ECMI Modeling Week 2017
Lappeenranta University of Technology

Snowplow Route Optimisation

Albert Miguel López (Autonomous University of Barcelona)
Carl Assmann (Bundeswehr University Munich)
Edyta Kabat (Jagellionian University of Cracow)
Gandalf Saxe (Technical University of Denmark)
Matthew Geleta (University of Oxford)
Sara Battiston (University of Milan)

Supervised by Miika Tolonen (Lappeenranta University of Technology)

September 2017

Contents

1 Introduction **2**

2 Problem Description **4**

3 Mathematical Formulation of the Problem **7**

3.1 The Chinese Postman Problem 7

3.2 Algorithms 8

4 Getting Data **13**

4.1 Generating Data Using the Google Street Maps API 13

4.2 Constructing the Road Network in Lappeenranta 14

4.3 Visualization of Solutions 15

5 Solution 1: Classical Solution to the CPP **18**

5.1 Solving the Basic CPP for Closed and Open Paths 18

5.2 Algorithm Scaling 21

5.3 Extension to two-way roads 22

6 Solution 2: Stochastic Algorithm **27**

6.1 Motivation 27

6.2 Finding a Penalty Scout Tour 27

6.3 Results using Penalty Scout 30

6.4 Modified Snowplow Problems 31

7 Discussion and Further Work **34**

References **37**

1 Introduction

During the cold Winter months in Finland and neighboring countries, a great deal of time and money must be spent on clearing snow from roadways and footpaths. Similar problems are well-known among residents of many other European countries, as well as those of Canada and Alaska. In some countries, snowfall is light and roads can be effectively cleared using relatively inexpensive means, such as by individual people with shovels, or by chemical methods such as salt treatment.

In Finland, the Winter snowfall is sufficiently heavy that roads must be cleared daily by large and expensive snow-plowing vehicles. If snow is not cleared quickly, thick sheets of ice may form on the roads. Ice is far more difficult and expensive to remove than soft snow. Moreover, storm drains may block if snow is not cleared regularly, not to mention that transportation is brought to a halt when snow-cover is too thick. An effective snow-plowing strategy is therefore a crucial economic challenge. A poorly designed plowing strategy carries a severe economic burden, costing the Finish government millions of Euros. As an indication of how severe this problem can be, Figure 1 shows a snow-covered street in Lappeenranta.

Figure 1: *A road in the Lappeeenranta area during winter.*

Mathematical optimisation supplies tools for designing efficient snow-plowing strategies. The task of snowplow routing is a (challenging) com-

binatorial optimization problem, and is an active research area in the algorithms and operational research communities. Some recent papers on the topic include the *genetic algorithms* described in [1, 2, 3]. The book by Campbell et al. [4] provides an overview of recent optimisation methods for snowplow route planning.

Report Overview

Our 2017 ECMI project was to develop an optimised snow-plowing strategy for the city of Lappeenranta. In this paper we report some of our main findings and contributions. Section 2 describes the snow-plowing problem and some variations thereof. Section 3 formulates the snow-plowing optimisation mathematically using the language of graph theory, and includes some relevant algorithms. Section 4 describes the real-world data that we used to construct the Lappeenranta road-network, and how the data were obtained. Section 5 provides an exact algorithm and a complete analytical solution to a simple version of the snow-plowing problem, and describes our software implementation. Section 6 provides a stochastic algorithm of our own invention that we used to solve more complicated versions of the snow-plowing problem, and present some results for the Lappeenranta road network. In this section we have also provided a visualisation of the solution, and an internet link to a video animation of the optimal snow-plowing route. Our findings are discussed in Section 7.

2 Problem Description

We were tasked with developing optimized snow-plowing routes for the town of Lappeenranta under various conditions. The optimized routes would then be provided to snowplow drivers. In the long-term, our work would be extended to a smart-phone application, and our current project can be thought of as laying foundations for the application.

The problems we considered were the following:

Problem 1: One driver. Given a connected subset of a road newtork, determine a route that visits every road while driving the shortest possible distance. Over the course of a year, such a snowplowing route would result in significant savings in terms of fuel consumption and vehicle maintenance.

Problem 2: Multiple drivers. As in Problem 1, but with multiple snowplow vehicles operating simultaneously.

Problem 3: Roads with priorities. As with Problem 1, but with a partial ordering in which roads are to be visited. For instance, highways must be cleared before residential roads, which must be cleared before pedestrian footpaths and cycle lanes.

Problem 4: One-way streets. As with Problem 1, except that some streets can only be travelled in one direction.

Problem 5: Two-way streets. As with Problem 1, except that some streets must be travelled twice to be completely cleared. Represents big streets that the machine can not clear in one go.

The following data were provided:

- A list of road names and their lengths — Figure 2
- A basic map — Figure 3
- A priority scheme for each road — Figure 4

KADUT	pituus m 58 574
I LK	**2 168**
LAVOLANTIE	1 277
MUNTERONKATU	198
RUOHOLAMMENKATU	560
SKINNARILANKATU	133
II LK	**10 059**
HIETAKALLIONKATU	164
HOPEAMÄENRAITTI	166
JUPITERINKATU	201
JÄRVINIITUNKATU	296
KAARTINKATU	110
KALLIOPELLONKATU	264
KARJAKATU	209
KIERTOKATU	116
KIVINIEMENRAITTI	65
KORPIKUNNAANKATU	302
KORPIMAANKATU	141
KORPIMETSÄNKATU	152
KORPISUONKATU	372

Figure 2: *Lappeenranta street names*

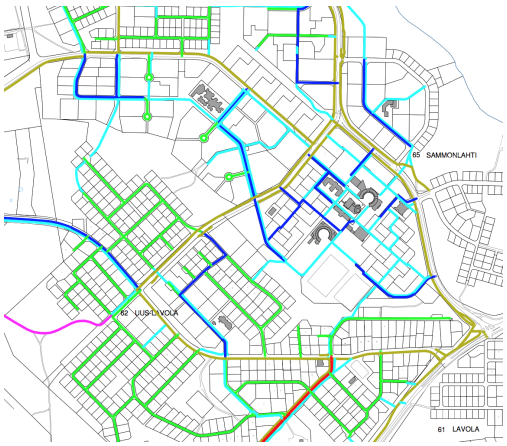

Figure 3: *Basic map of Lappeenranta*

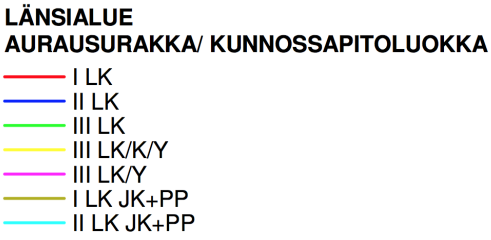

Figure 4: *Priority scheme*

As will be discussed in Section 4, the data provided was not sufficient to solve the problem, and alternative sources had to be used. Before we describe the data we used, let us describe how we formulated the problem mathematically.

3 Mathematical Formulation of the Problem

Let us refer to the snowplow route optimisation problem as (SP), for "Snowplow Problem". (SP) and its variants can be elegantly described using the language of graph theory, enabling the use of powerful graph-theoretic algorithms and theorems.

Some basic graph theory. A *graph* $\mathcal{G} = (V, E)$ consists of a non-empty set of nodes V and a non-empty set of edges E that connect nodes. $\mathcal{G}$ is called *connected* if every pair of nodes in V can be connected by a sequence of edges in E. A graph is called *directed* if the edges have a specified direction, and is otherwise called *undirected*.

In the context of (SP), we use a graph $\mathcal{G}$ to describe the road network; the nodes V represent street-intersections, and the edges E represent segments of road between each intersection. Distances are incorporated by weighting each edge $e \in E$ by the (non-negative) length $d(e)$ of the corresponding segment of road between each intersection.

A snowplow route takes the form of a *tour*, which is a sequence of nodes and edges of the form $T = (n_1, e_1, n_2, e_2, \ldots, n_l, e_l, n_{l+1})$. The same nodes and edges may appear more than once in a single tour. The total cost of the tour T is $c(T) = \sum_{j=1}^{l} d(e_j)$, which is the sum of the distances of individual edges. If the starting-node and ending-node are equal (that is, if $n_1 = n_{l+1}$) then the tour is said to be *closed*.

For some graphs, it is possible to find a tour T that contains every edge *exactly* once. Such graphs are called *Eulerian*, and the tour is called an *Euler tour*. As we shall see, the concept of an Eulerian graph will play an important role in the algorithms we use to solve (SP), because an Euler tour has minimum cost among all tours that contain every edge.

3.1 The Chinese Postman Problem

Having translated (SP) into the language of graph theory, let us point out an equivalence between (SP) and the well-known *Chinese Postman Problem*,

which is a classical problem in graph theory. The problem is the following: *given a connected undirected weighted graph $\mathcal{G}$, find a minimum-cost closed tour T^* that contains all edges at least once.* Mathematically, the problem is the following:

$$c(T^*) = \min_T c(T) \qquad \text{(CPP)}$$
$$\text{subject to } e \in T \quad \forall e \in E$$

Note that if $\mathcal{G}$ is a road network, then solving (CPP) is equivalent to solving (SP) for that network. Therefore an algorithm that solves (CPP) also solves (SP). There are some variants of (CPP) that are related to the variants of (SP):

Directed (CPP)**:** The Chinese Postman Problem using directed graphs is equivalent to (SP) with one-way roads.

Hierarchical (CPP)**:** This variant of the Chinese Postman Problem specifies a partial ordering of edges, such that some edges must appear before others in the (CPP) tour T. This variant is equivalent to (SP) with priorities for each road.

Fortunately, (CPP) is a well-studied problem, and there exist many efficient (that is, *polynomial-time*) algorithms for its solution. The algorithms we have used are described in Sub-section 3.2. The other (CPP) variants are much more difficult (most have been proven to be NP-hard), and for these we have developed stochastic and heuristic algorithms, which are described in Section 6.

3.2 Algorithms

The (CPP) algorithms we use are based on the properties of Eulerian graphs, and some preliminary facts about these graphs will be useful. Firstly, there exist efficient algorithms for finding Euler tours in Eulerian graphs. We use Algorithm 1 as a sub-routine in our (SP) algorithm.

The importance of Algorithm 1 for (CPP) is based on Proposition 3.1.

Algorithm 1: Hierholzer's Algorithm for Finding an Eulerian Tour

Input : Eulerian graph $\mathcal{G} = (V, E)$, and starting node $n_1 \in V$
Output: An Euler tour $T = (n_1, e_1, n_2, e_2, \ldots, n_l, e_l, n_{l+1} = n_1)$
Init : Set $E' \leftarrow \emptyset$, $E_1 = \emptyset$

Starting from n_1, trace any simple tour T, and call the set of traced edges E'
repeat
    Chose any vertex n in T incident to an edge not in E'
    Trace a second simple path T_1, starting from n, using edges in the set E/E'. Call the set of newly traced edges E_2
    Merge T_1 with the T at the intersection node n
    Set $E' \leftarrow E' \cup E_1$
until $E' == E$
Return the Euler tour T.

Proposition 3.1. *If $\mathcal{G}$ is an Eulerian graph or multigraph, then its Euler-tour solves* (CPP) *on $\mathcal{G}$.*

According to Proposition 3.1, if the (SP) road network is Eulerian, then Algorithm 1 solves (SP) optimally. Before applying Algorithm 1 to a real road network, we must determine whether or not the network is Eulerian. The following lemma is useful:

Lemma 3.1 (Eulerian Graph)**.** *A connected undirected graph is Eulerian if and only if the degree of every node is even [5].*

Lemma 3.1 provides an easy method for determining whether or not a graph is Eulerian, by simply checking the degree of each node.

Unfortunately, the Lappeenranta road network is not Eulerian. However, it turns out that (CPP) can be solved on a non-Eulerian graph $\mathcal{G}$ by first augmenting $\mathcal{G}$ to obtain a graph $\mathcal{G}'$ that is Eulerian. The augmentation can be implemented using using Algorithm 3, which will be discussed shortly. Armed with this knowledge, Algorithm 2 provides a method for solving the Chinese Postman Problem.

Algorithm 2: Overview of the (CPP) Algorithm

Input : An undirected connected graph $\mathcal{G} = (V, E)$

Output: A (CPP) tour $T = (n_1, e_1, n_2, e_2, \ldots, n_l, e_l, n_{l+1} = n_1)$

1 Determine if $\mathcal{G}$ is Eulerian using Lemma 3.1

2 If $\mathcal{G}$ is Eulerian, obtain an Euler-tour T using Algorithm 1 and stop

3 If $\mathcal{G}$ is not Eulerian, find a minimum-cost augmentation of $\mathcal{G}$ to construct an Eulerian multigraph $\mathcal{G}'$, and let T be and Euler-tour of $\mathcal{G}'$. Stop.

We implemented Step 3 of Algorithm 2 using the minimum-cost Eulerian augmentation algorithm due to [6]. The method is stated in Algorithm 3.

The main mathematical difficulty encountered in Algorithm 3 is the minimum-cost maximum-cardinality matching[1] sub-problem (MM) in Step 3. The first approach we tried was based on the following lemma.

Lemma 3.2 (Handshake Lemma)**.** *Every finite undirected graph has an even number of odd-degree nodes [5].*

Lemma 3.2 guarantees that the set S in Algorithm 3 has an even number of nodes, which means that any maximum-cardinality matching is in fact a pairing. Thus, one method of solving (MM) is to enumerate all pairings, and to select a pairing of minimal cost. We used this approach to successfully solve (MM) and (CPP) for some some small example graphs. However, the scaling of this method is combinatorial: if the number of nodes in S is $|S| = 2m$, then the number of pairs is ${}^{2m}C_m$, and the number of pairings is $(2m)!/(2^m)$. This poor scaling prohibits the use of our simple method for the map of Lappeenranta.

Fortunately there exists an algorithm for (MM) with $O(|S|^3)$ scaling. It is called the *Blossom Algorithm*, and is due to [7, 8].[2]

[1] A maximum matching (also known as maximum-cardinality matching) is a matching that contains the largest possible number of edges [9].

[2] The Blossom Algorithm is too involved to discuss in detail here, and we refer the reader to [10] for a detailed description. In our code, we used a Python implementation of the Blossom Algorithm written by Joris van Rantwijk, which is available from [11].

Algorithm 3: Minimal-Cost Eulerian Augmentation [6]

Input : An undirected connected weighted non-Eulerian graph $\mathcal{G} = (V, E)$.

Output: An Eulerian multigraph $\mathcal{G}' = (V, E')$ obtained by a minimal-cost augmentation of $\mathcal{G}$.

1 Set $S \subset V$ the subset of odd-degree nodes;

2 For each $n_i, n_j \in S$, set c_{ij} equal to the length of the shortest-path between n_i and n_j in $\mathcal{G}$;

3 Construct the complete undirected graph $\mathcal{G}_p = (S, E_p)$, whose edges have weights c_{ij};

4 From the set $\mathcal{P}$ of all possible matchings of nodes in S, select a minimum-cost maximum-cardinality matching M:

$$M \in \arg\min_{p \in \mathcal{P}} \text{cost}(p), \tag{MM}$$

subject to $\text{card}(p) \geq \text{card}(q) \quad \forall q \in \mathcal{P}$;

5 Return the graph $\mathcal{G}' = (V, E')$, where the edge-set E' consists of all original edges in E, and additional copies every edge along the shortest paths between paired nodes in M.

We implemented Step 2 using Dijkstra's algorithm, whose cost is $O(|T|^2)$.

We implemented Step 3 using the Blossom Algorithm [7, 8], whose cost is $O(|T|^3)$.

Note that previous discussion applies only to closed tours. If we allow for CPP solutions that start and end at different nodes, there are special cases in which an eulerian path can be found for non-eulerian graphs. $\mathcal{G}$ is called *semi-eulerian* if it has one single pair of odd nodes (we call them n_{o1} and n_{o2}). It can be proven that for semi-eulerian networks there exists at least one tour T with $n_1 = n_{o1}$ and $n_{l+1} = n_{o2}$ that goes through all edges exactly once, without need of backtracking. This is because starting/ending nodes have effectively one degree less, so with clever choice of a path $\mathcal{G}$ becomes eulerian in regards to the (CPP).

In the general case with more than one pair of odd nodes, we can still choose different start and ending points. If we select a pair from the optimal matching as n_1 and n_{l+1}, their degree is effectively reduced by one and we can remove them from the final path computation, since they can be considered as "even" nodes.

4 Getting Data

Having translated the snowplow routing problem into the language of graph theory, we turn now to the task of constructing the relevant graphs from real-world data.

4.1 Generating Data Using the Google Street Maps API

The only data provided by the city of Lappeenranta were lists of street names and their priorities. These lists contained streets which have to be plowed by the local contractor (Figure 2), but this information alone was insufficient for solving the problem. Our first task was therefore to use the given data to construct a usable graph representing the road network.

We decided to use APIs provided by Google Maps, partly due to prior experience in the team, and partly due to their ease of use and excellent documentation. The first API we used is called the *Google Geo Coding API*. It allows one to find the intersection of two streets, if one exists, as well as the coordinates of the intersection. We accessed this information from the API using Algorithm 4.

Algorithm 4: Get Intersections from API

Input : L =List of street names
Output: List of intersections (Street A, Street B, Coordinates)
Init : $l \leftarrow \emptyset$

```
for i ∈ L do
    for j ∈ L do
        ask Google for intersection between i and j
        if streets intersect, set l ← l ∪ (i, j, coords)
    end
end
```

Now having a list of all intersections and coordinates, we used the *Google Distance Matrix API* to compute the distances between each pair

of neighbouring intersections, which is the shortest route between those intersections. We used the Algorithm 5 to generate a distance matrix corresponding to the intersections.

Algorithm 5: Get Distance Matrix

Input : L, a list of intersection-names with coordinates
Output: Distance Matrix $A \in \mathbb{R}^{n \times n}$

```
for i ∈ L do
    for j ∈ L do
        if i and j are neighbors then
            ask the Google API the distance between them, and set
             A(i, j) ←distance
        else
            Set A(i, j) ← 0
        end
    end
end
```

Unfortunately, the Google Maps API is rate-limited, and we exceeded the daily limit when using the Lappeenranta street data. For every 1000 request above the free daily limit, Google charges €0.5. Using our code we would need approximately €50-100 to generate the distance matrix. Therefore we worked on a smaller version of the Lappeenranta street map, covering the priority 1 and 2 roads only.

4.2 Constructing the Road Network in Lappeenranta

As described above, due to financial constraints we ended up crafting the distance matrix of road intersections by hand for a smaller road network. We used Google Maps, and a service called "My Maps," where one can mark points manually and get the distances between each pair of points. The points we chose are shown in Figure 5. The roads which are connecting our 24 points cover all priority 1 and priority 2 roads. We decided that for

testing our solution attempts this network is sufficently large and complex.

4.3 Visualization of Solutions

In order to visually verify our solutions (as well as having fancy demo material), we wanted to animate the tours found by the two solvers.

We used the Google Maps Directions API to obtain coordinates for routes between read intersections and Google Maps Javascript API to animate a symbol on the maps. This could be done after the solution was converted from a list of nodes to a list of (latitude, longitude) coordinates, using the Google Maps Directions API.

The animations themselves and more details can be found at the project's Github page [12].

Figure 5: *Lappeenranta relevant intersections, illustrated as a Google Map.*

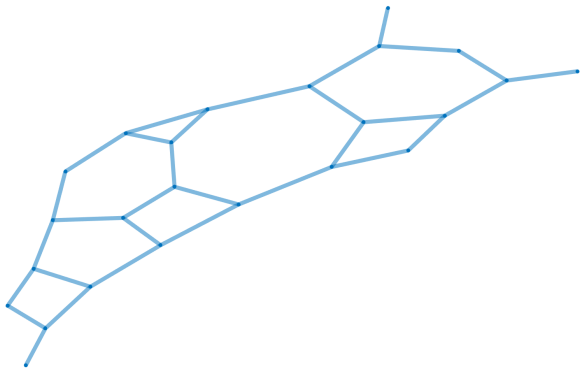

Figure 6: *The Lappeenranta snowplow network in the form of a planar graph.*

Figure 7: *Screenshot of animation.*

5 Solution 1: Classical Solution to the CPP

5.1 Solving the Basic CPP for Closed and Open Paths

We have adapted the theory described in Section 3 to a Python code. We used the following libraries:

- `Numpy:` basic numerical python package that allows the creation and manipulation of arrays.
- `Networkx:` used to generate, modify and draw graphs in python. Has typical graph theory algorithms efficiently implemented.
- `Pandas:` built on top of numpy, allows arrays to have arbitrary numbers of rows and columns, which helps with the manipulation of data. Used to create the distance matrix between nodes with are labelled non-numerically, such as streets labelled by letters or street names.
- `Itertools:` library that comes with python, used for efficient combinatorial functions.

The main part of the code is in the script `chinese_postman_algorithm.py`. The function `solve_chinese_postman_problem` takes as input a graph, a starting point, and an ending point, and returns the optimal path that goes through all edges at least once, together with the total distance covered. This is done using Algorithm 2, with a few modifications. The first step is to find the optimal graph extension $\mathcal{G}'$ that makes $\mathcal{G}$ eulerian, which is done through Algorithm 3. If the starting and ending nodes are the same, then Algorithm 8 is enough to find an optimal (CPP) tour T. Figure 8 shows an example of a non-eulerian graph $\mathcal{G}$ and its minimum-weight expansion $\mathcal{G}'$. Every double edge in $\mathcal{G}'$ represents a backtrack. Starting and ending at node 1, a possible solution obtained by using the Python algorithm is: $T = (1, 2, 4, 3, 4, 6, 7, 8, 3, 2, 1)$ with total length of 24. From section 3 we know that this length can not be reduced for any closed tour T that solves the CPP problem. Applying this method to the high-priority road map of

Lappeenranta gives a total distance of 30.525Km, which is shown in one of the videos created with the methods of Section 4.3.

But what happens if we want the starting and ending nodes to be different? There are different reasons that might motivate us to choose $n_1 \neq n_{l+1}$: in a real snow-plowing situation, we might want the plowing machine to go from one place of the map to another, for example from one truck station to the other. Furthermore the solution can be slightly optimized by starting at one of the odd nodes and ending at its minimum-cost pair. As discussed in Section 3.2, this removes the need to repeat the path between them, reducing the total distance covered. If there is only one odd pair, the resulting path is eulerian.

Let n_{o1} and n_{o2} be an odd node pair from the minimum weight matching. Algorithm 1 can not be used to find the path after the optimum augmentation $\mathcal{G}'$ since it only finds closed tours. We developed instead or our one code that finds open eulerian paths in semi-eulerian graphs. First we remove n_{o1}, n_{o2} from the minimal-cost augmentation, so that Algorithm 3 returns a semi-eulerian graph $\mathcal{G}'$, and then apply Algorithm 6 to find the optimal path. This can be understood as a deterministic scout tour, as opposed to the penalty scout tour described in Section 6, that eliminates visited edges and avoids the last node until the end, making sure that at any step the graph will not be left disconnected[3]. Applying this algorithm to the example in Figure 8, starting at $n_{o1} = 1$ and ending at $n_{o2} = 2$ returns the following open tour: $T = (1, 2, 3, 4, 3, 8, 6, 7, 6, 4, 2)$, with a total length of 22. Note that the distance has been reduced from the previous case using node 1 as start and ending point. If open tours are allowed, the real optimum path starts and ends at each node from the odd pair with largest shortest-path length, out of all the pairs obtained with the minimum weight matching.

[3] *Disconnected* means that there exists no path linking $n_i, n_j \in \mathcal{G}$ for at least one pair of i, j.

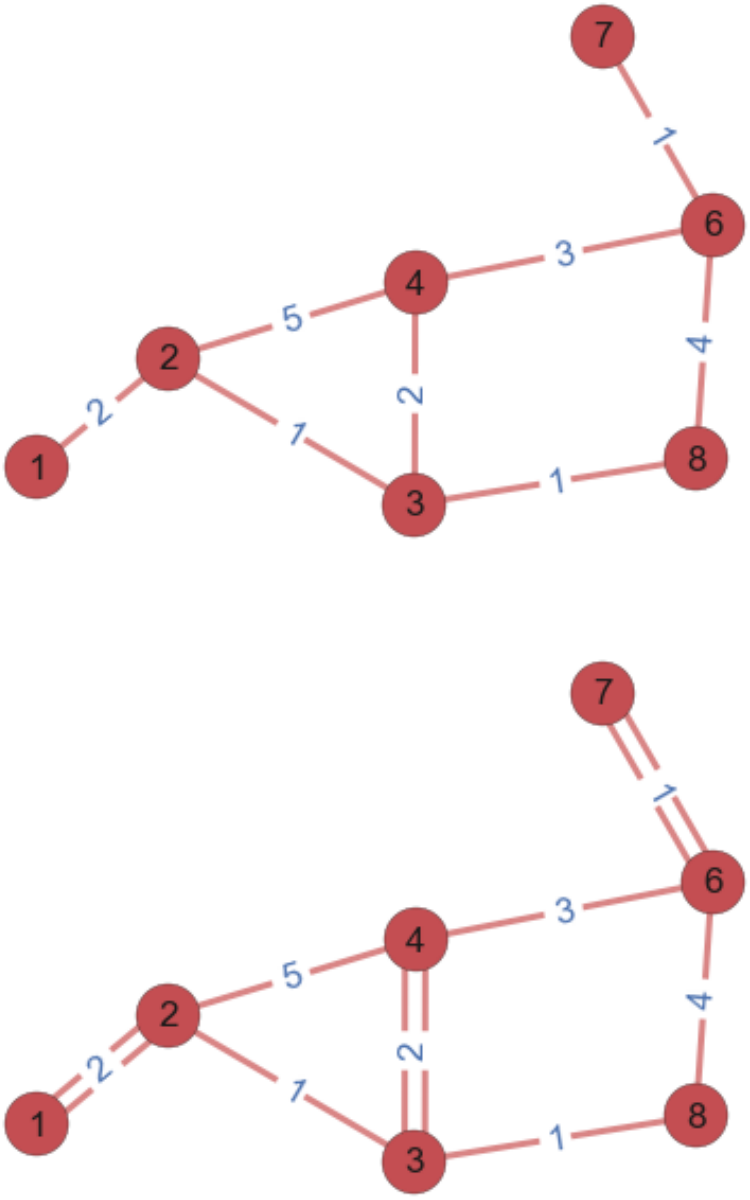

Figure 8: *Example of a non-eulerian graph (top) and its optimal eulerian expansion (bottom). The numbers in the edges mark the weight (or street length in the (SP) problem).*

5.2 Algorithm Scaling

In Section 3.2 we mentioned that finding the eulerian augmented graph $\mathcal{G}'$ with minimum weight scaled in a combinatorial way if the solution is found by brute-force[4]. This method does not scale at all when bigger networks are introduced. For 4 odd nodes there are only 3 combinations to choose, but for 20, which is the number we have for the Lappeenranta high priority road network, we already have more than 10^{16}! Note that the scaling is in regards to the number of odd nodes, since adding even nodes does have little impact on the computations.

The reasons to switch to the Blossom algorithm are obvious, since it provides polynomial scaling. To make sure that our implementation was correct we tested the Python code on randomly-generated networks of increasing size, and tested the validity of the solutions and the average time it takes to get them. The networks were generated using the Barabasi-Albert model which is known to create networks similar to those in real life [13]. The basic idea of the model is that it starts with a small set of fully connected nodes that is constantly growing, and the probability of creating an edge with a new node is larger the more neighbors an old node has. This results in hub-like networks, with a few nodes (the "hubs") being directly or closely linked with all others, making shortest-path lengths quite small. While this model is probably too simple to represent a real city, it will only be used to test the validity of the solutions and the efficiency of the algorithm.

Figure 9 shows the average computational time required to find a solution as the number of nodes increases. Note that a given random network of size 30 does not necessarily have less odd nodes than one of size 20, but since we are averaging over different tries we can expect the number of odd nodes to increase proportionally to the network size, on average. We fitted the obtained points to a third degree polynomial and saw that it fits quite well with an R^2 almost equal to one. This means that the scaling of the algorithm is polynomial and behaves as we expected. If we were to use the entire map of Lappeenranta with probably a few hundred of odd nodes

[4]In this case, brute-forcing means to test all possible combinations of odd node pairs.

we would expect the algorithm to finish in between a few seconds to less than an hour, which is quite good.

5.3 Extension to two-way roads

In general, extensions of the CPP such as the ones described in Section 2 are NP-hard and need heuristic tools to be solved. Our heuristic algorithm is described in Section 6, while more complex ones involving genetic algorithms can be found in [2, 1, 3]. Though there is one variation that can be solved in polynomial time with the classical algorithms, and that is Problem 5: two-way streets.

The idea is to extend the previous work to *multigraphs*, which are defined as graphs $\mathcal{G}_m$ that are allowed to have more than one edge for the same pair of nodes (and same direction, if the graph is directed). A two-way street is represented by a double edge with the same weight. To solve this problem we first transform $\mathcal{G}_m$ into a normal graph $\mathcal{G}$ with no multi-edges, and then apply Algorithm 2. The transformation follows these steps:

1. All nodes belonging to a multi-edge are multiplied, but without any of the original connections. For example, if there are two nodes A, B connected by a double edge then we create copies A', B' with no edges.

2. An edge is created between every copied pair, with a weight equal to the original edge.

3. Every copy is linked with its original node, with an arbitrary weight several orders of magnitude lower than any in the original graph

Figure 10 shows the application of this method to a simple multigraph with only one double edge. The cost of travelling from one node to its duplicate is almost negligible, so in practice it looks as if they were the same one. The transformation to a normal graph makes it possible to apply the algorithms described before, allowing us to get a (CPP) tour T. But T is defined for transformed graph $\mathcal{G}$, to obtain the multigraph tour T_m

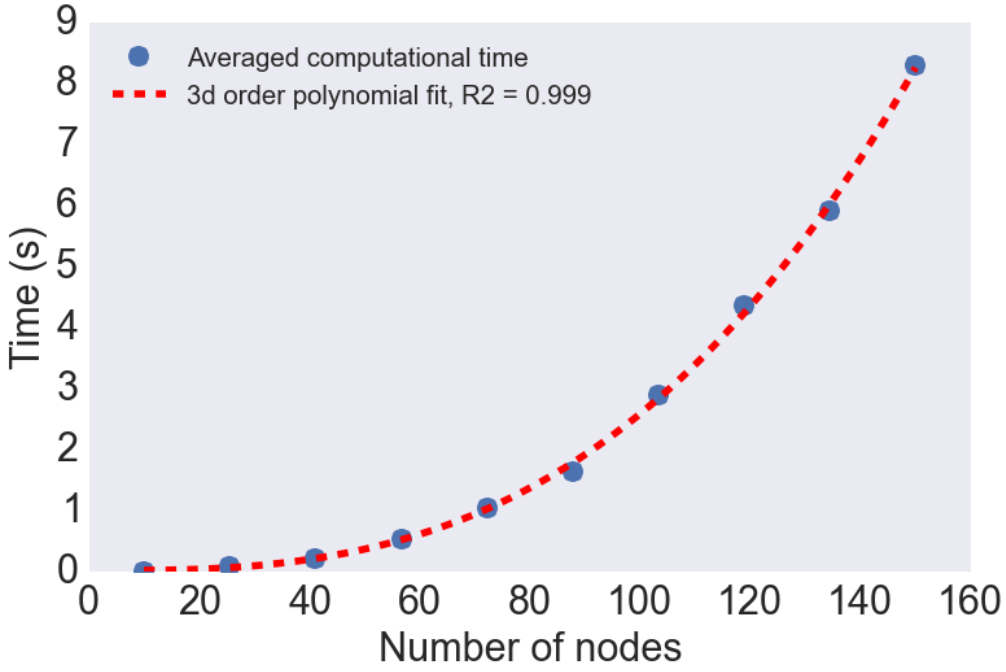

Figure 9: *Computation time as a function of network size. A third-order polynomial curve fits the data almost perfectly ($R^2 = 0.999$).*

we rename all the duplicate nodes (marked with ′, ″, etc.) to their original names and eliminate the crossings between a node and its copy. This way we find the optimal CPP path for the original multigraph $\mathcal{G}'$. As an example, $T = (A, B, B', C', C, B, A)$ would become $T_m = (A, B, C, B, A)$.

We tested this algorithm in the high-priority road map of Lappeenranta assuming all roads were two way. In this case, the resulting multigraph is eulerian (since all nodes have even degree) and so the optimum path is simply doing an eulerian path twice. The minimum distance required to cover all edges is then the sum of all distances multiplied by two. Using the modifications described above we effectively recovered this result, obtaining a path of total distance 48.471 km, which is twice the sum of all edges (24.2355 km)

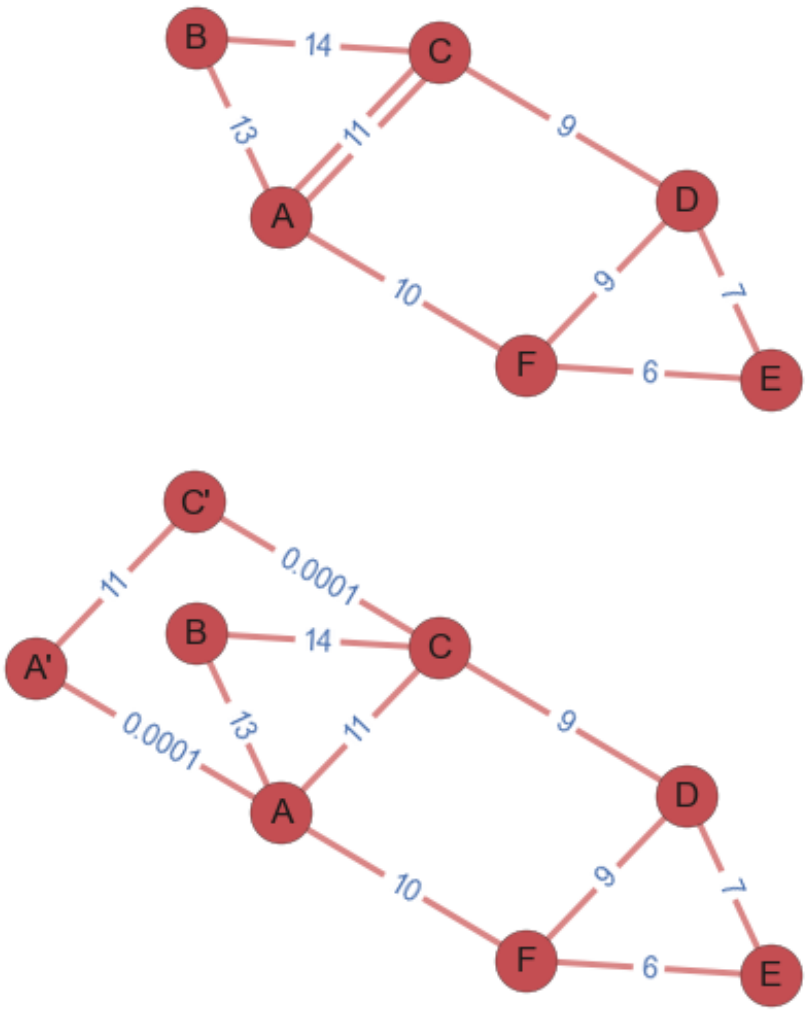

Figure 10: *Original multigraph (top) and its transformation to normal undirected graph (bottom). The multi-edged nodes A and C are duplicated.*

Algorithm 6: Semi-Eulerian tour

Input : An undirected connected weighted semi-Eulerian graph $\mathcal{G}' = (V, E)$. Starting and ending at nodes n_{o1} and n_{o2}.

Output: An open (CPP) tour

$T = (n_1 = n_{o1}, e_1, n_2, e_2, \ldots, n_l, e_l, n_{l+1} = n_{o2})$

Init : Start and End pair $p = (n_{o1}, n_{o2})$

$T \leftarrow (n_1)$

$n_{current} = n_{o1}$

repeat
  neigh $\leftarrow$ neighbors from $n_{current}$;
  repeat
    FoundNext $\leftarrow$ `True`;
    $n_{next} \leftarrow neigh_1$;
    if $n_{next} == n_{o2}$ && n_{o2} *has only one neighbor* **then**
      Move $neigh_1$ to the end of $neigh$ ▷ If $neigh = (n_3, n_5, n_6)$ then make $neigh \leftarrow (n_5, n_6, n_3)$;
      FoundNext $\leftarrow$ `False`
    else if $\hat{\mathcal{G}} = \mathcal{G} - (n_{current}, n_{next})$ *is disconnected* **then**
      Move $neigh_1$ to the end of $neigh$;
      FoundNext $\leftarrow$ `False`
  until *FoundNext* == `True`;
  $e_{next} \leftarrow (n_{current}, n_{next})$;
  $T \leftarrow$ append$(T, (e_{next}, n_{next}))$;
  Remove e_{next} from $\mathcal{G}$
until $n_{current}$ *has no neighbors*;

6 Solution 2: Stochastic Algorithm

6.1 Motivation

In Section 5 we tried to solve the snow plow optimisation problem using classical algorithms, which are able to find the optimum exactly. The problem with these algorithms is that even without looking at priorities, variations in the number of machines, different road sizes or other parameters, the complexity is very high. Our first model, covering parts of Lappeenranta's streets, allowed us to find an optimal solution covering all streets at least once (or twice).

But as soon as other parameters come into play, the complexity of these algorithms gets out of control (at least for a one week project).

Facing these problems, we saw the need to find a simpler and more customizable method of finding optimal or nearly optimal paths. We have devised a greedy algorithm that, generally speaking, attempts to find a path by estimating the next vertex step by step. This decision is made stochastically, by considering penalties (lower probability) for "less valuable paths." This approach we call the "penalty scout" method.

6.2 Finding a Penalty Scout Tour

Algorithm 7: Applying penalty to edges

Input : Graph $\mathcal{G} = (V, E)$, previous walking path W, neighbor node V_x

Output: Value of penalty for this edge

Init : Penalty $P = 0$

1 if V_x causes a U-turn : increase penalty P

2 if E_x which connects current position with V_x has been visited already: increase penalty P

3 if E_x has low priority: increase penalty $P \cdots$

4 return P

Algorithm 7 shows the idea of associating a value (in this case a penalty value) to every candidate vertex to be chosen next. The benefit is this approach is its flexibility — it is easily modifiable when additional parameters and constraints are used, and the algorithm is easy to tweak and fine-tune. Additional parameters could be length, number of visits, right/left turns, u-turns, priorities etc.

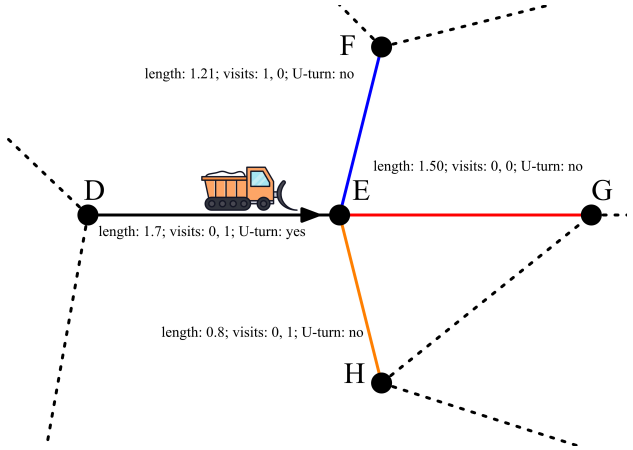

Figure 11: *How the Generate Walk Algorithm 8 "sees" the graph at each step.*

To choose each vertex, we implemented a `nextvertex()` function. This function creates a list of possible next points and uses the penalty algorithm 7 to assign each vertex a probability, and then generates a random number to choose the next point. In this way we reduce the probability of choosing directions that have already been plowed. But as we do not know if a candiate direction could be better in the long run, we do not want to reduce the probability to zero. Therefore, with this decision making protocol, we can be sure that our walk makes sense — preference is given to roads that are not yet plowed, and we do not make the same decision each time a vertex is re-visited.

Now we know how to choose the next point. If we put a penalty on multiple visits (which would lead to very small probabilities for any path that re-visits any one street many times), then we can be sure that we will cover all paths. Using the `nextvertex()` function, a complete tour can be generated with the walk algorithm 8.

Algorithm 8: Generate Walk

Input : Graph $\mathcal{G} = (V, E)$, starting (and ending) node W_0, and fitness function F

Output: Walk as a list of Nodes W, and a fitness of the path

Init : fitness $fit \leftarrow 0$, list of edges to cover $edgelist$, current location $loc = W_0$, $Walk = [W_0]$

1 **repeat**
2 Choose next vertex $V_k + 1$ using `nextvertex`
3 $edge = (loc, V_k + 1)$
4 Increase fitness $fit+ = F(edge)$
5 Perform step $\rightarrow loc = V_k + 1$
6 if $edge$ has not yet been visited, remove $edge$ from $edgelist$
7 **until** $|edgelist| = 0$ *and* $loc = W_0$
8 append loc to $Walk$
9 return $Walk$ and fit

Note that the fitness function could include the time needed to traverse each edge (which could be different for roads that have already been plowed) or the length of each edge. As we prefer lower (better) fitness values, the fitness function in our case is the objective function in an optimization problem. To improve the performance of the algorithm, its important to consider the goal of optimization when adding a penalty to candidate vertices.

Now the idea is to try many different paths and choose the path with the best fitness value (e.g. the shortest length/shortest time path) as the optimum. Here we generate a walk using Algorithm 8 and compare it to the best walk generated so far. If the new walk is better, we delete the old one and save the new one as the "best so far". After some predefined number of iterations is reached, we stop the algorithm and return the best candidate found so far.

Clearly we cannot be sure that this algorithm will give *the* optimal solution, which would cover every edge at least once with the shortest

Iteration	Length (km)
3	40.85
69	36.48
1233	33.08
53871	31.43
169931	30.5275

Table 1: *Results of the Penalty Scout algorithm, using 1 snowplow, and covering every street only once.*

possible total length. However, on several example graphs where the optimal solution was known, our penalty scout algorithm found the optimum solution within only a few (usually around 1000) iterations. The test graphs had up to 10 nodes.

6.3 Results using Penalty Scout

Motivated by this success, we started working on the graph shown in Figure 5, which covers all priority 1 roads and some others with a total number of 24 nodes. The total distance of all edges sums up to 24.2355 km. However, since the graph is not Eulerian, this is not the length of the optimal path — some roads must neccessarily be traversed more than once. The length of the optimal path is 30.5275 km, and was obtained using the classical algorithms from section 5.

The penalty scout algorithm is easily modified for more complicated problems. However, the penalty variables need to fit the characteristics of the graph. When applied to the graph shown in Figure 5, the penalty scout algorithm produced the results in Table 1.

The final solution in Table 1, which was found after 169931 iterations, matches exactly with the optimal solution found using classical (CPP) algorithm. This shows that our idea for the penalty scout algorithm is able to find the optimum. Another important observation is that fairly good

Iteration	Length (km)
23	73.16
142	55.98
452	50.01
3131	44.73
5608	41.99
92125	41.18

Table 2: *Using Penalty Scout algorithm, 2 machines, cover every street only once, total number of iterations: 100000*

solutions (solutions that are close to the optimum) can be found within less than 100 attempts. Hence we can find a compromise between optimality and calculation time. For the graph in Figure 5, 1000 iterations of the penalty scout algorithm requires approximately 2 seconds on our hardware.

6.4 Modified Snowplow Problems

We where also able to modify Algorithm 8 slightly to compute an approximately optimal path for *two machines* snow plowing at the same time. The lengths shown in Table 2 are the aggregated distances covered by both machines.

This result shows that, if we take 2 machines to plow the graph in Figure 5, then each machine needs to drive about only 20.6 kilometers instead of 30.5 kilometers. Therefore, the streets of Lappeenranta will be plowed faster using 2 machines, and we are able to deliver a nearly optimum path. Computation for 2 machines takes slightly longer, because the *Generate Walk Algorithm* 8 has to compute 2 paths at each iteration.

Table 3 shows similar results in the case of three snowplows.

According to Table 3, a nearly optimal solution for the graph in Figure 5 using 3 machines simultaneously ends up in approximately 19.6 kilometers. This is worse than with 2 machines, because 2 machines are able to cover

Iteration	Length (km)
15	133.40
698	85.67
1464	72.38
11130	62.41
52757	58.84

Table 3: *Using Penalty Scout algorithm, 3 machines, cover every street only once, total number of iterations: 100,000*

the graph using 20.6 kilometers per machine. However, this could change for larger graphs. Therefore we are not only able to deliver nearly optimum paths for different machine numbers, we can also tell how many machines cover the path most efficiently from a cost-benefit point of view.

Now let us consider a further modification of the snowplow problem. Obviously a street is not yet fully plowed if it has been visited only once. We have to visit every street at least one once in *each direction* (assuming that all streets are two-way streets). This can be done by adding attributes to the graph that count the number of visits for each direction. Then the list of edges ($edgeslist$) in our *Generate Walk Algorithm* 8 needs to contain every edge *twice* (once for each direction). Note that the total distance that our snowplow needs to cover then doubles from $dist = 24.25$ to $dist = 48.5$, which we know from Section 5 to be the optimal solution to the CPP. Table 4 provides some results for this problem, produced using the Penalty Scout method.

Moreover we tried to find a path that minimizes the *total time* needed, assuming that the speed while plowing is 15 kilometers per hour, and that the speed on already-plowed roads is 40 kilometers per hour. If you are interested in these results, feel free to contact us at [14].

Machines	Iteration	Length (km)	Machines	Iteration	Length (km)
1	4	88.29	2	1	116.13
1	223	68.02	2	1956	78.37
1	437	59.12	2	2062	76.80
1	17090	53.00	2	25286	63.92
–	–	–	2	90669	60.24

Table 4: *Using Penalty Scout algorithm, 1 and 2 machines, covering both directions, total number of iterations: 100,000*

Summary of this section

The methods we have provided can:

- deliver optimal or near-optimal results.
- deliver solutions for multiple machines covering the graph at the same time.
- deliver paths that visit high-priority roads first.
- decide how many machines can plow all streets with reasonable costs.
- change optimization rules to fit customers needs.
- let the customer choose between computation time and other measure of performance.

More tests are needed for larger scale graphs. It is also important to mention that using the Penalty Scout Algorithm 8 cannot tell us whether a solution is in fact exactly optimal.

7 Discussion and Further Work

Discussion. This project was done during the ECMI Modelling Week 2017 in Lappeenranta. Together, we where able to provide software that is able to solve our problem *exactly and efficiently*, at least in the simple case of one machine and with no priorities, as described in Section 5. Moreover, we invented a new method using heuristics that is able to provide optimal or nearly optimal solutions for more sophisticated problems. These problems include priority consideration, multiple snowplows, and visiting every road at least once in *both* directions. We called this method *Penalty Scout Algorithm*, and is described in Section 6. As we where provided with a list of street names only, we devised a method for turning this data set into a usable graph that models the Lappeenranta road retwork. We used this graph as input for each of our solution methods.

Of course, a complicated problem cannot be solved completely in a mere five days, and there is much to be done by way of further work. Let us mention some promising areas:

- Since the efficient analytical method (Section 5) is not able to solve the more complicated problems, more time could be spent in trying to find ways to solve these without heuristics.
- Our stochastic method requires a lot of computation time (compared to the classical method) to provide the same solution. The time needed may increase exponentially for larger graphs. Therefore a more efficient way of finding the optimum needs to be implemented (perhaps by using methods such as swarm intelligence or genetic algorithms).
- We were not able to construct the full graph of the original problem (as described in Section 4.1), because the free tier of the Google Maps API requests did not cover our needs. Ways to reduce API requests to the minimum to save money for generating the distance matrix out of the street names need to be found. Alternatively, with more development time a solution using *OpenStreetMap* [15] could be considered.

An interesting observation that we made while solving the snowplow problem with our algorithms is that the optimal paths tend to be somewhat unintuitive. This suggests that human intuition alone is quite unlikely to find the optimal path. We observe that paths devised by humans (specifically, members of our team) tended to cover the roads close to the current position first. We created a video, showing the optimal solution found for the simplest case (one machine visiting every road once) and uploaded it to our GitHub repository, which is open to the public. The link to the repository is https://github.com/GandalfSaxe/ecmi2017. Moreover, we saw that both of our solution attempts found a solution with the same total length, but different paths.

Further Work. To fully apply our solutions and solving techniques, there is still plenty of work to be done.

- The Google API script needs to be optimized according to the number of API requests and possible use of OpenStreetMaps could be investigated.
- The penalty scout algorithm needs to be optimized for lower computational time. Maybe a different setup for the penalty parameters would improve the quality of generated walks.
- Both scripts could be combined to get a piece of software that takes an address and a list of streets as an input and returns an optimal path.
- The solutions are currently displayed as a list of points (intersections), but a driver needs a list of driving instructions to follow this path. Automatically generating driving instructions would be a worthwhile development.
- With reduced computational time and more data, we could try to develop a dynamic optimization approach, so that driving instructions for each driver can be updated on-the-fly according to current fall of snow and traffic conditions.

- A smartphone application could be developed that displays the driving instructions to the drivers.

References

[1] T. Rao, S. Mitra, J. Zollweg, and F. Sahin. "Computing Optimal Snowplow Route Plans Using Genetic Algorithms". In: *Systems, Man, and Cybernetics (SMC), 2011 IEEE*. IEEE. 2011, pp. 2785–2790.

[2] T. Rao, S. Mitra, and J. Zollweg. "Snow-Plow Route Planning Using AI Search". In: *Systems, Man, and Cybernetics (SMC), 2011 IEEE*. IEEE. 2011, pp. 2791–2796.

[3] T. Rao, S. Mitra, and J. Zollweg. "Route Allocation for Multiple Snowplows Using Genetic Algorithms". In: *Systems, Man, and Cybernetics (SMC), 2012 IEEE*. IEEE. 2012, pp. 29–34.

[4] J. F. Campbell, A. Langevin, and N. Perrier. "Advances in Vehicle Routing for Snow Plowing". In: *Arc Routing: Problems, Methods, and Applications* 20 (2014), p. 321.

[5] L. Euler. "Solutio Problematis ad Geometriam Situs Pertinentis". In: *Commentarii Academiae Scientiarum Petropolitanae* 8 (1741), pp. 128–140.

[6] J. Edmonds and E. L. Johnson. "Matching, Euler Tours and the Chinese Postman". In: *Mathematical Programming* 5.1 (1973), pp. 88–124.

[7] J. Edmonds. "Paths, Trees, and Flowers". In: *Canadian Journal of Mathematics* 17.3 (1965), pp. 449–467.

[8] J. Edmonds. "Maximum Matching and a Polyhedron with 0,1-Vertices". In: *Journal of Research of the National Bureau of Standards B* 69.125-130 (1965), pp. 55–56.

[9] *Wikipedia Entry: Matching (graph theory)*. 2017. URL: https://en.wikipedia.org/wiki/Matching_(graph_theory).

[10] Z. Galil. "Efficient Algorithms for Finding Maximum Matching in Graphs". In: *ACM Computing Surveys (CSUR)* 18.1 (1986), pp. 23–38.

[11] J. van Rantwijk and N. Mancuso. *Maximum-Weight Maximum-Cardinality Matching NetworkX Code*. Ed. by A. Hagberg, D. Schult, and P. Swart. Copyright (C) 2011 by Nicholas Mancuso. All rights reserved under BSD license. 2011. URL: https://networkx.github.io/documentation/development/_modules/networkx/algorithms/matching.html.

[12] *ECMI 2017: Snowplow Route Optimization*. 2017. URL: https://github.com/GandalfSaxe/ecmi2017.

[13] A.-L. Barabási and R. Albert. “Emergence of Scaling in Random Networks”. In: *Science* 286 (1999), pp. 509–512.

[14] C. Assmann. *Additional Snowplot Results*. URL: carlassmann@me.com.

[15] M. Haklay and P. Weber. “OpenStreetMap: User-Generated Street Maps”. In: *IEEE Pervasive Computing* 7.4 (2008), pp. 12–18.