| + |D^+| \leq |V|$.

The core part of the algorithm is a linear programming problem, a transportation problem, a minimisation problem ... in fact, it is a generic problem, for which many algorithms are known [2]. We now review some alternative approaches to the cycle canceling algorithm used in §5.3 and Appendix A.

D.1. Matching

A *bipartite* graph is a graph that can be partitioned into two sets of vertices, where edges only connect vertices in the different sets. A *matching* of a graph is a set of edges that share no vertex. A *perfect bipartite matching*, then, is a set of edges that connect the vertices in each partite set one-to-one. If the edges are weighted, then of all possible matchings, some will be of minimal total cost.

There are two alternative ways to convert the linear programming to a minimum weight perfect bipartite matching problem (also known as minimum weight assignment problem).

- A bipartite graph is constructed that has its two sets of vertices corresponding to D^- and D^+, with vertex v is repeated $|\delta(v)|$ times so that a perfect matching is possible.
- A bipartite graph is constructed that has its two sets of vertices corresponding to D^- and D^+, with each vertex appearing once. A perfect matching

then finds a $(0,1)$ solution for f. The largest possible multiple of this solution is subtracted from the δ constraints on f, and the process repeated until the outstanding constraints are zero.

In either case, the weight of each edge is given by the cost of the corresponding least cost paths in the original graph G.

Having constructed a bipartite graph, the matching problem can be solved directly. For example, Knuth [32] provides exemplary (complete and fully documented) C code for the Hungarian algorithm. Kolman and Beck [36] provide a particularly clear explanation of this algorithm. The Hungarian algorithm also has elegant properties that makes it ideal for programming — invariants can be checked to assure that the program is correct [30]. Indeed the algorithm is one of the most popular and important algorithms in combinatorics (see, e.g., [30]) — so it is worth covering for educational reasons alone.

D.2. Network flow

The problem can be converted to a network flow problem as follows. If a multiple source/sink algorithm is available, simply treat each vertex in D^- as a source and each vertex in D^+ as a sink, and assign all edges a capacity the appropriate $\sum\delta$. Otherwise, in case only a single source and sink can be handled, add a 'source' vertex connected by edges to each vertex corresponding to D^-, and add a 'sink' vertex connected by edges from each vertex corresponding to D^+.

We now require the minimum cost flow through the graph at given flow levels. The flow level finds a perfect matching; the minimum cost minimises the CPT cost.

Sources to obtain code (in Pascal, C++, etc) for min-cost max-flow implementations are given by [51]; the Ford-Fulkerson algorithm is discussed in [2, 24, 27], and [55] gives a particularly clear discussion of its use for finding bipartite matchings.

D.3. Special cases

When all adjoined path costs are equal — as may easily occur in the rural CPT — (and in certain other cases), the problem can be solved by a greedy method. (In Section 4 we showed a greedy method can find a feasible solution; and from Section 5.3, if any improvements are possible there will be an alternating cycle of positive and negative costs with an overall negative sum — but when all the costs are equal the sum of any alternating cycle is zero. Thus any feasible solution for a problem with equal costs is an optimal solution.)

In the special case that costs are zero (cf. [23]), we may wish to pretend the costs are 1 to better distribute the use of edges in the CPT.