Master Thesis

Interactive Design of Metro Maps

Tim Janiak

Date of Submission:	01. Dezember 2021
Advisors:	Prof. Dr. Alexander Wolff
	Dr. Thomas van Dijk

Julius-Maximilians-Universität Würzburg
Lehrstuhl für Informatik I
Algorithmen und Komplexität

Abstract

Metro maps are one of the main ways for people using the public transport system to orient themselves. For these maps to be effective, they need to give a clear overview of the network's structure while abstracting away unnecessary detail. Designing such a map is a complex task, the automation of which has been widely studied. Despite this, human designed maps are usually still of higher quality then those created algorithmically. This is in part due to the fact that there can always be portions of the map where deviations from the optimal solution – according to the implementation – are preferable. To deal with these cases, we pursue an interactive approach for designing metro maps.

Since the metro map layout problem is computationally hard, we modify an existing heuristic to accept feedback of a designer and immediately incorporate it into the drawing. This heuristic does not always find a solution but has a low runtime. We discuss a variety of ways a designer may want to interact with the map and implement them to be used intuitively through a graphical interface. This way, the designer can add restrictions to the drawing where they deem appropriate and our prototype will try to respect them while still creating a good map.

We evaluated this system to determine how often it fails to find a solution, how long the calculations takes, and whether the tools for interaction help create a better drawing. Our approach struggles with large networks, but on small and medium sized ones it achieves a high success rate in interactive runtimes. Here, the tools can be used to improve metro maps, even in few steps. The inclusion of a human also allows our system to appropriately handle network specific peculiarities. We also showcase some situations, where our tools produce unexpected results and explain how to circumvent them. Finally, we conclude by discussing how our approach could be expanded to provide more functionality.

Zusammenfassung

Liniennetzpläne spielen eine zentrale Rolle für die Orientierung bei der Nutzung öffentlicher Verkehrsmittel. Damit solche Karten gut verwendbar sind, müssen sie die Struktur des Netzwerks wiedergeben, ohne dabei unwichtige Details zu zeigen. Solche Karten zu entwerfen ist eine dementsprechend komplexe Aufgabe und das Automatisieren dieses Prozesses wurde umfangreich erforscht. Dennoch sind Karten, die algorithmisch erstellt wurden, in der Regel von niedrigerer Qualität als die von erfahrenen DesignerInnen. Das ist zum Teil darauf zurückzuführen, dass es immer Teile einer Karte geben kann, bei denen es sinnvoll ist, von der laut Algorithmus optimalen Lösung abzuweichen. Um mit solchen Fällen umgehen zu können, verfolgen wir einen interaktiven Prozess zum Entwerfen von Netzpläne.

Da das Erstellen von Netzplänen eine hohe algorithmische Komplexität hat, wandeln wir eine heuristische Lösung so ab, dass sie Anregungen von DesignerInnen sofort in die Karte integrieren kann. Die Heuristik findet dabei zwar nicht immer eine Lösung, hat aber eine kurze Laufzeit. Wir implementieren verschiedene Möglichkeiten, um intuitiv über eine grafischen Schnittstelle mit einem Netzplan zu interagieren. Dies erlaubt es DesignerInnen, Teile des Plans nach Bedarf zu verändern, woraufhin unser Prototyp versucht die Änderung mit den anderen ästhetischen Kriterien zu vereinbaren.

Wir werten unseren Ansatz dahingehend aus, wie häufig keine Karte erstellt werden kann, wie lange die Berechnungen brauchen und ob die von uns zur Verfügung gestellten Werkzeuge zur Verbesserung von Netzpläne beitragen können. Der Prototyp hat zwar Probleme mit großen Netzwerken, aber auf kleinen und mittelgroßen ist er meist erfolgreich, hat dabei eine geringe Laufzeit und ermöglicht dadurch eine Verbesserung der Qualität der Karte in wenigen Schritten. Das Einbinden eines Menschen hilft außerdem mit Besonderheiten des Netzwerks angemessen umzugehen. Wir präsentieren auch Situationen, in denen unsere Werkzeuge unerwartete Veränderungen hervorrufen und erklären, wie diese verhindert werden können. Abschließend besprechen wir, wie die Funktionalität unseres Ansatzes erweitert werden kann.

Contents

1 Introduction

In public transport, schematic maps of the network are an essential tool for travelers to plan their route. The design of such maps plays a big role in its efficacy. As such, their creation is a time-intensive task and requires skilled designers who have to consider many factors to produce a legible map that is apt for orientation. An example for Würzburg's tram network, represented in a typical metro map style[1] as opposed to the geographic layout[2] is shown in Fig. 1.1. The attempt to automate the creation process of such maps has resulted in numerous approaches, aiming for slightly varying design criteria, using different algorithms. Despite this, most computed maps can still benefit from human corrections, necessitating a designer anyways [Nöl14]. To better incorporate this human feedback, we investigate an interactive approach of designing metro maps, an area that has been studied much less widely.

In particular, we algorithmically aid a designer during the creation of a metro map layout. That is, for an embedded graph, representing a network of stations connected by the routes available for public transit, we try to create a drawing that follows some layout rules which make it fit to be used as a metro map. In our case, these aesthetic goals include preservation of topology, octilinearity (curves follow one of eight directions), a minimum distance between vertices, monotony of edges and geographic accuracy. The designer can then repeatedly improve the drawing by adding some restrictions, like changing the placement of vertices or the shape of edges, which are integrated into the map in real time.

To accomplish this, we adapt the methodology used by Bast et al. [BBS20] to create metro map layouts. They assign each station a position in an octilinear grid graph and use shortest paths between those grid nodes to generate the curves for the edges. By changing the costs of edges in the grid, specific design criteria can be achieved. We can use this adaptability to incorporate the additional constraints requested by the designer. To find a good drawing this way, the authors propose a heuristic algorithm with a low runtime, which is required for use in an interactive process. The heuristic routes the edges through the grid one after another in an appropriate order, with previously added edges acting as obstacles.

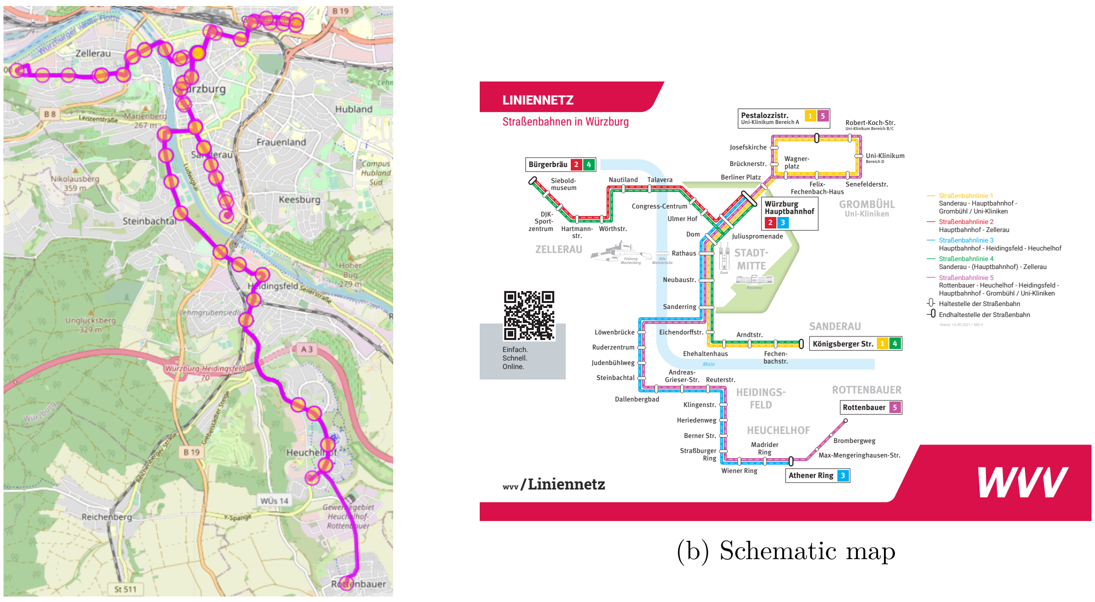
We create a working interface and implementation that uses this procedure to create an initial drawing. By recalculating the metro map after every modification through a designer, the change can be included with minor influence over the realization of other aesthetic rules. As means of interactions, we allow users to modify the parameters of the algorithm and provide multiple different tools of acting on the drawing directly in

[1]WVV – Liniennetz Straßenbahnen in Würzburg `https://www.wvv.de/mobil-b2c/liniennetzplan-wuerzburg/` (Nov. 2021)
[2]Based on OpenStreetMap data from Nov. 2021; data available under ODbL

an intuitive way. For example, they can drag vertices around to set their position in the drawing or click on edges to prevent bends in their curve.

This prototype can thus be used to lay out metro maps of higher quality and including more varied design criteria than those created by the algorithm on its own. Thanks to the automated integration of any requested changes, it also increases the designer's productivity, making it faster than creating a map completely manually. The iterative nature of our process tries to combine the advantages of both manual and algorithmic methods, more so, than simply providing a designer with a computer generated map as inspiration or as something to be fixed. In general, it is very hard to formalize a set of aesthetic requirements that produce good maps for every input network, especially when aiming for a reasonable runtime. So human designers still play an important role in the map creation process. With this in mind, our work shows that interactive approaches to metro map design are a promising idea that should be investigated further.

The remainder of this work is structured as follows. Related work is presented in Chapter 2. In Chapter 3, we then explain the approach of Bast et al. and our modifications to it in more detail and list the ways we allow designers to edit the drawing. The performance of our prototype is evaluated in Chapter 4, with regards to user experience and the quality of resulting maps. Lastly, Chapter 5 summarizes and interprets our results. Here, we also give an outlook for potential future work.



(a) Geographic network

(b) Schematic map

Fig. 1.1: Tram network in Würzburg. While (a) shows the geographic positions of stations and the paths the rails take, the simplified layout of the official map (b) allows users to easily understand the structure of the network. The geography is roughly preserved to help with orientation.

2 Related Work

This section provides an overview of publications related to the topics covered by this thesis, such as different parts of the metro map creation process and low-runtime or interactive approaches to this.

Problem Formulation Schematic maps play a significant role in public transit when it comes to planing routes [Guo11]. As such, their design has been extensively studied, especially the automatic schematization of railway networks into metro maps. There have been a lot of different approaches focusing on varied aspects of the process, a major part of which deals with where on the map stations are drawn and how they are connected. This, in combination with the labeling of stations has initially been called the metro map layout problem [HMDN04] and multiple algorithms for solving this have been proposed [SRMOW10], [WTH$^+$13].

However, since map labeling has seen a lot of research in general and because it can be performed somewhat independently from the graph layout, Nöllenburg [Nöl14] treated these two tasks as separate subproblems in the map creation process. A third step is drawing metro lines onto the curves of edges, in a way that minimizes crossings among them. Our approach however only deals with the layout problem. That is, we try to create a drawing for a graph representing the rail network, while following some design constraints.

For this problem, Nöllenburg provides a list of such design principles that are commonly applied. These pertain to orientation, uniformity, and legibility. Examples for the first include preservation of network topology and minimizing the relative displacement of stations to their neighbors. Uniformity is frequently realized by restricting edge orientations and using consistent edge lengths. Legibility can be promoted by drawing lines as monotone as possible and using a large angular resolution. Different algorithmic approaches are better suited, depending on which principles are pursued and whether they present hard constraints or are seen as something to optimize.

Optimality-Runtime Dichotomy Algorithms also differ in whether they aim for a high degree of optimization or for low runtimes. For example, Hong et al. [HMdN06] modified spring-/force-based algorithms to account for the specific requirements of metro map layouts. They suggest multiple methods, each aiming for a different combination of the design criteria minimum distance between vertices, straight edges, topology preservation, and octilinearity. While this approach does not necessarily achieve these criteria, it still creates a decent drawing in seconds.

Somewhat inversely to that approach, Nöllenburg and Wolff [NW10] create the map layout that guarantees octilinearity, topology preservation, and a minimum edge length, by using a mixed-integer program (MIP). The additional soft constraints (minimization of bend count, preservation of relative station position, and minimization of edge length in the drawing) can then be assigned an importance by changing cost factors in the MIP. Due to the size of the MIP, this approach can take hours to finish – especially when solving to proven optimality. However, suboptimal but early solutions, i.e. after seconds or minutes, depending on network size, are usually only marginally worse.

This commonly observed trade-off between optimality of a metro map layout and the required runtime has been demonstrated across a multitude of approaches by Wu et al. [WNTN19]. In this paper, the authors provide an overview of many state of the art layout methods in regards to runtime and problem scope, i.e. the complexity of combinatorial and geometric design criteria and whether they are optimized locally or globally. Besides showing that some methods sacrifice quality to reduce calculation times, they also show a trend toward approaches both becoming faster and considering more complex criteria by grouping publications by year of publication.

Interactive Approaches Even though some methods integrate multiple design criteria and optimize for those globally, the resulting maps are not necessarily on par with manually created ones and thus should – if possible – be revised by a designer [Nöl14]. This may be due to edge cases where drawings that are suboptimal according to the algorithm make for better maps or simply because some design criteria were not implemented or their importance was weighted incorrectly. Since there still is the need for a human to take part in the creation of metro maps, an approach that interactively incorporates feedback of a designer seems sensible.

Following this idea, Chivers and Rodgers [CR11] created a mobile application that allows users to create and position stations and connect them with edges. During this graph creation process, users can press a button to initiate an optimization step, that snaps stations to a free spot on a grid and then calculates the aesthetic score for that station and adjacent edges, which is based on octilinearity, edge crossings, straightness of lines, feature occlusion, and consistent edge lengths. For each station, they iteratively choose the best grid point in the vicinity of the original position, to optimize this score. The runtime of this process was not reported. The authors also mainly use their approach for metro map metaphors, that is to schematically present data without inherent positional data like relationships in social networks or which systems are shared by components of a website.

For editing geographic metro maps, Wang and Peng [WP15] provide a framework in which users can set the locations for some stations. Through least-squares optimization, they then let these stations approximate the provided position while also pursuing short and straight edges as well as evenly spaced vertices. They can create curvilinear and, by subsequent rotation of edges, octilinear drawings in less than a second. When designing their system, the authors also focused on the user experience by providing stability of unrelated regions of the layout when moving a station and consistency between user expectations and the resulting drawing.

Prioritization of Runtime Another important factor for the usability of an interactive tool is the system response time (SRT), i.e. how long it takes to produce an output after the user takes an action. As a guideline [Nie94], SRTs below 0.1 s make the user feel as though they are directly affecting the output without noticing that calculations are being performed. A SRT between 0.1 s and 1 s makes the delay noticeable but does not feel interruptive to the task the user is trying to accomplish. For higher SRTs, there should be an indication that the system is still working. While this does break the flow of users, they are usually willing to stay attentive to the system for about 10 s. Any operation longer than that should provide the ability to cancel it and show a percent-done indicator, or better yet an estimated finishing time. Notably, for graphical problem solving tasks, while the mean SRT matters, its variance across multiple uses of a function does not seem to be correlated with the time it takes users to find a solution [GS81]. However, according to Goodman et al., these results are not necessarily applicable to designing metro maps, since the way users engage with a task and thus how they handle SRTs, largely depends on the type of that task. For good usability in our case, the layout algorithm should not take longer than a few seconds, preferably with the ability to recalculate subsections of the drawing even more quickly. This would allow for true interactivity when editing the graph in a way that only causes local changes.

There are multiple approaches that achieve such short runtimes. For example, van Dijk and Lutz [vDL18] apply their method of drawing graphs with specified edge lengths to metro maps. They first assign each edge an octilinear direction and then optimize for each edge to align with its direction and for all edges to have the same length using the least squares method. This creates nearly octilinear drawings with well distributed stations in a few milliseconds.

Another fast (<1 s reported runtime) algorithm using least squares is that of Wang and Chi [WC11]. Here, the drawing focuses on the route between user provided start and destination stations. They first optimize for consistent edge lengths (with edges that are not on the route being shortened), high angular resolution, and low geographic displacement of stations. Subsequently they rotate every edge to the nearest octilinear direction.

Besides using a integer linear program (ILP), a recent publication by Bast et al. [BBS20] also includes a heuristic algorithm that has a low runtime and allows for recalculations of sections of the drawing. Here, the authors draw each edge by calculating a shortest path in an auxiliary grid graph in which edge costs model the design constraints. This approach is explained in detail in Chapter 3.1. Since then, the same authors [BBS21] expanded on their method to be able to also produce hexalinear and orthoradial layouts. They also experimented with using different sparse grids to reduce the problem size. While this barely impacted map quality, it also did not consistently reduce runtimes for either the optimal solution via ILP or for the heuristic that we will use. The original idea of their algorithm has the additional advantage of being easy to conceptually understand. The restriction that all edges become paths on a set octilinear grid makes it possible for users to predict outputs the system produces when interacting with the graph, which allows them to more efficiently use the tool.

3 Methodology

In this chapter, we explain our prototype and the algorithms behind it. We present relevant parts of the approach of Bast et al., which user interactions we allow, and how we realize them. Lastly, we give additional insight into some parts of our implementation.

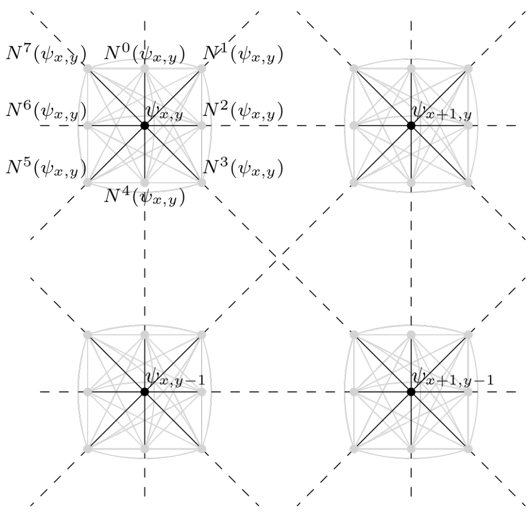
3.1 Metro Maps on Octilinear Grid Graphs

Bast, Brosi, and Storandt propose the idea of utilizing a grid graph to create a metro map drawing, by assigning each station a vertex in an octilinear grid graph and using paths in this grid as curves for drawing the edges [BBS20]. They solve their formalization of this problem optimally using an ILP, but also provide a much faster, well performing heuristic approach using shortest path routing on the grid graph. Regarding the criteria from Chapter 2, here the approach guarantees octilinearity, topology preservation, and map density, while trying to minimize edge length and the number of edge bends as well as maintaining geographical accuracy as best possible. In the following, we explain how their approach combines and achieves these constraints and afterwards note how we changed it to better fit the interactive use case. Lastly, we explain the different components of our interface.

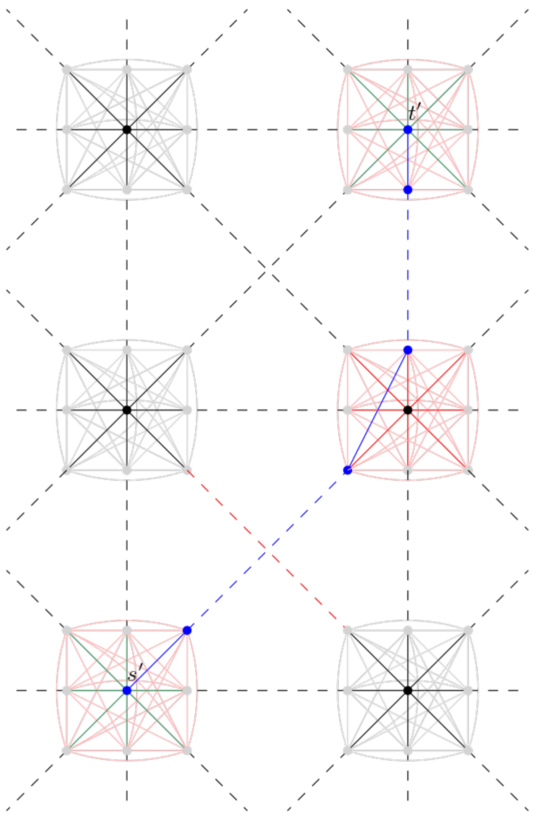
3.1.1 Heuristic Approach

Given an input graph $G = (V, E)$ with initial drawing $\mathcal{D}^*_G = (P^*, C^*)$, we want to find a drawing $\mathcal{D}'_G$ that fulfills the listed constraints. There cannot be any vertices $v \in V$ with $\deg(v) > 8$, since the approach preserves topology and in an octilinear drawing the curves of at most eight edges can leave any vertex. Furthermore, because the curves in the resulting drawing will not intersect, the input graph must also be planar.

Based on the area of the bounding box of the initial drawing A and a smallest distance between any two grid vertices D, Bast et al. set the grid graph $\Gamma = (\Psi, \Omega)$ as an $X \times Y$ grid with $X \times Y = \lceil A/D^2 \rceil$. A path in this grid will represent the curve for drawing an edge from the input graph. In order to encode all soft constraints as edge costs, they first introduce eight auxiliary vertices $N^0(\psi_{x,y}), \dots, N^7(\psi_{x,y})$ for every grid vertex $\psi_{x,y}$. All of these *port vertices* $N^i(\psi_{x,y})$ are connected to the corresponding grid vertex $\psi_{x,y}$. These original vertices are called *sink vertices*, with the edges between port and sink vertices labeled as *sink edges*. Furthermore, all port vertices of the same sink are connected via so called *bend edges*. These 9-cliques at the grid points are connected with neighboring (including diagonally) points through the corresponding ports. So there is a *grid edge* from the north port $N^0(\psi_{x,y})$ to the south port of the next clique above



(a) Vertex naming



(b) Edges after routing

Fig. 3.1: Sections of the grid graph. The sink edges are drawn with heavy lines, bend edges with light ones, and grid edges are dashed. Figure (a) shows the naming of the vertices in Γ with the port vertices only labeled for $\psi_{x,y}$. The blocked edges in the grid after routing an input edge (s, t) are shown in (b). The endpoints s and t are settled to the sink vertices s' and t'. The shortest path between those is colored blue. For the routing of the following input edges, none of the blue or red edges may be used. The green sink edges are still available to other input edges adjacent to s or t respectively.

$N^4(\psi_{x,y+1})$, one from the north-east port $N^1(\psi_{x,y})$ to the south-west of the clique to the top-right $N^5(\psi_{x+1,y+1})$, and so on. The structure of this graph is shown in Fig. 3.1a.

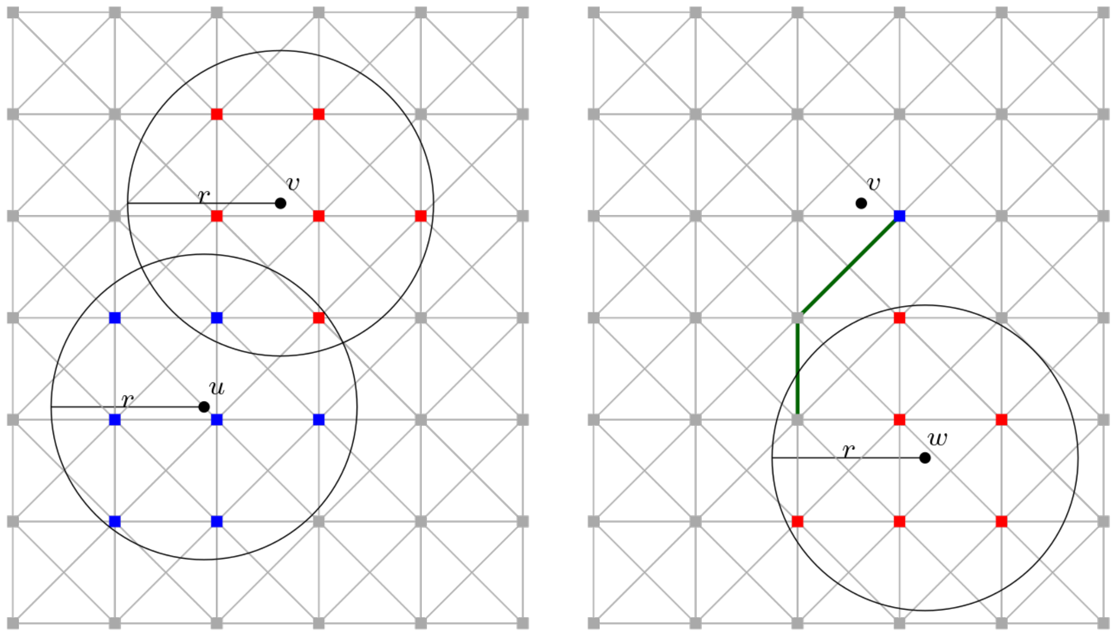
On this grid, all soft constraints can be encoded in edge costs. To promote short paths, the authors simply introduce a *hop cost* c_h for all grid edges. For a minimal amount of bends and to prefer shallow bends, they assign each bend edge a cost depending on the resulting angle of the bend in the path. For example, $(N^3(\psi_{x,y}), N^4(\psi_{x,y}))$ has the cost c_{45} and $(N^6(\psi_{x,y}), N^2(\psi_{x,y}))$ the cost c_{180}. Note that angles $\alpha > 180°$ look like $(360° - \alpha)$-bends, so we can use the equivalent cost and only need four costs $c_{45} \geq c_{90} \geq c_{135} \geq c_{180}$ which incentivize keeping angles as large as possible. A problem could arise, if the cost for a sharp bend is higher than that of two shallow bends which end in the same port: Then, a 90° bend could simulated as two 135° bends, for example. To prevent this, the costs must also obey $2c_{135} \geq c_{90}$ and $c_{180} + c_{135} \geq c_{45}$. Additionally, a cost c_s for each sink edge is necessary with $2c_s > c_{45}$ to prevent pathfinding through the sink instead of using bend edges, which also avoids bend costs. Since the input vertices will be mapped onto sinks in the grid, the sink edges can also be used to penalize moving vertices from their original position. When choosing a grid position for input vertex v, the cost of sink edges at $\psi_{x,y}$ are increased by $d(c_h + c_m)/D$, where c_m is some move cost and d is the euclidean distance between v in the initial drawing and $\psi_{x,y}$.

We will skip over the actual objective function and creation of the ILP here, as we only use the heuristic algorithm. This produces a drawing by finding a shortest path in the grid for each input edge, while disallowing paths over grid edges that were used for previous input edges. This means that edges will influence how the following edges will be drawn, so they need to be added in a sensible order. For this, the authors introduce the line degree $\mathrm{ldeg}(v)$ of input vertices as the sum of the number of lines on edges adjacent to v. Starting with a vertex v_0 with maximal line degree, they add all adjacent edges $(v_0, u_0), \dots, (v_0, u_k)$ to the edge order, so that $u_0, \dots, u_k$ are sorted decreasingly by line degree. They repeat this for different vertices v_i, until all edges are included, skipping over edges that are already part of the order. In round i, the vertex v_i has the highest line degree, only considering vertices adjacent to edges in the order and excluding $v_0, \dots, v_{i-1}$.

Now, the edges in the order are assigned a shortest path in the grid between a set of candidate grid points for each endpoint. So to route edge (s, t), the authors perform Dijkstra's algorithm between S and T, where S (and likewise T) contains all the sink vertices $\psi_{x,y}$ within a certain distance r around s (t) in the initial drawing, except for sinks which are already assigned to another input vertex. Vertices that are in both candidate sets are removed from the one where they have a greater distance to the corresponding edge endpoint. Additionally, if s (t) is adjacent to an already routed edge, it is considered *settled* since it already has a set position in the grid. In that case, there are no other candidates and thus $S = \{s\}$ $(T = \{t\})$. See Fig. 3.2 for an example of the candidate sets.

For the pathfinding between two sets of vertices S and T, instead of two vertices, we need to modify Dijkstra's algorithm slightly. To do so, we first add a dummy vertex s to the graph and connect it to all vertices in S, using edges with no cost. By starting the algorithm in s, we will have to traverse a vertex in S to reach T. Having multiple target vertices is also not a problem, since we can simply stop the algorithm as soon as any vertex in T is reached. Because vertices are considered in order according to the distance from s, the paths to the other vertices in T must be longer than the first one that was found. This way, we can simplify the case with two sets down to a normal application of Dijkstra's algorithm.

After an edge is routed, none of the sink or bend edges at any of the grid points the path passes through can be used for another input edge. Additionally, the grid edges that are part of the path and those crossed by diagonal edges may also not be used for routing again. Only the other sink edges of the grid vertex that s (t) settled on, are still available for other edges sharing that endpoint. Fig. 3.1b visualizes this for a short path. However, to preserve the circular edge ordering in the grid representation, some sink edges still get blocked, while the others will have an additional cost for every other adjacent input edges to also penalize bends at stations.



(a) Routing of input edge (u, v) (b) Routing of input edge (v, w)

Fig. 3.2: Candidates for grid positions when routing input edges. The labeled vertices represent the position of that input vertex in the initial drawing. In (a), the blue grid positions are all candidates that u can settle on, the red ones are candidates for v. After (u, v) was routed onto the green path in (b), the only candidate for v is the blue vertex it settled on. Here, the candidates for w are colored in red.

If some edge cannot be routed successfully, the authors restart the whole algorithm now using a shuffled edge order. Until one allows all edges to find a path, routing is repeatedly attempted using random edge orders. Once successful, they further improve the drawing through a local search. For this, they choose one vertex v and change the grid position it settled on to one of the 8 neighboring positions, reroute all edges adjacent to v and recalculate the costs across all edges. They try this for all input vertices and each of the positions (if no other vertex is settled there) and select the result with the lowest cost as the new drawing. This local search step is repeated until no more improvements are found, at which point the drawing is done.

The authors also propose using this algorithm on a simplified version of the input graph. For this so-called *deg-2 heuristic*, all vertices with degree 2 are contracted before routing the edges. In the drawing, they can then be reinserted equidistantly on the corresponding paths through the grid. This may however violate the map density constraint if too many vertices need to be inserted onto a short path. To combat this, they add a spring force on grid vertices connected by a path. If a path consisting of l grid edges cannot accommodate the k contracted stations, the cost of that path is increased by $(k + 1 - l)^2 c_c / 2k$, where c_c is a factor to tune this compression cost.

3.1.2 Modifications to the Heuristic

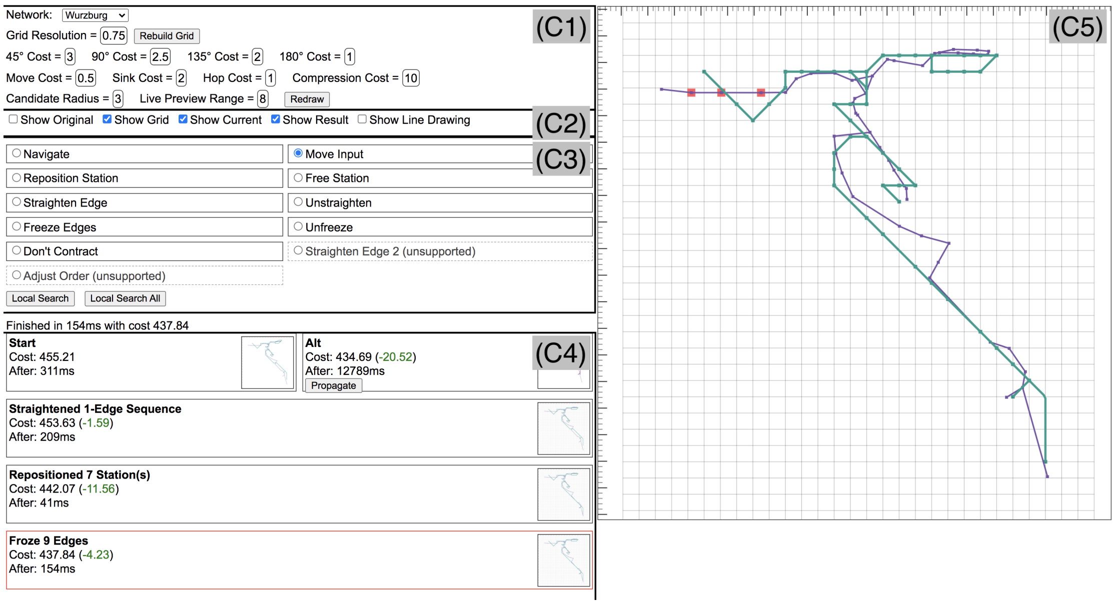
Our implementation of the heuristic algorithm differs slightly in each step, to better deal with restrictions added by the user. Besides allowing the user to modify parameters (like costs or grid sizing) directly, they can also influence the positioning of stations and shape of edges. The order in which edges are added has a high impact on whether a drawing can be found. Therefore, we modify the order to route edges affected by user added constraints earlier, as it may be more difficult to fit them into a drawing later on. Our method of determining the edge order is detailed in Section 3.3.

Candidate selection and routing itself are also modified for stations and edges edited by the user, depending on the tool, as explained in Section 3.2. In the rare case that routing is unsuccessful after some user interaction, we do not shuffle the edge order and retry, as that could lead to significantly different drawings for each randomization. Instead, we simply do not enforce the added restriction. Lastly, we exclude the local search step unless explicitly requested, due to runtime concerns and problems with subsequent interaction, as discussed in Section 3.2.5.

Graphical User Interface Figure 3.3 shows a screenshot of our website, the most important element of which is the canvas (C5) on the right. It is used to visualize the input graph as well as the resulting metro map and allows users to select the vertices and edges that they want to interact with. The selection and modification mechanism is determined by the currently selected tool (C3). For example, with the “Navigate”-tool, users can scroll up on the canvas to zoom in on the cursor location, scroll down to zoom out or click and drag the canvas to move the viewport to different parts of the drawing. The other tools can be used to add constraints to the drawing and are explained in detail in Section 3.2. The users can also influence the algorithm by changing the various cost parameters (C1) and choose which drawings they want shown (C2). The edit list (C4) is expanded by a new entry whenever a user interaction occurs. It names the type of edit, change in global cost and time required for the calculation. Users can also jump back to earlier versions or compare the looks of different ones by clicking the thumbnails on the right.

Graph Preprocessing and Representations We used the rail networks of London, Montreal, Sydney, Washington D.C., and Vienna based on GraphML files kindly provided by Nöllenburg, as well as the tram network of Würzburg[1], on which we did most of the testing. While we do use the geographic positions of stations for the initial drawing of the input graph, for edges we simply connect the endpoints with a straight line-segment and do not represent the actual route of the railway more accurately. This could potentially change the clockwise ordering of edges at a vertex. If this poses a problem, dummy vertices can be inserted on the offending edges with a position such that the original ordering is preserved. Similarly, if the initial drawing has intersecting edges (s,t) and (u,v), we add a vertex w at that point, remove the edges and instead add (s,w), (w,t),

[1]Manually edited, based on OpenStreetMap data from June 2021; data available under ODbL


Network: Wurzburg
Grid Resolution = 0.75 Rebuild Grid
45° Cost = 3 90° Cost = 2.5 135° Cost = 2 180° Cost = 1
Move Cost = 0.5 Sink Cost = 2 Hop Cost = 1 Compression Cost = 10
Candidate Radius = 3 Live Preview Range = 8 Redraw

☐ Show Original ☑ Show Grid ☑ Show Current ☑ Show Result ☐ Show Line Drawing

○ Navigate [✓ ◉ Move Input]
○ Reposition Station ○ Free Station
○ Straighten Edge ○ Unstraighten
○ Freeze Edges ○ Unfreeze
○ Don't Contract ○ Straighten Edge 2 (unsupported)
○ Adjust Order (unsupported)
Local Search Local Search All

Finished in 154ms with cost 437.84

Start
Cost: 455.21
After: 311ms

Alt
Cost: 434.69 (-20.52)
After: 12789ms
Propagate

Straightened 1-Edge Sequence
Cost: 453.63 (-1.59)
After: 209ms

Repositioned 7 Station(s)
Cost: 442.07 (-11.56)
After: 41ms

Froze 9 Edges
Cost: 437.84 (-4.23)
After: 154ms

Fig. 3.3: Overview of the components in our interface.

(u, w), and (w, v). An example for fixing these problems is show in Fig. 3.4. In input graphs that contain vertices with more than eight adjacent edges, they can be split into multiple, connected vertices at the same position, that distribute the edges among them. This was however not necessary for any of our networks. Users can view the resulting intersection-free, straight-line drawing based on the geographic positions of stations by checking “Show Original” at (C2).

The other checkboxes allow users to overlay other versions of the graph or meta information. The available options are demonstrated in Fig. 3.5. “Show Current” displays the graph on which the algorithm is actually run. This may be different from the initial drawing (the *original graph*), if degree 2 vertices are contracted or the geographic po-

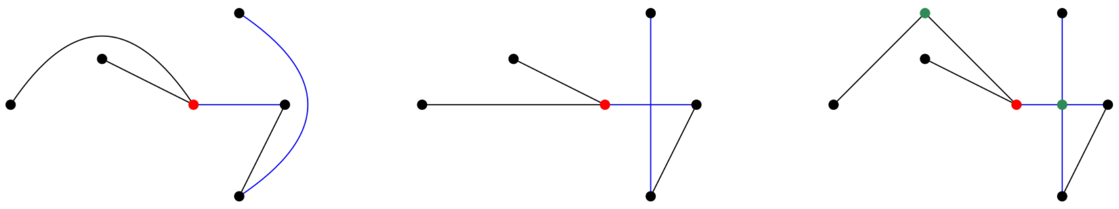


Fig. 3.4: Preprocessing of the input network. In the actual network on the left, rails run in a way which changes the topology when creating a drawing using the geographic locations of stations and representing edges as line-segments between them (middle). Here, the circular order of edges at the red vertex changes and an intersection of the blue edges is introduced. By adding new vertices, like the green ones on the right, we can create an initial drawing without intersection and – if required – one that respects the original circular edge orders.

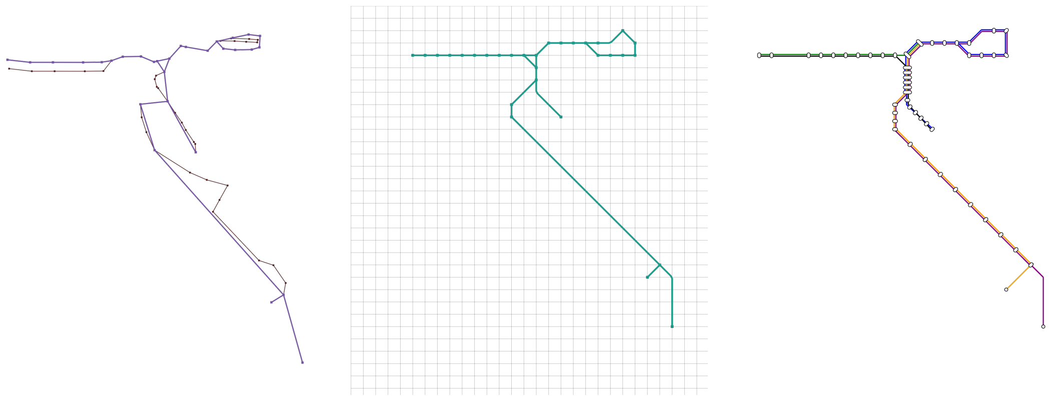
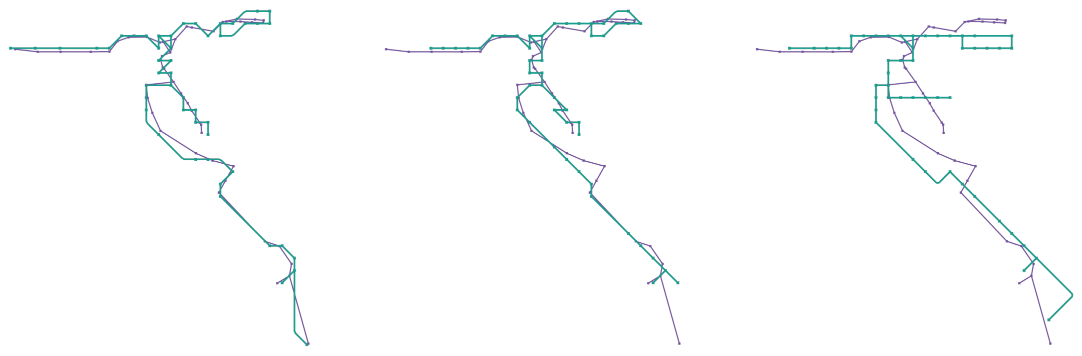

(a) “Original” and “Current” (b) “Result” and “Grid” (c) “Line Drawing”

Fig. 3.5: Different drawings that users can choose to view. The initial drawing and the current graph after moving and contracting some vertices are shown in brown and purple respectively in (a). The routed paths in the grid graph and a representation of the grid itself are visible in (b). Lastly, (c) shows stylized stations and the different lines of the resulting paths with contracted vertices reinserted.

sition of stations is changed. So unlike the input graph, this *current graph* will change with some user interactions. By default however, it is just the input graph after preprocessing as we do not contract vertices unless requested by the designer. “Show Grid” gives a schematic view of the underlying grid to indicate possible station positions. To avoid visual clutter, diagonal edges are not displayed. “Show Result” shows the shortest paths through the grid that edges were routed onto. An alternative visualization of this can be viewed by checking “Show Line Drawing”, which produces a more typical metro map style with stations as rounded rectangles and edges as bundles of line segments with colors corresponding to the lines. In this drawing contracted stations are also reinserted equidistantly on the corresponding paths, while they are simply omitted in the “Show Result” drawing. As its purpose is to just give a better idea of what the finished map might look like, the line drawing is created using a simple algorithm that does not minimize line crossings and even produces line intersections on singular edges.

Costs We let users directly change the parameters of the algorithm in (C1). The default values for all the costs and the *candidate radius* c_r (the factor which determines r when multiplied by D, see Fig. 3.2) are taken from the original paper. Figure 3.6 shows some examples of how these values can be modified to achieve different results. If users enter values that do not conform to the bend cost restrictions ($c_{45} \geq c_{90} \geq c_{135} \geq c_{180}$, $2c_s > c_{45}$, $2c_{135} \geq c_{90}$, and $c_{180} + c_{135} \geq c_{45}$), the offending values are highlighted, see Fig. 3.7. Bast et al. propose adding a constant offset to all bend costs to guarantee these constraints, but we decided to show the costs actually used in the algorithm so that the relationship among the different costs is more intuitive. If the inequalities are not met, a drawing can sometimes still be calculated, but it could include simulated bends, circumventing the higher costs, as explained in Section 3.1.1. If this happens, our



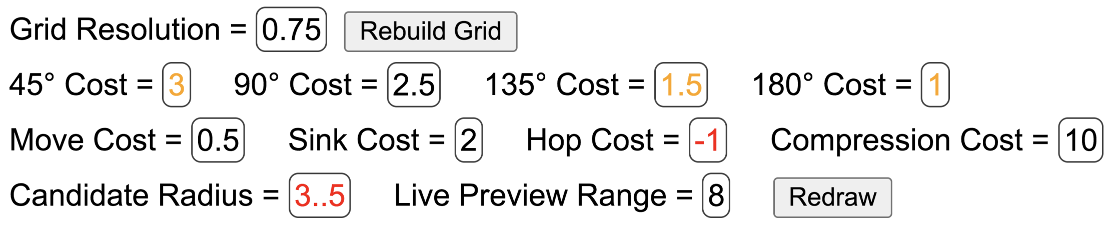
(a) Geographic accuracy (b) Shorter edges (c) Preferring right angles

Fig. 3.6: Different drawings resulting from cost adjustments. In (a), we set $c_m = 10$, so that the displacement penalty ("Move Cost") dominates over bend and hop costs. This results in a high geographical accuracy. If the goal is instead to get a more compact map, the hop cost can be increased: For (b), we used $c_h = 3$ and $c_r = 10$. By also increasing the candidate radius, we allow stations, like the left most vertex, to be placed further away from their original position without the need for a local search. Lastly, the bend costs can be adjusted for specific effects, like in (c). By setting $c_{45} = c_{135} = 4$, we deter those bends and instead mostly get right angles at the cost of edge length and geographic accuracy. Note that these costs do not adhere to the required inequalities, so some bends could have been simulated, this was however not the case here.

implementation will also fail to create the line drawing or calculate the global cost, as these algorithms expect every second vertex in the shortest paths to be in a new clique and for paths to only include sink vertices at the ends. This value can be calculated consistently after every interaction.

The *global cost* of the drawing, as shown for each edit in (C4), is a parameter to estimate the quality of a drawing. With interactions, it would not be consistent if we simply add up the costs for each path. This is because routing an edge initially will produce a different cost than when it gets added into an existing drawing. The latter may happen, when a user interaction only affects a few edges in which case only those may get recalculated. For example, consider Fig. 3.8 where an edge (s, t) is first routed based on just the initial drawing, with s and t not being settled on a grid position yet. Let $\deg(s) = 1$ and $\deg(t) = 2$ with another edge (t, u). Then the cost of the grid path of (s, t) consists of the displacement penalties of s and t and the hop and bend costs along the path. However, when (t, u) is already routed and the path of (s, t) needs to be recalculated because s got assigned a certain grid position, the path cost will consist of the same costs but also include a bend cost corresponding to the angle in t between the paths of (s, t) and (t, u). The global cost $\mathcal{C}$ instead consists of the hop and bend costs along the path of each edge and for each station, its displacement penalty and the bend costs between all pairs of adjacent paths.

While the cost can be useful to compare the quality of different drawings with the same


Grid Resolution = 0.75 Rebuild Grid

45° Cost = 3 90° Cost = 2.5 135° Cost = 1.5 180° Cost = 1

Move Cost = 0.5 Sink Cost = 2 Hop Cost = -1 Compression Cost = 10

Candidate Radius = 3..5 Live Preview Range = 8 Redraw

Fig. 3.7: Highlighting for bad parameter inputs. Numbers in orange indicate that these costs violate some cost inequality. Here $c_{180} + c_{135} \geq c_{45}$ is not fulfilled, thus the three involved costs are colored. Empty textboxes, negative values (here the "Hop Cost"), or non numerical inputs ("Candidate Radius") are ignored and the most recent valid value is used. The user is informed of this by the offending input being colored red.

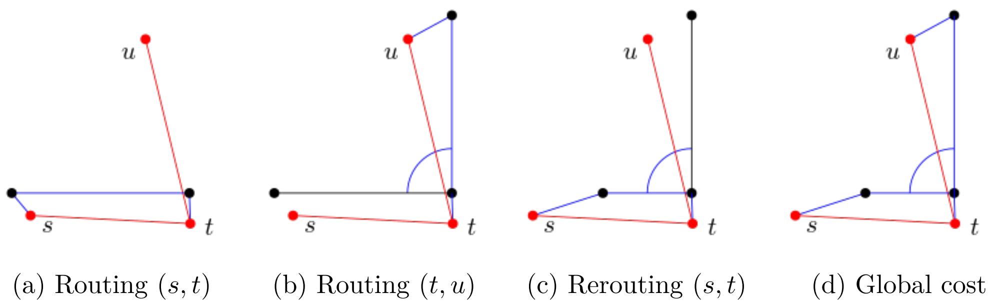


Fig. 3.8: Difference between global cost and sum of edge costs. Blue components represent the hop, movement and bend costs in each step. When finding a drawing for the red graph, we first route (s, t) in (a) and then (t, u) in (b). The sum over the costs of the paths includes the bend at t only once. If (s, t) is recalculated afterwards, as in (c), the bend at t has to be considered again, so the cost sum over all edges will count it twice. Instead we calculate the global cost of a drawing in an extra step, as in (d), considering each bend exactly once.

set of constraints, changes in the requirements, like requesting an edge to be straight, will usually also influence the global cost. These changes should however not necessarily be directly interpreted as qualitative differences. After all, one of the main reasons to pursue an interactive approach is to allow a designer to improve parts of the drawing, where a good solution according to the cost function, is unsatisfactory. We still denote the global cost and how much it changed for every interaction in the edit history ($C4$). Following the argument above, this is mainly useful for comparing the drawing before and after the local search as no restrictions are modified here.

Grid Similarly to the cost parameters, users can also change the size of cells in the grid to give stations and edges more options for where they can be drawn at the cost of increased runtimes or vice versa. Other than the costs which are read during routing, changing the grid requires the grid graph to be instantiated anew. This can be done by clicking "Rebuild Grid", at which point we first recalculate the dimensions of the grid,

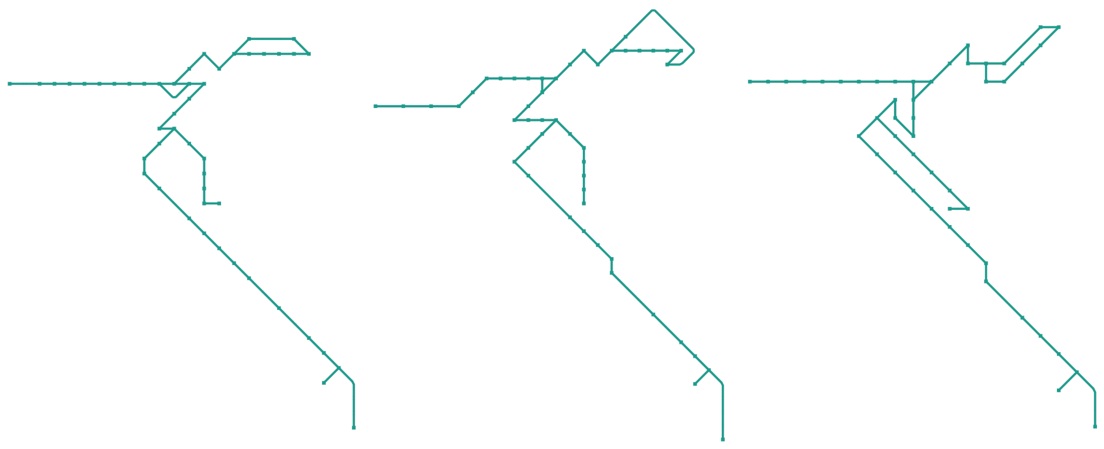

(a) $D = 0.7\bar{d}$ (b) $D = 0.65\bar{d}$ (c) $D = 0.85\bar{d}$

Fig. 3.9: Influence of the grid resolution on map quality and cost. Drawing (a) serves as baseline for comparison. Using a finer grid as in (b) can produce an arguably worse drawing. On the other hand, a drawing on a more coarse grid, as in (c) may also be considered worse than (a), but has a slightly lower global cost (2.3 % decrease).

by padding the bounding box of the initial drawing by 10 % of the corresponding side length on all four sides. The cell size is based on the average distance of all edges in the initial drawing $\bar{d}$. This, multiplied by the value for “Grid Resolution” in (C1) is the side length D of the cells. The default for this resolution factor is set to the highest multiple of 0.05 for which the algorithm finds a solution, but at most to 0.75, the value Bast et al. used.

When changing the grid resolution, some changes may not sensibly carry over to the new resolution. For example, if a station is set to a certain position on the grid, then increasing the granularity of the grid will move that grid position to the top left. Because of this, it makes sense to find a suitable resolution at the start and keep that during editing. Furthermore, even slight changes in D can significantly change the resulting drawing in dense parts of the graph. That the algorithm finds a solution on a certain grid, does also not necessarily mean that it is guaranteed to work on a finer grid. For example, the network of Vienna can be drawn on a grid with $D = 0.75\bar{d}$, but not on one with $D = 0.7\bar{d}$. Figure 3.9 shows some examples of the influence of the grid resolution. Varying it also changes the total cost somewhat independently of drawing quality, as a path will cross through fewer grid points and thus induce the hop cost and c_{180} less often on a more coarse grid. This makes it hard to objectively compare drawings based on different grid resolutions.

3.2 User Interactions

We allow users to influence the drawing in a number of intuitive and useful ways. For most of these interactions, they select an entry from the toolbox and use it on one of the graph representations in the canvas. Most tools add some restriction to a user selected set of vertices or edges, with a second tool removing the restrictions again. As an overview, our tools enable users to change routing in these ways:

- Move Input changes the position of vertices in the current graph, which determines candidate sets and movement penalties.
- Reposition Station sets which grid position a certain station should settle on.
- Straighten Edge prevents any bends in the path of an edge.
- Freeze Edges locks the paths of a group of edges, resulting in them always being routed the same way.
- Don't Contract specifies which degree 2 vertices are to be contracted.
- Local Search manually initiates the local search to polish the current drawing.

Selecting such a tool from the toolbox automatically makes an appropriate choice for which representations are shown, like hiding obtrusive drawings unnecessary for the task or overlaying the grid when dealing with grid positions. In the following, we give motivations for the different tools, explain how to use them and how we implemented them.

3.2.1 Selecting Elements

Before taking actions, most tools require the user to first select which objects should be affected. The drawing on which the selection is made varies, with some tools using the straight line drawing of the current graph and some the octilinear drawing on the grid. In both cases, we only allow connected components to be selected. The mechanism for making a selection is also similar for both. Although the same functionality could be implemented using buttons in the graphical user interface, we let user specify which selection mode to use by holding the corresponding key down, which allows for faster editing. We explain the different modes using Move Input as an example, which needs a selection of vertices on the current graph:

- No key – "Select object": The vertex closest to the mouse pointer is highlighted if it is close enough ($<$ 25 pixels). We call such a vertex v in the following. By simply clicking, just v becomes the selection – for the displacement tools, this vertex can also be dragged immediately.
- Q – "Select component": The closest vertex v and all vertices in the smallest connected component of the graph, if v were removed are highlighted. Clicking with Q pressed confirms this as the selection.
- W – "Select path": At first only v is highlighted, but the first click sets v as the start of a breadth-first search (BFS) for the vertex v' that the mouse is now closest

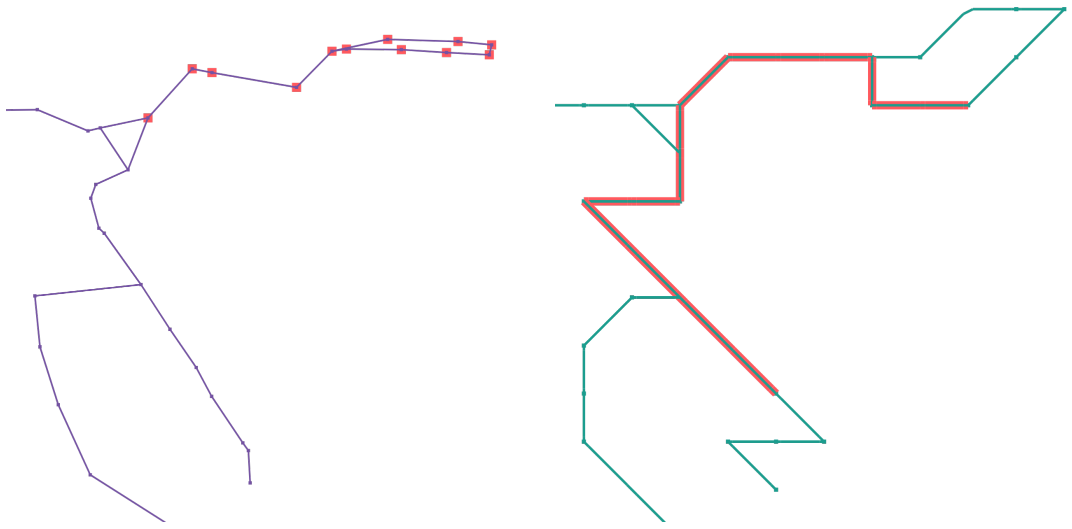

(a) Q – “Select component” (b) W – “Select path”

Fig. 3.10: The two main modes of making an initial selection. In (a), the vertices in a component of the current graph are selected. In (b), a path of edges on the grid is selected.

to. At this point, releasing W and moving the mouse, thus changing v', will cause the vertices along the path between v and v' to be highlighted. A second click will set that path (including the endpoints) as the selection.

- E – “Edit selection”: Refines an initial selection made using the previous modes. If v is part of the selection, clicking will exclude it, if doing so still leaves the selected vertices and their edges as a connected component. If v is instead only neighboring a selected vertex, a click will add it to the selection.
- R – “Reset selection”: Removes all vertices from the selection, to start over. This only needs to be pressed, not held as the other keys.

Note that for any key-presses to register, the canvas needs focus, so a user may need to click anywhere on it first. When the user is satisfied with the selection, they can initiate an action by dragging the objects or by hitting Enter, depending on the tool. The selection modes work the same way when selecting grid vertices. In this case, it only makes sense to choose from the sink vertices where stations settled, so only those are highlighted. The “Select component” and “Select path”-mode also execute their calculations on the corresponding vertices of the current graph.

If edges are being selected instead of vertices, the different selection modes behave similarly, this time highlighting edges, or their paths through the grid. Here, “Select component” highlights the edges in the biggest connected component of the graph if the closest edge to the mouse were removed. For “Select path”, a BFS between some endpoint of each of the two relevant edges is executed. We then simply highlights these two edges as well as any on the shortest path between the endpoints.

3.2.2 Moving Stations

An obvious way in which one may want to change the drawing is influencing where on the grid certain stations are placed. For example, to emphasize their geographic location or relative position to other stations, to make room for other elements, or to allow for better edge routing. A possible approach to allow this is simply moving the positions of vertices in the initial drawing. Even though this basically qualifies as preprocessing, by integrating this as a tool (MOVE INPUT), users can get quick feedback on how moving vertices influences the drawing. While this approach has the advantage of making use of the edge costs to find a trade-off between requested position and edge length and monotony, this compromise can also make the tool frustrating to use as stations are only placed somewhat close to their position in the initial drawing. So if users have a specific grid position for a station in mind, they might have to unintuitively distort the input to achieve this positioning. For more precise displacements, we provide the tool REPOSITION STATION, with which stations can be assigned a grid vertex on which they will be forced to settle. An example for what these tools can be used for is shown in Fig. 3.11.

For the first option, we let users select a set of vertices of the current graph and allow them to drag them to translate their position. The points are moved in real-time according to how much the mouse did since the mouse button was pressed. The new coordinates will be used to calculate displacement costs during routing, however, we still use the initial drawing of the input to determine topology. This means, the drawing of the current graph can contain intersections which will not be represented in the metro map drawing. Similarly, changes in circular edge orders through the MOVE INPUT-tool will be ignored. By keeping the input graph unmodified, users can also compare their edited version to the original network, to check whether its structure is still resembled by the current graph. If a high geographic accuracy of stations is desired, it could be beneficial to use the distance to the stations' positions in the initial drawing when calculating displacement penalties for the global cost. That way, movement of vertices caused by the users will be reflected negatively in the score. To increase the usefulness of this tool, we provide a live preview of the routing as the input nodes are being dragged. For this, we only recalculate the paths of edges adjacent to the selected vertices during dragging and perform a global recalculation when the mouse button is released. Since the grid graph is not updated for this, moving inputs too far outside the bounding box of the initial drawing can result in bad or failed routings.

The REPOSITION STATION-tool instead takes a selection of grid-points on which stations have been settled and lets users drag them to unoccupied grid vertices. The change of grid-position is calculated separately for x- and y-direction, by rounding the mouse movement in that direction since the drag started to the nearest multiples of the cell size D. This provides the coordinates of the targeted grid vertex. If they lie outside of the grid for any vertex in the selection, the displacement for all selected vertices is reduced until they are valid grid positions, so that they maintain their relative position. The targeted grid vertices are highlighted during dragging, with a live preview again rerouting the paths of edges adjacent to the selected vertices. This preview calculation

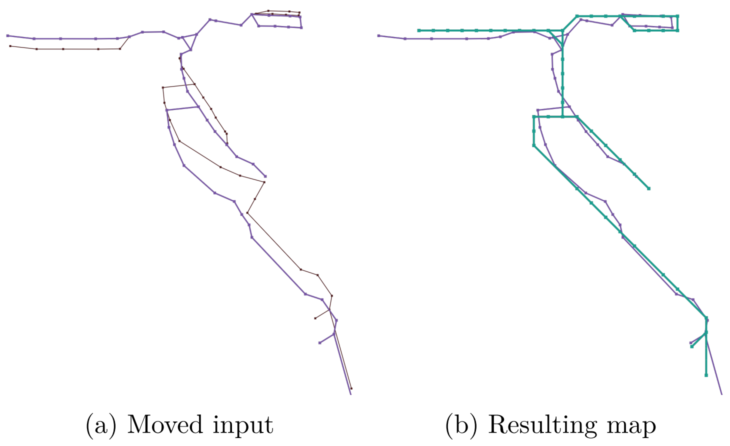

(a) Moved input (b) Resulting map (c) Repositioned stations

Fig. 3.11: Examples for using the displacement tools. In (a), we see how the purple current graph was changed from the brown original one using MOVE INPUT. By aligning some vertices, we can reduce the number of bends at the cost of geographical accuracy, as shown in (b). If the designer has a specific layout in mind, it can be achieved using REPOSITION STATION. We imitated that of the official map of Würzburg's network (see Fig. 1.1) in (c).

is skipped, if any targeted position is already occupied, because a station (excluding ones in the selection) is settled there or because a path (excluding those of edges adjacent to vertices in the selection) runs through it.

Similarly to MOVE INPUT, when the mouse button is released, the whole drawing is recalculated, after saving the requested grid-coordinates for each selected vertex. Now, whenever routing an edge, we not only check if the endpoints are already settled, but also whether this tool has previously been used to move them. If the latter is the case, the only grid-vertex in the corresponding candidate set is the one at the requested grid-coordinates. To prevent occupying the grid-position of a repositioned station v, we exclude that grid-vertex from the candidate sets of any edge not adjacent to v.

The requested position for a certain vertex can be updated by simply using the REPOSITION STATION-tool again, or it can be reset to being freely placed algorithmically by clicking on it while using the FREE STATION-tool. When selecting that tool, vertices with a set grid-position are highlighted in the metro map drawing. Clicking on one will also perform a recalculation of the whole drawing with that vertex now free.

For the live preview of both methods, we try to skip local recalculations if the time required for them is too great. Because this is performed whenever the mouse moves, longer wait times can significantly reduce the responsiveness of these tools. We use the distance of the moved stations to their neighbors as an indicator for how long routing will take. So for every edge that would get rerouted, we calculate the distance between where the station is dragged and the grid position of the other endpoint. If the sum of these distances is greater than some multiple r_p of D, we just show where the dragged stations will be placed without rerouting the adjacent edges. The preview radius r_p can be adjusted by users to fit their systems performance by entering a different value in "Live Preview Radius".

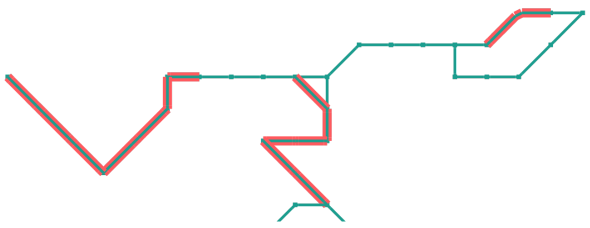
3.2.3 Straightening Edges

A certain edge or sequence of edges may include bends to reduce station displacement. If a designer deems it more appealing to sacrifice geographic accuracy for fewer bends, they should be able to request that edge sequence to be straight. This could also be used to guarantee no bends on edges which represent a part of the network with significant meaning, for example rails passing over a well-known bridge or, if applicable, the path between a certain start and destination station in a route-based map, as in [WC11]. To accomplish this for sequences of edges separated by degree 2 vertices, we again propose two mechanisms. Both first contract all vertices along the path, so that in the current graph only a singular edge needs to be straight. That way we do not have to deal with preventing bends at stations. Routing would also be likely to fail for long edge sequences otherwise, as the path of the first edge would set the trajectory for the whole sequence but only consider costs of its own path. After contracting, for one approach we simply set the cost for any bend except for c_{180} to ∞ before routing that edge. For the other, we calculate all combinations of endpoint candidates that lie on a straight path and use the cheapest bend-free path between those.

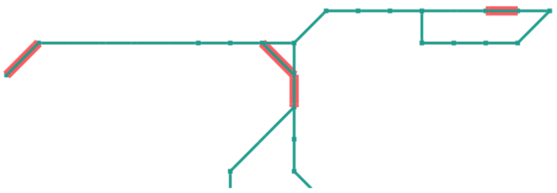
The first method – Straighten Edge – lets the user select a set of edges in the grid drawing. When they hit Enter, we check that all vertices along the edge path can be contracted. If this is impossible, we notify the user so that they can adjust the selection. Otherwise, the edge sequence is replaced by a single edge for which we denote that it should be straight and recalculate the drawing. Now, whenever such an edge is routed, we first set $c_{45} = c_{90} = c_{135} = \infty$, to stop bend edges form being used. The path might however still leave the straight line, by passing through sink vertices. To avoid this, we do not allow edges into sink vertices during routing of straight edges, unless the sink is in the target set, or one of the start vertices. This way, the resulting path will only consist of hop edges and 180°-bends. Besides changing bend costs, we also double the radius used to determine the candidate set for such edges, as their restrictive shape makes them harder to route successfully if only few grid vertices are possible endpoints. After routing, we reset the bend costs and candidate radius to what they were before, so subsequent edges are unaffected.

In some cases, straightening edges in this manner can also be used as an intuitive tool to improve the drawing, if the previous settling of endpoints of an edge results in it having to take an awkward path to avoid other paths. This is the case for the rightmost straightened edge in the example use case in Fig. 3.12a. However, this kind of problem could also be alleviated by simply moving the offending edge forward in the edge ordering, which is a byproduct of setting an edge as straight, see Section 3.3.

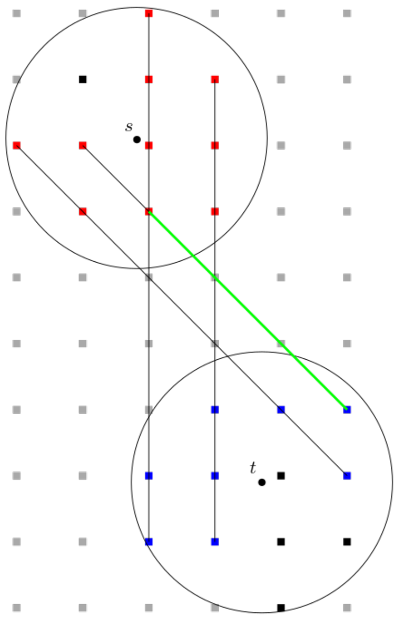
The other approach was only implemented for initial testing and because it performed worse than the first one, it was not incorporated with the rest of the algorithm. That means, Straighten Edge 2 only recalculates the drawing locally around the selected edge and other edits discard these changes again. With this tool, users can click an edge on the grid graph that they want to be drawn without bends. Using contractions, this could again be expanded to work with a sequence of edges separated by degree 2 vertices. The edge (s, t) corresponding to the selected curve and all other edges adjacent



(a) Initial routing



(b) After straightening



(c) Potential straight paths

Fig. 3.12: The figures on the left shows how Straighten Edge may be used to remove bends. The highlighted edges and paths in (a) are drawn as one straight edge each in (b). The edge in the 3-cycle was straightened as it was bend when the path underneath it was straightened. For the alternative approach, the search for a straight path is shown in (c). The colored vertices are the grid points from which a straight line path to a vertex in the other candidate set is possible. The shortest path – including displacement penalties – among these is drawn in green.

to s or t are removed from the drawing. The vertices s and t are also unsettled and we calculate their candidate sets S and T normally. Now we find all paths in the grid that connect some vertex in S to one in T with a straight line. An example for this is shown in Fig. 3.12c. For every such line, we then calculate the costs of all feasible paths along it. The cheapest path following any line is then used for routing the edge and determines where s and t are settled. Subsequently, the paths for the other previously removed edges are added again using the normal routing method.

3.2.4 Contracted Stations

Besides using contractions to straighten edge paths, we also see them as an opportunity for users to directly influence the drawing. They may want to keep some important stations while contracting others or want certain paths to follow the stations along it more closely. For this, the Don't Contract-tool allows them to individually choose which degree 2 vertices to contract. When selecting this tool, the vertices in the initial drawing are colored according to their contraction status, as shown in Fig. 3.13. Brown ones cannot be contracted, because they have the wrong amount of edges or a line starts

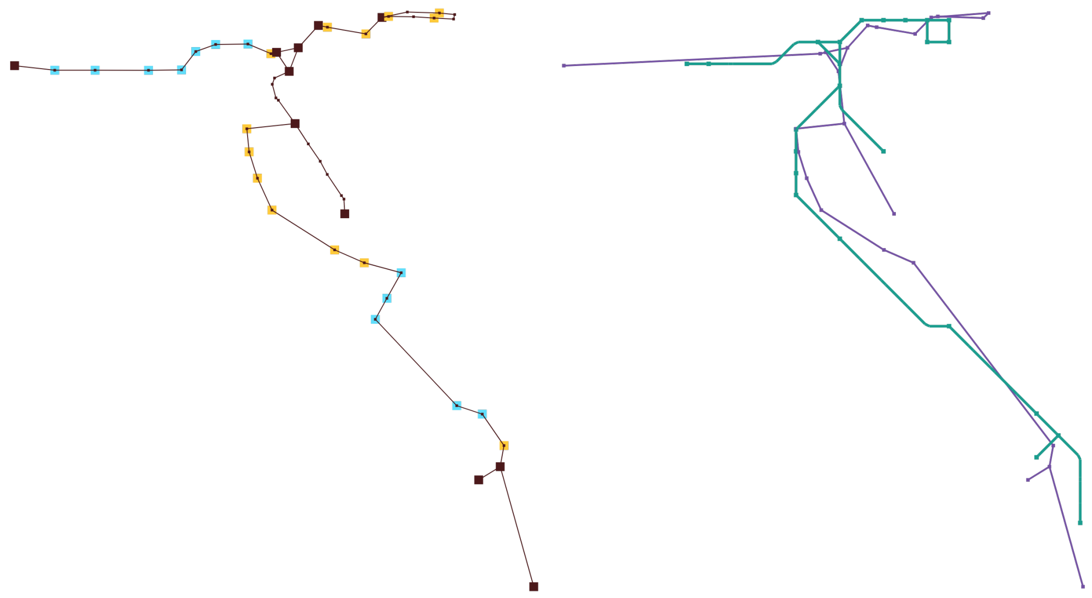

(a) Contraction highlighting (b) Resulting map

Fig. 3.13: The coloring when contracting stations is shown in (a). Here, some stations are contracted manually (no highlighting) or by straightened paths (blue), while others stay expanded (yellow and brown). The resulting current graph and metro map are shown in (b).

or terminates at that stations, whereas blue vertices will definitely be contracted, as they lie on a path selected as a straight edge. The remaining vertices are highlighted in yellow if they are exempt from contractions and blank otherwise. Clicking on either type toggles the vertex state from one option to the other. When the user hits Enter or switches to another tool, the current graph is rebuilt based on the new contraction rules and the drawing is recalculated accordingly.

3.2.5 Local Search and Frozen Edges

Besides the local search as described in Sec. 3.1.1, we also allow for an alternative approach Local Search All which instead of moving just one vertex in every step, repositions all vertices to the cheapest position within their grid-neighborhood. As with the original search, this is repeated until moving any vertex only increases the global cost.

Unlike Bast et al., we treat the local search of the algorithm as an optional polishing step, since the runtime is high for the minor reduction in global cost. Instead, we allow users to initiate this step by hitting the Local Search- or Local Search All-button. This will start the local search on the current version of the graph in a separate thread, so that the user can still interact with the interface during execution. When a result is obtained, it will show up in the edit history in the row with the graph version on which it was called. Since restrictions, that were made after the local search was started, will not be respected in its result and rerouting with the restriction discards optimizations of the local search, we try to keep as much of the drawing result from the search and only recalculate areas directly affected by the additional edits. If the user wants to apply the added restrictions to the changes from the local search, they can click “Propagate” on the entry in the version list.

In order to preserve most of the drawing when using this function, we introduce the concept of *Frozen Edges.* These edges will produce paths in the grid which, while translatable, keep their shape and orientation. When an edge (s, t) is frozen, we store at which port its path in the current drawing leaves the grid vertex where s settled and the sequence of bends (including 180°) taken. Now, when either endpoint of that edge settles onto a sink, the bend sequence is recreated – in reverse order if that endpoint is t. This means that first endpoint determines the position of the edge and thus, on which grid vertex the other endpoint settles. We also extend this idea to groups of paths that should additionally maintain their relative placement in the grid, by saving the offset in grid position of the start vertices to each other. Here, after settling any vertex that’s adjacent to an edge in this group, the paths of all edges in that group are given and can be added to the drawing according to the offsets and bend sequences. This can be used to freeze parts of the drawing the user is content with, so they stay the same on subsequent routings. By allowing the groups to still be translated (unless one of the adjacent vertices has a set position), they can more easily tolerate other constraints like adjacent straight edges.

For keeping most optimizations obtained by the local search, we freeze the whole graph and then remove all edges that were straightened and those removed or added as a result of contracted or expanded stations. The edges of each still connected part of the graph make up a frozen group. If any of them are adjacent to multiple positioned stations, we remove the edges adjacent to the one with the lowest degree from the frozen group, until at most one station has an assigned grid position. This way, most edges stay as the local search routed them, while still allowing the drawing to respect any newly added constraints.

If the user wants to make additional changes to the frozen parts, they first have to UNFREEZE them. This tool colors in all frozen paths and highlights the corresponding group when the mouse is close to one of them. By clicking, the selected group is unfrozen and the drawing recalculated with edges in that group being routed traditionally again. The inverse is possible with the FREEZE EDGE-tool. With this, users can select a set of paths in the drawing that they’d like to keep as they are and hit Enter to freeze them. Here, users may not select edges that are already part of a frozen group. Similarly, the selection for straightening edges also excludes frozen ones, while moving the grid position of a station, which is the endpoint of a frozen edge, automatically moves all vertices adjacent to the edges in that frozen group. This is to prevent inconsistencies between the different systems, that would result in routing requirements that are impossible to fulfill.

3.3 Edge Order

As the constraints added by users limit the amount of paths possible, the edge order should be designed to take edges with routing constraints into consideration. For example, it makes sense to route all frozen edges in a group as soon as one of their adjacent vertices is settled, so that no other edges can occupy the grid edges used for their paths. Straight edges should also be drawn before ones with arbitrary shape, as adding the latter kind into an already crowded drawing is easier. Similarly, preferring edges adjacent to positioned stations can also result in a better drawing. To account for these factors, we assign each edge a priority score depending on whether they are affected by restrictions. We first calculate the edge order as Bast et al. did. We use the ranks of edges in this order as a tiebreaker, if multiple share the same priority. So initially, the priority of each of the m edges is set to $m - r(e)$, where $r(e)$ is the position of edge e in the original edge order. If an edge is set as straight, its priority is increased by $2m$ and for each endpoint that has a set grid position, it is additionally increased by m. We choose this weighting of the restrictions because forcing edges to be straight seems like a bigger limitations than settling an endpoint. After all, in routing without any restrictions, all edges but the first have also at most one free endpoint. We did however also test different weight combinations in Section 4.1.

After sorting the edges by their priority, all straight ones and those where both endpoints are positioned will come first, followed by ones with one set endpoint and lastly ones unaffected by user restrictions. In order to deal with frozen edges, we first remove all of them from the order. Then, for each group, we find the first edge in the order that shares an endpoint with any of the edges in the frozen group. From that vertex, we execute a BFS, denoting the order in which the edges in that frozen group are passed. This sequence is then inserted into the actual edge order after the edge with the shared endpoint. So the priority of a frozen edge e is $p - r'(e)/(m+1)$, where p is the priority of the first edge that determined the position of the frozen group including e and $r'(e)$ is the position of e in that group's BFS. This way, frozen groups are inserted into the drawing as soon as their position is decided and their edges are added in an order that guarantees that each one has at least one endpoint already settled.

With this edge order in place we can potentially combine any set of user restrictions into one drawing. For this, we first create the current graph including to user specified contractions and those induced by requested straight paths. The positions of vertices in its initial drawing are given by the input graph unless they were previously displaced using the Move Input-tool. We then calculate the edge order as detailed above and add them to the drawing successively. In order to route an edge e we first check if it is frozen, in which case we follow the stored bend-sequence. Otherwise, we determine the candidate sets for the endpoints. Should a vertex v already be settled on a grid vertex ψ, this set only contains ψ. Similarly, if the position v was set using Reposition Station, the previously selected grid vertex is the only candidate. Otherwise, the set is made up of all unoccupied sink vertices within a radius around v. If the user requested e to be straight, we also set the bend costs (excluding c_{180}) to ∞. Now we find the shortest path in the grid that connects the two candidate sets and add it to the drawing. If required, the bend costs are reset before proceeding. This is repeated until all edges are routed or one fails because no path can be found for it. In the latter case, we display a warning to the user and let them remove the most recent restriction. After a drawing was created successfully, we calculate the global cost, update the canvas and add an entry to the edit history.

3.4 Software Engineering

Since the development of our prototype was a major part of this thesis, we give an overview of some interesting and challenging parts of the implementation in this section. For each of the three graphs (original, current and grid), we store a set of vertices and one for edges. Each vertex has a list containing the edges it is adjacent to and a point object for representation in the drawing. Edges in turn have a reference to both of their endpoints and to a line segment with the corresponding vertices' points as the start and end. The relationship between graphs is given by the elements referencing each other. So for example, an edge in the current graph has a list of the original edges it replaces and a grid vertex stores the station that settled there, if any.

Routing on the Grid Before and after routing an edge, the costs in the grid have to be adjusted as described in Sec. 3.1, to ensure that grid edges are only used once. There are multiple factors that can contribute to an edge being seen as occupied. We also need to be able to undo changes to this status, for example when removing paths for a live preview, the occupation status of all affected edges needs to be reverted to what it was previously. To do so, every grid edge g has a dictionary, mapping the current edges e that occupy g to a value, depending on the context:

- 1: Edge g is a bend or sink edge belonging to a grid position that the path of e passes through (or ends in).
- 2: Edge g is actually used in the path of e, or is a hop edge crossing such an edge.
- 3: This is added before routing the current edge (s, t) if necessary and removed afterwards again. Here, g is a sink edge that is blocked to preserve the circular edge order of s (or t). If s is already settled and other edges adjacent to s are already routed, we check which sink edges can still be used without changing the embedding. We also make sure that there is enough room to leave the grid vertex, for edges adjacent to s, that will be routed later.

We keep track of the maximum in this dictionary. If it is 2 or greater while routing (s, t), we set the cost for g to ∞. For a maximum of 1, we do not block g, if it is a sink edge adjacent to where an endpoint of e settled and if that endpoint is s or t. In this case, or for an empty dictionary, we use the normal cost of that edge.

These costs also change, depending on which current edge (s, t) is being routed, so every grid edge g also has a list of costs, the sum of which is used during pathfinding. Initially, all lists only contain the cost corresponding to the type of edge, that is the hop, sink, or one of the bend costs. Before finding a shortest path between the grid vertex sets S and T, the lists of sink edges adjacent to the candidates $\psi \in S \cup T$ are modified and reverted afterwards again. The cost of each such sink edge g is increased by the movement penalty based on the distance between ψ and s. For every other path already ending in ψ, we also add the bend cost corresponding to the angle between that path and the port through which g leaves ψ.

Handling Interactions To provide different functionalities for the various tools, each tool consists of a selection mode and a dictionary, mapping event types onto functions. When the user acts on the canvas in any way, e.g. moves the mouse, scrolls, or presses a key, the corresponding event is first passed to the selection mode of the current tool. Here it is either consumed if the action was still part of the selection process, or the event and a potential selection is passed on to the tools function corresponding to that event type. In this function, any logic or graphical changes specific to the tool are made. We implemented one selection mode for vertices and one for edges. The tools also specify on which graph the selection should be made. This allows us to reuse the same code for our current and potential future tools. Only slight changes have to be made to accommodate, whether a selection is made on the current or original graph versus the grid graph, where not all, but only the occupied elements are relevant.

As an example, Reposition Station needs a group of vertices of the grid graph as the selection and it has to react when the mouse is being dragged and where the user starts or stops dragging. Any clicks made while pressing “Q”, “W”, or “E” are captured by the selection mode, as the user is not changing the positions yet. When the selection mode recognizes that the vertices are now being dragged, it passes the mouse events and a list of the highlighted stations to the functions of the tool. Only in here are the requested positions and the new routes using them actually calculated.

With this system, we also simulate events when a tool is selected or deselected. This way, tools like Don’t Contract can add highlighting right away without any interactions, and subsequently remove it when another tool is selected. Furthermore, we can set a default behavior for certain events that applies to all tools, unless explicitly overwritten. We use this to allow translating the viewport by scrolling for all tools but “Navigate” and to simulate a release of the mouse button when the pointer leaves the canvas. This prevents inconsistencies with the internal state tracking whether the user is clicking. Similarly, if the pointer enters the canvas while the button is pressed, a mouse down event is called.

Line Drawing Whenever the metro map is successfully rebuilt, we also create a new line drawing for it. Such a drawing consists of two sets of geometric objects, one containing the rectangles for stations and one the line bundles for paths between them. Similarly to how vertices are always the same size, we want the stroke thickness of lines, the gaps between them, and thus the dimensions of the rectangles to be consistent regardless of zoom level. For this, the objects are drawn using offsets in pixel space around the converted location of grid vertices. If vertices were contracted, they are still drawn in the line drawing, dividing the corresponding line bundle into parts of equal length.

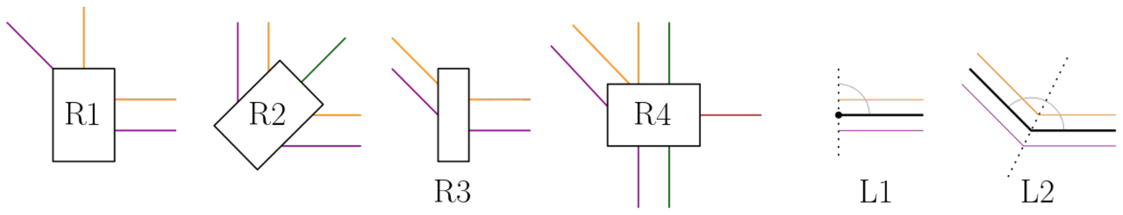


Fig. 3.14: Creation of the line drawing. The first four diagrams show the alignment and dimensions of the rectangle based on incoming lines. Rectangle R1 is aligned with the only edge with highest line count, R2 splits a tie between edges at an right angle, R3 is axis alignment because of a tie at a 135°-bend, and R4 aligns with the preferable vertical edges, since there are no horizontal ones in the 3-way tie. Additionally, R3 shows a change in the length of the shorter sides since there are no lines leaving at those sides. The two diagrams on the right show the dotted lines on which the endpoints of lines lie. This is given by the black path on the grid, both, when ending in a station (L1), and at bends along the path (L2).

When drawing rectangular stations, we first align them with an edge, i.e. set the longer side perpendicular to one of the edges leaving the station. For this, we use the port through which the highest number of lines leaves. If two directions tie, we try to align the longer side of the rectangle to halve the angle between them. If this angle is 45° or 135°, we stay axis aligned, as we want the rectangles to match the octilinear style. If more directions tie for first, we prefer alignment with horizontal edges over vertical ones if available, and use diagonals as a last choice. Examples for these different cases are visualized in Fig. 3.14. The number of lines on the alignment edge determines the length of the longer side. The length of the short side is increased, according to the maximum number of lines on edges orthogonal to the alignment edge.

To draw the line bundles, we need to calculate the coordinates of the endpoints of every line segment, that is, one point per bend for every metro line of the edge. When the paths terminate in a station, these points lie on a line through that vertex, perpendicular to the last hop edge. That way, the endpoints will be covered by the rectangle in most cases. For bends along the path, they are on the line halving the bend and running through the sink vertex at that location. Both of these cases are demonstrated in Fig. 3.14 on the right. Using these lines, we calculate the endpoints' offsets from the sink vertex at each bend. We use a set order in which the different metro lines occur. This order is followed to determine how far out from the sink vertices (and in which direction) each metro line is drawn. As this order simply rotates with the direction in which the path travels, many unnecessary line crossings are introduced, some even in between bends. We accept these local imperfections, as the line drawing can be calculated quickly and still gives a decent impression of the drawing in a typical metro map style. Optimizing such a line drawing for minimal crossings is a complex problem on its own and has been the focus of some dedicated research [BBS19].

Edit History and Concurrency The edit history was implemented before we had settled on a way of combining different interactions. At that point is was unclear whether new edits would just make local changes in the previous drawing, unlike the global recalculation at every step that we ended up using. Thus, we make a complete copy of all three graphs in their current state, which ensures that a previous drawing can be recreated exactly, without needing to execute any pathfinding. We also clone the line drawing and the state of the user interface, that is: which costs were entered, check boxes ticked, tool selected, and the position of the viewport.

In order to run the local search concurrently with the rest of the program, we execute it in a background thread, using a Web Worker[2]. Since this code runs in a separate context from the main script, we need to pass the current version of the graphs in serialized form to the worker. This would usually be done by converting all objects into the JSON-format. However, this does not allow for circular references and creates copies of an object if it is referenced in multiple places, instead of linking all references to the same instance. Our objects contain both of these types of references. For example, an edge stores its endpoints, which in turn both reference the edge.

To deal with this, we still convert the current version to JSON, but replace most occurrences of objects that may be referenced at multiple locations. In our implementation, these objects are either vertices, edges, or geometric objects, like line segments representing edges or rectangles for stations in the line drawing. Each such geometric object has a unique ID and the IDs for vertices (or for edges) are unique in the vertex-set (edge-set) of its graph. When serializing these sets or the set of all geometric objects, we store the actual values of these objects. Other references to the objects are replaced with a string, containing the objects ID, its type, i.e. edge, point, etc., and – if applicable – to which graph it belongs. Then, after deserializing the JSON-text in the worker, we walk through our object tree and replace all strings of that format with a reference to the corresponding object that is stored in one of the sets.

Unfortunately, both serialization and deserialization require a significant amount of time, in the order of hundreds of milliseconds for Würzburg's network. Even the creation of a deep copy, by traversing the object tree in the main script, is not much faster, simply because of the deep nesting of references and the large number of objects. Retrospectively, it would have made more sense to simply store which restrictions are active in any version and then recalculate the state of the grid, and thus the drawing from those. In fact, we successfully tried this approach to store the constraints used in our figures, to be able to recreate them consistently. These serializations could also be used to store the current version in the browser's cookies or to export and import different sets of restrictions.

[2]MDN Web Docs – Web Workers API: `https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API` (Nov. 2021)

4 Evaluation

We evaluate the usability of our approach by testing how frequently the algorithm fails and how long system response times (SRTs) are, both for recalculating the whole drawing and for the live preview during tool use. We also examine how stable the drawing is, that is, how strongly it changes globally when editing one part. Furthermore, we anecdotally show whether tools perform as expected, allowing users to achieve the desired effect, as well as occasions where they fail. Lastly, we present some examples of maps for different networks created with our method. Conducting a user study to evaluate how well our tools and algorithm aid in the creation of appealing and useful metro maps would be insightful, but was beyond the scope of this thesis. This could be used to inquire about the designers quality of experience while using the interface and to compare the resulting drawings to fully algorithmically generated ones.

It should be noted that we evaluate some data multiple times, using different tests, or to compare it to multiple other data sets. This changes the interpretation of the p-values and makes it more likely to incorrectly arrive at a result. To compensate, the p-values should be read as larger values than the ones listed when using multiple tests. This effect is however negligible in our case, since most of our results are overwhelmingly significant.

4.1 Edge Order

To assess, whether our method of generating the edge order is actually an improvement over the original order, we test both on multiple different sets of restrictions and see whether they find a solution, when adding edges according to their order. Here, we also use the global cost as an indicator of drawing quality, to see if one produces a better map when both succeed. Besides these two strategies we also try different weightings for the prioritization of restricted edges. We use the following combinations for increasing priorities of straightened edges (p_s) and of each positioned endpoint (p_p):

- $p_s = 0, p_p = 0$: The original edge order as used by Bast et al., here the restrictions do not influence when edges are routed. This serves as a baseline to compare the other combinations to.
- $p_s = 2, p_p = 1$: The weighting we used to ensure straight edges are among the first to get routed with set positions for endpoints mattering less, as those occur during routing without restrictions as well.
- $p_s = 1, p_p = 1$: Equal priority gain for both restrictions. Edges that have both endpoints positioned will be routed first here, followed by ones with one endpoint set and straight edges.

- $p_s = 1, p_p = 2$: Any endpoint being set makes edges appear early in this edge order. Straight edges are then added only before completely unrestricted edges.

We simulate user actions by randomly selecting some edges to be straight and specifying the positions for some stations, so that we can test the edge orders on different sets of constraints. We do not examine the influence of different displacements of input stations and contractions of stations, as for the purpose of comparing edge orders, these just act like different input graphs. Instead, we try all approaches on multiple networks, namely Würzburg, Vienna, and Sydney. Similarly, we only select singular edges to be straight and not paths including multiple edges. Again, all weightings deal with this in the same way, contracting the vertices along the path, resulting in a single edge that needs to be straight.

To generate restriction sets, we pick a random integer from $\{0, \dots, 7\}$ and then select that many edges to be drawn without bends. We generate another number in $\{0, \dots, 7\}$ to determine the amount of stations we reposition. For each of these, we set its position to a random grid-vertex in its candidate set, i.e. an unoccupied gridposition, that is less than $3D$ from the stations position in the initial drawing. While we do ensure that no two stations are positioned at the same grid position, we may still generate restrictions that are impossible to fulfill, such as requesting an edge to be straight while setting its endpoints to gridposition $(1, 1)$ and $(2, 3)$, or requesting grid positions that violate the input graph's topology. For comparing the different weightings, this does not present a problem as all strategies will fail on such restrictions and they are relevant for determining the success rate of the algorithm as well, since a user could also introduce constraints in a way that makes routing impossible. Nonetheless, only generating constraints that a user might set is not trivial, and our restrictions, might for example, include setting stations to arbitrary positions and straightening edges even though they were already drawn without bends. However, simulating enough inputs should still give an idea about the influence of the different priority values.

For each random set of constraints, we attempt routing with all four edge orders, one after the other, denoting the global cost of the drawing after a success or ∞ otherwise. To see, whether our weighting performs differently from the original order, we compared the costs using a Wilcoxon signed-rank test [HEK14]. This checks whether two paired sets of values $x_1, \dots, x_n$ and $y_1, \dots, y_n$ stem from the same distributions. For this, the absolute difference between the values in each pair is calculated. The differences are then sorted in ascending order, assigning each difference a rank $R_1, \dots, R_n$, with $R_i = 1$ for the pair with the smallest difference $|x_i - y_i|$. Note that any pairs with $x_i = y_i$ are discarded and not included in the ranking. We then sum up all the ranks of differences, where the value from the first set was greater than that of the second one. We do the same for the remaining ranks and take the lower of the two values as our test statistic[1]:

$$T = \min\left(\sum_{i=1}^{n} c_i R_i,\ \sum_{i=1}^{n}(1 - c_i) R_i\right) \quad \text{with } c_i = \begin{cases} 0, \text{ if } |x_i < y_i| \\ 1, \text{ if } |x_i > y_i| \end{cases} \tag{4.1}$$

[1]SciPy documentation – scipy.stats.wilcoxon `https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.wilcoxon.html` (Nov. 2021)

		Würzburg		Vienna		Sydney	
p_s	p_p	T	p	T	p	T	p
2	1	827	1.57×10^{-15}	331	7.25×10^{-14}	1616	1.78×10^{-3}
1	1	757	7.53×10^{-15}	328	2.70×10^{-13}	1596	2.16×10^{-3}
1	2	947	2.12×10^{-12}	335	2.15×10^{-12}	1563	5.18×10^{-3}

Tab. 4.1: Wilcoxon test comparing the drawings created using edge orders that gave straight edges (p_s) and those adjacent to positioned stations (p_p) different priorities. Three different weightings are tested against drawings from the original order ($p_s = p_p = 0$). The listed statistic T is the sum of either all positive- or all negative-rank differences, whichever is smaller and p is the test's p-value.

By approximating the distribution of this value under the assumption that the data sets share a distribution, we can calculate the likelihood of achieving the T we got. This is our p-value, the probability of wrongly declining the null hypothesis, that the sets follow the same distribution.

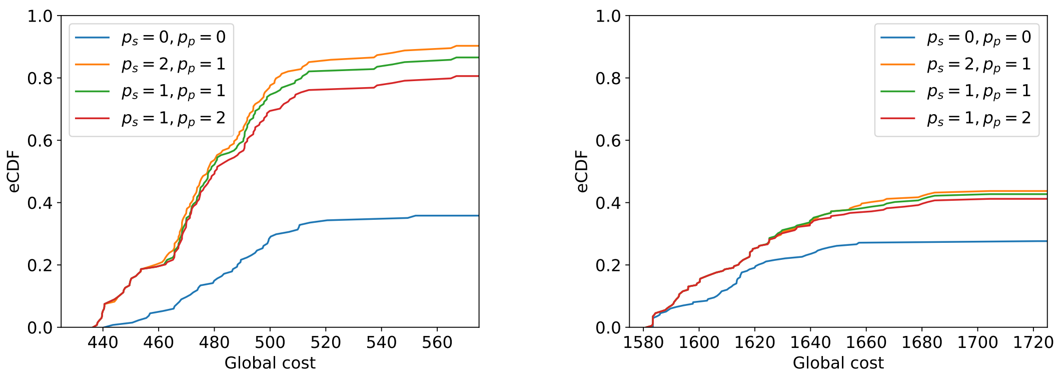
Note that as this test relies on ranking the samples, having some of them set to ∞ will not be interpreted as a bigger difference than any other value larger than all others. Value pairs were mainly dropped when both strategies did not find a drawing, or when there were few restrictions and thus both strategies obtaining the same costs. The results of comparing prioritizing methods to the original edge order are listed in Tab. 4.1. We evaluate 200 of the randomly generated constraints for a small (Würzburg: $n = 49$ stations, $m = 50$ edges), medium (Vienna: $n = 84, m = 90$), and large (Sydney: $n = 174, m = 183$) network and observe a clear difference in the results produced by every combination of priorities. Some of the weighted strategies also differ significantly from each other, this effect is however much less pronounced.

The success rates of each strategy and the average cost over runs where a drawing was found, as shown in Tab. 4.2, indicate the direction of the difference. With all new edge orders finding solutions for the two smaller networks more than twice as often as the original one and still significantly increasing the success rate on the large one. When the unmodified edge order does produce a drawing, it is on average not worse than those created by the other orders. There are some sets of restrictions, where it even achieved a lower global cost than any of the others, and even ones, where it was the only strategy that found a solution at all. But, due to the large discrepancy in success rate, a strategy utilizing priorities is clearly preferable.

This is emphasized by the empirical cumulative distribution functions (eCDF) of our data sets depicted in Fig. 4.1. As the unsuccessful routings are included as ∞ here, each strategy only reaches a y-value according to its success rate. In this plot we can see, that for any value x, the new strategies produced more drawings with global cost smaller than x, when compared to the original order. So inversely, on a random constraint set, we are more likely to achieve a lower cost using these orders. We can see that without pairing the results of the different priority orders by constraints, as they were for the Wilcoxon test, they perform rather similarly. However, the most intuitive weighting,

		Würzburg		Vienna		Sydney	
p_s	p_p	Success	Cost	Success	Cost	Success	Cost
0	0	36 %	485 ± 23	30 %	814 ± 23	28 %	1614 ± 25
2	1	90 %	478 ± 26	94 %	816 ± 23	44 %	1619 ± 29
1	1	87 %	477 ± 26	91 %	816 ± 24	43 %	1618 ± 29
1	2	81 %	477 ± 27	68 %	816 ± 24	42 %	1617 ± 29

Tab. 4.2: Success rate using different weights for edge order priorities, as well as the mean and standard deviation of the global cost for completed drawings.



(a) Würzburg (b) Sydney

Fig. 4.1: Distribution of global costs of drawings on random restrictions as an eCDF, visualizing the influence of the method by which edges were ordered.

namely $p_s = 2, p_p = 1$, performs best on all tested networks and has the highest overall success rate. Therefore, we use this strategy in our implementation.

Our data also indicates, that a high number of restrictions corresponds with solutions being harder to find and them having higher costs. To verify this, we again randomly select graph elements, which we afflict with restrictions to simulate interactions. This time however, instead of randomly setting the amount n_s of straightened edges and number n_p of positioned stations, we tried 100 restriction sets for each combination with $0 \leq n_s, n_p \leq 10$. Table 4.3 shows the resulting success rates. As expected, more of either restriction increases the number of fails, as they limit options during routing.

When it comes to the average global cost across 100 successful runs, as listed in Tab. 4.4, straightened edges produce unexpected results. Here, requiring more edges to be straight, tends to result in a lower cost. While straightening an edge does not inherently increase its own cost, the placement of the endpoints can cause adjacent edges to be longer and require more bends. We thus attribute this negative correlation to sampling bias. If the described displacement is significant, a drawing with the other restrictions will be hard to find, excluding the resulting high cost from our statistic. If

n_p n_s	0	1	2	3	4	5	6	7	8	9	10
0	100 %	89 %	85 %	66 %	57 %	59 %	47 %	45 %	30 %	20 %	21 %
1	100 %	91 %	80 %	71 %	60 %	54 %	42 %	41 %	36 %	24 %	17 %
2	97 %	88 %	83 %	67 %	60 %	44 %	36 %	36 %	24 %	21 %	19 %
3	96 %	86 %	78 %	64 %	60 %	45 %	37 %	33 %	23 %	23 %	24 %
4	94 %	74 %	68 %	58 %	50 %	43 %	37 %	30 %	28 %	21 %	12 %
5	85 %	76 %	65 %	49 %	41 %	53 %	37 %	23 %	21 %	14 %	19 %
6	73 %	74 %	60 %	55 %	41 %	31 %	32 %	28 %	14 %	15 %	11 %
7	70 %	66 %	54 %	45 %	38 %	27 %	22 %	22 %	11 %	11 %	8 %
8	72 %	57 %	53 %	33 %	33 %	23 %	21 %	29 %	13 %	10 %	6 %
9	57 %	49 %	47 %	36 %	20 %	30 %	13 %	11 %	12 %	9 %	9 %
10	54 %	47 %	39 %	28 %	27 %	24 %	16 %	13 %	13 %	10 %	5 %

Tab. 4.3: Influence of the number of positioned stations n_p and the number of straight edges n_s on the success rate of our heuristic on Würzburg's network. Stronger saturations correspond to fewer failed drawings. The headers are colored according to the mean success rate in that column or row.

the straightened edge instead does not force its neighboring edges to awkward paths, the cost will stay low and the algorithm is more likely to succeed. In these cases, the change in edge order can even be useful to align an edge with its geographic position, preventing displacement penalties. Some edges for which this happens are shown in Fig. 4.2.

Repositioning more stations on the other hands increases the average cost of successful drawings. This effect makes sense, as repositioning stations will usually result in a larger displacement penalty, as the station does not settle on the optimal grid vertex. This can additionally lead to adjacent edges requiring longer paths.

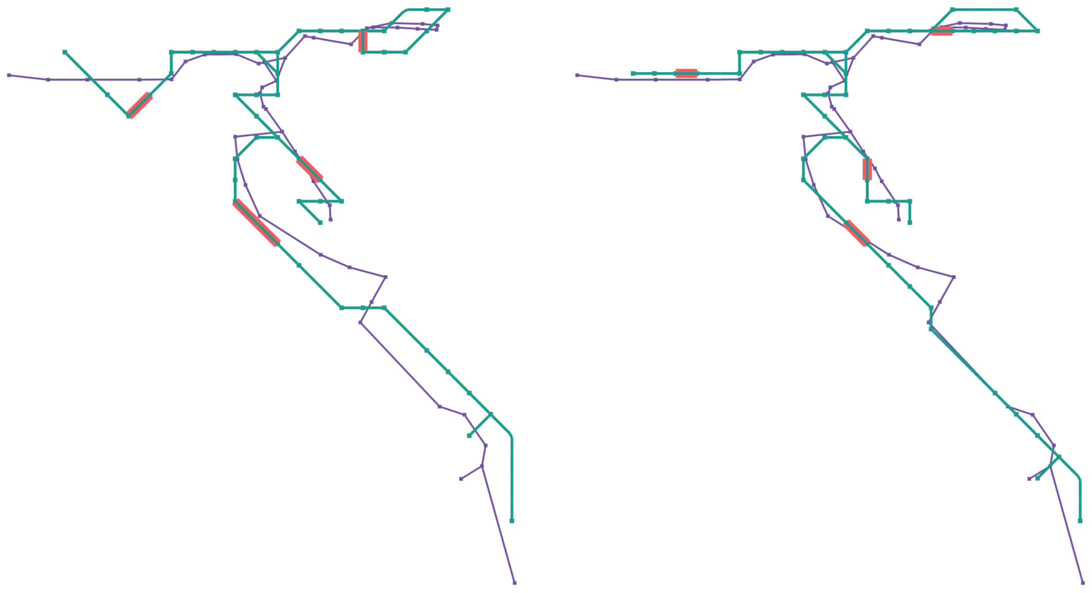
To quantify the correlation between the number of restrictions and success rate or cost, we use the sample Pearson correlation coefficient r [HEK14]. For two paired data sets $x_1, \dots, x_n$ and $y_1, \dots, y_n$ with the respective means $\bar{x}$ and $\bar{y}$, the correlation coefficient is given by

$$r = \frac{\sum_{i=1}^{n} x_i y_i - n\bar{x}\bar{y}}{\sqrt{\left(\sum_{i=1}^{n} x_i^2 - n\bar{x}^2\right)\left(\sum_{i=1}^{n} y_i^2 - n\bar{y}^2\right)}}. \tag{4.2}$$

This is the covariance of the sets, normalized by their standard deviations. Thus, $|r| = 1$ indicates a perfect linear correlation between the sets, that is, the values lie on a line when plotted against each other. Completely uncorrelated sets on the other hand have $r = 0$. A negative Person's r signifies an inverse correlation. As indicated by the colored tables, our values show strong and significant correlations in all cases, see Tab. 4.5.

n_s \ n_p	0	1	2	3	4	5	6	7	8	9	10
0	466	468	475	477	482	490	499	500	509	520	523
1	461	464	475	474	484	491	494	503	509	516	530
2	456	463	467	474	479	489	494	506	502	517	523
3	455	461	467	477	481	488	494	497	513	520	526
4	453	457	467	474	477	487	494	492	513	520	522
5	452	458	468	470	480	489	487	495	513	517	520
6	449	458	465	469	480	493	493	500	504	515	528
7	449	459	465	470	478	490	493	500	511	522	527
8	450	455	463	474	475	488	502	503	513	514	539
9	451	455	461	470	477	492	492	499	514	511	539
10	448	457	466	470	478	486	496	497	503	509	541

Tab. 4.4: Influence of the number of positioned stations n_p and the number of straight edges n_s on the average global cost of successfully created drawings of Würzburg's network. Stronger saturations correspond to a lower cost. The headers are colored according to the mean cost in that column or row.



(a) Before straightening

(b) After straightening

Fig. 4.2: Edges which result in a lower global cost when straightened. By straightening all four highlighted edges, the cost of the graph was reduced from 455 to 423, with the biggest subtraction stemming from the reduced displacement when straightening the lowest edge.

	Success Rate		Global Cost	
	r	p	r	p
n_s	-0.990	6.10×10^{-9}	-0.702	1.61×10^{-2}
n_p	-0.987	2.04×10^{-8}	0.996	5.66×10^{-11}

Tab. 4.5: Pearson correlation coefficients between the number of straight edges or the number of positioned stations with the likelihood of finding a drawing and with the resulting cost. The amount of active restraints of the other kind was again random, from 0 to 10.

4.2 Interactivity of Runtimes

To investigate whether our approach is sufficiently responsive, we briefly discuss the complexity of the steps in the algorithm and then test the response times of different parts of our system. Besides the complete recalculation of the drawing, we are also interested in the performance of the live updates during dragging of stations and that of the local searches.

Complexity For a $X \times Y$ grid graph, our whole algorithm runs in $\mathcal{O}(|E|XY \log XY)$, as did the original one without restrictions, if excluding the local search. Finding the shortest path of each input edge on the grid dominates here, which is done in both approaches. We first build the current graph, which can be done in $\mathcal{O}(|V| + |E|)$, by copying all non-contracted vertices and then adding edges by following the input edges across degree 2 vertices until another non contracted vertex is reached. The complexity of calculating the edge order increases because of the priority system. After finding the original order in $\mathcal{O}(|V| \log |V|)$, we assign a priority for each edge (assuming we can test whether edges are denoted as straight or stations as repositioned in constant time) and then sort by those priorities in $\mathcal{O}(|E| \log |E|)$. For reinserting frozen edges, we have to execute a breadth-first search per group. Since groups have to be connected, we only need to consider the edges in that group and with the different groups not sharing edges, this can be done in $\mathcal{O}(|V| + |E|)$ in total.

The time required for selecting candidate nodes stays the same (in $\mathcal{O}(XY)$ per station), but this step becomes unnecessary for endpoints of frozen edges and positioned stations. Some edges affected by constraints also have a reduced complexity for routing. Frozen edges are trivial again, here we only need to update costs of edges along the path in $\mathcal{O}(XY)$, as we do for every other routed edge as well. For straight edges, where one endpoint is settled, we can start our pathfinding at that vertex and with bend costs being infinite, the effective number of edges and vertices in the graph is in $\mathcal{O}(\max(X, Y))$, since we only have to explore a straight line in all eight directions. So Dijkstra will run in $\mathcal{O}(\max(X, Y) \log \max(X, Y))$ here. On the other hand, if the target of routing is a positioned station, the runtime will in practice be longer, since we have to find that specific vertex, instead of using the first sink of the target set that is reached. This is however still in $\mathcal{O}(XY \log XY)$.

Note that $\mathcal{O}(\Omega) = \mathcal{O}(\Psi) = \mathcal{O}(XY)$ in the grid graph (Ω,Ψ). Furthermore, in the input graph, we can ignore vertices without edges and thus $|V| \in \mathcal{O}(|E|)$. So in total, our runtime is in

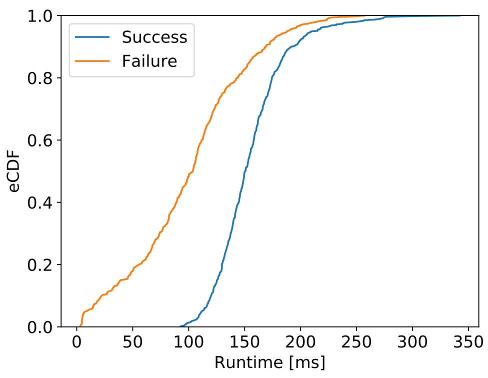
$$\begin{aligned}&\mathcal{O}(|V| + |E| + |V|\log|V| + |E|\log|E| + |V| + |E| + |V|XY + |E|(XY + XY\log XY))\\ &\subseteq \mathcal{O}(|E|XY\log XY).\end{aligned}$$

If we then want to assign the stations with degree 0 to the closest free grid vertex after the algorithm finishes, we need additional time in $\mathcal{O}(|V|XY)$.

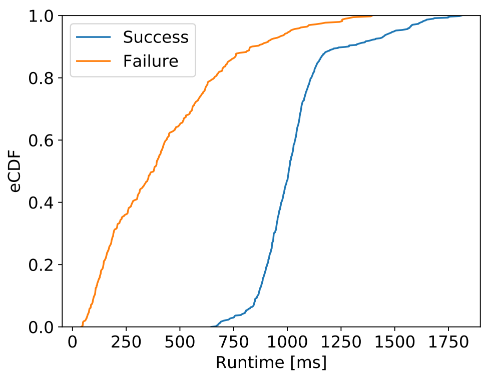
When using the Move Input- or Reposition Station-tools on an input vertex with degree k, we only reroute the adjacent edges and thus achieve a runtime in $\mathcal{O}(kXY\log XY)$. For each step in both types of local search, when exploring the neighboring positions of a vertex, all adjacent edges are routed eight times or less. Since this is done for all vertices, every edge is routed at most 16 times, so the required time is still in $\mathcal{O}(|E|XY\log XY)$. Even though for the search no current graph, edge order, or candidate sets need to be calculated, we expect this to take longer than calculating the drawing to begin with, due to edges being routed multiple times.

Complete Routing To actually test the runtime of the algorithm, we again generate random sets for constraints, this time only using ones where we find a drawing. Note that this could bias some results, as “harder” restrictions may for example result in longer edges which take more time to calculate. On the other hand, restrictions which we cannot solve for will stop the algorithm early, after the first edge that fails to route. To confirm that this second effect outweighs the first, we compared the time required to either find a solution, or fail trying in Fig. 4.3. Here, we used 1,000 restriction sets for each network and chose the maximal number of restriction, so that we achieve a success rate of roughly 50 %. So n_s and n_p for the network of Würzburg were picked from $\{0, \dots, 9\}$ and from $\{0, \dots, 4\}$ for Sydney's. We can see, that only about 15 % of failed runs in Würzburg and 5 % in Sydney take longer than the respective mean time for successful ones, so we can focus solely on the latter.

In order to see the runtime's dependence on added restrictions, we sweep the number of straight edges n_s from 0 to 10, finding 100 restriction sets with a random n_p from $\{0, \dots, 10\}$ for which we find a drawing. We then calculate the average runtime for each n_s and repeat the same with the roles of n_s and n_p swapped. The results are visualized in Fig. 4.4 and the correlations between restriction cost and runtime are listed in Tab. 4.6. We can see, that both types of restrictions negatively influence the runtime. This might seems surprising for straight edges, due to the virtually lowered grid size when routing them. However, our implementation selects an arbitrary endpoint of every edge as the start of the search for a shortest path, so we will only take advantage of that for about half the edges, where one endpoint is settled. The other half and those where both endpoints are still free, instead suffer from the larger candidate radius, which results in a greater number of vertices being visited early in routing. Additionally, straight edges might force their endpoints onto grid positions which causes subsequent adjacent edges to be longer. This, and the reduced size of the target sets, also contributes to the



(a) Würzburg



(b) Sydney

Fig. 4.3: Distribution of runtimes on random restrictions as an eCDF, visualizing the difference in time needed for successful and unsuccessful executions of the algorithm.

	Würzburg		Vienna		Sydney	
	r	p	r	p	r	p
n_s	0.86	7.6×10^{-4}	0.97	5×10^{-7}	0.98	1.2×10^{-7}
n_p	0.92	5.1×10^{-5}	0.32	0.34	0.98	4.2×10^{-8}

Tab. 4.6: Pearson correlation coefficients between runtime and the number of straight edges or the number of positioned stations.

runtime increase with more positioned stations. The placement of stations itself can of course significantly increase path lengths, too.

If we want the user to not be interrupted by this calculation, the SRT should be below 1 s. For smaller networks, our algorithm achieves this pretty consistently, for large ones however, the introduction of too many restrictions breaks this threshold. So our prototype is performing in the right order of magnitude but a bit too slow – especially if we consider the actual SRT, which additionally includes the time it takes to create the line drawing, calculate the global cost, draw the canvas, and clone the graphs for the edit history. But in a more powerful environment than in-browser or with some code optimizations, our approach should be usable even for big networks.

It should be noted, that other parameters can also have a significant influence on the runtime. This is obvious for changes in grid resolution and candidate radius, but for example, a large sink cost will also make the algorithm slower. This is because during pathfinding, the cost for taking the last step toward a sink vertex in the target set will increase accordingly. As we choose the edge with the lowest cost in Dijkstra, a lot of unnecessary edges of the grid are added before any sink edges, postponing when any target can be reached.

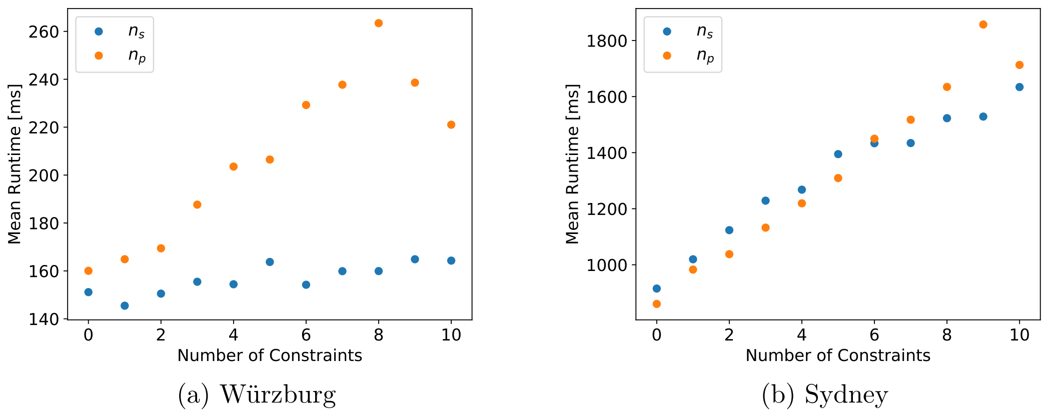


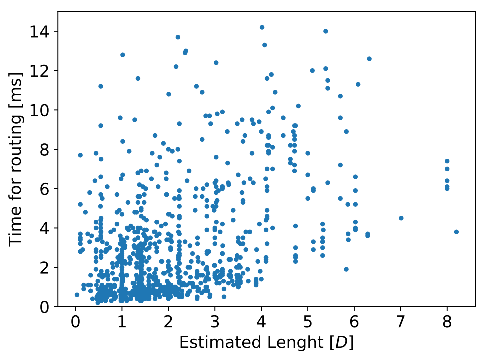
Fig. 4.4: Correlation between runtime and restriction count. In the series where n_s is set, the values for n_p are random and vice versa. For example, the drawings at $n_s = 0$ still include an average of five positioned stations.

Live Preview We are also interested in the runtime for the local recalculations while moving stations. To investigate this, we recorded the time it takes to route a single edge depending and its estimated length. Here, we exclude the preparation of candidate sets and focus just on the execution of Dijkstra's algorithm, because both endpoints have a set grid position when repositioning stations. As a reminder, the estimated length of an edge is the distance between where one endpoint is dragged (so either the position of an input station or a grid vertex) and the gridposition on which the other endpoint is settled. If this distance, normalized by the grid spacing D, is greater than the preview range r_p, no live preview is calculated.

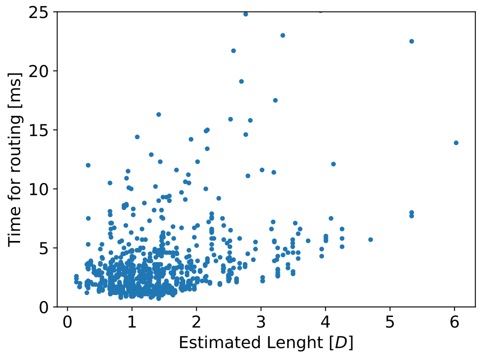
The results of this test over 1,000 edges are visualized in Fig. 4.5. The graphs for these measurements are again subject to randomized restrictions. We can see that higher estimations raise the minimal time required. However, some short edges also took quite long, resulting in a wide scattering of the runtimes. This may be due to the estimation not representing the actual length of resulting paths or external factors varying the calculation speed.

We test these datasets for correlation in Tab. 4.7. We find a significant but weak correlation on all three networks. We also checked how strongly the runtime depends on the actual length of a path, which is what our estimation tries to predict. Here, we see a more pronounced correlation, that is however still not as strong as expected. Again, external factors might be to blame for this. Another factor is that Person's r tests for linear correlations and we would expect the runtime to grow roughly quadratically with the number of hops on a path, since the pathfinding explores the grid in every direction. The minimal values for different lengths in Fig. 4.5 might hint at such a relationship, this could however also be coincidental.

In practice, the calculation of the live preview also includes updating the drawing, which increases with network size. Therefore, the value for r_p should be chosen more



(a) Würzburg



(b) Sydney

Fig. 4.5: Correlation between the runtime of routing an edge and its estimated length, as used for deciding whether to provide a live preview. The distance is given in multiples of the length of a grid cell D. Note that we excluded 15 data points with a higher runtime in (a) (22 in (b)), to keep the lower points discernible.

	Würzburg		Vienna		Sydney	
	r	p	r	p	r	p
Estimated	0.24	6.2×10^{-15}	0.22	3.2×10^{-12}	0.23	1.2×10^{-13}
Hops	0.36	3.6×10^{-32}	0.32	8.9×10^{-25}	0.45	5.4×10^{-51}

Tab. 4.7: Pearson correlation coefficients between runtime for routing an edge and their lengths. We tested the estimation used for deciding whether to provide a live preview and the amount of hops in the resulting grid path.

	Würzburg		Vienna		Sydney	
	t_s	m_s	t_s	m_s	t_s	m_s
SEARCH	(3.1 ± 0.6)s	17 ± 5	(5.1 ± 1.7)s	25 ± 4	(34 ± 12)s	23 ± 6
SEARCH ALL	(2.7 ± 0.6)s	3 ± 1	(4.5 ± 0.9)s	4 ± 2	(25 ± 3.6)s	2 ± 0.2

Tab. 4.8: Runtimes t_s for one step during either type of local search and how many steps m_s were required until no more improvements were found (± standard deviation).

conservatively on large networks. When repositioning just one station in Würzburg or Vienna however, the preview range could be increased to include basically the whole grid, while maintaining a responsive layout, that is, a system response time of less than 0.1 s. The default range is set to a lower value to accommodate for the longer runtime when moving inputs, which also requires the selection and preparation of a candidate set. However, by allowing users to change this value, they can adjust for their systems performance and personal requirements for responsiveness.

Local Search Lastly, we investigate the time needed for running a local search. For this, we again generate random but successfully routed restrictions and run the local search on the resulting map. The runtime for this varies greatly, as the number of steps until no more improvements are found changes depending on the constraints. Thus, we look at the more consistent time needed for a single step and how many of these steps were performed, see Tab. 4.8. Note that our results for this later value are only rough estimates, as we only performed 25 searches per network.

As expected, the runtime per step increases with network size. The type of search should not change this time, as both of them perform routings the same number of times. The observed decrease in runtime for LOCAL SEARCH ALL is likely caused by the browser slowing down as more steps are completed, due to problems with memory management. This effect is stronger for the LOCAL SEARCH as it requires more steps. This is because it just moves one vertex at a time, while LOCAL SEARCH ALL performs multiple improvements in each step. The number of steps required to reach a local minimum seems to depend on the specific network, and not directly from its size. The restrictions also influence this, but to a lesser degree. In general, both methods have a rather high expected runtime, with around 1 min to 13 min for LOCAL SEARCH and 10 s to 50 s for LOCAL SEARCH ALL, depending on the network. Performing these tasks in the background allows user to still interact with the graph during this, but for LOCAL SEARCH, they likely have to wait a long time for the results nonetheless, causing a bad user experience.

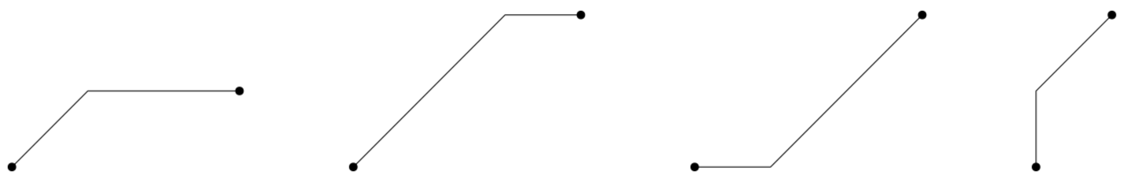


Fig. 4.6: Between these four drawings of the edge, only the first two are seen as equivalent and thus not counted. Here, the right station was moved vertically, which is penalized and so the stretching of the diagonal line segment in the edge to accommodate this change, is ignored. If the edge instead turns in another direction, like in the third image or leaves a station through another port, as in the last drawing, the edge counts as changed.

4.3 Stability of Drawings

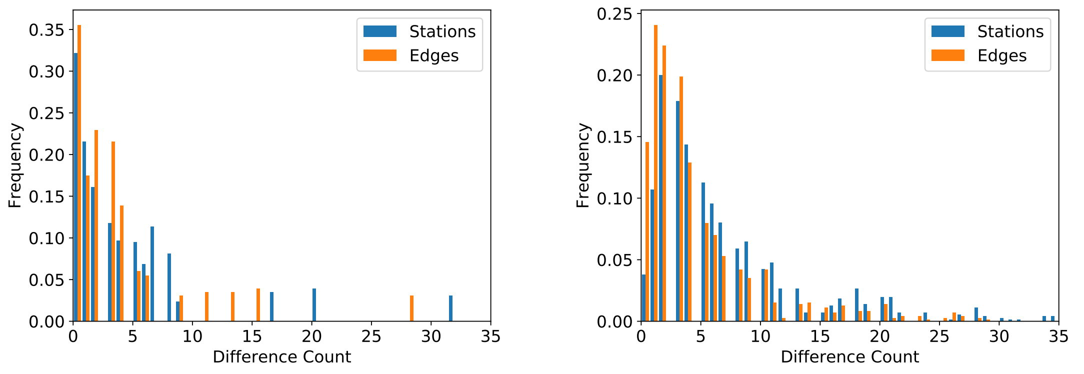
Besides low SRTs, the predictability of how tools behave greatly influences user experience. In our case, this means that there should be few changes to the rest of the drawing, when modifying one part of it. To quantify the difference between two drawings of the same network, we can count the number of stations that changed their grid position and how many edges have a different bend sequence. As shown in Fig. 4.6, the latter means, that an edge is only counted, if it cannot obtain the same shape as its counterpart, by changing the lengths of the line segments it is composed of. This way, changing the position of a station will not also count toward changing all its adjacent edges, if they can simply change in length. Intuitively, it will also be more noticeable and thus undesired to a user, when an edge gains or loses bends, or their angles change, as opposed to them just stretching in one direction.

To measure the unwanted side effects of using a tool, we calculate this difference between the drawings before and after the interaction. Here, we do not count changes directly affected by the new restriction. This means, for a straightened edge, changes in the edge itself and the adjacent stations are ignored. Likewise, for a repositioned station, the movement of the station and modifications of its adjacent edges are excluded from the difference score. We again generate 500 random sets of constraints and create a drawing for them. We then choose one more edge to straighten or station to randomly reposition and recalculate the drawing and its difference to the previous one. If routing fails before or after adding the last restriction, we discard that run.

The resulting average in the number of non-local stations and edges that change are listed in Tab. 4.9. The tendency for these values to decrease with network size is surprising at first, as there are more elements that could be changed. However, a large difference is mainly caused by editing the densest part of the network. If there is not enough room, the changes in the direct neighborhood make it so their surroundings also have to adapt for the drawing to fit. Relative to the size of the graph, Würzburg has the biggest area in which most grid positions are taken, whereas on the larger networks, it is more likely, that one of the parts on the outskirts is edited. These edits usually affect the drawing only locally, as there are enough free grid positions to move to. For Sydney, changes are even less likely to propagate, due to the higher default value for the

	Würzburg		Vienna		Sydney	
	m	n	m	n	m	n
Straighten Edge	3.6 (9)	4.4 (8)	2.2 (6)	3.1 (10)	1.1 (4)	0.6 (2)
Reposition Station	4.7 (10)	7.1 (17)	3.2 (9)	5.6 (15)	1.2 (3)	0.73 (2)

Tab. 4.9: Average number of edges m, that change their bend sequence and number of stations n, that settle on a different grid position, when interacting with a part of the drawing. The values in parentheses are the 90th percentile of that distribution, so for example, when repositioning a station in Vienna, only 10 % of the runs caused more than 15 stations to move.



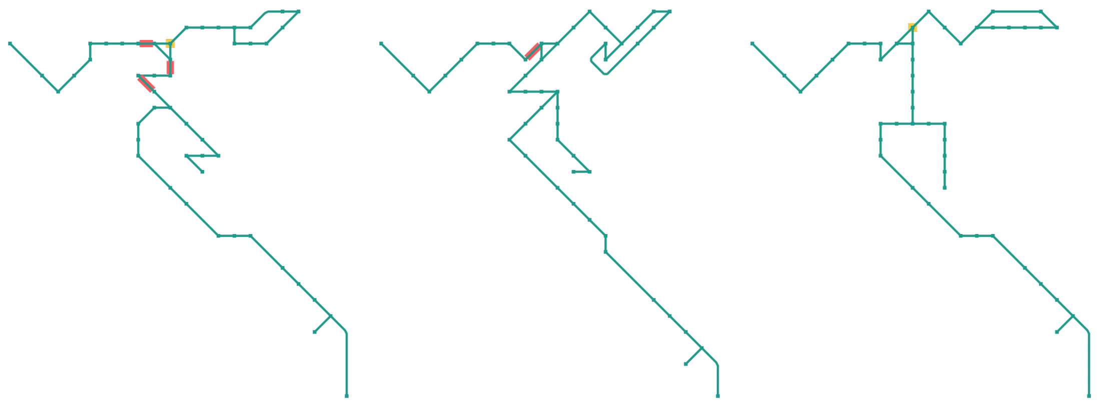
(a) Straighten Edge (b) Reposition Station

Fig. 4.7: Distributions of the amount of edges and stations that are changed when straightening an edge (a) or repositioning a station (b).

move penalty, which causes the metro map to only differ from the input, where it was directly modified. Lastly, there probably is a significant sampling bias again, as random interactions with dense parts of the drawing are unlikely to be be part of a successful routing, especially for the networks, that have a lower success rate to begin with.

For Würzburg, we show the distributions of these values in Fig. 4.7. The shape of the distributions is similar for the other networks, but falls off more quickly. In Würzburg, despite the other randomized restrictions, straightening some edges caused a consistent, large amount of changes. For example, there was only one edge responsible for the difference count of 28 edges and 32 stations. Similarly, the other four peaks above a difference count of 13 were also all caused by two edges, mostly because routing these edges earlier modifies a central part of the drawing, which causes changes in all other parts of the drawing. This effect is demonstrated in Fig. 4.8.

In the distribution for repositioning stations, such peaks are much less pronounced, mainly because there are multiple possible grid positions, where the station can be moved to, varying the resulting drawings. This randomness likely also contributes to the higher number of changes for this type of interaction. Due to the large radius, in which stations



(a) Problematic edges (b) After straightening (c) After repositioning

Fig. 4.8: Strong side effects of interactions. The drawing before adding the constraint is shown in (a), with the edges that cause the biggest changes highlighted. When straightening the top most one, the resulting drawing is (b), which differs in many parts that seem unconnected. Similarly, when instead placing the highlighted station on the closest grid position to its north, the routing will produce (c), with a lot of changed elements in the upper half of the drawing.

can be placed, the displacement of stations is bigger than when straightening an edge, where the endpoints usually only move one or two grid positions (or none). This then influences other parts of the drawing more often.

When actually using this tool sensibly and for small modifications, the difference often seems connected to the interaction, like neighboring stations and their edges being adjusted, or causing changes, that propagate away from the center. However, there are some stations in dense parts of the graph, where repositioning causes unexpected changes, as shown in Fig. 4.8c. In total, for interactions with most elements, the inconsistencies between drawings were not bothering us a lot, sometimes noticeably improving the drawing instead. In cases where they were unwanted, increasing the displacement penalty or undoing the change and freezing the regions that are to be maintained can help alleviate the problem. The changes can also be mitigated by repositioning a station between the edited vertex and the unexpectedly affected area, to stop propagation, or by simply interacting with an element in that area, to increase its priority in the edge order. Some more options for how this could be handled are mentioned in Chapter 5.

4.4 Tools in Practice

In the following, we explain what uses the different tools are well suited for, as well as discussing which shortcomings we noticed:

- Move Input (see Fig. 4.9): Useful for setting the rough position of vertices, like for reducing the density in some regions. Especially when moving many vertices, this

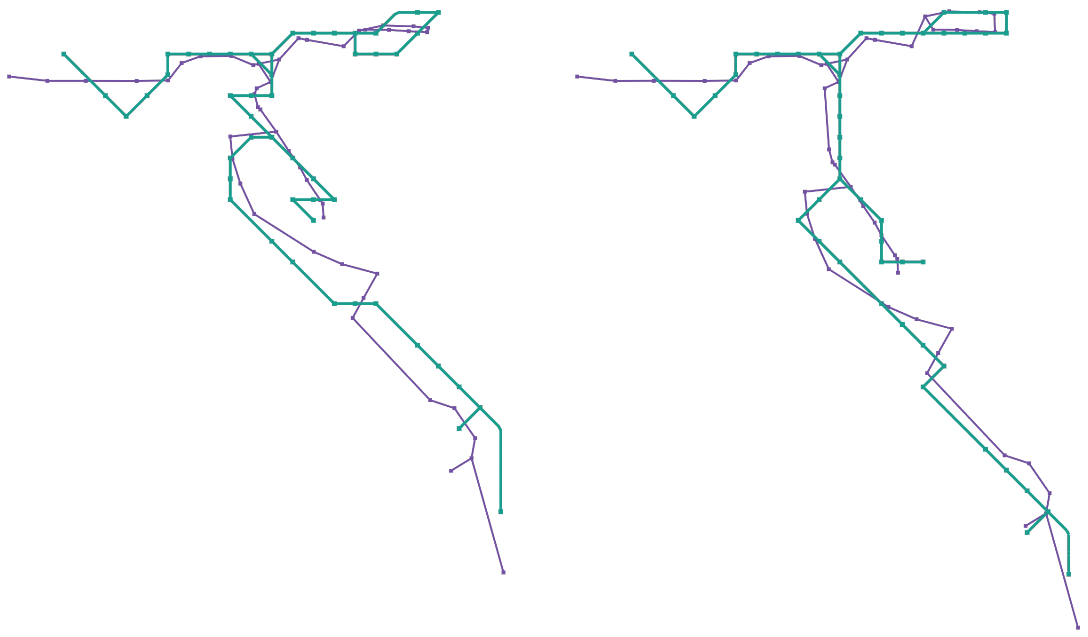
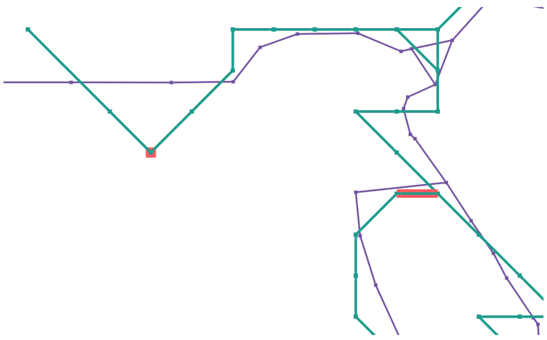
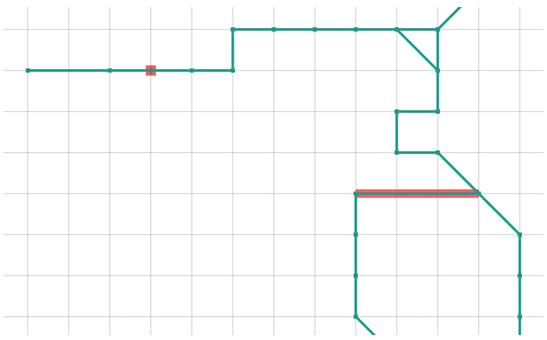

(a) Initial map (b) After moving inputs

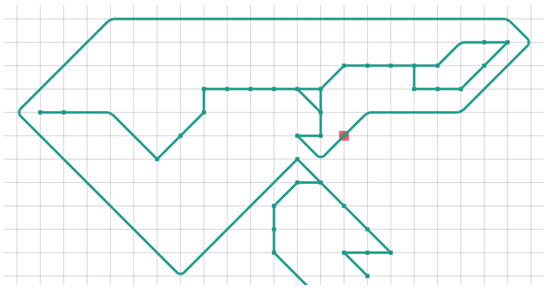
Fig. 4.9: Use cases for Move Input. The bends in the middle of the drawing due to the high density can be prevented by moving the lower half of the graph further down. For the cycle in the top right, we aligned the upper half with a grid line, to get a simpler shape.

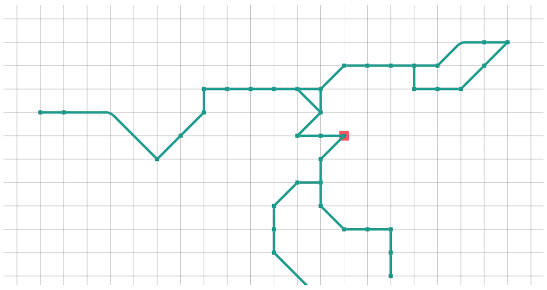
has the advantage over Reposition Station, that the affected vertices are still placed by the algorithm, allowing for relative movement between them. Can also be used to manipulate sections slightly, to align elements with the grid, to prevent bends. More precise interactions like this are hard to realize, as the resulting drawing is somewhat unpredictable and subject to change with other interactions. It might lead to loss of geographic accuracy, but the original graph can still be viewed as a reference.

- Reposition Station (see Fig. 4.10): Good for specifying certain parts exactly, for example, to highlight a geographic feature of the network. Can also help to remove unnecessary bends, if used on the right stations. These are however not always obvious. It might also tempt users into setting many vertices, disallowing potential improvements through the algorithm. Dragging vertices across other edges can lead to very long edges and thus unresponsiveness, due to the live preview, or even failed routings. Sometimes the preview routing also fails, preventing the action, while a full recalculation might work. In these cases, setting the preview range to 0 can help. Lastly, not permitting users to move stations to occupied grid positions is annoying, when the occupying spot would be freed through rerouting. This can be bypassed by the more cumbersome method of also moving the occupying vertex and freeing it afterwards again.


(a) Initial map


(b) After repositioning stations


(c) During live preview


(d) After complete routing

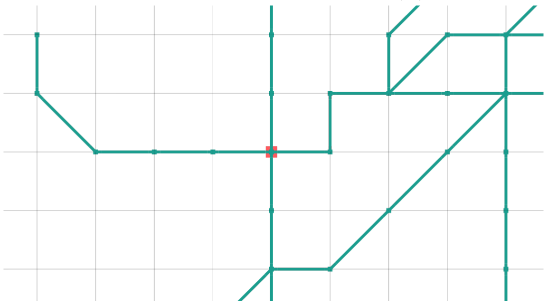
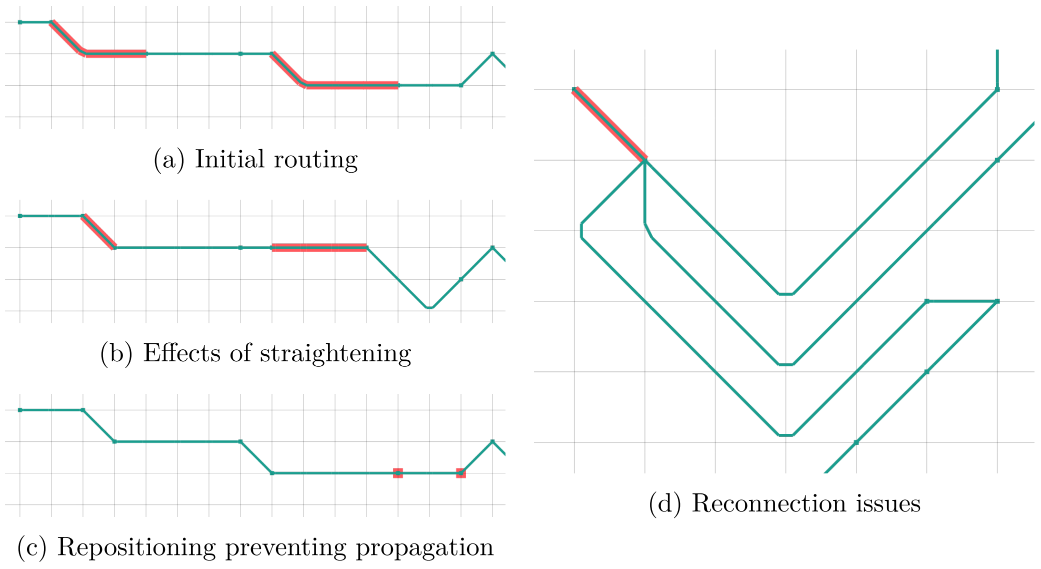

(e) Unmovable station

Fig. 4.10: Use cases for and issues with Reposition Station. At the highlighted edge in (a), the rails cross a river. By dragging the endpoints of it apart, we emphasize this landmark for orientation in (b). At the same time, by simply moving the highlighted vertex up two grid positions, we can align the path with its geography and remove bends. The locality of the live preview can result in very bad edges, even if a full recalculation of the drawing creates very reasonable edges. For example, in (c) we are dragging the highlighted vertex three spots east from where it was, forcing an edge to circumnavigate half the graph. When letting go, the drawing recovers as shown in (d). Lastly, the highlighted vertex in (e) cannot be moved to the occupied position to the left, even though that restriction would result in a successful routing. Furthermore, dragging it to any of its free diagonal neighbors is not allowed, because the live preview fails, since there is not enough room to route all four edge.


(a) Initial routing

(b) Effects of straightening

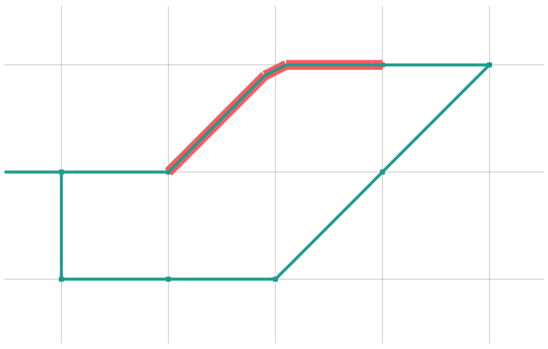
(c) Repositioning preventing propagation

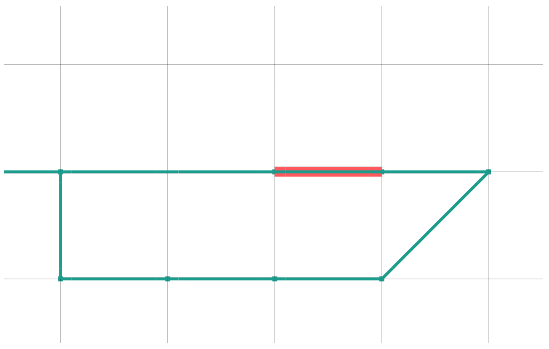
(d) Reconnection issues

Fig. 4.11: Problems with Straighten Edge. Straightening the highlighted edges in (a), simply moves the bend along to a station or to another edge, as in (b) for the left and right edges respectively. The latter effect was prevented in (c), by positioning the highlighted endpoints. Figure (d) shows awkward routings stemming from straightening the highlighted edge. This caused it to be routed first, independently of the center of the network to the right. Unfortunate angles in the first (top) connecting edge forced the others onto long paths as well.

- Straighten Edge: Has little applicability on a single edge, when the drawing mainly consists of short edges. In general, straightening single edges usually results in the bend occurring at one of the endpoints, or the less favorable option, of another edge being bend instead, as shown in Fig. 4.11. This can be counteracted by positioning the endpoints, which may however lead to failed routings. It is more useful for longer chains, however, due to the contractions, a local search then becomes necessary if a minimal distance between stations is to be maintained. The main use case for us was to align all stations on paths on the outskirts of the drawing, see Fig. 4.12. The high priority might however, causes them to be drawn with a minimal length, distorting the surroundings, which makes the user loose a sense of structure in the network, as in Fig. 4.11d. This is partially fixed by the compression penalty during local search, or can be avoided by excluding a degree 2 vertex from the path as a buffer.

- Don't Contract and Local Search: Similarly to straightened edges, manually contracted stations necessitate a local search at some point, which reduces their usefulness: Before the search, the edge does not behave like it will after "decompression" and afterwards the graph is less interactive due to most edges being frozen. The local search feels disruptive to the creation process in general because of this. If used when most interactions are already included, it can provide some small improvements, especially to compact the drawing (more so, if no vertices were contracted).


(a) Initial routing


(b) After straightening

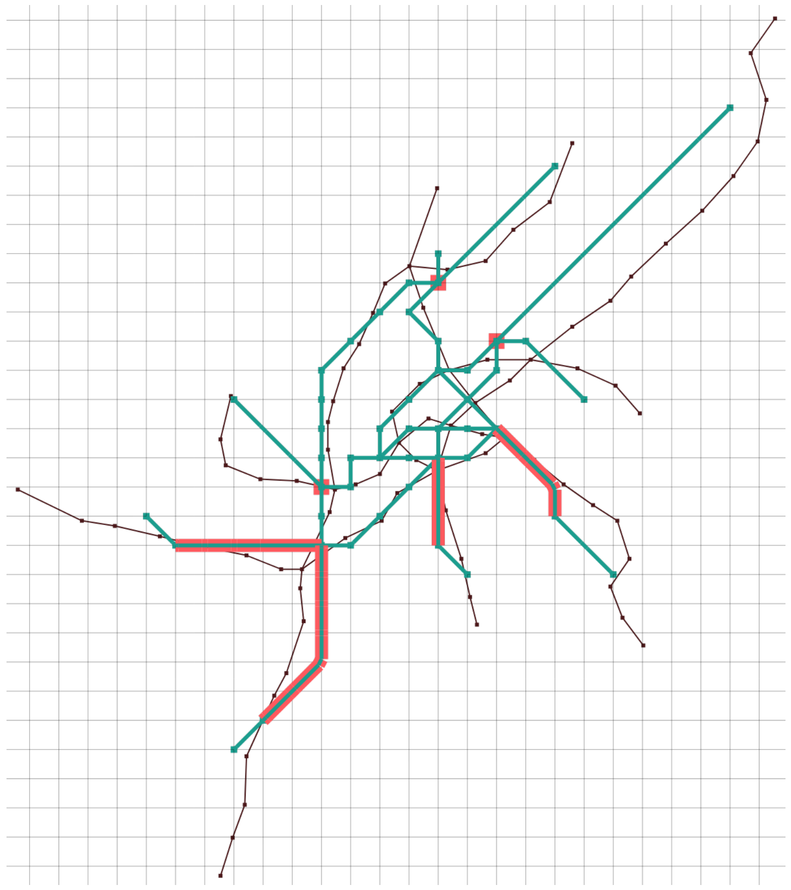
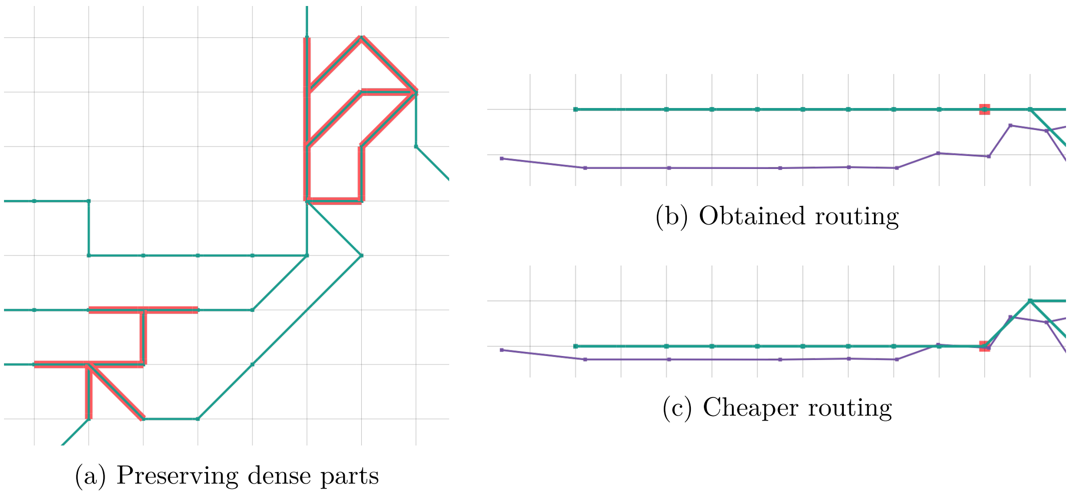

(c) Straightening of non-central paths

Fig. 4.12: Uses for Straighten Edge. Straightening the highlighted edge in (a) and the accompanying priority increase delivers a great result, as shown in (b). In (c), a common use case is shown, namely the straightening of paths on the periphery. Here, a mix of positioned stations (highlighted vertices) and non-contracted buffer edges (highlighted paths) prevents the distortion of central edges.


(a) Preserving dense parts

(b) Obtained routing

(c) Cheaper routing

Fig. 4.13: Use cases and issues with FREEZE EDGES. We modified crowded regions to our liking in (a) and then froze relevant parts (highlighted) to be able to edit the rest of the graph, without having to worry about them changing. This also helps with success rate, as these dense parts can otherwise become harder to route, when the edge order or neighboring edges change. In (b) the straight path to the left of the highlighted vertex is frozen. Since the highlighted vertex is placed first, it forced the rest of the frozen group onto the same y-Position. If the whole group was moved down, as in (c), the drawing would have higher geographical accuracy and a lower cost.

- FREEZE EDGES (see Fig. 4.13): Useful if there are separate dense sections of the network that require some manual improvements, after which they can be frozen. Since routing of the whole group is based on the first vertex and thus does not consider the displacement of the other stations in the group, a significant deviation from the geography is possible in some cases. Another annoyance is having to unfreeze a group, to straighten an edge in it, or, to contract an adjacent station. On the other hand, repositioning a whole frozen group as one unit feels intuitive.
- Selections and removing restrictions: The selection tools were adequate for most cases, since work on groups mainly happens on sections away from the center, "Select component" (Q) often only needs few adjustments using "Edit selection" (E). For straightening edges, the "Select path"-mode (W) is well suited, as here only paths are eligible anyways. However, the selection for freezing edges was cumbersome in some dense parts, where most had to be added individually, for example in Fig. 4.13a. Similarly, having to click every single vertex or edge when removing their constraints, or when setting contractions, is unpleasant.

We quantified the improvements through local searches by comparing the global cost of 25 drawings generated from random constraints before and after the search. The cost reductions for both search methods are listed in Tab. 4.10. The slower LOCAL SEARCH shows a slight tendency toward performing better than LOCAL SEARCH ALL. The improvements possible through either method do however depend on the network. In total, they did not decrease the cost in a meaningful way.

	Würzburg	Vienna	Sydney
Local Search	(5.6 ± 2.3) %	(4.8 ± 1.0) %	(1.8 ± 0.4) %
Local Search All	(4.5 ± 1.6) %	(4.1 ± 1.7) %	(1.9 ± 0.3) %

Tab. 4.10: Mean cost reduction through local search (± standard deviation).

	Montreal	Würzburg	Vienna	Sydney	Washington
Time	<2 min	<4 min	<15 min	<18 min	<21 min
Interactions	8	25	78	83	104
Failures	0	0	8	5	12
Stations	65	49	84	174	98

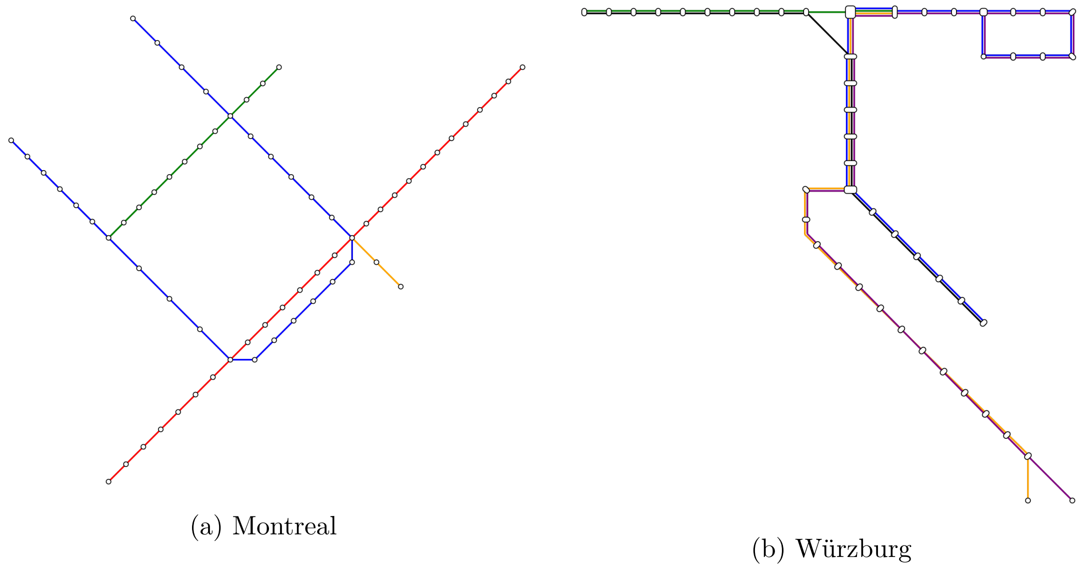
Tab. 4.11: Time and number of interactions required for finishing a metro map on different networks, as well as the number of interactions that resulted in a failed routing. The number of stations indicates the size of the network

Lastly, we present some maps created using our prototype. We recorded the time required and the number of interactions, as well as how often they resulted in a failed drawing. These values are listed in Tab. 4.11. It should be noted, that our familiarity with how the tools work speeds up the process and reduces the number of failed drawings. The quality of the maps on the other hand could likely still be greatly improved by an experienced designer. In general, we can see that small networks can be turned into metro maps in very little time using few interactions, while bigger ones require significantly more work. Another relevant factor are any specific goals the designer has in mind.

For example, if the designer does not mind different paths having an inconsistent spacing between stations, they can contract almost all vertices and use the equidistant reinserting on every edge, like we did on the Montreal network, see Fig. 4.14a. We then only had to aligned the corners of the central diamond to achieve a decent drawing.

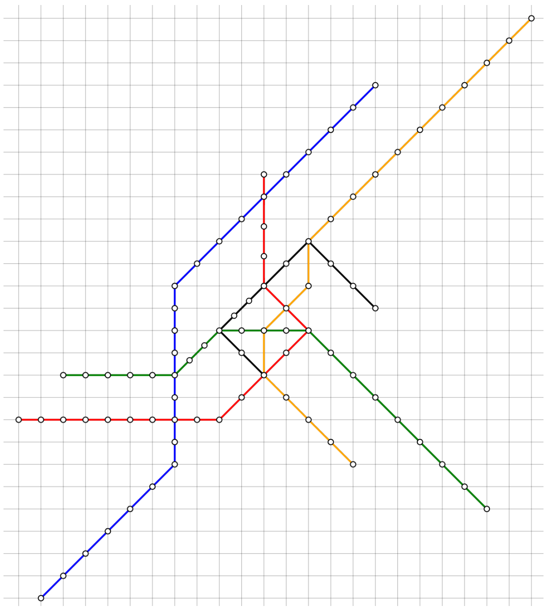
For Würzburg, see Fig. 4.14b, we tried to keep edges roughly the same length. To do so, we again contracted most stations, then adjusted the distance between the endpoints for an equal spacing of stations. This leads to some expansion in the more dense parts and edges between non-contractable stations are drawn as longer, since their endpoints are forced onto the grid.

In the map for Vieanna, see Fig. 4.14c, we stick to the grid almost completely by not contracting any vertices, except two in the most dense part, to allow for a more compact center. This grid alignment creates a tidy look. This is important in this network, since some edges run along each other, so different spacings between stations would be quite noticeable. Creating the middle of this map was somewhat cumbersome, as repositioning stations is often either disallowed by our prototype or fails, when most adjacent grid edges are occupied. For the straight edges on the outskirts, we moved the inputs to align the stations and then repositioned the component to graft it onto the center.



(a) Montreal

(b) Würzburg



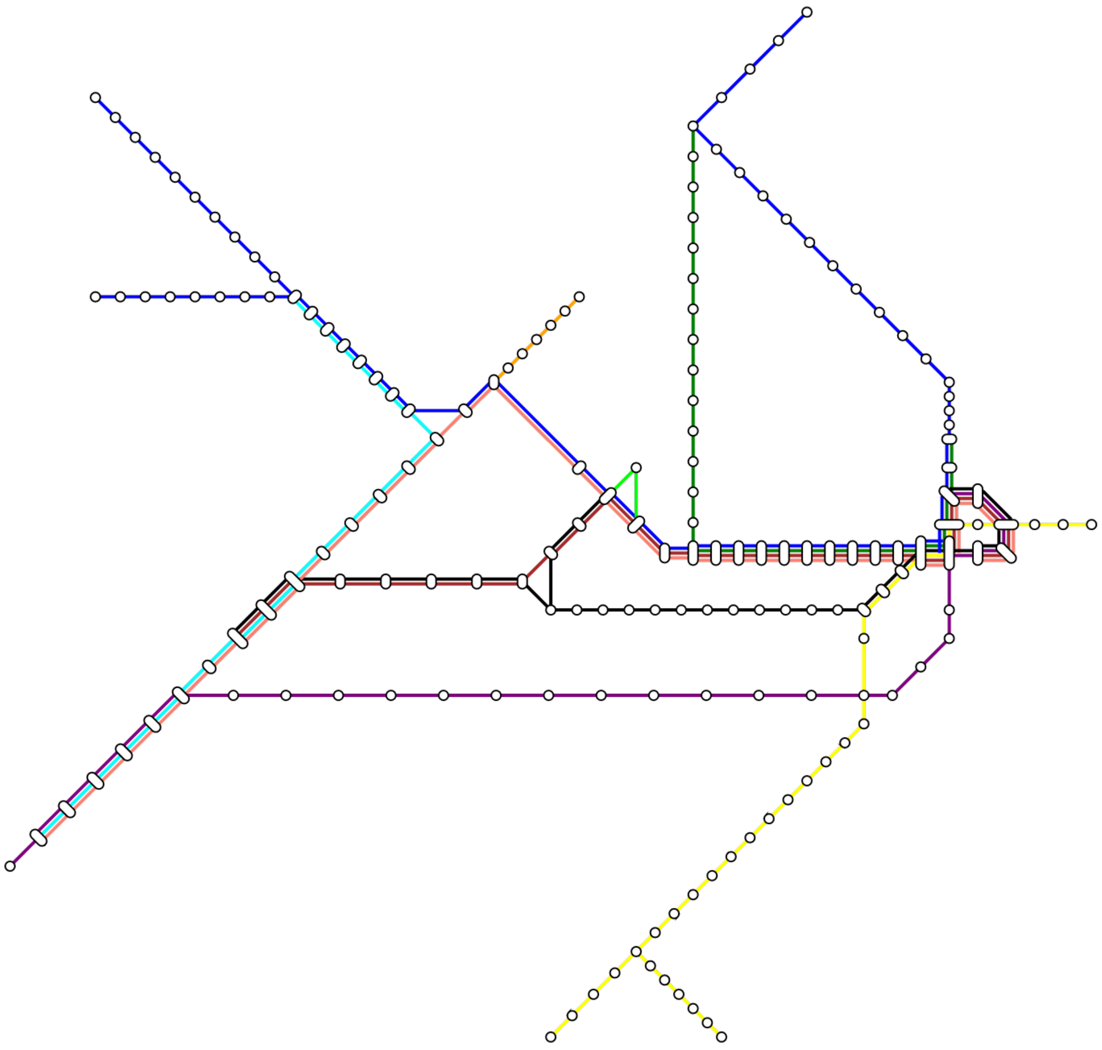
(c) Vienna

Fig. 4.14: Maps of small and medium-sized networks created using our prototype. The drawings show increasing levels of uniformity in edge length due to how contractions are handled. In (c) almost all stations lie on gridpoints.

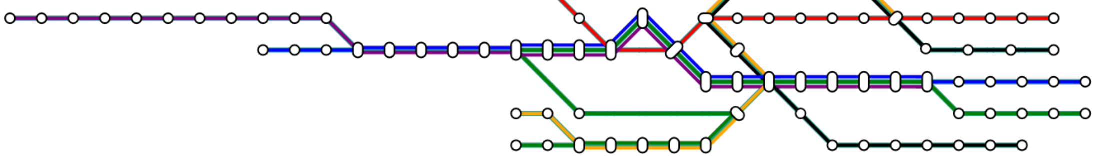
The dense part of Sydney, see Fig. 4.15a, provided similar challenges to that of Vieanna and required the most work. After manipulating this part to look satisfactory, we froze the corresponding edges and adjusted the rest of the map. We straightened a lot of the long paths, but only after repositioned the endpoints onto grid vertices that lay on a line. This prevents them from settling onto positions from which the other adjacent, straight edges are impossible to route.

For the final map we created, Washington, see Fig. 4.15b, we mostly disregarded geographic positions and instead tried to create a very compact drawing which still gives an overview of the lines and the stations. Such designs are frequently found in actual metros because of their clear structure. To achieve such a look, we first contracted most vertices and adjusted the resulting edges to align with the edge which has the most lines. We then refined the resulting routing by repositioning stations. Again, the edges being next to each other can make some interactions require multiple attempts. For this map we also focused on all vertices landing on gridpoints for consistent spacing. To do so, we had to un-contract and position the vertices at the 135°-bends, in order to avoid automatic reinsertion on paths with both axis-aligned grid edges and the longer diagonal ones. For example, if the second station from the left on the orange line is not positioned, it would be drawn a bit further right on that line.

We noticed the performance of our prototype drop after about 30 interactions on larger networks. We could remedy this by clearing previous version of the graphs from the edit history. Apart from some slow response times and failed routings, using the system is enjoyable and the tools can be used to achieve most goals. The time needed to create a map is also less than in a fully manual process and requires only few interactions on small networks. Furthermore, the added restrictions improved the initial map in all cases and allowed us to achieve a variation of styles.



(a) Sydney



(b) Washington

Fig. 4.15: Maps of larger networks created using our prototype. In (a) we focused more on geographic accuracy, whereas (b) is stylized and more compact.

5 Conclusion and Future Work

In this thesis, we explored an interactive process of creating metro map layouts which algorithmically integrates changes requested by a human designer. For this purpose, we adapted an existing approach for creating metro maps that uses shortest paths on an auxiliary graph to draw edges. By adding systems that handle additional constraints on certain elements, we can let a designer influence the drawing, while still benefiting from optimizations of our algorithm. We created a prototype which realizes this system and enables users to directly act on the metro map in multiple ways.

We provide two tools which can be used to change the locations of stations in the drawing (Move Input and Reposition Station) and two more that deal with the shape of groups of edges, removing their bends (Straighten Edge) or locking their curves (Freeze Edges). Furthermore, we let users opt into pre- and post-processing steps, namely the contraction of vertices and a local search. By also modifying the order in which edges are added, we are able to successfully create maps of smaller networks most of the time, even when there are many restrictions set. The system response time achieved by our prototype is also low enough for its use to feel responsive on moderately-sized networks.

While all tools have their use cases, some of them behave unexpectedly or create undesirable side effects, requiring additional steps to achieve the intended result. To this end, Reposition Station is probably the most useful tool for predictable, concrete changes to the map, whereas Move Input more indirectly suggests vertex placement. Straighten Edge is best suited to reduce complexity on the fringe of the network and Freeze Edges can be used to increase the consistency of the drawing across multiple edits. Changing costs and parameters, manually contracting stations, and the local search also provide helpful functions, yet are less useful during the main editing phase. They are instead most applicable at the start or end of the creation process.

All in all, our approach can be used to create layouts quite quickly and without too many steps. These maps are better than those created by the algorithm itself, but more importantly, they can realize different requirements special to the network, as determined by the designer. Because of the adaptability to different inputs thanks to the inclusion of a human, we believe that interactive approaches to metro map design provide a good alternative to fully automatic processes.

This conclusion does however come with the caveat that our evaluation in regards to map quality and user experience are based solely on our own usage. Thus, a user study should be conducted, including designers, who previously worked on metro maps. On the one hand, to better judge the usability of resulting maps and on the other hand, to see how tools are used, where they fall short and which additional functions would help during the creation process. For example, a tool that allows users to specify the bend angle at a station might be useful, or one to change the grid's resolution in selected regions.

Bast et al. proposed some ideas of how to modify the grid to achieve certain effects themselves [BBS20]. Some of these could be employed interactively. They create drawings that mainly use edges aligned in a certain direction, by increasing the hop costs for the others. They also used changes in costs to prohibit routing through parts of the grid that correspond to geographic features, like lakes or mountains. These are well suited to interactive use and could be implemented in our tool with little additional effort. Users could draw a polygon that they want unoccupied, or select a region where edges of a certain alignment are discouraged. The authors also experimented with assigning grid edges a cost based on their distance to the geographic course of the rails. This could be expanded to allow users to create paths or locations which influence the costs of surrounding edges, potentially with different sets for each input edge. We imagine that this could feel quite intuitive, with the added features attracting or repelling edges.

Furthermore, there are some improvements to our tools, that could lead to a better performance, or make them less cumbersome to use. We already mentioned that for lower runtimes, we should start routing every edge on the endpoint that has already settled, if available. When the other endpoint is a degree 1 vertex, the target set for routing can include all unoccupied grid vertices for a shorter path. To achieve a higher success rate, other edge orders, like the original or a randomized one, could be tried after ours fails. This would however decrease the drawing's stability. To counteract this, a cost for changing the shape of edges could be added, but this might lead to lower quality results.

To prevent drawing differences when using the displacement tools specifically, a way for users to revert to the routing at the end of the live preview and freezing the edges as they are, could be useful. Reposition Station can also be made more enjoyable to use by quietly skipping the preview if it fails and then attempting a global recalculation, even if another station settled on the targeted spot. Frozen edges can be made more user friendly by automatically unfreezing edges that are straightened or added/removed due to contractions, splitting the frozen group, if necessary. For frozen edges, unconnected groups also seem useful in some cases, e.g. to create parallel paths. The options for making selections can also be refined in general, for example, by providing methods to change the selection by more than one element at a time. A rectangular selection tool would be particularly helpful when removing restrictions.

The local search also has potential to be more powerful, if restricted edges are treated differently. For example, when moving one endpoint of a frozen edge, the other one needs to be translated in the same direction, or the routing is guaranteed to fail. Similarly, moving a vertex in a straight edge sometimes necessitates the other endpoint to follow. In an environment where concurrent calculations can be realized more easily, it might also make sense to execute a local search at every step, to serve as a preview and inspiration for the designer. Lastly, our edit history can be adapted to allow users to simply toggle each restriction on or off. This could be expanded, to include a slider for each modification, indicating how important the designer deems it. This could be implemented by regulating restrictions through increased costs, instead of as absolutes.

Thus, our approach serves as a proof of concept for interactive metro map design and our prototype has the potential to be easily expanded to include other features.

Bibliography

[BBS19] Hannah Bast, Patrick Brosi, and Sabine Storandt: Efficient generation of geographically accurate transit maps. *ACM Transactions on Spatial Algorithms and Systems (TSAS)*, 5(4):1–36, 2019.

[BBS20] Hannah Bast, Patrick Brosi, and Sabine Storandt: Metro maps on octilinear grid graphs. In *Computer Graphics Forum*, volume 39, pages 357–367. Wiley Online Library, 2020.

[BBS21] Hannah Bast, Patrick Brosi, and Sabine Storandt: Metro maps on flexible base grids. In *17th International Symposium on Spatial and Temporal Databases*, pages 12–22, 2021.

[CR11] Daniel Chivers and Peter Rodgers: Gesture-based input for drawing schematics on a mobile device. In *2011 15th International Conference on Information Visualisation*, pages 127–134. IEEE, 2011.

[GS81] Thomas J Goodman and Robert Spence: The effect of computer system response time variability on interactive graphical problem solving. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(3):207–216, 1981.

[Guo11] Zhan Guo: Mind the map! the impact of transit maps on path choice in public transit. *Transportation Research Part A: Policy and Practice*, 45(7):625–639, 2011.

[HEK14] Joachim Hartung, Bärbel Elpelt, and Karl Heinz Klösener: *Statistik*. Oldenbourg Wissenschaftsverlag, 2014.

[HMDN04] Seok Hee Hong, Damian Merrick, and Hugo AD Do Nascimento: The metro map layout problem. In *International Symposium on Graph Drawing*, pages 482–491. Springer, 2004.

[HMdN06] Seok Hee Hong, Damian Merrick, and Hugo AD do Nascimento: Automatic visualisation of metro maps. *Journal of Visual Languages & Computing*, 17(3):203–224, 2006.

[Nie94] Jakob Nielsen: *Usability engineering*. Morgan Kaufmann, 1994.

[Nöl14] Martin Nöllenburg: A survey on automated metro map layout methods. In *Schematic Mapping Workshop*, 2014.

[NW10] Martin Nollenburg and Alexander Wolff: Drawing and labeling high-quality metro maps by mixed-integer programming. *IEEE Transactions on Visualization and Computer Graphics*, 17(5):626–641, 2010.

[SRMOW10] Jonathan 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(1):101–114, 2010.

[vDL18] Thomas C van Dijk and Dieter Lutz: Realtime linear cartograms and metro maps. In *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 488–491, 2018.

[WC11] Yu Shuen Wang and Ming Te Chi: Focus+ context metro maps. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2528–2535, 2011.

[WNTN19] Hsiang Yun Wu, Benjamin Niedermann, Shigeo Takahashi, and Martin Nöllenburg: A survey on computing schematic network maps: The challenge to interactivity. In *Proceeding of the 2nd Schematic Mapping Workshop*, 2019.

[WP15] Yu Shuen Wang and Wan Yu Peng: Interactive metro map editing. *IEEE Transactions on Visualization and Computer Graphics*, 22(2):1115–1126, 2015.

[WTH$^+$13] Hsiang Yun Wu, Shigeo Takahashi, Daichi Hirono, Masatoshi Arikawa, Chun Cheng Lin, and Hsu Chun Yen: Spatially efficient design of annotated metro maps. In *Computer Graphics Forum*, volume 32, pages 261–270. Wiley Online Library, 2013.

Erklärung

Hiermit versichere ich die vorliegende Abschlussarbeit selbstständig verfasst zu haben, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben, und die Arbeit bisher oder gleichzeitig keiner anderen Prüfungsbehörde unter Erlangung eines akademischen Grades vorgelegt zu haben.

Würzburg, den 01. Dezember 2021

. . . .
Tim Janiak