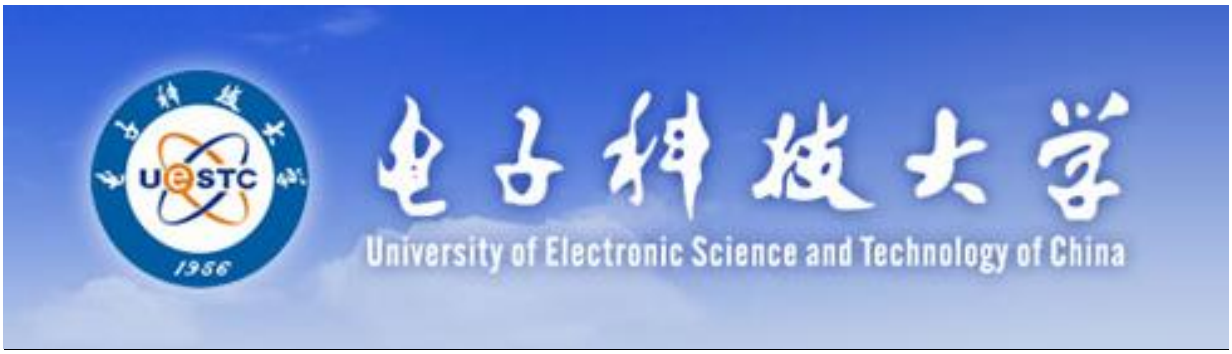


Title: The Chinese Postman Problem (CPP) and It's Solution

Author Name: Mamo Tadiyos Hailemichael

School: Computer Science and Engineering

Major: Computer Science and Technology

Teachers Name: Professor Gao Hui

Course Title: Discrete Mathematics

01 July, 2020

Table of Contents

Abstract

A routing problem arise in several areas of distribution management and have long been the object of study by mathematicians and operation researchers. It can be divided into two main categories: node routing and arc routing. The purpose of arc routing problems is to determine the shortest path and paths which is returning to the starting node by passing at least once from all the lines on a line. Chinese postman problem (CPP) which is one of arc routing problems defined that a postman picks up mails at the post office, delivers it, and then returns to the post office. He must cover each street in his area at least once. Subject to this situation, the CPP is referring to research how to cover every street and return to the post office under the least cost. In reality, there are many situations in which CPP can be used. We consider the Chinese Postman Problem, in which a mailman must deliver mail to houses in a neighborhood. The mailman must cover each side of the street that has houses, at least once. The focus of this paper is our attempt to discover the optimal path, or the least number of times each street is walked. The integration of algorithms from graph theory and operations research form the method used to explain solutions to the Chinese Postman Problem.

Keywords: CPP, Routing, shortest path, algorithm, optimal, graph theory.

1. Curriculum Design Tasks and Requirements:

1.1 Description to Chinese Postman Problem (CPP)

The Chinese Postman Problem was first discovered by Chinese mathematician Kwan Mei-Ko. This problem described a situation that a Chinese postman would face every day in his life. The exact problem was defined in Kwan's paper as, "A postman has to deliver letters to a given neighborhood. He must walk through all the roads within the neighborhood and back to the post-office. How can he plan his route so that he walks the shortest distance?" **[1].** This is the main problem that he wants to figure out to make his job easy. At that point, he generalized the issue by checking the number of edges connected to a node as, "Given a connected graph where 2n of the nodes are odd and all other nodes are even. Suppose we have to include a few edges to the chart with the following property: the number of edges included to any odd node is odd and that included to any even node is even. We need to minimize the overall length of the added edges." **[2].** Then, he solved this problem by his theorem, which is additionally in his paper, " For a set of included edges it is fundamental and adequate to be an ideal arrangement for the above issue taking after two conditions hold:

1. Between any two nodes, no more than one edge is included,
2. In any cycle of the extended graph, the overall length of the added edge is not greater than half of the overall length of the cycle." **[3].**

The Chinese postman who wishes to travel along each street to deliver letters. The Chinese Postman Problem (CPP) is curiously since it has numerous applications, could be a simply-stated issue, but for which there's no straightforward calculation. There are numerous varieties to the CPP, most strikingly whether the streets are one-way (typically the Directed CPP or DPP) and whether the postman has got to return back to where they begun (closed or open CPP).This paper is concerned specifically with the directed CPP, and provides algorithms for both closed and open solutions.

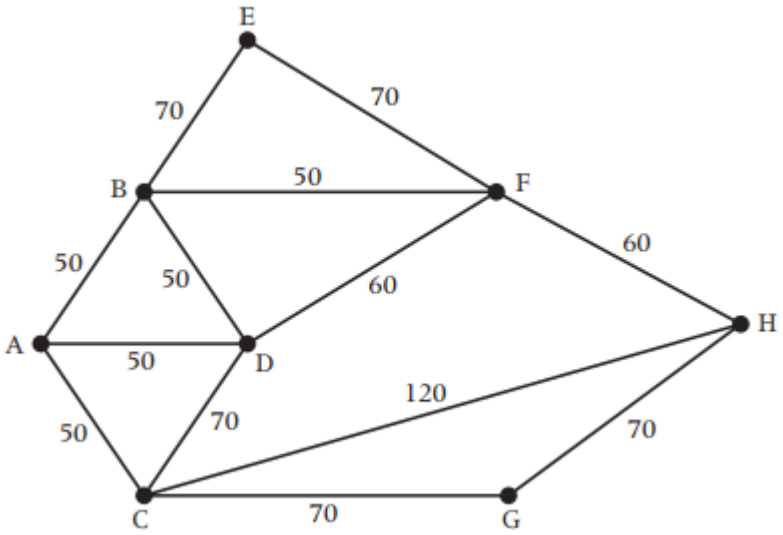
The CPP is one of the foremost interesting problems in Operations Research (OR). It determines its title from the fact that an early paper discussing this problem appeared in the journal Chinese Mathematics **[4]**. The CPP has a simple goal, to transverse each edge of a graph at least once with a minimum of backtracking. The great Swiss mathematician Leonhard Euler first examined this problem in the 18th century. Euler attempted to discover a way in which a parade procession may cross all 7 bridges exactly once in the city of Konigsberg, Prussia **[5]**. Euler demonstrated in 1736 that no solution to the onigsberg routing issue exists. He also determined a few common results that give the inspiration for the solution to the CPP. These results will be introduced in detail below.

The Chinese postman problem, also known as the route inspection problem, is to discover a circuit in an associated undirected graph that visits each edge at least once. Also, an adjusted arrangement minimizes the full fetched cost of the circuit. In case all vertices of the graph have even degree, at that point the ideal arrangement is the Euler circuit of the graph. In this case, the fetched cost of the arrangement is the sum of whole edge weights in the graph.

Within the case where the graph has a few odd edges, the arrangement is more complicated. To begin with identifying all nodes with odd degree. Following, match up the odd nodes such that the entire sum of the weights of the least paths between pairs is a least. At that point, take all the edges from the paths in the past step, and copy them. This guarantees that all nodes presently have even degree, whereas including the least conceivable weight to the graph. At last, the arrangement is the Euler circuit of the altered graph, and the fetched cost is the entire sum of whole edge weights in the original graph added with the entire sum of the weights of the copied edges.

The three primary steps (discover odd nodes, discover least paths, and discover Euler circuit) can all be figured out in polynomial time, so this problem can be illuminated in polynomial time.

Within the following example a postman should begin at A, walk along all 13 roads and return to A. The numbers on every edge denotes the length, in meters, of all street. The problem is to discover a path that uses entire edges of a graph with least length.



How seem a postman visit each letter on the graph in the shortest conceivable time?

Tackling this requires using a branch of mathematical science called **graph theory**, made by Leonard Euler. This arithmetic looks to decrease problems to network graphs like that shown above.

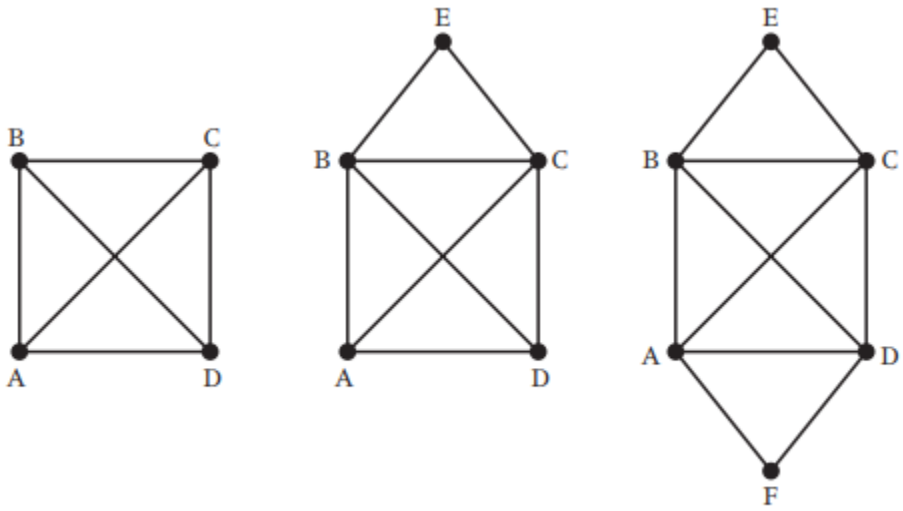
Before we can solve the above graph problem, we ought to get into a few terminologies on next page:

1.2 Pre-request Concepts and Terminologies of Graph

We will return to solving the above actual problem later, but initially we will look at drawing various graphs.

1.2.1 Traversable graph

A **traversable graph** is one that can be drawn without taking a pen from the paper and without retracing the same edge. In such a case the graph is said to have a **Eulerian trail**. [6].



Graph 1 Graph 2 Graph 3

Above we have 3 graphs. A graph which can be drawn without taking the write off the paper or re-tracing any steps is called as traversable (and has a Euler trail) [7]. So, based on this definition: Graph 1 is not traversable, Graph 2 is traversable as long as you begin at either A or D, and Graph 3 is traversable from all point that you begin. Then It turns out that what directs whether a graph is traversable or not is the arrange of their vertices.

Generally, in case we attempt drawing the three graphs shown above, we find:

- It is inconceivable to draw Graph1 without either taking the write off the paper or retracing an edge.
- We can draw Graph 2, but as it were by beginning at either A or D – in all case the route will conclude at the other vertex of D or A.
- Graph 3 can be drawn in any case of the beginning position and you'll continuously return to the beginning vertex.

What is the difference between these three graphs?

Looking at each letter we count the number of lines at the vertex. This is the order. For graph 1 we have 3 lines from A so A has an order of 3. All the vertices on graph 1 have an order of 3. For graph 2 we have the orders (from A, B, C, D, E in turn) 3, 4, 4, 3, 2. For graph 3 we have the orders 4,4,4,4,2,2.

This allows us to arrive at a rule for working out if a graph is traversable.

If all orders are even then a graph is traversable. If there are 2 odd vertices then we can find a traversable graph by starting at one of the odd vertices and finishing at the other. We need therefore to pair up any odd vertices on the graph.

- In order to establish the differences, we must consider the order of the vertices for each graph. We obtain the following:

Graph 1

Vertex	Order
A	3
B	3
C	3
D	3

Graph 2

Vertex	Order
A	3
B	4
C	4
D	3
E	2

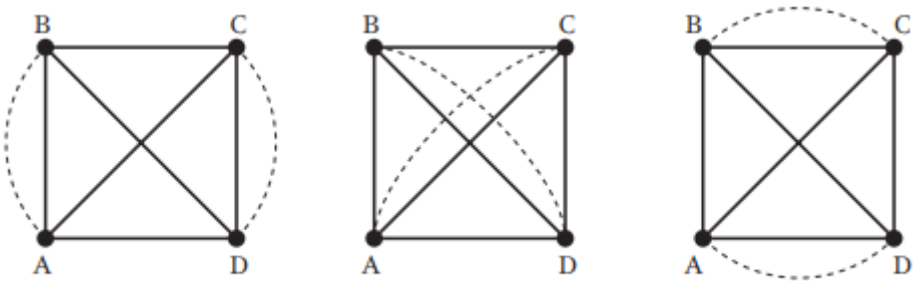
Graph 3

Vertex	Order
A	4
B	4
C	4
D	4
E	2
F	2

When the order of all the vertices is even, the graph is traversable and we can draw it. When there are two odd vertices, we can draw the graph but the start and end vertices are different. When there are four odd vertices the graph can't be drawn without repeating an edge.

A **Eulerian** trail uses all the edges of a graph. For a graph to be Eulerian all the vertices must be of even order. If a graph has two odd vertices then the graph is said to be **semi-Eulerian**. A trail can be drawn starting at one of the odd vertices and finishing at the other odd vertex.

To draw the graph with odd vertices, edges need to be repeated. To find such a trail we have to make the order of each vertex even. In graph 1 there are four vertices of odd order and we need to pair the vertices together by adding an extra edge to make the order of each vertex four. We can join AB and CD, or AC and BD, or AD and BC.



In each case the graph is now traversable.

1.2.2 Pairing odd vertices

Next, we need to understand how to pair the odd vertices. For example:

- If there are two odd vertices there is only one way of pairing them together.
- If there are four odd vertices (say A, B, C, D) there are three ways (either AB and CD or AC and BD or AD and BC) of pairing them together.
- How many ways are there of pairing six or more odd vertices together?
  If there are six odd vertices ABCDEF, then consider the vertex A. It can be paired with any of the other five vertices and still leave four odd vertices. We know that the four odd vertices can be paired in three ways; therefore, the number of ways of pairing six odd vertices is 5 x 3 x 1 = 15.
- Similarly, if there are eight odd vertices ABCDEFGH, then consider the first odd vertex A. This could be paired with any of the remaining seven vertices and still leave six odd vertices. We know that the six odd vertices can be paired in 15 ways therefore the number of ways of pairing eight odd vertices is 7 x 5 x 3 x 1 = 105 ways.

We can continue the process in the same way and the results are summarized in the following table.

Number of odd vertices	Number of possible pairings
2	1
4	$3 \times 1 = 3$
6	$5 \times 3 \times 1 = 15$
8	$7 \times 5 \times 3 \times 1 = 105$
10	$9 \times 7 \times 5 \times 3 \times 1 = 945$
n	$(n-1) \times (n-3) \times (n-5) \ldots \times 1$

- The general term rule to calculate how many ways n odd vertices can be paired up is:

  (n-1) x (n-3) x (n-5) … x 1.

Let's generalize several fundamental relationships between the vertices and edges of a graph.

- The sum of the degrees of the vertices of a graph is twice the number of edges.
- Each edge contributes two to the degree sum.
- In a graph, there is an even number of vertices having odd degree.
- Then the total sum of the degrees of the vertices of a graph is twice the number of the edges. Subtract from this sum the result of calculating the sum of the degrees of the even degree vertices. The result must be an even number. So, the sum of the odd degrees must also be even. Hence, there must be an even number of vertices of odd degree.

- A connected undirected graph is a Euler Graph thus processes a Euler Tour if and only if all vertices of the graph have even degree. [8].

2. Fundamentals:

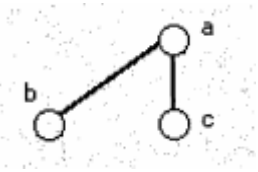
2.1 Definitions and Outline of the Algorithm

A graph G = (V, E) consists of two finite sets V and E. The elements of V are called vertices, and the elements of E are called edges. Each edge has a set of one or two vertices associated, to it, which are called its endpoints. If every edge has a specified orientation, the graph is called directed. If no edge has an orientation, the graph is undirected; and if some edges are directed and some are not, the graph is mixed. An edge connecting vertices i, j ∈ V will be denoted as the unordered pair (i, j). A multi-edge is a collection of two or more edges having identical vertices.

A subgraph H'= (V', E') of a graph G = (V, E) is a graph such that V'⊆V and E'⊆E. In this case, can only contain edges whose end points are in V'. If H is a subgraph of G, we may also say that G is a supergraph of H. A supergraph, G=H U {e}, is easily constructed by adding an edge {e} between any two vertices i and] in a graph H. If a graph has n vertices and each vertex has n-l edges, that graph is called a complete graph.

Any two vertices connected by an edge or any two edges connected by a vertex are said to be adjacent as seen in Figure 1, If vertex b is an endpoint of edge (b, a), then b is said to be incident to (b, a), and (b, a) is incident to b. The degree of a vertex v in an undirected graph G, denoted **deg(v)**, is the number of edges incident to v. For directed graphs, the indegree of a vertex is the number of edges leading into that vertex and its outdegree, the number of edges leading away from it.

Figure 1. Adjacent Edges. Vertex b is Incident to (b, a)



The characteristics of graphs we are interested in are identified by examination. Keeping in mind that my graph will be finite, the first characteristic we find is the graph could consist of nothing but even degree vertices. The degree of a vertex is the number of edges incident to that vertex. Euler's theorem states that if all vertices have even degree then there exists a Euler Tour, a tour which transverses every edge of a graph exactly once, and the optimal solution is ensured. [9].

3. Theoretical Design (and Implementation):

3.1 Solution for Chinese Postman Problem

To find a minimum Chinese postman route we must walk along each edge at least once and in addition we must also walk along the least pairings of odd vertices on one extra occasion.

An algorithm for finding an optimal Chinese postman route is:

Step 1: List all odd vertices.

Step 2: List all possible pairings of odd vertices.

Step 3: For each pairing find the edges that connect the vertices with the minimum weight.

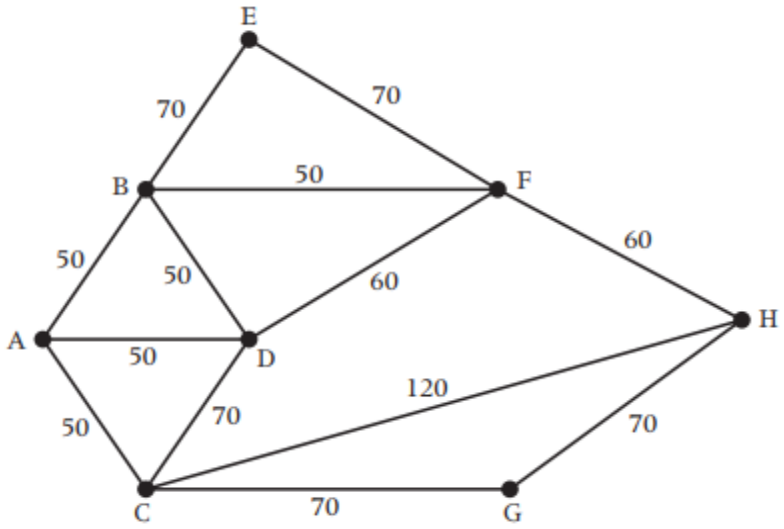
Step 4: Find the pairings such that the sum of the weights is minimized.

Step 5: On the original graph add the edges that have been found in Step 4.

Step 6: The length of an optimal Chinese postman route is the sum of all the edges added to the total found in Step 4.

Step 7: A route corresponding to this minimum weight can then be easily found.

So, we can now apply this to the original Chinese Postman Problem below:



Let's apply the above Chinese Postman Algorithm to this graph as follow:

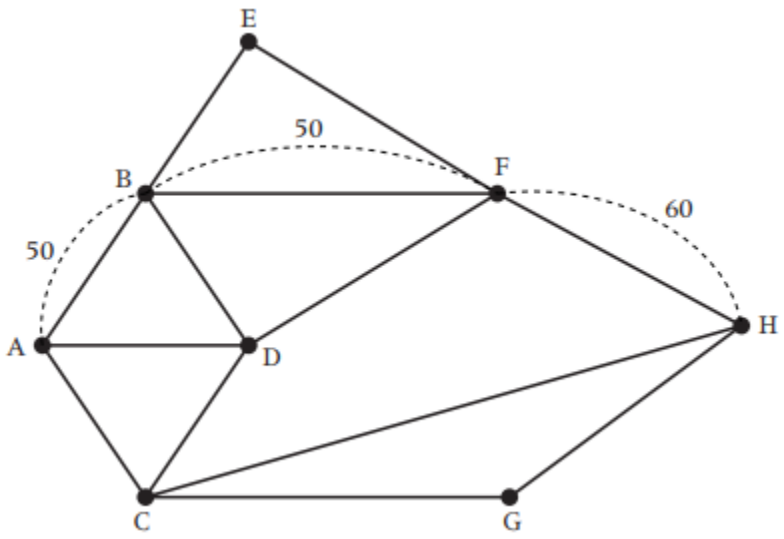
Step 1: We are able see that the only odd vertices are A and H.

Step 2: There's only one way of matching these odd vertices, to be specific AH.

Step 3: The least possible way of joining A to H is using the path AB, BF, FH (ABFH), a total length of:

50 + 50 + 60 = **160**. This is shown below:

Step 4: Draw these edges onto the above original network.



Step 5 and 6: The length of the optimal Chinese postman route is the sum of all the edges in the original network, which is 840 m, plus the answer found in Step 4, which is 160 m. Therefore, the optimum (minimum) distance it is possible to complete the route is 1000m.

Step 7: We now need to find a route of distance 1000m which includes the loop ABFH (or in reverse) which starts and finishes at one of the odd vertices. One possible route corresponding to this length is ADCGHCABDFBEFHFBA, but many other possible routes of the same minimum length can be found.

3.1.1 Finding an Optimal Route

We need to insert paths from those nodes with too few outgoing edges to those nodes with too few incoming edges. Since we aim to minimize our costs, these should be shortest paths. We will then use the costs of the shortest paths in the Matching Phase in order to determine the optimal set of shortest paths to use.

The method for finding the length of the Chinese postman route is quite straightforward, but to find the list of edges corresponding to this route can be quite tricky, especially in complicated networks. It is useful to calculate how many times each vertex will appear in a Chinese postman route. The following method should be applied before trying to find the route.

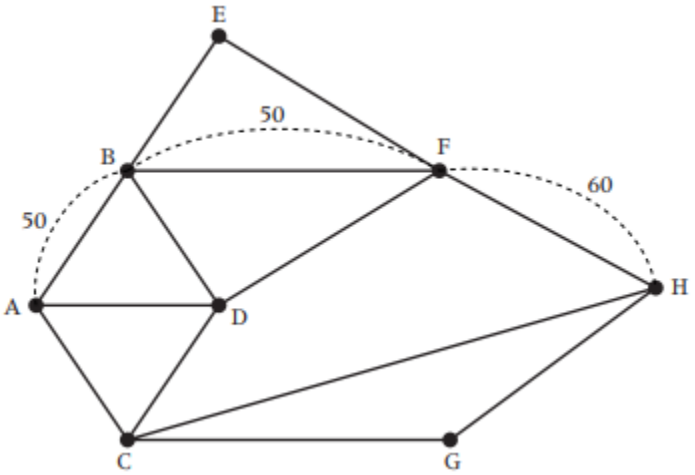
Step 1: On the original diagram add the extra edges to make the graph Eulerian.

Step 2: List the order of each vertex. At this stage each vertex will have an even order.

Step 3: The number of times each edge will appear in a Chinese postman route will be half the order of its vertex, with the exception being vertex A (the start/finish vertex), as this will appear on one extra occasion.

Referring to the diagram below, the orders of the vertices are as follows:

Vertex	Order
A	4
B	6
C	4
D	4
E	2
F	6
G	2
H	4



This indicates that the number of times each vertex will appear in the Chinese postman route is:

A $\frac{4}{2}$ = 2 + 1 = 3 B $\frac{6}{2}$ = 3

C $\frac{4}{2}$ = 2 D $\frac{4}{2}$ = 2

E $\frac{2}{2}$ = 1 F $\frac{6}{2}$ = 3

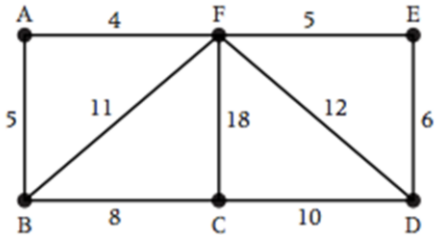
G $\frac{2}{2}$ = 1 H $\frac{4}{2}$ = 2

The number of vertices in the optimal Chinese postman route is:

3 + 3 + 2 + 2 + 1 + 3 + 1 + 2 = **17**.

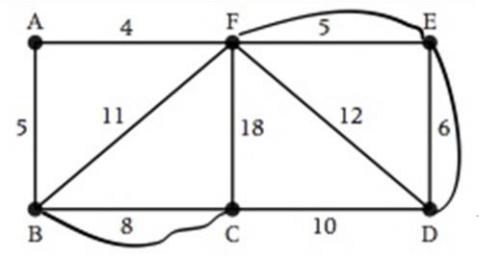
They may be in a different order than in the example above but they must have the number of vertices as indicated in the table

Let's again apply the above algorithm for the following example of graph to find a minimum Chinese postman route.



1. B, C, F and D are odd vertices.
2. BC-FD, BF-CD and BD-CF are possible pairings of odd vertices.
3. For each pairing find the edges that connect the vertices with the minimum weight:

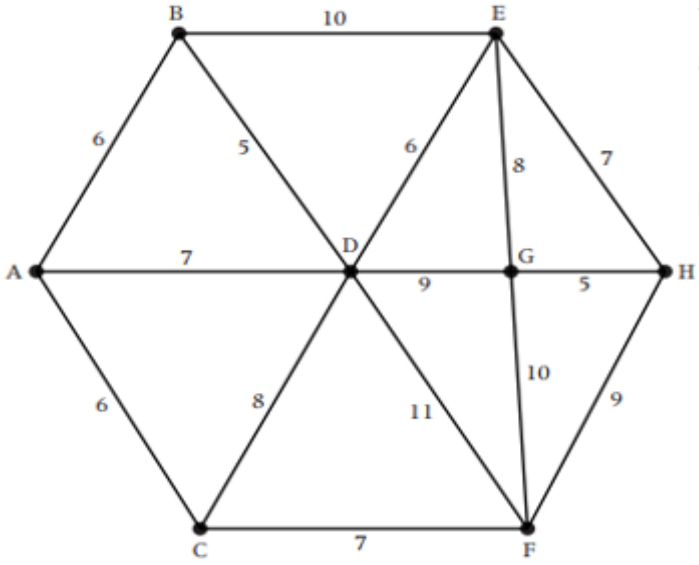
   BC-FD: 8+11=19. BF-CD: 9+10=19. BD-CF: 17+18=35.
4. Find the pairings such that the sum of the weights is minimized: BC-FD and BF-CD.
5. the original graph adds the edges that have been found in Step 4. (See following graph)



1. Total weight should be 79+19=**98**.
2. A possible route of this weight can be **AFEDFEDCFBCBA**

Let's see our final example of real-world application that how a supervisor and a council worker maintain the following road networks.

For example: A county council is responsible for maintaining the following network of roads. The number on each edge is the length of the road in miles.



The council office is based at A.

a) A council worker has to inspect all the roads, starting and finishing at A. Find the length of an optimal Chinese postman route.

b) A supervisor, based at A, also wishes to inspect all the roads. However, the supervisor lives at H and wishes to start his route at A and finish at H. Find the length of an optimal Chinese postman route for the supervisor.

Solution

a) There are four odd vertices: A, B, C and H.

There are three ways of pairing these odd vertices and the minimum length of each pairing is:

AB + CH = 6 + 16 = 22

AC + BH = 6 + 17 = 23

AH + BC = 21 + 12 = 33

Draw the edges AB and CH onto the network.

The length of all the roads in the network is 116.

The length of an optimal Chinese postman route for the worker is 116 + 22 = 138 miles.

b) Starting at A and finishing at H leaves two odd vertices B and C.

The minimum distance from B to C is 12.

The length of an optimal Chinese postman route for the supervisor is 116 + 12 = 128 miles.

3.2 Implementation of CPP Algorithm

The Chinese Postman Problem 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 [] []; // to check path cost is defined between vertices or not
        int path [] []; //
        float basicCost; // sum of the cost that traverse each arc once
        void solve ()
        {
                leastCostPaths ();
                checkValid ();
                findUnbalanced ();
                findFeasible ();
                while (improvements ());
        }
              .
              .
              .
            // Other definitions and declarations can be described below
              .
              .
              .
}
```

We use a simple adjacency matrix representation of graphs: arcs count 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;
}
```

Initialization in Java sets booleans to false and integers to zero: all elements of delta, arcs, neg and pos are 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.
```

Find shortest paths and costs

The Floyd-Warshall algorithm [10] can efficiently find and record shortest paths at the same time as establishing the costs, c, by building a matrix P (called path in the Java) such that the first arc of the least cost path u >>>v from vertex u to vertex v is (u, Puv).

Subsequent arcs along the path are found similarly; if (u, P_{uv}) is an arc to vertex w ≠ v then (u, P_{wv}) is the next arc to take towards v, and so on.

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

```
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. Conclusion Analysis

As illustrated by the example, the algorithm given in this paper provides a solution to the Chinese Postman Problem for directed graphs that 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 detail break-down process of the algorithm and we extended this problem in the real-world applications and provided a different optimization 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.

5. Harvest, experience and suggestions

The Chinese Postman Problem has many useful applications, including robot exploration, and analyzing 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 optimizations are possible, no substantially better algorithms are likely.

CPP may be used in several areas, the main real-world application areas are:

- Mail delivery
- Garbage collection
- Road gritting
- Highways streets cleaning, ice controls and also snow removal operations
- School bus and police patrol vehicles routing
- Water and newspaper distribution
- Effective web site determination
- Network algorithm checking and transmission line inspections
- For well-designed web sites, to check every link explicitly

Imagine trying to understand your mobile phone. Pressing buttons takes the phone to a new state, and corresponds to travelling an "edge". After some considerable work, one might obtain a map of how the phone works. However, one doesn't know if the map is correct or not, and the map may be too difficult or complex to test systematically. Given the map, a CPT will provide a systematic test sequence that will exercise each transition.

Another example that we can use CPP is if the 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.

References

[1] Brucker, P. 1981. The Chinese Postman Problem for Mixed Graphs. In Graph Theoretic Concepts in Computer Science. H. Noltemeier (ed.). Springer, Berlin, 254-366

[2] Edmonds, J. 1965a. The Chinese Postman's Problem. ORSA Bull. 13,73

[3] Win, Z. 1987. On the Windy Postman Problem on Eulerian Graphs. Math. Prog. 44, 97-112

[4] Picard, J.-C., and H. D. Ratliff. 1975. Minimum Cuts and Related Problems. Networks 5, 357-370

[5] Kaufmann, C. H. and G. J. Koehler. 1979. The Mixed Postman Problem. Discr. Appl. Math. 1, 89-103

[6] Chrirsofides, N. 1975. Graph Theory. An Algorithm Approach. Academic Press, London

[7] Frederickson 1978. Approximation Algorithms for Some Routing Problems. SIAM J. Comp. 7, 78-93

[8] Alfa, A. S., and D. Q. Liu. 1988. Postman Routing Problem in a Hierarchical Network; 127-138

[9] Harary, F. 1969. Graph Theory. Addison-Wesley, Reading, Mass

[10] Guan, M. 1962. Graph Programming Using Odd and Even Points. Chinese Math. 1, 273-277