The directed Chinese Postman Problem

Harold Thimbleby

UCLIC, University College London Interaction Centre, 26 Bedford Way, London, WC1H 0AP.
Email: h.thimbleby@ucl.ac.uk

SUMMARY

The Chinese Postman Problem has many applications, including robot exploration, and analysing interactive system and web site usability. This paper reviews the wide range of applications of the problem and presents complete, executable code to solve it for the case of directed multigraphs. A variation called the 'open Chinese Postman Problem' is also introduced and solved. Although optimisations are possible, no substantially better algorithms are likely.

KEY WORDS: Directed Chinese Postman Problem, Java, Usability Evaluation.

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* (CPP) is interesting because it has many applications, is a simply-stated problem, but for which there is no simple algorithm. There are many variations to the CPP, most notably whether the roads are one-way (this is the Directed CPP or DPP) and whether the postman has to return back to where they started (closed or open CPP). This paper is concerned specifically with the directed CPP, and provides algorithms for both closed and open solutions.

Although many pseudo-code descriptions of the CPP exist (e.g., [10]), no executable algorithm for it is available [18]. A typical reference says, "The details of the algorithm are too complicated to give here" [3]. Further examples from the literature are given in [22], but that is not the concern here — this paper provides and explains executable Java to solve the problem, and hence makes the algorithm and its application accessible to a wide audience. The code is given in its entirety in this paper (it was extracted automatically from the original source code [22]), and is also available from the web site `http://www.uclic.ucl.ac.uk/harold/cpp`, which provides code in both Java and *Mathematica*.

The aim of this paper is to motivate and exhibit a clear working algorithm rather than a commercial or especially efficient algorithm. However, our implementation of the CPP is

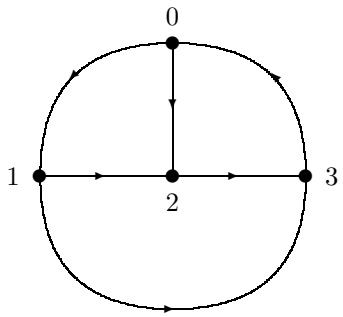

Figure 1. Taking each arc to be of equal weight, one optimal closed Chinese Postman Tour of this graph is $0, 1, 3, 0, 1, 2, 3, 0, 2, 3, 0$, traversing 10 arcs. A least cost open tour is $1, 3, 0, 1, 2, 3, 0, 2$, which traverses only 7 arcs.

elegant and interesting: the preliminary phase combines checking (to confirm it is given a valid problem) with initialising a data structure, which is then exploited in different ways in the subsequent phases of the implementation; we also use a single shortest paths routine for three different purposes.

2. Applications and variations of the Chinese Postman Problem

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 their office. This is a graph theoretic problem: roads are directed edges (arcs), and road junctions are vertices. The postman requires a *Chinese Postman Tour*, which we abbreviate CPT [9]. The postman probably wants a shortest tour, with few repeated street visits. The *cost* of a CPT is defined as the total arc 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 could arbitrarily repeat it and get a total cost lower and lower without bound.

The CPP has many applications from analysing DNA [16] to routing robots. Conventional applications of the CPP are concerned with routing more generally than postmen, as in routing snow ploughs or planning street maintenance. Figure 1 shows a simple, isolated village, with four intersections and six one-way streets. The least cost CPT in this example requires travelling down streets ten times; some streets are used more than once, but this is not always the case, and the optimal solution obviously depends on the relative costs of using each arc.

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 unexpected behaviour of the mobile. The length of the optimal CPT is therefore a measure of a machine's complexity [19].

As a concrete example, the Nokia 2110 mobile phone has a menu subsystem of 88 menu-items and 273 actions [12]; the optimal CPT of this takes 515 button presses, plus 79 presses (done at appropriate points) to check presses that do nothing, a total of 594 test operations. (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 [21]).

Web sites notoriously have broken links; but possibly worse is a link that takes the user to the wrong page. A web author should check every link of a site for correctness, certainly if it provides medical or legal information. 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. An optimal CPT gives a route around a web site (or other multimedia resource) that exercises every link, with minimal effort.

For well designed web sites, it is not necessary to check every link explicitly. The web site for Benjamin Franklin's House [11] had 66 pages and 1191 links at the time of writing. Its optimal CPT of 2248 steps is excessive for a human to follow unaided; without following a CPT users would do even more work and be unable to guarantee thorough checking. Ideally authoring systems should provide mechanisms to help check sites (for instance, so that the human checker can take a break and not lose track of where they were), in conjunction with using an optimal CPT algorithm to minimise workload. In fact, the Benjamin Franklin site was generated by compiling 78 pages (12 being design templates) needing only 201 explicit links (most being checked by the compiler itself). The CPT of this specification site had 241 steps, including the links the compiler can check itself [20]. The tenfold improvement makes checking the site manageable for a human.

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 arc 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 arcs, an algorithm has been found that takes at most $m\phi^{O(\log \phi)}$ steps [2], 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 important variations to the CPP. The problem may be closed, with the postman starting *and* returning to their office; in the open problem, the postman need not return. In the real world, where postmen or vehicles traverse a network of roads, this may not seem very useful because they will eventually need to return home; however, in testing once every transition has been tested it may not be necessary to return the device to its initial conditions. This is certainly true for testing web sites: once a user has tested a site, they can simply walk away and they do not have to return the browser to the home page. The optimal test sequence for a web site is therefore an open CPT.

If the open tour of an interactive device is much shorter than a closed tour, this suggests a possible design fault because it indicates that a tester is (and users in general are) better off walking away from a device than switching it off (or otherwise returning it to its initial state).

Orloff [13] 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. Further, there may be one or more postmen, with one or more offices, and each postman may be capacitated (e.g., in the total edge cost they can bear).

The *selecting CPP* (also known as the *rural CPP*) is a variation where the postman must visit certain roads but not necessarily all of them [15]. 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 [5, 17]. 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 where others, the ones to be checked, were created by hand.

The graph may be undirected, directed or mixed. Computationally, the undirected and directed cases are polynomial, whereas the mixed is NP-hard [14]. This paper considers the directed case on multidigraphs (graphs with parallel arcs): this case arises naturally in analysing the World Wide Web and finite state machines.

3. Definitions and outline of the algorithm

Discussion of graph-theoretic terms may be found in standard graph theory references: any of [1, 3, 6] cover everything required here.

In this paper we will consider graphs as collections of arcs $\langle label, i, j, c\rangle$, where *label* is an identifier for an arc from vertex i to j, and c the cost associated with it. This suits the problem of checking web sites: a vertex is a page, an arc is a link, the label of the arc could be hot text or a URL, 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. The goal is to determine a list of labels that, in order, constitute an optimal CPT.

The number of arcs going into a vertex v is the *in-degree* written $d^-(v)$, and the number of arcs pointing out of a vertex v is the *out-degree* written $d^+(v)$. Let δ be the difference between the in and out degrees: $\delta(v) = d^+(v) - d^-(v)$. If $\delta(v) = 0$, we say the vertex v is *balanced*.

An *Eulerian graph* is one that has a circuit traversing each arc *exactly* once, and which returns to the start vertex. A standard theorem is that a graph has an Euler circuit if and only if every vertex is balanced. Finding an Euler circuit has standard algorithms, and certain classes of Eulerian graph have trivial solutions, notably *randomly Eulerian* graphs [4]. An Euler circuit of a graph is an optimal CPT, since each arc is traversed exactly once. If a graph is not Eulerian, then more work must be done to find a CPT.

Let D^+ be the set of unbalanced vertices with an excess of out-going arcs, and D^- the set of unbalanced vertices with an excess of in-going arcs. More formally:

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

In the example graph, we have $D^+ = \{0, 1\}$ and $D^- = \{2, 3\}$, and the δ for each vertex happens to be either $+1$ or -1. Thus the CPT will have to walk extra paths to 'join up' the unbalanced vertices in D^- and D^+. For an optimal CPT, the specific choice of extra paths taken between D^- and D^+ will be chosen to have least total cost.

It is useful to have some notation. Let $i \rightsquigarrow j$ denote a least cost path from vertex i to j, and let c_{ij} be its cost. If there are multiple arcs between adjacent vertices, the least cost path will obviously take the cheapest arc of those available. We use (i, j) to denote, specifically, the cheapest arc from vertex i to j.

For the example graph, there are two ways to choose the set of extra paths. If one path is $2 \rightsquigarrow 0$, then the other path is $3 \rightsquigarrow 1$; the alternative is to use the paths $2 \rightsquigarrow 1$ and $3 \rightsquigarrow 0$. As it happens, in this example the choices have equal cost ($c_{21} + c_{30} = 3 + 1$, $c_{20} + c_{31} = 2 + 2$), and either can be used for an optimal CPT. We can now imagine the original graph augmented with the two new paths as new arcs: all vertices will be balanced and the result is an Eulerian graph, and *its* Eulerian circuit, which will be an optimal CPT of the original, can be obtained by a standard algorithm.

In the example, two new paths are required; in general k extra paths are required, where $k = \sum_{v \in D^+} \delta(v)$. The first of these k paths might go to any of the (at most) k vertices in D^+, the second can to any of the remaining $k - 1$, and the third to any of the remaining $k - 2$, and so on. In the worst case there are $k!$ choices, and clearly it would be an inefficient algorithm that examined all the choices and picked the least cost; instead more efficient approaches can be used.

In general a CPT may take some of the $D^- \rightsquigarrow D^+$ paths more than once. Let f_{ij} be the number of times the path $i \rightsquigarrow j$ must be taken — specifically, how many times the path must be added to the graph as an arc to make it Eulerian. We are now in a position to express the optimal CPT problem formally: finding the optimal tour amounts to finding how to minimise its excess cost, $\phi = \sum c_{ij} f_{ij}$, which is the additional cost of traversing the chosen additional paths between the unbalanced vertices. The total cost of the optimal CPT will be ϕ plus the total of each original arc weight, since every arc has to be traversed at least once in any case.

Original graph G (as list of arcs)	$(0,1),(0,2),(1,2),(1,3),(2,3),(3,0)$
Vertex δs	$\delta(0)=1,\ \delta(1)=1,\ \delta(2)=-1,\ \delta(3)=-1$
D^-	$\{2,3\}$
D^+	$\{0,1\}$
Variables	$f_{20}, f_{21}, f_{30}, f_{31}$
ϕ	$2f_{20}+3f_{21}+f_{30}+2f_{31}$
Constraints	$f_{20}+f_{30}=1,\ f_{21}+f_{31}=1,$
	$f_{20}+f_{21}=1,\ f_{30}+f_{31}=1$
Integer solution minimising ϕ	$f_{20}=0,\ f_{21}=1,\ f_{30}=1,\ f_{31}=0$
Extra paths to adjoin	$f_{ij}\times i\rightsquigarrow j=\{2\rightsquigarrow 1, 3\rightsquigarrow 0\}$
Adjoined to make Eulerian graph	$(0,1),(0,2),(1,3),(1,2),(2,3),(3,0),(2,1),(3,0)$
Euler circuit	$2,1,3,0,2,3,0,1,2$
Least cost path for arcs not in G	$2\rightsquigarrow 1\mapsto 2,3,0,1$
... splice in to make CPT of G	$2,3,0,1,3,0,2,3,0,1,2$

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

(If each arc weight is 1, ϕ is the least number of arcs that need to be added to make the graph Eulerian; this is the *deficiency* of the graph, as required for the robot exploration problem in §2.)

The algorithm can now be sketched mathematically:

$$\textbf{determine from the graph}: \quad \delta, D^-, D^+, c$$

$$\textbf{find } f \textbf{ to minimise}: \quad \sum c_{ij} f_{ij}$$

$$\textbf{subject to}: \quad \begin{cases} f_{ij} & & \text{integer} \\ f_{ij} & \geq & 0 \\ \sum_{j\in D^+} f_{ij} & = & -\delta(i) \\ \sum_{i\in D^-} f_{ij} & = & \delta(j) \end{cases}$$

$$\textbf{construct}: \quad \text{Eulerian circuit, by using least cost paths } i\rightsquigarrow j, \text{ each path repeated } f_{ij}>0 \text{ times}$$

The code we give below (§4 onwards) also includes error checking. Figure 2 summarises the calculations relevant for the example graph.

4. Java code for the closed CPP

An optimal CPT for the simple graph of Figure 1 can be found using the code developed in this paper:

```
CPP G = new CPP(4); // create a graph of four vertices

// add the arcs for the example graph
// addArc(<arc label>, <from vertex>, <to vertex>, <weight>
G.addArc("a", 0, 1, 1).addArc("b", 0, 2, 1).addArc("c", 1, 2, 1)
 .addArc("d", 1, 3, 1).addArc("e", 2, 3, 1).addArc("f", 3, 0, 1);

G.solve(); // find the CPT
G.printCPT(0); // print it, starting from vertex 0
System.out.println("\nCost = "+G.cost());
```

Here is the output:

```
Take arc b from 0 to 2
Take arc e from 2 to 3
Take arc f from 3 to 0
Take arc a from 0 to 1
Take arc c from 1 to 2
Take arc e from 2 to 3
Take arc f from 3 to 0
Take arc a from 0 to 1
Take arc d from 1 to 3
Take arc f from 3 to 0

Cost = 10.0
```

4.1. Class outline

The CPT algorithm is defined in a Java class `CPP`, but our procedural style of programming can be translated to languages such as C or Pascal directly.

After creating a new graph and defining its arcs (including their labels and weights), the main phases of the algorithm are called in sequence using the method `solve`, shown below, after which calling `printCPT(startVertex)` will print an optimal CPT starting from any vertex.

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

public class CPP
{   int N; // number of vertices
    int delta[]; // deltas of vertices
    int neg[], pos[]; // unbalanced vertices
    int arcs[][]; // adjacency matrix, counts arcs between vertices
    Vector label[][]; // vectors of labels of arcs (for each vertex pair)
    int f[][]; // repeated arcs in CPT
    float c[][]; // costs of cheapest arcs or paths
    String cheapestLabel[][]; // labels of cheapest arcs
    boolean defined[][]; // whether path cost is defined between vertices
    int path[][]; // spanning tree of the graph
    float basicCost; // total cost of traversing each arc once

    void solve()
    {   leastCostPaths();
        checkValid();
        findUnbalanced();
```

```
        findFeasible();
        while( improvements() );
    }

    ⋮
    // Other declarations are described below
    ⋮
}
```

We use a simple adjacency matrix represention of graphs: `arcs` counts the number of parallel arcs between each pair of vertices, along with subsidiary structures, such as vertex deltas that are needed for the algorithm. To avoid the complexity of changing the size of vectors and matrices dynamically, a `CPP` graph is instantiated with a fixed size:

```
// allocate array memory, and instantiate graph object
CPP(int vertices)
{   if( (N = vertices) <= 0 ) throw new Error("Graph is empty");
    delta = new int[N];
    defined = new boolean[N][N];
    label = new Vector[N][N];
    c = new float[N][N];
    f = new int[N][N];
    arcs = new int[N][N];
    cheapestLabel = new String[N][N];
    path = new int[N][N];
    basicCost = 0;
}
```

Initialisation in Java sets booleans to `false` and integers to zero: all elements of `delta`, `arcs`, `neg` and `pos` are initially zero. Hence when a graph is first created it has `N` vertices and no arcs. Graphs then are constructed by adding arcs. Some complexity in the code arises from keeping track of multiple labels for parallel arcs; otherwise we only need to know for any pair of vertices the number of arcs and the cheapest cost.

```
CPP addArc(String lab, int u, int v, float cost)
{   if( !defined[u][v] ) label[u][v] = new Vector();
    label[u][v].addElement(lab);
    basicCost += cost;
    if( !defined[u][v] || c[u][v] > cost )
    {   c[u][v] = cost;
        cheapestLabel[u][v] = lab;
        defined[u][v] = true;
        path[u][v] = v;
    }
    arcs[u][v]++;
    delta[u]++;
    delta[v]--;
    return this;
}
```

The final line, `return this`, allows `addArc` calls to be strung together conveniently.

4.2. Find shortest paths and costs

The Floyd-Warshall algorithm [18] can efficiently find and record shortest paths at the same time as establishing the costs, c, by building a matrix P (called, more explicitly, `path` in the Java) such that the first arc of the least cost path $u \rightsquigarrow v$ from vertex u to vertex v is (u, P_{uv}). Subsequent arcs along the path are found similarly; if (u, P_{uv}) is an arc to vertex $w \neq v$ then (u, P_{wv}) is the next arc to take towards v, and so on.

Since the matrix P defines paths from every vertex to any vertex v, it also defines spanning trees $\{(u, P_{uv}) \colon u \neq v\}$, each one rooted at a different vertex v. We will use this fact later.

The standard algorithm is modified to terminate if any negative cycle is found: because it terminates before possibly finding further cycles through the same vertex, the cycle is correctly defined by recording a single value in `path` — this fact is used in §4.5.

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

4.3. Check valid

Checking that a graph is strongly connected with no negative weight cycles is established by inspecting the shortest path costs. If some element of c is undefined, the graph is not connected. For any vertex i, c_{ii} is the cost of a least cost cycle including that vertex; so a graph with negative weight cycles will have $c_{ii} < 0$ for some i.

```
void checkValid()
{   for( int i = 0; i < N; i++ )
    {   for( int j = 0; j < N; j++ )
            if( !defined[i][j] ) throw new Error("Graph is not strongly connected");
        if( c[i][i] < 0 ) throw new Error("Graph has a negative cycle");
    }
}
```

4.4. Find unbalanced vertex sets

The vectors `neg` and `pos` represent the sets D^- and D^+, and they are built straight forwardly by examining `delta`, the δ of each vertex.

```
void findUnbalanced()
{   int nn = 0, np = 0; // number of vertices of negative/positive delta

    for( int i = 0; i < N; i++ )
```

```
        if( delta[i] < 0 ) nn++;
        else if( delta[i] > 0 ) np++;

    neg = new int[nn];
    pos = new int[np];
    nn = np = 0;
    for( int i = 0; i < N; i++ ) // initialise sets
        if( delta[i] < 0 ) neg[nn++] = i;
        else if( delta[i] > 0 ) pos[np++] = i;
}
```

4.5. Solve the optimisation problem

The optimisation problem may be solved in various ways. Care must be taken if a general purpose linear programming method is used: the example problem has a solution $f_{20} = f_{21} = f_{30} = f_{31} = 1/2$ where ϕ has the optimal value of 4 and the constraints are still satisfied *but* the f_{ij} are not integral. This is a problem we must avoid, since non-integral f are not acceptable: a valid CPT solution requires f be integral. Matching is one approach, and can be done efficiently in polynomial time, though no simple algorithm is available — Knuth's C code [8] implements the Hungarian algorithm and runs to about 9 book pages, 16 including comments. Other approaches are also long and tedious to program, though some code is freely available on the Internet. The min-cost max-flow algorithm is available in source form [18]; the Ford-Fulkerson algorithm is discussed in [1, 6].

Cycle canceling [7] can reuse the shortest path results already found, and we use it because it is concise. Cycle canceling starts with an approximate solution, then iteratively improves it. Here, the greedy method is used to establish an initial feasible solution: when the method terminates, the variable `f` satisfies the constraints but ϕ may not yet be optimal.

```
void findFeasible()
{   // delete next 3 lines to be faster, but non-reentrant
    int delta[] = new int[N];
    for( int i = 0; i < N; i++ )
        delta[i] = this.delta[i];

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

Notice how the class variable `delta` is copied to a local variable. As used in this paper, there is no requirement for any code to be re-entrant, but it is nevertheless good practice! The few lines that make the copy can be deleted if performance is at a premium. (Or Java's efficient `System.arraycopy` can be used.)

Cycle canceling uses a residual graph, and 'cancels' negative cycles in it. The `improvements` method returns `true` into the `while` loop of the `solve` method so it will be invoked repeatedly

while improvements are possible. The loop terminates when no further improvements are possible.

```
boolean improvements()
{   CPP residual = new CPP(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];
            residual.addArc(null, i, j, c[i][j]);
            if( f[i][j] != 0 ) residual.addArc(null, j, i, -c[i][j]);
        }
    }
    residual.leastCostPaths(); // find a negative cycle
    for( int i = 0; i < N; i++ )
        if( residual.c[i][i] < 0 ) // cancel the cycle (if any)
        {   int k = 0, u, v;
            boolean kunset = true;
            u = i; do // find k to cancel
            {   v = residual.path[u][i];
                if( residual.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 = residual.path[u][i];
                if( residual.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
}
```

Java's labelled `continue` command could have been used to avoid the separation of the control of the looping (in `solve()`) from the test to continue (commented 'have another go,' above); however, our coding style here is easier to translate to languages like C.

4.6. Cost

The cost of an optimal CPT is obtained straight forwardly:

```
float cost()
{   return basicCost+phi();
}

float phi()
{   float phi = 0;
    for( int i = 0; i < N; i++ )
        for( int j = 0; j < N; j++ )
            phi += c[i][j]*f[i][j];
    return phi;
}
```

4.7. Print the optimal CPT

We now have an optimal assignment to f, and can use this to implicitly construct a Eulerian graph. The optimal CPT is a Euler circuit of this graph.

An algorithm for following Euler circuits uses a spanning tree: to follow a circuit, take any arc from a vertex, but use the spanning tree arc last [6]. The matrix `path` found in §4.2 defines a spanning tree.

When at any vertex i on the tour, if $f_{ij} > 0$, follow the shortest path $i \rightsquigarrow j$. Since f counts how many times paths should be followed, decrement f_{ij} as each path is followed. Otherwise take an arc other than an arc in the spanning tree, if possible, else use the last arc (called a *bridge*) from the spanning tree. Since we allow graphs to have parallel arcs (with different labels), as each arc is traversed the arc count is decremented so that on the next traversal a different arc label will be used.

Calling `printCPT` requires prior initialisation with the `solve` method. Note that `printCPT` makes a local copy of `arcs` and `f` so that the original graph data is not erased as its CPT is printed. The method terminates when it visits a vertex where the bridge has already been used; this can only happen when the path completes the circuit on returning to the start vertex (this need not be the first time the circuit returns to the start vertex).

```
static final int NONE = -1; // anything < 0

int findPath(int from, int f[][]) // find a path between unbalanced vertices
{   for( int i = 0; i < N; i++ )
        if( f[from][i] > 0 ) return i;
    return NONE;
}

void printCPT(int startVertex)
{   int v = startVertex;

    // delete next 7 lines to be faster, but non-reentrant
    int arcs[][] = new int[N][N];
    int f[][] = new int[N][N];
    for( int i = 0; i < N; i++ )
        for( int j = 0; j < N; j++ )
        {   arcs[i][j] = this.arcs[i][j];
            f[i][j] = this.f[i][j];
        }

    while( true )
    {   int u = v;
        if( (v = findPath(u, f)) != NONE )
        {   f[u][v]--; // remove path
            for( int p; u != v; u = p ) // break down path into its arcs
            {   p = path[u][v];
                System.out.println("Take arc "+cheapestLabel[u][p]
                    +" from "+u+" to "+p);
            }
        }
        else
        {   int bridgeVertex = path[u][startVertex];
            if( arcs[u][bridgeVertex] == 0 )
                break; // finished if bridge already used
```

```
            v = bridgeVertex;
            for( int i = 0; i < N; i++ ) // find an unused arc, using bridge last
                if( i != bridgeVertex && arcs[u][i] > 0 )
                {   v = i;
                    break;
                }
            arcs[u][v]--; // decrement count of parallel arcs
            System.out.println("Take arc "+label[u][v].elementAt(arcs[u][v])
                +" from "+u+" to "+v); // use each arc label in turn
        }
    }
}
```

5. Java code for the open CPP

The CPP asks for a closed tour that brings the postman back to their start. In many applications, it is not necessary to return; for instance, in a testing application, once every arc has been tested, the device can be left in any state. It is not necessary for test purposes to return the device to its initial state (although this might only be a final 'switch off' transition).

Imagine that the optimal open solution starts at vertex u and finishes at v. The open solution can be obtained from a closed solution, found by introducing a new 'virtual' arc (v, u), provided the cost associated with this arc is so high that it is used only once in the closed tour (if the cost was low, the virtual arc might be used more than once to 'cheat'). If we assume u is given, to find the optimal open tour, simply solve the closed problem for each possible choice of v and choose any least cost solution.

We can do better. If the graph is Eulerian, then the optimal open CPT will anyway be a Euler circuit, which is closed. Hence to solve the open problem on a Eulerian graph no virtual arc is required: we just solve the closed problem and its solution will be the optimal solution to the open problem. Otherwise the vertices to consider for v need only be selected from D^-. The proof of this is easy: consider a choice of v not in D^-. This will be some vertex either balanced or with an excess of out arcs; with the addition of the virtual arc, it will have an excess of out arcs in either case. A CPT must necessarily use some in arcs to this vertex more than once. Choose any of those in arcs and move v to the initial vertex of that arc. We now have a lower cost tour (provided the arc cost was positive). The argument can be repeated until v is in the set D^-.

The set D^- is necessarily smaller than the set of all vertices, and typically smaller than half its size, so considering choices of v only from D^- represents a useful optimisation. Furthermore, if a least cost open tour is required without assuming a given u, a similar argument can be used to show u can be chosen from D^+.

The code for solving the open CPP shown below uses these ideas, with two provisos. First, our data structure for representing graphs does not make moving arcs convenient, and it is easier to reconstruct graphs for each possible location of v — in this case by storing calls to `addArc` so the graph can be reconstructed by iterating through a saved collection of arcs. Secondly, as the closed algorithm can put the virtual arc anywhere in a tour (which means the solution needs reorganising to end at the virtual arc), it is convenient to split the virtual arc with a virtual vertex w and start the tour there — this ensures that solutions *always* start

and finish with a virtual arc, which is therefore easy to discard. If the graph is Eulerian, we still add the two virtual arcs, so the output is consistent whether or not the graph is Eulerian.

```
class OpenCPP
{   class Arc
    {   String lab; int u, v; float cost;
        Arc(String lab, int u, int v, float cost)
        {   this.lab = lab;
            this.u = u;
            this.v = v;
            this.cost = cost;
        }
    }
    Vector arcs = new Vector();
    int N;

    OpenCPP(int vertices)
    {   N = vertices;
    }

    OpenCPP addArc(String lab, int u, int v, float cost)
    {   if( cost < 0 ) throw new Error("Graph has negative costs");
        arcs.addElement(new Arc(lab, u, v, cost));
        return this;
    }

    float printCPT(int startVertex)
    {   CPP bestGraph = null, g;
        float bestCost = 0, cost;
        int i = 0;
        do
        {   g = new CPP(N+1);
            for( int j = 0; j < arcs.size(); j++ )
            {   Arc it = (Arc) arcs.elementAt(j);
                g.addArc(it.lab, it.u, it.v, it.cost);
            }
            cost = g.basicCost;
            g.findUnbalanced(); // initialise g.neg on original graph
            g.addArc("'virtual start'", N, startVertex, cost);
            g.addArc("'virtual end'",
                // graph is Eulerian if neg.length=0
                g.neg.length == 0? startVertex: g.neg[i], N, cost);
            g.solve();
            if( bestGraph == null || bestCost > g.cost() )
            {   bestCost = g.cost();
                bestGraph = g;
            }
        } while( ++i < g.neg.length );
        System.out.println("Open CPT from "+startVertex+" (ignore virtual arcs)");
        bestGraph.printCPT(N);
        return cost+bestGraph.phi();
    }

}
```

For example, a least cost open tour of the example problem is $1, 3, 0, 1, 2, 3, 0, 2$ — which traverses only 7 arcs rather than the minimum of 10 required for a closed tour.

Different data structures for representing graphs could make moving the vertex v and the arc (v, w) more efficient. We did not use this approach because the clarity of the closed code was more important than its efficiency in supporting the open case.

6. The difficulty of the algorithm

It is certainly possible to solve the problem more efficiently than has been done here. Our code repeatedly relies on linear searches. The shortest paths algorithm found *all* shortest paths, but only the shortest paths between D^+ and D^- are used — although as a side effect of finding all shortest paths we also found spanning trees, one of which was later used to construct the Euler circuit efficiently. Our cycle canceling algorithm is not sophisticated, but it exploited the existing data structure built when finding shortest paths.

However, there are no significantly better algorithms available. The CPP certainly requires shortest paths, and it certainly requires a solution to the integer linear programming problem. However the form of linear programming problem can be reduced to a graph whose optimal CPT represents its solution. The CPP is therefore as hard as the best methods of solving this form of linear program: hence no significantly faster algorithm can be found for the CPP than can be found for this form of linear programming.

Figure 3 shows such a construction. In general: construct a graph with vertices $\{b,\ u_1, \ldots, u_n,\ v_1, \ldots, v_p\}$ where $n = |D^-|$ and $p = |D^+|$. Add the np arcs (u_i, v_j) with weights c_{ij}; for each vertex u_i add p parallel arcs (b, u_i) from b; for each vertex v_j add n parallel arcs (v_j, b) to b. Vertices u_i should have balance $-\delta(i)$: add this number of arcs (b, u_i); similarly add $\delta(j)$ arcs (v_j, b). To ensure the shortest paths $u_i \rightsquigarrow v_j$ are the arcs (u_i, v_j), no indirect path of the form (u_i, v', b, u', v_j) can have a cost less than c_{ij}: therefore assign arcs into and from b weights $\max(c_{ij})$.

7. Conclusions

The Chinese Postman Problem for directed graphs has many interesting and useful applications, and can be solved by an interesting combination of standard algorithms, yet it does not get the wide recognition it deserves. This paper reviewed the applications, and provided a complete Java solution for the problem.

On the one hand, it is disappointing not to find a short algorithm; on the other hand, the Chinese Postman Problem is all the more interesting for being a simple problem with a complex solution. It provides a nice case-study in integrating algorithms.

ACKNOWLEDGEMENTS

Harold Thimbleby is a Royal Society-Wolfson Research Merit Award Holder, and acknowledges their support for this research, The author is grateful for constructive comments from Paul Cairns, Paul Curzon, Herbert Fleischner, Matt Jones, Peter Ladkin, John McCarthy, James Orlin, Peter

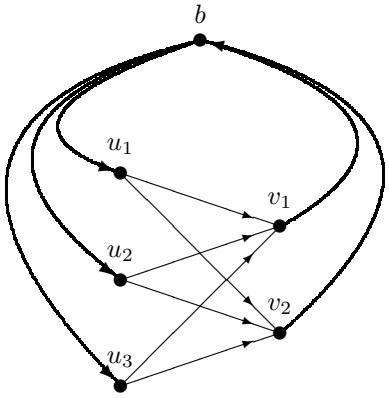

Figure 3. Example construction of a graph corresponding to the linear programming formulation of a CPP, where $|D^-| = 3$ and $|D^+| = 2$. Thicker lines represent multiple parallel arcs; the thin lines are single arcs (for all vertices, the in-degrees and out-degrees are equal).

Rowlinson and Will Thimbleby. This work was also supported by EPSRC Grants No. GR/J43110 and No. GR/K79376.

REFERENCES

1. R. K. Ahuja, T. L. Magnanti & J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*, Prentice-Hall (Simon and Schuster), 1993.
2. S. Albers & M. R. Henzinger, *Exploring Unknown Environments*, Digital Systems Research Center, SRC Technical Note 1997-014, 1997.
3. J. M. Aldous & R. J. Wilson, *Graphs and Applications*, The Open University, Springer Verlag, 2000.
4. B. Bollobás, *Graph Theory*, Springer-Verlag, 1979.
5. 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.
6. A. Gibbons, *Algorithmic Graph Theory*, Cambridge University Press, 1985.
7. A. V. Goldberg & R. E. Tarjan, "Finding Minimum-Cost Circulations by Canceling Negative Cycles," *Journal of the ACM*, **36**(4):873–886, 1989.
8. D. E. Knuth, *The Stanford GraphBase*, Addison-Wesley, 1993.
9. Kuan (Kwan or Guǎn) Mei-Ko, "Graphic Programming Using Odd or Even Points," *Chinese Mathematics*, **1**:273–277, 1962.
10. Y. Lin & Y. Zhao, "A New Algorithm for the Directed Chinese Postman Problem," *Computers and Operations Research*, **15**(6):577–584, 1988.
11. G. Marsden & H. Thimbleby, *Benjamin Franklin Centre*, `http://www.rsa.org.uk/franklin/`, 1998.
12. Nokia Mobile Phones, *Nokia 2110 User's Guide*, Issue 5, 1996.
13. C. S. Orloff, "A Fundamental Problem in Vehicle Routing," *Networks*, **4**(1):35–64, 1974.
14. C. H. Papadimitriou, "On the Complexity of Edge Traversing," *Journal of the ACM*, **23**:544–554, 1976.
15. W. L. Pearn & C. M. Liu, "Algorithms for the Rural Postman Problem." *Computers & Operations Research*, **22**(8):819–828, 1995.
16. P. A. Pevzner, H. Tang & M. Waterman, "An Eulerian path approach to DNA fragment assembly," *Proceedings of the National Academy of Sciences*, **98**(17):9748–9753, 2001.
17. 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.

18. S. Skiena, *The Algorithm Design Manual*, Springer Verlag, 1998.
19. 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.
20. H. W. Thimbleby, "Distributed Web Authoring," in S. Lobodzinski & I. Tomek, eds., *WebNet'97*, World Conference of the WWW, Internet, & Intranet, Toronto, pp1056–1083, Association for the Advancement of Computing in Education (AACE), 1997.
21. H. W. Thimbleby, "Visualising the Potential of Interactive Systems," *The 10th. IEEE International Conference on Image Analysis and Processing*, ICIAP'99, 670–677, 1999.
22. H. W. Thimbleby, "Explaining Computer Programs," *Software—Practice & Experience*, in press. See `http://www.uclic.ucl.ac.uk/harold/warp`.