



## **Master Thesis**

im Rahmen des Universitätslehrganges  
**"Geographical Information Science & Systems" (UNIGIS MSc)**  
am interfakultären Fachbereich für Geoinformatik (Z\_GIS)  
der Paris Lodron-Universität Salzburg

zum Thema

# **Von der topographischen Karte zum Liniennetzplan**

**Möglichkeiten der Automatisierung der graphischen  
Nachbearbeitung**

vorgelegt von

**Dipl. Ing. (FH) Thomas Zuberbühler**

103402, UNIGIS MSc Jahrgang 2014

zur Erlangung des Grades

"Master of Science (Geographical Information Science and Systems) - MSc (GIS)"

Zürich, den 4. Oktober 2016

# Danksagung

Danken möchte ich dem ganzen UNIGIS-Team für die hervorragende Betreuung während meiner Studienzzeit, insbesondere Ao. Univ. Prof. Dr. Josef Strobl und Dr. Gudrun Wallentin für die Betreuung meiner Masterarbeit und die hilfreichen Rückmeldungen auf meine Fragen.

Ein ganz besonderer Dank gilt meinem Stiefvater, Heinz-Günther Susssdorf, welcher mich mit zahlreichen Stunden Korrekturlesen unterstützt und als Fachfremder aufgezeigt hat, wo zusätzlicher Erklärungsbedarf bestand. Darüber hinaus hat er mich moralisch unterstützt und mich fortlaufend motiviert.

Ebenfalls fürs Gegenlesen mit fachlichem Blick möchte ich mich bei Michael Gerber, Nick Bucher und Tino Tschärner bedanken.

Meinem Arbeitgeber, HRM Systems AG, und meinen Arbeitskollegen möchte ich für die Unterstützung und die Möglichkeit danken, mein Studium berufsbegleitend zu absolvieren und die vorliegende Masterarbeit zu schreiben.

Weiterhin bedanke ich mich bei meiner Familie, Bruno Zuberbühler, Heidi Susssdorf-Jenni, Nadja Näf und Lis Zuberbühler, sowie bei meinen Freunden, die mich moralisch unterstützt haben und mir viel Geduld entgegen gebracht haben.

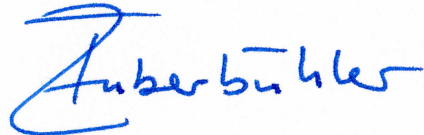
Besten Dank auch an die jeweiligen Transportunternehmen für die freundliche Genehmigung der Verwendung von Ausschnitten aus deren Liniennetzplänen.

# Eidesstattliche Erklärung

Ich versichere, diese Master Thesis ohne fremde Hilfe und ohne Verwendung anderer als der angeführten Quellen angefertigt zu haben, und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen wurde. Alle Ausführungen der Arbeit die wörtlich oder sinngemäss übernommen wurden, sind entsprechend gekennzeichnet.

Zürich, den 4. Oktober 2016

Thomas Zuberbühler

A handwritten signature in blue ink, reading "Zuberbühler". The signature is written in a cursive style with a large initial 'Z'.

# Kurzfassung

Die vorliegende Masterarbeit beschäftigt sich mit der Automatisierung von Teilaufgaben der manuellen graphischen Nachbearbeitung von Liniennetzplänen. Liniennetzpläne von Verkehrsnetzwerken sollen den Passagieren die Orientierung erleichtern. Zu diesem Zweck werden topographische Karten soweit vereinfacht, dass nur die notwendige Information ersichtlich ist. Weltbekannt wurde der Liniennetzplan der Londoner U-Bahn (Tube Map) von Henry Beck. Auf ihn gehen Regeln für die Darstellung von schematischen Liniennetzplänen zurück, z.B. die Darstellung von Streckensegmenten mit 45 Grad Winkeln oder einem Vielfachen davon (Oktilinearität). Solche Liniennetzpläne enthalten graphische Elemente (Signaturen), welche die Netzwerkinformation sowie Landmarken und eventuell Points of Interest repräsentieren.

Im Zuge seiner Literaturrecherche kommt der Autor zum Ergebnis, dass bereits eine Anzahl von Ansätzen zur automatisierten Erstellung von Liniennetzplänen in der Fachliteratur beschrieben wird. Einzelne dieser Ansätze erweisen sich als nützlich, um einen schematisierten Liniennetzplan zu erstellen, welcher ein Streckennetz abbildet, jedoch keine detaillierten Routeninformationen und auch keine anderen Kartenelemente berücksichtigt. Ausgehend von einem solchen schematisierten Streckennetz beschäftigt sich der Autor mit den Problemen, die entstehen, wenn dem Liniennetzplan Informationen hinzugefügt werden. Die Darstellung des reinen Streckennetzes bietet vielfach zu wenig Platz, um alle benötigten Informationen einzupflegen. Der Platzbedarf dieser Informationen bedingt eine laufende Anpassung der graphischen Darstellung, die immer komplexer wird, je mehr Informationen hinzugefügt werden. Hier setzt der Autor mit seiner Methode der Automatisierung an. Er entwickelt ein Verfahren, um den Platzbedarf der Routeninformationen und Haltestellen-Signaturen zu ermitteln. Um den benötigten Platz zu schaffen, müssen benachbarte Elemente iterativ verschoben (verdrängt) werden. Der Vorgang der Verdrängung erzeugt vor allem mit dem Kriterium der Oktilinearität Konflikte, die der Autor mit Hilfe der von ihm entwickelten Algorithmen korrigiert. Anhand von Beispielen werden zwei Korrekturverfahren demonstriert, wobei eines sich des Verfahrens der Skalierung bedient und das andere Knoten iterativ verschiebt, bis der Konflikt gelöst ist. Das zweite Verfahren erfordert unter Umständen die Wiederherstellung der Oktilinearität durch weitere Verschiebeprozesse.

Die Problemstellungen von Liniennetzplänen sind so unterschiedlich, dass immer wieder Einzelprobleme auftreten, die durch ein Standardverfahren nicht gelöst werden können. Auch im Beispiel des Autors ergeben sich Problemfälle, die mit seinem Ansatz nicht ge-

löst werden können, was aber erst bei der Implementierung deutlich wird. Für die Praxis bedeutet dies, dass immer mit Sonderfällen bei der Erstellung von Liniennetzplänen zu rechnen ist, die der manuellen Nachbearbeitung vorbehalten bleiben.

# Abstract

The purpose of this study is to investigate the automatization of manual post-processing graphical requirements of metro maps. Metro maps should ease the passenger's orientation within the network of a public transport system. For that reason, instead of using topographical maps, a simplified map is usually presented omitting irrelevant information. Probably the most famous metro map is Henry Beck's Tube Map of London City. Most of the today's used layout rules for schematic metro maps trace back to him, e.g. presenting the lines within an angle of 45 degree or a multiple of it (octilinearity). Beside network information, such maps might include graphical elements (signatures) such as landmarks and points of interest.

In the course of his study of literature the author came across variety of papers discussing approaches to generate a metro map. Some of this approaches are beneficial to generate metro maps of a simple network such as a underground train system. However, all of them do not address the issue of route segments with several routes such as often found in continental Europe where sometimes 10 or more routes are running on route segments. In addition, other graphical elements such as landmarks are not taken into account. Starting with an octilinear graph layout of a railway network, the author focusses on problems arising from adding route information. In most of the cases there is not enough space between the given map elements to include more information. The map must be changed to add such elements. This is an iterative process increasing the complexity with each iteration. The author develops an algorithm to solve the problems arising with this use case. He presents a method to determine the necessary space in order to add route information and station signatures. To increase space, neighbouring nodes are iteratively displaced. This process produces non-octilinear edges (lines) which are corrected by the author's algorithm. Based on examples, two approaches are presented. One approach uses scaling and the other moves nodes iteratively until enough space is available. The second approach may requires the restoration of the edge's octilinearity with further shift processes.

Problems of metro maps are often varying from map to map. Hence, these individual problems can not be solved completely with a routine method. This applies also to the author's examples. Whilst implementing, problems are produced which can not be solved with his approach. The author concludes that in real situations one has always to reckon with problems that require a manual post-processing.

# Inhaltsverzeichnis

<b>Danksagung</b>	<b>i</b>
<b>Eidesstattliche Erklärung</b>	<b>ii</b>
<b>Kurzfassung</b>	<b>iii</b>
<b>Abstract</b>	<b>v</b>
<b>Inhaltsverzeichnis</b>	<b>vi</b>
<b>1 Einleitung</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Zielsetzung und Relevanz . . . . .	1
1.3 Fragestellung . . . . .	2
1.4 Abgrenzung der Arbeit . . . . .	2
1.5 Vorgehen . . . . .	2
1.6 Gliederung der Arbeit . . . . .	4
<b>2 Liniennetzpläne</b>	<b>5</b>
2.1 Darstellung der Routen . . . . .	5
2.1.1 Map Styles . . . . .	5
2.1.2 Topographische Darstellung . . . . .	7
2.1.3 Halbschematische Darstellung . . . . .	8
2.1.4 Schematische Darstellung . . . . .	9
2.2 Graphische Elemente schematischer Karten . . . . .	11
2.2.1 Äussere Kartenelemente . . . . .	12
2.2.2 Liniensignaturen . . . . .	13
2.2.3 Punktsignaturen . . . . .	13
2.2.4 Flächensignaturen . . . . .	14
2.2.5 Beschriftung . . . . .	15
2.3 Design von Haltestellen . . . . .	16
2.4 Vor der Erstellung eines Liniennetzplanes . . . . .	17
2.4.1 Anforderungen bestimmen . . . . .	17
2.4.2 Beispiel für benötigte Daten . . . . .	18

2.5 Zusammenfassung . . . . .	19
<b>3 Manuelle Erstellung eines schematischen Liniennetzplanes</b>	<b>20</b>
3.1 Existierende Werkzeuge . . . . .	20
3.1.1 Software . . . . .	20
3.1.2 Metro Map Layout Regeln . . . . .	21
3.2 Vorgehensweise . . . . .	23
3.2.1 Ausgangsdaten . . . . .	23
3.2.2 Ansätze für die manuelle Erstellung . . . . .	24
3.3 Zusammenfassung . . . . .	28
<b>4 Vorhandene Ansätze und Möglichkeiten der Automatisierung</b>	<b>29</b>
4.1 Generierung von Metro Maps . . . . .	29
4.1.1 Spring Embedders . . . . .	29
4.1.2 Hill Climbing Algorithmus . . . . .	30
4.1.3 Mixed-Integer Linear Programming . . . . .	31
4.1.4 Focus + Context Metro Maps . . . . .	32
4.1.5 Weitere Algorithmen und Methoden . . . . .	32
4.2 Interaktives Bearbeiten von Metro Maps . . . . .	34
4.3 Automatisierte Bearbeitung von graphischen Elemente . . . . .	34
4.3.1 Beschriftung . . . . .	34
4.3.2 Darstellung der Linienführung . . . . .	35
4.3.3 Darstellung der Haltestellen und andere Verkehrsknoten . . . . .	39
4.3.4 Landmarken . . . . .	40
4.3.5 Spezielle Beschriftungen . . . . .	40
4.4 Zusammenfassung und Relevanz der Ansätze . . . . .	42
<b>5 Automatisierung bei der Darstellung der Linienführung</b>	<b>44</b>
5.1 Konflikte ermitteln . . . . .	44
5.1.1 Konflikte automatisch bestimmen . . . . .	44
5.2 Ansätze zur automatisierten Verdrängung . . . . .	49
5.2.1 Ansätze zur Verdrängung in der Kartographie . . . . .	49
5.2.2 Verdrängung in einem oktilinearen Layout . . . . .	50
5.3 Verdrängungsvektoren ermitteln . . . . .	53
5.4 Platz schaffen mit einer Skalierung . . . . .	55
5.4.1 Vorgehen . . . . .	55
5.4.2 Teilgraphen skalieren . . . . .	57
5.4.3 Berücksichtigung weiterer Kartenelemente . . . . .	58
5.5 Sequentieller Verdrängungsalgorithmus . . . . .	60
5.5.1 Vorgehen . . . . .	60



5.5.2 Berücksichtigung weiterer Kartenelemente . . . . .	68
<b>6 Implementierung</b>	<b>69</b>
6.1 Konfliktzonen ermitteln . . . . .	69
6.2 Skalierung . . . . .	71
6.2.1 Optimierungen . . . . .	72
6.3 Verdrängungsalgorithmus . . . . .	72
6.3.1 Ermittlung des Verdrängungsvektors . . . . .	73
6.3.2 Ermittlung der Verdrängungsgerade . . . . .	74
6.3.3 Verdrängungsphase . . . . .	75
6.3.4 Korrekturphase . . . . .	75
6.4 Optimierungen . . . . .	76
6.4.1 Bearbeitungsreihenfolge der Konflikte . . . . .	76
6.4.2 Oktilineare Konflikte . . . . .	77
6.5 Visualisierung . . . . .	79
6.6 Spezielle Probleme bei der Implementierung . . . . .	79
6.6.1 Numerische Präzision der Koordinaten . . . . .	79
6.6.2 Wiederholbarkeit . . . . .	79
6.6.3 Konflikte zwischen Knoten und diagonalen Kanten . . . . .	80
6.7 Resultate . . . . .	80
6.7.1 Beispiel 1 . . . . .	81
6.7.2 Beispiel 2 . . . . .	81
6.7.3 Beispiel 3 (S-Bahnstreckennetz der Stadt Zürich) . . . . .	83
<b>7 Diskussion und Ausblick</b>	<b>85</b>
7.1 Beantwortung der Fragestellungen . . . . .	86
7.1.1 Fragestellung 1 . . . . .	86
7.1.2 Fragestellung 2 . . . . .	86
7.1.3 Fragestellung 3 . . . . .	87
7.2 Schlussfolgerung . . . . .	88
7.3 Ausblick . . . . .	88
<b>Literatur</b>	<b>90</b>
<b>Tabellenverzeichnis</b>	<b>95</b>
<b>Abbildungsverzeichnis</b>	<b>96</b>
<b>Glossar</b>	<b>98</b>
<b>Akronyme</b>	<b>104</b>
<b>Notationen</b>	<b>105</b>

<b>Anhang</b>	<b>107</b>
<b>A Metro Map Layout Regeln</b>	<b>108</b>
<b>B Testdaten</b>	<b>109</b>
B.1 Beispiel 1 . . . . .	.109
B.1.1 Haltestellen . . . . .	.109
B.1.2 Streckensegmente . . . . .	.109
B.2 Beispiel 2 . . . . .	.110
B.2.1 Haltestellen . . . . .	.110
B.2.2 Streckensegmente . . . . .	.111
B.3 Beispiel 3 (S-Bahnstreckennetz der Stadt Zürich) . . . . .	.112
B.3.1 Haltestellen . . . . .	.112
B.3.2 Streckensegmente . . . . .	.113
<b>C Klassenstruktur</b>	<b>116</b>
<b>D Source Code</b>	<b>119</b>
D.1 Evaluation der Konflikte . . . . .	.119
D.1.1 Package <code>ch.geomo.tramaps.conflict.*</code> . . . . .	.119
D.2 Verdrängungsalgorithmus . . . . .	.133
D.2.1 Package <code>ch.geomo.tramaps.map.displacement.*</code> . . . . .	.133
D.2.2 Package <code>ch.geomo.tramaps.graph.layout</code> . . . . .	.144
D.3 Skalierung . . . . .	.148
D.3.1 Package <code>ch.geomo.tramaps.map.displacement.scale</code> . . . . .	.148
D.4 Weitere Core-Klassen . . . . .	.150
D.4.1 Package <code>ch.geomo.tramaps</code> . . . . .	.150
D.4.2 Package <code>ch.geomo.tramaps.map.*</code> . . . . .	.151
D.4.3 Package <code>ch.geomo.tramaps.conflict.buffer</code> . . . . .	.159
D.4.4 Package <code>ch.geomo.tramaps.graph.*</code> . . . . .	.163
D.5 Beispiele . . . . .	.178
D.5.1 Package <code>ch.geomo.tramaps.example</code> . . . . .	.178
D.6 Util-Klassen . . . . .	.183
D.6.1 Package <code>ch.geomo.util.collection.*</code> . . . . .	.183
D.6.2 Package <code>ch.geomo.util.geom.*</code> . . . . .	.199
D.6.3 Package <code>ch.geomo.util.logging</code> . . . . .	.207
D.6.4 Package <code>ch.geomo.util.math</code> . . . . .	.209

# 1 | Einleitung

## 1.1 Motivation

Seit vielen Jahren ist das Interesse des Autors an der Lösung kartographischer Problemstellungen gross. Mit der vorliegenden Arbeit geht er einem spezifischen Problem der Kartendarstellung nach. Sein besonderes Interesse gilt dabei der computergenerierten graphischen Ausgabe schematisierter Karten, wobei er eine gewisse Ästhetik anstrebt. Ende 2006 legte der Autor eine Diplomarbeit zum Thema Wanderroutenplaner vor. Schon zu dieser Zeit beschäftigte er sich mit der automatischen Generierung von Metro Maps. Da aber zu dieser Zeit noch kaum Ansätze zur Lösung dieser Aufgabe vorhanden waren, sprengte das Vorhaben den Rahmen einer Diplomarbeit. Seit dieser Zeit hat die Technik viele Fortschritte gemacht und der Autor ergriff die Gelegenheit mit der vorliegenden Arbeit sein vormaliges Interessengebiet erneut aufzugreifen.

## 1.2 Zielsetzung und Relevanz

Schematische Karten im öffentlichen Verkehr (z.B. Metro Maps) können mit Hilfe verschiedener Algorithmen automatisch generiert werden. Allerdings ist immer eine manuelle graphische Nachbearbeitung erforderlich, weshalb im Regelfall auf die automatisierte Generierung von Liniennetzplänen verzichtet wird.

Die vorliegende Arbeit möchte Vorschläge aufzeigen, wie die manuelle Bearbeitung der Daten durch automatisierte Prozesse effizienter gestaltet werden kann. Im Zuge dieser Automatisierung kann der Einbezug bestehender Algorithmen zu dieser Effizienz beitragen. Die Entwicklung weiterer Algorithmen durch den Autor soll die Aufgabe der manuellen graphischen Nachbearbeitung, die insbesondere beim Hinzufügen oder Anpassen von Routeninformationen immer komplexer wird, vereinfachen. Dadurch würde sich der personelle und zeitliche Aufwand für die graphische Nachbearbeitung minimieren.

## 1.3 Fragestellung

Die vorliegende Arbeit beschäftigt sich mit der Automatisierung von Teilaufgaben der manuellen graphischen Nachbearbeitung von Liniennetzplänen. Folgende Fragestellungen sollen in der Erarbeitung beantwortet werden:

- Welche graphischen Anforderungen an einen Liniennetzplan existieren?
- Wie kann die graphische Nachbearbeitung eines Liniennetzplans unterstützt werden und welche Algorithmen und Ansätze für die automatische Erstellung von Liniennetzplänen existieren?
- Wie kann mit einem teilautomatisierten Verfahren der Platzbedarf für die Darstellung von Routen berücksichtigt werden?

## 1.4 Abgrenzung der Arbeit

Der Autor präsentiert Lösungswege wie mit einem automatisierten Verfahren Routeninformationen berücksichtigt werden können. Die vorliegende Arbeit zeigt auf, wie der Platzbedarf für Routeninformationen (Linien-Signaturen) und Haltestellen-Signaturen ermittelt und berücksichtigt werden kann. Mit dem selben Verfahren können theoretisch auch andere Kartenelemente verarbeitet werden. Da jeder Liniennetzplan anderen Anforderungen genügen soll und auch die Aufgabenstellungen hinsichtlich der Kartenelemente unterschiedlich sein können, beschränkt sich die vorliegende Arbeit darauf einen Ansatz zu skizzieren, wie im Layout eines Liniennetzplans die notwendigen Routeninformationen und Haltestellen-Signaturen automatisch eingepflegt werden können.

## 1.5 Vorgehen

In Annäherung an das Thema der vorliegenden Arbeit sichtet der Autor verschiedene Varianten von Liniennetzplänen wie sie in der Fachliteratur vorgestellt werden. Dabei werden sowohl kartographische Anforderungen wie auch spezielle Layout-Anforderungen an Liniennetzpläne deutlich. Als relevant für das Thema der Arbeit erweisen sich insbesondere die Layout-Regeln für Metro Maps, welche auf den Prinzipien der Londoner Tube Map aufbauen.

Auf dieser Grundlage beschäftigt sich der Autor mit den Problemen der manuellen Erstellung eines schematischen Liniennetzplans. Dabei wird der Bedarf an Vereinfachung durch automatisierte Prozesse deutlich.

Mit einer weiteren Literaturrecherche kommt der Autor der vorliegenden Arbeit zum Ergebnis, dass bereits eine Anzahl von Ansätzen zur automatisierten Erstellung von Liniennetzplänen vorhanden ist. Die Autoren dieser Ansätze beschreiten unterschiedliche Wege, um Metro Maps zu generieren. Die Ergebnisse sind von unterschiedlicher graphischer Qualität. Einzelne dieser Ansätze beurteilt der Autor dieser Arbeit als durchaus nützlich, um einen schematisierten Liniennetzplan zu erstellen. Die so generierten Liniennetzpläne bilden ein Streckennetz ab, berücksichtigen jedoch keine detaillierten Routeninformationen und auch keine anderen Kartenelemente wie zum Beispiel Landmarken oder Points of Interest. Einzelne Ansätze beschäftigen sich mit der Automatisierung von weiteren Arbeitsschritten, zum Beispiel mit der Darstellung der Linienführung (Überkreuzungen, Farbwahl der Routen).

Ausgehend von einem schematisierten Streckennetz dessen Erstellung in der Literatur beschrieben ist beschäftigt sich der Autor mit den Problemen der manuellen Nachbearbeitung, die entstehen, wenn weitere Informationen hinzugefügt werden sollen. Dabei bedingt der Platzbedarf dieser zusätzlichen Informationen eine Verzerrung des ursprünglichen schematischen Layouts, welche iterativ korrigiert werden muss. Der Autor entwickelt ein Verfahren, wie der Platzbedarf der Routeninformationen und Haltestellen-Signaturen ermittelt werden kann. Um den benötigten Platz zu schaffen, müssen benachbarte Elemente verschoben (verdrängt) werden.

Angewendet auf Liniennetzpläne erzeugt der Vorgang der Verdrängung vor allem mit dem Kriterium der Oktilinearität Konflikte, welche der Autor aufzeigt und mit Hilfe der von ihm entwickelten Methoden korrigiert. Anhand von Beispielen demonstriert der Autor zwei Korrekturverfahren, wobei eines sich des Verfahrens der Skalierung bedient und das andere Knoten (Haltestellen) iterativ verschiebt, bis der Konflikt gelöst ist. Das zweite Verfahren erfordert unter Umständen die Wiederherstellung der Oktilinearität einzelner Kanten durch weitere Verschiebeprozesse adjazenter Knoten.

Die auf den oben aufgestellten Überlegung basierenden Algorithmen werden implementiert und die Ergebnisse ausgewertet. Anhand der Resultate werden Optimierungen vorgenommen, die zur Verbesserung der Ergebnisse beitragen.

## 1.6 Gliederung der Arbeit

Im Kapitel 1 werden die Fragestellung, die Eingrenzungen und das Vorgehen der vorliegenden Arbeit vorgestellt. Nach einer allgemeinen Einführung in die Natur von Liniennetzplänen im Kapitel 2, werden im Kapitel 3 die Prinzipien und das Vorgehen für die manuelle Erstellung solcher Liniennetzpläne erörtert. Im Hinblick auf eine teilautomatisierte Erstellung von Liniennetzplänen werden im Kapitel 4 verschiedene Ansätze präsentiert. Mit den vorhandenen Ansätzen ist es möglich mit Hilfe eines gegebenen Datensatzes ein oktilineares Liniennetz zu generieren. Auf dieser Grundlage werden im Kapitel 5 Probleme mit dem Platzbedarf diskutiert, die beim Einfügen von Routeninformationen und Haltestellen-Signaturen in den oktilinearen Liniennetzplan entstehen, und es werden Ansätze präsentiert, wie diese Probleme durch eine Teilautomatisierung gelöst werden können. Im Kapitel 6 werden die Algorithmen, welche im Kapitel 5 skizziert werden, anhand von Beispielen getestet. Bei der Implementierung zeigen sich Probleme, die zum Teil durch Optimierungen der Algorithmen behoben werden können. Eine Diskussion der Ergebnisse sowie ein Ausblick auf die mögliche Weiterbearbeitung von Problemen, die in der vorliegenden Arbeit nicht gelöst werden können, schliessen sich im Kapitel 7 an.

## 2 | Liniennetzpläne

Die Benutzung eines öffentlichen Transportsystems in einer unbekanntenen Stadt kann zu einer Herausforderung werden. Wie kommt man von A nach B? Wo muss man umsteigen? Nach wie vielen Haltestellen muss man wieder aussteigen? Normale Strassenkarten helfen womöglich nicht bei der Beantwortung dieser Fragen, wenn sie nicht auch Informationen über Haltestellen und Verkehrslinien enthalten. Selbst wenn solche Angaben beispielsweise in einem Stadtplan vorhanden sind, können sie meist nicht übersichtlich genug dargestellt werden, weil ihre Wahrnehmung durch die Vielzahl der anderen Kartendetails überlagert wird. Das Netzwerk eines öffentlichen Transportsystems ist weitaus informativer, wenn irrelevante Informationen weggelassen und die übrigen Kartenelemente durch semantische und geometrische Generalisierung vereinfacht werden.

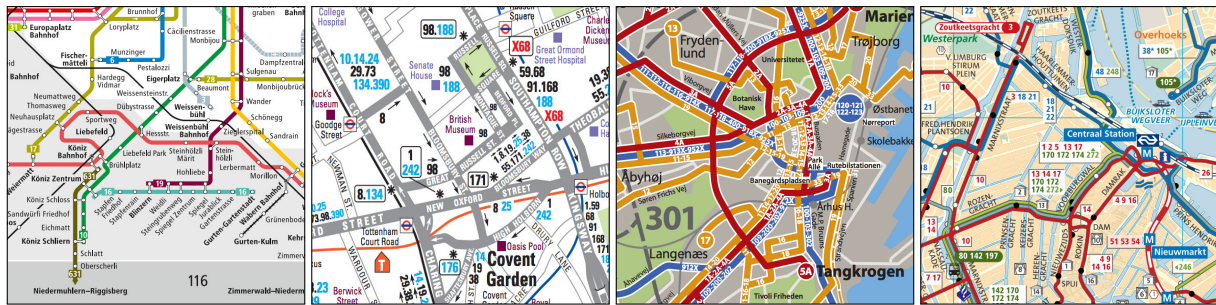
Liniennetzpläne existieren in vielen Städten rund um den Globus, um den Passagieren die Orientierung im Verkehrsnetzwerk zu erleichtern. Um hilfreich zu sein, müssen Liniennetzpläne so verständlich und leserlich wie möglich gestaltet werden. Während ein Mensch nur eine begrenzte Menge an Informationen auf einmal verarbeiten kann, verfügt er doch über eine ausgezeichnete Fähigkeit Muster (Pattern) zu erkennen (vgl. Tversky und P. U. Lee 1999, Seite 2). Wenn man dies berücksichtigt, macht es Sinn irrelevante Kartenelemente zu entfernen und ein leicht merkbare und vereinfachtes Layout des Netzwerkes darzustellen, um dem Passagier das Lesen der Karte zu vereinfachen und seine Fragen zu Transportmöglichkeiten zu beantworten.

### 2.1 Darstellung der Routen

Dieses Kapitel fokussiert sich auf die Darstellung der Routen und dessen Haltestellen eines (an der Haltestelle im Grossformat aufgehängten) Liniennetzplans. Auf weitere (wichtige) Aspekte, die ebenfalls bei der Darstellung berücksichtigt werden müssen, z.B. Kartengrösse, Verwendungszweck und Zielpublikum, wird in der vorliegende Arbeit nicht eingegangen. Für eine umfassendere Diskussion der Darstellung von Liniennetzpläne empfiehlt der Autor dieser Arbeit deshalb die Artikel von Morrison (1996), Avelar und Hurni (2006), Allard (2009) und Bain (2010), sowie den Leitfaden von Cain u. a. (2008).

#### 2.1.1 Map Styles

Es gibt nicht nur eine Art eine Route darzustellen. So hat Morrison (1996) bei einer Analyse von Liniennetzpläne westeuropäischer Städte mehrere Stile der kartographischen Repräsentation einer Route identifiziert. Die von ihm ermittelten Stile zur Darstellung von



(a) Bern (CH), 2016      (b) London (GB), 2016      (c) Århus (DK), 2015      (d) Amsterdam (NL), 2016

**Abbildung 2.1:** Beispiele für den French, Classic, Scandinavian und Dutch Style (von links nach Rechts: ©Libero Tarifverbund, ©Transport for London - Bus map reproduced by kind permission of Transport for London, ©Midttrafik, ©Gemeentelijk Vervoerbedrijf)

Routen unterscheiden sich insbesondere durch die Wahl der Liniensignatur einer Route, deren Beschriftung und das allfällige Typifizieren von Routen. Die folgende Zusammenfassung der Darstellungsstile beruht auf den von Morrison (1996, Seite 93 ff.) identifizierten Stilen.

### French Style

Der häufigste verwendete Stil identifiziert jede Route mit einer Farbe. Beschriftet wird die Route jeweils an den Endstationen. Dieser Stil wird z.B. im Liniennetzplan von Bern (vgl. Abbildung 2.1a) verwendet.

### Classic Style

Jeder Streckenabschnitt wird nur mit einer Linie dargestellt. Dies hat den Nachteil, dass aus der Liniensignatur weder die Route noch das Verkehrsmittel herausgelesen werden können. Einzig die entlang der Linie platzierte Beschriftung gibt Auskunft zu den Routen auf dem jeweiligen Streckenabschnitt. So macht beispielsweise der Liniennetzplan des Busstreckennetzes von London (vgl. Abbildung 2.1b) von diesem Stil Gebrauch.

### Scandinavian Style

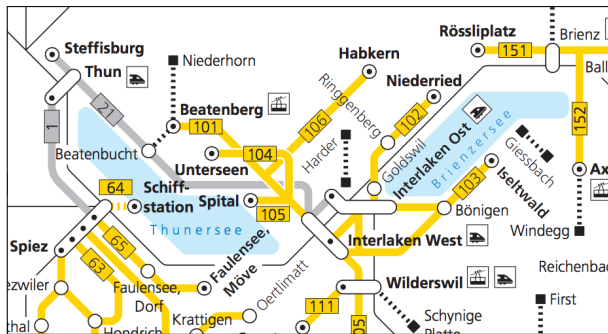
Dieser Stil ist dem Classic Style sehr ähnlich. Jedoch wird das Transportnetzwerk zusätzlich in zwei oder drei Gruppen eingeteilt und jeweils mit einer anderen Farbe versehen. Das Verkehrsnetz von Århus (Abbildung 2.1c) wird z.B. in drei Gruppen aufgeteilt: A-Busse (rot), Stadtbusse (orange) und Regionalbusse (blau).

### Dutch Style

Der Dutch Style basiert ebenfalls auf dem Classic Style. Im Unterschied zum Scandinavian Style wird die Darstellung der Routen aber nicht nach Transportnetzwerken unterteilt, sondern es wird nach Verkehrsmittel typifiziert. Dabei wird ein grosser Nachteil des Classic



## 2.1. Darstellung der Routen



**Abbildung 2.2:** Ausschnitt aus dem Postauto-Liniennetzplan des Berner Oberlandes (©PostAuto Region Bern)

Styles aufgehoben, da im Dutch Style zwischen den Verkehrsmitteln unterschieden werden kann und jedes Transportmittel eine eigene Liniensignatur hat. Beispielsweise wird im Liniennetzplan von Amsterdam (vgl. Abbildung 2.1d) die übliche kartographische Darstellung von Bahnstrecken verwendet.

Weitere Styles kann Morrison nicht identifizieren, doch können regionale und länder-spezifische Charakteristiken existieren (vgl. Morrison 1996, Seite 95). Der Autor dieser Arbeit hat festgestellt, dass z.B. Liniennetzpläne in der Deutschschweiz unabhängig vom Transportmittel vorwiegend den French Style verwenden, meistens kombiniert mit einem anderen Style für die Darstellung von sekundären Transportmitteln. Des Weiteren sind sie in der Regel schematisiert. Reguläre Busrouten (als sekundäres Transportmittel) werden häufig mit dem Classic Style visualisiert, während Postautorouten jeweils einheitlich mit Gelb (vgl. Abbildung 2.2), also im Scandinavian Style, dargestellt werden. Des Weiteren werden Anschlussrouten ausserhalb des jeweiligen Verkehrsverbunds in manchen Liniennetzplänen mit grauer Farbe und Schiffsrouten mit einer Bézierkurve dargestellt (vgl. Zürichsee in Abbildung 2.8).

Grundsätzlich sollte der French Style bevorzugt werden (vgl. Morrison 1996, Seite 96). Morrison empfiehlt, die Wahl des Darstellungsstil von der Anzahl verschiedener Verkehrsmittel abhängig zu machen. Existiert beispielsweise mehr als ein Verkehrsmittel, so macht es Sinn den French Style beim primären bzw. dominierenden Transportmittel anzuwenden und für weitere Verkehrsmittel den Classic Style zu verwenden (vgl. Morrison 1996, Seite 96). Dies ist gut im Liniennetzplan des Zürcher Verkehrsverbundes zu erkennen, wo die Busrouten im Classic Style dargestellt werden und für die S-Bahnlinien der French Style verwendet wird.

### 2.1.2 Topographische Darstellung

Liniennetzpläne mit einem topographischen Hintergrund und einer genauen Visualisierung der Streckenführung werden häufig für reine Bussysteme verwendet. Insbesondere im nordamerikanischen Raum, wo Städte normalerweise im Schachbrettmuster aufgebaut sind, findet sich diese Darstellungsweise, so zum Beispiel auch in Montréal (vgl. Abbildung



**Abbildung 2.3:** Ausschnitt aus dem Liniennetzplan von Montréal, Québec, 2016 (©Archives of the Société de transport de Montréal)



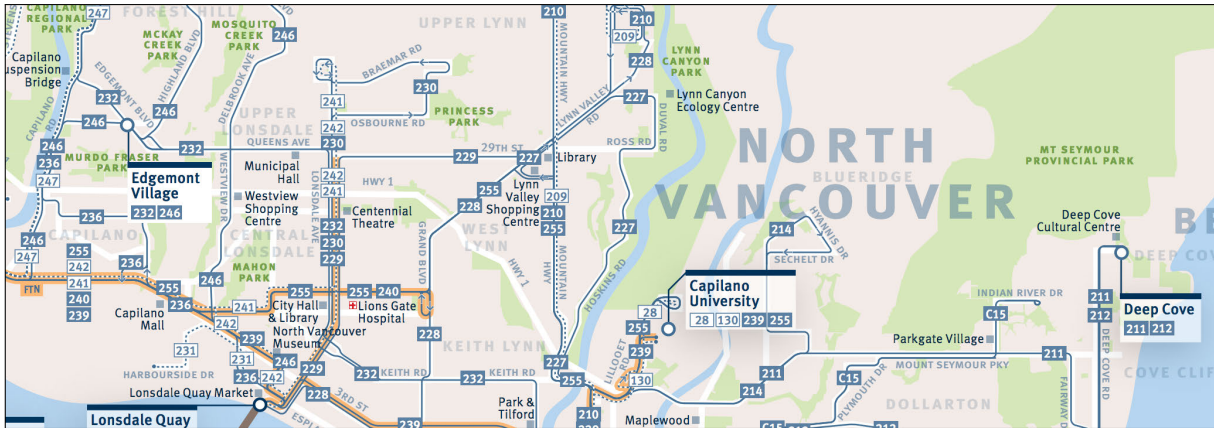
**Abbildung 2.4:** Ausschnitt aus dem Liniennetzplan von Amsterdam, 2016 (©Gemeentelijk Vervoerbedrijf)

2.3). Diese Art von Liniennetzplänen (Overlay Maps) bietet eine hohe Flexibilität und ist insbesondere durch den Einsatz von geographischen Informationssystemen (GIS) attraktiv (vgl. Cain u. a. 2008, Seite 5).

Die topographische Darstellung erlaubt einen sehr detailreichen Karteninhalt (vgl. Abbildung 2.4). Allerdings wird dabei dem Kartenleser die Sicht auf die wesentlichen Informationen zur Orientierung innerhalb des Transportnetzwerkes erschwert. Des Weiteren wird auch der Einsatz des French Style behindert, weil die Grundkarte die Wahl der Routenfarben aufgrund des fehlenden Kontrasts einschränkt (vgl. Morrison 1996, Seite 103). Nach Morrison ist deshalb ein einfarbiger Hintergrund zu bevorzugen und er empfiehlt für den Hintergrund die Farbe weiss oder grau, um einen möglichst grossen Kontrast zur Vordergrundfarbe zu erzielen.

### 2.1.3 Halbschematische Darstellung

Wenn ein topographischer Liniennetzplan weiter vereinfacht wird, entsteht eine Darstellung, die nach wie vor viel der ursprünglichen Topographie und Kartendetails abbildet, aber die Karte etwas verzerrt und die Routenführung teilweise schematisiert. Die Verzerrung entsteht unter anderem dadurch, dass die Strassenbreite unter Umständen angepasst werden muss, um alle Routen darzustellen. Gleichzeitig müssen andere Kartenele-



**Abbildung 2.5:** Ausschnitt aus dem Liniennetzplan von North Vancouver, British Columbia, 2016 (©Translink)

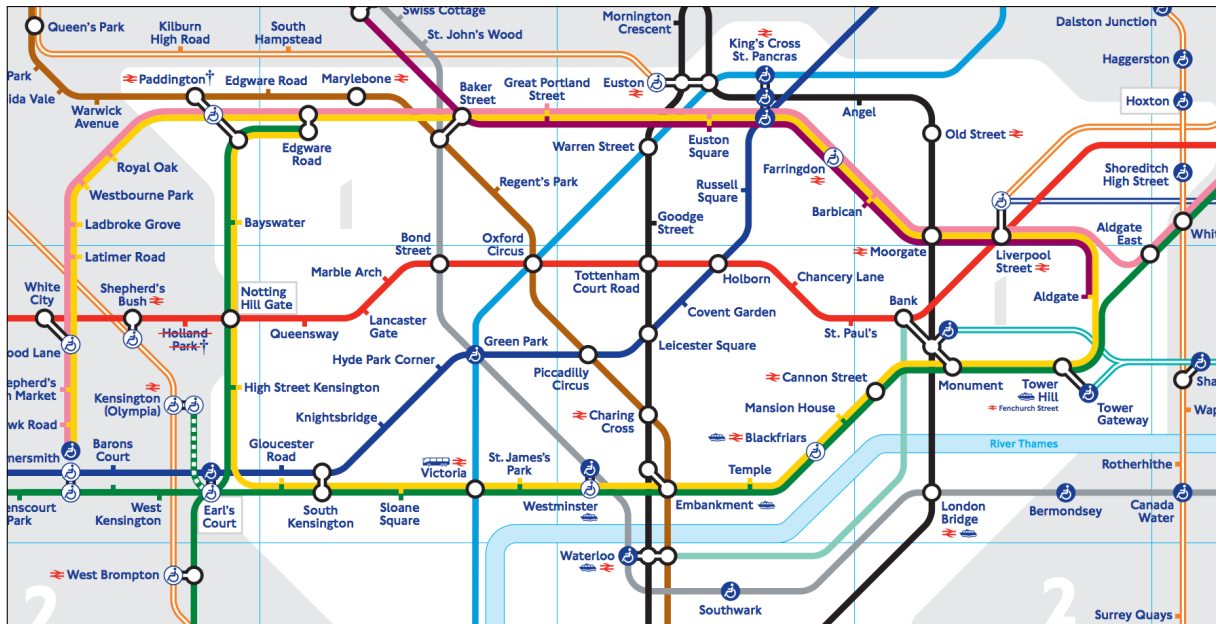
mente mit weniger Raum auskommen. Ein gutes Beispiel für eine solche Darstellung ist der Liniennetzplan Rhein-Ruhr (vgl. Morrison 1996, Seite 99f.). Cain u. a. (2008, Seite 5) nennt eine solche Darstellung halbschematisch, Allard (2009, Seite 84) spricht in diesem Zusammenhang von Hybrid Maps. Sie findet vor allem Verwendung in Regionen, die eine relativ regelmäßige Struktur aufweisen, wie beispielsweise das Schachbrettmuster amerikanischer Städte (vgl. Abbildung 2.5 und Cain u. a. 2008, Seite 5).

### 2.1.4 Schematische Darstellung

Eine schematische Karte abstrahiert die reale Gegebenheit und präsentiert die Routen des Transportmittels mit vereinfachten Linien. Diese Linien enthalten wenige bis gar keine Knicke und ihre Liniensegmente werden in der Regel oktilinear (vgl. Morrison 1996, Seite 97) gezeichnet. Für einen Passagier, der von A nach B kommen möchte, ist es im Normalfall irrelevant, ob eine Strecke kurvenreich ist. Das Layout wirkt aufgeräumter und erlaubt dadurch eine schnellere Orientierung auf der Karte. Des Weiteren kann der Kartenleser den Linien besser folgen.

Die wohl bekannteste schematische Darstellung eines Liniennetzplans ist Henry Becks Tube Map (vgl. Abbildung 2.6). Beck hatte die Idee die Leserlichkeit der Netzwerkkarte zu verbessern, indem er Linien und Winkel vereinfachte und verschiedene Farben für jede Route benutzte. Sein erster Entwurf stammt aus dem Jahr 1931. Zwei Jahre später wurde seine Karte publiziert (vgl. Garland 2008, Seite 17 bis 19). Indem Beck die Topologie des Netzwerks beibehält, berücksichtigt er ein wichtiges Prinzip der kartographischen Darstellung, welches unter dem Begriff Topological Pattern bekannt ist (vgl. Avelar und Hurni 2006, Seite 219). Darüber hinaus führt er das oktilineare Layout (Linienwinkel beschränkt auf ein Mehrfaches von  $45^\circ$ ) und gleichmäßige Abstände zwischen den Haltestellen ein. Letzteres erreicht Beck, indem er nur die relativen Positionen der Haltestellen berücksichtigt und eine Verzerrung des Massstabs erlaubt. Peripherien mit wenigen Routen werden

## 2.1. Darstellung der Routen



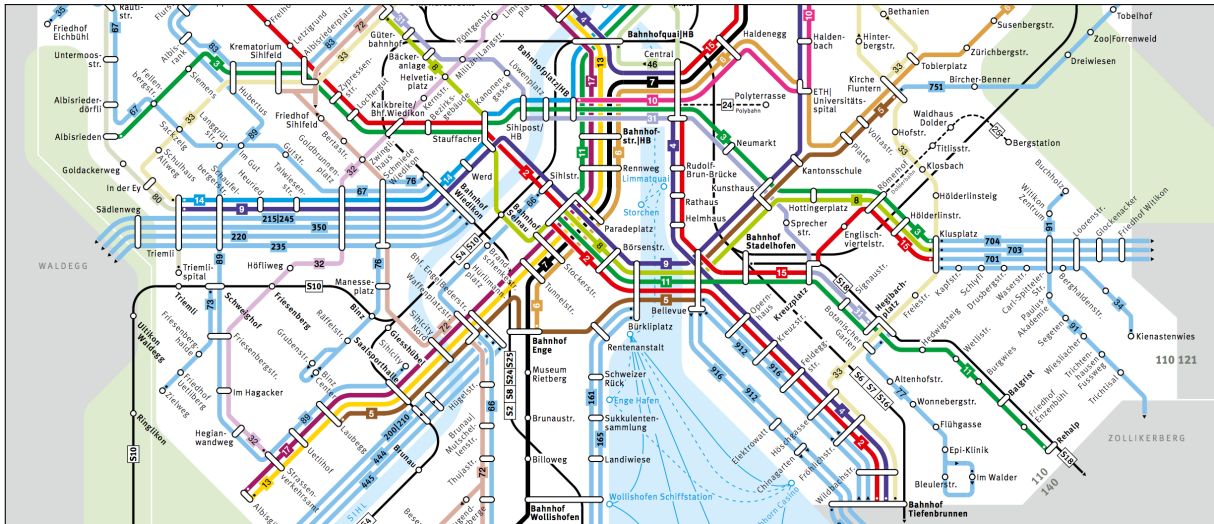
**Abbildung 2.6:** Ausschnitt aus dem Liniennetzplan von London, 2016 (©Transport for London - Tube map reproduced by kind permission of Transport for London)

dadurch kleiner dargestellt und das Zentrum, normalerweise mit vielen Routen, dementsprechend grösser. Dies ermöglichte Beck alle Peripherien darzustellen ohne auf Details im Zentrum zu verzichten (vgl. Jenny 2006, Seite 15<sup>1</sup>).

Neben Haltestellen, Streckensegmenten, Routeninformationen und den Metro Map Layout Regeln (vgl. Kapitel 3.1.2) können weitere (geo-)graphische Details zur Orientierung herangezogen werden. Es handelt sich in der Regel um vereinfachte Flussläufe, Seen oder Küstenlinien. Zuweilen werden Details von besonderem Interesse (Points of Interest) wie beispielsweise ein Flughafen als Symbol hinzugefügt.

<sup>1</sup> "Enlarging the centre in relation to the outlying regions allowed him to include all tentacles of the ramified network, while still clearly depicting the details of the central area." (Jenny 2006, Seite 15)

## 2.2. Graphische Elemente schematischer Karten



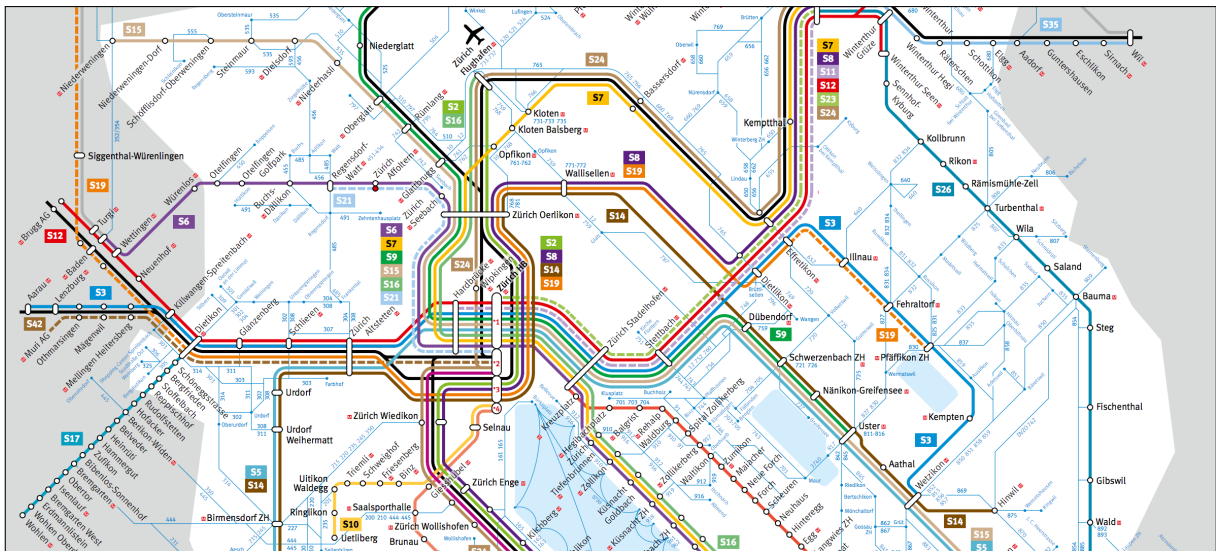
**Abbildung 2.7:** Ausschnitt aus dem Liniennetzplan der Stadt Zürich, 2015 (©ZVV/VBZ)

Diese Darstellungsart ist jedoch nicht für alle Verkehrsmittel sinnvoll, wie Morrison (1996, Seite 97) festhält. Schematische Karten werden u.a. bei Streckennetzen von Untergrundbahnen, Eisenbahnen oder Strassenbahnen eingesetzt. Bei einem reinen Busbetrieb sollte der Einsatz einer topologischen Darstellung bevorzugt werden. Da der Passagier eines Busses den realen Strassenverlauf wahrnimmt und dadurch auch seine mentale Karte geformt wird, kann der Passagier unter Umständen die schematische Karte nicht so leicht der Realität zuordnen (vgl. Morrison 1996, Seite 97). Allerdings werden schematische Karten vor allem in der Schweiz auch für reine Bus-Liniennetze und kombinierte Netzwerke mit verschiedenen Verkehrsmitteln verwendet<sup>2</sup>, beispielsweise in den Städten Zürich (vgl. Abbildung 2.7) und Bern.

## 2.2 Graphische Elemente schematischer Karten

Liniennetzpläne enthalten neben Haltestellen und Routen weitere graphische Elemente, z.B. stilisierte Flussverläufe oder Points of Interest. Die heutige Weiterentwicklung von Becks Tube Map (vgl. Abbildung 2.6) enthält neben dem Flussverlauf der Themse auch die Darstellung der Ticketzonen. Der Liniennetzplan des Zürcher Verkehrsverbunds (vgl. Abbildung 2.8) visualisiert die drei grössten Seen des Kanton Zürichs, die Grenzen des Verbundgebiets und ein Symbol, das den Flughafen Zürich repräsentiert. Darüber hinaus sind im Zürcher Liniennetzplan neben den S-Bahnlinien auch Busverbindungen zwischen den Ortschaften und Schiffsverbindungen eingezeichnet.

<sup>2</sup> In einer Korrespondenz (vgl. Journal of Maps, 2008, Volume 4, Issue 1, Seite c2) mit Silvania Avelar schreibt Alistair Morrison: “[...] schematic maps may [also] be effective for a tram or bus network in a city such as Zürich where all the tram and bus stops have well-defined names which are displayed prominently on the stops themselves, or are announced reliably within the vehicle”.



**Abbildung 2.8:** Ausschnitt aus dem S-Bahn-Liniennetzplan des Kantons Zürich, 2015 (© Zürcher Verkehrsverbund und PostAuto Region Zürich)

José Allard analysiert in seiner Dissertation Liniennetzpläne und unterteilt die gefundenen graphischen Elemente in 5 Kategorien (vgl. Allard 2009, Seite 54):

- Äussere Kartenelemente (Design Elements)
- Liniensignaturen (Line Symbols)
- Punktsignaturen (Point Symbols)
- Flächensignaturen (Area Symbols)
- Beschriftung (Lettering)

### 2.2.1 Äussere Kartenelemente

Genauso wie bei einer Strassenkarte werden in einem Liniennetzplan weitergehende Informationen und Erläuterungen am Kartenrand zur Verfügung gestellt. Diese helfen dem Kartenleser beim Verstehen und Interpretieren des Karteninhaltes. Dazu gehören insbesondere Titel, Legende, Massstab<sup>3</sup>, Angabe der Gültigkeit und der Herausgeber.

Eine Karte bzw. ein Liniennetzplan wird manchmal auch mit einem unabhängigen Gitternetz unterteilt. So hat u.a. die Tube Map (vgl. Abbildung 2.6) ein solches Netz. Zusammen mit einem Index lassen sich so die Haltestellen in einem grossen Transportnetzwerk schneller finden. Das Gitternetz ist dabei komplett unabhängig von den geographischen Gegebenheiten eingezeichnet und kann daher auch bei schematischen Karten mit grosser Verzerrung verwendet werden.

<sup>3</sup> Falls die Verwendung eines Massstabes aufgrund der geometrischen Verzerrung bei schematischen Karten nicht möglich ist, könnte ein Hinweis wie z.B. "Nicht skalierbar." (vgl. Allard 2009, Seite 83) angegeben werden.

### 2.2.2 Liniensignaturen

Liniensignaturen in einem Liniennetzplan geben Auskunft über die Art der Verkehrsmittel in einem Streckennetz. Es kann zwischen den primären Verkehrsmitteln und allenfalls zusätzlichen Verkehrsmitteln unterschieden werden. Andere Arten von Liniensignaturen, die sich auf topographische Gegebenheiten beziehen, werden normalerweise weggelassen, z.B. Höhenlinien, kleine Flussverläufe oder Wanderwege.

#### **Primäres Verkehrsmittel**

Liniennetzpläne dienen in der Regel dazu die Streckenverläufe eines Transportmittel schematisch darzustellen. In manchen Netzplänen werden ausser dem primären Transportmittel weitere Verkehrsmittels berücksichtigt. Mit Hilfe der Darstellung der Liniensignatur wird das primäre Transportmittel wird deshalb in den Fokus gerückt, beispielsweise durch die Linienbreite und die Farbgebung.

#### **Zusätzliche Verkehrsmittel**

Die Anzahl der Transportmittel und die Darstellung der Liniensignaturen haben einen grossen Einfluss auf das Layout des Liniennetzplanes. Neben dem primären Verkehrsmittel können zusätzliche Verkehrsmittel auf einem Liniennetzplan dargestellt werden. Ein Beispiel zeigt der Zürcher Liniennetzplan, bei dem die S-Bahn das primäre Transportmittel ist (vgl. Abbildung 2.8). Der Plan verweist auf drei weitere Verkehrsmittel, nämlich Schnellzüge, Busse und Schiffe. Im Zürcher Liniennetzplan sind zusätzliche Transportmittel mit anderen Liniensignaturen dargestellt, wobei die Wahl der geringeren Linienbreite und die Farbgebung die zusätzlichen Verkehrsmittel in den Hintergrund rücken. Auch sind die Informationen über zusätzliche Transportmittel auf ein Minimum reduziert. So werden werden z.B. bei den regionalen Buslinien nur grössere Ortschaften, nicht aber Haltestellen dargestellt. Durch diese Art der Darstellung wird das primäre Verkehrsmittel deutlich hervorgehoben.

Ein anderes Beispiel für einen Liniennetzplan mit zusätzlichen Verkehrsmittel ist der Plan der Massachusetts Bay Transportation Authority, welche neben dem U-Bahn-Netz, die wichtigsten Busrouten und Fährverbindungen enthält (vgl. [http://www.mbta.com/schedules\\_and\\_maps/subway](http://www.mbta.com/schedules_and_maps/subway), 30. September 2016).

### 2.2.3 Punktsignaturen

In einem Liniennetzplan werden normalerweise Haltestellen (Stops, Umsteigehaltestellen) und Orte von besonderem Interesse (Points of Interest) mit einer Punktsignatur dargestellt, die durch unterschiedliche graphische Elemente repräsentiert werden kann.

### **Haltestellen**

Ohne die Darstellung von Haltestellen ist ein Liniennetzplan nicht brauchbar. Die Visualisierung ist daher essentiell und sollte gut erkennbar sein. Die Darstellung erfolgt in der Regel durch eine qualitative Punktsignatur mit einer einfachen Form, beispielsweise mit einem Kreis (vgl. Abbildung 2.7) oder einem Tick (kleine Ausbuchtung der Linie, vgl. Abbildung 2.6).

### **Umsteigehaltestellen**

Eine Ausnahme bilden Umsteigehaltestellen. Obwohl sie ebenfalls als Punktobjekt vorliegen, werden Umsteigebahnhöfe normalerweise mit komplexeren Formen (z.B. mit einem Ring oder einem Polygon) dargestellt (vgl. Allard 2009, Seite 88 f.).

### **Points of Interest**

Points of Interest können z.B. auf einen Flughafen oder eine markante Sehenswürdigkeit verweisen. Sie sollten in einem Liniennetzplan sparsam verwendet werden. Richtig angewendet können Points of Interest den Kartenleser unterstützen, indem wichtige, dem ortsansässigen Passagier bekannte Orientierungspunkte hervorgehoben werden. Die Aufmerksamkeit wird dadurch auf bekannte Regionen des Transportnetzwerkes gerichtet (vgl. Avelar 2008, Seite 138; zit. n. HGSD 2005).

## **2.2.4 Flächensignaturen**

Flächensignaturen werden in der Regel für Landmarken und die Begrenzung von Gebieten verwendet.

### **Landmarken**

In Liniennetzplänen verwendete Landmarken sind in der Regel grosse Flächen, beispielsweise Wasserflächen, wichtige Flüsse und Waldgebiete. Sie sind eine gute Orientierungshilfe für den Kartenleser. Die Geometrie einer Landmarke ist in einem schematischen Liniennetzplan normalerweise ebenfalls vereinfacht. Die Darstellung von Streckensegmenten entlang einer Landmarke zeichnet die Aussenlinie des Polygons und des Streckensegments üblicherweise parallel (vgl. Abbildung 2.8). Parallel verlaufende Routen beeinflussen deshalb die Form bzw. Generalisierung der Landmarke. Umgekehrt wird die Ausrichtung der Streckensegmente durch die Landmarke beeinflusst.

### **Grenzen**

Grenzgebiete sind in der Regel administrative Regionen oder Tarifzonen. Die Darstellung kann sowohl durch eine Linien- als auch durch eine Flächensignatur erfolgen. Aus Sicht



des Autors dieser Arbeit ist die Verwendung einer Flächengeometrie jedoch sinnvoller, da die Topologie bei der Herstellung des Liniennetzplans besser überprüft und berücksichtigt werden kann.

Die ursprüngliche Form einer administrativen Region sollte bei der Vereinfachung erhalten bleiben. Dies hilft der mentalen Karte des Kartenlesers. Die Form einer Tarifzone ist jedoch in der Regel nicht festgelegt.

### **2.2.5 Beschriftung**

Die Beschriftung (Labelling) von Kartenelementen ist ein wichtiger Aspekt einer Karte. Beschriftungen (Labels) benennen Kartenelemente, beispielsweise die Haltestellen, Routen und Landmarken, und ermöglichen so die Unterscheidbarkeit von Kartenelementen gleicher Signatur. Damit die Labels richtig kommunizieren können, sollten diese so dargestellt werden, dass sie dem Kartenelement leicht zugeordnet werden können (vgl. Freeman 2005, Seite 2) und bestimmten Regeln der Platzierung folgen (vgl. Wood 2000).

#### **Haltestellen**

Der Ausgangspunkt und das Ziel der Reise müssen zur erfolgreichen Navigation identifiziert werden können. Daher ist jede Haltestelle in einem Liniennetzplan beschriftet<sup>4</sup>. Damit das Lesen der Beschriftung der Haltestellen entlang einer Route so einfach wie möglich ist, werden die Labels einheitlich ausgerichtet. Im Idealfall sollten deshalb alle Labels zwischen mindestens zwei Umsteigestationen auf der gleichen Seite liegen (vgl. Ehinger 2009, Seite 5) und die gleiche Ausrichtung haben.

#### **Routen**

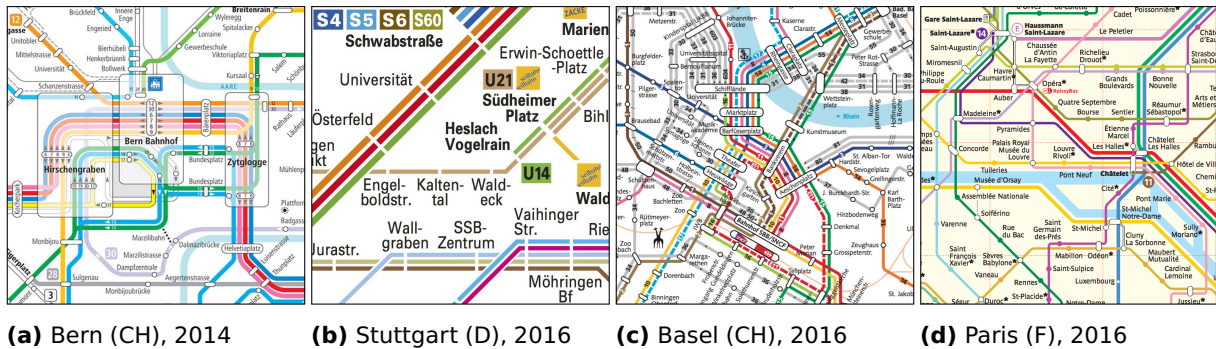
Liniensignaturen einer schematischen Karte verwenden in der Regel verschiedene Farben für verschiedene Routen (French Style, vgl. Kapitel 2.1.1). Es genügt allerdings nicht eine Route nur durch die Farbe und ohne Beschriftung darzustellen. Eine Unterscheidbarkeit ist nicht in jedem Fall gegeben, so kann ein Passagier mit einer Sehschwäche (z.B. Farbenblindheit) unter Umständen die Farbe Grün nicht erkennen.

#### **Points of Interest und Landmarken**

Points of Interest werden beschriftet, wenn das Kartensymbol (Punktsignatur) nicht selbsterklärend ist. Landmarken werden in der Regel beschriftet (vgl. Abbildung 2.6). Dem ortsunkundigen Kartenleser wird damit die Orientierung erleichtert. So wird beispielsweise

---

<sup>4</sup> Es gibt jedoch auch andere (experimentelle) Wege der Beschriftung. So verwendet eine inoffizielle Metro Map von Mexico City Symbole anstelle einer Beschriftung der Stationen (vgl. Archambault 2015; Allard 2009, Seite 71). Die offizielle Metro Map von Tokio verwendet neben einer Beschriftung zusätzlich eine Nummerierung der Haltestellen (vgl. Ovenden 2007, Seite 43).



**Abbildung 2.9:** Beispiele für das Design von Haltestellen (von links nach Rechts: ©Libero Tarifverbund, ©Verkehrs- und Tarifverbund Stuttgart GmbH, ©TNW Tarifverbund Nordwestschweiz, ©RATP (Paris Transport Authority) - All rights reserved)

die Ortschaft Pfäffikon SZ (am Zürichsee) weniger oft mit Pfäffikon ZH (am Pfäffikersee) verwechselt, falls dem Kartenleser bekannt ist, an welchem See die Ortschaft liegt. Es ist daher sinnvoll die Landmarken Zürichsee und Pfäffikersee zu beschriften.

## 2.3 Design von Haltestellen

Für die Darstellung der Haltestellen wird in dieser Arbeit zwischen (einfachen) Stops, Umsteigehaltestellen (Haltestellen mit Verzweigungen) und Endhaltestellen unterschieden. Stops verwenden normalerweise einfache Signaturen, beispielsweise Kreise (vgl. Abbildung 2.9a), langgezogene (abgerundete) Rechtecke (vgl. Abbildung 2.8), kurze Unterbrüche der Liniensignatur (vgl. Abbildung 2.9b) oder Ticks (vgl. Abbildung 2.6). Umsteigehaltestellen hingegen werden komplexere Signaturen dargestellt. Dabei handelt es sich häufig um Polygone (vgl. Abbildung 2.9c), die in der Regel in Grösse, Form und eventuell Farbe variieren.

Umsteigehaltestellen, welche etwas weiter auseinander liegen, werden häufig speziell dargestellt. Die Visualisierung kann z.B. mit Panels oder Brücken-Symbolen realisiert werden. In der Londoner Tube Map werden beispielsweise Brücken verwendet (vgl. Abbildung 2.6, z.B. Waterloo Station). Mit einem (transparentem) Panel kann eine grössere Fläche zusammengefasst werden (vgl. Abbildung 2.9a).

Endhaltestellen werden in der Regel ebenfalls gekennzeichnet. So wird im Liniennetzplan von Basel und Paris eine Endhaltestelle mit einem kleinen Punkt auf der Linie bzw. auf der Haltestelle visualisiert (vgl. Abbildung 2.9a und 2.9d). Vielfach werden Endhaltestellen im gleichen Design wie Umsteigehaltestellen dargestellt, während die Beschriftung hervorgehoben wird (vgl. Abbildung 2.9b sowie Allard 2009, Seite 89).

## **2.4 Vor der Erstellung eines Liniennetzplanes**

Im folgenden Abschnitt wird die Frage behandelt, welche Anforderungen für die Erstellung eines Liniennetzplanes erfüllt werden müssen und welche Daten benötigt werden. Es wird eine Reihe von Aspekten präsentiert, die bei der Wahl der richtigen Darstellung und des Karteninhaltes zu beachten sind. Des Weiteren werden die für die Erstellung benötigten Daten erläutert.

### **2.4.1 Anforderungen bestimmen**

Jede Karte hat einen spezifischen Verwendungszweck und soll mit dem dargestellten Karteninhalt dem Kartenleser räumliche Informationen zu einem bestimmten Thema vermitteln. Die Aufgaben eines Liniennetzplanes sind in erster Linie die Darstellung eines Transportnetzwerkes und dem Kartenleser die Routenplanung zu erleichtern. In die Visualisierung eines Liniennetzplanes fließen neben der Art des Transportnetzwerkes auch die lokalen und regionalen Gegebenheiten, mögliche Präferenzen des Kartenlesers und andere graphische Restriktionen ein.

#### **Art des Transportnetzwerkes**

Falls mehr als ein Transportmittel auf einem Liniennetzplan dargestellt werden soll, ist zu bestimmen, welches das dominierende Transportmittel ist. Eventuelle Routenüberschneidungen oder Routenvarianten sind ebenso zu berücksichtigen wie Distanzen zwischen Haltestellen (vgl. Avelar 2008, Seite 137). Je nach Transportmittel bieten sich verschiedene Arten der Darstellung an<sup>5</sup>.

#### **Lokale und regionale Gegebenheiten**

Morrison (1996, Seite 93 ff.) hat festgestellt, dass gewisse Map Styles in bestimmten Regionen dominieren. Der Autor dieser Arbeit ist daher der Meinung, dass bei der Wahl des Map Styles die regionale Dominanz eines Stils bedacht werden sollte.

Räumliche Auffälligkeiten sollten unbedingt im Layout des Streckennetz berücksichtigt werden. Der ortskundige Kartenleser kann so ein ihm bekanntes räumliches Muster erkennen, was die kognitive Karte des Lesers unterstützt (vgl. Tversky und P. U. Lee 1999, Seite 2). Dasselbe gilt für markante Sehenswürdigkeiten und andere Landmarken (vgl. Avelar 2008, Seite 138; zit. n. HGSD 2005).

---

<sup>5</sup> Vgl. die Überlegungen von Morrison (1996) über die geeignete Darstellung von Transportnetzwerk und Cain u. a. (2008), der einen Leitfaden für die Erstellung von Liniennetzplänen in Nordamerika präsentiert.

### Präferenzen des Kartenlesers

Avelar (2008) weist darauf hin, dass Passagiere möglicherweise gewisse Präferenzen haben, welche Design-Elemente und Map Styles mitbestimmen können<sup>6</sup>. Eine vorgängige Umfrage kann aus Sicht des Autors dieser Arbeit helfen solche Präferenzen zu erkennen. Des Weiteren kann es hilfreich sein regelmässige und wiederkehrende Rückmeldungen der Passagiere einzuholen (analog dem Vorgehen bei der Entwicklung der Benutzeroberfläche einer Software).

### Graphische Restriktionen

Graphische Restriktionen sind durch den Verwendungszweck und technische Gegebenheiten bedingt. Eine typische technische Restriktion ist das Druckmedium. Ein Beispiel für den Verwendungszweck der Karte sind Liniennetzpläne, die an der Haltestelle in einem Panel aufgehängt oder in einem faltbaren Fahrplan für die Hosentasche abgedruckt sind. Diese Zweckbestimmung hat einen Einfluss auf die Darstellung, sowohl auf den Detaillierungsgrad wie auch auf das Layout und die Farbwahl (vgl. Morrison 1996; Avelar 2008).

### 2.4.2 Beispiel für benötigte Daten

Nachdem die Anforderungen definiert sind, müssen die benötigten Daten für die Erstellung zusammengetragen werden. Im folgenden werden die erforderlichen Daten am Beispiel des Liniennetzplan des Zürcher Verkehrsbundes analysiert.

Der Liniennetzplan des Kanton Zürichs (vgl. Abbildung 2.8) visualisiert insgesamt vier Transportmittel, wobei die S-Bahn als primäres Verkehrsmittel identifizierbar ist. Jede S-Bahnlinie wird mit einer eigenen Farbe (French Style, vgl. Kapitel 2.1.1) dargestellt. Eine Ausnahme bilden die S-Bahnlinien ausserhalb des Verbundgebietes. Diese werden mit einer einheitlichen Farbe (grau) visualisiert. Wie bereits im Kapitel 2.2.2 erwähnt werden drei weitere Verkehrsmittel mit einem geringeren Detailgrad dargestellt. Weiter sind vier markante Gegebenheiten hervorgehoben, nämlich die drei wichtigsten Seen<sup>7</sup>, der Damm zwischen Rapperswil SG und Pfäffikon SZ, der Flughafen Zürich und das Verbundgebiet bzw. die Kantonsgrenze. Komplett weggelassen wurden Flüsse, wie Limmat und Rhein. Die Signatur der Haltestellen unterscheiden sich je nachdem, ob ein Zug immer oder nur zeitweise an der Haltestelle hält. Des Weiteren kann eine Haltestelle auch ein Mobility-Standort<sup>8</sup> sein und wird dementsprechend mit einem Symbol gekennzeichnet.

---

<sup>6</sup> "Transport users may have certain preferences that may determine the design elements and also style of map." (Avelar 2008, Seite 140)

<sup>7</sup> Sehr klein aber fast nicht wahrnehmbar ist auch ein vierter See (Türlersee) abgebildet.

<sup>8</sup> Mobility ist ein Carsharing Service.

Für die Erstellung des S-Bahn-Liniennetzplans des Zürcher Verkehrsverbundes werden also folgende Informationen benötigt:

- Haltestellen (Name, Position bzw. Lage, Mobility Standort, Halt auf Verlangen)
- Weitere Verkehrsknoten wie z.B. Abzweigungen
- Streckennetze (geom. vereinfacht) der einzelnen Transportmittel bzw. Betriebsmodi
- Routeninformationen (Name, Farbe, Teilbetrieb) der einzelnen Betriebsmodi
- Informationen zu den Landmarken (Geometrie und Lage, Art der Fläche)
- Position zur Platzierung des Flughafensymbols
- Geometrie und Lage der Kantons- bzw. Verbundgrenze

Aufgrund der unterschiedlichen Darstellung einzelner Transportmittel kann es sinnvoll sein die Streckennetze separat zu behandeln. Ein Streckennetz wird in Streckensegmente unterteilt. Jeder Streckenabschnitt zwischen zwei benachbarten Haltestellen oder anderen Verkehrsknoten bildet ein Streckensegment.

## 2.5 Zusammenfassung

Liniennetzpläne von Verkehrsnetzwerken sollen den Passagieren die Orientierung erleichtern. Zu diesem Zweck werden topographische Karten soweit vereinfacht, dass nur die notwendige Information ersichtlich ist. Henry Beck entwarf 1931 einen Liniennetzplan der Londoner U-Bahn, die unter dem Namen Tube Map weltbekannt wurde. Auf ihn gehen Regeln für die Darstellung von schematischen Liniennetzplänen zurück. Solche Liniennetzpläne enthalten verschiedene graphische Elemente (Signaturen), welche die Netzwerkinformation sowie Landmarken und eventuell Points of Interest repräsentieren. In der Regel wird ein primäres Verkehrsmittel graphisch hervorgehoben und Haltestellen mit einer geeigneten Punktsignatur markiert. Eine besondere Bedeutung kommt dabei der Beschriftung zu, die möglichst funktional gehalten werden muss. Alle notwendigen Daten fließen in die Gestaltung des Liniennetzplans ein und beeinflussen auf ihre Weise die graphische Gestaltung.

# 3 | Manuelle Erstellung eines schematischen Liniennetzplanes

Schematische Liniennetzpläne werden in der Regel manuell erstellt, da noch keine Computerprogramme existieren, die diese Aufgabe komplett lösen können.

## 3.1 Existierende Werkzeuge

Während Beck seine Tube Map noch am Zeichenbrett entworfen hat, werden heute Computerprogramme, insbesondere Grafik- oder Layout-Software, für die manuelle Erstellung eines Liniennetzplans eingesetzt. Becks Entwurf diente als Grundlage für Layoutregeln die bis heute Gültigkeit besitzen<sup>1</sup>. Die Anwendung dieser Layoutregeln ist seither ein zentrales Werkzeug bei der Erstellung schematischer Liniennetzpläne.

### 3.1.1 Software

**Grafik- und Layoutprogramme** Heute werden die meisten (schematischen) Liniennetzpläne durch Designer erstellt (vgl. Avelar 2002 und Allard 2009). Aufgrund der Tatsache, dass ein Liniennetzplan fast ausschliesslich aus grafischen Primitiven besteht, sind vektorbasierte Grafikprogramme besonders gut geeignet, da die Skalierbarkeit von Vektorgrafiken eine Ausgabe des Liniennetzplanes ohne Verlust in mehreren Grössen erlaubt. Des Weiteren bieten professionelle Grafikprogramme eine Reihe von Werkzeugen an, die die Bearbeitung des Layouts erleichtern, beispielsweise das Zeichnen von oktilinearen Linien oder die Snapping-Funktion. Ein Export in ein vektorbasiertes Datenformat, beispielsweise SVG, ermöglicht eine leichte Weiterverarbeitung bzw. das Finishing in einem Layoutprogramm.

**GIS Programme** Geographische Informationssysteme (GIS) werden bei topographischen Karten verwendet. Die wahrscheinlich bekannteste Integration schematischer Darstellung von räumlichen Daten ist die Erweiterung ArcGIS Schematics<sup>2</sup> von ESRI. ArcGIS Schematics generiert automatisch mit einem mitgelieferten oder benutzerdefinierten Layout eine schematische Darstellung der Daten. Des Weiteren können Änderungen der Daten direkt synchronisiert werden.

---

<sup>1</sup> Mehrere Forscher (u.a. Hong, Merrick und Nascimento 2006, Wolff 2007, Stott 2008, Wolff 2013) haben anhand Becks Tube Map und dessen Prinzipien Regelkataloge für das Kartenlayout formuliert. Eine Aufstellung des Regelkatalogs nach Wolff (2013) ist im Anhang A zu finden.

<sup>2</sup> vgl. <http://www.esri.com/software/arcgis/extensions/schematics> (22. September 2016)

**Visualisierungsprogramme** Seit ein paar Jahren gibt es neben den oben genannten professionellen Softwarelösungen auch Visualisierungsprogramme, wie z.B. Edraw Max<sup>3</sup> und Concept Draw PRO<sup>4</sup>, welche eine Symbolbibliothek zur Erstellung von Metro Maps anbieten.

Für die selbsterstellten Graphiken der vorliegenden Arbeit werden Adobe Illustrator CC 2015 und ArcGIS 10.4 verwendet. ArcGIS Schematics, Edraw Max und Concept Draw PRO werden in dieser Arbeit nicht weiter diskutiert. Für eine Einführung in ArcGIS Schematics empfiehlt der Autor dieser Arbeit die Online-Hilfe<sup>5</sup> von ArcGIS Schematics und den Blog-Eintrag über die Erstellung einer Route Map mit ArcGIS Schematics (vgl. Buckley 2007).

#### 3.1.2 Metro Map Layout Regeln

In der Tube Map vereinfacht Henry Beck durch Generalisierung der Streckensegmente und Verzerrung der Distanzen zwischen den Haltestellen die damalige Underground Railway Map von London. Gleichzeitig werden die Topologie des Netzwerkes und die relativen Positionen der Haltestellen beibehalten.

Die Wichtigsten von Becks Tube Map abgeleiteten Layoutregeln können wie folgt zusammengefasst werden (Nummerierung nach Wolff 2013, Seite 715)<sup>6</sup>:

**Regel 1** Oktilineare Ausrichtung der Streckensegmente.

**Regel 2** Topologie nicht verändern.

**Regel 3** Knicke entlang einer Route vermeiden.

**Regel 4** Relative Positionen der Haltestellen erhalten.

**Regel 5** Länge der Streckensegmente möglichst einheitlich und kurz halten.

**Regel 6** Jede Haltestelle soll gut lesbar beschriftet sein.

**Regel 7** Jede Route hat eine Farbe, welche sich gut von den anderen Routen abhebt.

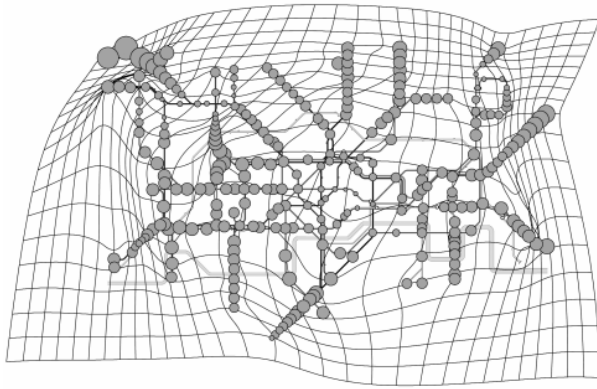
Diese Layoutregeln werden im Folgenden auf die vier thematischen Bereiche Haltestellen, Streckennetz, Routen und Topologie angewendet.

<sup>3</sup> vgl. <https://www.edrawsoft.com/subwaymap.php> (2. Juni 2016)

<sup>4</sup> vgl. <http://www.conceptdraw.com/examples/subway-map-symbols> (2. Juni 2016)

<sup>5</sup> vgl. <https://desktop.arcgis.com/en/arcmap/latest/extensions/schematics> (2. Juni 2016)

<sup>6</sup> Siehe auch Anhang A.



**Abbildung 3.1:** Beck's Tube Map mit einem überlagerten Grid, das die geometrische Verzerrung der Realität aufzeigt. Die Grösse der Kreise bezieht sich auf die Proportionen der Distanz zur ursprünglichen geographischen Position der Haltestelle. (Quelle: Jenny 2006, Seite 2)

#### **Haltestellen**

Die Abstände zwischen zwei Haltestellen sollten möglichst einheitlich sein. Dabei entsteht eine massive Verzerrung der geometrischen Realität (vgl. Abbildung 3.1). Lediglich die relative Position zur ursprünglichen geographischen Position bleibt so weit wie möglich erhalten. So werden die Abstände zwischen den Haltestellen in Gebieten mit einer kleinen Dichte von Haltestellen kleiner als in Wirklichkeit dargestellt ("Komprimierung", vgl. Bain 2010, Seite 2), während die Abstände zwischen den Haltestellen in Gebieten mit grosser Dichte grösser (Verdrängung) werden, wie u.a. Bain 2010 aufzeigt. Ein weiteres wichtiges Element ist die Beschriftung (Label) der Haltestellen. Dabei ist zu beachten, dass Labels keine andere Informationen überdecken und einheitlich positioniert werden. So sollten Labels entlang einer Linie möglichst auf der gleichen Seite der Linie positioniert werden (vgl. Wolff 2007, Seite 26).

#### **Streckennetz**

Durch die Einführung der Oktilinearität durch Beck haben alle Linien einen fixen Winkel von einem Vielfachen von 45 Grad. Die Linien werden soweit generalisiert, dass sie eine gerade Linie darstellen. Für eine bessere Visualisierung können (zusätzliche) Knicke in Streckensegmenten eingeführt werden. Solche Knicke können bestimmte Eigenschaften der ursprünglichen Geometrie hervorheben (vgl. Avelar und Hurni 2006, Seite 219), beispielsweise um anzudeuten, dass die Route um ein natürliches Hindernis (Berg, See) herumführt. Andererseits kann der Kartenleser einer Route mit möglichst wenigen Knicken besser folgen (vgl. Wolff 2007, Seite 26). Benachbarte Streckensegmente sollten eine möglichst einheitliche Länge haben und möglichst kurz gehalten werden (vgl. Wolff 2013, Seite 715)<sup>7</sup>. Neben der einheitlichen Linienlänge, welche die Karte übersichtlicher macht, verringert es nach Ansicht des Autors dieser Arbeit auch die Wahrscheinlichkeit einer Interpretation der Streckenlänge als Distanzangabe durch den Kartenleser, da damit dem Kartenleser die Verzerrung der Karte in Erinnerung gerufen wird.

<sup>7</sup> Milea u. a. (2012) gehen einen anderen Weg und schlagen in ihrem Artikel vor, die Länge der Streckensegmente relativ zur Reisezeit zu zeichnen.



### **Route**

Jede Route sollte mit einer einheitliche Farbe dargestellt werden. Dies erleichtert es einer Route auf der Karte zu folgen (vgl. Wolff 2007, Seite 26). Dabei ist jedoch der gewünschte Style (vgl. Kapitel 2.1.1) zu berücksichtigen und zwischen primären und sekundären Verkehrsmitteln im Layout zu unterscheiden.

### **Topologie**

Der ortskundige Kartenleser hat eine grobe Vorstellung im Kopf wie die geographischen Gegebenheiten der Region, in der er sich aufhält, beschaffen sind. Diese mentale Karte des Kartenlesers sollte der Liniennetzplan unterstützen. Deshalb müssen wesentliche Merkmale der Topologie erhalten bleiben, z.B. die Lage eines Distrikts oder die Beibehaltung der Himmelsrichtungen.

## **3.2 Vorgehensweise**

Im folgenden Abschnitt soll anhand eines vereinfachten Beispiels die Methodik aufgezeigt werden, welche bei der Erstellung eines Liniennetzplans zur Anwendung kommt. Ausgehend von einer topographischen Karte und den erforderlichen Daten des Liniennetzes (Haltestellen, Streckenverlauf) wird demonstriert, welche Arbeitsschritte erforderlich sind, um zu einer schematischen Darstellung zu gelangen. In weiteren Arbeitsschritten werden die oben beschriebenen Layoutregeln angewendet. Dabei soll auch die Komplexität dieses Vorgangs am Beispiel erläutert werden.

Als Beispiel dient das S-Bahnnetz des Zürcher Verkehrsbaus, das in einem Kartenausschnitt dargestellt werden soll. Der Kartenausschnitt beinhaltet die Stadt Zürich sowie einen Teil des Netzes entlang des Zürichsees. Aus der topographischen Karte übernommen werden ein Ausschnitt des Zürichsees und die Stadtgrenze der Stadt Zürich (vgl. Abbildung 3.2).

### **3.2.1 Ausgangsdaten**

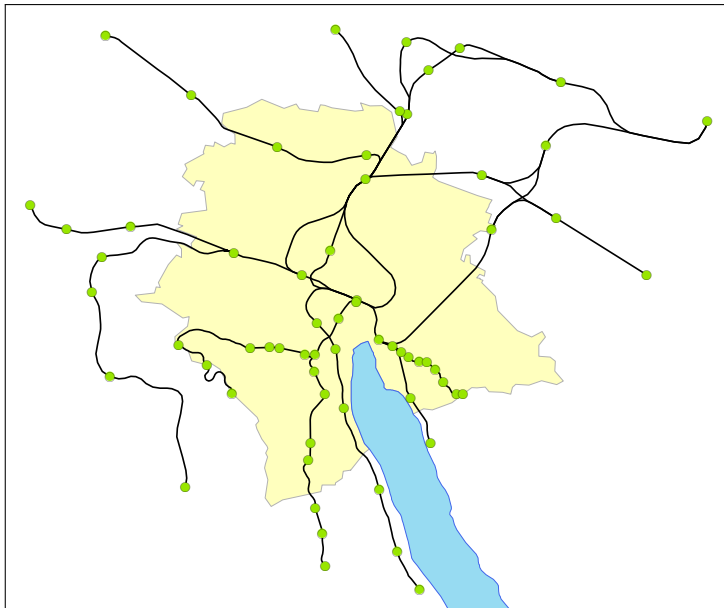
Damit ein Liniennetzplan erstellt werden kann, müssen zunächst die Ausgangsdaten beschafft werden. Zur Reduktion der Komplexität werden fiktive S-Bahnlinien dargestellt.

### **Haltestellen und Streckennetz**

Die Haltestellen und das Streckennetz (vgl. Abbildung 3.2) wurden im Rahmen eines UNIGIS Assignments<sup>8</sup> basierend auf OpenStreetMap Daten manuell durch den Autor dieser

---

<sup>8</sup> UNIGIS Modul: QGIS (<http://unigis.at/index.php/qgis-an-open-source-desktop-gis>; 29. Juni 2016), Einreichung des Assignments: Sommer 2015.



**Abbildung 3.2:** Ausgangslage zur Erstellung des Liniennetzplans im Zürcher Stadtgebiet (eigene Grafik, Datengrundlage: OpenStreetMap, swissBOUNDARIES<sup>3D</sup>)

Arbeit digitalisiert<sup>9</sup>.

### Landmarken

Die Polygone der Stadtgrenze und der Fläche des Zürichsees (vgl. Abbildung 3.2) wurden aus dem kostenlosen Datensatz swissBOUNDARIES<sup>3D</sup> (Ausgabe 2006)<sup>10</sup> von Swisstopo extrahiert.

### 3.2.2 Ansätze für die manuelle Erstellung

In diesem Abschnitt werden zwei mögliche Ansätze skizziert. Die Ansätze unterscheiden sich im Ausgangspunkt der Gestaltung des Liniennetzplans. Der erste Ansatz wählt das Streckennetz ohne Routeninformationen als Ausgangspunkt. Erst in einem weiteren Schritt werden die Routeninformationen hinzugefügt. Der zweite Ansatz berücksichtigt die Routeninformationen bereits im ersten Schritt des Erstellungsprozess. Dabei wird das Layout iterativ mit jedem einzelnen Streckensegment ausgebaut.

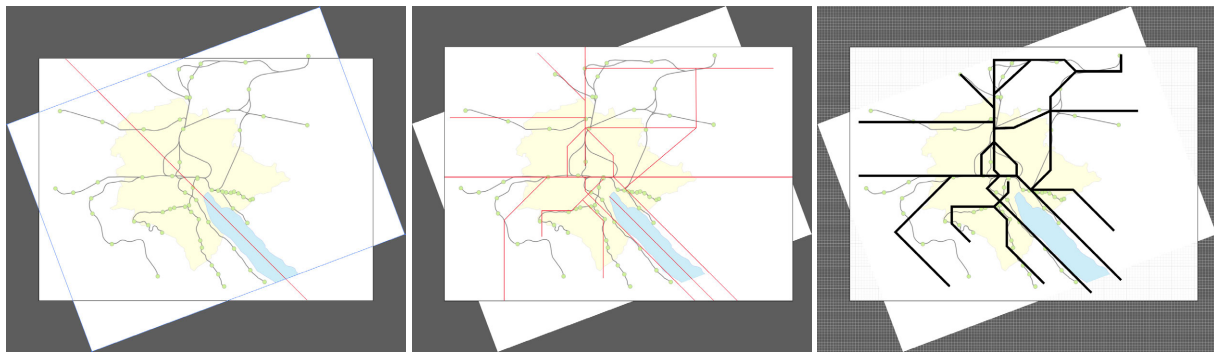
#### Ausgangspunkt Streckennetz

Aus den vorliegenden GIS-Daten wird mit einer GIS-Software, beispielsweise mit ArcGIS oder QGIS<sup>11</sup>, eine Karte erstellt. Die Karte beinhaltet das Streckennetz, die Haltestellen, die Gemeindegrenzen und die Darstellung des Zürichsees. Anschliessend wird die Karte in ein geeignetes Grafikformat (z.B.) exportiert. Das exportierte Bild wird später als Vorlage in einem vektorbasierten Grafikprogramm, beispielsweise Adobe Illustrator, verwendet.

<sup>9</sup> Der Digitalisierungsmasstab ist ca. 1:20000.

<sup>10</sup> Quelle: Bundesamt für Landestopographie, <http://www.swisstopo.admin.ch/internet/swisstopo/de/home/products/landscape/swissBOUNDARIES3D.html> (9. Juni 2016)

<sup>11</sup> vgl. <http://qgis.org/de/site> (16. Juli 2016)



(a) Ausrichtung See

(b) Hilfslinien

(c) Streckennetz

**Abbildung 3.3:** Schritte zum oktilinearen Streckennetz mit Adobe Illustrator CC an einem Beispiel. Schritt 1: Ausrichtung See, Schritt 2: Hilfslinien einzeichnen, Schritt 3: Streckennetz einzeichnen. (eigene Grafik, Datengrundlage: OpenStreetMap, swissBOUNDARIES<sup>3D</sup>)

Die Umwandlung der GIS-Daten in das Datenformat SVG wäre von grossem Nutzen. Aufgrund der Tatsache, dass graphische Primitive und Linien mit verschiedenen SVG-Elementen dargestellt werden können, wird der Vorteil gegenüber einer Bildvorlage jedoch wieder aufgehoben. Eine Weiterbearbeitung der Linien ist dadurch nur bedingt möglich. Andernfalls wäre es möglich die Geometrien bereits in der GIS-Software mit geeigneten Tools zu vereinfachen. Darüberhinaus könnte die Topologie während der Generalisierung automatisch überwacht werden. Alternativ besteht die Möglichkeit eine generalisierte Karte als Bildvorlage zu verwenden.

Aufgrund der geographischen Position des Zürcher Hauptbahnhofs, der geraden Form des oberen Zürichsees (vgl. Abbildung 3.2) und der Streckenführung entlang der Zürichseeufer ist es aus Sicht des Autors dieser Arbeit sinnvoll die Karte leicht gegen den Uhrzeigersinn zu drehen, sodass der Zürichsee in einem Winkel von 45 Grad gegenüber der Nord-Süd-Achse ausgerichtet ist (vgl. Abbildung 3.3a). Damit kann die Regel der Oktilinearität besser berücksichtigt werden, da Routen entlang des Zürichsees führen.

Um den Streckenverlauf nach den Regeln der Oktilinearität zu schematisieren, wird über die Ausgangskarte ein Netz von oktilinearen Hilfslinien gelegt, das den Streckenverlauf ungefähr nachzeichnet (vgl. Abbildung 3.3b).

In einem Korrekturprozess werden die schematischen Hilfslinien unter Beibehaltung der Oktilinearität dem Streckenverlauf angepasst (vgl. Abbildung 3.3c). Wenn nun Haltestellen hinzugefügt werden, entsteht die bereits diskutierte Verzerrung der geometrischen Realität. Die Abstände zwischen Haltestellen entsprechen nicht mehr den ursprünglichen topographischen Relationen. Die Gemeindegrenze und das Seeufer werden ebenfalls vereinfacht. Dabei ist besonders auf die Erhaltung der ursprünglichen Topologie zu achten. Wenn beispielsweise eine Haltestelle innerhalb der Gemeindegrenze liegt, so muss sie

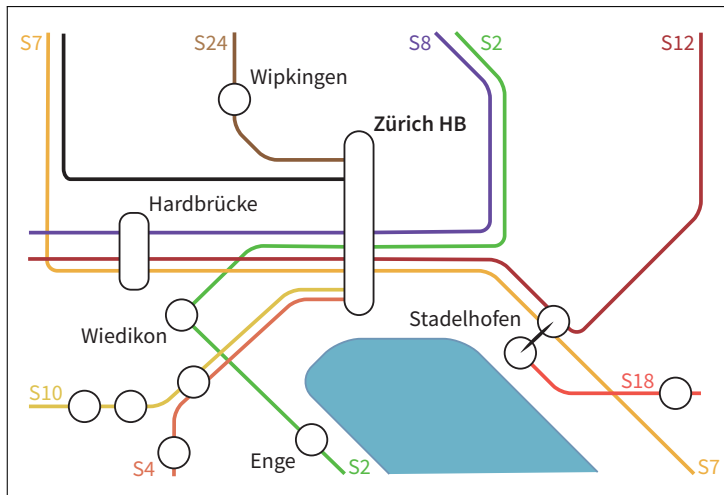


**Abbildung 3.4:** Einfacher Liniennetzplan (ohne Routen und Beschriftung) des S-Bahn-Streckennetz im Stadtgebiet von Zürich. (eigene Grafik)

auch nach der Vereinfachung innerhalb der Gemeindegrenze liegen.

Ist das Streckennetz eingezeichnet und sind die Haltestellen platziert, müssen die verschiedenen Routen, die auf einem Streckensegment verkehren, dargestellt (Liniensignatur) und beschriftet werden. Die Routeninformationen müssen iterativ hinzugefügt werden. Mit jeder Route verändern sich die Abstände zwischen den Haltestellen sowie zwischen adjazenten Linien. Dies erfordert einen zusätzlichen Aufwand, wenn man bedenkt, dass sich das Layout mit jeder Iteration verändert. Dasselbe gilt für die Beschriftung der Haltestellen. Der Liniennetzplan des Zürcher Verkehrsverbunds (vgl. Abbildung 2.8) zeigt eine solche Situation am Beispiel des Zürcher Hauptbahnhofs (mit Stopps von 21 S-Bahnlinien und Fernverkehrszügen).

Jede Beschriftung und jede Liniensignatur, die auf einem Streckensegment eingezeichnet werden, haben eine Auswirkung auf einen Bereich des Layouts oder gar auf das gesamte Layout. Beispielsweise beansprucht die Einzeichnung mehrerer Routen in einem Streckensegment mehr Raum. Dies wirkt sich auf die Darstellung der Haltestellen aus, die in Form und Proportionen dem Streckensegment angepasst werden müssen. Bei der Platzierung von Labels muss beachtet werden, dass keine wichtigen Elemente des Liniennetzplans überdeckt werden. Das bedingt eine weitere geometrische Verzerrung und eventuell eine Anpassung der Liniendarstellung in Relation zu den Landmarken oder anderen Linien. Dieser Prozess muss iterativ ausgeführt werden. Jedes neu hinzugefügte Element erfordert weitere Iterationen. Mit diesen Arbeitsschritten wächst die Komplexität um ein Vielfaches. Die manuelle Ausführung beansprucht daher immer mehr Zeit. Eine Automatisierung dieser Arbeitsschritte könnte den Vorgang vereinfachen und erheblich



**Abbildung 3.5:** Manuell erstellte Karte im Bereich des Zürcher Hauptbahnhofes mit einer *sehr* limitierten Darstellung der S-Bahnlinien. (eigene Grafik)

Zeit einsparen.

### Ausgangspunkt Streckensegment

Geht man bei der Erstellung eines schematischen Liniennetzplans von einem Streckensegment zwischen Haltestellen aus, ändert sich das Vorgehen. Jedes Streckensegment wird einzeln und unter Berücksichtigung aller Attribute (z.B. Beschriftung und Routen) gezeichnet. Begonnen wird mit einem geeigneten Liniensegment (z.B. in einer Region mit einer hohen Dichte an Haltestellen). Das Segment wird ausgearbeitet und anschliessend werden iterativ weitere Segmente hinzugefügt und bearbeitet. Dieses Vorgehen eignet sich besonders für die Detaildarstellung von Ausschnitten des Netzplans (z.B. wenn an einer Haltestelle lediglich die benachbarten Haltestellen und Routen dargestellt werden, vgl. Abbildung 3.5). Theoretisch könnte der gesamte Liniennetzplan auf diese Weise erstellt werden. Komplex ist bei diesem Vorgang lediglich die Bestimmung der relativen Positionen der Haltestellen. Ein solches Vorgehen könnte dann sinnvoll sein, wenn bei der Darstellung des Liniennetzplanes wenig Rücksicht auf topographische Gegebenheiten (z.B. relative Positionen der Haltestellen oder die Himmelsrichtungen) genommen werden muss. Andernfalls ergibt sich jedoch die selbe Komplexität und derselbe Arbeitsaufwand wie im ersten Ansatz.

### 3.3 Zusammenfassung

Eine Anzahl von vektorbasierten Grafikprogrammen eignet sich für die manuelle Bearbeitung des oktilinearen Layouts eines Liniennetzplanes. Voraussetzung ist geeignetes Datenmaterial (topographische oder vereinfachte Karte mit Routeninformationen) als Ausgangspunkt. Die Bearbeitung der Ausgangskarte soll so erfolgen, dass die Metro Map Layout Regeln weitgehend eingehalten werden. So soll beispielsweise das Streckennetz oktilinear visualisiert werden und die Abstände zwischen Haltestellen gleichmässig sein. Auch sollen wesentliche Merkmale der Topologie erhalten bleiben.

Wählt man das reine Streckennetz als Ausgangspunkt werden die Routeninformationen in einem weiteren Schritt iterativ hinzugefügt. Eine andere Möglichkeit besteht darin mit einem Streckensegment zu beginnen und die Routeninformationen einzuarbeiten, wobei die Karte iterativ um weitere Streckensegmente erweitert wird.

Die schematische Darstellung eines reinen Streckennetzes ohne Routeninformationen kann in der Regel mit einfachen Mittel erstellt werden. Werkzeuge und Layout-Regeln sind vorhanden. Je mehr Information der Darstellung hinzugefügt wird, desto anspruchsvoller wird die Aufgabe. Ab einem bestimmten Grad der Komplexität lassen sich zusätzliche Informationen nur noch mit Iterationen in den Liniennetzplan integrieren, da sie jeweils aufwendige Anpassungen des Layouts bedingen. Dieser Vorgang könnte durch eine Automatisierung optimiert werden.

## 4 | **Vorhandene Ansätze und Möglichkeiten der Automatisierung**

Im vorangegangenen Kapitel wird vorgeschlagen bei der Erstellung von schematischen Liniennetzplänen den Aufwand an Zeit und Arbeit mit Hilfe der Automatisierung von Arbeitsschritten zu minimieren. In diesem Kapitel soll aufgezeigt werden, welche Ansätze zur Automatisierung von Teilschritten bereits existieren und welche Werkzeuge die Möglichkeit bieten weitere Automatisierungsansätze zu entwerfen.

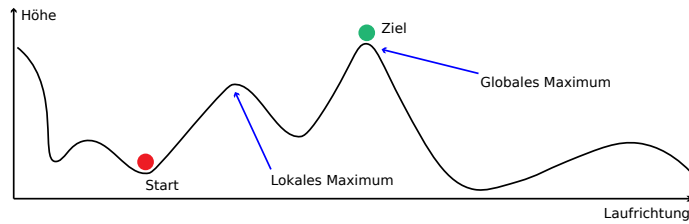
### 4.1 **Generierung von Metro Maps**

Für die Generierung von Metro Maps wurden in den letzten zehn Jahren verschiedene Ansätze der Automatisierung publiziert. In der Regel handelt es sich um Algorithmen, die von einem initialen Datensatz ausgehen und welche das Layout der Streckensegmente, Haltestellen und Routen unter Berücksichtigung der Metro Map Layout Regeln (vgl. Kapitel 3.1.2 und Anhang A) generieren können, wie die folgenden Beispiele zeigen. Liniennetzpläne können mit den Elementen der Graphentheorie, nämlich Knoten, Kanten und Pfade, beschrieben werden. Mit Hilfe von Algorithmen und der Graphentheorie kann der Datensatz im Form eines schematischen Liniennetzplan ausgegeben werden.

#### 4.1.1 **Spring Embedders**

##### **Hong, Merrick und Nascimento 2006**

Zu den ersten die sich der automatischen Generierung von schematischen Karten im öffentlichen Verkehr annehmen gehören Hong, Merrick und Nascimento (2006). Der von ihnen diskutierte Ansatz verwendet verschiedene Force-Directed Algorithmen (sogenannte Spring Embedders) zur Generierung eines Liniennetzplanes. Dabei handelt es sich um ein kräftebasiertes Layout-Verfahren, bei welchem sich die benachbarten Knoten nach festgelegten Kriterien anziehen, während sich die anderen Knoten abstossen. Allerdings berücksichtigt dieser Ansatz weder die Oktilinearität noch die ursprüngliche Topologie (vgl. Oke und Siddiqui 2015).



**Abbildung 4.1:** Lokales und globales Maximum im Hill Climbing Algorithmus anhand der Analogie dargestellt. (eigene Grafik)

### 4.1.2 Hill Climbing Algorithmus

#### Stott 2008

Einen anderen Ansatz für die Generierung einer Metro Map wählt Stott (2008) in seiner Dissertation und verwendet einen Hill Climbing Algorithmus. Ein Hill Climbing Algorithmus ist ein heuristisches Optimierungsverfahren. Dabei wird das Layout iterativ verändert, solange ein besseres Resultat erzielt wird. Das Resultat jeder Veränderung wird jeweils mit Kriterien bewertet. Das Problem dieses Algorithmus sind lokale Maxima, benötigt wird aber das globale Maximum (vgl. Abbildung 4.1). Muss beispielsweise vor der Besteigung eines Berggipfels eine andere Anhöhe überwunden werden und die Bewertung der Position erfolgt in Höhenmeter, so ist das lokale Maximum diese Anhöhe. Der Bergsteiger kann nicht weitergehen, ohne dass er wieder absteigt. Dadurch wird das Bewertungsergebnis schlechter.

Zur Erstellung einer Metro Map richtet Stott die geographischen Positionen der Knoten an einem Grid aus. Der Algorithmus iteriert über alle Knoten und prüft, ob eine bessere (Grid-)Position des jeweiligen Knotens vorliegt. Die möglichen Positionen sind jeweils von einem Radius und bestimmten Verschiebekriterien abhängig. Die Bewertung erfolgt über die Summe von gewichteten Bewertungskriterien, welche die Metro Map Layout Regeln (z.B. die Einhaltung der Oktilinearität) bewerten. Gibt es für einen Knoten eine bessere Position, so wird dieser dorthin verschoben. Die Optimierung der einzelnen Knoten wird solange wiederholt, bis ein besseres Resultat erzielt wird.

Das Problem mit den lokalen Maxima versucht Stott zu beheben, indem Knoten zusammengefasst werden (Clustering). Clusters werden als eine Einheit betrachtet und verschoben. So können überlange Brücken durch das Verschieben des Clusters gekürzt und unnötige Richtungswechsel entlang eines Pfads, bestehend aus Knoten mit einem Grad von 2, einfacher behoben werden.

Der Ansatz erzielt gute Resultate<sup>1</sup>. Aufgrund der Wechselwirkung zwischen den Bewertungskriterien werden jedoch nicht alle Metro Map Layout Regeln gleich gut durchgesetzt.

<sup>1</sup> Für eine Diskussion des Ansatzes von Stott vgl. Wolff 2007, Chivers 2014 und Oke und Siddiqui 2015.



Wird beispielsweise das Kriterium zur Reduktion der Kantenlänge stärker gewichtet, hat dies einen relevanten Einfluss auf die Oktilinearität (vgl. Wolff 2007, Seite 29).

### **Chivers 2014**

Der Ansatz von Stott wird von Chivers (2014) übernommen und optimiert. Während Richtungswechsel bei der Methode von Stott nur an einer Haltestelle möglich sind, führt Chivers sogenannte Bend Points ein und ermöglicht damit Knicke entlang der Kanten. Zusätzlich werden adjazente Kanten mit adjazenten Knoten, die einen Grad von 2 aufweisen, durch eine einzige Kante ersetzt. Dies verbessert die Performance, da weniger Knoten überprüft werden müssen. Durch die eingeführten Bend Points wird aber dennoch eine gewisse Flexibilität gewährt. Die Performance wird nochmals verbessert, indem Chivers die Anzahl möglicher Verschiebepositionen reduziert. Es kommen nur Positionen in Frage, die sich (ausgehend von der aktuellen Position) entlang einer oktilinearen Linie befinden.

Chivers's Erweiterungen umfassen auch eine neue Clustering-Methode und zusätzliche Bewertungskriterien. Die Gewichtung der Kriterien wird aufgrund der neuen Bewertungskriterien ebenfalls angepasst. Die letzte Erweiterung gilt dem Labelling. Labels müssen nicht nur horizontal platziert werden, sondern orientieren sich an vier möglichen Winkeln (vgl. Chivers 2014, Seite 81 ff.).

### **4.1.3 Mixed-Integer Linear Programming**

#### **Nöllenburg und Wolff 2011**

Einen anderen Weg gehen Nöllenburg und Wolff (2011). Zur Erstellung einer Metro Map verwenden sie Mixed-Integer Linear Programming (MIP<sup>2</sup>). Basierend auf Variablen, Bewertungsfunktionen und Bedingungen (durch Gleichungen und Ungleichungen gegeben) soll eine Zielfunktion optimiert werden. Die Metro Map Layout Regeln (vgl. Kapitel 3.1.2 und Anhang A) werden in Hard- und Soft-Constraints (Bedingungen) unterteilt. Während die Hard-Constraints (Regel 1 und 3) in die Nebenbedingungen einfließen und zwingend erfüllt werden müssen, werden die Soft-Constraints (Regeln 3 bis 6) in die Zielfunktion übernommen und sollen optimiert werden (vgl. Ehinger 2009, Seite 10).

Der Vorteil gegenüber dem Ansatz von Stott (2008) Ansatz ist die Unterscheidung von Hard- und Soft-Constraints. Dadurch werden die Regeln 1 bis 3 hinsichtlich ursprünglicher Einbettung, Oktilinearität und einheitlicher Distanz zwischen den Haltestellen auf jeden Fall eingehalten. Es gibt keine Wechselwirkungen mit anderen Regeln, die diese Bedingungen aufweichen könnten.

---

<sup>2</sup> Für eine Einführung in MIP vgl. Smith und Taskin 2008.

### **Oke und Siddiqui 2015**

Oke und Siddiqui (2015) vereinfachen den MIP Ansatz von Nöllenburg (2005), indem wesentliche Bedingungen abgeschwächt (relaxed) und die Anzahl der Bewertungsfunktionen auf zwei reduziert wird. Darüber hinaus führen zusätzliche Anpassungen an Gleichungen zu einer besseren Lösung nach dem Paretoprinzip (80/20-Regel). Die Modifizierungen ermöglichen auch eine Reduktion der Zeit, die für die Generierung benötigt wird (vgl. Oke und Siddiqui 2015, Seite 14).

### **Milea u. a. 2012**

Milea u. a. (2012) untersuchen die Möglichkeit eine Metro Map zu erstellen, welche dem Kartenleser ermöglicht, anhand der Kantenlängen die kürzeste Route von A nach B abzulesen. Dazu erweitern Milea u. a. den MIP-Algorithmus von Nöllenburg und Wolff (2011) mit einer weiteren Zielfunktion.

### **4.1.4 Focus + Context Metro Maps**

Einen anderen Ansatz zur Darstellung einzelner Routen in einem Transportsystem beschreiben Wang und Chi (2011). Eine Route wird in den Fokus gestellt, während das übrige Streckennetz im Hintergrund dargestellt wird. Wang und Chi glätten zunächst das geographische Layout des Streckennetzes. Gleichzeitig werden Kanten, die nicht im Fokus stehen, gekürzt. Anschliessend wird ein oktilineares Layout erstellt und im nächsten Schritt werden die Haltestellen beschriftet (vgl. Wang und Chi 2011, Seite 2530, Abbildung 2). Für den Vorgang werden Kriterien als Energy Terms formuliert und anschliessend in einer nicht-linearen Optimierung<sup>3</sup> angewendet. Der Ansatz von Wang und Chi kann für die Anzeige einzelner Routen bzw. Reisewege (z.B. kürzester Weg zwischen zwei Haltestellen) im Transportnetzwerk verwendet werden.

### **4.1.5 Weitere Algorithmen und Methoden**

Weitere Algorithmen und Methoden werden unter anderem von Avelar und Müller (2000), Cabello, Berg u. a. (2001), Merrick und Gudmundsson (2007), Dwyer, Hurst und Merrick (2008) und Sobati und Ahmad (2012) diskutiert. Sie werden im Folgenden kurz vorgestellt.

### **Avelar und Müller 2000**

Avelar und Müller (2000) diskutieren einen Algorithmus, welcher mit geometrischen Operationen eine schematische Karte erstellt, wobei die topologische Beziehungen zwischen den Kartenelementen (Knoten und Kanten) berücksichtigt werden. Der Algorith-

<sup>3</sup> Für eine Einführung in die nicht-lineare Optimierung vgl. <http://www.mit.edu/~9.520/spring08/Classes/optlecture.pdf> (7. Juli 2016).

mus verwendet den Douglas-Peucker Algorithmus<sup>4</sup>, um die Linien zu vereinfachen, und verkürzt die Streckensegmente in einem weiteren Schritt. Anschliessend wird in einem Optimierungsvorgang über alle Punkte (Knoten) iteriert. Dabei werden geometrische und ästhetische Bedingungen überprüft und mit geometrischen Operationen eine neue Position für jeden Punkt ermittelt. Falls das dabei entstehenden Streckensegment Konflikte (Kollision, z.B. Überschneidungen von Streckensegmenten) verursachen sollte, wird eine alternative Position berechnet. Der Optimierungsvorgang wird solange (manuell) wiederholt bis das gewünschte Layout erstellt wird.

### **Cabello, Berg u. a. 2001**

Cabello, Berg u. a. (2001) und Cabello und Kreveld (2015) diskutieren einen weiteren Ansatz. Dabei werden topographisch korrekte Streckensegmente mit oktilinearen Streckensegmenten (Connections) ausgetauscht. Diese Connections besitzen einen oder zwei Knicke, wobei die Position der Knicke so gewählt werden muss, dass die Connection keinen Konflikt mit einer anderen Connection oder einem Punkt verursacht. Wenn die Eigenschaft nicht erfüllt werden kann, so kann keine Lösung gefunden werden.

### **Merrick und Gudmundsson 2007**

Merrick und Gudmundsson (2007) und ein Jahr später Dwyer, Hurst und Merrick (2008) diskutieren einen Algorithmus basierend auf einer Least-Square Regression<sup>5</sup>. In einem ersten Schritt werden die Knoten eines Pfad bzw. einer Route in Blöcke aufgeteilt. Mit der Least-Square Regression wird dann eine Linie gesucht, die möglichst nahe an allen Knoten des Blocks verläuft.

### **Sobati und Ahmad 2012**

Der trigonometrischen Ansatz von Sobati und Ahmad (2012) entfernt zunächst die Knoten mit einem Grad von 2. Anschliessend wird der Graph auf die Planarität überprüft. Falls er nicht planar ist, werden Dummy-Knoten an den Kreuzungen eingeführt. Danach wird mit einer Breitensuche (Breadth-First Search) über die Knoten iteriert und dabei die neue Koordinate mit Hilfe der Trigonometrie (vgl. Sobati 2013, Seite 3) berechnet. Schliesslich werden die zuvor entfernten Knoten wieder eingefügt und abschliessend das Layout beschriftet. Auf diese Weise wird ein gutes oktilineares Layout generiert, andere Kriterien einer Metro Map wie z.B. gleichmässige Abstände zwischen den Haltestellen werden nicht ausreichend berücksichtigt.

---

<sup>4</sup> Für eine Beschreibung des Douglas-Peucker Algorithmus vgl. Douglas und Peucker 1973.

<sup>5</sup> Für eine Einführung in die Methode der kleinsten Quadrate vgl. [https://de.wikipedia.org/wiki/Methode\\_der\\_kleinsten\\_Quadrate](https://de.wikipedia.org/wiki/Methode_der_kleinsten_Quadrate) (5. August 2016).

## 4.2 Interaktives Bearbeiten von Metro Maps

### Wang und Peng 2015

Einen Schritt weiter als die reine Generierung der schematischen Darstellung der Metro Map präsentieren Wang und Peng (2015) in ihrem Artikel. Auch sie generieren eine Metro Map mit Hilfe eines Algorithmus, haben aber ein System entwickelt, das es dem Benutzer erlaubt, Haltestellen (Knoten) auf einer Zeichnungsfläche zu verschieben, worauf die Positionen der übrigen Knoten neu berechnet werden. Des Weiteren erlaubt das System auch das Manipulieren einer Selektion mehrerer Knoten durch gleichzeitiges Verschieben, Rotieren und Skalieren. In einem weiteren Arbeitsschritt kann das System die Haltestellen beschriften (Labels).

Nachdem sogenannte Handles (vom Benutzer gesetzte Knoten) definiert wurden, generiert der Algorithmus von Wang und Peng zuerst ein kurvenförmiges Layout und ignoriert dabei das oktilineare Layout komplett. Erst anschliessend wird daraus (in mehreren Iterationen) ein oktilineares Layout erstellt, wobei die Positionen der Handles unverändert bleiben.

### Chivers und Rodgers 2014

Basierend auf seiner Implementierung eines Hill Climbing Algorithmus (vgl. Kapitel 4.1.2) hat Chivers (2014) eine Applikation für Tablets (SchemaSketch<sup>6</sup>) geschrieben. Mit einer Gesture-basierten Eingabe können damit oktilineare Layouts gezeichnet werden.

## 4.3 Automatisierte Bearbeitung von graphischen Elemente

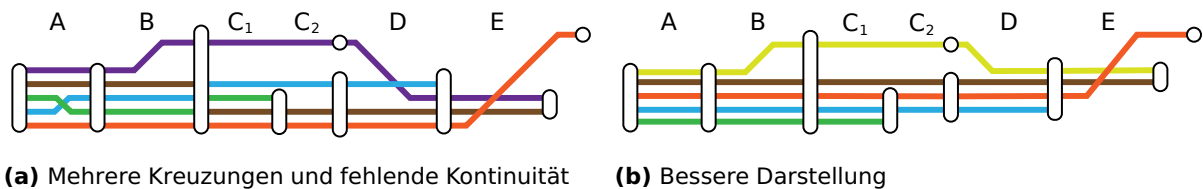
### 4.3.1 Beschriftung

Das Beschriften (Labeling) von räumlichen Elementen ist ein bekanntes Problem der Kartographie. Die Beschriftung (Label) sollte dem assoziierten Element eindeutig zugeordnet werden können. Weiter sollen Labels keine wichtige Informationen überdecken und möglichst konsistent bzw. einer Regel folgend platziert werden. Letzteres hilft dem Kartenleser beim Zuordnen des Labels bzw. beim Finden eines Elementes auf der Karte. Ein Liniennetzplan enthält in erster Linie Labels zur Beschriftung von Haltestellen und Routen. Landmarken wie Seen, Flüsse, grosse Waldgebiete werden in der Regel nicht beschriftet.

Einige der bereits diskutierten Ansätze zur Generierung von Metro Maps berücksichtigen die Platzierung von Labels der Haltestellen. Nöllenburg und Wolff (2011) beispielsweise verwendet dazu einen Graph-Labeling Approach (vgl. Nöllenburg und Wolff 2011, Seite

<sup>6</sup> Vgl. Chivers und Rodgers 2011 und für den Download der Applikation: <https://www.cs.kent.ac.uk/projects/schemasketch> (1. Juli 2016).

### 4.3. Automatisierte Bearbeitung von graphischen Elementen



**Abbildung 4.2:** Probleme bei der Darstellung der Linienführung. Abschnitt A: unnötige Kreuzung, B und C<sub>1</sub>: Routen sind nicht kontinuierlich eingezeichnet, C<sub>2</sub>: vermeidbare Lücke, D: vermeidbare Kreuzung, E: Kreuzung lässt sich nicht vermeiden, A bis E: violette und braune Route sind nicht gut unterscheidbar. Hinweis: Es handelt sich um zwei separate Haltestellen zwischen den Abschnitten C<sub>2</sub> und D. (eigene Grafik)

634)<sup>7</sup>. Allerdings erhöht die Berücksichtigung der Labels den Zeitbedarf zur Generierung massiv. Dadurch kann keine zufrieden stellende Qualität innerhalb eines adäquaten Zeitraums geliefert werden (vgl. Haurert und Niedermann 2015, Seite 689). Beispielsweise benötigt der Algorithmus von Nöllenburg und Wolff (2011) zur Generierung der Metro Map von Sydney 10 Stunden und 31 Minuten inklusive der Platzierung der Labels. Im Vergleich dazu benötigt die Generierung der Metro Map ohne Label nur 23 Minuten (vgl. Haurert und Niedermann 2015, Seite 689).

Das Problem der Beschriftung von räumlichen Daten ist ein allgemeines kartographisches Problem. Hierzu werden verschiedene Algorithmen (z.B. vgl. Edmondson u. a. 1996, Wagner und Wolff 1997, Wolff 1999) angeboten, die sich auf kartographische Probleme beziehen, aber nicht spezifisch für Metro Maps entwickelt wurden. Die Anwendung solcher Algorithmen hat keinen Einfluss auf die Kartenelemente. Das heißt, die Labels werden optimiert gesetzt, die zugrundeliegende Karte bleibt jedoch unverändert. Für die Erstellung eines Liniennetzplanes sind solche Hilfsmittel nur von begrenztem Wert, da ein solcher Map-Labeling-Algorithmus praktisch nach jeder Iteration bzw. nach jeder Anpassung des Layouts neu angewendet werden müsste.

#### 4.3.2 Darstellung der Linienführung

Im Kapitel 3.2.2 wird aufgezeigt, wie aufwändig der manuelle Prozess ist, wenn dem schematischen Netzplan Routeninformationen hinzugefügt werden. Wenn mehrere Routen auf dem gleichen Liniensegment dargestellt werden müssen, muss die Liniensignatur nicht nur vollständig sichtbar sein, sondern auch in der richtigen Reihenfolge visualisiert werden. Überkreuzungen sollten nach Möglichkeit vermieden werden und ebenso sollte auf eine kontinuierliche Darstellung der Reihenfolge in anderen Streckensegmenten geachtet werden (vgl. Abbildung 4.2).

Im Normalfall liegen die Routeninformationen nicht als Linien-Geometrien vor, sondern in einem Datenformat bzw. einer Tabelle, welche die Streckensegmente referenziert. Zu-

<sup>7</sup> Für eine Einführung zum Thema Graph-Labeling Algorithmen vgl. Kakoulis und Tollis 2008.

### 4.3. Automatisierte Bearbeitung von graphischen Elemente

sätzliche Linien müssen nacheinander hinzugefügt werden, wodurch das Streckensegment mehr Raum zur Darstellung benötigt. Benachbarte Elemente müssen entsprechend verdrängt werden. Die Verdrängung ist ein klassisches Werkzeug zur Generalisierung von Karten. Dabei wird die Geometrie eines Elements (z.B. eine Strasse) so angepasst, dass ein benachbartes Element (z.B. ein Fluss) ohne eine Überschneidung der Signaturen dargestellt werden kann.

#### **Platzbedarf der Routeninformationen**

Jede Anwendung einer Methode der Generalisierung (in diesem Fall: Verdrängung) ruft möglicherweise eine neue Konfliktsituation hervor. Die Generalisierung von Karten kann daher als Optimierungsproblem betrachtet werden (vgl. Lonergan und C. B. Jones 2001, Seite 287). Lonergan und C. B. Jones (2001) stellen in ihrem Artikel eine iterative Methode der Verdrängung vor. Dabei werden zuerst die schwierigsten Konflikte angegangen und anschliessend die daraus neu entstandenen Konflikte. Gleichzeitig werden Informationen zu den Änderungen gesammelt und ausgewertet, sodass eine optimierte Lösung gefunden werden kann.

Monnot, Hardy und D. Lee (2007) zeigen in ihrem Artikel eine Methode wie die Generalisierung einer Karte mit ArcGIS optimiert werden kann. Dabei wird ein Set von Regeln definiert, das anschliessend in einem "Optimizer-Kernel" für die Evaluierung der optimalen Lösung verwendet wird. Die generische Natur der Implementation erlaubt auch die Berücksichtigung von weiteren kombinatorischen Aspekten der kartographischen Darstellung. Dafür verwenden Monnot, Hardy und D. Lee (2007, Seite 9) ein zu dieser Arbeit passendes Beispiel<sup>8</sup>. Sie betrachten anhand der Metro Map von Paris die Aufgabe ein Bündel (Bundle) von Buslinien zu layouten, das einer gemeinsamen Richtungslinie folgt. Dabei soll die Kontinuität maximiert und Überkreuzungen minimiert werden, sodass der Anfang bzw. das Ende der Routen jeweils auf der Aussenseite des Bündels liegen (vgl. grüne und blaue Route in Abbildung 4.2b).

#### **Vermeidung von Überkreuzungen**

Das Problem der Vermeidung von Überkreuzungen wurde durch verschiedenen Autoren untersucht (vgl. Benkert u. a. 2007, Bekos u. a. 2008, Asquith, Gudmundsson und Merrick 2008, Nöllenburg 2010 und Fink und Pupyrev 2013<sup>9</sup>). Beispiele einer solchen Überkreuzung zeigen die Abbildung 3.5 (S2 und S7/S12 zwischen den Haltestellen Zürich HB und Hardbrücke) und die Abbildung 4.2a im Streckenabschnitt A. Bekos u. a. (2008) haben für dieses Problem den Begriff "Metro-Line Crossing Problem" (MLCM) eingeführt.

<sup>8</sup> "An example is laying out multiple bus routes which share a common centerline (as a bundle of offset lines) while maximizing continuity and minimizing crossings, and ensuring that start and end of routes lie at the outside of the route bundle." (Monnot, Hardy und D. Lee 2007, Seite 9)

<sup>9</sup> Aufzählung ist nicht abschliessend.

### 4.3. Automatisierte Bearbeitung von graphischen Elemente

Benkert u. a. (2007) haben sich als erstes der Behebung von Überkreuzungen angenommen. Dabei sollen die Routen entlang der Kanten eines Graphen (Streckennetz) so eingezeichnet werden, dass die Routenlinien so wenig wie möglich kreuzen. Benkert u. a. stellen dazu einen Ansatz vor, der die dynamische Programmierung<sup>10</sup> verwendet, um das Optimierungsproblem zu lösen. Der von Benkert u. a. diskutierte Ansatz funktioniert für eine Kante, kann aber nicht auf einen Pfad (Weg) angewendet werden. Dieses Problem greifen Bekos u. a. (2008) auf und zeigen einen Ansatz zur Lösung des MLCM-Problems entlang eines Pfads (MLCM-P) auf. Etwas später präsentieren Asquith, Gudmundsson und Merrick (2008) einen ILP<sup>11</sup>-Approach. Darauf basierend stellt Nöllenburg (2010) einen verbesserten Algorithmus und eine Zusammenfassung der verschiedenen Varianten des MLCM Problems vor (vgl. Nöllenburg 2010, Seite 385, Tabelle 1).

#### **Farbwahl der Routen**

Bereits im Kapitel 2.1.1 wurde erwähnt, dass bei der Verwendung des French Styles die einzelnen Farben der Routen maximal unterscheidbar sein sollten. Falls nun eine neue Route in einem bestehenden Liniennetzplan hinzukommt, muss eine Farbe gefunden werden, die sich soweit wie möglich von den bestehenden Farben abhebt. Kommen mehr als zwei neue Linien hinzu, wird es noch komplexer. In diesem Fall müssen die neuen Farben sich nicht nur von den bestehenden Routen, sondern auch voneinander abheben (vgl. Griffioen und Kiselev 2015, Seite 25). Griffioen und Kiselev (2015) analysieren in ihrem Artikel das Problem und präsentieren eine mathematische Strategie, um die besten Farben zu evaluieren.

#### **Visualisierung von Busrouten mit topographischen Darstellung**

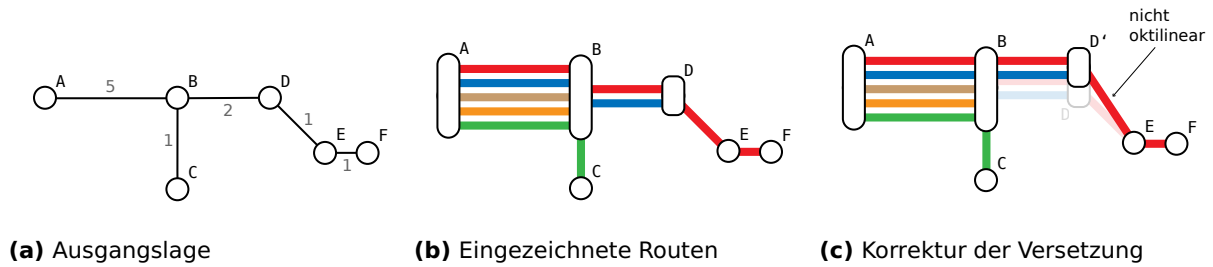
Sadahiro u. a. (2015) diskutieren die computerunterstützte Visualisierung von Busrouten und haben sechs ästhetische Kriterien identifiziert. Die Kriterien werden von Sadahiro u. a. in zwei Gruppen aufgeteilt. Die erste Gruppe beinhaltet Layout-Probleme, nämlich die Ausrichtung (Misalignment) entlang der Strasse, die Überlappung von Routeninformationen (Overlapping) und spitzwinklige Knick (Acute Bends). Bei einem Misalignment ist die Strasse nicht gleich der Mittellinie der dargestellten Routen. Überlappungen können auftreten, wenn zwei Strassen mit vielen Routen sehr nahe beieinander liegen, sodass sich die Darstellungen der Routen überschneiden. Spitzwinklige Winkel sind vor allem unästhetisch, falls mehrere Routen auf der Strecke verkehren.

Die zweite Gruppe enthält Kriterien, die sich mit dem Line Crossing Minimization Problem befassen oder in dessen Umfeld bewegen. Es handelt sich dabei um die Behebung

<sup>10</sup> Für eine Einführung in die dynamische Programmierung vgl. <https://people.eecs.berkeley.edu/~vazirani/algorithms/chap6.pdf> (6. August 2016).

<sup>11</sup> Inductive Logic Programming. Für eine Einführung vgl. [https://en.wikipedia.org/wiki/Inductive\\_logic\\_programming](https://en.wikipedia.org/wiki/Inductive_logic_programming) (6. August 2016).

### 4.3. Automatisierte Bearbeitung von graphischen Elementen



**Abbildung 4.3:** Korrekturproblem Versetzung. Die erste Abbildung zeigt die Ausgangslage. Der Kantenwert entspricht der Anzahl Routen auf dem Streckensegment. Werden die Routen wie in der mittleren Abbildung eingezeichnet, so entstehen Shifts (Versetzungen). Die Routen sind nicht kontinuierlich weitergeführt eingezeichnet. Um den Shift zu entfernen, muss der Mittelpunkt der Haltestelle D verschoben werden. Dabei entsteht jedoch ein nicht-oktilineares Streckensegment (vgl. Kante  $\overline{D'E}$  in der rechten Abbildung), welches korrigiert werden muss. (eigene Grafik)

von Lücken (Gaps, vgl. Abschnitt  $C_2$  in Abbildung 4.2a) zwischen Routen, die Versetzung (Shifts, vgl. Abschnitte B und  $C_1$  in Abbildung 4.2a) von Routen entlang mehrerer Streckensegmente ohne Abzweigungen und Knicke sowie von Überkreuzungen (Crossing, vgl. Abschnitt E in Abbildung 4.2a) von Routen.

Die von Sadahiro u. a. (2015) vorgestellte Methode geht diese Probleme unter der Verwendung einer topographischen Darstellung an. Dazu unterteilen Sadahiro u. a. die Methode in zwei Phasen. Die Line Layout Phase behandelt die Darstellung der Routen. Dabei sollen Gaps und Crossings reduziert werden. Da eine Wechselwirkung zwischen Gaps, Crossings und Shifts existiert, werden unter bestimmten Konditionen Versetzungen von Routenlinien akzeptiert. Die Map Layout Phase nimmt sich des Misalignments, der Überlappung und der spitzwinkligen Knicke an. Entlang der Streckensegmente werden die Routen eingezeichnet und anschliessend mit den Linien der gleichen Route des adjazenten Streckensegments verbunden. Dazu werden die Linien an den Enden jeweils verkürzt oder verlängert. Spitzwinklige Knicke werden behoben, indem zusätzliche Liniensegmente hinzugefügt werden. Die zusätzlichen Liniensegmente werden so eingefügt, dass der Abstand zwischen den Linien eingehalten wird. Überlappungen werden durch das Verschieben der Segmente behoben. Dabei werden die Segmente verschoben, die auf dem längeren Strassenabschnitt liegen. Dieser Ansatz behebt jedoch nicht alle Überlappungen, sodass eine manuelle Bearbeitung dieser Fälle nötig ist. Die Korrektur einer Überlappung kann auch zu einem neuen Misalignment führen. Sobald die Distanz zwischen dem Strassenabschnitt und dem nächsten Streckensegment grösser als ein vordefinierter Wert ist, werden die Misalignments ebenfalls durch das Verschieben der Endpunkte des Streckensegments behoben.

Abgesehen von der Behebung der Überlappung könnte die von Sadahiro u. a. (2015) vorgeschlagene Methode auch für schematische Karten verwendet werden. Aufgrund der Oktilinearität müsste jedoch die Behebung von Überlappungen anders angegangen wer-



### 4.3. Automatisierte Bearbeitung von graphischen Elementen

den, da unter Umständen nach jeder Verschiebung eines Streckensegments die oktilineare Ausrichtung der adjazenten Streckensegmente wiederhergestellt werden muss. Auch in einem schematischen Liniennetzplan kann ein Misalignment entstehen. Durch das Einzeichnen der Routen auf einem Streckensegment könnten unter Umständen adjazente Streckensegmente ihre oktilineare Ausrichtung verlieren, nämlich dann, wenn ein Streckensegment deutlich mehr eingehende Routen aufweist, als das fortlaufende Streckensegment (mit gleicher Ausrichtung) und der adjazente Knoten (vgl. Abbildung 4.3). Die Komplexität der Wiederherstellung der Oktilinearität ist daher abhängig von den jeweiligen adjazenten Streckensegmenten.

Die oben vorgestellten Lösungsansätze für die Darstellung der Linienführung können zum Teil automatisiert werden. Allerdings stehen diese Lösungsansätze jeweils für sich allein und sind nicht als Teil einer Werkzeugkette kombinierbar. Anzustreben wäre die Hintereinanderschaltung von Teillösungen zu einem automatisierteren Arbeitsprozess.

#### 4.3.3 Darstellung der Haltestellen und andere Verkehrsknoten

Haltestellen können auf verschiedene Weise dargestellt werden. Den grössten Einfluss auf die Darstellung hat die Anzahl der Routen, die einen Stopp an der Haltestelle haben. Von Bedeutung ist auch, in welchem Winkel die Routen zur Haltestelle führen. Ersteres beeinflusst vor allem die Dimension der Haltestelle, während die Anzahl der möglichen Winkel einen Einfluss auf die Form der Haltestelle haben kann.

Die Darstellung wird in der Regel vom Designer oder Kartograph bestimmt. Einige Darstellungsarten lassen sich besser automatisieren als andere. Ein Kreis oder ein Polygon als Punktsignatur kann an der Position der Haltestelle im Vordergrund dargestellt werden und überdeckt die Linien. Ein anderes Design stellt Haltestellen als Lücken (Gaps) dar, welche die Linien unterbrechen. Die Gaps sind transparent und zeigen den Hintergrund. Die Darstellung dieser Gaps ist einfacher zu automatisieren, wenn der Hintergrund einfarbig ist<sup>12</sup>. Falls aber ein mehrfarbiger Hintergrund vorhanden ist, müssen die Gaps explizit definiert werden.

Da beide notwendige Informationen, Anzahl Routen und Winkel der Routenlinien, bekannt sind, könnte die geometrische Form theoretisch berechnet werden. Mit einer Automatisierung könnte auch die Ausrichtung der Signatur berücksichtigt werden, beispielsweise bei der Verwendung von langgezogenen Rechtecken (vgl. Abbildung 2.8). Des Weiteren müssten Routen berücksichtigt werden können, die an der Haltestelle nicht halten, aber durchfahren. In der Regel wird die Haltestellen-Signatur vergrössert. Dies hat einen

---

<sup>12</sup> Bei einfarbigem Hintergrund könnte ein Symbol mit der selben Farbe verwendet werden.

### 4.3. Automatisierte Bearbeitung von graphischen Elemente

Einfluss auf die Beschriftung und die Lage von Points of Interest, sodass eine Neupositionierung der Beschriftung benötigt wird.

Andere Verkehrsknoten (Abbiegungen oder Kreuzungen), die keine Haltestelle darstellen, werden nicht visualisiert. Da die Routenlinien durch diese Punkte jedoch unterbrochen werden, müssten Linien der gleichen Route miteinander verbunden werden. Solche Verkehrsknoten können im Layout in Form eines abgerundeten Richtungswechsels ohne Unterbrechung dargestellt werden. Bei Kreuzungen ist zu definieren, welche Routen im Hintergrund und welche im Vordergrund darzustellen sind.

#### 4.3.4 Landmarken

##### Einfügen von Landmarken

Es gibt zwei Möglichkeiten Landmarken in einen Liniennetzplan einzufügen, nämlich die Berücksichtigung der Landmarken bei der (automatischen) Generierung der Karte oder das Einpflegen der Landmarken in der Nachbearbeitung. Das spätere Hinzufügen benötigt unter Umständen eine Anpassung des Layouts bzw. der Karte, wenn der Platzbedarf der Landmarke grösser als die vorhandene freie Fläche ist oder die Form unrealistisch verändert werden muss. Wird zusätzlich die Tatsache berücksichtigt, dass die Geometrie einer Landmarke in der Regel über Jahre unveränderlich ist, so spricht alles für ihre Berücksichtigung schon während der Generierung des Streckennetzes. In diesem Fall muss die Topologie der vereinfachten Fläche der Landmarke in Relation zu den Haltestellen sowie den Streckensegmenten eingehalten werden. Haltestellen müssen entweder innerhalb oder ausserhalb der Fläche platziert werden.

##### Berücksichtigung von Landmarken während der Generierung

Will man Landmarken bereits während der automatischen Generierung einer Karte berücksichtigen, eignen sich beispielsweise die Hill-Climbing-Algorithmen von Stott (2010) oder von Chivers (2014) mit einem Ausschlusskriterium zur Wahl der möglichen Knoten. Gleichzeitig müsste ein weiteres Bewertungskriterium hinzugefügt werden. Das neue Bewertungskriterium sollte topologische Fehler abstrafen, beispielsweise wenn ein Streckensegment einen See überquert (schneidet), obwohl das Streckensegment entlang des Sees führt.

#### 4.3.5 Spezielle Beschriftungen

##### Einfügen von Beschriftungen

Neben den Haltestellen und Routen können auch Landmarken und Regionen beschriftet werden. Labels von Regionen, z.B. eine Tarifzone, können ebenfalls als Landmarken

### 4.3. Automatisierte Bearbeitung von graphischen Elemente

behandelt werden, um eine Überschneidung zu verhindern. Falls aber eine Überschneidung möglich sein darf, jedoch vermieden werden soll, und beispielsweise ein Halo-Effekt verwendet wird, könnte die Abstrafung für das Überschneiden von Labels mit anderen Elementen reduziert werden.

Mit diesem Vorgehen kann sichergestellt werden, dass genug Raum für das Platzieren der Labels vorhanden ist. Geschieht das Einfügen in einem späteren Schritt, wie dies üblicherweise in topographischen Karten gemacht wird, wird mit einem Algorithmus basierend auf einem Regelset (und unter Berücksichtigung anderer Kartensignaturen und Labels) eine vorteilhafte Position des Labels ermittelt<sup>13</sup>.

Mit der Verwendung eines Map-Labeling-Algorithmus kann eine manuelle Anpassung des Layouts bzw. der Karte vermieden werden, beispielsweise, wenn zu einem späteren Zeitpunkt ein längerer Text benötigt wird. In der Nachbearbeitung ist aber sicherzustellen, dass andere Elemente nicht verdeckt und das Label selbst nicht von einem anderen Kartenelement überlagert wird.

#### **Automatisierung der nachträglichen Beschriftung**

Falls die Schriftgröße der weiteren Labels kein fixes Kriterium sein sollte, so könnten mit dem Algorithmus von Roessel (1989) mögliche rechteckige Flächen (Boxen) für das Einfügen eines Labels innerhalb von Polygonen gefunden werden. Diese Polygone werden durch Streckensegmente, Landmarken und Haltestellen (bzw. die Convex Hull der Punkt-signatur) definiert.

In einem weiteren Arbeitsschritt werden jeweils die rechteckigen Flächen (Boxen) mit der grössten Ausdehnung in allen Regionen ermittelt, die einem Labeltyp zugeordnet sind (z.B. Tarifzone). Mit diesen Boxen und dem jeweiligen Beschriftungstext kann nun die Schriftgröße ermittelt werden. Es muss die maximale Schriftgröße, welche für alle Texte in den jeweiligen Boxen angewendet werden kann, gefunden werden. Natürlich müssen weitere Kriterien und Prinzipien zur Platzierung von Labels berücksichtigt werden<sup>14</sup>. Sollte die Schriftgröße der Label im Voraus definiert sein, müsste das Layout vorgängig angepasst werden. Die Abstände zwischen den Kartenelementen müssten gross genug definiert werden, um eine Überlappung zu verhindern<sup>15</sup>.

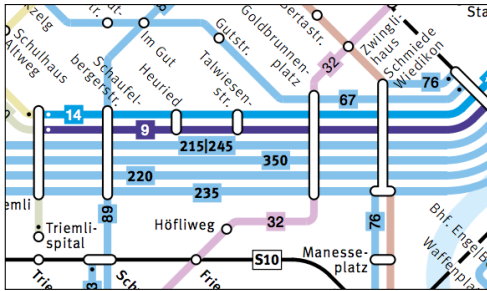
---

<sup>13</sup> Weiterführende Literatur zum Thema Map-Labeling wird in der Bibliographie von Prof. Dr. Alexander Wolff (Universität Würzburg) aufgeführt: <http://i11www.itl.uni-karlsruhe.de/~awolff/map-labeling/bibliography> (28. Juni 2016)

<sup>14</sup> Für eine Übersicht vgl. Wood 2000.

<sup>15</sup> Das Problem einer möglichen Überlappung stellt sich bei allen Kartenelementen, die automatisiert in das Layout eingefügt werden. Ein Lösungsansatz wird am Beispiel der Linienführung im nächsten Kapitel diskutiert.

#### 4.4. Zusammenfassung und Relevanz der Ansätze



**Abbildung 4.4:** Die Birmensdorferstrasse in Zürich wird von sehr vielen Bus- und Tramlinien befahren. Vergrößerter Ausschnitt aus der Abbildung 2.7. (© ZVV/VBZ)

### 4.4 Zusammenfassung und Relevanz der Ansätze

Verschiedene Fachleute haben sich bereits mit der automatischen Generierung von Metro Maps befasst und Ansätze einer Automatisierung entwickelt. In der Regel handelt es sich um Algorithmen, die von einem initialen Datensatz ausgehen und das Layout der Metro Map unter Berücksichtigung der Layout-Regeln generieren können. Alle Ansätze fokussieren sich auf das Layout und die Einhaltung der Layout Regeln.

Lediglich die Methoden von Stott und Nöllenburg (sowie deren Modifikationen) produzieren aus Sicht des Autors ein funktionales und ansprechendes Layout. Die Layouts der anderen Algorithmen bedürfen einer manuellen Korrektur. Als besonders nützlich erweist sich der Algorithmus von Stott (2008), der von Chivers (2014) übernommen und optimiert wurde. Ein qualitativ besseres Layout liefert der Ansatz von Nöllenburg und Wolff (2011), welcher die Layout Regeln in Hard- und Soft-Constraints unterteilt, wobei die Hard-Constraints unbedingt und die Soft-Constraints annähernd erfüllt sein müssen. Der Ansatz von Wang und Peng (2015) stellt einen weiteren Fortschritt dar, da er eine Nachbearbeitung des ausgegebenen Layouts erlaubt. Sie erweitern die Automatisierung der Arbeitsschritte zur Erstellung eines Liniennetzplanes, indem sie weitergehende Werkzeuge (in Form einer Benutzeroberfläche) zur Verfügung stellen. Allerdings können auch sie in ihrem System keine Landmarken berücksichtigen und die graphische Darstellung von mehreren Routen in einem Streckensegment gelingt nicht zufriedenstellend, denn die Liniensignaturen der Routen sind, abgesehen von der Farbe, nicht formatiert und kleben aneinander.

Alle diese Ansätze können einem einfachen Netzwerk (z.B. ein reines U-Bahnnetz ohne parallel geführte Routen) genügen, nicht jedoch einem dichten Netzwerk wie dem der Verkehrsbetriebe der Stadt Zürich, wo z.B. der Goldbrunnenplatz von acht Bus- und Tramrouten bedient wird (vgl. Abbildung 4.4). Weder die Signaturen der Routen (Breite, Farbe), noch deren Platzbedarf auf der Karte werden durch die oben erwähnten Algorithmen berücksichtigt.

Sadahiro u. a. (2015) beschäftigen sich mit der Darstellung der Routen in einem topographischen Liniennetzplan für Busse. Sie untersuchen Probleme der Darstellung paralleler

#### 4.4. Zusammenfassung und Relevanz der Ansätze

Routen (z.B. Layout-Probleme und Überlappungen) und präsentieren Vorschläge zur Lösung. Sie berücksichtigen die Oktilinearität nicht, ihre Vorschläge können aber auch für ein oktilineares Layout relevant sein.

Weitere graphische Aspekte wie die Berücksichtigung von Landmarken und weiteren topologische Gegebenheiten werden für die Generierung ebenfalls nicht einbezogen. Der Liniennetzplan der Stadt Zürich enthält zum Beispiel als Landmarken die Flüsse Sihl und Limmat sowie den Zürichsee und die grösseren Waldgebiete von Zürich. Des Weiteren wird eine vereinfachte Form der Stadtgrenze dargestellt. Dabei wird die Topologie zwischen Haltestellen, Landmarken und Stadtgebiet eingehalten. Dementsprechend ist der Bahnhof Schlieren ausserhalb der Stadtgrenze dargestellt, während der Bahnhof Stettbach auf der Stadtgrenze visualisiert ist, was der Topologie entspricht.

Verschiedene Autoren beschäftigen sich mit dem Problem der Überkreuzungen von Routen (MLCM) und präsentieren Vorschläge wie die Zahl der Überkreuzungen minimiert werden kann. Welche weiteren Möglichkeiten der Automatisierung von Arbeitsschritten bestehen, soll im folgenden Kapitel am Beispiel des Platzbedarfs von Liniensignaturen aufgezeigt werden.

## 5 | Automatisierung bei der Darstellung der Linienführung

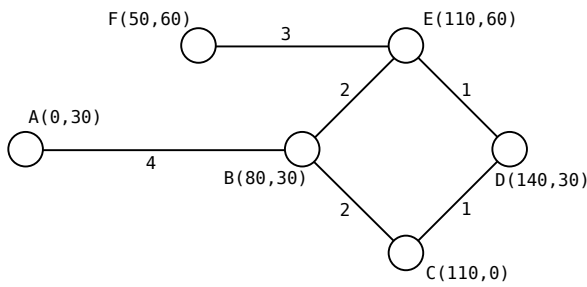
In diesem Kapitel sollen Möglichkeiten der Automatisierung bei der Nachbearbeitung von Liniennetzplänen aufgezeigt werden. Beim nachträglichen Einfügen von Kartenelementen tritt häufig das Problem einer Überlappung (Überschneidungen von Signaturen aufgrund des neuen Platzbedarfs) auf, die nur durch eine Anpassung des gesamten Layouts behoben werden kann. Welche Möglichkeiten es gibt, diese komplexe Aufgabe zumindest teilweise zu automatisieren, soll im Folgenden am Beispiel der Linienführung diskutiert werden. Ausgangspunkt ist eine bereits vorhandene schematische Darstellung eines Streckennetzes (z.B. Abbildung 5.1a) sowie ein Datensatz der die Routenführung auf einzelnen Streckensegmenten enthält. Die Aufgabe ist es im Streckennetz verschiedene Routen einzufügen. Dies hat eine Verdickung der Streckensegmente durch das Einfügen mehrerer parallelen Linien und teilweise auch eine vergrößerte Darstellung der Haltestellen-Symbole zur Folge (vgl. Abbildung 5.1b). Dadurch müssen unter Umständen andere Kartenelemente verschoben (verdrängt) werden, was sich auf Teile des Layouts oder womöglich auf das ganze Layout auswirkt. Statt sämtliche Verschiebungsmöglichkeiten manuell durchzutesten, soll eine weitgehend automatisierte Vorgehensweise präsentiert werden.

### 5.1 Konflikte ermitteln

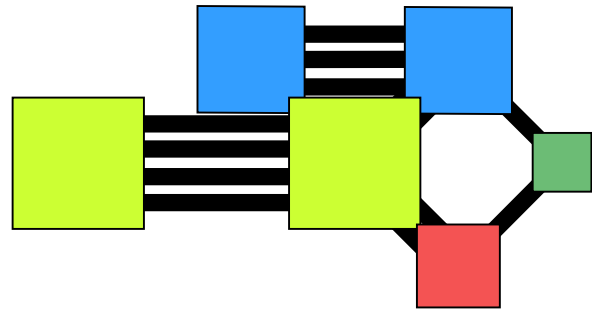
Ausgehend von der Information, wieviele Routen auf einem Streckensegment eingezeichnet werden müssen, kann automatisch ermittelt werden, wo Konflikte (Überschneidungen) vorhanden sind. In einem zweiten Schritt sollen die Konflikte behoben werden. Konflikte werden im Folgenden durch Konfliktpolygone dargestellt.

#### 5.1.1 Konflikte automatisch bestimmen

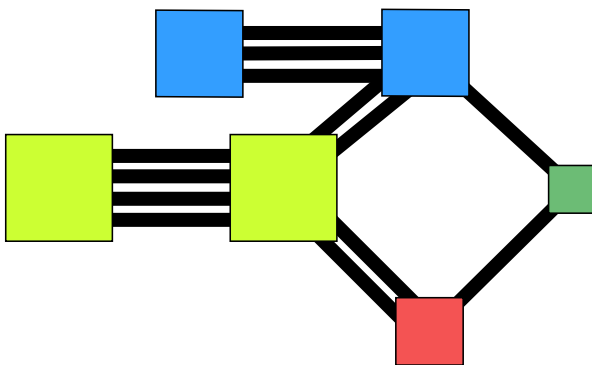
Für das Bestimmen der Konfliktpolygone werden zunächst alle Streckensegmente benötigt, wobei  $e$  ein einzelnes Streckensegment ist. Zusätzlich müssen die Routeninformationen der einzelnen Streckensegmente beigezogen werden. So repräsentiert  $l_e$  eine Route und  $L_e$  alle Routen entlang des Streckensegmentes  $e$ . Die Linienstärke der Route  $l$  ist  $w_l$ . So können beispielsweise verschiedene Verkehrsmittel durch verschiedene Linienstärken repräsentiert werden. Sind diese Informationen vorhanden, können die Konfliktpolygone mit Hilfe der im Folgenden dargestellten Arbeitsschritte automatisch bestimmt werden.



(a) Ausgangslage

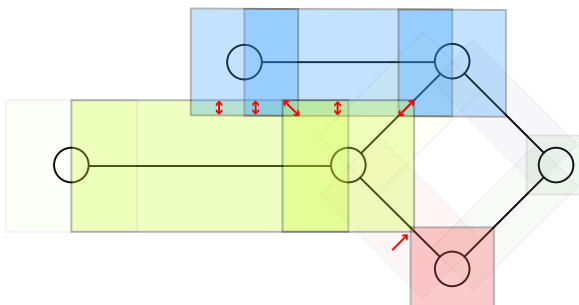


(b) Fehlender Raum für Routen

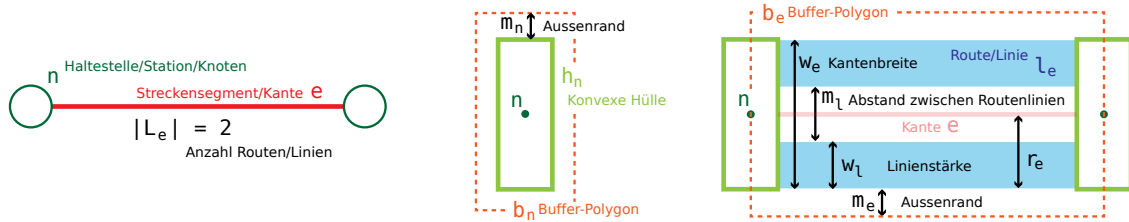


(c) Erwartetes Ergebnis

**Abbildung 5.1:** Problemstellung der Überlappung von Routeninformationen. Werte des Beispiels: Die Anzahl Routen  $|L_e|$  entspricht der Wertigkeit der Kante, die Linienstärke  $w_l$  sowie der minimale Abstand  $m_e$  zur gegenüberliegenden Haltestelle oder Streckensegment entsprechen je 25px, es gilt  $m_n = m_e$  und der Abstand zwischen den Routenlinien  $m_l$  ist 1px. Die Haltestelle werden als Quadrate dargestellt, wobei die Seitenlänge des Quadrats dem maximalen Wert von  $w_e$  aller adjazenten Kanten entspricht. (eigene Grafik)



**Abbildung 5.2:** Konfliktpolygone die entstehen, wenn Haltestellen und Streckensegmenten zu wenig Raum erhalten. Die umrandeten Rechtecke sind Buffer-Polygone aller Haltestellen und Streckensegmente. (eigene Grafik)



**Abbildung 5.3:** Zeichenerklärung für das Kapitel 5.1.1. (eigene Grafik)

In der Regel sind die einzelnen Routenlinien mit einem einheitlichen Abstand  $m_l$  (Margin) voneinander getrennt. Ein weiterer Abstand  $m_e$  soll sicherstellen, dass zwei Streckensegmente (oder auch Haltestellen) genug Abstand zueinander haben. Durch das Summieren aller Linienstärken sowie die Summe der Margins wird die maximale Ausdehnung  $w_e$  berechnet. Die Berechnung ist in der Gleichung 5.1 dargestellt.

$$w_e = \left( \sum_{l_e \in L_e} w_{l_e} \right) + (m_l * (|L_e| - 2) + m_e * 2) \quad (5.1)$$

Nachdem die Ausdehnung der Streckensegmente in die Breite berechnet wurde, werden nun Buffer-Polygone aller Streckensegmente benötigt. Für die Erstellung des Buffers wird die Distanz  $r_e = \frac{w_e}{2}$  verwendet, wobei die Enden der Buffer jeweils flach sind. Dazu soll die Funktion `createFlatBuffer(e, r_e)` verwendet werden.

Ebenso wird die Ausdehnung der Punktsignatur der Haltestelle  $n$  benötigt. Die Berechnung der Ausdehnung hängt von der Zielform des Symbols bzw. der Signatur ab. Mit einer Funktion `createStationConvexHull(n)` wird die konvexe Hülle  $h_n$  der Signatur  $n$  erstellt. Eine konvexe Hülle entspricht dem kleinsten konvexen Polygon, welches alle Punkte der Signatur beinhaltet. Anschliessend wird wieder mit der Funktion `createFlatBuffer(h_n, m_n)` ein Buffer  $b_n$  mit der Distanz  $m_e$  erzeugt. Der zusätzliche Buffer ist notwendig, damit ein Abstand  $m_n$  zwischen den Rändern der Haltestellen und Streckensegmente eingehalten werden kann<sup>1</sup>.

Zur Identifizierung der Konfliktpolygone werden nun alle Kombinationen zwischen den Buffer-Polygonen aller Kanten und Knoten getestet. Die Menge aller Buffer-Polygone wird durch das Set  $B = \{b_{e_1}, \dots, b_{e_{n-1}}, b_{e_n}, b_{n_1}, \dots, b_{n_{n-1}}, b_{n_n}\}$  repräsentiert. Dazu werden Kombinationen mit jeweils zwei Elementen erstellt. Paare mit adjazenten Knoten/Kante- und Kante/Kante-Kombinationen werden später jedoch ignoriert und nicht überprüft. Für das Erstellen der Kombination wird die Funktion `createBufferPairs(B)` verwendet.

<sup>1</sup> Der Abstand  $m_n$  ist in der Regel identisch mit  $m_e$ . Zu beachten gilt, dass der Abstand zwischen einem Streckensegment und einer Haltestelle die Summe der beiden Margins  $m_e$  und  $m_n$  beträgt.



Für jedes aus zwei Buffer-Polygonen ( $b_1$  und  $b_2$ ) bestehende Paar (Pair) wird nun die räumliche Relation überprüft. Es soll mit dem standardisiertem DE-9IM Modell<sup>2</sup> herausgefunden werden, ob sich die Innenflächen (Interior) der beiden Polygone überschneiden. Die maximale Anzahl der Dimensionen der Schnittmenge beider Polygon-(Innen)flächen muss 2 ergeben (vgl. Gleichung 5.2 und Strobl 2008, Seite 241 f.).

$$\dim(I(b_1) \cap I(b_2)) = 2 \quad (5.2)$$

Die Funktion  $\dim(\text{geom})$  berechnet die maximale Anzahl der Dimensionen<sup>3</sup> einer Geometrie  $\text{geom}$ , während die Funktion  $I(\text{geom})$  das Interior (Innenfläche) der Geometrie  $\text{geom}$  zurückgibt.

Anschliessend wird für jedes Polygon-Paar, auf welche die oben genannten Eigenschaften zutreffen, eine Clip-Operation durchgeführt. Das Resultat einer Clip-Operation ist ein Polygon, welches alle gemeinsame Punkte der beiden Polygone beinhaltet, und entspricht somit einem Konfliktpolygon. Die Clip-Operation wird mit der Funktion  $\text{clip}(b_1, b_2)$  durchgeführt, wobei  $b_1$  und  $b_2$  für die beiden Buffer-Polygone stehen.

Nachfolgend werden die Schritte zur Erstellung der Konfliktpolygone zusammengefasst und im Algorithmus 1 mit Pseudo Code dargestellt:

**Schritt 1** Erstelle ein Set B.

**Schritt 2** Für jedes Streckensegment:

- Berechne  $w_e = (\sum_{l_e \in L_e} w_{l_e}) + (m_l * (|L_e| - 2) + m_e * 2)$ .
- Erstelle einen Buffer mit der Distanz  $r_e$  und füge ihn dem Set B hinzu.

**Schritt 3** Für jede Haltestelle:

- Erstelle eine konvexe Hülle der Signatur bzw. des Symbols der Haltestelle.
- Erstelle einen Buffer mit der Distanz  $m_n$  und füge ihn dem Set B hinzu.

**Schritt 4** Erstelle Buffer-Paare aus dem Set B.

**Schritt 5** Für jedes Buffer-Paar im Set B:

- falls* keine adjazente Kante/Kante- oder Kante/Knoten-Kombination und sich die Innenfläche der Buffer-Polygone schneiden
- dann* erstelle ein Polygon der gemeinsamen Fläche

<sup>2</sup> Für eine Einführung in das DE-9IM Modell vgl. Egenhofer und Franzosa 1991, Clementini, Sharma und Egenhofer 1994, Clementini und Di Felice 1995 und Strobl 2008.

<sup>3</sup> Für eine Übersicht der Dimensionen vgl. Strobl 2008 (Seite 242).

---

**Algorithmus 1** Konfliktzonen bestimmen

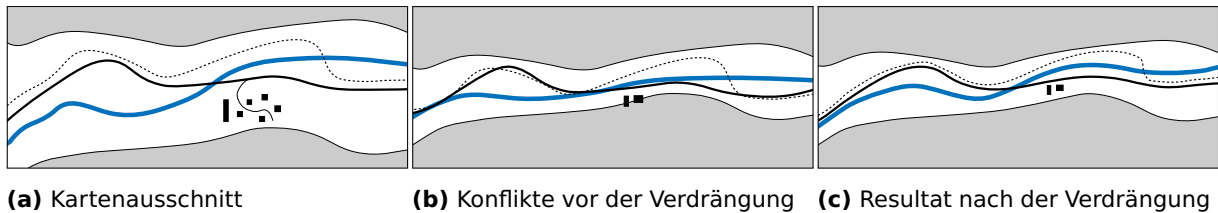
---

```

1 procedure getConflictPolygons(edges, nodes, routeMargin, edgeMargin, nodeMargin)
2
3   buffers = new Set()
4
5   for all edge in edges do
6     edgeWidth = 0
7     routes = edge.getRoutes()
8     for all route in routes do
9       edgeWidth = edgeWidth + route.getLineWidth();
10    end for
11    edgeWidth = edgeWidth + routeMargin * (routes.count() - 2) + edgeMargin * 2
12    buffer = createFlatBuffer(edge, edgeWidth / 2)
13    buffers.add(buffer)
14  end for
15
16  for all node in nodes do
17    stationConvexHull = createStationConvexHull(node)
18    buffer = createFlatBuffer(stationConvexHull, nodeMargin)
19    buffers.add(buffer)
20  end for
21
22  pairs = createBufferPairs(buffers)
23  conflictPolygons = new Set()
24
25  for all {buffer1, buffer2} in pairs do
26    if element of buffer1 is adjacent to element of buffer2 then
27      if one of the elements of {buffer1, buffer2} is not a node then
28        continue with next pair
29      end if
30    end if
31    if interior of buffer1 intersects with interior of buffer2 then
32      polygon = clip(buffer1, buffer2)
33      conflictPolygons.add(polygon)
34    end if
35  end for
36
37  return conflictPolygons
38
39 end procedure

```

---



**Abbildung 5.4:** Anwendung einer Verdrängung in einer Talebene. Der Kartenausschnitt aus der ersten Abbildung ist vergrößert. In der zweiten Abbildung wird er verkleinert dargestellt, was Konflikte erzeugt. In der letzten Abbildung wird die Talebene leicht verbreitert (Übertreibung), damit die Häuser dargestellt werden können und die Linien werden abgesetzt voneinander dargestellt (Verdrängung). (eigene Grafik)

## 5.2 Ansätze zur automatisierten Verdrängung

Nachdem die Konfliktzonen bestimmt sind, sollen Korrekturen vorgenommen werden, welche die Konflikte beheben. Dabei muss der Generalisierung von Kartenelementen, insbesondere dem Verfahren der Verdrängung, Aufmerksamkeit geschenkt werden. Es soll aufgezeigt werden, wie sich die Verdrängung auf die Darstellungsweise in der Kartographie auswirkt und welche speziellen Charakteristika die Verdrängung in einem oktilinearen Layout aufweist. Schliesslich sollen Lösungsansätze aufgezeigt werden, wie das Verfahren automatisiert werden kann.

### 5.2.1 Ansätze zur Verdrängung in der Kartographie

Die Verdrängung in der Kartographie ist ein Verfahren zur Generalisierung von räumlich dargestellten Informationen<sup>4</sup>. Benötigt ein Kartenelement mehr Raum für eine vollständige Darstellung der Signatur, so werden benachbarte Kartenelemente von ihrer ursprünglich geometrisch korrekten Lage verdrängt.

Im gezeigten Beispiel kann die Darstellung von Fluss, Strasse und Bahnlinie in einer Talebene nicht ohne Überlappung realisiert werden (vgl. Abbildung 5.4). Als Lösung können die Talebene verbreitert (Übertreibung<sup>5</sup>) und im dadurch entstanden Raum die drei Liniensignaturen voneinander abgesetzt dargestellt werden (vgl. Tyner 2014, Seite 86 f.).

Bereits seit den frühen Jahren der elektronischen Datenverarbeitung wird das Themengebiet der automatisierten Generalisierung von Karten untersucht. Brassel und Weibel (1988) geben einen Überblick der Arbeiten bis in die späten 1980er Jahre. Verschiedene Herangehensweisen für eine automatische Verdrängung von Objekten wurden seit dem Review von Brassel und Weibel diskutiert. So verwenden C. B. Jones, Bundy und J. M. Ware (1995) beispielsweise eine modifizierten Delaunay-Triangulierung zur Identifizierung und Behebung von Konflikten. Die Dreiecke werden mit den Eckpunkten der Polygone gebildet (vgl. Bader 2001, Seite 21). Die Verdrängung wird anschliessend durch Verschieben

<sup>4</sup> Vgl. Schweizerische Gesellschaft für Kartographie Arbeitsgruppe 1975 und Tyner 2014 (Seite 82 ff.).

<sup>5</sup> Ein weiteres Verfahren zur Generalisierung von Karteninformationen.

der Polygone erreicht und behandelt Konflikte lokal und sequentiell. Einen anderen Ansatz präsentieren Mackaness und Purves (2001) mit einem Algorithmus, der einem Hill Climbing-Algorithmus gleicht. Dabei werden iterativ Positionen in der Umgebung der Polygone geprüft und bewertet. Sobald eine bessere Position gefunden wird, wird das Polygon verschoben. Das Vorgehen wird in mehreren Iterationen wiederholt. Es werden unter Umständen auch Polygone verschoben, die keine Verdrängung benötigen würden. Die Methode von Mackaness und Purves (2001) bewertet die Lösung global unter Berücksichtigung aller Polygone und nicht einzelne, lokale Korrekturen wie dies bei C. B. Jones, Bundy und J. M. Ware (1995) der Fall ist. Weitere Ansätze<sup>6</sup> basieren u.a. auf dem Prinzip des Simulated Annealing (vgl. Lonergan und C. B. Jones 2001; J. M. Ware, Christopher B. Jones und Thomas 2003), einem evolutionären Algorithmus<sup>7</sup> (vgl. Wilson, J. M. Ware und J. A. Ware 2003) oder einem Tabu-Search-Approach<sup>8</sup> (vgl. J. M. Ware, Wilson u. a. 2002).

Die Verdrängung in der Kartographie ist in der Regel das Produkt des Zusammenspiels verschiedener Generalisierungsoperationen. Dazu gehören beispielsweise das Vereinfachen, Weglassen, Übertreiben oder Verschmelzen von Kartenelementen. Ein Liniennetzplan stellt insofern einen Sonderfall dar, als der Plan bereits sehr stark generalisiert ist. Wenn in einem Liniennetzplan Konflikte (Überlappungen) existieren, bleibt meist nur die Verdrängung übrig, um Raum für die korrigierte Darstellung zu schaffen.

### 5.2.2 Verdrängung in einem oktilinearen Layout

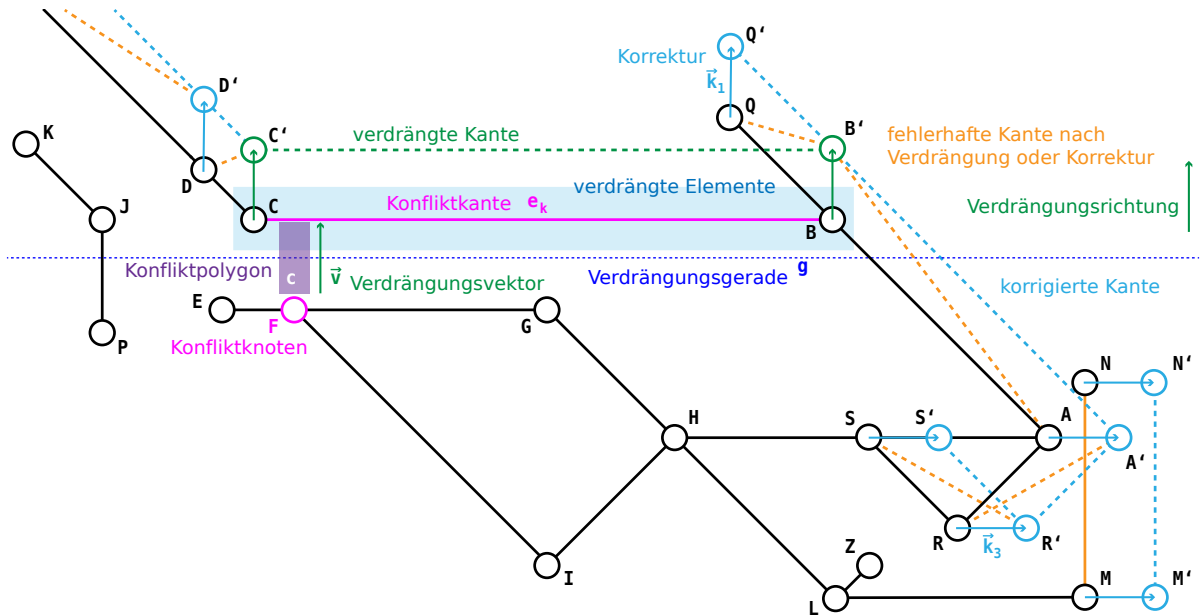
Die Anwendung einer Verdrängung in einem oktilinearen Layout wirft das Problem auf, dass die Winkeltreue nicht eingehalten werden kann. Die Verdrängung einer Kante oder eines Knoten wirkt sich auf alle Kanten (und Knoten) des Graphen aus (vgl. Abbildung 5.5a), sodass unter Umständen eine Korrektur der Oktilinearität betroffener Kanten notwendig ist. Dabei verändert sich jedoch auch die Topologie zwischen den Knoten verschiedener Komponenten (vgl. Kante  $\overline{JK}$  in Abbildung 5.5a) oder es kann unter Umständen zu einer Überlappung mehrerer Komponenten kommen. In solchen Fällen kann es sinnvoll sein, nicht nur einzelne Konfliktelemente, sondern eine ganze Seite entlang einer Verdrängungsgeraden  $g$  (eine Hilfslinie, welche den gesamten Netzwerkplan unterteilt) zu verschieben (vgl. Abbildung 5.5b).

Dabei wird jedoch noch nicht berücksichtigt, dass die Kanten des betroffenen Teilgraphen adjazente Knoten aufweisen können, welche auf beiden Seiten der Verdrängungsgeraden liegen (vgl. Kante  $e$  und  $\overline{PJ}$  in Abbildung 5.5b). Solche Kanten können im besten Fall

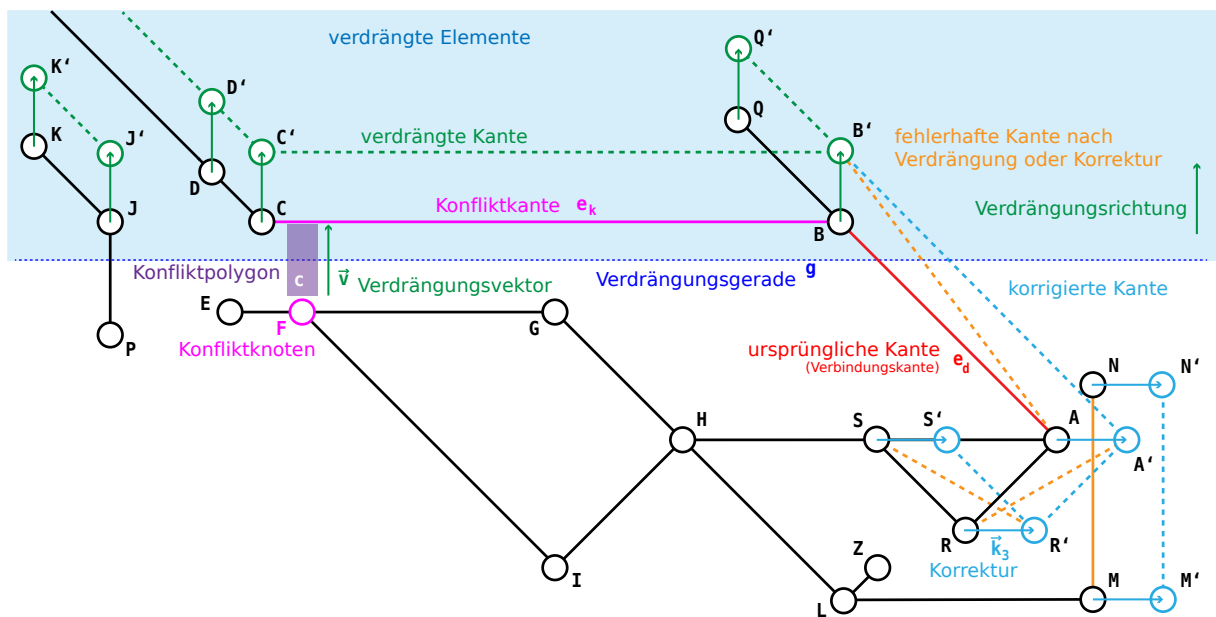
<sup>6</sup> Die Aufzählung ist nicht abschliessend. Für weitere Ansätze (bis 2001) vgl. Bader 2001 (Kapitel 2.5).

<sup>7</sup> Für eine Einführung in evolutionäre Algorithmen vgl. Jacobson und Kanber 2015.

<sup>8</sup> Für eine Einführung in Tabu Search vgl. [http://home.ifi.uio.no/infheur/Bakgrunn/Intro\\_to\\_TS\\_Gendreau.htm](http://home.ifi.uio.no/infheur/Bakgrunn/Intro_to_TS_Gendreau.htm) (16. Juli 2016).



(a) Verdrängung einzelner Konfliktelemente



(b) Verdrängung einer ganzen Seite der Verdrängungsgerade

**Abbildung 5.5:** Verdrängung im oktilinearen Layout. Kanten, die nach der Verdrängung eine Korrektur benötigen, sind orange dargestellt. (eigene Grafik)

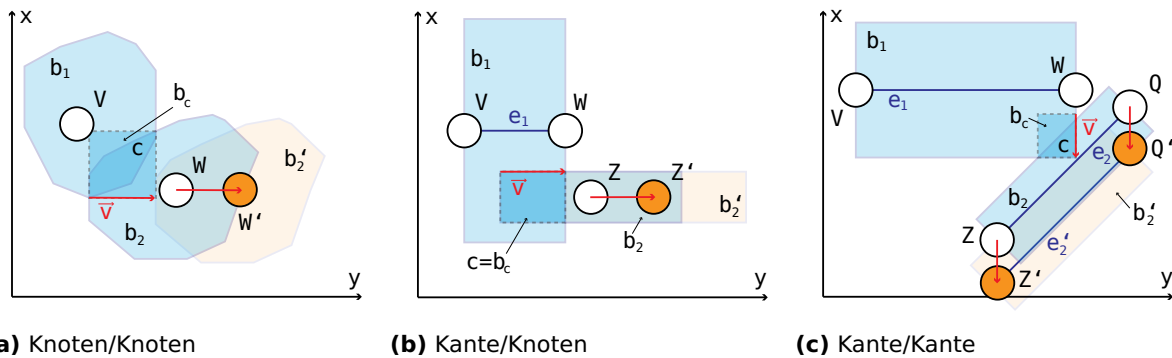
einfach verlängert werden, wenn sie die Verdrängungsgerade im 90 Grad Winkel schneiden oder berühren (vgl. Kante  $\overline{P\bar{J}}$  in Abbildung 5.5b). Schneidet oder berührt eine Kante aber mit einem 45 Grad Winkel, so wird sich deren Winkel zwangsläufig verändern, wenn einer der adjazenten Knoten verschoben wird (vgl. Kante  $e$  in Abbildung 5.5b). Es wird eine Korrektur der Verbindungskante  $e$  benötigt. Der Knoten  $A$  (auf der anderen Seite der Verdrängungsgerade) muss verschoben werden, um die Oktilinearität wiederherzustellen. Dazu wird ein Vektor  $\vec{k}$  benötigt, wobei die Länge in der Regel dem Vektor  $|\vec{v}|$  oder einem Vielfachen von  $|\vec{v}|$  entspricht. Dies kann zu weiteren nicht-oktilinearen Ausrichtungen von adjazenten Kanten führen (vgl. Kante  $\overline{A\bar{R}}$  in Abbildung 5.5b bevor  $R$  um  $\vec{k}$  verschoben wurde), welche iterativ korrigiert werden müssen. Eine solche Korrektur kann neue Konflikte erzeugen (vgl. Kante  $\overline{M\bar{N}}$  in Abbildung 5.5b).

Das Konfliktpolygon in der Abbildung 5.5a bzw. 5.5b entsteht durch den Knoten  $F$  (Haltestelle) und die Kanten  $e_k$  (Streckensegment), welche jeweils mehr Raum für ihre Darstellung benötigen und sich deshalb auf der Fläche des Konfliktpolygons  $c$  schneiden. Der Verdrängungsvektor  $\vec{v}$  repräsentiert den zusätzlichen Platzbedarf und die Richtung der Verdrängung. Die Verdrängung kann jedoch auch in die entgegengesetzte Richtung durch die Verwendung von  $-\vec{v}$  durchgeführt werden.

Neben den Winkeln werden unter Umständen auch die Topologie sowie die Distanzen zwischen Haltestellen verändert (vgl. Kante  $\overline{H\bar{S}'}$  in Abbildung 5.5b). Eine Veränderung der Topologie ist in der Abbildung 5.5b anhand der Knoten  $Z$  und  $S$  bzw.  $S'$  zu sehen. Die relativen Positionen der beiden Knoten zueinander werden zusätzlich<sup>9</sup> verändert. Die Frage stellt sich nun, ob eine Methode existiert, welche die Änderungen an Topologie und Distanzen so klein wie möglich hält, während das resultierende Layout gleichzeitig winkeltreu zum ursprünglichen Layout bleibt. Ein möglicher Ansatz wäre die Skalierung der Positionen von Haltestellen, wobei das Streckenverhältnis und die Winkel nicht verändert werden. Dies erhält die Topologie, lässt jedoch Randregionen grösser erscheinen.

In den folgenden Abschnitten soll das Vorgehen zur Ermittlung der Verdrängungsvektoren diskutiert werden und es wird geprüft, welche Konflikte zufriedenstellend mit einer Skalierung gelöst werden können. Im weiteren wird eine Methode präsentiert, welche Konflikte basierend auf Verdrängungsgeraden und Verdrängungsvektoren lösen soll. Für die Korrekturen ist ein Set von Regeln zu definieren, welches sicherstellen soll, dass Änderungen an Winkeln und Topologie so weit wie möglich vermieden werden.

<sup>9</sup> Die relative Genauigkeit der Lage zwischen den Knoten wurde unter Umständen bereits während des Zeichnens oder der Generierung des Streckennetzes verzerrt.



**Abbildung 5.6:** Beispiele für orthogonale Verdrängungsvektoren. Die blaue Fläche zeigt jeweils den Platzbedarf der jeweiligen Signaturen und die orange Fläche den neuen Buffer des verschobenen Elements. (eigene Grafik)

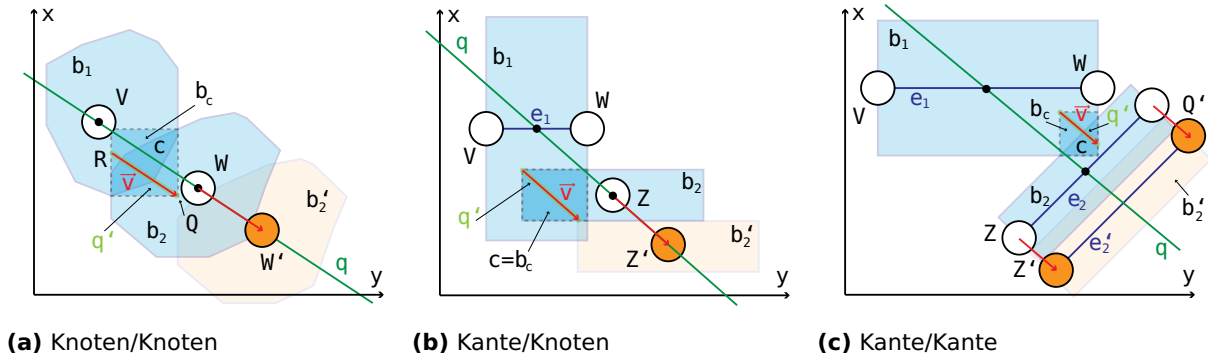
### 5.3 Verdrängungsvektoren ermitteln

Konfliktpolygone werden wie im Kapitel 5.1 beschrieben ermittelt. Basierend auf den Polygonen und den jeweiligen Konfliktelementen muss nun der entsprechende Verdrängungsvektor  $\vec{v}$  gefunden werden.

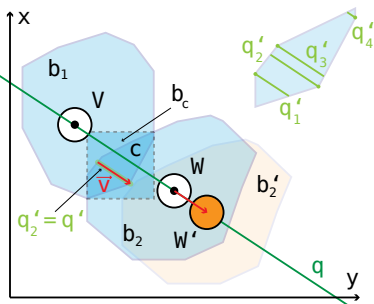
Der einfachste Ansatz ist die Erstellung einer Bounding Box für jedes Konfliktpolygon  $c$ . Eine Bounding Box steht parallel zur X-Achse und ist das kleinste Rechteck, welches das Konfliktpolygon einschliessen kann. Die kleinere Seite der Bounding Box  $b_c$  definiert dann den Vektor  $\vec{v}$  (vgl. Abbildung 5.6a und 5.6c). In einem Ausnahmefall kann jedoch nicht die kleinste Seite der Bounding Box zur Bestimmung des Vektors  $\vec{v}$  verwendet werden. Liegen nämlich drei Seiten der Bounding Box innerhalb einer der beiden konfliktverursachenden Signaturen (Konfliktbuffers,  $b_1$  und  $b_2$ ), müsste diejenige Seite den Vektor  $\vec{v}$  definieren, deren gegenüberliegende Seite sich ebenfalls innerhalb des Konfliktbuffers befindet (vgl. Abbildung 5.6b).

Das oben beschriebene Vorgehen verändert jedoch die relative Position zwischen den beiden Elementen, wie in der Abbildung 5.6b dargestellt. Die Position von  $Z'$  ist neu mehr östlich als südlich von  $W$  zu finden. Um die relativen Positionen zwischen den Elementen zu erhalten, dürfte die Verdrängung nicht auf orthogonale Richtungen eingeschränkt werden.

Ein einfacher Lösungsweg wäre die Verwendung einer Diagonale der Bounding Box  $b_c$ , die den Vektor  $\vec{v}$  definiert. Die Verwendung einer Diagonale zur Ermittlung des Vektors  $\vec{v}$  schwächt zwar eine mögliche Veränderung an den relativen Positionen, verursacht jedoch unter Umständen eine grössere Verdrängung als notwendig. Es muss daher eine andere Methode gefunden werden, um einen besseren Verdrängungsvektor zu ermitteln. Falls beide Konfliktelemente Knoten sind, bietet sich die Gerade  $q$  an, die durch die bei-



**Abbildung 5.7:** Beispiele für nicht-orthogonale Verdrängungsvektoren. Die blaue Fläche zeigt jeweils den Platzbedarf der jeweiligen Signaturen und die orange Fläche den neuen Buffer des verschobenen Elements. Die Situation in der Abbildung 5.7b benötigt eine zweite Iteration, um den Konflikt zu beheben. (eigene Grafik)



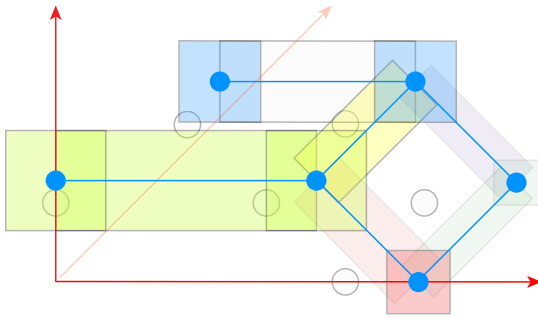
**Abbildung 5.8:** Verdrängungsvektoren mit Konfliktpolygon ermittelt. Es gibt vier parallele Linien  $q'_1$ ,  $q'_2$ ,  $q'_3$  und  $q'_4$  (oben rechts vergrößert dargestellt) die an einem Eckpunkt des Polygons beginnen und auf einer gegenüberliegenden Seite enden, wobei  $q'_2$  die längste Linie ist und somit den Vektor  $\vec{v}$  definiert. Die blaue Fläche zeigt wiederum den Platzbedarf der jeweiligen Signaturen und die orange Fläche den neuen Buffer des verschobenen Elements. (eigene Grafik)

den Punkte V und W gelegt wird. Nun wird eine parallele Linie  $q'$  mit der grösstmöglichen Länge innerhalb des Konfliktpolygons  $c$  gesucht. Die Endpunkte der Linie  $q'$  sind R und Q. Zur Vereinfachung wird anstelle des Konfliktpolygons  $c$  dessen Bounding Box  $b_c$  verwendet. Dadurch wird das Ergebnis zwar leicht verfälscht, erlaubt aber eine einfachere Findung der Linie  $q'$  indem R oder Q auf einem Eckpunkt der Bounding Box liegen. Der Vektor  $\vec{v}$  wird nun mit der Linie  $q'$  definiert (vgl. Abbildung 5.7a). Falls es sich jedoch bei mindestens einem Element um eine Kante handelt, so wird der Mittelpunkt der Kante für die Definition der Gerade  $q$  verwendet (vgl. Abbildung 5.7b und 5.7c).

Anstelle der Bounding Box  $b_c$  könnte auch das Konfliktpolygon  $c$  verwendet werden. Bei einem konvexen<sup>10</sup> Polygon ist die gesuchte Linie  $q'$  ebenfalls parallel zur Gerade  $q$ . Die Linie  $q'$  startet an dem Eckpunkt des Polygons, der am weitesten entfernt von der gegenüberliegende Seite des Polygons liegt, und endet am Schnittpunkt mit dieser Seite (vgl. Linie  $q'_2$  in Abbildung 5.8).

<sup>10</sup> Ein konkaves Polygon kommt selten vor. Das Polygon müsste durch eine Hilfskonstruktion seine konkave Eigenschaft verlieren und kann danach wie ein konvexes Polygon behandelt werden.





**Abbildung 5.9:** Das Beispiel aus Abbildung 5.2 nach der Skalierung. Der Platzbedarf um die Haltestellen bzw. Streckensegmente bleibt unverändert und wird mit Polygonen dargestellt. Dabei ist ersichtlich, dass durch die Skalierung alle Konflikte gelöst sind. (eigene Grafik)

## 5.4 Platz schaffen mit einer Skalierung

Eine Skalierung mit einem Faktor  $\lambda$  produziert eine ähnliche, jedoch vergrößerte Abbildung  $A'$  der ursprünglichen Abbildung  $A$ . Ähnliche Abbildungen sind immer winkeltreu. Sofern auch  $\lambda > 0$  gilt, bleibt die Topologie der Punkte nach der Skalierung ebenfalls unverändert.

Mit diesen beiden Eigenschaften kann somit die Topologie und die Oktilinearität des Layouts beibehalten werden. Wenn das Layout um einen Faktor  $\lambda$  skaliert wird, ändern sich die Signaturen (z.B. die Linienstärke) nicht. Während die Haltestellen, Labels und Points of Interest nur neu positioniert werden, werden Streckensegmente gestreckt und Landmarken vergrößert. Die Abbildung 5.9 zeigt die Lösung der Konflikte in Abbildung 5.2 mit Hilfe einer Skalierung.

### 5.4.1 Vorgehen

Zur Bestimmung des Faktors  $\lambda$  werden die im vorhergehenden Abschnitt ermittelten Verdrängungsvektoren herangezogen. Benötigt wird der Vektor mit der grössten Länge entlang der X- und Y-Achsen. Da die meisten Vektoren jedoch weder vertikal noch horizontal zur X-Achse sind, müssen für einen solchen Vektor  $\vec{v}$  die beiden Vektoren  $\vec{v}_1$  und  $\vec{v}_2$  ermittelt werden, wobei gilt  $\vec{v}_1 + \vec{v}_2 = \vec{v}$ . Vektor  $\vec{v}_1$  liegt parallel zur X-Achse, während  $\vec{v}_2$  parallel zur Y-Achse liegt ( $\vec{v}_1 \perp \vec{v}_2$ ). Für die Berechnung der Vektoren kann eine Vektorprojektion verwendet werden. Im Folgenden werden diese Vektoren mit  $\vec{v}_x$  und  $\vec{v}_y$  bezeichnet.

Sobald entlang der X- und Y-Achse die Vektoren mit der grössten Länge ermittelt sind, werden die Verhältnisse der Verdrängungsdistanz (Länge der Vektoren) bezogen auf die Höhe  $dy$  und Breite  $dx$  der Bounding Box beider Konfliktelementen des jeweiligen Konflikts berechnet. Das grössere der beiden Verhältnisse wird anschliessend als Skalierungsfaktor  $\lambda$  verwendet (vgl. Gleichung 5.3).

$$\lambda = \max_{c \in C} \left( \frac{dx_c + |\vec{v}_{x_c}|}{dx_c}, \frac{dy_c + |\vec{v}_{y_c}|}{dy_c} \right) \quad (5.3)$$

Die Skalierung des Graphen erfolgt nun mit dem Faktor  $\lambda$ . Die Ortsvektoren der Knoten und anderen Punkte (z.B. Positionen von Points of Interest) werden dazu mit  $\lambda$  multipliziert. Damit werden alle Konfliktzonen eliminiert. Dieses Verfahren hat jedoch Nachteile. Da die meisten Routen eines Transportnetzwerkes üblicherweise in einem oder mehreren Zentren zusammengeführt werden, sind die meisten Konfliktzonen in diesen Zentren und nicht in der Agglomeration zu finden. Eine Skalierung der ganzen Karte führt jedoch dazu, dass die Agglomeration mehr Darstellungsfläche (bei gleichbleibendem Layout) benötigt. Diese Regionen erscheinen dann grösser als erwünscht.

Die Ausdehnung der Karte wird naturgemäss ebenfalls um den Faktor  $\lambda$  vergrössert. Falls die Karte für das Zielmedium (z.B. eine Faltkarte) wieder verkleinert werden muss, müssen unter Umständen auch Kartenelemente und Labels kleiner dargestellt werden. Dies könnte unvorteilhaft sein.

Alle Schritte für die Ermittlung des Skalierungsfaktors  $\lambda$  werden im Algorithmus 2 mit Pseudo Code zusammengefasst und aufgezeigt. Der Algorithmus 3 zeigt das Zusammenspiel der Algorithmen 1 und 2.

---

#### Algorithmus 2 Skalierungsfaktor $\lambda$ ermitteln

---

```

1  procedure evaluateScaleFactor(conflicts)
2
3     maxRatioX = 1;
4     maxRatioY = 1;
5
6     for all conflict in conflicts do
7
8          $\vec{v}$  = getDisplacementVector(conflict)
9          $\vec{x}$  = (0, 1)                                     ▷ Vektor entlang der X-Achse
10         $\vec{v}_y$  =  $\vec{v}$  * (dotProduct( $\vec{x}$ ,  $\vec{v}$ ) / dotProduct( $\vec{v}$ ,  $\vec{v}$ ))           ▷ Projection
11         $\vec{v}_x$  =  $\vec{v}$  -  $\vec{v}_y$                                            ▷ Rejection
12
13        bbox = getBoundingBoxOf(conflict.getElementA(), conflict.getElementB())
14                                           ▷ Bounding Box beider Konfliktelemente
15
16        ratioX = (bbox.getWidth()+| $\vec{v}_x$ |)/bbox.getWidth()
17        maxRatioX = getMaxValue(maxRatioX, ratioX)
18
19        ratioY = (bbox.getHeight()+| $\vec{v}_y$ |)/bbox.getHeight()
20        maxRatioX = getMaxValue(maxRatioX, ratioX)
21
22    end for
23
24    return getMaxValue(maxRatioX, maxRatioY)
25
26 end procedure

```

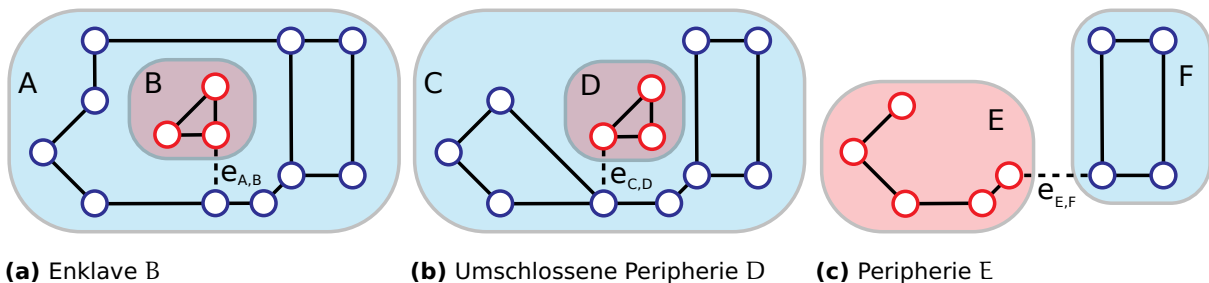
---

**Algorithmus 3** Platz schaffen mit Skalierung

```

1 procedure makeSpaceByScaling(map, routeMargin, edgeMargin, nodeMargin)
2
3   edges = map.getEdges()
4   nodes = map.getNodes()
5
6   conflicts = getConflictPolygons(edges, nodes, routeMargin, edgeMargin, nodeMargin)
7
8   if conflicts contains at least one conflict then
9
10    scaleFactor = getScaleFactor(conflicts)
11    scale(map, scaleFactor)
12
13  end if
14
15 end procedure

```



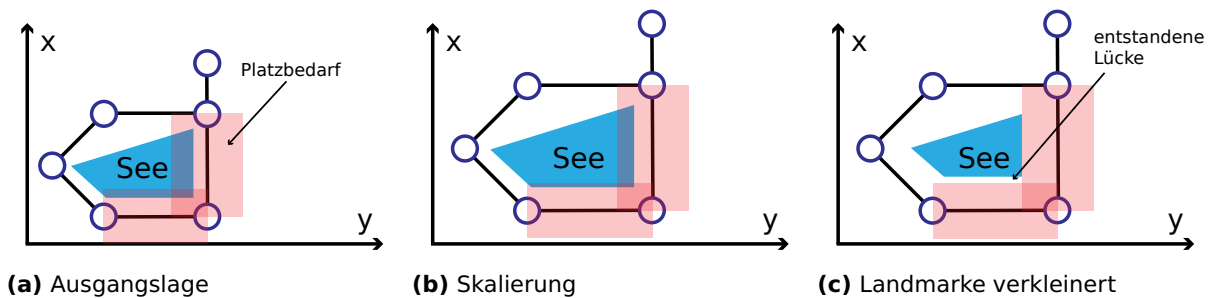
**Abbildung 5.10:** Problemregionen bei der Skalierung von Teilgraphen. Die Abbildungen zeigen von links nach rechts einen vollständig umschlossenen Teilgraphen (Enklave), einen Teilgraphen mit sehr nahen Nachbarn und einen abgesetzten Teilgraphen (Peripherie). Eine unabhängige Skalierung der Teilgraphen B und D könnte unter Umständen zu einer Überlappung mit den Teilgraphen A bzw. C führen. Die Skalierung der Peripherie ist relativ unproblematisch. (eigene Grafik)

### 5.4.2 Teilgraphen skalieren

Teilt man den Graph in mehrere Teilgraphen auf, so ist unter Umständen eine Skalierung der einzelnen Teilgraphen möglich. Dies würde verhindern, dass Regionen, die keine oder kleinere Konfliktzonen besitzen, unnötig vergrößert dargestellt werden.

Die durch das Entfernen von Brücken  $e_{k_1, k_2}$  entstehenden Komponenten werden somit einzeln skaliert. Für jede Komponente  $k$  wird der Faktor  $\lambda_k$  für die Skalierung bestimmt (vgl. Listing 2). Die skalierten Komponenten werden nach der Skalierung der einzelnen Teilgraphen wieder durch die zuvor entfernten Brücken verbunden. Die ursprünglichen Winkel bleiben gleich. Jedoch wird unter Umständen die Topologie verändert. Abhilfe kann eventuell eine Verlängerung oder Verkürzung der jeweiligen Brücke schaffen.

Falls jedoch durch das Entfernen einer Brücke  $e_{A, B}$  eine Enklave (also ein Teilgraph B innerhalb eines anderen, äusseren Teilgraphen A) entsteht, macht die Anwendung unterschiedlicher Faktoren  $\lambda_A$  und  $\lambda_B$  keinen Sinn. Falls nämlich  $\lambda_B > \lambda_A$  gilt, überlagert unter Umständen der Teilgraph B den ursprünglich äusseren Teilgraphen A nach der Skalierung.



**Abbildung 5.11:** Berücksichtigung der Landmarken bei der Skalierung (1). Die Landmarke wird zusammen mit den Positionen skaliert (mittlere Abbildung) und anschliessend um einen aus den Konflikten berechneten Faktor verkleinert (rechte Abbildung). (eigene Grafik)

Umgekehrt, falls  $\lambda_B < \lambda_A$  gilt, würde das Grössenverhältnis der Komponenten die Ästhetik des neuen Layouts möglicherweise stören. Die Brücke wird wieder eingefügt und beide Teilgraphen werden als ein einziger Teilgraph behandelt und mit dem grösseren Faktor skaliert (vgl. Abbildung 5.10a).

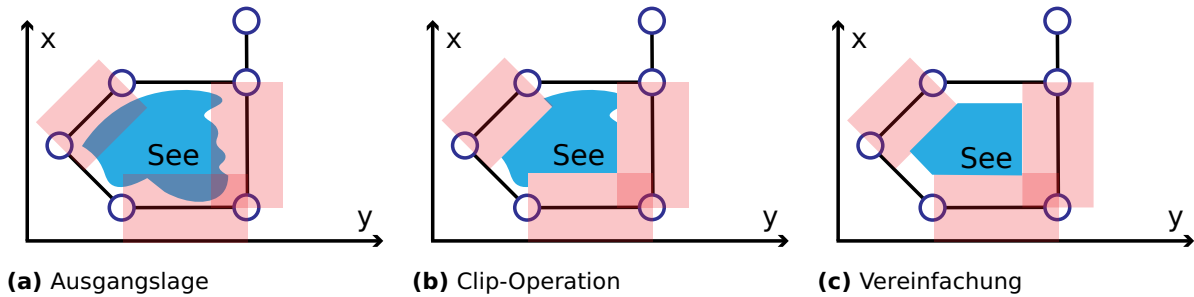
Dieses Vorgehen eignet sich vor allem für Peripherien. Allerdings kann auch das Resultat der Skalierung einer Peripherie zu einer Überlappung führen. Dies ist der Fall, wenn die Komponente D teilweise von einer anderen Komponente C umschlossen wird. Es muss daher geprüft werden, ob eine Überlappung auftritt ( $\lambda_D > \lambda_C$ ). Sollte dies der Fall sein, werden auch hier die beiden Teilgraphen als ein einziger Teilgraph behandelt, der nun mit dem grösseren Faktor skaliert wird (vgl. Abbildungen 5.10b und 5.10c).

### 5.4.3 Berücksichtigung weiterer Kartenelemente

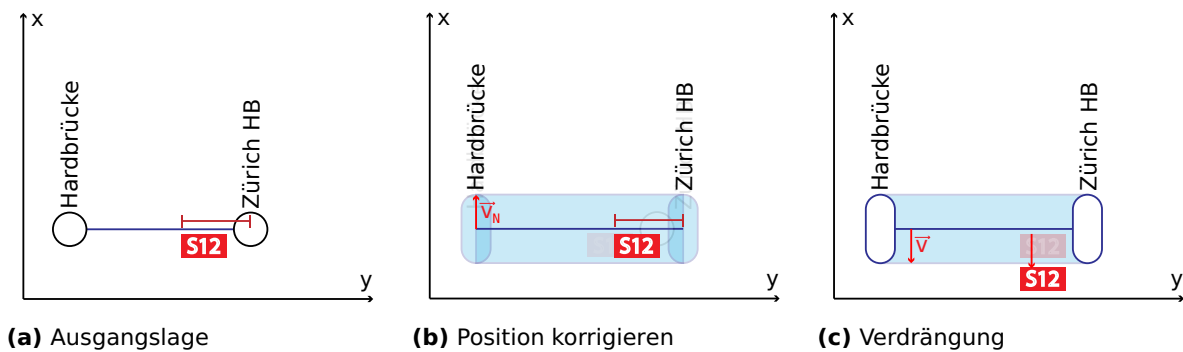
#### Landmarken

Konflikte zwischen Landmarken und Streckensegmenten bzw. Haltestellen können mit der Skalierung alleine häufig nicht behoben werden. Deshalb gilt es solche Konflikte und den notwendigen Platzbedarf zu ermitteln. Die Landmarke wird bei der initialen Skalierung ebenfalls berücksichtigt und entsprechend vergrössert. In einem zweiten Schritt wird mit einem eigenen Skalierungsfaktor (basierend auf dem eruierten Platzbedarf) vom Mittelpunkt der Landmarke ausgehend verkleinert. Der Nachteil an diesem Vorgehen ist, dass Streckensegmente mit einem geringeren Platzbedarf durch den Vorgang weiter von der Landmarke entfernt als erwünscht visualisiert werden und es entsteht eine Lücke (vgl. Abbildung 5.11).

Eine Lösung könnte das Entfernen derjenigen Bereiche der Landmarke, welche sich mit den Konfliktpolygonen überlappen, darstellen. Dabei wird unter Umständen die Form der Landmarke jedoch so verändert, dass die ursprüngliche Form nicht mehr erkennbar ist. Die Problematik tritt vor allem dann auf, wenn es sich um ein konkaves Polygon handelt oder die Landmarke einen hohen Detailgrad (sehr viele Eckpunkte) besitzt. Kleine



**Abbildung 5.12:** Berücksichtigung der Landmarken bei der Skalierung (2). Die Landmarke wird zusammen mit den Positionen skaliert (mittlere Abbildung), anschliessend werden die Konflikte mit einer Clip-Operation entfernt und vereinfacht (rechte Abbildung). (eigene Grafik)



**Abbildung 5.13:** Berücksichtigung der Labels entlang eines Streckensegments bei der Skalierung. Die Position der Haltestellen-Labels und der Routenbeschriftung entlang des Streckensegments wird korrigiert (mittlere Abbildung). Anschliessend wird das Label um den Platzbedarf verschoben (rechte Abbildung). (eigene Grafik)

Änderungen (z.B. ungewollte Ausbuchtungen) könnten durch eine erneute Vereinfachung (Simplification) oder Glättung (Smoothing) korrigiert werden (vgl. Abbildung 5.12).

### Labels und Points of Interest

Labels und Points of Interest, deren Position abhängig zu einer Haltestelle sind, müssen ebenfalls neu positioniert werden. Sie werden zuerst entsprechend der ursprünglichen Entfernung und Ausrichtung zur Haltestelle platziert. Danach werden die Konflikte ermittelt und die entsprechenden Verdrängungsvektoren berechnet. Da die freie Fläche (zwischen den Elementen) durch die Skalierung der Haltestellen-Positionen nicht kleiner wird, müssen keine weitere Elemente verdrängt werden. Das Label oder der Point of Interest wird einfach entlang des Verdrängungsvektors verschoben (vgl. Abbildung 5.13b).

Dasselbe Vorgehen kann für Labels mit einer Abhängigkeit zu einem Streckensegment verwendet werden. Bei der Platzierung vor der Ermittlung der Konflikte muss beachtet werden, dass die ursprüngliche Distanz zu einem Knoten oder das Verhältnis der Entfernung des Labels zu beiden Knoten der Kante erhalten bleibt. Die Länge des Vektors  $\vec{v}$  ist die Hälfte des Platzbedarf des Streckensegments  $\frac{w_e}{2}$  (für  $w_e$  vgl. Gleichung 5.1) und orthogonal zur (verlängerten) Kante (vgl. Abbildung 5.13).

## 5.5 Sequentieller Verdrängungsalgorithmus

Die Platzbeschaffung mit Hilfe einer Skalierung erzeugt häufig übergrosse Randregionen. Im folgenden Abschnitt wird ein anderer Ansatz zur Platzbeschaffung diskutiert. Dabei kommt ein sequentielles Prinzip zur Anwendung, welches pro Konflikt eine Verdrängungsgerade  $g$  ermittelt und alle Knoten auf einer Seite um den benötigten Platz verschiebt. Dabei entstehen nicht-oktilineare Kanten, deren Oktilinearität mit einer Anpassung des Layouts wiederhergestellt werden.

Die zu ermittelnde Verdrängungsgerade  $g$  liegt parallel zur X- oder Y-Achse und verläuft zwischen den beiden Konfliktelementen. Für jede Konfliktzone werden die Knoten auf der östlichen bzw. nördlichen Seite (Verdrängungsseite) der Verdrängungsgerade um die notwendige Verdrängung verschoben. Die Verdrängung erfolgt jeweils orthogonal zur Verdrängungsgerade nach Norden (Y-Achse) oder Osten (X-Achse).

Sobald die Knoten verschoben wurden, wird die Oktilinearität überprüft. Kanten, welche die Verdrängungsgerade  $g$  schneiden oder berühren werden im Folgenden Verbindungskanten genannt. Sie verlieren ihre oktilineare Ausrichtung durch den Verdrängungsprozess, ausser wenn die Verbindungskante  $e_d$  orthogonal zur Verdrängungsgerade  $g$  liegt. Muss die Oktilinearität wiederhergestellt werden, geschieht dies durch die Anwendung von Regeln, die der Algorithmus befolgt. Dadurch werden Knoten auf der Seite der Verdrängungsgerade  $g$  verschoben, deren Anpassung die kleinste Auswirkung auf das Layout hat. Ab einer bestimmten Komplexität wird die Möglichkeit geprüft, ob ein Knick in der betreffenden Verbindungskante  $e_d$  ein besseres Resultat erzielt.

Wurde die Oktilinearität wiederhergestellt, müssen die Konfliktpolygone erneut berechnet werden. Es könnten neue Konflikte entstanden und/oder idealerweise bestehende Konflikte mitgelöst worden sein. Anschliessend wird der Vorgang für den nächsten Konflikt wiederholt. Dies wird iterativ solange wiederholt, bis keine Konflikte mehr vorhanden sind (vgl. Algorithmus 4).

### 5.5.1 Vorgehen

#### Verdrängungsgerade ermitteln

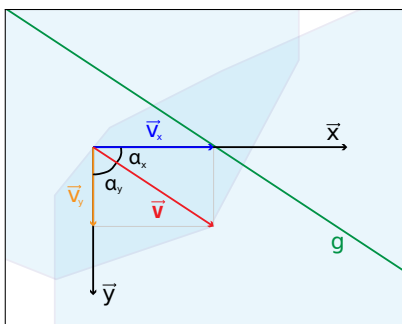
Die Verdrängungsgerade  $g$  liegt parallel zur X- oder Y-Achse, verläuft zwischen den beiden Konfliktelementen und ist orthogonal zum Verdrängungsvektor, der angewendet werden soll. Dabei soll die Verdrängungsgerade  $g$  Konfliktelemente nicht schneiden. Falls jedoch eine Kante/Kante-Kombination vorliegt und sich die Verlängerung einer Kante mit der anderen Kante (nicht verlängert) schneidet, so kann keine Verdrängungsgerade mit

**Algorithmus 4** Platz schaffen mit einem Verdrängungsalgorithmus

```

1 procedure makeSpaceByDisplacement(map, routeMargin, edgeMargin, nodeMargin)
2
3   edges = map.getEdges()
4   nodes = map.getNodes()
5
6   conflicts = getConflictPolygons(edges, nodes, routeMargin, edgeMargin, nodeMargin)
7
8   while conflicts is not empty do
9     conflict = getConflictWithBiggestImpact(conflicts)
10    displace(conflict, map)
11    conflicts = getConflictPolygons(edges, nodes, routeMargin, edgeMargin, nodeMargin)
12  end while
13
14 end procedure

```



**Abbildung 5.14:** Verdrängungsvektoren mit der kleinsten Verfälschung ermitteln. Die Grafik zeigt eine vergrößerte Version des Konfliktpolygons aus der Abbildung 5.8. In diesem Beispiel ist die Verfälschung des Layouts mit der Verwendung der Projektion des Vektors  $\vec{v}$  entlang von  $\vec{x}$  geringer weil  $\alpha_x < \alpha_y$ . (eigene Grafik)

diesen Eigenschaften gefunden werden<sup>11</sup>.

**Verdrängung**

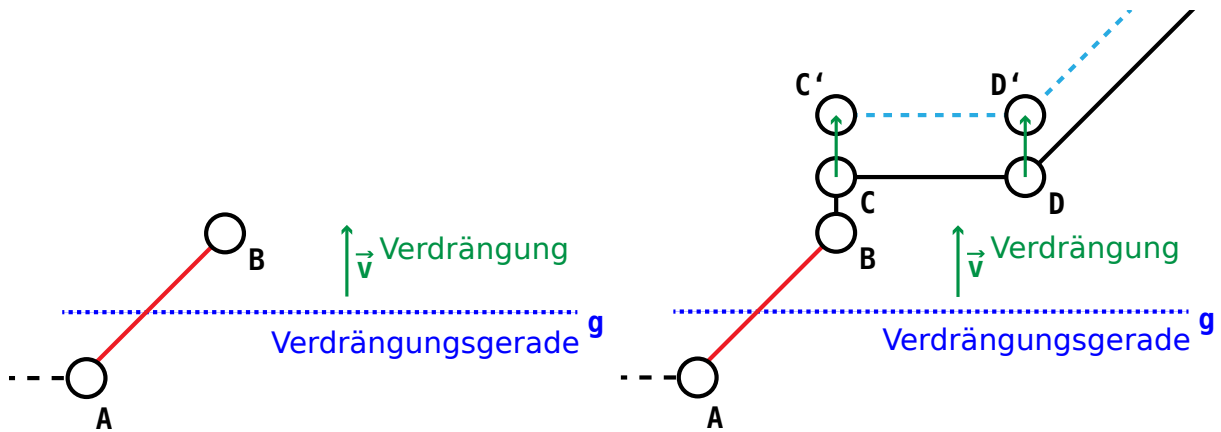
Für die Verdrängung wird die Verdrängungsrichtung gewählt, welche orthogonal zur Verdrängungsgerade  $g$  bzw. parallel zum Verdrängungsvektor liegt. Nun werden alle Knoten auf der festgelegten Verdrängungsseite um den Konfliktvektor  $\vec{v}$  verschoben. Eine Verdrängung findet jedoch immer entlang der X- oder Y-Achse statt<sup>12</sup> und hat eine positive Distanz. Falls nun  $\vec{v}$  nicht orthogonal zur X-Achse ist, muss diejenige Projektion des Vektors  $\vec{v}$  und eines Vektors entlang der X-Achse ( $\vec{x}$ ) und Y-Achse ( $\vec{y}$ ) gefunden werden, welche die kleinste Verfälschung verursacht. Dies ist die Projektion mit dem kleinsten Winkel  $\alpha$  zum Vektor  $\vec{v}$  (vgl. Abbildung 5.14)<sup>13</sup>.

Speziell behandelt wird ein Knoten  $B$  mit einer einzigen adjazenten Verbindungskante  $\overline{BC}$ . Falls nun der Knoten  $B$  nur einen adjazenten Knoten  $A$  besitzt (vgl. Abbildung 5.15a), wird auf eine Verdrängung des Knotens  $B$  verzichtet. Dasselbe gilt, wenn die Verbindungs-

<sup>11</sup> Auf diesen Sonderfall wird in diesem Kapitel nicht näher eingegangen. Ein Lösungsweg wäre beispielsweise die vorgängige Skalierung des Graphen mit den grössten Skalierungsfaktor zur Behebung aller Konflikte die keine orthogonale Verdrängungsgerade  $g$  aufweisen.

<sup>12</sup> Diese Arbeit beschränkt sich auf diese zwei Verdrängungsrichtungen. Theoretisch wäre auch eine Verdrängung im 45 Grad Winkel zur den Achsen durchführbar.

<sup>13</sup> Dies gilt nicht für einen Konflikt zwischen einem Knoten und einer Kante. Eine Lösung wird im Kapitel 6.3.2 vorgestellt.



(a)  $\deg_G(B) = 1$

(b)  $\deg_G(B) = 2$  und  $\overline{BC} \perp g$

**Abbildung 5.15:** Verzicht auf die Verdrängung des Knotens A falls  $\deg_G(B) = 1$  oder  $\deg_G(B) = 2$  und  $\overline{BC} \perp g$ . (eigene Grafik)

kante  $\overline{AB}$  eine adjazente Kante  $\overline{BC}$  aufweist und diese Kante orthogonal zur Verdrängungsgerade  $g$  liegt. In diesem Fall wird die Kante  $\overline{BC}$  lediglich verlängert (vgl. Abbildung 5.15b).

Falls eine Verbindungskante  $e_d$ , welche die Verdrängungsgerade  $g$  schneidet, gleichzeitig eine Brücke ist, könnte von der Verdrängung des Teilgraphen auf der Verdrängungsseite ebenfalls abgesehen werden. In diesem Fall wird die Topologie unter Umständen jedoch stärker verändert als erwünscht (vgl.  $\overline{JK}$  in Abbildung 5.5a).

### Wiederherstellung der Oktilinearität

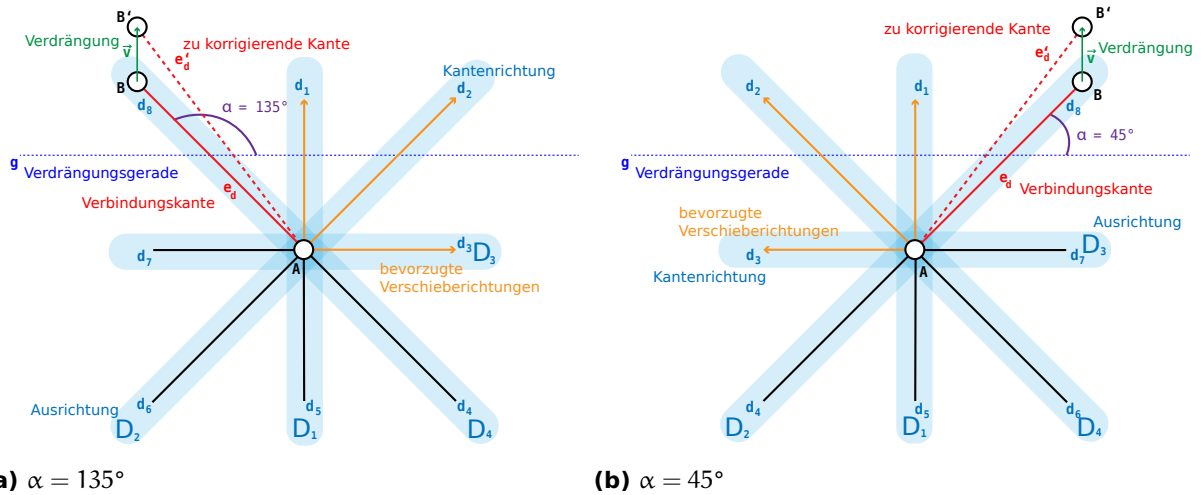
Die Wiederherstellung der Oktilinearität sollte jeweils auf der Seite einer Verbindungskante  $e_d$  durchgeführt werden, die den adjazenten Knoten mit den geringsten Anpassungskosten  $\kappa$  enthält. Der Aufwand der Wiederherstellung der Oktilinearität der Verbindungskante  $e_d$  und somit auch die Anpassungskosten  $\kappa$  sind abhängig von der Zusammensetzung der adjazenten Kanten der Verbindungskante.

Verbindungskanten mit einem Knoten, welche eine oder zwei adjazente Kanten besitzen, die entweder vertikal, horizontal oder im 90 Grad Winkel zu den Verbindungskanten stehen (vgl.  $D_1$ ,  $D_2$  und  $D_3$  in den Abbildungen 5.17a, 5.17b und 5.17c), können problemlos durch eine Verschiebung des adjazenten Knotens entlang der Verlängerung der adjazenten Kanten korrigiert werden (vgl. oranger Pfeil in der Abbildung 5.17). Solche Knoten werden in den folgenden Abschnitten als einfache Korrekturknoten bezeichnet.

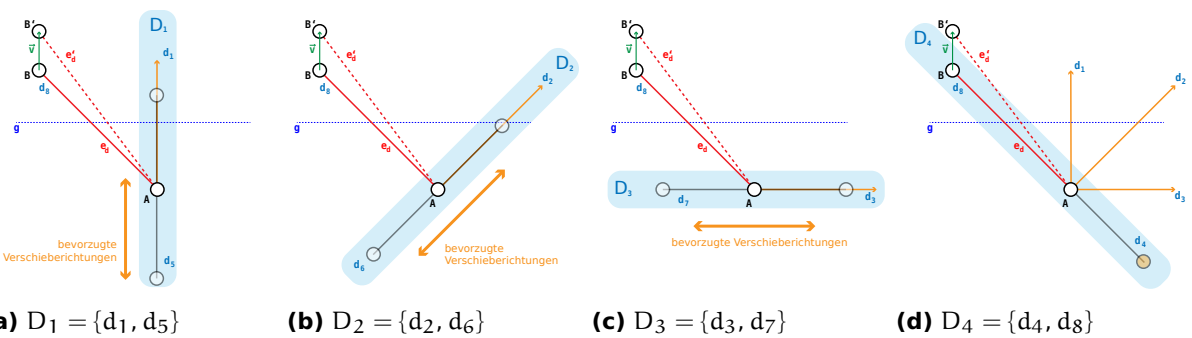
Hat die Verbindungskante  $e_d$  vor der Verdrängung jedoch eine adjazente Kante, die eine Verlängerung der Verbindungskante  $e_d$  darstellt (die gleiche Ausrichtung hat), kann nicht entlang der Verlängerung der adjazente(n) Kante(n) verschoben werden (vgl.  $D_4$  in



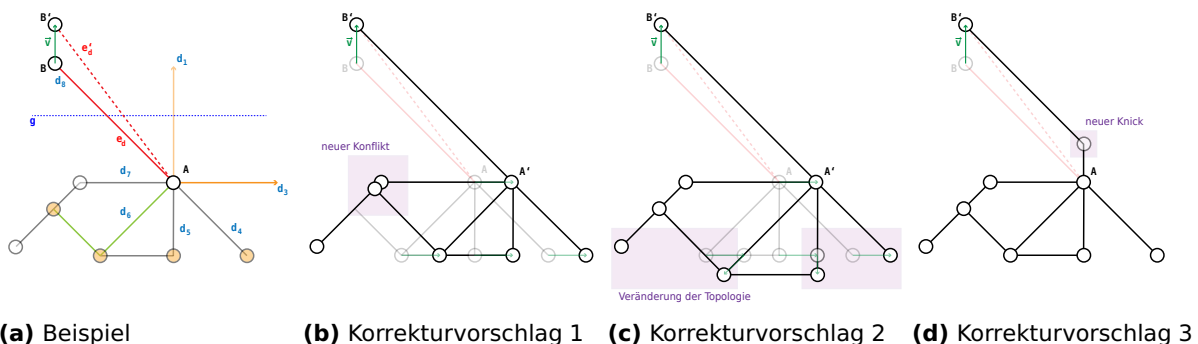
## 5.5. Sequentieller Verdrängungsalgorithmus



**Abbildung 5.16:** Mögliche Verschieberichtungen zur Wiederherstellung der Oktilinearität. Die linke Abbildung ist eine Spiegelung der rechten Abbildung. Daher ist die Nummerierung der Kantenrichtungen und der Ausrichtungen ebenfalls spiegelverkehrt. Die Verdrängungsgerade  $g$  ist entweder parallel zur X-Achse oder parallel zur Y-Achse. Mit den beiden Abbildungen sind nur zwei von vier möglichen Situationen dargestellt. Die nicht dargestellten Situationen können mit einer horizontalen Spiegelung der Abbildungen erreicht werden. (eigene Grafik)



**Abbildung 5.17:** Situationen mit nur einer Ausrichtung der adjazenten Kanten der Verbindungskante  $e_d$ . In den ersten drei Abbildungen ist das Verschieben entlang der verlängerten Linie der adjazenten Kante möglich. Falls die adjazente Kante eine Verlängerung der Verbindungskante (vgl. vierte Abbildung) ist, muss eine andere Verschieberichtung für den Knoten  $A$  gewählt werden. (eigene Grafik)



**Abbildung 5.18:** Situation mit Kombinationen von Ausrichtungen der adjazenten Kanten. Die orangen Knoten in der ersten Abbildung müssen in weiteren Iterationen korrigiert werden. Insbesondere bei den beiden grün eingezeichneten Kanten ist die Korrektur problematisch. Alle Korrekturvorschläge haben Nachteile. Während beim ersten Vorschlag ein neuer Konflikt entsteht und eine weitere Iteration benötigt wird, werden im zweiten Vorschlag die relativen Positionen unter den Haltestellen geändert. Im dritten Vorschlag wird ein neuer Knick hinzugefügt. (eigene Grafik)

Abbildung 5.17d). Dasselbe gilt für Kombinationen von verschiedenen Ausrichtungen der adjazenten Kanten. In diesem Fall steigt der Aufwand für die Wiederherstellung der Oktilinearität der Verbindungskante  $e_d$ , da eine Korrektur Auswirkungen auf deren adjazenten Kanten haben kann, wobei sich die Winkel dieser Kanten unter Umständen ebenfalls verändern (vgl. Knoten A, B, R und S in der Abbildung 5.5b). Die Abbildung 5.18 zeigt ein Beispiel und Korrekturvorschläge für eine Kombination von Ausrichtungen der adjazenten Kanten. Knoten, welche ein komplexeres Korrekturvorgehen benötigen, werden im Folgenden als komplexe Korrekturknoten bezeichnet.

Die Anpassungskosten  $\kappa$  hängen somit von der Summe adjazenter Kanten ab. Die Kosten der adjazenten Kanten werden rekursiv aufsummiert, solange diese wiederum eine Korrektur benötigen. Um einen Korrektur-Kreis zu verhindern, werden bereits besuchte Knoten nicht weiter traversiert und mit dem Wert 1000 bestraft. Ein Korrektur-Kreis kann entstehen, wenn bei allen vorgängig traversierten Knoten keine einfache Korrekturmöglichkeit angewendet werden können<sup>14</sup>. Um einen solchen Kreis aufzulösen, würden mehrere Iterationen benötigt. Die Konstellation müsste als Optimierungsproblem behandelt werden und mit einem entsprechenden Algorithmus (z.B. Hill Climbing) global gelöst werden.

Da wie in der Abbildung 5.18 ersichtlich verschiedene Korrekturmöglichkeiten und Kombinationen existieren können, handelt es sich bei den Anpassungskosten  $\kappa$  um keine genaue Kostenzahl, sondern um eine Abschätzung, die mit einem minimalen Aufwand evaluieren soll, welche Seite der Verdrängungsgerade  $g$  für die Korrektur geeigneter ist. Das Listing 5 skizziert die Berechnung der Anpassungskosten  $\kappa$  mit Pseudo Code.

Die Korrektur kann im schlechtesten Fall alle Knoten bzw. Kanten des Graphen betreffen oder es kann ein Korrektur-Kreis auftreten. Deshalb wird ab einer bestimmten Komplexität eine Layout-Korrektur, beispielsweise durch das Hinzufügen eines Knicks (Bend) auf der zu korrigierenden Kante, in Betracht gezogen. Als Indikator können hier ebenfalls die Anpassungskosten herangezogen werden. Dazu wird ein maximaler Wert  $\iota$  definiert, der nicht von beiden Seiten überschritten werden darf. Sind die Kosten auf beiden Seiten trotzdem höher, so wird eine Layout-Korrektur (Knicke) vorgenommen.

Der Knoten (auf der zu korrigierenden Seite der Verbindungskante  $e_d$ ) wird nun soweit in eine der acht möglichen Richtungen verschoben, bis die Oktilinearität wiederhergestellt werden kann. Anschliessend werden alle adjazenten Kanten auf ihre Oktilinearität überprüft. Gibt es eine adjazente Kante, die eine nicht-oktilineare Ausrichtung hat, so wird die Korrektur auch für diese Kante durchgeführt. Dies wird solange rekursiv wiederholt,

<sup>14</sup> Graphentheorie: Ein Kreis ist ein geschlossener Weg aus aufeinanderfolgenden Kanten.

**Algorithmus 5** Anpassungsknoten berechnen

---

```

1 procedure calculateAdjustmentCosts(connectionEdge, node, traversedNodes)
2
3   if node in traversedNodes then
4     return 1000                                ▷ Penalty, um Korrektur-Kreise zu vermeiden
5   end if
6
7   traversedNodes.add(node)
8
9   adjacentEdges = getAdjacentEdges(node).without(connectionEdge)
10
11  if both adjacentEdges has same direction as  $D_1$ ,  $D_2$  or  $D_3$  then
12    return 1                                    ▷ einfache Korrektur möglich
13  end if
14
15  costs = 1 + adjacentEdges.count()
16
17  for all adjacentEdge in adjacentEdges do
18    oNode = adjacentEdge.getOtherAdjacentNode(node)
19    costs = costs + calculateAdjustmentCosts(adjacentEdge, oNode, traversedNodes)
20  end for
21
22  return costs
23
24 end procedure

```

---

bis keine Korrektur mehr nötig ist (vgl. Algorithmus 6). Der letzte Knoten ist in der Regel ein einfacher Korrekturknoten. Falls die Korrekturdistanz grösser als die Kante entlang der Korrekturrichtung ist, so muss der adjazente Knoten ebenfalls in dieselbe Richtung verschoben werden (vgl. Kante  $\overline{CD}$  in Abbildung 5.19). Falls eine Kante verkleinert werden muss und dadurch ein neuer Konflikt entsteht, wird dieser Konflikt in einer der folgenden Iterationen korrigiert (vgl. Kante  $\overline{AE}$  in Abbildung 5.19).

**Algorithmus 6** Kante korrigieren

---

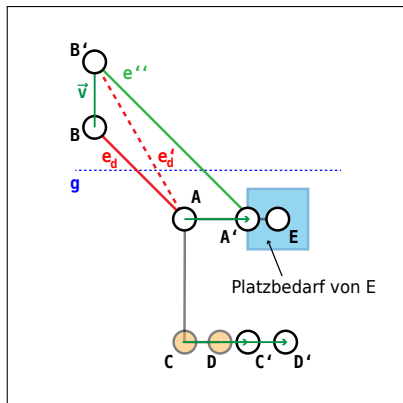
```

1 procedure correctEdge(edge, node)
2
3   direction = moveNodeUntilEdgeIsOctlilinear(node, edge)
4
5   edges = getNonOctlilinearAdjacentEdges(node).without(edge)
6
7   for all edge in edges do
8     correctEdge(edge, edge(node), direction)
9   end for
10
11 end procedure

```

---

Durch das Verschieben von Knoten können unter Umständen auch Überkreuzungen von Kanten auftreten (vgl. Kante  $\overline{MN}$  in Abbildung 5.5b). In diesem Fall muss die überkreuzte Kante  $\overline{MN}$  ebenfalls um die gleiche Korrekturdistanz verschoben werden. Dies hat jedoch zur Folge, dass adjazente Kanten der Kante unter Umständen keine oktilineare Ausrich-



**Abbildung 5.19:** Behandlung von kleinen Kanten nach der Verdrängung. Die Kante  $\overline{CD}$  ist kleiner als  $|\vec{v}|$ , sodass der Knoten D ebenfalls verschoben werden muss. Die neue Distanz zwischen  $A'$  und E verursacht aufgrund des Platzbedarfs von E einen neuen Konflikt. (eigene Grafik)

tung mehr haben und korrigiert werden müssen.

Die Korrektur der adjazenten Kanten der Verbindungskante  $e_d$  mit einem komplexen Korrekturknoten erfolgt in der Regel mit einer Verschiebung des Knotens in die Richtung, welche bei der Korrektur der Verbindungskante  $e_d$  verwendet wurde. Dies ist in der Abbildung 5.18b dargestellt. Alle Knoten werden nach rechts verschoben. Anders sieht es beim zweiten Korrekturvorschlag (vgl. Abbildung 5.18c) aus. Dort müsste jedes Mal aufs neue eine geeignete Verschieberichtung evaluiert werden.

Falls die Anpassungskosten  $\kappa$  grösser als die maximalen Anpassungskosten  $\iota$  sind, wird eine Layout-Korrektur vorgenommen (vgl. Zeile 16 im Algorithmus 7). Falls ein Korrekturkreis auftreten könnte, wurde mindestens ein Knoten bei der Berechnung der Anpassungskosten zweimal besucht und entsprechend bestraft. In diesem Fall ist  $\kappa$  immer grösser als  $\iota$ . Durch die Layout-Korrektur entsteht eine neue vertikale oder horizontale Kante und die Wahrscheinlichkeit von weiteren Korrekturkreisen wird minimiert.

Falls die Anpassungskosten  $\kappa$  beider Knoten der Verbindungskante  $e_d$  grösser als die maximale Anpassungskosten  $\iota$  sind, wird die Oktilinearität der Kante  $e_d$  mit zusätzlichen Knicken wiederhergestellt. Es sollen allerdings maximal 2 zusätzlichen Knoten (für die Knicke) eingeführt werden. Die ursprüngliche Kante wird anschliessend mit einem bzw. zwei Knoten sowie mit zwei bzw. drei Kanten ersetzt. Die Abbildung 5.20 zeigt mögliche Varianten für die Korrektur von nicht-oktilinearen Kanten durch Knicke.

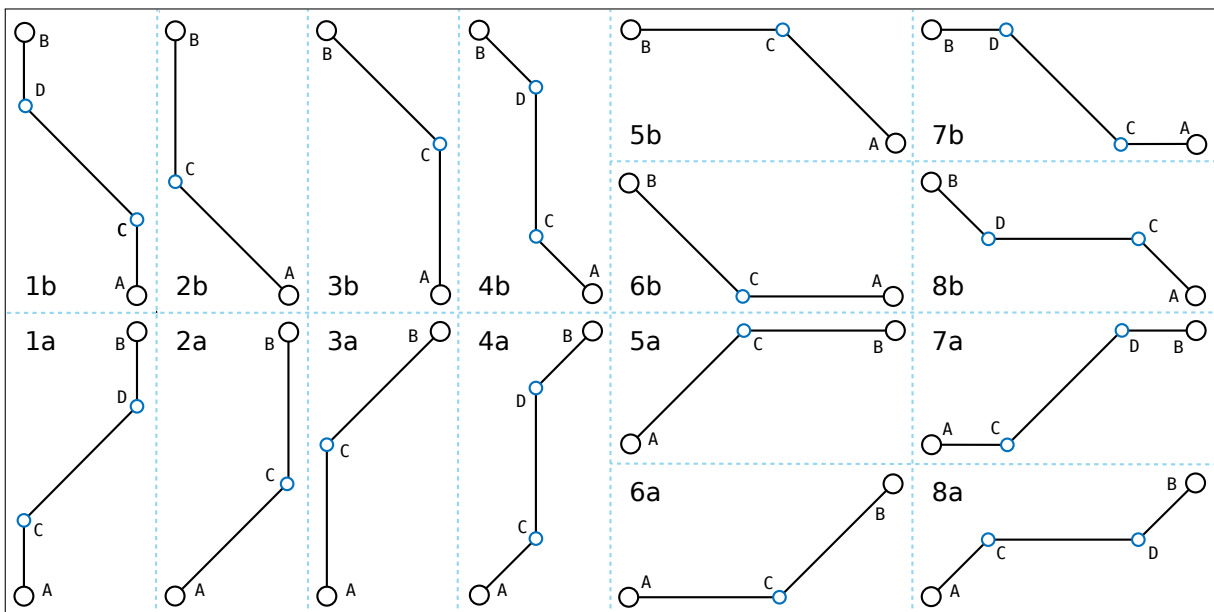
Die Wahl der Variante des Knicks ist abhängig von der Ausrichtung der Kante und von der Ausrichtung der adjazenten Kanten. Eingefügte Knicke sollten beispielsweise nicht auf einer adjazenten Kante liegen.

**Algorithmus 7** Konflikt auflösen

```

1  procedure displace(conflict, map)
2
3   $\vec{v}$  = conflict.getDisplacementVector()           ▷ Ermittle orthogonalen Vektor
4  g = conflict.getDisplacementLine()                ▷ Ermittle Verdrängungsgerade g
5
6  moveNodes(map, g,  $\vec{v}$ )                          ▷ Verschiebe Knoten orthogonal
7
8  edges = getNonOctilinearEdges(map, g)           ▷ Ermittle nicht-oktilineare Kanten
9
10 for all edge in edges do
11                                     ▷ Berechne Anpassungskosten
12    $\kappa_A$  = calculateAdjustmentCosts(edge, edge.getNodeA(), new Set())
13    $\kappa_B$  = calculateAdjustmentCosts(edge, edge.getNodeB(), new Set())
14
15   if getMinValue( $\kappa_A$ ,  $\kappa_B$ ) >  $\iota$  then           ▷  $\iota = 1000$ 
16     modifyEdge(edge)                                 ▷ Füge Knick hinzu
17   else                                             ▷ oder korrigiere Kante(n) rekursiv
18     if  $\kappa_A > \kappa_B$  then
19       correctEdge(edge, edge.getNodeB())
20     else
21       correctEdge(edge, edge.getNodeA())
22     end if
23   end if
24
25 end for
26
27 end procedure

```



**Abbildung 5.20:** Eine Auswahl von Korrekturvarianten durch Einfügung von Knicken zur Wiederherstellung der Oktilinearität. 1a bis 8a zeigt die Varianten für eine Diagonale von Südwest nach Nordost ( $D_2$ ) und 1b bis 8b für eine Diagonale von Südost nach Nordwest ( $D_4$ ). (eigene Grafik)

### Evaluierung der Verschieberichtung und Verschiebedistanz

Die möglichen Verschieberichtungen für einfache Korrekturknoten werden in der Abbildung 5.18 aufgezeigt. Für die Berechnung der Verschiebedistanz wird die Breite  $dx$  und Höhe  $dy$  der Bounding Box beider (adjazenten) Knoten der Verbindungskante  $e_d$  benötigt. Die Verschiebedistanz  $|\vec{k}|$  kann mit der Gleichung 5.4 berechnet werden.

$$|\vec{k}| = (|dx| - |dy|) \quad (5.4)$$

Die Evaluierung der Verschieberichtung von komplexen Korrekturknoten erweist sich als schwieriger und ist abhängig von den Kantenrichtungen der adjazenten Kanten sowie von der Korrekturkomplexität der adjazenten Knoten. Die Ermittlung der idealen Verschieberichtungen von komplexen Korrekturknoten ist eine aufwändige Prozedur und geht über den Rahmen der vorliegenden Arbeit hinaus.

### 5.5.2 Berücksichtigung weiterer Kartenelemente

#### Landmarken

Die Anwendung des Verdrängungsalgorithmus in einen Liniennetzplan mit Landmarken hat den Nachteil, dass die Form der Landmarken unter Umständen sehr stark verändert werden kann. Jedes Verschieben eines Knoten verändert die Form der Landmarke, indem diese gestaucht oder auseinander gezogen wird. Dieses Verhalten ergibt sich aus den unterschiedlichen Grössenänderungen der einzelnen Streckensegmente.

Um die Deformation von Landmarken zu verhindern, könnten qualifizierte Knoten eingeführt werden. Dabei werden alle direkt benachbarten Knoten der Landmarke einer Gruppe zugeordnet. Die Relation der Abstände zwischen den Knoten einer Gruppe darf nicht verändert werden. Der Verdrängungsalgorithmus kann die Knoten einer Gruppe nicht einzeln verschieben, sondern muss alle Knoten miteinander in dieselbe Richtung verschieben.

#### Labels und Points of Interest

Grundsätzlich können die Positionen der Labels und Points of Interest mit dem bereits im Kapitel 5.4.3 beschriebenen Vorgehen korrigiert werden. Da sich der Raum zur Platzierung der Labels und Points of Interest aber verkleinern kann, ist unter Umständen nach der Anwendung des Verdrängungsalgorithmus nicht mehr genügend Fläche zur Darstellung der Elemente vorhanden. Aus diesem Grund kann es auch hier sinnvoll sein mit qualifizierten Knoten zu arbeiten.

## 6 | Implementierung

Dieses Kapitel beschreibt die Implementierung von zwei der drei diskutierten Ansätze, nämlich die Platzbeschaffung mittels Skalierung und durch den Verdrängungsalgorithmus<sup>1</sup>. Für die Implementierung wird die objekt-orientierte Programmiersprache Java<sup>2</sup> (Version 8) verwendet. Als externe Bibliothek wird die JTS Topology Suite<sup>3</sup> (Version 1.14) verwendet.

In diesem Kapitel wird auszugsweise auf den Source Code, welcher für die Implementierung der diskutierten Ansätze im Kapitel 5 relevant ist, eingegangen. Des Weiteren werden die während der Implementierung auftretenden Probleme diskutiert.

### 6.1 Konfliktzonen ermitteln

Für die Evaluierung der Streckensegmente ist die Klasse `ConflictFinder` verantwortlich (vgl. D.1.1, `ConflictFinder`). Die Klasse `BufferConflict` repräsentiert einen Konflikt zwischen zwei Graph-Elementen (vgl. D.1.1, `ConflictFinder`).

#### Konfliktbuffer erstellen

Für die Erstellung eines Buffers könnte z.B. eine Minkowski Addition<sup>4</sup> verwendet werden. Die JST Topology Suite bietet jedoch bereits Funktionen zur Erstellung eines Buffers an, die in der vorliegenden Arbeit genutzt werden.

Ein Buffer eines Liniensegments mit flachen Enden kann mit der Klasse `Geometry` und der Methode `buffer(distance, int)` erstellt werden. Diese Methode wird von der Klasse `GeomUtil` abstrahiert und in der Klasse `EdgeBuffer` verwendet (vgl. Anhang D.6.2, `GeomUtil`). Mit dessen Methode `updateBuffer()` und `calculateEdgeWidth(double)`, einer Methode der Klasse `Edge` (vgl. Listing 6.1 und 6.2), wird die benötigte Breite zur Darstellung der Routeninformationen gemäss Gleichung 5.1 berechnet (vgl. Anhang D.4.4, `Edge`).

Die Methode `createStationConvexHull(node)` aus dem Listing 1 (Zeile 17) wurde indirekt mit der Klasse `SquareStationSignature` implementiert (vgl. Listing 6.3 und Anhang

---

<sup>1</sup> Nicht implementiert wird die Platzbeschaffung durch eine Skalierung einzelner Teilgraphen.

<sup>2</sup> vgl. <http://docs.oracle.com/javase/8/> (6. Juli 2016)

<sup>3</sup> vgl. <http://tsusiatsoftware.net/jts/main.html> (6. Juli 2016)

<sup>4</sup> Für eine Einführung vgl. [https://en.wikipedia.org/wiki/Minkowski\\_addition](https://en.wikipedia.org/wiki/Minkowski_addition) (6. Juli 2016).

```

1 // Auszug aus der Klasse: ch.geomo.tramaps.conflict.buffer.EdgeBuffer
2 private void updateBuffer() {
3     double width = edge.calculateEdgeWidth(routeMargin) + edgeMargin * 2;
4     buffer = GeomUtil.createBuffer(edge.getLineString(), width / 2, true);
5 }

```

**Listing 6.1:** Berechnung des Konfliktbuffers

```

1 // Auszug aus der Klasse: ch.geomo.tramaps.graph.Edge
2 public double calculateEdgeWidth(double routeMargin) {
3     double width = getRoutes().stream()
4         .mapToDouble(Route::getLineWidth)
5         .sum();
6     return width + routeMargin * (getRoutes().size() - 2);
7 }

```

**Listing 6.2:** Berechnung des Platzbedarfs einer Kante

D.4.2). Für die Visualisierung der Haltestellen-Polygone werden in diesem Abschnitt einfache Rechtecke verwendet. Die Seitenlänge des Rechtecks berücksichtigt die Anzahl Routen je Seite, wobei die Routen, welche diagonal von der Haltestelle wegführen, jeweils der linken bzw. rechten Seite zugeordnet werden. Eine alternative Implementierung zeigt die Klasse `RectangleStationSignature`, welche die Haltestelle als Rechteck repräsentiert (vgl. Anhang D.4.2).

Sobald alle Buffer (repräsentiert durch die Klassen `EdgeBuffer` und `NodeBuffer`) erstellt wurden, werden die Konflikte mit der Methode `getBufferConflicts()` der Klasse `ConflictFinder` ermittelt (vgl. Listing 6.4).

Die Konfliktpaare werden mit der Methode `toPairSet(Predicate)` der Schnittstelle `EnhancedSet`<sup>5</sup> erstellt und nach den im Kapitel 5.1.1 aufgestellten Regeln, nämlich in welchem Fall ein Konflikt zwischen zwei Elementen existiert, mit einem Predicate (vgl. Konstante `CONFLICT_PAIR_PREDICATE` in Listing 6.4) gefiltert .

```

1 // Auszug aus der Klasse: ch.geomo.tramaps.map.signature.SquareStationSignature
2 @Override
3 public void updateSignature() {
4     // gets all adjacent edges of node and calculates the required space
5     double width = node.getAdjacentEdges().stream()
6         .map(edge -> edge.calculateEdgeWidth(ROUTE_MARGIN))
7         .max(Double::compare)
8         .orElse(ROUTE_MARGIN);
9     // creates a convex polygon (no convex hull needed anymore)
10    signature = GeomUtil.createPolygon(node.getPoint(), width, width);
11    // ...
12 }

```

**Listing 6.3:** Berechnung der Haltestellensignatur

<sup>5</sup> Die Schnittstelle `EnhancedSet` wird von der Klasse `GSet` implementiert. `GSet` ist erweiterte Implementierung der Klasse `HashSet`.



```

1 // Auszug aus der Klasse: ch.geomo.tramaps.conflict.ConflictFinder
2 private final static Predicate<Pair<ElementBuffer>> CONFLICT_PAIR_PREDICATE = (Pair<ElementBuffer> pair) -> {
3     ElementBufferPair bufferPair = new ElementBufferPair(pair);
4     if (bufferPair.hasEqualElements()) {
5         return false;
6     }
7     else if (!bufferPair.hasAdjacentElements()) {
8         return true;
9     }
10    return bufferPair.isNodePair();
11 };
12
13 @NotNull
14 private EnhancedSet<Pair<ElementBuffer>> getConflictElements() {
15     EnhancedSet<Pair<ElementBuffer>> buffers = GSet.createSet(createEdgeBuffers(), createNodeBuffers());
16     return buffers.toPairSet(ConflictFinder.CONFLICT_PAIR_PREDICATE);
17 }
18
19 private static boolean intersects(@NotNull Pair<ElementBuffer> bufferPair) {
20     return bufferPair.getFirst().getBuffer().relate(bufferPair.getSecond().getBuffer(), "T*****");
21 }
22
23 @NotNull
24 private EnhancedList<Conflict> getBufferConflicts() {
25     return getConflictElements().stream()
26         .filter(ConflictFinder::intersects)
27         .map(BufferConflict::new)
28         .filter(BufferConflict::isNotSolved)
29         .distinct()
30         .collect(GCollectors.toList());
31 }

```

**Listing 6.4:** Ermittlung der Konflikte zwischen Elementen

Damit die Gleichung 5.2, also die Überschneidung der Innenfläche zweier Polygone, erfüllt ist, wird eine Implementierung des DE-9IM Modells benötigt. Diese Prüfung wird mit der Methode `intersects(Pair)` in Zeile 26 des Listings 6.4 durchgeführt. Dazu wird die Methode `relate(otherGeometry, pattern)` der Klasse `Geometry` verwendet.

Die in der Zeile 27 im Listing 6.4 erstellte Instanz der Klasse `BufferConflict` übernimmt nun die Clip-Operation und erstellt das Konfliktpolygon. Die Klasse `BufferConflict` ist verantwortlich für alle Informationen, die für die Lösung des Konfliktes nötig sind und stellt diese über eine entsprechende Schnittstelle zur Verfügung (vgl. Anhang D.1.1).

## 6.2 Skalierung

Die im Listing 2 beschriebene Ermittlung des Skalierungsfaktors wird in der Methode `evaluateScaleFactor(EnhancedList<Conflict>)` in der Klasse `ScaleHandler` umgesetzt (vgl. Listing 6.5 und Anhang D.3.1).

```

1 // Auszug aus der Klasse: ch.geomo.tramaps.map.displacement.scale.ScaleHandler
2 private double evaluateScaleFactor(@NotNull EnhancedList<Conflict> conflicts) {
3     double maxMoveX = conflicts.stream()
4         .map(conflict -> {
5             Envelope bbox = conflict.getElementBoundingBox();
6             double alongX = conflict.getDisplaceDistanceAlongX();
7             if (bbox.getWidth() != 0) {
8                 return (bbox.getWidth() + alongX) / bbox.getWidth();
9             }
10            return 1d;
11        })
12        .max(Double::compare)
13        .orElse(1d);
14    double maxMoveY = conflicts.stream()
15        .map(conflict -> {
16            Envelope bbox = conflict.getElementBoundingBox();
17            double alongY = conflict.getDisplaceDistanceAlongY();
18            if (bbox.getHeight() != 0) {
19                return (bbox.getHeight() + alongY) / bbox.getHeight();
20            }
21            return 1d;
22        })
23        .max(Double::compare)
24        .orElse(1d);
25    return Math.max(GeomUtil.makePrecise(Math.max(maxMoveX, maxMoveY)), 1.0001);
26 }

```

Listing 6.5: Berechnung des Skalierungsfaktors

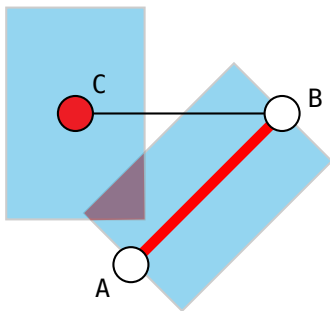


Abbildung 6.1: Spezieller Fall eines Konflikts zwischen Kante und Knoten. Knoten B der Konfliktkante ist adjazent zum Konfliktknoten C. Rot dargestellt sind die Konfliktelemente und das Konfliktpolygon. (eigene Grafik)

### 6.2.1 Optimierungen

#### Konflikte zwischen Knoten und diagonalen Kanten

Die Ausführung des Algorithmus sollte theoretisch alle Konflikte mit einer einzigen Iteration auflösen. Dies funktioniert mit allen Konfliktarten, ausser mit einem Konflikt bestehend aus einem Knoten und einer Kante, die einen adjazenten Knoten besitzt, der ebenfalls zum Konfliktknoten adjazent ist (vgl. Abbildung 6.1). In diesem Fall wird der Konflikt nur teilweise gelöst und es werden mehrere Iterationen benötigt bis der Konflikt vollständig gelöst werden kann. Um die Anzahl Iterationen mit sehr kleinen Faktoren zu senken, wird der Skalierungsfaktor auf den Wert 1.0001 aufgerundet (vgl. Zeile 25 im Listing 6.5).

## 6.3 Verdrängungsalgorithmus

Der folgende Abschnitt geht auf die Besonderheiten der Implementierung des Verdrängungsalgorithmus ein. Aufgrund der Vielzahl von Codezeilen wird in diesem Kapitel jeweils

```

1 // Auszug aus der Klasse: ch.geomo.util.geom.PolygonUtil
2 /**
3  * Returns a {@link Stream} set {@link LineString}s which are parallel to given
4  * {@link LineString} and within given {@link Polygon} while one endpoint is equal
5  * to a vertex and the other end point of the line string lies on the exterior
6  * set the polygon.
7  */
8 @NotNull
9 public static Stream<LineString> findParallellineString(@NotNull Polygon inPolygon,
10                                                         @NotNull LineString parallelTo) {
11     Envelope envelope = inPolygon.getEnvelopeInternal();
12     // scale line string order to be long enough to intersect with the polygons exterior
13     double factor = Math.max(envelope.getHeight(), envelope.getWidth()) * 2;
14     AffineTransformation scaleTransformation = new AffineTransformation();
15     scaleTransformation.scale(factor, factor);
16     Point endPoint = GeomUtil.createPoint(scaleTransformation.transform(parallelTo.getEndPoint()));
17     LineString scaledLineString = GeomUtil.createLineString(parallelTo.getStartPoint(), endPoint);
18     return Arrays.stream(inPolygon.getCoordinates())
19         .sequential()
20         // create a parallel line for each vertex
21         .map(vertex -> {
22             AffineTransformation translateTransformation = new AffineTransformation();
23             translateTransformation.translate(
24                 vertex.x - scaledLineString.getCentroid().getX(),
25                 vertex.y - scaledLineString.getCentroid().getY());
26             return translateTransformation.transform(scaledLineString);
27         })
28         .filter(Geometry::isValid)
29         // get line string within polygon
30         .filter(inPolygon::intersects)
31         .map(inPolygon::intersection)
32         .filter(geom -> geom instanceof LineString && !geom.isEmpty())
33         .map(geom -> (LineString) geom);
34 }
35
36 /**
37  * Finds the longest parallel {@link LineString} parallel to the given {@link LineString}
38  * and within the given {@link Polygon}.
39  */
40 @NotNull
41 public static Optional<LineString> findLongestParallellineString(@NotNull Polygon inPolygon,
42                                                                 @NotNull LineString parallelTo) {
43     return PolygonUtil.findParallellineString(inPolygon, parallelTo)
44         .max((l1, l2) -> Double.compare(l1.getLength(), l2.getLength()));
45 }

```

Listing 6.6: Erstellen des Verdrängungsvektors

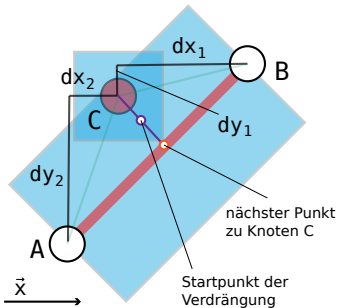
auf den Anhang verwiesen, um den Lesefluss nicht zu behindern.

### 6.3.1 Ermittlung des Verdrängungsvektors

Die Ermittlung des Verdrängungsvektors ist Aufgabe der Klasse `BufferConflict`<sup>6</sup>. Zuerst wird  $q$  bestimmt, indem mit der Methode `createQ()` eine Linie zwischen den Mittelpunkten beider Elemente erstellt wird. Anschliessend wird mit dem Konfliktpolygon und der Methode `findLongestParallellineString(Polygon, LineString)` der Hilfsklasse `PolygonUtil` (vgl. Listing 6.6) die parallele Linie  $q'$  ermittelt. Mit dem Ergebnis wird der Verdrängungsvektor<sup>7</sup> erstellt.

<sup>6</sup> Vgl. auch die abstrakte Klasse `AbstractConflict` im Anhang D.1.1.

<sup>7</sup> Die Klasse `MoveVector` erlaubt es einen Vektor aus einem `LineString`-Objekt zu erzeugen.



**Abbildung 6.2:** Beispiel für die Ermittlung der Verdrängungsgeraden bei einem Kante/Knoten-Konflikt. In diesem Fall ist die Verdrängungsrichtung entlang der X-Achse, da  $\max(dx_1, dx_2) < \max(dy_1, dy_2)$ . (eigene Grafik)

### 6.3.2 Ermittlung der Verdrängungsgerade

#### Knoten/Knoten- und Kante/Kante-Konflikte

Die Verdrängungsgerade wird in dieser Implementierung durch einen Punkt und der Verdrängungsrichtung definiert. Zusammen mit diesem Startpunkt (der Verdrängung) wird der beste Verdrängungsvektor entlang der X- oder Y-Achse ermittelt. Bei Konflikten mit gleichen Graph-Elementen (Kante/Kante oder Knoten/Knoten) wird die Projektion (entlang der X- oder Y-Achse,  $\vec{v}_x$  oder  $\vec{v}_y$ ) mit dem kleinsten Winkel zum Vektor  $\vec{v}$  gewählt (vgl. Abbildung 5.14) und somit auch die Verdrängungsrichtung bestimmt.

Der Startpunkt eines Knoten/Knoten-Konflikts ist der Mittelpunkt der Linie zwischen beiden Knoten. Bei einem Kante/Kante-Konflikt liegt der Startpunkt auf dem Mittelpunkt der gemeinsamen Fläche des Konfliktpolygons und der konvexen Hülle der beiden Kanten. Dies soll sicherstellen, dass der Punkt immer zwischen den beiden Kanten liegt<sup>8</sup>.

#### Kante/Knoten-Konflikte

Die noch verbleibende Konfliktart ist der Kante/Knoten-Konflikt. Wenn der Knoten eindeutig im Norden, Süden, Westen oder Osten der Kante liegt, also ausserhalb der Bounding Box der Kante, kann die Verdrängungsrichtung einfach bestimmt werden (Norden/Süden  $\hat{=}$  Verdrängung nach Norden; Westen/Osten  $\hat{=}$  Verdrängung nach Osten).

Handelt es sich jedoch um eine diagonale Kante und liegt der Knoten innerhalb der Bounding Box der Kante, wird eine etwas aufwändigere Methode benötigt. In diesem Fall müssen die Deltas der x- und y-Werte zwischen dem Knoten und den beiden adjazenten Knoten der Kante berechnet werden. Ist nun der grössere Wert der beiden Deltas des x-Wertes kleiner als der grössere Wert der beiden Deltas des y-Wertes, so wird entlang der X-Achse verschoben, andernfalls entlang der Y-Achse (vgl. Abbildung 6.2 und die Klasse `BufferConflict` in Anhang D.1.1).

<sup>8</sup> Für die Implementierung vgl. Klasse `BufferConflict` im Anhang D.1.1.

Für den Startpunkt der Verdrängung wird bei dieser Konfliktart der Mittelpunkt der Linie zwischen dem Knoten und dem am nächsten liegenden Punkt auf der Kante gewählt (vgl. Anhang D.1.1).

### 6.3.3 Verdrängungsphase

Die Klasse `NodeDisplacer` übernimmt die Aufgabe der Konfliktlösung durch Verdrängung und ist somit das Kernstück der Verdrängungsphase. Abhängig von der besten Verdrängungsrichtung (vgl. Abschnitt 6.3.2) und des Startpunkt der Verdrängung werden mit Hilfe dieser Klasse alle Punkte, die sich (je nach Verdrängungsrichtung) östlich oder nördlich des Startpunktes befinden, verdrängt bzw. verschoben.

Die im Kapitel 5.5.1 beschriebenen und in der Abbildung 5.15 dargestellten Ausnahmen werden ebenfalls berücksichtigt, sofern es sich nicht um eine konfliktrelevante Kante handelt. Unter einer konfliktrelevanten Kante ist die aktuelle Konfliktkante oder eine Kante zu verstehen, welche adjazent zu einem Konfliktelement ist (vgl. hierzu Methode `checkConnectionEdge(Node, boolean)` der Klasse `NodeDisplacer` im Anhang D.2.1).

Ändert sich die Position eines Knotens (vgl. Anhang D.4.4, `Node`), so werden abhängige Objekte (unter anderem die Kanten, vgl. Anhang D.4.4, `Edge`) durch die Implementierung des Observer-Pattern informiert und müssen daher nicht manuell aktualisiert werden. Das Observer-Pattern ist ein Entwurfsmuster aus der Software Entwicklung bei dem der Beobachter bei Änderungen benachrichtigt wird.

### 6.3.4 Korrekturphase

Die für die Korrekturphase relevanten Klassen sind `CostCalculator`, `EdgeAdjuster` und `DirectionEvaluator` sowie die Klasse `OctilinearEdgeBuilder` für das Einführen von Knicken<sup>9</sup>.

Die Klasse `CostCalculator` berechnet die Anpassungskosten  $\kappa$  für einen Korrekturknoten und implementiert das Listing 5. Die Berechnung der Kosten wird jedoch geringfügig angepasst. So wird zwischen einfachen Korrekturknoten mit einem Grad von 2 und 3 unterschieden, die jeweils einem anderen Wert zugeordnet werden (vgl. Anhang D.2.1). Diese Anpassung wurde eingeführt, um einfache Korrekturknoten mit einem Grad von 2 gegenüber von Korrekturknoten mit einem Grad von 3 bevorzugt zu behandeln.

---

<sup>9</sup> Für die Implementierung der Klassen vgl. Anhang D.

### Korrektur durch Verschieben

Das Verschieben der Knoten wird mit der Methode `moveNode(Edge, Node)` der Klasse `EdgeAdjuster` durchgeführt (vgl. Anhang D.2.1, `EdgeAdjuster`). Diese evaluiert in einem ersten Schritt, mit Hilfe der Klasse `DirectionEvaluator` die Verschieberichtung für einfache Korrekturknoten<sup>10</sup> (vgl. Anhang D.2.1, `DirectionEvaluator`). Anschliessend wird die neue Position des Knotens auf Überschneidungen mit anderen Graph-Elementen überprüft. Verschoben wird nur, falls keine Überschneidung existiert. Nach dem Verschieben wird überprüft, ob die neue Position einen Konflikt verursacht. Wenn ja, wird die alte Position wiederhergestellt. Dieses Verhalten muss eingeführt werden, da sonst die Korrektur durch die Verdrängungsphase unter Umständen rückgängig gemacht wird und dies zu einer Endlosschleife führt.

### Korrektur durch Knicke

Hat eine Verbindungskante  $e_d$  keine einfachen Korrekturknoten und sind die Anpassungskosten  $\kappa$  grösser als die maximalen Anpassungskosten  $\iota$ , so wird die Oktilinearität der Kante durch zusätzliche Knicke wiederhergestellt.

Diese Aufgabe übernehmen die Methode `correctEdgeByIntroducingBendNodes()` der Klasse `EdgeAdjuster` und die Klasse `OctilinearEdgeBuilder` (vgl. Anhang D.2.2). Da in der vorliegenden Arbeit keine komplexe Korrekturknoten verschoben werden, kann angenommen werden, dass die maximalen Anpassungskosten  $\iota$  immer 2 betragen. Unter Umständen kann es gewollt sein, dass erst bei sehr komplexen Korrekturknoten die Oktilinearität durch Knicke wiederhergestellt wird<sup>11</sup>.

## 6.4 Optimierungen

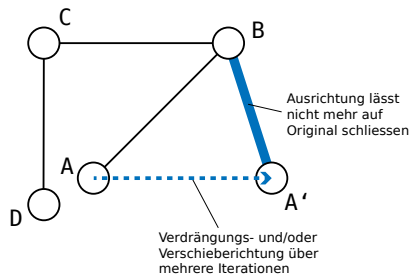
Bei der Implementierung des Verdrängungsalgorithmus müssen verschiedene Optimierungen vorgenommen werden. Zu den beiden wichtigsten Optimierungen gehören die Bearbeitungsreihenfolge der Konflikte und die Einführung einer neuen Konfliktart zur Sicherstellung der Erhaltung der ursprünglichen Ausrichtung einer Kante.

### 6.4.1 Bearbeitungsreihenfolge der Konflikte

Bei der Implementierung stellt sich heraus, dass die Bearbeitungsreihenfolge der Konflikte für das Ergebnis relevant ist. Damit eine geeignete Reihenfolge ermittelt werden kann, wird die Klasse `ConflictComparator` (vgl. Anhang D.1.1, `ConflictComparator`) eingeführt. Sie ist dafür verantwortlich, dass die Konflikte unter anderem gemäss einem

<sup>10</sup> Das Verschieben von komplexen Korrekturknoten wird in der vorliegenden Arbeit nicht behandelt.

<sup>11</sup> Die Oktilinearität wird auf jeden Fall mit der Korrektur oktilinearer Konflikte wiederhergestellt. Vgl. hierzu Kapitel 6.4.2.



**Abbildung 6.3:** Beispiel für den Verlust der ursprünglichen Ausrichtung. Die Kante  $\overline{AB}$  ist nach mehreren Iterationen nicht mehr nach Südwesten ausgerichtet sondern (dargestellt mit  $\overline{A'B}$ ) nach Südosten. (eigene Grafik)

Konflikt-Ranking (Klassifizierung) sortiert werden. Das Ranking der einzelnen Konflikttypen ist im Enum `ConflictType` definiert (vgl. Anhang D.1.1, `ConflictType`).

Durch die Auswertung verschiedener Rankings stellt sich heraus, dass durch die höhere Priorisierung der Knoten-Konflikte bessere Resultate erzielt werden können. Wird die Verarbeitung der Kanten zuerst durchgeführt, kann es sein, dass je nach Ausgangslage bzw. Graph keine Lösung gefunden wird. Dies kann insbesondere auftreten, wenn sehr wenig Raum für Korrekturen vorhanden ist und die Korrekturphase dadurch das Ergebnis der Verdrängungsphase beeinträchtigt wird. Werden Knoten-Konflikte zuerst gelöst, so wird schneller zusätzlicher Raum für Korrekturen geschaffen<sup>12</sup>.

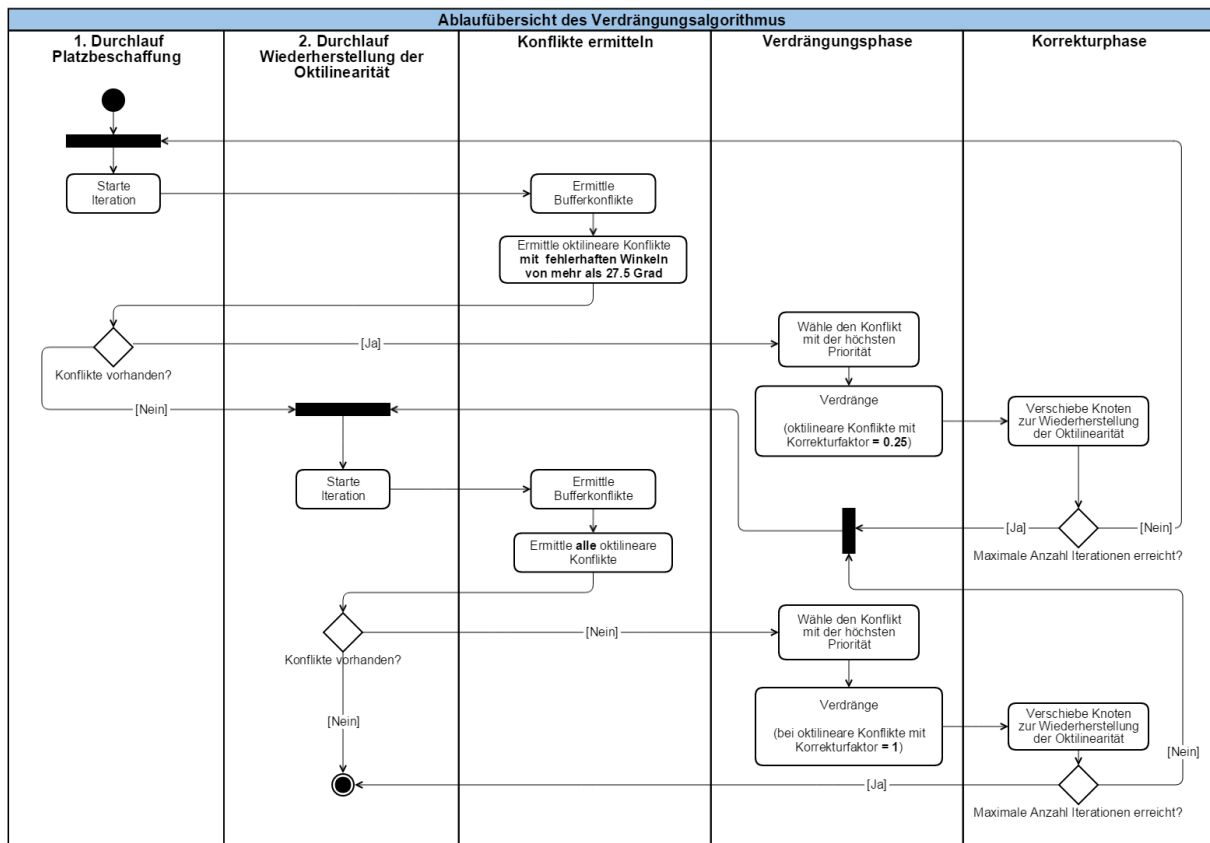
### 6.4.2 Oktilineare Konflikte

Da während der Korrekturphase nicht immer alle diagonalen Kanten oktilinear korrigiert werden können, kann es vorkommen, dass die diagonale Kante in der nächsten Iteration einen Winkel um mehr als 45 Grad gegenüber der originalen Ausrichtung hat. Um dies zu verhindern, werden in der Verdrängungsphase diese Winkel so korrigiert, dass der Winkel zur originalen Ausrichtung nie mehr als 45 Grad beträgt. Andernfalls könnte beispielsweise eine Kante, die ursprünglich nach Südwesten ausgerichtet war, nach Südosten ausgerichtet sein. Eine Korrektur dieser Kante ist unter Umständen nicht mehr möglich, nämlich spätestens dann, wenn ein anderer Konflikt diese Kante ebenfalls beeinflusst und einer der Endknoten womöglich verschoben wird (vgl. Abbildung 6.3).

Der Name "oktilinearer Konflikt" bezeichnet in diesem Fall die fehlerhafte Ausrichtung einer Kante. Eine solche fehlerhafte Ausrichtung wird in der Verdrängungsphase wie jeder andere Konflikt (Bufferkonflikt) behandelt und korrigiert.

Theoretisch könnten iterativ alle nicht-oktilinearen Kanten oktilinear ausgerichtet werden. Dies ist aber nicht sinnvoll, solange Bufferkonflikte vorhanden sind, da die vollständige Korrektur der oktilinearen Konflikte mit jeder Iteration aufgrund der notwendigen Verdrängung die Ausdehnung der Karte (ähnlich einer Skalierung) beeinflusst.

<sup>12</sup> Unter Umständen kann eine Anpassung der Priorisierung bei anderen Eingabedaten ein noch besseres Resultat erzielen.



**Abbildung 6.4:** Ablaufübersicht des Verdrängungsalgorithmus. (eigene Grafik)

Damit die ursprüngliche Ausrichtung einer Kante erhalten bleiben kann, ohne eine zu grosse Ausdehnung der Karte in Kauf zu nehmen, wird der Verdrängungsalgorithmus zweimal durchlaufen. Der erste Durchlauf (Platzbeschaffung) berücksichtigt nur oktilineare Konflikte mit einem fehlerhaften Winkel von mehr als 27.5 Grad und korrigiert den Winkel nicht vollständig. Das Mass der Winkelkorrektur kann über den Korrekturfaktor gewählt werden. In der Implementierung wird hierfür der Faktor 0.25 verwendet, wobei der Faktor 1 vollständige Oktilinearität herstellen würde. Der relativ kleine Faktor stellt sicher, dass die Ausrichtung der Kanten soweit wie möglich erhalten bleibt ohne die Karte zu sehr auszudehnen. Der zweite Durchlauf (Wiederherstellung der Oktilinearität) beginnt, sobald alle Bufferkonflikte gelöst sind, und korrigiert die verbleibenden oktilinearen Konflikte mit dem Korrekturfaktor 1. Dabei handelt es sich um alle nicht-oktilinearen Kanten, deren Oktilinearität während der Korrekturphase nicht hergestellt werden konnte. Unter Umständen entstehen neue Bufferkonflikte, die im gleichen Durchlauf wieder mit einer Verdrängungs- und Korrekturphase gelöst werden. Eine Übersicht des Ablaufs ist in der Abbildung 6.4 dargestellt<sup>13</sup>.

<sup>13</sup> Für die Implementierung vgl. insbesondere die Klassen `DisplaceLineSpaceHandler`, `ConflictFinder` und `OctilinearConflict` im Anhang D.



## 6.5 Visualisierung

Für die Visualisierung wird JavaFX 8 verwendet. Die Klasse `Met roMapDrawer` (vgl. Anhang D.4.2) ist für die Darstellung verantwortlich.

## 6.6 Spezielle Probleme bei der Implementierung

Im folgenden Abschnitt werden Probleme bei der Implementierung aufgezeigt, die in speziellen Fällen auftreten, und es werden mögliche Lösungen diskutiert.

### 6.6.1 Numerische Präzision der Koordinaten

Bei der Anwendung der Verdrängung treten Koordinatenwerte auf, die viele Nachkommastellen aufweisen. Die Überprüfung der räumlichen Relation könnte deswegen eine `TopologyException` werfen, nämlich dann, wenn der Algorithmus (für die Überprüfung der räumliche Relation) die betreffenden Koordinaten nicht korrekt zuordnen kann. Als Workaround lässt sich die numerische Präzision eingeschränken<sup>14</sup>. Dies kann bei Bedarf in der Klasse `GeomUtil` vorgenommen werden, indem die Konstante `PRECISION_MODEL` angepasst wird (vgl. Anhang D.6.2).

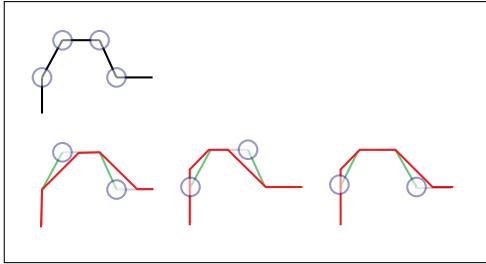
### 6.6.2 Wiederholbarkeit

Die Klasse `ConflictComparator` (vgl. Anhang D.1.1) ist verantwortlich, dass die Priorität der Konflikte und somit die Bearbeitungsreihenfolge einheitlich und konsistent ist, sodass der Algorithmus mit den gleichen Eingabedaten jedesmal das gleiche Resultat erzielt. Mit der Einführung des oktilinearen Konflikts (vgl. Klasse `OctilinearConflict` im Anhang D.1.1) ist dieselbe Reihenfolge nicht in jedem Fall gewährleistet. Insbesondere, wenn zwei oder mehr oktilineare Konflikte den gleichen Verschiebevektor besitzen, haben sie die gleiche Wertigkeit. Da aber diese Konflikte immer die vorhergehende und somit gemeinsame Verdrängungsgerade schneiden, hat dies im ersten Durchlauf (Platzbeschaffung, vgl. Abbildung 6.4) keinen Einfluss auf das Endresultat. Anders sieht es im zweiten Durchlauf (Wiederherstellung der Oktilinearität, vgl. Abbildung 6.4) aus. In diesem Fall könnte aber der `ConflictComparator` zusätzlich die Ausrichtung der Kante und die Entfernung der Kante zum Nullpunkt berücksichtigen<sup>15</sup>.

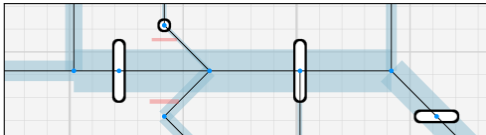
Ein weiterer kritische Punkt für die Wiederholbarkeit des Algorithmus ist die Korrekturphase. Die Bearbeitungsreihenfolge der nicht-oktilinearen Kanten muss ebenfalls nach einer festen Priorisierung erfolgen. Um bei einer Wiederholung die gleiche Reihenfolge

<sup>14</sup> vgl. auch <http://tsusiatsoftware.net/jts/jts-faq/jts-faq.html#D9> (25. September 2016)

<sup>15</sup> Diese Erweiterung wird nicht mehr im Rahmen der vorliegenden Arbeit umgesetzt.



**Abbildung 6.5:** Bearbeitungsreihenfolge der Konflikte und Knoten während der Korrekturphase. Grün eingezeichnet sind die zu korrigierenden Kanten. Die rot eingezeichneten Kantenverläufe unterscheiden sich dadurch, welche der blau umkreisten Knoten für die Korrektur gewählt werden. Dies illustriert die Relevanz der Bearbeitungsreihenfolge. (eigene Grafik)



**Abbildung 6.6:** Konflikte (rot) zwischen einem Knoten und einer Kante, welche adjazent zu einer Kante ist, die wiederum adjazent zum Konfliktknoten ist (vgl. Abbildung 6.1). Konflikte entstehen durch die Margins. (eigene Grafik)

zu erhalten, wird die Klasse `NonOctilinearEdgeComparator` eingeführt<sup>16</sup> (vgl. Anhang D.2.1). Die Abbildung 6.5 zeigt an einem einfachen Beispiel die Auswirkung auf das Layout, falls die Kanten nicht in der gleichen Reihenfolge abgearbeitet werden.

Weiter hat auch die Wahl des Korrekturknoten bei identischen Anpassungskosten  $\kappa$  einen Einfluss auf das Layout (vgl. Abbildung 6.5). Daher wird die Summe der Länge aller adjazenten Kanten (ohne die zu korrigierende Kante) der beiden Korrekturknoten verglichen und derjenige Korrekturknoten bevorzugt, der die grössere Summe aufweist (vgl. Klasse `EdgeAdjuster` im Anhang D.2.1), da in diesem Fall die Wahrscheinlichkeit grösser ist, dass mehr Platz für Korrekturen vorhanden ist.

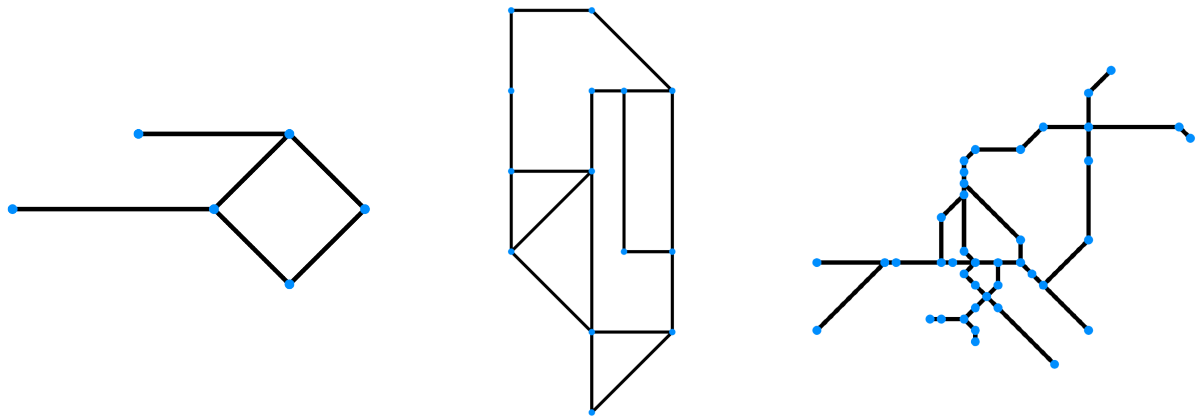
### 6.6.3 Konflikte zwischen Knoten und diagonalen Kanten

Sowohl bei der Skalierung als auch beim Verdrängungsalgorithmus treten Schwierigkeiten auf bei Konflikten bestehend aus einem Knoten und einer Kante, die einen adjazenten Knoten besitzt, der ebenfalls zum Konfliktknoten adjazent ist (vgl. Abbildung 6.1 und 6.6). Ein solcher Konflikt kann meist nur durch sehr viele Iterationen oder durch manuelles Eingreifen gelöst werden.

## 6.7 Resultate

Im folgenden Abschnitt wird die oben beschriebene Implementierung auf drei Graphen angewendet. Die Abbildung 6.7 zeigt die jeweiligen Ausgangslagen. Die Testdaten sind im Anhang B verfügbar.

<sup>16</sup> Die Klasse `NonOctilinearEdgeComparator` wird bei der Implementierung wie beschrieben verwendet, aber im Rahmen der vorliegenden Arbeit nicht weiter ausgearbeitet.



(a) Beispiel 1 aus Kapitel 5

(b) Beispiel 2

(c) S-Bahnnetz der Stadt Zürich

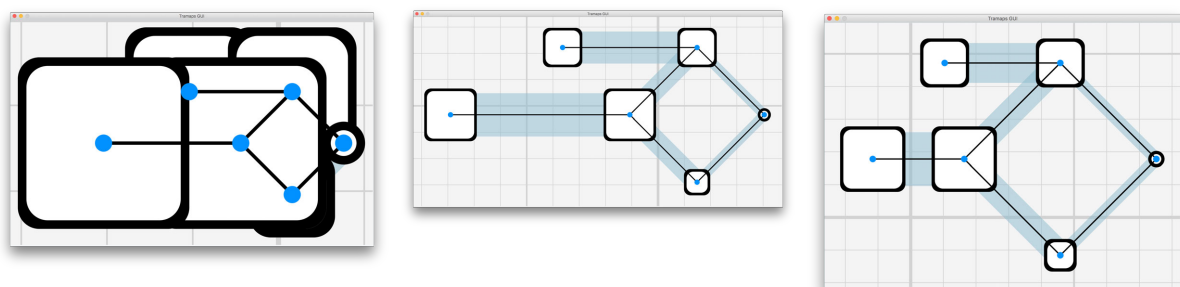
**Abbildung 6.7:** Ausgangslage Beispiele. (eigene Grafik)

### 6.7.1 Beispiel 1

Als einleitendes Beispiel wird der Graph aus der Abbildung 5.1a im Kapitel 5 verwendet<sup>17</sup>. Die Skalierung ist in der Abbildung 6.8b dargestellt. Das Resultat ist winkeltreu und die Topologie unverändert gegenüber der Ausgangslage. Der Verdrängungsalgorithmus hingegen staucht die Zeichnung etwas (vgl. Abbildung 6.8c).

### 6.7.2 Beispiel 2

Einen etwas komplexeren Graphen zeigt die Abbildung 6.7b<sup>18</sup>. Die Skalierung ist in der Abbildung 6.9b dargestellt. Das Resultat ist wiederum winkeltreu und die Topologie unverändert gegenüber der Ausgangslage. Der Verdrängungsalgorithmus hat in diesem Beispiel Knicke (linke Seite) eingeführt und staucht das Resultat im oberen Bereich (vgl. Abbildung 6.8c). Die eingeführten Knicke entsprechen in diesem Fall den Varianten 4a und 2b aus der Abbildung 5.20.



(a) Ohne Verdrängung

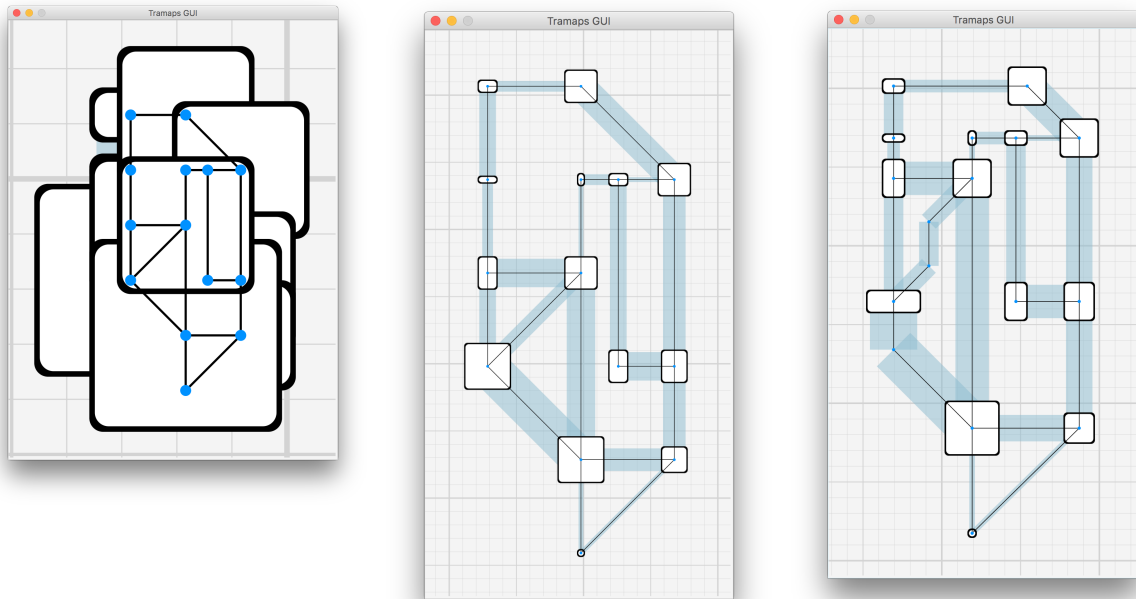
(b) Mit Skalierung

(c) Mit Verdrängungsalgorithmus

**Abbildung 6.8:** Anwendung der vorgestellten Ansätze am Beispiel aus Kapitel 5. (eigene Grafik)

<sup>17</sup> Das Beispiel 1 wird durch die Klasse `MetroMapChapterFive` repräsentiert (vgl. Anhang D.5.1).

<sup>18</sup> Das Beispiel 2 wird durch die Klasse `MetroMapExampleGraph` repräsentiert (vgl. Anhang D.5.1).

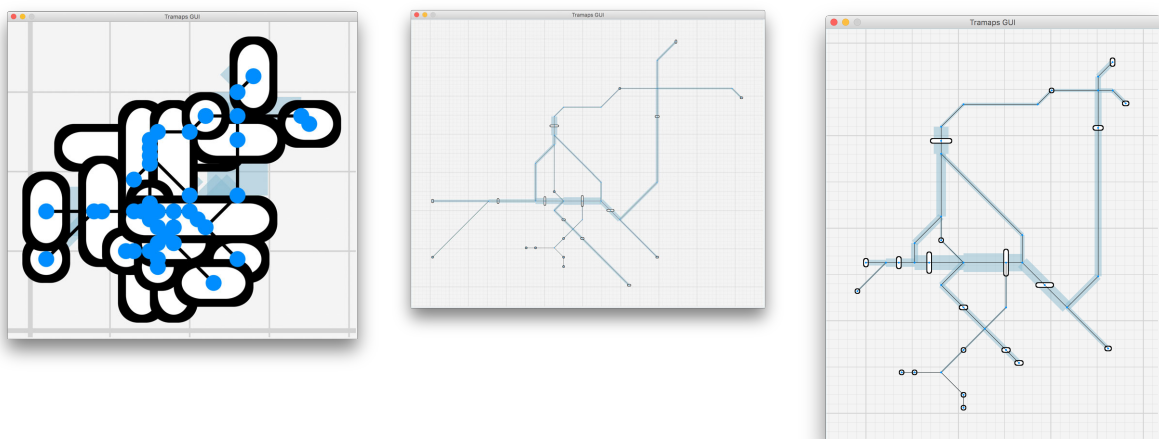


(a) Ohne Verdrängung

(b) Mit Skalierung

(c) Mit Verdrängungsalgorithmus

**Abbildung 6.9:** Anwendung der vorgestellten Ansätze auf einen komplexeren Graph. Die Ausdehnung nach Anwendung der Skalierung ist 835x1805 (zweite Abbildung) und nach Anwendung des Algorithmus 729x1488 (dritte Abbildung). (eigene Grafik)



(a) Ohne Verdrängung

(b) Mit Skalierung

(c) Mit Verdrängungsalgorithmus

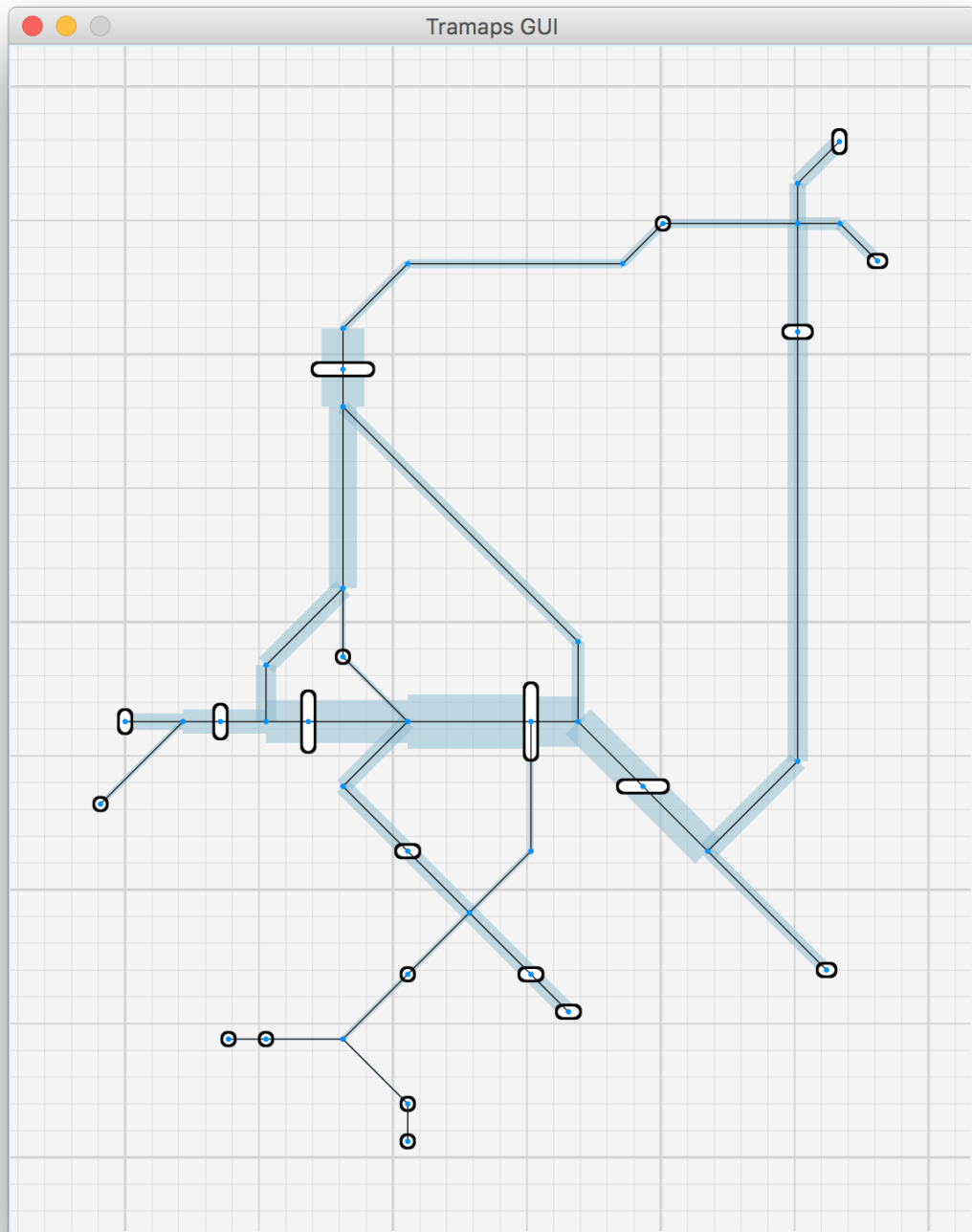
**Abbildung 6.10:** Anwendung der vorgestellten Ansätze auf das S-Bahn-Streckennetz der Stadt Zürich. Die Ausdehnung nach Anwendung der Skalierung ist 4077x3222 (zweite Abbildung) und nach Anwendung des Algorithmus 1476x1898 (dritte Abbildung). Die dritte Abbildung wird vergrößert in der Abbildung 6.11 dargestellt. (eigene Grafik)

### 6.7.3 Beispiel 3 (S-Bahnstreckennetz der Stadt Zürich)

In der Abbildung 6.7c wird das S-Bahnstreckennetz der Stadt Zürich dargestellt<sup>19</sup>. Das Resultat der Skalierung in der Abbildung 6.10b ist auch hier winkeltreu und die Topologie unverändert gegenüber der Ausgangslage. Es ist aber gut erkennbar, dass die ganze Karte stark ausgedehnt wird, während der Verdrängungsalgorithmus (vgl. Abbildung 6.10c) eine deutlich kleinere Ausdehnung produziert. Die Einsparung entsteht primär in den Peripherien, doch auch im Zentrum gibt es Einsparungen, vor allem dann, wenn eine Möglichkeit besteht die Knoten in der Korrekturphase zu verschieben. Der Vorteil des Verdrängungsalgorithmus gegenüber der Skalierung ist die kleiner Ausdehnung der Karte. Ein Nachteil ist es, dass er die Metro Map Layout Regel 5 (möglichst gleichmässiger Abstand der Haltestellen, vgl. Anhang A) nicht berücksichtigt.

---

<sup>19</sup> Das Beispiel 3 wird durch die Klasse `MetroMapZuerich` repräsentiert (vgl. Anhang D.5.1).



**Abbildung 6.11:** Anwendung des Verdrängungsalgorithmus auf das S-Bahn-Streckennetz der Stadt Zürich. Vergrösserte Darstellung. (eigene Grafik)

## 7 | Diskussion und Ausblick

Die vorliegende Arbeit hat zum Ziel Möglichkeiten für die automatisierte Nachbearbeitung von Liniennetzplänen aufzuzeigen. Dazu wird eine Analyse der graphischen Anforderungen eines Liniennetzplans und der bereits existierenden Werkzeuge und Ansätze durchgeführt. Weiter werden mögliche Vorgehensweisen für das manuelle Zeichnen von Liniennetzplänen präsentiert. Diese Vorgehensweisen fokussieren die Darstellung vieler Routen auf dem gleichen Streckensegment und die Probleme mit dem Platzbedarf, die sich dadurch ergeben. Es zeigt sich, dass bei der Erstellung eines Liniennetzplanes der Platzbedarf der Routeninformationen schwer abzuschätzen ist und dadurch mehrere Iterationen benötigt werden, um alle Routen wie gewünscht darzustellen. Der Kartenersteller muss dazu viel Zeit aufwenden, insbesondere, wenn eine nachträgliche Änderung vorgenommen werden soll.

Die vorliegende Arbeit greift diesen Anwendungsfall auf und diskutiert Ansätze der Verdrängung in einem Liniennetzplan. Dabei wird deutlich, dass bei einer Verdrängung in einem Liniennetzplan die Wiederherstellung der Oktilinearität eine Herausforderung darstellt. Um die Winkeltreue zu bewahren, wird deshalb zuerst ein Ansatz untersucht, der mit einer Skalierung der Punkte arbeitet und dadurch mehr Raum für die Routeninformationen zwischen den Haltestellen und Streckensegmenten schaffen soll. Durch die Anwendung der Skalierung werden jedoch die Peripherien ebenfalls vergrößert und sind unter Umständen stark überdimensioniert. Dies wirkt sich auf die Ausdehnung der ganzen Karte aus, welche dann bei Medien mit begrenzter Grösse (zum Beispiel eine Faltkarte) verkleinert dargestellt werden muss und dadurch unter Umständen kaum noch leserlich ist.

Daher wird ein zweiter Ansatz untersucht, welcher den benötigten Raum anhand einer Verdrängungsgerade durch das Verschieben aller Knoten auf einer Seite der Geraden beschafft. Dadurch entstehen jedoch nicht-oktilineare Kanten und es wird untersucht, wie die Oktilinearität wiederhergestellt werden kann.

Die vorgestellten Ansätze werden implementiert. Dabei stellt sich heraus, dass Optimierungen notwendig werden. Es treten Schwierigkeiten mit diagonalen Kanten auf, welche unter Umständen nach einigen Iterationen ihre Richtung ändern. Des Weiteren lässt sich feststellen, dass der Bearbeitungsreihenfolge der Konflikte eine nicht unbedeutende Rolle zukommt.

## 7.1 Beantwortung der Fragestellungen

### 7.1.1 Fragestellung 1

#### **Welche graphischen Anforderungen an einen Liniennetzplan existieren?**

Liniennetzpläne von Verkehrsnetzwerken sollen den Passagieren die Orientierung erleichtern. Zu diesem Zweck werden topographische Karten soweit vereinfacht, dass nur die notwendige Information ersichtlich ist. Es bestehen Metro Map Layout Regeln für die Darstellung von schematischen Liniennetzplänen. Solche Liniennetzpläne enthalten verschiedene graphische Elemente (Signaturen), welche die Netzwerkinformation sowie Landmarken und eventuell Points of Interest repräsentieren. In der Regel wird ein primäres Verkehrsmittel graphisch hervorgehoben und Haltestellen mit einer geeigneten Punktsignatur markiert. Eine besondere Bedeutung kommt dabei der Beschriftung zu, die möglichst funktional gehalten werden muss. Alle notwendigen Daten fließen in die Gestaltung des Liniennetzplans und beeinflussen auf ihre Weise die graphische Gestaltung. Voraussetzung ist geeignetes Datenmaterial (topographische oder vereinfachte Karte mit Routeninformationen) als Ausgangspunkt. Die Bearbeitung der Ausgangskarte soll so erfolgen, dass die Metro Map Layout Regeln weitgehend eingehalten werden. So soll beispielsweise das Streckennetz oktilinear visualisiert werden und die Abstände zwischen Haltestellen möglichst gleichmässig sein. Auch sollen wesentliche Merkmale der Topologie erhalten bleiben.

### 7.1.2 Fragestellung 2

#### **Wie kann die graphische Nachbearbeitung eines Liniennetzplans unterstützt werden und welche Algorithmen und Ansätze für die automatische Erstellung von Liniennetzplänen existieren?**

Die vorliegende Arbeit zeigt, dass bereits viele Komponenten für eine automatisierte Nachbearbeitung vorhanden sind. Es handelt sich einerseits um Algorithmen, die von einem initialen Datensatz ausgehen und das Layout der Metro Map unter Berücksichtigung der Layout-Regeln generieren können. Andere Ansätze bearbeiten Teilaufgaben, wie zum Beispiel das Labelling oder der Farbgebung der Routenlinien. Es gilt nun, diese Komponenten zusammenzufügen. Es verbleiben Fragestellungen, die aus Sicht des Autors in der Forschung noch nicht ausreichend untersucht worden sind. Hierzu zählen das MLCM-Problem und das im Kapitel 4.3.2 aufgezeigte Problem der Versetzung von Routen und der dadurch entstehenden nicht-oktilinearen Kanten (vgl. Abbildung 4.3c).

Vorstellbar wäre eine graphische Nachbearbeitung durch Hilfsmittel für konkrete Anwendungsfälle. So könnte beispielsweise eine Software sehr hilfreich sein, welche basie-



rend auf Metadaten bzw. Routeninformationen sicherstellt, dass die Streckensegmente und Haltestellen nicht zu nahe beieinander liegen. Wünschenswert wäre im Weiteren, dass Routen nicht mehr manuell gezeichnet werden müssen, sondern wiederum basierend auf den Metadaten ausgegeben werden. Hilfreich wäre, wenn der Benutzer die Darstellung wie in einem Grafikprogramm beliebig ändern kann, wobei mit Funktionen sichergestellt wird, dass beispielsweise die Abstände und Oktilinearität eingehalten werden. Darüber hinaus wäre auch die Einführung von qualifizierten Knoten zu prüfen sowie eine Funktion, welche die Darstellung der Routen (Konsistenz, Kreuzungen) bewertet und eventuell geeignete Hilfestellung zur Verfügung stellt.

### 7.1.3 Fragestellung 3

**Wie kann mit einem teilautomatisierten Verfahren der Platzbedarf für die Darstellung von Routen berücksichtigt werden?**

Die vorliegende Arbeit prüft zwei Vorgehensweisen, mit denen der Platzbedarf von Routeninformationen berücksichtigt werden kann. Durch Skalierung können Abstände vergrößert und Raum geschaffen werden. Der Vorteil der Skalierung ist die Bewahrung der Winkeltreue; der Nachteil besteht darin, dass die Peripherien unter Umständen stark überdimensioniert dargestellt werden. Eine andere Vorgehensweise ist Verdrängung, bei der Knoten auf einer Seite einer Verdrängungsgerade verschoben werden und dadurch der benötigte Raum geschaffen wird. Der Vorteil der Anwendung des Verdrängungsalgorithmus besteht darin, dass das Layout in Grundzügen erhalten und kompakt bleibt. Allerdings werden durch die Verdrängung die Winkel verändert, sodass in einem weiteren Schritt die Oktilinearität wieder hergestellt werden muss.

In einem ersten Schritt wird der Platzbedarf der Streckensegmente und Haltestellen-Signaturen berechnet und in Form eines Polygons repräsentiert. In einem weiteren Schritt werden die Konfliktzonen ermittelt und die Überschneidungen als Polygone dargestellt. Anhand dieser Konfliktpolygone wird ein Verdrängungsvektor definiert. Es wird eine Verdrängungsgerade parallel zur X- oder Y-Achse erstellt und die Knoten werden auf einer Seite der Geraden verschoben. Durch die Verdrängung können nicht-oktilineare Kanten entstehen. Bei der Wiederherstellung der Oktilinearität können weitere Konflikte auftreten, welche auf dieselbe Weise behoben werden. Dieser Prozess wird iterativ solange wiederholt, bis alle Konflikte gelöst sind.

## 7.2 Schlussfolgerung

Die Hypothese des Autors der vorliegenden Arbeit, dass es möglich sei mit Hilfe geeigneter Algorithmen die manuelle Nachbearbeitung eines Liniennetzplans zumindest teilweise zu automatisieren, hat sich bestätigt. Das in dieser Arbeit vorgestellte Verfahren kann einen Teil der manuellen Nachbearbeitung automatisieren und ein brauchbares Ergebnis liefern. Einschränkend ist festzuhalten, dass die Problemstellungen von Liniennetzplänen so unterschiedlich sind, dass immer wieder Einzelprobleme auftreten, die durch ein Standardverfahren nicht gelöst werden können. Aus Sicht des Autors der vorliegenden Arbeit macht es daher keinen Sinn die vollständig automatisierte Erstellung eines Liniennetzplans anzustreben. Auch im Beispiel des Autors der vorliegenden Arbeit ergeben sich Problemfälle, die mit seinem Ansatz nicht gelöst werden können, was aber erst bei der Implementierung deutlich wird. Für die Praxis bedeutet dies, dass immer mit Sonderfällen bei der Erstellung von Liniennetzplänen zu rechnen ist, die der manuellen Nachbearbeitung vorbehalten bleiben.

Sinnvoller ist es, wie oben beschrieben, ein Hilfsmittel zur Verfügung zu stellen, welches dem Kartenersteller möglichst alle Freiheiten eines Grafikprogramms gibt, ihn aber dennoch führt, um einen guten Liniennetzplan zu erstellen. Weiter würde dieses Hilfsmittel auch spätere Änderungen problemlos unterstützen.

## 7.3 Ausblick

Aus der vorliegenden Arbeit ergeben sich Fragestellungen, die in einer umfassenderen Erarbeitung oder mit einem anderen Blickwinkel aufgenommen werden könnten.

### **Berücksichtigung von weiteren Kartenelementen**

Ein Liniennetzplan besteht nicht nur aus Streckensegmenten und Haltestellen, sondern kann auch weitere Kartenelemente aufweisen. Diese müssten bei der Verdrängung ebenfalls einbezogen werden. Im Kapitel 5 werden bereits Ansätze skizziert, um Landmarken und Labels bei der Verdrängung zu berücksichtigen. Es bleibt zu prüfen, ob diese Ansätze realisierbar und sinnvoll sind.

### **Qualifizierte Knoten einführen**

Knoten die innerhalb einer Gruppe von Knoten die Relation zueinander bewahren sollen, können als qualifizierte Knoten definiert und in die Bearbeitung eingeführt werden. Dieses Vorgehen könnte eine zu grosse Deformation bzw. Änderung der Topologie verhindern. Dies kann relevant für die Berücksichtigung weiterer Kartenelemente oder für

die Erhaltung von Relationen einzelner Regionen sein. So könnte beispielsweise sichergestellt werden, dass zwei unabhängige Knoten auf beiden Seiten eines Sees auf gleicher Höhe liegen. Es könnte untersucht werden, welche Ergebnisse mit dem Ansatz des Autors der vorliegenden Arbeit erzielt werden können, wenn qualifizierte Knoten definiert und berücksichtigt werden.

### **Gleichzeitige Korrektur mehrerer Knoten**

Bereits Stott (2008) verwendet sogenannte Clusters in seinem Hill-Climbing-Algorithmus zur Generierung einer Metro Map (vgl. Kapitel 4.1.2). Ein Cluster ist eine Sammlung von Punkten, die als Einheit behandelt und verschoben werden. Angewendet auf den in dieser Arbeit vorgestellten Verdrängungsalgorithmus könnte dies die Entstehung überlanger Kanten verhindern, welche aufgrund eines anderen Konflikts entstehen (vgl. auch Kapitel 5.5.1).

### **Versetzung der Routen**

Nachdem in den Kapiteln 5 und 6 Ansätze diskutiert wurden, um Raum für die Routeninformationen zu schaffen, könnte auch das MLCM-Problem unter der Berücksichtigung der Oktilinearität angegangen werden. Dabei muss insbesondere der Oktilinearität bei der Versetzung von Routen Aufmerksamkeit geschenkt werden (vgl. Abbildung 4.3).

### **Behandlung von komplexen Korrekturknoten**

In der vorliegenden Arbeit wird die Behandlung von komplexen Korrekturknoten nicht berücksichtigt. Es wäre interessant herauszufinden, ob mit einer Korrektur dieser Knoten die Ausdehnung der gesamten Karte (gegenüber der im Kapitel 6 beschriebenen Implementierung) minimiert werden kann.

### **Berücksichtigung der Routeninformationen bei der Generierung**

Die vorliegende Arbeit beschreibt Ansätze, die von einem bereits generierten oder manuell erstellten oktilinearen Layout des Streckennetzes ausgehen. Auch werden die Konflikte mit dem im Kapitel 5 beschriebenen Verdrängungsalgorithmus lokal behoben. Es wäre zu prüfen, ob der Platzbedarf der Streckensegmente und Haltestellen mit einer Erweiterung eines bereits existierenden Algorithmus berücksichtigt und somit als Optimierungsproblem angegangen werden kann.

# Literatur

- Douglas, David H. und Thomas K. Peucker (1973). „Algorithms for the Reduction of the Number of Points Required to Represent a Digitized Line or Its Caricature“. In: *Cartographica: The International Journal for Geographic Information and Geovisualization* 10.2, S. 112–122.
- Schweizerische Gesellschaft für Kartographie Arbeitsgruppe (1975). „Kartographische Generalisierung“. In: *Kartographische Schriftenreihe*.
- Brassel, Kurt E. und Robert Weibel (1988). „A review and conceptual framework of automated map generalization“. In: *International journal of geographical information systems* 2.3, S. 229–244.
- Roessel, Jan W. van (1989). „An Algorithm for Locating Candidate Labeling Boxes Within a Polygon“. In: *Cartography and Geographic Information Science* 16.3, S. 201–209.
- Egenhofer, Max J. und Robert D. Franzosa (1991). „Point-set topological spatial relations“. In: *International journal of geographical information systems* 5.2, S. 161–174.
- Clementini, Eliseo, Jayant Sharma und Max J. Egenhofer (1994). „Modelling topological spatial relations: Strategies for query processing“. In: *Computers & Graphics* 18.6, S. 815–822.
- Clementini, Eliseo und Paolino Di Felice (1995). „A comparison of methods for representing topological relationships“. In: *Information Sciences - Applications* 3.3, S. 149–178.
- Jones, C. B., Geraint Ll. Bundy und J. Mark Ware (1995). „Map Generalization with a Triangulated Data Structure“. In: *Cartography and Geographic Information Science* 22.4, S. 317–331.
- Edmondson, Shawn u. a. (1996). „A General Cartographic Labelling Algorithm“. In: *Cartographica: The International Journal for Geographic Information and Geovisualization* 33.4, S. 13–24.
- Morrison, Alastair (1996). „Public Transport Maps in Western European Cities“. In: *The Cartographic Journal* 33.2, S. 93–110.
- Wagner, Frank und Alexander Wolff (1997). „A practical map labeling algorithm“. In: *Computational Geometry* 7.5-6, S. 387–404.
- Tversky, Barbara und Paul U. Lee (1999). „Pictorial and Verbal Tools for Conveying Routes“. In: *Graph Drawing*. Berlin, Heidelberg: Springer Berlin Heidelberg, S. 51–64.
- Wolff, Alexander (1999). „Automated Label Placement in Theory and Practice“. Diss.
- Avelar, Sylvania und Matthias Müller (2000). *Generating topologically correct schematic maps*. Techn. Ber.
- Wood, Clifford H (2000). „A Descriptive and Illustrated Guide for Type Placement on Small Scale Maps“. In: *The Cartographic Journal* 37.1, S. 5–18.

- Bader, Matthias (2001). „Energy Minimization Methods for Feature Displacement in Map Generalization“. Diss. Zürich.
- Cabello, Sergio, Mark de Berg u. a. (2001). „Schematization of Road Networks“. In: *SCG '01: Proceedings of the seventeenth annual symposium on Computational geometry*. New York, New York, USA: ACM Request Permissions, S. 33–39.
- Loneragan, M. und C. B. Jones (2001). „An Iterative Displacement Method for Conflict Resolution in Map Generalization“. In: *Algorithmica* 30.2, S. 287–301.
- Mackaness, William A. und R S Purves (2001). „Automated Displacement for Large Numbers of Discrete Map Objects“. In: *Algorithmica* 30.2, S. 302–311.
- Avelar, Sylvania (2002). „Schematic Maps On Demand“. Diss.
- Ware, J. Mark, Ian D. Wilson u. a. (2002). „A tabu search approach to automated map generalisation“. In: *GIS '02*. New York, New York, USA: ACM Press, S. 101–6.
- Ware, J. Mark, Christopher B. Jones und Nathan Thomas (2003). „Automated map generalization with multiple operators: a simulated annealing approach“. In: *International Journal of Geographical Information Science* 17.8, S. 743–769.
- Wilson, Ian D., J. Mark Ware und J. Andrew Ware (2003). „A Genetic Algorithm approach to cartographic map generalisation“. In: *Computers in Industry* 52.3, S. 291–304.
- Freeman, Herbert (2005). „Automated cartographic text placement“. In: *Pattern Recognition Letters* 26.3, S. 287–297.
- Nöllenburg, Martin (2005). „Automated Drawing of Metro Maps“. Magisterarb.
- Avelar, Sylvania und Lorenz Hurni (2006). „On the Design of Schematic Transport Maps“. In: *Cartographica: The International Journal for Geographic Information and Geovisualization* 41.3, S. 217–228.
- Hong, Seok-Hee, Damian Merrick und Hugo A. D. do Nascimento (2006). „Automatic visualisation of metro maps“. In: *Journal of Visual Languages and Computing* 17.3, S. 203–224.
- Jenny, Bernhard (2006). „Geometric distortion of schematic network maps“. In: *Bulletin of the Society of Cartographers* 40.1, S. 15–18.
- Benkert, Marc u. a. (2007). „Minimizing Intra-edge Crossings in Wiring Diagrams and Public Transportation Maps“. In: *Graph Drawing*. Berlin, Heidelberg: Springer Berlin Heidelberg, S. 270–281.
- Buckley, Aileen (2007). *Create route maps with the ArcGIS schematics extension*. url: <https://blogs.esri.com/esri/arcgis/2007/12/12/create-route-maps-with-the-arcgis-schematics-extension/> (besucht am 02.10.2016).
- Merrick, Damian und Joachim Gudmundsson (2007). „Path Simplification for Metro Map Layout“. In: *Graph Drawing*. Berlin, Heidelberg: Springer Berlin Heidelberg, S. 258–269.

- Monnot, Jean-Luc, Paul Hardy und Dan Lee (2007). „An optimization approach to constraint-based generalization in a commodity gis framework“. In: *International Cartographic Conference*.
- Ovenden, Mark (2007). *Transit Maps of the World*. Hrsg. von Paul E. Garbutt und Robert Schwandl. Penguin Books.
- Wolff, Alexander (2007). „Drawing subway maps: A survey“. In: *Informatik Forsch. Entw.* 22.1, S. 23–44.
- Asquith, Matthew, Joachim Gudmundsson und Damian Merrick (2008). „An ILP for the metro-line crossing problem.“ In: *CATS*.
- Avelar, Sylvania (2008). „Visualizing public transport networks: an experiment in Zurich“. In: *Journal of Maps* 4.1, S. 134–150.
- Bekos, Michael A. u. a. (2008). „Line Crossing Minimization on Metro Maps“. In: *Graph Drawing*. Berlin, Heidelberg: Springer Berlin Heidelberg, S. 231–242.
- Cain, Alasdair u. a. (2008). *Designing Printed Transit Information Materials*. Techn. Ber.
- Dwyer, Tim, Nathan Hurst und Damian Merrick (2008). „A Fast and Simple Heuristic for Metro Map Path Simplification“. In: *Advances in Visual Computing*. Berlin, Heidelberg: Springer Berlin Heidelberg, S. 22–30.
- Garland, Ken (2008). *Mr Beck's Underground Map. a history* by Ken Garland. Capital Transport Publishing.
- Kakoulis, Konstantinos G. und Ioannis G. Tollis (2008). „Labeling Algorithms“. In: *Graph Drawing*. Berlin, Heidelberg: Springer Berlin Heidelberg, S. 1–28.
- Smith, J. Cole und Z. Caner Taskın (2008). „A Tutorial Guide to Mixed-Integer Programming Models and Solution Techniques“. In: *Optimization in Medicine and Biology*. Hrsg. von G. J. Lim und E. K. Lee. Taylor und Francis, Auerbach Publications, S. 1–23.
- Stott, Jonathan (2008). „Automatic Layout of Metro Maps using Multicriteria Optimization“. Diss.
- Strobl, Christian (2008). „Dimensionally Extended Nine-Intersection Model (DE-9IM)“. In: *Encyclopedia of GIS*. Boston, MA: Springer US, S. 240–245.
- Allard, José (2009). „The Design of Public Transport Maps“. Diss.
- Ehinger, Christian (2009). „Modellierung des Layout-Problems von U-Bahn-Karten“. Diss.
- Bain, Peter (2010). „Aspects of Transit Map Design“. In: *Parsons Journal of Information Mapping* 11.3, S. 1–6.
- Nöllenburg, Martin (2010). „An Improved Algorithm for the Metro-line Crossing Minimization Problem“. In: *Graph Drawing*. Berlin, Heidelberg: Springer Berlin Heidelberg, S. 381–392.
- Stott, Jonathan u. a. (2010). „Automatic Metro Map Layout Using Multicriteria Optimization“. In: *IEEE Trans. Visual. Comput. Graphics* 17.1, S. 101–114.

- Chivers, Daniel und Peter Rodgers (2011). „Gesture-Based Input for Drawing Schematics on a Mobile Device“. In: *2011 15th International Conference Information Visualisation (IV)*. IEEE, S. 127–134.
- Nöllenburg, Martin und Alexander Wolff (2011). „Drawing and Labeling High-Quality Metro Maps by Mixed-Integer Programming.“ In: *IEEE Trans. Visual. Comput. Graphics* 17.5, S. 626–641.
- Wang, Yu-Shuen und Ming-Te Chi (2011). „Focus+Context Metro Maps“. In: *IEEE Trans. Visual. Comput. Graphics* 17.12, S. 101–114.
- Milea, Tal u. a. (2012). „Shortest-Paths Preserving Metro Maps“. In: *Graph Drawing*. Berlin, Heidelberg: Springer Berlin Heidelberg, S. 445–446.
- Sobati, Somayeh Moghadam und Absetan Ahmad (2012). „A new trigonometric Method for automatic visualization of metro map layout “. In: S. 1–5.
- Fink, Martin und Sergey Pupyrev (2013). „Metro-Line Crossing Minimization: Hardness, Approximations, and Tractable Cases“. In: *arXiv*, S. 1–20. arXiv: 1306.2079.
- Sobati, Somayeh Moghadam (2013). „New algorithm for automatic visualization of metro map“. In: *International Journal of Computer Applications* 10.4, S. 225–229.
- Wolff, Alexander (2013). „Graph Drawing and Cartography“. In: *Handbook of Graph Drawing and Visualization*, S. 697–736.
- Chivers, Daniel (2014). „Improving automated Layout Techniques for the Production of Schematic Diagrams“. Diss.
- Chivers, Daniel und Peter Rodgers (2014). „Octilinear Force-Directed Layout with Mental Map Preservation for Schematic Diagrams“. In: *Graph Drawing*. Berlin, Heidelberg: Springer Berlin Heidelberg, S. 1–8.
- Tyner, Judith A. (2014). *Principles of Map Design*. Guilford Publications.
- Archambault, Richard (2015). *Mexico City Metro System*. url: <http://mexicometro.org/wp-content/uploads/Mexico-City-Metro-Map-October-2015.pdf> (besucht am 02.10.2016).
- Cabello, Sergio und Marc van Kreveld (2015). *Automated Production of Schematic Networks*.
- Griffioen, Simone und Arthemy Kiselev (2015). „Painting new lines: Maximizing color difference in metro maps“. In: *arXiv*. arXiv: 1504.00140v1 [math.OC].
- Hauert, Jan-Henrik und Benjamin Niedermann (2015). „An Algorithmic Framework for Labeling Network Maps“. In: *arXiv Chapter 54*, S. 689–700. arXiv: 1505.00164v1 [cs.CG].
- Jacobson, Lee und Burak Kanber (2015). *Genetic Algorithms in Java Basics*. Berkeley, CA: Apress.
- Oke, Olufolajimi und Sauleh Siddiqui (2015). „Efficient automated schematic map drawing using multiobjective mixed integer programming“. In: *Computers and Operation Research* 61.C, S. 1–17.

- Sadahiro, Yukio u. a. (2015). „Computer-aided design of bus route maps“. In: *Cartography and Geographic Information Science* 43.4, S. 361–376.
- Wang, Yu-Shuen und Wan-Yu Peng (2015). „Interactive Metro Map Editing“. In: *IEEE Trans. Visual. Comput. Graphics* 22.2, S. 1115–1126.



# Tabellenverzeichnis

B.1 Haltestellen im Beispiel 1 . . . . .	.109
B.2 Streckensegmente im Beispiel 1 . . . . .	.109
B.3 Haltestellen im Beispiel 2 . . . . .	.110
B.4 Streckensegmente im Beispiel 2 . . . . .	.111
B.5 Haltestellen im Beispiel 3 (Teil 1) . . . . .	.112
B.6 Haltestellen im Beispiel 3 (Teil 2) . . . . .	.113
B.7 Streckensegmente im Beispiel 3 (Teil 1) . . . . .	.113
B.8 Streckensegmente im Beispiel 3 (Teil 2) . . . . .	.114

# Abbildungsverzeichnis

2.1	Beispiele für den French, Classic, Scandinavian und Dutch Style . . . . .	6
2.2	Ausschnitt aus einem Postauto-Liniennetzplan . . . . .	7
2.3	Ausschnitt aus dem Liniennetzplan von Montréal . . . . .	8
2.4	Ausschnitt aus dem Liniennetzplan von Amsterdam . . . . .	8
2.5	Ausschnitt aus dem Liniennetzplan von North Vancouver . . . . .	9
2.6	Ausschnitt aus dem Liniennetzplan von London . . . . .	10
2.7	Ausschnitt aus dem Liniennetzplan der Stadt Zürich . . . . .	11
2.8	Ausschnitt aus dem Liniennetzplan des Kantons Zürich . . . . .	12
2.9	Beispiele für das Design von Haltestellen . . . . .	16
3.1	Geometrische Verzerrung der Tube Map von London . . . . .	22
3.2	Ausgangslage Liniennetzplan . . . . .	24
3.3	Schritte zum oktilinearen Streckennetz mit Adobe Illustrator CC . . . . .	25
3.4	Einfacher Liniennetzplan ohne Routen . . . . .	26
3.5	Manuell erstellte Karte im Bereich des Zürcher Hauptbahnhofes . . . . .	27
4.1	Lokales und globales Maximum im Hill Climbing Algorithmus . . . . .	30
4.2	Probleme bei der Darstellung der Linienführung . . . . .	35
4.3	Korrekturproblem Versetzung . . . . .	38
4.4	Beispiel von Streckensegmenten mit sehr vielen Routen . . . . .	42
5.1	Problemstellung der Überlappung von Routeninformationen . . . . .	45
5.2	Konfliktpolygone . . . . .	45
5.3	Zeichenerklärung für das Kapitel 5.1.1 . . . . .	46
5.4	Anwendung einer Verdrängung in einer Talebene . . . . .	49
5.5	Verdrängung im oktilinearen Layout . . . . .	51
5.6	Beispiele für orthogonale Verdrängungsvektoren . . . . .	53
5.7	Beispiele für nicht-orthogonale Verdrängungsvektoren . . . . .	54
5.8	Verdrängungsvektoren mit Konfliktpolygon ermittelt . . . . .	54
5.9	Abbildung 5.2 nach der Skalierung . . . . .	55
5.10	Problemregionen bei der Skalierung von Teilgraphen . . . . .	57
5.11	Berücksichtigung der Landmarken bei der Skalierung (1) . . . . .	58
5.12	Berücksichtigung der Landmarken bei der Skalierung (2) . . . . .	59

5.13	Berücksichtigung der Labels bei der Skalierung . . . . .	59
5.14	Verdrängungsvektoren mit der kleinsten Verfälschung ermitteln . . . . .	61
5.15	Verzicht auf die Verdrängung des Knotens . . . . .	62
5.16	Mögliche Verschieberichtungen zur Wiederherstellung der Oktilinearität .	63
5.17	Situationen mit nur einer Ausrichtung der adjazenten Kanten . . . . .	63
5.18	Situation mit Kombinationen von Ausrichtungen der adjazenten Kanten . .	63
5.19	Behandlung von kleinen Kanten nach der Verdrängung . . . . .	66
5.20	Wiederherstellung der Oktilinearität durch Knicke . . . . .	67
6.1	Spezieller Fall eines Konflikts zwischen Kante und Knoten . . . . .	72
6.2	Ermittlung der Verdrängungsgeraden bei einem Kante/Knoten-Konflikt . .	74
6.3	Beispiel für den Verlust der ursprünglichen Ausrichtung . . . . .	77
6.4	Ablaufübersicht des Verdrängungsalgorithmus . . . . .	78
6.5	Bearbeitungsreihenfolge der Konflikte der Kanten und Knoten . . . . .	80
6.6	Spezielle Konflikte zwischen einer Kante und einem Knoten . . . . .	80
6.7	Ausgangslage Beispiele . . . . .	81
6.8	Anwendung der vorgestellten Ansätze am Beispiel aus Kapitel 5 . . . . .	81
6.9	Anwendung der vorgestellten Ansätze auf einen komplexeren Graph . . .	82
6.10	Anwendung der vorgestellten Ansätze auf ein Streckennetz . . . . .	82
6.11	Anwendung der vorgestellten Ansätze auf ein Streckennetz (vergrössert) .	84
B.1	Ausgangslage für den Algorithmus (S-Bahn-Streckennetz Stadt Zürich) . .	115
C.1	Klassen des Core-Package . . . . .	117
C.2	Klassen des Util-Package . . . . .	118

# Glossar

**Adjazente Kante/Knoten** Graphentheorie; Eine Kante ist adjazent zu einem Knoten, wenn es sich bei dem Knoten um einen Endknoten der Kante handelt und umgekehrt. Eine Kante ist adjazent zu einer anderen Kante, falls beide einen gemeinsamen adjazenten Knoten besitzen. Beide Endknoten einer Kante sind ebenfalls adjazent zueinander.

**ArcGIS** Ein geographisches Informationssystem der Firma ESRI.

**ArcGIS Schematics** Eine Erweiterung von ArcGIS für die schematische Darstellung von räumlichen Daten.

**Bounding Box** Eine Bounding Box steht parallel zur X-Achse und ist das kleinste Rechteck, welches ein oder mehrere geometrische Elemente einschließen kann.

**Breadth-First Search** Ein Algorithmus zur Traversierung eines Graphen; dabei werden die benachbarten Knoten zuerst untersucht, bevor die nächst tiefere Ebene überprüft wird.

**Brücke** Graphentheorie; eine Kante bei deren Entfernung zwei Komponenten entstehen.

**Buffer** Ein Polygon, dessen Aussenlinie eine spezifizierte Distanz zu einem anderen geometrischen Element hat.

**Bézierkurve** Eine Kurve die mit Parametern beschrieben und häufig in vektorbasierten Computergrafiken verwendet wird.

**Centroid** Engl. für den Schwerpunkt einer geometrischen Form.

**Clipping** Das Resultat einer Clip-Operation mit einem Polygon (als Clipper) ist die gemeinsame Fläche beider Polygone oder bei einer Linie die Linienabschnitte innerhalb des Clippers.

**Clustering** Engl.; steht für eine Häufung; ein Cluster beschreibt alle Elemente innerhalb einer Häufung.

**Concept Draw PRO** Ein Visualisierungsprogramm mit einer Symbolbibliothek zur Erstellung von Metro Maps.

**DE-9IM** Abkürzung für Dimensionally Extended Nine Intersection Model; ein topologisches Model und Standard zur Beschreibung von räumlichen Beziehungen.

- Delaunay-Triangulierung** Die Delaunay-Triangulierung ist eine Triangulierung mit speziellen Eigenschaften. Bei einer Triangulierung werden Dreiecke aus einer Menge von Punkten erstellt, wobei die Eckpunkte der Dreiecke jeweils aus den Punkten bestehen.
- Douglas-Peucker Algorithmus** Ein Algorithmus zur Kurvenglättung. Das Ziel ist es eine Linie durch Weglassen von Knoten zu vereinfachen, sodass jedoch die ursprüngliche Form noch erkennbar ist.
- Edraw Max** Ein Visualisierungsprogramm mit einer Symbolbibliothek zur Erstellung von Metro Maps.
- Enklave** Ein Gebiet welches vollständig von einem anderen Gebiet eingeschlossen ist.
- Evolutionärer Algorithmus** Ein heuristisches Optimierungsverfahren basierend auf der Funktionsweise der Evolution in der Natur.
- Force-Directed Algorithmus** Ein Algorithmus zum Zeichnen von Graphen basierend auf anziehenden und abstossenden Kräften.
- Generalisierung** Durch die Anwendung eines Generalisierungsvorgangs (wie z.B. Typifizierung, Verdrängung oder Übertreibung) wird der Karteninhalt vereinfacht. Das Ziel der Generalisierung ist die Verbesserung der Lesbarkeit und Verständlichkeit.
- GIS** Abkürzung für ein geographisches Informationssystem.
- Graph** Graphentheorie; eine Datenstruktur aus Objekten (Knoten) und allfälligen Verbindungen (Kanten) zwischen diesen Objekten.
- Graphische Primitive** Eine elementare geometrische Form. Solche Grundformen sind unter anderem Linien, Punkte und Polygone.
- Heuristik** Verfahren zur Findung einer möglichst guten Lösung mit begrenzten bzw. unvollständigen Informationen.
- Hill Climbing Algorithmus** Ein heuristisches Optimierungsverfahren. Dabei wird das Layout iterativ verändert, solange ein besseres Resultat erzielt wird. Das Resultat jeder Veränderung wird jeweils mit Kriterien bewertet.
- Kante** Graphentheorie; verbindet zwei Objekte eines Graphen miteinander.
- Knoten** Graphentheorie; ein Objekt im Graph. In Liniennetzplan kann ein Knoten jeweils eine Haltestelle, Kreuzung, Weiche oder einen Knick beschreiben.

- Kognitive Karte** Eine mentale Repräsentation geographischer Gegebenheiten.
- Komponente** Graphentheorie; eine Komponente ist ein Teilgraph der nicht durch eine Kante mit einem anderen verbunden ist.
- Konfliktbuffer** Der Buffer einer Signatur zur Feststellung eines Konflikts.
- Konfliktpolygon** Ein Konfliktpolygon entsteht durch eine Überschneidung zweier Konfliktbuffer.
- Konfliktzone** Die Fläche des Konfliktpolygons.
- Konkaves Polygon** Ein konkaves Polygon besitzt mindestens einen Innenwinkel der größer als 180 Grad ist.
- Konvexe Hülle** Eine konvexe Hülle entspricht dem kleinsten konvexen Polygon, welches alle Punkte der geometrischen Elemente beinhaltet.
- Konvexes Polygon** Ein konvexes Polygon besitzt nur Innenwinkel, die kleiner als 180 Grad sind.
- Korrektur-Kreis** Vgl. Kreis. In der vorliegenden Arbeit wird die Bezeichnung Korrektur-Kreis nur verwendet, wenn der Korrekturprozess der Oktilinearität auf einen Knoten trifft welcher bereits traversiert/korrigiert wurde.
- Kreis** Graphentheorie; Ein geschlossener Weg in einem Graphen.
- Label** Engl. für eine Beschriftung, beispielsweise einer Haltestelle oder einer Landmarke.
- Least-Square Regression** Ein Verfahren zur Findung einer Kurve, die möglichst nahe an den Knoten verläuft.
- Linie** Eine Linie im geometrischen Sinn. In der vorliegenden Arbeit wird der alleinstehende Begriff "Linie" nicht für eine Route verwendet, ausser er wird mit einem anderen Wort, z.B. Buslinie, verwendet.
- Mentale Karte** Vgl. kognitive Karte.
- Metro Map** Ein Liniennetzplan; der Begriff Metro Map wird jedoch meistens für reine U-Bahn-Netzwerke verwendet.
- MIP** Abkürzung für Mixed Integer Programming; engl. für ganzzahlige lineare Optimierung; Optimierungsverfahren basierend auf Gleichungen.
- MLCM** Abkürzung für Metro Line Crossing Problem; beschreibt die Problematik der Überkreuzung von Routenlinien in einem Liniennetzplan.

**Mobility** Ein Carsharing Service in der Schweiz.

**Observer-Pattern** Ein Entwurfsmuster aus der Software Entwicklung. Das Observer-Objekt ist als Beobachter bei einem anderen Objekt angemeldet und wird benachrichtigt, falls Änderungen vorliegen.

**Oktilinearität** Darstellung eines Graphen, dessen Kanten in einem Winkel von 45 Grad oder einem Mehrfachen zur X-Achse dargestellt sind.

**Optimierungsproblem** Bei einem Optimierungsproblem wird eine optimale Lösung zu einer Problemstellung gesucht, da die Ermittlung der besten Lösung nicht (in nützlicher Zeit) durchgeführt werden kann.

**Optimierungsverfahren** Ein Verfahren zur Findung einer optimalen Lösung, beispielsweise durch eine graphische oder iterative Optimierung.

**Orthogonalität** Darstellung eines Graphen, dessen Kanten in einem Winkel von 90 Grad oder einem Mehrfachen zur X-Achse dargestellt sind.

**Ortsvektor** Ein Vektor mit einem festen Bezugspunkt.

**Overlay Map** Eine Karte mit überlagerten, zusätzlichen Informationen.

**Panel** Eine Anzeigetafel für Informationen, z.B. bei einer Haltestelle.

**Paretoprinzip** Die 80/20 Regel sagt aus, dass 80% der Ergebnisse mit 20% Aufwand erzielt werden können.

**Pattern** Engl. für ein Muster. Ein Pattern kann von dem Betrachter wiedererkannt werden und ist daher einfacher zu merken.

**Pfad** Graphentheorie; eine Menge aus aufeinanderfolgenden (adjazenten) Kanten innerhalb eines Graphen.

**Point of Interest** Engl. für einen Ort von besonderem Interesse, beispielsweise ein Flughafen oder eine Sehenswürdigkeit.

**Projektion** Vgl. Vektorprojektion.

**Pseudo Code** Pseudo Code ist ein Quelltext, der keiner vom Computer interpretierbaren Programmiersprache entspricht. Er dienen zur Veranschaulichung von Algorithmen.

**Rejection** Vgl. Vektorprojektion.

**Rekursion** Eine Rekursion bezeichnet in der Software-Entwicklung die Eigenschaft einer Funktion, welche sich selbst wieder aufruft.

**Route** Eine Route bezeichnet den Streckenverlauf eines Transportmittels. In der vorliegende Arbeit wird der Streckenverlauf von Transportnetzwerke immer als Routen bezeichnet. Der alleinstehende Term "Linie" wird für eine Linie im geometrischen Sinn verwendet.

**Routeninformation** Alle Routen und dessen Attribute auf einem Streckensegment.

**SchemaSketch** Eine experimentelle Applikation für Tablets zur Erstellung von Metro Maps.

**Set** Engl. für eine Menge.

**Signatur** Die Darstellung eines Kartenelementes.

**Simplification** Eine Generalisierungsmethode bei der Details einer Darstellung entfernt werden. Dies kann beispielsweise die Vereinfachung einer Form oder das Zusammenfassen von mehreren Kartenelementen zu einem Element sein.

**Simulated Annealing** Ein heuritisches Optimierungsverfahren.

**Smoothing** Ein Glättungsverfahren von Linien und anderen Formen.

**Snapping** Ein Vorgang, der die Position eines Punktes, welcher in einer bestimmten Distanz zu einem anderen Punkt liegt, diesem Punkt angleicht.

**Spring Embedder** Vgl. Force-Directed Algorithmus.

**Streckensegment** Eine Strecke zwischen zwei Haltestellen. Gleichbedeutend mit einer Kante im darunterliegenden Graph.

**SVG** Abkürzung für Scalable Vector Graphics; ein XML-basiertes Datenformat für zweidimensionale Vektorgrafiken.

**Tabu-Search-Approach** Ein heuritisches Optimierungsverfahren. Der Namen kommt von der angelegten Tabu-Liste, deren Inhalt bei weiteren Iterationen nicht für mögliche Lösungen verwendet werden darf.

**Tarifzone** Ein Verkehrsverbund teilt sein Verkehrsnetz zur Festlegung der Fahrpreise in Tarifzonen ein.

**Teilgraph** Graphentheorie; ein Teilgraph enthält eine Untermenge der Knoten und Kanten eines Graphen.

**Typifizieren** Ein Generalisierungsvorgang in der Kartographie, bei dem Kartenelemente eines gleichen Typs in reduzierter Menge dargestellt werden. Dabei bleibt die räumliche Verteilung erhalten.



**Vektorprojektion** Eine Vektorprojektion ist die orthogonale Projektion eines Vektors entlang einer geraden Linie. Die Rejektion steht senkrecht zur Projektion und wenn sie zur Projektion dazu addiert wird, erhält man den ursprünglichen Vektor.

**Verbindungskante** Eine Kante, dessen adjazenten Knoten je auf einer Seite der Verdrängungsgerade liegen.

**Verbundgebiet** Das Gebiet eines Verkehrsverbunds.

**Verdrängung** Ein Vorgang in der Kartographie, der für die Generalisierung von Karten verwendet wird; Kartenelemente werden aufgrund des Platzbedarfs durch Signaturen anderer Elemente aus ihrer ursprünglichen Position verdrängt, sodass alle Signaturen dargestellt werden können.

**Verdrängungsgerade** Eine zur X-Achse orthogonale Gerade, welche die Knoten eines Graphen aufteilt. Die Knoten östlich oder nördlich (je nach Winkel) werden jeweils während des Verdrängungsprozesses verschoben.

**Verdrängungsvektor** Der Vektor, der die Richtung und Distanz der Verdrängung beschreibt.

**Vertex** Engl. für einen Knoten bzw. Punkt auf einer Linie.

**Verzerrung** Eine Verzerrung der Karte entsteht beim Strecken, Ändern von Winkeln oder dem Verändern von Flächen. Bei der verzerrten Karte handelt es sich somit um keine affine Abbildung der realen Situation.

**Winkeltreue** Die Darstellung von Kartenelementen oder geometrischen Formen ohne Winkelverzerrung; die Winkel werden unverändert dargestellt.

# Akronyme

**DE-9IM** Dimensionally Extended Nine Intersection Model

**GIS** Geographisches Informationssystem

**ILP** Inductive Logic Programming

**MIP** Mixed-Integer Linear Programming

**MLCM** Metro Line Crossing Problem

**POI** Point of Interest

**RATP** Paris Transport Authority

**SVG** Scalable Vector Graphics

**TNW** Tarifverbund Nordwestschweiz

**VBZ** Verkehrsbetriebe Zürich

**ZVV** Zürcher Verkehrsverbund

# Notationen

$e$	Streckensegment, Kante
$m_e$	Mindestabstand der Kante $e$ zu einem anderen Element
$l$	Route, Pfad, Weg
$l_e$	Route $l$ entlang des Streckensegments $e$
$L_e$	Set aller Routen entlang des Streckensegments $e$
$ L_e $	Anzahl Routen auf dem Streckensegment $e$
$m_l$	Margin (Abstand) zwischen Routenlinien
$w_l$	Linienstärke der Route $l$
$w_e$	Benötigte Darstellungsbreite der Kante $e$
$r_e$	Halbe Darstellungsbreite der Kante $e$ , $r_e = \frac{w_e}{2}$
$b_e$	Bufferpolygon der Kante $e$
$n$	Haltestelle, Station, Knoten
$h_n$	Konvexe Hülle der Haltestellen-Signatur
$m_n$	Mindestabstand der Haltestelle $n$ zu einem anderen Element
$b_n$	Bufferpolygon des Knotens $n$
$B$	Set aller Bufferpolygone (der Knoten <i>und</i> Kanten) im Liniennetzplan
$g$	Verdrängungsgerade
$\lambda$	Skalierungsfaktor
$c$	Konfliktpolygon
$C$	Set aller Konfliktpolygone
$b_c$	Bounding Box des Konfliktpolygons
$e_d$	Verbindungskante
$e_k$	Konfliktkante (Konfliktelement)
$n_k$	Konfliktknoten (Konfliktelement)
$q$	Gerade durch den Centroid zweier Konfliktelemente
$q'$	Parallele Linie zu $q$ , die an einem Eckpunkt des Konfliktpolygons beginnt und auf einer gegenüberliegenden Seite endet
$\vec{v}$	Verdrängungsvektor
$\vec{k}$	Korrekturvektor
$\vec{x}$	Vektor mit einer Distanz von 1 entlang der X-Achse
$\vec{y}$	Vektor mit einer Distanz von 1 entlang der Y-Achse
$\vec{v}_x$	Projektion des Vektors $\vec{v}$ entlang der X-Achse
$\vec{v}_y$	Projektion des Vektors $\vec{v}$ entlang der Y-Achse

- $\kappa$  Anpassungskosten
- $\iota$  Maximale Anpassungskosten
- $x(n)$  X-Wert der Position des Knotens  $n$
- $y(n)$  Y-Wert der Position des Knotens  $n$
- $dx$  Breite einer Bounding Box oder Delta der X-Werte von zwei Punkten
- $dy$  Höhe einer Bounding Box oder Delta der Y-Werte von zwei Punkten

# Anhang

# A | Metro Map Layout Regeln

Der nachfolgende Regelkatalog (R1 bis R7) ist vollständig von Wolff 2013 (Seite 715) übernommen und wiedergegeben:

- (R1) Restrict the drawing of edges to the octilinear directions.
- (R2) Do not change the geographical network topology. This is crucial to support the mental map of the passengers.
- (R3) Avoid bends along individual metro lines, especially in interchange stations, to keep them easy to follow for map readers. If bends cannot be avoided, obtuse angles are preferred over acute angles.
- (R4) Preserve the relative position between stations to avoid confusion with the mental map. For example, a station being north of some other station in reality should not be placed south of it in the metro map.
- (R5) Keep edge lengths between adjacent stations as uniform as possible with a strict minimum length. This usually implies enlarging the city center at the expense of the periphery.
- (R6) Stations must be labeled and station names should not obscure other labels or parts of the network. Horizontal labels are preferred and labels along the track between two interchanges should use the same side of the corresponding path if possible.
- (R7) Use distinctive colors to denote the different metro lines. This means that edges used by multiple lines are drawn thicker and use colored copies for each line.

## B | Testdaten

Für die vorliegende Arbeit wird neben dem sehr einfachen Graphen aus Abbildung 5.1a zwei weitere Graphen verwendet.

### B.1 Beispiel 1

Alle benötigten Angaben zu den Testdaten können der Klasse `MetroMapChapterFive` im Anhang D.5.1 oder aus den folgenden Tabellen entnommen werden. Für das Beispiel werden folgende Margins verwendet:  $m_l = 1$ ,  $m_e = 25$  und  $m_n = 25$ . Für die Linienstärke einer Route gilt  $w_l = 20$ . Weiter repräsentiert die Klasse `SquareStationSignature` die Haltestelle-Signatur (vgl. Anhang D.4.2).

#### B.1.1 Haltestellen

Name	X	Y
A	0	30
B	80	30
C	110	0
D	140	30
E	110	60
F	50	60

**Tabelle B.1:** Haltestellen im Beispiel 1

#### B.1.2 Streckensegmente

von	nach	Anzahl Routen
A	B	4
B	C	2
C	D	1
D	E	1
E	F	3
B	E	2

**Tabelle B.2:** Streckensegmente im Beispiel 1

## B.2 Beispiel 2

Alle benötigten Angaben zu den Testdaten können der Klasse `MetroMapExampleGraph` im Anhang D.5.1 oder aus den folgenden Tabellen entnommen werden. Für das Beispiel werden folgende Margins verwendet:  $m_l = 2$ ,  $m_e = 25$  und  $m_n = 25$ . Für die Linienstärke einer Route gilt  $w_l = 20$ . Weiter repräsentiert die Klasse `RectangleStationSignature` die Haltestelle-Signatur (vgl. Anhang D.4.2).

### B.2.1 Haltestellen

Name	X	Y
A	150	200
B	150	100
C	200	100
D	200	150
E	200	250
F	150	300
G	100	300
H	100	200
I	100	150
J	150	250
K	170	250
N	100	250
O	170	150
S	150	50

**Tabelle B.3:** Haltestellen im Beispiel 2



**B.2.2 Streckensegmente**

<b>von</b>	<b>nach</b>	<b>Anzahl Routen</b>
A	B	5
C	B	4
C	D	4
D	E	4
E	F	5
F	G	2
G	N	3
N	H	2
H	A	5
H	I	3
I	B	7
I	A	3
A	J	1
J	K	2
E	K	1
O	K	3
O	D	5
B	S	1
C	S	1

**Tabelle B.4:** Streckensegmente im Beispiel 2

### B.3 Beispiel 3 (S-Bahnstreckennetz der Stadt Zürich)

Für das S-Bahn-Streckennetz der Stadt Zürich wird die Abbildung B.1 für die Position der Knoten verwendet<sup>1</sup>. Die Routeninformationen wird aus dem offiziellen Liniennetzplan entnommen (vgl. Abbildung 2.8).

Alle benötigten Angaben zu den Testdaten können der Klasse `MetroMapZuerich` im Anhang D.5.1 oder aus den folgenden Tabellen entnommen werden. Für das Beispiel werden folgende Margins verwendet:  $m_l = 2$ ,  $m_e = 25$  und  $m_n = 25$ . Für die Linienstärke einer Route gilt  $w_l = 5$ . Weiter repräsentiert die Klasse `RectangleStationSignature` die Haltestelle-Signatur (vgl. Anhang D.4.2).

#### B.3.1 Haltestellen

Name	X	Y
Zürich HB	120	105
C1 (Kreuzung Sihlhölzli)	115	90
Selnau	110	85
J1 (Giesshübel/Binz)	105	80
Binz	95	80
Friesenberg	90	80
Giesshübel	110	75
Saalsporthalle	110	70
B1	120	95
J2 (Wiedikon/Wipkingen/Zürich HB)	110	105
Wiedikon	110	95
Enge	120	85
Wollishofen	145	60
B2	105	100
Hardbrücke	100	105
Altstetten	75	105
J3 (Oerlikon/Hardbrücke/Altstetten)	95	105
J4 (Oerlikon/Hardbrücke/Wipkingen)	105	135
J5 (Oerlikon/Hardbrücke/Wipkingen/Zürich HB)	105	140
Oerlikon	105	145
Wipkingen	105	110

**Tabelle B.5:** Haltestellen im Beispiel 3 (Teil 1)

<sup>1</sup> Für die Testausführung wird jedoch nur die Haltestellen innerhalb der Stadt Zürich verwendet.

### B.3. Beispiel 3 (S-Bahnstreckennetz der Stadt Zürich)

<b>Name</b>	<b>X</b>	<b>Y</b>
B3	95	125
J6 (Stadelhofen/Zürich HB)	130	105
Stadelhofen	135	100
J7 (Tiefenbrunnen/Stettbach)	140	95
Tiefenbrunnen	160	75
B4	130	115
J8 (Oerlikon/Wallisellen)	105	150
Wallisellen	140	165
J9 (Stettbach/Wallisellen/Dietlikon/Dübendorf)	160	165
B5	110	155
B6	130	155
Stettbach	160	150
B7	160	115
Dietlikon	170	190
B8	160	180
Dübendorf	205	160
B9	200	165
Schlieren	40	105
Urdorf	40	75
J10 (Schlieren/Urdorf)	70	105

**Tabelle B.6:** Haltestellen im Beispiel 3 (Teil 2)

#### B.3.2 Streckensegmente

<b>von</b>	<b>nach</b>	<b>Anzahl Routen</b>
Zürich HB	B1	2
C1	B1	2
C1	Selnau	2
J1	Selnau	2
J1	Giesshübel	1
Saalsporthalle	Giesshübel	1
J1	Binz	1
Friesenberg	Binz	1
Zürich HB	J2	16
B2	J2	5

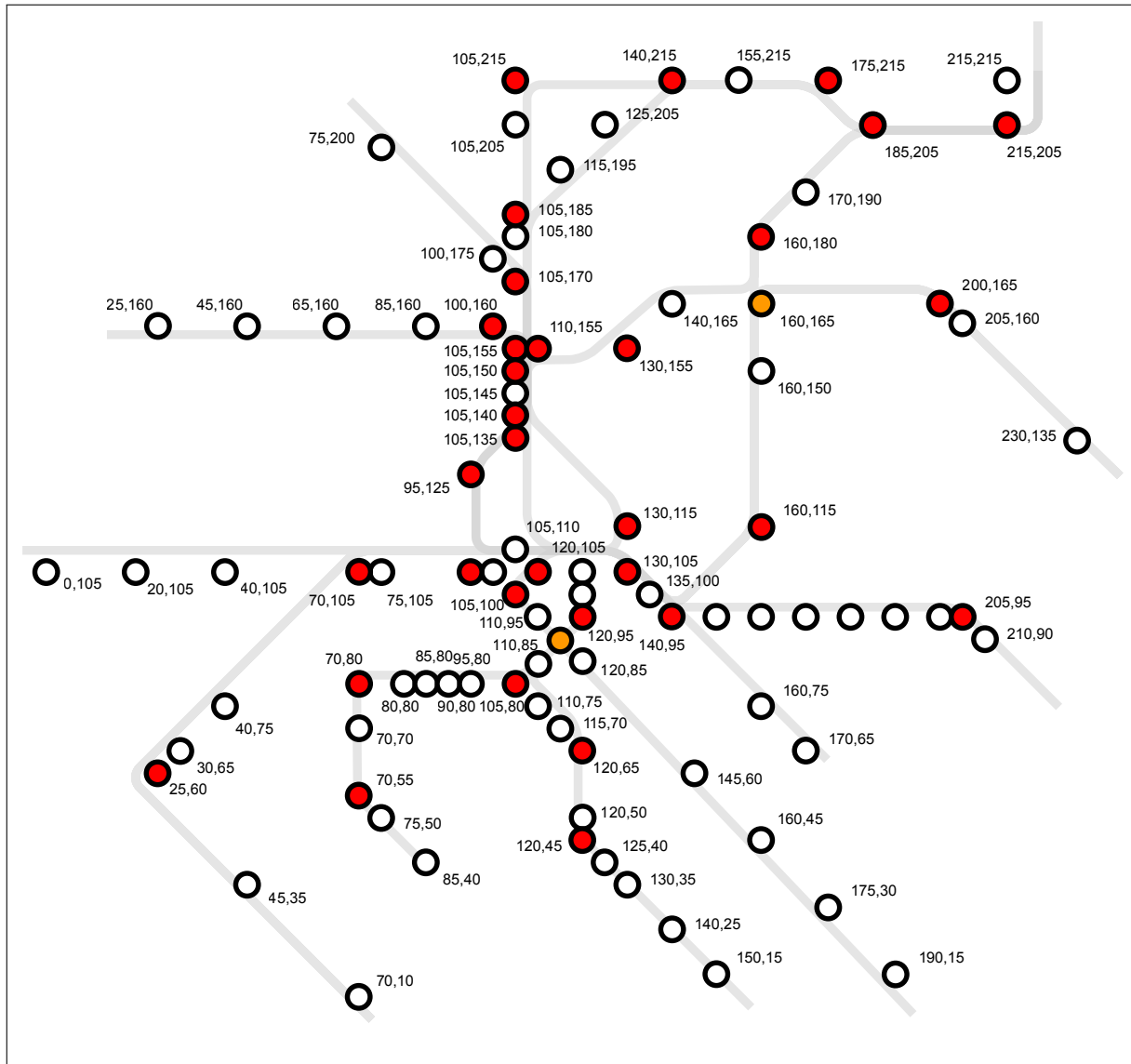
**Tabelle B.7:** Streckensegmente im Beispiel 3 (Teil 1)

B.3. Beispiel 3 (S-Bahnstreckennetz der Stadt Zürich)

<b>von</b>	<b>nach</b>	<b>Anzahl Routen</b>
B2	Wiedikon	5
C1	Wiedikon	5
C1	Enge	5
Wollishofen	Enge	5
J2	Hardbrücke	12
J3	Hardbrücke	12
J3	B3	6
J4	B3	6
J4	J5	8
Oerlikon	J5	12
J3	Altstetten	7
J2	Wipkingen	2
Zürich HB	J6	14
B4	J6	4
B4	J5	4
J6	Stadelhofen	10
J7	Stadelhofen	10
J7	Tiefenbrunnen	4
Oerlikon	J8	12
B5	J8	3
B5	B6	3
Wallisellen	B6	3
Wallisellen	J9	3
Stettbach	J9	6
Stettbach	B7	6
J7	B7	6
J9	B8	5
Dietlikon	B8	5
J9	B9	4
Dübendorf	B9	4
Altstetten	J10	7
Urdorf	J10	2
Schlieren	J10	5

**Tabelle B.8:** Streckensegmente im Beispiel 3 (Teil 2)

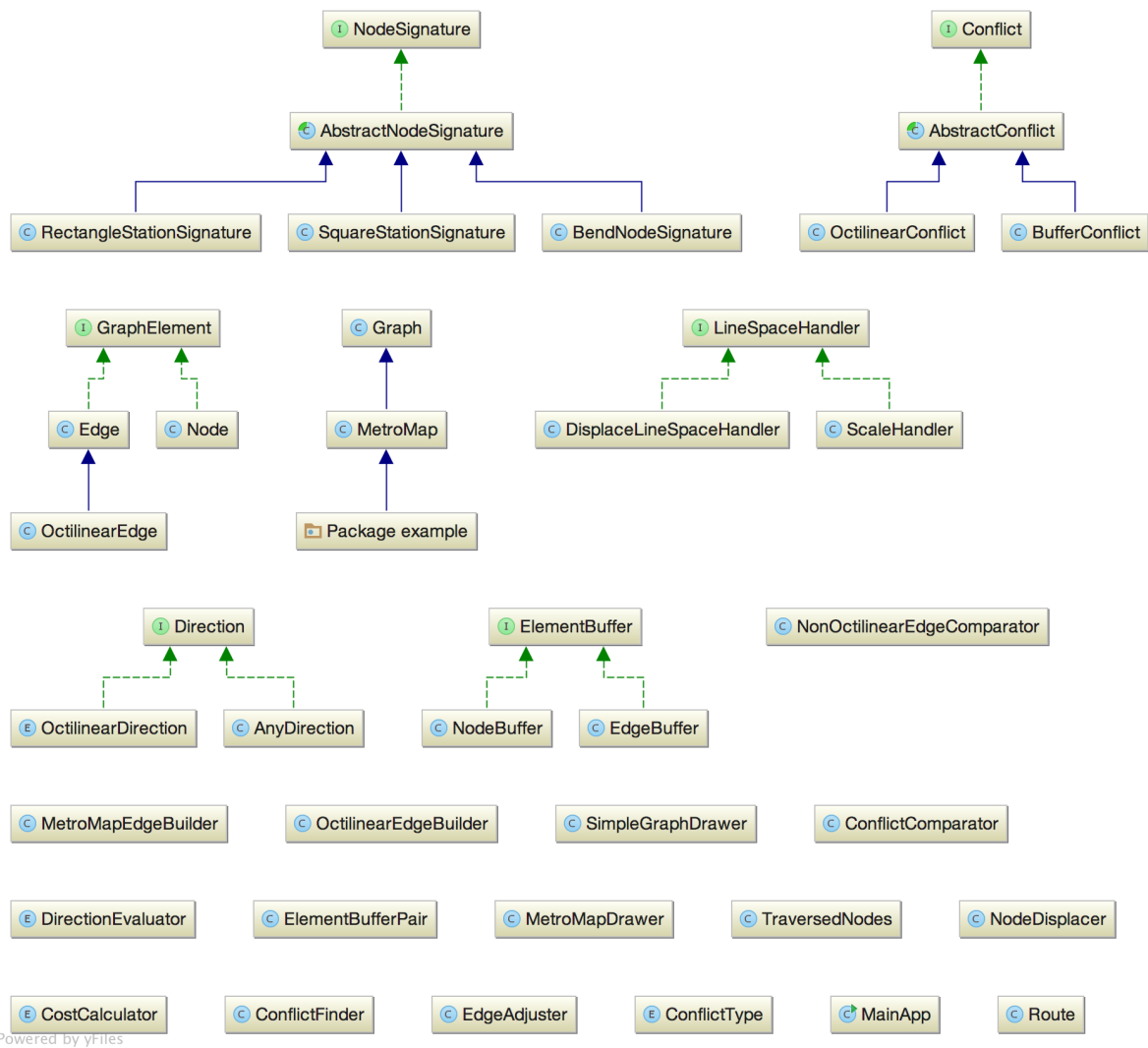
### B.3. Beispiel 3 (S-Bahnstreckennetz der Stadt Zürich)



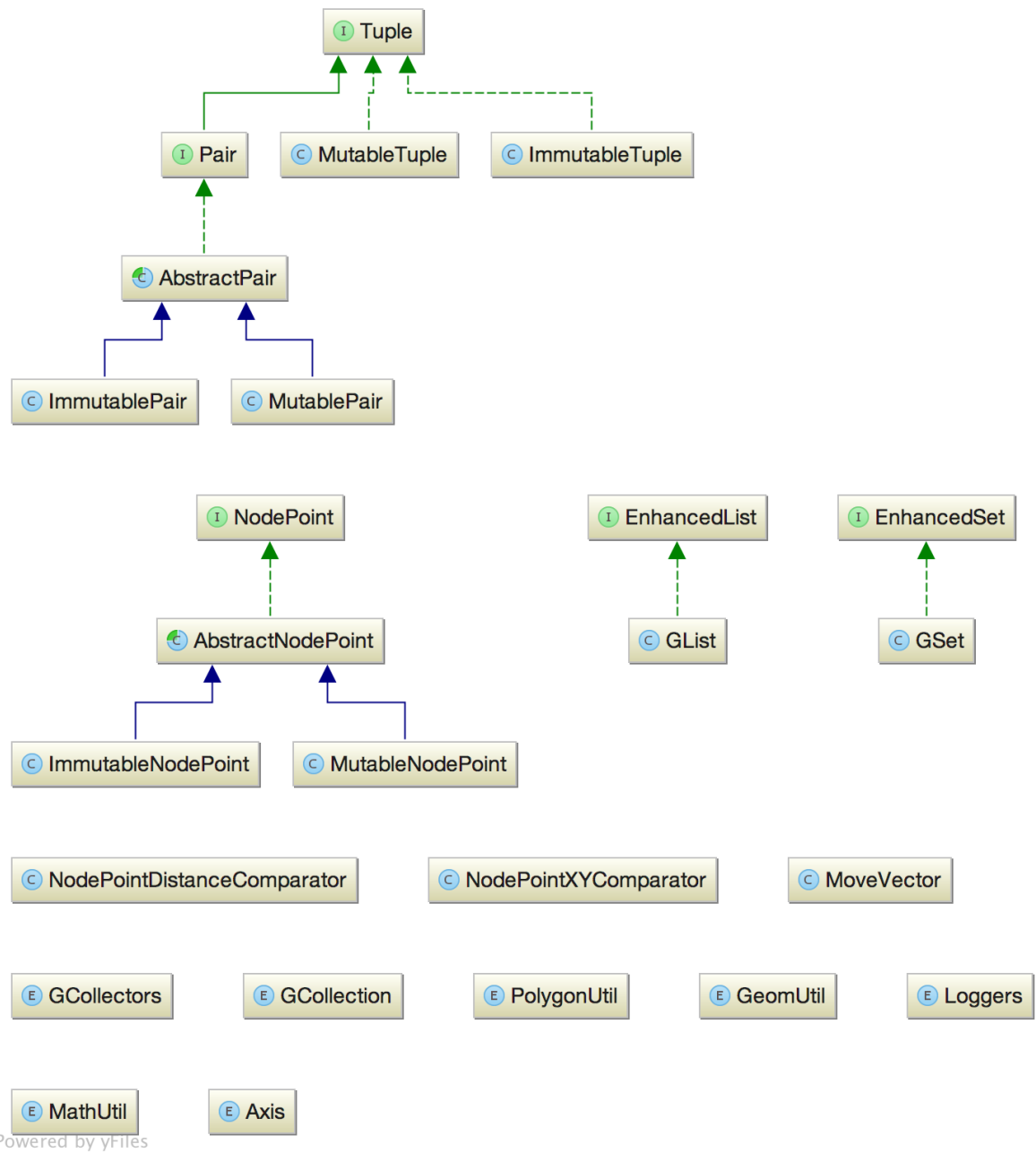
**Abbildung B.1:** Ausgangslage des S-Bahn-Streckennetzes der Stadt Zürich mit eingezeichneten Positionen der Haltestellen (weiss), Knicke (rot) und Kreuzungen (orange). Die Routeninformationen können aus dem offiziellen Liniennetzplan (vgl. Abbildung 2.8) entnommen werden. (eigene Grafik)

## C | Klassenstruktur

Die Abbildungen C.1 und C.2 zeigen die Klassen des Core- and Util-Packages. Alle Klassen sind während und für die Masterarbeit entstanden. Insgesamt handelt es sich um 4662 Zeilen Code (LOC). Dazu kommen weitere 1363 Zeilen Kommentare.



**Abbildung C.1:** Klassenhierarchie des Core-Package. (eigene Grafik, generiert mit IntelliJ IDEA 2016.2)



**Abbildung C.2:** Klassenhierarchie des Util-Package. (eigene Grafik, generiert mit IntelliJ IDEA 2016.2)



# D | Source Code

Dieses Kapitel enthält alle Core- und Util-Klassen die im Rahmen dieser Masterarbeit erstellt wurden und im Package `ch.geomo.tramaps.*` verwendet werden.

## D.1 Evaluation der Konflikte

### D.1.1 Package `ch.geomo.tramaps.conflict.*`

#### AbstractConflict.java

```
1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.tramaps.conflict;
6
7  import ch.geomo.tramaps.conflict.buffer.ElementBuffer;
8  import ch.geomo.tramaps.conflict.buffer.ElementBufferPair;
9  import ch.geomo.tramaps.graph.Edge;
10 import ch.geomo.tramaps.graph.GraphElement;
11 import ch.geomo.tramaps.graph.Node;
12 import ch.geomo.tramaps.graph.direction.OctilinearDirection;
13 import ch.geomo.util.collection.pair.Pair;
14 import ch.geomo.util.geom.Axis;
15 import ch.geomo.util.geom.GeomUtil;
16 import ch.geomo.util.math.MoveVector;
17 import com.vividsolutions.jts.geom.Coordinate;
18 import com.vividsolutions.jts.geom.Envelope;
19 import com.vividsolutions.jts.geom.Geometry;
20 import com.vividsolutions.jts.geom.GeometryCollection;
21 import org.jetbrains.annotations.NotNull;
22
23 import java.util.Arrays;
24 import java.util.List;
25 import java.util.Objects;
26 import java.util.stream.Collectors;
27 import java.util.stream.Stream;
28
29 import static ch.geomo.tramaps.conflict.ConflictFinder.CONFLICT_COMPARATOR;
30 import static ch.geomo.util.geom.Axis.X;
31
32 /**
33 * Provides a common implementation of certain methods and properties of a {@link Conflict} to reduce redundancy and
34 * duplicated code.
35 */
36 public abstract class AbstractConflict implements Conflict {
37
38     protected final ElementBufferPair buffers;
39
40     /**
41     * Projection of the displace vector along the x-axis and its rejection.
42     */
43     protected Pair<MoveVector> projection;
44
45     protected ConflictType conflictType;
46     protected MoveVector displaceVector;
47     protected MoveVector bestDisplaceVector;
48     protected Axis bestDisplaceAxis;
49     protected Coordinate bestDisplaceStartPoint;
50     protected boolean solved = false;
51
52     public AbstractConflict(@NotNull Pair<ElementBuffer> bufferPair) {
53         buffers = new ElementBufferPair(bufferPair.getFirst(), bufferPair.getSecond());
54     }
55
56     /**
```

```

57     * @return the {@link MoveVector} along x-axis
58     */
59     @NotNull
60     protected MoveVector getMoveVectorAlongX() {
61         return projection.getFirst();
62     }
63
64     /**
65     * @return the {@link MoveVector} along y-axis
66     */
67     @NotNull
68     protected MoveVector getMoveVectorAlongY() {
69         return projection.getSecond();
70     }
71
72     @NotNull
73     @Override
74     public ConflictType getConflictType() {
75         return conflictType;
76     }
77
78     @NotNull
79     @Override
80     public MoveVector getDisplaceVector() {
81         return displaceVector;
82     }
83
84     /**
85     * @return best move direction along an axis
86     */
87     @NotNull
88     @Override
89     public OctilinearDirection getBestDisplaceDirection() {
90         if (bestDisplaceAxis == X) {
91             return OctilinearDirection.EAST;
92         }
93         return OctilinearDirection.NORTH;
94     }
95
96     /**
97     * @return best move distance along the best move direction
98     */
99     @Override
100    public double getBestDisplaceDistance() {
101        return Math.ceil(Math.abs(bestDisplaceVector.length()));
102    }
103
104    @NotNull
105    @Override
106    public ElementBuffer getBufferA() {
107        return buffers.first();
108    }
109
110    @NotNull
111    @Override
112    public ElementBuffer getBufferB() {
113        return buffers.second();
114    }
115
116    /**
117    * @return all nodes as a list (list's size is 0, 1 or 2 nodes depending on the conflict type)
118    */
119    @NotNull
120    public List<Node> getNodes() {
121        return Stream.of(getBufferA().getElement(), getBufferB().getElement())
122            .filter(element -> element instanceof Node)
123            .map(element -> (Node) element)
124            .collect(Collectors.toList());
125    }
126
127    /**
128    * @return all edges as a list (list's size is 0, 1 or 2 edges depending on the conflict type)
129    */
130    @NotNull
131    public List<Edge> getEdges() {
132        return Stream.of(getBufferA().getElement(), getBufferB().getElement())
133            .filter(element -> element instanceof Edge)
134            .map(element -> (Edge) element)
135            .collect(Collectors.toList());
136    }

```

```

137
138 @Override
139 public boolean isSolved() {
140     return solved;
141 }
142
143 @NotNull
144 @Override
145 public Coordinate getDisplaceOriginPoint() {
146     return bestDisplaceStartPoint;
147 }
148
149 /**
150  * @return the <b>ceiled</b> displace distance along x-axis
151  */
152 @Override
153 public double getDisplaceDistanceAlongX() {
154     return Math.ceil(Math.abs(projection.getFirst().length()));
155 }
156
157 /**
158  * @return the <b>ceiled</b> displace distance along y-axis
159  */
160 @Override
161 public double getDisplaceDistanceAlongY() {
162     return Math.ceil(Math.abs(projection.getSecond().length()));
163 }
164
165 /**
166  * @return if given element is a conflict element
167  */
168 private boolean isConflictElement(@NotNull GraphElement graphElement) {
169     return graphElement.equals(getBufferA().getElement())
170         || graphElement.equals(getBufferB().getElement());
171 }
172
173 /**
174  * @return if given element is adjacent to a conflict element
175  */
176 private boolean isAdjacentToConflictElement(@NotNull GraphElement graphElement) {
177     return graphElement.isAdjacent(getBufferA().getElement())
178         || graphElement.isAdjacent(getBufferB().getElement());
179 }
180
181 /**
182  * @return if given element is a conflict element or adjacent to a conflict element
183  */
184 @Override
185 public boolean isConflictRelated(@NotNull GraphElement graphElement) {
186     return isConflictElement(graphElement) || isAdjacentToConflictElement(graphElement);
187 }
188
189 @NotNull
190 @Override
191 public Envelope getElementBoundingBox() {
192
193     Geometry geomA = getBufferA().getElement().getGeometry();
194     Geometry geomB = getBufferB().getElement().getGeometry();
195
196     // if (getConflictType() == ConflictType.NODE_EDGE) {
197     //     Node node = getNodes().get(0);
198     //     Edge edge = getEdges().get(0);
199     //     if (isEdgeAdjacentNodeConflict(node, edge)) {
200     //         // refactoring required: extract to an util method since function is already used twice
201     //         Node adjacentNode = node.getAdjacentEdges().stream()
202     //             .map(e -> e.getOtherNode(node))
203     //             .filter(n -> n.equals(edge.getNodeA()) || n.equals(edge.getNodeB()))
204     //             .findFirst()
205     //             .orElseThrow(IllegalArgumentException::new);
206     //         geomA = node.getGeometry();
207     //         geomB = edge.getOtherNode(adjacentNode).getGeometry();
208     //     }
209     // }
210
211     GeometryCollection col = GeomUtil.createCollection(Arrays.asList(geomA, geomB));
212     return col.getEnvelopeInternal();
213 }
214 }
215
216 /**

```

```

217     * @return true if this conflict is a node/edge conflict and does have an adjacent node with one end node of the edge
218     */
219     protected boolean isEdgeAdjacentNodeConflict(@NotNull Node node, @NotNull Edge edge) {
220         if (edge.getOriginalDirection(edge.getNodeA()).isDiagonal()) {
221             return node.getAdjacentEdges().stream()
222                 .map(e -> e.getOtherNode(node))
223                 .anyMatch(n -> n.equals(edge.getNodeA()) || n.equals(edge.getNodeB()));
224         }
225         return false;
226     }
227
228     @Override
229     public int compareTo(@NotNull Conflict o) {
230         return CONFLICT_COMPARATOR.compare(this, o);
231     }
232
233     @Override
234     public boolean equals(Object obj) {
235         return obj instanceof AbstractConflict
236             && (Objects.equals(buffers, ((AbstractConflict) obj).buffers));
237     }
238
239     @Override
240     public int hashCode() {
241         return Objects.hash(buffers);
242     }
243
244     @Override
245     public String toString() {
246         return getClass().getSimpleName() + ": {" +
247             "elements=[" + getBufferA() + ", " + getBufferB() + "], " +
248             "distance=" + getBestDisplaceDistance() + ", " +
249             "point=" + bestDisplaceStartPoint + ", " +
250             "axis=" + bestDisplaceAxis + "}";
251     }
252 }
253 }

```

## BufferConflict.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.tramaps.conflict;
6
7  import ch.geomo.tramaps.conflict.buffer.ElementBuffer;
8  import ch.geomo.tramaps.graph.Edge;
9  import ch.geomo.tramaps.graph.Node;
10 import ch.geomo.util.collection.pair.Pair;
11 import ch.geomo.util.geom.GeomUtil;
12 import ch.geomo.util.geom.PolygonUtil;
13 import ch.geomo.util.math.MoveVector;
14 import com.vividsolutions.jts.geom.*;
15 import com.vividsolutions.jts.operation.distance.DistanceOp;
16 import org.geotools.geometry.jts.JTS;
17 import org.jetbrains.annotations.NotNull;
18
19 import java.util.Collection;
20 import java.util.stream.Stream;
21
22 import static ch.geomo.tramaps.conflict.ConflictType.NODE_EDGE;
23 import static ch.geomo.util.geom.Axis.X;
24 import static ch.geomo.util.geom.Axis.Y;
25
26 /**
27 * Represents conflicts between overlapping elements/buffers, eg. node/edge, edge/edge or node/node.
28 * @see ConflictType
29 */
30 public class BufferConflict extends AbstractConflict {
31
32     private Polygon conflictPolygon;
33     private Geometry conflictArea;
34
35     public BufferConflict(@NotNull Pair<ElementBuffer> bufferPair) {
36         super(bufferPair);
37         evaluateConflictType();
38         initConflict();
39     }

```

```

40
41  /**
42  * Evaluates the type of this conflict in order to be prioritized and easier comparision.
43  */
44  private void evaluateConflictType() {
45      if (buffers.isNodePair() && buffers.hasAdjacentElements()) {
46          if (isAdjacentNodeNodeDiagonal()) {
47              conflictType = ConflictType.ADJACENT_NODE_NODE_DIAGONAL;
48          }
49          else {
50              conflictType = ConflictType.ADJACENT_NODE_NODE;
51          }
52      }
53      else if (buffers.isNodePair()) {
54          conflictType = ConflictType.NODE_NODE;
55      }
56      else if (buffers.isEdgePair()) {
57          conflictType = ConflictType.EDGE_EDGE;
58      }
59      else {
60          conflictType = NODE_EDGE;
61      }
62  }
63
64  /**
65  * @return true if the shared adjacent edge of both conflict nodes is diagonal
66  */
67  private boolean isAdjacentNodeNodeDiagonal() {
68      return getNodes().get(0)
69          .getAdjacentEdgeWith(getNodes().get(1)).getOriginalDirection(getNodes().get(0))
70          .isDiagonal();
71  }
72
73  /**
74  * Initialize this conflict.
75  */
76  private void initConflict() {
77
78      initConflictArea();
79
80      // create conflict polygon
81      conflictPolygon = createConflictPolygon();
82      //Loggers.info(this, "Conflict Polygon: " + conflictPolygon);
83
84      // create exact move vector
85      displaceVector = PolygonUtil.findLongestParallelLineString(conflictPolygon, createQ())
86          .map(MoveVector::new)
87          .orElse(new MoveVector());
88      //Loggers.info(this, "Displace Vector:" + displaceVector);
89
90      projection = displaceVector.getProjection(MoveVector.VECTOR_ALONG_X_AXIS);
91
92      // default values
93      bestDisplaceStartPoint = conflictPolygon.getCentroid().getCoordinate();
94
95      // handle different conflict types
96      switch (conflictType) {
97          case NODE_EDGE: {
98              initNodeEdgeConflict(getNodes().get(0), getEdges().get(0));
99              break;
100          }
101          case NODE_NODE:
102          case ADJACENT_NODE_NODE:
103          case ADJACENT_NODE_NODE_DIAGONAL: {
104              initNodeNodeConflict(getNodes().get(0), getNodes().get(1));
105              break;
106          }
107          case EDGE_EDGE: {
108              initEdgeEdgeConflict();
109              break;
110          }
111      }
112
113      // if (conflictType == NODE_EDGE) {
114      //     Node node = getNodes().get(0);
115      //     Edge edge = getEdges().get(0);
116      //     if (node.isAdjacent(edge.getNodeA()) || node.isAdjacent(edge.getNodeB())) {
117      //         // we currently ignore this kind set conflict
118      //         // fix required: remove workaround and find another solution
119      //         solved = true;

```

```

120 //     }
121 //     }
122
123     if (conflictPolygon.isEmpty()) {
124         solved = true;
125     }
126
127 }
128
129 /**
130  * @return polygon set the intersecting area set both buffers
131  */
132 @NotNull
133 private Polygon createConflictPolygon() {
134
135     Geometry geometry = getBufferA().getBuffer().intersection(getBufferB().getBuffer());
136     //     if (getConflictType() == NODE_EDGE) {
137     //         Node node = getNodes().get(0);
138     //         Edge edge = getEdges().get(0);
139     //         if (isEdgeAdjacentNodeConflict(node, edge)) {
140     //             return JTS.toGeometry(geometry.getEnvelopeInternal());
141     //         }
142     //     }
143
144     if (geometry instanceof Polygon) {
145         return (Polygon) geometry;
146     }
147     // result of intersection is not a polygon, it's a geometry with a dimension of 1 or less, so we do not
148     // care about this conflict at the moment since it's a very small one
149     // Loggers.info(this, "Cannot create conflict polygon. Result was: " + geometry);
150     return GeomUtil.createEmptyPolygon();
151 }
152
153 /**
154  * @return the {@link LineString} between the buffer element's centroid
155  */
156 @NotNull
157 private LineString createQ() {
158     return GeomUtil.createLineString(getBufferA().getElement().getCentroid(), getBufferB().getElement().getCentroid());
159 }
160
161 /**
162  * Initializes and creates the conflict area. The conflict area is within or identical with the conflict polygon
163  * and allows to define the starting point for the displacement. The starting point must always within the
164  * conflict area, otherwise it would not be between both conflict elements and therefore never solve the conflict.
165  */
166 private void initConflictArea() {
167
168     switch (conflictType) {
169         case NODE_NODE:
170         case ADJACENT_NODE_NODE:
171         case ADJACENT_NODE_NODE_DIAGONAL: {
172             conflictArea = GeomUtil.createLineString(getNodes().get(0), getNodes().get(1));
173             break;
174         }
175         case NODE_EDGE: {
176             conflictArea = GeomUtil.createPolygon(getNodes().get(0), getEdges().get(0).getNodeA(), getEdges().get(0).getNodeB(),
177                 getNodes().get(0));
178             break;
179         }
180         case EDGE_EDGE: {
181
182             Node a1 = getEdges().get(0).getNodeA();
183             Node b1 = getEdges().get(0).getNodeB();
184             Node a2 = getEdges().get(1).getNodeA();
185             Node b2 = getEdges().get(1).getNodeB();
186
187             // improvement required: replace try and error approach
188             conflictArea = GeomUtil.createPolygon(a1, b1, b2, a2, a1);
189             if (!conflictArea.isValid()) {
190                 conflictArea = GeomUtil.createPolygon(a1, b1, a2, b2, a1);
191                 if (!conflictArea.isValid()) {
192                     Coordinate[] coordinates = Stream.of(a1, a2, b1, b2).map(Node::getCoordinate).toArray(Coordinate[]::new);
193                     conflictArea = GeomUtil.createLineString(coordinates);
194                 }
195             }
196             break;
197         }
198     }

```

```

199     }
200
201   }
202
203   /**
204    * Evaluates and initialize the best displace vector as well as the best displace axis.
205    */
206   private void initBestDisplaceVector() {
207       // choosing the projection with the smallest angle to an axis
208       double dx = Math.abs(getBufferA().getElement().getCentroid().getX() - getBufferB().getElement().getCentroid().getX());
209       double dy = Math.abs(getBufferA().getElement().getCentroid().getY() - getBufferB().getElement().getCentroid().getY());
210       if (dy < dx) {
211           bestDisplaceVector = getMoveVectorAlongX();
212           bestDisplaceAxis = X;
213       }
214       else {
215           bestDisplaceVector = getMoveVectorAlongY();
216           bestDisplaceAxis = Y;
217       }
218   }
219
220   /**
221    * Initialize an edge/edge conflict.
222    */
223   private void initEdgeEdgeConflict() {
224       initBestDisplaceVector();
225       try {
226           Geometry intersection = conflictPolygon.intersection(conflictArea);
227           if (intersection.isEmpty()) {
228               solved = true;
229           }
230           else {
231               // only point within the conflict area can be handled since starting points outside of the conflict area
232               // may not be between both edges and would never solve the conflict
233               bestDisplaceStartPoint = intersection.getCentroid().getCoordinate();
234           }
235       }
236       catch (TopologyException e) {
237           // fix required: remove workaround for TopologyException
238           bestDisplaceStartPoint = conflictArea.getCentroid().getCoordinate();
239       }
240   }
241
242   /**
243    * Initialize an node/edge conflict.
244    */
245   private void initNodeEdgeConflict(@NotNull Node node, @NotNull Edge edge) {
246
247       if (node.isWestOf(edge) || node.isEastOf(edge)) {
248           bestDisplaceVector = getMoveVectorAlongX();
249           bestDisplaceAxis = X;
250       }
251       else if (node.isSouthOf(edge) || node.isNorthOf(edge)) {
252           bestDisplaceVector = getMoveVectorAlongY();
253           bestDisplaceAxis = Y;
254       }
255       else {
256           // find the best move direction done by calculating the deltas between the node
257           // and both end nodes of the edge, move along the other axis than the axis with
258           // the bigger delta
259           double dxa = Math.abs(node.getX() - edge.getNodeA().getX());
260           double dya = Math.abs(node.getY() - edge.getNodeA().getY());
261           double dxb = Math.abs(node.getX() - edge.getNodeB().getX());
262           double dyb = Math.abs(node.getY() - edge.getNodeB().getY());
263           if (Math.max(dxa, dxb) < Math.max(dya, dyb)) {
264               bestDisplaceVector = getMoveVectorAlongX();
265               bestDisplaceAxis = X;
266           }
267           else {
268               bestDisplaceVector = getMoveVectorAlongY();
269               bestDisplaceAxis = Y;
270           }
271       }
272
273       // if (isEdgeAdjacentNodeConflict(node, edge)) {
274       //     Envelope bbox = conflictPolygon.getEnvelopeInternal();
275       //     if (bbox.getWidth() > bbox.getHeight()) {
276       //         bestDisplaceAxis = X;
277       //         bestDisplaceVector = new MoveVector(bbox.getWidth(), 0);
278       //     }

```

```

279 //         else {
280 //             bestDisplaceAxis = Y;
281 //             bestDisplaceVector = new MoveVector(0, bbox.getHeight());
282 //         }
283 //     }
284
285 // set the best displace start point half way on the line between the nearest points of the edge and the node
286 Coordinate nearestPoint = DistanceOp.nearestPoints(edge.getGeometry(), node.getGeometry())[0];
287 LineString line = GeomUtil.createLineString(node.getCoordinate(), nearestPoint);
288 bestDisplaceStartPoint = line.getCentroid().getCoordinate();
289
290 }
291
292 /**
293  * Initialize an node/node conflict.
294  */
295 private void initNodeNodeConflict(@NotNull Node node1, @NotNull Node node2) {
296     if (node1.getX() == node2.getX() && node1.getY() == node2.getY()) {
297         throw new IllegalStateException("Both conflict nodes has equal position. Please check input data!");
298     }
299     initBestDisplaceVector();
300     bestDisplaceStartPoint = GeomUtil.createLineString(node1, node2).getCentroid().getCoordinate();
301 }
302
303 /**
304  * @return the conflict polygon
305  */
306 public Polygon getConflictPolygon() {
307     return conflictPolygon;
308 }
309
310 /**
311  * Evaluates if both conflict elements are neighbors. This is given when the conflict area is not crossed
312  * by another edge.
313  * @return true if the given edges <b>do not</b> cross with the conflict area
314  */
315 @SuppressWarnings("unused")
316 protected boolean hasElementNeighborhood(@NotNull Collection<Edge> edges) {
317     return edges.stream()
318         .noneMatch(edge -> conflictArea.crosses(edge.getLineString()));
319 }
320
321 }

```

## Conflict.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.tramaps.conflict;
6
7  import ch.geomo.tramaps.conflict.buffer.ElementBuffer;
8  import ch.geomo.tramaps.graph.GraphElement;
9  import ch.geomo.tramaps.graph.direction.OctilinearDirection;
10 import ch.geomo.util.math.MoveVector;
11 import com.vividsolutions.jts.geom.Coordinate;
12 import com.vividsolutions.jts.geom.Envelope;
13 import org.jetbrains.annotations.NotNull;
14
15 /**
16  * Represents a {@link Conflict} within a graph layout.
17  */
18 public interface Conflict extends Comparable<Conflict> {
19
20     /**
21      * @return the necessary displace distance along the <b>x</b>-axis to solve this conflict
22      */
23     double getDisplaceDistanceAlongX();
24
25     /**
26      * @return the necessary displace distance along the <b>y</b>-axis to solve this conflict
27      */
28     double getDisplaceDistanceAlongY();
29
30     /**
31      * @return the type of this conflict
32      */
33     @NotNull

```



```

34 ConflictType getConflictType();
35
36 /**
37  * @return the required displace vector to solve this conflict (v)
38  */
39 @NotNull
40 MoveVector getDisplaceVector();
41
42 /**
43  * @return the best displacement direction based on the best displacement axis
44  */
45 @NotNull
46 OctilinearDirection getBestDisplaceDirection();
47
48 /**
49  * @return the best displacement distance based on the best displacement axis
50  */
51 double getBestDisplaceDistance();
52
53 /**
54  * @return a point (p) through which the displacement line (g) is running
55  */
56 @NotNull
57 Coordinate getDisplaceOriginPoint();
58
59 /**
60  * @return true if this {@link Conflict} has <b>not</b> been solved meanwhile
61  */
62 default boolean isNotSolved() {
63     return !isSolved();
64 }
65
66 /**
67  * @return true if this {@link Conflict} has been solved meanwhile
68  */
69 boolean isSolved();
70
71 /**
72  * @return the first buffer element
73  */
74 @NotNull
75 ElementBuffer getBufferA();
76
77 /**
78  * @return the second buffer element
79  */
80 @NotNull
81 ElementBuffer getBufferB();
82
83 /**
84  * @return true if the given {@link GraphElement} is related to this conflict
85  */
86 boolean isConflictRelated(@NotNull GraphElement graphElement);
87
88 @NotNull
89 Envelope getElementBoundingBox();
90
91 }

```

## ConflictComparator.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.tramaps.conflict;
6
7  import java.util.Comparator;
8
9  /**
10  * Compares two conflicts in order to prioritise them for conflict solving.
11  */
12  public class ConflictComparator implements Comparator<Conflict> {
13
14      @Override
15      public int compare(Conflict o1, Conflict o2) {
16
17          int rank1 = o1.getConflictType().getConflictRank();
18          int rank2 = o2.getConflictType().getConflictRank();

```

```

19     if (rank1 != rank2) {
20         return Integer.compare(rank1, rank2);
21     }
22
23     double length1a = o1.getBestDisplaceDistance();
24     double length2a = o2.getBestDisplaceDistance();
25     if (length1a != length2a) {
26         return Double.compare(length1a, length2a);
27     }
28
29     double length1b = o1.getDisplaceVector().length();
30     double length2b = o2.getDisplaceVector().length();
31     if (length1b != length2b) {
32         return Double.compare(length1b, length2b);
33     }
34
35     double x1 = o1.getDisplaceDistanceAlongX();
36     double x2 = o2.getDisplaceDistanceAlongX();
37     if (x1 != x2) {
38         return Double.compare(x1, x2);
39     }
40
41     double y1 = o1.getDisplaceDistanceAlongY();
42     double y2 = o2.getDisplaceDistanceAlongY();
43     if (y1 != y2) {
44         return Double.compare(y1, y2);
45     }
46
47     // fix/analysis required: after displacing nodes more than one OctilinearConflict
48     // may occur and must be solved. however, these conflict has an equal distance
49     // and displace vector to be solved. therefore they have currently a randomised
50     // sorting among themselves
51     return 0;
52 }
53 }
54 }
55 }

```

## ConflictFinder.java

```

1  /*
2   * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3   */
4
5  package ch.geomo.tramaps.conflict;
6
7  import ch.geomo.tramaps.conflict.buffer.EdgeBuffer;
8  import ch.geomo.tramaps.conflict.buffer.ElementBuffer;
9  import ch.geomo.tramaps.conflict.buffer.ElementBufferPair;
10 import ch.geomo.tramaps.conflict.buffer.NodeBuffer;
11 import ch.geomo.tramaps.graph.Edge;
12 import ch.geomo.tramaps.graph.Node;
13 import ch.geomo.tramaps.map.MetroMap;
14 import ch.geomo.util.collection.GCollectors;
15 import ch.geomo.util.collection.list.EnhancedList;
16 import ch.geomo.util.collection.pair.Pair;
17 import ch.geomo.util.collection.set.EnhancedSet;
18 import ch.geomo.util.collection.set.GSet;
19 import org.jetbrains.annotations.NotNull;
20
21 import java.util.Comparator;
22 import java.util.function.Predicate;
23 import java.util.stream.Stream;
24
25 /**
26  * Provides methods to find conflicts.
27  */
28 public class ConflictFinder {
29
30     /**
31      * Comparator to compare {@link Conflict} instances.
32      */
33     public final static Comparator<Conflict> CONFLICT_COMPARATOR = new ConflictComparator();
34
35     /**
36      * {@link Predicate} returns true if both elements are not equal and not adjacent or at least one element is a node.
37      */
38     private final static Predicate<Pair<ElementBuffer>> CONFLICT_PAIR_PREDICATE = (Pair<ElementBuffer> pair) -> {
39

```

```

40     ElementBufferPair bufferPair = new ElementBufferPair(pair);
41
42     if (bufferPair.hasEqualElements()) {
43         return false;
44     }
45     else if (!bufferPair.hasAdjacentElements()) {
46         return true;
47     }
48     return bufferPair.isNodePair();
49
50 };
51
52 private final MetroMap map;
53
54 private final double routeMargin;
55 private final double edgeMargin;
56 private final double nodeMargin;
57
58 public ConflictFinder(@NotNull MetroMap map, double routeMargin, double edgeMargin, double nodeMargin) {
59     this.map = map;
60     this.routeMargin = routeMargin;
61     this.edgeMargin = edgeMargin;
62     this.nodeMargin = nodeMargin;
63 }
64
65 /**
66  * @return all edge buffers as a {@link Stream}
67  */
68 @NotNull
69 private Stream<ElementBuffer> createEdgeBuffers() {
70     return map.getEdges().stream()
71         .map(edge -> new EdgeBuffer(edge, routeMargin, edgeMargin));
72 }
73
74 /**
75  * @return all node buffers as a {@link Stream}
76  */
77 @NotNull
78 private Stream<ElementBuffer> createNodeBuffers() {
79     return map.getNodes().stream()
80         .map(node -> new NodeBuffer(node, nodeMargin));
81 }
82
83 /**
84  * @return all current {@link BufferConflict}s
85  */
86 @NotNull
87 private EnhancedList<Conflict> getBufferConflicts() {
88     return getConflictElements().stream()
89         // check interior intersection
90         .filter(ConflictFinder::intersects)
91         // create conflict
92         .map(BufferConflict::new)
93         // filter conflicts which do not cross with other (not-conflict related) edges
94         // .filter(conflict -> conflict.hasElementNeighborhood(map.getEdges()))
95         // filter unsolved conflicts
96         .filter(BufferConflict::isNotSolved)
97         // remove duplicates
98         .distinct()
99         .collect(GCollectors.toList());
100 }
101
102 /**
103  * @return all current {@link OctilinearConflict}s
104  */
105 @NotNull
106 private EnhancedList<Conflict> getOctilinearConflicts(double correctionFactor, boolean majorMisalignmentOnly) {
107     // fix/improvement required: buffers are not required, should be rewritten without using set of buffers
108     return getConflictElements().stream()
109         // check conflict
110         .filter(bufferPair -> hasOctilinearConflict(bufferPair, majorMisalignmentOnly))
111         // create conflict
112         .map(bufferPair -> new OctilinearConflict(bufferPair, correctionFactor))
113         // remove duplicates
114         .distinct()
115         // sort conflicts (smallest conflict first)
116         .sorted()
117         .collect(GCollectors.toList());
118 }
119

```

```

120  /**
121  * Evaluates if given buffer pair does have a misalignment respectively a non-octilinear edge. If the second
122  * parameters is true, then only misalignment with a wrong angle greater than 27.5 degree will be considered.
123  * @return if the given buffer pair does have a misalignment
124  */
125  private boolean hasOctilinearConflict(@NotNull Pair<ElementBuffer> bufferPair, boolean majorMisalignmentOnly) {
126      if (bufferPair.stream().allMatch(buffer -> buffer instanceof NodeBuffer)) {
127          Node a = (Node) bufferPair.getFirst().getElement();
128          Node b = (Node) bufferPair.getSecond().getElement();
129          Edge adjacentEdge = a.getAdjacentEdgeWith(b);
130          if (adjacentEdge != null && adjacentEdge.isNotOctilinear()) {
131              return !majorMisalignmentOnly || adjacentEdge.hasMajorMisalignment();
132          }
133      }
134      return false;
135  }
136
137  /**
138  * @return all pairs of conflict elements
139  */
140  @NotNull
141  private EnhancedSet<Pair<ElementBuffer>> getConflictElements() {
142      EnhancedSet<ElementBuffer> buffers = GSet.createSet(createEdgeBuffers(), createNodeBuffers());
143      return buffers.toPairSet(ConflictFinder.CONFLICT_PAIR_PREDICATE);
144  }
145
146  /**
147  * Returns all {@link BufferConflict}s and {@link OctilinearConflict}s. The first parameter configures an instance
148  * of {@link OctilinearConflict} in order to initialize its move vector. If the second parameter is true, only
149  * only misalignment with a wrong angle greater than 27.5 degree will be considered.
150  * @return all {@link BufferConflict}s and {@link OctilinearConflict}s
151  */
152  @NotNull
153  public EnhancedList<Conflict> getConflicts(double correctionFactor, boolean majorMisalignmentOnly) {
154      return getBufferConflicts()
155          .union(getOctilinearConflicts(correctionFactor, majorMisalignmentOnly))
156          .sortElements(CONFLICT_COMPARATOR);
157  }
158
159  /**
160  * @return true if the interior of the element buffers intersects
161  */
162  private static boolean intersects(@NotNull Pair<ElementBuffer> bufferPair) {
163      return bufferPair.getFirst().getBuffer().relate(bufferPair.getSecond().getBuffer(), "T*****");
164  }
165
166  /**
167  * @return true if the given {@link Node} and the given {@link Edge} has a conflict
168  */
169  public static boolean hasConflict(@NotNull Node node, @NotNull Edge edge, @NotNull MetroMap map) {
170      NodeBuffer nodeBuffer = new NodeBuffer(node, map.getNodeMargin());
171      EdgeBuffer edgeBuffer = new EdgeBuffer(edge, map.getRouteMargin(), map.getEdgeMargin());
172      Pair<ElementBuffer> bufferPair = Pair.of(nodeBuffer, edgeBuffer);
173      boolean intersects = intersects(bufferPair);
174      if (intersects) {
175          BufferConflict conflict = new BufferConflict(bufferPair);
176          return !conflict.isSolved();
177      }
178      return false;
179  }
180
181  /**
182  * @return true if the given {@link Node}s has a conflict
183  */
184  public static boolean hasConflict(@NotNull Node node1, @NotNull Node node2, @NotNull MetroMap map) {
185      NodeBuffer nodeBuffer1 = new NodeBuffer(node1, map.getNodeMargin());
186      NodeBuffer nodeBuffer2 = new NodeBuffer(node2, map.getNodeMargin());
187      Pair<ElementBuffer> bufferPair = Pair.of(nodeBuffer1, nodeBuffer2);
188      boolean intersects = intersects(bufferPair);
189      if (intersects) {
190          BufferConflict conflict = new BufferConflict(bufferPair);
191          return !conflict.isSolved();
192      }
193      return false;
194  }
195
196  }

```

## ConflictType.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.tramaps.conflict;
6
7  /**
8   * Represents the type of a {@link Conflict}.
9   */
10 public enum ConflictType {
11
12     /**
13      * Represents an {@link OctilinearConflict} which occurs if a diagonal edge
14      * is not octilinear.
15      */
16     OCTILINEAR(50),
17     /**
18      * Represents a {@link BufferConflict} with two adjacent nodes which shares
19      * an identical edge.
20      */
21     ADJACENT_NODE_NODE_DIAGONAL(50),
22     /**
23      * Represents a {@link BufferConflict} with two adjacent nodes which <b>do not</b>
24      * share an identical edge.
25      */
26     ADJACENT_NODE_NODE(25),
27     /**
28      * Represents a {@link BufferConflict} with two <b>non-adjacent</b> nodes.
29      */
30     NODE_NODE(20),
31     /**
32      * Represents a {@link BufferConflict} between a node and a non-adjacent edge.
33      */
34     NODE_EDGE(10),
35     /**
36      * Represents a {@link BufferConflict} between two non-adjacent edges.
37      */
38     EDGE_EDGE(5);
39
40     private final int conflictRank;
41
42     ConflictType(int conflictRank) {
43         this.conflictRank = conflictRank;
44     }
45
46     /**
47      * @return the rank defining the priority order to solve the conflict
48      */
49     public int getConflictRank() {
50         return conflictRank;
51     }
52 }
53 }

```

## OctilinearConflict.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.tramaps.conflict;
6
7  import ch.geomo.tramaps.conflict.buffer.ElementBuffer;
8  import ch.geomo.tramaps.graph.Edge;
9  import ch.geomo.tramaps.graph.Node;
10 import ch.geomo.util.collection.pair.Pair;
11 import ch.geomo.util.geom.Axis;
12 import ch.geomo.util.math.MoveVector;
13 import com.vividsolutions.jts.math.Vector2D;
14 import org.jetbrains.annotations.NotNull;
15
16 /**
17  * Conflict between two adjacent nodes with a diagonal edge. This kind of conflict is given when the difference between
18  * the angle of the original octilinear diagonal and the current angle is greater than 27.5 degree.
19  * <p>
20  * Using this conflict avoids prevents a diagonal edge from changing to another quadrant of the cartesian coordinate
21  * system.
22  * <p>
23  * The property {@link OctilinearConflict#correctionFactor} regulates how much the angle should be corrected. The

```

```

24  * value 1 (one) should restore the octilinearity may leads to a very stretched layout when correcting these kinds
25  * of {@link Conflict} iteratively.
26  * @see ConflictFinder#hasOctilinearConflict(Pair, boolean)
27  */
28  public class OctilinearConflict extends AbstractConflict {
29
30      private final double correctionFactor;
31
32      public OctilinearConflict(@NotNull Pair<ElementBuffer> bufferPair, double correctionFactor) {
33          super(bufferPair);
34          this.correctionFactor = correctionFactor;
35          conflictType = ConflictType.OCTILINEAR;
36          initConflict();
37      }
38
39      /**
40       * Initialize this conflict.
41       */
42      private void initConflict() {
43
44          Node nodeA = getNodes().get(0);
45          Node nodeB = getNodes().get(1);
46          Edge adjacentEdge = nodeA.getAdjacentEdgeWith(nodeB);
47
48          if (adjacentEdge == null) {
49              solved = true;
50              return;
51          }
52
53          double dx = Math.abs(nodeA.getX() - nodeB.getX());
54          double dy = Math.abs(nodeA.getY() - nodeB.getY());
55
56          double diff = Math.abs(dx - dy) * correctionFactor;
57
58          if (dx > dy) {
59              bestDisplaceAxis = Axis.Y;
60              displaceVector = new MoveVector(new Vector2D(0, diff));
61          }
62          else {
63              bestDisplaceAxis = Axis.X;
64              displaceVector = new MoveVector(new Vector2D(diff, 0));
65          }
66          bestDisplaceVector = displaceVector;
67          bestDisplaceStartPoint = adjacentEdge.getCentroid().getCoordinate();
68
69          projection = displaceVector.getProjection(MoveVector.VECTOR_ALONG_X_AXIS);
70
71          if (!adjacentEdge.hasMajorMisalignment()) {
72              solved = true;
73          }
74      }
75  }
76
77  /**
78   * Returns the displace distance along the x-axis. In order to solve the conflict when
79   * using a correction factor of 1, this implementation does not return a rounded value
80   * like the default implementation does.
81   * @return AbstractConflict#getDisplaceDistanceAlongX
82   */
83  @Override
84  public double getDisplaceDistanceAlongX() {
85      return Math.abs(projection.getFirst().length());
86  }
87
88  /**
89   * Returns the displace distance along the y-axis. In order to solve the conflict when
90   * using a correction factor of 1, this implementation does not return a rounded value
91   * like the default implementation does.
92   * @return AbstractConflict#getDisplaceDistanceAlongY
93   */
94  @Override
95  public double getDisplaceDistanceAlongY() {
96      return Math.abs(projection.getSecond().length());
97  }
98
99  }

```

## D.2 Verdrängungsalgorithmus

### D.2.1 Package `ch.geomo.tramaps.map.displacement.*`

#### LineSpaceHandler.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.tramaps.map.displacement;
6
7  import ch.geomo.tramaps.MainApp;
8
9  /**
10 * Convenience interface for {@link MainApp} in order to easily replace algorithms/approaches.
11 */
12 public interface LineSpaceHandler {
13
14     /**
15     * Starts algorithm and makes space for line and station signatures.
16     */
17     void makeSpace();
18
19 }

```

#### DisplaceLineSpaceHandler.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.tramaps.map.displacement.alg;
6
7  import ch.geomo.tramaps.conflict.BufferConflict;
8  import ch.geomo.tramaps.conflict.Conflict;
9  import ch.geomo.tramaps.conflict.ConflictType;
10 import ch.geomo.tramaps.graph.Edge;
11 import ch.geomo.tramaps.graph.Node;
12 import ch.geomo.tramaps.map.MetroMap;
13 import ch.geomo.tramaps.map.displacement.LineSpaceHandler;
14 import ch.geomo.tramaps.map.displacement.alg.adjustment.EdgeAdjuster;
15 import ch.geomo.util.collection.list.EnhancedList;
16 import ch.geomo.util.logging.Loggers;
17 import com.vividolutions.jts.geom.Envelope;
18 import org.jetbrains.annotations.NotNull;
19 import org.jetbrains.annotations.Nullable;
20
21 /**
22 * This {@link LineSpaceHandler} implementation makes space by displacing and moving nodes of the underlying graph.
23 */
24 public class DisplaceLineSpaceHandler implements LineSpaceHandler {
25
26     /**
27     * Max iteration until algorithm will be terminated when not found a non-conflict solution.
28     */
29     private static final int MAX_ITERATIONS = 200;
30
31     private final MetroMap map;
32
33     public DisplaceLineSpaceHandler(@NotNull MetroMap map) {
34         this.map = map;
35     }
36
37     /**
38     * Iterates over all non-octilinear edges and corrects them.
39     */
40     private void correctNonOctilinearEdges() {
41         Loggers.info(this, "Non-Octilinear edges: " + map.countNonOctilinearEdges());
42         map.getEdges().stream()
43             .filter(Edge::isNotOctilinear)
44             .sorted(new NonOctilinearEdgeComparator())
45             .forEach(edge -> EdgeAdjuster.correctEdge(map, edge));
46     }
47
48     // private boolean hasOnlyEdgeAdjacentNodeConflicts(@NotNull EnhancedList<Conflict> conflicts) {

```

## D.2. Verdrängungsalgorithmus

```
49 //     return conflicts.allMatch(conflict -> {
50 //         if (conflict.getConflictType() == ConflictType.NODE_EDGE) {
51 //             Node node = ((BufferConflict) conflict).getNodes().get(0);
52 //             Edge edge = ((BufferConflict) conflict).getEdges().get(0);
53 //             return node.getAdjacentEdges().stream()
54 //                 .map(e -> e.getOtherNode(node))
55 //                 .anyMatch(n -> n.equals(edge.getNodeA()) || n.equals(edge.getNodeB()));
56 //         }
57 //     return false;
58 // });
59 // }
60
61 /**
62  * Makes space for line and station signatures by displacing and moving nodes recursively.
63  */
64 private void makeSpace(int lastIteration, @Nullable Conflict lastConflict, double correctionFactor, boolean majorMisalignmentOnly) {
65
66     int currentIteration = lastIteration + 1;
67
68     EnhancedList<Conflict> conflicts = map.evaluateConflicts(true, correctionFactor, majorMisalignmentOnly);
69
70     Loggers.separator(this);
71     Loggers.info(this, "Start iteration: {0}", currentIteration);
72
73     if (!conflicts.isEmpty()) {
74
75         Loggers.warning(this, "Conflicts found: {0}", conflicts.size());
76
77         Conflict conflict = conflicts.get(0);
78         if (lastConflict != null
79             && conflicts.size() > 1
80             && conflict.getBufferA().getElement().equals(lastConflict.getBufferA().getElement())
81             && conflict.getBufferB().getElement().equals(lastConflict.getBufferB().getElement())) {
82
83             // skip conflict to give another conflict a chance to be solved
84             Loggers.warning(this, "Skip conflict for one iteration... Take next one.");
85             conflict = conflicts.get(1);
86
87         }
88
89         Loggers.flag(this, "Handle conflict: {0}", conflict);
90         NodeDisplacer.displace(map, conflict);
91
92         // try to move nodes to correct non-octilinear edges
93         correctNonOctilinearEdges();
94
95         Loggers.warning(this, "Uncorrected non-octilinear edges found: {0}", map.countNonOctilinearEdges());
96
97         // repeat as long as max iteration is not reached
98         if (currentIteration < MAX_ITERATIONS) {
99             makeSpace(currentIteration, conflict, correctionFactor, majorMisalignmentOnly);
100         }
101         else {
102             Loggers.separator(this);
103             Loggers.warning(this, "Max number set iteration reached. Stop algorithm.");
104         }
105     }
106     else {
107         Loggers.separator(this);
108         Loggers.info(this, "No (more) conflicts found.");
109     }
110 }
111
112 }
113
114 /**
115  * @return the bounding box size as a {@link String}
116  */
117 @NotNull
118 private String getBoundingBoxString() {
119     Envelope mapBoundingBox = map.getBoundingBox();
120     return "Size: " + (int) Math.ceil(mapBoundingBox.getWidth()) + "x" + (int) Math.ceil(mapBoundingBox.getHeight());
121 }
122
123 /**
124  * Starts algorithm and makes space for line and station signatures by displacing and moving nodes.
125  */
126 @Override
127 public void makeSpace() {
128
```



```

129     Loggers.separator(this);
130     Loggers.info(this, "Start TRAMAPS algorithm");
131
132     Loggers.separator(this);
133     Loggers.info(this, "Make space for edge and node signatures...");
134     makeSpace(0, null, 0.25, true);
135
136     Loggers.separator(this);
137     Loggers.info(this, "Restore octilinearity...");
138     makeSpace(0, null, 1, false);
139
140     Loggers.separator(this);
141     Loggers.info(this, getBoundingBoxString());
142     map.evaluateConflicts(true)
143         .doIfNotEmpty(list -> Loggers.warning(this, "Remaining conflicts found! :-("))
144         .forEach(conflict -> Loggers.warning(this, "-> {0}", conflict));
145     Loggers.separator(this);
146
147 }
148
149 }

```

## NodeDisplacer.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.tramaps.map.displacement.alg;
6
7  import ch.geomo.tramaps.conflict.BufferConflict;
8  import ch.geomo.tramaps.conflict.Conflict;
9  import ch.geomo.tramaps.graph.Edge;
10 import ch.geomo.tramaps.graph.Node;
11 import ch.geomo.tramaps.graph.direction.OctilinearDirection;
12 import ch.geomo.tramaps.map.MetroMap;
13 import ch.geomo.util.collection.GCollection;
14 import ch.geomo.util.collection.list.EnhancedList;
15 import ch.geomo.util.collection.set.EnhancedSet;
16 import com.vividsolutions.jts.geom.Coordinate;
17 import org.jetbrains.annotations.NotNull;
18
19 import static ch.geomo.tramaps.graph.direction.OctilinearDirection.EAST;
20 import static ch.geomo.tramaps.graph.direction.OctilinearDirection.NORTH;
21
22 /**
23 * Displace nodes based on the given {@link BufferConflict}.
24 */
25 public class NodeDisplacer {
26
27     private final MetroMap map;
28     private final Conflict conflict;
29     private final OctilinearDirection displaceDirection;
30
31     // future improvement: introduce factory class in order to reuse instances
32     public NodeDisplacer(@NotNull MetroMap map, @NotNull Conflict conflict) {
33         this.map = map;
34         this.conflict = conflict;
35         displaceDirection = conflict.getBestDisplaceDirection();
36     }
37
38     /**
39     * @return true if given direction is equals to the displace direction
40     */
41     private boolean isDisplaceDirection(@NotNull OctilinearDirection direction) {
42         return displaceDirection == direction;
43     }
44
45     /**
46     * Checks if given node is adjacent to a connection edge and (re-)evaluates if given node is displaceable.
47     * @return true if displaceable
48     */
49     private boolean checkConnectionEdge(@NotNull Node node, boolean displaceable) {
50         if (hasConnectionEdge(node) && !conflict.isConflictRelated(node)) {
51             if (node.getNodeDegree() == 1) {
52                 return !displaceable;
53             }
54             EnhancedSet<Edge> connectionEdges = getConnectionEdges(node);
55             if (connectionEdges.hasMoreThanOneElement()) {

```

```

56         return displaceable;
57     }
58     Edge connectionEdge = connectionEdges.first().orElse(null);
59     if (connectionEdge == null) {
60         throw new IllegalStateException("Should never happen since this node has a connection edge!");
61     }
62     Node otherNode = connectionEdge.getOtherNode(node);
63     if (isSimpleConnectionEdgeNode(node, connectionEdge) {
64         if ((otherNode.isNorthOf(node) && isDisplaceDirection(NORTH))
65             || (otherNode.isEastOf(node) && isDisplaceDirection(EAST))) {
66             return false;
67         }
68         return displaceable || otherNode.getNodeDegree() != 1;
69     }
70 }
71 return displaceable;
72 }
73
74 /**
75  * @return true if given node is simple to move
76  */
77 private boolean isSimpleConnectionEdgeNode(@NotNull Node node, @NotNull Edge connectionEdge) {
78     return node.getAdjacentEdges().stream()
79         .filter(connectionEdge::isNotEquals)
80         .map(edge -> edge.getOriginalDirection(node))
81         .noneMatch(direction -> {
82             if (isDisplaceDirection(NORTH)) {
83                 return !direction.isVertical();
84             }
85             return !direction.isHorizontal();
86         });
87 }
88
89 /**
90  * @return true if node can be moved northwards.
91  */
92 private boolean isDisplaceableToNorth(@NotNull Node node) {
93     boolean displaceable = node.getPoint().getY() > conflict.getDisplaceOriginPoint().y;
94     return checkConnectionEdge(node, displaceable);
95 }
96
97 /**
98  * @return true if node can be moved eastwards.
99  */
100 private boolean isDisplaceableToEast(@NotNull Node node) {
101     boolean displaceable = node.getPoint().getX() > conflict.getDisplaceOriginPoint().x;
102     return checkConnectionEdge(node, displaceable);
103 }
104
105 /**
106  * Starts the displacement process and displace nodes according to {@link #isDisplaceableToNorth(Node)}
107  * respectively {@link #isDisplaceableToEast(Node)}.
108  */
109 public void displace() {
110
111     EnhancedList<Node> displacedNodes = GCollection.list();
112
113     if (isDisplaceDirection(EAST)) {
114         map.getNodes().stream()
115             .filter(this::isDisplaceableToEast)
116             .forEach(node -> {
117                 node.updateX(node.getX() + conflict.getDisplaceDistanceAlongX());
118                 displacedNodes.add(node);
119             });
120     }
121     else { // NORTH
122         map.getNodes().stream()
123             .filter(this::isDisplaceableToNorth)
124             .forEach(node -> {
125                 node.updateY(node.getY() + conflict.getDisplaceDistanceAlongY());
126                 displacedNodes.add(node);
127             });
128     }
129 }
130 }
131
132 /**
133  * @return true if given edge is a connection edge
134  */
135 private boolean isConnectionEdge(@NotNull Edge edge) {

```

```

136     Coordinate displaceOriginPoint = conflict.getDisplaceOriginPoint();
137     if (isDisplaceDirection(NORTH)) {
138         return edge.getNodeA().isNorthOf(displaceOriginPoint) && edge.getNodeB().isSouthOf(displaceOriginPoint)
139             || edge.getNodeB().isNorthOf(displaceOriginPoint) && edge.getNodeA().isSouthOf(displaceOriginPoint);
140     }
141     return edge.getNodeA().isEastOf(displaceOriginPoint) && edge.getNodeB().isWestOf(displaceOriginPoint)
142         || edge.getNodeB().isEastOf(displaceOriginPoint) && edge.getNodeA().isWestOf(displaceOriginPoint);
143 }
144
145 /**
146  * @return true if given node has at least one adjacent connection edge
147  */
148 private boolean hasConnectionEdge(@NotNull Node node) {
149     return node.getAdjacentEdges().anyMatch(this::isConnectionEdge);
150 }
151
152 /**
153  * @return a set of connection edges
154  */
155 @NotNull
156 private EnhancedSet<Edge> getConnectionEdges(@NotNull Node node) {
157     return node.getAdjacentEdges(this::isConnectionEdge);
158 }
159
160 /**
161  * Starts the displacement process and displace nodes according to {@link #isDisplaceableToNorth(Node)}
162  * respectively {@link #isDisplaceableToEast(Node)}.
163  * <p>
164  * Creates internally a new instance of {@link NodeDisplacer} and invokes {@link NodeDisplacer#displace()}.
165  * @see NodeDisplacer#displace()
166  */
167 public static void displace(@NotNull MetroMap map, @NotNull Conflict conflict) {
168     new NodeDisplacer(map, conflict).displace();
169 }
170
171 }

```

## TraversedNodes.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.tramaps.map.displacement.alg;
6
7  import ch.geomo.tramaps.graph.Node;
8  import ch.geomo.util.collection.GCollection;
9  import ch.geomo.util.collection.list.EnhancedList;
10 import org.jetbrains.annotations.NotNull;
11
12 /**
13  * A transfer object which works as a guard to keep track set already visited nodes.
14  */
15 public class TraversedNodes {
16
17     private final EnhancedList<Node> traversedNodes;
18
19     public TraversedNodes() {
20         traversedNodes = GCollection.list();
21     }
22
23     /**
24      * @return true if given {@link Node} is already traversed/visited
25      */
26     public boolean hasAlreadyVisited(@NotNull Node node) {
27         return traversedNodes.contains(node);
28     }
29
30     /**
31      * Marks the given {@link Node} as visited. Adds the node to the cache set traversed nodes.
32      */
33     public void visited(@NotNull Node node) {
34         traversedNodes.add(node);
35     }
36
37 }

```

## NonOctilinearEdgeComparator.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.tramaps.map.displacement.alg;
6
7  import ch.geomo.tramaps.graph.Edge;
8
9  import java.util.Comparator;
10
11 public class NonOctilinearEdgeComparator implements Comparator<Edge> {
12
13     @Override
14     public int compare(Edge e1, Edge e2) {
15         if (e1.getLength() != e2.getLength()) {
16             return Double.compare(e1.getLength(), e2.getLength());
17         }
18         // improvement required: implement better comparision
19         return Double.compare(e1.getRoutes().size(), e2.getRoutes().size());
20     }
21
22 }

```

## CostCalculator.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.tramaps.map.displacement.alg.adjustment;
6
7  import ch.geomo.tramaps.graph.Edge;
8  import ch.geomo.tramaps.graph.Node;
9  import ch.geomo.tramaps.graph.direction.Direction;
10 import ch.geomo.tramaps.map.displacement.alg.TraversedNodes;
11 import ch.geomo.util.collection.list.EnhancedList;
12 import ch.geomo.util.collection.set.EnhancedSet;
13 import org.jetbrains.annotations.NotNull;
14
15 import java.util.List;
16
17 public enum CostCalculator {
18     ;
19
20     private static final double CORRECT_CIRCLE_PENALTY = 1000;
21
22     public static boolean isSimpleNode(@NotNull Edge connectionEdge, @NotNull Node node) {
23
24         EnhancedSet<Edge> adjacentEdges = node.getAdjacentEdges(connectionEdge);
25
26         if (adjacentEdges.size() == 0 || adjacentEdges.size() == 1) {
27             return true;
28         }
29         else if (adjacentEdges.size() > 2) {
30             return false;
31         }
32
33         EnhancedList<Direction> directions = adjacentEdges
34             .map(edge -> edge.getOriginalDirection(node))
35             .toList();
36         return directions.get(0).isOpposite(directions.get(1));
37     }
38 }
39
40 /**
41 * Calculates the costs to adjust given {@link Edge} by moving given {@link Node}. The {@link List} set traversed
42 * nodes is needed to avoid correction circles.
43 */
44 public static double calculate(@NotNull Edge connectionEdge, @NotNull Node node, @NotNull TraversedNodes guard) {
45
46     if (guard.hasAlreadyVisited(node)) {
47         return CORRECT_CIRCLE_PENALTY;
48     }
49
50     guard.visited(node);
51
52     if (node.getNodeDegree() == 1) {
53         return 0;
54     }

```

```

55
56     if (isSimpleNode(connectionEdge, node)) {
57         if (node.getNodeDegree() == 2) {
58             return 1;
59         }
60         return 2;
61     }
62
63     EnhancedSet<Edge> adjacentEdges = node.getAdjacentEdges(connectionEdge);
64
65     double costs = 2 + adjacentEdges.size();
66
67     for (Edge adjacentEdge : adjacentEdges) {
68         Node otherNode = adjacentEdge.getOtherNode(node);
69         costs = costs + calculate(adjacentEdge, otherNode, guard);
70     }
71
72     return costs;
73
74 }
75
76 }

```

## DirectionEvaluator.java

```

1  /*
2   * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3   */
4
5  package ch.geomo.tramaps.map.displacement.alg.adjustment;
6
7  import ch.geomo.tramaps.graph.Edge;
8  import ch.geomo.tramaps.graph.Node;
9  import ch.geomo.tramaps.graph.direction.Direction;
10 import ch.geomo.tramaps.graph.direction.OctilinearDirection;
11 import ch.geomo.util.collection.GCollectors;
12 import ch.geomo.util.collection.list.EnhancedList;
13 import ch.geomo.util.logging.Loggers;
14 import ch.geomo.util.math.MoveVector;
15 import org.jetbrains.annotations.NotNull;
16
17 /**
18  * Provides methods to evaluate the best move vector in order to correct a non-octilinear edge.
19  */
20 public enum DirectionEvaluator {
21     ;
22
23     /**
24      * Evaluates the best move vector for a node with a degree of 1 and given connection edge.
25      * @return the best move vector to correct given connection edge
26      */
27     @NotNull
28     public static MoveVector evaluateSingleNodeDirection(@NotNull Node moveableNode, @NotNull Edge connectionEdge) {
29
30         double dx = Math.abs(connectionEdge.getNodeA().getX() - connectionEdge.getNodeB().getX());
31         double dy = Math.abs(connectionEdge.getNodeA().getY() - connectionEdge.getNodeB().getY());
32         double diff = Math.abs(dx - dy);
33
34         Node otherNode = connectionEdge.getOtherNode(moveableNode);
35         OctilinearDirection originalDirection = connectionEdge.getOriginalDirection(otherNode).toOctilinear();
36         double angle = originalDirection.getAngleTo(connectionEdge.getDirection(otherNode));
37
38         switch (originalDirection.opposite()) {
39             case NORTH_EAST: {
40                 if (angle < 90) {
41                     return new MoveVector(0, -diff);
42                 }
43                 return new MoveVector(-diff, 0);
44             }
45             case SOUTH_EAST: {
46                 if (angle < 90) {
47                     return new MoveVector(-diff, 0);
48                 }
49                 return new MoveVector(0, diff);
50             }
51             case SOUTH_WEST: {
52                 if (angle < 90) {
53                     return new MoveVector(0, diff);
54                 }

```

```

55         return new MoveVector(diff, 0);
56     }
57     case NORTH_WEST: {
58         if (angle < 90) {
59             return new MoveVector(diff, 0);
60         }
61         return new MoveVector(0, -diff);
62     }
63     default: {
64         // analysis required: do we reach this point? when yes, how can this case be solved?
65         Loggers.info(DirectionEvaluator.class, "Single node {0} has a non-diagonal connection edge. -> Not (yet) treated/implemented.");
66         return new MoveVector(0, 0);
67     }
68 }
69
70 }
71
72 @NotNull
73 private static EnhancedList<OctilinearDirection> getAdjacentEdgeDirections(@NotNull Node node, @NotNull Edge connectionEdge) {
74     return node.getAdjacentEdges().stream()
75         .filter(connectionEdge::isNotEquals)
76         .map(edge -> edge.getOriginalDirection(node).toOctilinear())
77         .collect(GCollectors.toList());
78 }
79
80 @NotNull
81 public static MoveVector evaluateDirection(@NotNull Node moveableNode, @NotNull Edge connectionEdge) {
82
83     double dx = Math.abs(connectionEdge.getNodeA().getX() - connectionEdge.getNodeB().getX());
84     double dy = Math.abs(connectionEdge.getNodeA().getY() - connectionEdge.getNodeB().getY());
85     double diff = Math.abs(dx - dy);
86
87     Node otherNode = connectionEdge.getOtherNode(moveableNode);
88     EnhancedList<OctilinearDirection> directions = getAdjacentEdgeDirections(moveableNode, connectionEdge);
89
90     if (directions.allMatch(Direction::isVertical)) {
91         if (dy > dx) {
92             if (otherNode.isNorthOf(moveableNode)) {
93                 return new MoveVector(0, diff);
94             }
95             return new MoveVector(0, -diff);
96         }
97         if (otherNode.isNorthOf(moveableNode)) {
98             return new MoveVector(0, -diff);
99         }
100        return new MoveVector(0, diff);
101    }
102    else if (directions.allMatch(Direction::isHorizontal)) {
103        if (dx > dy) {
104            if (otherNode.isEastOf(moveableNode)) {
105                return new MoveVector(diff, 0);
106            }
107            return new MoveVector(-diff, 0);
108        }
109        if (otherNode.isEastOf(moveableNode)) {
110            return new MoveVector(-diff, 0);
111        }
112        return new MoveVector(diff, 0);
113    }
114    else if (directions.hasOneElement()) {
115
116        Edge adjacentEdge = moveableNode.getAdjacentEdges(connectionEdge)
117            .first()
118            .orElseThrow(IllegalStateException::new);
119        Direction adjacentEdgeDirection = adjacentEdge.getDirection(moveableNode);
120
121        // future improvement: analyse if we could either ease the layout to improve quality or if we
122        // could correct that node but which may not help since always one edge will remain
123        // non-octilinear. furthermore it may decrease the quality depending on the new angle of the edge
124        if (!adjacentEdgeDirection.isOctilinear()) {
125            OctilinearDirection originalConnectionEdgeDirection = connectionEdge.getOriginalDirection(moveableNode).toOctilinear();
126            if (originalConnectionEdgeDirection.getAngle() == directions.get(0).opposite().getAngle()) {
127                // ease layout (no full correction)
128                // LineString line = GeomUtil.createLineString(connectionEdge.getOtherNode(moveableNode),
129                    adjacentEdge.getOtherNode(moveableNode));
130                // Point centroid = line.getCentroid();
131                // return new MoveVector(moveableNode.getPoint(), centroid);
132                return new MoveVector(0, 0);
133            }
134        }
135    }
136 }

```

```

134
135     }
136
137     return new MoveVector(0, 0);
138
139     }
140
141 }

```

## EdgeAdjuster.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.tramaps.map.displacement.alg.adjustment;
6
7  import ch.geomo.tramaps.conflict.ConflictFinder;
8  import ch.geomo.tramaps.graph.Edge;
9  import ch.geomo.tramaps.graph.Graph;
10 import ch.geomo.tramaps.graph.Node;
11 import ch.geomo.tramaps.graph.layout.OctilinearEdge;
12 import ch.geomo.tramaps.graph.layout.OctilinearEdgeBuilder;
13 import ch.geomo.tramaps.map.MetroMap;
14 import ch.geomo.tramaps.map.displacement.alg.TraversedNodes;
15 import ch.geomo.util.collection.pair.Pair;
16 import ch.geomo.util.geom.GeomUtil;
17 import ch.geomo.util.logging.Loggers;
18 import ch.geomo.util.math.MoveVector;
19 import com.vividolutions.jts.geom.Point;
20 import org.jetbrains.annotations.NotNull;
21
22 import java.util.List;
23 import java.util.stream.Collectors;
24
25 /**
26 * Provides functionality to adjust a non-octilinear edge.
27 */
28 public class EdgeAdjuster {
29
30     private static final double MAX_ADJUSTMENT_COSTS = 100;
31
32     private final MetroMap map;
33     private final Edge edge;
34     private final double maxAdjustmentCosts;
35
36     // future improvement: introduce factory class in order to reuse instances
37     public EdgeAdjuster(@NotNull MetroMap map, @NotNull Edge edge, double maxAdjustmentCosts) {
38         this.map = map;
39         this.edge = edge;
40         this.maxAdjustmentCosts = maxAdjustmentCosts;
41     }
42
43     @NotNull
44     private Node getNodeA() {
45         return edge.getNodeA();
46     }
47
48     @NotNull
49     private Node getNodeB() {
50         return edge.getNodeB();
51     }
52
53     private double getAdjacentEdgeLengthSum(@NotNull Node node) {
54         return node.getAdjacentEdges(edge).stream()
55             .mapToDouble(Edge::getLength)
56             .sum();
57     }
58
59     public void correctEdge() {
60
61         Loggers.info(this, "Correct edge {0}...", edge.getName());
62
63         double scoreA = CostCalculator.calculate(edge, getNodeA(), new TraversedNodes());
64         double scoreB = CostCalculator.calculate(edge, getNodeB(), new TraversedNodes());
65
66         if (scoreA == scoreB) {

```

## D.2. Verdrängungsalgorithmus

```
69     // use node with a bigger length of his adjacent edge (more space for movements)
70     if (getAdjacentEdgeLengthSum(getNodeA()) > getAdjacentEdgeLengthSum(getNodeB())) {
71         scoreA++;
72     }
73     else {
74         scoreB++;
75     }
76 }
77
78 Loggers.info(this, "Adjustment costs for adjacent nodes: {0}/{1}", scoreA, scoreB);
79
80 if (scoreA > maxAdjustmentCosts && scoreB > maxAdjustmentCosts) {
81     correctEdgeByIntroducingBendNodes();
82 }
83 else if (scoreA < scoreB) {
84     correctEdgeByMovingNode(edge, getNodeA(), new TraversedNodes());
85 }
86 else {
87     correctEdgeByMovingNode(edge, getNodeB(), new TraversedNodes());
88 }
89
90 Loggers.info(this, "Correction is done.");
91
92 }
93
94 /**
95  * Introduces a bend node for given {@link Edge}. The given {@link Edge} instance will be destroyed.
96  */
97 private void correctEdgeByIntroducingBendNodes() {
98
99     // create octilinear edge
100    OctilinearEdge octilinearEdge = new OctilinearEdgeBuilder()
101        .setOriginalEdge(edge)
102        .build();
103
104    Pair<Node> vertices = octilinearEdge.getVertices();
105
106    Loggers.info(this, "Introduce bends {0} to edge {1}...", vertices, edge.getName());
107
108    if (vertices.hasNonNullValues()) {
109
110        // only one vertex
111        if (vertices.second() == null) {
112            map.addNodes(vertices.first());
113            edge.getNodeA()
114                .createAdjacentEdgeTo(vertices.first(), edge.getRoutes())
115                .createAdjacentEdgeTo(edge.getNodeB(), edge.getRoutes());
116        }
117        else {
118            map.addNodes(vertices.first(), vertices.second());
119            edge.getNodeA()
120                .createAdjacentEdgeTo(vertices.first(), edge.getRoutes())
121                .createAdjacentEdgeTo(vertices.second(), edge.getRoutes())
122                .createAdjacentEdgeTo(edge.getNodeB(), edge.getRoutes());
123        }
124
125        Loggers.info(this, "Octilinear edge created: {0}", edge);
126
127        // remove old edge
128        edge.destroy();
129
130        // numbers set nodes has changed, edge cache must be flagged for rebuild
131        map.updateGraph();
132    }
133    else {
134        Loggers.warning(this, "No octilinear edge created: {0}", edge);
135    }
136 }
137
138 }
139
140 /**
141  * Moves given {@link Node} in a certain direction to correct the given {@link Edge}'s octilinearity. Prefers to
142  * move in the given (last) move direction if two choices are equal weighted.
143  */
144 private void moveNode(@NotNull Edge connectionEdge, @NotNull Node moveableNode) {
145
146     MoveVector moveVector;
147
148     if (CostCalculator.isSimpleNode(connectionEdge, moveableNode) {
```



## D.2. Verdrängungsalgorithmus

```
149
150     // evaluate direction
151     if (!moveableNode.getAdjacentEdges(connectionEdge).isEmpty()) {
152         Loggers.info(this, "Evaluate move direction of simple node {0}...", moveableNode.getName());
153         moveVector = DirectionEvaluator.evaluateDirection(moveableNode, connectionEdge);
154     }
155     else {
156         Loggers.info(this, "Evaluate move direction of single node {0}...", moveableNode.getName());
157         moveVector = DirectionEvaluator.evaluateSingleNodeDirection(moveableNode, connectionEdge);
158     }
159 }
160 }
161 else {
162     Loggers.flag(this, "Node {0} is not a simple node. Node will not be moved.", moveableNode.getName());
163     moveVector = new MoveVector(0, 0);
164 }
165
166 Loggers.info(this, "Evaluated move vector: {0}", moveVector);
167
168 Point movePoint = GeomUtil.createMovePoint(moveableNode.getPoint(), moveVector);
169
170 boolean overlapsOtherEdges = overlapsWithOtherEdges(moveableNode, movePoint, connectionEdge, map);
171 boolean overlapsAdjacentNode = overlapsWithAdjacentNodes(moveableNode, connectionEdge, moveVector);
172 boolean overlapsOtherNodes = overlapsWithOtherNodes(moveableNode, movePoint);
173
174 if (!overlapsAdjacentNode && !overlapsOtherNodes && !overlapsOtherEdges) {
175
176     Point originalPoint = moveableNode.toPoint(); // clone original position for possible revert operation
177
178     Loggers.flag(this, "Move node {0} using vector {1}.", moveableNode.getName(), moveVector);
179     moveableNode.updatePosition(movePoint.getCoordinate());
180
181     Node otherNode = connectionEdge.getOtherNode(moveableNode);
182
183     // check for new conflicts
184     if (otherNode.getAdjacentEdges(connectionEdge).anyMatch(edge -> ConflictFinder.hasConflict(moveableNode, edge, map))) {
185         Loggers.flag(this, "Revert node movement. Occurs edge/node conflict.");
186         moveableNode.updatePosition(originalPoint.getCoordinate());
187     }
188     else if (moveableNode.getAdjacentEdges().anyMatch(edge -> ConflictFinder.hasConflict(moveableNode, edge.getOtherNode(moveableNode),
189         map))) {
190         Loggers.flag(this, "Revert node movement. Occurs adjacent node/node conflict.");
191         moveableNode.updatePosition(originalPoint.getCoordinate());
192     }
193     else {
194         Loggers.info(this, "New position for node {0}.", moveableNode.getName());
195     }
196 }
197 else {
198     // to be done: log reason
199     Loggers.info(this, "Cannot move node {0}!", moveableNode.getName());
200 }
201 }
202 }
203
204 /**
205  * @return true if new position is equals to a position of another node
206  */
207 private boolean overlapsWithOtherNodes(@NotNull Node moveableNode, @NotNull Point movePoint) {
208     return map.getNodes().stream()
209         .filter(moveableNode::isNotEquals)
210         .map(Node::getCoordinate)
211         .anyMatch(coordinate -> movePoint.getCoordinate().equals(coordinate));
212 }
213
214 /**
215  * @return true if moving the node does overlap another node after moving
216  */
217 private boolean overlapsWithAdjacentNodes(@NotNull Node moveableNode, @NotNull Edge connectionEdge, @NotNull MoveVector moveVector) {
218     // analysis required: how to improve this method? currently only working if moved along the adjacent edge
219     return moveableNode.getAdjacentEdges(connectionEdge)
220         .anyMatch(adjEdge -> adjEdge.getLength() <= moveVector.length());
221 }
222
223 /**
224  * @return true if new position would intersect with any other edge when moving
225  */
226 private boolean overlapsWithOtherEdges(@NotNull Node moveableNode, @NotNull Point movePoint, @NotNull Edge connectionEdge, @NotNull Graph graph) {
227     return moveableNode.getAdjacentEdges().stream()
```

```

228         .map(edge -> edge.getOtherNode(moveableNode))
229         .map(node -> GeomUtil.createLineString(movePoint, node.getPoint()))
230         .anyMatch(lineString -> graph.getEdges().stream()
231             // ignore adjacent edges
232             .filter(edge -> !moveableNode.getAdjacentEdges().contains(edge))
233             // test intersection
234             .filter(edge -> edge.getLineString().relate(lineString, "T*****"))
235             .peek(edge -> Loggers.warning(this, "Edge {0} would intersect with {1}!", edge.getName(), connectionEdge.getName()))
236             .findAny()
237             .isPresent());
238     }
239
240     /**
241     * Corrects the direction set given {@link Edge} recursively by moving the given {@link Node}.
242     */
243     private void correctEdgeByMovingNode(@NotNull Edge edge, @NotNull Node moveableNode, @NotNull TraversedNodes guard) {
244
245         if (guard.hasAlreadyVisited(moveableNode)) {
246             Loggers.warning(this, "Node {0} was already visited. Abort edge correction!", moveableNode.getName());
247             return;
248         }
249
250         guard.visited(moveableNode);
251
252         moveNode(edge, moveableNode);
253
254         List<Edge> nonOctilinearEdges = moveableNode.getAdjacentEdges().stream()
255             .filter(Edge::isNotOctilinear)
256             .filter(edge::isNotEquals)
257             .collect(Collectors.toList());
258
259         for (Edge nonOctilinearEdge : nonOctilinearEdges) {
260             Node otherNode = nonOctilinearEdge.getOtherNode(moveableNode);
261             // tail-end recursion
262             correctEdgeByMovingNode(nonOctilinearEdge, otherNode, guard);
263         }
264     }
265
266     public static void correctEdge(@NotNull MetroMap map, @NotNull Edge edge) {
267         new EdgeAdjuster(map, edge, MAX_ADJUSTMENT_COSTS).correctEdge();
268     }
269
270 }
271 }

```

## D.2.2 Package ch.geomo.tramaps.graph.layout

### OctilinearEdge.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.tramaps.graph.layout;
6
7  import ch.geomo.tramaps.graph.Edge;
8  import ch.geomo.tramaps.graph.Node;
9  import ch.geomo.util.collection.pair.MutablePair;
10 import ch.geomo.util.collection.pair.Pair;
11 import ch.geomo.util.geom.GeomUtil;
12 import com.vividsolutions.jts.geom.Coordinate;
13 import com.vividsolutions.jts.geom.LineString;
14 import org.jetbrains.annotations.NotNull;
15
16 import java.util.Observable;
17 import java.util.stream.Stream;
18
19 /**
20 * Represents an edge which is octilinear and has one or two vertices.
21 */
22 public class OctilinearEdge extends Edge {
23
24     private final MutablePair<Node> vertices;
25
26     private LineString lineString;
27
28     public OctilinearEdge(@NotNull Edge edge) {
29         super(edge.getNodeA(), edge.getNodeB());

```

```

30     setName(edge.getName());
31     addRoutes(edge.getRoutes());
32     vertices = new MutablePair<>();
33     updateOctilinearEdge();
34 }
35
36 private void updateOctilinearEdge() {
37     if (!vertices.hasNonNullValues()) {
38         lineString = null;
39     }
40     Coordinate[] coordinates = getNodeStream()
41         .map(Node::getCoordinate)
42         .toArray(Coordinate[]::new);
43     lineString = GeomUtil.createLineString(coordinates);
44     updateEdge();
45 }
46
47 /**
48  * @return an ordered {@link Stream} set {@link Node}s (node A, vertices, node B)
49  */
50 @NotNull
51 public Stream<Node> getNodeStream() {
52     return Stream.concat(Stream.of(getNodeA()), Stream.concat(vertices.nonNullStream(), Stream.of(getNodeA())));
53 }
54
55 /**
56  * @return a {@link Pair} set vertex {@link Node}s set this edge, may contain null values
57  */
58 @NotNull
59 public Pair<Node> getVertices() {
60     return vertices;
61 }
62
63 public void setVertices(@NotNull Pair<Node> vertices) {
64     this.vertices.replaceValues(vertices);
65     if (vertices.first() == null && vertices.second() != null) {
66         this.vertices.swapValues();
67     }
68     updateOctilinearEdge();
69 }
70
71 @NotNull
72 @Override
73 public LineString getLineString() {
74     if (lineString == null) {
75         return super.getLineString();
76     }
77     return lineString;
78 }
79
80 @Override
81 public void update(Observable o, Object arg) {
82     updateOctilinearEdge();
83     super.update(o, arg);
84 }
85
86 @Override
87 public String toString() {
88     return "OctilinearEdge: {lineString= " + lineString + ", vertices= " + vertices + "}";
89 }
90
91 }

```

## OctilinearEdgeBuilder.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.tramaps.graph.layout;
6
7  import ch.geomo.tramaps.graph.Edge;
8  import ch.geomo.tramaps.graph.Node;
9  import ch.geomo.tramaps.graph.direction.OctilinearDirection;
10 import ch.geomo.tramaps.map.signature.BendNodeSignature;
11 import ch.geomo.util.collection.GCollectors;
12 import ch.geomo.util.collection.list.EnhancedList;
13 import ch.geomo.util.collection.pair.MutablePair;
14 import ch.geomo.util.logging.Loggers;

```

## D.2. Verdrängungsalgorithmus

```
15 import org.jetbrains.annotations.NotNull;
16
17 import static ch.geomo.tramaps.graph.direction.OctilinearDirection.*;
18
19 public class OctilinearEdgeBuilder {
20
21     private final MutablePair<Node> vertices;
22
23     private Edge originalEdge;
24
25     public OctilinearEdgeBuilder() {
26         vertices = new MutablePair<>();
27     }
28
29     @NotNull
30     public OctilinearEdgeBuilder setOriginalEdge(@NotNull Edge edge) {
31         originalEdge = edge;
32         return this;
33     }
34
35     @NotNull
36     public OctilinearEdge build() {
37         createNodes();
38         OctilinearEdge octilinearEdge = new OctilinearEdge(originalEdge);
39         octilinearEdge.setVertices(vertices);
40         return octilinearEdge;
41     }
42
43     private Node getNodeA() {
44         return originalEdge.getNodeA();
45     }
46
47     private Node getNodeB() {
48         return originalEdge.getNodeB();
49     }
50
51     private void createNodeC(double x, double y) {
52         vertices.set(0, new Node("", x, y, BendNodeSignature::new));
53     }
54
55     private void createNodeD(double x, double y) {
56         vertices.set(1, new Node("", x, y, BendNodeSignature::new));
57     }
58
59     private boolean isReversedOrder() {
60         return !getNodeA().isSouthOf(getNodeB());
61     }
62
63     /**
64      * Creates the vertices based on the original edge and checks the result
65      */
66     private void createNodes() {
67
68         Loggers.info(this, "Create vertices for octilinear edge.");
69
70         vertices.clear();
71
72         Node a = isReversedOrder() ? getNodeB() : getNodeA();
73         Node b = isReversedOrder() ? getNodeA() : getNodeB();
74
75         EnhancedList<OctilinearDirection> adjacentEdgeDirectionsA = a.getAdjacentEdges().stream()
76             .filter(edge -> originalEdge.isNotEquals(edge))
77             .map(edge -> edge.getOriginalDirection(a).toOctilinear())
78             .collect(GCollectors.toList());
79
80         EnhancedList<OctilinearDirection> adjacentEdgeDirectionsB = b.getAdjacentEdges().stream()
81             .filter(edge -> originalEdge.isNotEquals(edge))
82             .map(edge -> edge.getOriginalDirection(b).toOctilinear())
83             .collect(GCollectors.toList());
84
85         double dx = Math.abs(b.getX() - a.getX());
86         double dy = Math.abs(b.getY() - a.getY());
87
88         OctilinearDirection originalEdgeDirection = originalEdge.getOriginalDirection(a).toOctilinear();
89
90         double diff = Math.abs(dx - dy);
91
92         if (originalEdgeDirection == OctilinearDirection.NORTH_EAST) {
93             if (dx > dy) {
94                 if (!adjacentEdgeDirectionsA.contains(EAST) && !adjacentEdgeDirectionsB.contains(WEST)) { // 7a
```

## D.2. Verdrängungsalgorithmus

```
95         double x1 = a.getX() + diff / 2;
96         double y1 = a.getY();
97         double x2 = b.getX() - diff / 2;
98         double y2 = b.getY();
99         createNodeC(x1, y1);
100        createNodeD(x2, y2);
101    }
102    else if (!adjacentEdgeDirectionsA.contains(EAST)) { // 6a
103        double x = a.getX() + diff;
104        double y = a.getY();
105        createNodeC(x, y);
106    }
107    else if (!adjacentEdgeDirectionsB.contains(WEST)) { // 5a
108        double x = b.getX() - diff;
109        double y = b.getY();
110        createNodeC(x, y);
111    }
112    else { // 8a
113        double x1 = a.getX() + dy / 2;
114        double y1 = a.getY() + dy / 2;
115        double x2 = b.getX() - dy / 2;
116        double y2 = b.getY() - dy / 2;
117        createNodeC(x1, y1);
118        createNodeD(x2, y2);
119    }
120 }
121 else { // dy > dx
122     if (!adjacentEdgeDirectionsA.contains(NORTH) && !adjacentEdgeDirectionsB.contains(SOUTH)) { // 1a
123         double x1 = a.getX();
124         double y1 = a.getY() + diff / 2;
125         double x2 = b.getX();
126         double y2 = b.getY() - diff / 2;
127         createNodeC(x1, y1);
128         createNodeD(x2, y2);
129     }
130     else if (!adjacentEdgeDirectionsA.contains(NORTH)) { // 3a
131         double x = a.getX();
132         double y = a.getY() + diff;
133         createNodeC(x, y);
134     }
135     else if (!adjacentEdgeDirectionsB.contains(SOUTH)) { // 2a
136         double x = b.getX();
137         double y = b.getY() - diff;
138         createNodeC(x, y);
139     }
140     else { // 4a
141         double x1 = a.getX() + dx / 2;
142         double y1 = a.getY() + dx / 2;
143         double x2 = b.getX() - dx / 2;
144         double y2 = b.getY() - dx / 2;
145         createNodeC(x1, y1);
146         createNodeD(x2, y2);
147     }
148 }
149 }
150 else if (originalEdgeDirection == OctilinearDirection.NORTH_WEST) {
151     if (dx > dy) {
152         if (!adjacentEdgeDirectionsA.contains(EAST) && !adjacentEdgeDirectionsB.contains(WEST)) { // 7b
153             double x1 = a.getX() - diff / 2;
154             double y1 = a.getY();
155             double x2 = b.getX() + diff / 2;
156             double y2 = b.getY();
157             createNodeC(x1, y1);
158             createNodeD(x2, y2);
159         }
160         else if (!adjacentEdgeDirectionsA.contains(WEST)) { // 6b
161             double x = a.getX() - diff;
162             double y = a.getY();
163             createNodeC(x, y);
164         }
165         else if (!adjacentEdgeDirectionsB.contains(EAST)) { // 5b
166             double x = b.getX() + diff;
167             double y = b.getY();
168             createNodeC(x, y);
169         }
170         else { // 8b
171             double x1 = a.getX() - dy / 2;
172             double y1 = a.getY() + dy / 2;
173             double x2 = b.getX() + dy / 2;
174             double y2 = b.getY() - dy / 2;
```

```

175         createNodeC(x1, y1);
176         createNodeD(x2, y2);
177     }
178 }
179 else { // dy > dx
180     if (!adjacentEdgeDirectionsA.contains(NORTH) && !adjacentEdgeDirectionsB.contains(SOUTH)) { // 1b
181         double x1 = a.getX();
182         double y1 = a.getY() + diff / 2;
183         double x2 = b.getX();
184         double y2 = b.getY() - diff / 2;
185         createNodeC(x1, y1);
186         createNodeD(x2, y2);
187     }
188     else if (!adjacentEdgeDirectionsA.contains(NORTH)) { // 3b
189         double x = a.getX();
190         double y = a.getY() + diff;
191         createNodeC(x, y);
192     }
193     else if (!adjacentEdgeDirectionsB.contains(SOUTH)) { // 2b
194         double x = b.getX();
195         double y = b.getY() - diff;
196         createNodeC(x, y);
197     }
198     else { // 4b
199         double x1 = a.getX() - dx / 2;
200         double y1 = a.getY() + dx / 2;
201         double x2 = b.getX() + dx / 2;
202         double y2 = b.getY() - dx / 2;
203         createNodeC(x1, y1);
204         createNodeD(x2, y2);
205     }
206 }
207 }
208
209 if (vertices.nonNullStream().count() == 1) {
210     vertices.get(0).setName(getNodeA().getName() + "-" + getNodeB().getName());
211 }
212 else if (vertices.nonNullStream().count() == 2) {
213     if (isReversedOrder()) {
214         vertices.swapValues();
215     }
216     vertices.get(0).setName(getNodeA().getName() + "+-" + getNodeB().getName());
217     vertices.get(1).setName(getNodeA().getName() + "-+" + getNodeB().getName());
218 }
219
220 }
221
222 }

```

## D.3 Skalierung

### D.3.1 Package `ch.geomo.tramaps.map.displacement.scale`

#### ScaleHandler.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.tramaps.map.displacement.scale;
6
7  import ch.geomo.tramaps.conflict.Conflict;
8  import ch.geomo.tramaps.graph.direction.OctilinearDirection;
9  import ch.geomo.tramaps.map.MetroMap;
10 import ch.geomo.tramaps.map.displacement.LineSpaceHandler;
11 import ch.geomo.util.collection.list.EnhancedList;
12 import ch.geomo.util.geom.GeomUtil;
13 import ch.geomo.util.geom.PolygonUtil;
14 import ch.geomo.util.logging.Loggers;
15 import com.vividsolutions.jts.geom.Envelope;
16 import com.vividsolutions.jts.geom.Geometry;
17 import com.vividsolutions.jts.geom.GeometryCollection;
18 import com.vividsolutions.jts.geom.util.AffineTransformation;
19 import org.jetbrains.annotations.NotNull;
20
21 import java.util.Arrays;
22

```

```

23 /**
24  * This {@link LineSpaceHandler} implementation makes space by scaling the underlying graph.
25  */
26 public class ScaleHandler implements LineSpaceHandler {
27
28     /**
29      * Max iteration until algorithm will be terminated when not found a non-conflict solution.
30      */
31     private static final int MAX_ITERATIONS = 100;
32
33     private final MetroMap map;
34
35     public ScaleHandler(@NotNull MetroMap map) {
36         this.map = map;
37     }
38
39     /**
40      * Makes space for line and station signatures by scaling nodes recursively.
41      */
42     private double evaluateScaleFactor(@NotNull EnhancedList<Conflict> conflicts) {
43
44         double maxMoveX = conflicts.stream()
45             .map(conflict -> {
46                 Envelope bbox = conflict.getElementBoundingBox();
47                 double alongX = conflict.getDisplaceDistanceAlongX();
48                 if (bbox.getWidth() != 0) {
49                     return (bbox.getWidth() + alongX) / bbox.getWidth();
50                 }
51                 return 1d;
52             })
53             .max(Double::compare)
54             .orElse(1d);
55         double maxMoveY = conflicts.stream()
56             .map(conflict -> {
57                 Envelope bbox = conflict.getElementBoundingBox();
58                 double alongY = conflict.getDisplaceDistanceAlongY();
59                 if (bbox.getHeight() != 0) {
60                     return (bbox.getHeight() + alongY) / bbox.getHeight();
61                 }
62                 return 1d;
63             })
64             .max(Double::compare)
65             .orElse(1d);
66
67         return Math.max(GeomUtil.makePrecise(Math.max(maxMoveX, maxMoveY)), 1.0001);
68     }
69 }
70
71 /**
72  * Scales the map with given scale factor.
73  */
74 private void scale(double scaleFactor) {
75     map.getNodes().forEach(node -> node.updatePosition(node.getX()*scaleFactor, node.getY()*scaleFactor));
76 }
77
78 private void makeSpace(int lastIteration) {
79
80     int currentIteration = lastIteration + 1;
81
82     EnhancedList<Conflict> conflicts = map.evaluateConflicts(true);
83
84     Loggers.separator(this);
85     Loggers.info(this, "Iteration: {0}", currentIteration);
86
87     if (!conflicts.isEmpty()) {
88
89         Loggers.warning(this, "Conflicts found: {0}", conflicts.size());
90
91         double scaleFactor = evaluateScaleFactor(conflicts);
92         Loggers.info(this, "Use scale factor: " + scaleFactor);
93         scale(scaleFactor);
94
95         if (currentIteration < MAX_ITERATIONS) {
96             makeSpace(currentIteration);
97         }
98         else {
99             Loggers.separator(this);
100            Loggers.warning(this, "Max number set iteration reached. Stop algorithm.");
101            Loggers.info(this, getBoundingBoxString());
102            Loggers.separator(this);

```

```

103     }
104     }
105     }
106     else {
107         Loggers.separator(this);
108         Loggers.info(this, "No (more) conflicts found.");
109         Loggers.info(this, getBoundingBoxString());
110         Loggers.separator(this);
111     }
112 }
113 }
114
115 /**
116  * @return the bounding box size as a {@link String}
117  */
118 @NotNull
119 private String getBoundingBoxString() {
120     Envelope mapBoundingBox = map.getBoundingBox();
121     return "Size: " + (int) Math.ceil(mapBoundingBox.getWidth()) + "x" + (int) Math.ceil(mapBoundingBox.getHeight());
122 }
123
124 /**
125  * Starts algorithm and makes space for line and station signatures by scaling the node positions.
126  */
127 @Override
128 public void makeSpace() {
129     makeSpace(0);
130 }
131
132 }

```

## D.4 Weitere Core-Klassen

### D.4.1 Package ch.geomo.tramaps

#### MainApp.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.tramaps;
6
7  import ch.geomo.tramaps.example.*;
8  import ch.geomo.tramaps.map.MetroMap;
9  import ch.geomo.tramaps.map.MetroMapDrawer;
10 import ch.geomo.tramaps.map.SimpleGraphDrawer;
11 import ch.geomo.tramaps.map.displacement.LineSpaceHandler;
12 import ch.geomo.tramaps.map.displacement.alg.DisplaceLineSpaceHandler;
13 import ch.geomo.tramaps.map.displacement.scale.ScaleHandler;
14 import com.vividsolutions.jts.geom.Envelope;
15 import javafx.application.Application;
16 import javafx.scene.Group;
17 import javafx.scene.Scene;
18 import javafx.scene.canvas.Canvas;
19 import javafx.scene.canvas.GraphicsContext;
20 import javafx.scene.control.ScrollPane;
21 import javafx.stage.Stage;
22 import org.jetbrains.annotations.NotNull;
23
24 import java.io.IOException;
25 import java.util.function.Supplier;
26
27 public class MainApp extends Application {
28
29     private static final int DRAWING_MARGIN = 50;
30
31     private static final double MAX_HEIGHT = 600;
32     private static final double MAX_WIDTH = 1400;
33
34     private MetroMap map;
35     private Stage stage;
36
37     private void makeSpace(@NotNull Supplier<LineSpaceHandler> makeSpaceHandlerSupplier) {
38         LineSpaceHandler handler = makeSpaceHandlerSupplier.get();
39         handler.makeSpace();
40     }

```



```

41
42 @Override
43 public void start(@NotNull Stage primaryStage) throws Exception {
44
45     stage = primaryStage;
46     stage.setTitle("Tramaps GUI");
47
48     //map = new MetroMapChapterFive();
49     map = new MetroMapExampleGraph();
50     //map = new MetroMapZuerich();
51
52     //makeSpace() -> new ScaleHandler(map));
53     makeSpace() -> new DisplaceLineSpaceHandler(map));
54
55     drawMetroMap();
56
57 }
58
59 private void drawMetroMap() {
60
61     Envelope bbox = map.getBoundingBox();
62
63     double scaleFactor = MAX_HEIGHT / bbox.getHeight();
64
65     double scaledHeight = bbox.getHeight() * scaleFactor + DRAWING_MARGIN * 2;
66     double scaledWidth = bbox.getWidth() * scaleFactor + DRAWING_MARGIN * 2;
67
68     Canvas canvas = new Canvas(scaledWidth, scaledHeight);
69     GraphicsContext context = canvas.getGraphicsContext2D();
70
71     MetroMapDrawer drawer = new MetroMapDrawer(map, DRAWING_MARGIN, scaleFactor, false, false);
72     drawer.draw(context, bbox);
73
74     // workaround: scaling is done when drawing otherwise an exception may happen:
75     // -> Requested texture dimension exceeds maximum texture size
76     // canvas.setScaleX(scaleFactor);
77     // canvas.setScaleY(scaleFactor);
78
79     showMetroMap(canvas, scaledHeight, scaledWidth);
80
81 }
82
83 private void showMetroMap(@NotNull Canvas canvas, double height, double width) {
84
85     Group group = new Group();
86     group.getChildren().add(canvas);
87
88     ScrollPane scrollPane = new ScrollPane(group);
89     scrollPane.setStyle("-fx-focus-color: transparent;");
90     Scene scene = new Scene(scrollPane, width + 5, height + 5);
91
92     stage.setScene(scene);
93     stage.setWidth(Math.min(MAX_WIDTH, width) + 5);
94     stage.setResizable(false);
95     stage.show();
96
97 }
98
99 public static void main(String... args) throws IOException {
100     MainApp.launch(args);
101 }
102
103 }

```

## D.4.2 Package `ch.geomo.tramaps.map.*`

### MetroMap.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.tramaps.map;
6
7  import ch.geomo.tramaps.conflict.Conflict;
8  import ch.geomo.tramaps.conflict.ConflictFinder;
9  import ch.geomo.tramaps.graph.Edge;
10 import ch.geomo.tramaps.graph.Graph;

```

```

11 import ch.geomo.tramaps.graph.Node;
12 import ch.geomo.tramaps.map.signature.BendNodeSignature;
13 import ch.geomo.util.collection.list.EnhancedList;
14 import org.jetbrains.annotations.NotNull;
15
16 public class MetroMap extends Graph {
17
18     private final ConflictFinder conflictFinder;
19
20     private final double routeMargin;
21     private final double edgeMargin;
22     private final double nodeMargin;
23
24     private int bendCount = 0;
25     private int crossingCount = 0;
26     private int junctionCount = 0;
27
28     public MetroMap(double routeMargin, double edgeMargin, double nodeMargin) {
29         super();
30         this.routeMargin = routeMargin;
31         this.edgeMargin = edgeMargin;
32         this.nodeMargin = nodeMargin;
33         conflictFinder = new ConflictFinder(this, routeMargin, edgeMargin, nodeMargin);
34     }
35
36     public double getRouteMargin() {
37         return routeMargin;
38     }
39
40     public double getEdgeMargin() {
41         return edgeMargin;
42     }
43
44     public double getNodeMargin() {
45         return nodeMargin;
46     }
47
48     /**
49      * @return a sorted {@link EnhancedList} of conflicts
50      */
51     @NotNull
52     public EnhancedList<Conflict> evaluateConflicts(boolean biggestConflictFirst) {
53         return conflictFinder.getConflicts(0.25, true)
54             .reverseIf() -> biggestConflictFirst);
55     }
56
57     /**
58      * @return a sorted {@link EnhancedList} of conflicts
59      */
60     @NotNull
61     public EnhancedList<Conflict> evaluateConflicts(boolean biggestConflictFirst, double correctionFactor, boolean majorMisalignmentOnly) {
62         return conflictFinder.getConflicts(correctionFactor, majorMisalignmentOnly)
63             .reverseIf() -> biggestConflictFirst);
64     }
65
66     public long countNonOctilinearEdges() {
67         return getEdges().stream()
68             .filter(Edge::isNotOctilinear)
69             .count();
70     }
71
72     @NotNull
73     public Node createCrossingNode(double x, double y) {
74         Node node = new Node("C" + (++crossingCount), x, y, BendNodeSignature::new);
75         addNodes(node);
76         return node;
77     }
78
79     @NotNull
80     public Node createJunctionNode(double x, double y) {
81         Node node = new Node("J" + (++junctionCount), x, y, BendNodeSignature::new);
82         addNodes(node);
83         return node;
84     }
85
86     @NotNull
87     public Node createBendNode(double x, double y) {
88         Node node = new Node("B" + (++bendCount), x, y, BendNodeSignature::new);
89         addNodes(node);
90         return node;

```

```

91     }
92
93 }

```

## MetroMapDrawer.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.tramaps.map;
6
7  import ch.geomo.tramaps.conflict.BufferConflict;
8  import ch.geomo.tramaps.graph.Edge;
9  import ch.geomo.tramaps.map.signature.BendNodeSignature;
10 import com.vividsolutions.jts.geom.Envelope;
11 import javafx.scene.canvas.GraphicsContext;
12 import javafx.scene.paint.Color;
13 import javafx.scene.shape.StrokeLineCap;
14 import javafx.scene.text.Font;
15 import org.jetbrains.annotations.NotNull;
16
17 public class MetroMapDrawer {
18
19     private final MetroMap map;
20     private final double margin;
21     private final double scaleFactor;
22
23     private final boolean showNodeName;
24     private final boolean showNodePosition;
25
26     public MetroMapDrawer(@NotNull MetroMap map, double margin, double scaleFactor, boolean showNodeName, boolean showNodePosition) {
27         this.map = map;
28         this.margin = margin;
29         this.scaleFactor = scaleFactor;
30         this.showNodeName = showNodeName;
31         this.showNodePosition = showNodePosition;
32     }
33
34     private void drawEdge(@NotNull Edge edge, @NotNull GraphicsContext context) {
35         context.strokeLine(edge.getNodeA().getX() * scaleFactor, -edge.getNodeA().getY() * scaleFactor, edge.getNodeB().getX() * scaleFactor,
36             -edge.getNodeB().getY() * scaleFactor);
37     }
38
39     public void draw(@NotNull GraphicsContext context, @NotNull Envelope bbox) {
40
41         // start drawing at the top left
42         context.translate(-bbox.getMinX() * scaleFactor + margin, bbox.getMaxY() * scaleFactor + margin);
43
44         int max = (int) Math.ceil(Math.max(bbox.getMaxX(), bbox.getMaxY() * -1) + 1000);
45         for (int i = -max, j = 0; i < max; i = i + 50, j = j + 50) {
46             context.setStroke(Color.LIGHTGRAY);
47             context.setLineWidth(j == 0 || j % 500 == 0 ? 5 * scaleFactor : (j % 250 == 0) ? 3 * scaleFactor : 1 * scaleFactor);
48             context.strokeLine(i * scaleFactor, max * scaleFactor, i * scaleFactor, -max * 2 * scaleFactor);
49             context.strokeLine(-max * scaleFactor, -i * scaleFactor, max * 2 * scaleFactor, -i * scaleFactor);
50         }
51
52         map.getEdges().stream()
53             .filter(Edge::hasRoutes)
54             .forEach(edge -> {
55                 double width = edge.calculateEdgeWidth(map.getRouteMargin()) * scaleFactor;
56                 context.setLineWidth(width);
57                 context.setStroke(Color.rgb(139, 187, 206, 0.5d));
58                 if (edge.isNotOctilinear()) {
59                     context.setStroke(Color.rgb(227, 74, 93, 0.5d));
60                 }
61                 context.setLineCap(StrokeLineCap.BUTT);
62                 drawEdge(edge, context);
63             });
64
65         map.getNodes().stream()
66             .filter(node -> !(node.getNodeSignature() instanceof BendNodeSignature))
67             .forEach(node -> {
68                 Envelope station = node.getNodeSignature().getGeometry().getEnvelopeInternal();
69                 context.setFill(Color.BLACK);
70                 context.fillRoundRect((station.getMinX() - 5) * scaleFactor, (-station.getMaxY() - 5) * scaleFactor, (station.getWidth() + 10) *
71                     scaleFactor, (station.getHeight() + 10) * scaleFactor, 25 * scaleFactor, 25 * scaleFactor);
72                 context.setFill(Color.WHITE);
73                 context.fillRoundRect(station.getMinX() * scaleFactor, -station.getMaxY() * scaleFactor, station.getWidth() * scaleFactor,
74                     station.getHeight() * scaleFactor, 25 * scaleFactor, 25 * scaleFactor);

```

```

71     });
72     map.getEdges().forEach(edge -> {
73         context.setLineWidth(2 * scaleFactor);
74         context.setStroke(Color.BLACK);
75         drawEdge(edge, context);
76     });
77     context.translate(-5 * scaleFactor, -5 * scaleFactor);
78     map.getNodes().forEach(node -> {
79         context.setFill(Color.rgb(0, 145, 255));
80         context.fillOval(node.getX() * scaleFactor, -node.getY() * scaleFactor, 10 * scaleFactor, 10 * scaleFactor);
81     });
82
83     context.translate(5 * scaleFactor, 5 * scaleFactor);
84     // map.evaluateConflicts(true)
85     //     .filter(conflict -> conflict instanceof BufferConflict)
86     //     .forEach(conflict -> {
87         context.setFill(Color.rgb(240, 88, 88, 0.4));
88         // Envelope bbox2 = ((BufferConflict) conflict).getConflictPolygon().getEnvelopeInternal();
89         // context.fillRect(bbox2.getMinX() * scaleFactor, -bbox2.getMaxY() * scaleFactor, bbox2.getWidth() * scaleFactor,
90         //     bbox2.getHeight() * scaleFactor);
91     });
92
93     if (showNodeName) {
94         Font font = Font.font(7);
95         map.getNodes().forEach(node -> {
96             Envelope station = node.getNodeSignature().getGeometry().getEnvelopeInternal();
97             context.setStroke(Color.BLACK);
98             context.setFont(font);
99             context.setLineWidth(0.5);
100            if (showNodePosition) {
101                context.strokeText(node.getName() + "(" + Math.round(node.getX()) + "/" + Math.round(node.getY()) + ")", station.getMinX() *
102                    scaleFactor - 50 * scaleFactor, -station.getMaxY() * scaleFactor - 20 * scaleFactor);
103            }
104            else {
105                if (node.getNodeSignature() instanceof BendNodeSignature) {
106                    // context.strokeText(node.getName(), node.getPoint().getX() * scaleFactor - 50 * scaleFactor, -node.getPoint().getY() *
107                    //     scaleFactor - 20 * scaleFactor);
108                }
109                else {
110                    context.strokeText(node.getName(), station.getMinX() * scaleFactor - 20 * scaleFactor, -station.getMaxY() * scaleFactor - 20
111                    * scaleFactor);
112                }
113            }
114        });
115    }

```

## MetroMapEdgeBuilder.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.tramaps.map;
6
7  import ch.geomo.tramaps.graph.Node;
8  import ch.geomo.tramaps.graph.Route;
9  import ch.geomo.tramaps.map.signature.RectangleStationSignature;
10 import org.jetbrains.annotations.NotNull;
11
12 import java.util.LinkedList;
13
14 public class MetroMapEdgeBuilder {
15
16     private final MetroMap map;
17     private final LinkedList<Node> nodes;
18
19     private Route[] routes = new Route[1];
20
21     private boolean built = false;
22
23     public MetroMapEdgeBuilder(@NotNull MetroMap map) {
24         this.map = map;
25         nodes = new LinkedList<>();
26     }
27

```

```

28     @NotNull
29     public MetroMapEdgeBuilder routes(@NotNull Route... routes) {
30         this.routes = routes;
31         return this;
32     }
33
34     @NotNull
35     public MetroMapEdgeBuilder station(@NotNull Node node) {
36         nodes.add(node);
37         return this;
38     }
39
40     @NotNull
41     public MetroMapEdgeBuilder junction(@NotNull Node node) {
42         nodes.add(node);
43         return this;
44     }
45
46     @NotNull
47     public MetroMapEdgeBuilder crossing(@NotNull Node node) {
48         nodes.add(node);
49         return this;
50     }
51
52     @NotNull
53     public MetroMapEdgeBuilder station(double x, double y, @NotNull String name) {
54         nodes.add(map.createNode(x, y, name, RectangleStationSignature::new));
55         return this;
56     }
57
58     @NotNull
59     public MetroMapEdgeBuilder bend(double x, double y) {
60         nodes.add(map.createBendNode(x, y));
61         return this;
62     }
63
64     @NotNull
65     public LinkedList<Node> create() {
66         return build();
67     }
68
69     @NotNull
70     public LinkedList<Node> build() {
71         if (built) {
72             return nodes;
73         }
74         built = true;
75         for (int i = 0; i < (nodes.size() - 1); i++) {
76             map.createEdge(nodes.get(i), nodes.get(i + 1), routes);
77         }
78         return nodes;
79     }
80
81 }

```

## NodeSignature.java

```

1  /*
2   * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3   */
4
5  package ch.geomo.tramaps.map.signature;
6
7  import com.vividsolutions.jts.geom.Geometry;
8  import com.vividsolutions.jts.geom.Polygon;
9  import org.jetbrains.annotations.NotNull;
10
11  import java.util.Observable;
12  import java.util.Observer;
13
14  /**
15   * Represents a signature set a node.
16   */
17  public interface NodeSignature extends Observer {
18
19      /**
20       * @return the convex hull set the signature's geometry
21       * @see #getGeometry()
22       */

```

```

23     @NotNull
24     Geometry getConvexHull();
25
26     /**
27      * @return the signature's geometry
28      */
29     @NotNull
30     Polygon getGeometry();
31
32     /**
33      * Updates the signature.
34      */
35     void updateSignature();
36
37     /**
38      * Implementing class set {@link NodeSignature} must extend {@link Observable}. Doing so, this
39      * method must not be overridden.
40      */
41     @SuppressWarnings("unused")
42     void addObserver(Observer o);
43
44 }

```

### AbstractNodeSignature.java

```

1  /*
2   * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3   */
4
5  package ch.geomo.tramaps.map.signature;
6
7  import ch.geomo.tramaps.graph.Node;
8  import com.vividsolutions.jts.geom.Geometry;
9  import com.vividsolutions.jts.geom.Polygon;
10 import org.jetbrains.annotations.NotNull;
11
12 import java.util.Observable;
13
14 public abstract class AbstractNodeSignature extends Observable implements NodeSignature {
15
16     protected final Node node;
17
18     protected Polygon signature;
19
20     public AbstractNodeSignature(@NotNull Node node) {
21         this.node = node;
22         node.addObserver(this);
23         updateSignature();
24     }
25
26     @Override
27     public void update(Observable o, Object arg) {
28         updateSignature();
29     }
30
31     @NotNull
32     @Override
33     public Geometry getConvexHull() {
34         return signature.convexHull();
35     }
36
37     @NotNull
38     @Override
39     public Polygon getGeometry() {
40         return signature;
41     }
42
43 }

```

### BendNodeSignature.java

```

1  /*
2   * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3   */
4
5  package ch.geomo.tramaps.map.signature;
6
7  import ch.geomo.tramaps.graph.Node;

```

```

8 import ch.geomo.util.geom.GeomUtil;
9 import org.jetbrains.annotations.NotNull;
10
11 public class BendNodeSignature extends AbstractNodeSignature {
12
13     public BendNodeSignature(@NotNull Node node) {
14         super(node);
15     }
16
17     /**
18      * Updates the signature properties. Recalculates the signature geometry if the node's
19      * x- and y-value where changed.
20      */
21     @Override
22     public void updateSignature() {
23         double width = node.getAdjacentEdges().stream()
24             .filter(edge -> !edge.getOriginalDirection(edge.getNodeA()).isHorizontal())
25             .map(edge -> edge.calculateEdgeWidth(0))
26             .max(Double::compare)
27             .orElse(0d);
28         double height = node.getAdjacentEdges().stream()
29             .filter(edge -> !edge.getOriginalDirection(edge.getNodeA()).isVertical())
30             .map(edge -> edge.calculateEdgeWidth(0))
31             .max(Double::compare)
32             .orElse(0d);
33         signature = GeomUtil.createPolygon(node.getPoint(), Math.max(width, 20), Math.max(height, 20));
34         setChanged();
35         notifyObservers();
36     }
37 }
38 }

```

## RectangleStationSignature.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.tramaps.map.signature;
6
7  import ch.geomo.tramaps.graph.Edge;
8  import ch.geomo.tramaps.graph.Node;
9  import ch.geomo.tramaps.graph.direction.Direction;
10 import ch.geomo.tramaps.graph.direction.OctilinearDirection;
11 import ch.geomo.util.geom.GeomUtil;
12 import org.jetbrains.annotations.NotNull;
13
14 import java.util.function.Predicate;
15 import java.util.stream.Stream;
16
17 /**
18  * A simple implementation set a station signature. The signature's form
19  * is a rectangle.
20  */
21 public class RectangleStationSignature extends AbstractNodeSignature {
22
23     private static final double MIN_SIDE_LENGTH = 20d;
24     private static final double STATION_MARGIN = 5d;
25
26     public RectangleStationSignature(@NotNull Node node) {
27         super(node);
28     }
29
30     /**
31      * Updates the signature properties. Recalculates the signature geometry if the node's
32      * x- and y-value where changed.
33      */
34     @Override
35     public void updateSignature() {
36
37         if (hasOnlyDiagonalDirections()) {
38
39             double ne = getSize(edge -> {
40                 OctilinearDirection direction = edge.getOriginalDirection(node).toOctilinear();
41                 return direction == OctilinearDirection.NORTH_EAST || direction == OctilinearDirection.SOUTH_WEST;
42             });
43             double nw = getSize(edge -> {
44                 OctilinearDirection direction = edge.getOriginalDirection(node).toOctilinear();
45                 return direction == OctilinearDirection.NORTH_WEST || direction == OctilinearDirection.SOUTH_EAST;

```

```

46     });
47
48     // future idea: rotate signature
49     signature = GeomUtil.createPolygon(node.getPoint(), Math.max(nw, MIN_SIDE_LENGTH), Math.max(ne, MIN_SIDE_LENGTH));
50
51     }
52     else {
53
54         double width = getSize(edge -> !edge.getOriginalDirection(node).isHorizontal());
55         double height = getSize(edge -> !edge.getOriginalDirection(node).isVertical());
56
57         if (hasNoVerticalDirections() && hasNoDiagonalDirections()) {
58             width = MIN_SIDE_LENGTH;
59         }
60         else if (hasNoHorizontalDirections() && hasNoDiagonalDirections()) {
61             height = MIN_SIDE_LENGTH;
62         }
63
64         signature = GeomUtil.createPolygon(node.getPoint(), Math.max(width, MIN_SIDE_LENGTH), Math.max(height, MIN_SIDE_LENGTH));
65
66     }
67
68     setChanged();
69     notifyObservers();
70
71 }
72
73 private double getSize(@NotNull Predicate<Edge> predicate) {
74     return node.getAdjacentEdges().stream()
75         .filter(predicate)
76         .map(edge -> edge.calculateEdgeWidth(STATION_MARGIN))
77         .max(Double::compare)
78         .orElse(STATION_MARGIN);
79 }
80
81 @NotNull
82 private Stream<OctilinearDirection> getAdjacentEdgeDirections() {
83     return node.getAdjacentEdges().stream()
84         .map(edge -> edge.getOriginalDirection(node))
85         .map(Direction::toOctilinear);
86 }
87
88 private boolean hasOnlyDiagonalDirections() {
89     return getAdjacentEdgeDirections()
90         .allMatch(Direction::isDiagonal);
91 }
92
93 private boolean hasNoDiagonalDirections() {
94     return getAdjacentEdgeDirections()
95         .noneMatch(Direction::isDiagonal);
96 }
97
98 private boolean hasNoVerticalDirections() {
99     return getAdjacentEdgeDirections()
100         .noneMatch(Direction::isVertical);
101 }
102
103 private boolean hasNoHorizontalDirections() {
104     return getAdjacentEdgeDirections()
105         .noneMatch(Direction::isHorizontal);
106 }
107
108 }

```

## SquareStationSignature.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.tramaps.map.signature;
6
7  import ch.geomo.tramaps.graph.Node;
8  import ch.geomo.util.geom.GeomUtil;
9  import org.jetbrains.annotations.NotNull;
10
11 /**
12  * A simple implementation set a station signature. The signature's form
13  * is a square.

```



```

14  */
15  public class SquareStationSignature extends AbstractNodeSignature {
16
17      private static final double ROUTE_MARGIN = 5d;
18
19      public SquareStationSignature(@NotNull Node node) {
20          super(node);
21      }
22
23      /**
24       * Updates the signature properties. Recalculates the signature geometry if the node's
25       * x- and y-value where changed.
26       */
27      @Override
28      public void updateSignature() {
29          double width = node.getAdjacentEdges().stream()
30              .map(edge -> edge.calculateEdgeWidth(ROUTE_MARGIN))
31              .max(Double::compare)
32              .orElse(ROUTE_MARGIN);
33          signature = GeomUtil.createPolygon(node.getPoint(), width, width);
34          setChanged();
35          notifyObservers();
36      }
37
38  }

```

### D.4.3 Package ch.geomo.tramaps.conflict.buffer

#### EdgeBuffer.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.tramaps.conflict.buffer;
6
7  import ch.geomo.tramaps.graph.Edge;
8  import ch.geomo.tramaps.graph.GraphElement;
9  import ch.geomo.util.geom.GeomUtil;
10 import com.vividsolutions.jts.geom.Polygon;
11 import org.jetbrains.annotations.NotNull;
12
13 import java.util.Objects;
14 import java.util.Observable;
15
16 /**
17  * Represents the buffer of a {@link Edge}.
18  */
19 public class EdgeBuffer implements ElementBuffer {
20
21     private final Edge edge;
22     private final double routeMargin;
23     private final double edgeMargin;
24
25     private Polygon buffer;
26
27     public EdgeBuffer(@NotNull Edge edge, double routeMargin, double edgeMargin) {
28         this.edge = edge;
29         edge.addObserver(this);
30         this.routeMargin = routeMargin;
31         this.edgeMargin = edgeMargin;
32         updateBuffer();
33     }
34
35     /**
36      * Initialize or updates this buffer representation.
37      */
38     private void updateBuffer() {
39         double width = edge.calculateEdgeWidth(routeMargin) + edgeMargin * 2;
40         buffer = GeomUtil.createBuffer(edge.getLineString(), width / 2, true);
41     }
42
43     @NotNull
44     @Override
45     public Polygon getBuffer() {
46         return buffer;
47     }
48

```

```

49     @NotNull
50     @Override
51     public GraphElement getElement() {
52         return edge;
53     }
54
55     /**
56      * Notifies the observers.
57      */
58     @Override
59     public void update(Observable o, Object arg) {
60         updateBuffer();
61     }
62
63     @Override
64     public boolean equals(Object obj) {
65         return obj instanceof EdgeBuffer
66             && Objects.equals(edge, ((EdgeBuffer) obj).edge)
67             && Objects.equals(edgeMargin, ((EdgeBuffer) obj).edgeMargin)
68             && Objects.equals(routeMargin, ((EdgeBuffer) obj).routeMargin);
69     }
70
71     @Override
72     public int hashCode() {
73         return Objects.hash(edge, edgeMargin, routeMargin);
74     }
75
76     @Override
77     public String toString() {
78         return "EdgeBuffer: {" + edge + "}";
79     }
80 }

```

### ElementBuffer.java

```

1  /*
2   * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3   */
4
5  package ch.geomo.tramaps.conflict.buffer;
6
7  import ch.geomo.tramaps.graph.Edge;
8  import ch.geomo.tramaps.graph.GraphElement;
9  import ch.geomo.tramaps.graph.Node;
10 import com.vividolutions.jts.geom.Polygon;
11 import org.jetbrains.annotations.NotNull;
12
13 import java.util.Observer;
14
15 /**
16  * Represents a buffer of a {@link Node} or an {@link Edge} implementing the {@link Observer} pattern.
17  */
18 public interface ElementBuffer extends Observer {
19
20     /**
21      * @return the buffer of the {@link #getElement()}
22      */
23     @NotNull
24     Polygon getBuffer();
25
26     /**
27      * @return the element
28      */
29     @NotNull
30     GraphElement getElement();
31 }
32 }

```

### ElementBufferPair.java

```

1  /*
2   * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3   */
4
5  package ch.geomo.tramaps.conflict.buffer;
6
7  import ch.geomo.util.collection.pair.Pair;
8  import org.jetbrains.annotations.NotNull;

```

```

9
10 import java.util.Objects;
11
12 /**
13  * A specialized {@link Pair} implementation of {@link ElementBuffer}s.
14  */
15 public class ElementBufferPair implements Pair<ElementBuffer> {
16
17     private final ElementBuffer first;
18     private final ElementBuffer second;
19
20     public ElementBufferPair(@NotNull ElementBuffer first, @NotNull ElementBuffer second) {
21         this.first = first;
22         this.second = second;
23     }
24
25     /**
26      * Creates a new instance of {@link ElementBufferPair}. Both values of given
27      * {@link Pair} instance must not be null!
28      * @throws NullPointerException if one value set the {@link Pair} is null
29      */
30     public ElementBufferPair(@NotNull Pair<ElementBuffer> pair) {
31         this(pair.first(), pair.second());
32         Objects.requireNonNull(first);
33         Objects.requireNonNull(second);
34     }
35
36     /**
37      * @see Pair#getFirst()
38      */
39     @Override
40     public ElementBuffer getFirst() {
41         return first;
42     }
43
44     /**
45      * @see Pair#getFirst()
46      */
47     @Override
48     public ElementBuffer getSecond() {
49         return second;
50     }
51
52     /**
53      * @return true if both elements are equals
54      */
55     public boolean hasEqualElements() {
56         return first.equals(second);
57     }
58
59     /**
60      * @return true if the elements are adjacent to each other
61      */
62     public boolean hasAdjacentElements() {
63         return first.getElement().isAdjacent(second.getElement());
64     }
65
66     /**
67      * @return true if both elements are nodes
68      */
69     public boolean isNodePair() {
70         return first instanceof NodeBuffer && second instanceof NodeBuffer;
71     }
72
73     /**
74      * @return true if both elements are edges
75      */
76     public boolean isEdgePair() {
77         return first instanceof EdgeBuffer && second instanceof EdgeBuffer;
78     }
79
80     @Override
81     public int hashCode() {
82         // fix required: remove workaround -> but an ElementBufferPair must be
83         // equal independent to the order set the values
84         return Objects.hash(first.hashCode() + second.hashCode());
85     }
86
87     @Override
88     public boolean equals(Object obj) {

```

```

89     return obj instanceof ElementBufferPair
90         && ((Objects.equals(first, ((ElementBufferPair) obj).getFirst()) && Objects.equals(second, ((ElementBufferPair) obj).getSecond()))
91         || (Objects.equals(second, ((ElementBufferPair) obj).getFirst()) && Objects.equals(first, ((ElementBufferPair) obj).getSecond())));
92 }
93
94 }

```

## NodeBuffer.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.tramaps.conflict.buffer;
6
7  import ch.geomo.tramaps.graph.GraphElement;
8  import ch.geomo.tramaps.graph.Node;
9  import ch.geomo.util.geom.GeomUtil;
10 import ch.geomo.util.logging.Loggers;
11 import com.vividsolutions.jts.geom.Envelope;
12 import com.vividsolutions.jts.geom.Polygon;
13 import org.jetbrains.annotations.NotNull;
14
15 import java.util.Objects;
16 import java.util.Observable;
17
18 /**
19  * Represents the buffer of a {@link Node}.
20  */
21 public class NodeBuffer implements ElementBuffer {
22
23     protected final Node node;
24     protected final double margin;
25
26     protected Polygon buffer;
27
28     public NodeBuffer(@NotNull Node node, double margin) {
29         this.node = node;
30         node.addObserver(this);
31         this.margin = margin;
32         updateBuffer();
33     }
34
35     /**
36     * Initialize or updates this buffer representation.
37     */
38     private void updateBuffer() {
39         Polygon geometry = node.getNodeSignature().getGeometry();
40         buffer = GeomUtil.createBuffer(geometry, margin, true);
41     }
42
43     @NotNull
44     @Override
45     public Polygon getBuffer() {
46         return buffer;
47     }
48
49     @NotNull
50     @Override
51     public GraphElement getElement() {
52         return node;
53     }
54
55     /**
56     * Notifies the observers.
57     */
58     @Override
59     public void update(Observable o, Object arg) {
60         updateBuffer();
61     }
62
63     @Override
64     public boolean equals(Object obj) {
65         return obj instanceof NodeBuffer
66             && Objects.equals(node, ((NodeBuffer) obj).node)
67             && Objects.equals(margin, ((NodeBuffer) obj).margin);
68     }
69
70     @Override

```

```

71     public int hashCode() {
72         return Objects.hash(node, margin);
73     }
74
75     @Override
76     public String toString() {
77         return "NodeBuffer: {" + node + "}";
78     }
79
80 }

```

## D.4.4 Package ch.geomo.tramaps.graph.\*

### Edge.java

```

1  /*
2   * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3   */
4
5  package ch.geomo.tramaps.graph;
6
7  import ch.geomo.tramaps.graph.direction.AnyDirection;
8  import ch.geomo.tramaps.graph.direction.Direction;
9  import ch.geomo.util.collection.GCollection;
10 import ch.geomo.util.collection.pair.Pair;
11 import ch.geomo.util.collection.set.EnhancedSet;
12 import ch.geomo.util.geom.GeomUtil;
13 import ch.geomo.util.geom.point.NodePoint;
14 import ch.geomo.util.logging.Loggers;
15 import com.vividsolutions.jts.geom.Geometry;
16 import com.vividsolutions.jts.geom.LineString;
17 import org.jetbrains.annotations.Contract;
18 import org.jetbrains.annotations.NotNull;
19 import org.jetbrains.annotations.Nullable;
20
21 import java.util.*;
22
23 /**
24  * Represents an edge within a {@link Graph}. Each edge has a name, a position, a start {@link Node}, an end node and
25  * a {@link EnhancedSet} of routes. When creating a new instance of {@link Edge}, the instance will
26  * automatically adds itself as an adjacent edge to the two nodes and observes them.
27  * <p>
28  * Note: When comparing two edges, the position set the nodes won't be considered.
29  */
30 public class Edge extends Observable implements Observer, GraphElement {
31
32     private final Pair<Node> nodePair;
33     private final EnhancedSet<Route> routes;
34     private final Direction originalDirection;
35
36     private String name;
37
38     private LineString lineString;
39     private Direction direction;
40
41     private boolean destroyed = false;
42
43     public Edge(@NotNull Node nodeA, @NotNull Node nodeB, @NotNull Route... routes) {
44
45         nodePair = Pair.of(nodeA, nodeB);
46         this.routes = GCollection.set(routes);
47
48         updateEdge();
49
50         // cache original direction set this edge
51         originalDirection = AnyDirection.fromAngle(rotateAngle());
52
53         nodeA.addAdjacentEdge(this);
54         nodeB.addAdjacentEdge(this);
55     }
56
57
58     /**
59     * Calculates the edge width set this edge using given margin between
60     * the routes.
61     * @return the width set this edge
62     */
63     public double calculateEdgeWidth(double routeMargin) {

```

```

64     double width = getRoutes().stream()
65         .mapToDouble(Route::getLineWidth)
66         .sum();
67     return width + routeMargin * (getRoutes().size() - 2);
68 }
69
70 @NotNull
71 public String getName() {
72     return Optional.ofNullable(name)
73         .orElse(getNodeA() + " <-> " + getNodeB());
74 }
75
76 public void setName(@Nullable String name) {
77     this.name = name;
78 }
79
80 @NotNull
81 public Node getNodeA() {
82     return nodePair.getFirst();
83 }
84
85 @NotNull
86 public Node getNodeB() {
87     return nodePair.getSecond();
88 }
89
90 /**
91  * @return the angle between Y axis and this edge (starting north, clockwise)
92  */
93 private double calculateAngle() {
94     double angle = NodePoint.calculateAngle(getNodeA(), getNodeB());
95     // accepting an imprecision set one degree
96     if (angle % 45 < 1 || angle % 45 > 44) {
97         return Math.round(angle / 45) * 45;
98     }
99     return angle;
100 }
101
102 /**
103  * Updates the {@link LineString} representation and notifies Observers.
104  */
105 protected final void updateEdge() {
106     lineString = GeomUtil.createLineString(getNodeA(), getNodeB());
107     double angle = calculateAngle();
108     direction = AnyDirection.fromAngle(angle);
109     setChanged();
110     notifyObservers();
111 }
112
113 /**
114  * Adds given routes to this edge and notifies Observers. Ignores
115  * duplicated routes.
116  */
117 public void addRoutes(@NotNull Collection<Route> routes) {
118     this.routes.addAll(routes);
119     setChanged();
120     notifyObservers();
121 }
122
123 /**
124  * @return an {@link EnhancedSet} containing all routes
125  */
126 @NotNull
127 public EnhancedSet<Route> getRoutes() {
128     return routes;
129 }
130
131 /**
132  * Gets the adjacent {@link Node} set given {@link Node} on the other side
133  * set this edge. May throws a {@link NoSuchElementException} if given
134  * node is not a start or end node set this edge. To avoid this exception,
135  * test first with {@link #isAdjacent(Node)}.
136  * @return the adjacent {@link Node} set given {@link Node}
137  * @throws NoSuchElementException if given node is not an end node set this edge
138  */
139 @NotNull
140 public Node getOtherNode(@NotNull Node node) {
141     return nodePair.getOtherValue(node);
142 }
143

```

```

144 @Override
145 @Contract("null->false")
146 public boolean isAdjacent(@NotNull Edge edge) {
147     return getNodeA().getAdjacentEdges().contains(edge) || getNodeB().getAdjacentEdges().contains(edge);
148 }
149
150 @Override
151 @Contract("null->false")
152 public boolean isAdjacent(@NotNull Node node) {
153     return nodePair.contains(node);
154 }
155
156 /**
157  * @return the {@link LineString} representation of this edge
158  */
159 @NotNull
160 public LineString getLineString() {
161     return lineString;
162 }
163
164 @NotNull
165 @Override
166 public Geometry getGeometry() {
167     return getLineString();
168 }
169
170 /**
171  * Returns the {@link Direction} set this edge starting at given {@link Node}. Returns
172  * the given {@link Direction} if given {@link Node} is null.
173  */
174 @NotNull
175 private Direction getDirection(@Nullable Node node, @NotNull Direction direction) {
176     if (node == null || getNodeA().equals(node)) {
177         return direction;
178     }
179     else if (getNodeB().equals(node)) {
180         return direction.opposite();
181     }
182     String message = "Node " + node.getName() + " must be equals to an end node set " + getName() + ".";
183     throw new IllegalStateException(message);
184 }
185
186 /**
187  * @return the <b>current</b> direction of this edge starting at given {@link Node}
188  * @throws IllegalArgumentException if given node is neither equal to node A nor node B
189  */
190 @NotNull
191 public Direction getDirection(@NotNull Node node) {
192     return getDirection(node, direction);
193 }
194
195 /**
196  * @return the <b>original</b> direction of this edge starting at given {@link Node}
197  * @throws IllegalArgumentException if given node is neither equal to node A nor node B
198  */
199 @NotNull
200 public Direction getOriginalDirection(@NotNull Node node) {
201     return getDirection(node, originalDirection);
202 }
203
204 /**
205  * @return true if this edge has an octilinear angle
206  */
207 public boolean isOctilinear() {
208     return getDirection(getNodeA(), direction).isOctilinear();
209 }
210
211 /**
212  * @return true if this edge has a <b>non-</b>octilinear angle
213  */
214 public boolean isNotOctilinear() {
215     return !isOctilinear();
216 }
217
218 /**
219  * @return true if this instance is <b>not</b> equals with given {@link Edge}
220  */
221 @Contract(value = "null->true")
222 public boolean isNotEquals(@Nullable Edge edge) {
223     return !equals(edge);

```

```

224     }
225
226     /**
227      * Returns true if this edge was previously destroyed. If destroyed this edge is
228      * disconnected from the start and end node.
229      * @return true if this edge is marked as destroy
230      */
231     public boolean destroyed() {
232         return destroyed;
233     }
234
235     /**
236      * Remove this edge from the list set the adjacent edges in the start and end
237      * node. Marks this edge as destroyed and unsubscribe all observers.
238      */
239     public void destroy() {
240         // remove from adjacent nodes
241         getNodeA().removeAdjacentEdge(this);
242         getNodeB().removeAdjacentEdge(this);
243         destroyed = true;
244         // notify observers a last time
245         setChanged();
246         notifyObservers();
247         // unsubscribe all observers
248         deleteObservers();
249     }
250
251     /**
252      * @return true if this edge has routes
253      */
254     public boolean hasRoutes() {
255         return !routes.isEmpty();
256     }
257
258     /**
259      * @return the edge length based on the related {@link LineString}
260      */
261     public double getLength() {
262         return lineString.getLength();
263     }
264
265     /**
266      * @return true if the angle between the original octilinear direction and the current direction is greater than 27.5 degrees
267      */
268     public boolean hasMajorMisalignment() {
269         return getOriginalDirection(getNodeA()).toOctilinear() != getDirection(getNodeA()).toOctilinear();
270     }
271
272     @Override
273     public void update(Observable o, Object arg) {
274         updateEdge();
275         Loggers.debug(this, toString() + " updated. New direction is " + direction + ".");
276     }
277
278     @Override
279     public boolean equals(Object obj) {
280         return obj instanceof Edge
281             && Objects.equals(name, ((Edge) obj).name)
282             && Objects.equals(nodePair, ((Edge) obj).nodePair)
283             // && CollectionUtil.equals(routes, ((Edge) obj).routes)
284             && destroyed == ((Edge) obj).destroyed;
285     }
286
287     @Override
288     public int hashCode() {
289         // return Objects.hash(name, nodePair, routes, destroyed);
290         return Objects.hash(name, nodePair, destroyed);
291     }
292
293     @Override
294     public String toString() {
295         return "Edge: {" + getName() + "}";
296     }
297
298 }

```

## Graph.java

```
1 /*
```



```

2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.tramaps.graph;
6
7  import ch.geomo.tramaps.map.signature.NodeSignature;
8  import ch.geomo.util.collection.set.EnhancedSet;
9  import ch.geomo.util.collection.set.GSet;
10 import ch.geomo.util.geom.GeomUtil;
11 import com.vividsolutions.jts.geom.Envelope;
12 import com.vividsolutions.jts.geom.Geometry;
13 import org.jetbrains.annotations.NotNull;
14
15 import java.util.function.Function;
16
17 public class Graph {
18
19     private final EnhancedSet<Node> nodes;
20
21     private EnhancedSet<Edge> edgeCache;
22
23     public Graph() {
24         nodes = GSet.emptySet();
25     }
26
27     private void clearCache() {
28         edgeCache = null;
29     }
30
31     private void buildCache() {
32         if (edgeCache == null) {
33             edgeCache = nodes.flatMap(node -> node.getAdjacentEdges().stream());
34         }
35     }
36
37     public void addNodes(@NotNull Node... nodes) {
38         this.nodes.addElements(nodes);
39         clearCache();
40     }
41
42     @NotNull
43     private EnhancedSet<Edge> getEdgeCache() {
44         buildCache();
45         return edgeCache;
46     }
47
48     @NotNull
49     public EnhancedSet<Edge> getEdges() {
50         return GSet.createSet(getEdgeCache());
51     }
52
53     @NotNull
54     public EnhancedSet<Node> getNodes() {
55         return GSet.createSet(nodes);
56     }
57
58     /**
59     * Calculates the bounding box with a collection set all edge and node signature geometries.
60     * @return a bounding box set all edge and node signatures
61     * @see #getEdgeGeometries()
62     * @see #getNodeSignatureGeometries()
63     */
64     @NotNull
65     public Envelope getBoundingBox() {
66         return GeomUtil
67             .createCollection(getEdgeGeometries(), getNodeSignatureGeometries())
68             .getEnvelopeInternal();
69     }
70
71     @NotNull
72     private EnhancedSet<Geometry> getEdgeGeometries() {
73         return GSet.createSet(getEdgeCache().map(Edge::getLineString));
74     }
75
76     @NotNull
77     private EnhancedSet<Geometry> getNodeSignatureGeometries() {
78         return GSet.createSet(nodes.stream()
79             .map(Node::getNodeSignature)
80             .map(NodeSignature::getGeometry));
81     }

```

```

82
83  /**
84   * Reset and flags edge cache for a rebuild. Removes deleted nodes returning true when
85   * invoking {@link Node#destroyed()}.
86   */
87  public void updateGraph() {
88      // remove deleted nodes
89      nodes.removeIf(Node::destroyed);
90      // reset edge cache -> will be created again when accessing next time
91      clearCache();
92  }
93
94  /**
95   * Creates a new {@link Node} and adds the node to this instance.
96   * @return the created node
97   */
98  @NotNull
99  public Node createNode(double x, double y, @NotNull String name, @NotNull Function<Node, NodeSignature> nodeSignatureFactory) {
100      Node node = new Node(name, x, y, nodeSignatureFactory);
101      addNodes(node);
102      return node;
103  }
104
105  /**
106   * Creates a new {@link Edge}.
107   * @return the created edge
108   */
109  @NotNull
110  public Edge createEdge(@NotNull Node nodeA, @NotNull Node nodeB, @NotNull Route... routes) {
111      Edge edge = new Edge(nodeA, nodeB, routes);
112      clearCache();
113      return edge;
114  }
115
116  @NotNull
117  @Override
118  public String toString() {
119      return "Graph[Edges[" + getEdges() + "],Nodes[" + getNodes() + " ]]";
120  }
121
122  }

```

## GraphElement.java

```

1  /*
2   * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3   */
4
5  package ch.geomo.tramaps.graph;
6
7  import com.vividsolutions.jts.geom.Geometry;
8  import com.vividsolutions.jts.geom.Point;
9  import org.jetbrains.annotations.NotNull;
10
11  /**
12   * Represents a {@link Node} or an {@link Edge} set a {@link Graph}.
13   */
14  public interface GraphElement {
15
16      /**
17       * @return true if given {@link GraphElement} is adjacent to this {@link GraphElement}
18       */
19      default boolean isAdjacent(@NotNull GraphElement element) {
20          if (element instanceof Edge) {
21              return isAdjacent((Edge) element);
22          }
23          else if (element instanceof Node) {
24              return isAdjacent((Node) element);
25          }
26          return false;
27      }
28
29      /**
30       * @return true if given {@link Edge} is adjacent to this {@link GraphElement}
31       */
32      boolean isAdjacent(@NotNull Edge edge);
33
34      /**
35       * @return true if given {@link Edge} is adjacent to this {@link GraphElement}

```

```

36     */
37     boolean isAdjacent(@NotNull Node node);
38
39     @NotNull
40     String getName();
41
42     /**
43      * @return the {@link Geometry} representation set this {@link GraphElement}
44      */
45     @NotNull
46     Geometry getGeometry();
47
48     /**
49      * @return the centroid set the {@link Geometry} representation set this {@link GraphElement}
50      */
51     @NotNull
52     default Point getCentroid() {
53         return getGeometry().getCentroid();
54     }
55
56     /**
57      * @return true if this {@link GraphElement} was previously destroyed
58      */
59     boolean destroyed();
60
61     /**
62      * Mark this {@link GraphElement} as destroyed.
63      */
64     void destroy();
65 }
66 }

```

## Node.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.tramaps.graph;
6
7  import ch.geomo.tramaps.map.signature.NodeSignature;
8  import ch.geomo.util.collection.set.EnhancedSet;
9  import ch.geomo.util.collection.set.GSet;
10 import ch.geomo.util.geom.GeomUtil;
11 import ch.geomo.util.geom.point.NodePoint;
12 import com.vividsolutions.jts.geom.Coordinate;
13 import com.vividsolutions.jts.geom.Geometry;
14 import com.vividsolutions.jts.geom.Point;
15 import org.jetbrains.annotations.Contract;
16 import org.jetbrains.annotations.NotNull;
17 import org.jetbrains.annotations.Nullable;
18
19 import java.util.*;
20 import java.util.function.Function;
21 import java.util.function.Predicate;
22
23 /**
24  * Represents a node within a {@link Graph}. Each node has a name, a position and a {@link NodeSignature}.
25  * <p>
26  * Note: When comparing two nodes, the position won't be considered.
27  */
28 public class Node extends Observable implements GraphElement, NodePoint {
29
30     private final EnhancedSet<Edge> adjacentEdges;
31     private final NodeSignature signature;
32
33     private String name;
34
35     private Point point;
36
37     private boolean destroyed = false;
38
39     public Node(@NotNull String name, double x, double y, @NotNull Function<Node, NodeSignature> nodeSignatureFactory) {
40         this(name, GeomUtil.createPoint(x, y), nodeSignatureFactory);
41     }
42
43     private Node(@NotNull String name, @NotNull Point point, @NotNull Function<Node, NodeSignature> nodeSignatureFactory) {
44         this.name = name;
45         this.point = point;

```

```

46     adjacentEdges = GSet.createSet();
47     signature = nodeSignatureFactory.apply(this);
48 }
49
50 @NotNull
51 public String getName() {
52     return name;
53 }
54
55 public void setName(@NotNull String name) {
56     this.name = name;
57 }
58
59 /**
60  * Creates a new adjacent edge between this and the given node. Subscribes immediately the new edge
61  * as an {@link java.util.Observer}.
62  * @return the newly connected node (allowing to chain this method)
63  */
64 public Node createAdjacentEdgeTo(@NotNull Node node, @NotNull Set<Route> routes) {
65     Edge edge = new Edge(this, node);
66     edge.addRoutes(routes);
67     adjacentEdges.add(edge);
68     addObserver(edge);
69     setChanged();
70     notifyObservers();
71     return node;
72 }
73
74 /**
75  * Adds a new adjacent edge but ignores given edge if neither node A nor node B is equals to this instance.
76  * Subscribes the given edge as an {@link Observer} but does not notify the other observers!
77  */
78 /* package-private */ void addAdjacentEdge(@NotNull Edge edge) {
79     if (!equals(edge.getNodeA()) && !equals(edge.getNodeB())) {
80         return;
81     }
82     adjacentEdges.add(edge);
83     addObserver(edge);
84     setChanged();
85     notifyObservers();
86 }
87
88 /**
89  * Removes an adjacent edge. Nodes will be untouched. Unsubscribe the edge as an {@link Observer}.
90  */
91 public void removeAdjacentEdge(@NotNull Edge edge) {
92     if (!isAdjacent(edge)) {
93         return;
94     }
95     adjacentEdges.remove(edge);
96     deleteObserver(edge);
97     setChanged();
98     notifyObservers();
99 }
100
101 /**
102  * @return an {@link EnhancedSet} of all adjacent edges
103  */
104 @NotNull
105 public EnhancedSet<Edge> getAdjacentEdges() {
106     return adjacentEdges;
107 }
108
109 /**
110  * @return the adjacent edge shared with this node and the given node or null if they are not adjacent
111  */
112 @Nullable
113 public Edge getAdjacentEdgeWith(@NotNull Node otherNode) {
114     return adjacentEdges.filter(edge -> edge.isAdjacent(this) && edge.isAdjacent(otherNode)).first().orElse(null);
115 }
116
117 /**
118  * @return an {@link EnhancedSet} of all adjacent edges <b>without the given edge</b>
119  */
120 @NotNull
121 public EnhancedSet<Edge> getAdjacentEdges(@NotNull Edge without) {
122     return adjacentEdges.without(without::equals);
123 }
124
125 /**

```

```

126     * @return an {@link EnhancedSet} of all adjacent edges <b>matching given {@link Predicate}</b>
127     */
128     @NotNull
129     public EnhancedSet<Edge> getAdjacentEdges(@NotNull Predicate<Edge> predicate) {
130         return adjacentEdges.filter(predicate);
131     }
132
133     /**
134     * Returns the node's geometry. Alias for {@link #getPoint()} in order to satisfy {@link GraphElement}.
135     */
136     @NotNull
137     @Override
138     public Geometry getGeometry() {
139         return getPoint();
140     }
141
142     @Override
143     @Contract("null->false")
144     public boolean isAdjacent(@Nullable Edge edge) {
145         return edge != null && getAdjacentEdges().contains(edge);
146     }
147
148     @Override
149     @Contract("null->false")
150     public boolean isAdjacent(@Nullable Node node) {
151         return node != null && getAdjacentEdges().anyMatch(node::isAdjacent);
152     }
153
154     /**
155     * @return the {@link NodeSignature} of this instance
156     */
157     @NotNull
158     public NodeSignature getNodeSignature() {
159         return signature;
160     }
161
162     /**
163     * @return the x-value of this node's position/coordinate
164     */
165     @Override
166     public double getX() {
167         return point.getX();
168     }
169
170     /**
171     * @return the y-value of this node's position/coordinate
172     */
173     @Override
174     public double getY() {
175         return point.getY();
176     }
177
178     /**
179     * Updates the node's position/coordinate and notifies the {@link Observer}s.
180     */
181     public void updatePosition(double x, double y) {
182         point = GeomUtil.createPoint(x, y);
183         setChanged();
184         notifyObservers();
185     }
186
187     /**
188     * @see #updatePosition(double, double)
189     */
190     public void updatePosition(@NotNull Coordinate coordinate) {
191         updatePosition(coordinate.x, coordinate.y);
192     }
193
194     /**
195     * Updates the x-value of the node's position/coordinate and notifies the {@link Observer}s.
196     * @see #updatePosition(double, double)
197     */
198     public void updateX(double x) {
199         updatePosition(x, getY());
200     }
201
202     /**
203     * Updates the y-value of the node's position/coordinate and notifies the {@link Observer}s.
204     * @see #updatePosition(double, double)
205     */

```

```

206 public void updateY(double y) {
207     updatePosition(getX(), y);
208 }
209
210 /**
211  * @return a <b>new</b> instance of the encapsulated {@link Point} representation of this node
212  */
213 @NotNull
214 @Override
215 public Point toPoint() {
216     return GeomUtil.clonePoint(point);
217 }
218
219 /**
220  * @return the <b>same</b> instance of the encapsulated {@link Point} representation of this node
221  * @see #toPoint() if you need a new instance
222  */
223 @NotNull
224 public Point getPoint() {
225     return point;
226 }
227
228 /**
229  * @return a <b>new</b> instance of the encapsulated {@link Point#getCoordinate()} representation of this node
230  */
231 @NotNull
232 @Override
233 public Coordinate toCoordinate() {
234     Coordinate coordinate = GeomUtil.createCoordinate(point.getCoordinate());
235     if (coordinate != null) {
236         return coordinate;
237     }
238     throw new IllegalStateException("Should never reach this point. A node always has a point geometry.");
239 }
240
241 /**
242  * @return the <b>same</b> instance of the encapsulated {@link Point#getCoordinate()} representation of this node
243  * @see #toCoordinate() if you need a new instance
244  */
245 @NotNull
246 public Coordinate getCoordinate() {
247     return point.getCoordinate();
248 }
249
250 @Override
251 public boolean destroyed() {
252     return destroyed;
253 }
254
255 @Override
256 public void destroy() {
257     // remove adjacent nodes
258     getAdjacentEdges().forEach(Edge::destroy);
259     destroyed = true;
260     // notify observers a last time
261     setChanged();
262     notifyObservers();
263     // unsubscribe all observers
264     deleteObservers();
265 }
266
267 /**
268  * @return the degree of this node
269  */
270 public int getNodeDegree() {
271     return adjacentEdges.size();
272 }
273
274 public boolean isSouthOf(@NotNull Coordinate coordinate) {
275     return getY() < coordinate.y;
276 }
277
278 public boolean isSouthOf(@NotNull Node... node) {
279     return Arrays.stream(node)
280         .allMatch(n -> isSouthOf(n.getCoordinate()));
281 }
282
283 public boolean isSouthOf(@NotNull Edge edge) {
284     return isSouthOf(edge.getNodeA(), edge.getNodeB());
285 }

```

```

286
287 public boolean isNorthOf(@NotNull Coordinate coordinate) {
288     return getY() > coordinate.y;
289 }
290
291 public boolean isNorthOf(@NotNull Node... node) {
292     return Arrays.stream(node)
293         .allMatch(n -> isNorthOf(n.getCoordinate()));
294 }
295
296 public boolean isNorthOf(@NotNull Edge edge) {
297     return isNorthOf(edge.getNodeA(), edge.getNodeB());
298 }
299
300 public boolean isEastOf(@NotNull Coordinate coordinate) {
301     return getX() > coordinate.x;
302 }
303
304 public boolean isEastOf(@NotNull Node... node) {
305     return Arrays.stream(node)
306         .allMatch(n -> isEastOf(n.getCoordinate()));
307 }
308
309 public boolean isEastOf(@NotNull Edge edge) {
310     return isEastOf(edge.getNodeA(), edge.getNodeB());
311 }
312
313 public boolean isWestOf(@NotNull Coordinate coordinate) {
314     return getX() < coordinate.x;
315 }
316
317 public boolean isWestOf(@NotNull Node... node) {
318     return Arrays.stream(node)
319         .allMatch(n -> isWestOf(n.getCoordinate()));
320 }
321
322 public boolean isWestOf(@NotNull Edge edge) {
323     return isWestOf(edge.getNodeA(), edge.getNodeB());
324 }
325
326 @Contract("null -> true")
327 public boolean isNotEquals(@Nullable Node node) {
328     return node == null || !equals(node);
329 }
330
331 @Override
332 public boolean equals(Object obj) {
333     // since a node is equals to the same node but at another position, position
334     // is not used to check equality
335     return obj instanceof Node
336         && Objects.equals(name, ((Node) obj).name)
337         && destroyed == ((Node) obj).destroyed;
338 }
339
340 @Override
341 public int hashCode() {
342     // hashCode and equals must be matching: a.equals(b) == (a.hashCode() == b.hashCode())
343     // therefore position of this node is transient and not used to calculate hash code
344     return Objects.hash(name, destroyed);
345 }
346
347 @Override
348 public String toString() {
349     return "Node: {" + name + "(" + getX() + "/" + getY() + ")"}";
350 }
351
352 }

```

## Route.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.tramaps.graph;
6
7  import javafx.scene.paint.Color;
8  import org.jetbrains.annotations.NotNull;
9

```

```

10 import java.util.Objects;
11
12 public class Route {
13
14     private final double lineWidth;
15     private final Color lineColor;
16     private final String name;
17
18     public Route(@NotNull String name, double lineWidth, @NotNull Color lineColor) {
19         this.name = name;
20         this.lineWidth = lineWidth;
21         this.lineColor = lineColor;
22     }
23
24     @NotNull
25     public String getName() {
26         return name;
27     }
28
29     @NotNull
30     public Color getLineColor() {
31         return lineColor;
32     }
33
34     public double getLineWidth() {
35         return lineWidth;
36     }
37
38     @Override
39     public boolean equals(Object obj) {
40         return obj instanceof Route && Objects.equals(name, ((Route) obj).getName())
41             && Objects.equals(lineColor, ((Route) obj).getLineColor())
42             && lineWidth == ((Route) obj).getLineWidth();
43     }
44
45     @Override
46     public int hashCode() {
47         return Objects.hash(name, lineColor, lineWidth);
48     }
49
50     @Override
51     public String toString() {
52         return "Route: {" + name + "}";
53     }
54
55 }

```

## AnyDirection.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.tramaps.graph.direction;
6
7  import org.jetbrains.annotations.NotNull;
8
9  import java.util.Objects;
10
11 public class AnyDirection implements Direction {
12
13     private final double angle;
14     private Direction oppositeDirection;
15
16     protected AnyDirection(double angle) {
17         this.angle = angle % 360;
18     }
19
20     @Override
21     public double getAngle() {
22         return angle;
23     }
24
25     public boolean isVertical() {
26         return angle % 180 == 0;
27     }
28
29     public boolean isHorizontal() {
30         return angle % 90 == 0 && !isVertical();

```



```

31     }
32
33     /**
34     * Returns the closest octilinear direction for this direction. Rounds to the
35     * octilinear angle which is closer to the angle set this instance. Rounds up
36     * if the distance to both possible octilinear angle is exactly half set 45 getNodeDegree.
37     * @return the closest octilinear direction
38     */
39     @NotNull
40     @Override
41     public OctilinearDirection toOctilinear() {
42         Direction direction;
43         double diff = angle % 45;
44         if (diff < 45 / 2) {
45             direction = fromAngle(angle - diff);
46         }
47         else {
48             direction = fromAngle(angle + (45 - diff));
49         }
50         return direction.toOctilinear();
51     }
52
53     /**
54     * @return the opposite direction
55     */
56     @NotNull
57     @Override
58     public Direction opposite() {
59         if (oppositeDirection == null) {
60             // calculate once when accessing first time
61             oppositeDirection = fromAngle(Math.abs(360 - angle));
62         }
63         return oppositeDirection;
64     }
65
66     @Override
67     public boolean equals(Object obj) {
68         return obj instanceof AnyDirection
69             && angle == ((AnyDirection) obj).getAngle();
70     }
71
72     @Override
73     public int hashCode() {
74         return Objects.hash(getAngle());
75     }
76
77     @Override
78     public String toString() {
79         return "AnyDirection: {" + angle + " degree}";
80     }
81
82     /**
83     * Creates a {@link Direction} instance with given angle. Returns a instance set
84     * {@link OctilinearDirection} if the given angle is a multiple set 45 getNodeDegree.
85     * Otherwise an instance set {@link AnyDirection} will be returned.
86     */
87     @NotNull
88     public static Direction fromAngle(double angle) {
89         if (Direction.isOctilinear(angle)) {
90             return OctilinearDirection.fromAngle(angle);
91         }
92         return new AnyDirection(angle);
93     }
94
95 }

```

## Direction.java

```

1  /**
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.tramaps.graph.direction;
6
7  import org.jetbrains.annotations.NotNull;
8
9  public interface Direction {
10
11     /**

```

```

12     * @return the angle set the direction
13     */
14     double getAngle();
15
16     /**
17     * @return true if horizontal to x-axis
18     */
19     boolean isHorizontal();
20
21     /**
22     * @return true if vertical to y-axis
23     */
24     boolean isVertical();
25
26     /**
27     * @return true if neither horizontal nor vertical
28     */
29     default boolean isDiagonal() {
30         return !isHorizontal() && !isVertical();
31     }
32
33     /**
34     * Returns the closest octilinear direction for this direction. Depending on
35     * the implementation the angle will be rounded up or down.
36     * @return the closest octilinear direction
37     */
38     @NotNull
39     OctilinearDirection toOctilinear();
40
41     /**
42     * @return the opposite direction
43     */
44     @NotNull
45     Direction opposite();
46
47     /**
48     * @return true if given {@link Direction} is the opposite direction set this instance
49     */
50     default boolean isOpposite(@NotNull Direction direction) {
51         return opposite().equals(direction);
52     }
53
54     /**
55     * @return true if angle set this instance is octilinear
56     */
57     default boolean isOctilinear() {
58         return isOctilinear(getAngle());
59     }
60
61     /**
62     * @return the angle between given direction and this instance (clockwise)
63     */
64     default double getAngleTo(Direction direction) {
65         return Math.abs((direction.getAngle() - getAngle() + 360) % 360);
66     }
67
68     /**
69     * @return true if given angle is octilinear
70     */
71     static boolean isOctilinear(double angle) {
72         return angle % 45 == 0;
73     }
74
75 }

```

## OctilinearDirection.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.tramaps.graph.direction;
6
7  import ch.geomo.util.math.MoveVector;
8  import org.jetbrains.annotations.NotNull;
9
10 import java.util.Arrays;
11
12 public enum OctilinearDirection implements Direction {

```

```

13
14 NORTH(0d, new MoveVector(0, 1)),
15 NORTH_EAST(45d, new MoveVector(1, 1)),
16 EAST(90d, new MoveVector(1, 0)),
17 SOUTH_EAST(135d, new MoveVector(1, -1)),
18 SOUTH(180d, new MoveVector(0, -1)),
19 SOUTH_WEST(225d, new MoveVector(-1, -1)),
20 WEST(270d, new MoveVector(-1, 0)),
21 NORTH_WEST(315d, new MoveVector(-1, 1));
22
23 private final double angle;
24 private final MoveVector vector;
25
26 OctilinearDirection(double angle, @NotNull MoveVector vector) {
27     this.angle = angle;
28     this.vector = vector;
29 }
30
31 public double getAngle() {
32     return angle;
33 }
34
35 @NotNull
36 @SuppressWarnings("unused")
37 public MoveVector getVector() {
38     return vector;
39 }
40
41 @Override
42 public boolean isHorizontal() {
43     return this == EAST || this == WEST;
44 }
45
46 @Override
47 public boolean isVertical() {
48     return this == NORTH || this == SOUTH;
49 }
50
51 /**
52  * Returns the closest octilinear direction for this direction. Since
53  * this implementation set {@link Direction} is always octilinear, the
54  * current instance will always be returned.
55  * @return current instance
56  */
57 @NotNull
58 @Override
59 public OctilinearDirection toOctilinear() {
60     return this;
61 }
62
63 /**
64  * @return the opposite direction set this direction
65  */
66 @NotNull
67 public OctilinearDirection opposite() {
68     switch (this) {
69         case NORTH_EAST:
70             return SOUTH_WEST;
71         case EAST:
72             return WEST;
73         case SOUTH_EAST:
74             return NORTH_WEST;
75         case SOUTH:
76             return NORTH;
77         case SOUTH_WEST:
78             return NORTH_EAST;
79         case WEST:
80             return EAST;
81         case NORTH_WEST:
82             return SOUTH_EAST;
83         case NORTH:
84             return SOUTH;
85     }
86     throw new IllegalStateException("Should never reach this point.");
87 }
88
89 /**
90  * Finds the octilinear direction for given angle. If angle is not a multiple
91  * of 45 getNodeDegree, an octilinear direction will be evaluated using
92  * {@link AnyDirection#toOctilinear()}.

```

```

93     * @return the octilinear direction for given angle
94     */
95     @NotNull
96     public static OctilinearDirection fromAngle(double angle) {
97         return Arrays.stream(values())
98             .filter(direction -> direction.angle == (angle + 360) % 360)
99             .findFirst()
100            .orElseThrow(IllegalStateException::new);
101    }
102
103 }

```

## D.5 Beispiele

### D.5.1 Package ch.geomo.tramaps.example

#### MetroMapChapterFive.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.tramaps.example;
6
7  import ch.geomo.tramaps.graph.Node;
8  import ch.geomo.tramaps.graph.Route;
9  import ch.geomo.tramaps.map.MetroMap;
10 import ch.geomo.tramaps.map.signature.NodeSignature;
11 import ch.geomo.tramaps.map.signature.SquareStationSignature;
12 import javafx.scene.paint.Color;
13
14 import java.util.function.Function;
15
16 public class MetroMapChapterFive extends MetroMap {
17
18     public MetroMapChapterFive() {
19
20         super(1, 25, 25);
21
22         Route line1 = new Route("U1", 20, Color.BLUE);
23         Route line2 = new Route("U2", 20, Color.RED);
24         Route line3 = new Route("U3", 20, Color.GREEN);
25         Route line4 = new Route("U4", 20, Color.YELLOW);
26
27         Function<Node, NodeSignature> signatureFunction = SquareStationSignature::new;
28
29         Node a = createNode(0, 30, "A", signatureFunction);
30         Node b = createNode(80, 30, "B", signatureFunction);
31         Node c = createNode(110, 0, "C", signatureFunction);
32         Node d = createNode(140, 30, "D", signatureFunction);
33         Node e = createNode(110, 60, "E", signatureFunction);
34         Node f = createNode(50, 60, "F", signatureFunction);
35
36         createEdge(a, b, line1, line2, line3, line4);
37         createEdge(c, b, line1, line2);
38         createEdge(c, d, line1);
39         createEdge(d, e, line1);
40         createEdge(e, f, line1, line3, line4);
41         createEdge(b, e, line3, line4);
42     }
43 }
44
45 }

```

#### MetroMapExampleGraph.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.tramaps.example;
6
7  import ch.geomo.tramaps.graph.Node;
8  import ch.geomo.tramaps.graph.Route;
9  import ch.geomo.tramaps.map.MetroMap;

```

```

10 import ch.geomo.tramaps.map.signature.NodeSignature;
11 import ch.geomo.tramaps.map.signature.RectangleStationSignature;
12 import javafx.scene.paint.Color;
13
14 import java.util.function.Function;
15
16 public class MetroMapExampleGraph extends MetroMap {
17
18     public MetroMapExampleGraph() {
19
20         super(2, 25, 25);
21
22         Function<Node, NodeSignature> signatureFunction = RectangleStationSignature::new;
23
24         Node a = createNode(150, 200, "A", signatureFunction);
25         Node b = createNode(150, 100, "B", signatureFunction);
26         Node c = createNode(200, 100, "C", signatureFunction);
27         Node d = createNode(200, 150, "D", signatureFunction);
28         Node e = createNode(200, 250, "E", signatureFunction);
29         Node f = createNode(150, 300, "F", signatureFunction);
30         Node g = createNode(100, 300, "G", signatureFunction);
31         Node h = createNode(100, 200, "H", signatureFunction);
32         Node i = createNode(100, 150, "I", signatureFunction);
33         Node j = createNode(150, 250, "J", signatureFunction);
34         Node k = createNode(170, 250, "K", signatureFunction);
35         Node n = createNode(100, 250, "N", signatureFunction);
36         Node o = createNode(170, 150, "O", signatureFunction);
37         Node s = createNode(150, 50, "S", signatureFunction);
38
39         Route line1 = new Route("U1", 20, Color.BLUE);
40         Route line2 = new Route("U2", 20, Color.RED);
41         Route line3 = new Route("U3", 20, Color.GREEN);
42         Route line4 = new Route("U4", 20, Color.YELLOW);
43         Route line5 = new Route("U5", 20, Color.ORANGE);
44         Route line6 = new Route("U6", 20, Color.MAGENTA);
45         Route line7 = new Route("U7", 20, Color.BLACK);
46
47         createEdge(a, b, line1, line2, line3, line4, line5);
48         createEdge(c, b, line1, line2, line4, line5);
49         createEdge(c, d, line1, line2, line4, line5);
50         createEdge(d, e, line1, line2, line4, line5);
51         createEdge(e, f, line1, line2, line4, line5, line6);
52         createEdge(f, g, line1, line6);
53         createEdge(g, n, line1, line2, line4);
54         createEdge(n, h, line1, line2);
55         createEdge(h, a, line1, line2, line3, line6, line7);
56         createEdge(h, i, line1, line3, line6);
57         createEdge(i, b, line1, line2, line3, line4, line5, line6, line7);
58         createEdge(i, a, line1, line4, line5);
59         createEdge(a, j, line5);
60         createEdge(j, k, line5, line2);
61         createEdge(e, k, line2);
62         createEdge(o, k, line2, line5, line4);
63         createEdge(o, d, line1, line2, line4, line5, line6);
64         createEdge(b, s, line1);
65         createEdge(c, s, line1);
66
67     }
68
69 }

```

## MetroMapZuerich.java

```

1  /*
2   * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3   */
4
5  package ch.geomo.tramaps.example;
6
7  import ch.geomo.tramaps.graph.Node;
8  import ch.geomo.tramaps.graph.Route;
9  import ch.geomo.tramaps.map.MetroMap;
10 import ch.geomo.tramaps.map.MetroMapEdgeBuilder;
11 import ch.geomo.tramaps.map.signature.RectangleStationSignature;
12 import javafx.scene.paint.Color;
13 import org.jetbrains.annotations.NotNull;
14
15 public class MetroMapZuerich extends MetroMap {
16

```

```

17 public MetroMapZuerich() {
18
19     super(2, 25, 25);
20
21     Route s2 = new Route("S2", 5, Color.BLACK);
22     Route s3 = new Route("S3", 5, Color.BLACK);
23     Route s4 = new Route("S4", 5, Color.BLACK);
24     Route s5 = new Route("S5", 5, Color.BLACK);
25     Route s6 = new Route("S6", 5, Color.BLACK);
26     Route s7 = new Route("S7", 5, Color.BLACK);
27     Route s8 = new Route("S8", 5, Color.BLACK);
28     Route s9 = new Route("S9", 5, Color.BLACK);
29     Route s10 = new Route("S10", 5, Color.BLACK);
30     Route s11 = new Route("S10", 5, Color.BLACK);
31     Route s12 = new Route("S12", 5, Color.BLACK);
32     Route s14 = new Route("S14", 5, Color.BLACK);
33     Route s15 = new Route("S15", 5, Color.BLACK);
34     Route s16 = new Route("S16", 5, Color.BLACK);
35     Route s18 = new Route("S18", 5, Color.BLACK);
36     Route s19 = new Route("S19", 5, Color.BLACK);
37     Route s21 = new Route("S21", 5, Color.BLACK);
38     Route s24 = new Route("S24", 5, Color.BLACK);
39     Route s25 = new Route("S25", 5, Color.BLACK);
40     Route s42 = new Route("S24", 5, Color.BLACK);
41     Route sbb = new Route("SBB", 5, Color.BLACK);
42
43     // not using umlauts in order to not have trouble with latex ;- )
44     // when including this class
45     Node hb = createNode("Zuerich HB", 120, 105);
46
47     Node uetlibergTrainCrossing = createCrossingNode(115, 90); // C1
48     Node selnau = createNode("Selnau", 110, 85);
49     Node selnauToBinzGiesshuebel = createJunctionNode(105, 80); // J1
50     Node binz = createNode("Binz", 95, 80);
51     Node friesenberg = createNode("Friesenberg", 90, 80);
52     Node giesshuebel = createNode("Giesshuebel", 110, 75);
53     Node saalsporthalle = createNode("Saalsporthalle", 110, 70);
54
55     new MetroMapEdgeBuilder(this)
56         .routes(s4, s10)
57         .station(hb)
58         .bend(120, 95) // B1
59         .crossing(uetlibergTrainCrossing)
60         .station(selnau)
61         .junction(selnauToBinzGiesshuebel)
62         .create();
63
64     new MetroMapEdgeBuilder(this)
65         .routes(s4)
66         .junction(selnauToBinzGiesshuebel)
67         .station(giesshuebel)
68         .station(saalsporthalle)
69         .create();
70
71     new MetroMapEdgeBuilder(this)
72         .routes(s10)
73         .junction(selnauToBinzGiesshuebel)
74         .station(binz)
75         .station(friesenberg)
76         .create();
77
78     Node hbToHardbrueckeWipkingenWiedikon = createJunctionNode(110, 105); // J2
79
80     Node wiedikon = createNode("Wiedikon", 110, 95);
81     Node enge = createNode("Enge", 120, 85);
82     Node wollishofen = createNode("Wollishofen", 145, 60);
83
84     new MetroMapEdgeBuilder(this)
85         .routes(s2, s3, s5, s6, s7, s8, s9, s11, s12, s15, s16, s21, s24, s25, s42, sbb)
86         .station(hb)
87         .junction(hbToHardbrueckeWipkingenWiedikon)
88         .create();
89
90     new MetroMapEdgeBuilder(this)
91         .routes(s2, s8, s24, s25, sbb)
92         .junction(hbToHardbrueckeWipkingenWiedikon)
93         .bend(105, 100) // B2
94         .station(wiedikon)
95         .crossing(uetlibergTrainCrossing)
96         .station(enge)

```

```

97         .station(wollishofen)
98         .create();
99
100 Node hardbruecke = createNode("Hardbruecke", 100, 105);
101 Node altstetten = createNode("Altstetten", 75, 105);
102 Node hardbrueckeToOerlikonAltstetten = createJunctionNode(95, 105); // J3
103 Node oerlikonToWipkingenHardbruecke = createJunctionNode(105, 135); // J4
104 Node oerlikonToHbWipkingenHardbruecke = createJunctionNode(105, 140); // J5
105 Node oerlikon = createNode("Oerlikon", 105, 145);
106 Node wipkingen = createNode("Wipkingen", 105, 110);
107
108 new MetroMapEdgeBuilder(this)
109     .routes(s3, s5, s6, s7, s9, s11, s12, s15, s16, s21, s42, sbb)
110     .junction(hbToHardbrueckeWipkingenWiedikon)
111     .station(hardbruecke)
112     .create();
113
114 new MetroMapEdgeBuilder(this)
115     .routes(s3, s5, s6, s7, s9, s11, s12, s15, s16, s21, s42, sbb)
116     .station(hardbruecke)
117     .junction(hardbrueckeToOerlikonAltstetten)
118     .create();
119
120 new MetroMapEdgeBuilder(this)
121     .routes(s6, s7, s9, s15, s16, s21)
122     .junction(hardbrueckeToOerlikonAltstetten)
123     .bend(95, 125)
124     .junction(oerlikonToWipkingenHardbruecke)
125     .create();
126
127 new MetroMapEdgeBuilder(this)
128     .routes(s6, s7, s9, s15, s16, s21, s24, sbb)
129     .junction(oerlikonToWipkingenHardbruecke)
130     .junction(oerlikonToHbWipkingenHardbruecke)
131     .create();
132
133 new MetroMapEdgeBuilder(this)
134     .routes(s2, s6, s7, s8, s9, s14, s15, s16, s19, s21, s24, sbb)
135     .junction(oerlikonToHbWipkingenHardbruecke)
136     .station(oerlikon)
137     .create();
138
139 new MetroMapEdgeBuilder(this)
140     .routes(s3, s42, s12, sbb, s19, s5, s14)
141     .junction(hardbrueckeToOerlikonAltstetten)
142     .station(altstetten)
143     .create();
144
145 new MetroMapEdgeBuilder(this)
146     .routes(s24, sbb)
147     .junction(hbToHardbrueckeWipkingenWiedikon)
148     .station(wipkingen)
149     .junction(oerlikonToWipkingenHardbruecke)
150     .create();
151
152 Node stadelhofenToHb = createJunctionNode(130, 105); // J6
153 Node stadelhofen = createNode("Stadelhofen", 135, 100);
154 Node stadelhofenToTiefenbrunnenStettbach = createJunctionNode(140, 95); // J7
155 Node tiefenbrunnen = createNode("Tiefenbrunnen", 160, 75);
156
157 new MetroMapEdgeBuilder(this)
158     .routes(s2, s8, s14, s19, s9, s5, s15, s24, s11, s3, s7, s6, s16, s12)
159     .station(hb)
160     .junction(stadelhofenToHb)
161     .create();
162
163 new MetroMapEdgeBuilder(this)
164     .routes(s2, s8, s14, s19)
165     .junction(stadelhofenToHb)
166     .bend(130, 115)
167     .junction(oerlikonToHbWipkingenHardbruecke)
168     .create();
169
170 new MetroMapEdgeBuilder(this)
171     .routes(s9, s5, s15, s24, s11, s3, s7, s6, s16, s12)
172     .junction(stadelhofenToHb)
173     .station(stadelhofen)
174     .create();
175
176 new MetroMapEdgeBuilder(this)

```

```

177         .routes(s18, s16, s6, s7, s11, s12, s3, s9, s15, s5)
178         .station(stadelhofen)
179         .junction(stadelhofenToTiefenbrunnenStettbach)
180         .create();
181
182     new MetroMapEdgeBuilder(this)
183         .routes(s18, s16, s6, s7)
184         .junction(stadelhofenToTiefenbrunnenStettbach)
185         .station(tiefenbrunnen)
186         .create();
187
188     Node oerlikonToWallisellen = createJunctionNode(105, 150); // J8
189
190     new MetroMapEdgeBuilder(this)
191         .routes(s2, s6, s7, s8, s9, s14, s15, s16, s19, s21, s24, sbb)
192         .station(oerlikon)
193         .junction(oerlikonToWallisellen)
194         .create();
195
196     Node wallisellen = createNode("Wallisellen", 140, 165);
197     Node crossJunction = createJunctionNode(160, 165); // J9
198
199     new MetroMapEdgeBuilder(this)
200         .routes(s8, s14, s19)
201         .junction(oerlikonToWallisellen)
202         .bend(110, 155) // B10
203         .bend(130, 155) // B11
204         .station(wallisellen)
205         .junction(crossJunction)
206         .create();
207
208     Node stettbach = createNode("Stettbach", 160, 150);
209
210     new MetroMapEdgeBuilder(this)
211         .routes(s11, s12, s3, s9, s15, s5)
212         .junction(crossJunction)
213         .station(stettbach)
214         .bend(160, 115) // B12
215         .junction(stadelhofenToTiefenbrunnenStettbach)
216         .create();
217
218     Node dietlikon = createNode("Dietlikon", 170, 190);
219
220     new MetroMapEdgeBuilder(this)
221         .routes(s8, s19, s3, s11, s12)
222         .junction(crossJunction)
223         .bend(160, 180) // B13
224         .station(dietlikon)
225         .create();
226
227     Node duebendorf = createNode("Duebendorf", 205, 160);
228
229     new MetroMapEdgeBuilder(this)
230         .routes(s14, s9, s15, s5)
231         .junction(crossJunction)
232         .bend(200, 165) // B14
233         .station(duebendorf)
234         .create();
235
236     Node schlieren = createNode("Schlieren", 40, 105);
237     Node urdorf = createNode("Urdorf", 40, 75);
238
239     Node altstettenToSchlierenUrdorf = createJunctionNode(70, 105);
240
241     new MetroMapEdgeBuilder(this)
242         .routes(s3, s42, s12, sbb, s19, s5, s14)
243         .station(altstetten)
244         .junction(altstettenToSchlierenUrdorf)
245         .create();
246
247     new MetroMapEdgeBuilder(this)
248         .routes(s5, s14)
249         .junction(altstettenToSchlierenUrdorf)
250         .station(urdorf)
251         .create();
252
253     new MetroMapEdgeBuilder(this)
254         .routes(s3, s42, s12, sbb, s19)
255         .junction(altstettenToSchlierenUrdorf)
256         .station(schlieren)

```



```

257         .create();
258     }
259 }
260
261 /**
262  * @return a node for given position and name using a {@link RectangleStationSignature}
263  */
264 private Node createNode(@NotNull String name, double x, double y) {
265     return createNode(x, y, name, RectangleStationSignature::new);
266 }
267
268 }

```

## D.6 Util-Klassen

### D.6.1 Package ch.geomo.util.collection.\*

#### GCollection.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.util.collection;
6
7  import ch.geomo.util.collection.list.EnhancedList;
8  import ch.geomo.util.collection.list.GList;
9  import ch.geomo.util.collection.pair.Pair;
10 import ch.geomo.util.collection.set.EnhancedSet;
11 import ch.geomo.util.collection.set.GSet;
12 import ch.geomo.util.collection.tuple.Tuple;
13 import org.jetbrains.annotations.NotNull;
14 import org.jetbrains.annotations.Nullable;
15
16 import java.util.Collection;
17
18 public enum GCollection {
19
20     /* factory class */;
21
22     @NotNull
23     @SafeVarargs
24     public static <E> EnhancedList<E> list(@NotNull E... elements) {
25         return GList.createList(elements);
26     }
27
28     @NotNull
29     public static <E> EnhancedList<E> list(@NotNull Collection<E> c) {
30         return GList.createList(c);
31     }
32
33     @NotNull
34     public static <E> EnhancedList<E> mergeLists(@NotNull Collection<E> c1, @NotNull Collection<E> c2) {
35         return GList.merge(c1, c2);
36     }
37
38     @NotNull
39     @SafeVarargs
40     public static <E> EnhancedSet<E> set(@NotNull E... elements) {
41         return GSet.createSet(elements);
42     }
43
44     @NotNull
45     public static <E> Pair<E> pair(@Nullable E firstElement, @Nullable E secondElement) {
46         return Pair.of(firstElement, secondElement);
47     }
48
49     @NotNull
50     public static <E> Pair<E> mutablePair(@Nullable E firstElement, @Nullable E secondElement) {
51         return Pair.of(firstElement, secondElement, true);
52     }
53
54     @NotNull
55     public static <T, S> Tuple<T, S> tuple(@Nullable T firstElement, @Nullable S secondElement) {
56         return Tuple.createTuple(firstElement, secondElement);
57     }
58 }

```

```

59     @NotNull
60     public static <T, S> Tuple<T, S> mutableTuple(@Nullable T firstElement, @Nullable S secondElement) {
61         return Tuple.createTuple(firstElement, secondElement, true);
62     }
63
64 }

```

## GCollectors.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.util.collection;
6
7  import ch.geomo.util.collection.list.EnhancedList;
8  import ch.geomo.util.collection.list.GList;
9  import ch.geomo.util.collection.set.EnhancedSet;
10 import ch.geomo.util.collection.set.GSet;
11 import org.jetbrains.annotations.NotNull;
12
13 import java.util.List;
14 import java.util.Set;
15 import java.util.stream.Collector;
16
17 /**
18 * Provides {@link Collector} implementations for {@link EnhancedList} and {@link EnhancedSet}.
19 */
20 public enum GCollectors {
21
22     ;
23
24     @NotNull
25     public static <T> Collector<T, ?, EnhancedList<T>> toList() {
26         return Collector.of(
27             GList::new,
28             List::add,
29             (left, right) -> {
30                 left.addAll(right);
31                 return left;
32             },
33             Collector.Characteristics.IDENTITY_FINISH
34         );
35     }
36
37     @NotNull
38     public static <T> Collector<T, ?, EnhancedSet<T>> toSet() {
39         return Collector.of(
40             GSet::new,
41             Set::add,
42             (left, right) -> {
43                 left.addAll(right);
44                 return left;
45             },
46             Collector.Characteristics.UNORDERED,
47             Collector.Characteristics.IDENTITY_FINISH
48         );
49     }
50
51 }

```

## EnhancedList.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.util.collection.list;
6
7  import ch.geomo.util.collection.pair.Pair;
8  import ch.geomo.util.collection.set.EnhancedSet;
9  import org.jetbrains.annotations.NotNull;
10
11 import java.util.Collection;
12 import java.util.Comparator;
13 import java.util.List;
14 import java.util.Optional;
15 import java.util.function.BooleanSupplier;

```

```

16 import java.util.function.Consumer;
17 import java.util.function.Function;
18 import java.util.function.Predicate;
19 import java.util.stream.Stream;
20
21 public interface EnhancedList<E> extends List<E> {
22
23     @NotNull
24     EnhancedList<E> union(@NotNull Collection<E> list);
25
26     @NotNull
27     EnhancedList<E> intersection(@NotNull Collection<E> list);
28
29     @NotNull
30     Pair<EnhancedList<E>> diff(@NotNull Collection<E> list);
31
32     boolean anyMatch(@NotNull Predicate<E> predicate);
33
34     boolean allMatch(@NotNull Predicate<E> predicate);
35
36     boolean noneMatch(@NotNull Predicate<E> predicate);
37
38     boolean contains(@NotNull Collection<E> list);
39
40     boolean hasEqualContent(@NotNull Collection<E> list);
41
42     @NotNull
43     Optional<E> first();
44
45     boolean hasOneElement();
46
47     boolean hasMoreThanOneElement();
48
49     @NotNull
50     EnhancedList<E> reverse();
51
52     @NotNull
53     EnhancedList<E> reverseIf(@NotNull BooleanSupplier supplier);
54
55     @NotNull
56     EnhancedList<E> addElements(@NotNull E... elements);
57
58     @NotNull
59     EnhancedList<E> addElements(@NotNull Collection<E> elements);
60
61     @NotNull
62     EnhancedList<E> addElements(@NotNull Stream<E> elementStream);
63
64     @NotNull
65     EnhancedList<E> removeElements(@NotNull E... elements);
66
67     @NotNull
68     EnhancedList<E> removeElements(@NotNull Predicate<E> predicate);
69
70     @NotNull
71     EnhancedList<E> keepElements(@NotNull Predicate<E> predicate);
72
73     @NotNull
74     EnhancedList<E> without(@NotNull Predicate<E> predicate);
75
76     @NotNull
77     EnhancedList<E> filter(@NotNull Predicate<E> predicate);
78
79     @NotNull
80     EnhancedList<E> doIfNotEmpty(@NotNull Consumer<EnhancedList<E>> consumer);
81
82     @NotNull <T> EnhancedList<T> map(@NotNull Function<E, T> function);
83
84     @NotNull
85     EnhancedList<Pair<E>> toPairList();
86
87     @NotNull
88     EnhancedList<Pair<E>> toPairList(@NotNull Predicate<Pair<E>> predicate);
89
90     @NotNull
91     Stream<Pair<E>> toPairStream(@NotNull Predicate<Pair<E>> predicate);
92
93     @NotNull
94     EnhancedSet<E> toSet();
95

```

```

96     @NotNull
97     EnhancedList<E> sortElements(@NotNull Comparator<? super E> c);
98
99     @NotNull
100    @Override
101    E[] toArray();
102
103 }

```

## GList.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.util.collection.list;
6
7  import ch.geomo.util.collection.pair.Pair;
8  import ch.geomo.util.collection.set.EnhancedSet;
9  import ch.geomo.util.collection.set.GSet;
10 import org.jetbrains.annotations.NotNull;
11
12 import java.util.*;
13 import java.util.function.BooleanSupplier;
14 import java.util.function.Consumer;
15 import java.util.function.Function;
16 import java.util.function.Predicate;
17 import java.util.stream.Collectors;
18 import java.util.stream.Stream;
19
20 public class GList<E> extends ArrayList<E> implements EnhancedList<E> {
21
22     public GList() {
23         super();
24     }
25
26     public GList(@NotNull Collection<E> c) {
27         super(c);
28     }
29
30     @SafeVarargs
31     public GList(@NotNull E... elements) {
32         super(Arrays.asList(elements));
33     }
34
35     public GList(@NotNull Stream<E> stream) {
36         super(stream.collect(Collectors.toList()));
37     }
38
39     @NotNull
40     @Override
41     public Optional<E> first() {
42         return isEmpty() ? Optional.empty() : Optional.of(get(0));
43     }
44
45     @Override
46     public boolean hasOneElement() {
47         return size() == 1;
48     }
49
50     @Override
51     public boolean hasMoreThanOneElement() {
52         return size() > 1;
53     }
54
55     @NotNull
56     @Override
57     public EnhancedList<E> union(@NotNull Collection<E> list) {
58         return createList(list).addElements(this);
59     }
60
61     @NotNull
62     @Override
63     public EnhancedList<E> intersection(@NotNull Collection<E> list) {
64         return createList(stream()
65             .filter(list::contains));
66     }
67
68     @NotNull

```

```

69     @Override
70     public Pair<EnhancedList<E>> diff(@NotNull Collection<E> list) {
71         EnhancedList<E> intersection = intersection(list);
72         EnhancedList<E> thisList = createList(stream()
73             .filter(e -> !intersection.contains(e)));
74         EnhancedList<E> otherList = createList(list.stream()
75             .filter(e -> !intersection.contains(e)));
76         return Pair.of(thisList, otherList);
77     }
78
79     @NotNull
80     @Override
81     public EnhancedList<E> without(@NotNull Predicate<E> predicate) {
82         return filter(e -> !predicate.test(e));
83     }
84
85     @NotNull
86     @Override
87     public EnhancedList<E> filter(@NotNull Predicate<E> predicate) {
88         return createList(stream()
89             .filter(predicate));
90     }
91
92     @NotNull
93     public EnhancedList<E> doIfNotEmpty(@NotNull Consumer<EnhancedList<E>> consumer) {
94         if (!isEmpty()) {
95             consumer.accept(this);
96         }
97         return this;
98     }
99
100    @NotNull
101    @Override
102    public EnhancedList<E> reverse() {
103        Collections.reverse(this);
104        return this;
105    }
106
107    @NotNull
108    public EnhancedList<E> reverseIf(@NotNull BooleanSupplier supplier) {
109        if (supplier.getAsBoolean()) {
110            return reverse();
111        }
112        return this;
113    }
114
115    @Override
116    public boolean anyMatch(@NotNull Predicate<E> predicate) {
117        return stream().anyMatch(predicate);
118    }
119
120    @Override
121    public boolean allMatch(@NotNull Predicate<E> predicate) {
122        return stream().allMatch(predicate);
123    }
124
125    @Override
126    public boolean noneMatch(@NotNull Predicate<E> predicate) {
127        return stream().noneMatch(predicate);
128    }
129
130    @Override
131    public boolean contains(@NotNull Collection<E> list) {
132        return stream().allMatch(list::contains);
133    }
134
135    @Override
136    public boolean hasEqualContent(@NotNull Collection<E> list) {
137        if (list.size() != list.size()) {
138            return false;
139        }
140        return anyMatch(list::contains);
141    }
142
143    @NotNull
144    @Override
145    public final EnhancedList<E> addElements(@NotNull Stream<E> elementStream) {
146        return addElements(elementStream.collect(Collectors.toList()));
147    }
148

```

```

149     @NotNull
150     @Override
151     @SafeVarargs
152     public final EnhancedList<E> addElements(@NotNull E... elements) {
153         return addElements(Arrays.asList(elements));
154     }
155
156     @NotNull
157     @Override
158     public final EnhancedList<E> addElements(@NotNull Collection<E> elements) {
159         addAll(elements);
160         return this;
161     }
162
163     @NotNull
164     @Override
165     @SafeVarargs
166     public final EnhancedList<E> removeElements(@NotNull E... elements) {
167         removeAll(Arrays.asList(elements));
168         return this;
169     }
170
171     @NotNull
172     @Override
173     public EnhancedList<E> removeElements(@NotNull Predicate<E> predicate) {
174         removeIf(predicate);
175         return this;
176     }
177
178     @NotNull
179     @Override
180     public EnhancedList<E> keepElements(@NotNull Predicate<E> predicate) {
181         return removeElements(e -> !predicate.test(e));
182     }
183
184     @NotNull
185     @Override
186     public <T> EnhancedList<T> map(@NotNull Function<E, T> function) {
187         return createList(stream()
188             .map(function));
189     }
190
191     @NotNull
192     @Override
193     public EnhancedList<Pair<E>> toPairList() {
194         return toPairList(e -> true);
195     }
196
197     @NotNull
198     @Override
199     public EnhancedList<Pair<E>> toPairList(@NotNull Predicate<Pair<E>> predicate) {
200         return createList(toPairStream(predicate).collect(Collectors.toList()));
201     }
202
203     @NotNull
204     @Override
205     public Stream<Pair<E>> toPairStream(@NotNull Predicate<Pair<E>> predicate) {
206
207         if (isEmpty()) {
208             return Stream.of();
209         }
210
211         return stream()
212             .flatMap(v1 -> stream()
213                 .map(v2 -> Pair.of(v1, v2)))
214             .filter(predicate)
215             .distinct();
216     }
217
218
219     @NotNull
220     @Override
221     public EnhancedSet<E> toSet() {
222         return GSet.createSet(this);
223     }
224
225     @NotNull
226     @Override
227     public EnhancedList<E> sortElements(@NotNull Comparator<? super E> c) {
228         sort(c);

```

```

229     return this;
230 }
231
232 @NotNull
233 @Override
234 @SuppressWarnings("unchecked")
235 public E[] toArray() {
236     return (E[]) super.toArray();
237 }
238
239 @NotNull
240 public static <E> EnhancedList<E> cast(@NotNull List<E> list) {
241     if (list instanceof EnhancedList) {
242         return (EnhancedList<E>) list;
243     }
244     throw new ClassCastException("Cannot cast List to EnhancedList.");
245 }
246
247 @NotNull
248 public static <E> EnhancedList<E> createList(@NotNull Collection<E> c) {
249     return new GList<>(c);
250 }
251
252 @NotNull
253 public static <E> EnhancedList<E> createList(@NotNull Stream<E> stream) {
254     return new GList<>(stream);
255 }
256
257 @NotNull
258 public static <E> EnhancedList<E> createList(@NotNull Stream<E> stream1, @NotNull Stream<E> stream2) {
259     return new GList<>(Stream.concat(stream1, stream2));
260 }
261
262 @NotNull
263 @SafeVarargs
264 public static <E> EnhancedList<E> createList(@NotNull E... elements) {
265     return new GList<>(elements);
266 }
267
268 @NotNull
269 public static <E> EnhancedList<E> merge(@NotNull Collection<? extends E> c1, @NotNull Collection<? extends E> c2) {
270     return new GList<>(Stream.concat(c1.stream(), c2.stream()));
271 }
272
273 @NotNull
274 @SafeVarargs
275 public static <E> EnhancedList<E> merge(@NotNull Collection<E> c1, @NotNull Collection<E> c2, @NotNull Collection<E>... moreCollections) {
276     GList<E> list = new GList<>(Stream.concat(c1.stream(), c2.stream()));
277     Arrays.stream(moreCollections).forEach(list::addElement);
278     return list;
279 }
280
281 }

```

## AbstractPair.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.util.collection.pair;
6
7  import java.util.Objects;
8
9  public abstract class AbstractPair<T> implements Pair<T> {
10
11     protected T first;
12     protected T second;
13
14     public AbstractPair() {
15     }
16
17     public AbstractPair(Pair<T> pair) {
18         if (pair != null) {
19             first = pair.getFirst();
20             second = pair.getSecond();
21         }
22     }
23

```

```

24     public AbstractPair(T first, T second) {
25         this.first = first;
26         this.second = second;
27     }
28
29     @Override
30     public T getFirst() {
31         return first;
32     }
33
34     @Override
35     public T getSecond() {
36         return second;
37     }
38
39     @Override
40     public boolean equals(Object obj) {
41         return obj instanceof AbstractPair
42             && ((Objects.equals(first, ((AbstractPair) obj).first) && Objects.equals(second, ((AbstractPair) obj).second))
43             || (Objects.equals(first, ((AbstractPair) obj).second) && Objects.equals(second, ((AbstractPair) obj).first)));
44     }
45
46     @Override
47     public int hashCode() {
48         return Objects.hash(first, second);
49     }
50
51     @Override
52     public String toString() {
53         return getClass().getSimpleName() + ": {" + first + ", " + second + "}";
54     }
55 }
56 }

```

## ImmutablePair.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.util.collection.pair;
6
7  import org.jetbrains.annotations.Nullable;
8
9  /**
10 * An implementation set an immutable pair.
11 */
12 public class ImmutablePair<T> extends AbstractPair<T> {
13
14     public ImmutablePair(@Nullable T first, @Nullable T second) {
15         super(first, second);
16     }
17
18 }

```

## Pair.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.util.collection.pair;
6
7  import ch.geomo.util.collection.tuple.Tuple;
8  import org.jetbrains.annotations.NotNull;
9
10 import java.util.*;
11 import java.util.stream.Collectors;
12 import java.util.stream.Stream;
13
14 /**
15 * Represents a pair of one type. Two pairs are always equal if both set the first pair values
16 * are equals in any combination to the other pair's values.
17 */
18 public interface Pair<T> extends Tuple<T, T> {
19
20     /**
21      * @return false if element is equals to null at the given index

```



```

22     * @throws IndexOutOfBoundsException if index is neither 1 or 0
23     */
24     default boolean isNull(int index) {
25         return get(index) == null;
26     }
27
28     /**
29     * Gets the other value. Throws a {@link NoSuchElementException} if given value is not an item of the current pair.
30     * @throws NoSuchElementException if given value is not a value of the current {@link Pair}
31     */
32     default T getOtherValue(T value) {
33         if (Objects.equals(get(0), value)) {
34             return get(1);
35         }
36         if (Objects.equals(get(1), value)) {
37             return get(0);
38         }
39         throw new NoSuchElementException("Given value is not an item of this pair!");
40     }
41
42     /**
43     * @return a {@link Stream} for this pair, note: stream may contain null values!
44     */
45     default Stream<T> stream() {
46         return Stream.of(getFirst(), getSecond());
47     }
48
49     /**
50     * @return a {@link Stream} for this pair without null value
51     */
52     default Stream<T> nonNullStream() {
53         return Stream.of(getFirst(), getSecond())
54             .filter(node -> node != null);
55     }
56
57     /**
58     * @return true if given value is contained in this instance
59     */
60     default boolean contains(T value) {
61         return Objects.equals(value, get(0)) || Objects.equals(value, get(1));
62     }
63
64     /**
65     * Gets the first or second value by index. Index start with 0 in order to be consistent with other APIs. Only
66     * 0 and 1 are allowed since only two items are hold by a pair.
67     * @return the element at the given index
68     * @throws IndexOutOfBoundsException if index is neither 1 or 0
69     */
70     default T get(int index) {
71         if (index > 1 && index < 0) {
72             throw new IndexOutOfBoundsException("An index > 1 or < 0 is not allowed.");
73         }
74         return index == 0 ? getFirst() : getSecond();
75     }
76
77     @NotNull
78     default List<T> toList() {
79         return stream()
80             .filter(value -> value != null)
81             .collect(Collectors.toList());
82     }
83
84     default boolean hasNonNullValues() {
85         return first() != null || second() != null;
86     }
87
88     /**
89     * @return a new {@link ImmutablePair} with given elements.
90     */
91     @NotNull
92     static <T> Pair<T> of(T first, T second) {
93         return new ImmutablePair<>(first, second);
94     }
95
96     /**
97     * @return a new {@link MutablePair} or {@link ImmutablePair} with given elements depending the given boolean
98     */
99     @NotNull
100     static <T> Pair<T> of(T first, T second, boolean mutable) {
101         if (mutable) {

```

```

102         return new MutablePair<>(first, second);
103     }
104     return new ImmutablePair<>(first, second);
105 }
106
107 /**
108  * @return a {@link Set} of {@link ImmutablePair} with given collections
109  */
110 @NotNull
111 static <T> Set<Pair<T>> from(Collection<T> coll, Collection<T> col2) {
112     if (coll == null || col2 == null) {
113         return Collections.emptySet();
114     }
115     return coll.stream()
116         .flatMap(v1 -> col2.stream()
117             .map(v2 -> Pair.of(v1, v2)))
118         .collect(Collectors.toSet());
119 }
120
121 }

```

## MutablePair.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.util.collection.pair;
6
7  import org.jetbrains.annotations.NotNull;
8  import org.jetbrains.annotations.Nullable;
9
10 /**
11  * An implementation set a mutable pair.
12  */
13 public class MutablePair<T> extends AbstractPair<T> {
14
15     public MutablePair() {
16         super();
17     }
18
19     public MutablePair(@Nullable Pair<T> pair) {
20         super(pair);
21     }
22
23     public MutablePair(@Nullable T first, @Nullable T second) {
24         super(first, second);
25     }
26
27     /**
28     * Sets/updates the first value.
29     */
30     public void setFirst(@Nullable T first) {
31         this.first = first;
32     }
33
34     /**
35     * Sets/updates the second value.
36     */
37     public void setSecond(@Nullable T second) {
38         this.second = second;
39     }
40
41     /**
42     * Sets/updates the value at given index.
43     * @throws IndexOutOfBoundsException if index is neither 1 or 0
44     */
45     public void set(int index, @Nullable T value) {
46
47         if (index < 0 || index > 1) {
48             throw new IndexOutOfBoundsException("An index > 1 or < 0 is not allowed.");
49         }
50
51         if (index == 0) {
52             first = value;
53         }
54         else {
55             second = value;
56         }

```

```

57
58 }
59
60 /**
61  * Replaces values with values of the given pair.
62  */
63 public void replaceValues(@NotNull Pair<T> pair) {
64     first = pair.first();
65     second = pair.second();
66 }
67
68 /**
69  * Swaps first with second value.
70  */
71 public void swapValues() {
72     T firstValue = first;
73     first = second;
74     second = firstValue;
75 }
76
77 /**
78  * Sets first and second value to null.
79  */
80 public void clear() {
81     first = null;
82     second = null;
83 }
84
85 }

```

## EnhancedSet.java

```

1  /*
2   * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3   */
4
5  package ch.geomo.util.collection.set;
6
7  import ch.geomo.util.collection.list.EnhancedList;
8  import ch.geomo.util.collection.pair.Pair;
9  import org.jetbrains.annotations.NotNull;
10
11 import java.util.Collection;
12 import java.util.Optional;
13 import java.util.Set;
14 import java.util.function.Function;
15 import java.util.function.Predicate;
16 import java.util.stream.Stream;
17
18 public interface EnhancedSet<E> extends Set<E> {
19
20     @NotNull
21     EnhancedSet<E> union(@NotNull Collection<E> list);
22
23     @NotNull
24     EnhancedSet<E> intersection(@NotNull Collection<E> list);
25
26     @NotNull
27     Pair<EnhancedSet<E>> diff(@NotNull Collection<E> list);
28
29     /**
30      * Convenience method. Gets the first element of this {@link Set}. The element
31      * is probably different to the last method call. Do only use this
32      * method if you are sure that only one element is available.
33      * @return the first element of this {@link Set}
34      * @see #hasOneElement()
35      */
36     @NotNull
37     Optional<E> first();
38
39     @SuppressWarnings("unused")
40     boolean hasOneElement();
41
42     boolean hasMoreThanOneElement();
43
44     boolean anyMatch(@NotNull Predicate<E> predicate);
45
46     boolean contains(@NotNull Collection<E> list);
47

```

```

48     boolean hasEqualContent(@NotNull Collection<E> list);
49
50     @NotNull
51     EnhancedSet<E> addElements(@NotNull E... elements);
52
53     @NotNull
54     EnhancedSet<E> addElements(@NotNull Collection<E> elements);
55
56     @NotNull
57     EnhancedSet<E> addElements(@NotNull Stream<E> elementStream);
58
59     @NotNull
60     EnhancedSet<E> removeElements(@NotNull E... elements);
61
62     @NotNull
63     EnhancedSet<E> removeElements(@NotNull Predicate<E> predicate);
64
65     @NotNull
66     EnhancedSet<E> keepElements(@NotNull Predicate<E> predicate);
67
68     @NotNull
69     EnhancedSet<E> without(@NotNull Predicate<E> predicate);
70
71     @NotNull
72     EnhancedSet<E> filter(@NotNull Predicate<E> predicate);
73
74     @NotNull <T> EnhancedSet<T> map(@NotNull Function<E, T> function);
75
76     @NotNull <T> EnhancedSet<T> flatMap(Function<E, ? extends Stream<? extends T>> mapper);
77
78     @NotNull
79     EnhancedSet<Pair<E>> toPairSet();
80
81     @NotNull
82     EnhancedSet<Pair<E>> toPairSet(@NotNull Predicate<Pair<E>> predicate);
83
84     @NotNull
85     Stream<Pair<E>> toPairStream(@NotNull Predicate<Pair<E>> predicate);
86
87     @NotNull
88     EnhancedList<E> toList();
89
90     @NotNull
91     @Override
92     E[] toArray();
93
94 }

```

## GSet.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.util.collection.set;
6
7  import ch.geomo.util.collection.GCollectors;
8  import ch.geomo.util.collection.list.EnhancedList;
9  import ch.geomo.util.collection.list.GList;
10 import ch.geomo.util.collection.pair.Pair;
11 import org.jetbrains.annotations.NotNull;
12
13 import java.util.*;
14 import java.util.function.Function;
15 import java.util.function.Predicate;
16 import java.util.stream.Collectors;
17 import java.util.stream.Stream;
18
19 public class GSet<E> extends HashSet<E> implements EnhancedSet<E> {
20
21     public GSet() {
22         super();
23     }
24
25     public GSet(@NotNull Collection<E> c) {
26         super(c);
27     }
28
29     @SafeVarargs

```

```

30 public GSet(@NotNull E... elements) {
31     super(Arrays.asList(elements));
32 }
33
34 public GSet(@NotNull Stream<E> stream) {
35     super(stream.collect(Collectors.toList()));
36 }
37
38 @NotNull
39 @Override
40 public Optional<E> first() {
41     return stream().findFirst();
42 }
43
44 @Override
45 public boolean hasOneElement() {
46     return size() == 1;
47 }
48
49 @Override
50 public boolean hasMoreThanOneElement() {
51     return size() > 1;
52 }
53
54 @NotNull
55 @Override
56 public EnhancedSet<E> union(@NotNull Collection<E> c) {
57     return createSet(c.addElements(this);
58 }
59
60 @NotNull
61 @Override
62 public EnhancedSet<E> intersection(@NotNull Collection<E> c) {
63     return createSet(stream()
64         .filter(c::contains));
65 }
66
67 @NotNull
68 @Override
69 public Pair<EnhancedSet<E>> diff(@NotNull Collection<E> c) {
70     List<E> intersection = intersection(c).toList();
71     EnhancedSet<E> thisCollection = createSet(stream()
72         .filter(e -> !intersection.contains(e)));
73     EnhancedSet<E> otherCollection = createSet(c.stream()
74         .filter(e -> !intersection.contains(e)));
75     return Pair.of(thisCollection, otherCollection);
76 }
77
78 @NotNull
79 @Override
80 public EnhancedSet<E> without(@NotNull Predicate<E> predicate) {
81     return filter(e -> !predicate.test(e));
82 }
83
84 @NotNull
85 @Override
86 public EnhancedSet<E> filter(@NotNull Predicate<E> predicate) {
87     return createSet(stream()
88         .filter(predicate));
89 }
90
91 @Override
92 public boolean anyMatch(@NotNull Predicate<E> predicate) {
93     return stream().anyMatch(predicate);
94 }
95
96 @Override
97 public boolean contains(@NotNull Collection<E> c) {
98     return stream().allMatch(c::contains);
99 }
100
101 @Override
102 public boolean hasEqualContent(@NotNull Collection<E> c) {
103     if (size() != c.size()) {
104         return false;
105     }
106     return anyMatch(c::contains);
107 }
108
109 @NotNull

```

```

110  @Override
111  public final EnhancedSet<E> addElements(@NotNull Stream<E> elementStream) {
112      return addElements(elementStream.collect(Collectors.toList()));
113  }
114
115  @NotNull
116  @Override
117  @SafeVarargs
118  public final EnhancedSet<E> addElements(@NotNull E... elements) {
119      return addElements(Arrays.asList(elements));
120  }
121
122  @NotNull
123  @Override
124  public final EnhancedSet<E> addElements(@NotNull Collection<E> elements) {
125      addAll(elements);
126      return this;
127  }
128
129  @NotNull
130  @Override
131  @SafeVarargs
132  public final EnhancedSet<E> removeElements(@NotNull E... elements) {
133      removeAll(Arrays.asList(elements));
134      return this;
135  }
136
137  @NotNull
138  @Override
139  public EnhancedSet<E> removeElements(@NotNull Predicate<E> predicate) {
140      removeIf(predicate);
141      return this;
142  }
143
144  @NotNull
145  @Override
146  public EnhancedSet<E> keepElements(@NotNull Predicate<E> predicate) {
147      return removeElements(e -> !predicate.test(e));
148  }
149
150  @NotNull
151  @Override
152  public <T> EnhancedSet<T> map(@NotNull Function<E, T> function) {
153      return createSet(stream()
154          .map(function));
155  }
156
157  @NotNull
158  @Override
159  public <T> EnhancedSet<T> flatMap(Function<E, ? extends Stream<? extends T>> mapper) {
160      return createSet(stream()
161          .flatMap(mapper));
162  }
163
164  @NotNull
165  @Override
166  public EnhancedSet<Pair<E>> toPairSet() {
167      return toPairSet(e -> true);
168  }
169
170  @NotNull
171  @Override
172  public EnhancedSet<Pair<E>> toPairSet(@NotNull Predicate<Pair<E>> predicate) {
173      return toPairStream(predicate).collect(GCollectors.toSet());
174  }
175  }
176
177  @NotNull
178  @Override
179  public Stream<Pair<E>> toPairStream(@NotNull Predicate<Pair<E>> predicate) {
180
181      if (isEmpty()) {
182          return Stream.of();
183      }
184
185      return stream()
186          .flatMap(v1 -> stream()
187              .map(v2 -> Pair.of(v1, v2)))
188          .filter(predicate)
189          .distinct();

```

```

190
191 }
192
193 @NotNull
194 @Override
195 public EnhancedList<E> toList() {
196     return GList.createList(this);
197 }
198
199 @NotNull
200 @Override
201 @SuppressWarnings("unchecked")
202 public E[] toArray() {
203     return (E[]) super.toArray();
204 }
205
206 @NotNull
207 public static <E> EnhancedSet<E> emptySet() {
208     return new GSet<>();
209 }
210
211 @NotNull
212 public static <E> EnhancedSet<E> createSet(@NotNull Collection<E> c) {
213     return new GSet<>(c);
214 }
215
216 @NotNull
217 public static <E> EnhancedSet<E> createSet(@NotNull Stream<E> stream) {
218     return new GSet<>(stream);
219 }
220
221 @NotNull
222 public static <E> EnhancedSet<E> createSet(@NotNull Stream<E> stream1, @NotNull Stream<E> stream2) {
223     return new GSet<>(Stream.concat(stream1, stream2));
224 }
225
226 @NotNull
227 @SafeVarargs
228 public static <E> EnhancedSet<E> createSet(@NotNull E... elements) {
229     return new GSet<>(elements);
230 }
231
232 }

```

## Tuple.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.util.collection.tuple;
6
7  import org.jetbrains.annotations.NotNull;
8
9  public interface Tuple<T, S> {
10
11     /**
12     * @return the first value
13     */
14     T getFirst();
15
16     /**
17     * Alias for {@link #getFirst()}.
18     */
19     default T first() {
20         return getFirst();
21     }
22
23     /**
24     * @return the second value
25     */
26     S getSecond();
27
28     /**
29     * Alias for {@link #getSecond()}.
30     */
31     default S second() {
32         return getSecond();
33     }

```

```

34
35  /**
36   * @return a new {@link ImmutableTuple} with given elements.
37   */
38  @NotNull
39  static <T, S> Tuple<T, S> createTuple(T first, S second) {
40      return new ImmutableTuple<>(first, second);
41  }
42
43  /**
44   * @return a new {@link ImmutableTuple} with given elements.
45   */
46  @NotNull
47  static <T, S> Tuple<T, S> createTuple(T first, S second, boolean mutable) {
48      if (mutable) {
49          return new MutableTuple<>(first, second);
50      }
51      return new ImmutableTuple<>(first, second);
52  }
53
54 }

```

### ImmutableTuple.java

```

1  /*
2   * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3   */
4
5  package ch.geomo.util.collection.tuple;
6
7  import org.jetbrains.annotations.Nullable;
8
9  public class ImmutableTuple<T, S> implements Tuple<T, S> {
10
11     private final T firstValue;
12     private final S secondValue;
13
14     public ImmutableTuple(@Nullable T firstValue, @Nullable S secondValue) {
15         this.firstValue = firstValue;
16         this.secondValue = secondValue;
17     }
18
19     @Override
20     public T getFirst() {
21         return firstValue;
22     }
23
24     @Override
25     public S getSecond() {
26         return secondValue;
27     }
28
29 }

```

### MutableTuple.java

```

1  /*
2   * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3   */
4
5  package ch.geomo.util.collection.tuple;
6
7  public class MutableTuple<T, S> implements Tuple<T, S> {
8
9     private T firstValue;
10    private S secondValue;
11
12    public MutableTuple(T firstValue, S secondValue) {
13        this.firstValue = firstValue;
14        this.secondValue = secondValue;
15    }
16
17    @Override
18    public T getFirst() {
19        return firstValue;
20    }
21
22    public void setFirst(T firstValue) {

```



```

23     this.firstValue = firstValue;
24 }
25
26 @Override
27 public S getSecond() {
28     return secondValue;
29 }
30
31 public void setSecond(S secondValue) {
32     this.secondValue = secondValue;
33 }
34
35 }

```

## D.6.2 Package ch.geomo.util.geom.\*

### Axis.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.util.geom;
6
7  /**
8   * Represents an axis in a 2D cartesian coordinate system.
9   */
10 public enum Axis {
11     X, // horizontal, west < east
12     Y // vertical, south < north
13 }

```

### GeomUtil.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.util.geom;
6
7  import ch.geomo.util.geom.point.NodePoint;
8  import ch.geomo.util.math.MoveVector;
9  import com.vividsolutions.jts.geom.*;
10 import com.vividsolutions.jts.operation.buffer.BufferParameters;
11 import org.jetbrains.annotations.Contract;
12 import org.jetbrains.annotations.NotNull;
13 import org.jetbrains.annotations.Nullable;
14
15 import java.util.ArrayList;
16 import java.util.Collection;
17 import java.util.List;
18 import java.util.stream.Collectors;
19 import java.util.stream.Stream;
20
21 /**
22  * Provides helper methods for creating and manipulating geometries.
23  */
24 public enum GeomUtil {
25
26     /* util class */;
27
28     private static final PrecisionModel PRECISION_MODEL = new PrecisionModel(10000);
29     private static final GeometryFactory GEOMETRY_FACTORY = new GeometryFactory(PRECISION_MODEL);
30
31     /**
32      * @return the {@link GeometryFactory} used by {@link GeomUtil}
33      */
34     @NotNull
35     public static GeometryFactory getGeometryFactory() {
36         return GEOMETRY_FACTORY;
37     }
38
39     /**
40      * @return the {@link PrecisionModel} used by {@link GeomUtil}
41      */
42     @NotNull
43     public static PrecisionModel getPrecisionModel() {

```

```

44     return PRECISION_MODEL;
45 }
46
47 /**
48  * @return the same instance set coordinate but made precise
49  */
50 @NotNull
51 public static Coordinate makePrecise(@NotNull Coordinate coordinate) {
52     getPrecisionModel().makePrecise(coordinate);
53     return coordinate;
54 }
55
56 /**
57  * @return the same instance set coordinate but made precise
58  */
59 public static double makePrecise(double value) {
60     return getPrecisionModel().makePrecise(value);
61 }
62
63 /**
64  * @return a buffer from given {@link Geometry} and a certain distance
65  */
66 @NotNull
67 public static Polygon createBuffer(@NotNull Geometry geom, double distance, boolean useSquareEndCap) {
68     if (useSquareEndCap) {
69         return (Polygon) geom.buffer(distance, BufferParameters.DEFAULT_QUADRANT_SEGMENTS, BufferParameters.CAP_FLAT);
70     }
71     return (Polygon) geom.buffer(distance);
72 }
73
74 /**
75  * Creates a {@link Coordinate} and makes the {@link Coordinate} precise using {@link #getPrecisionModel()}.
76  * @return a {@link Coordinate} set given x/y value pair
77  */
78 @NotNull
79 public static Coordinate createCoordinate(double x, double y) {
80     return makePrecise(new Coordinate(x, y));
81 }
82
83 /**
84  * Creates a new {@link Coordinate} with the copy constructor ({@link Coordinate#Coordinate(Coordinate)}) and
85  * makes the {@link Coordinate} precise using {@link #getPrecisionModel()}.
86  * @return a new instance set {@link Coordinate}
87  */
88 @Nullable
89 @Contract("null->null")
90 public static Coordinate createCoordinate(@Nullable Coordinate coordinate) {
91     if (coordinate == null) {
92         return null;
93     }
94     // use copy-constructor in order to modify and return a new instance
95     return makePrecise(new Coordinate(coordinate));
96 }
97
98 /**
99  * @return an empty {@link Polygon}
100  */
101 @NotNull
102 public static Polygon createEmptyPolygon() {
103     return GEOMETRY_FACTORY.createPolygon((Coordinate[]) null);
104 }
105
106 /**
107  * @return a new Point at the position of the given {@link Point} moved with given vector
108  */
109 @NotNull
110 public static Point createMovePoint(@NotNull Point point, @NotNull MoveVector moveVector) {
111     return GeomUtil.createPoint(point.getX() + moveVector.getX(), point.getY() + moveVector.getY());
112 }
113
114 /**
115  * @return a polygon with given centroid, width and height
116  */
117 @NotNull
118 public static Polygon createPolygon(@NotNull Point centroid, double width, double height) {
119     Coordinate a = createCoordinate(centroid.getX() - width / 2, centroid.getY() - height / 2);
120     Coordinate b = createCoordinate(centroid.getX() - width / 2, centroid.getY() + height / 2);
121     Coordinate c = createCoordinate(centroid.getX() + width / 2, centroid.getY() + height / 2);
122     Coordinate d = createCoordinate(centroid.getX() + width / 2, centroid.getY() - height / 2);
123     return GEOMETRY_FACTORY.createPolygon(new Coordinate[]{a, b, c, d, a});

```

```

124 }
125
126 public static Polygon createPolygon(@NotNull NodePoint... points) {
127     Coordinate[] coordinates = Stream.of(points)
128         .map(NodePoint::toCoordinate)
129         .toArray(Coordinate[]::new);
130     return GEOMETRY_FACTORY.createPolygon(coordinates);
131 }
132
133 public static Polygon createPolygon(@NotNull Point... points) {
134     Coordinate[] coordinates = Stream.of(points)
135         .map(Point::getCoordinate)
136         .toArray(Coordinate[]::new);
137     return GEOMETRY_FACTORY.createPolygon(coordinates);
138 }
139
140 /**
141  * @return a point with given x- and y-values
142  */
143 @NotNull
144 public static Point createPoint(double x, double y) {
145     return GEOMETRY_FACTORY.createPoint(createCoordinate(x, y));
146 }
147
148 /**
149  * @return a point with given coordinate
150  */
151 @NotNull
152 public static Point createPoint(@NotNull Coordinate coordinate) {
153     return GEOMETRY_FACTORY.createPoint(createCoordinate(coordinate));
154 }
155
156 /**
157  * @return a new point from {@link Geometry#getCoordinate()}
158  */
159 @NotNull
160 public static Point createPoint(@NotNull Geometry geometry) {
161     return GEOMETRY_FACTORY.createPoint(makePrecise(geometry.getCoordinate()));
162 }
163
164 /**
165  * @return a new instance set {@link Point} with x- and y-values set given {@link Point}
166  */
167 @NotNull
168 public static Point clonePoint(@NotNull Point point) {
169     return createPoint(point.getX(), point.getY());
170 }
171
172 /**
173  * @return an instance set {@link GeometryCollection} with geometries provided by given {@link Stream}
174  */
175 @NotNull
176 public static GeometryCollection createCollection(@NotNull Stream<? extends Geometry> stream) {
177     return GEOMETRY_FACTORY.createGeometryCollection(stream.toArray(Geometry[]::new));
178 }
179
180 /**
181  * @return an instance set {@link GeometryCollection} with values set given {@link Collection}s
182  */
183 @NotNull
184 @SafeVarargs
185 public static GeometryCollection createCollection(@NotNull Collection<? extends Geometry>... collections) {
186     Collection<Geometry> merged = Stream.of(collections)
187         .flatMap(Collection::stream)
188         .collect(Collectors.toSet());
189     return GEOMETRY_FACTORY.createGeometryCollection(merged.toArray(new Geometry[0]));
190 }
191
192 /**
193  * @return a {@link Stream} set {@link Geometry} from given {@link GeometryCollection}
194  */
195 @NotNull
196 public static Stream<Geometry> toStream(@NotNull GeometryCollection collection) {
197     List<Geometry> output = new ArrayList<>();
198     for (int i = 0; i < collection.getNumGeometries(); i++) {
199         output.add(collection.getGeometryN(i));
200     }
201     return output.stream();
202 }
203

```

```

204  /**
205   * @return a {@link LineString} with given x/y value pairs
206   */
207   @NotNull
208   public static LineString createLineString(double x1, double y1, double x2, double y2) {
209       return GEOMETRY_FACTORY.createLineString(new Coordinate[]{createCoordinate(x1, y1), createCoordinate(x2, y2)});
210   }
211
212   @NotNull
213   public static LineString createLineString(@Nullable Coordinate... points) {
214       if (points == null) {
215           return GEOMETRY_FACTORY.createLineString((Coordinate[]) null);
216       }
217       return GEOMETRY_FACTORY.createLineString(Stream.of(points)
218           .map(GeomUtil::createCoordinate)
219           .toArray(Coordinate[]::new));
220   }
221
222   @NotNull
223   public static LineString createLineString(@NotNull Point pointA, @NotNull Point pointB) {
224       return createLineString(pointA.getCoordinate(), pointB.getCoordinate());
225   }
226
227   @NotNull
228   public static LineString createLineString(@NotNull NodePoint nodeA, @NotNull NodePoint nodeB) {
229       return createLineString(nodeA.toCoordinate(), nodeB.toCoordinate());
230   }
231
232   /**
233   * Returns the angle between two adjacent lines connecting p1 and p2 with circleCenterPoint.
234   */
235   public static double getAngleBetween(@NotNull NodePoint circleCenterPoint, @NotNull NodePoint p1, @NotNull NodePoint p2) {
236       double angle1 = Math.atan2(p1.getY() - circleCenterPoint.getY(), p1.getX() - circleCenterPoint.getX());
237       double angle2 = Math.atan2(p2.getY() - circleCenterPoint.getY(), p2.getX() - circleCenterPoint.getX());
238       return angle1 - angle2;
239   }
240
241   public static double getAngleBetweenAsDegree(@NotNull NodePoint circleCenterPoint, @NotNull NodePoint p1, @NotNull NodePoint p2) {
242       double angle = getAngleBetween(circleCenterPoint, p1, p2);
243       return Math.toDegrees(angle);
244   }
245
246   }

```

## PolygonUtil.java

```

1  /**
2   * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3   */
4
5  package ch.geomo.util.geom;
6
7  import com.vividsolutions.jts.geom.*;
8  import com.vividsolutions.jts.geom.util.AffineTransformation;
9  import com.vividsolutions.jts.geom.util.LineStringExtractor;
10 import com.vividsolutions.jts.operation.polygonize.Polygonizer;
11 import org.jetbrains.annotations.NotNull;
12
13 import java.util.Arrays;
14 import java.util.Collection;
15 import java.util.List;
16 import java.util.Optional;
17 import java.util.stream.Stream;
18
19 public enum PolygonUtil {
20
21     /** util class */;
22
23     /**
24     * Returns a {@link Stream} set {@link LineString}s which are parallel to given {@link LineString}
25     * and within given {@link Polygon} while one endpoint is equal to a vertex and the other end
26     * point of the line string lies on the exterior set the polygon.
27     */
28     @NotNull
29     public static Stream<LineString> findParallelLineString(@NotNull Polygon inPolygon, @NotNull LineString parallelTo) {
30
31         Envelope envelope = inPolygon.getEnvelopeInternal();
32
33         // scale line string order to be long enough to intersect with the polygons exterior

```

```

34     double factor = Math.max(envelope.getHeight(), envelope.getWidth()) * 2;
35     AffineTransformation scaleTransformation = new AffineTransformation();
36     scaleTransformation.scale(factor, factor);
37     LineString scaledLineString = GeomUtil.createLineString(parallelTo.getStartPoint(),
38         GeomUtil.createPoint(scaleTransformation.transform(parallelTo.getEndPoint())));
39
40     return Arrays.stream(inPolygon.getCoordinates())
41         .sequential()
42         // create a parallel line for each vertex
43         .map(vertex -> {
44             AffineTransformation translateTransformation = new AffineTransformation();
45             translateTransformation.translate(vertex.x - scaledLineString.getCentroid().getX(), vertex.y -
46                 scaledLineString.getCentroid().getY());
47             return translateTransformation.transform(scaledLineString);
48         })
49         .filter(Geometry::isValid)
50         // get line string within polygon
51         .filter(inPolygon::intersects)
52         .map(inPolygon::intersection)
53         .filter(geom -> geom instanceof LineString && !geom.isEmpty())
54         .map(geom -> (LineString) geom);
55
56     }
57
58     /**
59     * Finds the longest parallel {@link LineString} parallel to the given {@link LineString} and within
60     * the given {@link Polygon}.
61     */
62     @NotNull
63     public static Optional<LineString> findLongestParallelLineString(@NotNull Polygon inPolygon, @NotNull LineString parallelTo) {
64         return PolygonUtil.findParallelLineString(inPolygon, parallelTo)
65             .max((l1, l2) -> Double.compare(l1.getLength(), l2.getLength()));
66     }
67
68     /**
69     * Creates polygons from given {@link LineString} or {@link GeometryCollection} set {@link LineString}.
70     * <p>
71     * Note: Returns an empty {@link GeometryCollection} if {@link Geometry} is not a {@link LineString}
72     * or a {@link GeometryCollection} set {@link LineString}.
73     * @return a {@link GeometryCollection} set polygons
74     * @see <a href="http://suite.opengeo.org/ee/docs/4.5/processing/wpsjava/index.html">OpenGeo Suite Enterprise Docs</a>
75     * @see <a href="http://gis.stackexchange.com/a/190002/21355">JTS: split arbitrary polygon by a line (stackoverflow.com)</a>
76     */
77     @NotNull
78     public static GeometryCollection polygonize(@NotNull Geometry geometry) {
79         @SuppressWarnings("unchecked")
80         List<LineString> lines = LineStringExtractor.getLines(geometry);
81         if (lines.isEmpty()) {
82             return GeomUtil.createCollection();
83         }
84         Polygonizer polygonizer = new Polygonizer();
85         polygonizer.add(lines);
86         @SuppressWarnings("unchecked")
87         Collection<Polygon> polygonCollection = polygonizer.getPolygons();
88         return GeomUtil.createCollection(polygonCollection);
89     }
90
91     /**
92     * Splits given polygon with given {@link LineString} or collection set {@link LineString}. Returns
93     * an empty {@link GeometryCollection}.
94     * @return a {@link GeometryCollection} set polygons
95     * @see <a href="http://suite.opengeo.org/ee/docs/4.5/processing/wpsjava/index.html">OpenGeo Suite Enterprise Docs</a>
96     * @see <a href="http://gis.stackexchange.com/a/190002/21355">JTS: split arbitrary polygon by a line (stackoverflow.com)</a>
97     */
98     @NotNull
99     public static GeometryCollection splitPolygon(@NotNull Geometry polygon, @NotNull Geometry lineStrings) {
100         GeometryCollection lines;
101         if (lineStrings instanceof GeometryCollection) {
102             Stream<Geometry> boundaryStream = Stream.of(polygon.getBoundary());
103             Stream<Geometry> lineStringStream = GeomUtil.toStream((GeometryCollection) lineStrings);
104             lines = GeomUtil.createCollection(Stream.concat(boundaryStream, lineStringStream));
105         }
106         else {
107             lines = (GeometryCollection) polygon.getBoundary().union(lineStrings);
108         }
109         Stream<Polygon> stream = GeomUtil.toStream(polygonize(lines))
110             .map(geom -> (Polygon) geom)
111             // polygons which are inside the polygon
112             .filter(p -> polygon.contains(p.getInteriorPoint()));
113         return GeomUtil.createCollection(stream);

```

```

112     }
113
114 }

```

## NodePoint.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.util.geom.point;
6
7  import ch.geomo.util.geom.GeomUtil;
8  import com.vividsolutions.jts.geom.Coordinate;
9  import com.vividsolutions.jts.geom.Point;
10 import org.jetbrains.annotations.Contract;
11 import org.jetbrains.annotations.NotNull;
12 import org.jetbrains.annotations.Nullable;
13
14 public interface NodePoint {
15
16     /**
17      * @return x-value
18      */
19     double getX();
20
21     /**
22      * @return y-value
23      */
24     double getY();
25
26     /**
27      * @return a <b>new</b> instance set {@link Coordinate}
28      */
29     @NotNull
30     Coordinate toCoordinate();
31
32     /**
33      * @return a <b>new</b> instance set {@link Point}
34      */
35     @NotNull
36     Point toPoint();
37
38     /**
39      * @return distance between this point and the other given point
40      */
41     default double calculateDistanceTo(@NotNull NodePoint point) {
42         return calculateDistanceTo(point.getX(), point.getY());
43     }
44
45     /**
46      * @return distance between this point and the other given coordinate
47      */
48     default double calculateDistanceTo(double x, double y) {
49         return calculateDistanceTo(new Coordinate(x, y));
50     }
51
52     /**
53      * @return distance between this point and the other given coordinate
54      */
55     default double calculateDistanceTo(@NotNull Coordinate coordinate) {
56         return toCoordinate().distance(coordinate);
57     }
58
59     /**
60      * @return an {@link ImmutableNodePoint} with given x- and y-values
61      */
62     @NotNull
63     static NodePoint of(double x, double y) {
64         return new ImmutableNodePoint(x, y);
65     }
66
67     /**
68      * @return an {@link ImmutableNodePoint} with the x- and y-values set given {@link Point}
69      */
70     @Nullable
71     @Contract("null->null")
72     static NodePoint of(@Nullable Point point) {
73         if (point == null) {

```

```

74         return null;
75     }
76     return NodePoint.of(point.getX(), point.getY());
77 }
78
79 static double calculateAngle(NodePoint nodeA, NodePoint nodeB) {
80     double angle = GeomUtil.getAngleBetweenAsDegree(nodeA, NodePoint.of(nodeA.getX(), nodeA.getY() + 5d), nodeB);
81     if (angle < 0) {
82         angle = (angle + 360) % 360;
83     }
84     return Math.abs(angle);
85 }
86
87 }

```

### ImmutableNodePoint.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.util.geom.point;
6
7  /**
8   * Immutable implementation set {@link NodePoint}.
9   */
10 public class ImmutableNodePoint extends AbstractNodePoint {
11
12     public ImmutableNodePoint(double x, double y) {
13         super(x, y);
14     }
15
16 }

```

### AbstractNodePoint.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.util.geom.point;
6
7  import ch.geomo.util.geom.GeomUtil;
8  import com.vividsolutions.jts.geom.Coordinate;
9  import com.vividsolutions.jts.geom.Point;
10 import org.jetbrains.annotations.NotNull;
11
12 import java.util.Objects;
13
14 public abstract class AbstractNodePoint implements NodePoint {
15
16     protected double x;
17     protected double y;
18
19     public AbstractNodePoint(double x, double y) {
20         this.x = x;
21         this.y = y;
22     }
23
24     @Override
25     public double getX() {
26         return x;
27     }
28
29     @Override
30     public double getY() {
31         return y;
32     }
33
34     @NotNull
35     @Override
36     public Coordinate toCoordinate() {
37         return GeomUtil.createCoordinate(x, y);
38     }
39
40     @NotNull
41     @Override
42     public Point toPoint() {

```

```

43     return GeomUtil.createPoint(x, y);
44 }
45
46 @Override
47 public boolean equals(Object obj) {
48     return obj instanceof NodePoint
49         && getY() == ((NodePoint) obj).getY()
50         && getX() == ((NodePoint) obj).getX();
51 }
52
53 @Override
54 public int hashCode() {
55     return Objects.hash(getX(), getY());
56 }
57
58 @Override
59 public String toString() {
60     return getX() + "/" + getY();
61 }
62
63 }

```

## MutableNodePoint.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.util.geom.point;
6
7  /**
8   * Mutable implementation set {@link NodePoint}.
9   */
10 public class MutableNodePoint extends AbstractNodePoint {
11
12     public MutableNodePoint(double x, double y) {
13         super(x, y);
14     }
15
16     public MutableNodePoint(NodePoint point) {
17         super(point.getX(), point.getY());
18     }
19
20     public void setX(double x) {
21         super.x = x;
22     }
23
24     public void setY(double y) {
25         super.y = y;
26     }
27
28     public void moveX(int moveDistance) {
29         setX(getX() + moveDistance);
30     }
31
32     public void moveY(int moveDistance) {
33         setY(getY() + moveDistance);
34     }
35
36 }

```

## NodePointDistanceComparator.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.util.geom.point;
6
7  import org.jetbrains.annotations.NotNull;
8
9  import java.util.Comparator;
10
11 /**
12 * Compares two {@link NodePoint} to their distance to an origin {@link NodePoint}. Closest point first,
13 * farthest point last.
14 */
15 public class NodePointDistanceComparator<P extends NodePoint> implements Comparator<P> {

```



```

16
17     private final static Comparator<NodePoint> XY_COMPARATOR = new NodePointXYComparator<>();
18
19     private final NodePoint originPoint;
20
21     public NodePointDistanceComparator() {
22         originPoint = NodePoint.of(0, 0);
23     }
24
25     public NodePointDistanceComparator(@NotNull P originPoint) {
26         this.originPoint = originPoint;
27     }
28
29     /**
30      * Returns a positive int value if first point's distance to the origin point is less
31      * than the distance set the second point to the origin point. Otherwise a negative
32      * int value will be returned. If both points are equals, first point is considered
33      * closer to the origin point.
34      */
35     @Override
36     @SuppressWarnings("unchecked") // -> safe raw type usage
37     public int compare(P p1, P p2) {
38         return Double.compare(p1.calculateDistanceTo(originPoint), p2.calculateDistanceTo(originPoint));
39     }
40
41 }

```

## NodePointXYComparator.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.util.geom.point;
6
7  import java.util.Comparator;
8
9  /**
10   * Sorts instances set {@link NodePoint}s along x/y axis, while x is higher weighted than y. Returns 1 when x set o1 is
11   * greater than x set o2 as well as when both x are equals but y set o1 is greater.
12   */
13   public class NodePointXYComparator<N extends NodePoint> implements Comparator<N> {
14
15       @Override
16       public int compare(N o1, N o2) {
17           if (o1.equals(o2)) {
18               return 0;
19           }
20           if (o1.getX() > o2.getX() || (o1.getX() == o2.getX() && o1.getY() > o2.getY())) {
21               return 1;
22           }
23           return -1;
24       }
25
26   }

```

## D.6.3 Package ch.geomo.util.logging

### Loggers.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.util.logging;
6
7  import org.jetbrains.annotations.NotNull;
8
9  import java.io.IOException;
10 import java.text.MessageFormat;
11 import java.util.HashMap;
12 import java.util.Map;
13 import java.util.logging.LogManager;
14 import java.util.logging.Logger;
15
16 public enum Loggers {
17

```

```

18  /* util class */;
19
20  private static final Map<Class, Logger> cache = new HashMap<>();
21
22  static {
23      // currently not working standalone -> log file won't be found when running standalone
24      System.setProperty("java.util.logging.config.file", "./src/main/resources/logging.properties");
25      try {
26          LogManager.getLogManager().readConfiguration();
27      }
28      catch (IOException e) {
29          Logger.getLogger().severe("Cannot read Logger configuration!");
30      }
31  }
32
33  @NotNull
34  public static Logger get(@NotNull Object obj) {
35      return getLogger(obj);
36  }
37
38  public static void flag(@NotNull Object obj, @NotNull String message) {
39      get(obj).info(" * " + message);
40  }
41
42  public static void flag(@NotNull Object obj, @NotNull String message, @NotNull Object... params) {
43      flag(obj, MessageFormat.format(message, (Object[]) params));
44  }
45
46  public static void separator(@NotNull Object obj) {
47      info(obj, "=====");
48  }
49
50  public static void debug(@NotNull Object obj, @NotNull String message) {
51      get(obj).finest(" " + message);
52  }
53
54  public static void info(@NotNull Object obj, @NotNull String message, char markCharacter) {
55      get(obj).info(" " + markCharacter + " " + message);
56  }
57
58  public static void info(@NotNull Object obj, @NotNull String message) {
59      info(obj, message, 'i');
60  }
61
62  public static void info(@NotNull Object obj, @NotNull String message, @NotNull Object... params) {
63      info(obj, MessageFormat.format(message, (Object[]) params), 'i');
64  }
65
66  public static void warning(@NotNull Object obj, @NotNull String message) {
67      // get(obj).warning(" ! " + message);
68  }
69
70  public static void warning(@NotNull Object obj, @NotNull String message, @NotNull Object... params) {
71      get(obj).warning(" ! " + MessageFormat.format(message, (Object[]) params));
72  }
73
74  public static void error(@NotNull Object obj, @NotNull String message) {
75      get(obj).severe(" E " + message);
76  }
77
78  public static void error(@NotNull Object obj, @NotNull String message, @NotNull Object... params) {
79      get(obj).severe(" E " + MessageFormat.format(message, (Object[]) params));
80  }
81
82  @NotNull
83  public static Logger get(@NotNull Class objClass) {
84      return getLogger(objClass);
85  }
86
87  @NotNull
88  public static Logger getLogger(@NotNull Object obj) {
89      if (obj instanceof Class) {
90          return get(obj);
91      }
92      return get(obj.getClass());
93  }
94
95  @NotNull
96  public static Logger getLogger(@NotNull Class objClass) {
97      Logger logger = cache.get(objClass);

```

```

98     if (logger == null) {
99         logger = Logger.getLogger(objClass.getSimpleName());
100        cache.put(objClass, logger);
101    }
102    return logger;
103 }
104 }
105 }

```

## D.6.4 Package ch.geomo.util.math

### MoveVector.java

```

1  /*
2  * Copyright (c) 2016 Thomas Zuberbuehler. All rights reserved.
3  */
4
5  package ch.geomo.util.math;
6
7  import ch.geomo.util.collection.pair.Pair;
8  import ch.geomo.util.geom.GeomUtil;
9  import com.vividsolutions.jts.geom.Coordinate;
10 import com.vividsolutions.jts.geom.LineString;
11 import com.vividsolutions.jts.geom.Point;
12 import com.vividsolutions.jts.math.Vector2D;
13 import org.jetbrains.annotations.NotNull;
14
15 /**
16 * An implementation set {@link Vector2D} providing constructor to create a vector from a {@link LineString} as well
17 * as getter-methods returning x- and y-values made precise using {@link GeomUtil#getPrecisionModel()}.
18 * @see Vector2D
19 */
20 public class MoveVector extends Vector2D {
21
22     public static final Vector2D VECTOR_ALONG_X_AXIS = new Vector2D(1, 0);
23     public static final Vector2D VECTOR_ALONG_Y_AXIS = new Vector2D(0, 1);
24
25     public MoveVector() {
26         super(0, 0);
27     }
28
29     public MoveVector(double x, double y) {
30         super(x, y);
31     }
32
33     @SuppressWarnings("unused")
34     public MoveVector(@NotNull Point from, @NotNull Point to) {
35         this(from.getCoordinate(), to.getCoordinate());
36     }
37
38     public MoveVector(@NotNull Coordinate from, @NotNull Coordinate to) {
39         super(from, to);
40     }
41
42     public MoveVector(@NotNull Vector2D vector) {
43         super(vector);
44     }
45
46     public MoveVector(@NotNull LineString lineString) {
47         super(lineString.getStartPoint().getCoordinate(), lineString.getEndPoint().getCoordinate());
48     }
49
50     /**
51 * @return the x-value made precise using {@link GeomUtil#getPrecisionModel()}
52 */
53     @Override
54     public double getX() {
55         return GeomUtil.getPrecisionModel().makePrecise(super.getX());
56     }
57
58     /**
59 * @return the y-value made precise using {@link GeomUtil#getPrecisionModel()}
60 */
61     @Override
62     public double getY() {
63         return GeomUtil.getPrecisionModel().makePrecise(super.getY());
64     }
65 }

```

```
66  /**
67   * @return the projection this vector along given vector
68   */
69   @NotNull
70   public Pair<MoveVector> getProjection(@NotNull Vector2D alongVector) {
71       return getProjection(this, alongVector);
72   }
73
74   @Override
75   public String toString() {
76       return "MoveVector: {x= " + getX() + ", y=" + getY() + "}";
77   }
78
79   /**
80   * @return the projection set this vector along given vector
81   */
82   @NotNull
83   public static Pair<MoveVector> getProjection(@NotNull Vector2D vector, @NotNull Vector2D alongVector) {
84       MoveVector projection = new MoveVector(alongVector.multiply(vector.dot(alongVector) / alongVector.dot(alongVector)));
85       MoveVector rejection = new MoveVector(vector.subtract(projection));
86       return Pair.of(projection, rejection);
87   }
88
89 }
```