Eindhoven University of Technology

Applied Mathematics

Combinitorial Optimization

The Chinese Postman Problem in undirected and directed graphs

Bachelor Final project

Author:
Lucy Verberk

Supervisor:
Dr. Judith Keijsper

July 11, 2019

Abstract

In this report the Chinese Postman Problem (CPP) for undirected, directed and mixed graphs will be considered. Solution methods for the undirected and directed CPP will be discussed. For the mixed CPP, some suggestions for further research are mentioned. An application of routing gritters and snow-shovel trucks in the city of Eindhoven in the Netherlands will be considered.

Contents

Chapter 1

Introduction

Consider the following problem: a postman wishes to deliver his letters covering a minimum distance and returning to his starting point. It is obvious that he needs to traverse all roads at least once, to deliver his letters there, but he should avoid covering too many roads more than once.

This problem is known as the Chinese Postman Problem, but is also sometimes referred to as the Route Inspection Problem. It is a famous combinatorial-optimization problem and was first studied by the Chinese mathematician Meigu Guan (Romanised as Mei-Ko Kwan) in 1960 [8].

Some variations of this problem:

- The Open Chinese Postman Problem, where the postman does not have to end at his starting point.
- The Directed Chinese Postman Problem, in which there are only one-way roads.
- The Mixed Chinese Postman Problem, in which there are both one-way and two-way roads.
- The Windy Postman Problem, in which the cost for traversing a road may differ when traversing it in one or the other direction. This can be due to, for example, wind that is going in a certain direction or a road that goes uphill/downhill depending on the direction you take.

The problem and its variations have many practical applications. In this report the undirected, directed and mixed Chinese Postman problem will be discussed. Also an application in the city of Eindhoven in the Netherlands will be considered. Namely, routes for gritters and snow shovel trucks.

Chapter 2

Prerequisites

In this report, basic knowledge from graph theory, totally unimodular matrices and polyhedra is used. Therefore these will be discussed as prerequisites here.

2.1 Graph Theory

An **undirected graph** $G = (V, E)$ consists of a set of **vertices** V with a set of **edges** E that connect the vertices in V. An edge $e = \{u, v\}$ is an unordered pair of vertices that are connected in the graph. The nodes u and v are called the **endpoints** of the edge e, e is said to be **incident** with u and v and u and v are said to be **adjacent**. A graph is called **simple** if between every pair of vertices, there is at most one edge. Otherwise it is called a **multigraph**. The **degree** of a vertex is the number of edges that are incident with that vertex. It is denoted as $d(v)$ for each vertex v.

A **directed graph** (or digraph) $D = (V, A)$ consists of a set of vertices V and a set of 'directed edges' A called **arcs**. An arc (u, v) is an ordered pair of vertices that are connected in a graph. The arc (u, v) **leaves** from u and **arrives** in v. Define, for a vertex v in V:

$$\delta^+(v) := \{a \in A \mid a \text{ leaves from } v\},$$

$$\delta^-(v) := \{a \in A \mid a \text{ arrives in } v\}.$$

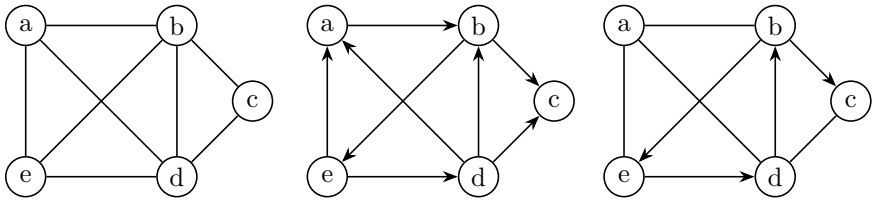
The **outdegree** of v is $d^+(v) := |\delta^+(v)|$ and the **indegree** is $d^-(v) := |\delta^-(v)|$. Let δ be the difference between the in- and outdegree, defined as:

$$\delta(v) = d^+(v) - d^-(v).$$

We call a vertex **balanced** if $\delta(v) = 0$.

A **mixed graph** G is a graph with both undirected and directed edges,

so $G = (V, E \cup A)$. The **degree** of a vertex is the number of edges that are incident with that vertex, both undirected and directed edges. The outdegree and the indegree are the same as defined before. So the **outdegree** is the number of directed edges leaving from a vertex and the **indegree** is the number of directed edges arriving in a vertex. A vertex is called **balanced** if the total degree is even and if the in- and outdegree are the same.



(a) undirected graph (b) directed graph (c) mixed graph

Figure 2.1: examples of graphs

A **walk** from a vertex $u \in V$ to a vertex $v \in V$ (or $u - v$ walk) in a graph is a sequence of vertices $(u = v_0, v_1, \ldots, v_k = v)$ such that v_{i-1} and v_i are connected for $i = 1, 2, \ldots, k$. A **closed walk** is a walk $(v_0, v_1, \ldots, v_k)$ such that $v_0 = v_k$. A **directed** or **mixed walk** is a walk that respects the direction of the arcs it traverses.

A **path** from u to v (or $u - v$ path) is a walk $(u = v_0, v_1, \ldots, v_k = v)$ in which all vertices are distinct. So for all $i, j \in \{0, 1, \ldots, k\}$ such that $i \neq j$: $v_i \neq v_j$. Note that if there is a $u - v$ walk, there is also a $u - v$ path.

Sometimes it might be convenient to denote a walk or a path by the edges (or arcs) it traverses, instead of the vertices. For example in a multigraph, where there can be more than one edge between two vertices.

Given a length function l over the edges (or arcs), $l : E \to \mathbb{R}_+$, the length of a walk $W = (e_0, e_1, \ldots, e_k)$, denoted by its edges, is defined as

$$\sum_{i=0}^{k} l(e_i).$$

An undirected graph is called **connected** if between every pair of vertices there exists a path.

A directed graph is called **strongly connected** if there is a directed path between every pair of vertices. So for two vertices u and v, there is a directed path from u to v, *and* there is a directed path from v to u. Note that in a strongly connected graph, every vertex has both an in- and outdegree of at least 1.

A mixed graph is called **weakly connected** if, considering all edges as undirected, the graph is connected. A mixed graph is called **strongly con-**

nected if the digraph obtained by replacing every undirected edge $\{u, v\}$ by a pair of arcs, (u, v) and (v, u), is strongly connected.

An **Euler tour** in an undirected/directed/mixed (multi)graph G is an undirected/directed/mixed closed walk that traverses every edge/arc exactly once. If there exists an Euler tour in G, then G is called a **Eulerian graph**.

A **postman tour** in an undirected/directed/mixed (multi)graph G is an undirected/directed/mixed closed walk that traverses every edge/arc at least once.

A **matching** in a graph (V, E) is a subset of the edges, $E' \subseteq E$, such that each vertex is incident with at most one edge in E'. A matching is **perfect** if each vertex is incident with exactly one edge in E'. A vertex v is **covered** by a matching if it is incident with a matching edge.

Given a function $b : V \to \mathbb{Z}_+$, a **b-matching** is a function $x : E \to \mathbb{Z}_+$ such that

$$\sum_{e \ni v} x(e) \leq b(v).$$

Note that, in contrast to a normal matching, an edge can be in the b-matching more than once. A vertex v is called **maximally covered** by a b-matching x if

$$\sum_{e \ni v} x(e) = b(v).$$

A b-matching is called **perfect** if each vertex v is maximally covered.

A **complete graph** K_n is a graph on n vertices with an edge between every pair of vertices.

Two sets are said to be **disjoint** if they have no elements in common. And a set of vertices in a graph is said to be **independent** if there is no edge between any pair of vertices in that set.

A **bipartite graph** (or bigraph) $B = (U \cup V, E)$ is a graph whose vertex set consists of two disjoint, independent sets U and V. So all edges connect a vertex from U with a vertex from V. There are no edges between vertices both in U or both in V. A bipartite graph $B = (U \cup V, E)$ is called **complete** if there is an edge between every pair of vertices u and v, for $u \in U$ and $v \in V$.

Lemma 2.1 (Handshaking Lemma)**.** *For every graph $G = (V, E)$, the sum of all vertex degrees is even. Indeed,*

$$\sum_{v \in V} d(v) = 2|E|.$$

Proof. Every edge is incident with two vertices. If you count the degree of a vertex, you count how many edges are incident with this vertex. So in particular, an arbitrary edge $\{u, v\}$ will be counted in the degree of u, as well as in the degree of v. In total, if you sum the degrees of all vertices, you count all edges twice. So

$$\sum_{v \in V} d(v) = 2|E|,$$

and thus the sum of all vertex degrees is even. □

For directed graphs there is a similar Lemma.

Lemma 2.2 (Handshaking Dilemma)**.** *For every directed graph $D = (V, A)$, the sum of all outdegrees equals the sum of all indegrees,*

$$\sum_{v \in V} d^+(v) = \sum_{v \in V} d^-(v).$$

Proof. Each arc is an in-going (out-going) arc for exactly one vertex. So if you sum the indegrees (outdegrees) of all vertices, you count all arcs. Thus indeed,

$$|A| = \sum_{v \in V} d^+(v) = \sum_{v \in V} d^-(v).$$

□

2.2 Totally Unimodular Matrices

A matrix is **totally unimodular** (TU) if all square non-singular submatrices are unimodular. An **unimodular** matrix is a square integer matrix that has determinant 1 or -1. So in other words, a matrix is totally unimodular if all square submatrices have determinant 0, 1 or -1.

Theorem 2.1. *The incidence matrix of a bipartite graph is TU.*

Proof. Let $B = (V \cup W, E)$ be some bipartite graph. Then each element a_{ij} of the incidence matrix A is defined as

$$a_{ij} = \begin{cases} 1 & \text{if edge } j \text{ is incident with vertex } i, \\ 0 & \text{otherwise.} \end{cases} \tag{2.1}$$

From that we can conclude that each column of A has exactly two nonzero elements. Namely two 1's. Assume A is not TU. Then there must exists a smallest square submatrix C that has $\det(C) \notin \{-1, 0, 1\}$. Let us say C is an $n \times n$ matrix. Then C does not have any column of only zeros, because then the determinant would be zero.

Assume C has a column with only one nonzero element. This column can be placed as the first column, because that only changes the sign of the determinant, not the magnitude. For the same reason, this nonzero element can be placed in the top row. So we can just assume C looks as follows:

$$\begin{pmatrix} 1 & \boldsymbol{a}^T \\ \mathbf{0} & C' \end{pmatrix},$$

with $\boldsymbol{a}$ and $\mathbf{0}$ vectors, and C' a $(n-1) \times (n-1)$ matrix. Then by expansion across the first column the following holds:

$$\{-1, 0, 1\} \ni \det(C) = 1 \cdot \det(C').$$

But that means $\det(C') \notin \{-1, 0, 1\}$, which contradicts that C was the smallest submatrix with that property. Therefore the only option left is that C has exactly two nonzero elements, so two 1's, in each column.

Note that for the incidence matrix of a bipartite graph, the rows can be divided into the two sets of vertices V and W, such that when only looking at one set, there is exactly one 1 in each column. That is because each edge is between a vertex of V and a vertex of W. This also holds for C, because there are two 1's in each column of C. If you multiply the rows of vertices in W by -1, which you can do because it can only change the sign of the determinant, each column then has exactly one 1 and one -1. Then each column of C will add up to 0. That means that adding the rows of C gives the all 0 vector, which means the rows are linearly dependent. But that means that the determinant of C is 0. Which is a contradiction with our assumption that $\det(C) \notin \{-1, 0, 1\}$.

So that must mean the incidence matrix A is actually TU! □

Theorem 2.2. *If A is TU, then each of the matrices A^T, $[A, -A]$, $[A, I]$ is TU.*

Proof. A is TU, so $\det(A') \in \{-1, 0, 1\}$, for all square submatrices A' of A.

All square submatrices of A^T, are the transpose of a square submatrix of A. The determinant of a matrix and its transpose are the same. So the determinant of the square submatrices of A^T will also be in the set $\{-1, 0, 1\}$. Therefore, A^T is TU.

Square submatrices of $[A, -A]$ can be completely in A, then the determinant is indeed in $\{-1, 0, 1\}$. They can also be completely in $-A$, then the determinant is in $\pm\{-1, 0, 1\} = \{-1, 0, 1\}$, so that is also okay. But square submatrices can also be a combination of A and $-A$. If one chooses a few columns of A and a few of $-A$, two things can happen. First it could happen that for two of the chosen columns, one is the negative of the other, so for example column i of A and column i of $-A$. Then the determinant is 0, so that is okay. Second it could happen that no two columns are the negative of each other, so if column i of A is chosen, then column i of $-A$ is

not chosen. Lets say c columns of $-A$ are in the submatrix. By multiplying each of those c columns of $-A$ by -1, the determinant changes by a factor of $(-1)^c$. So the determinant of the submatrix is $(-1)^c$ times the determinant of a submatrix of A. Multiplying with ± 1 does not change the fact that the outcome is in the set $\{-1, 0, 1\}$, so in this case it is also okay. Therefore, $[A, -A]$ is TU.

Square submatrices of $[A, I]$ can be completely in A, then it is okay. They can also be completely in I, then the determinant is 0 or 1, so that is also okay. If the square submatrix is a combination of A and I, one can just expand the determinant across the columns of I. Every time you expand to a column of I the determinant becomes ± 1 times a smaller determinant. Continuing like this, you eventually end up with ± 1 times a smaller submatrix of A. So then the outcome will indeed be in the set $\{-1, 0, 1\}$. Therefore, also $[A, I]$ is TU. □

From the theorem it can be derived that if the matrix A is TU, then so is the matrix $\begin{pmatrix} A \\ -A \end{pmatrix}$.

Lemma 2.3. *If U is a unimodular matrix, U^{-1} is also a unimodular matrix.*

Proof. U is unimodular, so it has integer entries and $\det(U) = \pm 1$. From

$$UU^{-1} = I,$$

it can be seen that

$$\det(U)\det(U^{-1}) = \det(I).$$

Which implies

$$\det(U^{-1}) = \det(I)/\det(U) = 1/(\pm 1) = \pm 1.$$

By Cramer's rule:

$$U^{-1} = \frac{1}{\det(U)}\text{adj}(U) = \pm\text{adj}(U),$$

where $\text{adj}(U)$ is the adjugate matrix of U. The **adjugate** of U is the transpose of the cofactor matrix C of U. An element C_{ij} of the cofactor matrix C is defined as the determinant of U without the i'th row and j'th column. This means that if U has only integer entries, so has C and therefore so has $\text{adj}(U)$. That means also U^{-1} has integer entries.

So $\det(U^{-1}) = \pm 1$ and U^{-1} has only integer entries, which means U^{-1} is also unimodular. □

2.3 Polyhedra

A **polyhedron** P is the solution set of a system of finitely many linear inequalities. So $P = \{\boldsymbol{x} \mid A\boldsymbol{x} \leq \boldsymbol{b}\}$, for some matrix A and some vector $\boldsymbol{b}$. A set $H_{c,d}$ is a **hyperplane** if there is a nonzero row vector $\boldsymbol{c}$ and a $d \in \mathbb{R}$, such that $H_{c,d} = \{\boldsymbol{x} \mid \boldsymbol{c}\boldsymbol{x} = d\}$. A hyperplane $H_{c,d}$ is called a **supporting hyperplane** of a polyhedron P if $\max\{\boldsymbol{c}\boldsymbol{x} \mid \boldsymbol{x} \in P\} = d$. A **face** F is the intersection of P and some supporting hyperplane H. So $F = P \cap H$. A point $\boldsymbol{v} \in P$ is a **vertex** of P if $\{\boldsymbol{v}\}$ is a face.

The following lemma is Lemma 28 of [10].

Lemma 2.4. *Let A be an $m \times n$ matrix, $\boldsymbol{b} \in \mathbb{R}^m$ and $P = \{\boldsymbol{x} \in \mathbb{R}^n \mid A\boldsymbol{x} \leq \boldsymbol{b}\}$. Let $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_m$ be the rows of A. Then the following are equivalent for a nonempty set $F \subseteq P$:*

(1) F is a face of P, and

(2) $F = \{\boldsymbol{x} \in P \mid \boldsymbol{a}_i\boldsymbol{x} = b_i$ for all $i \in J\}$ for some $J \subseteq \{1, \ldots, m\}$.

So if F is a face of P, for a subset of the rows of the inequality $A\boldsymbol{x} \leq \boldsymbol{b}$, equality holds. This can be formulated as $A'\boldsymbol{x} = \boldsymbol{b}'$, where A' and $\boldsymbol{b}'$ consist of a subset of the rows of A and $\boldsymbol{b}$, respectively.

Now consider some vertex $\boldsymbol{v}$ of P, then $\{\boldsymbol{v}\}$ is a face of P, so we can use Lemma 2.4:

$$\{\boldsymbol{v}\} = \{\boldsymbol{x} \in P \mid \boldsymbol{a}_i\boldsymbol{x} = b_i \text{ for all } i \in J\} \text{ for some } J \subseteq \{1, \ldots, m\}. \tag{2.2}$$

Theorem 2.3. *If $\boldsymbol{v} \in \mathbb{R}^n$ is a vertex of $P = \{\boldsymbol{x} \mid A\boldsymbol{x} \leq \boldsymbol{b}\}$, with*

$$\{\boldsymbol{v}\} = \{\boldsymbol{x} \in P \mid \boldsymbol{a}_i\boldsymbol{x} = b_i \text{ for all } i \in J\} \tag{2.2a}$$

for some $J \subseteq \{1, \ldots, m\}$, then $|J| \geq n$.

Proof. From (2.2a) we see that there is only one vertex that satisfies $\boldsymbol{a}_i\boldsymbol{x} = b_i$ for all $i \in J$. It is only possible to have only one solution in $\mathbb{R}^n$ if you have a number of equations of which n are independent. So that must mean $|J| \geq n$. □

From this theorem we conclude that if $\boldsymbol{v}$ is a vertex of P, then for at least n rows of the inequality $A\boldsymbol{v} \leq \boldsymbol{b}$, equality holds. However, only independent equations are needed to determine the solution. So, there is an independent subset of n rows of the inequality $A\boldsymbol{v} \leq \boldsymbol{b}$, for which equality holds. That means we could write $A'\boldsymbol{v} = \boldsymbol{b}'$, where A' and $\boldsymbol{b}'$ are a subset of n independent rows of A and $\boldsymbol{b}$. That means A' has full rank.

The following two definitions of convex combination and polytope are taken from [10].

We say that $y \in \mathbb{R}^n$ is a **convex combination** of $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k \in \mathbb{R}^n$, if there exists $\lambda_1, \ldots, \lambda_k \geq 0$ such that $y = \lambda_1 \boldsymbol{x}_1 + \ldots + \lambda_k \boldsymbol{x}_k$ and $\lambda_1 + \ldots + \lambda_k = 1$.

A **polytope** is the convex hull of a finite set of points, i.e. P is a polytope if there exists k vectors $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k$ such that

$$\begin{aligned} P &= \text{conv.hull}\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k\} \\ &= \{\lambda_1 \boldsymbol{x}_1 + \ldots + \lambda_k \boldsymbol{x}_k \mid \lambda_1 + \ldots + \lambda_k = 1 \text{ and } \lambda_1, \ldots, \lambda_k \geq 0\}. \end{aligned}$$

The following theorem is a rewritten version of a statement in exercise 21 of chapter 3 of [10].

Theorem 2.4. *Let $\boldsymbol{x} \in P$, for some polyhedron P. Then $\boldsymbol{x}$ is a vertex of P if and only if $\boldsymbol{x} \notin$ conv.hull$\{P \setminus \{\boldsymbol{x}\}\}$.*

Note that this theorem implies that all vectors in $P = \text{conv.hull}\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k\}$, that are not one of the vectors $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k$, are not vertices. If some $\boldsymbol{x}_i$ is a convex combination of the other $\boldsymbol{x}_j$'s, $j \neq i$, then it will be in the convex hull of those points. So then $\boldsymbol{x}_i$ will also not be a vertex. From all this we can conclude that the vertices of $P = \text{conv.hull}\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k\}$ form a subset of the vectors $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k$.

Theorem 2.5. *Let A be TU and $\boldsymbol{b}$ be integral. Let P be the polyhedron $P = \{\boldsymbol{x} \mid A\boldsymbol{x} \leq \boldsymbol{b}\}$. If $\boldsymbol{x}$ is a vertex of P, then $\boldsymbol{x}$ is integral.*

Proof. If $\boldsymbol{x}$ is a vertex of P, then there is a matrix A' of full rank that consists of a subset of rows of A such that $A'\boldsymbol{x} = \boldsymbol{b}'$, where $\boldsymbol{b}'$ consists of a subset of corresponding rows of $\boldsymbol{b}$. A' has full rank, so $\det(A') \neq 0$. A' is a square submatrix of A and A is TU, so that means A' must be unimodular. As we have seen in Lemma 2.3, then also $(A')^{-1}$ is unimodular. Moreover, $\boldsymbol{b}$ is integral, so also $\boldsymbol{b}'$ is integral.

We can now write $\boldsymbol{x}$ as $\boldsymbol{x} = (A')^{-1}\boldsymbol{b}'$. Because both $(A')^{-1}$ and $\boldsymbol{b}'$ are integral, $\boldsymbol{x}$ must also be integral. □

The following theorem is from [10] (Theorem 27):

Theorem 2.6. *Let $P \subseteq \mathbb{R}^n$. Then P is a polytope if and only if P is a bounded polyhedron.*

Linear optimisation problems in polytopes have the following property.

Theorem 2.7. *Let P be a polytope. Then a problem of the form $\min\{\boldsymbol{c}^T\boldsymbol{x} \mid \boldsymbol{x} \in P\}$ attains its optimum solution always in at least one vertex.*

Proof. First note that P is a polytope, so by Theorem 2.6 it is a bounded polyhedron. Because P is bounded, an optimisation problem over P will always have an optimal solution.

Assume the minimum value of the problem is m, so:

$$\min\{\boldsymbol{c}^T\boldsymbol{x} \mid \boldsymbol{x} \in P\} = m.$$

Now define the hyperplane $H = \{\boldsymbol{x} \mid \boldsymbol{c}^T\boldsymbol{x} = m\}$, then H is a supporting hyperplane for P. That means that $P \cap H$ is a face. Define

$$F = P \cap H = \{\boldsymbol{x} \mid \boldsymbol{x} \in P, \boldsymbol{c}^T\boldsymbol{x} = m\}.$$

Then each point on F is a point in P and each point satisfies $\boldsymbol{c}^T\boldsymbol{x} = m$, so in each point on F the optimum solution is attained. In particular, in a polytope there is always at least one vertex on a face, so there is always at least one vertex that attains the minimum. □

Theorem 2.8. *If A is TU, $\boldsymbol{b}$ is integral and $P = \{\boldsymbol{x} \mid A\boldsymbol{x} \geq \boldsymbol{b}, \boldsymbol{x} \geq 0\}$ is a polytope, then a problem of the form $\min\{\boldsymbol{c}^T\boldsymbol{x} \mid A\boldsymbol{x} \geq \boldsymbol{b}, \boldsymbol{x} \geq 0\}$ attains an integral optimum solution in a vertex.*

Proof. The set of solutions for the minimisation problem is the polyhedron $\{\boldsymbol{x} \mid A\boldsymbol{x} \geq \boldsymbol{b}, \boldsymbol{x} \geq 0\}$. Note that this can be rewritten as the polyhedron $\{\boldsymbol{x} \mid A'\boldsymbol{x} \leq \boldsymbol{b}'\}$, with $A' = \begin{pmatrix} -A \\ -I \end{pmatrix}$ and $\boldsymbol{b}' = \begin{pmatrix} -\boldsymbol{b} \\ \boldsymbol{0} \end{pmatrix}$. Here A' is still TU by Theorem 2.2, and $\boldsymbol{b}'$ is still integral. So then we know by Theorem 2.5 this polyhedron has integral vertices.

By Theorem 2.7 we know there is at least one vertex in which the optimal solution is attained, if P is a polytope. Taking this together means there is an integral optimal solution, that is attained in at least one vertex, since P is a polytope. □

Chapter 3

The Undirected Problem

The problem we will discuss in this chapter is the following.

Problem 3.1. Given a connected, undirected graph $G = (V, E)$ with length function $l : E \to \mathbb{R}_+$, find a minimum length postman tour.

If G is Eulerian the problem is easy. There exists at least one Euler tour, and all possible Euler tours have the same length:

$$\sum_{e \in E} l(e).$$

Any Euler tour is an optimal solution in this case.

If G is not Eulerian, certain edges need to be traversed more than once, to be able to traverse all edges at least once. To find a minimum solution, one has to find a set of edges of minimum total length, that are traversed more than once.

Theorem 3.1 (Euler)**.** *Let G be a connected, undirected (multi)graph. Then G is Eulerian if and only if every vertex in G has even degree.*

Proof. "⇒" G is an Eulerian graph, so there exists at least one Euler tour. Assume there is a vertex v that has odd degree. Every time the tour passes this vertex, it enters the vertex via one edge, and leaves the vertex via an *other* edge. This is because all edges are visited exactly once. So the vertex has at least these two edges. If the tour passes this vertex again, it has to enter and leave via two *other* edges. So still an even number of edges incident with this vertex are used. To use an odd number of edges, the tour has to only leave, or enter this vertex once. But that must mean it is either the beginning or ending of the tour. But an Euler tour has the same start and endpoint, so it must be both the start and the end of the tour, and then again an even number of edges is used.

So all vertices of G must have even degree.

"⇐" Assume there is some longest walk W visiting all edges at most once, that is not an Euler tour. If this is not a closed walk, it ends in some vertex. But that means there is an odd number of edges incident with this vertex that were used by the walk, but the degree of this vertex is even. So there is at least one edge incident with this vertex not yet used in the walk W. Therefore this edge can be added to W, which makes W longer and keeps the property that all edges are visited at most once. So that is a contradiction.

If W was a closed walk, there is at least one edge e of the graph that is not used in the walk. But because the graph is connected, this edge is connected, via some path P, to a vertex v on W. One can assume that P only consists of unused edges, because otherwise the edge e was already connected to W via some other vertex than v. So P can be added to W to create a longer walk, which contradicts that W is the longest walk. □

From this theorem we conclude that if a graph is not Eulerian it must have vertices of odd degree. Define T as the set of all vertices with odd degree. From the Handshaking Lemma, see Lemma 2.1, we can can deduce that $|T|$ must be even.

A **T-join** is a subset $F \subseteq E$ such that in the graph $G' = (V, F)$, $d(v)$ is odd for each $v \in T$ and $d(v)$ is even for $v \notin T$.

Theorem 3.2. *Let $G = (V, E)$ be a graph and let $l : E \to \mathbb{R}_+$ be a positive length function. A minimum cardinality/length T-join F is a collection of paths pairwise connecting the vertices in T.*

Proof. We will prove this by induction on the size of T. Because $|T|$ is always even, the base case is $|T| = 2$.

Base case ($|T| = 2$): let $T = \{s, t\}$. Let F be a minimum cardinality/length T-join. Then in the graph $G' = (V, F)$, s and t are the only vertices of odd degree. In G', start a walk in s, then one can keep walking until one comes at t. This is because each vertex, except for s and t, has even degree. So if one enters a vertex that is not s or t, there must also be an unused edge to leave that vertex. If one comes back in s, then there is an even number of edges incident with s used, so there must be another unused edge that can be used to leave s again. Therefore there must be an $s - t$ walk.

If there is a closed sub-walk W in this $s - t$ walk, then the edges of this closed walk can be deleted from F, without losing the property that F contains an $s - t$ path. This makes the cardinality/length of the walk smaller, because the cardinality/length of all edges is positive. So in a minimum cardinality/length T-join there will be no closed walks. Which means F is the set of edges of an $s-t$ path. So in the case $|T| = 2$, the theorem is correct.

Induction step: let the set F be a minimum cardinality/length T-join

with $|T| = 2k$, $k \geq 1$.
Induction hypothesis: if $|T| \leq 2(k-1)$, then every T-join of minimum cardinality/length is a collection of paths pairwise connecting the vertices in T. Consider some vertex $s \in T$. Similar to the base case, it can be argued that there is an $s-t$ path in $G' = (V, F)$, for some $t \in T$. Call the edge set of this path P. Now consider the set $T' = T \setminus \{s, t\}$, then $|T'| = 2(k-1)$. Delete the edges of the $s-t$ path P from F to create a T'-join: $F' = F \setminus P$.

If F' is not a minimum cardinality/length T'-join then there would be another T'-join F'' with smaller cardinality/length. Adding P to F'' would create a T-join with smaller cardinality/length than the minimum cardinality/length T-join F. That is a contradiction, so F' must be a minimum cardinality/length T'-join.

Then by the induction hypothesis, F' is a collection of paths pairwise connecting the vertices in T'. Then adding the edges in P to F', to get F, gives a collection of paths pairwise connecting the vertices in T. So F is a collection of paths pairwise connecting the vertices in T.

By induction on $|T|$ we have proven that a minimum cardinality/length T-join is a collection of paths pairwise connecting the vertices in T. □

Consider some minimum length T-join F. According to the theorem above, F is a collection of paths connecting the vertices in T. Consider some edge e, this edge is possibly in several of such paths. Define for each edge e, x_e as the number of times edge e is in the T-join.

Theorem 3.3. *Let G be a graph and l be a positive length function. Let F be a minimum length T-join and for each edge e let x_e be the number of times edge e is F. Then x_e is either 0 or 1 for all edges e.*

Proof. We have a minimum length T-join F, so according to Theorem 3.2 F is a collection of paths connecting the vertices in T. Assume F has the fewest possible copies of all edges. Let L be the length of F. Assume there is an edge e such that $x_e \geq 2$. Then e is in at least two distinct paths in (V, F). Suppose e is in the paths $(a, \ldots, b)$ and $(c, \ldots, d)$, and maybe more paths.

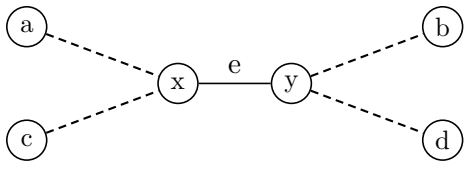


Figure 3.1: paths $(a, \ldots, b)$ and $(c, \ldots, d)$

Now consider a set F' that is almost the same as F, the only change is that the edges of the paths $(a, \ldots, b)$ and $(c, \ldots, d)$ are replaced by the edges

of the walks $(a, \ldots, x, \ldots, c)$ and $(b, \ldots, y, \ldots, d)$. So F' contains the same edges as F, except that it contains two fewer copies of e. That means all vertices in V have the same degree in the graphs (V, F) and (V, F'), only the degrees of x and y are two smaller in the graph (V, F'). That does not change the parity of the degrees in x and y, so F' is also a T-join. The length L' of F' is:

$$L' = L - 2l(e) < L,$$

because $l(e)$ is positive. That contradicts the fact that F is a minimum length T-join. So our assumption that $x_e \geq 2$ for some edge must be wrong. □

From the previous two theorems it can be concluded that a minimum length T-join is a collection of paths connecting the vertices in T, such that each edge e is not or once in the T-join.

Now create an auxiliary graph G' from G, by putting x_e extra copies of e in G'. Note that this new graph G' has the property that all vertices have even degree. Therefore we have found a Eulerian graph. Then the problem is solved, each Euler tour is an optimal solution. Back to the original graph G this translates to the postman tour where each edge e must be traversed $x_e + 1$ times. To make sure it is the optimal postman tour for G, one has to find a T-join of minimum length.

Theorem 3.4. *Let $G = (V, E)$ be a graph with positive length function $l : E \to \mathbb{R}_+$. Let T be the set of vertices of odd degree in G. A minimum length T-join is an optimal solution for the Chinese Postman Problem in G.*

Proof. The length of a postman tour is:

$$\sum_{e \in E} l(e) + \sum_{e \in F} l(e),$$

where F is the multi-set of edges that are traversed more than once. Or, in other words, F is the set of edges that are added to the graph, to make it Eulerian. The first sum is always the same, so when minimising the length of the postman tour it is only necessary to minimise the length of the edges in F.

Adding F must make the graph Eulerian, so the graph $(V, E \cup F)$ only has vertices of even degree. To make all degrees even in that graph it must be that in the graph (V, F), $d(v)$ is odd for all $v \in T$ and $d(v)$ is even for all $v \notin T$. But that is the definition of a T-join!

So the set F of edges that are traversed more than once, is a T-join. Then minimising the total length over all edges in F results in a minimum length T-join. Therefore a minimum length T-join is indeed an optimal solution for the Chinese Postman Problem in a graph G. □

From the previous two theorems, it can be concluded that in an optimal postman tour, created by a minimum length T-join, each edge is traversed once or twice. Once if it is not in the T-join and twice if it is in the T-join.

But how should one find such a T-join? Start by making another auxiliary graph: a complete graph $K_{|T|}$, consisting of all vertices in T and edges between all pairs of vertices. Then all edges get assigned lengths: if $s, t \in T$, then the edge $\{s, t\}$ gets the length of the shortest $s - t$ path in the original graph G. And the edge is also associated with such a shortest path. Now our new graph $K_{|T|}$ consists of all shortest paths between the vertices of T. Then it leaves us to find a perfect matching of minimum length in this graph.

Theorem 3.5. *In a complete graph on an even number of vertices, a perfect matching exists.*

Proof. Assume it is not true. So there is a complete graph on an even number of vertices, with a maximum cardinality matching that is not perfect. That means that there are at least two vertices that are not incident with an edge from the matching. But the graph is complete, so there is an edge between these two vertices. This edge is not in the matching yet, and because its endpoints are not incident with any edges from the matching, this edge can be added to the matching. This is in contradiction with our assumption that it was a maximum cardinality matching. □

Theorem 3.6. *Let $G = (V, E)$ be a graph, l be a positive length function and let T be the set of all vertices of odd degree in G. Let K be the auxiliary complete graph on the vertices of T. Then a minimum length perfect matching in the auxiliary graph K corresponds with a minimum length T-join.*

Proof. Let L be the minimum length of a perfect matching in the auxiliary graph and let L' be the minimum length of a T-join.

Suppose $L < L'$. Let M be a perfect matching in K of weight L. Each edge $e \in M$ corresponds with a path in G that connects two vertices in T. Taking all the paths that corresponds with a matching edge together, gives also a total length L. Let the edge set of those paths be F. Since M is a perfect matching, each vertex in T is an endpoint of one of the paths. In the graph (V, F), the endpoints of the paths have odd degree, and all other points have even degree. Thus in the graph (V, F), $d(v)$ is odd for all vertices in T and even otherwise. Which is the definition of a T-join. So a perfect matching in the auxiliary graph of length L corresponds to a T-join F in the original graph, also of length L. But that means we have found a T-join of smaller length than the minimum length T-join. That is a contradiction.

Suppose $L > L'$. Let F' be a T-join of length L'. By Theorem 3.2, F' is a collection of paths pairwise connecting the vertices of T. Such paths

correspond with edges in the auxiliary graph. An edge $e = \{u, v\}$ in the auxiliary graph has the length of a shortest $u - v$ path in G, while the paths in F might not be shortest paths. The paths are pairwise connecting the vertices of T, so each vertex in T is the endpoint of exactly one such path. Which means that the edges corresponding to the paths, form a perfect matching in the auxiliary graph, of at most length L'. So a minimum length T-join in the original graph of length L' corresponds to a perfect matching in the auxiliary graph, of at most length L'. But that means we have found a perfect matching of smaller length than the minimum length of a perfect matching. That is a contradiction.

We have found that neither $L < L'$ nor $L > L'$ is true, so $L = L'$. Therefore a minimum length perfect matching in the auxiliary graph corresponds with a minimum length T-join in the original graph. □

Example 3.1. Consider the graph in Figure 3.2, the edges have no given length, so it is assumed all edges have length one.

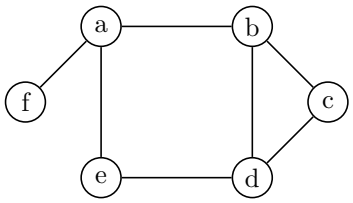


Figure 3.2: graph G

From the graph it is seen that $T = \{a, b, d, f\}$ and $|T| = 4$. A new complete graph will be made with these four vertices. The length of the shortest paths between all these vertices is easily seen in the graph. This results in the following graph:

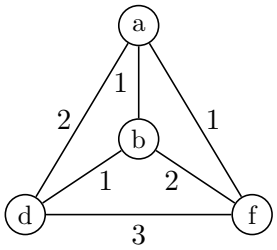


Figure 3.3: K_4 with given length

Note that the edges in this graph represent the shortest path in the original graph. So for example, $\{a.b\}$ represents only the edge $\{a, b\}$ and $\{d, f\}$ can represent the path (d, b, a, f) and the path (d, e, a, f). Both paths have the same length, so one can just choose which path is represented by the edge.

From Figure 3.3 it can be easily seen that $\{\{a, f\}, \{b, d\}\}$ is a minimum length T-join. Both edges in the T-join represent a path in G consisting of only one edge. So there are two edges in the T-join, which both only appear once in this T-join. So we create a new graph G' from G, by putting one extra copy of both edges in G'.

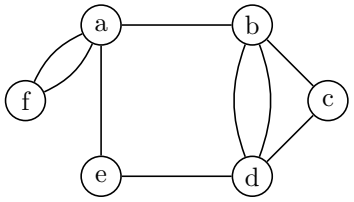


Figure 3.4: graph G' with added edges

Now we see that all vertices indeed have even degree, so the graph is Eulerian. This graph can be translated back to the original graph G by giving each edge a value, which represents exactly how many times it must be traversed.

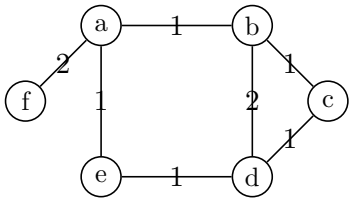


Figure 3.5: graph G with added values for the edges

Now any Euler tour for G' is an optimal postman tour for G, for example: $(a, b, c, d, b, d, e, a, f, a)$. Note that it is not necessary to specify which edge between a and f one uses in the Euler tour of graph G', because they both translate back to the same edge in G. However, if G would be a multigraph, it would be important to specify when which edge is used.

How to find shortest $s - t$ paths, minimum weight perfect matchings and Euler tours will be discussed in the following sections.

3.1 Finding a Shortest $s - t$ Path

In general it is not so easy to immediately see the shortest $s-t$ path, therefore Dijkstra's algorithm for finding shortest path from a vertex s will be used. This algorithm can only be used if the length function is nonnegative. Since the graphs in this problem represent road networks for the postman, the lengths of the edges can represent the lengths of the respective real roads,

or maybe the time it takes to travel over a road, the cost of the petrol used, etcetera. All these things are of course represented by nonnegative numbers.

The algorithm calculates the shortest path from s to all other vertices. In this case one only wants the shortest path between all pairs s, $t \in T$. So when executing the algorithm on a vertex $s \in T$, it can be terminated when the distances to all $t \in T$, are determined. The shortest paths can be determined from the predecessors.

Here is a variant of Dijkstra's algorithm (Section 7.2 of [13]), for this specific case:

Algorithm 1: Dijkstra
Input: A graph $G = (V, E)$ with a nonnegative length function $l : E \to \mathbb{R}^+$, a set T, and a vertex $s \in V$.
Output: Shortest distance from s to all vertices in T.

```
1  U = V
2  f(s) = 0
3  for u ≠ s do
4      f(u) = ∞
5  while U contains vertices that are in T do
6      Find u ∈ U, and if possible u ∈ T, such that f(u) is minimum,
        and remove u from U.
7      for e = {u, v} ∈ E do
8          if f(v) > f(u) + l(e) then
9              f(v) = f(u) + l(e)
10             pred(v) = u
```

The algorithm has a running time of $O(|V^2|)$, and has the following results:

- Upon termination of the algorithm $f(t)$ is the shortest distance from s to t, for all $t \in T$.
- Upon termination, for all $t \in T$ the shortest $s-t$ path can be seen in the tree rooted in s, given by (V, E'), where $E' = \{\{\text{pred}(v), v\} \mid v \in V \backslash \{s\}\}$. In this case it might not be a shortest path tree, because the algorithm can terminate before U is completely empty. So for some vertices in $U \setminus T$, the distance is not necessarily minimum. If $T = V$, then it does result in a shortest path tree.

Example 3.2. Consider the following graph.

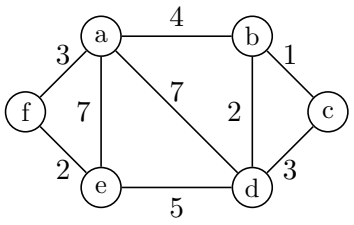


Figure 3.6: graph with length function

Take vertex a as the start vertex s and take $T = V$. The steps of the algorithm will be displayed in the following table. The bold numbers are the minimum values of $f(u)$ for the corresponding iteration.

iteration	a	b	c	d	e	f	predecessor
0	**0**	∞	∞	∞	∞	∞	
1		4	∞	7	7	**3**	p(b)=a, ~~p(d)=a~~, ~~p(e)=a~~, p(f)=a
2		**4**	∞	7	5		p(e) = f
3			**5**	6	5		p(c)=b, p(d)=b
4				6	**5**		
5				**6**			

Table 3.1: execution of Dijkstra's algorithm

For each vertex v in T, the length of the shortest $a - v$ path is the bold number under v in the table. The shortest path tree rooted at a can be found by using the predecessors.

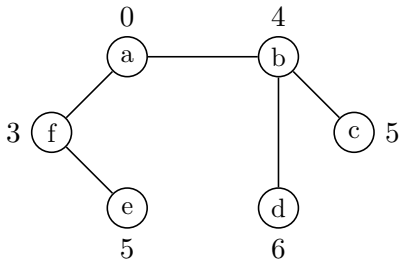


Figure 3.7: shortest path tree

The length of the shortest path from a to a particular vertex is displayed above that vertex. And the shortest path can be found by following the edges. For example, the shortest $a - d$ path is (a, b, d) and has length 6.

3.2 Finding a minimum Length Perfect Matching

In this section we discuss the following problem.

Problem 3.2. Given an undirected complete graph $G = (V, E)$ on an even number of vertices, and given a length function $l : E \to \mathbb{R}_+$, find a minimum length perfect matching.

To find a maximum cardinality matching, one can use the blossom algorithm [2], which was developed by Jack Edmonds and first published in 1965.

But our problem is about a complete graph on an even number of vertices, in which, as seen in Theorem 3.5, a perfect matching exists. Which means maximising the cardinality is not needed. Instead, the length of the edges used in the matching needs to be minimised, while keeping the matching perfect. Edmonds also wrote papers about a generalisation of the blossom algorithm, in which he covers this subject [1, 3].

Let $\boldsymbol{\chi}_M \in \{0,1\}^E$ be the incidence vector of a matching M in $G = (V, E)$. Then the entries χ_e of $\boldsymbol{\chi}_M$ are defined as

$$\chi_e = \begin{cases} 1 & \text{if } e \in M, \\ 0 & \text{if } e \notin M. \end{cases}$$

The **matching polytope** is defined as

$$P = \text{conv.hull}\{\boldsymbol{\chi}_M \mid M \text{ matching in } G\}, \tag{3.1}$$

and the **perfect matching polytope** is similarly defined as

$$P = \text{conv.hull}\{\boldsymbol{\chi}_M \mid M \text{ perfect matching in } G\}. \tag{3.2}$$

We want to have a perfect matching, so all vertices v must be incident with exactly one edge. In other words

$$\sum_{e \ni v} \chi_e = 1, \text{ for all } v. \tag{3.3}$$

Define A as the $V \times E$ incidence matrix of the complete graph G. Then for an element a_{ij} in A:

$$a_{ij} = \begin{cases} 1 & \text{if edge } j \text{ is incident with vertex } i, \\ 0 & \text{otherwise.} \end{cases}$$

Then Equation (3.3) can be rewritten as $A\boldsymbol{\chi}_M = \mathbf{1}$. Here $\mathbf{1}$ is the all one vector of size $|V|$.

The given length function l assigns a length $l(e)$ to each edge e. The total length of the matching is

$$\sum_{e \in M} l(e) = \sum_e l(e)\chi_e.$$

Define $\boldsymbol{l}$ as the vector of $l(e)$'s, then this can be rewritten as $\boldsymbol{l}^T\boldsymbol{\chi}_M$.

Finding a minimum length perfect matching M is equivalent to minimising the total length of the matching ($\boldsymbol{l}^T\boldsymbol{x}$) over all incidence vectors of edge sets ($\boldsymbol{x} \in \{0,1\}^E$), with the constraint that the edge set is a perfect matching ($A\boldsymbol{x} = \mathbf{1}$). So one would like to solve the integer linear program:

$$\min\{\boldsymbol{l}^T\boldsymbol{x} \mid A\boldsymbol{x} = \mathbf{1}, \boldsymbol{x} \in \{0,1\}^E\}. \tag{3.4}$$

Consider the following matrix and vector,

$$A' = \begin{pmatrix} A \\ -A \end{pmatrix}, \boldsymbol{b} = \begin{pmatrix} \mathbf{1} \\ -\mathbf{1} \end{pmatrix},$$

Then the equation $A'\boldsymbol{x} \geq \boldsymbol{b}$ is equivalent to

$$A\boldsymbol{x} \geq \mathbf{1} \wedge -A\boldsymbol{x} \geq -\mathbf{1},$$

or equivalently:

$$A\boldsymbol{x} = \mathbf{1}.$$

Note that $\boldsymbol{x} \geq 0$ and $A\boldsymbol{x} = \mathbf{1}$ together imply $\boldsymbol{x} \leq 1$. Then the problem in Equation (3.4) can also be written as

$$\min\{\boldsymbol{l}^T\boldsymbol{x} \mid A'\boldsymbol{x} \geq \boldsymbol{b}, \boldsymbol{x} \geq 0, \boldsymbol{x} \in \mathbb{Z}^E\}. \tag{3.5}$$

If A would be TU, then as a result of Theorem 2.2, A' would also be TU. And then as can be seen in Theorem 2.8, Equation (3.5), without the constraint $\boldsymbol{x} \in \mathbb{Z}^E$, would have integer optimal solutions. That would be nice, because a linear program is easier to solve than an integer linear program. But in general, incidence matrices are not TU. Consider the following example.

Example 3.3. Consider the incidence matrix of the complete graph on three vertices.

$$\begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

The determinant of this matrix equals:

$$\det \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} = 1 \cdot \det \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} - 1 \cdot \det \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} = -1 - 1 = -2.$$

So this matrix is not totally unimodular. This matrix is a submatrix of all incidence matrices of complete graphs on three or more vertices. So all of those are not totally unimodular. That does not mean the optimal solution to a linear problem with this constraint matrix cannot be integer, but it does mean that the existence of an integer optimal solution is not guaranteed. To guarantee the optimal solution is integer, the constraint $\boldsymbol{x} \in \mathbb{Z}^E$ is necessary.

It would still be nice to have a linear program instead of an integer linear program. Consider the following constraint that Edmonds stated in [1], where the real variables x correspond to edges $e \in E$ of a finite graph G,

> *For every subset S of $2r+1$ nodes in G, where r is a strictly positive integer $\sum x \leq r$ (summed over $x \in R$), where R is the set of variables corresponding to the edges of G with both ends in S.*

In our notation this constraint becomes:

$$\text{for each } S \subseteq V \text{ on an odd number of vertices, } \sum_{e \subset S} x_e \leq \lfloor \frac{1}{2}|S| \rfloor. \qquad (3.6)$$

It is easily verified that the incidence vector of a (perfect) matching satisfies this constraint. Each vertex is incident with exactly one matching edge, thus in a set S of $2r+1$ vertices, there can be at most $2r$ vertices that are incident with a matching edge with both endpoints in S. That means there is at least one vertex left that, in a perfect matching, then must be incident with a matching edge that has exactly one endpoint in S. Note that this holds for *perfect* matchings, not for general matchings. But because we want to find a perfect matching, we suspect that the constraint in Equation (3.6) can be replaced by the following constraint:

$$\text{for each } S \subseteq V \text{ on an odd number of vertices, } \sum_{e \in \delta(S)} x_e \geq 1, \qquad (3.7)$$

where $\delta(S)$ is the set of edges with exactly one endpoint in S. We will come back to this later.

Note that the constraint in Equation (3.6) is not implied by the constraints:

$$A\boldsymbol{x} = \mathbf{1} \wedge \boldsymbol{x} \geq 0.$$

Take for example the complete graph on three vertices, and $x_e = \frac{1}{2}$ for all edges. Then the constraints above are satisfied, but the constraint from Edmonds for $S = V$ is not true: $\frac{3}{2} \not\leq 1$. Also the constraint in Equation (3.7) is not satisfied in this situation for $S = V$: $0 \not\geq 1$, so that one is also not implied by the other two constraints. Note however that in a graph on an odd number of vertices, this constraint can never be satisfied. Which is not weird, because in a graph on an odd number of vertices, no perfect matching can exist.

Now we would like to rewrite both constraints (3.6) and (3.7) in matrix notation. Define $\mathcal{S}$ as the set of all odd subsets S of V, so: $\mathcal{S} = \{S \subseteq V \text{ odd}\}$. We will now first rewrite (3.6).

Each set S contains certain vertices, so we define a $0,1$ vector $\boldsymbol{s}$, that indicates if a vertex is, or is not in S. Then we create a $\mathcal{S} \times V$ matrix T where each row is such a vector $\boldsymbol{s}$. So each row indicates which vertices are in the set of that row. Then we can also create a vector $\boldsymbol{r}$ that contains the value $\lfloor \frac{1}{2}|S| \rfloor$ for each possible odd subset S, in the same order as in T.

Now we want to translate T to a matrix that indicates which edges are in which set, instead of which vertices. You could multiply T by the incidence matrix A of the graph. Then the result is a $\mathcal{S} \times E$ matrix with entries $0, 1, 2$. The entries represent how many endpoints an edge has in a certain subset. But we only want to have the edges that have two endpoints in a certain subset. This can be achieved by dividing each entry by 2, and then taking the floor. So our desired matrix is

$$T' = \lfloor \frac{1}{2} TA \rfloor.$$

So now constraint (3.6) can simply be written as $T'\boldsymbol{x} \leq \boldsymbol{r}$.

Now we will rewrite constraint (3.7). For each set S, $\delta(S)$ contains certain edges. So we define a 0,1 vector $\boldsymbol{s}$, that indicates if an edge is, or is not, in $\delta(S)$. Then we create a $\mathcal{S} \times E$ matrix B where each row is a vector $\boldsymbol{s}$. Now each row indicates, for some set S, which edges are, or are not in $\delta(S)$. With this we can already rewrite the constraint to: $B\boldsymbol{x} \geq \mathbf{1}$.

Now that we have a way to write the constraints in matrix notation, we would like to verify if they, combined with our other constraints, result in the same polyhedron. Note that it is easily verified, that with the requirement $x \in \mathbb{Z}^E$ the two constraints are the same. And because we eventually want to lose the integrality constraint, we have already left it out in the following theorem.

Theorem 3.7. *The following two polyhedra are equal:*

$$P_1 = \{\boldsymbol{x} \mid A'\boldsymbol{x} \geq \boldsymbol{b}, T'\boldsymbol{x} \leq \boldsymbol{r}, \boldsymbol{x} \geq 0\},$$

$$P_2 = \{\boldsymbol{x} \mid A'\boldsymbol{x} \geq \boldsymbol{b}, B\boldsymbol{x} \geq \mathbf{1}, \boldsymbol{x} \geq 0\}.$$

Proof. Whether $\boldsymbol{x} \in P_1$ or $\boldsymbol{x} \in P_2$, $A'\boldsymbol{x} \geq \boldsymbol{b}$ always holds. This equation is equivalent to $A'\boldsymbol{x} = \mathbf{1}$. Which was a rewriting of Equation (3.3):

$$\sum_{e \ni v} x_e = 1, \text{ for all } v. \tag{3.3}$$

Consider the following sum

$$\sum_{v \in S} \sum_{e \ni v} x_e.$$

It sums over all e that are incident with a vertex that is in S. If $e \subset S$, then e is incident with two vertices in S, so that one is counted twice. If $e \in \delta(S)$,

then it is only incident with one vertex in S, so that one is counted once. That means:

$$\sum_{v \in S} \sum_{e \ni v} x_e = 2 \sum_{e \subset S} x_e + \sum_{e \in \delta(S)} x_e.$$

Now using Equation (3.3) on the left hand side:

$$\sum_{v \in S} \sum_{e \ni v} x_e = \sum_{v \in S} 1 = |S|.$$

Assume $S \subseteq V$ and $|S| = 2r + 1$. Taking the two equations above together results in:

$$2 \sum_{e \subset S} x_e + \sum_{e \in \delta(S)} x_e = 2r + 1. \tag{3.8}$$

Assume $\boldsymbol{x} \in P_1$, then we know $T'\boldsymbol{x} \leq \boldsymbol{r}$. Which came from the constraint in Equation (3.6), a bit rewritten:

$$\text{for each } S \subseteq V \text{ on } 2r + 1 \text{ vertices, } \sum_{e \subset S} x_e \leq r. \tag{3.6a}$$

Using this in Equation (3.8), we get:

$$2r + 1 \leq 2r + \sum_{e \in \delta(S)} x_e,$$

for each odd subset $S \subseteq V$ which implies $\sum_{e \in \delta(S)} x_e \geq 1$ for each odd subset $S \subseteq V$. Which is constraint (3.7), which is the same as $B\boldsymbol{x} \geq \mathbf{1}$. Thus if $\boldsymbol{x} \in P_1$, then $\boldsymbol{x}$ satisfies $B\boldsymbol{x} \geq \mathbf{1}$ and obviously $\boldsymbol{x}$ also satisfies the other two constraints of P_2. So $\boldsymbol{x} \in P_2$, which implies $P_1 \subseteq P_2$.

Now assume $\boldsymbol{x} \in P_2$, then we know $B\boldsymbol{x} \geq \mathbf{1}$. Which came from the constraint in Equation (3.7), a bit rewritten:

$$\text{for each } S \subseteq V \text{ on } 2r + 1 \text{ vertices, } \sum_{e \in \delta(S)} x_e \geq 1. \tag{3.7a}$$

Using this in Equation (3.8), we get:

$$2r + 1 \geq 2 \sum_{e \subset S} x_e + 1$$

for each odd subset $S \subseteq V$. That implies $\sum_{e \subset S} x_e \leq r$ for each odd subset $S \subseteq V$. That is the same as constraint (3.6), which is the same as $T'\boldsymbol{x} \leq \boldsymbol{r}$. Thus if $\boldsymbol{x} \in P_2$, then $\boldsymbol{x}$ satisfies $T'\boldsymbol{x} \leq \boldsymbol{r}$ and obviously $\boldsymbol{x}$ also satisfies the other two constraints of P_1. So $\boldsymbol{x} \in P_1$, which implies $P_2 \subseteq P_1$.

So we have $P_1 \subseteq P_2$ and $P_2 \subseteq P_1$, thus P_1 and P_2 must be equal. □

As a result:

$$\min\{\boldsymbol{l}^T\boldsymbol{x} \mid \boldsymbol{x} \in P_1\} = \min\{\boldsymbol{l}^T\boldsymbol{x} \mid \boldsymbol{x} \in P_2\}.$$

Now that we have established that we can use any one of the two constraints (3.6) and (3.7), we would like to know if with one of these extra constraint, the integrality requirement can be left out. In Section 5.4 of [14] the following theorem is stated (Corollary 5.6a).

Theorem 3.8 (Edmonds' perfect matching polytope theorem)**.** *The perfect matching polytope of any graph $G = (V, E)$ is determined by*

$$\begin{array}{lrll} \text{(i)} & x_e & \geq 0 & \text{for each } e \in E; \\ \text{(ii)} & \sum_{e \ni v} x_e & = 1 & \text{for each } v \in V; \\ \text{(iii)} & \sum_{e \subset \delta(U)} x_e & \geq 1 & \text{for each odd subset } U \text{ of } V. \end{array}$$

This theorem uses all the constraints that we also have in P_2, so we will from now on use P_2. From this theorem we can conclude that $P_2 = P$, for the perfect matching polytope P (3.2) of a graph G. From Theorem 2.4 we can deduce that all vertices of P are incidence vectors $\boldsymbol{\chi}_M$ for matchings M, which are integral vectors. So P, and therefore also P_2, has only integer valued vertices. Then we can conclude from Theorem 2.7, that there is always at least one vertex in which the optimum is attained. So we now know there is always at least one integral optimal solution if P has at least one vertex. Thus from the theorem above we can conclude that adding the constraint $B\boldsymbol{x} \geq \mathbf{1}$ to the ILP in (3.5) will indeed imply that integral optimal solutions exists, without the integrality requirement. Because all vertices are integral.

Note that P has at least one vertex if G has at least one perfect matching.

Adding the constraint $B\boldsymbol{x} \geq \mathbf{1}$ to the ILP in (3.5), and leaving out the integrality constraint, results in the LP:

$$\min\{\boldsymbol{l}^T\boldsymbol{x} \mid A'\boldsymbol{x} \geq \boldsymbol{b}, B\boldsymbol{x} \geq \mathbf{1}, \boldsymbol{x} \geq 0\}. \tag{3.9}$$

Theorem 3.9. *The optimal solution of the ILP in (3.5) is equal to the optimal solution of the LP in (3.9).*

Proof. By Theorem 3.8, we know that $P_2 = P$. By Theorem 2.7, we know there is always a vertex in which the optimal solution is attained. All vertices of the perfect matching polytope are incidence vectors of perfect matchings. So the LP in (3.9) attains its minimum for some $\boldsymbol{\chi}_M$, where M is a perfect matching.

The set Q formed by all $\boldsymbol{x}$ that satisfy the constraints of the ILP in (3.5) consists of perfect matching vectors. That means $Q \subseteq P$. From that we can conclude that

$$\min\{\boldsymbol{l}^T\boldsymbol{x} \mid \boldsymbol{x} \in Q\} \geq \min\{\boldsymbol{l}^T\boldsymbol{x} \mid \boldsymbol{x} \in P\} = \boldsymbol{l}^T\boldsymbol{\chi}_M.$$

But $\boldsymbol{\chi}_M$ is a perfect matching vector, thus $\boldsymbol{\chi}_M \in Q$. That means

$$\min\{\boldsymbol{l}^T\boldsymbol{x} \mid \boldsymbol{x} \in Q\} \leq \boldsymbol{l}^T\boldsymbol{\chi}_M.$$

From the above two inequalities we can conclude that

$$\min\{\boldsymbol{l}^T\boldsymbol{x} \mid \boldsymbol{x} \in Q\} = \boldsymbol{l}^T\boldsymbol{\chi}_M.$$

Thus the ILP (3.5) and LP (3.9) have the same optimal solution(s). □

So solving Problem 3.2 is now reduced to solving the LP in (3.9). To do this, we need to construct the matrix B. However, for a graph on $|V|$ vertices, B would need $O(2^{|V|})$ rows. For large graphs, it is not really feasible to create B and then solve the linear program in reasonable time. So perhaps there is a better way to do this.

One could leave out the constraint $B\boldsymbol{x} \geq \mathbf{1}$, to get the problem of Equation (3.5), but then without the integer constraint.

$$\min\{\boldsymbol{l}^T\boldsymbol{x} \mid A'\boldsymbol{x} \geq \boldsymbol{b}, \boldsymbol{x} \geq 0\} \tag{3.10}$$

Then this can be solved by using linear programming, for example with the simplex algorithm (Chapter 4 of [10]). If the solution is integral, great! If not, check for which odd subset the constraint (3.7) is violated, and add the constraint for that odd subset. Continue this process, until you find an integral solution. This way, only the useful constraints get added, and not all of the $O(2^{|V|})$ at once.

How should one check for which subset the constraint is not satisfied? Idea: find a minimum weight cut $\delta(S)$, such that $|S|$ is odd. To find such a minimum odd cut, we use an algorithm from [12], which will be explained in the next section. Then if the weight of this minimum weight cut is less than 1, then the constraint for set S can be added to the set of constraints. Given a cost function c on the edges, the weight of a cut is

$$\sum_{e \in \delta(S)} c(e).$$

In our case we take $\boldsymbol{x}$, for which the last LP attained its optimal solution, as the cost function. Then the constraint we need to check becomes:

$$\sum_{e \in \delta(S)} x_e \geq 1,$$

for each odd cut $\delta(S)$. In fact it suffices to check it for the minimum weight odd cut that can be found using the algorithm described in [12].

We check this constraint if we have found a non-integral solution. If all odd subsets S satisfied the constraint (3.7), we would have found an integral

solution. That means the constraint is violated for at least one odd subset. In particular, the minimum weight cut will violate the constraint. So, in summary, if we find a non-integral solution, the minimum weight cut will violate the constraint, so we can add the constraint for that cut to the set of constraints.

3.2.1 Finding a Minimum Weight Cut

To compute a minimum weight cut, we use Algorithm 1 from [12]. Before we state the algorithm a few definitions from [12] must be explained. Let $G = (V, E)$ be a graph and let T be a subset of the vertices, such that $|T|$ is even. A set S is called T**-odd**, if $|S \cap T|$ is odd. So if S contains an odd number of elements of T. Otherwise S is called T**-even**. If S is T-odd, $\delta(S)$ is called an **odd** T**-cut**. The **complement** of a set $S \subseteq V$ is denoted as $\bar{S} = V \setminus S$.

Given a T-even set S, denote by G_S the graph obtained from G by identifying all vertices in S as one single vertex, and $T_S = T \setminus S$. When $S = \{s, t\}$, the shorthand notation $G_S = G_{s,t}$, $T_S = T_{s,t}$ is used.

The algorithm computes a minimum weight odd T-cut. In our case, we want an odd cut of all the vertices, so we will take $T = V$. Then an odd V-cut for a V-odd set S, is just a cut for an odd subset S, which is what we need to find a violated constraint. As mentioned in the previous section, we will use the vector $\boldsymbol{x}$, for which the last run of the LP was optimal, as cost function.

Algorithm 2: Min_T-cut(G, T, c)
Input: A graph G, a subset T of vertices and a cost function c.
Output: The cost of a minimum odd T-cut.

1 if $T = \emptyset$ then return ∞; *comment*: G contains no odd T-cut
2 let s and t be any two different nodes in T;
3 let $\delta(S)$ be a minimum $\{s, t\}$-cut;
4 if S is T-odd then return $\min\{c(\delta(S)), \text{Min\_}T\text{-cut}(G_{s,t}, T_{s,t}, c)\}$; else return $\min\{\text{Min\_}T\text{-cut}(G_S, T_S, c), \text{Min\_}T\text{-cut}(G_{\bar{S}}, T_{\bar{S}}, c)\}$.

The goal of the algorithm is to find a minimum odd T-cut. First it considers two vertices, s and t, of T and uses a simpler algorithm (explained later) to find a minimum $\{s, t\}$-cut $\delta(S)$. This set S can be T-odd or T-even.

If S is T-odd, $\delta(S)$ is an odd cut, so that is good. For the minimum odd T-cut, s and t are either on different sides ($s \in S$ and $t \in \bar{S}$,) and $\delta(S)$ is the minimum odd cut, or s and t are on the same side (both in S or both in $\bar{S}$,) and we are looking for a different cut. When looking for this different cut, s and t can be forced to be on the same side by considering the graph $G_{s,t}$, to make sure the set T remains even, we now consider the new set $T_{s,t}$.

Now if S is T-even, then $|S \cap T|$ is even. First, consider some odd T_S-cut

$\delta(S_1)$ in G_S. Then S_1 is T_S odd, which means

$$|S_1 \cap T_S| = |S_1 \cap (T \setminus S)| = |(S_1 \setminus S) \cap T|$$

is odd. If the (contracted) vertex S is in the set S_1 in G_S, then $S_1 \cup S$ is T-odd in G, since $|(S_1 \setminus S) \cap T|$ is odd and $|S \cap T|$ is even. If the contracted vertex S is not in the set S_1 in G_S, then S_1 is T-odd in G, since $S_1 \cap S = \varnothing$, and therefore $|S_1 \cap T| = |S_1 \setminus S \cap T|$, which is odd. So in both cases this odd T_S-cut in G_S corresponds to an odd T-cut in G. The same can be concluded if one considers some odd $T_{\bar{S}}$-cut in $G_{\bar{S}}$. This means that

$$\text{Min\_}T\text{-cut}(G, T, c) \leq \min\{\text{Min\_}T\text{-cut}(G_S, T_S, c), \text{Min\_}T\text{-cut}(G_{\bar{S}}, T_{\bar{S}}, c)\}. \tag{3.11}$$

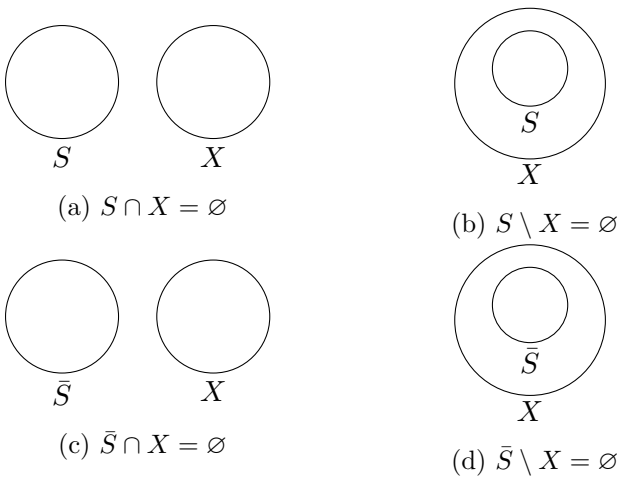
To see the other inequality, we first consider the following definitions from [12]. **Switching** a set S means replacing S by its complement $\bar{S}$. If $S \cap X \neq \varnothing$ for every possible switching of S and X, then S and X are said to **cross**. Equivalently, S and X cross if each of the sets

$$S \cap X,\ S \cap \bar{X} = S \setminus X,\ \bar{S} \cap X,\ \bar{S} \cap \bar{X} = \bar{S} \setminus X, \tag{3.12}$$

is non-empty. Now consider the following lemma (Lemma 2.4 in [12]):

Lemma 3.1. *Let T_1, T_2 be even cardinality subsets of V. Let $\delta(S_1)$ be a minimum odd T_1-cut and assume that S_1 is T_2-even. Then there exists a minimum odd T_2-cut $\delta(S_2)$ such that S_1 and S_2 do not cross.*

Recall that we are considering an $\{s,t\}$-cut $\delta(S)$ such that S is T-even. Let $T_1 = \{s,t\}$ and $T_2 = T$. Then $\delta(S)$ is an odd T_1-cut and S is T_2-even. Then according to Lemma 3.1, there exists a T-odd set X such that $\delta(X)$ is a minimum odd T-cut in G and such that S and X do not cross. That means at least one of the sets in Equation (3.12) must be empty. The following diagram illustrates all these situations in G.



(a) $S \cap X = \varnothing$ (b) $S \setminus X = \varnothing$

(c) $\bar{S} \cap X = \varnothing$ (d) $\bar{S} \setminus X = \varnothing$

Figure 3.8: Possibilities if S and X do not cross.

Consider Figure 3.8a. Because $T_S = T \setminus S$, X is T-odd and $S \cap X = \varnothing$, X is T_S-odd. That means $\delta(X)$ is also an odd T_S-cut in G_S. Now consider Figure 3.8b. Because $T_S = T \setminus S$, X is T-odd and S is T-even, X is T_S-odd. That means that also in this case $\delta(X)$ is an odd T_S-cut in G_S. In a similar way it can be seen that in the Figures 3.8c and 3.8d, $\delta(X)$ is also an odd $T_{\bar{S}}$-cut in $G_{\bar{S}}$. That means that the odd T-cut $\delta(X)$ in G is also either an odd T_S-cut in G_S or an odd $T_{\bar{S}}$-cut in $G_{\bar{S}}$. This means that

$$\text{Min\_}T\text{-cut}(G, T, c) \geq \min\{\text{Min\_}T\text{-cut}(G_S, T_S, c), \text{Min\_}T\text{-cut}(G_{\bar{S}}, T_{\bar{S}}, c)\}. \tag{3.13}$$

By (3.11) and (3.13) we conclude that

$$\text{Min\_}T\text{-cut}(G, T, c) = \min\{\text{Min\_}T\text{-cut}(G_S, T_S, c), \text{Min\_}T\text{-cut}(G_{\bar{S}}, T_{\bar{S}}, c)\},$$

if S is T-even.

In [12] it is not explained how to find a minimum $\{s, t\}$-cut, which is needed in line 3. But one can for example use the Ford-Fulkerson algorithm (Section 4.3 of [14]) for finding max-flow/min-cut. However this algorithm works with directed graphs, and we have an undirected graph. You could replace each edge $\{u, v\}$ by the arcs (u, v) and (v, u). The capacity is our weight function. The new arcs both get the capacity of the original edge. However, when there is a flow through an arc, the capacity of the corresponding reverse arc becomes the original capacity *plus* the value of the flow in the other direction.

As an example: if the capacity of an edge $\{u, v\}$ is x, then both arcs get capacity x. Now if there is a flow of value y through (u, v), then the capacity of (v, u) changes to $x + y$. This makes sense, because there still can go x through this arc, but now also an extra y to 'cancel' the flow in the reverse direction.

I used Wolfram Mathematica [6] for programming. I found that when looking for maximum flow, one can use the function FindMaximumFlow. In the details of that function, the following is stated:

For undirected graphs, edges are taken to have flows in both directions at the same time and same capacities.

So Mathematica interprets a flow in an undirected graph the same as is mentioned above.

3.3 Finding an Euler Tour

Now that one can make a graph Eulerian, it remains to find an Euler tour in a Eulerian multigraph.

Start at some vertex s and start following unused edges, until you come back at s. Note that you will always come back to s because all vertices have even degree. So if you enter any other vertex while following unused edges, this vertex must have at least one unused edge left, which you can use to leave the vertex.

Doing this creates a closed walk W that consists of all edges at most once, because you only follow unused edges. If could happen that not all edges are in W yet. Consider an unused edge, and suppose that it is not incident with a vertex in W. Then this edge is incident with two vertices u and u' which are not in W. Because the graph is connected, there is a path of unused edges between u and a vertex v in W. That means that v is the endpoint of an unused edge. So as long as there are unused edges, there is some unused edge incident with a vertex in W. Because each edge in W contributes an even number to the degree of each vertex and each vertex has even total degree, this vertex v must be the endpoint of an even number of unused edges, so at least two unused edges. Thus there is an edge you can use to start a new walk leaving from v, starting this new walk, you will eventually come back to v via some other unused edge. Note that this can be reasoned in a same way as for the vertex s in which we started.

Then the new closed walk W' is created from W by following W until you arrive at vertex v, then follow the new part from v back to v. When back at v you can proceed with W. Keep modifying the tour in this way until there are no unused edges left.

This is also known as Hierholzer's algorithm. In 1873 he wrote a paper about this [5].

Note that, if one uses the notation that each edge gets a value of how many times it needs to be traversed, instead of that the edges get doubled, this algorithm also works. Only then 'unused' should be replaced by 'has a value greater or equal to 1'. And if one traverse an edge, instead that is goes from unused to used, one should decrease its value by 1.

Example 3.4. Consider the following graph.

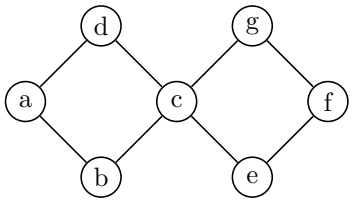


Figure 3.9: graph

Start at vertex a. Then you can for example find the closed walk (a, b, c, d, a). Then vertex c still has unused edges, so start there, then you can find the

closed walk (c, e, f, g, c). Adding these two walks together gives the walk $(a, b, c, e, f, g, c, d, a)$. Now all edges have been used, so we have found our Euler tour.

Chapter 4

The Directed Problem

The problem considered in this chapter is the following.

Problem 4.1. Given a strongly connected, directed graph $D = (V, A)$ with length function $l : A \to \mathbb{R}_+$, find a minimum length postman tour.

Again, if D is Eulerian the problem is easy. But what if D is not Eulerian?

Theorem 4.1. *Let D be a strongly connected, directed (multi)graph. Then D is Eulerian if and only if every vertex in D is balanced.*

Proof. "$\Rightarrow$" D is Eulerian, so there is a Euler tour. In the same way as in the proof for undirected graphs, one can argue that every time the tour passes a vertex it uses one in-going arc and one out-going arc. And again the start and endpoint are the same, thus for all vertices the same number of in-going arcs and out-going arcs are traversed. Also all arcs are traversed, because it is an Euler tour. So all vertices are incident with as many in-going arcs as out-going arcs, which means all vertices are balanced.

"$\Leftarrow$" Let there be some longest walk W that traverses all arcs at most once, but is not a Euler tour. If W is not closed, we have used one more in-going arc than out-going arc from the vertex in which the walk ends. But the vertex is balanced, so we can add an out-going arc that has not been traversed yet, which makes the walk longer.

So then W must be a closed walk that is not a Euler tour. W is a closed walk, so uses, from each vertex the same number of in- and out-going arcs. Analogous to the proof of Theorem 3.1: because the graph is strongly connected, not all arcs are used in W and all vertices are balanced, there must be at least one vertex v in W that has two unused arcs: one going out and one going in. Also analogous to the proof of Theorem 3.1, one can argue that one can use these arcs to visit vertices using arcs not in W, until one is back in v. Then merging this new part with W, makes the walk longer.

This contradicts the fact that W already was a longest walk. Therefore D must be Eulerian. □

So in this case, if G is not Eulerian, it must mean that not all vertices are balanced. Remember, vertices are balanced if $\delta(v) = d^+(v) - d^-(v) = 0$. Define

$$T^+ := \{v \in V \mid \delta(v) > 0\},$$
$$T^- := \{v \in V \mid \delta(v) < 0\}.$$

All vertices in T^+ have more edges leaving than entering and T^- has more edges entering than leaving. So to make this situation balanced again we will add arcs to the graph by looking for minimum length directed paths from vertices in T^- to vertices in T^+. This translates to a weighted complete bipartite graph $B = (T^- \cup T^+, E)$. Each edge represents a shortest, directed $T^- - T^+$ path. The weight of edge $e = (u, v)$, where $u \in T^-$ and $v \in T^+$, represents the length of a shortest directed path from u to v.

In the undirected problem we started looking for a minimum length perfect matching, but the situation is a bit different here. The difference between the in- and outdegree of a vertex can be greater than one, but we want to have it equal to zero. To achieve this, one wants that a vertex v is incident with exactly $|\delta(v)|$ matching edges. Thus we will be looking for a minimum length perfect b-matching, with $b(v) = |\delta(v)|$.

From the Handshaking Dilemma, see Lemma 2.2, we can deduce the following for graph $B = (T^- \cup T^+, E)$:

$$\sum_{v \in T^- \cup T^+} d^+(v) = \sum_{v \in T^- \cup T^+} d^-(v).$$

Expanding the sum and bringing the sums over T^+ both to the left and the sums over T^- both to the right, gives

$$\sum_{v \in T^+} d^+(v) - \sum_{v \in T^+} d^-(v) = \sum_{v \in T^-} d^-(v) - \sum_{v \in T^-} d^+(v).$$

Taking the sums on one side together, by using $d^+(v) - d^-(v) = \delta(v)$, gives

$$\sum_{v \in T^+} \delta(v) = -\sum_{v \in T^-} \delta(v).$$

In other words, T^+ has as many 'spare' arcs leaving as T^- has entering, and in particular

$$\sum_{v \in T^+} b(v) = \sum_{v \in T^+} |\delta(v)| = \sum_{v \in T^-} |\delta(v)| = \sum_{v \in T^-} b(v). \tag{4.1}$$

Theorem 4.2. *Given a complete bipartite graph $B = (U \cup V, E)$ and a function $b : V \cup U \to \mathbb{Z}_+$. There exists a perfect b-matching in B if and only if*

$$\sum_{u \in U} b(u) = \sum_{v \in V} b(v).$$

Proof. "⇒" Consider a perfect b-matching x. In bipartite graphs each edge is incident with a vertex in U and a vertex in V. So also each matching edge is incident with a vertex in U and a vertex in V. That means if we sum over all matching edges incident with some vertex $u \in U$, then we count all matching edges. In other words:

$$\sum_{u \in U} \sum_{e \ni u} x(e) = \sum_{e \in E} x(e).$$

Note that on the right hand side, we sum over all $e \in E$, because if e is not in the matching, $x(e) = 0$ anyway. That also holds if you sum over all matching edges incident with some vertex $v \in V$:

$$\sum_{v \in V} \sum_{e \ni v} x(e) = \sum_{e \in E} x(e).$$

Taking the above two together results in:

$$\sum_{u \in U} \sum_{e \ni u} x(e) = \sum_{v \in V} \sum_{e \ni v} x(e). \tag{4.2}$$

We are considering a perfect b-matching, so for each $v \in U \cup V$:

$$\sum_{e \ni v} x(e) = b(v).$$

That means

$$\sum_{u \in U} \sum_{e \ni u} x(e) = \sum_{u \in U} b(u),$$

and

$$\sum_{v \in V} \sum_{e \ni v} x(e) = \sum_{v \in V} b(v).$$

Substituting the above two expressions in Equation (4.2) gives what we wanted to proof:

$$\sum_{u \in U} b(u) = \sum_{v \in V} b(v).$$

"⇐" Let us assume that we have a maximum cardinality matching x in our graph B that is not a perfect b-matching. That must mean there is at least one vertex that is not maximally covered by x. Because

$$\sum_{u \in U} b(u) = \sum_{v \in V} b(v),$$

there must be at least one vertex u in U and at least one vertex v in V that are both not maximally covered, that means

$$\sum_{e \ni v} x(e) < b(v), \text{ for both } u \text{ and } v.$$

Since x and b are both integer valued functions, the equation above implies:

$$\sum_{e \ni v} x(e) \leq b(v) - 1, \text{ for both } u \text{ and } v.$$

The bipartite graph B is complete, so there is an edge $e = \{u, v\}$. Because of the equation above, we know the value of $x(e)$, for this particular edge e, can be increased by one. Which contradicts that x is of maximum cardinality. So our assumption that x is not a perfect b-matching must be wrong. □

In our graph B we are going to be looking for a minimum length perfect b-matching. And because we have derived (4.1), the theorem above tells us that a perfect b-matching exists!

Example 4.1. It is again assumed all edges have weight 1.

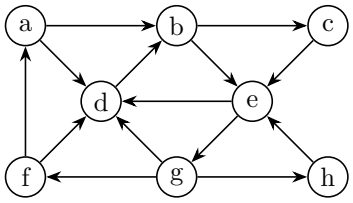


Figure 4.1: directed graph

From the graph one can see that $T^+ = \{a, f, g\}$ and $T^- = \{d, e\}$. To draw the bipartite graph one needs to calculate the values of δ for these vertices: $\delta(a) = 1$, $\delta(f) = 1$, $\delta(g) = 2$, $\delta(d) = -3$ and $\delta(e) = -1$.

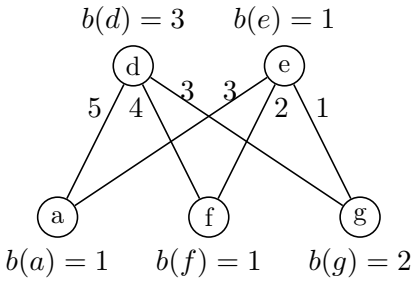


Figure 4.2: complete bipartite graph

In the next section Dijkstra's algorithm is executed on this graph to find shortest paths from vertex d. From that one can derive the lengths of the edges incident with d. The lengths of the edges incident with e can be easily seen in the original graph. For example, the length of the edge $\{e, a\}$ is 3 and the edge represents the directed path (e, g, f, a).

Vertex e should be incident with only one edge in the matching, so if you choose that edge, the edges incident with d must go to the remaining

vertices. That means there are only three possible matchings and it is easily verified that all these matchings have a total length of 13. Therefore all are of minimum length, so one can just choose one of the three. Take for example the matching with the following edges: $\{d, a\}$, $\{d, f\}$, $\{d, g\}$ and $\{e, g\}$. Adding the corresponding paths to the original graph, creates the following new graph:

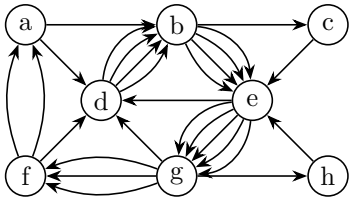


Figure 4.3: digraph with added arcs

Which indeed has 13 edges more than the original graph. As in the undirected case, this new graph can be translated back to the original graph, by giving each arc a value, which represents how many times it must be traversed.

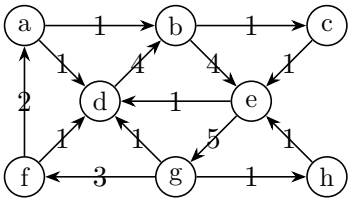


Figure 4.4: digraph with number of times arcs need to be traversed

In Section 4.3 it is demonstrated how to find an Euler tour in this graph, with as result: $(a, b, c, e, g, h, e, g, f, a, d, b, e, g, f, d, b, e, d, b, e, g, d, b, e, g, f, a)$.

Note that in contrast with the undirected case, where all edges are added at most once, in this case arcs can be added twice, or even more often. See the figure below, where arc e is needed at least twice.

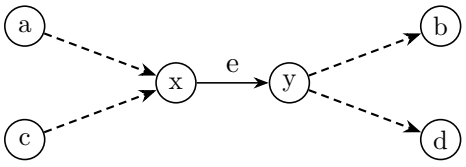


Figure 4.5: directed paths both containing the same arc e

Leaving out arc e is not possible, because the paths $(a, \ldots, x)$ and $(c, \ldots, x)$ are directed walks, and can neither be combined into the directed walk

$(a, \ldots, x, \ldots, c)$, nor into to directed walk $(c, \ldots, x, \ldots, a)$.

However, in the following situation it is possible to leave arcs out.

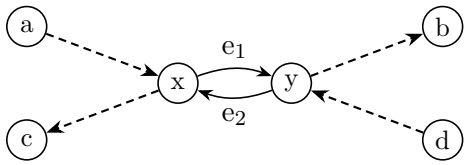


Figure 4.6: arcs in both directions between x and y

Now it is possible to leave out both arc e_1 and e_2. Because then the directed walks $(a, \ldots, b)$ and $(d, \ldots, c)$ can be changed to the directed walks $(a, \ldots, x, \ldots, c)$ and $(d, \ldots, y, \ldots, b)$.

4.1 Finding a Shortest Directed $s - t$ Path

Again Dijkstra's algorithm (1) can be used. This time it needs to be executed on all vertices in T^-, to find directed shortest paths to the vertices in T^+. So one chooses a start vertex from T^- and chooses $T = T^+$.

Example 4.2. Consider the graph of Figure 4.1, with $T^+ = \{a, f, g\}$ and $T^- = \{d, e\}$. Take $d \in T^-$ as start vertex s and take $T = T^+$.

iteration	a	b	c	d	e	f	g	h	predecessor
0	∞	∞	∞	**0**	∞	∞	∞	∞	
1	∞	**1**	∞		∞	∞	∞	∞	p(b) = d
2	∞		**2**		2	∞	∞	∞	p(c) = b, p(e) = b
3	∞				**2**	∞	∞	∞	
4	∞					∞	**3**	∞	p(g) = e
5	∞					**4**		4	p(f) = g, p(h) = g
6	5							**4**	p(a) = f
7	**5**								

Table 4.1: execution of Dijkstra's algorithm

Looking at the predecessors we find the following shortest path tree.

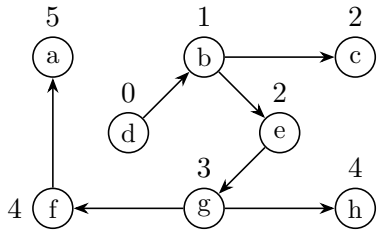


Figure 4.7: directed shortest path tree

The length from d to a particular vertex is displayed above that vertex. The shortest path can be found by following the edges. For example, the shortest $d-a$ path is (d, b, e, g, f, a) and has length 5.

4.2 Finding a minimum Length Perfect b-Matching in a Bipartite Graph

The problem under consideration in this section is the following.

Problem 4.2. Given a bipartite graph $B = (V \cup W, E)$, with given length function $l : E \to \mathbb{R}_+$, find a minimum length perfect b-matching for a given function $b : V \cup W \to \mathbb{Z}_+$.

Let x be a b-matching in B. We want to have a perfect b-matching, so we want each vertex v to be maximally covered. In other words,

$$\sum_{e \ni v} x(e) = b(v), \text{ for all } v \in V. \tag{4.3}$$

Define A as the $(V \cup W) \times E$ incidence matrix of the graph B. Then the elements of A are as defined in Equation (2.1):

$$a_{ij} = \begin{cases} 1 & \text{if edge } j \text{ is incident with vertex } i, \\ 0 & \text{otherwise.} \end{cases} \tag{2.1}$$

Define $\boldsymbol{\chi}$ as the vector of $x(e)$'s and $\boldsymbol{b}$ as the vector of $b(v)$'s. Then Equation (4.3) can be rewritten as $A\boldsymbol{\chi} = \boldsymbol{b}$.

The given length function l assigns a length $l(e)$ to each edge e. Then the total length of the matching is

$$\sum_{e \in M} l(e) = \sum_{e} l(e)x(e).$$

Define $\boldsymbol{l}$ as the vector of $l(e)$'s, then this can be rewritten as $\boldsymbol{l}^T\boldsymbol{\chi}$.

Analogous to the undirected case we want to find a minimum length perfect b-matching, which is equivalent to minimising $\boldsymbol{l}^T\boldsymbol{x}$, over all $\boldsymbol{x} \in \mathbb{Z}_+^E$, with the constraint that $\boldsymbol{x}$ is the incidence vector of a perfect b-matching. So one would like to solve the integer linear program:

$$\min\{\boldsymbol{l}^T\boldsymbol{x} \mid A\boldsymbol{x} = \boldsymbol{b}, \boldsymbol{x} \geq 0, \boldsymbol{x} \in \mathbb{Z}^E\}. \tag{4.4}$$

Consider the following matrix and vector,

$$A' = \begin{pmatrix} A \\ -A \end{pmatrix}, \boldsymbol{b}' = \begin{pmatrix} \boldsymbol{b} \\ -\boldsymbol{b} \end{pmatrix}.$$

Then the equation $A'\boldsymbol{x} \geq \boldsymbol{b}'$ is equivalent to

$$A\boldsymbol{x} \geq \boldsymbol{b} \wedge -A\boldsymbol{x} \geq -\boldsymbol{b},$$

or equivalently:

$$A\boldsymbol{x} = \boldsymbol{b}.$$

Then the problem in Equation (4.4) can also be written as

$$\min\{\boldsymbol{l}^T\boldsymbol{x} \mid A'\boldsymbol{x} \geq \boldsymbol{b}', \boldsymbol{x} \geq 0, \boldsymbol{x} \in \mathbb{Z}^E\}. \tag{4.5}$$

In Equation (4.4) the matrix A is the incidence matrix of a bipartite graph. So by Theorem 2.1 this matrix is TU. Then by Theorem 2.2 the matrix A' in Equation (4.5) is also TU. The vector $\boldsymbol{b}$ is integral, then also the vector $\boldsymbol{b}'$, is integral. By (4.3) we know that the sum over all edges incident with some vertex v, is $b(v)$. Which means that each edge incident with v can have a value of at most $b(v)$. Each edge is incident with two vertices, say $e = \{u, v\}$, that means $x_e \leq \min\{b(u), b(v)\}$. This means that in the polyhedron

$$P = \{\boldsymbol{x} \mid A'\boldsymbol{x} \geq \boldsymbol{b}', \boldsymbol{x} \geq 0\},$$

each x_e is within the range

$$0 \leq x_e \leq \min\{b(u), b(v)\} \leq \max_{v \in V}\{b(v)\}.$$

Therefore P is a bounded polyhedron.

So then, by Theorem 2.8, the problem in Equation (4.5) attains an integral optimal solution in a vertex, also without the constraint $\boldsymbol{x} \in \mathbb{Z}^E$. So we can rewrite it as

$$\min\{\boldsymbol{l}^T\boldsymbol{x} \mid A'\boldsymbol{x} \geq \boldsymbol{b}', \boldsymbol{x} \geq 0\}. \tag{4.6}$$

So now we have changed to original integer linear program in Equation (4.4), to a normal linear program in Equation (4.6).

In conclusion, the problem of finding a minimum length perfect b-matching can be solved by using linear programming on Equation (4.6). When using a method (like the simplex algorithm, see Chapter 4 of [10]) that looks for an optimum *vertex* of the polyhedron, this results in an integral optimal solution vector $\boldsymbol{x}$. From this vector it can be seen how often an edge is in the b-matching.

4.3 Finding an Euler Tour

To find a Euler tour in a directed Eulerian multigraph, one can also use Hierholzer's algorithm, see Section 3.3. Only in this case it works because all vertices are balanced and the graph is strongly connected. Here we will show an example in which the edges have values.

Example 4.3. We will demonstrate this method on the graph in Figure 4.4, by starting at vertex a. We can then for example find the walk

$(a, b, c, e, g, h, e, g, f, a)$. Now we have to decrease the values of the edges, each time they are passed, by 1. That results in the following graph.

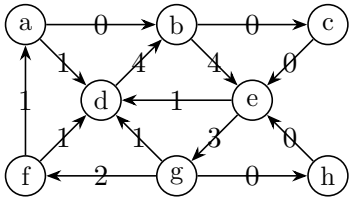


Figure 4.8: directed graph with decreased values

It can be seen that we can again start in vertex a, to for example get: (a, d, b, e, g, f, a). We can merge this with the first walk, by replacing the last a by this walk: $(a, b, c, e, g, h, e, g, f, a, d, b, e, g, f, a)$. Then we are left with the following graph.

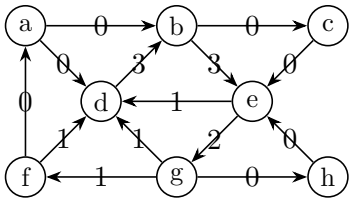


Figure 4.9: directed graph with decreased values

Now we cannot start in a anymore. But we can for example start in f to find $(f, d, b, e, d, b, e, g, d, b, e, g, f)$. And then all values are decreased to zero, so we are finished. We can merge this walk with the previous walk, by replacing the last f by this walk, to get the Euler tour: $(a, b, c, e, g, h, e, g, f, a, d, b, e, g, f, d, b, e, d, b, e, g, d, b, e, g, f, a)$.

Chapter 5

The Mixed Problem

To my regret I did not have enough time to study this problem in depth. I have however read through some literature. So I will state the problem, and state some literature and ideas that might be helpful, for perhaps further research.

Problem 5.1. Given a strongly connected, mixed graph $G = (V, E \cup A)$ with length function $l : E \cup A \to \mathbb{R}_+$, find a minimum length postman tour.

From Theorems 3.1 and 4.1, we derive that a mixed graph is Eulerian if each vertex is balanced.

Firstly, some quotes from articles that I have read. In [3] the following theorem is stated.

There exists a postman tour if and only if there is no proper subset S of nodes such that every edge meeting one node in S and one node not in S is directed away from the node in S.

In [9] the following lemma is stated.

In any optimum postman route, if an undirected arc (x, y) is traversed in both directions, then arc (x, y) is traversed exactly two times.

And it states that from this lemma, the following follows.

The postman problem for mixed networks is solved by deciding for each undirected arc (x, y) whether it will be crossed (a) only from x to y, (b) only from y to x, or (c) exactly once in each direction.

However, it is not explained how the decision for each undirected arc should be made.

Next an idea from [4]: replace all undirected edges $e = \{i, j\}$ and their value x_e by arcs $a_1 = (i, j)$ and $a_2 = (j, i)$ with values x_{ij} and x_{ji}, respectively. Then instead of that the one undirected edge must be visited at least

once, so $x_e \geq 1$, we now have that at least one of the arcs must be visited at least once, so $x_{ij} + x_{ji} \geq 1$, where $x_{ij}, x_{ji} \geq 0$. Then it can be solved as the directed case. When a tour is found, the arcs should be translated back to their original edge and the values x_{ij} and x_{ji} should be added to know how many times edge e is in the tour.

For further research into the mixed Chinese postman problem I recommend to consider the sources mentioned in this chapter. I think in particular [4] is a good source.

Chapter 6

De-icing the Streets of Eindhoven

During the winter period, icy and snowy roads are a well known problem. The roads get very slippery, and it becomes dangerous to drive on them. Therefore measures are taken. Snow-shovel trucks and gritters are sent out to clean and de-ice the roads of all the snow and ice.

So certain roads need to be traversed, at least once, in order to be cleaned. It is preferred that the total route has minimum cost. Because there are one-way and two-way streets, this sounds a lot like the mixed Chinese postman problem. The first difference we note is that de-icing is mostly done per city, but cities can be too big for just one truck. So that means there need to be several trucks, that all drive a part of the total. That means, several postman tours, together covering all the roads.

6.1 Information

To get more information on this subject, I visited the city hall of Eindhoven to talk to Ruud Jacobs [7]. He is responsible for the routes for the snow-shovel trucks and gritters in the city of Eindhoven. When I was there I learned there are a lot of things that need to be taken into account.

- Streets can be one-way or two-way.
- Gritters can spread salt in a width of 2 to 12 meters; gritters for bicycle lanes can spread salt in a width of 2 to 6 meters.
- Snow-shovel trucks come in many different sizes, but in Eindhoven they use a truck that can work in a width of 1.8 meters for bicycle lanes, and 1.5 meters for small bicycle lanes; for normal roads it depends on the width of the road, but there are trucks with a working width between 2.1 and 3 meters.

- Multi-lane roads may need to be traversed more than once, to cover their whole width.
- The number one rule is that straight roads need to be cleaned in one go, if you come at a crossing, you first must go straight. This is because drivers do not expect to go from a de-iced road to a still icy road, when going straight, so if it would still be icy, that can be dangerous.
- There is continuously work being done on roads in the city, for construction or for extension of the road network, so routes must be adaptive.
- There is one central depot for salt, where all trucks need to go at least once on their round. At the depot there is circa 2000 tons dry road salts (NaCl), and circa 90 m^3 brine (NaCl solution 22%) stored.
- Gritters are allowed to go 60 km/h, and snow-shovel trucks are allowed to go 30 km/h. On bicycle lanes, gritters can go 30 km/h, just like other road users. Snow shovel trucks can go 10-15 km/h.

As you might notice, trucks do not have to start at the depot, but it is mandatory that they visit it at least once during their round. This is because truckers are allowed to take their truck, with salt in it, home. But they do need to come by the depot, to get extra salt and to report that they are working. This means that when making the routes you can take the depot as central starting and endpoint for all routes. But in practice, truckers can start on any point of their route. For example on the point that is closest to their home.

I also learned which are some basic roads that need to be cleaned.

- Roads that are used by emergency service, like ambulance, fire brigade and police. They must be able to reach their destination in a legally determined time interval.
- Access roads to social services, like hospitals, police headquarters, fire station, care homes, tax/custom office, etcetera.
- Roads that are used by public transport buses.
- The big and widely used roads. These are mainly the approach and neighbourhood access roads.
- The access roads to the city.
- Bicycle lanes. Often there are smaller vehicles for those, because bicycle lanes are smaller than normal roads, and because they have a lower carrying capacity.

- Industrial areas (so people can get to work).
- Roads around high schools. Roads around primary school are only taken along when there is snow.

So not all roads need to be cleaned. For example in a neighbourhood with a lot of homes, and small streets with speed breakers. These are difficult roads to de-ice, and those roads are not that much used, so they are often not de-iced. Primary schools often lay in such neighbourhoods, therefore they are only taken along when there is snow.

There is a difference between icy roads and snow on the roads. Those cases are handled differently. If there is snow on the roads more roads get cleaned. For example, roads around retirement homes usually do not get cleaned, but when it snows also those roads get cleaned.

There is also a difference between different pavements of the roads. Like baked paving blocks become slippery faster than concrete paving blocks, but baked paving blocks also dry faster. On asphalt roads there are no joints were the water can go, therefore asphalt roads dry the slowest. A road can only get slippery if there is water and if the temperature of the pavement (so not the air temperature) is 0 degrees, or lower.

6.2 Current Situation Eindhoven

In Eindhoven they work with a 'microstation' where they have a map of the city, in which they can draw the routes by hand. They also have another program, which everyone at city hall has access to, in which an overview of the routes can be seen. You can choose there to see a map or a picture of the city, with the routes on top. Those pictures are taken once a year and are useful to keep track of new roads, and for example to see how wide a road is, or how many lanes it has.

So in this program the standard routes are made, but when the routes are actually needed there are a few things that the coordinator keeps in mind.

- Which day is it? Is it during the week, or during the weekend?
- Which time of the day is it? Is it at night, and thus very quiet on the streets? Is it during rush hour? Or can it be finished before rush hour begins? Which time of the day it is, is also important for the temperature of the road. During the morning the temperature of the road surface increases, because of the sunlight. But at the end of the afternoon, the road surface starts to cool down again, when the sun sets.
- Are there events? For example, it is preferred not to go cleaning the roads when a soccer game at the PSV stadium is just finished, because

then the roads are full of traffic. Also at Saturday night from midnight to four in the morning, the roads around 'Stratumseind' are avoided.

These things have an influence on how many trucks will be put to work. If it is in the middle of the night and the morning rush hour is far away, not many trucks are needed and they can take their time. But if rush hour is almost there, many trucks are used, with the hope that the streets can be done before rush hour starts.

There is a live GPS connection with all the spreaders that are attached with the trucks. This is can be used to see if they are already at work, and driving the correct route. It can also be used to see a lot of data, for example: how fast they drive, where they drive, where and how much and how wide they spread salt.

Some roads are tough to do and require that they are done several times. An example of such a road in Eindhoven is the John F Kennedylaan. That is a big road, that has multiple lanes going in both directions. At the moment, if it has snowed, there are multiple trucks (3 to 4) going over all the lanes at the same time, to clean the whole road at once. But then you also have the driveways and exits, they need to be done separately, because the whole road needs to be done at once, because of the number one rule. Then there are also bicycle lanes on both sides of the road. There is one smaller truck that can do this.

At last there are the mopeds (brommers in Dutch), they must drive on the bicycle lane, but when approaching a crossing, they need to go on the normal road, and after a crossing they need to go back to the bicycle lane. So for that there are special crossings that go from bicycle lane to road and from road to bicycle lane. There is one smaller truck that drives the whole John F Kennedylaan for only these few small crossings.

Gritters do not have to spread salt continuously. So if they go over a road that already has salt on it, or over a road that does not have to be de-iced, than they can turn off the spreader. Therefore in optimising routes, the amount of used salt is not important, that is always the same. However what can be optimised is the total distance, because more distance costs more fuel. Or the total time, because time is money.

When I was talking with Ruud, he mentioned that he is especially interested in the balance between costs and how many vehicles he puts to work for one cleaning action. More vehicles means more workers that have to be payed, but more vehicles also means the cleaning action can be done in less time.

6.3 Towards a Model for Eindhoven

As was mentioned in the introduction of this chapter, the problem of routing snow shovel trucks and gritters sounds a lot like the mixed Chinese postman problem, only the problem asks for several postman tours, instead of one.

My idea was to model the problem as a mixed graph. So crossings are the vertices, and roads are the edges. Because it can happen that roads need to be visited more than once, my idea was to give a value to the roads. This value then indicates how many times these roads must at least be visited. Roads that have separate lanes for opposite directions will get directed edges in both directions. Also roads that have no separate lanes, but are too wide to clean at once, are preferred to be visited once in each direction. Because if the truck drives it twice in the same direction, it once has to drive on the wrong side of the road. So also those roads get arcs in each direction.

When only considering the roads that need to be cleaned, this can result in a disconnected graph. To ensure that we have a strongly connected graph, all roads are added to the graph. But if they do not necessarily have to be visited, they get a value of 0.

So now the problem differs in two ways from the mixed Chinese postman problem: (1) the problem asks for several postman tours, and (2) the edges do not have to traversed at least once, but they have to be traversed at least some assigned nonnegative value.

In [11], the problem of vehicle routing for snow plowing is discussed. It is refered to as '*multiple hierarchical Chinese Postman Problem*' (m-HCPP). The multiple stands for more than one postman tour, similar to our addition (1) above. Our addition (2) they solve by adding extra copies of the edges, instead of giving them a value. They also included the word 'hierarchical'. Hierarchical means there are classes of roads that are of different importance. Like main roads are the most important, and therefore they are in the highest class. Those roads need to be cleaned first. Next you have, for example, the connecting roads. And at last you could have the residential roads. However, class upgrading is allowed. That means that a road of lower class can be cleaned before all roads of higher classes are cleaned. This can for example be beneficial, if one needs to go over the road anyway to go from one road of higher class to another road of higher class. If one goes over the road anyway, it is better to also clean the road then.

This problem I also discussed with Ruud. If a truck comes over a road that has to be cleaned by another truck, but that is not yet done, then that truck can immediately clean that road. It could for example happen that the truck which was supposed to clean that road, just had some delay in his route, but it could also happen that the truck ran into some problems, and is now unable to go clean that road. Anyway, the truck that drives over the road and sees it is not yet cleaned, does not know what the reason is, so also

does not know if the truck that has to do it will still come or not, therefore that truck can clean that road.

In [11], vehicle road segment dependencies are taken into account. That means not all vehicles are allowed to go over all roads. Like we mentioned before, there are special vehicles for the bicycle lanes, they can go only over the bicycle lanes. And the normal vehicles for the roads, cannot go over the bicycle lanes. The paper also considers different costs for roads, when cleaning and when not cleaning. This is because there can be different restrictions on the speed when cleaning and when not cleaning. As optimisation objective they consider the service completion time, which means the time which it takes for all roads to be cleaned. And thus not the time it takes, after all roads are cleaned, to drive all trucks back to the depot.

This problem is a NP-hard problem, that means there is no efficient algorithm known to solve the problem optimally. In [11], some heuristics to solve the problem are proposed. The results of that approach shows that this approach gives better results than the current routes of the city, of which they had data. Solving the problem using this approach did take a few hours computing time, but because it only needs to be computed once before every winter season, they found it a satisfactory computing time. However, as was mentioned before, there is continuously work being done on the roads in Eindhoven, also during the winter. This means it might be necessary to change the route during the winter period. In [11] this is shortly mentioned:

> *A faster solution approach would, however, be required so that the model can be used in real time to determine the changes to be made following an equipment breakdown or weather change.*

But still the solution approach of [11] is quite good. If the approach has found a solution, it might be not so hard to adapt that solution by hand when there is work being done on a road. This adaptation does not have to be optimal, one should just take into account that road work takes time.

For further research into the application of routing snow shovel trucks and gritters in Eindhoven, I still recommend to consider [11] and compare it to the situation in Eindhoven.

Bibliography

[1] Jack Edmonds. Maximum matching and a polyhedron with 0,1-vertices. *J. Res. Natl. Bur. Stand., Sect. B*, 69:125–130, 1965.

[2] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[3] Jack Edmonds and Ellis L. Johnson. Matching, Euler tours and the Chinese postman. *Mathematical Programming*, 5:88–124, 1973.

[4] Maximilian P.L. Haslbeck. Algorithms for the mixed Chinese postman problem. Final report for an interdisciplinary project, Technische Universität München, 2015.

[5] Carl Hierholzer and Chr. Wiener. Ueber die möglichkeit, einen linienzug ohne wiederholung und ohne unterbrechung zu umfahren. *Mathematische Annalen*, 6(1):30–32, March 1873.

[6] Wolfram Research, Inc. Mathematica, Version 12.0. Champaign, IL, 2019.

[7] Ruud Jacobs. Information about de-icing and snow-shoveling. Personal communication, City Hall Eindhoven, contacted in 2019.

[8] Mei-ko M Kwan. Graphic programming using odd or even points. *Acta Math. Sinica*, 10:263–266, 1960.

[9] Edward Minieka. The Chinese postman problem for mixed networks. *Management Science*, 25(7):643–648, July 1979.

[10] Rudi Pendavingh. Linear optimization. Lecture notes, Eindhoven University of Technology, 2017.

[11] Nathalie Perrier and Andre Langevin. Vehicle routing for urban snow plowing operations. *Transportation Science*, 42(1):44–56, February 2008.

[12] Romeo Rizzi. A simple minimum T-cut algorithm. *Discrete Applied Mathematics*, 129:539–544, 2003.

[13] Alexander Schrijver. *Combinatorial Optimization, Polyhedra and Efficiency*, volume A. Springer, 2003.

[14] Alexander Schrijver. A course in combinatorial optimization. Lecture notes, University of Amsterdam, 2017.

Appendices

Appendix A

Code for Undirected Case

When I was using Mathematica I experienced some problems using the function VertexContract, therefore I have written my own vertex contract function.

```
VertexContractOwn[G_,V_]:=(
(*Input: graph G and set of vertices V that need to be contracted.*)
(*Output: graph g where the set of vertices V is contracted into a single vertex. This
      single vertex is the first vertex given in the set V.*)
  If[Length[V]==0 || Length[V]==1,Return[G]];
  g=G;
(*The vertices will be contracted one by one. So first V[[1]] and V[[2]] into V[[1]],
then V[[1]] and V[[3]] into V[[1]], and so on, until all vertices of V are contracted
      into vertex V[[1]].*)
  For[i=2,i<=Length[V],i++,
    Ed=EdgeList[g];
    gbefore=g;
    g=VertexDelete[g,V[[i]]];
(*Vertex V[[i]] and all edges incident with V[[i]] are deleted from the graph.
Consider some edge V[[i]]-a, if a is not V[[1]] then the edge V[[1]]-a needs to be
      added to the graph, with the weight of edge V[[i]]-a. If V[[1]]-a was already an
      edge in the graph, the weight of V[[i]]-a gets added to the weight edge V[[1]]-a.
There is a difference between V[[i]]-a and a-V[[i]], so it is checked which is the
      case.
Mathematica prefers that for an edge a-b, a<b.*)
    For[j=1,j<=Length[Ed],j++,
      edge=Ed[[j]];
      If[MemberQ[edge,V[[i]]],
        weight=PropertyValue[{gbefore,edge},EdgeWeight];
        If[edge[[1]]==V[[i]],
          If[edge[[2]]!=V[[1]],
            If[MemberQ[Ed,edge[[2]] \[UndirectedEdge] V[[1]]],
              weight0=PropertyValue[{gbefore,edge[[2]] \[UndirectedEdge] V[[1]]},
                  EdgeWeight];
              PropertyValue[{g,edge[[2]] \[UndirectedEdge] V[[1]]}, EdgeWeight]=
                  weight0+weight,
              If[MemberQ[Ed,V[[1]] \[UndirectedEdge] edge[[2]]],
                weight0=PropertyValue[{gbefore,V[[1]] \[UndirectedEdge] edge[[2]]},
                    EdgeWeight];
```

```
27                  PropertyValue[{g,V[[1]] \[UndirectedEdge] edge [[2]]}, EdgeWeight]=
                        weightO+weight,
28                  If[edge[[2]]<V[[1]],
29                    g=EdgeAdd[g,edge[[2]] \[UndirectedEdge] V[[1]]];
30                    PropertyValue[{g,edge[[2]] \[UndirectedEdge] V[[1]]}, EdgeWeight]=
                          weight,
31                    g=EdgeAdd[g,V[[1]] \[UndirectedEdge] edge[[2]]];
32                    PropertyValue[{g,V[[1]] \[UndirectedEdge] edge [[2]]}, EdgeWeight]=
                          weight
33                  ];
34
35                ]
36              ]
37            ],
38            If[edge[[1]]!=V[[1]],
39              If[MemberQ[Ed,edge[[1]] \[UndirectedEdge] V[[1]]],
40                weightO=PropertyValue[{gbefore,edge[[1]] \[UndirectedEdge] V[[1]]},
                      EdgeWeight];
41                PropertyValue[{g,edge[[1]] \[UndirectedEdge] V[[1]]}, EdgeWeight]=
                      weightO+weight,
42                If[MemberQ[Ed,V[[1]] \[UndirectedEdge] edge[[1]]],
43                  weightO=PropertyValue[{gbefore,V[[1]] \[UndirectedEdge] edge [[1]]},
                        EdgeWeight];
44                  PropertyValue[{g,V[[1]] \[UndirectedEdge] edge [[1]]}, EdgeWeight]=
                        weightO+weight,
45                  If[edge[[1]]<V[[1]],
46                    g=EdgeAdd[g,edge[[1]] \[UndirectedEdge] V[[1]]];
47                    PropertyValue[{g,edge[[1]] \[UndirectedEdge] V[[1]]}, EdgeWeight]=
                          weight,
48                    g=EdgeAdd[g,V[[1]] \[UndirectedEdge] edge [[1]]];
49                    PropertyValue[{g,V[[1]] \[UndirectedEdge] edge [[1]]}, EdgeWeight]=
                          weight
50                  ];
51
52                ]
53              ]
54            ]
55          ]
56        ]
57      ]
58    ];
59    Return[g]
60  )
```

Next I have written a function to find a minimum odd T-cut. This function only works when the given graph has a positive weight function. This is because the used function FindEdgeCut only works with a positive weight function.

```
1  (*For the function VertrexContractOwn to work, run the file with that function first.
       *)
2
3  MinTcut[G_,T_]:=
4  (*Input: graph G and set of vertices T, such that |T| is even.*)
```

```
(*Output: the value of a minimal odd T-cut delta(S), and the set S.*)
Module[{s,t,stCutEdges,stCut,g,ConComp,S,T1,G1,MinCut1Set,MinCut1,S1,MinCut,Smin,
    T2,G2,MinCut2Set,MinCut2,S2,SCompl,T3,G3,MinCut3Set,MinCut3,S3},
  If[OddQ[Length[T]],Print["You can only fill in T of even length."]];
  If[Length[T]==0,
    Return[{Infinity,{}}],

    (*Find a minimum s-t cut.*)
    s=T[[1]];t=T[[2]];
    stCutEdges=FindEdgeCut[G,s,t];
    stCut=EdgeConnectivity[G,s,t];

    (*Determine the set S.*)
    g=EdgeDelete[G,stCutEdges];
    ConComp=ConnectedComponents[g];
    For[i=1,i<=Length[ConComp],i++,
      If[MemberQ[ConComp[[i]],s],
        S=ConComp[[i]]
      ]
    ];

    (*Check if S is T-odd or T-even.*)
    If[OddQ[Length[Intersection[S,T]]],
      (*Determine Subscript[T, s,t], Subscript[G, s,t] and MinTcut[Subscript[G, s,t],
          Subscript[T, s,t]].*)
      T1=Complement[T,{s,t}];G1=VertexContractOwn[G,{s,t}];
      MinCut1Set=MinTcut[G1,T1];
      MinCut1=MinCut1Set[[1]];S1=MinCut1Set[[2]];

      (*Determine minimum of minimum s-t cut and MinTcut[Subscript[G, s,t],Subscript[
          T, s,t]].*)
      MinCut=Min[stCut,MinCut1];
      (*Set Smin to the correct set S.*)
      If[MinCut==stCut,
        Smin=S,
        If[MemberQ[S1,s],
          Smin=Union[S1,{s,t}],
          Smin=S1
        ]
      ],
      (*Determine Subscript[T, S], Subscript[G, S] and MinTcut[Subscript[G, S],
          Subscript[T, S]].*)
      T2=Complement[T,S];G2=VertexContractOwn[G,S];
      MinCut2Set=MinTcut[G2,T2];
      MinCut2=MinCut2Set[[1]];S2=MinCut2Set[[2]];

      (*Determine Subscript[T, Scomplement], Subscript[G, Scomplement] and MinTcut[
          Subscript[G, Scomplement],Subscript[T, Scomplement]].*)
      SCompl=Complement[T,S];
      T3=Intersection[T,S];G3=VertexContractOwn[G,SCompl];
      MinCut3Set=MinTcut[G3,T3];
      MinCut3=MinCut3Set[[1]];S3=MinCut3Set[[2]];

      (*Determine minimum of MinTcut[Subscript[G, S],Subscript[T, S]] and MinTcut[
```

```
            Subscript[G, Scomplement],Subscript[T, Scomplement]].*)
54        MinCut=Min[MinCut2,MinCut3];
55        (*Set Smin to the correct set S.*)
56        If[MinCut==MinCut2,
57          If[MemberQ[S2,S[[1]]],
58            Smin=Union[S2,S],
59            Smin=S2
60          ],
61          If[MemberQ[MinCut3Set[[2]],SCompl[[1]]],
62            Smin=Union[S3,SCompl],
63            Smin=S3
64          ]
65        ]
66      ];
67      Return[{MinCut,Smin}]
68    ];
69  ]
```

At last, the code to solve the undirected Chinese Postman Problem. In this code we call the function MinTcut with $\boldsymbol{x}$ as weight function, and $\boldsymbol{x}$ might have zeros. To resolve this problem I added 10^{-9} to each value in $\boldsymbol{x}$ (lines $71-77$). I added 10^{-9} because with edge weights smaller or equal to 10^{-10} the function FindEdgeCut did not work.

Note that adding something to each value of the weight function could change the minimum cut, because then cuts of fewer edges might be preferred. However, we are assuming that for the small example graphs we considered, and for the weight functions arising from these, the correct minimum cuts are still found by the Mathematica function FindEdgeCut, when 10^{-9} is added to each value in $\boldsymbol{x}$.

```
1  (*For the function MinTcut to work, run the file with that function first .*)
2
3  (*To create random connected graph.*)
4  n=10; (*vertices*)
5  m=15; (*edges*)
6  c=RandomInteger[{1,10},m]; (*weight function*)
7  g1=RandomGraph[{n,m},EdgeWeight->c,VertexLabels->"Name",EdgeLabels->"
       EdgeWeight"];
8  g2=ConnectedComponents[g1];
9  G=Subgraph[g1,g2[[1]]]
10
11 (*Check if G Eulerian.*)
12 If[EulerianGraphQ[G],
13   Print["This graph is Eulerian, the Euler tour is"];
14   FindEulerianCycle[G],
15   Print["This graph is not Eulerian."];
16
17   (*Create T, the set of verticeses of odd degree*)
18   T={};
19   For[i=1,i<=n,i++,
20     If[OddQ[VertexDegree[G,i]],AppendTo[T,i]]
21   ];
```

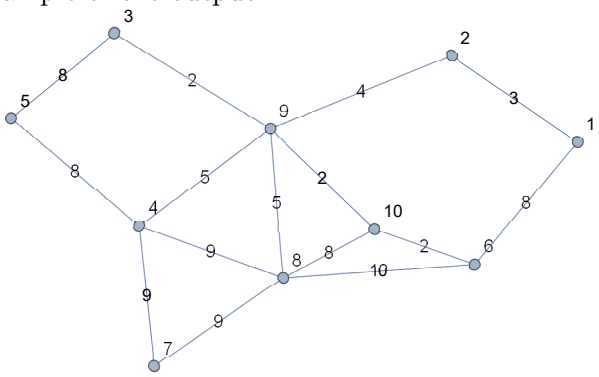
```
22  p=Length[T];
23
24  (*Determine the shortest s-t paths, and their length using Dijkstra's algorithm.*)
25  l={};shp={};
26  For[i=1,i<=p,i++,
27    For[j=i+1,j<=p,j++,
28      AppendTo[l,GraphDistance[G,T[[i]],T[[j]], Method->"Dijkstra"]];
29      AppendTo[shp,FindShortestPath[G,T[[i]],T[[j]], Method->"Dijkstra"]];
30    ]
31  ];
32
33  (*Create auxiliary complete graph, with vertices the elements in T.*)
34  Edges={};
35  For[i=1,i<=p-1,i++,
36    For[j=i+1,j<=p,j++,
37      AppendTo[Edges,T[[i]] \[UndirectedEdge] T[[j]]]
38    ]
39  ];
40
41  (*Make Kaux without weight, such that weights can later be changed.*)
42  Kaux=Graph[T,Edges,VertexLabels->"Name"];
43  K=Graph[Kaux,EdgeWeight->l,EdgeLabels->"EdgeWeight"];
44  Print[" Auxiliary complete graph"];
45  Print[K];
46
47  (*Create A' and b.*)
48  A=Join[IncidenceMatrix[K],-IncidenceMatrix[K]];
49  b=Join[Array[1&,p],-Array[1&,p]];
50
51  (*Print shortest path vector to check in resulting graph.*)
52  Print["Corresponding shortest paths" ];
53  Print[shp];
54
55  (*Find minimal length perfect matching using the Simplex method.
56  LinearProgramming[c,A,b] solves the LP: min{cx|Ax>=b,x>=0}.*)
57  x=LinearProgramming[l,A,b,Method->"Simplex"];
58
59  (*Check if x is an integer vector.*)
60  IntegerVector:=Apply[And]@*Map[IntegerQ @* Rationalize];
61  Test=True;
62  While[Test,
63  (*If the constraints for all odd subsets S are added, an integer x will be found.
64  That means the while loop will always stop eventually.*)
65    If[IntegerVector[x],
66      (*x is integer, so stop the while loop.*)
67      Test=False,
68      (*x is not integer, so find odd subset for which the extra constraint does not
            hold.*)
69      Kh=Graph[Kaux,EdgeWeight->x,EdgeLabels->"EdgeWeight"];
70
71      (*The function MinTcut does not work with edge weights of zero. To prevent that
            edge weights of zero occur, increase each edge weight by 10^(-9).
72      Adding a value to each edge may result in a different mininmum cut, one that
            contains fewer edges.
```

```
73      To make sure we still get the same cut we add a very small value, namely
            10^(-9).*)
74      For[i=1,i<=Length[Edges],i++,
75        OriginalWeight=PropertyValue[{Kh,Edges[[i]]}, EdgeWeight];
76        PropertyValue[{Kh,Edges[[i ]]}, EdgeWeight]=OriginalWeight+10^(-9)
77      ];
78
79      (*Determine minimum odd T-cut.*)
80      MinCutSet=MinTcut[Kh,T];
81      S=MinCutSet[[2]];
82
83      (*Create delta(S).*)
84      deltaS=EdgeList[Kh,S[[1]] \[UndirectedEdge] _];
85      For[i=2,i<=Length[S],i++,
86        edges=EdgeList[Kh,S[[i]] \[UndirectedEdge] _];
87        For[j=1,j<=Length[edges],j++,
88          If[MemberQ[deltaS,edges[[j]]],
89            deltaS=Complement[deltaS,{edges[[j]]}],
90            AppendTo[deltaS,edges[[j ]]]
91          ]
92        ]
93      ];
94
95      (*Increasing each edge weight by 10^(-9) results in the value of the cut being
            increased by Length[deltaS]*10^(-9), this needs to be undone.*)
96      MinCut=MinCutSet[[1]]-Length[deltaS]*10^(-9);
97
98      If[MinCut<1,
99        (*Add constraint for S, run linear program again.*)
100       (*Make a vector s of edges in delta(S).*)
101       s={};
102       For[i=1,i<=Length[Edges],i++,
103         If[MemberQ[deltaS,Edges[[i]]],
104           s=AppendTo[s,1],
105           s=AppendTo[s,0]
106         ]
107       ];
108       (*Append s to A, append 1 to b, run linear program again.*)
109       AppendTo[A,s];
110       AppendTo[b,1];
111       x=LinearProgramming[l,A,b,Method->"Simplex"],
112       (*x is not integer, so this is not supposed to happen.*)
113       Return["Error"];
114     ]
115   ]
116 ];
117
118 Print["Minimal length perfect matching is"];
119 Print[x];
120
121 (*For each i such that x[[i]]=1, we want to add the corresponding shortest path to
        our graph.*)
122 (*To do this we first need to make shp into a list of edges instead of vertices.*)
123 For[i=1,i<=Length[x],i++,
```

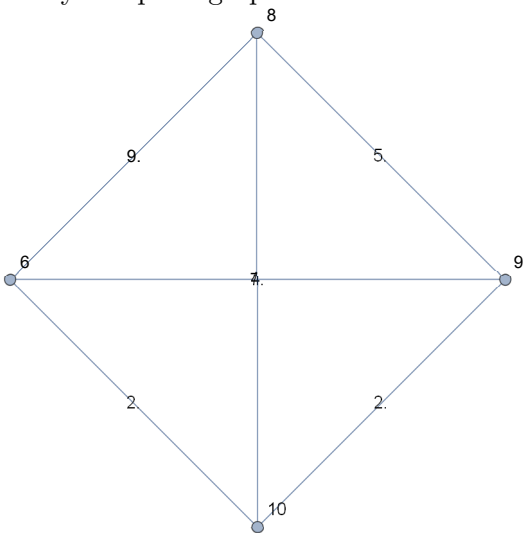
```
124       If[x[[i]]==1,
125         G=EdgeAdd[G,UndirectedEdge@@@Partition[shp[[i]],2,1]]
126       ]
127     ];
128     Print["Graph with extra edges to make it Eulerian."];
129     Print[G];
130
131     (*Finally, the postman tour.*)
132     Print["The postman tour is"];
133     FindEulerianCycle[G]
134   ]
```

An example of the output:

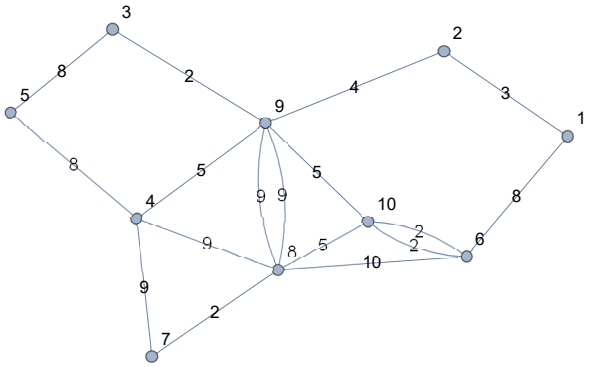
Out[•]=



This graph is not Eulerian.
Auxiliary complete graph



Corresponding shortest paths
$\{\{6, 10, 9, 8\}, \{6, 10, 9\}, \{6, 10\}, \{8, 9\}, \{8, 9, 10\}, \{9, 10\}\}$
Minimal length perfect matching is
$\{0., 0., 1., 1., 0., 0.\}$
Graph with extra edges to make it Eulerian.



The postman tour is

$\{\{1-6, 6-10, 10-9, 9-8, 8-10, 10-6, 6-8, 8-9, 9-4, 4-8, 8-7,$

$7-4, 4-5, 5-3, 3-9, 9-2, 2-1\}\}$

Appendix B

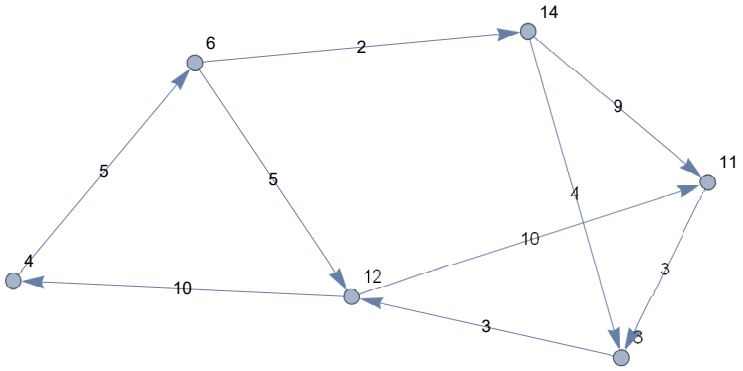
Code for Directed Case

```
(*To create a random strongly connected directed graph.*)
n=15; (*vertices*)
m=23; (*edges*)
c=RandomInteger[{1,10},m]; (*weight function*)
g1=RandomGraph[{n,m},EdgeWeight->c,VertexLabels->"Name",EdgeLabels->"
    EdgeWeight" ];
g2=DirectedGraph[g1,"Random"];
g3=ConnectedComponents[g2];
v=With[{L=Length/@g3},Pick[g3,L,Max@L]];
(*v is a list of all strongly connected components of maximum size, so only use one
    element of v to create a strongly connected graph*)
G=Subgraph[g2,v[[1]]]

(*Check if G Eulerian.*)
If[EulerianGraphQ[G],
   Print["This graph is Eulerian, the Euler tour is"];
   FindEulerianCycle[G],
   Print["This graph is not Eulerian."];

   (*Create T^+ (A) and T^- (B), simultaneously keep track of the values of the b
       function (bv)*)
   A={}; B={}; bv={};
   For[i=1,i<=n,i++,
      If[VertexOutDegree[G,i]>VertexInDegree[G,i],
         AppendTo[A,i];
         AppendTo[bv,VertexOutDegree[G,i]-VertexInDegree[G,i]];
      ]
   ];
   For[i=1,i<=n,i++,
      If[VertexInDegree[G,i]>VertexOutDegree[G,i],
         AppendTo[B,i];
         AppendTo[bv,VertexInDegree[G,i]-VertexOutDegree[G,i]]
      ]
   ];
   p=Length[A]; q=Length[B];

   (*Determine the shortest b-a paths and their lengths using Dijkstra's algorithm.
```

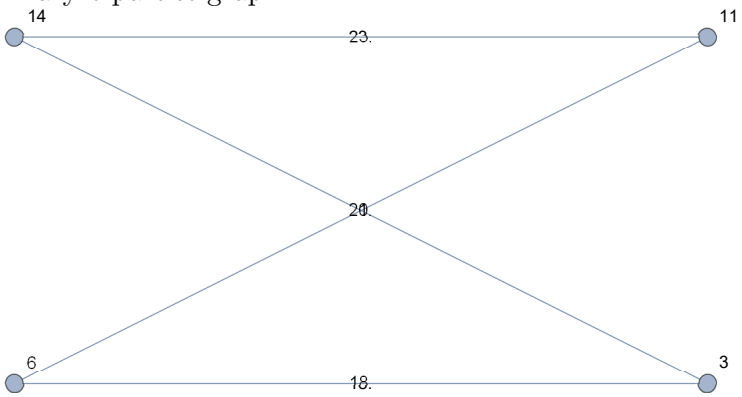
```
        These are stored in the lists shp and l. The shortest paths are stored as
        sequence of vertices .*)
35    l={};shp={};
36    For[i=1,i<=q,i++,
37        For[j=1,j<=p,j++,
38            AppendTo[l,GraphDistance[G,B[[i]],A[[j ]], Method->"Dijkstra"]];
39            AppendTo[shp,FindShortestPath[G,B[[i]],A[[j ]], Method->"Dijkstra"]]
40        ]
41    ];
42
43    (*Create auxiliary complete bipartite graph, with as vertices the elements in A
        and B.*)
44    BiG=Graph[Join[A,B],Flatten[Outer[UndirectedEdge,B,A]],EdgeWeight->l,
        VertexLabels->"Name",EdgeLabels->"EdgeWeight",GraphLayout->"
        BipartiteEmbedding"];
45    Print[" Auxiliary  bipartite  graph"];
46    Print[BiG];
47
48    (*Create A' and b'.*)
49    Inc=Join[IncidenceMatrix[BiG],-IncidenceMatrix[BiG]];
50    b=Join[bv,-bv];
51
52    (*Print shortest path vector to check in resulting graph.*)
53    Print["Corresponding shortest paths"];
54    Print[shp];
55
56    (*Find minimal length perfect b-matching using the Simplex method.
        LinearProgramming[c,A,b] solves the LP: min{cx|Ax>=b,x>=0}.*)
57    x=LinearProgramming[l,Inc,b,Method->"Simplex"];
58    Print["Minimal length perfect b-matching is"];
59    Print[x];
60
61    (*For each i such that x[[i]]=1, we want to add the corresponding shortest path to
        our graph. To do this we first need to make shp into a list of edges instead
        of vertices .*)
62    For[i=1,i<=Length[x],i++,
63        If[x[[i]]==1,
64            G=EdgeAdd[G,DirectedEdge@@@Partition[shp[[i]],2,1]]
65        ]
66    ];
67    Print["Graph with extra edges to make it Eulerian."];
68    Print[G];
69
70    (*Finally, the postman tour.*)
71    Print["The postman tour is"];
72    FindEulerianCycle[G]
73 ]
```

An example of the output:

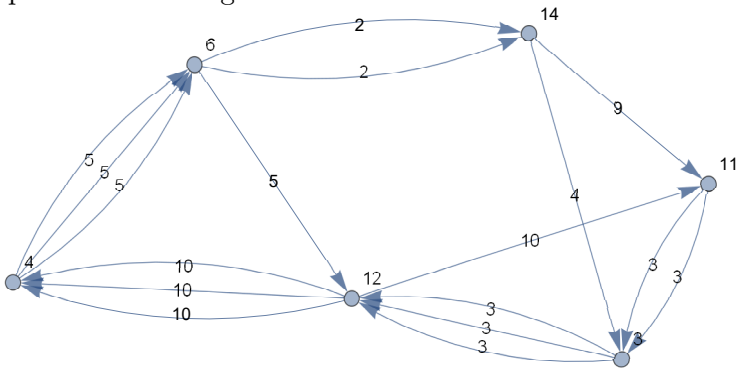
Out[ ]=



This graph is not Eulerian.
Auxiliary bipartite graph



Corresponding shortest paths
$\{\{3, 12, 4, 6\}, \{3, 12, 4, 6, 14\}, \{11, 3, 12, 4, 6\}, \{11, 3, 12, 4, 6, 14\}\}$
Minimal length perfect b-matching is
$\{0., 1., 1., 0.\}$
Graph with extra edges to make it Eulerian.



The postman tour is

$\{\{3 \to 12, 12 \to 4, 4 \to 6, 6 \to 12, 12 \to 4, 4 \to 6, 6 \to 14, 14 \to 3,$

$3 \to 12, 12 \to 4, 4 \to 6, 6 \to 14, 14 \to 11, 11 \to 3, 3 \to 12, 12 \to 11, 11 \to 3\}\}$