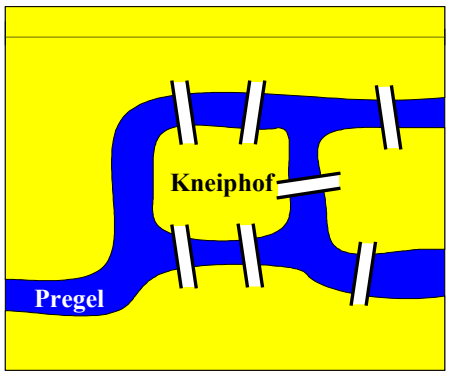
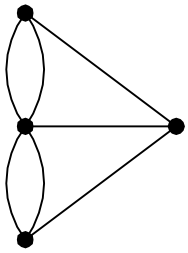
Königsberg Bridge Problem

In the early eighteenth century, the mediaeval town of Königsberg in Prussia had a central island (the Kneiphof) around which the Pregel river flowed before dividing in two. The four parts of the town were linked by seven bridges as shown on the diagram.



Königsberg Bridge Problem: *is it possible to find a route through Königsberg, beginning and ending at the same point, that crosses each bridge* exactly *once*?

In 1731, Leonhard Euler[1] published a paper that solved the problem and, at the same time, gave birth to the subject of graph theory. Euler first represented the essential features of Königsberg by a graph – the parts of the city are represented by vertices and the bridges are represented by edges.



In graph theory terms, the problem is: *find a trail of edges, beginning and ending at the same vertex, that traverses each edge exactly once.*

Notation. A trail which begins and ends at the same vertex and includes every edge exactly once is now called an **Eulerian trail**; a graph is **Eulerian** if it has an Eulerian trail.

In his 1736 paper, Euler proved the following theorem. The **degree** of a vertex is the number of edges incident with it.

Euler's Theorem. *A connected graph G is Eulerian if and only if every vertex has even degree.*

[1] Leonhard Euler (1707–1783) was born in Switzerland but spent most of his life in St Petersberg and Berlin. was probably the most prolific mathematician of all time, his collected works filling more than 70 volumes. More than any other person, Euler is responsible for much of the mathematical notation that is in use today.

Euler's theorem gives a simple characterisation that allows us to determine whether a graph is Eulerian. There is also a simple algorithm for constructing and Eulerian trail where one exists.

Fleury's Algorithm

Let G be a connected graph. If G is Eulerian then the following algorithm will produce an Eulerian trail in G. In a connected graph G, a **bridge** is an edge which, if removed, produces a disconnected graph.

Step 1. Choose any vertex v of G and set *current vertex* equal to v and *current trail* equal to the empty sequence of edges.

Step 2. Select any edge e incident with the *current vertex* **but** choosing a bridge only if there is no alternative.

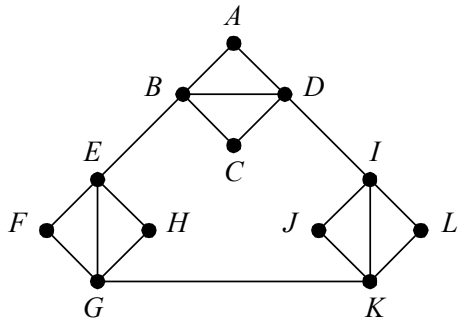
Step 3. Add e to the *current trail* and set the *current vertex* equal to the vertex at the 'other end' of e.

Step 4. Delete e from the graph. Delete any isolated vertices.

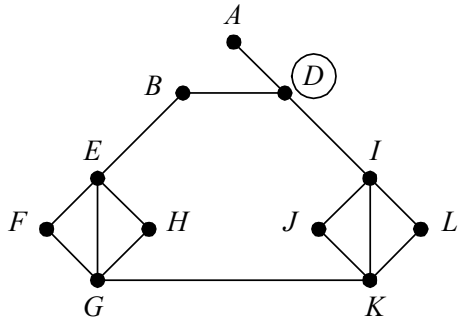
Step 5. Repeat steps 2 – 4 until all edges have been deleted from G. The final *current trail* is an Eulerian trail in G.

Example

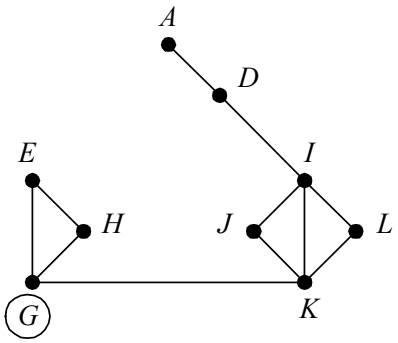
Apply Fleury's algorithm, beginning with vertex A, to find an Eulerian trail in the following graph. In applying the algorithm, at each stage chose the edge (from those available) which visits the vertex which comes first in alphabetical order.



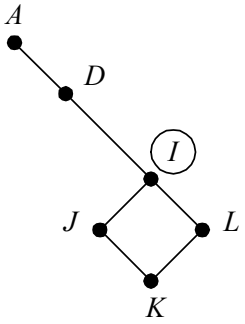
Starting at A, choose AB, BC, CD. This gives the following graph (with the current vertex circled).



The edge DA is a bridge; choose DB, BE, EF, FG to produce the following graph.



(*GK* is a bridge.) Choose *GE*, *EH*, *HG*, *GK*, *KI* to give the following.



The edge *ID* is a bridge so choose *IJ* followed by *JK*, *KL*, *LI*, *ID*, *DA*.
The complete trail is:

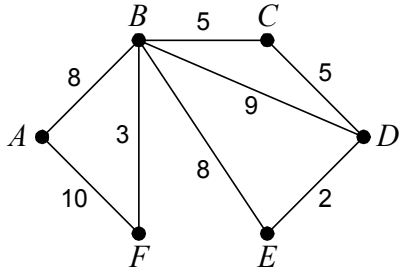
A B C D B E F G E H G K I J K L I D A.

Travelling Salesperson Problem

Given a network of roads connecting a collection of towns, a travelling salesperson has to visit each town and return to his or her starting point. The **Travelling Salesperson Problem** (TSP) is to choose a route which minimises the total distance of the round trip. Note that we shall use *distance* and *weight* interchangeably.

Example

The following weighted graph represents such a network of roads connecting villages *A*, *B*, …, *F*.



A shortest round-trip route visiting each village is _____

Mathematical Formulation of the Problem

What, precisely, do we mean by a 'round trip route'? In graphs, the obvious notion of a 'round trip route' is a cycle. So we might try to formulate the problem as:

find a minimum weight cycle passing through every vertex.

The problem is that the graph above does not have *any* cycle that passes through every vertex. Why not?

Definition

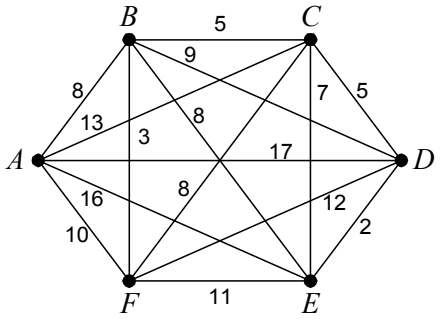
In a graph, a cycle that passes through every vertex is called a **Hamiltonian cycle**.[2]

It is usual to reformulate the problem by replacing the original weighted graph representing the actual connections between the locations with a **complete** graph in which every vertex is joined to every other vertex.

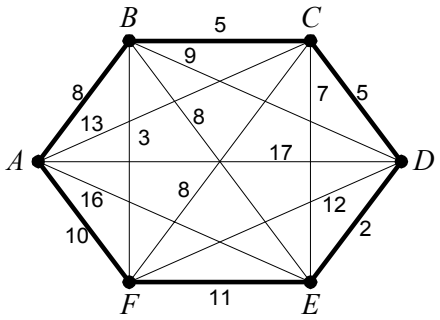
What should be the weights of the edges of the complete graph?

Example

The complete graph representing the 'village network' in the previous example is given below.

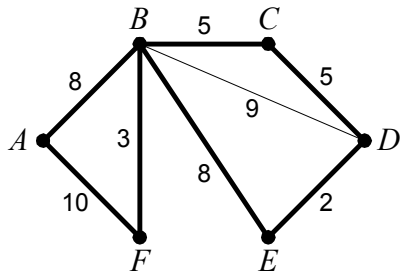


A minimum weight Hamiltonian cycle is shown below (with total length 41).



[2] Sir William Rowan Hamilton (1805–65) was Ireland's most gifted mathematician-scientist. As a 22 year-old undergraduate he was elected Professor of Astronomy and Astronomer Royal of Ireland. In fact he made little contribution to astronomy; his most significant work was in mathematics and physics. In 1843 he discovered the quaternions – a sort of generalized complex numbers – and he devoted most of the rest of his life to their study. His name is also associated with the Hamiltonian operator used in physics, particularly wave mechanics. The name Hamiltonian cycle in graph theory derives from a game, called the *Icosian Game*, that Hamilton invented which involved finding Hamiltonian cycles in the graph of a dodecahedron.

Since each edge in the complete graph corresponds to a path in the original graph, we can construct the 'round trip route' in the original graph as follows. (Simply replace the edge EF in the complete graph with the path EBF that is represents in the original graph.)



We can now give the classical mathematical formulation of the travelling salesperson problem.

Travelling Salesperson Problem

Given a complete weighted graph, find a Hamiltonian cycle of minimum weight.

Every complete graph has a Hamiltonian cycle. In fact, a complete graph with n vertices has many Hamiltonian cycles.

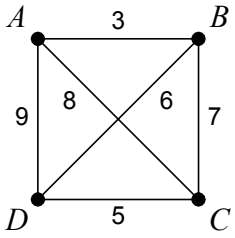
Question: How many Hamiltonian cycles are there in a complete graph with n vertices?

Algorithm for the Travelling Salesperson Problem

There is an obvious 'brute force' algorithm for the TSP:

1. List all Hamiltonian cycles.
2. Calculate the weight (length) of each cycle.
3. Select a cycle with minimum total weight. (Note that there may be several such cycles.)

Example



Beginning and ending at A and ignoring the orientation of cycles, there are 3 Hamiltonian cycles:

Cycle	Weight
ABCDA	24
ABDCA	22
ACBDA	30

The minimum weight Hamiltonian cycle is $ABDCA$ with weight 22.

The cycle is equivalent to $BDCAB$, $DCABD$, $CABDC$ which start at a different vertex and $ACDBA$, $CDBAC$, $DBACD$, $BACDB$ which have the opposite orientation

How efficient is the algorithm?

Suppose the algorithm is implemented on a fast computer that can find and calculate the length of 1 million Hamiltonian cycles per second. By storing the shortest cycle yet discovered as it goes along, we may assume that finding a shortest cycle takes no additional time. The following table gives the length of time for the computer to execute the algorithm.

Number of vertices n	Number of Hamiltonian cycles $(n-1)!/2$	Time taken to execute the algorithm
10	181 444	0.2 seconds
15	4.4×10^{10}	12 hours
20	16.1×10^{16}	1929 years
25	3.1×10^{23}	98 371 449 centuries
30	4.4×10^{30}	1.4×10^{17} years

Clearly the algorithm is not practical to use even for quite moderately sized networks.

Instead we seek

- upper and lower bounds for the minimum total length
- heuristic algorithms that give a 'reasonably good' solution in a reasonable time.

Upper and lower bounds

Suppose k is the true minimum length of a Hamiltonian cycle in a weighted graph G. Although we may be unable to find k (if G has more than just a few vertices), it may be possible to find a range of values in which k lies.

Any number L such that $L \le k$ is called a **lower bound** for k and any number U such that $k \le U$ is called an **upper bound** for k. If we can find both an upper and a lower bound for k then we have 'framed' k within these values:

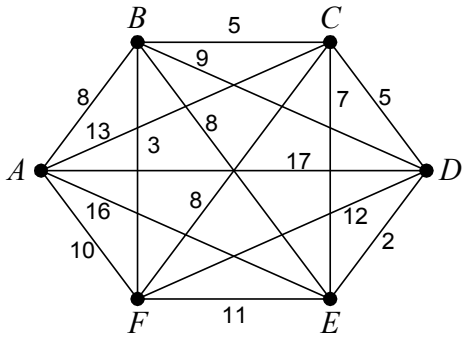
$$L \le k \le U.$$

Upper bounds are easy: the length of *any* Hamiltonian cycle is an upper bound for the TSP since k is the length of the shortest Hamiltonian cycle. However, to find a *good* upper bound, we wish to find a Hamiltonian cycle that is not 'too long'. There are a number of approaches to this.

Nearest Neighbour Algorithm

1. Select any vertex v as starting vertex.
2. Travel along an edge to the nearest vertex that has not yet been visited.
3. Repeat step 2 until all vertices have been visited.
4. Return to the starting vertex v.

Example



Beginning at A: $ABFCDEA$ length = 8 + 3 + 8 + 5 + 2 + 16 = 42.

Beginning at B: $BFCDEAB$ length = 42.

Beginning at C:

Beginning at D:

Beginning at E:

Beginning at F:

Usually the nearest neighbour algorithm produces a reasonably short Hamiltonian cycle. But there are graphs on which it performs very badly indeed. In the following theoretical sense, the nearest neighbour algorithm is very poor: given *any* positive integer N, there exists a complete weighted graph and a starting vertex such the nearest neighbour algorithm produces a cycle whose length is at least N times the true minimum length k.

A slightly more complicated algorithm, which is guaranteed to produce a cycle with length no more than twice the minimum is the nearest insertion algorithm. The basic step in the algorithm is to take a cycle in the graph (that does not include all the vertices) and to enlarge it at minimum 'cost' (that is, with minimum increase in total length).

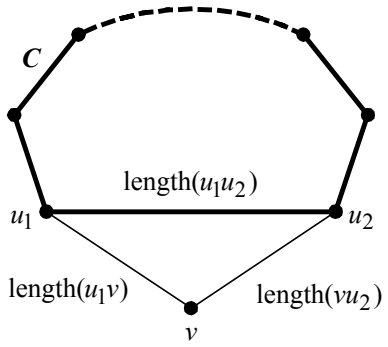
Nearest Insertion Algorithm

1. Choose any vertex v. Select the edge vw with minimum length and let the initial 'cycle' C be vwv.
2. Select an edge of minimum weight joining a vertex in C with a vertex, v say, not in C.

3. This step enlarges the cycle C to include the new vertex v. Choose a pair u_1 and u_2 of adjacent vertices in C such that

$$I = \text{length}(u_1v) + \text{length}(vu_2) - \text{length}(u_1u_2)$$

is a minimum. The expression I represents the increase in total length of C when v is inserted in between u_1 and u_2.

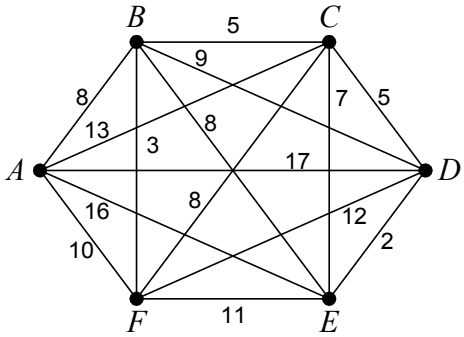


Enlarge C by adding edges u_1v and vu_2 and deleting edge u_1u_2.

4. Repeat steps 2 and 3 until all vertices are included in C.

Example

Using the weighted graph in the previous example.



Step 1. Beginning at vertex A, the first 'cycle' is ABA.

Step 2. The 'nearest' vertex to A or B is F.

Step 3. Since there are only two vertices in C F is inserted between them to give the cycle $ABFA$.

Step 2. The 'nearest' vertex to A, B or F is C.

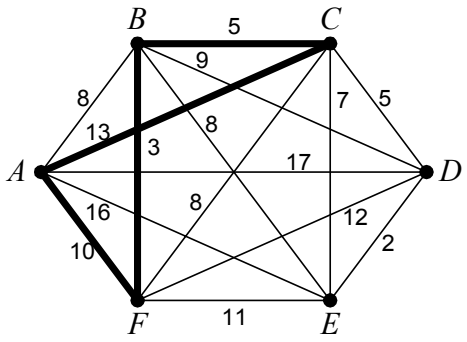
Step 3. $I_{AB} = \text{length}(AC) + \text{length}(CB) - \text{length}(AB) = 13 + 5 - 8 = 10$

$I_{BF} = \text{length}(BC) + \text{length}(CF) - \text{length}(BF) = 5 + 8 - 3 = 10$

$I_{FA} = \text{length}(FC) + \text{length}(CA) - \text{length}(FA) = 8 + 13 - 10 = 11$

We may choose to insert C between A and B or between B and F.

We choose to insert between A and B giving cycle $ACBFA$.



Step 2. The 'nearest' vertex to A, B, C or F is D.

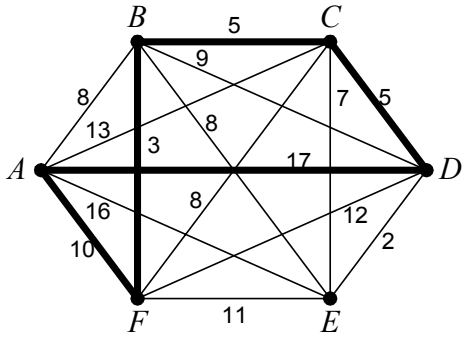
Step 3. I_{AC} = length(AD) + length(DC) – length(AC) = 17 + 5 – 13 = 9

I_{CB} = length(CD) + length(DB) – length(CB) = 5 + 9 – 5 = 9

I_{BF} = length(BD) + length(DF) – length(BF) = 9 + 12 – 3 = 18

I_{FA} = length(FD) + length(DA) – length(FA) = 12 + 17 – 10 = 19

We may choose to insert D between A and C or between C and B.
We choose to insert between A and C giving cycle $ADCBFA$.



Step 2. The 'nearest' vertex to A, B, C, D or F is E.

Step 3. I_{AD} = length(AE) + length(ED) – length(AD) = 16 + 2 – 17 = 1

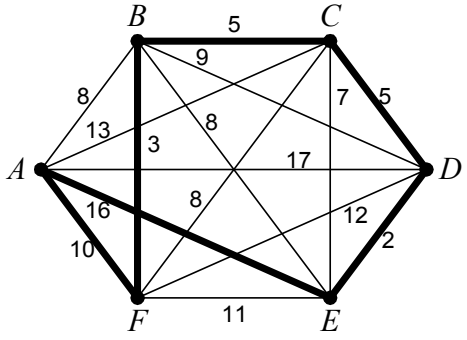
I_{DC} = length(DE) + length(EC) – length(DC) = 2 + 7 – 5 = 4

I_{CB} = length(CE) + length(EB) – length(CB) = 7 + 8 – 5 = 10

I_{BF} = length(BE) + length(EF) – length(BF) = 8 + 11 – 3 = 16

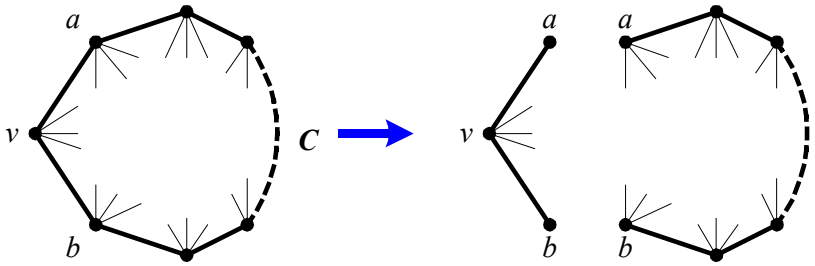
I_{FA} = length(FE) + length(EA) – length(FA) = 11 + 16 – 10 = 17

Insert E between A and D to give Hamiltonian cycle $AEDCBFA$ with total length $16 + 2 + 5 + 5 + 3 + 10 = 41$.



Lower Bounds

Suppose we have a minimum weight Hamiltonian cycle C in a complete weighted graph G. Consider a vertex, v say. The cycle C contains two edges incident with v; call these edges av and vb. If we delete these two edges, the rest of C is a path from a to b. We can think of this path as a special kind of spanning tree in the graph $G - \{v\}$ obtained from G by removing the vertex v.



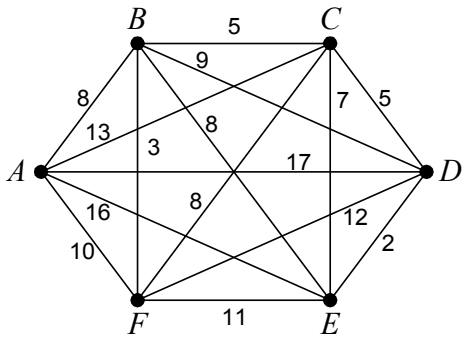
The length of this path is greater than or equal to the weight of a minimum spanning tree in $G - \{v\}$. Also $w(av) + w(vb)$ is greater than or equal to the sum of weights of the two smallest weight edges incident with v.

Thus the following algorithm will produce a lower bound for the TSP for G.

Lower Bound Algorithm

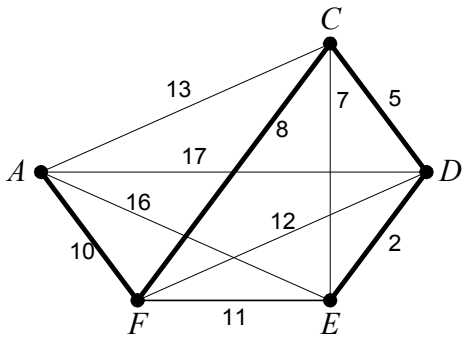
1. Choose any vertex v and delete it and all of its incident edges from the graph.
2. Find a minimum weight spanning tree in the resulting graph – use Kruskal's or Prim's algorithm.
3. Find the two edges incident with v that have the smallest weight.
4. A lower bound is the sum of the weights of the minimum weight spanning tree found at step 2 and the two edges found at step 3.

Example



Step 1. Choose vertex B.

Step 2. A minimum weight spanning tree contains the edges AF, FC, CD, DE, and has weight $10 + 8 + 5 + 2 = 25$.



Step 3. The two edges of smallest weight incident with B are BF and BC;
$w(BF) + w(BC) = 3 + 5 = 8$.

Step 4. Lower bound = 25 + 8 = 33.

I.e. $33 \leq$ weight of minimum weight Hamiltonian cycle

We can obtain a collection of lower bounds by applying the algorithm removing each vertex in turn. Which of the set of lower bounds will be the best?

Chinese Postman Problem or Route Inspection Problem

Problem: *given a weighted graph, find a route of minimum weight that traverses every edge at least once and returns to the starting vertex.*

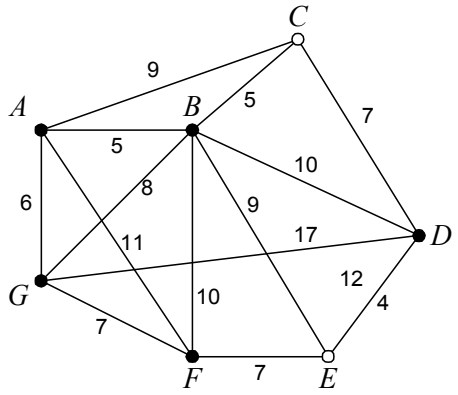
This is a 'postman's problem' because a postman needs to traverse every street (edge) in his or her round so as not to miss out any houses. 'Chinese' refers to the nationality of the mathematician – Mei-ko Kwan – who first considered the problem in 1962, and not to the nationality of the postman!

Clearly, if the graph is Eulerian then any Eulerian trail solves the problem because it traverses each edge exactly once. If the graph is not Eulerian then

- it will have vertices of odd degree
- some edges will need to be traversed more than once.

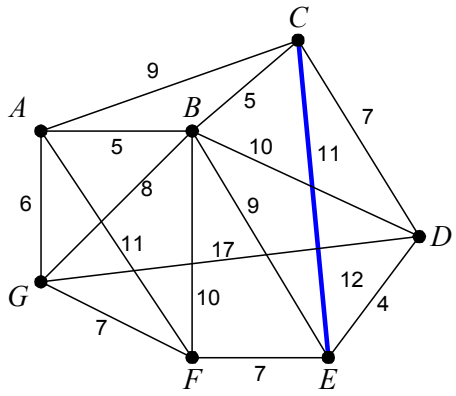
Example

Consider the following weighted graph.



There are two vertices of odd degree C and E. If we were to join these two vertices with an edge then all vertices would have even degree and Fleury's algorithm will find an Eulerian trial in the resulting graph. The newly added edge will correspond to the shortest path in the original graph from C to E and the edges on this shortest path will be repeated.

In this example, we add edge CE with weight 11 corresponding to the path CDE which is the shortest path from C to E.



Suppose, now, there are four vertices of odd degree in a graph. Then we need to pair off the vertices; for each pair, find the shortest path between the vertices and add an edge to the graph with this weight. However there are three such pairings. For example, if the vertices of odd degree are $A, B, C, D,$ the possible pairings are:

AB and CD
AC and BD
AD and BC

For each pairing we need to determine the weights of the additional edges that need to be added (corresponding to edges of the original graph that will be traversed more than once) and select the pairing that minimises the additional weight.

If there are four vertices of odd degree, say $A, B, C, D, E, F,$ then there are 15 possible pairings:

$AB,\ CD,\ EF$
$AB,\ CE,\ DF$
$AB,\ CF,\ DE$
$AC,\ BD,\ EF$
$AC,\ BE,\ DF$
$AC,\ BF,\ DE$
$AD,\ BC,\ EF$
$AD,\ BD,\ CF$
$AD,\ BE,\ CD$
$AE,\ BC,\ DF$
$AE,\ BD,\ CF$
$AE,\ BF,\ CD$
$AF,\ BC,\ DE$
$AF,\ BD,\ CE$
$AF,\ BE,\ CD$

'Chinese Postman' Algorithm

Step 1 List all the vertices of odd degree.

Step 2 Form all possible pairings of vertices of odd degree.

For each pairing:

- for each pair of vertices find the shortest path between them
- add the weights of the shortest paths between all pairs of vertices.

Step 3 Choose the pairing with the smallest sum of weights of the shortest paths between all pairs of vertices.

For each pair of vertices, add an edge to the original graph with weight equal to the length of the shortest path between them.

Step 4 Use Fleury's algorithm to find an Eulerian trail in the resulting graph. Each of the added edges needs to be interpreted as a path of edges in the original graph that are traversed more than once.

Complexity of Chinese Postman Algorithm

The complexity of the algorithm depends on the number of ways of forming pairings. How many ways are there to do this?

With 2 vertices of odd degree:	there is 1 pairing.
With 4 vertices of odd degree:	there are 3 pairings.
With 6 vertices of odd degree:	select any vertex; there are 5 choices of a vertex to pair with it; then there are 4 vertices left and we know there are 3 pairings in this case. Hence there are $5 \times 3 = 15$ pairings.
With 8 vertices of odd degree:	select any vertex; there are 7 choices of a vertex to pair with it; then there are 6 vertices left and we know there are 5×3 pairings in this case. Hence there are $7 \times 5 \times 3 = 105$ pairings.
With 10 vertices of odd degree:	there are $9 \times 7 \times 5 \times 3 = 945$ pairings.

In general, with $2n$ vertices of odd degree, there are $(2n-1) \times (2n-3) \times \ldots \times 5 \times 3 \times 1$ pairings.

We can write this more compactly as follows:

$$(2n-1)\times(2n-3)\times\ldots\times 5\times 3\times 1$$

$$= \frac{2n\times(2n-1)\times(2n-2)\times(2n-3)\times(2n-4)\times\ldots\times 6\times 5\times 4\times 3\times 2\times 1}{2n \times (2n-2) \times (2n-4)\times\ldots\times 6 \times 4 \times 2}$$

$$= \frac{(2n)!}{2(n)\times 2(n-1)\times 2(n-2)\times \ldots \times 2(3)\times 2(2)\times 2(1)}$$

$$= \frac{(2n)!}{2\times 2\times 2\times \ldots \times 2\times n\times(n-1)\times \ldots \times 2\times 1}$$

$$= \frac{(2n)!}{2^n n!}$$

The following table shows how this expression grows as n increases.

n	$\frac{(2n)!}{2^n n!}$
1	1
2	3
3	15
4	105
5	945
6	10 395
7	135 135
8	2 027 025
9	34 459 425
10	654 729 075
11	13 749 310 575
12	316 234 143 225
13	7 905 853 580 625
14	213 458 046 676 875
15	6 190 283 353 629 375
16	1 918 987 839 622 510 625
17	63 326 569 870 762 850 625
18	221 643 095 476 699 771 875
19	8 200 794 532 637 891 559 375
20	319 830 968 772 877 770 815 625 = 3×10^{23}