Kwan's Postman Problem

Neither Rain, Nor Sleet, Nor Dark Of Night Shall Stay These Couriers
From The Swift Completion Of Their Appointed Rounds.
– *James Farley Post Office in New York City*

Figure 1: Cliff Clavin of TV's *Cheers*

Cliff Clavin, dedicated but deeply lazy postman, must walk every street in his route while delivering mail and then return home. Cliff might have to travel some streets multiple times to reach every street. How can Cliff plan a route that minimizes the walking?

If we interpret the streets as edges of a graph with weights describing distance, then this is a combinatorial minimization problem. Mei-Ko Kwan in

1962 proposed an algorithm to solve the *postman problem.* A more rigorous model for the problem is the following:

Problem. Given a connected weighted graph G, how can one compute a minimal weight closed walk that contains every edge of G?

Jargon. For completeness let's define all the terms in the problem: A *weighted graph* $G = (V, E, w)$ is a set of *vertices* V (in our problem these represent street junctions) together with a set E of pairs of vertices called *edges* (representing streets) and positive real-valued function on the edges called the *weight* $w : E \to (0, \infty)$. Here the weight of the edge represents the length of the street. Vertices that share an edge are *adjacent.* If v is a vertex then number of adjacent vertices is called the *degree* of v. A *walk* of G is a string of vertices such that vertices adjacent in the string must also be adjacent in the graph. More explicitly, a walk length n is a string $s : n + 1 \to V$ such that for all $i = 0, \ldots, n - 1$ we have that $\{s_i, s_{i+1}\}$ is an edge of the graph. The walk is *closed* if it ends where it began: $s(n) = s(0)$. The weight of the walk is the sum of the weights of its edges: $w(s) = \sum_{i=0}^{n-1} w(\{s_i, s_{i+1}\})$. Two vertices vertices of a graph are *connected* if there is a walk between them. A graph is *connected* if any two vertices are connected. Recall further that a *multigraph* allows multiple edges between the same vertices. An *Eulerian circuit* is a closed walk that contains every edge of the graph exactly once. When a graph or multi-graph has an Eulerian circuit, we just say that the graph is *Eulerian.*

Let's return to thinking about the substance of the problem. A really easy algorithm can solve the problem in some cases. Euler proved the following:

Theorem. A multigraph is Eulerian if and only if it is connected and every vertex has even degree.

If the graph is Eulerian, then any Eulerian circuit is the postman's best route; every Eulerian circuit has weight equal to the total weight of the graph. There's an easy algorithm for constructing an Eulerian circuit for an Eulerian graph. Iteratively choose edges while being careful not to disconnect the remaining edges of the graph:

Fleury's Algorithm: to construct an Eulerian circuit from an Eulerian graph

```
input Eulerian graph $G = (V, E)$ and starting vertex $v \in V$
    initialize $H \leftarrow \varnothing$ as subgraph of used up edges and vertices
    initialize $w \leftarrow v$ as the walk
    initialize $x$ as $v$
    while $H \neq G$ do:
        if there is any edge $\{x, y\}$ from $x$ such that $G \setminus (H \cup \{\{x, y\}\})$ is connected set $e \leftarrow \{x, y\}$
        else $e \leftarrow \{x, y\}$ the only edge from $x$ in $G \setminus H$
        append $y$ to $w$
        add $e$ to $H$
        if $x$ has degree 0 in $G \setminus H$ add $x$ to $H$
        update $x \leftarrow y$
output Eulerian circuit $w$
```

Question: The algorithm above isn't fully described as we haven't specified a subroutine that checks if a graph is connected. Write some psuedocode for an algorithm that can check if a graph is connected.

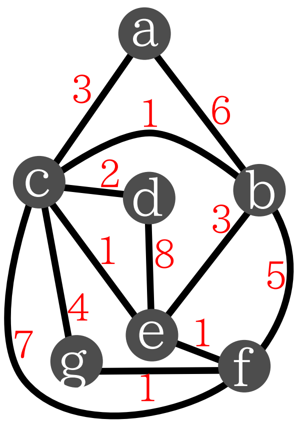

Figure 2: An Eulerian graph

Let's solve the postman problem for the graph in Figure 2. Say Cliff will start and end his route at a. Every vertex has even degree, so we can construct an Eulerian circuit with exactly the weight of the full graph. Start at a and always choose the vertex lowest in alphabetical order provided you don't disconnect the remaining graph from a. We get the weight 42 closed walk $abcdebfecgfca$.

But not all graphs are Eulerian, sometimes Cliff will have to travel streets multiple times to reach every street and get back home. Consider the graph in figure 3. Vertices c and e are both odd degree. Vertex c has degree 5, so Cliff will arrive at c three times, leaving only two streets to leave c. Inevitably, Cliff must repeat an edge that is incident to c. The same is true of vertex e. To be efficient Cliff should reuse small weight edges. Use Dijkstra's algorithm to identify a minimal walk

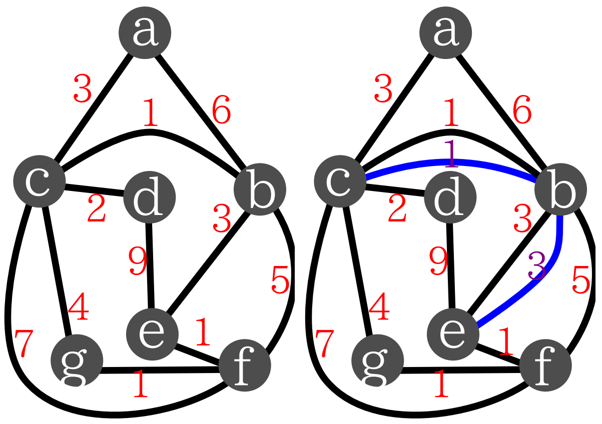

Figure 3: An non-Eulerian graph and its minimal Eulerianized multigraph

from c to e. You'll find there is a weight 4 walk cbe. That is the path that Cliff should rewalk. Double those edges of that walk in the graph to obtain an Eulerian multigraph. Then we can use Fleury's algorithm to calculate a route. A minimal weight closed walk containing every edge is given by the weight 46 walk $abcbebfedcgfca$. Note that this is the total weight of the original graph, 42, plus the weight of the minimal walk between the odd vertices.

In general the graph might have many vertices of odd degree! The procedure is the same: we will pair off all the odd degree vertices by cloning edges along walks connecting them to obtain an Eulerian multigraph. We just need to be sure that we pair off odd degree vertices in a minimal way. Pairing the vertices off like this is called by graph theorists a *minimal perfect matching*.

Algorithm: to compute a minimal weight closed walk containing every edge of a graph

```
input weighted graph $G = (V, E, w)$
    identify odd degree vertices $W$ of $G$
    use Dijkstra's algorithm to compute least weight walks between all odd vertices $W$
    construct complete graph $K_W$ with vertex set $W$ and edges equal to the minimal walks in $G$
    compute a minimal perfect matching of $K_W$
    construct multigraph $H$ from $G$ by double edges of the walks in the perfect matching
    use Fleury's algorithm to compute $w$ Eulerian circuit of $H$
output $w$
```

Again we've left some subroutines unspecified. But let's attempt the algorithm with the graph G in figure 4 anyway. We identify four odd degree veritces: b, c, e and f. Now use Dijkstra's to compute the distance between every pair of odd vertices. We can think of this as a complete graph with the edges weighted by Dijkstra distance in the original graph, as shown in figure 5. Now we decide which vertices we should connect with walks in G to make our Eulerian multigraph. There are three perfect matchings:

$(be)(cf)$ weight 3+5=8
$(bc)(ef)$ weight 1+9=10
$(bf)(ce)$ weight 6+4=10

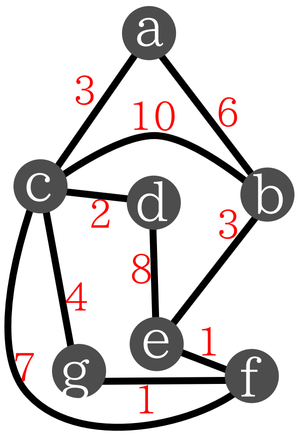

Figure 4: More odd degree vertices

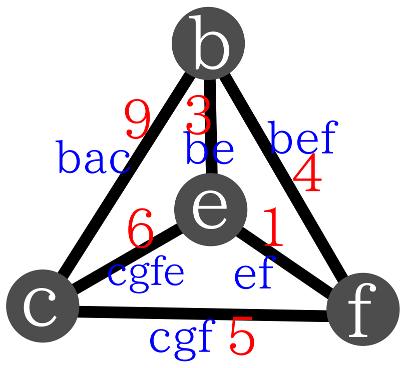

Figure 5: Distance graph of the odd vertices of G

So we see the minimal perfect matching has b connected to e and c to f. (Note that (ef) is only weight 1, so it might seem like a good idea to pair, but it forces you to pair b and c incurring a large cost. That's why a greedy approach to the postman problem can fail in general.) We will then double the edges c, g, g, f, and b, e in G to get the Eulerian multigraph shown in figure 6. Now we can write down a minimal closed walk with every edge of G. The walk $abcdebefcgfgca$ should do. It has weight 52=44+8.

Question: Will Cliff ever have to walk down a street more than twice? Prove that the minimal Eulerian multigraph obtained from a graph has at most two edges between any pair of vertices.

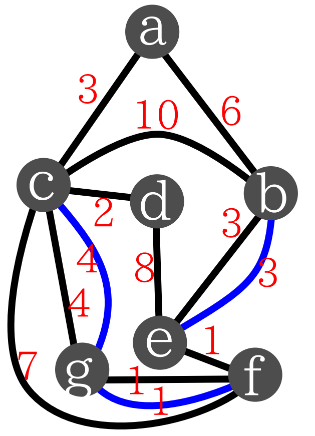

Figure 6: Minimal Eulerianized multigraph

The postman problem is in complexity class P, although that is not at all evident from the algorithm described above. The computational bottleneck in our algorithm comes from the pefect matching. In our example we compute the minimal perfect matching by a brute force comparison– there were only 3 perfect matchings anyway. How bad could it be in general? Very.

Question How many ways are there to pair off $n = 2k$ vertices? Show that there are

$$\frac{(2k)!}{k!2^k} = \prod_{j=1}^{k} 2j - 1$$

If we use Stirling's approximation to simplify the factorials, we find that a brute force approach to the matching will requires checking $\mathcal{O}\left(\left(\frac{n}{e}\right)^{n/2}\right)$ possibilities. This is completely untenable. Is this as bad a problem as the traveling salesman? In fact there are efficient known algorithms to compute minimal perfect matchings. One such algorithm is Edmund's Blossom Algorithm. Extremely efficient algorithms are known. See for example:

Micali, Silvio; Vazirani, Vijay (1980). *An $O(V^{1/2}E)$ algorithm for finding maximum matching in general graphs.* 21st Annual Symposium on Foundations of Computer Science,. IEEE Computer Society Press, New York. pp. 1727.