Enhancing Geospatial Data for Passenger Transport Systems

Dissertation

for the award of the degree

“Doctor rerum naturalium” (Dr.rer.nat)

of the Georg-August Universtität Göttingen

within the doctoral programm *Environmental Informatics*

of the Georg-August Universitiy School of Science (GAUSS) submitted by

Armin Hahn

from Göttingen

Göttingen, 2021

Thesis Commitee

- Prof. Dr. Martin Kappas
  Georg-August-Universität, Göttingen
  Geographisches Institut - Abteilung Kartographie, GIS und Fernerkundung
- Prof. Dr. Stephan Herminghaus
  Max-Planck-Institut für Dynamik und Selbstorganisation, Göttingen
  Dynamik komplexer Fluide
- Prof. Dr.-Ing. Marcus Baum
  Georg-August-Universität, Göttingen
  Institut für Informatik - Abteilung Data Fusion

Members of the Examination Board

- Reviewer:
  Prof. Dr. Martin Kappas
  Georg-August-Universität, Göttingen
  Geographisches Institut - Abteilung Kartographie, GIS und Fernerkundung
- Second Reviewer:
  Prof. Dr. Stephan Herminghaus
  Max-Planck-Institut für Dynamik und Selbstorganisation, Göttingen
  Dynamik komplexer Fluide

Further members of the Examination Board

- Prof. Dr.-Ing. Marcus Baum
  Georg-August-Universität, Göttingen
  Institut für Informatik - Abteilung Data Fusion

- Prof. Dr. Winfried Kurth
  Georg-August-Universität, Göttingen
  Institut für Informatik - Abteilung Ökoinformatik, Biometrie und Waldwachstum

- Prof. Dr.-Ing. Bernd Stock
  HAWK - Hochschule für angewandte Wissenschaft und Kunst, Göttingen
  Fakultät für Ingenieurwissenschaften

- Dr. Daniel Wyss
  Georg-August-Universität, Göttingen
  Geographisches Institut - Abteilung Kartographie, GIS und Fernerkundung

Date of the oral examination: 20.12.2021

Acknowledgements

I would like to thank everyone who accompanied and supported me during my Ph.D. I would especially like to acknowledge my supervisors Prof. Dr. Martin Kappas, Prof. Dr. Stephan Herminghaus and Prof. Dr.-Ing. Marcus Baum for their assistance at every stage of the research project. I would also like to express my gratitude to the examination board for their time and feedback. Thanks to the Max Planck Institute for Dynamics and Self-Organization, especially to the whole EcoBus team, with whom I was able to gain a lot of valuable experience in the pilot projects. I would also like to thank the Leipziger Verkehrsbetriebe and my colleagues there, as this meant that I never lost touch with the practical side of passenger transportation. I very much appreciated the interdisciplinary work and the exchange between different kinds of researchers such as physicists, computer scientists, geographers, traffic economists and economists. Furthermore, I would like to especially thank my colleagues in the office for their help, support and sometimes for pleasant distractions to rest my mind outside of my research. In particular, I would like to thank Wiard Frühling for his cooperation, discussions, patience and support. Finally, my appreciation also goes out to my friends and family.

Abstract

This thesis presents and evaluates new solutions and developed software frameworks to challenges from a geoinformatics perspective that have resulted from participation in demand responsive transport (DRT) projects. Thereby, the focus is on geospatial data, such as the road network or general map data, which are used for routing in the context of passenger transportation systems.

Such DRT systems have the potential to reduce various detrimental effects, such as the consumption of finite resources (e.g. fossil fuels for the inefficient motorized private transport), environmental pollution (e.g. air pollution by particulate matter), or congestion at peak times in urban regions through more efficient mobility offers, and can thus simultaneously contribute to mitigating the anthropogenic climate change.

Within the scope of this work, the focus is set on two main points. We especially concentrate on the aspect of how an enhancement of geospatial data could contribute to improvements for passenger transport systems. The interdisciplinarity of geographers was also used in this work to combine the fields of mobility research, geoinformatics, computer science, and mathematics (graph theory), and thus to consider problems across disciplines.

First, a performant approach for determining network distances was developed that could be an alternative to the usage of Euclidean distance in transportation services and transportation research. Even nowadays, the Euclidean distance is often used to determine the distance between two points on the road network to avoid a computationally costly calculation of exact network distances. However, the use of Euclidean distance can lead to inaccurate distances, if the actual path on the road network is a much larger detour than the beeline. A common example for this problem involves rivers, where a small Euclidean distance may be calculated between a point on one side of a river to a point on the other side of the river, but if there is no bridge in the immediate vicinity, a much larger detour, and thus a much larger travel time, must be taken than calculated by the Euclidean distance. Another example where such problems occur more frequently is road networks with many oneways. However, these problems can occur anywhere. We present an approach, which provides approximated network distances. This can be useful if exact network distances are not required, e.g. for rough estimations or preselections in ride-pooling scenarios, to evaluate whether two requests can possibly be pooled. Calculating the exact network distances with routing engines (shortest path algorithms) can be very calculation-intensive, especially for many parallel (and iterative) calculations on large networks. Since a precalculation of all shortest path distances would be a reasonable solution but is not practical for large networks, we partition the road network and determine proxies for each partition. Proxies then represent the area for the respective partition. The size of the partition and hence the acceptable deviation (inaccuracy or the degree of generalization of the road network) can be set by a parameter. Based on the proxies, a complete graph is created, which data can be stored in a lookup table and network distances can be read easily. Thus, the performance for identifying network distances depends on the search algorithm, which scales linearly in the worst case, regardless of the network size, whereas conventional shortest path algorithms scale worse on large networks. In the evaluation, this approach showed potential use for the future.

Second, this work also deals with the problem of so-called road snapping. This is the determination of stop locations at the start and end of a calculated route between two addresses. Road snapping thus describes the process of determining reference points on the road network for given start and destination points that are not located directly on the road network. Conventional routing engines use the perpendicular distance for the determination, which can lead to insufficient calculated snapping points, respectively stop locations, since the actual access to buildings is not taken into account. In the context of supported DRT projects, insufficient stop locations can not only be dangerous pick-up locations on highly trafficked highways but also can lead to delays in the time schedule, because bus drivers need to find a suitable spot for the boarding of passengers. Such time delays can interfere with future trips and the whole time schedule. We developed an alternative approach that uses remote sensing and the cost distance analysis method to determine the most likely access to buildings and thus more reasonable stop locations. Therefore, the assumption was made, that the access to buildings consists of few vegetation cover, minimal slope of the terrain and the calculated path should not cross building footprints. For this approach, open source data were used and the parameters for the cost distance analysis were determined by using a vegetation index, a high-resolution elevation model using light detection and ranging (LiDAR) data, and building footprints from OpenStreetMap. Thus, the so-called least cost path can be calculated, which reflects the most likely path from a building to the road network. Accordingly, optimized snapping points, respectively stop locations have been determined, that consider the actual access to the building, which conventional approaches do not consider. For the evaluation, the used parameters were weighted differently, which allowed determining a suitable weight combination of these parameters. Furthermore, the results were compared and validated with a conventional routing engine, which uses the perpendicular distance. The presented approach achieves depending on the weight combinations a validation-rate up to 90.3%, whereas the routing engine only achieves a validation-rate of 81.4%. These results show that the presented approach could be used in the future to precompute optimized snapping points, thus avoiding misunderstandings, delays, and dangerous pick-up and drop-off locations in the context of passenger transportation systems with a to-door service.

Zusammenfassung

In dieser Dissertation werden neue Lösungen und entwickelte Software-Frameworks für Herausforderungen aus Sicht der Geoinformatik vorgestellt und bewertet, die sich aus der Teilnahme an demand responsive transport (DRT)-Projekten ergeben haben. Dabei liegt der Fokus auf Geodaten, wie dem Straßennetz oder allgemeinen Kartendaten, die für ein Routing im Kontext von Personenbeförderungssystemen genutzt werden.

Solche DRT Systeme haben das Potenzial durch effizientere Mobilitätsangebote verschiedene Entwicklungen, wie den Verbrauch von endlichen Ressourcen (z.B. den Verbrauch von fossilen Treibstoffen für den meistens ineffizienten motorisierten Individualverkehr), Umweltverschmutzungen (z.B. Luftverschmutzung durch Feinstaub) oder Staus zu Stoßzeiten in urbanen Regionen zu reduzieren und können somit zeitgleich einen Beitrag zur Bekämpfung des anthropogenen Klimawandels leisten.

Im Rahmen dieser Arbeit werden zwei Hauptschwerpunkte untersucht. Dabei wird insbesondere der übergeordnete Aspekt betrachtet, wie eine verbesserte Nutzung von Geodaten dazu beitragen kann, moderne Transportsysteme attraktiver zu gestalten. Die Interdisziplinarität der Geographie biete dabei den Vorteil, die Bereiche Mobilitätsforschung, Geoinformatik, Informatik und Mathematik (Graphentheorie) miteinander zu verbinden und somit disziplinübergreifende Probleme holistisch betrachten und lösen zu können.

Es wurde ein Ansatz zur performanten Bestimmung von Netzwerkdistanzen entwickelt, der die Nutzung der Euklidischen Distanz im Transportbereich und in der Mobilitätsforschung ersetzen könnte. Auch heutzutage wird noch die Euklidische Distanz zur Bestimmung der Distanz zwischen zwei Punkten im Straßennetz verwendet, um so eine rechenintensive Berechnung von exakten Netzwerkdistanzen zu vermeiden. Die Verwendung der Euklidischen Distanz kann jedoch zu sehr ungenauen Distanzen führen, wenn der tatsächliche Weg auf dem Straßennetz ein viel größerer Umweg als die Euklidische Distanz (Luftlinie) ist. Ein Beispiel für dieses Problem kann bei Flüssen auftreten, bei denen zwar eine geringe Euklidische Distanz zwischen einem Punkt auf der einen Seite des Flusses und einem Punkt auf der anderen Seite des Flusses ermittelt werden kann, aber es keine Brücke in unmittelbare Nähe gibt und somit ein größerer Umweg und

eine größere Fahrzeit in Kauf genommen werden muss, als ursprünglich anhand der Euklidischen Distanz ermittelt wurde. Ein weiteres Beispiel wo solche Probleme häufiger auftreten sind Straßennetze mit vielen Einbahnstraßen. Allgemein können diese Probleme jedoch überall auftreten. Der in dieser Dissertation neu entwickelte Ansatz liefert angenäherte Netzwerkdistanzen, ist aber weniger rechenintensiv als bisherige Algorithmen. Das kann nützlich sein, wenn keine exakten Netzwerkdistanzen benötigt werden, aber die Problematik bzw. die Ungenauigkeit der Euklidischen Distanz in einigen Fällen vermieden werden soll. Beispielsweise kann diese Ansatz für eine grobe Abschätzung oder Vorauswahl für die Berechnung von möglichen Fahrgemeinschaften Anwendung finden, wenn überprüft werden soll, ob zwei Reisewünsche für eine Fahrgemeinschaft berücksichtigt werden sollen oder nicht. Eine Berechnung der exakten Netzwerkdistanzen mit Routenplanern (basierend auf Kürzeste-Wege-Algorithmen) kann sehr rechenintensiv sein, insbesondere wenn viele Berechnungen parallel und iterativ für große Straßennetze durchgeführt werden. Eine Vorberechnung aller kürzesten Pfade ist zwar möglich und würde das Problem der benötigten Rechenleistung umgehen, jedoch ist das besonders bei großen Netzwerkgraphen nicht praktikabel. Daher wird in dem hier vorgestellten Ansatz das Straßennetz partitioniert und für jede Partition wird ein sogenannter Proxy definiert, der den Bereich seiner Partition repräsentiert. Die Größe der Partitionen und damit auch die akzeptierte Ungenauigkeit bzw. der Grad der Generalisierung des Straßennetzes kann anhand eines Parameters bestimmt werden. Mithilfe der Proxies wird ein vollständiger Graph erstellt, dessen Daten in einer Lookup-Tabelle gespeichert werden und mit einem Suchalgorithmus können dann dort Netzwerkdistanzen aller kürzesten Wege ausgelesen werden. Die Performance dieses Ansatzes hängt dabei von dem Suchalgorithmus für die Lookup-Tabelle ab, der im schlechtesten Fall linear mit der Netzwerkgröße skaliert, während die meisten herkömmlichen Algorithmen zur Bestimmung von Netzwerkdistanzen mit großen Netzwerken schlechter skalieren. In der Auswertung zeigte dieser Ansatz Potenzial für eine zukünftige Anwendung.

Im Rahmen dieser Arbeit wurde auch das für DRT Projekte wichtige Problem des sogenannten road snappings bearbeitet. Dabei geht es um die Bestimmung von Haltepunkten am Anfang und Ende einer berechneten Route für zwei Adressen. Road snapping beschreibt also den Prozess zur Bestimmung von Referenzpunkten auf dem Straßennetz für Start- und Zielpunkte, die nicht direkt auf dem Straßennetz liegen. Herkömmliche Routenplaner nutzen für die Bestimmung die perpendikulare Distanz, wodurch es zu ungenügenden Haltepunkten kommen kann, da der tatsächliche Zugang zu den Adressen bzw. Gebäuden nicht berücksichtig wird. Im Kontext der begleiteten DRT Projekte bedeuteten ungenügende Haltepunkte zum Beispiel nicht nur gefährliche Abholorte an stark befahrenen Bundesstraßen, sondern auch Zeitverzögerungen, weil Busfahrer einen geeigneten Haltpunkt oder den Fahrgast suchen mussten, da die Fahrgäste einen

anderen Haltepunkt erwarteten. Dieses Problem tritt besonders bei Mobilitätsangeboten auf, die zusätzlich auch Buchungen via Callcenter ermöglichen, sodass kein Abholort auf einer Karte angezeigt werden kann. Die dadurch entstehenden Zeitverzögerungen können dazu führen, dass darauffolgende Fahrten nicht nach Zeitplan stattfinden und sich Verzögerungen kaskadisch immer weiter vergrößern können. Es wurde eine Alternative entwickelt, die anhand von Fernerkundung und der Methode der Kostendistanz-Analyse den wahrscheinlichsten Zugang zu Gebäuden berechnet und somit sinnvolle Haltepunkte bestimmt. Dafür wurde die Annahme getroffen, dass der Zugang zu Gebäuden nur eine geringe Vegetationsbedeckung und eine minimale Steigung des Geländes aufweist sowie der ermittelte Pfad von der Straße zur berücksichtigten Adresse nicht durch andere Gebäude führt. Für die Bestimmung sogenannter günstigster Kostenpfade durch die Methode der Kostendistanz-Analyse wurden aus Open Source Daten folgende Parameter bestimmt: Die Vegetationsbedeckung (anhand eines Vegetationsindexes), die Steigungen eines hochauflösenden Geländemodells (anhand von light detection and ranging (LiDAR) Daten) sowie die Grundrisse der Gebäude (von OpenStreetMap). Die ermittelten Pfade repräsentieren den wahrscheinlichsten Weg von einem Gebäude zum Straßennetz. Durch den Schnittpunkt dieser Pfade mit dem Straßennetz sind somit optimierte Haltepunkte bestimmt worden, die den Weg zum Eingang von Gebäuden berücksichtigen. Für die Evaluation wurden die verwendeten Parameter in verschiedene Iterationen unterschiedlich gewichtet, wodurch als Resultat sinnvolle Gewichtungskombinationen der Parameter ermittelt werden konnten. Weiterhin wurden die Ergebnisse mit einem herkömmlichen Routenplaner, der die perpendikulare Distanz verwendet, verglichen und validiert. Die Haltepunkte von dem vorgestellten Ansatz erreichten je nach verwendeter Gewichtung eine Validierungs-Rate von bis zu 90.3%, wohingegen der Routenplaner nur eine Validierungs-Rate von 81.4% erreichte. Die Ergebnisse zeigen, dass der vorgestellte Ansatz zukünftig genutzt werden kann, um optimierte Haltepunkte vorzuberechnen, wodurch Missverstädnisse, Verzögerungen und gefährliche Haltepunkte im Rahmen von Personenbeförderungssystemen mit einem Tür-zu-Tür Angebot vermieden werden können.

Contents

List of Figures

List of Tables

Acronyms

ACSA Accumulative Cost Surface Analysis.

AOI area of interest.

API application programming interface.

APSP all pairs shortest path.

BFS breadth-first search.

CIR color infrared.

CRP customizable route planning.

DARP dial-a-ride problem.

DPS distance-preserving subgraphs.

DRAs deterministic routing areas.

DRRP demand responsive ride pooling.

DRT demand responsive transport.

FC-GBOP FluidC-Generalization based on Proxies.

FMLM first mile/last mile.

GHG greenhouse gas.

GIS geographic information system.

GPS global positioning system.

IPCC Intergovernmental Panel on Climate Change.

IQR interquartile range.

kNN k nearest neighbor.

LiDAR light detection and ranging.

LVB Leipziger Vekehrsbetriebe.

MGP multilevel graph partitioning.

MPT motorized private transport.

MSP mobility service providers.

NDVI normalized difference vegetation index.

NIR near-infrared.

NRW North Rhine-Westphalia.

OSRM Open Source Routing Machine.

PCD precomputed cluster distances.

TNC transport network company.

VRP vehicle routing problem.

1. Introduction

Traffic and logistics are a major cause of environmental damage due to its emissions (e.g. carbon dioxides, particular matter) and its inefficient usage of resources in motorized private transport (MPT) [1, 2]. According to the Intergovernmental Panel on Climate Change (IPCC), reducing the greenhouse gas (GHG) emissions will be challenging if the increasing trend of emissions from passenger and freight transport cannot be stopped, since they could outweigh all mitigation measures [3]. Further, it is predicted that without aggressive and sustained applied mitigation policies, the transport emissions could be 12 Gt CO_2 eq/year by 2050 (for 2019 the total emissions for Europe accounted 5.57 Gt [4]). The European Parliament state that transport causes nearly 30% of the total CO_2 emissions of Europe, where road transportation alone causes 72% of these emissions [5].

The increasing demand for mobility in the last decades led to congestion in urban areas. Simultaneously, insufficient public transport is a growing challenge in rural areas. Thus, more people either depend on MPT or they move to urban areas, which will further increase the high traffic volume. Hence it is important to optimize mobility for the future. Recently, there has been a growing interest in so-called demand responsive transport (DRT) systems, demand responsive ride pooling (DRRP) or dynamic and flexible ride-sharing [6, 7, 1]. Such systems can help to optimize the mobility and accordingly the current transport systems because they use resources for mobility more efficiently by combining similar trips (ride-matching problem) [8, 1].

These transport systems can complement or replace inefficient static schedules of tram or bus lines. This means they can act as a feeder or distributor for public transportation, which is also known as the First Mile / Last Mile problem. Consequently, public transport in combination with DRT could compete better with MPT. Otherwise, in some cases, DRT systems can also be seen as an attractive alternative to MPT without public transport. Overall, such systems have to be comfortable, fast, and reliable to be part of the mobility in the future [6].

Even though the concept of ride-sharing is not new, more and more mobility projects with demand driven concepts emerged in the last decade. This is due to a growing interest in optimizing mobility with more technical capabilities (smartphones, computing power) and digitalization (e.g. online payment systems) [1, 2, 6]. For instance, in the past, the planning of trips for similar dial-a-ride systems was often carried out by hand using Microsoft Excel or Outlook [9]. However, technology and digitalization can be used to automate such processes. Users can book trips via smartphones and the pooling of transportation requests can be performed by advanced pooling algorithms and more available computing power [10].

1.1. Motivation

The challenges related to mobility are very complex. On the one hand, urbanization and the associated higher demand for mobility in confined spaces such as cities, lead to congestion and overloaded public transport. On the other hand, rural exodus also plays a role, since in rural areas the economic viability of public transport is often problematic and, if the demand decreased, the offered services by public transport will decrease too. This makes rural areas even less attractive without MPT and creates a feedback mechanism of urbanization.

Especially the aspects of environmental pollution and inefficient use of finite resources for the MPT indicate how important a change in mobility is. In order to make the mobility of the future sustainable and more efficient, this must be seen as a holistic problem that requires interdisciplinary solution strategies, which includes for example the following research areas: *vehicle routing problem*, enhancing single and multicriteria routing and its performance (finding the shortest or fastest path in a complex network), thus additionally *complex networks* and *graph theory* should also be considered. Furthermore, an enhanced use of *geospatial data* is also a part that should be considered, since the information about road closures (constructions) or traffic jams must be managed and maintained for a routing engine. Also, geospatial data such as the road network (e.g. OpenStreetMap), addresses, and house numbers as well as the access to buildings play an important role for accurate and comfortable routing. In the future, this could be especially important for autonomous driving, when no human driver can compensate for erroneous or inaccurate routing.

Geographers are predestined for such interdisciplinary challenges. In this thesis, the experiences of the following supported pilots of DRT projects have been included:

1. Ecobus: Phase 1 - area of interest in a rural area (small scale)
2. Ecobus: Phase 2 - area of interest in a rural area (medium scale)

3. Flexa - area of interest in an urban area with the focus on feeding and supporting intermodal transportation

The pilots from Ecobus[1] were a funded research project carried out by the Max Planck Institute for Dynamics and Self-Organization (Dynamics of Complex Fluids). The Flexa[2] project is part of the offered services by the transport company Leipziger Vekehrsbetriebe (LVB).

1.2. Mobility Concepts

There are a variety of different modern, flexible transport systems, all of which basically pursue the same goal: they want to offer a more resource-efficient alternative to the conventional motorized private transport (MPT). In addition, most of the systems operate to replace fixed routes of bus lines, which are not efficient outside peak hours [9].

Some of the typical names of such modern, flexible transportation systems are demand responsive transport (DRT) systems, demand responsive ride pooling (DRRP), ride-pooling, or ride-sharing. Whereby some already make distinctions here. Aydin, Gokasar, and Kalan [6] state that the difference between so-called dial-a-ride problem (DARP) programs would be the driver supply, which means that in DARP, the drivers are provided by a company, whereas drivers in ride-sharing systems are independent entities. In this thesis, we do not use such a distinction between passenger transport systems, as long as they have a similar goal and hence have similar requirements regarding technology and information management (e.g. geospatial data and routing for to-door services).

Nevertheless, it is important to distinguish some modern systems from others, as they lead to different developments. Such demand-oriented transport systems, especially the ones in cooperation with public transport or as a part of public transport, can actually achieve a more resource efficient transportation, as it strengthens the public transport. In particular, such systems can act as a so-called feeder for public transport or they can take over the first mile/last mile (FMLM). In the case of similar mobility services such as Uber[3], Lyft[4], Grab[5] or Moia[6] [8, 1] etc., there is a risk that they will act as competitors to public transport in urban areas and that potential customers who have already used public transport will switch to such services and not, as is necessary, users from MPT. Therefore, it is reasonable that such mobility services should be provided

1 https://www.ecobus.jetzt/home.html
2 https://www.l.de/verkehrsbetriebe/kundenservice/services/flexa
3 www.uber.com
4 www.lyft.com
5 www.grab.com
6 www.moia.io

by transport network company (TNC) or mobility service providers (MSP) or at least should be included as cooperation partners so that public transport is strengthened by such mobility concepts.

For further literature, we refer to Masoud and Jayakrishnan [1] who present a comparison of different mobility concepts, to Jittrapirom *et al.* [11] with an overview of further concepts and their aims, as well as to Böhler [12], even if it is not completely up to date, but they provide a handbook for planning flexible forms of service in public transport in Germany.

1.3. Main Contribution

Due to the opportunity to participate in supported DRT projects, it was possible to identify potential enhancements for passenger transportation systems that geoinformatics can provide. In this thesis, theoretical and real operational problems in passenger transportation are considered, and moreover, implementations for some selected optimization potentials are presented. The focus of the described challenges and optimization potentials is on how geodata can be used and improved for this purpose in the field of passenger transportation.

We neither focus on theoretical algorithms for the vehicle routing problem (VRP) nor on algorithms for pooling nor on matching for dynamic and flexible mobility systems. For this, we refer to the comprehensive overview from Masoud and Jayakrishnan [1].

1.4. Outline

In the second chapter examples of challenges in supported DRT projects are presented. They are categorized in challenges based on performance issues due to the complexity of road networks and computations for such networks and in challenges based on inaccurate stop locations for to-door transportation. In the third chapter, we present basic concepts of graph theory, routing techniques, approximation algorithms, map matching, and cost distance analysis. In the following chapter, related work is introduced. Chapters five and six revisit the categorized challenges from chapter two and present some solution strategies and concepts for them. In chapter seven, the main results from chapters five and six are discussed and considered in terms of how geospatial data can be enhanced in the context of passenger transportation, and the methodologies are compared to other approaches from the literature. Further, the potentials and future work are described before in chapter eight, the results of this thesis are concluded.

2. Challenges in Modern Mobility Concepts

In this chapter, challenges that occurred in supported DRT projects are presented. The projects were introduced in section 1.1.

These challenges include performance issues for the determination of network distances in the context of modern passenger transport systems, as well as the impact of erroneous or incomplete map data on routing and in particular on so-called stop locations. For the incomplete map data, OpenStreetMap is mainly used as an example, since both free and commercial routing engines often use data from OpenStreetMap.

2.1. Performance of Network Distance Computations

Even if navigation and routing is an everyday task, there is still a need for improvements [13], especially if navigation and routing systems are used for autonomous cars or for very specific and reliable to-door routing, when the driver does not have enough local knowledge like professional taxi drivers. Further, most popular online maps or routing engines are used to compute single criterion queries. In practice, however, queries based on multiple criteria are more useful, such as the shortest or fastest route, while trying to avoid tolls or congested roads [14]. For more and more upcoming ride-sharing services and DRT systems, this becomes more relevant, since drivers for such transport services must rely on accurate routing in general and on accurately calculated stop locations.

Euclidean distances are widely used in transportation practice and transportation research as a measurement between two points on the road network, due to historic difficulties in calculating network distances and due to the assumption, that the ratio between Euclidean distance and network distance on a homogeneous network tend to be constant [15]. However, only the grid-like Manhattan road pattern can be seen as a homogeneous network, but this assumption can not be applied to road networks in general. It is arguable that Euclidean distance is sufficient for approximate estimations of network distance when a small circuity value is present. This value describes the ratio of the distance on the road network to the Euclidean distance [16]. Still, the probability of miscalculations and hence delays decrease with a smaller circuity value but doesn't prevent errors due to the Euclidean distance approach. Therefore, Shang *et al.* [17] recommend using the real distance between two objects on the road network rather than the Euclidean distance.

Nevertheless, Euclidean distance is still used in current mobility and transportation research. For example, Czioska *et al.* [18] use the Euclidean distance to cluster customers into temporary and spatially similar groups for evaluating the feasibility of shared rides. Another example of Euclidean distance used in transportation and transportation research is in modern mobility services such as Uber[1]. Therefore, it is interesting to find the k nearest neighbor (kNN). Shen *et al.* [19] describe that existing studies focused on kNN for moving objects are still based on Euclidean distance constraints.

Figure 2.1 shows an extreme example of the difference between Euclidean distance and network distance. Assuming an identical speed, the time delay between the calculated time to get from the origin to the destination by Euclidean distance, compared to the actual network distance, is in this case about factor 20.

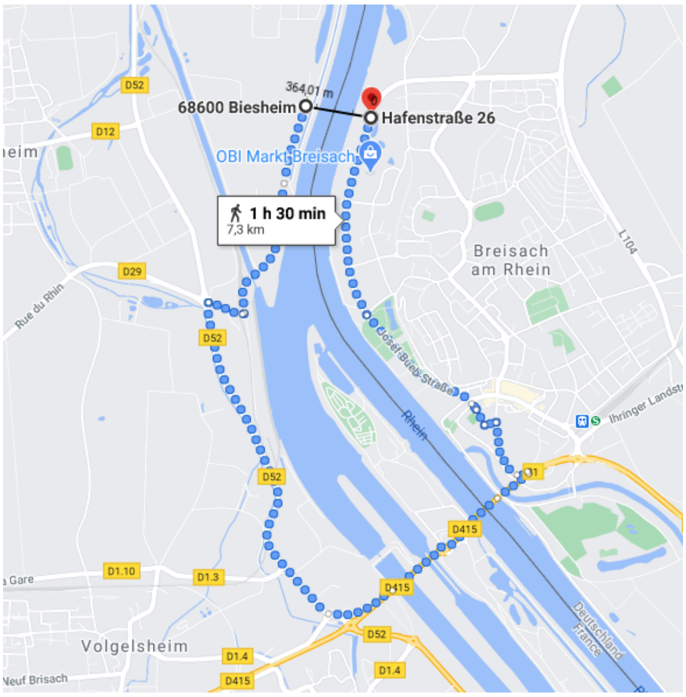


Figure 2.1.: Difference between network distance and Euclidean distance. Assumptions in transport planning based on Euclidean distance can lead to miscalculations for distance and time. The difference of distance or travel time, assuming the same speed, is about the factor 20.

Figure 2.2 depicts the use of Euclidean distance for a concept of a demand responsive transport (DRT) system from Masoud and Jayakrishnan [1], precisely the concept for filtering suitable stops for ride pooling. The origin and destination of a predetermined trip

[1] https://www.uber.com/

are $f1$ and $f2$, the numbered nodes are possible stops. An ellipsoid is used to determine the stops that are candidates for an acceptable detour, e.g., potential stops that can be combined with the predetermined trip between $f1$ and $f2$. This ellipsoid is based on Euclidean distance, e.g., an acceptable spatial or temporal detour, but it does not take into account the actual distance on the road network. Since buffers, e.g. for time windows, are used, the ellipsoid is not symmetric with respect to the origin and the destination. The plane of the ellipsoid can describe time or space (spatio-temporal), but is only an approximation to reduce the number of stops to be considered. It can occur that stops are theoretically reachable, hence are within the ellipsoid, but are not reachable in practice due to the difference between edge costs (network distance) and Euclidean distance.

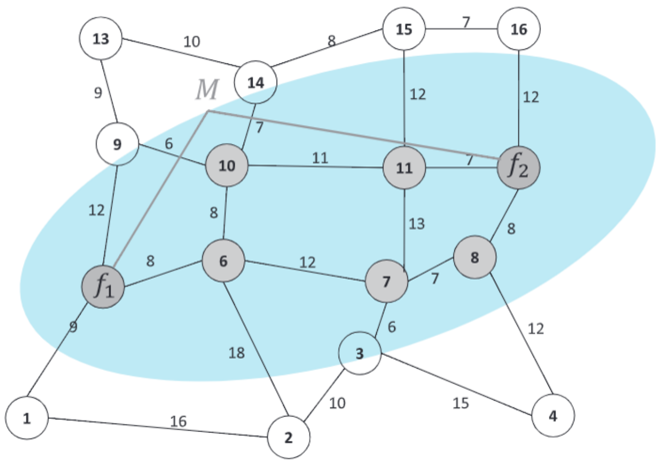


Figure 2.2.: A network graph showing eligible stops for ride pooling with enumerated stops as nodes and edge weights between nodes. A trip between $f1$ and $f2$ is already given. For this given trip combinable stops, hence possible detours are determined using an ellipsoid based on Euclidean distance. Thus, the number of eligible stops is reduced, since only stops within this ellipsoid are considered close enough for a detour to them without violating constraints such as the arrival time window for the original trip. However, the actual distance on the road network is not considered [1].

The computational complexity behind DRT systems can be easily underestimated. The performance of calculations can be a limiting factor for the prevalence of such systems, especially on a large scale [1, 20]. Such calculations include calculations of combinable trips, but also the reiteration and evaluation process for changing conditions due to new incoming trips or delays in intermodal trip planning. To combine two trips, many critical calculations and constraints have to be done and checked, e.g. the distance of the detour or the additional travel time for a passenger [21]. To find the optimal combination of trips, these calculations have to be done as fast as possible to check given constraints and compare different combinations of similar travel requests. These calculations can demand significant computing power when dealing with a large number of requests and routes.

In the literature, the problem of performance of such calculations, especially for shortest paths or network distance calculations, is treated differently.

Wang *et al.* [14] describe that multicriteria exact shortest path queries are proven to be NP-hard [22, 23] and thus the most existing algorithms are approximative solutions [23, 24, 25] (*cf.* section 3.3) which use a parameter to limit the acceptable range of the results as a constraint. They point out that for large road networks the existing methods are still too expensive.

Maue, Sanders, and Matijevic [26] describe that an extreme way to accelerate shortest path queries for static transportation networks is to precompute all shortest path distances. Yet this is not practical for large networks since it requires quadratic space and preprocessing time. For small networks, this can be a feasible solution, especially if possible origins and destinations are limited. Then, a precalculated distance matrix, which is often provided by modern routing engines, can be sufficient. If the possible origins and destinations are not limited to a number of addresses and the network gets larger, the distance matrix can get unnecessarily complex for certain applications.

There are many routing techniques and optimizations to calculate the shortest path (A*, bidirectional Dijkstra, contraction hierarchies), hence the network distance between two points. In section 3.2 the main concepts of such routing techniques are introduced. Nevertheless, these algorithms scale with the size of the network (run time $\approx O(n^2)$, *cf.* section 3.3). For this reason, we will focus on generalizing the road network to reduce the complexity of the network graph and hence the complexity for calculations such as network distances queries. This enables a combination of a reduced network graph and optimized routing algorithms for further performance improvements. We assume that approximated network distances can be useful and sufficient for some purposes in passenger transportation, such as reducing the number of potential stops in the ride-pooling process of DRT systems. Instead of using Euclidean distance as Masoud and Jayakrishnan [1], it may be sufficient to use the approximated network distances derived from a generalized road network. Even if exact network distances can not be obtained from a generalized road network by adjusting the degree of generalization, an acceptable inaccuracy of the network distances and a concurrent performance improvement can be achieved.

2.2. Optimized Pick-up and Drop-off Locations in to-Door Services

The following section refers especially to the application of routing for to-door services without limited origin and destinations (addresses), such as those given for bus lines.

Snapping or road snapping describes the assignment of a single coordinate or an address to a reference point, a so-called snapping point on the road network as a start or end point of a route. Road snapping is thematically related to so-called map matching, which methodology is explained in section 3.4.

So far, road snapping in most conventional routing engines is based on perpendicular distance, the shortest distance between a point and a line, hence the shortest distance between an address or a coordinate and a segment of the road network. To avoid inaccurate or misleading snapping points, fixed stops or bus stations have been mostly used so far in transportation services besides taxis. In to-door transportation, the calculated snapping points were less relevant in transportation research and passenger transportation, since to-door transportation was mostly performed by cab drivers (taxis) with local knowledge. Another approach that can be used for modern, flexible demand transport systems, is described by Czioska *et al.* [18], who determine efficient meeting points. They state that DRT systems mostly operate on a to-door policy. Instead of using real to-door services, they determine meeting points for similar requests. This would offer several benefits, such as fewer stops and less traveled kilometers, but customers have to accept a walk to meeting points [18]. The method of this approach can be described as follows. The meeting points are determined in three steps: First, the customers are clustered into temporary and spatially similar groups. Second, meeting points for boarding (pick-up) and alighting (drop-off) are calculated for each cluster. Third, a neighborhood search algorithm is used to obtain the vehicle routes, that pass through all the calculated meeting points while respecting requirements such as the passengers' time constraints. This approach is one way to avoid the issue with insufficient stop locations and miscommunications of pick-up locations but requires acceptance of longer walks to the meeting points, which does not comply with the requirements of to-door services and transportation of elderly, hampered, or disabled people. Consequently, optimal snapping points for to-door services become more relevant for transportation services and transportation research.

Typical snapping problems can occur for large building complexes with several entrances, such as hospitals or university campuses, buildings directly located on intersections, or buildings between two parallel roads with an identical name. There are some commercial services that try to solve the problem like what3words[2] or Google Plus Codes[3]. However, they only offer the possibility to determine different building entrances with shorter coordinates by users, but not to determine meaningful snapping points or calculating them automatically. Nevertheless, in most conventional routing engines snapping problems still occur. Figure 2.3 depicts an example snapping point located on a highway, where it is dangerous and most likely not possible to pick-up or drop-off passengers. In this figure,

[2] `https://what3words.com`
[3] `https://grid.plus.codes/`

the routing from Google Maps shows the access to the destination with the blue dashed line and the reference, hence snapping point, directly on a highway. The more suitable and realistic approach to the building is shown with an orange dashed line, where several parking spots are available.

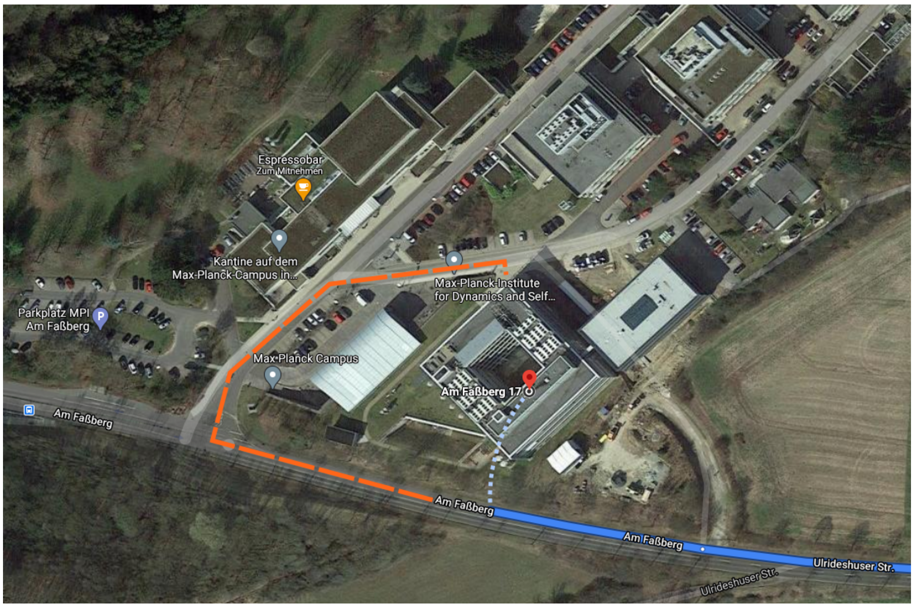


Figure 2.3.: Road snapping based on perpendicular distance from Google Maps shows an insufficient snapping point without direct access to the building. The orange dashed line shows the correct access to the building. The dotted line depicts the access to the building by Google Maps [27].

We can assume that Google Maps uses an enhanced technique for road snapping, that uses additionally to the perpendicular distance a matching of names with the given address and surrounding road names. However, this technique is still not sufficient as shown in Figure 2.4. In this figure, the snapping point is not located on the road in the southeast, even if the shortest perpendicular distance would lead to a snapping point on this road. Instead, the road northwest is used as a reference for the given address, since the road name and the address have a matching name. The actual access to the building is depicted with the orange dashed line.

Even if such snapping problems don't occur often, it shows that state-of-the-art routing engines like Google Maps yet have problems with accurate snapping points. For a reliable and comfortable to-door mobility service, such problems should be avoided.

Another, more theoretical problem that has not been encountered in the pilot projects, but may occur when multiple reasonable reference points for passenger boarding are available on the road network. For this theoretical showcase, we do not care about the perpendicular distance. As an example, Figure 2.5 shows three possible locations for the boarding of passengers. The yellow line represents the most reasonable stop location, but depending on the direction and the destination of the route the other options

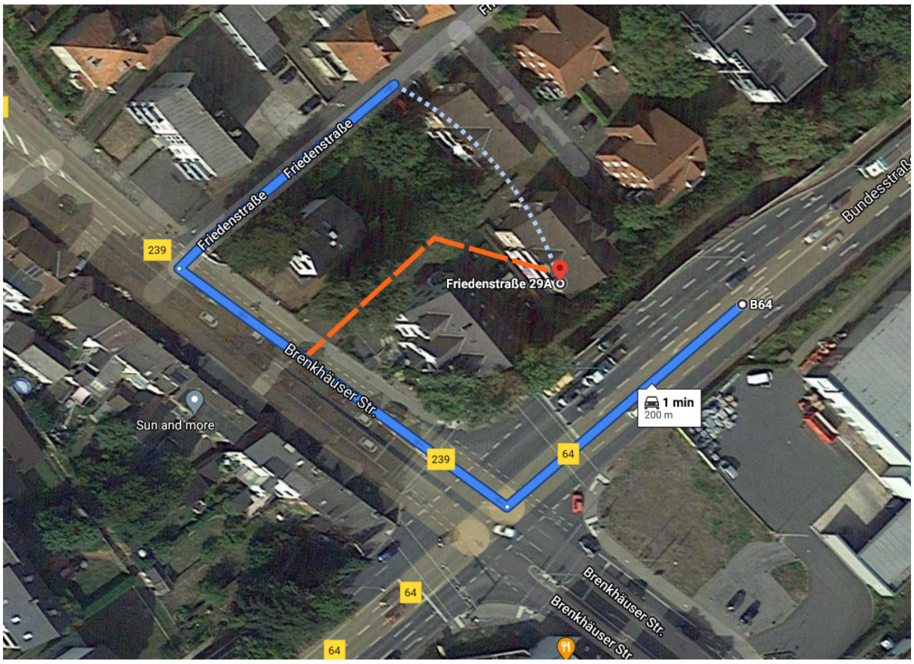


Figure 2.4.: Problems in current road snapping by perpendicular distance. Even if a matching name of the road and the given address is used as an additional feature, the actual access to the building (orange dashed line) differs from the result from Google Maps. The dotted line depicts the access to the building by Google Maps [28]. In the Appendix A (Figure A.5) is a true color image with a similar extent and a higher resolution depicted.

can be preferable for boarding. The quality of map data will also influence the choice of boarding locations by routing engines. This can be especially relevant when using OpenStreetMap data. In OpenStreetMap properties, so-called tags such as “private” or “service”, are assigned to features (e.g. roads, buildings, areas). Map data with these tags are then ignored by routing engines to avoid routing on private properties. Due to a community-driven validation of the map data, it can happen that some assigned properties are not consistent or even wrong and consequently, the quality of the snapping by routing engines is influenced. In Figure 2.5, the parking lot (yellow line) could theoretically be assigned with the property “private”, making the other stop locations more reasonable.

Supplementary to the snapping problems, missing map data also lead to challenges in supported pilot projects. Figure 2.6 shows an example of a missing road, which leads also to problematic snapping. The road segment highlighted with pink dots was missing in this case but it could also have a wrong property (e.g. private road) and then it will not be considered for snapping in most routing engines. This leads to a snapping point north of the river, because the blue line represents the shortest distance to the next road segment, while the acceptable alternative (red line) is longer and is thus not considered as a snapping point.

Another showcase is depicted in Figure 2.7. The service roads to the buildings were missing, which led to a snapping point on the highway (Bundestrasse B64).

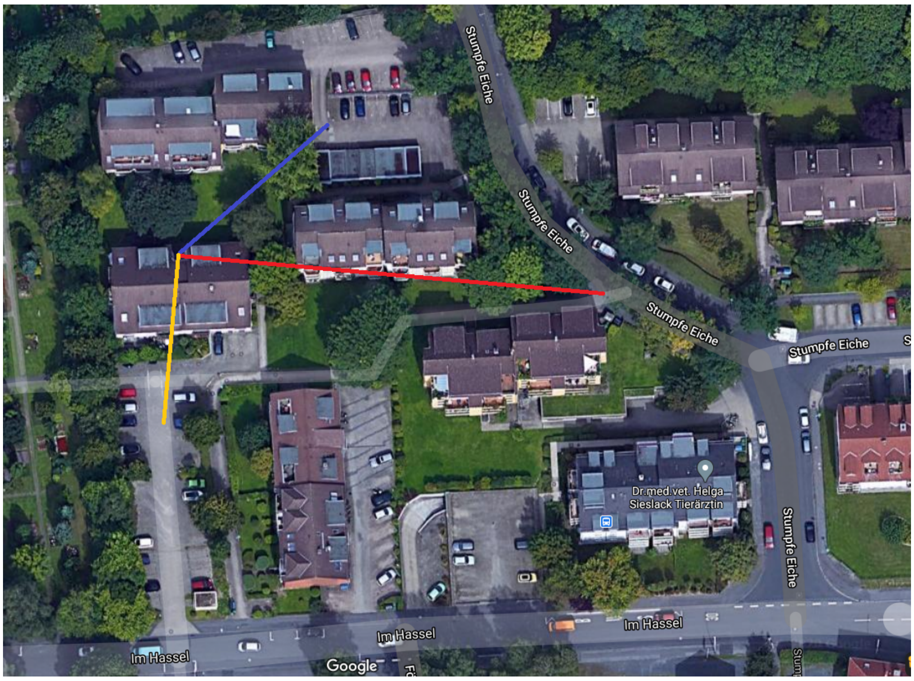


Figure 2.5.: Three possible stop locations for a building that can be reasonable for passenger boarding, depending on the given route, its driving direction, and the quality of the map data.

These examples show that an accurate determination of snapping points becomes more relevant for upcoming to-door services besides taxis. The communication of stop locations for boarding passengers is also crucial. With modern smartphones and apps, this can be done by highlighting the calculated pick-up location on maps. In the supported pilot projects in rural areas, we encountered the challenge, that the mobility demand of older people without smartphones must also be met. This is why booking via a call center was also made possible as part of the pilot. Here, verbal communication of exact pick-up locations becomes a serious challenge.

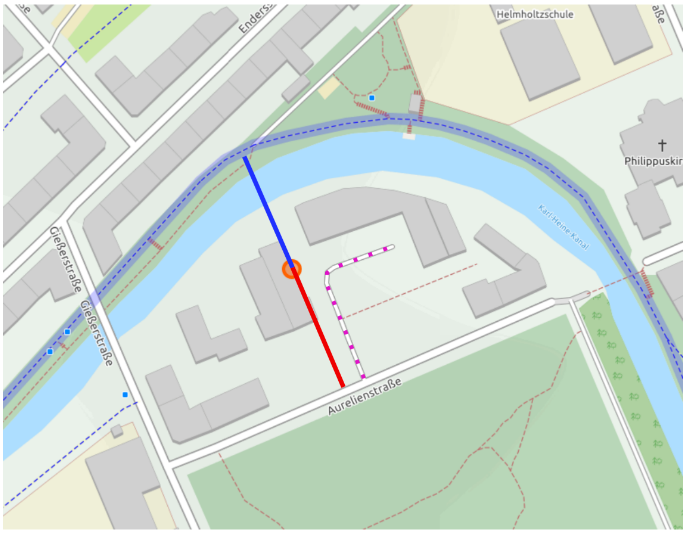


Figure 2.6.: Effects of missing map data on road snapping. The road segment highlighted with pink dots was missing or can be theoretically tagged with wrong information, which leads to a snapping point north of the river. This is due to the shorter distance to the next road segment, represented by the blue line and the acceptable alternative (red line) is not considered due to the larger distance. Based on the missing map data or inaccuracies, detours and miscommunication for pick-up and drop-off locations can occur. This visualization is based on the exact results of routing engines using OpenStreetMap data.

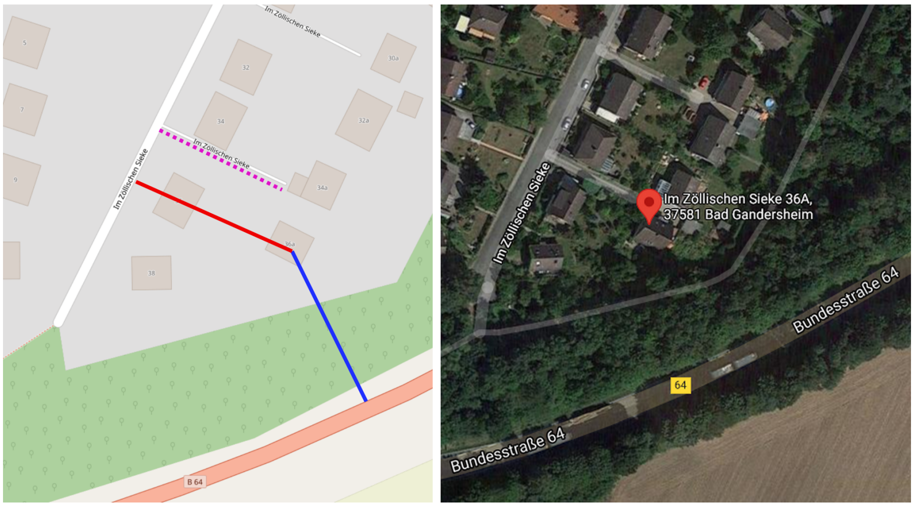


Figure 2.7.: Snapping problem caused by missing map data. The road segment highlighted with pink dots was missing in OpenStreetMap, which results in a reference to the road network represented by the blue line. Thus, a snapping point on a highway was used, since the red line represents a larger distance than the blue one. On the right is a similar extent of the region with a satellite image from Google Maps to give a realistic impression of that area. This visualization is based on the exact results of routing engines using OpenStreetMap data.

3. Preliminaries

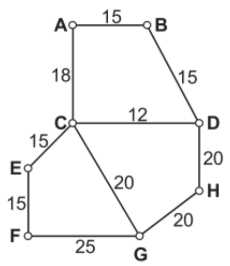
3.1. Graph Theory

Graph theory is a branch of mathematics. The origin of this subfield can be traced back to Leonhard Euler (1707-1783) and his solution for the problem of the seven bridges of Königsberg. He was asked to find a route or circuit over seven bridges in Königsberg, with the condition that each bridge should be crossed only once. He was able to prove by means of a graph, that such a circuit is not possible [29]. A graph $G = (V, E)$ consists of a finite set of nodes $V = \{V_1, V_2, V_3, \ldots, V_n\}$ and edges E, defined by pairs of nodes $E_1 = \{V_i, V_j\}$. Edges can be assigned arbitrary data, such as distance, velocity, or other properties of the node pair's relationships, which are then referred to as edge weights. In the literature, the terms nodes, vertex, or vertices are often used synonymously. Edges are also referred to as segments or arcs. By using graphs, relations can be represented as (complex) networks and mathematical calculations and analyses can be performed for such networks. Besides the classical fields of mathematics and computer science (e.g. networks of communication or parallel computing), graph theory is also used in chemistry (isomorphism of molecules), biology (spread of diseases and parasites), neurosciences (networks of nerves), and in social sciences (social networks and relationships) [30, 31]. In the context of this thesis, graph theory is used for cartographic purposes, since a road network (topological network) can also be viewed as a graph. Edges represent road segments and nodes represent intersections or the start and end points of the segments. Graph theory has already been used in numerous cartographic studies for the generalization of road networks [32, 33, 34, 35, 36, 37, 38, 39].

The graph theory allows a variety of different calculations and analyses such as connectivity analyses or the so-called traveling salesman problem. The traveling salesman problem is a classical problem of combinatorial optimization, where a sequence of nodes is searched that covers all nodes of the graph and visits all nodes except the starting point exactly once with minimal edge weights (e.g. distance) [40]. In the context of road networks, graph theory also involves the *Dijkstra's algorithm*, which is an important basis for routing problems [41]. This algorithm computes within a graph the shortest or most favorable path in terms of edge weights from a starting node to a destination node. This algorithm and other routing techniques are introduced in section 3.2.

Further terms, which need a closer terminological consideration, are *adjacency* and *incidence*. They describe the relations of objects to each other in a graph. Adjacency characterizes elements of the same type (two nodes) in a graph that are adjacent, hence direct neighbors. Incidence, on the other hand, is characterized by two elements of different types being adjacent (e.g. an edge and a node).

The data of the graph can be stored in a so-called adjacency matrix or alternatively in multiple lists. Thereby, for each node, a list with all the adjacent nodes and the weights is stored. Figure 3.1 depicts an example of a weighted graph and the corresponding adjacency matrix as well as the alternative storage in multiple lists. Lange [40] states that in practice, the storage method of multiple lists is mostly used since it requires less space than the adjacency matrix.



	A	B	C	D	E	F	G	H
A	0	15	18	∞	∞	∞	∞	∞
B	15	0	∞	15	∞	∞	∞	∞
C	18	∞	0	12	15	∞	20	∞
D	∞	15	12	0	∞	∞	∞	20
E	∞	∞	15	∞	0	15	∞	∞
F	∞	∞	∞	∞	15	0	25	∞
G	∞	∞	20	∞	∞	25	0	20
H	∞	∞	∞	20	∞	∞	20	0

A 15 → B 18 → C
B 15 → A 15 → D
C 18 → A 12 → D 15 → E 20 → G
D 15 → B 12 → C 20 → H
E 15 → C 15 → F
F 15 → E 25 → G
G 20 → F 20 → C 20 → H
H 20 → G 20 → D

Figure 3.1.: Visualization of a weighted graph (left), the corresponding adjacency matrix (middle), and the alternative storage in multiple lists (right)[40].

Network graphs can be basically divided into digraphs, multidigraphs, and weighted graphs. Digraphs are characterized by the fact, that edges are directed. This means they have an assigned direction, such as oneway roads. Multidigraphs have the additional property, that between two nodes, multiple edges can exist. A weighted graph describes the property of weighted edges in a graph. Thereby arbitrary data can be stored as weights. For road networks, the distance, travel time (speed limit), or general properties of the roads are stored, such as the name or the condition of the road. Further examples of edge properties or weights from practice are documented by OpenStreetMap [42, 43]. Consequently, road networks are mostly categorized as weighted multidigraphs.

The *degree* a node is also called valency [37], gives the number of adjacent nodes $D(V_n)$. In Figure 3.2, the node V_7 has a degree of 5 because it is directly connected or adjacent to 5 different nodes. A special case for the degree of a node arises for loops as in V_{11}. For a loop in an undirected graph, a degree of 2 is calculated, because each outgoing edge is interpreted as a neighboring node. Thus, if no digraph is given where the loop has the attribute "oneway", then one outgoing edge is counted for each direction. With respect to Figure 3.2, this leads to $D(V_{11}) = 5$. The node V_7 has a central role because removing this node would result in two separate graphs. These graphs are then called *subgraphs*. If removing of a node creates subgraphs or disconnected edges, then such a node is called *articulation vertex* [35]. If subgraphs are created by removing edges, these edges are called *disconnected set*. In Figure 3.2 the disconnected set would consist of the edges E_6 and E_7. Such nodes and edges should ideally be identified before generalization so that they are not removed during the generalization process. Thus, isolated nodes and unwanted disconnected subgraphs can be prevented. A further terminological concept is the so-called *dead-end*. Dead-ends can be either be roads with an end, but also be connection points to the road network outside of the selected extract of a road network.

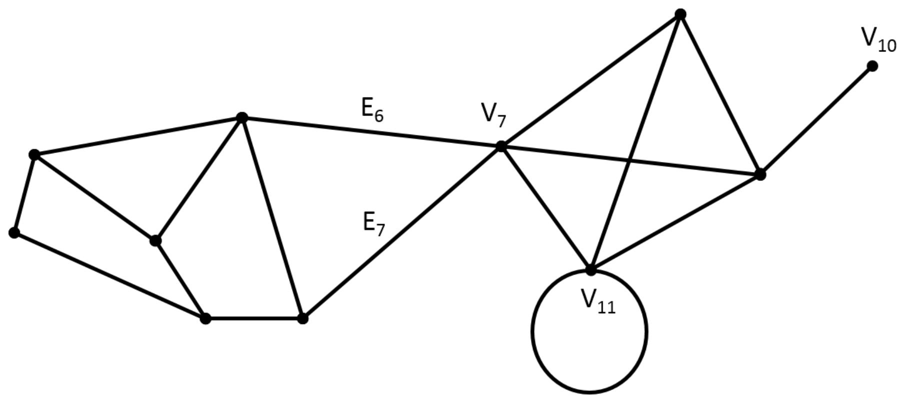


Figure 3.2.: Graph with articulation vertex V_7, a disconnected set E_6 and E_7, a dead-end V_{10} and a loop V_{11}.

Another basic concept in the context of graph theory that should be mentioned are the so-called *centrality measures*. In general, centrality measures represent information about the connectivity of the whole graph or of single nodes. An example of this is the centrality measure *connectivity*. The connectivity value can be between 0 and 1. If the value is 1, all nodes of a graph are connected. Figure 3.3 shows exemplary connectivity values with the corresponding graphs. The following formula can be used to calculate the connectivity [39]:

$$Connectivity = \frac{\sum_{i \in N} \sum_{j \in N} a_{ij}}{N(N-1)} \tag{3.1}$$

where a_{ij} is the path between the two nodes i and j, and N is the number of all paths. This measure can be used to validate a generalization of a graph, as unintentionally resulting isolated nodes or detached subgraphs can be identified.

Another centrality measure is the *betweenness centrality*. Betweenness centrality can be considered as a measure of the relevance of a node. This measure indicates the frequency of shortest paths passing a node. Specifically, this can be illustrated in Figure 3.4. All

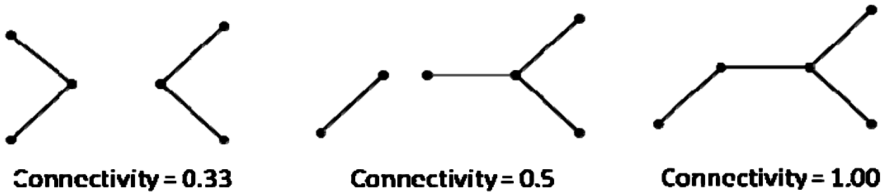


Figure 3.3.: Example of the centrality measure connectivity [39].

shortest paths from region C_1 to region C_2 pass through node V. This leads to the highest betweenness centrality value of V.

Accordingly to Jiang and Claramunt [34], the betweenness centrality for a node V_i can be calculated as follows:

$$Betweenness(V_i) = \sum_{j=1}^{N} \sum_{k=1}^{j-1} \frac{P_{ikj}}{P_{ij}} \tag{3.2}$$

where P_{ij} is the amount of shortest paths between the nodes i and j, and P_{ijk} describes the amount of shortest paths between i and j through k.

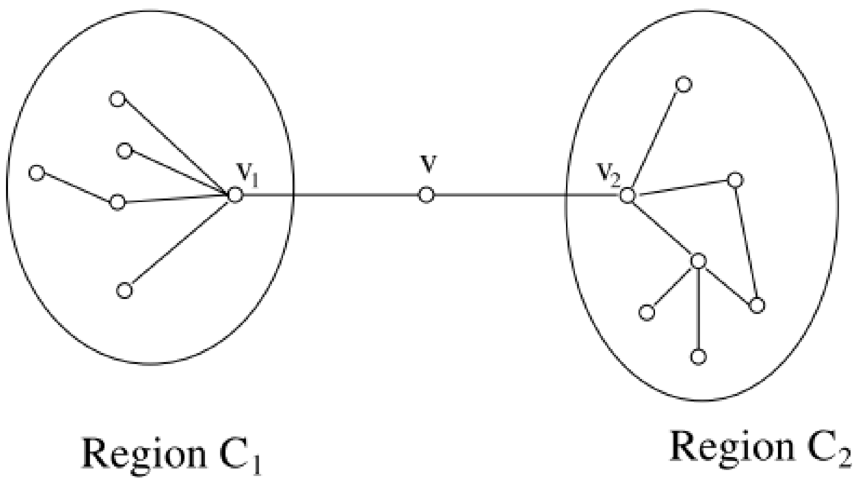


Figure 3.4.: Visualization of the betweenness centrality. All shortest paths between C_1 to C_2 pass through the node V, which leads to a high betweenness value of V [44].

There are numerous centrality measures. For a more detailed overview, we refer to [45, 46, 47] and for the closeness centrality to subsection 5.2.4.

Another part of graph theory is the distinction between different types of network graphs. For example, they can be categorized into types such as the dual or the complete graph. The dual graph represents inverted graphs, where edges of a normal graph are represented as nodes and vice versa. Consequently, nodes represent roads and edges indicate intersections [34]. Figure 3.5 depicts an example of the normal and the corresponding dual graph.

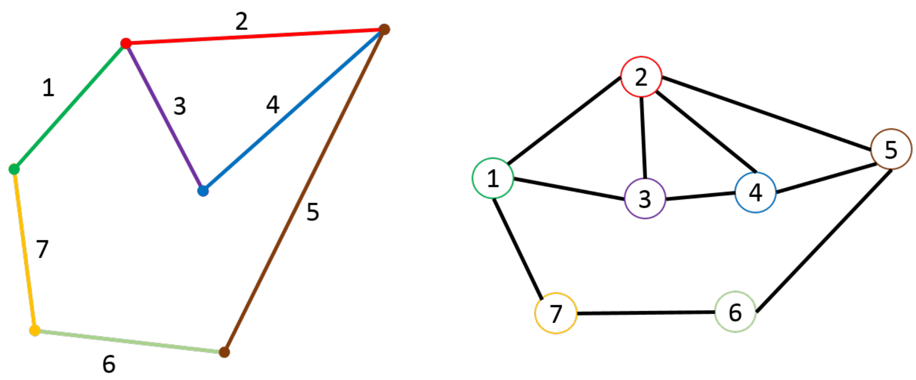


Figure 3.5.: Schematic representation of a dual graph (right) from the original graph (left).

Figure 3.6 depicts a complete graph K with 8 nodes. Complete graphs are characterized by the fact, that every node is directly adjacent to all other nodes. Such graphs can be very complex since they reflect every possible shortest path with one edge. The number of edges of a complete graph can be calculated with the following equation:

$$K_m = n * \frac{n-1}{2} \tag{3.3}$$

where n is the number of nodes.

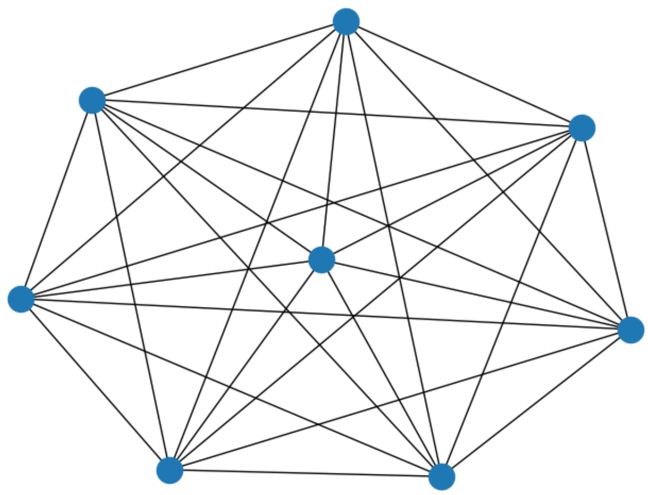


Figure 3.6.: Example of a complete graph K with eight nodes. Each node is directly connected to all other nodes.

3.2. Routing Techniques

In this section, the basic techniques of many routing tools are presented in a simplified form. The workflow of the well-known Dijkstra algorithm [41] is also introduced as an example. Many optimizations for routing in network graphs or solving the so-called shortest path problem are based on this algorithm. In section 4.1, more specific algorithms and heuristics are introduced that improve the performance of routing, based on graph partitioning or so-called arc flags or label hubs.

Routing, or solving the shortest path problem, is a very large and broad research field, that can be applied to various disciplines. For example, routing is relevant for telecommunication, computer networks, but also for routing engines in the context of navigation and transportation.

Using the information available as edge weights, most routing techniques can determine not only the shortest path (using distances from edge weights), but also the travel time or use any other arbitrary data stored as edge weights to calculate the path with the least sum of edge costs. Hence, the path between two nodes in a graph with the least cost is determined, using the edge weights as costs. There are special features that have to be considered since they limit some algorithms, such as negative edge weights or directed and undirected graphs. However, this is neglected in this section.

For the sake of clarity, we want to emphasize that a shortest path reflects the order of the visited nodes or edges. In contrast, the shortest path cost or the shortest path distance reflect the sum of the edge weights of this path. In the context of this thesis, the focus is on shortest path distances. However, the method presented in chapter 5 can be applied to other purposes as well.

Figure 3.7 illustrates Dijkstra's algorithm. Given is a graph $G = (V, E)$ with start node V_B and target node V_E highlighted. The edge weights can be arbitrary data. In the initial phase, all nodes except the start node are assigned to infinite costs. Then recursively the adjacent nodes of already visited nodes are visited and the costs are updated if the cost is lower than the already assigned cost. This process runs until the target node is reached from all possible nodes. This algorithm has the disadvantage, that all nodes in a graph have to be visited, even if they do not play a role for the shortest path to a given target node. This is illustrated in Figure 3.7 using the node A, whose reachability costs is also determined.

In modern optimizations of routing algorithms, there is mostly a tradeoff between flexibility, customizability, and performance [13]. There is an abundance of different routing techniques, optimizations, and combinations of these techniques. In the following we

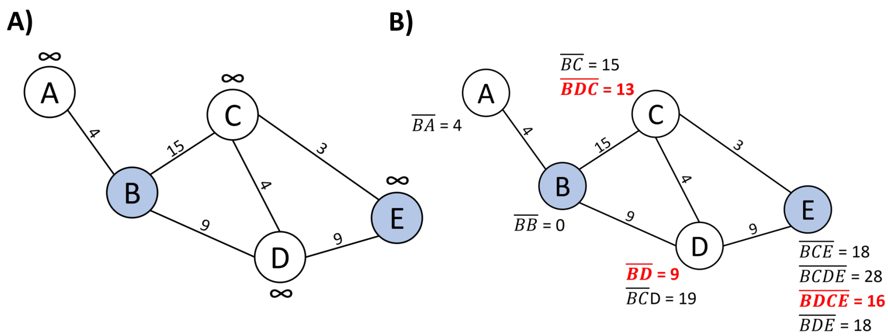


Figure 3.7.: Example of Dijkstra's algorithm. Finding the shortest, respectively the least cost path between the highlighted nodes V_B and V_E. In *A)* the initial phase is visualized. Every node except the starting node is assigned a cost of infinity. In each step, the adjacent nodes from the starting node and from the already visited nodes are then recursively updated with their respective costs. The least costs are taken over and assigned to the node. In *B)* the paths to each node and their associated costs are shown. In red is the path with the least cost highlighted, that is finally assigned to the node. The least cost path, or the shortest path, assuming edge weights are distances, from V_B to V_E is using the path B, D, C, E with a total distance of 16.

want to give a brief overview of some common routing techniques besides Dijkstra's algorithm:

- **Bidirectional Dijkstra [48]:** This algorithm is Dijkstra's algorithm implemented with bidirectional search. Bidirectional search can be implemented for some other routing algorithms as well. The search process then starts parallel from the origin and the destination node, hence from two directions.
- **A* (Goal-Directed Technique) [49]:** This algorithm is based on Dijkstra. It uses a heuristic to optimize the efficiency of the search path by searching in a targeted manner rather than checking all nodes of a graph, as it is the case with the original Dijkstra algorithm. Here, the Euclidean distance is used to restrict the search path.
- **Contraction Hierarchies [50]:** This technique has a comparatively high performance and is implemented in many navigation and routing engines. The heuristic behind this technique can be described by contracting important junctions. This is done by adding edges as shortcuts between important junctions, thus routing between two nodes can use the shortcuts and not the full path (not every node) has to be considered. This approach requires preprocessing, hence dynamically updates of edge weights are not possible. Thus, for routing engines like Open Source Routing Machine (OSRM), Contraction Hierarchies is fast, but for example, road closures can not be updated dynamically due to the required preprocessing [51].
- **Floyd Warshall [52, 53]:** This algorithm is very specific and is mostly not used in common routing engines, since this algorithm is used to identify all pairs shortest

paths (APSP). This can be useful if distances are precomputed and stored in a distance matrix.

- **breadth-first search (BFS) [54]:** This algorithm visits all nodes in a search from a given starting node (root) until the searched node is found. In this process, all directly adjacent nodes are always visited first. In contrast, the depth-first search first searches in the depth of a graph or tree and visits not step by step all directly neighboring nodes.

For a detailed overview of state-of-the-art routing techniques in terms of functionality, performance, and shortcomings, we refer to Bast *et al.* [13] for the most current overview and to Delling *et al.* [55] for an overview specifically focused on road networks.

3.3. Approximation Algorithms

Before approximation algorithms can be discussed in more detail, the basics of computational complexity must be clarified. Computational complexity allows describing the resources needed to solve complex problems. Thus, the complexity of problems can be classified. For this purpose, either the O notation is used or the complexity is reflected by a function $f(n)$, where n represents the size of the input.

For example, algorithms are divided into the complexity classes P, NP, or NP-complete. P (polynomial time) means that the problem can be solved by an algorithm in polynomial time, e.g. n^2, n^3, whereas NP-problems (non-deterministic polynomial time) require mostly exponential time, e.g. 2^n, 3^n. NP-complete is the next step, which means that both, the calculation and the validation always take exponential time.

Simplified, P-problems are all problems that a computer can solve and validate in a reasonable time. While NP-problems can be validated in polynomial time, but it may take an exponential time to solve the problem. To be precise, solving problems means determining the optimal solution.

Most combinatorial optimization problems are NP-hard, such as multicriteria shortest paths algorithms [22, 23] or the well-known traveling salesman problem [56] (*cf.* section 3.1).

For such problems, so-called approximation algorithms are mainly used in practice. Approximation algorithms for discrete optimization problems relax the requirement of finding an optimal solution, but the goal is to relax this as little as possible. Thereby, heuristics and assumptions are used to approximate the optimal solution. Consequently, if not only the optimal solution is acceptable, multiple solutions can be good enough and acceptable in some cases. This means results for approximation algorithms can be nondeterministic since the same parameters used for the same algorithm can lead to different acceptable results, depending on the used algorithm or heuristic. Here, randomness as part of a heuristic can be named as an example. For a more detailed overview of approximation algorithms, we refer to Williamson and Shmoys [56] and Johnson [57].

3.4. Map Matching

The preliminaries described below have also been adopted for the publication Hahn, Frühling, and Schlüter [58]. Further, another publication process in the International Scientific Journal - Transport Problems[1] is in progress.

Nowadays, the global positioning system (GPS) is used for almost every navigation. However, the determination of the position always contains a certain inaccuracy. Thin *et al.* [59] present and compare different methods to compensate the inaccuracy of GPS. The so-called map matching is one of these techniques. Pereira, Costa, and Pereira [60] define map matching as the task of relating a geographic point or a sequence of points to a logical model of the real world, such as road networks. Map matching can be divided into real-time map matching and offline map matching. A typical application for real-time map matching is live navigation, where the determined position should be directly located on the road and not next to the road or on the wrong lane, despite the inaccuracies of GPS. Offline map matching instead is mostly used to reconstruct the most likely path of a given GPS-track by assigning the points to nearby roads. Therefore, conventional routing engines use the perpendicular distance, which is the shortest line between a point and a line (e.g. a road segment). This can be calculated using the formula in Equation 3.4:

$$d_{perpendicular} = \frac{|Ax + By + C|}{\sqrt{A^2 + B^2}} \tag{3.4}$$

where $d_{perpendicular}$ is the distance from a point defined by (x, y) to a line defined by $Ax + By + C = 0$.

In this thesis, we focus on so-called road snapping, which is thematically and technically strongly related to offline map matching.

[1] ISSN 1896 - 0596

We define road snapping as the process of determining start and end point of a route which represent reference points for a given origin and destination on the road network. However, the intention of road snapping is different from that of offline map matching, which is why different factors are considered in road snapping than in offline map matching. Here, for example, the use of matching road names with the name of a given address can be mentioned (*cf.* Figure 2.4), which is usually not taken into account in typical offline map matching. Examples of road snapping services by conventional routing engines, that are based on perpendicular distance are the Google application programming interface (API)[2] or the Nearest API[3] from Open Source Routing Machine (OSRM).

There are many publications on real-time map matching [60, 61, 62, 63, 64, 65], but a few on offline map matching and road snapping [66], as it has received little research focus for a number of reasons. In the past, improving map matching, offline and especially real-time map matching were generally more relevant to enhance routing and postprocessing GPS-tracks. The accuracy of snapping points for pick-up and drop-off locations was less important in the past for an e.g. transport network company (TNC) as already mentioned in section 2.2. This is changed due to the increasing demand for transportation with to-door services, offered by other entities besides taxis. Accurate road snapping may also be of great interest for autonomous driving in the future, as insufficient snapping points, as sometimes derived from conventional routing engines, could no longer be compensated by the smart behavior of a human driver.

3.5. Cost Distance Analysis

As already indicated in section 3.4, the preliminaries described below have also been adopted for the publication Hahn, Frühling, and Schlüter [58]. Further, another publication process in the International Scientific Journal - Transport Problems[4] is in progress.

Even though the methodology of a cost distance analysis is sufficiently known and documented in the field of geoinformatics, we also want to address an audience of mobility researchers, which is why we explain the basics of a cost distance analysis in this chapter.

Cost distance is a “procedure for determining least cost paths across continuous surfaces, typically using grid representations” [67]. Cost distance is based on the concept, that movement in continuous space requires efforts of different kinds. Therefore, not only

[2] `https://developers.google.com/maps/documentation/roads/snap`
[3] `http://project-osrm.org/docs/v5.5.1/api/#services`
[4] ISSN 1896 - 0596

the length of a route e.g in Euclidean space but also its difficulty influence the time or cost of completing the route. The concepts of Euclidean distance and cost distance are compared in Figure 3.8.

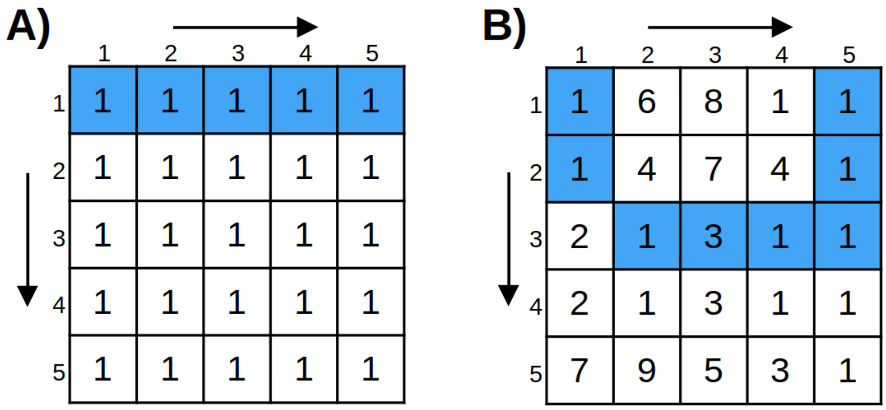


Figure 3.8.: Different distance metrics for finding the least cost path from (1:1) to (1:5) highlighted in blue. A) Least cost path by the Euclidean distance (cost = 5). B) Least cost path by cost distance (cost = 10).

Cost distance analyses, first defined in the 1950s as *Cost Based Proximity Analysis* [68], are widely used in areas such as cartography, archaeology and computer science. Some possible applications are road planning [69] and the reconstruction of ancient roads with known start and end points [70]. In many studies, the resulting paths are considered as realistic [69, 71]. Nowadays, the calculation of the least cost path based on cost distance is implemented in most geographic information system (GIS) [72].

The identification of the least cost path between two points on a grid can be done as follows. So-called source cells are given points, that refer to possible destinations.

In a cost distance analysis, for every cell in a grid, the costs of paths to all source cells are computed and compared. Consequently, the computational time scales with the number of source cells and with the resolution of the grid or raster. A cost surface is needed as an input, which can be seen as a gridded representation of a graph, describing the cost per grid-cell. In a grid representation of a graph, cell centres represent the nodes of a graph with costs passing the nodes. They are connected via edges with adjacent nodes, respectively adjacent cells [73].

There are several possible neighbourhood types that determine the number and relations of adjacent nodes of a cell. The most common ones are shown in Figure 3.9.

The weights of the edges are calculated for horizontal and vertical neighbours as shown in Equation 3.5. For diagonal neighbours the Equation 3.6 was used.

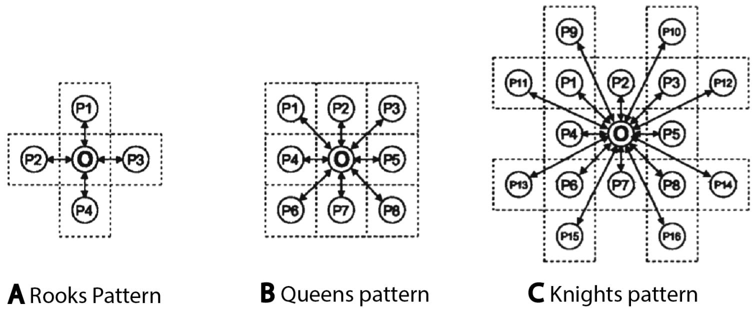


Figure 3.9.: Different neighbourhood types in gridded data representation: A) Rooks pattern - 4 linked neighbours for each cell. B) Queens pattern - 8 linked neighbours for each cell. C) Knights pattern - 16 linked neighbours for each cell. [69]

$$a_1 = \frac{(cost_1 + cost_2)}{2} \tag{3.5}$$

$$b_1 = \sqrt{2} * \frac{(cost_1 + cost_2)}{2} \tag{3.6}$$

When multiple parameters should be considered, different cost surfaces can be amalgamated into a merged cost surface. For a single cost surface or a merged cost surface, an accumulative cost surface and a backlink raster are calculated based on the cost surfaces. Therefore, in most implementations of cost distance analysis, Dijkstra's shortest path algorithm is used [74]. We used a well-established modification of this algorithm [69] to calculate the cost from each cell to the next source cell with the least cost, resulting in an accumulative cost surface and a backlink raster. The cells of a backlink raster contain coded direction values linking to the next cell on the least cost path to the source cell. The cells of an accumulative cost surface contain the actual cost of the path to the source cell. The backlink raster can then be used to track the least cost path from any cell to the next source cell with the least cost [75]. Figure 3.10 shows a simplified example of a cost distance analysis with equally weighted cost surfaces.

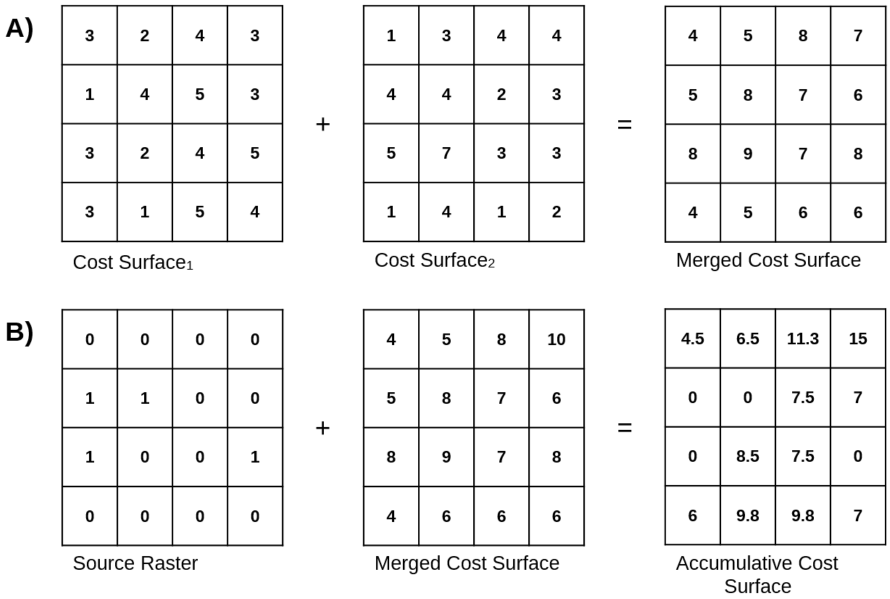


Figure 3.10.: A) Generation of a Merged Cost Surface with equally weighted Cost Surfaces. B) Generation of the Accumulative Cost Surface using the Queens pattern and Equation 3.5 and Equation 3.6. In each cell, the least costs to the next Source Cell is stored.

4. Related Work

4.1. Graph Partitioning of Road Networks

Many research domains make use of graph theory to constitute and analyze relations in data as already introduced in section 3.1. Thus, graph partitioning can play a role as part of methodologies for many different research areas, resulting in different aims and approaches just for graph partitioning. Nevertheless, graph partitioning and clustering in networks are key tools for processing and analyzing large complex networks, independent of the research area [76]. Hence, graph partitioning can be also seen as a generalization process.

Graph partitioning is a very versatile and broad field of expertise. Schulz [77] states that *"it is quite fascinating that the problem of dividing a graph into a given number of blocks having roughly equal size, such that some objective function is minimized, literally has application everywhere. For example, solving the graph partitioning problem can help to balance load and minimize communication in scientific simulations [78, 79, 80], can speed up Dijkstra's algorithm [81, 82], and in general is useful technique in the route planning area [83, 84, 85] [...]."*[1] Furthermore, such methods can be applied to e.g. social networks or arbitrary information, that can be stored in network graphs. This is particularly interesting against the background of big data since larger amounts of data can be better processed due to the partitioning. In this thesis, however, we focus on graph partitioning from the point of view for optimizing routing performance and generalization of road networks.

We present a condensed and simplified overview of the main approaches in the literature with the focus on road network partitioning and routing enhancement, that can be used to optimize e.g. the pooling process in demand responsive transport (DRT) systems. Routing enhancement can be carried out by improvements of the routing algorithms (*cf.* section 3.2) itself (e.g. A*, Bellman-Ford, bidirectional Dijkstra, etc.) or with preprocessing

[1] Remark: This direct quote has been modified by adjusting the source citations to match with the bibliography of the present thesis.

and modifications of the underlying data. In this work, we focus on the latter one, which can often be combined with enhanced routing algorithms.

Bichot and Siarry [86], Buluç *et al.* [87], Pavlopoulos *et al.* [88], and Schaeffer [31] report comprehensive overviews of graph clustering algorithms and fundamentals, mainly for application in parallel computing, biology, and chemistry. Despite the abundance of graph clustering methods, the application for road networks is less frequently investigated.

Enhancement of the computation time of shortest path algorithms in road networks, partitioning the network, preprocessing, and modifications of routing algorithms are the most common methods used in the existing literature.

In the following, we present the basic ideas of the most relevant approaches from the literature.

Arc-Flag: Reducing the search size
The widely used Arc-flag approach, introduced by Lauther [81] and improved by Hilger *et al.* [89], Köhler, Möhring, and Schilling [90], and Möhring *et al.* [82] are based on the idea of edges with additional binary information, in particular, whether they are part of a shortest path to a given region of the network. In this approach, the network is divided into simple, rectangular regions using a grid [81] or kd-trees and quad-trees [82, 89]. Due to the flags, the search size for a shortest path is reduced. Still, the preprocessing is very expensive, a modification of the used routing algorithm is needed and the space complexity depends on the partitioning because each arc has flags equal to the number of partitions.

Natural cuts: Minimizing edges between partitions
Another widely used approach, presented in the groundbreaking paper of Delling *et al.* [91], is based on the idea of natural cuts. This approach is divided into two steps. First, minimum-cuts are computed to identify dense regions of the graph. Second, search heuristics are used to create final partitions. Delling *et al.* [91] state that their resulting algorithm PUNCH (Partitioning Using Natural Cut Heuristic) is well suited for road networks since the natural cuts can be compared with bridges, mountain passes, or ferries, where a separation of a road network is reasonable, referring to large, continental-sized networks. Based on minimal or natural cuts many similar approaches emerged.

Customizable Route Planning
The customizable route planning (CRP) approach, presented by Delling *et al.* [85] and Delling and Werneck [92] is based on the principle of separating the topology of the graph as an overlay graph from cost metrics which can be changed dynamically. It consists of two main stages. *i*) metric-independent preprocessing, which generates an overlay graph based on the topology of the primal graph. Therefore a graph partitioning

algorithm is needed. Delling *et al.* [85] and Delling and Werneck [92] used the PUNCH algorithm from Delling *et al.* [91], but the partitioning algorithm can be exchanged. For each partition, incoming arcs are denoted as entry points of the partition and outgoing arcs as exit points. The overlay graph is then the bipartite graph with directed shortcuts between entry points and each exit point within the same partition. *ii*) The next stage computes actual costs of the overlay graph based on given metrics. The metric and hence the cost can be changed and updated fast and independently.

Proxies and precomputed distances
Jung and Pramanik [93] developed HiTi (Hierarchical Multi), a graph model to structure the topology of road maps. They partition large graphs into smaller subgraphs and precompute shortest paths between boundary nodes of each subgraph and finally store them in a hierarchical manner, which is then used by their new proposed shortest path algorithm SPAH. They used simple grid graphs as a representation for road networks.

Yan *et al.* [94] and Xu and Jacobsen [95] make also use of boundary nodes to partition gridded road networks. Therefore, they used the area of the whole graph, seen as a polygon. They calculated cuts (shortest paths between border nodes on the contour) of the graph with equally distributed border nodes to divide the graph into zones. Yan *et al.* [94] performed minor adjustments to generate distance-preserving subgraphs (DPS) with predefined route sources and targets based on the zones. Their approach can be used in logistics plannings where logistic hubs as predefined route sources and targets are static and known before the partitioning process.

Maue, Sanders, and Matijevic [26] describe that an extreme way to accelerate shortest path queries for static transportation networks is to precompute all shortest path distances. Yet it is not practical for large networks since it requires quadratic space and preprocessing time. Thus, they explore an approach to speed up queries by precomputing and storing only some shortest path distances, resulting in so-called precomputed cluster distances (PCD), which can be seen as a lookup table with the precomputed distances between given clusters. Therefore they used a k-centering clustering, which partitions the graph into random k-clusters with similar sizes. For each cluster, they create a new node v' with zero edge weight to every border node of the same cluster. Thereby the search size can be pruned when routing through clusters because only the newly added node and the border nodes have to be considered. The shortest path search runs in two phases. First, a bidirectional search from starting node s to target node t is performed until the search boundaries meet or until the distance between s, t and the respective border nodes of their cluster is found. The distance table with precomputed distances between clusters can then be used to look up the missing distance between the two corresponding clusters, where s and t are located.

Eapen and Beegom [96] and Ma *et al.* [97] use the concept of deterministic routing areas (DRAs) and corresponding proxies, which are small subgraphs and their representative nodes. To identify the DRAs, bi-connected components and some minor improvements (size restriction, assignment of leaf nodes) are used. A bi-connected component is a maximal set of edges, such that any two edges of the set lie on a cycle. Based on the DRAs, a reduced graph is created where DRAs are replaced with proxy nodes. The distances within each DRA between every node and its proxy are calculated and stored in a lookup table. The query stage runs differently depending on the location of starting node s and target node t. If they are within the same DRA, the precomputed paths or distances can be used from the lookup table. If they are located in different DRAs, a normal routing algorithm runs on the reduced graph between the corresponding proxies for s and t. The additional distance from the proxies to s, respectively t can then be read from the lookup table.

Various interdisciplinary approaches

Raghavan, Albert, and Kumara [98] introduced the label propagation algorithm (LPA) for detecting communities in social networks. This approach is still known for its efficiency and scalability, as it runs almost in linear time. The procedure of this algorithm starts with an initial phase where every node gets a unique label. In every further step, nodes adopt the label which occurs most in neighboring nodes. Therefore the algorithm uses the structure of the graph, hence the connections of nodes to their neighbors. If no label is in the majority (tie), like in the first iteration, then the assignment is random. Thus, the final results are nondeterministic. This algorithm runs until no labels are changed in an iteration and then nodes with the same label are grouped into a community, respectively into a partition of the graph. A special feature compared to other approaches is, that no prior analysis or knowledge about the graph is needed, such as the number of communities or the size of communities.

Modularity is another widely used approach to detect communities in social networks. The basic idea from Newman [99] is to use Modularity as a best-fitting function or measure, resulting in a score for partitions. It evaluates given partitions by relating the proportion of edges within a community to randomly distributed edges. Partitions are changed until a sufficient score is reached. For this approach, the number and size of communities are required as a given parameter. Consequently, prior analysis of the network is important, as the optimal number and size of communities depend on the network and its structure.

Anwar *et al.* [100] presented a dynamic clustering of urban road networks based on flow data. They aim to partition the network based on congestion to reduce the complexity of the network and hence the computing load for routing management could be optimized.

A modified k-clustering, a density-based clustering with a given number of clusters was used to create a density peak graph, and then as a final result, the partitioning of the network.

Another approach by Shoman and Gülgen [101] used the concept of centrality measures for thinning a road network to enhance the labeling of roads for visual purposes. The idea of centrality measures is to determine an indicator for the importance of nodes. Common centrality measures they considered are closeness, betweenness, straightness and reach. Less important nodes and corresponding edges are omitted in the generalization process.

Multilevel Graph Partitioning

The multilevel graph partitioning (MGP) is a widely used heuristic in graph partitioning in general, but it is independent of the partitioning process itself. It comprises three phases. *i*) Coarsening a graph, mostly done by contracting nodes until the graph is small enough for computational complex partitioning algorithms. *ii*) Initial partitioning by using any partitioning algorithm that uses edge weights. *iii*) Uncoarsening the graph to its finer, primal level while mapping the partitions to the finer level. Additionally, the partitions are often improved in this step by some iterative improvement heuristic (e.g. local search), because partitions based on a coarse level correspond to big changes on the primal level, which might be not optimal on the primal level. This heuristic enables the application for graph partitioning algorithms with a higher calculation complexity on large networks, due to the coarse level and the opportunity of parallelizing the computations. For more information, we refer to the multilevel graph partitioning section in [87].

Commonly used software

Commonly used software in the existing literature are METIS[2] and SCOTCH[3]. Both are not fully up-to-date. Still, there are widely used algorithms implemented. For more recent work, we want to highlight the software published by the Karlsruhe High Quality Partitioning Group (KaHIP)[4]. One focus of the software is to parallelize computations to speed-up the partitioning process itself for large networks. Since this is not the main focus of our work, we will not go further into detail.

4.2. Effects on Incomplete Map Data on to-Door Services

Inaccurate or missing map data is a huge problem for routing engines, especially for the determination of stop locations in passenger transport systems. Such stop locations, that

[2] `http://glaros.dtc.umn.edu/gkhome/metis/metis/overview`
[3] `https://gforge.inria.fr/projects/scotch/`
[4] `http://algo2.iti.kit.edu/kahip/`

are determined by road snapping (*cf.* section 3.4) can lead to miscommunication between the driver and the customer if they are not reasonable. Commercial navigation providers usually charge for so-called map curation and map updates and do not make their data available for others, since the quality of the routing depends directly on the quality of the underlying data. Nevertheless, there are also many open source solutions based on the map material of OpenStreetMap[5]. This project provides open source geospatial data and it relies on mapping and map curation of the community. Everyone has the possibility to correct flaws in the maps or to add missing information without big hurdles. Due to the large community, changes in the map material, such as road closures, are often adapted more quickly than in alternative sources [102]. Maier [102] presents a brief overview of OpenStreetMap (structure of the data and potential usage) for spatial economic research. Haklay [103] provides an overview on how good the quality of the data from OpenStreetMap is compared to alternatives.

Other publications, that are more focused on solving the problem of missing OpenStreetMap data are from Ort, Paull, and Rus [104] and Funke, Schirrmeister, and Storandt [105]. Ort, Paull, and Rus [104] propose an approach combined with LiDAR data to avoid problems caused by incomplete map data in the context of autonomous vehicle navigation and Funke, Schirrmeister, and Storandt [105] published an approach on how to extrapolate missing OpenStreetMap data in road networks.

However, in this thesis, the focus is more on the effects of missing or inaccurate map data for routing purposes. There is only few published work on mapping automation for OpenStreetMap, as this problem has not been solved yet completely and both, commercial providers like What3Words[6] and HereMaps[7] as well as OpenStreetMap rely on mapping and feedback from users. Especially missing information such as house numbers can cause critical problems for a routing based on OpenStreetMap data since then, only the centroid of the correct road is used as a destination and not the correct address. The coverage and the quality of the OpenStreetMap data vary by area. Services such as *regio-osm*[8] offer overviews e.g. of the coverage of mapped house numbers for cities.

The publication of Hu *et al.* [106] belongs to one of the few publications in this area that deal with the same thematic problem, as described in Figure 2.2. They focused on determining the entrance of public buildings and highlight the problems for routing and navigation services (e.g. from Google Maps), but have a particular focus on pedestrian navigation. Therefore, they use statistical learning, or classification (weighted random forest, balanced random forest, and smoteBoost). They assumed that the position of the

[5] `https://www.openstreetmap.de/`
[6] `https://what3words.com`
[7] `https://www.here.com/`
[8] `https://regio-osm.de/`

entrance of public buildings has certain patterns. Intrinsic features such as the distance from the outer building footprint (edges) to the centroid and extrinsic features such as the shortest path to the main road were used. Hu *et al.* [106] state, that most entrance detection methods rely on the analysis of street-level images (e.g. image recognition), and in contrast to that, their approach relies only on data from OpenStreetMap.

Another publication that aims for the detection of entrances is published by Kang *et al.* [107]. They aim to improve autonomous navigation for robots by using street-level images and distinguish for example objects like windows and doors with image recognition to consequently determine the entrance of buildings. However, this can barely be used for the navigation described in the context of the present thesis.

5. Performance of Network Distance Computations

5.1. Central Ideas

In section 2.1, drawbacks related to the use of Euclidean distance and limitations of the performance of algorithms for calculating network distances in the context of transportation services or transportation research have been described.

In this chapter, we present an approach based on generalizing the network graph and therefore reducing the complexity for computations or even for precomputing network distances. For this purpose, we make the following assumptions:

1. For some purposes, approximated network distances are sufficient. They are less error-prone than distances calculated by Euclidean distances. A possible application could be a preselection of considered stops when similar travel requests should be pooled.
2. To construct edges for the generalized graph, we used the shortest paths between selected nodes on the primal graph and add these shortest paths as new edges on the generalized graph. Therefore, we assume, that in practice mainly the shortest distance is used.

Graph partitioning techniques (*cf.* section 4.1) were used to generalize the road network, hence the primal network graph to create a generalized graph.

For this generalized graph, it is more feasible to precompute network distances and store them e.g. in a lookup table, for which then only a search algorithm is needed, such as linear search, which scales linearly instead of common algorithms such as A* or (bidirectional) Dijkstra (*cf.* section 3.2), which do not scale linearly with the size of the network (run time $\approx O(n^2)$).

For our approach, the graph is divided into different partitions and for each partition, a reasonable proxy is computed using a centrality measure (*cf.* section 3.1). All shortest paths between the proxies of the partitions are then extracted from the primal graph and transferred to the generalized graph as edges. This results in a complete graph $K_{reduced}$ with realistic network distances between the proxies. The complete graph $K_{reduced}$ is then stored in an adjacency matrix and we implemented a prototype for network distance queries to compare the performance with conventional implementations of A* and (bidirectional) Dijkstra. A parameter that limits the size of partitions in network distance is used to manipulate and estimate the accuracy of the approximated network distances.

5.2. Methods

5.2.1. Area of Interest

The selection of a suitable area of interest (AOI) can be challenging due to different types of patterns in road networks, such as the five road patterns from Southworth and Ben-Joseph [108]: Gddiron, fragmented parallel, warped parallel, loops and lollipops and lollipops on a stick (*cf.* Figure 5.1). The topological structure of a road network has an impact on the results of partitioning algorithms. Since we aim for real-world application, we want to avoid using only a grid graph, which only represents manhattan-like networks. Another challenge is the visualization of large road networks as well as the way of functioning of the reduction process in detail. Therefore, we chose smaller road networks and visualize oversimplifications as a showcase. We selected different types of road networks that can be designated as gridded-like, concentric-like, mixed, and twisted road network patterns. The extent and the primal road networks for each selected AOI are shown in the Appendix A. Basic properties as the circuity, the total street length, and the area for each AOI are shown in Table 5.1.

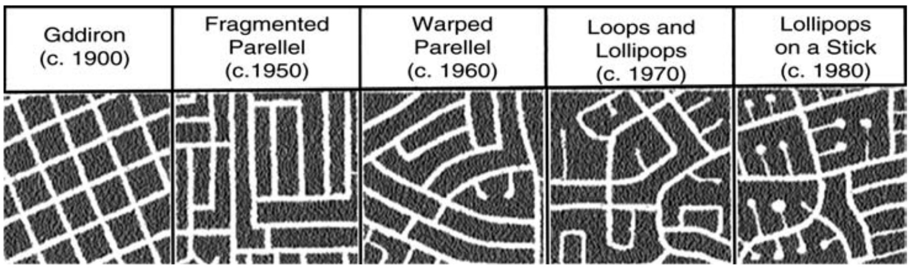


Figure 5.1.: Five patterns in road networks from [108].

	circuity	total street length [m]	area [km^2]
Göttingen	1.064	226575	25.90
Krefeld	1.021	73328	3.98
Málaga	1.143	57045	4.01
Soest	1.059	125969	10.08

Table 5.1.: Basic properties for each AOI.

5.2.2. Data

We created a weighted undirected Graph $G(V, E)$, where V is the set of nodes and E the set of edges consisting of V_u, V_v. This is done based on data from OpenStreetMap[1].

5.2.3. Partitioning

We considered approaches from different research areas for the partitioning of networks. To the best of our knowledge, approaches from social sciences to detect communities have not yet been considered for partitioning road networks. We selected the FluidC algorithm proposed by Parés *et al.* [109], which is based on the idea of fluids interacting in an environment, expanding and contracting as a result of that interaction. This idea could be also feasible for the partition of road networks into similar parts without prior analysis or knowledge of the network, which is needed for widely used partitioning and clustering algorithms such as k-means. This requires, for example, the number of clusters, hence partitions, which depend on the size and structure of the network. The FluidC algorithm is an enhancement of the already mentioned label propagation algorithm [98] in section 4.1. The process of the FluidC algorithm starts to assign random nodes to k-partitions. These partitions expand and push until a balanced, stable state in the sense of density is found. In simple terms, for each community the density with a range between 0 and 1 is calculated with:

$$Density(c) = \frac{1}{v \in c} \tag{5.1}$$

and nodes are assigned to the nearest community respecting the topology with the lower density to reach a balanced partition. Figure 5.2 depicts the workflow of the FluidC algorithm for two partitions. For a more detailed description of the algorithm, we refer to Parés *et al.* [109].

1 `https://www.openstreetmap.org/`

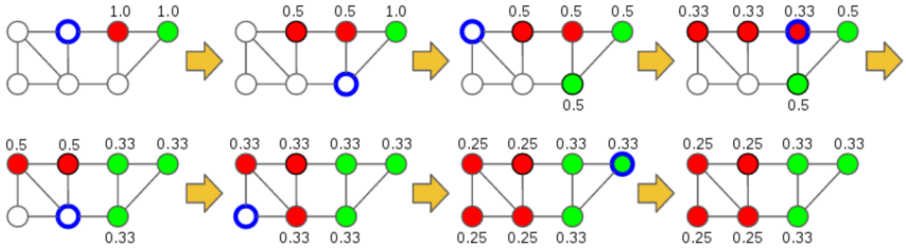


Figure 5.2.: Workflow of the FluidC algorithm with two partitions in green and red. The calculated density for assigned nodes determines to which partition a considered node (blue circle) will be assigned [109].

We used the implementation of the FluidC algorithm from networkx[2] and enhanced this implementation with an evaluation to detect a suitable size of k-partitions, using a maximum acceptable distance deviation. After an initial partition, a proxy for each partition will be determined (*cf.* subsection 5.2.4). From this proxy, the shortest path costs (distance) to all other nodes within the same partition will be calculated. If the cost of any shortest path is bigger than the given distance deviation parameter, the number of k is incremented until partitions with the given requirements are found. This regularization mechanism ensures, that after the final generalization each node is reachable within the used distance deviation from the corresponding proxy. By manipulating the distance deviation, the degree of generalization can be modified. The smaller the distance deviation, the more partitions will be created and the resulting approximated network distances are more accurate.

5.2.4. Determination of Partition Proxies

In order to determine the proxies for partitions, we considered centrality measures (*cf.* section 3.1). We selected the centrality measure *closeness centrality*. The node with the highest closeness centrality for each partition is selected as a proxy. Since the closeness centrality value of a node is influenced by the whole graph, the closeness values are calculated for each partition, respectively subgraph, separately in each iteration.

Closeness centrality is the average length of the shortest path between the selected node to all other nodes. For the closeness centrality, it is relevant if the given graph is directed or undirected due to the difference between incoming and outgoing paths for the selected node. Thus, we transformed the road network into an undirected graph. The closeness centrality was calculated with Equation 5.2:

[2] `https://networkx.github.io/documentation/stable/reference/algorithms/generated/networkx.algorithms.community.asyn_fluid.asyn_fluidc.html`

$$C_{Closeness}(u) = \frac{n-1}{\sum_{n-1}^{v=1} d(v,u)} \quad (5.2)$$

where $d(v,u)$ is the distance of the shortest path between node v and u, n is the number of nodes, that can reach the selected node u.

5.2.5. Building a Generalized Graph

To create a generalized graph, we assumed that mainly the shortest paths are used in transportation and hence the shortest path between proxies on the primal graph can represent almost realistic trips between these points.

The generalized, complete graph $K_{reduced}$ (*cf.* section 3.1) is based on the set of proxies V'. The edges are generated depending on the shortest paths between the proxies on the primal graph. The costs of the shortest paths are added as the edge weights. The geometric properties from the original path can also be transferred to the new edges, but since we focus on enhancing the computing time, the original properties of the edges, except the distance on the road network will be neglected. All properties of edges that are part of a shortest path could be transferred to the new edge of the complete graph. However, cases would have to be taken into account that e.g. different road names or different types of roads have to be either combined or generalized. To examine the presented approach more closely, only the distance of the edge weights was taken and summed up and assigned to the new edge of the complete graph. In the future, other properties should be able to be transferred without major challenges.

The generalized graph can then be created as a complete graph, where each node is directly connected to every other node. This information can also be stored in a lookup table for fast processing. The performance of lookups from such a table depends on the search algorithm, e.g. for linear search, it runs at worst with a run time $O(n)$.

5.2.6. Network Distance Queries

To enable queries to retrieve network distances, a preprocessing is required once to extend the properties of all nodes in the primal graph by the ID of the referring proxy. Then the referring proxies P_s and P_t can be read for a routing request from V_s to V_t. Approximate network distance between V_s to V_t, thus the distance between the proxies P_s and P_t can then be read from the adjacency matrix or lookup table from $K_{reduced}$.

The resulting distances tend to underestimate the exact network distances, since the distances from the assignment of nodes (e.g. V_s, V_t) to proxies (e.g. P_s, P_t) are not taken into account. However, these missing distances are smaller than the used distance deviation. To minimize this underestimation of network distances, a corrective factor can be added to the network distance between the two proxies. For example, for P_s and P_t the distance deviation could be added for the start and the target node so that the final network distance between any two nodes is either exact or overestimated. Thus a corrective factor of a maximum of two times the distance deviation can be added. A smaller corrective factor may be more reasonable.

5.2.7. Scaling and Variability

To identify the behavior for different conditions, we applied this approach for each AOI, introduced in subsection 5.2.1, with different distance deviations. Due to the influence of the road network pattern, it is very difficult to determine an equation to calculate reasonable distance deviations on the basis of the area or the total street length. Initial tests for each AOI lead to the following used distance deviations, shown in Table 5.2. Thereby, we aimed to have a comparative number of partitions (roughly between 10 and 25 partitions) for the constant parameter. For the scaling investigation, a minimum and a maximum distance deviation were determined. The minimum value was determined by decreasing the distance deviation until the computation time has increased significantly and the computation was almost impracticable. The maximum value was determined so that only a few partitions will be created. Between these two values, 15 evenly distributed distance deviations were determined as parameters. For each distance deviation, multiple iterations were performed to evaluate the behavior, e.g. the fluctuation of the results.

	constant [m]	scaling minimum [m]	scaling maximum [m]
Göttingen	3500	2.300	5.000
Krefeld	1500	600	2.000
Málaga	1000	400	1800
Soest	2000	1300	3000

Table 5.2.: Used distance deviations in the evaluation for the part with a constant distance deviation and the part with a changing distance deviation (scaling between minimum and maximum).

5.2.8. Evaluation

We split the evaluation into two parts. In the first part, we analyze the behavior with a constant distance deviation and in the second part, we analyze the behavior with a changing distance deviation to evaluate the scaling of the approach (*cf.* subsection 5.2.7). We have structured these two parts as follows: For each aspect investigated, we have summarized in a clear way what the aim of the investigation is, why we are studying it and what analyses and interpretations this study allows, and finally, which methods are used to measure and present the results.

For each of the investigated aspects, we ran every calculation with $n = 14$ iterations, using the exact same parameters to obtain a basic statistical overview. This allows evaluating the results of the nondeterministic approach, e.g. with respect to the scattering.

5.2.8.1. Constant parameter

1. Number of partitions
Aim of investigation: Variability of the resulting number of partitions for n iterations.
Analysis and interpretation: Reviewing the scattering of the results helps to define the impact of randomness in the nondeterministic approach.
Methods: Using descriptive statistics, visualized by scatter plots.

2. Size reduction
Aim of investigation: Comparing the size (the number of edges) for all precomputed shortest paths of the primal and the reduced graph, hence the enhancement of the generalization regarding the size for precomputed shortest paths.
Analysis and interpretation: Indicates the size reduction for precomputed paths for used distance deviation and shows the potential use for large network graphs.
Methods: Using the ratio of the complete graphs K_{primal} and $K_{reduced}$ for n iterations, visualized in scatter plots.

3. All pairs shortest path (APSP)
Aim of investigation: Uniform distribution of proxies.
Analysis and interpretation: The used algorithm FluidC leads to nondeterministic results, due to the influence of randomness by selecting nodes. The mean distance between all proxies, hence all shortest path distances are an indicator if the proxies are reasonably homogeneously distributed in the road network, which allows the conclusion that the proxies represent the primal road network somehow realistic.
Methods: Using error bars to visualize the variation of the mean distance of all shortest path distances between proxies.

4. Performance
Aim of investigation: Performance of network distance queries for the presented approach in comparison to A*, Dijkstra, and bidirectional Dijkstra.
Analysis and interpretation: Shows the potential speed-up and difference compared to conventional algorithms. To be comparable in the performance calculation, we used implementations in the programming language python using the same libraries and not implementations that are faster due to more performant programming language (e.g. C++).
Methods: Using a random set of node pairs (50 pairs) and take the mean of the processing time in seconds for *n* iterations and visualize the results with boxplots.

We want to point out, that this compares approximate network distances versus exact network distances.

5.2.8.2. Scaling parameter

Overall, 15 different distance deviations were used and for each distance deviation $n = 14$ iterations were performed.

1. Number of partitions

Aim of investigation: Variability of the resulting number of partitions for n iterations and 15 different distance deviations.

Analysis and interpretation: Indicates the scaling of the number of partitions e.g. linear or exponential if the distance deviations change linearly. This can help to estimate the size of the complete graph $K_{reduced}$.

Methods: Creating boxplots for each distance deviation and repetitions with the same parameter of $n = 14$.

2. Size reduction

Aim of investigation: Comparing the size (the number of edges) for all precomputed shortest paths of the primal and the reduced graph, hence the enhancement of the generalization regarding the size for precomputed shortest paths.

Analysis and interpretation: Indicates the size reduction for precomputed paths for used distance deviations and shows the potential use for large network graphs.

Methods: Using the ratio of the complete graphs K_{primal} and $K_{reduced}$ for different distance deviations. Since for each distance deviation multiple repetitions were performed (with n iterations), boxplots for each distance deviation were created.

3. All pairs shortest path (APSP)

Aim of investigation: Uniform distribution of proxies.

Analysis and interpretation: The used algorithm FluidC leads to nondeterministic results, due to the influence of randomness by selecting nodes. The mean distance between all proxies, hence all shortest path distances are an indicator if the proxies are reasonably homogeneously distributed in the road network, which allows the conclusion, that the proxies represent the primal road network somehow realistic.

Methods: The weighted mean and weighted standard deviation are computed and visualized in scatter plots. For each distance deviation, n iterations were performed. For the weighted mean $\bar{x}^*$ and the weighted standard deviation s^* we used the following equations:

$$\bar{x}^* = \frac{\sum_{i=1}^{N} w_i x_i}{\sum_{i=1}^{N} w_i}. \tag{5.3}$$

$$s^* = \sqrt{\frac{\sum_{i=1}^{N} w_i (x_i - \bar{x}^*)^2}{\frac{(M-1)}{M} \sum_{i=1}^{N} w_i}} \tag{5.4}$$

4. Variations of APSP-distances with respect to the distance deviation
Aim of investigation: The percentage variation of the distance between proxies (weighted standard deviation) related to the used distance deviation (the given parameter).
Analysis and interpretation: The proportional variation with respect to the selected parameter is helpful to classify the magnitude of the variation, and consequently it allows to draw the conclusion, whether proxies are reasonably homogeneously distributed.
Methods: Using the ratio of the weighted standard deviation and the used distance deviation in percent.

5. Performance
Aim of investigation: Performance of network distance queries for the presented approach in comparison to A*, Dijkstra and bidirectional Dijkstra.
Analysis and interpretation: Shows the potential speed-up and difference compared to conventional algorithms. To be comparable in the performance calculation, we used implementations in the programming language python using the same libraries and not implementations that are faster due to a more performant programming language (e.g. C++).
Methods: Using a random set of node pairs (50 pairs) and take the mean of the processing time in seconds for *n* iterations and visualize the results with boxplots.

We want to point out, that this compares approximate network distances versus exact network distances.

5.3. Own Software Package

For the presented approach, a framework called FluidC-Generalization based on Proxies (FC-GBOP) was programmed (~2000 lines of python code). Thus, the approach including the evaluation can be carried out. This framework contains some additional features. Therefore and for the whole project, including documentation, we refer to the following link of the repository.

`https://github.com/fauceta/FC-GBOP`

The main source code is shown in Appendix B.1 to give a brief overview of the amount of programming work. This printout is not formatted and is not intended for practical use. We want to point out, that the design of this framework is focused on functionality rather than on the performance of the preprocessing stage.

5.4. Results

In this section, some results are presented as examples and for visualization. In the next sections subsection 5.4.1 and subsection 5.4.2, the results of the evaluation are shown and for reasons of comprehensibility, the results are also interpreted directly after the results for the respective aspect so an assignment of results and discussion (interpretation) becomes more comprehensible. In section 5.5 the main concluding results are summarized.

Methodically, we want to emphasize that the comparison of some figures can be misleading due to different scaling. We are aware of this and have therefore tried to apply the following rule. If the value ranges are uniform, they are used uniformly. However, if the range of values differs so much from other comparable figures that the axes and the scaling would have to be stretched even further and the readability would suffer, the scaling was adjusted in powers of 10 and the different scale was explicitly pointed out for reasons of transparency.

Figure 5.3, Figure 5.4, Figure 5.5, and Figure 5.6 show examples of the FluidC approach to generalize road networks. These figures are intended for visualization of the approach, which is why very simple parameters have been used. On the left, the primal road networks are shown. The partitions are colored and the corresponding proxies are marked by black squares. On the right side, the reduced complete graphs $K_{reduced}$ are shown, that are based on the proxies and the realistic edge weights between them.

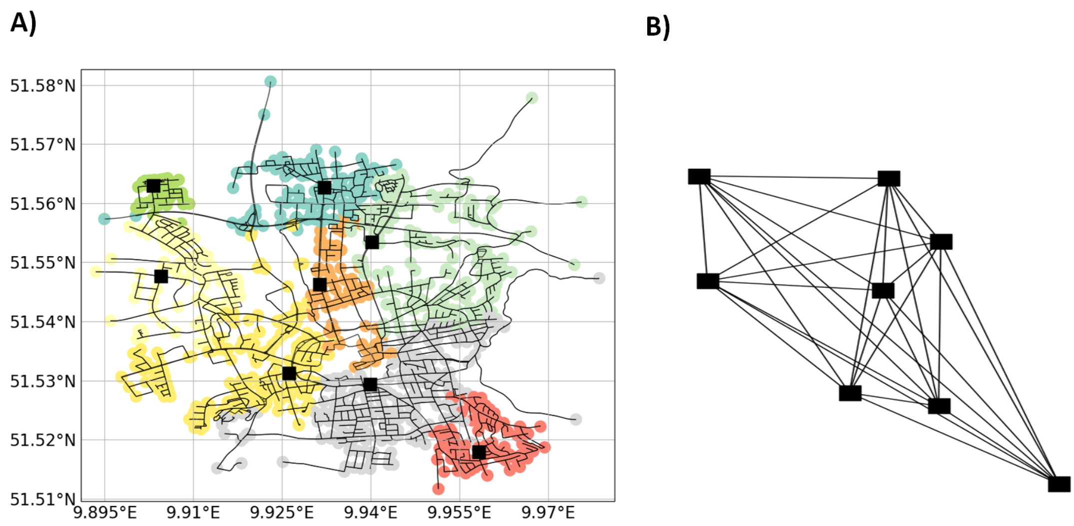


Figure 5.3.: In *A*) an example of partitions is visualized in colored nodes and their proxies by black squares with a distance deviation of 4500m for Göttingen as a showcase. In *B*) the corresponding reduced complete graph $K_{reduced}$ is presented.

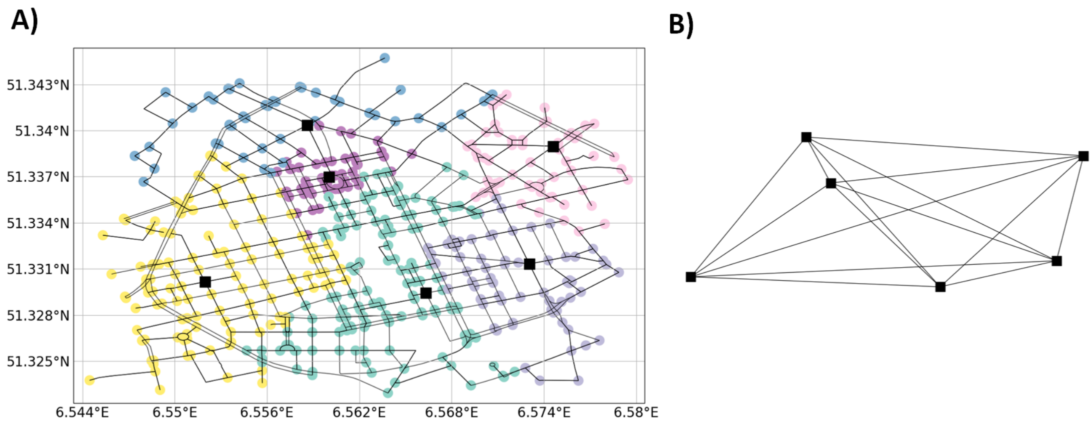


Figure 5.4.: In *A*) an example of partitions is visualized in colored nodes and their proxies by black squares with a distance deviation of 1500m for Krefeld as a showcase. In *B*) the corresponding reduced complete graph $K_{reduced}$ is presented.

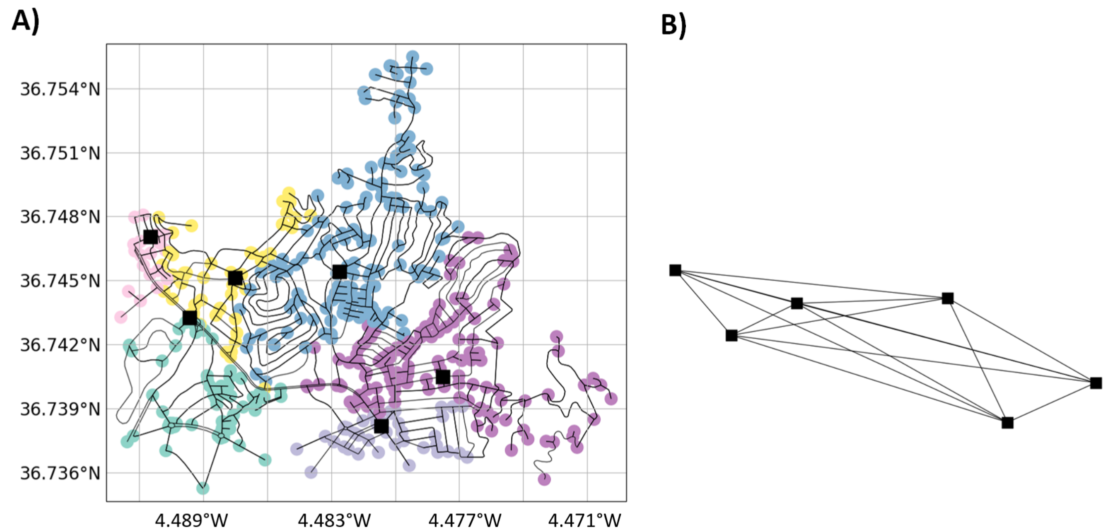


Figure 5.5.: In *A*) an example of partitions is visualized in colored nodes and their proxies by black squares with a distance deviation of 1400m for Málaga as a showcase. In *B*) the corresponding reduced complete graph $K_{reduced}$ is presented.

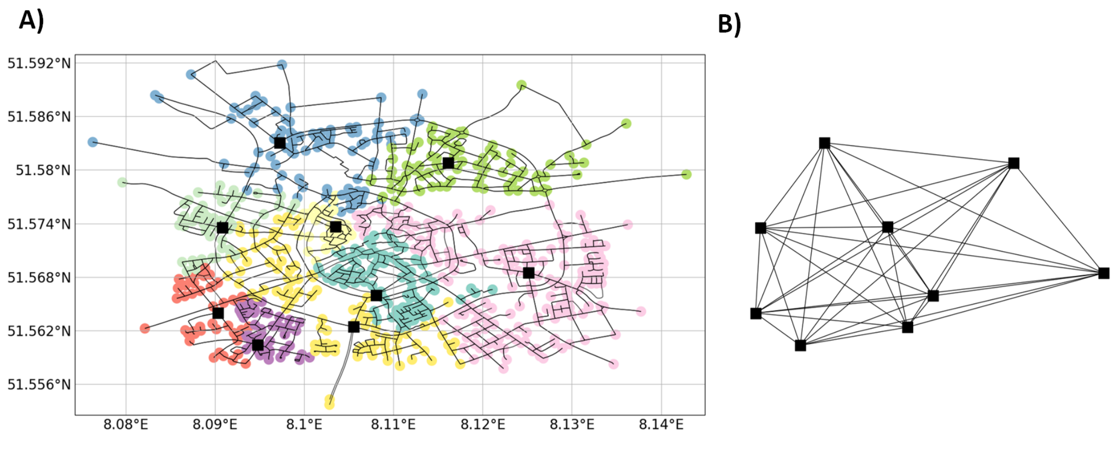


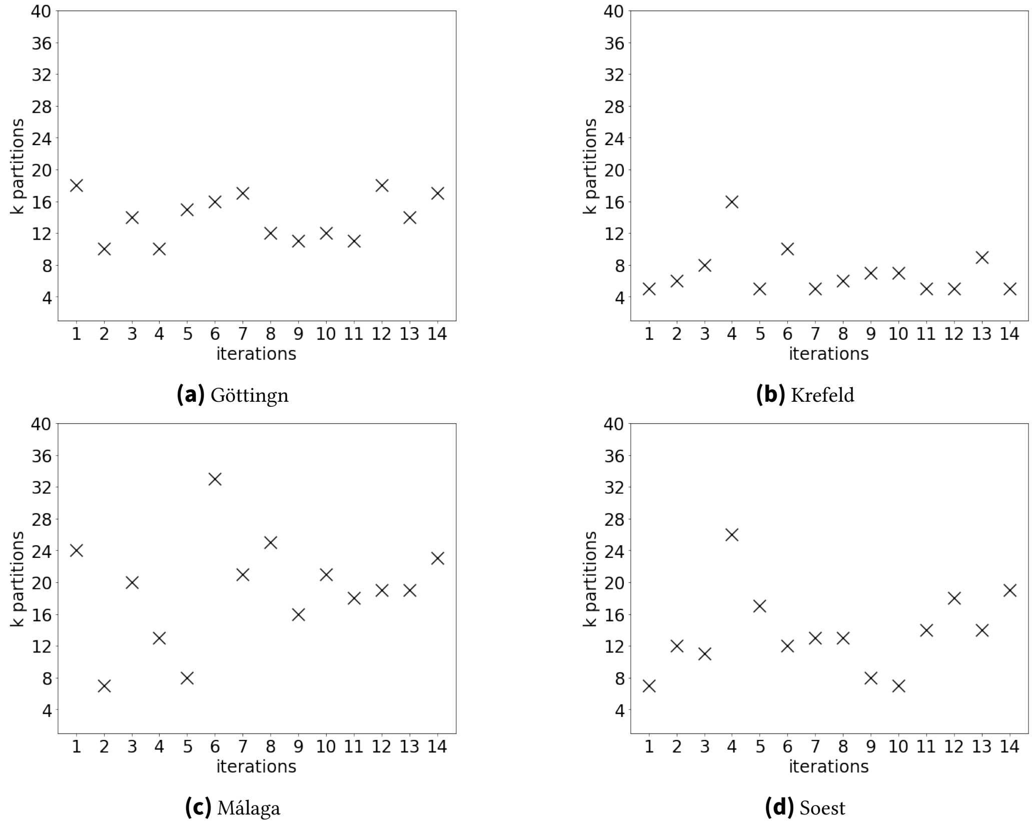
Figure 5.6.: In *A*) an example of partitions is visualized in colored nodes and their proxies by black squares with a distance deviation of 2300m for Soest as a showcase. In *B*) the corresponding reduced complete graph $K_{reduced}$ is presented.

5.4.1. Constant Parameter

Regarding the procedure of the evaluation for a constant distance deviation, described in subsubsection 5.2.8.1, the resulting plots are shown in the following. For the constant distance deviation, multiple iterations ($n = 14$) were performed for each AOI. The used distance deviations are listed in Table 5.2.

1. Number of partitions

Figure 5.7 shows the scattering of the number of resulting partitions. Overall, Málaga shows the highest variation of 26 between the smallest (7) and the largest number of partitions (33). In contrast, Göttingen has the least variation of 8. It can be seen, that the results for Málaga show by far the most scattering, followed by the results for Soest and the results for Göttingen and Krefeld have a comparatively low scattering. This behavior can be explained due to the different road patterns since Málaga has the most convoluted road pattern, followed by Soest, Göttingen and Krefeld. In Table 5.3 the basic statistics of the results are shown.



(a) Göttingn

(b) Krefeld

(c) Málaga

(d) Soest

Figure 5.7.: Number of k partitions for a constant distance deviation.

	minimum	maximum	mean	standard deviation
Göttingen	10	18	13.9	3
Krefeld	5	16	7.1	3
Málaga	7	33	19.1	7
Soest	7	26	13.6	5

Table 5.3.: Basic statistics for the number of k partitions.

2. Size reduction

Figure 5.8 depicts the ratio of the edges of the complete graphs K_{primal} and $K_{reduced}$. This represents the size of the networks, when all shortest paths would be precomputed (*cf.* complete graphs in section 3.1). The size of the precomputed network K_{primal} does not change, but the size of the graph $K_{reduced}$. This is directly related to the number of partitions. Based on the value, it can be seen that the magnitude of $K_{reduced}$ is much smaller than that of K_{primal}. This shows the potential reduction in size using the FluidC approach for all precomputed shortest paths. Table 5.4 shows the basic statistics corresponding to Figure 5.8.

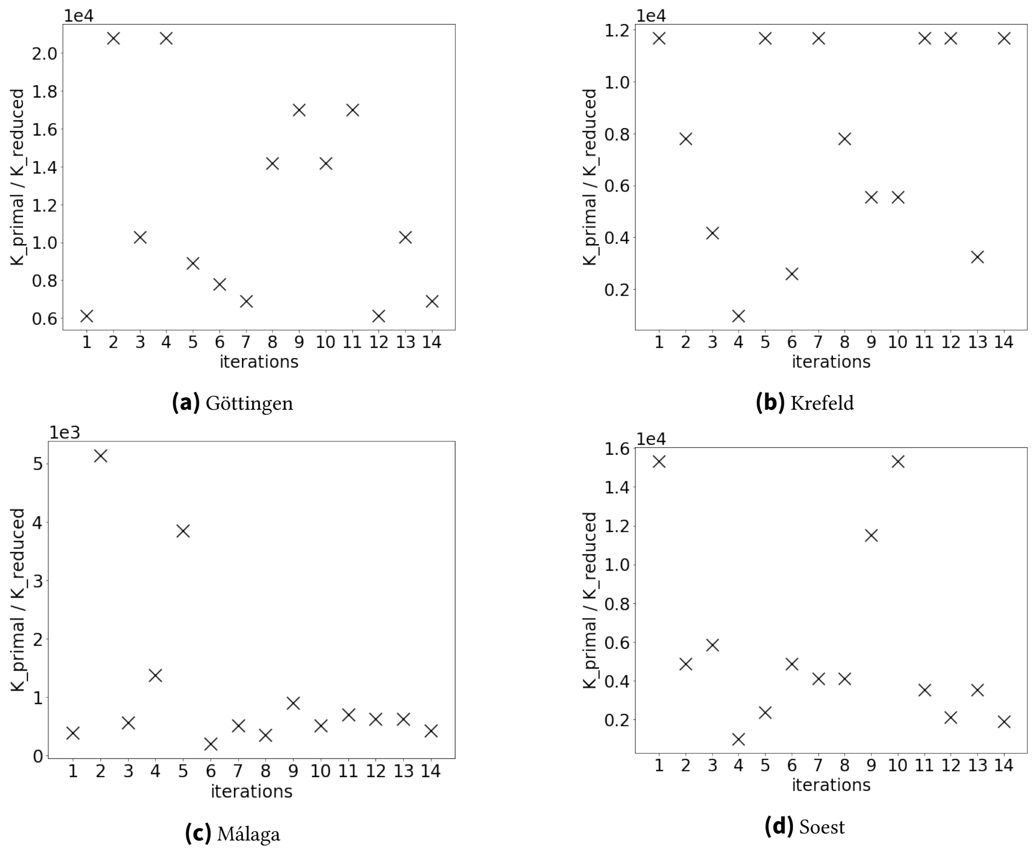


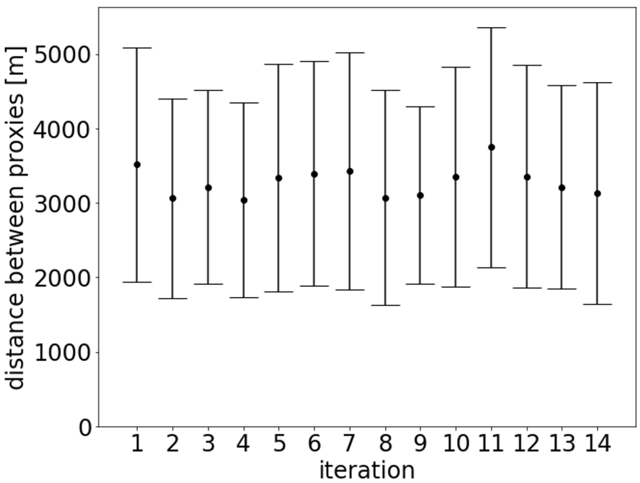
Figure 5.8.: Ratio of the size of the complete graphs $K_{reduced}$ and K_{primal}.

	minimum	maximum	mean	standard deviation
Göttingen	6120	20808	11954	5138
Krefeld	974	11688	7702	3855
Málaga	204	5137	1158	1409
Soest	991	15333	5747	4595

Table 5.4.: Basic statistics for the ratio of the complete graphs $K_{reduced}$ and K_{primal}.

3. All pairs shortest path (APSP)

Figure 5.9, Figure 5.10, Figure 5.11 and Figure 5.12 depict the variability of all shortest path distances between proxies. A small variation is indicative of an even distribution of the proxies on the road network. The figures show a large range of distances between proxies, but the mean values seem to be more or less constant for every AOI. A larger range of values is logically more likely for real road networks than for theoretical or symmetrically constructed networks such as gridded ones. Furthermore, the range of values should also be seen in the context of the used distance deviation (*cf.* Table 5.2). The fluctuation of the mean values is therefore more meaningful. Málaga is the only AOI with a larger range than the used distance deviation, which can be traced back to the fact, that this is the most convoluted network of all considered networks. In addition, only the mean values for Málaga show a comparatively larger fluctuation.

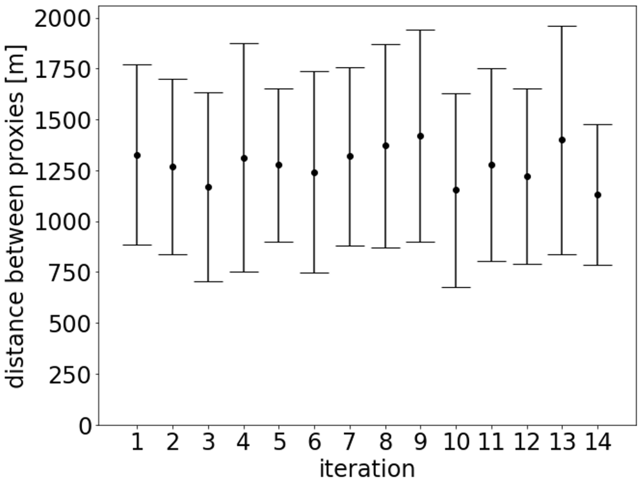


iteration	k partitions
1	18
2	10
3	14
4	10
5	15
6	16
7	17
8	12
9	11
10	12
11	11
12	18
13	14
14	17

Figure 5.9.: APSP-distances for Göttingen (constant).

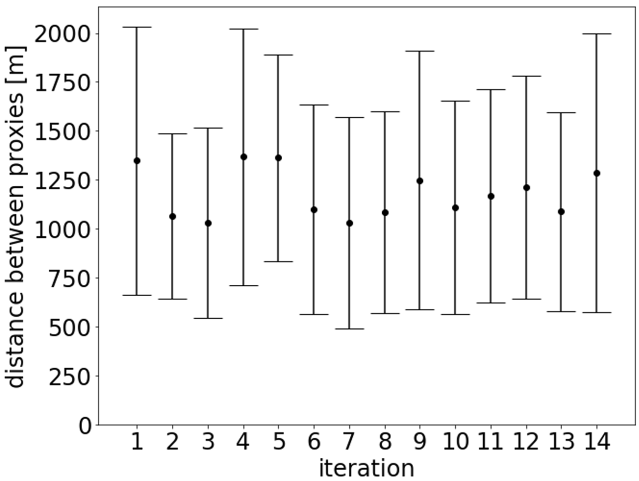
4. Performance

Figure 5.13 depicts the performance of the routing algorithms Dijkstra, bidirectional Dijkstra, A* and the presented approach based on FluidC. In these figures, the approach based on FluidC is called “reduced”, because instead of returning exact network distances as the other algorithms do, this approach only returns reduced network distances. For each iteration, 50 random node pairs were used to measure the time for the network distance queries. For every AOI, Dijkstra and bidirectional Dijkstra show by far the worst



iteration	k partitions
1	5
2	6
3	8
4	16
5	5
6	10
7	5
8	6
9	7
10	7
11	5
12	5
13	9
14	5

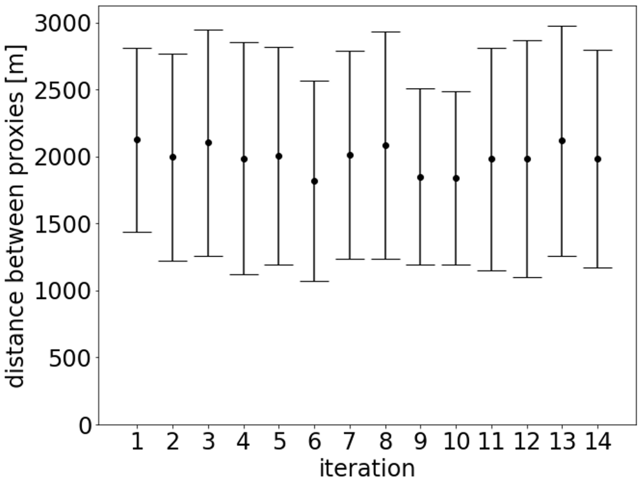
Figure 5.10.: APSP-distances for Krefeld (constant).



iteration	k partitions
1	24
2	7
3	20
4	13
5	8
6	33
7	21
8	25
9	16
10	21
11	18
12	19
13	19
14	23

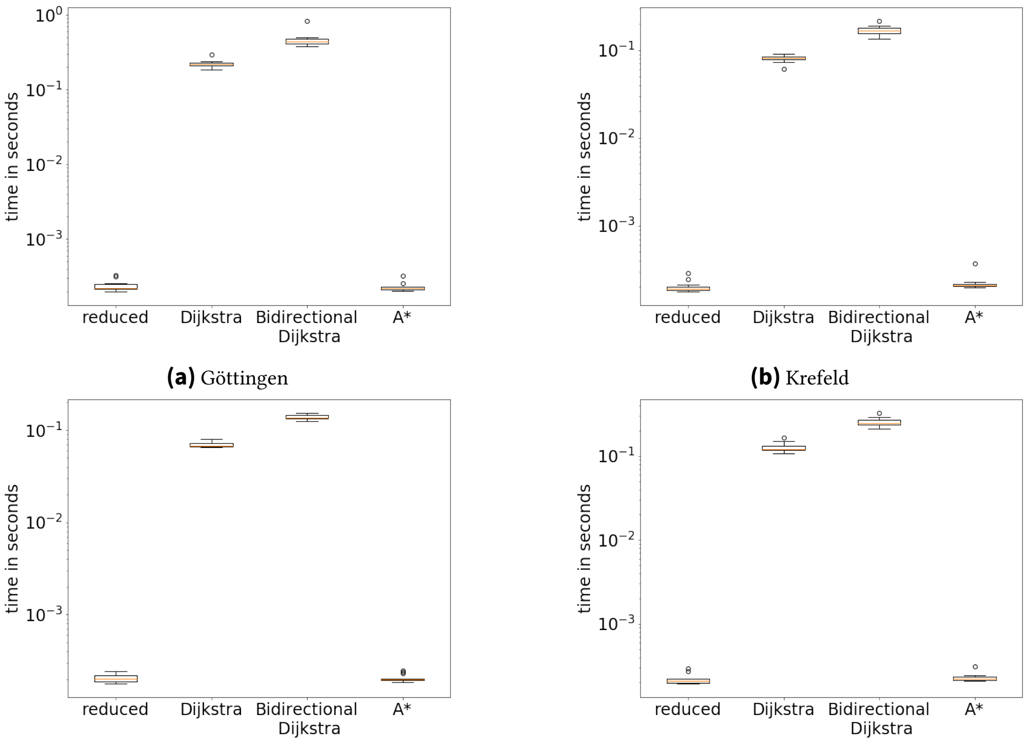
Figure 5.11.: APSP-distances for Málaga (constant).

performance of the considered algorithms. The reduced approach (FluidC) and A* show only minor differences but perform by far the best.



iteration	k partitions
1	7
2	12
3	11
4	26
5	17
6	12
7	13
8	13
9	8
10	7
11	14
12	18
13	14
14	19

Figure 5.12.: APSP-distances for Soest (constant).



(a) Göttingen

(b) Krefeld

(c) Málaga

(d) Soest

Figure 5.13.: Performance of different routing approaches for each AOI.

5.4.2. Scaling Parameter

In this section, the results for a changing distance deviation are presented, which shows the scaling of the introduced approach. Again, for each distance deviation, multiple iterations ($n = 14$) were performed for each AOI. Between the selected minimum and maximum (*cf.* subsection 5.2.7 and Table 5.2) 15 equally distributed distance deviations were used.

1. Number of partitions

Figure 5.14, Figure 5.15, Figure 5.16 and Figure 5.17 show the variation of the resulting k partitions for different distance deviations. Overall, for these figures, three main facts can be derived. First, the larger the distance deviation gets, the smaller the number of k partitions is. Second, in each figure, an asymptotic tendency can be recognized. There seems to be a certain level of distance deviation, above which the number of k partitions hardly change. Third, the variability per distance deviation within the iterations decreases with an increasing distance deviation.

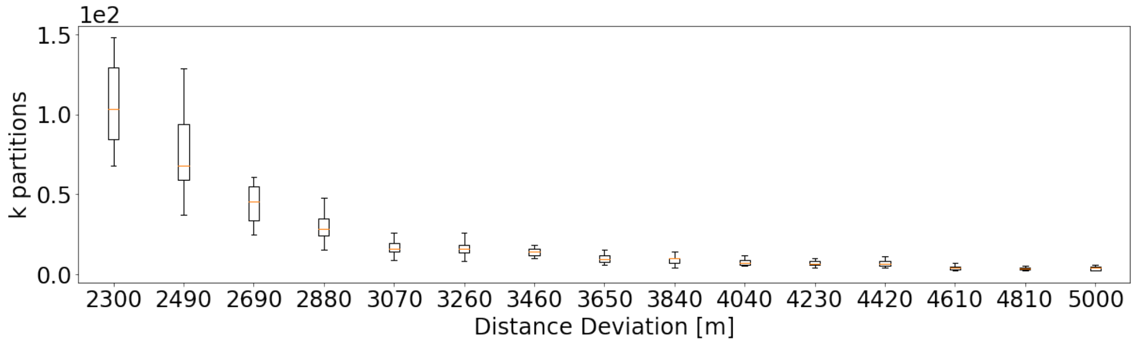


Figure 5.14.: Number of k partitions versus distance deviation for the AOI Göttingen.

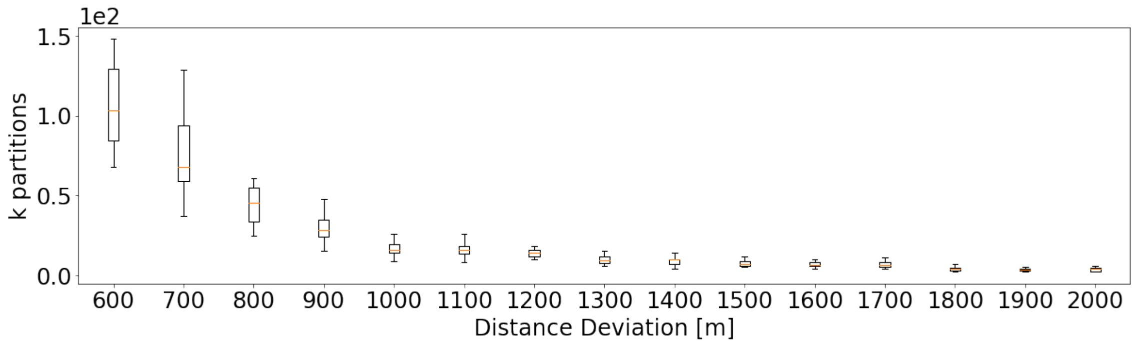


Figure 5.15.: Number of k partitions versus distance deviation for the AOI Krefeld.

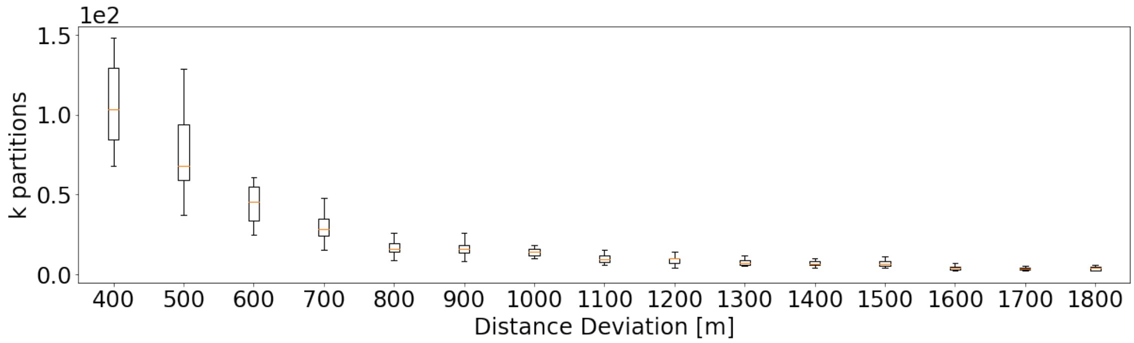


Figure 5.16.: Number of k partitions versus distance deviation for the AOI Málaga.

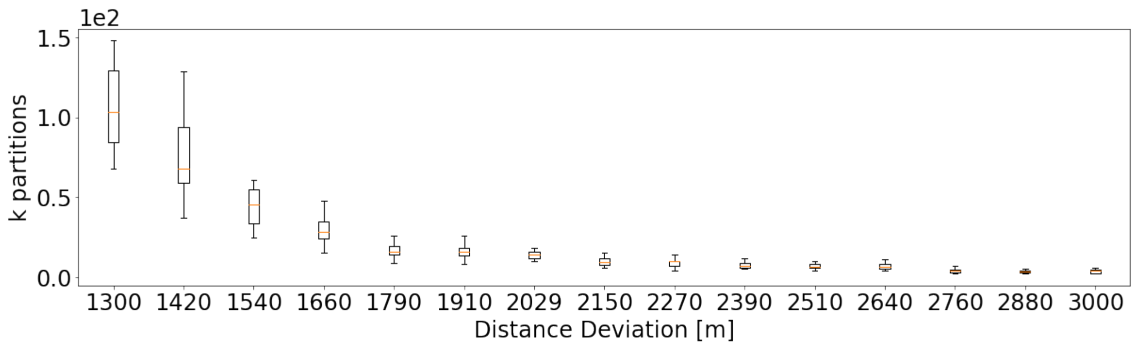


Figure 5.17.: Number of k partitions versus distance deviation for the AOI Soest.

2. Size reduction

Figure 5.18, Figure 5.19, Figure 5.20 and Figure 5.21 visualize the behavior of the ratio of K_{primal} and $K_{reduced}$ for a changing distance deviation. The behavior depicted in the plots seems to be more or less linear, but considering the logarithmic scale on the y-axis, it indicates an exponential scaling. That means, the use of high distance deviations results in an exponential size reduction for all precomputed shortest paths, comparing the primal complete graph with the reduced complete graph.

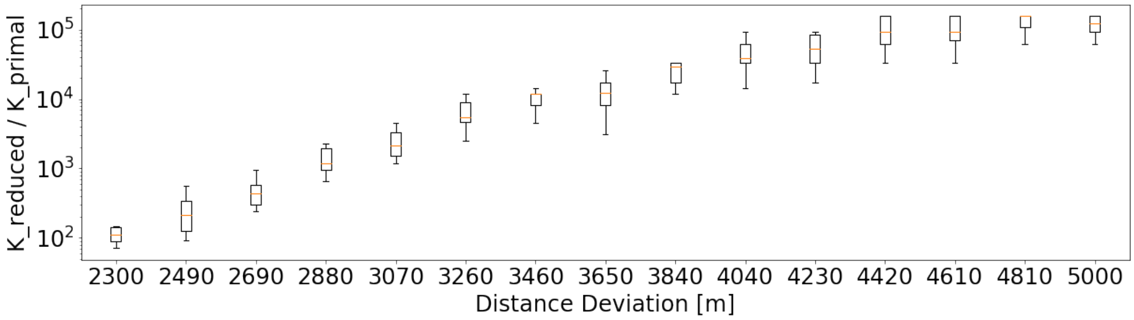


Figure 5.18.: Ratio of complete graphs versus distance deviation for the AOI Göttingen.

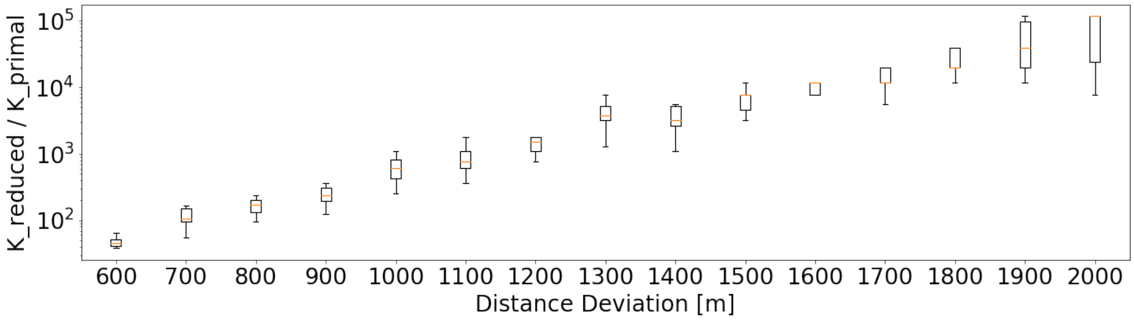


Figure 5.19.: Ratio of complete graphs versus distance deviation for the AOI Krefeld.

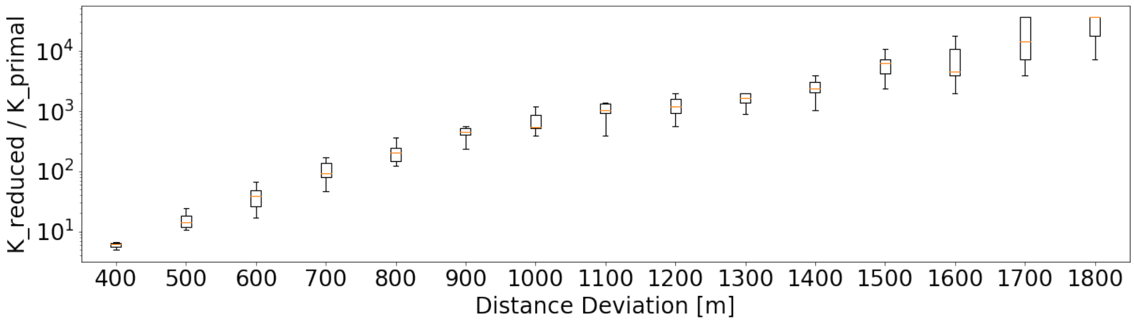


Figure 5.20.: Ratio of complete graphs versus distance deviation for the AOI Málaga.

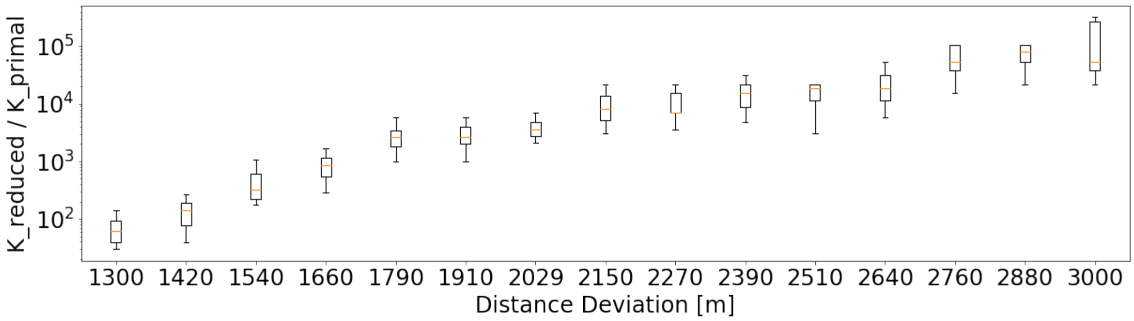
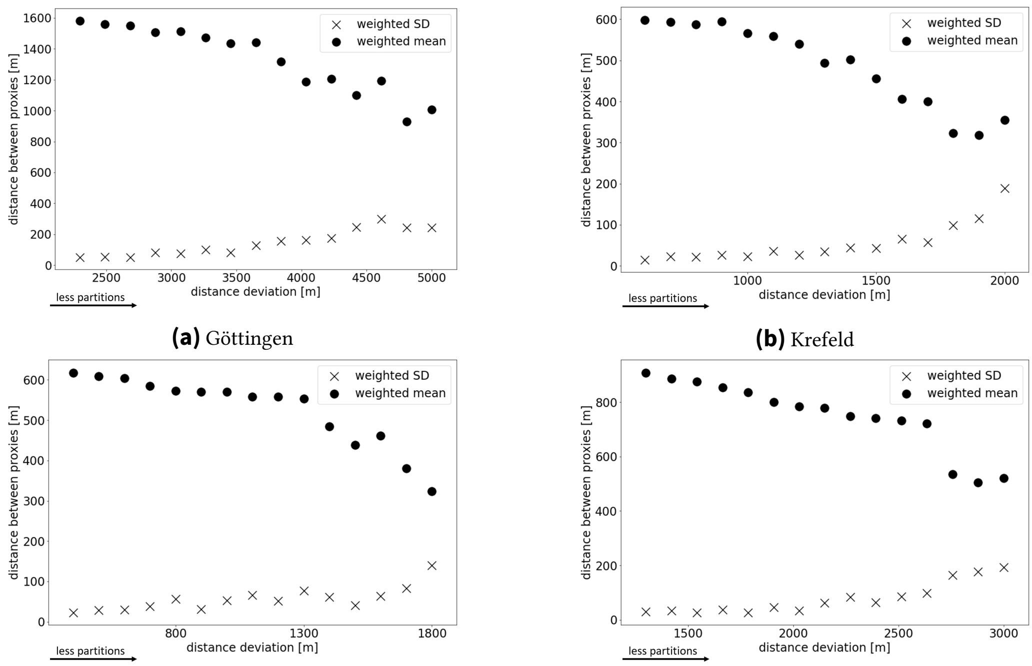


Figure 5.21.: Ratio of complete graphs versus distance deviation for the AOI Soest.

3. All pairs shortest path (APSP)

Figure 5.22 depicts the weighted mean and weighted standard deviation of all shortest path network distances of $K_{reduced}$ for a changing distance deviation. The mean value allows conclusions on the distribution of proxies for different iterations. The resulting distances, which are represented by the weighted mean value, are also influenced by the selected distance deviation. With an increasing distance deviation, the mean value decreases. This behavior can be explained by Figure 5.23. The lower the number of partitions, the lower the total distance of APSP-distances and consequently also the mean distances.

The weighted standard deviation represents the variability of the weighted mean values, which indicates a meaningfulness of the drawn conclusions based on the weighted mean values.



(a) Göttingen **(b)** Krefeld **(c)** Málaga **(d)** Soest

Figure 5.22.: Weighted mean and weighted standard deviation of APSP-distances for a changing distance deviation.

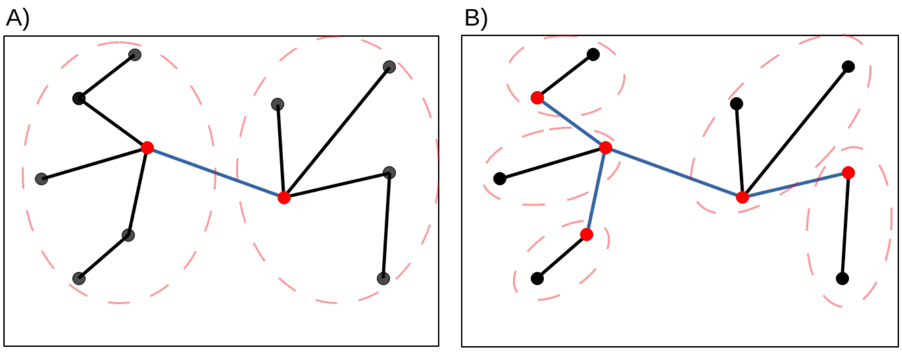
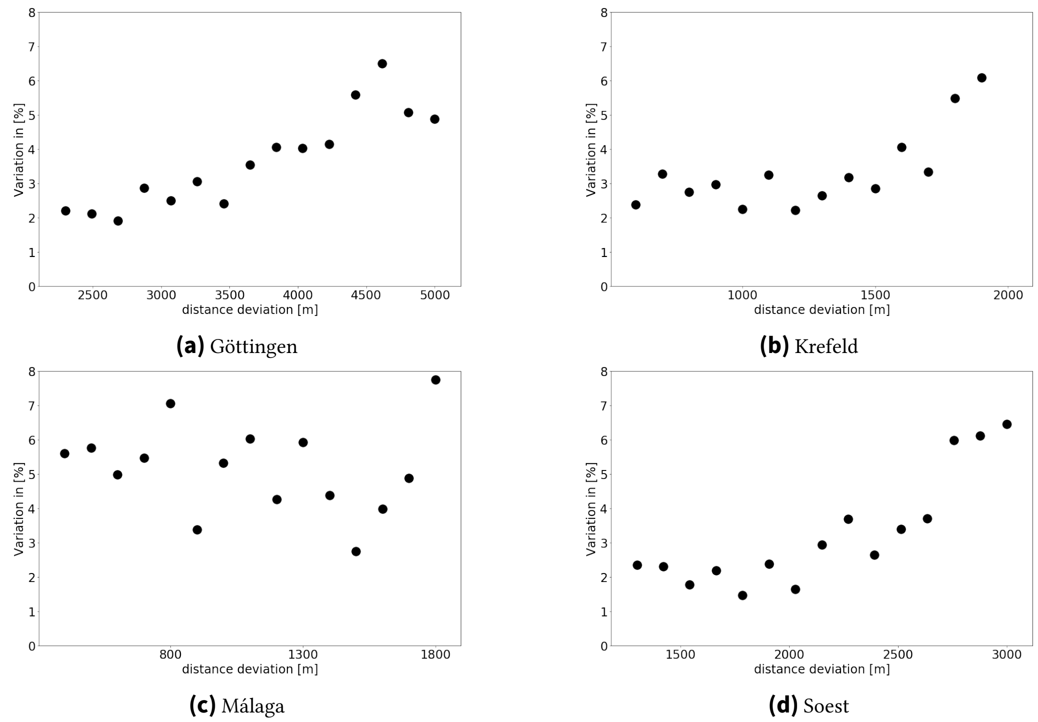


Figure 5.23.: Dependence of all pairs shortest path (APSP)-distances between proxies and the number of partitions. A) Two partitions (red circled area), hence two proxies (red nodes) and all shortest paths between proxies are highlighted in blue. In B) the same network is used, but instead with 5 partitions, resulting in more shortest paths between proxies. Consequently, the more partitions, the larger the value of APSP-distances gets in most cases.

4. Variations of APSP-distances with respect to the distance deviation

In Figure 5.24 the variation of the distance between proxies related to the used distance deviation is shown in percent for each AOI. For Göttingen, Krefeld and Soest a small, linear increasing tendency can be recognized. Nevertheless, the main point of this plot is, that all values are within a range of small values. The highest variation for all areas and distance deviations is 8% (Málaga), while the lowest one amounts to 1.5% (Soest).

The last two plots show that proxies seem to be distributed roughly even on the road network and are not distributed in clusters. This can be related to a problem of common cluster algorithms, when a low minimum of cluster size is needed for density-based clustering algorithms, such as k-means or DBSCAN and an abundance of micro-clusters should be avoided in high-density regions [110]. This could occur when normal clustering algorithms are applied to nodes of the network graph and consequently the primal network graph would be poorly represented by the generalized form. In our presented approach, we can assume that the reduced network $K_{reduced}$ represents somehow realistically the primal network since the proxies are mostly evenly distributed (no clustering) and the centrality measure closeness is used to incorporate a weighting of central and important nodes. Furthermore, the small percentage variation shows that the distance deviation works well as a restrictive parameter and only small deviations occur in the final resulting partitions.



(a) Göttingen

(b) Krefeld

(c) Málaga

(d) Soest

Figure 5.24.: The percentage variation of the distance between the proxies related to the distance deviation.

5. Performance

Figure 5.25, Figure 5.26, Figure 5.27 and Figure 5.28 show the performance of the respective approach for the AOI Göttingen. Special attention must be paid to the scaling on the y-axis since the reduced approach is faster by a factor of about 1000. For every approach, only relatively small variations are evident from the boxplots and the used distance deviation has only a small effect on the performance, since the mean of the boxplots in Figure 5.25 seems to be constant. The distance deviation only influences the performance of the reduced approach (resulting size of the network) but is also shown on the x-axis for the other algorithms for easier comparison.

The scattering, which can be seen from the boxplots in Figure 5.25 and which deviates significantly from the natural deviations, respectively scattering (see boxplots in Figure 5.26, Figure 5.27 and Figure 5.28), can be attributed to the lookup table or linear search. If the searched node pair is at the beginning of the adjacency matrix of the graph, the network distance can be delivered much faster than in case the searched node pair is at the end of the adjacency matrix. This can explain the deviations in performance for the approach based on FluidC and could be improved e.g. by other search algorithms instead of linear search.

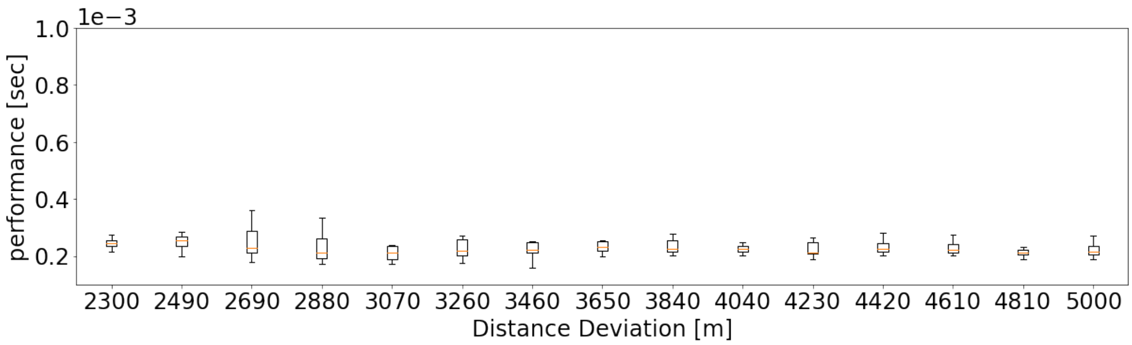


Figure 5.25.: Performance of the reduced approach based on FluidC for the AOI Göttingen.

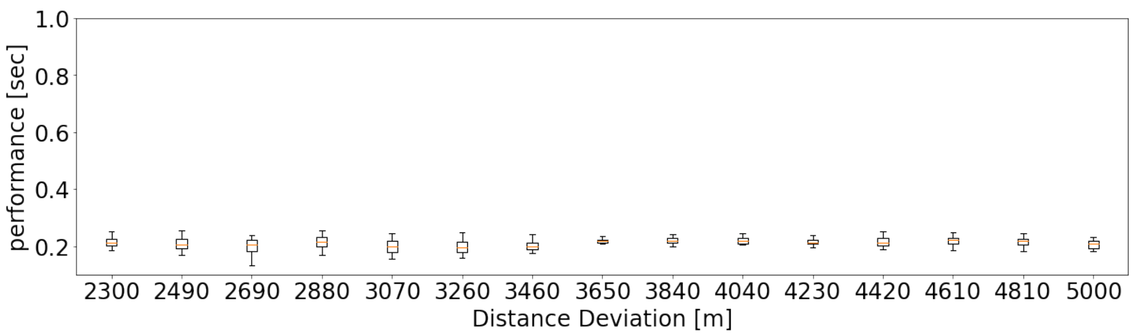


Figure 5.26.: Performance of the Dijkstra algorithm for the AOI Göttingen.

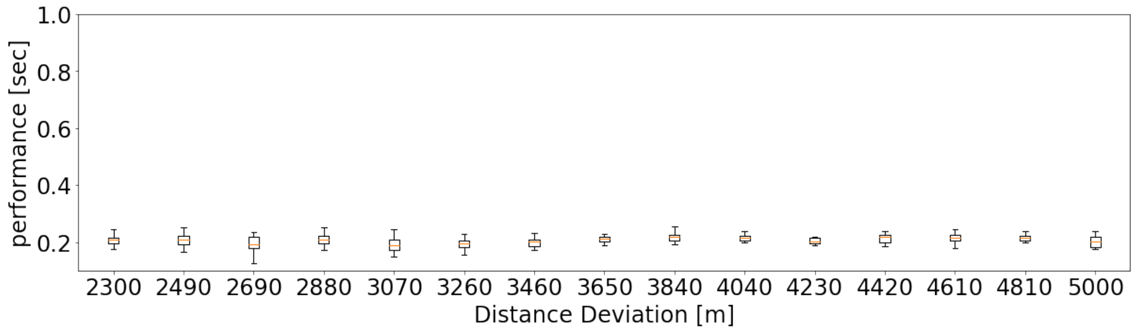


Figure 5.27.: Performance of the bidirectional Dijkstra algorithm for the AOI Göttingen.

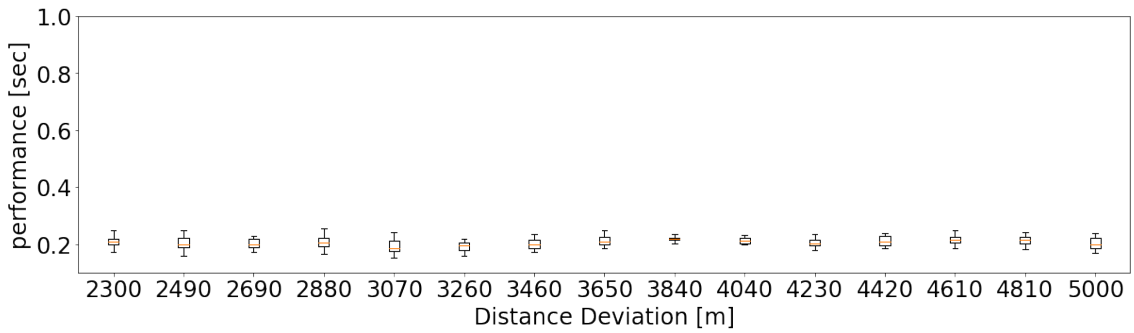


Figure 5.28.: Performance of the A* algorithm for the AOI Göttingen.

The resulting plots for the performance regarding the AOI Krefeld are depicted in Figure 5.29, Figure 5.30, Figure 5.31 and Figure 5.32. Again, we want to highlight the different scaling on the y-axis, when comparing the performance for Krefeld. Overall, for a gridded network such as the road network from Krefeld, the reduced approach based on FluidC seemed to be the most performing algorithm. Except for an outlier for the smallest distance deviation, the performance is mostly constant and faster than the other algorithms. Dijkstra, bidirectional Dijkstra, and A* show a larger scattering and dispersion of values, with the fastest performance being slower than the slowest performance by the approach based on FluidC.

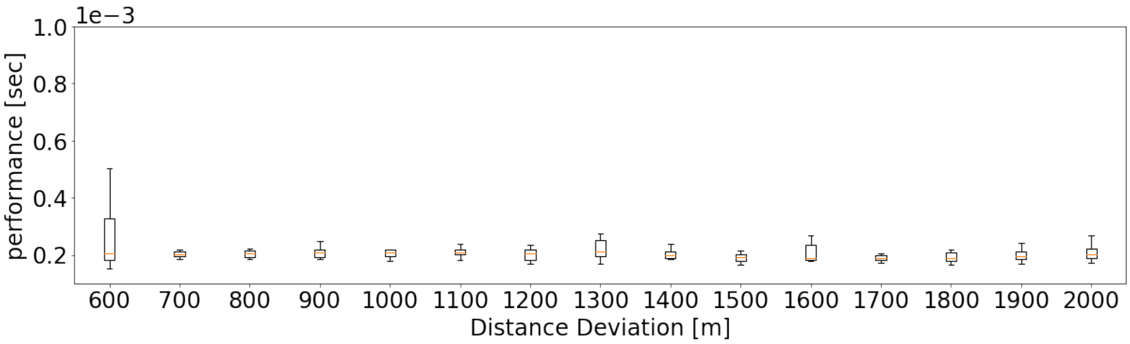


Figure 5.29.: Performance of the reduced approach based on FluidC for the AOI Krefeld.

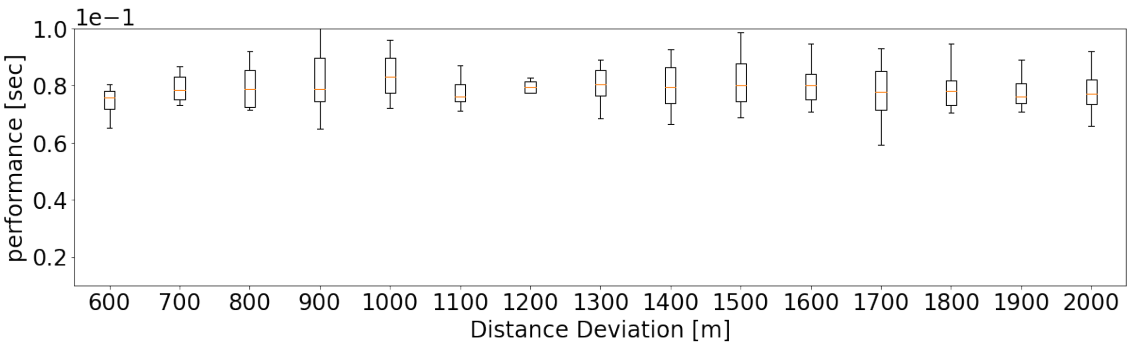


Figure 5.30.: Performance of the Dijkstra algorithm for the AOI Krefeld.

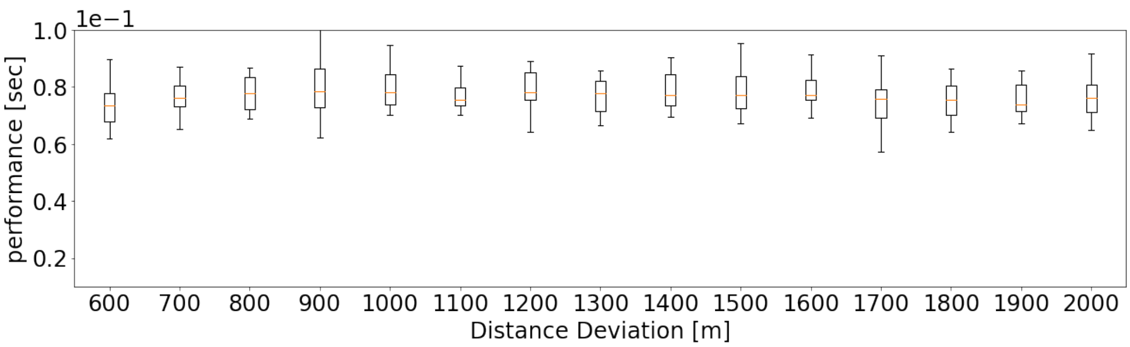


Figure 5.31.: Performance of the bidirectional Dijkstra algorithm for the AOI Krefeld.

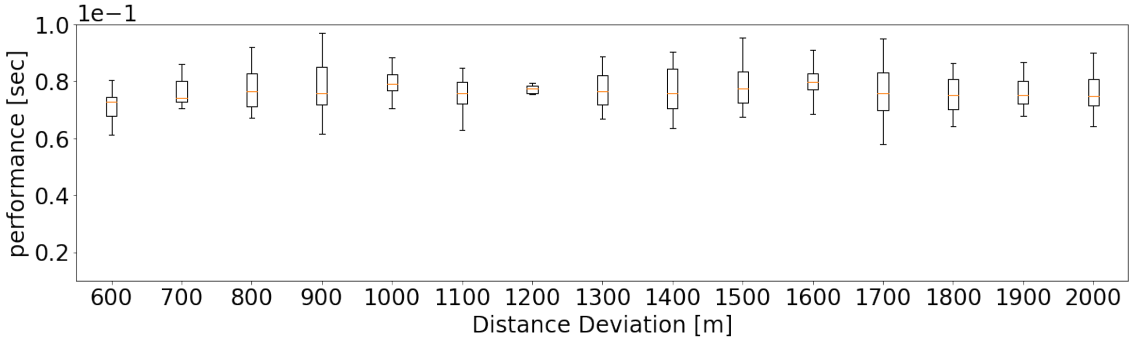


Figure 5.32.: Performance of the A* algorithm for the AOI Krefeld.

In Figure 5.33, Figure 5.34, Figure 5.35 and Figure 5.36 are the results for the performance for the AOI Málaga. For Málaga, a very similar trend to the results to Krefeld can be seen. There are only minimal deviations in the value ranges.

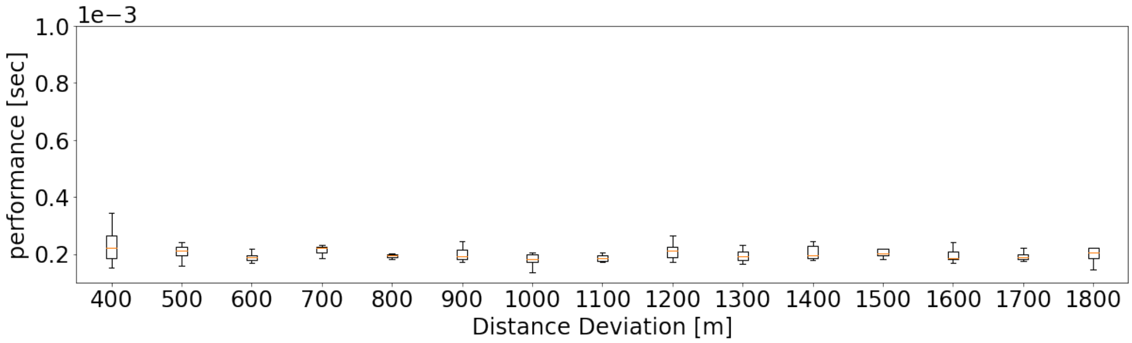


Figure 5.33.: Performance of the reduced approach based on FluidC for the AOI Málaga.

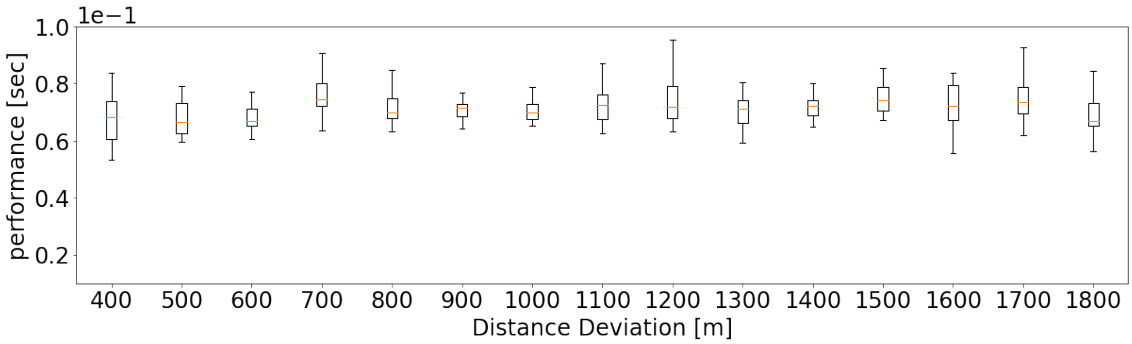


Figure 5.34.: Performance of the Dijkstra algorithm for the AOI Málaga.

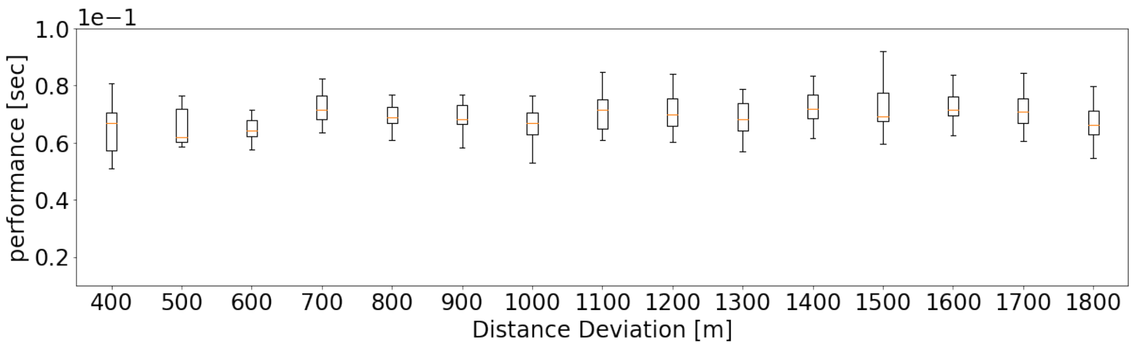


Figure 5.35.: Performance of the bidirectional Dijkstra algorithm for the AOI Málaga.

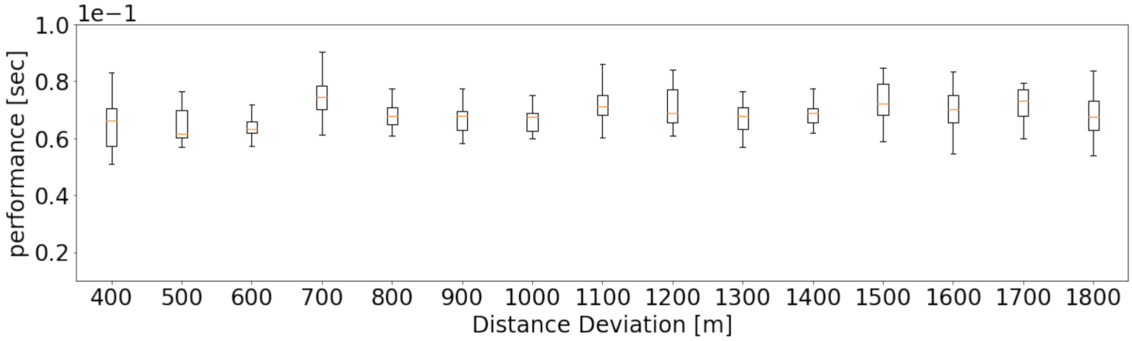


Figure 5.36.: Performance of the A* algorithm for the AOI Málaga.

Figure 5.37, Figure 5.38, Figure 5.39 and Figure 5.40 report the performance for Soest. Dijkstra, bidirectional Dijkstra, and A* seem to perform very similarly on this road network. These algorithms seem to be constant with a small dispersion of values. In contrast, the reduced approach seems to have a larger dispersion. However, the different scaling on the y-axis should be taken into account. Table 5.5 shows the maximum values (absolute and relative) for the interquartile range (IQR) and the range between the lower and the upper whisker of the boxplots. The ratio to the median was used to determine the relative dispersion. It can be seen that the absolute difference for the reduced approach is much smaller than for the other approaches, i.e. there a much smaller differences in the performance. However, the relative differences in relation to the median are larger than for the other approaches. The values in Table 5.5 illustrate the differences between the considered approaches, using Soest as an example.

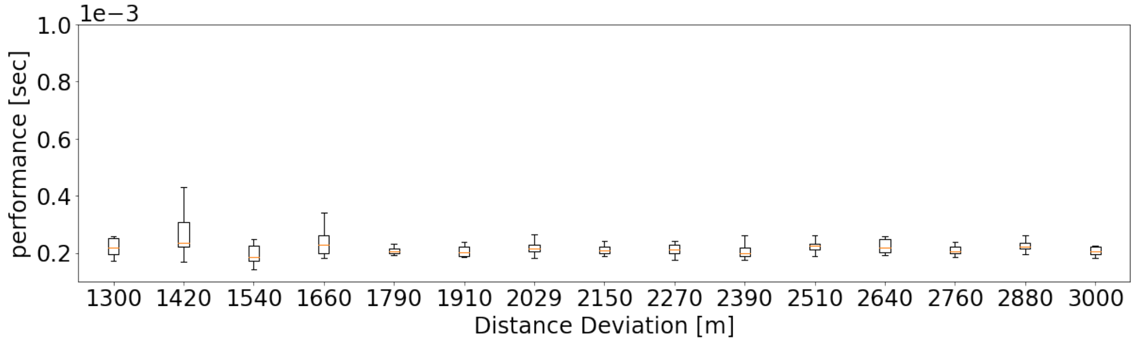


Figure 5.37.: Performance of the reduced approach based on FluidC for the AOI Soest.

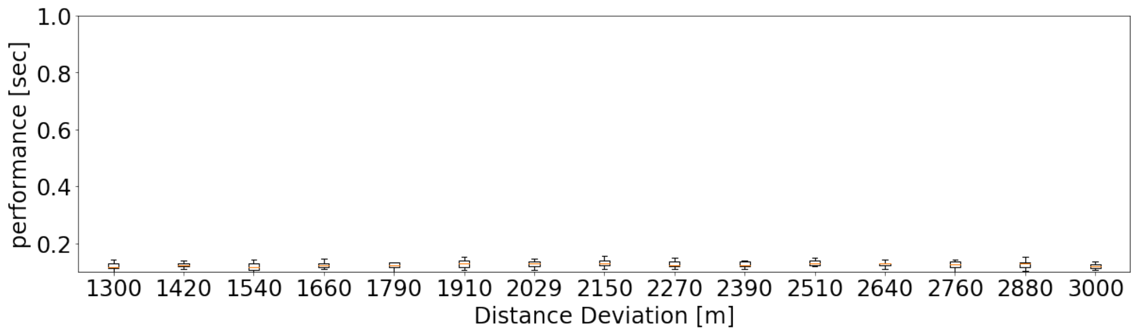


Figure 5.38.: Performance of the Dijkstra algorithm for the AOI Soest.

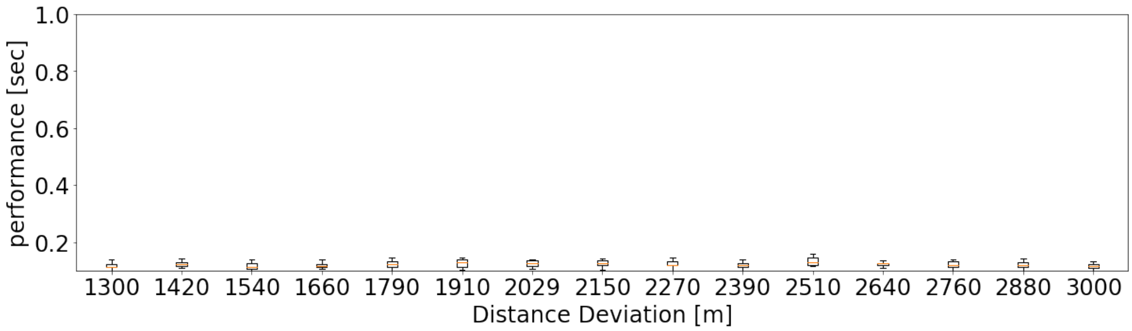


Figure 5.39.: Performance of the bidirectional Dijkstra algorithm for the AOI Soest.

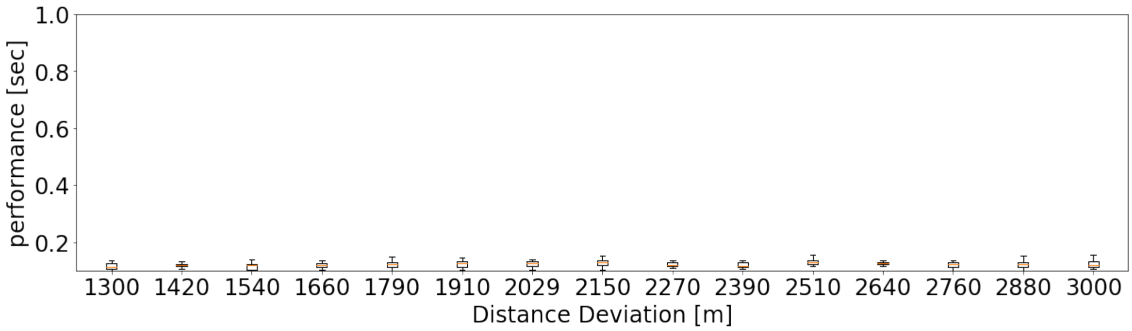


Figure 5.40.: Performance of the A* algorithm for the AOI Soest.

	max. IQR		max. Δ Whiskers	
	absolute	relative [%]	absolute	relative [%]
Reduced	0.00007	36	0.00015	67
Dijkstra	0.02445	19	0.05041	41
Bidirectional Dijkstra	0.02267	19	0.05451	41
A*	0.02234	18	0.06197	51

Table 5.5.: Maximal interquartile range (IQR) and the maximal difference between the lower and the upper whisker of the boxplots for the AOI Soest regarding the performance. The relative values were calculated in relation to the respective median.

5.5. Summary

In this chapter, we have presented an approach that has the potential to provide performant approximated network distances. The approach is based on a modified algorithm for graph partitioning (FluidC) and on the concept of precalculating all pairs shortest path (APSP)-distances. This precalculation is used for the implemented prototype routing, which uses a linear search algorithm. This algorithm scales normally faster than linear and at worst linear (run time $\approx O(n)$), whereas the best competing algorithm considered in this work (A*) scales quadratically (run time $\approx O(n^2)$).

The general concept of the presented approach is a segmentation of the road network into partitions. For each partition, one node, called proxy, is chosen that best represents the respective partition. The degree of generalization is regularized by setting the maximal acceptable distance deviation between a proxy and all other nodes within the respective partition.

For evaluation of this approach, various real-world road networks were used to demonstrate the functionality for different patterns in road networks. The behavior of this approach was examined for several aspects as described in subsection 5.2.8. Multiple iterations were performed to obtain a statistical basis to evaluate the fluctuations of the nondeterministic results of the used algorithm FluidC. By this, the effect of various distance deviations is quantified for the investigated real-world road network patterns.

The results of multiple iterations show, that the pattern of the road network has a very strong influence. For example, the variation of the number for resulting partitions depends on the network, especially for convoluted road networks this fluctuation is high. Consequently, the size of the complete graphs also fluctuates. For more symmetric and gridded-like road networks graphs, the number of partitions varies less.

The homogeneous distribution of the resulting proxies on the road network was investigated, allowing a conclusion whether the generalized network graph realistically represents the original network. Our evaluation shows that the proxies and consequently also the partitions are reasonably evenly distributed on the road network.

In terms of performance, the approach was compared with implementations of some conventional algorithms that provide exact network distances (Dijkstra, bidirectional Dijkstra, and A*). We would like to emphasize here, that this comparison is only meaningful to a certain extent since none of the implementations of the algorithms are designed for maximum performance and exact network distances are compared with approximated network distances resulting from the presented approach. Thus, these results have to be interpreted carefully. Nevertheless, the results show tendencies and allow a rough assessment and comparison of the different approaches.

The performance of the considered algorithms was compared to the presented, also called "reduced approach". This approach always performs best or sometimes similar to the A* algorithm. The Dijkstra and bidirectional Dijkstra algorithms could not compete.

6. Optimized Pick-up and Drop-off Locations

6.1. Central Ideas

Content of this chapter is partly published in Hahn, Frühling, and Schlüter [58] and Hahn, Frühling, and Schlüter [111]. In addition, a publication process in the International Scientific Journal - Transport Problems[1] is in progress.

The challenges for stop locations, especially in passenger transportation, were already introduced in section 2.1. In the current chapter, we present an approach to determine optimized stop locations for transport services using remote sensing data. Therefore, we want to enhance common snapping techniques, that are based on perpendicular distance, which is related to the research area of offline map matching (*cf.* section 3.4). Still, there are minor differences, such as considering a matching name of a given address and surrounding road names. Further, the terms and meaning of map matching and snapping sometimes differ in literature and practice. Therefore, we define the term road snapping, which is characterized by the fact that it only serves the purpose of determining the start and end points of a route on a road network for given addresses or coordinates. These points will be called snapping points.

In the presented approach the already known method of cost distance analysis (*cf.* section 3.5) is used to identify the most likely access to buildings, which in turn results in optimized snapping points, hence stop locations.

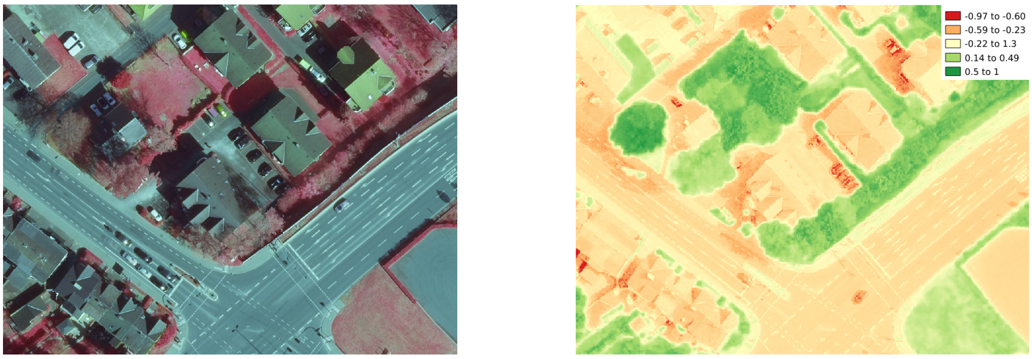
We assume, that the most realistic path from the road network to the building consists of minimal vegetation cover, minimal slope of the terrain, and the path could not go through building footprints.

[1] ISSN 1896 - 0596

6.2. Methods

The method of cost distance analysis was already introduced in section 3.5. We performed such a cost distance analysis with cost surfaces for the parameters vegetation, slope, and building footprints. Therefore, we used multispectral images to determine the vegetation cover based on the vegetation index normalized difference vegetation index (NDVI), light detection and ranging (LiDAR) data for modeling the slope of the terrain, building footprints from OpenStreetMap, and the road network from OpenStreetMap. For the detection of vegetation, NDVI and color infrared (CIR) images (*cf.* Figure 6.1) were considered. CIR images are false color images using the wavelengths for near-infrared (NIR), red, and green. However, the identification of vegetation using the NDVI led to better results, so the CIR images were not considered further.

We used thresholds for the parameters vegetation and slope to distinguish between cells with no-vegetation and vegetation, and between cells with passable or not-passable slope (*cf.* subsection 6.2.3). Consequently, only binary rasters are used instead of continuous data.



(a) Color infrared image

(b) Normalized difference vegetation index

Figure 6.1.: Extract of the AOI with a false color composite color infrared (CIR) image (left) and the vegetation index normalized difference vegetation index (NDVI) to detect vegetation. A true color composite (RGB) of this extent is attached in the Appendix A (Figure A.5).

The road network was transformed to a source raster, where cells represent the existence of road segments, further called source cells. To avoid an unnecessary computation complexity due to a high amount of source cells, we only generated a source cell every 3 meters on the road network, which we still consider as sufficiently accurate (*cf.* Figure 6.2).

Based on the building footprints, a binary raster with cells representing the existence or absence of buildings was created. The centroids of these building footprints will further be called destination cells. Then a merged cost surface is generated, by merging and weighting the cost surfaces of vegetation, slope, and building footprints.

Figure 6.2.: Extract from AOI with the generated source cells as the centroid of the red circles. These source cells were generated with a spacing of 3 meters. This resolution is considered sufficient for accurate snapping.

Further, the accumulative cost surface and the backlink raster were calculated, where each cell in the accumulative cost surface represents the costs from said cell to the source cell, that can be reached with the least cost using the merged cost surface. The complete least cost paths between destination (centroids of buildings) and source (road network) cells can then be generated using the coded direction values in the backlink raster. The last point of the least cost path describes the source cell and thus the snapping point for the corresponding destination cell on the road network.

We evaluated our results by comparing the calculated snapping points using cost distance with the snapping points from the conventional routing engine OSRM[2], which uses the perpendicular distance. Therefore we applied a so-called ideal snapping area (*cf.* section 6.2.4), which defines an area where snapping points are considered correct. This area is based on manually set geographical points, which were used as ground truth data. For the considered AOI we set 495 ideal snapping points. We also evaluated the weighting of the classes vegetation, slope, and building footprints. Only the odd numbers from 1 to 9 were used as weights for each class to reduce the calculation time.

The number of total weight combinations were calculated according to Equation 6.1:

$$iterations_{weighting} = n^k \tag{6.1}$$

[2] `http://project-osrm.org/`

where n is the number of possible weights (1,3,5,7,9) and k is the number of classes (vegetation, slope, and building footprints). Accordingly, we still have 125 different weighting combinations in total, which means 125 iterations of a complete cost distance analysis. As a result, we have validation-rates describing the percentage of snapping points within the ideal snapping area for each weight combination. This enables a detailed analysis of a reasonable weighting and a comparison between the calculated snapping points based on cost distance and the snapping points from OSRM.

We would also like to emphasize, that the weighting of the cost surfaces means higher weighting results in a higher cost, and hence the parameter has less influence on the least cost path. To illustrate this using an example, a high weighting of vegetation costs leads to least cost paths that strictly avoid vegetation, hence to be precise, the cost and not the feature or parameter is weighted.

6.2.1. Area of Interest

A small extent of the used area of interest (AOI) was already shown in Figure 2.4. It is located in the town of Höxter. Höxter is a medium-sized town in the southwest of North Rhine-Westphalia (NRW) in Germany. This city extends over 158.16 km^2 with a population of 29.112 [112]. The used AOI is square-shaped, 1 km^2 in size and located at the centre of Höxter. Most of the landcover in the AOI are residential areas, but there are some industrial complexes in the centre, north-west and east of the AOI. The coordinates confining the geographical extent of the AOI are shown in Figure 6.3. This AOI was chosen due to the availability of high-resolution data and local knowledge of that area.

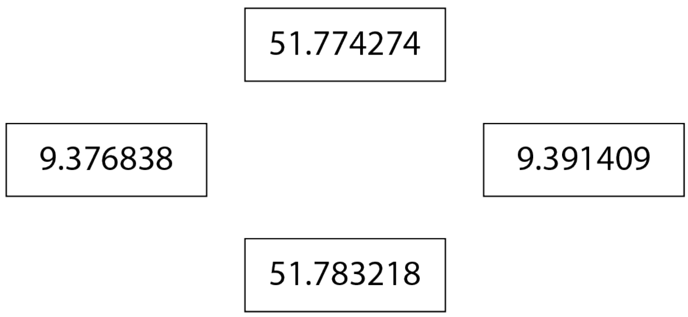


Figure 6.3.: Coordinates confining the extent of the AOI (used EPSG:4326).

6.2.2. Data

We derived the road network and the building footprints from OpenStreetMap contributors [113]. The aerial imagery and light detection and ranging (LiDAR) data can be obtained from the OpenGeoData project [114] for NRW. We filtered the road network data resulting in a road network comprising only publicly accessible roads. The aerial imagery is a multispectral image containing red and near-infrared (NIR) wavelengths, hence the normalized difference vegetation index (NDVI) can easily be calculated with:

$$NDVI = \frac{NIR - Red}{NIR + Red} \tag{6.2}$$

The point cloud from the LiDAR data was used to generate a grid, where each cell represents the slope value of the terrain. Therefore, the point cloud had to be preprocessed. We used the open source tool LAStool[3] to create xyz data and interpolate missing cells before calculating the slope for each cell.

For each considered parameter (vegetation, slope, building footprints, and the road network), a grid representation was generated with a cell size of 0.2 x 0.2 meters, thus the cells of the cost surfaces overlap exactly and cells from cost surfaces and the source raster refer to the same position in the AOI.

6.2.3. Generation of Snapping Points by Cost-Distance

We determine thresholds for the classes vegetation and slope on an empiric basis. The results are shown Table 6.1.

Class	Derived from	Threshold
vegetation	NDVI raster	> 0.2
no-vegetation	NDVI raster	≤ 0.2
passable	slope raster	≤ 11
not passable	slope raster	> 11

Table 6.1.: Classes of the cost distance analysis, the source, and the used thresholds for the binary concept.

As mentioned before, using continuous data without thresholds was also considered, but initial tests and results showed that a binary concept with a clear distinction between e.g. passable or not passable cells regarding the slope leads to clearer least cost paths. These thresholds are based on empirical samples in the study area and should not be applied to other areas without testing. To distinguish cells into vegetation or no-vegetation, the NDVI with a value of 0.2 was used, whereas for the distinction for passable and not passable cells a degree of 11 was chosen.

[3] `http://lastools.org/`

To be able to calculate the least cost paths from building addresses to the road network, the allowed movement of paths has to be defined first. Considering only vertical and horizontal movement can be seen as sufficient. However, an additional diagonal movement is feasible and improves the quality and accuracy of the least cost paths. Consequently, we decided to used the Queens pattern as a neighbourhood type (*cf.* Figure 3.9).

6.2.4. Evaluation of Snapping Points

Ideal snapping points for buildings were predetermined as points on the road network, which are most likely accessible from the building, hence the start point on the road network to the entrance. If a building has more than one possibility to gain access from the road network, multiple ideal snapping points were set. Unfortunately, ground truth data from transport services or entities e.g. pick-up and drop-off locations from taxis are difficult to obtain due to privacy concerns. Consequently, we set 495 reference points as ground truth data using aerial images and local knowledge about the accesses of buildings.

A line from each building's centroid to its ideal snapping point on the road network was generated. This ensures a spatial relation between the two points. The first vertex of the line represents the building's centroid and the second vertex represents the ideal snapping point. This line and its second vertex can be compared to lines from the building's centroid to calculated snapping points by cost distance and to the snapping points obtained from OSRM.

Considering a maximum acceptable distance from the ideal snapping point to calculated snapping points results in a circled area around the ideal snapping point. In the first validation step, we checked if the calculated snapping point is located inside this area. However, if the distance between the building and the road is short, a calculated snapping point might be validated even if the ideal snapping point is in another direction from the building's centroid position, hence on another road. To also consider the direction, in the next validation step, we compare the difference in bearings between the two lines from the building's centroid to the ideal snapping point and to the calculated snapping point. Thus, a maximum acceptable degree for the angle between these two lines was used as a threshold.

This leads to an area around the ideal snapping point which is further called ideal snapping area and is illustrated in Figure 6.4. To validate a snapping point, we checked if the point is located inside this ideal snapping area.

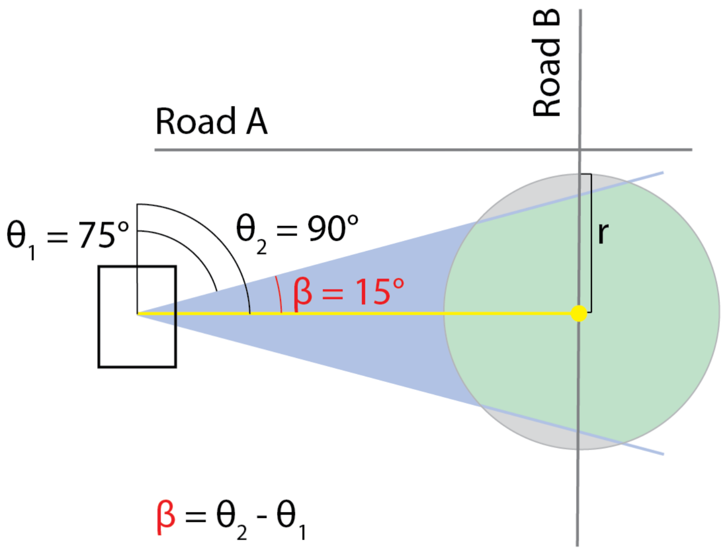


Figure 6.4.: Concept of ideal snapping area. The ideal snapping area is defined by a vector (yellow line) from the centroid of building *B* to the ideal snapping point (yellow point), a maximum distance which is defined by *r* and a direction which is defined by the maximum allowed difference in bearings. In this example, the allowed difference in bearings β between θ_1 and θ_2 is 15°. The ideal snapping area is shown in green. An ideal snapping area restricted only by *r* could lead to acceptable snapping points on Road A and Road B if *r* would be larger.

Consequently, a calculated snapping point is only validated if the difference between the calculated and the manually set ideal snapping point regarding distance and direction is below the predefined threshold. With a growing distance between the points and the same maximum acceptable difference in bearings, the size of the ideal snapping area increases until the area is defined only by the radius of the circle, based on the maximum allowed distance. For the process of evaluation, the maximum acceptable distance between the points was set to 25 meters, and the maximum difference in bearings between the line from the building's centroid to the calculated snapping point and the ideal snapping point was set to 70°.

6.3. Own Software Package and Patent Application

To carry out the cost distance analyses, hence to calculate optimized snapping points and further to perform the evaluation, another framework called Accumulative Cost Surface Analysis (ACSA) was programmed (~2200 lines of python code.) This framework and the technical documentation for required preparation, e.g. setting up the routing engine, preprocessing, and modeling of data, is accessible in the following repository:

`https://github.com/fauceta/ACSA`.

However, the main source code as a simple printout is attached in Appendix B.2. To access the source code for practical use, we recommend the code from the repository.

For the described approach, the Max-Planck-Gesellschaft zur Förderung der Wissenschaften e.V. has filed in the above method as a European patent application[4].

6.4. Results

In the AOI 403 out of 495 calculated snapping points by perpendicular distance using the Nearest API from OSRM are inside the ideal snapping area, which leads to a validation-rate of 81.4%. The calculated snapping points by cost distance show different validation-rates, depending on the weighting of the parameters. Thus, the validation-rate varies from 84.8% to 90.3%.

A detailed analysis of the weighting and the validation-rate allows scoring the weighting of each parameter. Figure 6.5 shows for each weight combination the enhancement of the validation-rate compared to the validation-rate without weighting the parameters. A trend can be seen, that a higher cost of the parameter slope leads to higher validation-rates whereas the lower cost of the parameter slope results in lower validation-rates. For the parameter vegetation and building footprints, no such clear trend can be identified. However, the highest validation-rates were achieved with a medium cost of vegetation and building footprints.

Figure 6.6 depicts the distribution of the validation-rates. All validation-rates of our presented cost distance approach are higher than the validation-rates based on perpendicular distance, with the highest validation-rates being achieved most frequently.

An additional cost distance analysis with more weight combinations for a small extract of the AOI was performed. The same address respectively the same building was used for illustration, as already in Figure 2.4, where insufficient snapping of Google Maps was shown. Figure 6.7 depicts the impact of the weighting on the quality of the least cost paths. The most reasonable least cost path, hence the snapping point is represented by the blue line, whereas the other least cost paths serve as extreme examples and do not reflect reasonable access to the building. However, the red least cost path results in a very similar stop location as from Google Maps.

Figure 6.8 shows least cost paths for multiple buildings in comparison to results from the routing engine OSRM. For the considered buildings, the resulting snapping points based on least cost path and based on perpendicular distance are represented by the

[4] Number: 20180864.9 - 1001

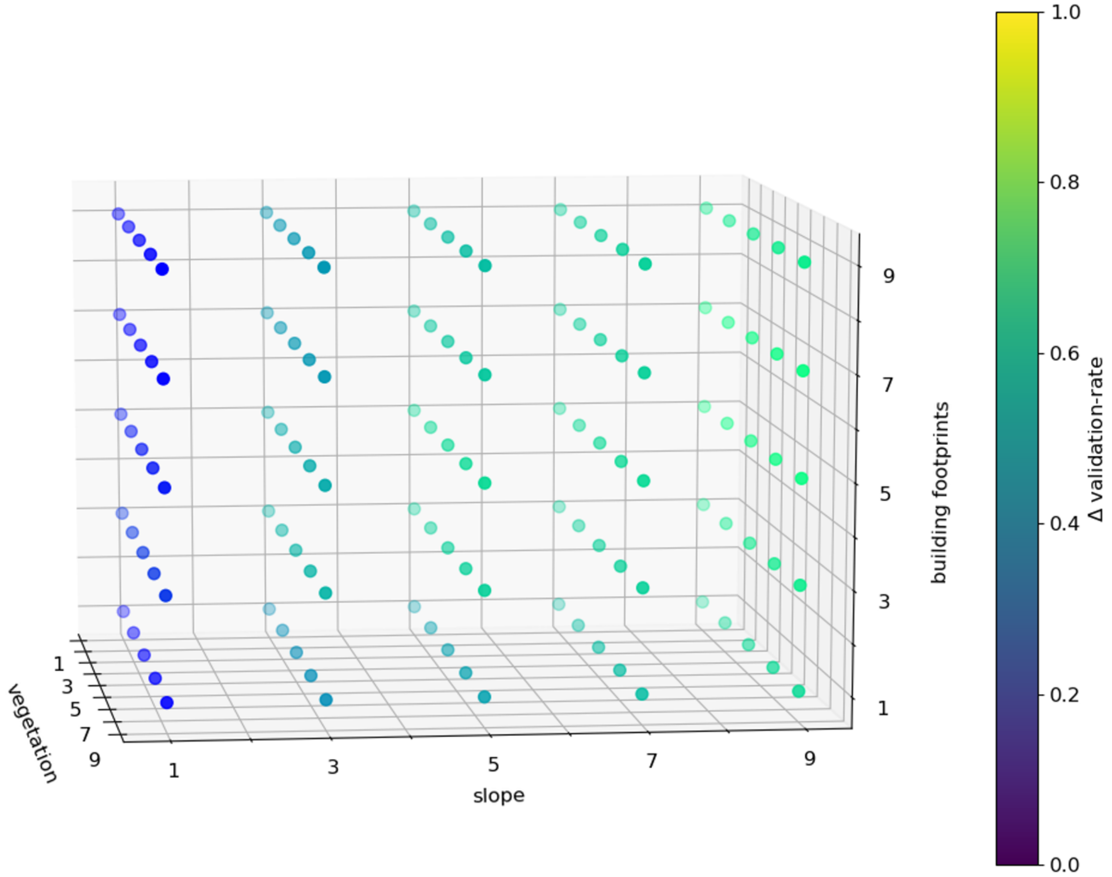


Figure 6.5.: Effects of weights on the validation-rate. For each weight combination, the difference of the validation-rate compared the validation-rate without weighting parameters. Additionally, the data was normalized to a value range between 0 and 1 to allow better visualization.

intersections with the road network. For both approaches, any coordinate instead of only building addresses can be used. This example shows, that the resulting snapping points based on least cost paths are overall more reasonable in the selected extract.

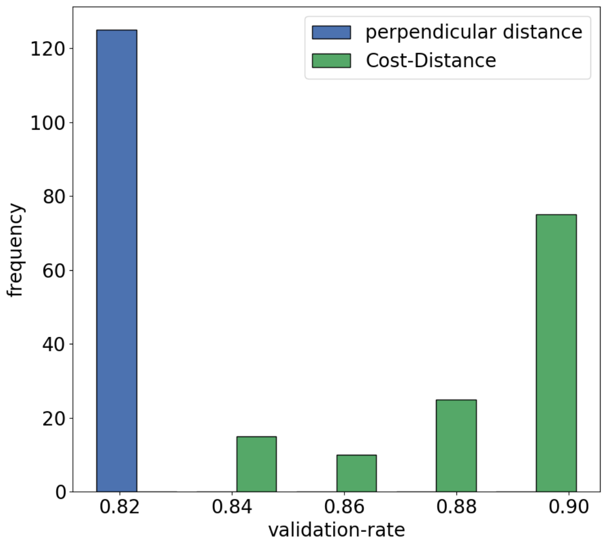


Figure 6.6.: Histogram of the distribution of the validation-rates. In total, 125 cost distance analyses were performed. The validation-rate based on perpendicular distance is not affected by the weighting, therefore in the 125 iterations the validation-rates do not change, whereas the validation-rates for the approach based on cost distance are affected by the weighting.

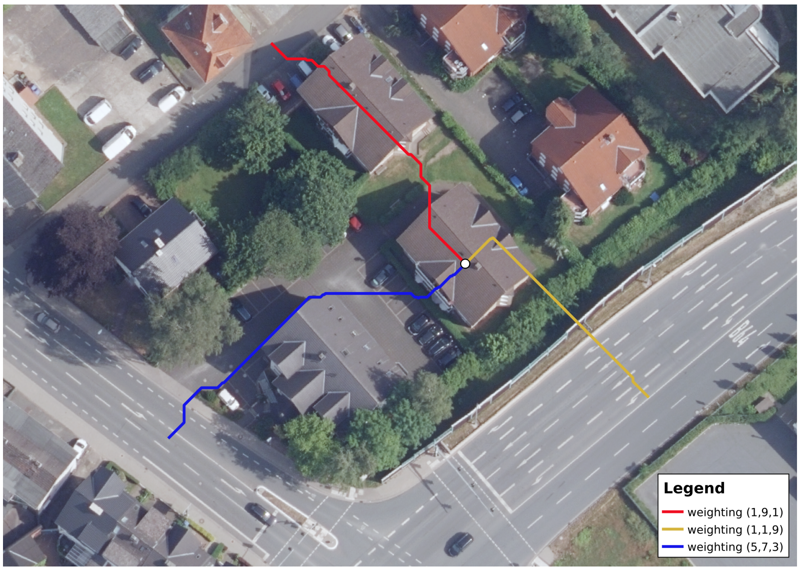


Figure 6.7.: Influence of the weighting using the example from Figure 2.4. The weighting with a cost of 5 (vegetation), 7 (slope), and 3 (building footprints) results in a least cost path (blue) which reflects the realistic access to the building, whereas the other extreme examples show, that important parameters are almost ignored, if not adequately weighted. The red least cost path represents a similar stop location, as retrieved from Google Maps, shown in the example in Figure 2.4.

Figure 6.8.: Multiple least cost paths (blue) in comparison with the results from the routing engine OSRM, which is based on perpendicular distance. The intersections with the roads represent the resulting snapping point, hence stop location for the given address.

6.5. Summary

We presented an approach to identify optimized stop locations for passenger transportation with to-door services based on the common method of cost distance analysis. This can be useful for mobility service providers (MSP) or a transport network company (TNC), which offer such services, especially in combination with public transport or multimodal transportation. Time delays, that are caused by finding a reasonable stop location, can interfere with the plans of future trips and the time schedules. This could be prevented with precalculated optimized stop locations. Therefore, we used remote sensing and carried out the cost distance analysis with the parameters vegetation cover, slope of the terrain, and building footprints. We assumed, that the most likely path from buildings to the road network is characterized by minimal vegetation cover, and minimal slope of the terrain, and that building footprints must not be crossed. These parameters were weighted differently and evaluated to identify a reasonable weighting of these parameters. Further, the resulting snapping points from cost distance were compared to a conventional routing engine (Open Source Routing Machine (OSRM)), which is based on perpendicular distance.

The results show that the approach based on cost distance outperforms the snapping points from the routing engine, as evidenced by the high validation-rates (up to 90.3%), compared to the routing engine (81.4%). Furthermore, the highest validation-rates are achieved more frequently (*cf.* Figure 6.6).

Given the high computational complexity of cost distances analyses, an application of the presented approach is particularly suitable for a limited area in combination with a one-time preprocessing, where the corresponding snapping points are calculated and stored for each address or even for every pixel of the considered area. For dynamic adjustments such as changes in the road network due to road closures or construction zones, the cost distance analysis would have to be performed again. However, on the one hand, road closures and construction zones rarely affect road snapping and on the other hand, in the precalculated result of the one-time preprocessing, the affected areas could be adjusted manually.

For the considered area of interest, the evaluation showed that the slope parameter should have a particularly high cost in order to obtain the most reasonable snapping points. For the other parameters, there is no clear trend and the best results were obtained when these parameters had medium costs. Still, these results are valid only for the considered study area and cannot be easily transferred to other areas. The given conditions in an area such as the vegetation cover, the topography, and the built-up area have an influence on a reasonable weight combination. For example, slopes of hedges, bushes, and fences in a rather flat area are a very good indicator to determine the access to buildings, whereas, in a more mountainous area, the access to buildings may also have a higher slope. Consequently, this parameter should have less influence there.

7. Discussion

In the context of the supported DRT projects, we tried to identify all the potentials for the improvement in modern, flexible passenger transportation. The evaluation of the projects covered theoretical aspects as well as very real problems in operation. From the viewpoint of geoinformatics, two challenges seemed worth to be investigated in more detail in this thesis.

The first one is the up to now common use of Euclidean distance for calculations of network distances, whereby inherent pitfalls may become relevant when the real distance on the road network differs markedly from Euclidean distance. The second is to assign appropriate locations to the requester's location for passenger boarding. Both, generalization of road networks for the purpose of simplified routing as well as the use of geospatial data (e.g. remote sensing data) for optimizing the accuracy of start and end points of calculated routes were addressed. As different as these topics may seem, both can contribute significantly to an enhancement of modern passenger transport systems.

In chapter 5, an own approach was presented, with which approximated network distances can be easily determined and this approach should be seen as an alternative to the Euclidean distance in transportation practice and transportation research. Our evaluation showed, that the presented approach shows potential for further usage and that it works for different road network patterns. In Chapter 6, we introduced an approach to determine optimized stop locations, using the method of cost distance analysis and remote sensing data. We assumed, that the path from buildings to the road network consists of few vegetation cover, minimal slope of the terrain, and that building footprints should not be crossed. We compared our results to a conventional routing engine, which is based on the error-prone perpendicular distance. We achieved a higher validation-rate (up to 90.3%) than the conventional routing engine (81.4%) and we could evaluate a reasonable weighting of the used parameters.

In the following, we will relate the used methods to approaches from literature, evaluate the used methodology, hence highlighting disadvantages and advantages and we recommend optimization potentials and future work. Therefore, we will distinguish between the work regarding an enhancement of the performance for network distance computations (*cf.* chapter 5) and the optimization for pick-up and drop-off locations (*cf.* chapter 6).

References to previous research

Performance of network distance computations
The presented approach shows potential to replace the usage of Euclidean distance in transportation and transportation research. Euclidean distance is still used today to determine the distance between two points on the road network, shown among others in [18, 1, 19]. Even if this method can be feasible for road networks with a small circuity value, pitfalls as depicted in Figure 2.1 may occur. Reasons for the use of Euclidean distances in these fields are either historically caused [15] or due to less required computing power compared to a determination of exact network distances. The calculation of exact network distances can still be very time-consuming and costly if many calculations have to be performed for parallel queries. Therefore, Maue, Sanders, and Matijevic [26] suggest as an extreme way to precalculate all pairs shortest path (APSP)-distances, so no calculations need to be performed and results can be looked up. However, this is not feasible for large networks. This is why we and other approaches from the literature use graph partitioning of the underlying data to reduce the required computational power for solving such queries (*cf.* section 4.1). In the conventional approaches for partitioning of road networks, most approaches from literature do not consider methods cross-disciplinary for the partitioning. Since both, graph theory and graph partitioning have applications in many different areas of science, there are plenty of different methods. We refer to the literature presented in section 4.1. To the best of our knowledge, no algorithm that is actually intended for community detection in social networks has been applied to road network partitioning so far. The concept of the algorithm FluidC we adopted from social sciences, may be suitable for the partitioning of road networks. From this point of view, our approach is innovative. As already mentioned in the introduction of this algorithm (*cf.* subsection 5.2.3), the basic idea is, that fluids interact with each other and contract and expand until a balanced state is reached.

In section 4.1, we presented a brief literature review addressing the main concepts of the considered approaches for conventional road network partitioning in the context of routing enhancement. In our developed approach, we combine some advantages and benefits published in the literature. The published concepts of partitioning the road network and the basic idea of using proxies have been proven successful in the past. Jung and Pramanik [93], Yu, Lee, and Munro-Stasiuk [69], and Xu and Jacobsen [95] present an approach that uses graph partitioning for routing enhancement. They determine so-called distance-preserving subgraphs (DPS), which can be seen as an equivalent to partitions (*cf.*

section 4.1). However, there are some drawbacks in their concept, since the applicability is limited to predefined targets, e.g. logistic hubs. Therefore, these approaches are not suitable for flexible passenger transportation systems.

Further similarities between our approach and the conventional ones from literature are precalculations of shortest path distances. Maue, Sanders, and Matijevic [26] use precomputed cluster distances (PCD) and lookup tables and Eapen and Beegom [96] and Ma *et al.* [97] use so-called deterministic routing areas (DRAs).

Most previous approaches have mainly used only gridded and symmetrically constructed networks and did not evaluate their algorithms for different road network patterns. So Jung and Pramanik [93], Yu, Lee, and Munro-Stasiuk [69], and Xu and Jacobsen [95] have only used grid networks and other approaches from literature used for example continental-sized road networks [91], where the difference between urban or rural road network patterns get less relevant on the large scale since only larger highways are considered. Both examples are hardly suitable for the application for flexible passenger transport systems.

A direct quantitative comparison of own results to results from the literature was not performed, because the used road networks differ, a quantitative evaluation of partitioning is not standardized, and the used implementations in different programming languages are not comparable. Instead, a detailed, quantitative evaluation was performed for real road networks.

We want also to mention for the sake of completeness that there are other approaches such as multilevel graph partitioning (MGP) and the software METIS, SCOTCH, and KaHIP (*cf.* section 4.1). The original purpose of these approaches is mostly the optimization for parallel computing in computer sciences. Also, some of the algorithms are protected by patents, and implementations of these algorithms are not easily accessible. Our approach, on the other hand, can be easily used and modified.

<u>Optimzed pick-up and drop-off locations</u>
Inaccuracies in routing in the context of passenger transportation can lead to problems such as misunderstandings between customers and drivers and time delays in the schedule. Especially inaccuracies at the start and end points of a route, hence pick-up and drop-off locations can lead to problems. This can be caused by incorrect, inaccurate, or missing map data as well as by an insufficient road snapping technique.

In research, there has been little focus on improving road snapping for the purpose of better stop locations. There are many publications on real-time and offline map matching (*cf.* section 3.4), but they have a different intention and do not consider the entrances to buildings at the start and endpoints of a route to determine reasonable stop locations.

The publication from Hu *et al.* [106] is one of the few that addresses the same issue as described in Figure 2.2. They try to identify the entrances of buildings in order to optimize pedestrian routing to public buildings. Therefore, they use statistical learning methods for building footprints from OpenStreetMap data. To the best of our knowledge, our presented approach is the only one that uses remote sensing for this goal. Hu *et al.* [106] plan to also use satellite imagery complementary to their approach in the future, but they have exploited little potential so far. According to Hu *et al.* [106], most approaches in the literature for determining building entrances are based on the analysis of street-level images such as image recognition of Google Street-View images. Such data and its applications are often limited and therefore currently not used in the context of passenger transport systems.

Other approaches from literature in the context of flexible passenger transport systems have somehow circumvented the limited accuracy of to-door routing, e.g. by determining meeting points [18] or by using predefined stops as used in the supported project Flexa[1]. To-door services have been mainly reserved for taxi companies, that do not require accurate routing due to local knowledge. For private use, the demand for a more accurate routing was apparently not high enough so far, as a small delay due to searching for the building entrance and stop locations is unlikely to have serious consequences. In transportation services, even small delays can interfere with the plans of future trips and time schedules. Especially in combination with intermodal trip planning, this can result in missing connection trains.

Another aspect already mentioned, is missing or incorrect map data (*cf.* section 2.2). Many routing engines use data from OpenStreetMap. Consequently, the quality of the results from the routing engine depends on the quality of the map data. If e.g. house numbers are missing, the centroid of the road is used for routing, which leads to very large inaccuracies, especially on long roads. There are services that offer the coverage of tagged house numbers in OpenStreetMap for some areas[2]. However, improving the coverage of tagged house numbers is mostly still done by the users by hand, as described in this blog [115].

Funke, Schirrmeister, and Storandt [105] published an approach for automatic extrapolation of missing OpenStreetMap data, focusing on missing street names. Nevertheless, the experience gained in the supported pilot projects showed, that most work still has to be done by hand. Despite that, the quality of OpenStreetMap data for routing could compete in the considered areas with commercial alternatives such as HereMaps[3], especially regarding the accuracy of constructions and road closures.

[1] https://www.l.de/verkehrsbetriebe/kundenservice/services/flexa
[2] https://regio-osm.de/
[3] https://www.here.com/

Critical view of methodology and future work

Performance of network distance computations
We are aware, that in our evaluation exact shortest path distances are compared with approximate shortest path distances, which need to be interpreted with caution. Furthermore, the parameters used have been chosen to show the potentials and the functioning of the presented approach. We did not primarily determine parameters for practical application, since depending on the AOI and given requirements, such as the maximal acceptable distance between proxies and all nodes within the same partition, the parameters can be completely different.

For future work, we suggest investigating the presented approach in more detail for an application in transportation practice. Also, the suitability of this approach for other disciplines could be investigated, too. It is conceivable, that this approach can also be applied to e.g. social networks or other disciplines using network graphs. Especially, in the context of big data, a generalization of network graphs can be worthwhile.

Optimized pick-up and drop-off locations
The presented approach for determining optimized stop locations is somehow limited for applications by the required computation time. An application can therefore be recommended if the calculation is done as a preprocessing and for a bounded area to restrict the complexity. Both, the application and further research on this approach are limited due to data availability. More ground truth data would be preferable for a more extensive evaluation. Data from taxi companies are difficult to obtain due to privacy issues and if such data are available, the other required data, such as LiDAR data are often not freely accessible or not affordable in a sufficient resolution.

Further research could concentrate on improving the calculation time, and testing and evaluating this approach for different regions as well as for the interpolation of house numbers. The cell size of 0.3 x 0.3 meters could be changed to reduce the computation time. However, when choosing the resolution, the guideline from Shannon, Whittaker, and Nyquist should be taken into account, which states that the cell size of a grid should be at least $2 * \sqrt{2}$ times smaller than the smallest detail to be kept [116, 117]. Considering the grid representation (*cf.* Figure 3.9), another pattern could be used, but it would hardly lead to any advantages. Consequently, we do not recommend changing the chosen Queens pattern. In addition, interpolation of missing house numbers should be investigated. Therefore, it should be assumed that there is a clear rule in the assignment of house numbers, such as odd house numbers on the one side and even house numbers on the other side of the road. Then, for buildings the assigned roads can be determined by the cost distance method and the rule for assigning house numbers can be applied. Such interpolated house numbers should be interpreted with caution, since there are sometimes further address additions, e.g. characters, and a certain inaccuracy of the results can be expected. However, the most realistic application for this approach is for DRT systems with a small or medium-sized area. For each address in this area, snapping points could be precalculated and stored, so they could easily be read when needed without calculating the snapping points and preventing the usage of the error-prone perpendicular distance. This would also allow manual adjustments for stop locations if these points would be stored e.g. in a lookup table since manually modified snapping can be implemented.

8. Conclusion

In this thesis, we picked up two challenges we identified to have the potential to improve modern, flexible passenger transport systems by using methods from geoinformatics. First, an alternative to the Euclidean distance was developed by providing fast approximated network distances. Second, an approach for determining enhanced stop locations for passenger transport systems with to-door services was set up by using remote sensing. These two challenges were addressed in an interdisciplinary manner so that methods and strengths from different disciplines were combined.

A new flexible and robust approach was presented, that generalizes complex network graphs such as road networks, and can further provide approximated network distances without having to resort to routing engines, respectively shortest path algorithms. They can require a lot of computing power, especially for many parallel queries and for large road networks. In the evaluation, different aspects of the used approximation algorithm and hence resulting nondeterministic results were investigated. These aspects consist of deviations of the number of partitions, the potential reduction for precalculating all pairs shortest path (APSP)-distances, distribution of partitions, and the performance in comparison to conventional shortest path algorithms. For each of these aspects, statistics were used to describe the behavior of our algorithm. Due to the design of approximation algorithms or precisely the used FluidC algorithm, nondeterministic results can arise. For this reason, the statistical investigations were performed for several iterations with unchanged parameters. Further, the scalability of this approach was also considered and investigated. It was shown, that the presented approach copes well with different road network patterns, which is often neglected in research. Comparable approaches from literature often evaluate their algorithms only on symmetric or constructed network graphs, such as symmetrically gridded road networks, which reflect only the results for Manhattan-like road networks, but not for many other real-world road network patterns. By testing algorithms on irregular road networks, an interpretation and evaluation is more difficult, but rather show the potential application for practical use. The comparison of the performances indicates, that the presented approach has potential, but this should be investigated more closely in the future by exploiting an improved implementation in other, more performant programming languages.

Furthermore, in chapter 6 an approach to determine the access to buildings using the method of cost distance analysis and remote sensing data, was presented. This technique can be used to determine optimized stop locations, that could potentially be applied to e.g. demand responsive transport (DRT) systems with a to-door service. In conventional routing engines, such as Google Maps or Open Source Routing Machine (OSRM), sometimes dangerous stop locations at heavily trafficked highways or insufficient stop locations without direct access to the corresponding building are calculated, which could lead to delays and misunderstandings between drivers and passengers. Such problems could be avoided by the presented approach. To the best of our knowledge, a research gap has been identified here, since little comparable research exists and even state-of-the-art routing engines like Google Maps provide inaccurate snapping points.

In a comparison of our approach and the conventional routing engine OSRM, which computes the stop locations based on perpendicular distance, the results show that our approach outperforms the conventional alternative. We could achieve a validation-rate up to 90.3%, whereas the conventional routing engine reaches a validation-rate of 81.4%.

In this work, we have shown some potential contributions that geoinformatics can offer to make efficient passenger transportation systems more attractive compared to motorized private transport (MPT), thus contributing further to mitigating the developments of the anthropogenic climate change.

Bibliography

[1] N. Masoud and R. Jayakrishnan. "A real-time algorithm to solve the peer-to-peer ride-matching problem in a flexible ridesharing system". In: *Transportation Research Part B: Methodological* 106 (Dec. 2017), pp. 218–236. DOI: 10.1016/j.trb.2017.10.006. URL: https://doi.org/10.1016/j.trb.2017.10.006.

[2] M. Tamannaei and I. Irandoost. "Carpooling problem: A new mathematical model, branch-and-bound, and heuristic beam search algorithm". In: *Journal of Intelligent Transportation Systems* 23.3 (Nov. 2018), pp. 203–215. DOI: 10.1080/15472450.2018.1484739. URL: https://doi.org/10.1080/15472450.2018.1484739.

[3] R. Sims *et al.* "Transport". In: *Climate Change 2014: Mitigation of Climate Change. Contribution of Working Group III to the Fifth Assessment Report of the Intergovernmental Panel on Climate Change.* Ed. by Edenhofer O. *et al.* Cambridge, United Kingdom and New York, NY, USA: Cambridge University Press, 2014. Chap. 8, pp. 599–670.

[4] EDGAR - Emissions Database for Global Atmospheric Research. *2019 Fossil CO2 Total Emissions.* https://edgar.jrc.ec.europa.eu/. 2019.

[5] European Parliament. *CO2 emissions from cars: facts and figures (infographics): News: European Parliament.* https://www.europarl.europa.eu/news/en/headlines/society/20190313STO31218/co2-emissions-from-cars-facts-and-figures-infographics. 2019.

[6] O. F. Aydin, I. Gokasar, and O. Kalan. "Matching algorithm for improving ride-sharing by incorporating route splits and social factors". In: *PLOS ONE* 15.3 (Mar. 2020). Ed. by Chen Lv, e0229674. DOI: 10.1371/journal.pone.0229674. URL: https://doi.org/10.1371/journal.pone.0229674.

[7] S. Fahnenschreiber *et al.* "A Multi-modal Routing Approach Combining Dynamic Ride-sharing and Public Transport". In: *Transportation Research Procedia* 13 (2016), pp. 176–183. DOI: 10.1016/j.trpro.2016.05.018. URL: https://doi.org/10.1016/j.trpro.2016.05.018.

[8] J. Ke, H. Yang, and Z. Zheng. "On ride-pooling and traffic congestion". In: *Transportation Research Part B: Methodological* 142 (Dec. 2020), pp. 213–231. DOI: 10.1016/j.trb.2020.10.003. URL: https://doi.org/10.1016/j.trb.2020.10.003.

[9] A. König and J. Grippenkoven. *From public mobility on demand to autonomous public mobility on demand – Learning from dial-a-ride services in Germany*. Ed. by Eric Sucky *et al.* 2017. URL: https://elib.dlr.de/104956/.

[10] J. Alonso-Mora *et al.* "On-demand high-capacity ride-sharing via dynamic trip-vehicle assignment". In: *Proceedings of the National Academy of Sciences* 114.3 (Jan. 2017), pp. 462–467. DOI: 10.1073/pnas.1611675114. URL: https://doi.org/10.1073/pnas.1611675114.

[11] P. Jittrapirom *et al.* "Mobility as a Service: A Critical Review of Definitions, Assessments of Schemes, and Key Challenges". In: *Urban Planning* 2.2 (June 2017), pp. 13–25. DOI: 10.17645/up.v2i2.931. URL: https://doi.org/10.17645/up.v2i2.931.

[12] S. Böhler. *Handbuch zur Planung flexibler Bedienungsformen im ÖPNV*. Dec. 2020. ISBN: 9783879940387.

[13] H. Bast *et al.* "Route Planning in Transportation Networks". In: *Algorithm Engineering*. Springer International Publishing, 2016, pp. 19–80. DOI: 10.1007/978-3-319-49487-6_2. URL: https://doi.org/10.1007/978-3-319-49487-6_2.

[14] S. Wang *et al.* "Effective Indexing for Approximate Constrained Shortest Path Queries on Large Road Networks". In: *Proc. VLDB Endow.* 10.2 (Oct. 2016), pp. 61–72. ISSN: 2150-8097. DOI: 10.14778/3015274.3015277. URL: https://doi.org/10.14778/3015274.3015277.

[15] D. Levinson and A. El-geneidy. *Network Circuity and the Location of Home and Work*. 2007.

[16] J. Huang and D. Levinson. "Circuity in urban transit networks". In: *Journal of Transport Geography* 48 (Oct. 2015), pp. 145–153. DOI: 10.1016/j.jtrangeo.2015.09.004. URL: https://doi.org/10.1016/j.jtrangeo.2015.09.004.

[17] S. Shang *et al.* "Trajectory similarity join in spatial networks". In: *Proceedings of the VLDB Endowment* 10.11 (2017), pp. 1178–1189. ISSN: 21508097. DOI: 10.14778/3137628.3137630.

[18] P. Czioska *et al.* "Real-world meeting points for shared demand-responsive transportation systems". In: *Public Transport* 11.2 (July 2019), pp. 341–377. DOI: 10.1007/s12469-019-00207-y. URL: https://doi.org/10.1007/s12469-019-00207-y.

[19] B. Shen *et al.* "V-Tree: Efficient kNN Search on Moving Objects with Road-Network Constraints". In: *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, Apr. 2017. DOI: 10.1109/icde.2017.115. URL: https://doi.org/10.1109/icde.2017.115.

[20] N. Agatz *et al.* “Optimization for dynamic ride-sharing: A review”. In: *European Journal of Operational Research* 223.2 (Dec. 2012), pp. 295–303. DOI: 10.1016/j.ejor.2012.05.028. URL: https://doi.org/10.1016/j.ejor.2012.05.028.

[21] M. Hyland and H. S. Mahmassani. “Operational benefits and challenges of shared-ride automated mobility-on-demand services”. In: *Transportation Research Part A: Policy and Practice* 134 (Apr. 2020), pp. 251–270. DOI: 10.1016/j.tra.2020.02.017. URL: https://doi.org/10.1016/j.tra.2020.02.017.

[22] M. Garey and D. Johnson. “Computers and Intractability: A Guide to the Theory of NP-Completeness”. In: 1978.

[23] R. Hassin. “Approximation Schemes for the Restricted Shortest Path Problem”. In: *Mathematics of Operations Research* 17.1 (1992), pp. 36–42. ISSN: 0364765X, 15265471. URL: http://www.jstor.org/stable/3689891.

[24] D. H. Lorenz and D. Raz. “A simple efficient approximation scheme for the restricted shortest path problem”. In: *Operations Research Letters* 28.5 (2001), pp. 213–219. ISSN: 0167-6377. DOI: https://doi.org/10.1016/S0167-6377(01)00069-4. URL: https://www.sciencedirect.com/science/article/pii/S0167637701000694.

[25] G. Tsaggouris and C. Zaroliagis. “Multiobjective Optimization: Improved FPTAS for Shortest Paths and Non-Linear Objectives with Applications”. In: *Theory of Computing Systems* 45.1 (Nov. 2007), pp. 162–186. DOI: 10.1007/s00224-007-9096-4. URL: https://doi.org/10.1007/s00224-007-9096-4.

[26] J. Maue, P. Sanders, and D. Matijevic. “Goal-directed shortest-path queries using precomputed cluster distances”. In: *Journal of Experimental Algorithmics* 14 (Dec. 2009), p. 3.2. DOI: 10.1145/1498698.1564502. URL: https://doi.org/10.1145/1498698.1564502.

[27] Google Maps. *Query Am Fassberg 17, Goettingen, Germany.* https://www.google.de/maps/place/Am+Fa%C3%9Fberg+17,+37077+G%C3%B6ttingen/@51.5605163,9.9660005,681m/data=!3m2!1e3!4b1!4m5!3m4!1s0x47a4d503fe8feebf:0x2c2b721e7a1fcf62!8m2!3d51.5605163!4d9.9681893. 2021.

[28] Google Maps. *Query Friedenstrasse 29A, Hoexter, Germany.* https://www.google.de/maps/place/Am+Fa%C3%9Fberg+17,+37077+G%C3%B6ttingen/@51.5605163,9.9660005,681m/data=!3m2!1e3!4b1!4m5!3m4!1s0x47a4d503fe8feebf:0x2c2b721e7a1fcf62!8m2!3d51.5605163!4d9.9681893. 2021.

[29] A. Beutelspacher. *Diskrete Mathematik für Einsteiger.* 5. Auflage. Springer-Verlag GmbH, Oct. 14, 2014. ISBN: 9783658057800.

[30] M. Newman. “Detecting community structure in networks”. In: *The European Physical Journal B - Condensed Matter* 38.2 (Mar. 2004), pp. 321–330. DOI: 10.1140/epjb/e2004-00124-y. URL: https://doi.org/10.1140/epjb/e2004-00124-y.

[31] S. E. Schaeffer. “Graph clustering”. In: *Computer Science Review* 1.1 (Aug. 2007), pp. 27–64. DOI: 10.1016/j.cosrev.2007.05.001. URL: https://doi.org/10.1016/j.cosrev.2007.05.001.

[32] S. Benz and R. Weibel. “Road network selection for medium scales using an extended stroke-mesh combination algorithm”. In: *Cartographic and Geographic Information Science* 41.4 (2014), pp. 323–339. DOI: 10.1080/15230406.2014.928482.

[33] A. Edwardes and W. Mackaness. “Intelligent Generalisation of urban road networks”. In: *Proceedings of GIS Research UK* (2000), pp. 81–85.

[34] B. Jiang and C. Claramunt. “A structural approach to model generalisation of an urban street network”. In: *Geoinformatica* 8.2 (2004), pp. 151–171. DOI: 10.1023/b:gein.0000017746.44824.70.

[35] W. A. Mackaness and M. K. Beard. “Use of graph theory to support generalisation”. In: *Cartography and Geographic Information Systems* 20 (1993), pp. 210–221.

[36] W. A. Mackaness and G. A. Mackechnie. “Automating the detection and simplification of junctions in road networks”. In: *Geoinformatica* 200.3 (1999), pp. 185–200.

[37] R. Thomson and D. Richardson. “A graph theory approach to road network generalisation”. In: *Proceedings 18th ICA/ACI International Cartographic Conference* (1995), pp. 1871–1880.

[38] R. Thomson and D. Richardson. “The ‘Good Continuation’ Principle of Perceptual Organization applied to the Generalization of Road Networks”. In: *Proceedings of the ICA 19th International Cartographic Conference* (1999), pp. 14–21.

[39] Q. Zhou and Z. Li. “Evaluation of Properties to determine the Importance of individual Roads for Map Generalization”. In: *Lecture Notes in Geoinformation and Cartography*. Springer Berlin Heidelberg, 2011, pp. 459–475. DOI: 10.1007/978-3-642-19143-5_26. URL: https://doi.org/10.1007/978-3-642-19143-5_26.

[40] N. de Lange. *Geoinformatik*. Springer Berlin Heidelberg, 2013. DOI: 10.1007/978-3-642-34807-5.

[41] E. W. Dijkstra. “A Note on Two Problems in Connexion with Graphs”. In: *NUMERISCHE MATHEMATIK* 1.1 (1959), pp. 269–271.

[42] OpenStreetMap. *Attributierung von Straßen in Deutschland*. 2020. URL: https://wiki.openstreetmap.org/wiki/Attributierung_von_Stra%5C%C3%5C%9Fen_in_Deutschland.

[43] OpenStreetMap. *Map Features*. 2021. URL: https://wiki.openstreetmap.org/wiki/Map_features.

[44] M. Barthelemy. “Betweenness centrality in large complex networks”. In: *The European Physical Journal B - Condensed Matter* 38.2 (2004), pp. 163–168. DOI: 10.1140/epjb/e2004-00111-4.

[45] J. Golbeck. “Analyzing networks”. In: *Introduction to Social Media Investigation*. Elsevier, 2015, pp. 221–235. DOI: 10.1016/b978-0-12-801656-5.00021-4. URL: https://doi.org/10.1016/b978-0-12-801656-5.00021-4.

[46] A. Landherr, B. Friedl, and J. Heidemann. “A Critical Review of Centrality Measures in Social Networks”. In: *Business & Information Systems Engineering* 2.6 (Oct. 2010), pp. 371–385. DOI: 10.1007/s12599-010-0127-3. URL: https://doi.org/10.1007/s12599-010-0127-3.

[47] L. Metcalf and W. Casey. “Graph theory”. In: *Cybersecurity and Applied Mathematics*. Elsevier, 2016, pp. 67–94. DOI: 10.1016/b978-0-12-804452-0.00005-1. URL: https://doi.org/10.1016/b978-0-12-804452-0.00005-1.

[48] G. Vaira and O. Kurasova. “Parallel Bidirectional Dijkstra’s Shortest Path Algorithm”. In: *Databases and Information Systems VI - Selected Papers from the Ninth International Baltic Conference, DB&IS 2010, July 5-7, 2010, Riga, Latvia*. Ed. by Janis Barzdins and Marite Kirikova. Vol. 224. Frontiers in Artificial Intelligence and Applications. IOS Press, 2010, pp. 422–435. DOI: 10.3233/978-1-60750-688-1-422. URL: https://doi.org/10.3233/978-1-60750-688-1-422.

[49] P. E. Hart, N. J. Nilsson, and B. Raphael. “A Formal Basis for the Heuristic Determination of Minimum Cost Paths”. In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. DOI: 10.1109/TSSC.1968.300136.

[50] R. Geisberger *et al.* “Exact Routing in Large Road Networks Using Contraction Hierarchies”. In: *Transportation Science* 46.3 (2012), pp. 388–404. ISSN: 1526-5447. DOI: 10.1287/trsc.1110.0401. URL: https://doi.org/10.1287/trsc.1110.0401.

[51] Project-OSRM. *How to run the tool chain*. https://github.com/Project-OSRM/osrm-backend/wiki/Running-OSRM. 2017.

[52] R. W. Floyd. “Algorithm 97: Shortest Path”. In: *Commun. ACM* 5.6 (June 1962), p. 345. ISSN: 0001-0782. DOI: 10.1145/367766.368168. URL: https://doi.org/10.1145/367766.368168.

[53] S. Warshall. “A Theorem on Boolean Matrices”. In: *J. ACM* 9.1 (1962), pp. 11–12. ISSN: 0004-5411. DOI: 10.1145/321105.321107. URL: https://doi.org/10.1145/321105.321107.

[54] R. Rojas *et al.* “Konrad Zuses Plankalkül — Seine Genese und eine moderne Implementierung”. In: *Geschichten der Informatik.* Springer Berlin Heidelberg, 2004, pp. 215–235. DOI: 10.1007/978-3-642-18631-8_9. URL: https://doi.org/10.1007/978-3-642-18631-8_9.

[55] D. Delling *et al.* “Engineering Route Planning Algorithms”. In: *Algorithmics of Large and Complex Networks.* Springer Berlin Heidelberg, 2009, pp. 117–139. DOI: 10.1007/978-3-642-02094-0_7. URL: https://doi.org/10.1007/978-3-642-02094-0_7.

[56] D. P. Williamson and D. B. Shmoys. *The Design of Approximation Algorithms.* 1st. USA: Cambridge University Press, 2011. ISBN: 0521195276.

[57] D. S. Johnson. “Approximation algorithms for combinatorial problems”. In: *Journal of Computer and System Sciences* 9.3 (Dec. 1974), pp. 256–278. DOI: 10.1016/s0022-0000(74)80044-9. URL: https://doi.org/10.1016/s0022-0000(74)80044-9.

[58] A. Hahn, W. Frühling, and J. Schlüter. “Determination of optimized pick-up and drop-off locations in transport routing - A cost distance approach”. In: *XIII International Scientific Conference and X International Symposium of Young Researchers. Conference Proceedings.* Ed. by Aleksander Sladkowski. Katowice, Poland: Silesian University of Technology - Faculty of Transport and Aviation Engineering, 2021. ISBN: 978-83-959742-1-2.

[59] Li Nyen Thin *et al.* “GPS Systems Literature: Inaccuracy Factors And Effective Solutions”. In: *International journal of Computer Networks & Communications* 8.2 (Apr. 2016), pp. 123–131. DOI: 10.5121/ijcnc.2016.8211. URL: https://doi.org/10.5121/ijcnc.2016.8211.

[60] F. C. Pereira, H. Costa, and N. M. Pereira. “An off-line map-matching algorithm for incomplete map databases”. In: *European Transport Research Review* 1.3 (Sept. 2009), pp. 107–124. DOI: 10.1007/s12544-009-0013-6. URL: https://doi.org/10.1007/s12544-009-0013-6.

[61] M. Hashemi and H. A. Karimi. “A critical review of real-time map-matching algorithms: Current issues and future directions”. In: *Computers, Environment and Urban Systems* 48 (2014), pp. 153–165. ISSN: 0198-9715. DOI: https://doi.org/10.1016/j.compenvurbsys.2014.07.009. URL: https://www.sciencedirect.com/science/article/pii/S0198971514000908.

[62] M. He *et al.* “An enhanced weight-based real-time map matching algorithm for complex urban networks”. In: *Physica A: Statistical Mechanics and its Applications* 534 (Nov. 2019), p. 122318. DOI: 10.1016/j.physa.2019.122318. URL: https://doi.org/10.1016/j.physa.2019.122318.

[63] L. Knapen *et al.* “Likelihood-based offline map matching of GPS recordings using global trace information”. In: *Transportation Research Part C: Emerging Technologies* 93 (Aug. 2018), pp. 13–35. DOI: 10.1016/j.trc.2018.05.014. URL: https://doi.org/10.1016/j.trc.2018.05.014.

[64] M. A. Quddus, W. Y. Ochieng, and R. B. Noland. “Current map-matching algorithms for transport applications: State-of-the art and future research directions”. In: *Transportation Research Part C: Emerging Technologies* 15.5 (Oct. 2007), pp. 312–328. DOI: 10.1016/j.trc.2007.05.002. URL: https://doi.org/10.1016/j.trc.2007.05.002.

[65] D. Zhang, Y. Dong, and Z. Guo. “A turning point-based offline map matching algorithm for urban road networks”. In: *Information Sciences* 565 (July 2021), pp. 32–45. DOI: 10.1016/j.ins.2021.02.052. URL: https://doi.org/10.1016/j.ins.2021.02.052.

[66] H Yin and O. Wolfson. “A weight-based map matching method in moving objects databases”. In: vol. 16. July 2004, pp. 437–438. ISBN: 0-7695-2146-0. DOI: 10.1109/SSDM.2004.1311248.

[67] M. J. de Smith, M. F. Goodchild, and P. A. & Associates Longley. *Geospatial Analysis 6th Edition.* 2018.

[68] D. H. Douglas. “Least-cost Path in GIS Using an Accumulated Cost Surface and Slopelines”. In: *Cartographica: The International Journal for Geographic Information and Geovisualization* 31 (Oct. 1994), pp. 37–51. DOI: 10.3138/D327-0323-2JUT-016M.

[69] C. Yu, J. Lee, and M. Munro-Stasiuk. “Research Article: Extensions to least-cost path algorithms for roadway planning”. In: *International Journal of Geographical Information Science* 17.4 (2003), pp. 361–376. DOI: 10.1080/1365881031000072645.

[70] M. van Leusen. “Pattern to process: methodological investigations into the formation and interpretation of spatial patterns in archaeological landscapes”. In: (Jan. 2002).

[71] G. Rees. “Least-Cost Paths in Mountainous Terrain”. In: *Computers & Geosciences* 30 (Apr. 2004), pp. 203–209. DOI: 10.1016/j.cageo.2003.11.001.

[72] W. Warntz. “Transportation, Social Physics, And The Law Of Refraction”. In: *The Professional Geographer* 9 (Feb. 2005), pp. 2–7. DOI: 10.1111/j.0033-0124.1957.094_2.x.

[73] W. Collischonn and J. Pilar. “A direction dependent least-cost-path algorithm for roads and canals”. In: *International Journal of Geographical Information Science* 14 (June 2000), pp. 397–406. DOI: 10.1080/13658810050024304.

[74] T.H. Cormen *et al. Introduction to Algorithms*. MIT Press, 2001.

[75] J. Xu and R. G. Lathrop. “Improving cost-path tracing in a raster data format”. In: *Computers & Geosciences* 20.10 (1994), pp. 1455–1465. ISSN: 0098-3004. DOI: `https://doi.org/10.1016/0098-3004(94)90105-8`. URL: `http://www.sciencedirect.com/science/article/pii/0098300494901058`.

[76] Y. Akhremtsev, P. Sanders, and C. Schulz. “(Semi-)External Algorithms for Graph Partitioning and Clustering”. In: *2015 Proceedings of the Seventeenth Workshop on Algorithm Engineering and Experiments (ALENEX)*. Society for Industrial and Applied Mathematics, Dec. 2014, pp. 33–43. DOI: `10.1137/1.9781611973754.4`. URL: `https://doi.org/10.1137/1.9781611973754.4`.

[77] C. Schulz. “High Quality Graph Partitioning”. PhD thesis. Karlsruher Institute for Technology, 2013.

[78] K. Schloegel, G. Karypis, and V. Kumar. “Graph Partitioning for High-Performance Scientific Simulations”. In: *Sourcebook of Parallel Computing*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003, pp. 491–541. ISBN: 1558608710.

[79] Ü. V. Çatalyürek and C. Aykanat. “Decomposing irregularly sparse matrices for parallel matrix-vector multiplication”. In: *Parallel Algorithms for Irregularly Structured Problems*. Ed. by Alfonso Ferreira *et al.* Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 75–86. ISBN: 978-3-540-68808-2.

[80] J. Fietz *et al.* “Optimized Hybrid Parallel Lattice Boltzmann Fluid Flow Simulations on Complex Geometries”. In: *Euro-Par 2012 Parallel Processing*. Ed. by Christos Kaklamanis, Theodore Papatheodorou, and Paul G. Spirakis. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 818–829. ISBN: 978-3-642-32820-6.

[81] U. Lauther. “An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background”. In: *Geoinformation und Mobilität-von der Forschung zur praktischen Anwendung* 22 (2004), pp. 219–230.

[82] R. Möhring *et al.* “Partitioning graphs to speedup Dijkstra’s algorithm”. In: *Journal of Experimental Algorithmics* 11 (Feb. 2007), p. 2.8. DOI: `10.1145/1187436.1216585`. URL: `https://doi.org/10.1145/1187436.1216585`.

[83] D. Luxen and D. Schieferdecker. “Candidate Sets for Alternative Routes in Road Networks”. In: *Experimental Algorithms*. Ed. by Ralf Klasing. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 260–270. ISBN: 978-3-642-30850-5.

[84] S. Lloyd. “Least squares quantization in PCM”. In: *IEEE Transactions on Information Theory* 28.2 (1982), pp. 129–137. DOI: `10.1109/TIT.1982.1056489`.

[85] D. Delling *et al.* "Customizable Route Planning". In: *Experimental Algorithms*. Springer Berlin Heidelberg, 2011, pp. 376–387. DOI: 10.1007/978-3-642-20662-7_32. URL: https://doi.org/10.1007/978-3-642-20662-7_32.

[86] C.E. Bichot and P. Siarry. "Graph Partitioning". In: *Encyclopedia of Parallel Computing*. Springer US, 2011, pp. 805–808. DOI: 10.1007/978-0-387-09766-4_92. URL: https://doi.org/10.1007/978-0-387-09766-4_92.

[87] A. Buluç *et al.* "Recent Advances in Graph Partitioning". In: *Algorithm Engineering*. Springer International Publishing, 2016, pp. 117–158. DOI: 10.1007/978-3-319-49487-6_4. URL: https://doi.org/10.1007/978-3-319-49487-6_4.

[88] G. Pavlopoulos *et al.* "Using graph theory to analyze biological networks". In: *BioData Mining* 4.1 (Apr. 2011). DOI: 10.1186/1756-0381-4-10. URL: https://doi.org/10.1186/1756-0381-4-10.

[89] M. Hilger *et al.* "Fast point-to-point shortest path computations with arc-flags". In: *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, July 2009, pp. 41–72. DOI: 10.1090/dimacs/074/03. URL: https://doi.org/10.1090/dimacs/074/03.

[90] E. Köhler, R. Möhring, and H. Schilling. "Acceleration of Shortest Path and Constrained Shortest Path Computation". In: *Experimental and Efficient Algorithms*. Springer Berlin Heidelberg, 2005, pp. 126–138. DOI: 10.1007/11427186_13. URL: https://doi.org/10.1007/11427186_13.

[91] D. Delling *et al.* "Graph Partitioning with Natural Cuts". In: *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, May 2011. DOI: 10.1109/ipdps.2011.108. URL: https://doi.org/10.1109/ipdps.2011.108.

[92] D. Delling and R. F. Werneck. "Faster Customization of Road Networks". In: *Experimental Algorithms*. Springer Berlin Heidelberg, 2013, pp. 30–42. DOI: 10.1007/978-3-642-38527-8_5. URL: https://doi.org/10.1007/978-3-642-38527-8_5.

[93] S. Jung and S. Pramanik. "An efficient path computation model for hierarchically structured topographical road maps". In: *IEEE Transactions on Knowledge and Data Engineering* 14.5 (Sept. 2002), pp. 1029–1046. DOI: 10.1109/tkde.2002.1033772. URL: https://doi.org/10.1109/tkde.2002.1033772.

[94] D. Yan *et al.* "Finding distance-preserving subgraphs in large road networks". In: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, Apr. 2013. DOI: 10.1109/icde.2013.6544861. URL: https://doi.org/10.1109/icde.2013.6544861.

[95] Z. Xu and H. A. Jacobsen. "Processing proximity relations in road networks". In: *Proceedings of the 2010 international conference on Management of data - SIGMOD '10*. ACM Press, 2010. DOI: 10.1145/1807167.1807196. URL: https://doi.org/10.1145/1807167.1807196.

[96] N. Eapen and A. Beegom. "A linear time pre-processing for optimization of shortest path and distance algorithms". In: *2017 IEEE International Conference on Signal Processing, Informatics, Communication and Energy Systems (SPICES)*. IEEE, Aug. 2017. DOI: 10.1109/spices.2017.8091313. URL: https://doi.org/10.1109/spices.2017.8091313.

[97] S. Ma *et al.* "Proxies for Shortest Path and Distance Queries". In: *IEEE Transactions on Knowledge and Data Engineering* 28.7 (July 2016), pp. 1835–1850. DOI: 10.1109/tkde.2016.2531667. URL: https://doi.org/10.1109/tkde.2016.2531667.

[98] U. N. Raghavan, R. Albert, and S. Kumara. "Near linear time algorithm to detect community structures in large-scale networks". In: *Physical Review E* 76.3 (Sept. 2007). DOI: 10.1103/physreve.76.036106. URL: https://doi.org/10.1103/physreve.76.036106.

[99] M. Newman. "Modularity and community structure in networks". In: *Proceedings of the National Academy of Sciences* 103.23 (May 2006), pp. 8577–8582. DOI: 10.1073/pnas.0601602103. URL: https://doi.org/10.1073/pnas.0601602103.

[100] T. Anwar *et al.* "Partitioning road networks using density peak graphs: Efficiency vs. accuracy". In: *Information Systems* 64 (Mar. 2017), pp. 22–40. DOI: 10.1016/j.is.2016.09.006. URL: https://doi.org/10.1016/j.is.2016.09.006.

[101] W. Shoman and F. Gülgen. "Centrality Based Hierarchy for Generalizing and Labeling Street Features in Multi Resolution Maps". In: July 2016.

[102] G. Maier. "OpenStreetMap, the Wikipedia Map". In: *REGION* 1.1 (Dec. 2014), R3–R10. DOI: 10.18335/region.v1i1.70. URL: https://doi.org/10.18335/region.v1i1.70.

[103] M. Haklay. "How Good is Volunteered Geographical Information? A Comparative Study of OpenStreetMap and Ordnance Survey Datasets". In: *Environment and Planning B: Planning and Design* 37.4 (Aug. 2010), pp. 682–703. DOI: 10.1068/b35097. URL: https://doi.org/10.1068/b35097.

[104] T. Ort, L. Paull, and D. Rus. "Autonomous Vehicle Navigation in Rural Environments Without Detailed Prior Maps". In: May 2018, pp. 2040–2047. DOI: 10.1109/ICRA.2018.8460519.

[105] S. Funke, R. Schirrmeister, and S. Storandt. "Automatic Extrapolation of Missing Road Network Data in Openstreetmap". In: *Proceedings of the 2nd International Conference on Mining Urban Data - Volume 1392*. MUD'15. Lille, France: CEUR-WS.org, 2015, pp. 27–35.

[106] X. Hu *et al.* "Tagging the main entrances of public buildings based on OpenStreetMap and binary imbalanced learning". In: *International Journal of Geographical Information Science* (Feb. 2021), pp. 1–29. DOI: 10.1080/13658816.2020.1861282. URL: https://doi.org/10.1080/13658816.2020.1861282.

[107] S.J. Kang *et al.* "Entrance Detection of Buildings Using Multiple Cues". In: *Intelligent Information and Database Systems*. Springer Berlin Heidelberg, 2010, pp. 251–260. DOI: 10.1007/978-3-642-12145-6_26. URL: https://doi.org/10.1007/978-3-642-12145-6_26.

[108] M. Southworth and E. Ben-Joseph. *Streets and the Shaping of Towns and Cities -*. None. Washington: Island Press, 2003. ISBN: 978-1-559-63916-3.

[109] F. Parés *et al. Fluid Communities: A Competitive, Scalable and Diverse Community Detection Algorithm*. 2017. eprint: arXiv:1703.09307.

[110] C. Malzer and M. Baum. "A Hybrid Approach To Hierarchical Density-based Cluster Selection". In: Sept. 2020, pp. 223–228. DOI: 10.1109/MFI49285.2020.9235263.

[111] A. Hahn, W. Frühling, and J. Schlüter. "Using open-source high resolution remote sensing data to determine the access to buildings in the context of passenger transport". In: (Mar. 2021). DOI: 10.5194/egusphere-egu21-9408. URL: https://doi.org/10.5194/egusphere-egu21-9408.

[112] Landesdatenbank-NRW. *Die Landesdatenbank NRW*. https://www.landesdatenbank.nrw.de/ldbnrw/online/dat. 2018. URL: https://www.landesdatenbank.nrw.de/ldbnrw/online/dat (visited on 06/27/2018).

[113] OpenStreetMap contributors. *Planet dump retrieved from https://planet.osm.org*. https://www.openstreetmap.org. 2017.

[114] Information und Technik - Nordrhein-Westfalen. *OpenGeoData Nordrhein-Westfalen retrieved from https://planet.osm.org*. https://www.opengeodata.nrw.de/produkte/. 2017.

[115] OSM Blog - unknown author. *Von einem, der auszog, viel zu viele Hausnummern zu mappen*. https://blog.openstreetmap.de/blog/2012/01/von-einem-der-auszog-viel-zu-viele-hausnummern-zu-mappen/. 2012.

[116] D. P. Huijsmans and A. M. Vossepoel. "Informatie in Gedigitaliseerde Beelden,volume One: Introduction". In: (1989).

[117] A.V. Oppenheim *et al. Signals & Systems*. Prentice-Hall signal processing series. Prentice Hall, 1983. ISBN: 9780138147570.

Glossary

DARP "Ride-sharing aims to bring together travelers with similar routes and schedules, and the idea is similar to the traditional dial-a-ride problem (DARP). The difference between ride-sharing and DARP program is the type of driver supply; in a DARP, drivers are provided by a company within the DARP program, whereas drivers in ride-sharing systems are independent entities."[6].

distance deviation In this thesis, the distance deviation is a parameter to adjust and ensure an acceptable size of partitions and hence the degree of generalization.

Euclidean distance The Euclidean distance describes the length of a straight line between two points in Euclidean space.

First Mile / Last Mile problem The First Mile / Last Mile problem in logistics and transportation describes the first, respectively the last leg for customers or mailings to their final destination. As an example from logistics, the delivery from hubs to individual addresses is time-consuming and expensive..

intermodal Intermodal transportation describes a trip consisting of different transport modes within a trip. For example: A trip consist of a ride-sharing concept that delivers the passenger to a train station and then the trips continues by train.

perpendicular distance The perpendicular distance describes the shortest distance between a point and a point on a line in Euclidean space.

road snapping Road snapping describes the assignment of coordinates or addresses to reference point on the road network. This is needed to determine the start- and endpoint of a route in the road network.

snapping point A snapping point is a reference point for a coordinate or an address on a road network, that results from road snapping or map matching. This is needed to determine the start- and endpoint of a route for given coordinates or addresses that are not directly located on the road network.

stop locations We used the term stop locations as a simplification for pickup and drop off locations in the context of passenger transportation.

to-door In this thesis, to-door services or routing describes mobility concepts such as DRT-systems, where either requests from any address (from door) or to any address (to door) or both (door-to-door) is allowed. Often peer-to-peer is used as a synonym for door-to-door.

traveling salesman problem The traveling salesman problem is a classic basic problem in combinatorial optimization, where a sequence of nodes is sought that covers all nodes of the graph and all nodes except the starting point are visited exactly once and the end of the sequence is the starting point. Here the length of the path must be minimally short.

variation The variation or also variability describes the behaviour of the data in a general sense. This should not be mistaken with variance.

vehicle routing problem The vehicle routing problem describes the problem of finding optimal routes for multiple vehicles and a given set of stops in transportation. Thereby, parameters such as the capacity of the vehicles or time windows for considered pick-up passengers are common restrictions. If this problem is reduced from a fleet of vehicles to one vehicle and only the shortest route should be found, regardless of other parameters, this can be seen as an equivalent to the traveling salesman problem.

A. Appendix

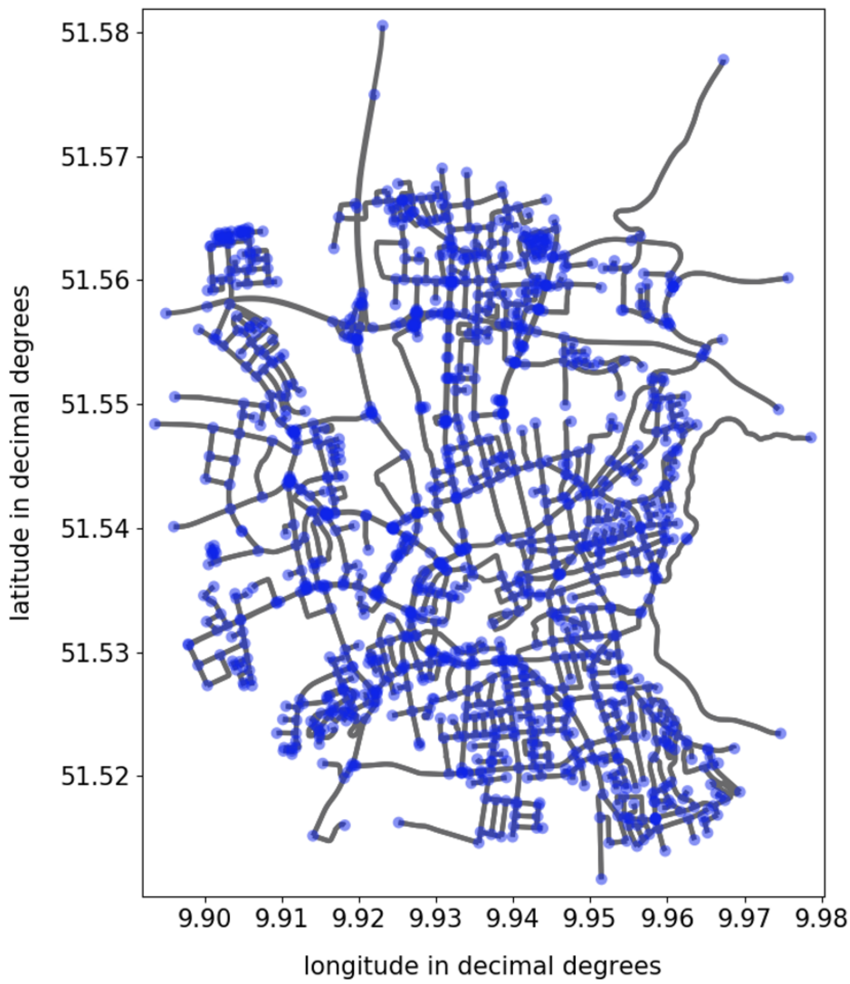


Figure A.1.: Primal road network of Göttingen.

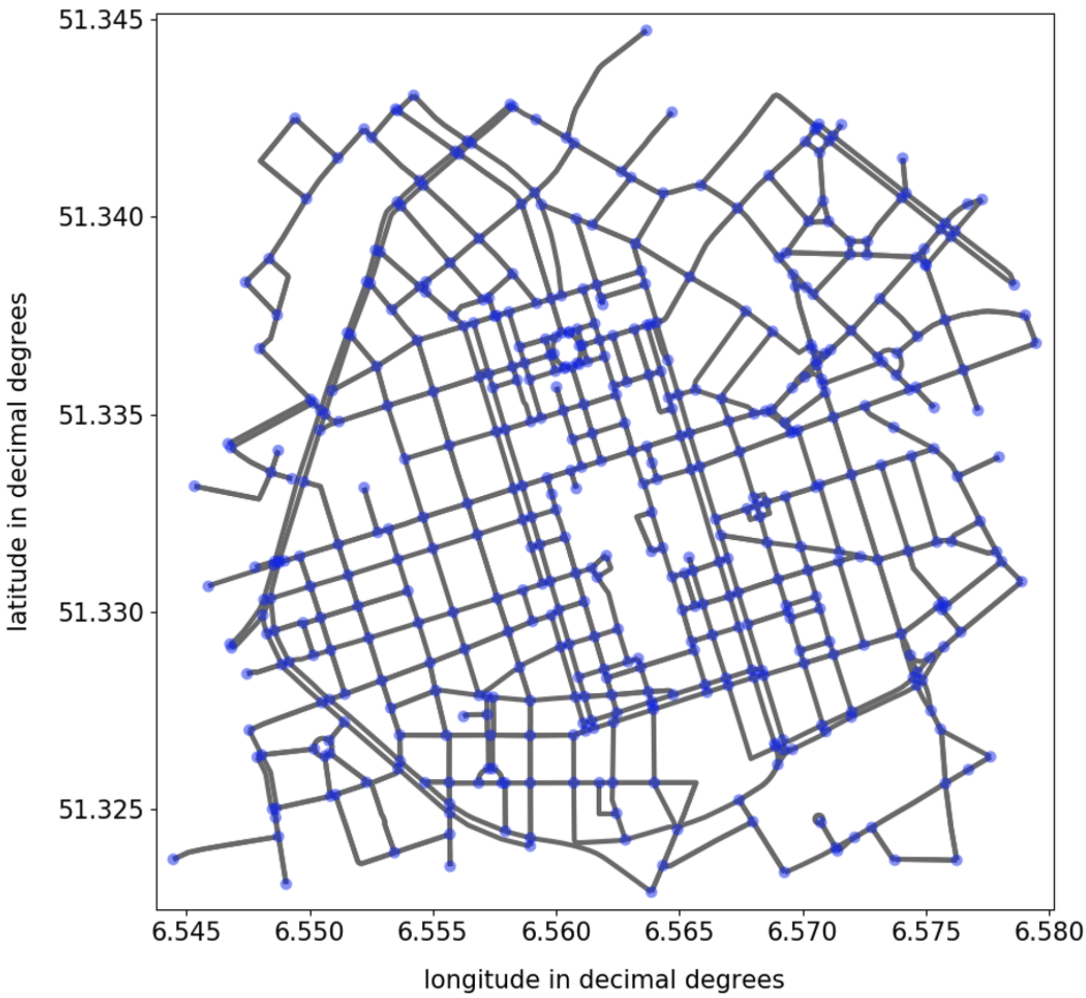


Figure A.2.: Primal road network of Krefeld.

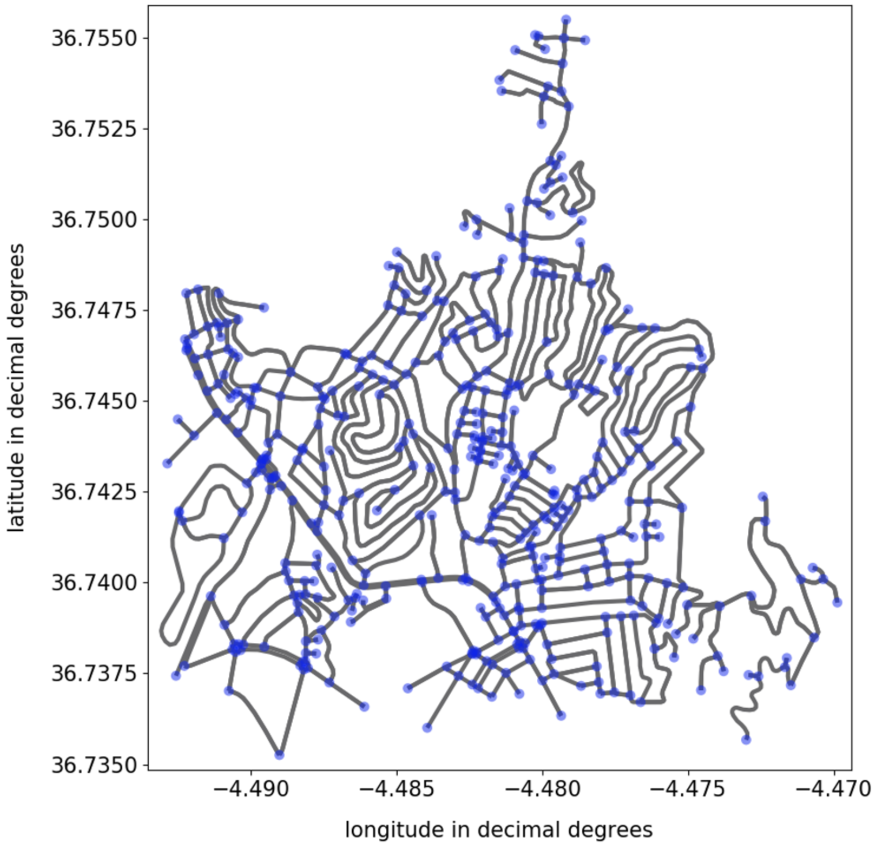


Figure A.3.: Primal road network of Málaga.

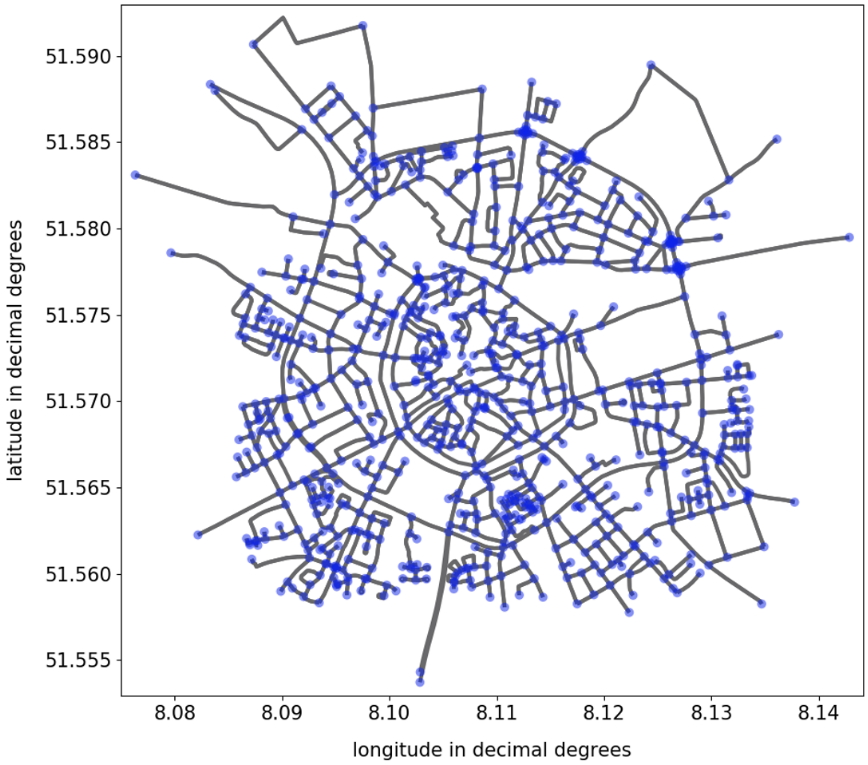


Figure A.4.: Primal road network of Soest.

	north	south	east	west
Göttingen	51.567068	51.515715	9.966399	9.900996
Krefeld	51.342099	51.324141	6.576635	6.548003
Málaga	36.755269	36.737246	-4.469599	-4.491996
Soest	51.5871	51.5593	8.1336	8.0865

Table A.1.: Extent of the considered road networks in chapter 5 (EPSG:4325).

Figure A.5.: True color composite (RGB) of an extend in the AOI Höxter. A similar extent was shown in Figure 2.4.

B. Appendix

B.1. Source Code for FluidC-Generalization based on Proxies (FC-GBOP)

The following code is intended to provide an overview of the software. To work with it, we recommend using the tested code from the repository. Python version 3.7.6 was used and the required libraries with their version numbers are listed in the requirements.txt in the repository.

Repository: `https://github.com/fauceta/FC-GBOP`

Requirements: `https://github.com/fauceta/FC-GBOP/blob/master/requirements.txt`

For inquiries please contact: armin.hahn@ds.mpg.de

```
# Defines properties of the area of interest

class AreaOfInterest:
        shapefileName = ''
        north = 0.0
        south = 0.0
        east = 0.0
        west = 0.0
        abbreviation = ''
        center = 0, 0

goettingen = AreaOfInterest()
goettingen.shapefileName = 'Goettingen'
goettingen.north = 51.567068
goettingen.south = 51.515715
goettingen.east = 9.966399
goettingen.west = 9.900996
goettingen.abbreviation = 'goe'

krefeld = AreaOfInterest()
krefeld.shapefileName = 'Krefeld'
krefeld.north = 51.3420997
krefeld.south = 51.3241411
krefeld.east = 6.5766356
krefeld.west = 6.5480031
krefeld.abbreviation = 'kre'

soest = AreaOfInterest()
soest.shapefileName = 'Soest'
soest.north = 51.5871
soest.south = 51.5593
soest.east = 8.1336
soest.west = 8.0865
soest.abbreviation = 'soe'
```

```

malaga = AreaOfInterest()
malaga.shapefileName = 'malaga'
malaga.north = 36.75526978941597
malaga.south = 36.737246196202555
malaga.east = -4.491996074660827
malaga.west = -4.46959984835477
malaga.center = 36.746258, -4.480798
malaga.abbreviation = 'malaga'
```

Listing B.1: AOI.py

```
import osmnx as ox, networkx as nx, geopandas as gpd, pandas as pd, numpy as np, matplotlib.pyplot as plt,
     matplotlib.patches as mpatches, matplotlib.colors as mpc, matplotlib.cm as mcm, cartopy.crs as ccrs

import itertools, random, request, timeit

from networkx.algorithms import community
from cartopy.mpl.gridliner import (
    LONGITUDE_FORMATTER,
    LATITUDE_FORMATTER,
)
from cartopy.io.img_tiles import GoogleTiles


################## Graph Partitioning ##################
def evalOptimumK(G, distanceDeviation, k=2, algorithm='fluid', proxy_centrality='BC'):
    """
    main function that loads data, partitions and evaluate the partitions
    implemented algorithms: fluid, lpa, modularity or kernighan_lin
    implemented centrality: BC (Betweennness), CL (Closeness)
    """
    while(True):
        #create the partitions and write the community IDs in the nodes DF
        nodes, edges = createPartition(G, algorithm=algorithm, k=k)

        #creates subgraphes and identifies the proxies with pagerank
        subgraphs = partition_graph(G, nodes)

        df_proxies = find_proxies(G, nodes, subgraphs, k, proxy_centrality = proxy_centrality)

        #prepares data subgraphs and proxies in one dataframe
        df_proxies['subgraph'] = [v for k, v in subgraphs.items()]

        #check the reachability for each community
        print (f'...processing k-value: {k}')
        accessability = communityReachability(df_proxies, distanceDeviation)

        if algorithm == 'fluid':
            if accessability == False:
                k+= 1
                continue
            else:
                print(f'found optimum k {k}')
                break
        else:
            break
    return nodes, edges, df_proxies, subgraphs


def getPrimalGraph(coords):
    """
    created the networkx graph with osmxn for given coordinates. Either a bbox with
    4 coords or 2 coords as a point. E.g. [north, south, east, west]
    """

    if len(coords) == 4:

        G = ox.graph_from_bbox(north=coords[0], south=coords[1], east=coords[2], west=coords[3], network_type="
            drive", truncate_by_edge=True)

    elif len(coords) == 2:
        G = ox.graph_from_point(coords, network_type='drive', truncate_by_edge=True, distance=1000)

    else:
        raise ValueError('Cannot read type of bbox.')

    return G

```

```python

def createPartition(G, algorithm, k):
    """
    creates partitions based on a graph G (networkx graph), given algorithm and a parameter k for the number of partitions, needed for some algorithms
    algorithm = fluid, lpa, modularity or kernighan_lin
    """

    nodes, edges = ox.save_load.graph_to_gdfs(G, nodes=True, edges=True)

    nodes["community"] = 0

    if algorithm == 'fluid':
        communities = community.asyn_fluidc(G.to_undirected(), k)

    elif algorithm == 'lpa':
        communities = community.label_propagation_communities(G.to_undirected())

    elif algorithm == 'modularity':
        communities = community.greedy_modularity_communities(G.to_undirected(), weight='length')

    elif algorithm == 'kernighan_lin':
        communities = community.kernighan_lin_bisection(G.to_undirected(), weight=None)

    else:
        raise ValueError('unsupported algorithm')

    for idx, partition in enumerate(communities):
        for node in partition:
            nodes.at[node, "community"] = idx + 1
    return nodes, edges


def partition_graph(G, nodes):
   """
   partitioning of the graph based on properties of nodes (assigned partition)
   returns subgraphs in a dict with G1,G2 as key and networkx subgraphs as values
   """
    # use k or get the amount of communities
    sum_communties = nodes.community.max()

    community_nodes = []

    #iterate for each community
    for i in range(sum_communties):
        community_nodes.append([nodes.loc[nodes['community'] == i+1]])          #need i+1 because community starts with 1 not with zero

    #split the graph by community value
    subgraphs = {}
    for idx, partition in enumerate(community_nodes):
        suffix = idx + 1
        subgraphs["G" + str(suffix)] = G.subgraph(list(community_nodes[idx][0]['osmid']))
    return subgraphs


def find_proxies(G, nodes, subgraphs, k, proxy_centrality, verbose=False):
    """
    accepts proxy centrality: 'BC', 'CL' or dict with weighting {'BC':0.5, 'CL':0.5}
    """
    global proxies
    proxies = {}

    #calc betweennes proxies in reference to the original graph, added in global proxies
    betweenness_proxies(G, subgraphs)

    #calc closeness proxies in reference to their subgraph, added in global proxies
    closeness_proxies(G, subgraphs)

    if verbose == True:
        import pprint
        print('Subgraphs and their proxies with corresponding centrality measures:')
        pprint.pprint(proxies)


    if proxy_centrality == 'BC':
        #selects only BC proxies from proxy dict
        selected_proxies = {key:value for (key,value) in proxies.items() if key[-11:] == 'betweenness'}
    elif proxy_centrality == 'CL':
        #selects only CL proxies from proxy dict
```

```python
        selected_proxies = {key:value for (key,value) in proxies.items() if key[-9:] == 'closeness'}
    elif type(proxy_centrality) == dict:
        print(f'not implemented yet {k}')
    else:
        raise ValueError ('unknown proxy_centrality value')

    #creates a seperate gdf for proxies
    proxy_frames = []

    #extract the osmids from nested dict
    proxies_osmid = []
    for k1, v1 in selected_proxies.items():
        for k2, v2 in v1.items():
            proxies_osmid.append(k2)

    for item in proxies_osmid:
        proxy_frames.append(nodes.loc[nodes['osmid'] == item])

    df_proxies = pd.concat(proxy_frames)
    return df_proxies


def betweenness_proxies(G, subgraphs):
    """
    Calculates the betweenness centrality for subgraphs. Input subgraphs as key-value pair (dicts)
    """
    betweenness_orig = nx.betweenness_centrality(nx.DiGraph(G), weight='length')

    #proxy = {} # use global variable instead
    #iterate over the different subgraphs (Gn)
    for key, value in subgraphs.items():
        #create directed graph for the subgraph
        subgraph_dir = nx.DiGraph(value)

        #select the node within the subgraph with the highest value in betweenness_orig
        subgraph_centrality_osmid = None
        subgraph_centrality_value = 0
        for osmid in subgraph_dir.nodes():
            if (subgraph_centrality_value < betweenness_orig.get(osmid)):
                subgraph_centrality_value = betweenness_orig.get(osmid)
                subgraph_centrality_osmid = osmid

        #write the partition and the proxy into proxy dict
        proxies[f'{key}_betweenness'] = {subgraph_centrality_osmid:subgraph_centrality_value}


def closeness_proxies(G, subgraphs):
    """
    Calculates the closeness centrality for subgraphs. Input subgraphs as key-value pair (dicts)
    """
    #proxy = {} # use global variable instead
    for key, value in subgraphs.items():
        subgraph_dir = nx.DiGraph(value)
        #calculate closeness for the subgraph
        closeness_centrality = nx.closeness_centrality(subgraph_dir, distance='length')
        #get the osmid with the highest closeness value
        closeness_proxy = max(closeness_centrality, key=closeness_centrality.get)
        #need global variable proxy
        proxies[f'{key}_closeness'] = {closeness_proxy:closeness_centrality.get(closeness_proxy)}


def weight_proxies(weighting, subgraphs, k):
    """
    weights CL and BC with the given dict weighting - this function is may be irrelevant, not only max BC/CL for calculation)
    """
    #pseudo_proxies for testing
    p = {'G1_betweenness': {60434219: 0.08739129284107451}, 'G2_betweenness': {305204955: 0.1813449781659389}, 'G3_betweenness': {579926865: 0.13271033478893743}, 'G4_betweenness': {60346725: 0.16642530104538839}, 'G5_betweenness': {1912529768: 0.09292258832870187}, 'G1_closeness': {28123858: 0.0005414683856711313}, 'G2_closeness': {28199172: 0.0008807034474547498}, 'G3_closeness': {28095715: 0.0005816236962446243}, 'G4_closeness': {60437845: 0.0008652616490743784}, 'G5_closeness': {4254093089: 0.0011351970347807095}}

    #get all keys of the dict into a list
    list_of_keys = [*p]

    #set i to 1, due k min is 2 partitions and numbering for partitions starts with 1
    i = 1

    #loop from i=1 until i==k ; for each partition selecting the BC/CL keys
```

```
    while i < (k+1):
        #select keys for partition i
        tmp_keys = [item for item in list_of_keys if item.startswith('G{}'.format(i))]
        #print(tmp_keys, i)

        #for items in tmp_key extract BC/CL
        BC_key = [x for x in tmp_keys if x.endswith('betweenness')][0]
        #CL_key = [x for x in tmp_keys if x.endswith('closeness')][0]

        BC_pair = p.get(BC_key)
        #CL_pair = p.get(CL_key)
        #using list comprehension to get the first value of the k-v-pair (osmid-centrality_value)
        BC = [(BC_pair[x]) for x in list(BC_pair)][0]

        #weighting the nodes with BC for partition i, gettingen subgraph[i] and calc

        #CL = [(CL_pair[x]) for x in list(CL_pair)][0]
        #calc_BCCL()
        #modified_proxies
        print('BC: {}, CL: {}, i: {}'.format(BC, None, i))
        i += 1


def communityReachability(df_proxies, distanceDeviation):
    """
    calculates the reachability for proxies. Are all nodes within a community
    reachable with a given distance (in meter as edgeweight 'length' is used)
    """"
    accessability = True
    for idx, row in df_proxies.iterrows():
        print (f'''\t ...processing reachability for community {row['community']}''')
        proxy = row['osmid']
        subgraph = row['subgraph']

        reachability = nx.single_source_dijkstra_path_length(subgraph, proxy, weight='length')

        #get nodes with higher distance deviation
        for key, value in reachability.items():
            if value > distanceDeviation:
                print(key,value)
                accessability = False
                # break #return to exit the for loop of df_proxies enhancing the performance because other
                    communities shouldnt be considered anymore
                return accessability

    return accessability


def getBBoxFromAddress(address):
    """
    needs address as str
    returns bbox in west, south, east, north
    """
    G = ox.graph_from_address(address, network_type="drive", truncate_by_edge=True)
    nodes = ox.save_load.graph_to_gdfs(G, nodes=True)
    return nodes.total_bounds


################# Graph construction #################
def constructGraph(G, df_proxies, nodes, edges, completeGraph = True):
    """
    Creates a new complete graph, based on a set of nodes (df_proxies)
    """

    G2 = nx.MultiGraph()
    proxies_list = list(df_proxies['osmid'])

    #add nodes, edges to the new generalized graph
    G2 = addNodesToGraph(G2, proxies_list, nodes)

    #prepare the edges for reconstruction
    possibleRoutes = calcAllPossibleRoutes(G, proxies_list)
    if completeGraph == False:
        splittedRoutes = splitRoutes(possibleRoutes, proxies_list)
        splittedRoutes = cleanRouteSteps(splittedRoutes)
        nodePairsFromRoute = getEdgePairFromRoute(splittedRoutes)

    else:
        nodePairsFromRoute = getEdgePairFromRoute(possibleRoutes)

```

```
    #cleaned duplicates/reversed routes in createEdgeList
    edgeList = createEdgesList(nodePairsFromRoute, edges)

    #round the edgweight on two decimals
    edgeList = [(elem[0], round(elem[1], 2)) for elem in edgeList]

    G2 = addEdgeList(edgeList, G2)

    return G2


def getOsmAttributes(listofOsmID, nodes):
    """
    get OpenStreetMap attributes. Modified from getPointInformationfromOsmID
    """

    if nodes is None:
        raise ValueError ('nodes not defined')

    results = []
    for item in nodes.itertuples():
        if str(item.osmid) in str(listofOsmID):
            results.append([('osmID', item.osmid), ('x', item.x), ('y',item.y), ('geometry', item.geometry), ('community', item.community)])

    # if len(shapelyobjects) != len(listofOsmID):
    #     raise ValueError ('length of input and output not equal')

    return results


def addNodesToGraph(G2, listofOsmIDs, nodes):
    """
    modified from addOsmIDAndShapelyPointToGraph
    adds nodes to a graph from a list of osmIDs (converting the IDs to coords)
    returns a networkx graph
    """

    osm_attributes = getOsmAttributes(listofOsmIDs, nodes)

    for idx, item in enumerate(osm_attributes):
        G2.add_node(item[0][1], osmID=item[0], x=item[1], y=item[2], community=item[4], xy=(item[1][1],item[2][1]))

        #examples of the properties
        # item[0][1] = ID            28127489
        # item[0] = osmID            ('osmID', 60346417)
        # item[1] = x                ('x', 9.9140377)
        # item[2] = y                ('y', 51.5484371)
        # item[3] = shapelyPoint     ('geometry', <shapely.geometry.point.Point object at 0x0000022BE1994408>)
        # item[4] = community        ('community', 4)
        # (item[1][1],item[2][1])    (9.9140377, 51.5484371) #need for plotting and getting pos
    return G2


def getLengthfromNodePair(tpl, edges):
    """
    extracts the length of an edge by using the nodepair
    """

    for idx, row in edges.iterrows():
        if (tpl[0] == int(row['u']) and tpl[1] == int(row['v']) or tpl[0] == int(row['v']) and tpl[1] == int(row['u'])):
            #print ("Index:" + str(idx) + " " + str(row['length']))
            return float(row['length'])


def cleanEdgeListFromDuplicates(edgeList):
    """"
    remove duplicates from the edgelist eg. [[(1,2),5],[(3,4),6],[(6,3),9], [(2,1),5]]
    results in [[(1, 2), 5], [(3, 4), 6], [(6, 3), 9]]
    """

    cleanEdgeList = []
    for item in edgeList:
        if str(item[0][::-1]) in str(edgeList):
            if (str(item[0][::-1]) not in str(cleanEdgeList)) and (str(item[0]) not in str(cleanEdgeList)):
                cleanEdgeList.append(item)
        else:
            cleanEdgeList.append(item)
```

```

    return cleanEdgeList


def createEdgesList(nodePairsFromRoute, edges):
    """
    getting length attribute for every u,v in all possible routes between representators
    """

    newEdges = []
    for idx, route in enumerate(nodePairsFromRoute):
        distance = 0
        newEdgeUV = (route[0][0], route[-1][-1])
        for edge in route:
            distance += getLengthfromNodePair(edge, edges)
        newEdges.append([newEdgeUV, distance])

    edgeList = cleanEdgeListFromDuplicates(newEdges)

    return edgeList


def addEdgeList(newEdges, rG):
    """
    add edges and its edgeweight (length) from an edgeslist (newEdges) to rG
    """
    for item in newEdges:
        rG.add_edge(item[0][0], item[0][1], length=item[1])
    return rG


def getEdgePairFromRoute(possibleRoutes):
    """"
    returns a list with tuple pairs with the adjacent nodes (u,v for edges) of all possible routes
    """

    allEdges = []
    for route in possibleRoutes:
        allEdges.append(list(zip(route[:-1], route[1:])))

    return allEdges


def cleanRouteSteps(splittedroutes):
    """
    remove dublicates
    """

    res = []
    for item in splittedroutes:
        if item not in res:
            res.append(item)
    return res


def splitRoutes(data, splitters):
    """
    split the routes by given splitters. E.g. routes over other proxies will stop and split the routes, resulting only in routes between proxies without proxies within the route
    """

    results = []
    for route in data:
        found = 0
        for idx, r in enumerate(route[1:-1], 1):  # start idx at 1
            if r in splitters:
                temp = route[found:idx+1]  # +1 to capture the splitter value
                results.append(temp)
                found = idx
        remaining = route[found:]
        results.append(remaining)
    return results


def calcAllPossibleRoutes(G, representators):
    """
    returns a list with all possible shortest routes between all possible nodepairs
    """

    possiblePairs = list(itertools.combinations(representators,2))
```

```
    #reciprocal=True keep edges that appear in both directions in the original graph
    G_undirected = G.to_undirected()

    possibleRoutes = []
    criticalPairs = []

    for item in possiblePairs:
        try:
            route = nx.shortest_path(G_undirected, item[0], item[1], weight='length')
            possibleRoutes.append(route)
        except:
            criticalPairs.append(item)
            pass

    if len(criticalPairs) > 0:
        raise ValueError ('lenght of criticalPairs > 0 - some routes are missing!')

    return possibleRoutes


def updateDfProxiesInteriorNodes(df_proxies):
    """
    returns an updated df_proxies with a new column of all osmids for each partition/proxy
    """

    interior_nodes = []
    for index, row in df_proxies.iterrows():
        interior_nodes.append(list(row['subgraph'].nodes()))
    df_proxies['interior_nodes'] = interior_nodes
    return df_proxies


def sumNetworkDistances(G):
    """
    calculates the sum of the network distances
    """
    G_undirected = G.to_undirected()
    sum_network_distances = sum([d['length'] for u, v, d in G_undirected.edges(data=True)])
    return sum_network_distances


def updateDfProxiesSubgraphSize(df_proxies):
    """
    Updates the df_proxies with the attribute of subgraph size and the sum of network distances
    """
    G_size = []
    sum_network_distance = []
    for idx, row in df_proxies.iterrows():
        subgraph = row['subgraph']
        G_size.append(subgraph.size(weight='length'))
        sum_network_distance.append(sumNetworkDistances(subgraph))
    df_proxies['G_size'] = G_size
    df_proxies['sum_network_distance'] = sum_network_distance
    return df_proxies


#################### Plotting #######################
def plotPartition(nodes, edges, df_proxies):
    """
    plots the partitions in colored nodes
    """

    fig = plt.figure(figsize=(20, 20))
    ax = fig.add_subplot(1, 1, 1, projection=ccrs.PlateCarree())

    bbox = getBBoxForPlotting(nodes)
    ax.set_extent(bbox, crs=ccrs.PlateCarree())

    #background map
#   ax.add_image(imagery, 12, alpha=0.5)

    #plot edges
    edges.plot(
        ax=ax,
        edgecolor="black",
        linewidth=1,
        facecolor="none",
        zorder=2,
        alpha=0.8,
    )
```

```
    #plot nodes
    nodes.plot(
        ax=ax,
        marker="o",
        markersize=200,
        # edge_color = '#909090',
        column="community",
        cmap="Set3",
        zorder=1,
        legend=False,
        categorical=True,
    )

    #plot proxies
    df_proxies.plot(
        ax=ax,
        marker="s",
        markersize = 250,
        color="black",
        zorder=5,
    )

    #plot grid with coords
    gl = ax.gridlines(draw_labels=True)
    gl.xlabels_top = gl.ylabels_right = False
    gl.xformatter = LONGITUDE_FORMATTER
    gl.yformatter = LATITUDE_FORMATTER
    gl.xlabel_style = {'size': 20}
    gl.ylabel_style = {'size': 20}

    # plt.title(
    #     "Community and Proxy Detection in Road Networks",
    #     {"fontsize": 30},
    #     pad=40,
    # )
    plt.show()


def plotGraph(G2, nodes, labels=False):
    """
    plots the new reduced graph
    """
    xmin, ymin, xmax, ymax = nodes.total_bounds
    #Boundings plus three percent
    xmin = xmin - (xmax - xmin)*0.03
    xmax = xmax + (xmax - xmin)*0.03
    ymin = ymin - (ymax - ymin)*0.03
    ymax = ymax + (ymax - ymin)*0.03

    pos = nx.get_node_attributes(G2, 'xy')
    fig, ax = plt.subplots(figsize=(15, 15))
    ax.set_xlim(left=xmin, right=xmax)
    ax.set_ylim(ymin, ymax)

    import pylab
    nx.draw(G2,pos, node_color='black', node_shape='s', node_size=250)
    # specifiy edge labels explicitly
    if labels == True:
        edge_labels=dict([((u,v,),d['length']) for u,v,d in G2.edges(data=True)])
        nx.draw_networkx_edge_labels(G2,pos,edge_labels=edge_labels)

    pylab.show()


def markNode(G, nodeList):
    """
    plot with a marked nodes. List of osmIDs as input
    """

    nc = ['r' if node in nodeList else '#757575' for node in G.nodes()]
     ox.plot_graph(G, fig_height=10, fig_width=10, node_color=nc, node_size=20, node_zorder=3, edge_linewidth=3)


#### functions get_color_list, get_node_colors_by_stats from G.Boeing - Credits
# https://github.com/gboeing/osmnx
def get_color_list(n, color_map='plasma', start=0, end=1):
    return [mcm.get_cmap(color_map)(x) for x in np.linspace(start, end, n)]


```

```python
def get_node_colors_by_stat(G, data, start=0, end=1):
    df = pd.DataFrame(data=pd.Series(data).sort_values(), columns=['value'])
    df['colors'] = get_color_list(len(df), start=start, end=end)
    df = df.reindex(G.nodes())
    return df['colors'].tolist()


def plotCentralities(G, centrality='betweenness_centrality'):
    """
    plots the centraliy values as plasma colors
    credits to G. Boeing https://github.com/gboeing/osmnx-examples/blob/master/notebooks/06-example-osmnx-networkx
        .ipynb
    accepts centrality input 'betweenness_centrality', 'closeness_centrality' and every other extended stat (not
        tested)
    """

    extended_stats = ox.extended_stats(G, ecc=True, bc=True, cc=True)
    nc = get_node_colors_by_stat(G, data=extended_stats[centrality])
    fig, ax = ox.plot_graph(G, node_color=nc, node_edgecolor='gray', node_size=20, node_zorder=2)

    # for plotting the aoi with betweenness in same pattern (size, axis etc.)
    # bbox_tpl #north,south,east,west as tuple
    # ox.plot_graph(G, bbox_tpl, fig_height=15, fig_width=15, node_color=nc, axis_off=False, equal_aspect=True,
        node_size=50, node_zorder=3, edge_linewidth=2)


def plotScalingResult(data, distances, x_axis ='scaling_soe', y_axis ='k_partitions', exp_ployfit=2):
    """
    plots the scaling as a graph with a fitting curve
    """

    #get the x-labels as list from dict distances
    x_axis = list(np.linspace(distances[x_axis][0],distances[x_axis][1],distances[x_axis][2]))

    if y_axis == 'reduction':
        plt.plot(x_axis, data[2], 'b+')
        plt.ticklabel_format(style='sci', axis='y', scilimits=(0,0))
        plt.ylabel('K_reduced / K_primal')
        plt.xlabel('distance deviation [m]')
        plt.show()

        #fitting
        x = x_axis
        y = list(data[2])

        #polynomial
        z = np.polyfit(x,y,exp_ployfit)
        f = np.poly1d(z)

        x1 = np.linspace(x[0], x[-1], 50)
        y1 = f(x1)

        plt.plot(x,y,'k+', x1, y1)

    elif y_axis == 'k_partitions':
        #plotting xdistance deviation - y k_partitions
        plt.plot(x_axis, data[3], 'b+')
        # plt.ticklabel_format(style='sci', axis='y', scilimits=(0,0))
        plt.ylabel('k partitions')
        plt.xlabel('distance deviation [m]')
        plt.show()

        #fitting
        x = x_axis
        y = list(data[3])

        #polynomial
        z = np.polyfit(x,y,exp_ployfit)
        f = np.poly1d(z)

        x1 = np.linspace(x[0], x[-1], 50)
        y1 = f(x1)

        plt.plot(x,y,'k+', x1, y1)


def plotBoxplotsOfDF(df_y, list_xaxis, xlabel, ylabel, yscale='log', ymin=None, ymax=None, x_ticks=None):
    """
    plotting boxplots of given dataframes. Used to present the scaling of the approach.
    """
```

```
    for idx, item in enumerate(list_xaxis):
        list_xaxis[idx] = int(item)

    #round values
    list_xaxis = [round_value(i) for i in list_xaxis]

    #invert data on yaxis to fit increasing xaxis
    y = df_y[df_y.columns[::-1]]

    #convert to y dataframe to nested list so for each nested list a boxplot can be calculated
    nestedList = []
    for i in y.iterrows():
        nestedList.append(list(i[1]))

    #create plot
    fig = plt.figure(figsize=(20,5))
    ax = plt.subplot(111)
    for idx, item in enumerate(nestedList):
        ax.boxplot(item, positions=[-idx], showfliers=False)
    ax.set_xticklabels(list_xaxis)
    if yscale == 'sci':
        ax.ticklabel_format(style='sci', axis='y', scilimits=(0,0))
    elif yscale == 'log':
        ax.set_yscale('log')
    plt.ylabel(ylabel)
    plt.xlabel(xlabel)
    if ymin and ymax != None:
        plt.ylim([ymin, ymax])
    plt.show()


def plotBoxplotsOfPerformanceDF(df_y, list_xaxis, xlabel, ylabel, yscale='log', ymin=None, ymax=None):
    """
    plotting boxplots of given dataframes regarding the performance. Used to present the scaling of the approach.
    """

    #invert data and cast x-axis to int
    #list_xaxis = list_xaxis[::-1]
    for idx, item in enumerate(list_xaxis):
        list_xaxis[idx] = int(item)

    #round values
    list_xaxis = [round_value(i) for i in list_xaxis]

    #invert data on yaxis to fit increasing xaxis
    y = df_y[df_y.columns[::-1]]

    #convert to y dataframe to nested list so for each nested list a boxplot can be calculated
    nestedList = []
    for i in y.iterrows():
        nestedList.append(list(i[1]))

    #create plot
    fig = plt.figure(figsize=(20,5))
    ax = plt.subplot(111)
    for idx, item in enumerate(nestedList):
        ax.boxplot(item, positions=[-idx], showfliers=False)
    ax.set_xticklabels(list_xaxis)
    if yscale == 'sci':
        ax.ticklabel_format(style='sci', axis='y', scilimits=(0,0))
    elif yscale == 'log':
        ax.set_yscale('log')
    plt.ylabel(ylabel)
    plt.xlabel(xlabel)
    if ymin and ymax != None:
        plt.ylim([ymin, ymax])
    else:
        #plt.ylim([0.01,0.10])
        min_value = df_y.min()[0]
        num_decimals = getNumberOfFloatZeros(min_value)
        ymin,ymax = getMinMaxExponent(num_decimals)
        plt.ylim([ymin,ymax])
    plt.show()

def plotErrorBars(mean, std, figsize, label_xaxis, label_yaxis):
    """
    plot error bars for mean and standard deviation
    """

```

```
    x_pos = list(range(1,len(mean)+1))
    plt.figure(figsize=figsize)
    plt.bar(x_pos, mean, yerr=std, align='center', ecolor='black', capsize=15, alpha=0)
    plt.ylabel(label_yaxis)
    plt.xlabel(label_xaxis)
    plt.xticks(x_pos)
    plt.scatter(x_pos, mean, color='black')
    plt.show()


################### Save_n_Load #####################
def saveProcessedData(G2, nodes, edges, df_proxies, outputName, outputInfo, outputDir='./data/', time_stamp=False)
    :
    """
    save the results into csv (nodes, edges, df_proxies) and
    graphs G,G2 and subgraphs as graphML.
    """

    from datetime import datetime
    import os
    time_stamp = datetime.now().strftime("_%Y_%m_%d_%H_%M")

    if time_stamp == True:
        folder_name = f'{outputDir}{outputName}{outputInfo[0]}{time_stamp}/'
        # folder_name = outputDir + outputName + str(outputInfo[0]) + time_stamp + '/'
        if not os.path.exists(folder_name):
            os.mkdir(folder_name)

        nodes.to_csv(f'{folder_name}{outputName}_nodes{time_stamp}.csv')
        edges.to_csv(f'{folder_name}{outputName}_edges{time_stamp}.csv')
        df_proxies.to_csv(f'{folder_name}{outputName}_proxies{time_stamp}.csv')

        saveNxGraph(G2, filename=f'G_reduced_{outputName}{time_stamp}.json', outputDir=folder_name)

        with open(f'{folder_name}info_{outputName}{time_stamp}.txt','w') as file:
            file.write(str(outputInfo))
    else:
        if not os.path.exists(outputDir):
            os.mkdir(outputDir)

        nodes.to_csv(f'{outputDir}{outputName}_nodes.csv')
        edges.to_csv(f'{outputDir}{outputName}_edges.csv')
        df_proxies.to_csv(f'{outputDir}{outputName}_proxies.csv')

        saveNxGraph(G2, filename=f'G_reduced_{outputName}.json', outputDir=outputDir)

        with open(f'{outputDir}info_{outputName}.txt','w') as file:
            file.write(str(outputInfo))


def loadProcessedData(folder_path, key, debug=False):
    """
    load processed data. Needs folderpath, returns G_reduced, nodes, edges, proxies
    """

    from pathlib import Path
    path = Path(folder_path)
    files = [f for f in path.iterdir() if f.match("*.*")]

    G, nodes, edges, df_proxies = None, None, None, None

    for file in files:
        if ('reduced' in str(file)) and (key in str(file)):
            G = loadNxGraph(file)
        elif 'nodes' in str(file) and key in str(file):
            nodes = gpd.read_file(file, GEOM_POSSIBLE_NAMES="geometry", KEEP_GEOM_COLUMNS="NO")
        elif 'edges' in str(file) and key in str(file):
            edges = gpd.read_file(file, GEOM_POSSIBLE_NAMES="geometry", KEEP_GEOM_COLUMNS="NO")
        elif 'proxies' in str(file) and key in str(file):
            df_proxies = gpd.read_file(file, GEOM_POSSIBLE_NAMES="geometry", KEEP_GEOM_COLUMNS="NO")
        elif 'info' in str(file) and key in str(file):
            with open(file, 'r') as info_text:
                print('')
                print('Info - DD and coords: ' + info_text.read())

        else:
            if debug == True:
                print('could not read file: ' + str(file))

    return G, nodes, edges, df_proxies
```

```


def loadNxGraph(file_path):
    """
    load graph from json style data
    """

    import json
    with open(file_path) as json_file:
        data = json.load(json_file)
    G = nx.adjacency_graph(data)
    return G


def saveNxGraph(G, filename='G_reduced.json', outputDir='./data/'):
    """
    saves the nx graph
    """

    import json
    output = outputDir + filename
    data = nx.adjacency_data(G)
    # data = json.dumps(data)
    with open(output, 'w') as file:
        file.write(json.dumps(data))


def jsonToFile(file, filename, outputDir='./data/'):
    """
    saves a python dict to json file
    """

    import json
    output = outputDir + filename
    with open(output, 'w') as json_file:
            json.dump(file, json_file)


def fileToJson(file):
    """
    reads a file and parse it to json type (dict)
    """

    import json
    with open(file, 'r') as myfile:
        data=myfile.read()
    # parse file
    obj = json.loads(data)
    return obj


def listToFile(list_array, filename='G_reduced.json', outputDir='./data/'):
    """
    stores a list into a file.
    """

    import os
    if not os.path.exists(outputDir):
        os.mkdir(outputDir)
        print(outputDir + ' created.')

    filepath = outputDir + filename
    with open(filepath, 'a') as file:
        for item in list_array:
            file.write('{}\n'.format(item))


def nestedListsToCSV(nested_list, filename='file.csv', outputDir='./data/'):
    """
    writes a nested list to a csv file
    """

    import os, csv
    if not os.path.exists(outputDir):
        os.mkdir(outputDir)
        print(outputDir + ' created.')

    filepath = outputDir + filename
    with open(filepath, 'a') as file:
        writer = csv.writer(file)
```

```
        for item in nested_list:
            writer.writerow(item)


def fileToList(pathToFile):
    """
    loads a file and stores the content in a list
    """

    list_array = []
    with open(pathToFile, 'r') as file:
        for line in file:
            list_array.append(float(line[:-1]))
    return list_array


def loadResults(folder_path, key1, key2):
    """
    Only for separate plotting. Loads results and filters by keywords.
    Parameters
    ----------
    folder_path : TYPE STRING
    key1 : STRING
        Abbreviaton.
    key2 : STRING
        k_partitions, performance_primal, performance_reduced, scaling, space.

    Returns List
    -------
    """

    from pathlib import Path
    path = Path(folder_path)
    files = [f for f in path.iterdir() if f.match("*.*")]

    loaded_list = []
    for file in files:
        if key1 in str(file) and key2 in str(file):
            loaded_list = fileToList(str(file))
    return loaded_list

def loadListsFromFile(folder_path, key1, key2):
    """
    similar to loadResults, but can read lists from text files
    """

    from pathlib import Path
    path = Path(folder_path)
    files = [f for f in path.iterdir() if f.match("*.*")]

    for file in files:
        if key1 in str(file) and key2 in str(file):
            df = pd.read_csv(file, delimiter=',', header=None)
            #remove brackets
            df[0] = df[0].str.strip('[')
            df[3] = df[3].str.strip(']')
            #cast column 2 to float
            df[0] = pd.to_numeric(df[0], errors='coerce')
            df[1] = pd.to_numeric(df[1], errors='coerce')
            df[2] = pd.to_numeric(df[2], errors='coerce')
            df[3] = pd.to_numeric(df[3], errors='coerce')

            return df


def loadScalingDataFromFile(folder_path, key):
    """
    similar to loadResults, but can read csv into pandas df
    """

    from pathlib import Path
    path = Path(folder_path)
    files = [f for f in path.iterdir() if f.match("*.csv")]

    K_primal = ''
    K_reduced = ''
    K_partitions = ''

    for file in files:
        if key in str(file) and 'K_primal' in str(file):
```

```
            K_primal = file
        elif key in str(file) and 'K_reduced' in str(file):
            K_reduced = file
        elif key in str(file) and 'K_partitions' in str(file):
            K_partitions = file

    df_primal = pd.read_csv(K_primal, header=None)
    df_reduced = pd.read_csv(K_reduced, header=None)
    df_partitions = pd.read_csv(K_partitions, header=None)

    return df_primal, df_reduced, df_partitions


def loadCSV2DF(folder_path, key):
    """
    reads all csv files in a folder, load them into multiple dataframes
    dataframes are returned in a list, in case multiple csv files are found
    """

    from pathlib import Path
    path = Path(folder_path)
    files = [f for f in path.iterdir() if f.match("*.csv*")]

    DFs = []
    for file in files:
        if key in str(file):
            DFs.append(pd.read_csv(file))
    return DFs


def readFixAllShortestPaths(results, key):
    """
    read all shortest paths (constant) results for plotting.
    """

    from pathlib import Path
    folder_path = results
    path = Path(folder_path)
    file = [f for f in path.iterdir() if f.match(f'*{key}*.csv')][0]
    #problem that for each row not every column has values. Read it as strings and split it then add NaNs
    tmp_df = pd.read_csv(file, sep='^', header=None, prefix='X')
    tmp_df2 = tmp_df.X0.str.split(',', expand=True)
    del tmp_df['X0']
    tmp_df = pd.concat([tmp_df, tmp_df2], axis=1)
    #convert str values into float
    df = tmp_df.apply(pd.to_numeric)
    return df


def getFileByKey(folder_path, key):
    """
    filters files in a folderpath by a key
    """

    from pathlib import Path
    path = Path(folder_path)
    files = [f for f in path.iterdir() if f.match(key)]
    return files


#################### Lookup ########################
def referingProxies(G, nodes):
    """
    returns a dict with the osmid as keys and the referring proxy as value
    """

    proxydict = {}
    for item in G.nodes():
        proxydict[item] = nodes.loc[nodes['osmid'] == item]['community'].values[0]
    return proxydict


def lookupDistance(startProxy, targetProxy, adjMatrix):
    """
    simple lookup in a matrix without special/performant algorithms.
    """
    s = startProxy -1
    t = targetProxy -1

    return adjMatrix[s][t]
```

```


#################### Evaluation #######################
def all_shortest_paths_statistics(G, weight='length'):
    """
    iterate over each node and computes all shortest path lengths
    """

    nested_path_lengths = []
    for node in G.nodes():
        nested_path_lengths.append(list(nx.single_source_dijkstra_path_length(G, source=node, weight=weight).values()))
    flatList = [item for elem in nested_path_lengths for item in elem]
    #filtering zeros, if origin and destination node is the same
    res = [n for n in flatList if n != 0]
    res_dict = {'mean':np.array(res).mean(),'min':np.array(res).min(), 'max':np.array(res).max(),'std':np.array(res).std(), 'median':np.median(np.array(res)), 'total_n':len(res)}

    return res, res_dict


def rndNodePairs(G, numberOfPairs):
    """
    generate random nodepairs to test the performance for routing/lookup
    """

    i = 0
    rndNodepairs = []
    while i < numberOfPairs:
        rndNodepairs.append((random.choice(list(G.nodes())), random.choice(list(G.nodes()))))
        i+=1
    return rndNodepairs


def performancePrimalGraph(G, nodePairs):
    """
    measures the time for a standard shortest path routing on the primal graph
    """

    start_time = timeit.default_timer()
    criticalPairs = []
    for pair in nodePairs:
        try:
            nx.shortest_path_length(G, source=pair[0], target=pair[1], weight='length')

        except:
            try:
                nx.shortest_path_length(G, source=pair[1], target=pair[0], weight='length')
            except:
                criticalPairs.append(pair)
                pass
    # print('critical pairs: ' + str(criticalPairs))
    stop_time = timeit.default_timer()
    print(f'Critical Pairs: {len(criticalPairs)}')
    print(f'Time - primal: {stop_time-start_time}')
    return (stop_time-start_time)


def performancePrimalGraph_dijkstra(G, nodePairs):
    """
    measures the time for a shortest path routing (dijkstra) on the primal graph
    """

    start_time = timeit.default_timer()
    criticalPairs = []
    for pair in nodePairs:
        try:
            nx.shortest_path_length(G, source=pair[0], target=pair[1], weight='length')

        except:
            try:
                nx.shortest_path_length(G, source=pair[1], target=pair[0], weight='length')
            except:
                criticalPairs.append(pair)
                pass
    # print('critical pairs: ' + str(criticalPairs))
    stop_time = timeit.default_timer()
    print(f'Critical Pairs: {len(criticalPairs)}')
    print(f'Time - primal dijkstra: {stop_time-start_time}')
    return (stop_time-start_time)
```

```python


def performancePrimalGraph_bidirectional_dijk(G, nodePairs):
    """
    measures the time for a shortest path routing (bidirectional Dijkstra) on the primal graph
    """

    start_time = timeit.default_timer()
    criticalPairs = []
    for pair in nodePairs:
        try:
            nx.bidirectional_dijkstra(G, source=pair[0], target=pair[1], weight='length')

        except:
            try:
                nx.bidirectional_dijkstra(G, source=pair[1], target=pair[0], weight='length')
            except:
                criticalPairs.append(pair)
                pass
    # print('critical pairs: ' + str(criticalPairs))
    stop_time = timeit.default_timer()
    print(f'Critical Pairs: {len(criticalPairs)}')
    print(f'Time - primal bidrectional dijk: {stop_time-start_time}')
    return (stop_time-start_time)


def performancePrimalGraph_astar(G, nodePairs):
    """
    measures the time for a shortest path routing (Astar) on the primal graph
    """

    start_time = timeit.default_timer()
    criticalPairs = []
    for pair in nodePairs:
        try:
            nx.star_path_length(G, source=pair[0], target=pair[1], weight='length')

        except:
            try:
                nx.star_path_length(G, source=pair[1], target=pair[0], weight='length')
            except:
                criticalPairs.append(pair)
                pass
    # print('critical pairs: ' + str(criticalPairs))
    stop_time = timeit.default_timer()
    print(f'Critical Pairs: {len(criticalPairs)}')
    print(f'Time - primal astar: {stop_time-start_time}')
    return (stop_time-start_time)


def performanceReducedGraph(adjMatrix, proxydict, nodePairs):
    """
    measures the time for a lookup in the adjacency matrix as an alternative to standard shortest path routing.
    """

    start_time = timeit.default_timer()
    for pair in nodePairs:
        lookupDistance(proxydict[pair[0]], proxydict[pair[1]], adjMatrix)
    stop_time = timeit.default_timer()
    print(f'Time - reduced: {stop_time-start_time}')
    return (stop_time-start_time)


def performance_RoutingMachine(nodePairsCoords):
    """
    only nhttps://github.com/gboeing/osmnxetwork distances as testing from osrm routing machine. Is difficult to compare (local routing machine vs online + internet connection/speed + hardware)
    """

    start_time = timeit.default_timer()
    url = "https://router.project-osrm.org/route/v1/driving/{lon1},{lat1};{lon2},{lat2}?overview=full&geometries=geojson"
    #url = "http://141.5.109.117:5000/route/v1/driving/{lon1},{lat1};{lon2},{lat2}?overview=full&geometries=geojson"

    for pair in nodePairsCoords:
        url = url.format(lat1=pair[0][0], lon1=pair[0][1], lat2=pair[1][0], lon2=pair[1][1])
        data = requests.get(url).json()
        # route = data['routes'][0]['geometry']
        route = data['routes'][0]['legs'][0]['distance']
```

```
    stop_time = timeit.default_timer()
    print(f'Time - OSRM: {stop_time-start_time}')
    return (stop_time-start_time)
    # return route


def osmIdToLatLon(osmid, nodes):
    """
    returns lat/lon for a node based on osmID
    """

    lat = nodes.query(f'osmid == {osmid}').y.values[0]
    lon = nodes.query(f'osmid == {osmid}').x.values[0]
    return (lat, lon)


def rndNodePairsToCoords(nodePairs, nodes):
    """
    transform random node pairs to lat/lon coordinates so a query for a routing machine can be carried out
    """

    rndNodePairsCoords = []
    for pair in nodePairs:
        rndNodePairsCoords.append((osmIdToLatLon(pair[0], nodes),(osmIdToLatLon(pair[1], nodes))))
    return rndNodePairsCoords


##################### Utils ##########################
def calcCircuity(G, circuity_dist='gc'):
    """
    average circuity: sum of edge lengths divided by sum of straight-line
    distance between edge endpoints. first load all the edges origin and
    destination coordinates as a dataframe, then calculate the straight-line
    distance
    """

    from osmnx.utils import great_circle_vec
    from osmnx.utils import euclidean_dist_vec

    edge_length_total = sum([d['length'] for u, v, d in G.edges(data=True)])

    coords = np.array([[G.nodes[u]['y'], G.nodes[u]['x'], G.nodes[v]['y'], G.nodes[v]['x']] for u, v, k in G.edges
        (keys=True)])
    df_coords = pd.DataFrame(coords, columns=['u_y', 'u_x', 'v_y', 'v_x'])
    if circuity_dist == 'gc':
        gc_distances = great_circle_vec(lat1=df_coords['u_y'],
                                        lng1=df_coords['u_x'],
                                        lat2=df_coords['v_y'],
                                        lng2=df_coords['v_x'])
    elif circuity_dist == 'euclidean':
        gc_distances = euclidean_dist_vec(y1=df_coords['u_y'],
                                          x1=df_coords['u_x'],
                                          y2=df_coords['v_y'],
                                          x2=df_coords['v_x'])
    else:
        raise ValueError('circuity_dist must be "gc" or "euclidean"')

    gc_distances = gc_distances.fillna(value=0)
    try:
        circuity_avg = edge_length_total / gc_distances.sum()
    except ZeroDivisionError:
        circuity_avg = np.nan

    return circuity_avg


def projCoords(orig_epsg, dest_epsg, x, y):
    """
    fransforms coordinates by using the epsg-code.
    Common epsg-codes:
        WGS84 lat lon (decimal, unit:degree):   4326
        WGS 84/ UTM 32 N (unit: meters):        32632
        WGS 84/ UTM 33 N (unit: meters):        32633
        WGS 84/ Pseudo Mercator (unit: meters): 3857
    Parameters:

    epsg-codes: str
    example: 'epsg:4326'
    """

```

```
    from pyproj import Proj, transform

    inProj = Proj(orig_epsg)
    outProj = Proj(dest_epsg)
    x2,y2 = transform(inProj,outProj,x,y)
    return x2,y2


def getAreaFromBBox(minLon, minLat, maxLon, maxLat, dest_epsg, projected=True):
    """
    fransforms the coordinates of the bbox in (epsg:4326) to a given target projection by the destination epsg. Calculates the area based on the given units of the epsg projection (meters or kilometers).
    Example:
    WGS 84/ UTM 32 N (unit: meters): 32632
    www.epsg.io
    https://gis.stackexchange.com/questions/59087/calculating-bounding-box-size
    """

    proj_minLon, proj_minLat = projCoords('epsg:4326', dest_epsg, minLon, minLat)
    proj_maxLon, proj_maxLat = projCoords('epsg:4326', dest_epsg, maxLon, maxLat)

    if projected == False:
        print('not implemented yet. Example for spheres: https://gis.stackexchange.com/questions/59087/calculating-bounding-box-size')
    else:
        area = (proj_minLon - proj_maxLon) * (proj_minLat - proj_maxLat)

    return area


def graphStatistics(G_primal, G_reduced, nodes, dest_epsg):
    """
    returns statistics of the graph:
    circuity
    area of the bbox
    distance of all shortest paths
    returns the reduction in percent
    """

    minx, miny, maxx, maxy = nodes.total_bounds
    area = getAreaFromBBox(minx, miny, maxx, maxy, dest_epsg=dest_epsg)
    # area = getAreaFromBBox(bbox[2], bbox[0], bbox[3], bbox[1], dest_epsg=dest_epsg)

    #to undirected graphs
    G_primal_undirected = G_primal.to_undirected()
    G_reduced_undirected = G_reduced.to_undirected()

    sum_network_distances_primal = sum([d['length'] for u, v, d in G_primal_undirected.edges(data=True)])
    sum_network_distances_reduced = sum([d['length'] for u, v, d in G_reduced_undirected.edges(data=True)])

    print(f'''\n Graph stats:\n circuity: {round(calcCircuity(G_primal, circuity_dist='gc'),6)}''')

    print(f' area: {round(area/1000000,2)} km^2')
    print(f' total street length - primal: {round(sum_network_distances_primal,0)}m')
    print(f' total street length - reduced: {sum_network_distances_reduced}m')
    print(f' reduction: {round(100-(sum_network_distances_reduced/sum_network_distances_primal)*100,2)}%')
    print(f' is complete: {isComplete(G_reduced)}')

    res = {} #units
    res.update({'area':f'{round(area/1000000,2)}'}) # km^2
    res.update({'total street length - primal':f'{round(sum_network_distances_primal,0)}'}) #meter
    res.update({'total street length - reduced':f'{sum_network_distances_reduced}'}) #meter
    res.update({'reduction':f'{round(100-(sum_network_distances_reduced/sum_network_distances_primal)*100,2)}'}) # percentage
    res.update({'is complete':f'{isComplete(G_reduced)}'})

    return res
    # return round(100-(sum_network_distances_reduced/sum_network_distances_primal)*100,2)


def dfStatisticsPerRow(df):
    """
    returns statistics (mean, median, std) per row of a dataframe
    """

    mean = []
    median = []
    std = []

    for row in df.iterrows():
```

```
        tmp = np.array(row)[1]
        mean.append(tmp.mean())
        median.append(tmp.median())
        std.append(tmp.std())
    return mean, median, std


def calcEdgesComplete(G):
    """
    calculates the number of edges the complete graph would have
    """

    n = len(G.nodes())
    m = (n*(n-1))/2
    return m


def getBBoxForPlotting(nodes, margin=0.03):
    """
    take total bounds of nodes and add margin for plotting, default three percent
    """

    xmin, ymin, xmax, ymax = nodes.total_bounds

    xmin = xmin - (xmax - xmin)*margin
    xmax = xmax + (xmax - xmin)*margin
    ymin = ymin - (ymax - ymin)*margin
    ymax = ymax + (ymax - ymin)*margin

    bbox = xmin, xmax, ymin, ymax
    return bbox


def isComplete(G):
    """
    checks if the graph is a complete graph. Returns boolean
    """

    n = len(G.nodes())
    if (n*(n-1))/2 == len(G.edges()):
        return True
    else:
        return False


def getAdjMatrix(G, check=False, structure='df'):
    """
    creates the adjacency matrix as a pandas df
    sort the nodes by community id
    """

    unsortedNodeList = list(G.nodes('community'))
    unsortedNodeList.sort(key = lambda x: x[1][1])

    sortedNodeList = [item[0] for item in unsortedNodeList]

    df = nx.to_pandas_adjacency(G, nodelist=sortedNodeList, weight='length')

    if check == True:
        if (sortedNodeList[0], sortedNodeList[-1]) == (df.columns[0], df.columns[-1]):
            print('sorting check ok')
        else:
            print('sorting failed')

    if structure == 'df':
        return df
    elif structure == 'np':
        return df.to_numpy()


def listStatistics(list_array):
    """
    prints standard numpy statistics (min, max, mean, std, variance of a list
    returns a dict
    """

    results = np.array(list_array)

    dict_results = {}
    dict_results['variance'] = np.var(results)
```

```
    dict_results['mean'] = np.mean(results)
    dict_results['median'] = np.median(results)
    dict_results['deviation'] = np.std(results)
    dict_results['min'] = np.min(results)
    dict_results['max'] = np.max(results)

    print(f'''variance: {dict_results['variance']}''')
    print(f'''mean: {dict_results['mean']}''')
    print(f'''median: {dict_results['median']}''')
    print(f'''deviation: {dict_results['deviation']}''')
    print(f'''min: {dict_results['min']}''')
    print(f'''max: {dict_results['max']}\n''')

    return dict_results


def getScalingIterations(info, scaling_DD):
    """
    get the steps of the uses parameter for the scaling approach
    """

    #if DD is not int - may be an old relict
    scaling_DD = [int(DD) for DD in scaling_DD]
    #create the iterations steps
    iterations_steps = list(range(info['scaling_DD_step']))
    #zip scaling_DD and iterations_steps then cast it as string, replace ", " with "_" and remove brakets
    zipped_info = list(zip(scaling_DD, iterations_steps))
    res = []
    for item in zipped_info:
        res.append(str(item).replace(', ', '_'))
    #regular expressions to remove brakets
    import re
    res = ([re.sub('[^a-zA-Z0-9-_]+', '', x) for x in res])
    return res


def weightedAvgAndStd(values, weights):
    """
    Return the weighted average/mean and standard deviation.
    values, weights -- Numpy ndarrays with the same shape.
    """

    import math
    average = np.average(values, weights=weights)
    # Fast and numerically precise:
    variance = np.average((values-average)**2, weights=weights)
    return (average, math.sqrt(variance))


def isZero(n):
    """
    checks if n is zero.
    """

    try:
        if int(n) == 0:
            return True
        else:
            return False
    except:
        return False


def getNumberOfFloatZeros(number):
    """
    get number of digits of a float which are null. Helpful for small values to determine the correct exponent for
        better plotting
    """

    str_num = str(number).split('.')[1]
    for idx, char in enumerate(str_num):
        if isZero(char) == True:
            continue
        else:
            break
    return idx


def getMinMaxExponent(min_decimals):
    """
```

```
    get the minimal exponent for plotting. For multiple datasets, check how many digits (zeros) a floating number
        has after less than 1.
    """

    min_limit = 1*10**-(min_decimals+1)
    max_limit = 1*10**-(min_decimals)
    return (min_limit, max_limit)


def round_value(value, digit=2):
    """
    round values to a default digit of 2
    """

    return int(round(value * 0.001,digit)*1000)


###################### Server #########################
def initArgParser():
    """
    reads external parameter from console, returns a dict with AOI, start, stop
    step, iterations
    """

    import argparse
    parser = argparse.ArgumentParser(description='Parameters AOI, fix_DD, scaling_start, scaling_stop,
        scaling_step')

    parser.add_argument('AOI', type=str,
                        help='A required integer positional argument')

    parser.add_argument('fix_DD', type=int,
                        help='A required integer positional argument')

    parser.add_argument('scaling_DD_start', type=int,
                        help='A required integer positional argument')

    parser.add_argument('scaling_DD_stop', type=int,
                        help='A required integer positional argument')

    parser.add_argument('scaling_DD_step', type=int,
                        help='A required integer positional argument')

    parser.add_argument('iterations', type=int,
                        help='A required integer positional argument')

    args = parser.parse_args()

    params = {'AOI': args.AOI, 'fix_DD': args.fix_DD, 'scaling_DD_start': args.scaling_DD_start,
              'scaling_DD_stop':args.scaling_DD_stop, 'scaling_DD_step':args.scaling_DD_step,
              'iterations': args.iterations}
    return params
```

Listing B.2: Community_proxies.py

```
################# Load Parameters #####################
import AOI,community_proxies
import os, json
import numpy as np, networkx as nx
from datetime import datetime

start_time = datetime.now().strftime("\%H:\%M:\%S/\%d.\%m.\%Y")

try:
    params = community_proxies.initArgParser()

    AOI = getattr(AOI, params['AOI'])

    if  params['AOI'] == 'malaga':
        coords = AOI.center
    else:
        coords = [AOI.north, AOI.south, AOI.east, AOI.west]

    fix_distance_deviation = params['fix_DD']
    scaling_distance_deviation = np.linspace(params['scaling_DD_start'], params['scaling_DD_stop'], params['
        scaling_DD_step'])

    n = params['iterations']
except:
```

```
    raise ValueError('Input parameters could not be read')

time_stamp = datetime.now().strftime("\%Y_\%m_\%d_\%H_\%M")
calc_time_start = datetime.now()

output_folder = f'./results/{time_stamp}_{AOI.abbreviation}/'
if not os.path.exists(output_folder):
    os.mkdir(output_folder)

#write the parameters from console / setup in a json-file
with open(output_folder + 'info_' + AOI.abbreviation + '.json','w') as file:
    file.write(json.dumps(params))

#write the distance deviations for scaling into a file (mostly x-axis for plots)
community_proxies.listToFile(scaling_distance_deviation, filename='scaling_distance_deviations_{}.txt'.format(AOI.
    abbreviation), outputDir= output_folder)


################## constant DD #######################
##### Order
### Kpartitions with iterations n
### ratio complete graphs K_primal, K_reduced
### performance
### shortest paths distances - distribution
### stores graph files

k_partitions = []
primal_complete = []
reduced_complete = []
numberOfPairs = 50
results_performance_primal_dijk = []
results_performance_primal_bidijk = []
results_performance_primal_astar = []
results_performance_reduced = []
G = community_proxies.getPrimalGraph(coords)
sp_distances = []
sp_distances_dict = {}


for x in range(1,n):
    nodes, edges, df_proxies, subgraphs = community_proxies.evalOptimumK(G,
            fix_distance_deviation, k=2, algorithm='fluid', proxy_centrality='CL')

    ## k partitions
    k_partitions.append(len(subgraphs))

    print(f'Finished iteration {x} of {n} in part K_partitions.')
    print(f'Duration since start - in hours: {divmod((datetime.now() - calc_time_start).total_seconds(), 3600)
        [0]}, in mins: {divmod((datetime.now() - calc_time_start).total_seconds(), 60)[0]}.')

    ## size - construct a new reduced graph
    G2 = community_proxies.constructGraph(G, df_proxies, nodes, edges, completeGraph = True)

    primal_complete.append(community_proxies.calcEdgesComplete(G))
    reduced_complete.append(community_proxies.calcEdgesComplete(G2))
    print(f'Finished iteration {x} of {n} in part size.')
    print(f'Duration since start - in hours: {divmod((datetime.now() - calc_time_start).total_seconds(), 3600)
        [0]}, in mins: {divmod((datetime.now() - calc_time_start).total_seconds(), 60)[0]}.')

    ##performance
    #get adjacency matrix in np or df
    adjMatrix = community_proxies.getAdjMatrix(G2, check=False, structure='np')

    #prepare a performant lookup to identify the proxies for every primal node
    referringProxies = community_proxies.referingProxies(G, nodes)
    G_dir = nx.DiGraph(G)
    rndNodePairs = community_proxies.rndNodePairs(G, numberOfPairs=numberOfPairs)

    results_performance_primal_dijk.append(community_proxies.performancePrimalGraph_dijkstra(G_dir, rndNodePairs))
    results_performance_primal_bidijk.append(community_proxies.performancePrimalGraph_bidirectional_dijk(G_dir,
        rndNodePairs))
    results_performance_primal_astar.append(community_proxies.performancePrimalGraph_astar(G_dir, rndNodePairs))
    results_performance_reduced.append(community_proxies.performanceReducedGraph(adjMatrix, referringProxies,
        rndNodePairs))
    print(f'Finished iteration {x} of {n} in part performance.')
    print(f'Duration since start - in hours: {divmod((datetime.now() - calc_time_start).total_seconds(), 3600)
        [0]}, in mins: {divmod((datetime.now() - calc_time_start).total_seconds(), 60)[0]}.')

    ## shortest paths distances - distribution
    # sp_distances.append(list(community_proxies.all_shortest_paths_statistics(G2, weight='length')))
```

```
    raw_statistics, dict_statistics = community_proxies.all_shortest_paths_statistics(G2, weight='length')
    sp_distances.append(raw_statistics)
    sp_distances_dict.update(dict_statistics)


### export data
community_proxies.listToFile(k_partitions, filename='fix_k_partitions_{}.txt'.format(AOI.abbreviation), outputDir=output_folder)
community_proxies.listToFile(primal_complete, filename='fix_K_primal_{}.txt'.format(AOI.abbreviation), outputDir=output_folder)
community_proxies.listToFile(reduced_complete, filename='fix_K_reduced_{}.txt'.format(AOI.abbreviation), outputDir=output_folder)
community_proxies.listToFile(results_performance_primal_dijk, filename='fix_performance_primal_dijk_{}.txt'.format(AOI.abbreviation), outputDir=output_folder)
community_proxies.listToFile(results_performance_primal_bidijk, filename='fix_performance_primal_bidijk_{}.txt'.format(AOI.abbreviation), outputDir=output_folder)
community_proxies.listToFile(results_performance_primal_astar, filename='fix_performance_primal_astar_{}.txt'.format(AOI.abbreviation), outputDir=output_folder)
community_proxies.listToFile(results_performance_reduced, filename='fix_performance_reduced_{}.txt'.format(AOI.abbreviation), outputDir=output_folder)
community_proxies.nestedListsToCSV(sp_distances, filename='fix_distribution_all_shortest_path_distances_raw_{}.csv'.format(AOI.abbreviation), outputDir=output_folder)
community_proxies.jsonToFile(sp_distances_dict, filename='fix_distribution_all_shortest_path_distances_statistics_{}.json'.format(AOI.abbreviation), outputDir=output_folder)


#################### scaling DD #####################
scaling_primal_complete = []
scaling_reduced_complete = []
scaling_k_partitions = []
results_performance_primal_dijk = []
results_performance_primal_bidijk = []
results_performance_primal_astar = []
results_performance_reduced = []
scaling_sp_distances = []
scaling_sp_distances_dict = {}

for distance in scaling_distance_deviation:
    tmp_scaling_primal_complete = []
    tmp_scaling_reduced_complete = []
    tmp_scaling_k_partitions = []
    tmp_results_performance_primal_dijk = []
    tmp_results_performance_primal_bidijk = []
    tmp_results_performance_primal_astar = []
    tmp_results_performance_reduced = []
    tmp_scaling_sp_distances = []

    for iteration in range(1,n):
        nodes, edges, df_proxies, subgraphs = community_proxies.evalOptimumK(G,
                distance, k=2, algorithm='fluid', proxy_centrality='CL')

        #construct a new reduced graph
        G2 = community_proxies.constructGraph(G, df_proxies, nodes, edges, completeGraph = True)

        #store the graph, nodes and subgraphs for further postprocessing
        info = (distance, coords)
        df_proxies = community_proxies.updateDfProxiesInteriorNodes(df_proxies)
        df_proxies = community_proxies.updateDfProxiesSubgraphSize(df_proxies)
        community_proxies.saveProcessedData(G2, nodes, edges, df_proxies, outputName=f'{AOI.abbreviation}_{int(distance)}_iteration_{iteration}', outputInfo=info, outputDir=f'{output_folder}/data/')

        tmp_scaling_primal_complete.append(community_proxies.calcEdgesComplete(G))
        tmp_scaling_reduced_complete.append(community_proxies.calcEdgesComplete(G2))
        tmp_scaling_k_partitions.append(len(G2.nodes()))
        print(f'Finished iteration {iteration} of {n} for distance {distance} in part size scaling.')
        print(f'Duration since start - in hours: {divmod((datetime.now() - calc_time_start).total_seconds(), 3600)[0]}, in mins: {divmod((datetime.now() - calc_time_start).total_seconds(), 60)[0]}.')

        #get adjacency matrix in np or df
        adjMatrix = community_proxies.getAdjMatrix(G2, check=False, structure='np')

        #prepare a performant lookup to identify the proxies for every primal node
        referringProxies = community_proxies.referingProxies(G, nodes)
        rndNodePairs = community_proxies.rndNodePairs(G, numberOfPairs=numberOfPairs)

        #tmp_results_performance_primal.append(community_proxies.performancePrimalGraph(G, rndNodePairs))
        tmp_results_performance_primal_dijk.append(community_proxies.performancePrimalGraph(G_dir, rndNodePairs))
        tmp_results_performance_primal_bidijk.append(community_proxies.performancePrimalGraph(G_dir, rndNodePairs))
        tmp_results_performance_primal_astar.append(community_proxies.performancePrimalGraph(G_dir, rndNodePairs))
```

```
        tmp_results_performance_reduced.append(community_proxies.performanceReducedGraph(adjMatrix, referringProxies, rndNodePairs))

        #all shortest path distances
        raw_statistics, dict_statistics = community_proxies.all_shortest_paths_statistics(G2, weight='length')
        # tmp_scaling_sp_distances.append(community_proxies.all_shortest_paths_statistics(G2, weight='length'))
        tmp_scaling_sp_distances.append(raw_statistics)
        tmp_scaling_sp_distances_dict = {f'''{int(distance)}_{iteration}''':dict_statistics}
        scaling_sp_distances_dict.update(tmp_scaling_sp_distances_dict)



    scaling_primal_complete.append(tmp_scaling_primal_complete)
    scaling_reduced_complete.append(tmp_scaling_reduced_complete)
    scaling_k_partitions.append(tmp_scaling_k_partitions)
    results_performance_primal_dijk.append(tmp_results_performance_primal_dijk)
    results_performance_primal_bidijk.append(tmp_results_performance_primal_bidijk)
    results_performance_primal_astar.append(tmp_results_performance_primal_astar)
    results_performance_reduced.append(tmp_results_performance_reduced)
    scaling_sp_distances.append(tmp_scaling_sp_distances)


### export the results - completeG2, completeG, reduction
community_proxies.nestedListsToCSV(scaling_primal_complete, filename='scaling_K_primal_{}.csv'.format(AOI.abbreviation), outputDir= output_folder)
community_proxies.nestedListsToCSV(scaling_reduced_complete, filename='scaling_K_reduced_{}.csv'.format(AOI.abbreviation), outputDir= output_folder)
community_proxies.nestedListsToCSV(scaling_k_partitions, filename='scaling_K_partitions_{}.csv'.format(AOI.abbreviation), outputDir= output_folder)
community_proxies.nestedListsToCSV(results_performance_primal_dijk, filename='scaling_performance_primal_dijk_{}.csv'.format(AOI.abbreviation), outputDir=output_folder)
community_proxies.nestedListsToCSV(results_performance_primal_bidijk, filename='scaling_performance_primal_bidijk_{}.csv'.format(AOI.abbreviation), outputDir=output_folder)
community_proxies.nestedListsToCSV(results_performance_primal_astar, filename='scaling_performance_primal_astar_{}.csv'.format(AOI.abbreviation), outputDir=output_folder)
community_proxies.nestedListsToCSV(results_performance_reduced, filename='scaling_performance_reduced_{}.csv'.format(AOI.abbreviation), outputDir=output_folder)
community_proxies.nestedListsToCSV(scaling_sp_distances, filename='scaling_distribution_all_shortest_path_distances_raw_{}.csv'.format(AOI.abbreviation), outputDir=output_folder)
community_proxies.jsonToFile(scaling_sp_distances_dict, filename='scaling_distribution_all_shortest_path_distances_statistics_{}.json'.format(AOI.abbreviation), outputDir=output_folder)

end_time = datetime.now().strftime("\%H:\%M:\%S/\%d.\%m.\%Y")
print('start: ' + str(start_time))
print('end: ' + str(end_time))
```

Listing B.3: FCGBOP.py

B.2. Source Code for Accumulative Cost Surface Analysis (ACSA)

The following code is intended to provide an overview of the software. To work with it, we recommend using the tested code from the repository. Python version 3.7.6 was used and the required libraries with their version numbers are listed in the requirements.txt in the repository.

Repository: `https://github.com/fauceta/ACSA`

Requirements: `https://github.com/fauceta/ACSA/blob/master/requirements.txt`

For inquiries please contact: armin.hahn@ds.mpg.de

```
import itertools, numpy as np, gdal,sys, time, osr
from PIL import Image
from sortedcontainers import SortedList
from pathlib import Path
import utils
from time import time

def acc_cost_hori_vert(cost1, cost2):
        """
        calculates the cost to travel from one cell to another in horizntal ore diagonal direction
        """
        return (cost1 + cost2) / 2

def acc_cost_diagonal(cost1, cost2):
        return (1.414214 * (cost1 + cost2) / 2)

def zero_source_cells(source_raster):
        """
        sets all cells in the calculation raster to 0 when they are source cells in the source raster
        """
        global active_list
        dim_n, dim_m = len(source_raster[1]), len(source_raster[0])
        #dim_n, dim_m = source_raster.shape[0], source_raster.shape[1]
        calc_raster = np.full([dim_n, dim_m], 0.).astype(float)


        array_positions = np.array(list(zip(np.where(source_raster == 1)[0], np.where(source_raster == 1)[1])))
        for row in array_positions:
                calc_raster[row[0], row[1]] = 0
                active_list.add((0, (row[0], row[1])))

        array_positions = np.array(list(zip(np.where(source_raster == 0)[0], np.where(source_raster == 0)[1])))
        for row in array_positions:
                calc_raster[row[0], row[1]] = 700000
        return calc_raster

def write_to_backlink_raster(neighbor_cell_position):
        """
        return direction value depending on the relative position which is given
        """
        a, b = neighbor_cell_position[0], neighbor_cell_position[1]
        if (a == 0 and b == 1):
                return 1
        elif (a == 1 and b == 1):
                return 2
        elif (a == 1 and b == 0):
                return 3
        elif (a == 1 and b == -1):
                return 4
        elif (a == 0 and b == -1):
                return 5
        elif (a == -1 and b == -1):
```

```
        return 6
    elif (a == -1 and b == 0):
        return 7
    elif (a == -1 and b == 1):
        return 8


def find_neighbors(cell, calc_raster):
    """
    detect neighbours (Queens Pattern), checks if these positions are inside the matrix
    and returns the neighbour positions in a list.
    """

    i = cell[0]
    j = cell[1]
    neighbours_list = []
    neighbor_positions=(
        (-1,0),         #above
        (0,-1),         #left
        (1,0),          #below
        (0,1),          #right
        (-1,1),         #above right
        (1,1),          #below right
        (1,-1),         #below left
        (-1,-1)         #above left
    )
    #loop all neighbours of input cell
    for neighbor in neighbor_positions:
        #calculate absolute cell position in matrix
        cell = i + neighbor[0], j + neighbor[1]
        # check if neighbor inside matrix and not a Source Cell
        if np.all(0 <= cell[0] < dim_m) and np.all(0 <= cell[1] < dim_m) and np.all(calc_raster[cell] != 0):
            neighbours_list.append((cell))

    return neighbours_list

def acs_Algorithm(active_list, merged_cost_array, calc_raster, backlink_raster, output_raster):
    """
    implementation of the modified dijkstra algorithm to generate an Accumulative Cost Surface and a Backlink Raster
    """

    global dim_m, dim_n

    while(1):

        if (len(active_list) == 0):
            break


        min_active_list = active_list[0]
        min_active_list_value = active_list[0][0]

        current_cell_position = (min_active_list[1][0], min_active_list[1][1])
        current_cell_value = merged_cost_array[current_cell_position] # the vlaue of the current cell position

        #remove the lowest cell in the active_list
        active_list.pop(0)
        output_raster[current_cell_position] = min_active_list_value# write to output raster

        #detect all neighbours on moores neighbourhood
        neighbors = find_neighbors(current_cell_position, calc_raster)  #find neighbor positions of the cell


        i = 0
        #loop through all neighbors of the current cell
        for n in neighbors:

            #get the cell value of the neighbor
            neighbor_cell_value = merged_cost_array[n[0], n[1]]

            if (output_raster[neighbors[i][0], neighbors[i][1]] == 0): ##only if neighbor has no entry in the output raster
                neighbor_cell_position = np.subtract(neighbors[i], current_cell_position) #get the relative position of the neighbor to the current cell e.g. (0,1) (-1,0), ...

                #Calculate Cost to travel from one cell to another
```

```python
                if (neighbor_cell_position[0] == 0) or (neighbor_cell_position[1] == 0): # vertical
                    value = min_active_list_value + acc_cost_hori_vert(current_cell_value, neighbor_cell_value)
                else: # diagonal
                    value = min_active_list_value + acc_cost_diagonal(current_cell_value, neighbor_cell_value)

                #get the old value in the calc_raster
                old_value = calc_raster[neighbors[i][0], neighbors[i][1]]

                #if the calculated value is lower
                if (value < old_value):

                    #replace current value in the calc_raster use the new value

                    calc_raster[neighbors[i][0], neighbors[i][1]] = value

                    #calculate Direction value
                    direction_value = write_to_backlink_raster(neighbor_cell_position)

                    # write direction value to the Backlink Raster
                    backlink_raster[neighbors[i][0], neighbors[i][1]] = direction_value


                    #add calculated value to the active_list
                    active_list.add((calc_raster[neighbors[i]], (neighbors[i][0],neighbors[i][1])))

                    #700000.0 indicates that the cell is unvisited.
                    if (old_value != 700000.0):

                        #remove value from the active list
                        active_list.remove((old_value,(neighbors[i][0],neighbors[i][1])))
            i = i + 1

    #return backlink and accumulative cost surface (output_raster)
    return backlink_raster, output_raster

def array_to_raster(newRasterfn, dataset, array, d_type):
    """
    save GTiff file from numpy.array
    This function was derived from https://gis.stackexchange.com/questions/247906/how-to-create-an-rgb-geotiff-file-raster-from-bands-using-the-gdal-python-module
    """
    dim_m, dim_n = array.shape[1], array.shape[0]
    originX, pixelWidth, b, originY, d, pixelHeight = dataset.GetGeoTransform()

    driver = gdal.GetDriverByName('GTiff')

    # set data type to save.
    GDT_d_type = gdal.GDT_Unknown
    if d_type == "Byte":
        GDT_d_type = gdal.GDT_Byte
    elif d_type == "Float32":
        GDT_d_type = gdal.GDT_Float32
    else:
        print("Not supported data type.")

    # set number of band.
    if array.ndim == 2:
        band_num = 1
    else:
        band_num = array.shape[2]

    outRaster = driver.Create(newRasterfn, dim_m, dim_n, band_num, GDT_d_type)
    outRaster.SetGeoTransform((originX, pixelWidth, 0, originY, 0, pixelHeight))

    for b in range(band_num):
        outband = outRaster.GetRasterBand(b + 1)
        if band_num == 1:
            outband.WriteArray(array)
        else:
            outband.WriteArray(array[:,:,b])

    # setteing srs from input tif file.
    prj=dataset.GetProjection()
    outRasterSRS = osr.SpatialReference(wkt=prj)
    outRaster.SetProjection(outRasterSRS.ExportToWkt())
    outband.FlushCache()
```

```
        return outRaster

def change_values_by_threshhold(input_array, threshhold, values):
        """
        changes all values to one of two possible values in an array depending on the current a value and a
             threshold
        Example:
        change_values_by_threshold(array, 2, (0,4))
        replace every value in array with 0 if below 2 or if higher than two value of the cell is set to 4
        """
        array_positions = np.array(list(zip(np.where(input_array < threshhold)[0], np.where(input_array <
             threshhold)[1]))) ##get positions in array
        for row in array_positions:
                input_array[row[0], row[1]] = values[0]

        array_positions = np.array(list(zip(np.where(input_array >= threshhold)[0], np.where(input_array >=
             threshhold)[1]))) ##get positions in array
        for row in array_positions:
                input_array[row[0], row[1]] = values[1]

        return input_array

def delete_values_by_threshhold(array_input, array_change, change_value):
        """
        deletes value in an array_input depending on the values in array_change and the change_value.
        """
        #identify array positions
        array_positions = np.array(list(zip(np.where(array_input == change_value)[0], np.where(array_input ==
             change_value)[1]))) ##get positions in array
        #set all values in the array to 0
        for row in array_positions:
                array_change[row[0], row[1]] = 0
        return array_change

def create_cost_surfaces(data, thresholds, output_folder, weightComb):
        """ """
        global active_list, dim_m, dim_n

        #read .tif files to arrays
        ndvi_array = np.array(data.vegetation.ReadAsArray())
        slope_array = np.array(data.slope.ReadAsArray())
        building_array = np.array(data.buildings_raster.ReadAsArray())
        street_array = np.array(data.road_network.ReadAsArray())

        #get the two dimensions of the numpy array
        dim_n, dim_m = len(ndvi_array[1]), len(ndvi_array[0])

        #initialize weighting to their features
        ndvi_weight, slope_weight, buildings_weight = weightComb[0], weightComb[1], weightComb[2]
        paved_weight = 1

        #create filename based on weight
        filename = f'{ndvi_weight}{slope_weight}{buildings_weight}'

        #set paths for backlink raster
        filepath_backlink = f'{output_folder}/Backlink_Raster/backlink_{filename}.tif'
        utils.createFolder(f'{output_folder}/Backlink_Raster/')

        my_file = Path(filepath_backlink)
        #if !my_file.is_file():
        #       raise.

        #apply thresholds for binary results
        # 0 = no-vegetation, 1 = vegetation ; 0 = passable, 1 = not_passable
        # to reduce computations, the weighting can replace the binary 1, since the weighting is 1x weighting =
             weighting
        # vegetation when higher then trashold, passable when lower than threshold
        ndvi_array = change_values_by_threshhold(ndvi_array,thresholds.vegetation,(ndvi_weight, 0))
        slope_array = change_values_by_threshhold(slope_array,thresholds.slope, (0,slope_weight))

        #set weighting for buildings
        array_positions = np.array(list(zip(np.where(building_array == 1)[0], np.where(building_array == 1)[1])))
             ##get positions in array
        for row in array_positions:
                building_array[row[0], row[1]] = buildings_weight

        array_positions = np.array(list(zip(np.where(building_array == 127)[0], np.where(building_array == 127)
             [1]))) ##get positions in array
        for row in array_positions:
                building_array[row[0], row[1]] = 0
```

```

        #remove values from the slope raster where building array has the value 1 (building present)
        slope_array = delete_values_by_threshhold(building_array, slope_array, 1)

        #generate merged cost array based on the weights and the discrete arrays
        merged_cost_array = (ndvi_array) + (building_array) + (slope_array) # calculate cost surface array with weighted input arrays

        #set cells to paved_weight where cells are 0 in merged cost array
        array_positions = np.array(list(zip(np.where(merged_cost_array == 0)[0], np.where(merged_cost_array == 0)[1]))) ##get positions in array
        for row in array_positions:
            merged_cost_array[row[0], row[1]] = paved_weight

        #export array for merged_cost_array as a .tif file
        path = f'cost_raster{filename}.tif'
        filepath = f'{output_folder}/Cost_Surfaces/{path}'
        utils.createFolder(f'{output_folder}/Cost_Surfaces/')
        array_to_raster(str(filepath), data.vegetation, merged_cost_array, "Float32")

        #remove possible negatives
        merged_cost_array = np.absolute(merged_cost_array)
        #convert to float values
        merged_cost_array = merged_cost_array.astype(float)
        source_raster = street_array


        #initialize list of active cells
        active_list = []
        active_list = SortedList(active_list)

        output_raster = np.full([dim_n, dim_m], 0.).astype(float)
        backlink_raster = np.full([dim_n, dim_m], 0.).astype(float)

        #set source cells to 0 in the output raster
        calc_raster = zero_source_cells(source_raster)
        backlink_raster, output_raster = acs_Algorithm(active_list, merged_cost_array, calc_raster, backlink_raster, output_raster)

        path_accum_cost = f'acc_cost_{filename}.tif'
        filepath_accum = f'{output_folder}/Accumulative_Cost_Surfaces/{path_accum_cost}'
        utils.createFolder(f'{output_folder}/Accumulative_Cost_Surfaces/')

        array_to_raster(filepath_backlink, data.road_network, backlink_raster, "Byte")
        array_to_raster(filepath_accum, data.road_network, output_raster, "Float32")
```

Listing B.4: acsa.py

```
import numpy as np, geopandas as gpd, pandas as pd, itertools
from osgeo import ogr, gdal, osr
from math import sin, cos, sqrt, atan2, radians, degrees
from pathlib import Path
from pyproj import Proj
import utils
from time import time

def read_shape_to_array(input_shape, reference_img, output_image):
    """
    this functions burns shapefiles into a numpy array with the value 1 for objects in the shapefile and returns this grid
    """
    #initialise paramaters
    g_format = 'GTiff'
    datatype = gdal.GDT_Byte
    raster_value = 1
    # Get properties of reference_img
    img = gdal.Open(reference_img, gdal.GA_ReadOnly)
    # load the shp
    shp = ogr.Open(input_shape)
    shp_layer = shp.GetLayer()
    # rasterise the shp using raster_value
    output = gdal.GetDriverByName(g_format).Create(output_image, img.RasterXSize, img.RasterYSize, 1, datatype, options=['COMPRESS=DEFLATE'])
    output.SetProjection(img.GetProjectionRef())
    output.SetGeoTransform(img.GetGeoTransform())
    # store raster in band 1 of the tiff
    gdal.RasterizeLayer(output, [1], shp_layer, burn_values=[raster_value])
    #reset parameters
    output,  = None
```

```
    img = None
    shp = None
    shape_array = gdal.Open(output_image).ReadAsArray()
    return(shape_array)


def shortest_path(backlink_raster_array, destination):
    """
    calculates and returns the shortest path on a Backlink Raster for a given Destination to the closest Source Cell.
    """
    current_position = destination
    path_list = []# initate path list with the destination position
    while (1): #loop until current cell is source cell
    #[(,),(,)]
        value = backlink_raster_array[current_position[0], current_position[1]]
        #print(value)

        path_list.append(current_position)
        if(value == 0):
            calculated_snapping_point =  pixel2coord(current_position[0], current_position[1])#path_coords_list[-1][0], path_coords_list[-1][1] #last element
            return path_list
        elif(value == 1):
            relative_position = (0, -1)
        elif(value == 2):
            relative_position = (-1, -1)
        elif(value == 3):
            relative_position = (-1, 0)
        elif(value == 4):
            relative_position = (-1, 1)
        elif(value == 5):
            relative_position = (0, 1)
        elif(value == 6):
            relative_position = (1, 1)
        elif(value == 7):
            relative_position = (1, 0)
        elif(value == 8):
            relative_position = (1, -1)
        current_position = (current_position[0] + relative_position[0], current_position[1] + relative_position[1])



def burn_raster_by_polygon(reference_img, output_image, polygon):
    """
    this functions burns polygons into a numpy array with the value 1 for objects in the shapefile and returns this grid
    """
    g_format = 'GTiff'
    datatype = gdal.GDT_Byte
    raster_value = 1

    # Get projection
    Image = gdal.Open(reference_img, gdal.GA_ReadOnly)
    Output = gdal.GetDriverByName(g_format).Create(output_image, Image.RasterXSize, Image.RasterYSize, 1, datatype)
    Output.SetProjection(Image.GetProjectionRef())
    Output.SetGeoTransform(Image.GetGeoTransform())

    rast_ogr_ds = \
    ogr.GetDriverByName('Memory').CreateDataSource('wrk')
    sr = osr.SpatialReference()
    sr.ImportFromEPSG(25832)
    rast_mem_lyr = rast_ogr_ds.CreateLayer('poly', srs=sr)

    wkt_geom = (str(polygon))
    feat = ogr.Feature( rast_mem_lyr.GetLayerDefn() )
    feat.SetGeometryDirectly( ogr.Geometry(wkt = wkt_geom) )
    rast_mem_lyr.CreateFeature( feat )

    gdal.RasterizeLayer(Output, [1], rast_mem_lyr, burn_values = [raster_value] )

def pixel2coord(px, py):
    """
    this function calculates and returns the global coordinates for a position in a numpy array by a given reference image
    """
    global xOff, xSize, b, yOff, d, ySize
    xp = xSize * px + b * py + xOff
```

```
    yp = d * px + ySize * py + yOff
    return(xp, yp)

def coords2pixel(xp, yp):
    """
    this function calculates and returns the position in a numpy array to global coordinates by a given reference image
    """
    global xOff, xSize, b, yOff, d, ySize
    px = int((xp - xOff) / xSize)
    py = int((yp - yOff )/ ySize)
    return(px, py)

def pixel2coordPath(path):
    """
    this function returns the path in global coordinates for a path of positions a two dimensional numpy array
    """
    # get columns and rows of your image from gdalinfo
    coord_list = []
    for tuple in path:
        current_coords = pixel2coord(tuple[1], tuple[0])
        coord_list.append(current_coords)

    return coord_list

def find_neighbors(cell):
    """
    this function detects the absolute positions of a cell in an array of the neighbours based on Moores neighbourhood
    """
    i = cell[0]
    j = cell[1]
    neighbor_positions=(
        (-1,0), (0,-1), (1,0), (0,1), (-1,1), (1,1), (1,-1), (-1,-1)
    )
    neighbors_list = []
    for neighbor in neighbor_positions:
        cell = i + neighbor[0], j + neighbor[1] #calculate neighbor cell position in matrix
        if np.all(0 <= cell[0] < dim_m) and np.all(0 <= cell[1] < dim_m): #and np.all(calc_raster[cell] != 0):# check if neighbor inside matrix and not a source cell
          neighbors_list.append((cell)) #appen cell position to neighbors_list
    return neighbors_list

def find_minimum_edge_value(min_value_position):
    """
    this function returns the minimum edge value of a polygon in the accumulative cost surface
    """
    active_list = [(0,(min_value_position))]
    edge_values = []

    while(len(active_list) > 0):
        min_active_list = active_list[0]
        active_list.remove(active_list[0])
        current_cell_position = (min_active_list[1][0], min_active_list[1][1])
        current_cell_value = indiv_acc_cost_raster[current_cell_position] # the vlaue of the current cell position

        neighbors = find_neighbors(current_cell_position)       #find neighbor positions of the cell

        i = 0
        #loop through all neighbors of the source cell
        for n in neighbors:
        #get the cell value of the neighbor
            neighbor_cell_value = indiv_acc_cost_raster[n[0], n[1]]
            #only if neighbor has no entry in the output raster
            if (output_raster[neighbors[i][0], neighbors[i][1]] == 0):
                #get the old value
                old_value = indiv_acc_cost_raster[neighbors[i][0], neighbors[i][1]]
                #get the relative position of the neighbor to the current cell e.g. (0,1) (-1,0), ...
                neighbor_cell_position = np.subtract(neighbors[i], current_cell_position)
                if ((neighbor_cell_value != 0) and (output_raster[neighbors[i][0], neighbors[i][1]] == 0)):
                    edge_values.append((neighbor_cell_value, (neighbors[i][0],neighbors[i][1])))
                #if the calculated value is lower than the current value in the indiv_acc_cost_raster use the new value
                elif (old_value == 0):
                    active_list.append((indiv_acc_cost_raster[neighbors[i]], (neighbors[i][0],neighbors[i][1])))
                #write to output raster
                output_raster[(neighbors[i][0],neighbors[i][1])] = 1
            i = i + 1
    if(len(edge_values) > 0 ):
        return min(edge_values)
```

```python


def calculate_bearing(latlong1,latlong2):
    """
    calculates the bearing of two coordinate pairs (tuples)
    """
    lat1 = radians(latlong1[0])
    lon1 = latlong1[1]
    lat2 = radians(latlong2[0])
    lon2 = latlong2[1]
    dLon = radians(lon2-lon1)
    y = sin(dLon) * cos(lat2)
    x = cos(lat1)*sin(lat2) - sin(lat1)*cos(lat2)*cos(dLon)
    brng = (degrees(atan2(y, x))+360)%360
    return brng

def shortest_paths(data, output_folder, weightComb):
    """
    calculates and returns the shortest paths from buildings to cells in the street raster
    """

    global xOff, xSize, b, yOff, d, ySize


    ndvi_weight, slope_weight, buildings_weight = weightComb[0], weightComb[1], weightComb[2]
    paved_weight = 1

    filename = f'{ndvi_weight}{slope_weight}{buildings_weight}'


    path = f'calculated_snapping_cell_{filename}.csv'
    file_path_points = f'{output_folder}/calculated_snapping_points/{path}'


    path_backlink = f'backlink_{filename}.tif'
    file_path_backlink = f'{output_folder}/Backlink_Raster/{path_backlink}'
    #my_file = Path(str(filepath))
    #if my_file.is_file():

    backlink_raster = gdal.Open(file_path_backlink)
    backlink_raster_array = np.array(backlink_raster.ReadAsArray())
    path_accum_cost = f'acc_cost_{filename}.tif'
    file_path_accum = f'{output_folder}/Accumulative_Cost_Surfaces/{path_accum_cost}'
    acc_cost_raster = gdal.Open(file_path_accum)
    acc_cost_raster_array = np.array(acc_cost_raster.ReadAsArray())
    #get dimensions of the grid
    dim_n, dim_m = acc_cost_raster_array.shape
    #rows, colms = len(acc_cost_raster_array), len(acc_cost_raster_array[0])
    #dim_m, dim_n = rows, colms #vertauscht? n=rows, m=colums
    output_raster = np.full([dim_m, dim_n], 0)
    xOff, xSize, b, yOff, d, ySize = backlink_raster.GetGeoTransform()
    dataframeCalculated = pd.DataFrame(columns=['id', 'building_lng', 'building_lat', 'snap_lng_calc', 'snap_lat_calc', 'calc_bearing'])

    # loop polygons here
    #1 burn raster
    #2 read as array
    #3 loop create individual accumulative raster
    #4 find least cost path and save it somewhere
    #5 recacluate to real coordinates  https://scriptndebug.wordpress.com/2014/11/24/latitudelongitude-of-each-pixel-using-python-and-gdal/

    building_polygons = data.buildings_shape #shapefile imported with gpd
    #building_polygons = building_polygons.to_crs('EPSG:25832') #not needed an raises FutureWarning because deprecated syntax

    for building_polygon in building_polygons.geometry:
        centroid = building_polygon.centroid #get centroid of current polygon

        centroid_position = coords2pixel(centroid.coords[0][0],centroid.coords[0][1])
        """
        The next comment lines are required to use the edge cell of a building polygon as Destination Cells
        """
        #burn_raster_by_polygon(reference_img, output_image, building_polygon) #create tif with current polygon
        #polygon_array = gdal.Open(output_image).ReadAsArray() # read created .tif

        #dim_n, dim_m = len(polygon_array), len(polygon_array[0]) #get dimensions of polygon array (same as accumulative cost raster)
        #indiv_acc_cost_raster = acc_cost_raster_array # create new raster based on accumulative cost raster

```

```
        #array_positions = np.array(list(zip(np.where(polygon_array == 1)[0], np.where(polygon_array == 1)[1]))) #loop all position representing current polygon
        #for row in array_positions:
        #       indiv_acc_cost_raster[row[0], row[1]] = 0 # set cells representing current building to 0

        #find least cost edge cell
        #if (len(array_positions) > 0): ## only if a building was fround
            #any_building_cell = ((array_positions[0][0], array_positions[0][1])) #any cell of the building

        #minimum_edge_value_position = find_minimum_edge_value(any_building_cell)# find the minimal accumulative cost value at cells adjacent to building cells

        #if (minimum_edge_value_position):
        #       centroid_position = minimum_edge_value_position

        #centroid_position = centroid_position[1]
        if (centroid_position[0] <= dim_m and centroid_position[1] <= dim_n):
            path = shortest_path(backlink_raster_array, (centroid_position[1], centroid_position[0]))


            #path on backlink_raster to next source cell
            path_coords_list = pixel2coordPath(path)

            #add centroid to the list
            path_coords_list.insert(0, centroid.coords[0])

            #first element (start point)
            building_centroid = path_coords_list[0]

            #last element (end point)
            calculated_snapping_point = path_coords_list[-1]

            building_centroid = utils.projCoords(origEpsg='epsg:25832', destEpsg='epsg:4326', x=building_centroid[0], y=building_centroid[1])
            snapping_point = utils.projCoords(origEpsg='epsg:25832', destEpsg='epsg:4326', x=calculated_snapping_point[0], y=calculated_snapping_point[1])
            #print(f'building_centroid: {building_centroid}, snapping_point: {snapping_point}')
            bearing = calculate_bearing(building_centroid, snapping_point)
            #relation = ("Building: ",building_centroid , "   Snapping: ", snapping_point, "Bearing: ", bearing)

            #dataframeCalculated = dataframeCalculated.append({'id': str(round(building_centroid[1], 6)) + str(round(building_centroid[0], 6)),'building_lng':building_centroid[1],'building_lat': building_centroid[0], 'snap_lng_calc': snapping_point[1], 'snap_lat_calc': snapping_point[0], 'calc_bearing': bearing}, ignore_index=True)
            # id_rounded_coord is the coord of building_centroid rounded - building_centroid[1] = lat, building_centroid[0] = lng
            dataframeCalculated = dataframeCalculated.append({'id': f'{round(building_centroid[0], 6)}_{round(building_centroid[1], 6)}','building_lng':building_centroid[1],'building_lat':building_centroid[0], 'snap_lng_calc': snapping_point[1], 'snap_lat_calc': snapping_point[0], 'calc_bearing': bearing}, ignore_index=True)


    path = f'calculated_snapping_cell_{filename}.csv'
    utils.createFolder(f'{output_folder}/calculated_snapping_points/')
    file_path = f'{output_folder}/calculated_snapping_points/{path}'

    dataframeCalculated.to_csv(file_path)
    #else:
    #    continue
```

Listing B.5: snapping.py

```
import shapely, itertools, urllib.request, json, osmnx as ox, geopandas as gpd, shapely, pandas as pd, numpy as np, csv, matplotlib.pyplot as plt, matplotlib.colors
from pyproj import Proj, transform
from pathlib import Path
from shapely.geometry import Point
from math import sin, cos, sqrt, atan2, radians, degrees
from statistics import mean
from mpl_toolkits.mplot3d import Axes3D
import utils

#GPPTOLp7


# def download_building_polygons(place):
#     """
#     downloads buildings from OSRM based on a place.
#     A shapefile containing the building polygons is returned
```

```
#       """
#       #B = ox.buildings.buildings_from_place(place, retain_invalid = False)
#       #ox.save_load.save_gdf_shapefile(B, filename='buildings', folder="I://implementation//osmnx") #save buildings
#       root_folder = Path("data/")
#       building_polygons_shp = root_folder / "input_data" / "building_shapes" / "buildings_3.shp"
#       building_poly = gpd.read_file(str(building_polygons_shp)) #read buildings
#       return building_poly

# def polygon_to_points(self, poly):
#       """
#       calculates centroids of polygons and stores the points in a shapefile.
#       The points are returned.
#       """
#       ##Shapre Area
#       shapeArea = poly['geometry'].area
#       # copy poly to new GeoDataFrame
#       points = poly.copy()
#       # change the geometry
#       points.geometry = points['geometry'].centroid
#       #ox.save_load.save_gdf_shapefile(points, filename='buildingCentroids', folder="F:/ArcGIS_CostDistance//data") # save centroids
#       # same crs
#       points.crs = poly.crs
#       return points

def calculate_bearing(latlong1,latlong2):
    """
    calculates the bearing of a line based on two coordinate pairs stored in tuples.
    the bearing is returned
    """
    lat1 = radians(latlong1[0])
    lon1 = latlong1[1]
    lat2 = radians(latlong2[0])
    lon2 = latlong2[1]
    dLon = radians(lon2-lon1)
    y = sin(dLon) * cos(lat2)
    x = cos(lat1)*sin(lat2) - sin(lat1)*cos(lat2)*cos(dLon)
    brng = (degrees(atan2(y, x))+360)%360
    return brng

def ideal_dataframe(snapping_lines):
    """
    This function generates and returns a dataframe containing all building centroids
    and the related ideal snapping points.
    """
    dataframeIdeal = pd.DataFrame(columns=['id', 'building_lng', 'building_lat', 'snap_lng_ideal', 'snap_lat_ideal', 'ideal_bearing'])
    i=0
    for row in snapping_lines.iterrows(): ##iterate geopanda dataframe
        current_line_geometry = snapping_lines['geometry'][i]

        point_A = current_line_geometry.coords[0] #Building
        point_B = current_line_geometry.coords[1] #Street

        lat_Building, lng_Building = utils.projCoords(origEpsg='epsg:25832', destEpsg='epsg:4326', x=point_A[0], y=point_A[1])
        lat_Street, lng_Street = utils.projCoords(origEpsg='epsg:25832', destEpsg='epsg:4326', x=point_B[0], y=point_B[1])

        # inProj = Proj('epsg:25832') #wgs84/utm zone 32n
        # outProj = Proj('epsg:4326') # wgs84

        # a_x, a_y = transform(inProj, outProj, point_A[0], point_A[1]) ## vertex 1
        # b_x, b_y = transform(inProj, outProj, point_B[0], point_B[1]) ## vertex 1

        bearing = calculate_bearing((lat_Building, lng_Building), (lat_Street,lng_Street))

        dataframeIdeal = dataframeIdeal.append({'id': f'{round(lat_Building,6)}_{round(lng_Building, 6)}','building_lat':lat_Building, 'building_lng':lng_Building, 'snap_lat_ideal': lat_Street, 'snap_lng_ideal': lng_Street, 'ideal_bearing': bearing}, ignore_index=True)

        i = i+1
    return dataframeIdeal

def nearest_dataframe(shape_gdf, baseURL):
    """
    calculates for every point the perpendicular distance to the closest point on the street network
    This function requires a working OSRM Server with the Nearest API
    """
```

```
    osrm_server = f'{baseURL}/nearest/v1/driving/'
    i = 0
    dataframeNearest = pd.DataFrame(columns=['id', 'snap_lng_nearest', 'snap_lat_nearest', 'nearest_bearing'])
    #calc centroids of the shapes
    shape_gdf['centroid'] = shape_gdf.centroid

    for row in shape_gdf.iterrows():
        point = shape_gdf['centroid'][i]
        #proj to epsg 4326 from
        latInput, lngInput  = utils.projCoords(origEpsg='epsg:25832', destEpsg='epsg:4326', x=point.x, y=point.y)
        latLngString =  f'{lngInput},{latInput}'
        # print(f'{osrm_server}{latLngString}.json')
        with urllib.request.urlopen(osrm_server + latLngString + ".json") as url:
            data = json.loads(url.read().decode())
            dict = data['waypoints'] #dict with coords in here
            coords = dict[0]['location'] #coords here
            lngOutput = coords[0]
            latOutput = coords[1]
            pointSnap = Point(coords[0], coords[1]) #point on the street network

            bearing = calculate_bearing((latInput, lngInput), (coords[1], coords[0]))
        dataframeNearest = dataframeNearest.append({'id': f'{round(latInput, 6)}_{round(lngInput, 6)}', 'snap_lat_nearest': coords[1], 'snap_lng_nearest': coords[0], 'nearest_bearing': bearing}, ignore_index=True)
        i += 1

    return dataframeNearest

def merge_dataframes(df1,df2):
    """
    generates and returns a merged dataframe by using a joint on the field id using two dataframes
    """
    merged_df = df1.merge(df2, left_on='id', right_on='id', how='inner') #merge dataframes
    return merged_df

def calculate_distance(row, type):
    """
    calculates the distance by the Haversine formula and returns the values in meters
    """
    lat1 = radians(row['snap_lat_ideal'])
    lon1 = radians(row['snap_lng_ideal'])
    if (type == "nearest"):
        lat2 = radians(row['snap_lat_nearest'])
        lon2 = radians(row['snap_lng_nearest'])
    elif (type == "calc"):
        lat2 = radians(row['snap_lat_calc'])
        lon2 = radians(row['snap_lng_calc'])

    R = 6373.0 #earth radius
    dlon = lon2 - lon1
    dlat = lat2 - lat1
    a = sin(dlat / 2)**2 + cos(lat1) * cos(lat2) * sin(dlon / 2)**2
    c = 2 * atan2(sqrt(a), sqrt(1 - a))
    distance = R * c * 1000 #in meters
    return distance

def evaluate(row, type, threshold_bearings = 70, threshold_distance = 25):
    """
    This function returns true if the distance and bearings is below defined thresholds, otherwise false
    """
    # threshold_distance = 25
    # threshold_bearings = 70
    if (type == "nearest"):
        distance = row['distance_ideal_nearest']
        bearing_difference = row['bearing_difference_ideal_nearest']
    elif (type == "calc"):
        distance = row['distance_ideal_calculated']
        bearing_difference = row['bearing_difference_ideal_calculated']

    if (distance < threshold_distance and bearing_difference < threshold_bearings):
        return 1 #true
    else:
        return 0 #false

def compare_bearings(row, type):
    """
    calculates and returns the difference between two bearings (angle between bearing lines)
    """
    if (type == "nearest"):
        ideal_bearing = row['ideal_bearing']
```

```
            nearest_bearing = row['nearest_bearing']
        elif (type == "calc"):
            ideal_bearing = row['ideal_bearing']
            nearest_bearing = row['calc_bearing']
        bearing_difference = (ideal_bearing - nearest_bearing) % 360
        if (bearing_difference < -180):
            bearing_difference += 360
        if (bearing_difference >= 180):
            bearing_difference -= 360
        return(abs(bearing_difference))

def evaluation(data, output_folder, thresholds, weightComb, baseURL):
        """
        This function
        """

        dataframeIdeal = ideal_dataframe(data.snapping_lines)
        dataframeNearest = nearest_dataframe(data.buildings_shape, baseURL)

        merged_df = merge_dataframes(dataframeNearest, dataframeIdeal)

        # tuple_list = []
        ndvi_list = []
        slope_list = []
        building_list = []
        validated_list = []


        validated_rate = 0

        ndvi_weight, slope_weight, buildings_weight = weightComb[0], weightComb[1], weightComb[2]
        #paved_weight = 1

        filename = f'{ndvi_weight}{slope_weight}{buildings_weight}'
        path = f'calculated_snapping_cell_{filename}.csv'
        file_path = f'{output_folder}/calculated_snapping_points/{path}'
        #my_file = Path(str(file_path))
        #print(my_file)
        #    if my_file.is_file():
        calculated_snapping_df = pd.read_csv(file_path)
        merged_all_df = merge_dataframes(merged_df, calculated_snapping_df)
        #print(merged_all_df)
        merged_all_df['distance_ideal_nearest'] = merged_all_df.apply(calculate_distance, args=("nearest",), axis=1) #
             calculate distance between nearest and ideal
        merged_all_df['bearing_difference_ideal_nearest'] = merged_all_df.apply(compare_bearings, args=("nearest",),
             axis=1) #calculate distance between nearest and ideal
        merged_all_df['evaluated_nearest'] = merged_all_df.apply(evaluate, args=("nearest",thresholds.
             bearings_evaluation, thresholds.distance_evaluation), axis=1) #evluation 1 = true , 0 = false


        merged_all_df['distance_ideal_calculated'] = merged_all_df.apply(calculate_distance, args=("calc",), axis=1) #
             calculate distance between nearest and ideal
        merged_all_df['bearing_difference_ideal_calculated'] = merged_all_df.apply(compare_bearings,args=("calc",),
             axis=1) #calculate distance between nearest and ideal
        merged_all_df['evaluated_calculated'] = merged_all_df.apply(evaluate, args=("calc",thresholds.
             bearings_evaluation, thresholds.distance_evaluation), axis=1) #evluation 1 = true , 0 = false

        # counts_calc = merged_all_df['evaluated_calculated'].value_counts()
        # counts_nearest = merged_all_df['evaluated_nearest'].value_counts()

        # tuple = (filename, counts_calc[1])
        # tuple_list.append(tuple)

        path = f'nearest_calculated_ideal_{filename}.csv'
        utils.createFolder(f'{output_folder}/Evaluation/')
        file_path = f'{output_folder}/Evaluation/{path}'
        merged_all_df.to_csv(file_path, encoding='utf-8')
```

Listing B.6: evaluation.py

```
from osgeo import gdal, ogr, osr
from pathlib import Path
from pyproj import Proj, transform
from fiona.crs import from_epsg
from rasterio import mask
import numpy as np, pandas as pd
import overpy, fiona, os
import rasterio

```

```
def clip_raster_by_bbox(file, bbox, output_name='clip.tif'):
    """[summary]

    :param file: [description]
    :type file: [type]
    :param bbox: [bbox with ulx uly lrx lry; min_x, max_y, max_x, min_y]
    :type bbox: [type]
    :param output_name: [description], defaults to 'clip.tif'
    :type output_name: str, optional
    """
    dataset = gdal.Open(file)
    dataset = gdal.Translate(output_name, dataset, projWin = bbox)

def mask_raster(file_path, shapely_bbox, output_file='clip.tif'):

    data = rasterio.open(file_path)
    out_img_array, out_transform = mask(dataset = data, shapes=shapely_bbox, crop=True)
    out_meta = data.meta.copy()

    with rasterio.open(output_file, "w") as dest:
        dest.write(out_img_array)


def projCoords(origEpsg, destEpsg, x, y):
    """
    Transform coordinates by using the epsg-code.
    Common epsg-codes:
        WGS84 lat lon (decimal, unit:degree): 4326
        WGS 84/ UTM 32 N (unit: meters):      32632
        WGS 84/ UTM 33 N (unit: meters):      32633
    Parameters:

    epsg-codes: str
    example: 'epsg:4326'

    returns lat, lng for epsg:4326
    """
    inProj = Proj(origEpsg)
    outProj = Proj(destEpsg)
    x2,y2 = transform(inProj,outProj,x,y)
    return x2,y2

def getExtentfromShape(shapefile):
    """
    Get the extent of a shapefile. Returns the coordinates in
    west, east, sout, north
    min_x, max_x, min_y, max_y

    Parameters
    ----------
    shapefiles : str
        Path to the shapefile.
    """
    file = ogr.Open(shapefile)
    layer = file.GetLayer()
    extent = layer.GetExtent()
    return extent

def getNoDataValue(rasterfn):
    raster = gdal.Open(rasterfn)
    band = raster.GetRasterBand(1)
    return band.GetNoDataValue()

def array2raster(newRasterfn,rasterOrigin,pixelWidth,pixelHeight,array):

    cols = array.shape[1]
    rows = array.shape[0]
    originX = rasterOrigin[0]
    originY = rasterOrigin[1]

    driver = gdal.GetDriverByName('GTiff')
    outRaster = driver.Create(newRasterfn, cols, rows, 1, gdal.GDT_Int16 ) #GDT_Byte ersetzt damit values ueber
        255 zulaessig sind
    outRaster.SetGeoTransform((originX, pixelWidth, 0, originY, 0, pixelHeight))
    outband = outRaster.GetRasterBand(1)
    outband.WriteArray(array)
    outRasterSRS = osr.SpatialReference()
    outRasterSRS.ImportFromEPSG(32633)
    outRaster.SetProjection(outRasterSRS.ExportToWkt())
```

```python
    outband.FlushCache()

def coord2pixelOffset(rasterfn,x,y):
    raster = gdal.Open(rasterfn)
    geotransform = raster.GetGeoTransform()
    originX = geotransform[0]
    originY = geotransform[3]
    pixelWidth = geotransform[1]
    pixelHeight = geotransform[5]
    xOffset = int((x - originX)/pixelWidth)
    yOffset = int((y - originY)/pixelHeight)
    return xOffset,yOffset

def rasterFromSHP(shapefile, outputfile, pxSize):
    driver = ogr.GetDriverByName('ESRI Shapefile')
    dataFile = driver.Open(shapefile, 0) #0 read only 1 write
    layer = dataFile.GetLayer()
    spatialRef = layer.GetSpatialRef()
    feature = layer.GetNextFeature()
    geom = feature.GetGeometryRef()
    spatialRef = geom.GetSpatialReference()

    #create raster
    NoDataValue = -9999
    xmin, xmax, ymin, ymax = layer.GetExtent()

    # Create the destination data source
    x_res = int((xmax - xmin) / pxSize)
    y_res = int((ymax - ymin) / pxSize)
    target_ds = gdal.GetDriverByName('GTiff').Create(outputfile, x_res, y_res, 1, gdal.GDT_Byte)
    target_ds.SetGeoTransform((xmin, pxSize, 0, ymax, 0, -pxSize))
    band = target_ds.GetRasterBand(1)
    band.SetNoDataValue(NoDataValue)

    # Rasterize
    gdal.RasterizeLayer(target_ds, [1], layer, burn_values=[0])

    raster = gdal.Open(outputfile, gdal.GA_Update)
    return raster

def read_shape_to_array(input_shape, reference_img, output_image, rasterValue):
    """
    this functions burns shapefiles into a numpy array with the rasterValue for objects in the
    shapefile and returns this grid
    """
    #initialise paramaters

    gdalformat = 'GTiff'
    datatype = gdal.GDT_Byte
#    raster_value = rasterValue
    # Get properties of reference_img
    img = gdal.Open(reference_img, gdal.GA_ReadOnly)
    # load the shp
    shp = ogr.Open(input_shape)
    shp_layer = shp.GetLayer()
    # rasterise the shp using raster_value
    output = gdal.GetDriverByName(gdalformat).Create(output_image, img.RasterXSize, img.
    RasterYSize, 1, datatype, options=['COMPRESS=DEFLATE'])
    output.SetProjection(img.GetProjectionRef())
    output.SetGeoTransform(img.GetGeoTransform())
    # store raster in band 1 of the tiff
    gdal.RasterizeLayer(output, [1], shp_layer, burn_values=[rasterValue])
    #reset parameters
    output = None
    img = None
    shp = None
    shape_array = gdal.Open(output_image).ReadAsArray()
    return(shape_array)

def raster2array(rasterfn):
    raster = gdal.Open(rasterfn)
    band = raster.GetRasterBand(1)
    array = band.ReadAsArray()
    return array

def coords2pixel(xUTM, yUTM, raster):
    """
    Converts UTM Coords into pixelCoords.

    Parameters
```

```
    ----------
    raster : osgeo.gdal.Dataset
        gdal.Open(pathToRaster) if needed.
    """
    #raster = gdal.Open(raster)
    geotransform = raster.GetGeoTransform()
    xOrig = geotransform[0]
    yOrig = geotransform[3]
    xSize = geotransform[1]
    ySize = geotransform[5]
    px = int((xUTM - xOrig) / xSize)
    py = int((yUTM - yOrig )/ ySize)
    return (px, py)

def createFolder(folder_path):
    """Checks if the given path is a folder, if it does not exists, create the folder.

    :param path: [description]
    :type path: str
    """
    if not os.path.exists(folder_path):
        os.mkdir(folder_path)

def generate_validation_rates(output_folder):
    """Generates a csv file based of all csv-files from the evaluation (nearest_calculaed_ideal).

    :param path: [description]
    :type path: str
    """
    weighting_list = []
    validation_calc = []
    validation_nearest = []

    path = Path(f'{output_folder}/Evaluation/')
    files = [f for f in path.iterdir() if f.match("nearest_calculated_ideal_*.csv*")]

    for file in files:
        weighting = str(file)[-7:-4]

        df = pd.read_csv(file)
        counts_calc = df['evaluated_calculated'].value_counts()
        counts_nearest = df['evaluated_nearest'].value_counts()

        #calculates the validation rate true out of all entries
        vali_calc = counts_calc[1]/len(df)
        vali_nearest = counts_nearest[1]/len(df)

        weighting_list.append(weighting)
        validation_calc.append(vali_calc)
        validation_nearest.append(vali_nearest)

    df_result = pd.DataFrame()
    df_result['weighting'] = weighting_list
    df_result['vali_calc'] = validation_calc
    df_result['vali_nearest'] = validation_nearest

    df_result.to_csv(f'{output_folder}/Evaluation/validation_rates.csv')

def already_processed(file_path):
    """Checks if a file exists, returns true or false

    :param file_path: [description]
    :type file_path: [str]
    :return: [bool]
    :rtype: [type]
    """

    my_file = Path(file_path)
    return my_file.is_file()

def generate_lcp(filepath_backlink_raster, destination_latlon):

    backlink_raster = gdal.Open(filepath_backlink_raster)

    #create numpy array from raster
    backlink_array = np.array(backlink_raster.ReadAsArray())

    #calculate cellposition in raster from latlon
    utm_coords = projCoords('epsg:4326', 'epsg:25832', destination_latlon[0], destination_latlon[1])

```

```
    #round to one digit since 0.2m for each pixel are relevant
    x_utm, y_utm = [round(item,1) for item in utm_coords]
    cellposition_x, cellposition_y = coord2pixelOffset(filepath_backlink_raster, x_utm, y_utm)

    current_position = (cellposition_x, cellposition_y)


    path_list = []
    path_list.append(current_position)
    while (1): #loop until current cell is source cell

        #get the cell value for the current cell, starting with destination. Depending
        #on the cell value (0= source, ... 8=diagonal, lower right cell), add the cell_position
        #of the path to the source cell to the path_list
        cell_value = backlink_array[current_position[1], current_position[0]]


        if(cell_value == 0):
            return path_list
        elif(cell_value == 1):
            current_position = tuple(map(lambda i,j: i+j, current_position, (-1,0))) #annahme x,y, bei 1 liegt die
                source celle links daneben
            path_list.append(current_position)
        elif(cell_value == 2):
            current_position = tuple(map(lambda i,j: i+j, current_position, (-1,-1)))
            path_list.append(current_position)
        elif(cell_value == 3):
            current_position = tuple(map(lambda i,j: i+j, current_position, (0,-1)))
            path_list.append(current_position)
        elif(cell_value == 4):
            current_position = tuple(map(lambda i,j: i+j, current_position, (1,-1)))
            path_list.append(current_position)
        elif(cell_value == 5):
            current_position = tuple(map(lambda i,j: i+j, current_position, (1,0)))
            path_list.append(current_position)
        elif(cell_value == 6):
            current_position = tuple(map(lambda i,j: i+j, current_position, (1,1)))
            path_list.append(current_position)
        elif(cell_value == 7):
            current_position = tuple(map(lambda i,j: i+j, current_position, (0,1)))
            path_list.append(current_position)
        elif(cell_value == 8):
            current_position = tuple(map(lambda i,j: i+j, current_position, (-1,1)))
            path_list.append(current_position)

def pixel2coord(filepath_raster, px, py):
# ==============================================================================
#     calculates the UTM coord for a pixel coord. For rasters the coordinate of
#     a pixel is on the upper left corner. To get the center of a pixel, half
#     a pixelSize is added for each direction (x,y)
# ==============================================================================
    raster = gdal.Open(filepath_raster)
    geotransform = raster.GetGeoTransform()
    xOrig = geotransform[0]
    yOrig = geotransform[3]
    xSize = geotransform[1]
    ySize = geotransform[5]
    x_utm = px * xSize + xOrig + (xSize/2)
    y_utm = py * ySize + yOrig + (ySize/2)
    return (x_utm, y_utm)

def createGeojson(nested_coords):
    import json
    template = {
                  "type": "FeatureCollection",
                  "features": [
                    {
                      "type": "Feature",
                      "properties": {},
                      "geometry": {
                        "type": "LineString",
                        "coordinates": nested_coords

                      }
                    }
                  ]
                }
    res = json.dumps(template)
    return res
```

```
def lonlat_lcp(filepath_raster, lcp):
    latlon_coords = []
    for item in lcp:
        x_utm, y_utm = pixel2coord(filepath_raster, item[0], item[1])
        lat, lon = projCoords('epsg:25832', 'epsg:4326', x_utm, y_utm)
        latlon_coords.append([lon, lat])
    return latlon_coords

def lcp_geojson(filepath_backlink, latlon, output_dir=None):
    lcp = generate_lcp(filepath_backlink, latlon)
    lcp_lonlat = lonlat_lcp(filepath_backlink, lcp)
    geojson = createGeojson(lcp_lonlat)
    if output_dir == None:
        return geojson
    else:
        with open(f'{output_dir}', 'w') as file:
            file.write(geojson)

def histogramm_validation_rate(fp_validation_rates, col, bins, xlabel, ylabel, fontsize, figSize=(10,10)):
    #histogramm_validation_rate(fp_validation_rate, 'vali_calc', 6, 'validation-rate', 'frequency', 20)
    import matplotlib.pyplot as plt
    df = pd.read_csv(fp_validation_rates)
    df = df[col]
    fig, ax = plt.subplots(figsize=figSize)
    df.plot.hist(grid=True, bins=bins, rwidth=0.5, color='#607c8e', ax=ax)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.rcParams.update({'font.size': fontsize})
    plt.show()
    return fig,ax

def mixed_histogramm_validation_rate(fp_validation_rates, col_cd, col_pd, bins, xlabel, ylabel, fontsize, figSize=(10,10)):
    #histogramm_validation_rate(fp_validation_rate, 'vali_calc', 6, 'validation-rate', 'frequency', 20)
    import matplotlib.pyplot as plt
    df = pd.read_csv(fp_validation_rates)
    df_cd = df[col_cd]
    df_pd = df[col_pd]
    fig, ax = plt.subplots(figsize=figSize)
    plt.style.use('seaborn-deep')
    plt.hist([df_pd, df_cd], bins, label= ['perpendicular distance', 'Cost-Distance'], rwidth=None, edgecolor="k")
    plt.legend(loc='upper right')
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.rcParams.update({'font.size': fontsize})
    plt.rcParams["patch.force_edgecolor"] = True
    plt.show()
    return fig,ax

def plot_validation_weights(fp_validation_rates, fontsize=12):
    from mpl_toolkits.mplot3d import Axes3D
    import matplotlib
    import matplotlib.pyplot as plt

    df = pd.read_csv(fp_validation_rates)

    vegetation_list = [] #ndvi
    slope_list = [] #slope
    buildings_list = [] #buildings

    values_list = []

    for idx, row in df.iterrows():
        vegetation_list.append(int(str(row['weighting'])[0]))
        slope_list.append(int(str(row['weighting'])[1]))
        buildings_list.append(int(str(row['weighting'])[2]))

        values_list.append(row['vali_calc'])

    #offset abziehen
    offset = min(values_list)

    values_offset = [value - offset for value in values_list]

    #alternative normalize data
    norm = matplotlib.colors.Normalize(vmin=min(values_offset), vmax=max(values_offset))
    colormap = plt.get_cmap("winter")

    fig = plt.figure()
    ax3D = fig.add_subplot(111, projection='3d')
```

```
    x = vegetation_list
    y = slope_list
    z = buildings_list

    p = ax3D.scatter(x, y, z, s=30, c=colormap(norm(values_offset)), marker='o')
    # p = ax3D.scatter(x, y, z, s=10, c=colormap(values_offset), marker='o')


    ax3D.set_xlabel('vegetation', labelpad=12)
    ax3D.set_ylabel('slope', labelpad=12)
    ax3D.set_zlabel('building footprints', labelpad=12)

    #set tick labels to every nth (2) tick
    for label in ax3D.axes.xaxis.get_ticklabels()[::2]:
        label.set_visible(False)
    for label in ax3D.axes.yaxis.get_ticklabels()[::2]:
        label.set_visible(False)
    for label in ax3D.axes.zaxis.get_ticklabels()[::2]:
        label.set_visible(False)

    cbar = plt.colorbar(p)
    cbar.set_label(''r'$\Delta$ validation-rate')
    plt.rcParams.update({'font.size': fontsize})
    plt.show()
```

Listing B.7: utils.py

```
from pathlib import Path
import utils, acsa, snapping, evaluation
import rasterio, gdal, itertools
from shapely import geometry
import geopandas as gpd
from datetime import datetime

class dataSet:
    vegetation = None
    slope = None
    buildings_raster = None
    buildings_shape = None
    road_network = None
    snapping_lines = None
    building_centroids = None

class thresholds:
    vegetation = None
    slope = None
    bearings_evaluation = None
    distance_evaluation = None


##set paths
#input
root_folder = Path('../data/input_data/tile_32526_5736_15_06_2017')
output_folder = Path('../data/output_data')
# tmp_folder = Path('../data/tmp')


##Open data
data = dataSet()

# data.road_network = gdal.Open(f'{tmp_folder}/tmp_roads.tif')
data.road_network = gdal.Open(f'{root_folder}/road_network/roads_3m_steps.tif')

# data.vegetation = gdal.Open(f'{tmp_folder}/tmp_vegetation.tif')
data.vegetation = gdal.Open(f'{root_folder}/vegetation/NDVI_res_20cm.tif')

# data.slope = gdal.Open(f'{tmp_folder}/tmp_slope.tif')
data.slope = gdal.Open(f'{root_folder}/LiDar_DEM/slope_fill_no_data.tif')

# data.buildings_raster = gdal.Open(f'{tmp_folder}/tmp_buildings.tif')
data.buildings_raster = gdal.Open(f'{root_folder}/buildings_raster/buildings_raster.tif')

# data.buildings_shape = gpd.read_file(f'{tmp_folder}/tmp_buildings_shape.shp')
data.buildings_shape = gpd.read_file(f'{root_folder}/buildings_shape/buildings_no_garage_parking.shp')

# data.snapping_lines = gpd.read_file(f'{tmp_folder}/tmp_snapping_lines.shp')
data.snapping_lines = gpd.read_file(f'{root_folder}/snapping_lines_full/ideal_lines.shp')

```

```
#set thresholds
thresholds = thresholds()
thresholds.vegetation =  0.2
thresholds.slope = 11
thresholds.bearings_evaluation = 70
thresholds.distance_evaluation = 25

### set weight combinations
weight_combinations_list = [range(1,11,2),range(1,11,2),range(1,11,2)]
weight_combinations_list = list(itertools.product(*weight_combinations_list))
# weight_combinations_list = [(4,6,7), (5,6,7), (6,7,8)] #(ndvi, slope, building)

#timer for start of weight combinations
start_time = datetime.now()

#iterate of weight_combs
for idx, weightComb in enumerate(weight_combinations_list):

    #set timer for every iteration
    weight_iteration_time = datetime.now()

    #create cost surfaces, accumulative cost surfaces and backlink raster for each weighting
    print(f'...processing cost surfaces for iteration {idx+1} of {len(weight_combinations_list)}')
    if utils.already_processed(f'{output_folder}/Accumulative_Cost_Surfaces/acc_cost_{weightComb[0]}{weightComb[1]}{weightComb[2]}.tif'):
        pass
    else:
        acsa.create_cost_surfaces(data, thresholds, output_folder, weightComb)

    #create the snapping points
    print('...processing snapping points')
    if utils.already_processed(f'{output_folder}/calculated_snapping_points/calculated_snapping_cell_{weightComb[0]}{weightComb[1]}{weightComb[2]}.csv'):
        pass
    else:
        snapping.shortest_paths(data, output_folder, weightComb)

    #evaluating the snapping points
    print('...evaluating snapping points')
    if utils.already_processed(f'{output_folder}/Evaluation/nearest_calculated_ideal_{weightComb[0]}{weightComb[1]}{weightComb[2]}.csv'):
        pass
    else:
        evaluation.evaluation(data, output_folder, thresholds, weightComb, 'http://localhost:5000')

    print(f'Time since start: {datetime.now() - start_time}.\nTime since iteration start: {datetime.now() - weight_iteration_time}')

#generate csv file with validation rates
utils.generate_validation_rates(output_folder)
print(f'Final processing time: {datetime.now() - start_time}')
```

Listing B.8: main.py