FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Artistic Metro Maps

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Oliver Pilizar
Matrikelnummer 01525787

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: PhD Hsiang-Yun Wu

Wien, 9. März 2020

Oliver Pilizar	Hsiang-Yun Wu

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Artistic Metro Maps

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Oliver Pilizar
Registration Number 01525787

to the Faculty of Informatics
at the TU Wien
Advisor: PhD Hsiang-Yun Wu

Vienna, 9th March, 2020

Oliver Pilizar | Hsiang-Yun Wu

Erklärung zur Verfassung der Arbeit

Oliver Pilizar

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 9. März 2020

Oliver Pilizar

Danksagung

Zuallererst möchte ich mich bei meiner Betreuerin PhD Hsiang-Yun Wu für die durchgehend nützlichen Informationen und Hinweise bedanken. Zusätzlich möchte ich mich bei Sebastian Kürten, dem Erzeuger von OpenMetroMaps (dem Basis-Framework dieser Arbeit), nicht für dessen Bereitstellung, sondern auch die ausführliche Antwort auf meine Frage, bedanken. Schließlich möchte ich mich auch bei meiner Familie für die Möglichkeit dem Studium nachzugehen bedanken, aber auch bei meinen Freunden, die die Zeit, die ich damit verbracht habe, umso besser gemacht haben.

Acknowledgements

First of all I want to thank my advisor PhD Hsiang-Yun Wu for supporting me by providing useful information and advice throughout my time working on the thesis. Additionally I want to thank Sebastian Kürten who is the creator of OpenMetroMaps, the main framework used in this thesis, for not only providing the framework, but also answering my question so thoroughly. Finally I want to thank my family for providing me with the opportunity to study, and also my friends who made my time doing so even better.

Kurzfassung

Die Erzeugung von visuell ansprechenden U-Bahn Netzplänen benötigen normalerweise einen Designer und eine Menge Aufwand. Und obwohl die automatische Erzeugung von regulären U-Bahn Netzplänen mittels diverser Methoden bereits mehrfach gemacht wurde, haben keine den Fokus auf den artistischen Aspekt gelegt. Um den Prozess für den Designer einfacher zu machen stellt diese Arbeit eine Methode vor die automatisch Netzpläne erstellt, welche dann entweder gleich so verwendet werden können, oder als Basis für den weiteren Designprozess dienen. Das Ziel dieser Arbeit ist es eine Methode zu finden und darauf aufbauend einen Prototypen zu erzeugen, der U-Bahn Netzpläne in beliebigen Formen generiert und dazu nur einen Plan und eine Kontur als Eingabe benötigt. Zusätzliche Parameter sollen, falls erwünscht, Anpassungen ermöglichen. Das prinzipielle Konzept ist zuerst den Plan und die Kontur für die weiteren kleinsten Quadrate Berechnungen vorzubereiten, welche dann den Plan in die gewünschte Form bringen und ihm das typische Aussehen eines U-Bahn Netzplans geben. Der Algorithmus wurde auf zwei verschiedene Pläne und sieben unterschiedliche Formen angewandt, um dessen Resultate darzustellen. Diese zeigen, dass die vorgestellte Methode in der Lage ist derartige artistische Netzpläne in beliebigen Formen zu erzeugen, weitere Anpassungen eines Designers um den Netzplan zu finalisieren sind allerdings notwendig.

Abstract

The creation of visually pleasing artistic metro maps usually requires a designer and a lot of effort, and while the automatic generation of regular metro maps has been done via several methods, none focus on the artistic aspect. To make the process easier for designers this thesis introduces a method that automatically creates maps that can either be used as they are, or used as baseline for the future design process. The goal of this thesis is to find a method and based on that create a prototype that generates metro maps in arbitrary shapes that simply requires the map and contour as input. Additional parameters are supposed to allow a user to make adjustments if so desired. The general approach is to first prepare the map as well as the contour for the following least squares calculations that reshape the map in a way to fit the contour and then create the look of a typical metro map. To test the algorithm and showcase its results it is applied to two different maps and seven different shapes. These results indicate that the introduced approach is capable of creating metro maps in arbitrary shapes, but need further adjustments by a designer to finalize the map.

Contents

CHAPTER 1

Introduction

Metro maps are a useful tool to navigate through and visualize metro networks. Usually such maps are drawn inside a rectangular shape, whether that be on public information displays or inside tourist leaflets. However, that kind of shape is not always beneficial. It would for example be a lot nicer to find such maps in a representative shape for the city, possibly inside a tourist leaflet or as a souvenir to take home. Furthermore it would make it easier to direct a reader's attention towards a map, whether that be inside a magazine, a newspaper, on a website or on a public information display. Lastly it would be a lot easier to fit a map inside any given context, if the shape were adjustable. To create such a map a person would usually have to create it from scratch, planning out the general positions of the metro's lines, carefully placing every station and thinking about how the edges would connect, so the map ends up planar.

It would therefore be desirable to have an algorithm that converts any metro map into one that fits any given shape at least to the extend of providing meaningful help to whoever is creating it. Since it is important to not only fit the shape, but also keep the general look of a metro map, this property would also have to be considered for the algorithm. And while such metro map generating algorithms exist, none do, that also allow the creation of arbitrary shapes.

This does however introduce one of the main challenges of this thesis. The part of the algorithm that creates the metro map layout needs to not destroy the generated shape, while the part that generates the shape needs to allow enough space for the map to deform into a metro map like layout. Since those two properties are contradicting to a certain extend, it is important to find a way for both of them to be fulfilled, without having to introduce computationally intensive repetitions or checks.

The objective of this thesis is to create an algorithm that assists designers in creating artistic maps in arbitrary shapes. This should not only make the task easier and faster, but also possibly improve the results or at the very least show a different possible solution

which might introduce new ideas. This means that the algorithm should be able to take any metro map as input as well as a contour and produce a result in real-time. Besides a number of parameters to adjust the result the algorithm is supposed to work fully automated, meaning that no further influence after providing the initial input shall be required. In case of a schematic map the result is supposed to keep its metro map look, while in case of a non-schematic map that look is supposed to be created. The resulting map is then also supposed to fit the given contour without any drastic errors, so only small adjustments need to be made by the designer.

To achieve the desired results the algorithm is split into several parts, first adjusting the scaling of the map and the contour to allow for a better process and the using constraints calculated in a least squares sense to reshape the map and create the correct look.

The result of the thesis is then a prototype tool that converts the map given as an xml-file and the contour, given as a list of nodes, into a visualized map that fits the above mentioned criteria. Furthermore the important step of adjusting the shape is visualized to give the user a better idea of what is happening. This is supposed to help the user with changes to the parameters, which for example describe to what extent certain properties of the map should be taken into consideration.

In summary the contribution to this problem includes:

- Finding a way to fit a node-link diagram (the metro map) into a given contour
- Finding a way to create a metro map like look by replicating typical characteristics, that works well in collaboration with the previous point
- Creating a prototype, which allows the testing and adjustment of the approach (as a plug-in of OpenMetroMaps [OMM])

1.1 Structure of the thesis

The thesis is separated into several chapters, each focusing on different topics. While this chapter provides some general information about the thesis, the next one “Chapter 2 – Related Work” introduces different approaches for various steps of the algorithm. “Chapter 3 – Methodology” gives a brief introduction into the used methods throughout the process whereas “Chapter 4 – Implementation” explains the actual implementation of the algorithm via small code snippets, text and an accompanying visual example. “Chapter 5 – Results” then demonstrates several results, both visually and statistically regarding their computational time. Lastly "Chapter 6 - Conclusion & Future Work" provides a short conclusion and information concerning possible improvements and methods to do so.

CHAPTER 2

Related Work

The following chapter introduces some of the related work and describes why certain characteristics were used in this thesis.

First it should be stated that the two main steps, the Deformer phase, which is supposed to create the metro map look and the Reshaper phase, which is supposed to reshape the map into the desired contour are treated separately. Deformer phase throughout this paper is referencing the least squares equations necessary to express the different criteria to create a metro map look in combination with the cg-method which combines and solves these equations. The Reshaper phase, similar to the Deformer phase, stands for the least squares approach and the cg-method that result in a map layout that fits a given contour.

Since the Reshaper phase takes place first during the algorithm it will also be handled first here. The basic idea is to change the appearance of the map without destroying the topology. There are various different approaches to achieve this, some of which are briefly listed in the work of Gibson and Mirtich [GM97] where they are split into several categories. The idea is to either rely on physical models that for example use the principle of springs or magnetic fields to find an optimal position while non-physical methods use geometric properties.

The work by Gibson et al. [GFV13] compares a variety of force-directed layout algorithms, which are physical models, and compares them regarding the number of nodes they can handle, the time it takes to compute the result and several other criteria. Since for the Reshape phase the most important aspects are the handling of a certain number of nodes usually present in metro maps, interactivity when it comes to the time the algorithm takes and the prevention of intersection several of the listed algorithm are already not suitable. And even after finding one that satisfies the mentioned criteria, the most important one, the predefined form of the end result is not taken into consideration. Usually these kind of algorithms produce clusters, which is not desired here meaning that

further adjustments have to be made. And while such adjustments are certainly possible the process would potentially be rather unintuitive or complex to formulate for arbitrary shapes.

A different approach is using the similarity between the problem of shaping a 2D graph and manipulating a 2D image, since it is possible to simply view the graph as a 2D image rather than a data structure.

Several approaches exist to manipulate the form of images. One of them is the Free-form deformation (FFD) [T.D14] which overlays overlays the image with a coordinate grid. Whenever a grid cell is then manipulated all other cells move accordingly. The approach by Setaluri et al. [SWM+14] also uses a grid overlay to create deformations in combination with non-linear constraints to express the desired result. Non-linear constraints have, as mentioned in the paper itself, the advantage of being able to express more advanced properties, however they do so by trading away computation speed. While this approach would probably work and be fast enough for interactivity, the creation of the grid is a step that can be avoided by using different approaches more suitable to the manipulation of a graph.

On the other hand several approaches exist that use skeletons [LCF00] to more accurately represent deformations of for instance human legs. Since the image to shape is however just a graph, and the result of the Reshaping phase is going to be altered in the step afterwards anyway such specific physical attributes are not necessary and would be destroyed afterwards regardless.

Similarly to the method used in this thesis Weng et al. [WXW+06] also use a least squares optimization. A number of points is inserted into the 2D shape to construct an inner graph which the least squares terms are working with. However, the equations are non-linear, which would usually result in higher computation times. However, an approach to circumvent this problem is introduced to remain interactive. Very similarly to this the approach by T. Igarashi et al. (ARAP) [IMH05] first creates a triangulation inside the 2D shape and then uses least squares to adjust it. It does so however, by formulating the minimization problem via several linear equations to gain speed. The advantage of this approach compared to the one by Weng et al. does not lie in the least squares optimization, but in the fact, that the nodes of the metro map can be used as nodes for the triangulation, meaning that no new points have to be introduced. Finally T. Igarashi and Y. Igarashi show a method [II09] that uses the same approach as ARAP, but apply the constraints on the edges instead of the faces of the generated triangles. This approach will serve as a guideline for the Reshaper phase of this thesis' algorithm.

Regarding the generation of metro maps several methods have been created over time. These vary from physically based ones like a spring algorithm to least squares optimizations. Hong et al. [HMdN04] introduce different approaches based on the GEM algorithm, the PrEd algorithm and a magnetic spring algorithm. They mostly work on a preprocessed graph, that has nodes with a degree of 2 removed to allow for easier movement.

Stott et al. [SRMOW11] on the other hand use a hill climbing algorithm to combine the typical criteria suited for metro maps. In addition to that clustering methods are introduced to not only move each node separately in an attempt to avoid local minima.

Since there is quite a large variation in attempts and implementations of such there are also a couple of studies showcasing the advantages and disadvantages of these variations. One such study by Nöllenburg [Nöl14] first introduces a set of ten design rules and several approaches made over the past years to achieve those. It also includes label positioning techniques, which some of the previously mentioned methods also do.

Another study by Wu et al. [WN19] focuses more on the choice of approach used for a specific purpose. The metric used is therefore a combination of computation time and local vs. global optima of the result. Two of the fastest methods mentioned in the study are the ones from the papers "Focus+Context Metro Maps" [WC11] and "Interactive Metro Map Editing" [WP16]. Both of them have the disadvantage of falling onto the local optima side of the spectrum, which is to be expected from fast algorithms, they do however still produce good results. Additionally both of them use the same principle, which is the least squares optimization, which coincidentally is also the approach chosen for the Reshaper phase for this thesis. These papers will therefore serve as a guideline for the Deformer part of this thesis.

Another approach that uses least squares is introduced by Lutz [Wür14], which uses a linear approach with a single iteration to gain speed, rather than using an iterative solver like the one used in this thesis, that allows a visualized step by step transformation of the map.

CHAPTER 3

Methodology

This chapter introduces some of the most important techniques, as well as a short explanation of the notation. In general the structure of this chapter is the following:

- Least Squares

  This section serves as an introduction to the basic concept, that is used to solve the later introduced constraints. Furthermore it provides an explanation of the basic notation.

- Method of Conjugate Gradients

  In this section the actual method of solving of the constraints is introduced

- Delaunay Triangulations

  This method plays an important part in one of the main steps of this paper's algorithm, the Reshaper, and is therefore shortly introduced here.

- Polygon Calculations

  Since over the course of the algorithm several smaller calculations are necessary, this section serves as an overview of what they might be and where they are explained in more detail.

- Combining the Map and the Hull

  Since the map and the contour have to be brought into some kind of relation to influence each other this section explains how and why that happens.

- Fitting the Contour – Reshaper

  This section introduces one of the main parts of the algorithm, by conveying the main idea, its prerequisites and the formulas that make up the process.

- Creating the Metro Map – Deformer

  This is the second and also last main part of the algorithm. Since it consists of several smaller parts each of them is given a designated mini-section to explain the idea and the formulas that are used to realize those ideas.

3.1 Least Squares

One method to solve linear equations is using least squares [Sel13]. It is used to find an approximate solution for overdetermined systems, which means a number of equations that has less unknown variables than equations. In matrix form this would be a "tall" matrix, meaning more rows than columns. Such a system is written as

$$\boldsymbol{Ax} = \boldsymbol{b} \tag{3.1}$$

where A represents a matrix and x and b represent vectors. Throughout this paper matrices will be denoted in bold upper-case letters, while vectors will be denoted in bold lower-case letters. Such "tall" matrices will be used throughout this paper, since there is going to be a row for each edge/node in the graph, which is therefore going to be more than the number of unknown variables. These matrices will be part of linear equation systems that represent certain constraints of the map, which need to be solved.

Since it is necessary to add weights to these systems in the here introduced algorithm it is necessary to include those here as well. The slightly changed term now becomes

$$\boldsymbol{WAx} = \boldsymbol{Wb} \tag{3.2}$$

where W represents the weight matrix, which is a diagonal matrix that sets the weight for every equation. This weight can be chosen freely meaning it does not have to be for example between 0 and 1 or add up to anything. It is however still important to test out which weight suits a certain criteria the best, since as will be shown later, multiple of these terms will be used together. It is therefore important to find a good balance between the weights, since the outcome can drastically change depending on what was chosen.

3.2 Method of Conjugate Gradients

The method of conjugate gradients (cg-method) [HS52] is used to solve a system of linear equations, such as the ones introduced in the previous section. It is an iterative method, that is applicable to sparse systems, such as the ones that will be created during the Reshaper and Deformer phase of the algorithm. What this means is that the matrix will have a lot more entries that are 0 than anything else. The basic steps are as follows:

1. Initial step

   During this step the first estimate x_0 is chosen and the corresponding residual r_0, as well as the direction p_0 are calculated.

2. Iterative step

   Based on the initial values, or if there has already been an iteration, based on the previous values, the next values for the estimate, residual and direction are calculated. If during any iteration the error falls below a set threshold the iteration stops and the current estimate becomes the result. The same applies if the number of iterations reaches a set maximum.

To make the following equations work, matrix A needs to be symmetric and positive definite. Since the systems that will later on be solved via this method are neither of those it is necessary to prepare them as follows:

$$\boldsymbol{A_{new}} = \boldsymbol{A}^T\boldsymbol{A} \tag{3.3}$$

$$\boldsymbol{b_{new}} = \boldsymbol{A}^T\boldsymbol{b} \tag{3.4}$$

The initial step to calculate the error and residual is rather simple. Matrix A is multiplied with the initial estimate and the difference to the expected result b is taken. The iterative step follows the equations shown in 3.5 [HS52] where x stands for an estimate, r for a residual and p for a direction.

$$\begin{aligned} \alpha_i &= \frac{|r_i|^2}{(p_i, Ap_i)} \\ x_{i+1} &= x_i + a_i p_i \\ r_{i+1} &= r_i - a_i A p_i \\ \beta_i &= \frac{|r_{i+1}|^2}{|r_i|^2} \\ p_{i+1} &= r_{i+1} + b_i p_i \end{aligned} \tag{3.5}$$

3.3 Delaunay Triangulation

The Delaunay triangulation [CDS12] is a commonly used method of creating meshes out of a set of finite points. While working with 2D points the Delaunay triangulations offers a big advantage, in that it maximizes the minimum angle of the generated mesh. Since the paper by T. Igarashi et al. [IMH05], on which the idea of the Reshaper is based on, mentions that the best results are achieved with near-equiliteral triangles this property of the Delaunay triangulation comes in handy. Especially so, since the points that will be used to create the mesh depend on the metro map that is used as input, meaning the distribution of points is not equal by any means and any way to generate semi-equal triangles is valuable. The created mesh will later on be necessary as input for the Reshaper phase.

3.4 Polygon Calculations

During the introduced algorithm, several calculations regarding polygons will be performed. The reason for that is that during some steps the graph's nodes will be treated as points of a polygon to find for example the convex hull, points on the graph's contour with a certain distance separating them, etc. Since there is no one method of doing so, each of these calculations will be explained when they come up while describing the algorithm's implementation.

3.5 Combining the Map and the Hull

To move the map in accordance to the contour they have to be put into some kind of relationship. Since the contour is the end goal of the Reshaper it is necessary to create something that later on reflects the contour, which in this case is the hull of the graph. The hull is created slightly larger than the minimal one to avoid problems during movements of the map's nodes. After the hull's creation the number of nodes that form its shape is adjusted to be equal to the contour's number of nodes. This will later on allow the movement of each node to its according position on the contour. Since at this point the map and the hull are still not connected in any relevant way both of them are combined into a single graph. This graph, after the Delaunay triangulation has been applied to it, will serve as input of the Reshaper, which will then actually move the hull alongside the map into the shape of the contour.

3.6 Fitting the Contour - Reshaper

This is the first big step of the algorithm. The goal for this step is to adjust the form of the metro map into the desired shape. The basic idea is to move certain nodes of the graph and move the rest accordingly to them. Since it is unknown where any stations are supposed to go and only the outer contour is known, the moving nodes will be the hull that is encapsulating all of the metro nodes. To move all nodes accordingly to the movement of a few, the prerequisite for the graph is that it needs to be turned into a triangle mesh, which is what the previously mentioned Delaunay triangulation is used for. The basic idea is to move certain nodes, so called handles (the nodes that build the hull), and then obtain the new coordinates of every other non-handle node by trying to minimize the distortion of the mesh's triangles. Since there is no way to describe that distortion as a linear function, a non-linear one has to be approximated by several linear ones. For this algorithm the distortion is not focused on the triangle's faces, but on the edges connecting them, which does not have significant impact on the result, but simply

changes the form of the functions.

$$\Omega_r = \Omega_{r1} + w * \Omega_{r2} \tag{3.6}$$

$$\Omega_{r1} = \sum_{\{i,j\}\epsilon E} |(\boldsymbol{v_i'} - \boldsymbol{v_j'}) - (\boldsymbol{v_i} - \boldsymbol{v_j})|^2 \tag{3.7}$$

$$\Omega_{r2} = \sum_{\{i\}\epsilon E} |\boldsymbol{v_i'} - \boldsymbol{C_i}|^2 \tag{3.8}$$

Ω_r combines the two constraints into a single function, while Ω_{r1} minimizes the edge deformation by trying to keep them similar to their previous state and Ω_{r2} minimizes the distance of the handle nodes to their designated positions C_i. The attached weight w to Ω_{r2} is necessary to adjust how serious the handle positions should be considered. While the weight will start out low during the course of the algorithm it will gradually increase until the desired positions are reached. Lastly $\boldsymbol{v_i}$ and $\boldsymbol{v_j}$ represent the nodes connected by an edge while $\boldsymbol{v_i'}$ and $\boldsymbol{v_j'}$ represent the same nodes after they have been moved. [II09].

3.7 Creating the Metro Map - Deformer

After the metro map has been reshaped into the desired contour, the typical layout for such a map has been destroyed. To restore the metro map feeling several of the layout's properties have to be changed. First of all, metro maps usually have edges of similar lengths. Secondly the angle of neighboring edges is usually quite high, if not the maximum, which in case of 2 edges would we 180 degree or a straight line, in case of 4 edges 90 degrees and so on. Thirdly the layout is usually octilinear, meaning that there are at most 8 neighboring edges and angles in an interval of 45 degrees. To achieve each of these properties they will be expressed as a constraint in the form of an energy term and later on solved in a least squares sense. The Deformer largely consists of 2 steps, each including the constraints just mentioned. First of all, it should probably be explained why there are two different steps involved, rather than just one. The reason is, that using octilinear constraints is very expensive since they are highly non-linear, so separating the two steps actually nets a better performance overall. That is calculating the smooth deformation first and then moving the edges to the closest position that results in an octilinear layout [WP16]. To provide an overview of the necessary individual constraints each of them is listed here with a short description of what they do. More detail about

each of them is then provided in the following sections.

$$\Omega_l = \sum_{\{i,j\}\epsilon E} |(\boldsymbol{v_i'} - \boldsymbol{v_j'}) - \boldsymbol{s_{ij}}\boldsymbol{R_{ij}}(\boldsymbol{v_i} - \boldsymbol{v_j})|^2 \tag{3.9}$$

$$\Omega_a = \sum_{\boldsymbol{v_i'}\epsilon \boldsymbol{V'}} \sum_{\{i,j\}\{i,k\}\epsilon N(i)} |(\boldsymbol{v_i'} - (\boldsymbol{v_j'} + \boldsymbol{u_{jk}'} + M_{jk}\boldsymbol{u_{jk}'}))|^2 \tag{3.10}$$

$$\Omega_p = \sum_{\boldsymbol{v_i'}\epsilon \boldsymbol{V'}} |(\boldsymbol{v_i'} - \boldsymbol{v_i})|^2 \tag{3.11}$$

$$\Omega_o = \sum_{\{i,j\}\epsilon E} |(\tilde{\boldsymbol{v_i}} - \tilde{\boldsymbol{v_j}}) - O(\boldsymbol{v_i'} - \boldsymbol{v_j'})|^2 \tag{3.12}$$

$$\tag{3.13}$$

Ω_l is the constraint necessary to achieve the regular edge length, Ω_a for the maximal angles, Ω_p is a positional constraint that restricts movement of nodes, as to not have them move too far and Ω_o creates the octilinearity

3.7.1 Smooth Deformation

The first step of this deformation is called "smooth deformation". It consists of two of the previously mentioned constraints, the regular edge length and maximal angles, in addition to a third one, a positional constraint. The reason for it is rather simple. Since a map already exists and the goal is to adjust that map, a general sense of similarity should be retained. That is keeping the edges in a topological sense the way that they are, which should not be altered anyways, but also the general positions of the nodes. While it is impossible to keep the exact positions, it is certainly desirable to keep them close to their origin. All of these constraints have different importance. Straight lines or four different edges going all in different directions is for example something that is very much desired, to make the map look more metro-map-like. Similarly a uniform length for the edges is something that is also rather important. Since the graph has already been reshaped, the geographical accuracy is already lost, and while the shape should stay roughly the same keeping the nodes close to their previous positions is in comparison much less important. These ideas should then be reflected when choosing the weights. All three of these constraints combined will then create new coordinates for each node, that will then be passed on to the next step, where the only remaining constraint, the octilinear layout, will be calculated.

Regular Edge Length

The edges of a typical metro map are usually about the same length. During the reshape phase it is very likely to have not only the nodes re-positioned, but also the edges rotated and changed in length. It is therefore important to recreate the property of same length edges. Compared to the other constraints it is however of relatively low importance, which is why the associated weight will be set relatively low. The general formula, or

energy term, for this constraint looks like this [WC11]:

$$\Omega_l = \sum_{\{i,j\}\epsilon E} |(\boldsymbol{v_i'} - \boldsymbol{v_j'}) - \boldsymbol{s_{ij}}\boldsymbol{R_{ij}}(\boldsymbol{v_i} - \boldsymbol{v_j})|^2 \tag{3.14}$$

$$s_{ij} = \frac{D}{|\boldsymbol{v_i} - \boldsymbol{v_j}|} \tag{3.15}$$

$$R_{ij} = \begin{bmatrix} \cos\theta_{ij} & -\sin\theta_{ij} \\ \sin\theta_{ij} & \cos\theta_{ij} \end{bmatrix} \tag{3.16}$$

D represents the expected length, meaning the average edge length of the whole map. s_{ij} is therefore the scaling factor that individually needs to be applied to each edge, to get to that value. $\boldsymbol{R_{ij}}$ is the rotation matrix that is necessary since the angles of the edges are going to change during each iteration of the deformation. Therefore the calculations have to be updated to reflect the newly created angles. Since $\boldsymbol{R_{ij}}$ is unknown while setting the matrix it will be computed in each iteration during the conjugate gradient step and updated accordingly. Therefore as an initial value $\boldsymbol{R_{ij}}$ will be set to an identity matrix. Figure 3.1 shows how the constraint works without the rotation matrix.

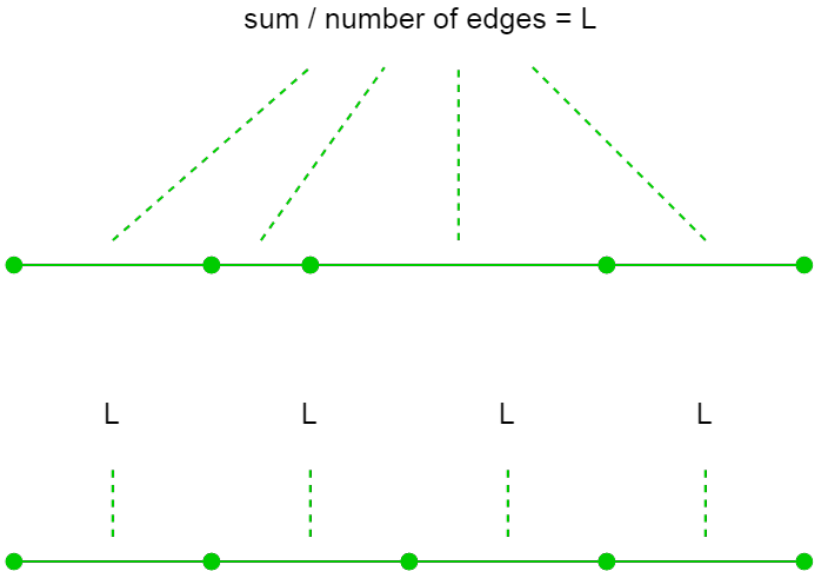


Figure 3.1: Visual representation of the regular edge length constraint.

Maximal Angles

Angles between incident edges of a metro map are usually maximized, or at least close to it, meaning that the edges are as far apart from each other as possible. Exceptions usually only occur based on the geographical layout or the underlying form of the map. This is especially true when a node has only 2 neighbors. Without good reason a metro line

would not be drawn in a zigzag shape. It is therefore very important to find an energy term that straightens lines and also maximizes the angle for more than 2 neighbors per node. The energy term

$$\Omega_a = \sum_{\boldsymbol{v_i'} \epsilon \boldsymbol{V'}} \sum_{\{i,j\}\{i,k\} \epsilon N(i)} |(\boldsymbol{v_i'} - (\boldsymbol{v_j'} + \boldsymbol{u_{jk}'} + M_{jk}\boldsymbol{u_{jk}'}))|^2 \tag{3.17}$$

$$u_{jk}' = \frac{1}{2}(v_k' - v_j') \tag{3.18}$$

$$M_{jk} = R(\frac{\pi}{2}) \tan(\frac{\pi - \theta}{2}) \tag{3.19}$$

does exactly that [Wu16]. In contrast to the energy term that concerns regular edge length, this one iterates over every node first and then for each iterates over every neighbor of that node, denoted by N(i). θ represents the desired maximized angle, which has to be calculated for each node individually via the formula

$$\theta = \frac{2 * \pi}{f} \tag{3.20}$$

where f represents the number of the current node's neighbors. $R(\frac{\pi}{2})$ on the other hand is just a simple 90 degree rotation in counter-clockwise direction. Figure 3.2 shows how the constraint works. Note that $\boldsymbol{u_{jk}'}$ is represented via the blue dotted line while $M_{jk}\boldsymbol{u_{jk}'}$ is represented via the red dotted line.

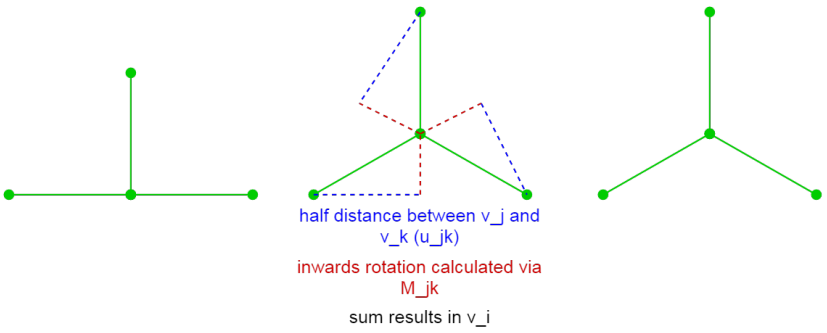


Figure 3.2: Visual representation of the maximal angle constraint.

Positional Constraint

Lastly the positional constraint is expressed by the distance between the node before and after repositioning and is therefore rather simple. The energy term [WC11] is given as

$$\Omega_p = \sum_{\boldsymbol{v_i'} \epsilon \boldsymbol{V'}} |(\boldsymbol{v_i'} - \boldsymbol{v_i})|^2 \tag{3.21}$$

Since this constraint requires the least amount of calculations and the representation of the calculation in matrix form can be done rather easily the following equation illustrates

an example for a graph with three nodes, where w represents the weight associated to the constraint. Without it, matrix A would be an identity matrix.

$$\begin{bmatrix} w & 0 & 0 & 0 & 0 & 0 \\ 0 & w & 0 & 0 & 0 & 0 \\ 0 & 0 & w & 0 & 0 & 0 \\ 0 & 0 & 0 & w & 0 & 0 \\ 0 & 0 & 0 & 0 & w & 0 \\ 0 & 0 & 0 & 0 & 0 & w \end{bmatrix} \begin{bmatrix} x'_0 \\ y'_0 \\ x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \end{bmatrix} = \begin{bmatrix} x_0 \\ y_0 \\ x_1 \\ y_1 \\ x_2 \\ y_2 \end{bmatrix}$$

Solving the Constraints

Now that all three constraints are defined they need to be combined into a single matrix to compute the new node positions. The idea is to minimize the total of all these constraints so the formula looks like this

$$\Omega_1 = w_l * \Omega_l + w_a * \Omega_a + w_p * \Omega_p \tag{3.22}$$

To allow the importance of each of these constraints to be adjusted a weight is added to each of the them. The positional constraint for example has a rather low importance compared to the others. To create the results shown in the paper the positional constraint's weight was set to 0.01, enough not to be ignored and able to prevent potential collapse of the nodes, but barely changing the result of the other constraints. Meanwhile the edge length weight was set to 10000 and the max angle weight to 15000. The values do however not have to be that extreme to achieve results. To calculate the result in matrix form the constraints are simply concatenated below each other. To solve this system in an iterative way the already introduces conjugate gradient method is applied.

Figure 3.3 shows the difference between the geographically correct Vienna metro map before and after smooth deformation.

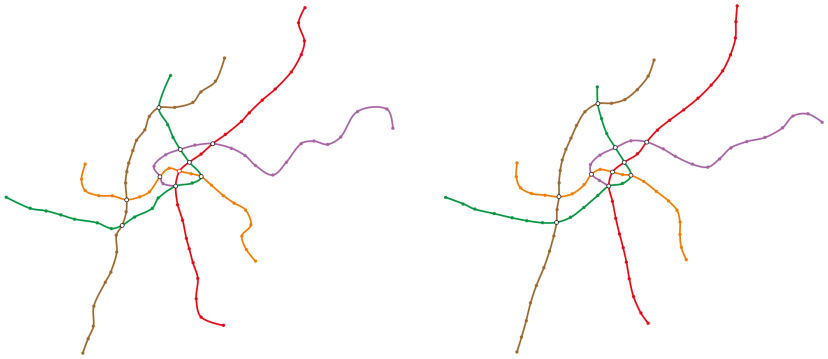


Figure 3.3: Vienna's geometrically correct metro map before (left) and after (right) smooth deformation

3.7.2 Octilinear Deformation

The second big step after obtaining the result from the smooth deformation is to adjust the edges. To do so, not only the constraint to move edges to the closest octilinear position is needed, but also the positional constraint, that was already introduced during the smooth deformation. The reason for that will be explained after the octilinear constraint is introduced.

Octilinear Edges

$$\Omega_o = \sum_{\{i,j\}\epsilon E} |(\tilde{\boldsymbol{v}}_{\boldsymbol{i}} - \tilde{\boldsymbol{v}}_{\boldsymbol{j}}) - O(\boldsymbol{v}'_{\boldsymbol{i}} - \boldsymbol{v}'_{\boldsymbol{j}})|^2 \tag{3.23}$$

O represents a function that rotates the edge to its closest octilinear position. This calculation needs to be pre-computed, meaning that for each edge it has to be decided in which direction to move and by how much before getting to the matrix form. [Wu16] Figure 3.4 shows how that the constraint first decides between which octilinear angle to take and then move accordingly.

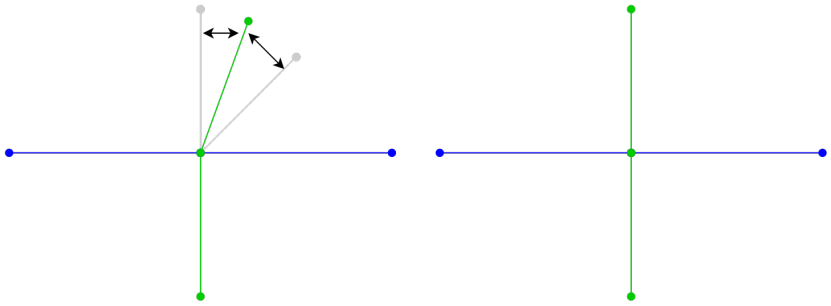


Figure 3.4: Visual representation of the octilinear constraint.

Positional Constraint

Since the energy term Ω_o tries to minimize the difference between every edge before and after rotation, but does not set any restrictions on the actual positions of the nodes the term would always solve correctly if the positions for $\boldsymbol{v}_{\boldsymbol{i}}$ and $\boldsymbol{v}_{\boldsymbol{j}}$ would simply be the same. The easiest way to see that is to just insert 0 for every node's x and y value. The energy term would always accept that, but the map would certainly not look like a metro map anymore. To stop this from happening the positional constraint Ω_p is introduced once again. This way the map will not collapse into a single point, but keep its general shape.

Solving the Constraints

$$\Omega_2 = w_o * \Omega_o + w_p * \Omega_p \tag{3.24}$$

Just like with the smooth deformation a simple concatenation of both energy terms combined with their own weights is sufficient. Since the positional constraint is again, not the main focus of this calculation the weight can be set rather small in comparison to the octilinear one. To achieve the same results as the given examples in this paper the weight for the octilinear deformation was set to 1000 while the positional constraint was set to 0.015, again just enough to not ignore the constraint, but very small so it would not have a huge role in determining the result.

Figure 3.5 shows the difference between the smooth geometrically correct Vienna metro map before and after octilinear deformation.

Figure 3.5: Smooth version of Vienna's geometrically correct metro map before (left) and after (right) octilinear deformation

3.7.3 Preventing Edge Intersections

Preventing edge intersections is not only important to keep the topology of the metro map in order, but also to prevent the map from breaking out of its convex hull and therefore the desired shape in which it is meant to be reshaped into. Since this constraint is used in the Reshaper-Phase as well as in the Deformation-Phase it is going to be described here in a separate paragraph. The basic idea is to stop any node from crossing any edge by setting a minimal distance between each node and edge and enforcing that, whenever this distance is overstepped. The energy term looks as follows [WC11]:

$$\Omega_i(\boldsymbol{v'_{i\to jk}}) = |(\boldsymbol{v'_i} - \boldsymbol{p'_{jk}}) - \delta_{i\to jk}(\boldsymbol{v_i} - \boldsymbol{p_{jk}})|^2 \tag{3.25}$$

$$\boldsymbol{p_{jk}} = r * \boldsymbol{v_j} + (1 - r) * \boldsymbol{v_k} \tag{3.26}$$

$$\delta_{i\to jk} = \frac{\varepsilon}{|\boldsymbol{v_i} - \boldsymbol{p_{jk}}|} \tag{3.27}$$

ε represents the minimal distance chosen, which may need to be adjusted depending on the scale of the map. If for example the map is rather dense the parameter for the distance needs to be set to a small value since otherwise every node-edge pair may be effected, which will not only move the map in unexpected ways, but also take a lot of

computation time. The problem with a small distance ε is however, that it may be crossed in a single iteration which will most likely lead to unwanted crossings. $\boldsymbol{p_{jk}}$ is the point closest to a node on an edge, represented by its distance to the edge's start- and endpoint. Since this point is not defined anywhere and potentially changes during each iteration of movement it needs to be pre-computed every time for each existing node-edge pair in the graph. If the distance from node $\boldsymbol{v_i'}$ to $\boldsymbol{p_{jk}}$ falls under ε the constraint Ω_i needs to be enforced and added to the sum of other constraints, which depending on the current phase of the algorithm is either the Reshaper or Deformer. Otherwise nothing is added for the particular node-edge pair.

CHAPTER 4

Implementation

The following chapter describes the order of operations used during the algorithm as well as details about each step. Additionally the used libraries are briefly explained and an example is given that changes with each step as well as a pipeline for a general overview.

To implement the algorithm the following external parts were used:

- **A general framework [OMM]** which serves as a tool to visualize the metro maps and which provides an easy way to add new algorithms to manipulate those maps.
- **A Delaunay Triangulator [DEL]** that is used to create a mesh from the metro station nodes, which is needed for the Reshaper phase.
- **A library for matrix calculations [EJM]** which are needed for both the Reshaper and the Deformer phase, since the equations used for these steps are all in matrix form.

4.1 General Concept

Before describing each step in more detail this part is about giving the reader a short overview of what is happening. This is to make it clear why each step is happening at what point and how that relates to the ones before and after. The goal of the algorithm is to take a metro map graph as input and output the same one reshaped into a given contour while keeping the appearance of a metro map. This means that the first main step is to reshape the map and then to recreate the right look in a second step. The idea to reshape the map is to set a number of handle positions along the convex hull of the metro map and then to move those handles in such a way that they form the desired contour. During this process any intersections that did not previously exist are

not allowed, meaning that the nodes inside keep the metro maps topology and do not escape the contour.

4.2 Visualized Pipeline

Now that the basic concept has been explained this section focuses on a visual introduction to the algorithm. Before the algorithm is explained in detail Fig. 4.1 gives a visual overview by providing the interim results after each step. The map used for this visualization is Vienna's geographically correct metro map and the shape is a rough contour of the St. Stephen's Cathedral approximated by nine points.

4.3 Input Data

4.3.1 The Metro Map

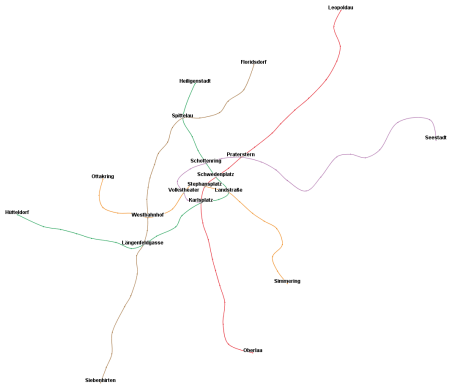
The input used is an xml-file that describes each station, line and the view. Each station consists of the real world longitude and latitude as well as the station's name. Each line has parameters indicating whether or not it is circular, what color it is and its name. Of course each stop is also listed, referred to by its name in order. Lastly the view describes the representation of the metro map. Just like each line the view also has parameters like a name, its dimensions and coordinates. Inside the view each line is represented as an edge at the start, referred to by its name followed by every station below that. Here the stations get their coordinates, that are used during visual representation. The xml-file shown in 4.1 is an example for how such a file would look like. Fig. 4.2 shows how the representation of the xml-file in the program looks like.

4.3.2 The Contour

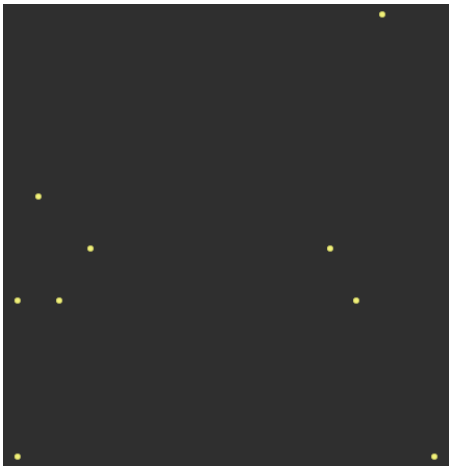
The contour is given by a number of points that form the desired hull. For a simple rectangle 4 points, each with an x and a y coordinate in the correct order would be sufficient. Since these points will later on correspond to the hull around the metro map it is important to also start with the node corresponding to the first node of the metro map's hull.

4.3.3 Other Parameters

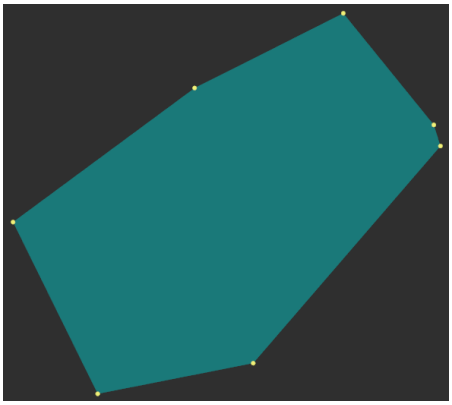
These parameters are the weights for each of the already mentioned constraints, as well as the minimum distance between a node an any edge that should not be crossed, the number of iterations for the cg-method, and the desired number of points that will approximate the hulls. Changing any of these parameters can drastically change the result, as well as the time it takes to calculate a result. An example for a very time sensitive parameter would be the minimum distance between a node and any edge. Setting this parameter to



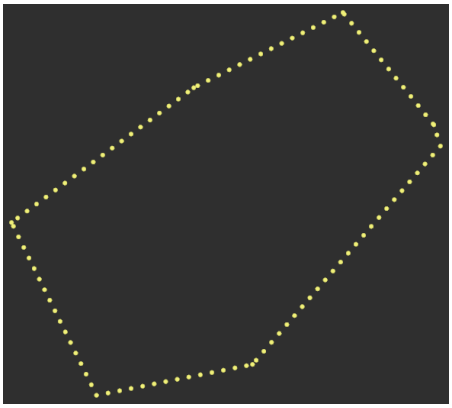
(a) Step 1: Input map



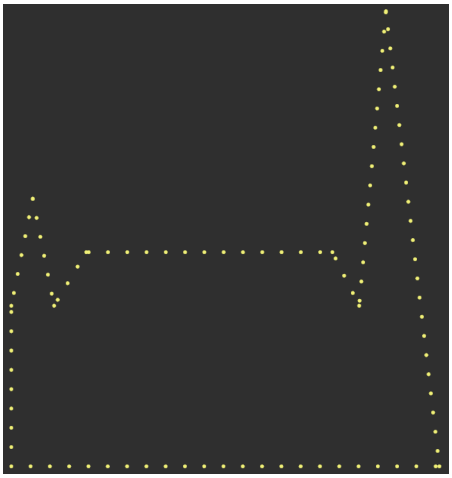
(b) Step 1: Input shape



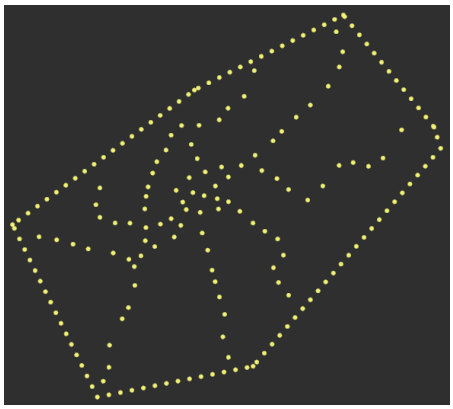
(c) Step 2: Calculated convex hull



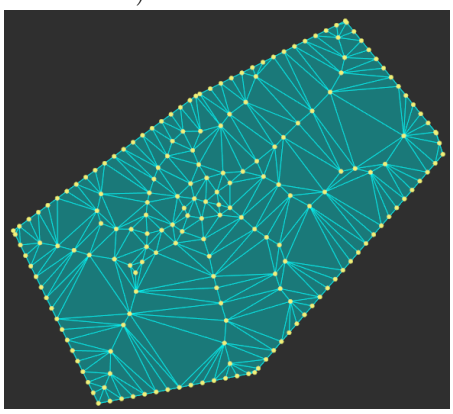
(d) Step 3: Calculated points on Polygon (convex hull)



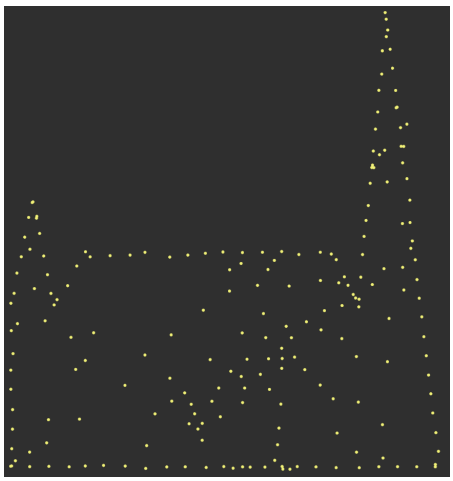
(e) Step 3: Calculated points on Polygon (shape)



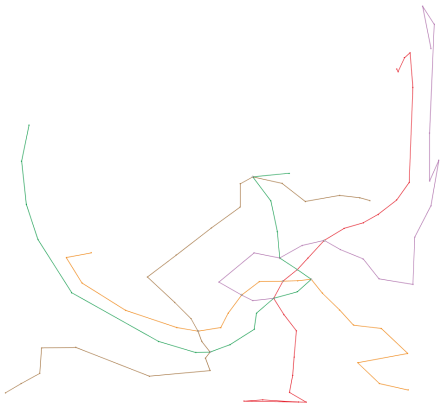
(f) Step 4: Combined graph (map and convex hull)



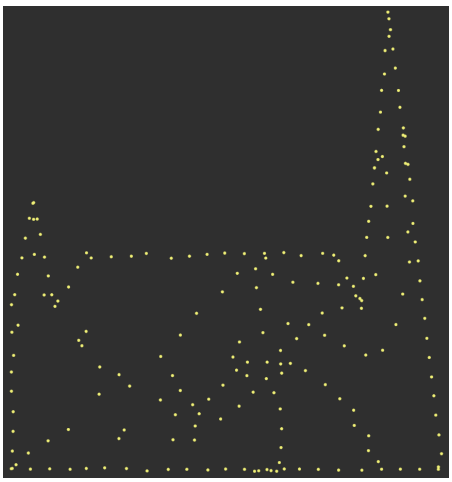
(g) Step 5: Triangulated combined graph



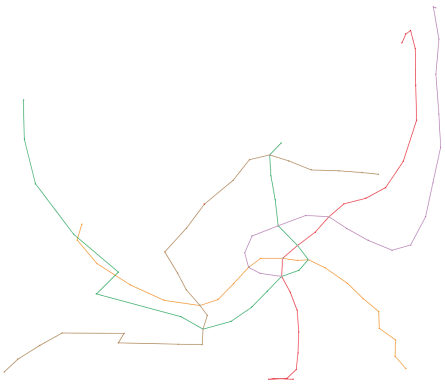
(h) Step 6: Reshaped combined graph (view of points)



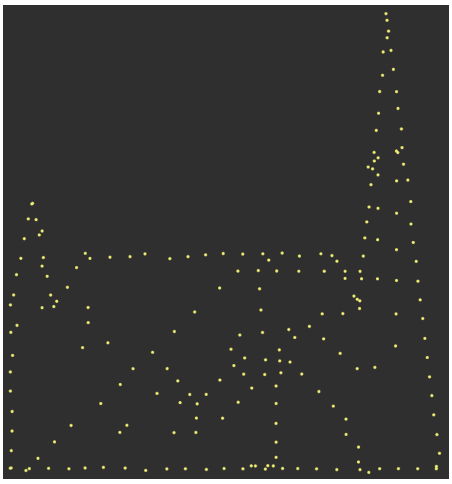
(i) Step 6: Reshaped combined graph (view of map)



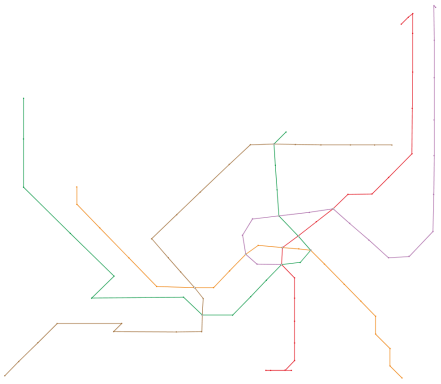
(j) Step 7: Smooth deformed graph (view of points)



(k) Step 7: Smooth deformed graph (view of map)



(l) Step 8: Octilinear deformed graph (view of points)



(m) Step 8: Octilinear deformed graph (view of map)

Figure 4.1: Algorithm's pipeline in images

Listing 4.1: Example xml input

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<omm-file version="1.0.0">
  <stations>
    <station lat="48.263424" lon="16.451707" name="A"/>
    <station lat="78.153707" lon="16.382446" name="B"/>
    <station lat="69.216756" lon="16.341840" name="C"/>
    <station lat="12.238260" lon="16.424787" name="D"/>
    <station lat="124.263424" lon="16.451707" name="E"/>
  </stations>
  <lines>
    <line circular="false" color="#E20613" name="U1">
      <stop station="B"/>
      <stop station="A"/>
      <stop station="C"/>
    </line>
    <line circular="false" color="#029540" name="U2">
      <stop station="D"/>
      <stop station="A"/>
      <stop station="E"/>
    </line>
  </lines>
  <view name="Wien" scene-height="902.146012" scene-width="
    ↪ 1000.000000" start-x="460.290214" start-y="442.606657">
    <edges line="U1"/>
    <edges line="U2"/>
    <station name="A" x="500.000000" y="500.000000"/>
    <station name="B" x="300.000000" y="500.000000"/>
    <station name="C" x="700.000000" y="550.000000"/>
    <station name="D" x="550.000000" y="300.000000"/>
    <station name="E" x="500.000000" y="700.000000"/>
  </view>
</omm-file>
```

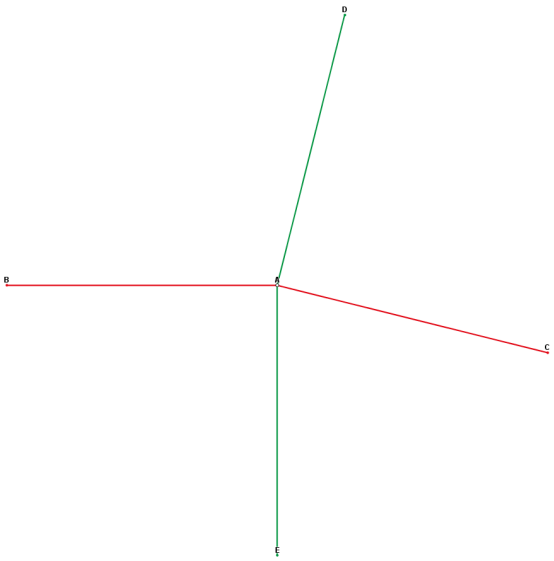


Figure 4.2: Visual representation of the example xml-file

a very low value will most likely reduce the time the computation takes, but it will also have an effect on possible intersections. As already mentioned previously, whenever this minimum distance is overstepped an additional row is added to the constraint matrix. Since every node has to be checked with every edge this can go out of hand very fast when having a bigger value for the minimum distance. However, while having a bigger value may take longer, it is also a lot harder to overstep it in a single iteration before the check takes place and a node potentially flips to the other side of an edge and creates an intersection. It is therefore important to choose the right value for each map. The same applies to different degrees to the other parameters. While their sensitivity varies, a drastic change will lead to drastic changes regarding the result and/or the time it takes to reach that result.

4.4 The Algorithm

4.4.1 Converting the Input Data

The framework responsible for the graph's visualization is also the one reading the xml-file and converting the data [OMM]. Other parameters are handled in the algorithm's class itself. The fist thing happening during the algorithm is the conversion of data, that was created from the xml-file to a self made CustomGraph class. It consists of a list of CustomNodes which represent the stations, CustomEdges which represent the connections between stations, an adjacency matrix and three maps to access nodes and Ids. Each CustomNode consists of a name, the coordinates used during representation, the original Node that was created via the framework, a list of lines it belongs to and an id. The CustomEdge on the other hand is just a pair of CustomNodes. At this point the graph

can already be displayed the way it was given to the program, as shown in Fig. 4.3.

Figure 4.3: Visualization of the initial graph

4.4.2 Creating the map's hull - Graham Scan

After obtaining information about the metro map and the desired shape, the first step is to create the convex hull for the metro map by using the Graham Scan [GRS]. The nodes from the metro map will be used as input for this part, and points will refer to the coordinates of each of those nodes. The first step is to find the bottom-most point and in case several points share that same y coordinate, the one with the lowest x coordinate is used. This point is the starting point of the convex hull. The second step is to sort all the remaining nodes by their angle in counter-clockwise direction. In case several points share the same angle the nearest one is put first. After that is done, a loop goes through all the points except the bottom-most one and removes any that have the same angle, except the one furthest away. Since this can lead to a list of nodes containing only 1 or 2 nodes, which would not make it possible to create a convex hull, it is necessary to check for that and only continue from here if there are indeed 3 or more nodes in that list. At this point the preparations for the main part of algorithm are done. The next step is to combine the correct nodes to the convex hull. To do so an empty stack is created and the bottom-most node, as well as the first and second node in the sorted list are added. From here on out the 3 top nodes of the stack (p_x, p_y, p_z) will be used to check whether

or not their orientation is counter-clockwise. In case it is p_y will be added to the convex hull and if not it will be removed from the stack and not added. After that the next node in the sorted list is pushed onto the stack.

- In case p_y was added to the hull the node p_x is now irrelevant, p_y becomes p_x, p_z becomes p_y and the newly added node becomes p_z.
- In case p_y was not added p_x remains p_x, p_y is removed, p_z becomes p_y and the newly added node becomes p_z

To make the concept clearer Fig. 4.4 shows an example of the process. Note that red stands for currently checked, green for confirmed, gray for discarded and black for not yet processed.

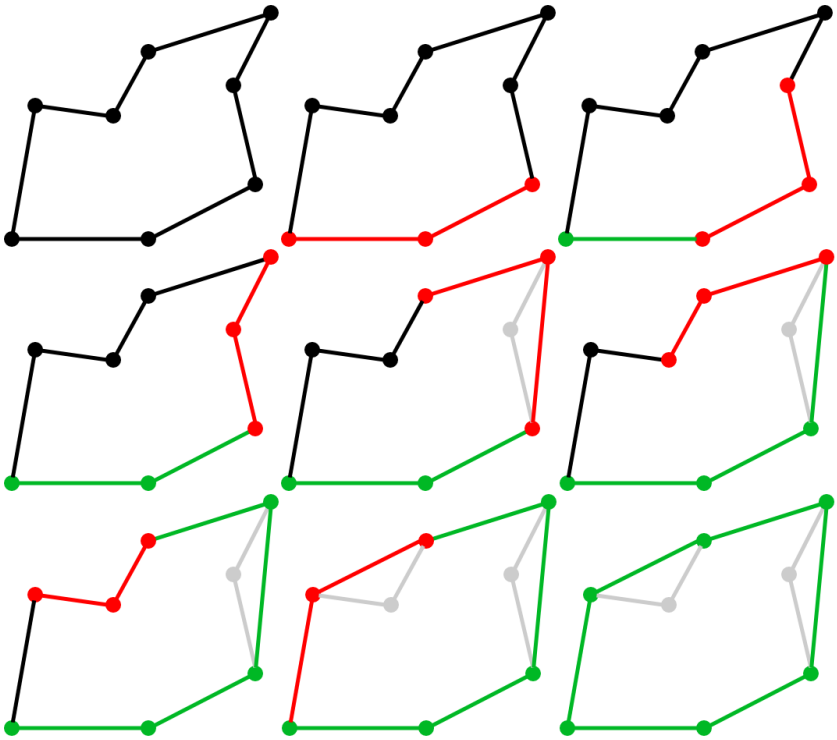


Figure 4.4: Visualization of the graham algorithm (top left to bottom right). red edges: currently checked, green edges: confirmed, gray edges: discarded, black edges: not yet processed

To ensure that all nodes fit inside the contour later on, instead of landing on the line the calculated convex hull is slightly scaled up. To do so adding a single coordinate point between a node and the contour would suffice so the scaling factor can be chosen rather small. In this case the x and y coordinates are multiplied with 1.0001. Since that also

means moving the coordinates to the bottom-right, the now scaled up hull needs to be moved back slightly. To do so the centroids of both the calculated convex hull and the scaled up version are calculated. Afterwards the scaled up one is moved back by the difference between the centroids' x and y coordinates. That is to ensure that all nodes are now still inside the hull and the scaling basically just happens around the center of the hull. The formulas to calculate the centroid of a polygon look as follows:

$$C_x = \frac{1}{6A} \sum_{i=0}^{n-1} (x_i + x_{i+1})(x_i * y_{i+1} - x_{i+1} * y_1) \tag{4.1}$$

$$C_y = \frac{1}{6A} \sum_{i=0}^{n-1} (y_i + y_{i+1})(x_i * y_{i+1} - x_{i+1} * y_1) \tag{4.2}$$

$$A = \frac{1}{2} \sum_{i=0}^{n-1} (x_i * y_{i+1} - x_{i+1} * y_i) \tag{4.3}$$

The only criteria for these formulas to work is to have the points be in order of their appearance around the polygon's perimeter, which is already the case. Applying those formulas is then a matter of running a for loop for the sums and then calculating C_x and C_y by simply dividing the interim results by $6A$ as in 4.1 and 4.2.

4.4.3 Adjusting the contour

After the convex hull for the metro map is obtained it is necessary to adjust the desired shape in a way to make all points of the metro map fit easily as to reduce possible complications, which mostly occur when movements in narrow areas happen. To do so the area of the contour is adjusted to be the same as the one of the convex hull. Since the coordinates of the convex hull are known for every node the calculation of its area is a matter of a single formula

$$A = \left| \frac{(x_1 * y_2 - y_1 * x_2) + (x_2 * y_3 - y_2 * x_3) + ... + (x_n * y_1 - y_n * x_1)}{2} \right|$$

The important things to consider here are that the coordinates have to be in order and that the last coordinates have to be paired with the first ones at the end to complete the polygon. Since the coordinates given for the contour are required to be input in order, and the convex hull is also created and saved in order, the application of the formula is straightforward.

After the areas for both the contour and the convex hull are acquired the square root of the result of their division is enough to find find the scaling factor, that then needs to be applied to every point of the contour. The reason to use the square root is that as already mentioned, the scaling factor needs to be used in combination with every node's coordinates to change their position and since the area scales quadratically while the coordinates (or the sides) do not, the square root is required. After that is done, both polygons (the convex hull and the contour) now have the same area. This does

not guarantee, that no complications occur, but it at least reduces the risk compared to having a large hull area and a small contour area. Furthermore it provides a baseline that allows scaling according to the area of the contour (now also the area of the hull) with a single additional scaling factor, in case that further increasing the area is able to solve problems. This of course all heavily depends on the shape of the contour and is most likely only useful when more complex contours are used.

4.4.4 Calculating Points on both Polygons

During the Reshape-Phase every node on the convex hull needs to be moved to a corresponding position on the contour, which means both of them need to be described by the same number of coordinates to allow a one-to-one translation. Since the number of coordinates for the contour depends on the input and can be as low as 3 and basically have no upper limit it is important to know how many points to use to approximate the shape. This value is given via a parameter.

The first step is to calculate the length of the polygon's perimeter. To do so the formula for the length of a line between two points is used for every side of the polygon and results are then added up. The only important thing to remember here, is again to close the polygon or in other words not forgetting the line connecting the last and the first node.

Now that the length for the polygon has been calculated and the number of nodes is given via a parameter, dividing the length by the number of points results in the distance between two points on the polygon. This calculation is necessary, since the next step will be following each line of the polygon's perimeter for the calculated distance before creating a point there and then doing the same thing until closing the polygon by reaching the starting point again. Notable things to consider here are to also add the corner points of both polygons, no matter whether or not a point lands on a corner point's position. The reason for that depends on whether the algorithm runs for the convex hull or the contour. In case of the convex hull it is so that the actual nodes do not land outside the contour instead of on or inside of it, which would be possible, since the result would just be an approximate shape whose accuracy depends on the number of hull nodes given per the parameter.

The problem here is that in case everything works as intended the metro map's nodes outside would not be able to cross the convex hull to move to the inside during the Reshape-Phase. This would lead to a result with stations outside the contour.

In case of the contour it would lead to a possible inaccurate result because as with the convex hull the result is just an estimate, which means the resulting polygon given by the calculated points could be missing some small details or in the worst case even important feature. This would be easily possible if the contour contains sharp angles, meaning that even a relatively small distance between points could easily cut off a corner. Figure 4.5 shows an example of cut corner points.

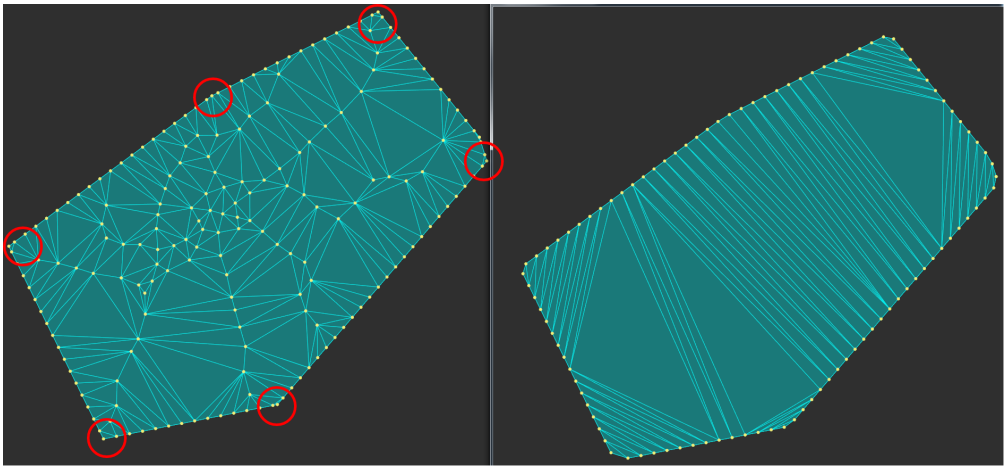


Figure 4.5: Points missing from the hull (missing points marked)

Given the convex hull / contour points and the calculated distance algorithm 4.1 describes how to uniformly distribute any number of points along the convex hull and the contour respectively.

4.4.5 Combining the Graph and the Convex Hull

To move the metro map according to the contour the map needs handle positions. These handle positions are the positions of the convex hull's points. Since these are already calculated at this point, the next step is to combine the map positions and the convex hull positions into a single graph. This process does not require any complex calculations and is rather straightforward, because the edges of the graphs do not overlap, which in turn makes the combination of their respective adjacency matrices a matter of changing ids. Since only 2 graphs are to be combined the ids of the first graph can stay the same, while the ones of the second one have to be offset by the size of the first graph. These changes then have to be put into the newly created adjacency matrix.

4.4.6 Triangulating the combined Graph

For this step an external library is used [DEL]. It requires a list of 2D vectors which represent each node's x and y positions and after calculation returns a list of 2D triangles. To do so the library utilizes the Delaunay triangulation algorithm, already briefly explained in the methodology section.

4.4.7 Calculating and aligning the centroids

The positions given for the metro map and the ones for the contour could not only differ by the area they enclose, but also by their general positions. The problem of the differing areas has already been solved, but moving nodes by possibly extremely large distances is

Algorithm 4.1: Points on Polygon

Input: A list of nodes representing the hull or the contour, The desired distance between 2 points on the polygon

Output: A list of points on the polygon with the given distance between them plus the corner points of the original polygon

```
1  add first node of nodes also to the end of nodes;
2  for each node except last one in nodes do
3      set startPosition to current node's position;
4      set endPosition to next node's position;
5      set tmpPosition to current node's position;
6      if slope of startPosition and endPosition is infinite then
7          if distance overflow ≠ 0 then
8              add overflow to currentPosition in y direction;
9              if tmpPosition is not between startPosition and endPosition then
10                 calculate distance overflow from endPosition;
11                 tmpPosition = endPosition;
12             end
13             else
14                 add tmpPosition to result;
15             end
16         end
17         while tmpPosition is between startPosition and endPosition do
18             add distance to tmpPosition; add tmpPosition to result
19         end
           // the last point does not lie on the polygon
20         remove last point from result;
21         calculate distance overflow from endPosition;
22         tmpPosition = endPosition;
23     end
24     else if slope of startPosition and endPosition is 0 then
           // same as above, but add overflow to currentPosition
              in x direction
25     end
26     else
           // same as above, but calculate in which direction to
              add distance overflow
27         set dx to distanceOverflow/√(1 + (slope * slope));
28         set dy to slope * dx;
29         add dx to tmpPosition's x coordinate;
30         add dy to tmpPosition's y coordinate;
31     end
32 end
33 return result;
```

a source for errors and inaccuracies. To solve this problem the contour is moved over the metro map before reshaping the metro map. This is done by aligning the centroids of the metro map's convex hull and the contour similar to when convex hull was scaled up around its center.

4.4.8 Reshaper

Now that everything related to the hulls and the triangulation is done the next step is the actual reshaping of the metro map. This segment is split into three parts. The first one describes preparation steps, the second one how the Reshaper is implemented, while the third part addresses the context in which it is used. The important parameters for the Reshaper are the combined graph, the calculated list of triangles and the weight that determines how much the process should do during a single call.

Preparations

The first step is to create a neighbor list from the triangles returned via the Delaunay triangulation. This neighbor list differs from the regular metro map not only in that the convex hull is included, but the edges connecting nodes are not the ones from the metro map, but the ones generated during triangulation. This means that the connecting edges have no similarity with the metro map and need to be created completely from scratch. This is done by creating a map (neighbormap) that contains the id for every node of the combined graph as key and a list of ids of its neighbors. While the keys can be created by simply iterating over the combined graph, the list of neighbors is filled by accessing all 3 of the triangle's corner points from the triangle list and cross referencing their positions with the ones from the graph. The reason it has to be done this way is that every other information, besides the position, is lost when converting to the triangles due to using a library. After that is done the edges are created according to the newly created neighbor map. After that is done the Reshaper is given a map that consists of ids as keys and positions as values. The ids refer to the points of the combined graph that are supposed to move while the positions are the goals for those nodes, or in other words the points on the contour that were previously calculated. Obviously the ids in that map only address the hull nodes of the combined graph.

Implementation

Since it is rather confusing to follow the implementation without an explanation or formulas as context, both of those will be given before the actual implementation for this section.

The basic idea is to use an unknown rotation matrix T on the original edge vector $\boldsymbol{v_j} - \boldsymbol{v_j}$ and move nearby vertices accordingly to new locations. To derive the rotation matrix 4 nodes around the edge are used as samples. The following formulas describe this process

where

$$\boldsymbol{T_k} = \begin{bmatrix} c_k & s_k \\ -s_k & ck \end{bmatrix}$$

describes the rotation matrix,

$$\boldsymbol{T_k} = \sum_{v \epsilon N(e_k)} |\boldsymbol{T_k v} - \boldsymbol{v'}|^2$$

is the least squares form to calculate T_k based on the neighboring nodes,

$$\{c_k, s_k\} = \sum_{v \epsilon N(e_k)} | \begin{bmatrix} c_k & s_k \\ -s_k & ck \end{bmatrix} \begin{bmatrix} v_x \\ v_y \end{bmatrix} - \begin{bmatrix} v'_x \\ v'_y \end{bmatrix} |^2$$

shows how to calculate c_k and s_k and,

$$\{c_k, s_k\} = | \begin{bmatrix} v_{ix} & v_{iy} \\ v_{iy} & -v_{ix} \\ v_{jx} & v_{jy} \\ v_{jy} & -v_{jx} \\ v_{lx} & v_{ly} \\ v_{lx} & -v_{lx} \\ v_{mx} & v_{my} \\ v_{my} & -v_{mx} \end{bmatrix} \begin{bmatrix} c_k \\ s_k \end{bmatrix} - \begin{bmatrix} v'_{ix} \\ v'_{iy} \\ v'_{jx} \\ v'_{jy} \\ v'_{lx} \\ v'_{ly} \\ v'_{mx} \\ v'_{my} \end{bmatrix} |^2$$

rearranges the previous formula to separate c_k and s_k to make their calculation possible and

$$\{c_k, s_k\} = |G_k \begin{bmatrix} c_k \\ s_k \end{bmatrix} - \begin{bmatrix} v'_{ix} \\ v'_{iy} \\ \vdots \end{bmatrix} |^2$$

shortens the formula for convenience by replacing the matrix consisting of all the neighbors with the variable G_k.

$N(e_k)$ represents the neighbors of the current edge as shown in Fig. 4.6.

Via $\boldsymbol{x} = (\boldsymbol{A}^T \boldsymbol{A})^{-1} \boldsymbol{A}^T \boldsymbol{b}$ it is then possible to calculate c_k and s_k. All this is done to then be able to create the first part of the constraint, which minimizes the edge deformation and looks now, including T_k, like this:

$$\Omega_{r1} = \sum_{\{i,j\} \epsilon E} |(\boldsymbol{v'_i} - \boldsymbol{v'_j}) - T_{ij}(\boldsymbol{v_i} - \boldsymbol{v_j})|^2$$

After some rearrangement the final formula that needs to be replicated in the algorithm looks like:

$$\Omega_{r1} = \sum_{\{i,j\} \epsilon E} |(\boldsymbol{v'_i} - \boldsymbol{v'_j}) - \begin{bmatrix} e_{kx} & e_{ky} \\ e_{ky} & -e_{kx} \end{bmatrix} (\boldsymbol{G_k}^T \boldsymbol{G_k})^{-1} \boldsymbol{G_k}^T \begin{bmatrix} v'_{ix} \\ v'_{iy} \\ \vdots \end{bmatrix} |^2$$

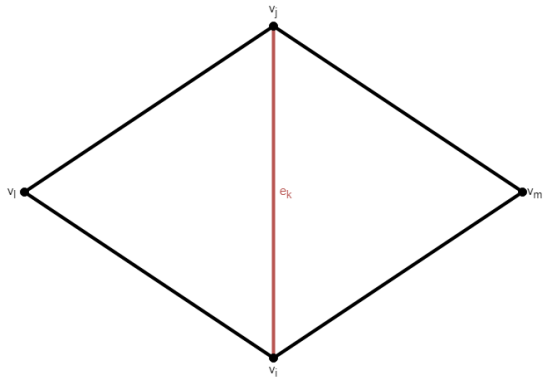


Figure 4.6: edge neighbors used for matrix T_k

And now that the goal and the steps to reach that goal are explained the actual description of the implementation starts:

To create the edge deformation constraint one main loop is required that iterates over all edges, given by the triangulation (not the edges of the metro map). The first step is to create G_k and the formula from it that results in c_k and s_k. To do so the upper part of G_k is created by addressing the nodes that are connected by the edge. After that is done the next lines are added via a loop, since there could only be one neighbor in case the edge is on the boundary of the triangulation. Most of the time however, two neighbors will be added, which results in four more lines for G_k. After that is done the calculation of c_k and s_k begins by simply transposing and inverting G_k to recreate the formula.

Next the matrix $\begin{bmatrix} e_{kx} & e_{ky} \\ e_{ky} & -e_{kx} \end{bmatrix}$ is created by once again addressing the nodes that are connected by the current edge and creating the matrix out of them. Next the matrix that represents $\boldsymbol{A}$ is created. This matrix will always look like $\begin{bmatrix} -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix}$ no matter the actual indices of the nodes in $\boldsymbol{x}$ or in other words $\begin{bmatrix} v'_{ix} \\ v'_{iy} \\ \vdots \end{bmatrix}$ because the actual indices are not important right now and the intermediate results given via this calculation so far will be shifted to the correct positions in a later step. Now that all parts of the formula are created the calculation of $(\boldsymbol{v'_i} - \boldsymbol{v'_j}) - \begin{bmatrix} e_{kx} & e_{ky} \\ e_{ky} & -e_{kx} \end{bmatrix} (\boldsymbol{G_k}^T \boldsymbol{G_k})^{-1} \boldsymbol{G_k}^T$ is done. This intermediate result will be called $\boldsymbol{H}$.

In the next step the contents of $\boldsymbol{H}$ will be shifted to the correct positions to create the matrix $\boldsymbol{A_u}$ (u stands for upper, since this is only the upper half of matrix $\boldsymbol{A}$). $\boldsymbol{A_u}$ reflects the results from all $\boldsymbol{H}$ matrices that were created for each edge and now correctly align with the vector $\boldsymbol{x}$ to create the least squares form necessary to calculate the new

positions. The shift is done by simply accessing the nodes' ids and moving the results of $\boldsymbol{H}$ to the corresponding column. The row is determined by the order the individual $\boldsymbol{H}$ matrices were created in.

Since the vector $\boldsymbol{b_u}$ for this part of the constraint only contains 0, a zero vector in the same size as the matrix $\boldsymbol{A_u}$ is created. With that the upper half of the Reshaper matrix that represents the edge deformation is done.

Next up is the lower part that represents the movement of handle nodes. Since the ids and the new positions are given via a map, this map is now iterated over with a loop. The ids again represent the columns that need to be filled, while the row is again determined by the order in the map (although the order of rows ultimately does not matter, it just has to align with $\boldsymbol{b_l}$). The values of $\boldsymbol{A_l}$ are all zero, besides the filled in values which are all equal to the weight of the constraint, which is also given as a parameter. $\boldsymbol{b_l}$ on the other hand contains the desired x and y coordinates multiplied with the same weight. After both are filled, the lower part of the constraint is now also done and all that is left is to concatenate $\boldsymbol{A_u}$ with $\boldsymbol{A_l}$ and $\boldsymbol{b_u}$ with $\boldsymbol{b_l}$.

Usage

Algorithm 4.2 illustrates how the conjugate gradient method is used in conjunction with the Reshaper to slowly move every node to its designated position.

The reason why the weight is slowly increased over the course of the cg-method is to avoid extreme movement in a single step and therefore minimize the risk of creating new intersections.

4.4.9 Deformer - Smooth Deformation

The Deformer, in contrast to the Reshaper, works on the original, if reshaped, graph instead of on the combined graph. The first part is the smooth deformation, which as already mentioned earlier consists of three constraints. The general structure of the algorithm is to calculate each of these three constraints separately, then combine them into a single matrix and afterwards solve them with the conjugate gradient method.

Regular Edge Length

While this constraint contains a multiplier to scale, and a rotation matrix to adjust each edge individually, during this step only the scaling is set. The reason for that is that the rotation needs to be updated after each movement, or in other words during each iteration of the conjugate gradient method, which will happen later on. The general idea is to produce the form $\boldsymbol{Ax} = \boldsymbol{b}$ that is mentioned under 3.1 Least Squares. Algorithm 4.3 gives an overview of what happens during this part, and can also serve as a reference for other constraints.

Algorithm 4.2: Reshape iterations

```
initialize errorThreshold;
initialize step with 1;
initialize number of iterations with 0;
initialize constraintWeight with a small value below 1;
initialize maxConstraintWeight with a larger value;
calculate positions for current Reshaper values;
while true do
    if step == 1 then
        if cg-method is initialized then
            run initialized cg-method iteration;
        end
        else
            run uninitialized cg-method iteration;
            set cg-method to initialized;
        end
        increase number of iterations;
        set position of graph to cg-method's result;
        if error from cg-method < errorThreshold OR number of iterations ≥
          number of max-iterations then
            if constraintWeight < maxConstraintWeight then
                reset cg-method to uninitialized;
                reset number of iterations to 0;
                increase constraintWeight;
                calculate positions for current Reshaper values including new
                  constraintWeight;
            end
            else
                increase step by 1;
                skip the rest of the current iteration;
            end
        end
    end
    if step == 2 then
        run Deformer's smooth deformation;
        run Deformer's octilinear deformation;
        revert any changes to node positions used for better results;
        break;
    end
end
```

Algorithm 4.3: Regular Edge Length

```
1  calculate averageEdgeLength;
2  create matrix A with correct size;
3  create vector b with correct size;
4  for edge in graph do
5  |  point1 = startpoint of edge;
6  |  point2 = endpoint of edge;
7  |  set next row of b to point1.xCoordinate - p2.xCoordinate;
8  |  set next row of b to point1.yCoordinate - p2.yCoordinate;
9  end
10 for edge in graph do
11 |  point1 = startpoint of edge;
12 |  point2 = endpoint of edge;
13 |  calculate edgeLength;
14 |  calculate edgeMultiplier via averageEdgeLength/edgeLength;
15 |  set point1's x cell of A to 1/edgeMultiplier;
16 |  set point2's x cell of A to −1/edgeMultiplier;
17 |  go to next row;
18 |  set point1's y cell of A to 1/edgeMultiplier;
19 |  set point2's y cell of A to −1/edgeMultiplier;
20 |  go to next row;
21 end
22 create weightMatrix;
23 multiplay weightMatrix with A;
24 multiplay weightMatrix with b;
```

Maximal Angles

This constraint requires rows in the matrix for each neighbor of the current node, unless there is only one, which is rarely the case. It is possible to either add rows in every iteration or to create the zero matrix with the correct size first and then address it correctly in each step. Here the second method of creating the matrix first is used, which means that a loop goes over every node and adds the number of neighbors, in case it is more than 1, to a counter. After that a matrix with the counter's value as the number of rows and two times the number of nodes (since the vector $\boldsymbol{x}$ consists of every node's x and y coordinates) as the number of rows is created. The next step is to fill every row of that newly created matrix $\boldsymbol{A}$ with correct values. To do so a loop iterates over every node where first the angle Θ and the value of the tangent are calculated. Then the value for node $\boldsymbol{v_i'}$ or in other words the current node is set, which in this case is 1. Then all of the node's neighbours are sorted by their angle.

Next a second inner loop is created which iterates the sorted neighbors. In that loop the values for the nodes $\boldsymbol{v_j'}$ and $\boldsymbol{v_k'}$ are set. For $\boldsymbol{v_j'}$ these values are $(-1+0.5+R(tanVal*0.5))$

while for $\boldsymbol{v}'_{\boldsymbol{k}}$ they are $(-0.5 - R(tanVal * 0.5))$. These values can be directly taken from the constraint's formula by simple extracting the values that come alongside either $\boldsymbol{v}'_{\boldsymbol{j}}$ or $\boldsymbol{v}'_{\boldsymbol{k}}$. The 90 degree counter-clockwise rotation can be achieved by translating the x and y coordinates x/y to -y/x. Now all that needs to be done is to set these values at the correct column position and not to forget the last pair of neighbors that connects the last and the first neighbor.

For the vector $\boldsymbol{b}$ a simple zero vector with the same number of rows as $\boldsymbol{A}$ is sufficient, since the goal of the constraint is for the subtraction to reach 0. After that the weight is added, which means creating a diagonal matrix with the desired weight as value and multiplying it with $\boldsymbol{A}$ and $\boldsymbol{b}$ (which since it is 0 here is not necessary) respectively.

Positional Constraint

The positional constraint is rather simple, since a single loop and no prior calculations are enough. To set matrix $\boldsymbol{A}$ the creation of an identity matrix is sufficient, since the layout used for the coordinates of a single node is $\begin{bmatrix} x & 0 \\ 0 & y \end{bmatrix}$. To create vector $\boldsymbol{b}$ a loop is created that iterates every node and adds the x or y coordinate depending on the current row resulting in a vector with two times the number of nodes as size. Afterwards, just like with the other constraints the weight matrix is created and multiplied with $\boldsymbol{A}$ and $\boldsymbol{b}$.

Solving the Constraints

Solving all three constraints requires combining them into a single matrix $\boldsymbol{A}$ and a single vector $\boldsymbol{b}$. Since the conjugate gradient method requires a symmetric and positive definite matrix, $\boldsymbol{A}$ and $\boldsymbol{b}$ need to be prepared accordingly as mentioned earlier when explaining the cg-method.

After that the initial residual is calculated by multiplying $\boldsymbol{A}$ with the estimate, which is just the current positions of the nodes, and then subtracting the result from $\boldsymbol{b}$. If the residual's calculated value is below a given threshold the cg-method stops here, but in case it is not the iterative part starts.

A for loop is started that stops after either being below the threshold or after a given number of iterations. Since at this point the edge rotation has still not been updated, but an estimate which has been calculated in the previous step has already changed positions it is necessary to calculate that rotation matrix for each edge now, before doing anything else involving the constraint. To do so a loop iterates over every edge and calculates the difference between the current and previous angle and saves those values in a list. Afterwards one big rotation matrix consisting of all the single rotations is created and filled according to the created list and multiplied with the old edge constraint matrix. After that is done the edge intersection constraint is calculated, which will be explained in more detail in a separate section.

After all that is done matrix $\boldsymbol{A}$ and vector $\boldsymbol{b}$ are updated with the just created matrices. Next the formulas of the cg-method are used to calculate the new residual value which is checked against the threshold. In case the error is not small enough the loop starts over with the newly calculated values.

4.4.10 Deformer - Octilinear Deformation

Now that the layout has been deformed "smoothly" the next step is to create the correct angles of 45 degree intervals. This step consists of the octilinear constraint and the positional constraint, that was already explained earlier. For that reasons the explanation of the positional constraint will be skipped here.

Octilinear Edges

The first step is to calculate the current angle α of every edge. Next the result of α mod 45 is calculated to decide whether the angle is closer to the next or the previous 45 interval. Depending on the result that value β is calculated by $45 * \lfloor \frac{angle}{45} \rfloor$ or $45 * \lceil \frac{angle}{45} \rceil$. Next the difference between α and β is calculated to know by how much the edge has to be rotated.

Next the matrix $\boldsymbol{A}$ and the vector $\boldsymbol{b}$ are created. To express the edges in $\boldsymbol{A}$ 1 and -1 are set at the corresponding positions in each row while the expected result of the rotation is set in $\boldsymbol{b}$. As always with these constraints the weight matrix is created as a last step and multiplied with $\boldsymbol{A}$ and $\boldsymbol{b}$.

Solving the Constraints

This step is basically the same as the one for the smooth deformation, with some steps missing. Since there is no rotation matrix to update that part is unnecessary. The intersection check is also skipped and while it is possible to break out of the contour or create intersections in the map, the results so far have not looked significantly worse because of it. It also allows for a clearer octilinear layout, since there are no constraints to reduce its influence. In the Result & Discussion section this will be addressed further, but for now it is only necessary to know that this part is also omitted.

That means that for the octilinear deformation only the combination of the constraints and the implementation of the cg-method are necessary. As input this step takes the positions given by the smooth deformation.

4.4.11 Preventing Edge Intersections

Calculating the Edge intersection prevention consists of two main steps. The first one is the preparation of distances and the check to see if any of those are lower than the defined threshold. The second one is then the creation of the matrix and the vertices based on those results.

Since every node has to be checked with every edge it is necessary to have two loops, one for each. Inside the second loop the distance between the current node and the current edge is then calculated. Important to note here is that the edge is finite, meaning that the formulas used have to reflect that. Not only the distance but also the point that is closest on the edge to the current node are then returned. Only if the returned distance falls below the threshold the loop continues with the current node and edge and the creation of $\boldsymbol{A}$ and $\boldsymbol{b}$ begins.

Two new rows for the x and y coordinates are added to $\boldsymbol{b}$, each containing $\delta * (nodeCoordinate - closestPointCoordinate)$ where $\delta = \frac{threshold}{calcDistance}$. For $\boldsymbol{A}$ first the position of the current node is set to 1 for both x and y. Since $\boldsymbol{x}$ does not contain the closest point to the node it needs to be expressed via other existing nodes. In case the start- p_s and endpoint p_e of the edge lie on the same x coordinate the values for p_s and p_e are set to 0.5 each. The same applies to the y coordinate. Otherwise it is necessary to calculate the percentage distance of point p from p_s and p_e and use those values inside $\boldsymbol{A}$ at the correct node indices representing p_s and p_e.

After all that is done the same calculations repeat for the distances to the contour edges. It is important to consider that neither p_s nor p_e can be the current node and that nodes on the contour have to be skipped altogether, since it is not desirable to move the contour, but only the graph nodes.

CHAPTER 5

Results

To test the algorithm, several maps have been used in combination with different contours. The maps used were the schematic Berlin metro map, the schematic and the geographic Vienna metro map as well as the schematic Stuttgart S-Bahn map. The contours used are shown in the graphics, represented by their points, that were created during the algorithm.

Not only are there differences when it comes to the quality of the visual results, but also when it comes to computation time. Table 5.1 shows the times for all maps when combined with the St. Stephen's Cathedral contour.

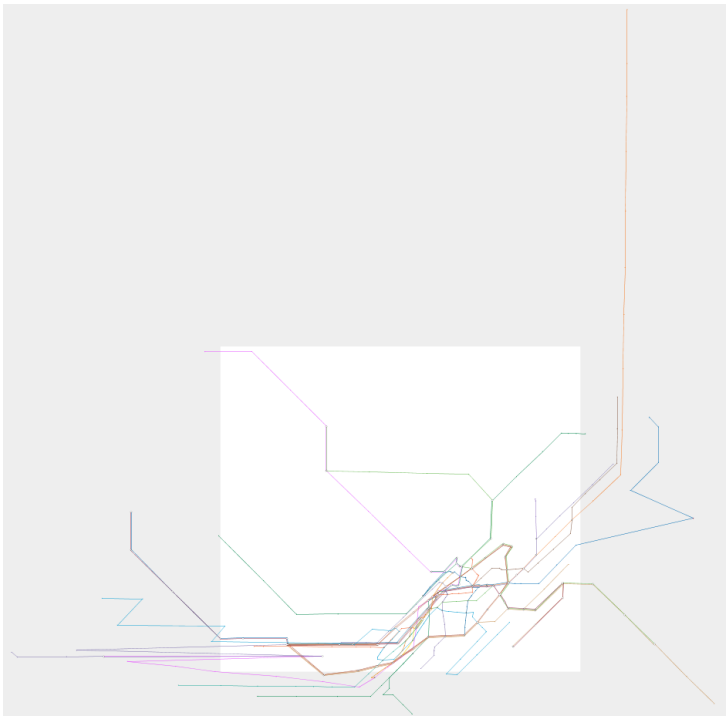
Note that the number of different weights the Reshaper has to go through is fixed. This means that the intermediate results may not vary much from the end result, they do however all run through all weight increases. However it is possible that some maps terminate faster inside a weight iteration than others, since they reach the error threshold for that specific weight faster. That said it is pretty clear from those results that more complex maps, that consist of more nodes and edges are a lot slower than easier maps. While the number of nodes and edges roughly triples the time for each step is multiple times slower than that. Two more things that are worth mentioning here are that the map was redrawn during each test, like it would be when used regularly, meaning that the results are probably slightly slower than they would be without animation. And

map	#nodes	#edges	reshape	smooth	octi.	total
Vienna geometric	98	104	37416	1267	20	38703
Vienna schematic	98	104	33825	1080	20	34925
Berlin	311	357	467454	18445	641	486540
Stuttgart	83	84	24191	809	13	25013

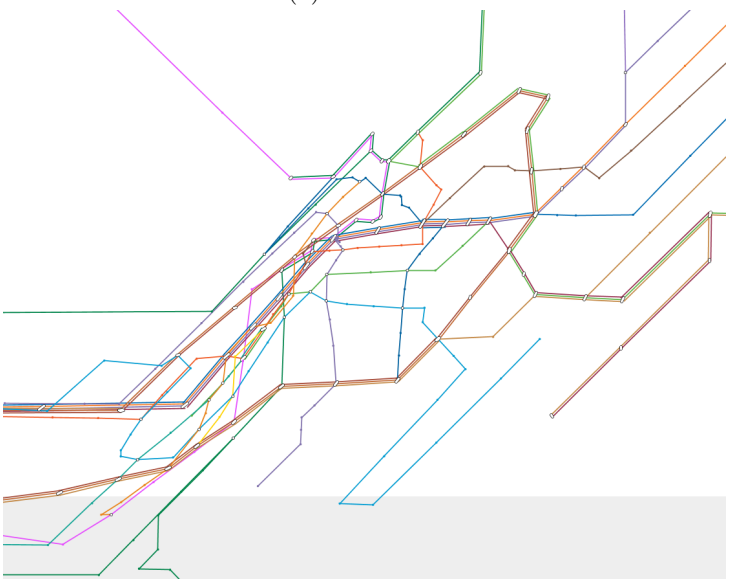
Table 5.1: Calculation times for different parts of the algorithm in milliseconds

lastly, for each of those tests the constraint to prevent intersections has not been used during the Reshaper phase. This effects the Berlin map the most, in the sense that since it consists of more nodes and edges, more checks would have to be made here in comparison to the other maps. Obviously the parameters could be adjusted individually for each of these maps to potentially better the results, but since that would defeat the purpose of the comparison, all maps used the exact same.

Next up are the visual results. For this only the geographic Vienna metro map and the schematic Stuttgart S-Bahn map (Fig. 5.2) were tested. The reason for that is that the difference between those two is the greatest, while also having acceptable results. As shown in Fig. 5.1 the result for the Berlin map is rather unpleasant, since a lot of intersections were introduced. However, the general shape of the map, as well as the properties of the map in general are similar to the other results and relatively good considering the size. The tested shapes are shown in Fig. 5.3. The results (Fig. 5.4 - 5.10) show that for both maps (Vienna and Stuttgart), the general shape has been properly adjusted. However some unexpected intersections were introduced during computation, which based on these results seem to stem from either indentations or narrow contours. These intersections either happen on the map itself where they would cause confusion as to whether or not there is a station (since intersection usually indicate those), or they happen between the map and the contour where they break out of the given shape.



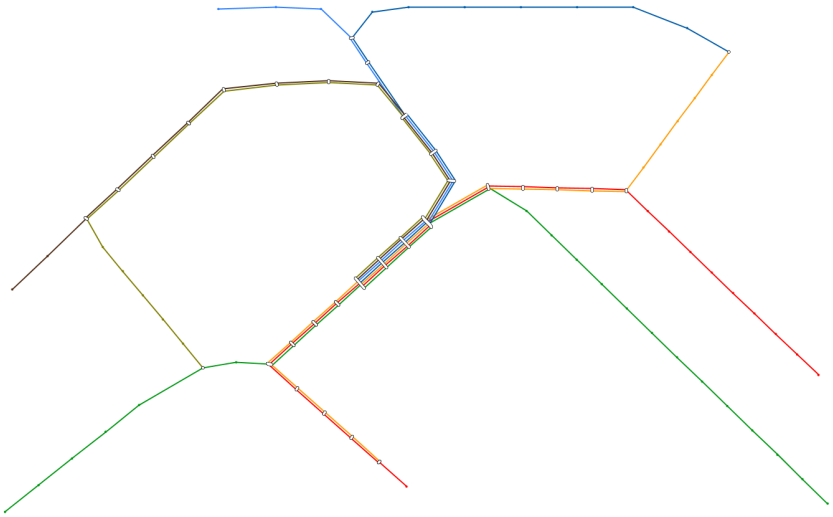
(a) Overview



(b) Zoomed-in view

Figure 5.1: Berlin's schematic metro map after reshaping into St. Stephen's Cathedral shape

(a) Vienna's geographically correct metro map



(b) Stuttgart's schematic S-Bahn map

Figure 5.2: Initial maps used as input

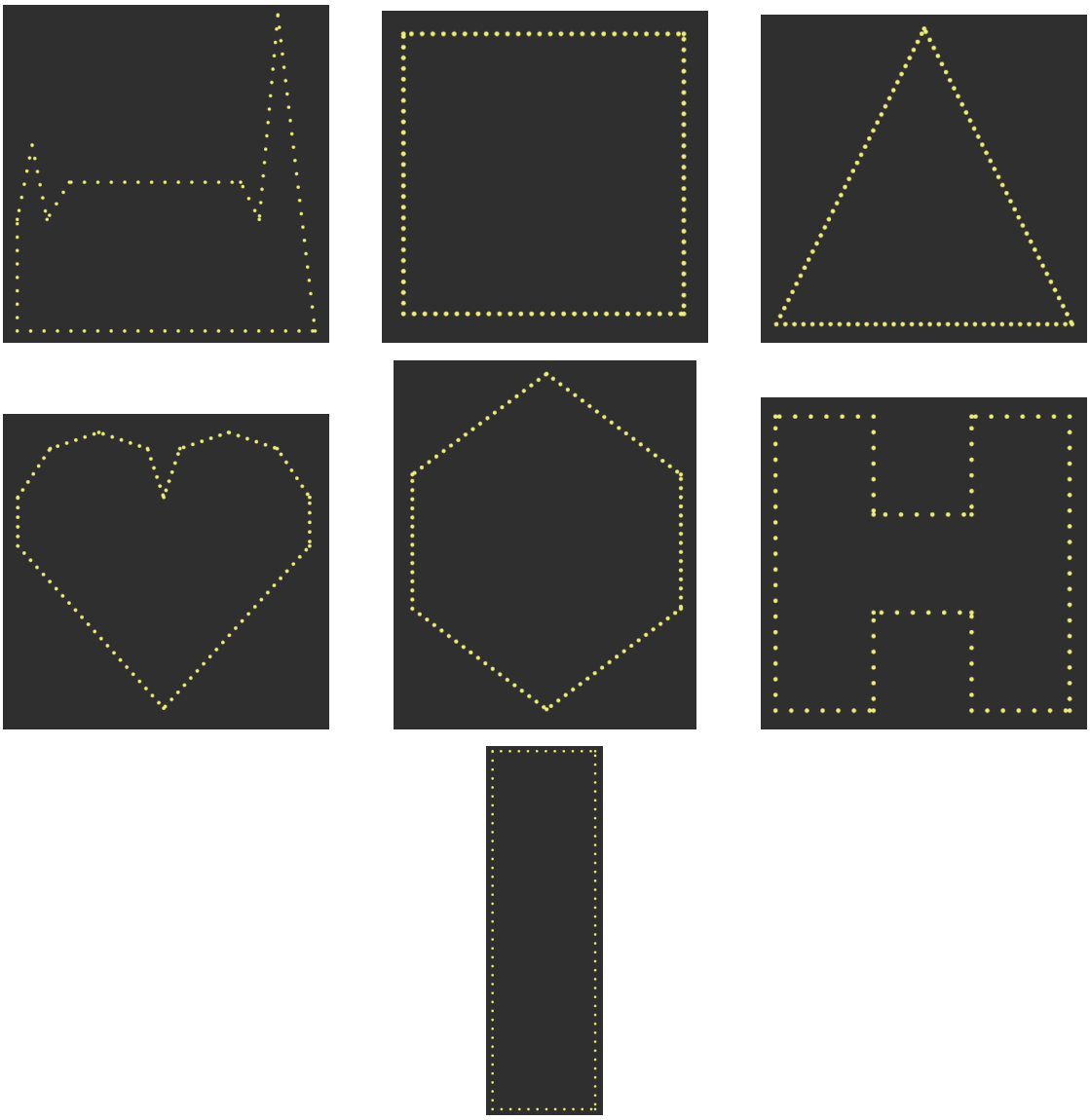
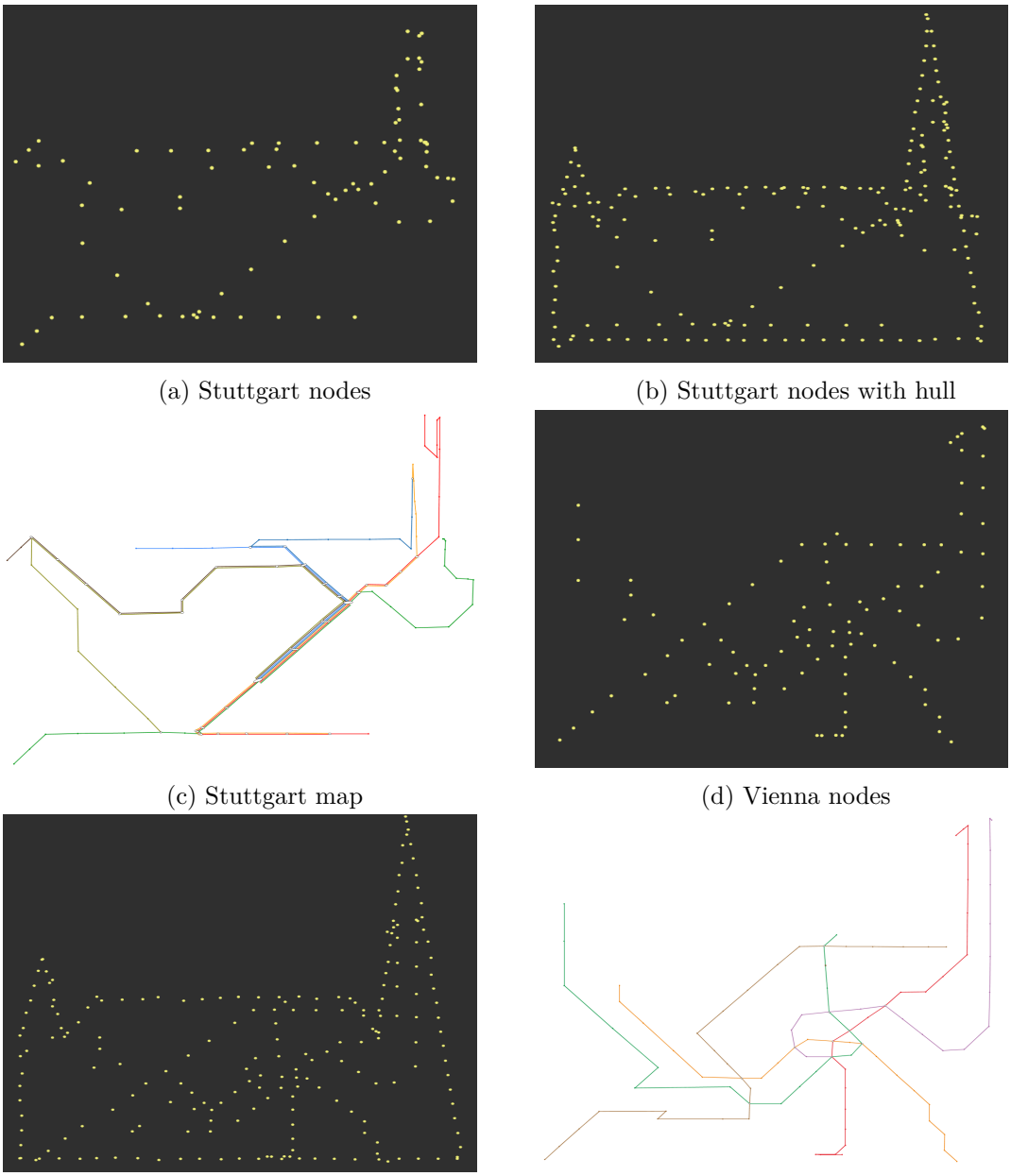


Figure 5.3: Shapes used as input



(a) Stuttgart nodes

(b) Stuttgart nodes with hull

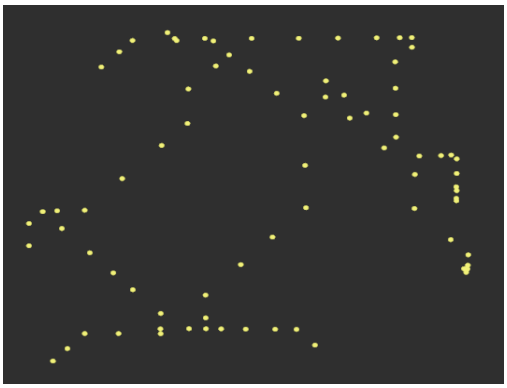
(c) Stuttgart map

(d) Vienna nodes

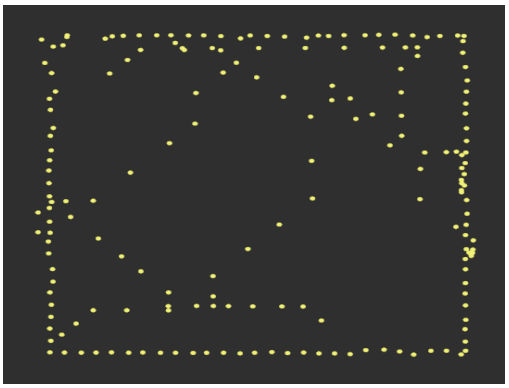
(e) Vienna nodes with hull

(f) Vienna map

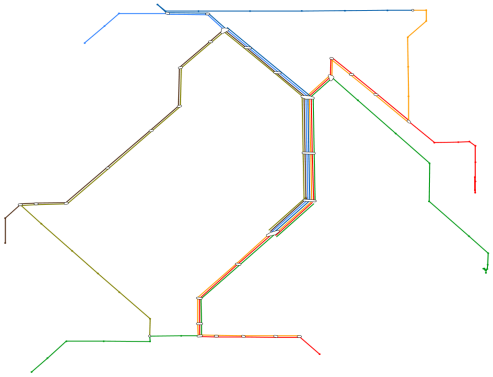
Figure 5.4: Results for St. Stephen's Cathedral shape



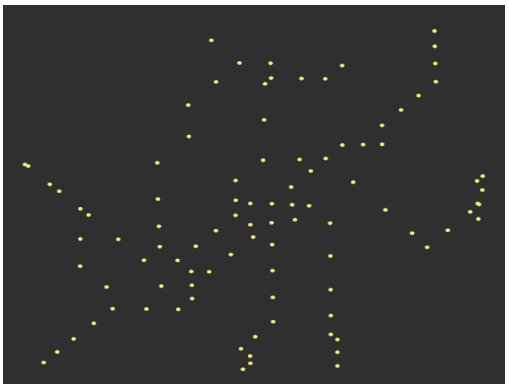
(a) Stuttgart nodes



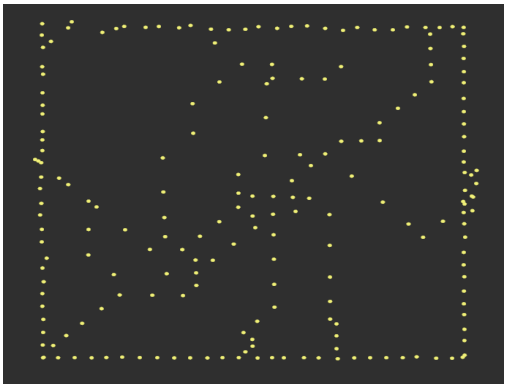
(b) Stuttgart nodes with hull



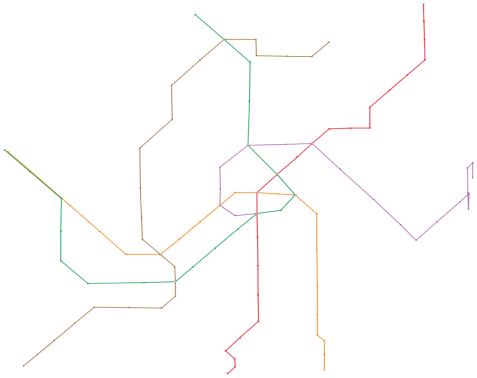
(c) Stuttgart map



(d) Vienna nodes

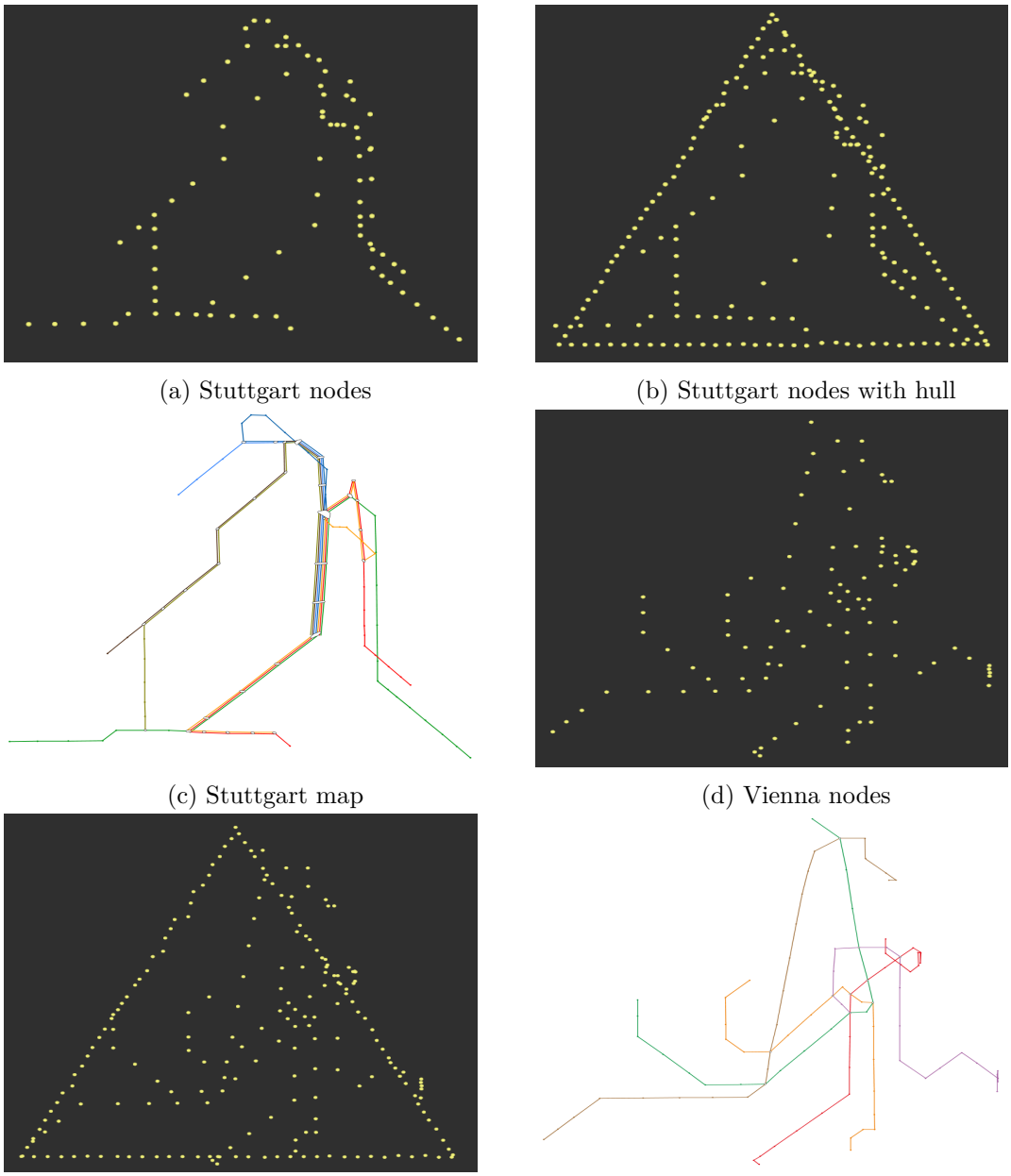


(e) Vienna nodes with hull



(f) Vienna map

Figure 5.5: Results for square shape



(a) Stuttgart nodes

(b) Stuttgart nodes with hull

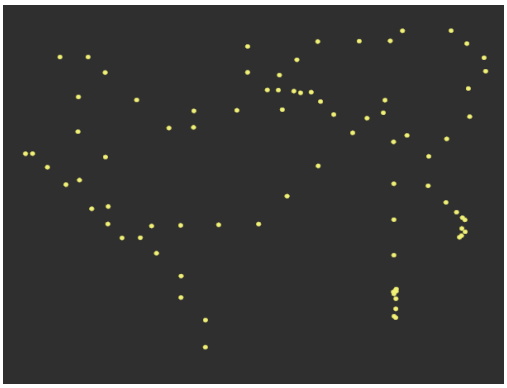
(c) Stuttgart map

(d) Vienna nodes

(e) Vienna nodes with hull

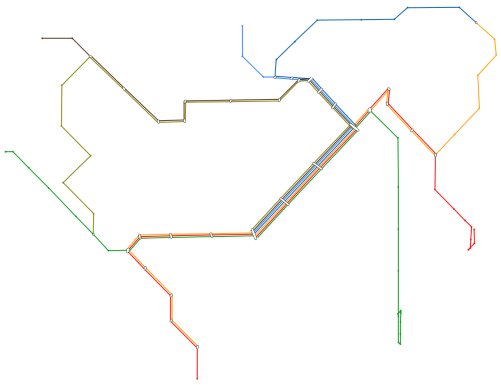
(f) Vienna map

Figure 5.6: Results for triangle shape

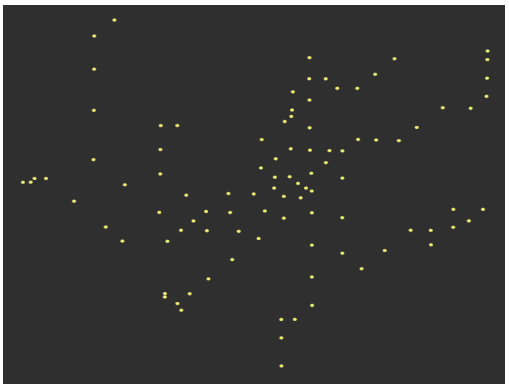


(a) Stuttgart nodes

(b) Stuttgart nodes with hull

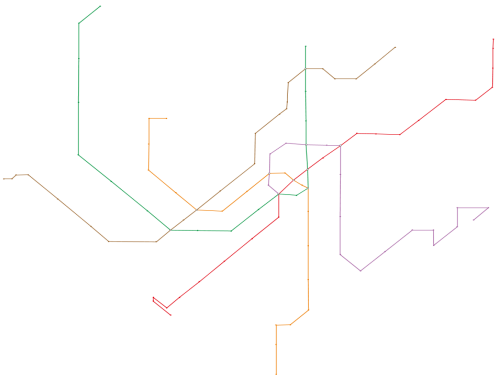


(c) Stuttgart map



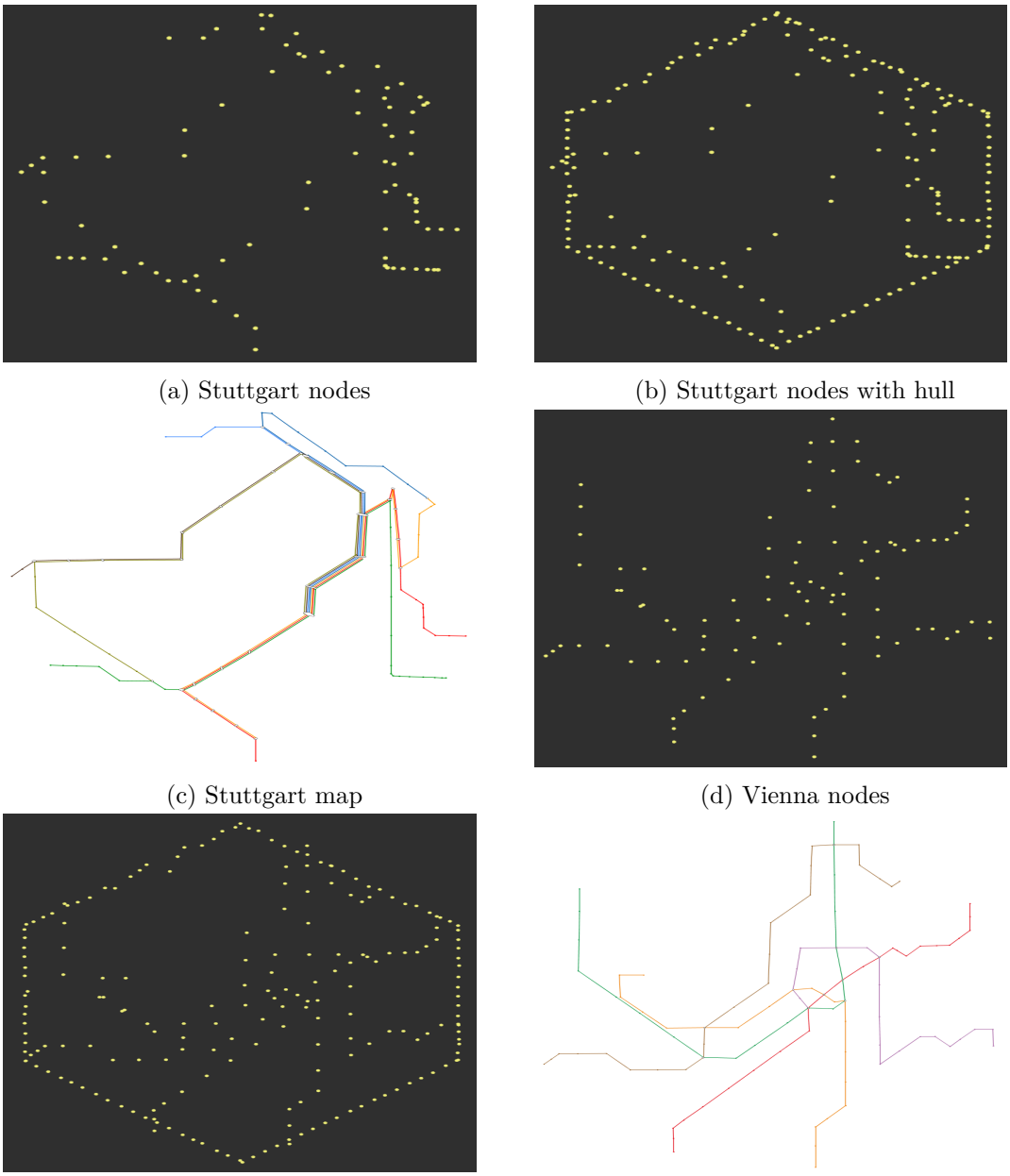
(d) Vienna nodes

(e) Vienna nodes with hull



(f) Vienna map

Figure 5.7: Results for heart shape



(a) Stuttgart nodes

(b) Stuttgart nodes with hull

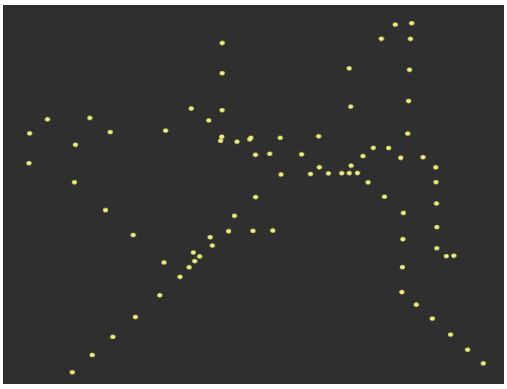
(c) Stuttgart map

(d) Vienna nodes

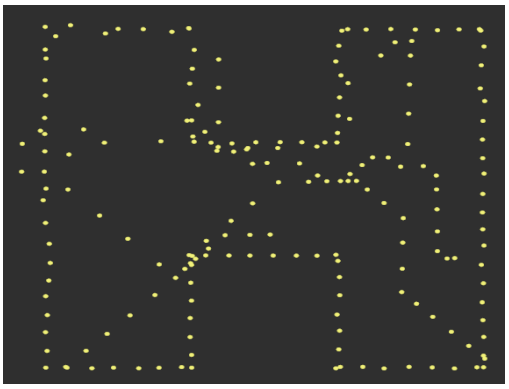
(e) Vienna nodes with hull

(f) Vienna map

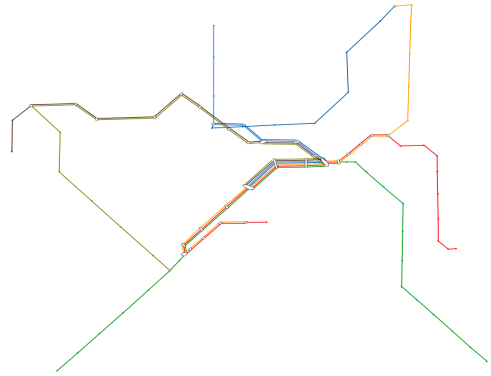
Figure 5.8: Results for hexagon shape



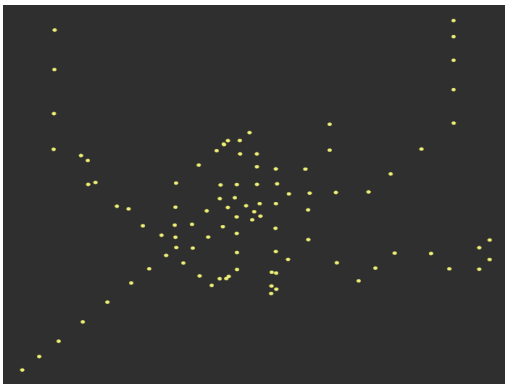
(a) Stuttgart nodes



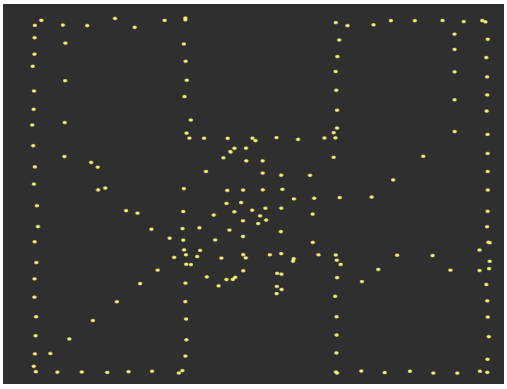
(b) Stuttgart nodes with hull



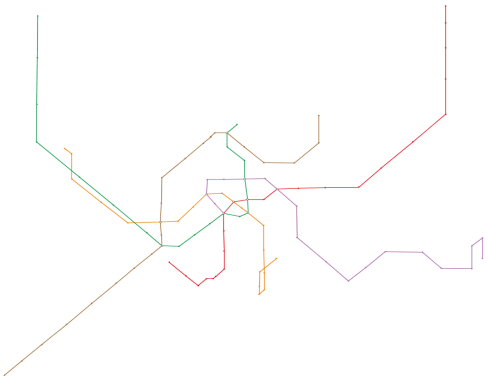
(c) Stuttgart map



(d) Vienna nodes

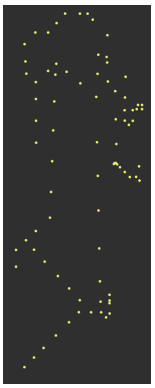


(e) Vienna nodes with hull

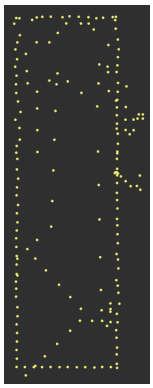


(f) Vienna map

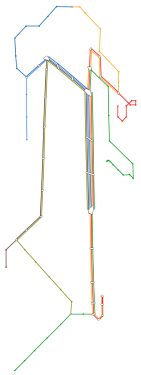
Figure 5.9: Results for H-letter shape



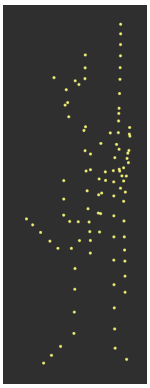
(a) Stuttgart nodes



(b) Stuttgart nodes with hull

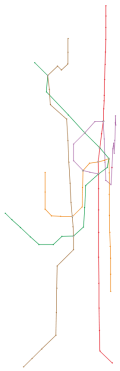


(c) Stuttgart map



(d) Vienna nodes

(e) Vienna nodes with hull



(f) Vienna map

Figure 5.10: Results for narrow rectangle shape

CHAPTER 6

Conclusion & Future Work

This chapter acts as a conclusion of the thesis and furthermore addresses areas that could be improved and methods to potentially do so.

6.1 Conclusion

The goal of the thesis was to find an automated method that uses a map and a contour as input and that generates a metro map confined within the given contour. The idea was to create a result that can be used as it is or as a basis for smaller adjustments by a designer. In general the introduced method does just that, however there certainly are some limitations. For one, as mentioned in the previous chapter, larger graphs can lead to unsatisfactory results. Beyond that the limitation of only having a contour without holes can certainly be a drawback for some use cases. However as already mentioned and as shown in the figures of the previous chapter, the introduced method for maps of a size similar to the one of Vienna's metro system creates satisfying solutions to the initially proposed problem.

6.2 Future Work

6.2.1 Selecting a Starting Point

In the current version of the algorithm the hull's starting point is always the bottom-most point and in case there are several on the same level, the left-most one is chosen. This means that the contour has to start with the point that corresponds to the calculated one, which leads to the problem that different maps that are supposed to be reshaped into the same contour possibly each need a separate deceleration of that contour. While this is not a huge problem it is inconvenient and should probably be addressed in case further iterations of this algorithm are created.

6.2.2 A different Hull

The convex hull is a good way to create a hull for a map that is rather dense, since in that case not much empty space would be included. However, since that may not always be the case it would be advantageous to use a hull that removes as much empty space as possible, so the contour later on is filled more completely and therefore more recognizable by the map alone. One such method is the alpha hull. Figure 6.1 shows the potential problem of using the convex hull, by highlighting the difference between the empty spaces.

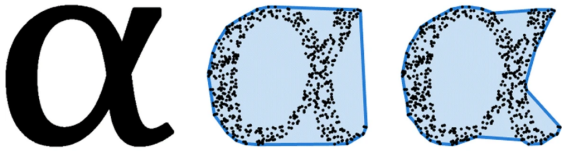


Figure 6.1: Difference between convex hull (middle) and an alpha shape (right) [GBB18]

6.2.3 Splitting the process

The previously mentioned improvements are pretty sure to accomplish better results or are simply more convenient. This idea is more theoretical. Since the algorithm can be applied on the result of itself it is possible to create a complex contour in a couple of steps, rather than doing it in one iteration. Since narrow areas and indentations seem to cause problems when looking at the results the approach for such shapes could be to first create a basic shape and try in further steps to slowly adapt to the desired shape by only expanding, if possible. Whether or not this approach would accomplishes better results remains to be seen.

6.2.4 Avoiding Intersections

As can be seen from the results, the algorithm does sometimes create new intersections, especially when working with a more complex and larger graph. The current way in which these intersections are checked is to look at the distance between a node and every other edge that it itself is not part of and see if the minimum distance that was given still remains after each iteration of movement. The movement is however only restricted by the weights and not set to a specific maximum value. That means that it is still possible to cross an edge in a single iteration by simply moving further than expected. Therefore a possible way to better the intersection problem is to restrict the movement of a node per iteration to a custom value, depending on the usual distances prevalent within the graph. While this may not solve all intersections it would most likely improve the result.

6.2.5 Improving Edge Lengths

The edge length of metro maps is usually roughly the same across the whole graph. The results of the algorithm however show that there are plenty of edges with highly varying edge lengths. This is the case, even after including an edge length constraint in the Deformer. A possible improvement would be to add that constraint to the Reshaper as well. However, the Reshaper already takes up a lot of computation time, especially with larger maps, meaning that adding more calculations there may not be the best solution. It would therefore be necessary to either weigh the benefits with the potential consequences or find a different solution altogether.

List of Figures

List of Tables

List of Algorithms

Bibliography

[CDS12] Siu-Wing Cheng, Tamal K. Dey, and Jonathan Richard Shewchuk. Delaunay mesh generation. In *Chapman and Hall / CRC computer and information science series*, 2012.

[DEL] Delaunay-Triangulator. `https://github.com/jdiemke/delaunay-triangulator`. Accessed: 2020-03-03.

[EJM] Efficient Java Matrix Library (EJML). `http://ejml.org/wiki/index.php?title=Main_Page`. Accessed: 2020-03-03.

[GBB18] James D. Gardiner, Julia Behnsen, and Charlotte A. Brassey. Alpha shapes: determining 3d shape complexity across morphologically diverse structures. *BMC Evolutionary Biology*, 18(1), December 2018.

[GFV13] Helen Gibson, Joe Faith, and Paul Vickers. A survey of two-dimensional graph layout techniques for information visualisation. *Information Visualization*, 12:324 – 357, 2013.

[GM97] Sarah F. F. Gibson and Brian Mirtich. A survey of deformable modeling in computer graphics. 1997.

[GRS] Graham Scan. `https://www.geeksforgeeks.org/convex-hull-set-2-graham-scan/`. Accessed: 2020-03-03.

[HMdN04] Seok-Hee Hong, Damian Merrick, and Hugo A. D. do Nascimento. The metro map layout problem. In *InVis.au*, 2004.

[HS52] Magnus R. Hestenes and Eduard Stiefel. Methods of conjugate gradients for solving linear systems. pages 409–412, 1952.

[II09] Takeo Igarashi and Yuki Igarashi. Implementing as-rigid-as-possible shape manipulation and surface flattening. *J. Graphics, GPU, & Game Tools*, 14:17–30, 2009.

[IMH05] Takeo Igarashi, Tomer Moscovich, and John F. Hughes. As-rigid-as-possible shape manipulation. 2005.

[LCF00] John P. Lewis, Matt Cordner, and Nickson Fong. Pose space deformation: a unified approach to shape interpolation and skeleton-driven deformation. In *SIGGRAPH '00*, 2000.

[Nöl14] Martin Nöllenburg. A survey on automated metro map layout methods. 2014.

[OMM] Open Metro Maps. `https://www.openmetromaps.org/`. Accessed: 2020-03-03.

[Sel13] Ivan W. Selesnick. Least squares with examples in signal processing 1. pages 1–2, 2013.

[SRMOW11] Jonathan M. Stott, Peter Rodgers, Juan Carlos Martinez-Ovando, and Stephen G. Walker. Automatic metro map layout using multicriteria optimization. *IEEE Transactions on Visualization and Computer Graphics*, 17:101–114, 2011.

[SWM+14] Rajsekhar Setaluri, Yu Wang, Nathan Mitchell, Ladislav Kavan, and Eftychios Sifakis. Fast grid-based nonlinear elasticity for 2d deformations. In *SCA '14*, 2014.

[T.D14] T.D. DeRosea and M. Meyer. Free-form deformation (ffd), 2014.

[WC11] Yu-Shuen Wang and Ming-Te Chi. Focus+context metro maps. *IEEE Transactions on Visualization and Computer Graphics*, 17:2528–2535, 2011.

[WN19] Hsiang-Yun Wu and Benjamin Niedermann. A survey on computing schematic network maps : The challenge to interactivity. 2019.

[WP16] Yu-Shuen Wang and Wan-Yu Peng. Interactive metro map editing. *IEEE Transactions on Visualization and Computer Graphics*, 22:1115–1126, 2016.

[Wu16] Hsiang-Yun Wu. Focus+context metro map layout and annotation. In *SCCG '16*, 2016.

[Wür14] J. Würzburg. Realtime linear cartograms using least-squares optimisation. 2014.

[WXW+06] Yanlin Weng, Weiwei Xu, Yanchen Wu, Kun Zhou, and Baining Guo. 2d shape deformation using nonlinear least squares optimization. *The Visual Computer*, 22:653–660, 2006.