Master's Thesis

Pathfinding in Dynamic Road Networks

by
Stefan Schefbäck
stefan.schefbaeck@student.tugraz.at
Registration Number: 9612363

at
Knowledge Technologies Institute
Graz University of Technology

Advisor: Assoc.Prof. Dipl.-Ing. Dr.techn. Denis Helic

Graz, January 2017

Copyright © 2017 Stefan Schefbäck

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, ______________________ ______________________

Date Signature

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am ______________________ ______________________

Datum Unterschrift

Zusammenfassung

Die Suche nach kürzesten Wegen im Straßennetz ist ein Forschungsfeld, das in den letzten Jahren eine bedeutende Entwicklung erfahren hat. Der wachsende Bedarf, nicht nur für die individuelle Routenplanung, sondern auch für die Verkehrsplanung, Simulationen und die bevorstehenden autonomen Fahrzeuge erfordern Algorithmen, die dynamische Netzwerke kontinentaler Größe mit häufigen Aktualisierungen verarbeiten können.

Für diese Arbeit wurden verschiedene Algorithmen, von den einfachsten bis zu einigen der neuesten und schnellsten, aus verschiedenen Kategorien ausgewählt und jeder von ihnen wurde im Detail in der Praxis analysiert. Ihr Verhalten auf vollständigen und kontrahierten Straßennetzen mit unterschiedlichen Metriken wurde untersucht und miteinander verglichen. Es wurde gezeigt, wie die Vorverarbeitungsphasen der anspruchsvolleren Algorithmen in der Praxis funktionieren und welchen Einfluss unterschiedliche Netzwerke auf sie haben. Eine große Sammlung von Straßennetzen aus der ganzen Welt mit unterschiedlichen Größen, von Städten bis zu Kontinenten, wurde verwendet, um mögliche Unterschiede aufzuzeigen und um zu analysieren wie die verschiedenen Algorithmen mit ihnen zurecht kommen. Schließlich wurde untersucht, welche Algorithmen für große und dynamische Netzwerke geeignet sind und ob es mögliche Nischenbereiche gibt.

Es wurde festgestellt, dass selbst das Verhalten der einfachsten Algorithmen durch die Verwendung unterschiedlicher Metriken beeinflusst wird. Im Gegensatz dazu betraf eine Änderung in der Topologie nur die komplexeren Algorithmen. Die Ergebnisse für A* zeigten den (in diesem Fall negativen) Einfluss der Hardware und der Implementierung auf die Ausführungsgeschwindigkeit. Es wurde entdeckt, dass die schnellen Abfragezeiten von Contraction Hierarchies auf den small-world Effekt zurückzuführen sind und dass Contraction Hierarchies und Customizable Route Planning gleichermaßen durch unterschiedliche Topologien beeinflusst werden, obwohl sie auf unterschiedlichen Konzepten basieren. Die Ergebnisse über die große Anzahl an Straßennetzen zeigte, dass es nur geringe Unterschiede gibt und nur wenige Netzwerke, wie zum Beispiel Buenos Aires, herausstechen. Der abschließende Leistungsvergleich zeigte, dass nur der Customizable Route Planning Algorithmus schnell genug für kontinentale Netze mit häufigen Aktualisierungen ist. Der „A*, Landmarks, Triangle Inequality“ Algorithmus in Kombination mit einem schnellen Vorverarbeitungsverfahren war schnell genug um als eine Alternative für kleinere und mittlere Netzwerke betrachtet werden zu können.

Abstract

Finding shortest paths in road networks is a research field that has seen significant developments in the recent years. The growing needs not only for individual route planning but also for transportation planning, simulations and the upcoming autonomous vehicles require algorithms that can handle dynamic networks of continental size with frequent updates.

In this work, several algorithms ranging from the simplest ones to some of the newest and fastest from different categories were chosen. Each of them was analyzed in detail in a practical setup. Their behavior on full and contracted road networks with different metrics was investigated and compared with each other. It was shown how the preprocessing phases of the more sophisticated algorithms work in practice and how they are affected by different networks. A large collection of road networks from all over the world with different sizes, ranging from cities to continents, was used to find possible distinctions between them and to analyze how the different algorithms handle them. Finally it was examined which algorithms are suitable for large and dynamic networks and if there are possible niche areas.

It was found that even the behavior of the simplest algorithms is influenced by using different metrics. On the contrary, a change in the topology only affected the more complex algorithms. The results for A* demonstrated the (in this case negative) influence of the hardware and implementation on the performance. It was discovered that the fast query times of Contraction Hierarchies are a result of the small-world effect and that Contraction Hierarchies and Customizable Route Planning are equally effected by different topologies, even though they are based on different concepts. The evaluation of the large selection of road networks showed that the differences between them are small and only a few networks such as Buenos Aires are standing out. The final performance comparison showed that only the Customizable Route Planning algorithm is fast enough for continental sized networks with frequent updates. The "A*, Landmarks, Triangle Inequality" algorithm used with a fast preprocessing method was fast enough to be considered as an alternative for small and medium sized networks.

Acknowledgements

I want to thank my friend Kiro, who had to listen to all my complaining during the writing of this work, for his motivational support. I am also very grateful for all the freedom I have been granted by my supervisor Denis Helic. Finally, I would like to thank my parents for their support throughout my life. I dedicate this work to my pet Frogger, who has accompanied me during my studies.

Contents

1 Introduction **1**
1.1 Motivation . . . 2
1.2 Outline . . . 2

2 Theoretical Overview **3**
2.1 Preliminaries . . . 3
2.1.1 Graphs . . . 3
2.1.2 Shortest Path Finding . . . 4
2.1.3 Synonyms . . . 4
2.2 Introduction . . . 5
2.3 Selected Algorithms . . . 5
2.4 Dijkstra's Algorithm . . . 6
2.4.1 Algorithm . . . 6
2.4.2 Runtime . . . 7
2.4.3 Performance Improvements . . . 7
2.5 Bidirectional Dijkstra's Algorithm . . . 8
2.5.1 Algorithm . . . 8
2.6 A* . . . 9
2.6.1 Heuristic . . . 9
2.6.2 Algorithm . . . 10
2.6.3 Runtime . . . 11
2.7 Bidirectional A* . . . 11
2.8 A*, Landmarks, Triangle Inequality (ALT) . . . 12
2.8.1 Triangle Inequality . . . 12
2.8.2 Landmarks . . . 14
2.8.3 Algorithm . . . 15
2.9 Contraction Hierarchies (CH) . . . 15
2.9.1 Hierarchies . . . 16
2.9.2 Shortcuts . . . 16
2.9.3 Node Ordering . . . 17
2.9.4 Preprocessing Algorithm . . . 17

2.9.5 Preprocessing Improvements . . . 18
2.9.6 Query Algorithm . . . 18
2.9.7 Query Improvements . . . 19
2.9.8 Path Reconstruction . . . 19
2.10 Customizable Route Planning (CRP) . . . 19
2.10.1 Graph Partitioning . . . 20
2.10.2 Overlay Graph . . . 21
2.10.3 Preprocessing . . . 22
2.10.4 Query Algorithm . . . 23
2.10.5 Path Reconstruction . . . 23
2.11 Partitioning Using Natural Cut Heuristics (PUNCH) . . . 24
2.11.1 Overview . . . 24
2.11.2 Preliminaries . . . 24
2.11.3 Filtering . . . 25
2.11.4 Assembly . . . 26

3 Framework and Dataset **29**
3.1 Framework . . . 29
3.1.1 Programming Language and Libraries . . . 29
3.1.2 Graph Data Structure . . . 29
3.1.3 Priority Queue . . . 31
3.1.4 Customizable Route Planning (CRP) and Natural Cuts . . . 32
3.1.5 Contraction Hierarchies (CH) . . . 34
3.1.6 A*, Landmarks, Triangle Inequality (ALT) . . . 34
3.1.7 Other Components . . . 34
3.1.8 Final Remarks . . . 35
3.2 Dataset . . . 36
3.2.1 Metrics . . . 37
3.2.2 Full and Contracted Networks . . . 38
3.2.3 Copyright and License . . . 40

4 Experimental Analysis **41**
4.1 Preliminaries . . . 41
4.2 Experimental Setup . . . 41
4.3 Dijkstra's Algorithm . . . 42
4.3.1 Synthetic Grids . . . 42
4.3.2 Road Networks . . . 43
4.3.3 Results & Conclusion . . . 47
4.4 Bidirectional Dijkstra's Algorithm . . . 48
4.4.1 Synthetic Grids . . . 48
4.4.2 Road Networks . . . 49

4.4.3 Results & Conclusion . . . 52
4.5 A*, Bidirectional A* . . . 53
4.5.1 Synthetic Grids . . . 53
4.5.2 Road Networks . . . 53
4.5.3 Results & Conclusion . . . 56
4.6 A*, Landmarks, Triangle Inequality (ALT), Bidirectional ALT . . . 57
4.6.1 Synthetic Grids . . . 57
4.6.2 Landmark Selection . . . 61
4.6.3 Search on Road Networks . . . 64
4.6.4 Results & Conclusion . . . 67
4.7 Contraction Hierarchies (CH) . . . 69
4.7.1 Synthetic Grids . . . 69
4.7.2 Preprocessing on Road Networks . . . 70
4.7.3 Search on Road Networks . . . 77
4.7.4 Results & Conclusion . . . 78
4.8 Customizable Route Planning (CRP) . . . 81
4.8.1 Synthetic Grids . . . 81
4.8.2 Partitioning on Road Networks . . . 83
4.8.3 Search on Road Networks . . . 93
4.8.4 Results & Conclusion . . . 93

5 Performance Comparison 97
5.1 Experimental Setup . . . 97
5.2 Query Times . . . 99
5.2.1 Geometric Distances . . . 99
5.2.2 Travel Times . . . 101
5.2.3 Conclusion . . . 102
5.3 Node Efficiency . . . 104
5.3.1 Full Networks . . . 104
5.3.2 Contracted Networks . . . 105
5.3.3 Conclusion . . . 106
5.4 Edge Efficiency . . . 108
5.4.1 Conclusion . . . 108
5.5 Preprocessing . . . 110
5.5.1 Optimizations . . . 113
5.5.2 Conclusion . . . 113

6 Conclusion 115
6.1 Future Work . . . 116

Bibliography 117

A Additional Key Figures **123**
A.1 Dataset . . . 123

B Additional Results **129**
B.1 ALT, Bidirectional ALT . . . 129
B.2 Edge Efficiency with Travel Times . . . 130

C Framework **132**
C.1 Compiling . . . 132
C.2 Graph Files . . . 132
C.3 User Manual . . . 133

List of Figures

2.1 Bidirectional search example . . . 9
2.2 Triangle inequality . . . 13
2.3 Deviation of the estimated lower bound . . . 13
2.4 Stall-on-demand example . . . 19
2.5 Graph partitioning . . . 20
2.6 Overlay graph . . . 21
2.7 Finding a natural cut . . . 25
2.8 Creating a fragment graph . . . 26
2.9 Temporary graph of local search . . . 27

3.1 Illustration of the graph data structure . . . 30
3.2 Natural cuts work flow overview . . . 32
3.3 Distribution of edges for different speed limits . . . 38
3.4 Unrestricted path contraction . . . 38
3.5 Restricted path contraction . . . 39
3.6 Number of nodes and edges in full and contracted networks. . . . 39

4.1 Dijkstra's search on synthetic grid graphs . . . 42
4.2 Dijkstra's search on Greater London Area . . . 43
4.3 Dijkstra's search on Greater Chicago Area . . . 46
4.4 Bidirectional Dijkstra's search on synthetic grid graphs . . . 48
4.5 Bidirectional Dijkstra's search on Greater London Area . . . 50
4.6 Bidirectional Dijkstra's search on Greater Chicago Area . . . 51
4.7 Unidirectional and Bidirectional A* search on synthetic grid graphs . . . 54
4.8 Unidirectional and Bidirectional A* search on the Greater London and Chicago Area with geometric distances. . . . 55
4.9 ALT and Bidirectional ALT search on uniform synthetic grid graphs . . . 57
4.10 ALT and Bidirectional ALT search on non-uniform synthetic grid graphs . 59
4.11 ALT and Bidirectional ALT search lined up with landmarks . . . 60
4.12 Bidirectional ALT search on a uniform synthetic grid with floating point and integer weights. . . . 60
4.13 Partitioning a graph by coordinates . . . 62

4.14 ALT and Bidirectional ALT search on Greater London Area with geometric distances and travel times. 65
4.15 ALT and Bidirectional ALT search on Greater Chicago Area with geometric distances and travel times. 66
4.16 CH search on synthetic grid graphs 69
4.17 Contracting corner nodes 70
4.18 Edge and degree distribution for the road network of Graz 71
4.19 CH network layout for various hierarchy level ranges for the road network of Graz 73
4.20 Number of shortcuts created by CH for all networks 74
4.21 Road network section of Atlanta and Buenos Aires 75
4.22 Time required for preprocessing with CH for all networks 75
4.23 Ratios between full and contracted versions of a network for CH 76
4.24 Time required for preprocessing per normal edge with CH for all networks 76
4.25 CH search on Graz with travel times 77
4.26 CH search on Greater London and Chicago Area with travel times 79
4.27 CRP search on synthetic grid graphs 82
4.28 CRP partitions of a grid graph. 82
4.29 CRP partition of North America with rivers 84
4.30 CRP partition of continental Europe with rivers 85
4.31 CRP partition of Greater London Area with rivers and railways 85
4.32 CRP partition of Greater Chicago Area with rivers and railways 86
4.33 CRP partition of London City with rivers and railways 87
4.34 CRP partition of Chicago City with rivers and railways 88
4.35 Percentage of boundary nodes at the given level 89
4.36 Number of boundary nodes and edges per cell over all networks 90
4.37 Time required for preprocessing with CRP for all networks 90
4.38 Partition comparison of Greater London Area's full and contracted network 91
4.39 Time required for preprocessing with CRP for all contracted networks with different cell size 92
4.40 CRP search on Greater London and Chicago Area with travel times . . . 94

5.1 Average query times for all full networks with geometric distances 99
5.2 Bidirectional A* search on Seattle 100
5.3 Average query times for all contracted networks with geometric distances 100
5.4 Average query times for all full networks with travel times 101
5.5 Average query times for all contracted networks with travel times 102
5.6 Road network of California and a section from Buenos Aires 103
5.7 Average node efficiency for all full networks with geometric distances and travel times 104

5.8 Road network section of Dallas and Tokyo 105
5.9 Average node efficiency for all contracted networks with geometric distances and travel times 106
5.10 Average node efficiency comparison for all contracted networks between geometric distances and travel times 107
5.11 Average edge efficiency for all networks with geometric distances 109
5.12 Sequential preprocessing time for all networks 111
5.13 Comparison of CRP customization time and number of boundary nodes . 112

B.1 Average edge efficiency for all full networks with travel times 130
B.2 Average edge efficiency for all contracted networks with travel times . . . 131

List of Tables

3.1 Settings used for PUNCH . . . 33
3.2 Categories and number of networks . . . 36
3.3 Mean number of nodes per category . . . 36
3.4 Default speed limits . . . 37

4.1 Average performance of Dijkstra's algorithm with geometric distances and travel times on the Greater London Area . . . 44
4.2 Average distribution of edges from shortest paths among different road types for different metrics on the Greater London Area . . . 44
4.3 Average performance of Dijkstra's algorithm with geometric distances and travel times on the Greater Chicago Area . . . 45
4.4 Average distribution of edges from shortest paths among different road types for different metrics on the Greater Chicago Area . . . 47
4.5 Average performance of Dijkstra's algorithm with geometric distances and travel times . . . 47
4.6 Average performance of a bidirectional Dijkstra's algorithm with geometric distances and travel times . . . 52
4.7 Average performance of bidirectional Dijkstra's algorithm with geometric distances and travel times . . . 56
4.8 Average performance of ALT and Bidirectional ALT . . . 58
4.9 Average efficiency comparison of the modified Partition-Corners algorithm with randomly placed landmarks (Bidirectional ALT) . . . 63
4.10 Average efficiency comparison of the modified with the original Partition-Corners algorithm (Bidirectional ALT) . . . 63
4.11 Average performance of ALT and Bidirectional ALT with different metrics 67
4.12 Average node efficiency of ALT and Bidirectional ALT on full and contracted networks . . . 68
4.13 Degree distribution for the road network of Graz . . . 71
4.14 Efficiency comparison between all algorithms . . . 78
4.15 Average performance of CH with geometric distances and travel times . . 80
4.16 Average performance of CH on full and contracted networks . . . 80
4.17 Cell sizes . . . 89

4.18 Comparison of the median number of boundary nodes per cell per level with different cell sizes . . . 92
4.19 Comparison of the average query times on contracted networks with different cell sizes . . . 93
4.20 Average performance of CRP with geometric distances and travel times . 95
4.21 Average performance of CRP on full and contracted networks . . . 95

5.1 Full network number to name LUT . . . 98
5.2 Contracted network number to name LUT . . . 98

A.1 List of road networks and their key figures of North American cities and metropolitan areas . . . 123
A.2 List of road networks and their key figures of North American cities and metropolitan areas. (continuation) . . . 124
A.3 List of road networks and their key figures of European cities and metropolitan areas . . . 125
A.4 List of road networks and their key figures of Other World cities and metropolitan areas . . . 126
A.5 List of road networks and their key figures of North American states and regions . . . 127
A.6 List of road networks and their key figures of European countries and regions . . . 127
A.7 List of road networks and their key figures of Other World countries and regions . . . 128
A.8 List of road networks and their key figures of continental size . . . 128

B.1 Efficiency comparison of the modified Partition-Corners algorithm with randomly placed landmarks (Unidirectional ALT) . . . 129
B.2 Efficiency comparison of the modified with the original Partition-Corners algorithm (Unidirectional ALT) . . . 130

Chapter 1

Introduction

Finding the shortest or quickest path between two places is a common task many people are confronted with every day. Just two decades ago this problem had to be solved by hand, based on personal or other's experience and the help of outdated road atlases. And while it was possible to find the shortest path this way, finding the quickest was a game of chance due to the lack of knowledge of things such as possible congestions, road works or accidents. Advances in computers, the opening of the Global Positioning System (GPS) to the public and the endless effort of various institutions (e.g. the United States Census Bureau[1] or OpenStreetMap[2]) to create, maintain and provide digital road maps improved this dramatically. Nowadays, shortest paths can be queried in an instant on personal computers, automotive navigation systems and smart phones. But the efforts in this field are just being started. Galileo[3], the European equivalent to GPS, went operational in December 2016 [7] for even more accurate positioning data and roads are increasingly equipped with sensors for real time traffic information. Combined with the growing networking it allows for better routing of traffic not only on an individual basis but also on a grand scale. Also, the research in autonomous vehicles has made huge advances in the recent years with first prototypes being on the road [43]. These vehicles will increase the need for fast, accurate and detailed path finding even more.

However, path finding is not only a matter of real traffic on real road networks but also in the virtual world. With the ever increasing number of vehicles and growing cities, transportation planning is getting more complex, requiring virtual models that can simulate traffic consisting of a large number of vehicles [56]. Another virtual application are computer games, which are getting more sophisticated every year, simulating growing amounts of entities finding their way on virtual maps.

[1] https://www.census.gov/geo/maps-data/data/tiger-line.html
[2] http://www.openstreetmap.org
[3] http://www.esa.int/Our_Activities/Navigation/Galileo/What_is_Galileo

1.1 Motivation

Due to the growing needs and available data, path finding has been rediscovered as important research field in the recent years with significant developments [3]. Algorithms are now able to answer shortest path queries in less than a millisecond on continental sized networks consisting of tens of millions of nodes. However, there are still many problems that have only been solved partially or not at all, such as continental sized journey planning with public transport or the multi-modal route planning problem, which considers different modes of transportation at once.

Published papers of recently developed algorithms and techniques have in common that they include detailed theoretical/mathematical descriptions and practical evaluations on a few large road networks that are widely used as benchmark instances. But a practical analysis on how the algorithms interact with road networks and behavioral differences compared to other methods are often treated as matters of secondary importance. Furthermore, the given performance comparisons tend to focus on the largest networks, paying less attention to other networks and their possible differences and impacts.

To investigate these less researched aspects, several techniques from different categories are selected for this work, ranging from Dijkstra's algorithm to some of the newest and fastest, such as Contraction Hierarchies and Customizable Route Planning. The research questions are focused on the practical behavior and differences of the algorithms on real road networks of various size and origin. Instead of a pure theoretical comparison, the behavioral patterns are highlighted in a practical setup and inspected for possible differences between them. The impacts of road networks of different size and origin with different metrics are examined and the question for possible distinctions in networks from all over the world is investigated. Furthermore, the query and preprocessing times are compared to answer the question which of the algorithms are suitable for dynamic networks with frequent updates and if there are niche areas for the slower ones.

1.2 Outline

In Chapter 2, the theory behind the algorithms used in this work is explained. The framework and dataset used for the experimental analysis and performance comparison are presented in Chapter 3. A detailed analysis of the behavior and practical performance of each algorithm is given in Chapter 4. In Chapter 5, the query times of all algorithms are compared against each other and the preprocessing times are analyzed. The appendix contains additional key figures for the road networks in the dataset and information for compiling and using the framework.

Chapter 2

Theoretical Overview

This chapter gives an theoretical overview of the algorithms and concepts used in this work starting with the basic building blocks followed by the high sophisticated algorithms build upon them.

2.1 Preliminaries

Some basic concepts and notations that are used throughout this work are presented in the following.

2.1.1 Graphs

A graph $G = (V, E)$ consists of a set V of vertices and a set E of edges. Every edge is connected to two vertices and denoted as $(u, v) \in E$. Edges that start and end at the same vertex are called loops. In this work only edges between distinct vertices are considered. A graph can be directed or undirected. In a directed graph, every edge has a direction, pointing from one vertex to another. An edge pointing from a vertex u to a vertex v is called *outgoing* at u and *incoming* at v. Edges running in opposite directions between the same pair of vertices can be combined into a bidirectional edge, i.e. one edge that points in both directions. In a weighted graph, a weight $w(u, v)$ is associated with every edge. In this work only positive weights are considered.

The number of edges connected to a vertex is the so called *degree* of it. There is no limit to the degree of a vertex. If no edge is connected to it, the degree is zero. Vertices that are directly connected with each other are called neighbors. Traversing an edge to reach another vertex is also called a *hop*.

A path is a sequence of connected vertices. It requires at least a start vertex s and a target vertex t. A path where s and t are equal is called a cycle. The length of a path equals the distance $d(s, t)$ which is the sum of the weights of its edges. The shortest path is the one with the smallest length from all paths between s and t. In an undirected graph, if a path from s to t exists, the reverse of it constitutes the path from t to s,

because the edges are undirected. In a directed graph, a path from s to t only has to be traversable in one direction along its directed edges and the same may not be true for the reverse from t to s.

In an undirected graph, a connected component is a subgraph where every pair of vertices is connected with a path. In a directed graph, this is called a strongly connected component. The difference is that in an undirected graph a path can be traversed in both directions, which is not true with directed graphs (see above). This means that not every vertex that is connected by a path is also part of the strongly connected component.

2.1.2 Shortest Path Finding

Several abbreviations and phrases have been established in connection with path finding which are described in the following:

- SPSP: single pair shortest path, one-to-one; finding the shortest path between a pair of nodes
- APSP: all pair shortest path, many-to-many; finding the shortest path between all pairs of nodes
- SSSP: single source shortest path, one-to-all, all-to-one; finding the shortest path from one node to all other nodes or vice versa
- *Node Efficiency*: Is calculated as the percentage of the number of nodes that have been settled during a search in relation to the number of nodes in the shortest path. A node can only be settled once.
- *Edge Efficiency*: Is calculated as the percentage of the number of edges that have been scanned during a search in relation to the number of edges in the shortest path. The same edge can be scanned several times.

2.1.3 Synonyms

Several different terms are used synonymously in this work:

- A network is the same as a graph.
- A vertex is also called a node.
- The weight of an edge is also called cost.
- The sum of the weights of all edges from a path are equal to the paths length which is the same as the distance between its start and end node.
- The geometric distance is the same as the spherical distance, spatial distance or great circle distance.

- A motorway is the same as a freeway and a Autobahn, describing an arterial road for high volume traffic.

2.2 Introduction

The problem of finding shortest paths between vertices in graphs has been solved in the middle of the twentieth century with the Bellman-Ford algorithm [4], Dijkstra's algorithm [13] and the Floyd-Warshall algorithm [19, 33, 69]. Since then, many more algorithms and speedup techniques built upon them have been proposed to improve their running times on all or certain types of graphs.

With the increased interest in path finding on road networks in recent years, many new methods specialized on this type of graph have been proposed. A recent and thorough overview of many algorithms is given by Bast et al. in [3] who also divide them into the following categories:

- *Goal directed techniques* try to direct the search towards the goal e.g. by applying heuristics or precomputing paths.
- *Separator based techniques* divide the graph along vertices or edges to create a smaller overlay graph which is then used for queries.
- *Hierarchical techniques* are focused on possible hierarchical structures in road networks such as arterial roads and try to reduce the number of visited vertices by exploiting them.
- *Bounded hop techniques* reduce the size of the graph by precomputing paths between pairs of vertices.

Many methods consist of a preprocessing phase that modifies the graph. Because road networks have a dynamic nature with varying traffic, accidents, road works etc. which can be unpredictable, preprocessed data will become inaccurate over time. This means that the data has to be repaired or the search algorithm has to be adapted which in every case costs time or reduces the performance of queries (if optimality is not dropped). This can impact algorithms with a time-consuming preprocessing phase severely, rendering them useless for highly dynamic networks.

2.3 Selected Algorithms

For this work, several algorithms from different categories are selected to provide a broad comparison. Dijkstra's algorithm [13] is chosen because it is used as a building block in many other techniques and it is the performance benchmark every other algorithm has to compete with.

From the goal directed category, A* [31] and "A*, Landmarks, Triangle Inequality" (ALT) [27] are selected. A* is one of the first algorithms that tried to improve the running time of Dijkstra's algorithm by applying a heuristic. Because of its age (it was proposed in 1968) it is well known and many variations exist. ALT can be seen as the continuation of A*, which tries to improve its shortcomings by providing better bounds and not requiring a heuristic, which makes it suitable for any metric. Its disadvantage is the required preprocessing phase.

From the category of hierarchical techniques, Contraction Hierarchies [23] is taken, which is the continuation of Highway Hierarchies [54] and Highway Node Routing [58]. It is currently used by the Open Source Routing Machine[1] (OSRM) [38]. By contracting nodes according to its importance in the networks hierarchy and adding shortcuts to the graph, its query times are among the fastest.

Customizable Route Planning (CRP) [9] is chosen from the separator based category because it is one of the most sophisticated and fastest techniques developed recently, used by the the search engine Bing from Microsoft [44]. Its main advantage is that its preprocessing phase is split into a metric independent and a customization phase, which allows it to incorporate weight changes quickly.

2.4 Dijkstra's Algorithm

One of the first algorithms to solve the path finding problem is Dijkstra's algorithm [13]. It finds the path with the lowest distance from a source node to all other nodes (single source shortest path problem) and can also be used to find the shortest path to only one node by terminating the algorithm after the target has been found (single pair shortest path problem).

2.4.1 Algorithm

The underlying idea of the algorithm, summarized from [13] in this section, is to always follow the path with the shortest tentative distance available. When a node is reached (settled), the used path must be optimal, because every other path has a longer distance [2, p. 208]. If more than one path with the same distance exists, it depends on the implementation which one is chosen.

For a given graph $G = (V, E)$ with a set V of vertices (nodes) and a set E of edges with nonnegative weights, the algorithm works as follows:

1. Create two empty sets named *closed set* and *open set*.

2. Select a source node $s \in V$, mark it as active node and add it to the open set.

[1] http://project-osrm.org/

3. Assign to every node $n \in V$ the tentative distance $\delta(s, n)$, which is the shortest distance between it and the source node s found so far. For the source node s this distance is 0, every other node's distance is initialized with ∞.

4. Add all nodes v not in a set yet that are direct neighbors of the active node u to the open set.

5. *Relaxation*: For all nodes v in the open set that are direct neighbors of the active node u, calculate the tentative distance $\delta(s, v) = \delta(s, u) + min(w(u, v))$, which is the current tentative distance δ assigned to u, plus the weight w of the edge from u to v with the smallest weight. Compare the result with the distance δ' currently assigned to v. If the new distance δ is smaller, assign it to v and set the active node u as predecessor node at v.

6. Move the active node u from the open set to the closed set. The node u is now *settled* and the final distance from the source node s to u is known.

7. *Expansion*: Search the node in the open set that has the shortest tentative distance assigned and make it the new active node. If this node is the target node t, the shortest path between s and t has been found and the algorithm can be stopped. If the open set is empty, all shortest paths from s have been found and the algorithm can also be stopped. Otherwise go back to step 4.

After the algorithm has finished, the assigned tentative distance δ of a node t is equal to the shortest distance from the source node s to t. If the distance is ∞, then no path exists between s and t. With the predecessor nodes that have been assigned at step 5, the shortest path leading from s to t can be reconstructed.

2.4.2 Runtime

The algorithm loops over $n = |V|$ nodes and at each iteration the node with the shortest tentative distance has to be found and removed from the open set. Also, new neighboring nodes get added to the open set which can happen at most n times and the tentative distance of neighboring nodes may be updated, which can happen at most $m = |E|$ times. This means an implementation with a simple data structure such as an unsorted array as open set has $n * n$ find and remove minimum (*removeMin*) operations, n *inserts* of new neighbors and at most m distance updates (also called *decreaseKey* in use with priority queues), which means the runtime is $T(n) = n * n + n + m = O(n^2 + m)$.

2.4.3 Performance Improvements

The theoretical runtime can be improved by using a more sophisticated data structure for the open set. A priority queue implemented with a binary heap [71], which takes $\log n$ time for each of the 3 operations *removeMin*, *insert* and *decreaseKey*, has a runtime

of $T(n) = n * (\log n + \log n) + m * \log n = O((n + m) \log n)$. This can be further improved by using a Fibonacci heap [20], which takes $\log n$ time for *removeMin* and only $O(1)$ amortized time for *insert* and *decreaseKey*, yielding a runtime of $T(n) = n * (\log n + O(1)) + m * O(1) = O(n \log n + m)$.

Chen et al. thoroughly compare the practical performance of Dijkstra's algorithm with 10 different priority queue implementations in [6]. The fastest implementation on every tested graph class uses a *sequence heap* [53], which is optimized for the cache/memory hierarchy of modern computers.

2.5 Bidirectional Dijkstra's Algorithm

A bidirectional search, first suggested by Dantzig [8], solves the single pair shortest path problem by simultaneously expanding from the source and target node. Used in combination with Dijkstra's algorithm, a forward search from the source node and a backward search from the target node is executed until the wave fronts of both searches meet. Correct termination procedures for this strategy were discussed in [16, 25, 46, 50]. Compared to the unidirectional approach, faster runtimes can be achieved because fewer nodes are visited [51].

2.5.1 Algorithm

The bidirectional Dijkstra's algorithm works under the same principle as the unidirectional version, i.e. to always follow the path with the shortest tentative distance (see Section 2.4 on page 6). The only differences are the handling of the graph in the backward search, the alternation between the forward and backward search and the termination procedure.

For undirected graphs, the backward search works equal to the forward search. For directed graphs, one has to consider that the path has to lead from the source node to the target node, not vice versa. This means that at the relaxation step, only neighboring nodes with *outgoing edges* to the active node are considered. Also, the reconstructed path has to be reversed.

A naive method of alternation would be switching between forward and backward search every other iteration, but better performance can be achieved with a cardinality comparison between the open sets of both searches [50]. The cardinality of the open set reflects the local density of the network surrounding the settled nodes. By executing the search with the smaller open set, less work has to be done because of the sparser surrounding network and the search fronts may meet earlier.

One possible termination procedure, as described in [50, p. 13], works as follows. The search terminates and a shortest path is found if the same node $v \in V$ is settled in the forward and backward search. Otherwise, if all nodes have been settled but none in both searches, no path exists between the source and target node. The shortest path

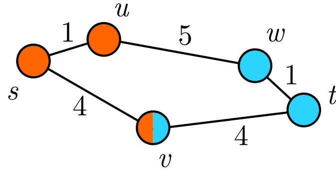


Figure 2.1: Bidirectional search example. The nodes $\{s, u, v\}$ are settled in the forward search and $\{t, w, v\}$ in the backward search. The node v responsible for termination is not on the shortest path $p = \langle s, u, w, t \rangle$ [16, p. 7].

does not necessarily contain the node v, but a settled node u_i in the forward search and a settled node w_i in the backward search, with an edge from u_i to w_i, as illustrated in Figure 2.1.

Let S_f contain all settled nodes in the forward search and S_b all settled nodes in the backward search. The shortest path is then defined as in Equation 2.1, where δ denotes the tentative distance and w the edge weight.

$$distance(s,t) = min(\delta(s,u_i) + w(u_i,w_j) + \delta(w_j,t)) \text{ with } u_i \in S_f, w_j \in S_b. \quad (2.1)$$

No shorter path including a node $i \notin \{S_f \cup S_b\}$ not settled yet exists, because by the definition of Dijkstra's algorithm, the distance to i must be greater than the distance to v in both searches, which means $\delta(s,i) + \delta(i,t) \geq \delta(s,v) + \delta(v,t)$.

Goldberg [25] proposed a stronger termination procedure. Let v_f and v_b be the node with the shortest tentative distance in the open set of the forward and backward search, respectively. By maintaining the length μ of the shortest path discovered so far (corresponding to Equation 2.1), the search can be terminated if $\delta(s, v_f) + \delta(v_b, t) \geq \mu$. No shorter path can exist for the same reason as given above.

2.6 A*

The A* algorithm [31] (also called *"A star"*), first described by Peter Hart, Nils Nilsson and Bertram Raphael, is from the category of *goal directed* techniques and extends Dijkstra's algorithm with a heuristic to find the shortest path between a start and a target node (single pair shortest path problem). It is summarized from [31] in the following section.

2.6.1 Heuristic

The algorithm works after the same principle as Dijkstra's algorithm, which is to always follow the path with the shortest distance. But instead of only considering the tentative distance $g(x)$ from the source node s to the current intermediate node x, it uses the total distance $f(x) = g(x) + h(x)$, which also includes the remaining distance $h(x)$ to the target node t. This remaining distance $h(x)$ is estimated with a heuristic function.

Depending on the properties of the heuristic, different behaviors can be achieved:

- If $h(x)$ always returns 0 (equal to not using any heuristic), A* behaves like Dijkstra's algorithm.
- If $h(x)$ returns an *admissible* heuristic, A* can have a lower runtime than Dijkstra's algorithm. An admissible heuristic means that the distance to the target node is never overestimated, which is required for the result to be optimal.
- If $h(x)$ additionally satisfies the condition $h(x) \leq distance(x, y) + h(y)$ for every node in the graph, it is also *monotone* (or *consistent*) and can further improve the runtime, because nodes only have to be visited once. (A monotone admissible heuristic is also called *dual feasible* [32].)
- If optimality can be neglected, the runtime can be additionally lowered in certain cases by multiplying $h(x)$ with an $\epsilon > 1$.

The runtime depends on the quality of the heuristic. If the estimates do not correlate with the actual distances, the benefits over Dijkstra's algorithm diminish. Possible heuristics for a road network are, for example, the Euclidean distance or the Manhattan distance[2].

2.6.2 Algorithm

In the following description, the open set utilizes a priority queue, with the total distance f as priority. The used heuristic must be monotone and admissible. For a given graph $G = (V, E)$ with a set V of vertices (nodes) and a set E of edges with nonnegative weights, the algorithm works as follows:

1. Create two empty sets named *closed set* and *open set*.
2. Select a source node $s \in V$, mark it as active node and add it to the open set.
3. Assign to every node $n \in V$ the tentative distance $\delta(s, n)$, which is the shortest distance between it and the source node s found so far. For the source node s this distance is 0, every other node's distance is initialized with ∞.
4. *Relaxation*: For all nodes v not in the closed set yet that are direct neighbors of the active node u:
    - Calculate the tentative distance $g(v) = g(u) + min(w(u, v))$, which is the current tentative distance g assigned to u, plus the weight w of the edge from u to v with the smallest weight.

[2] Also called City-block metric, as defined in [12, p. 204].

- Compare the result with the distance g' currently assigned to v. If the new distance g is greater, skip this node and continue with the next neighboring node. Otherwise, assign it to v and set the active node u as predecessor node at v.
- Calculate the total distance $f(v) = g(v) + \epsilon * h(v)$, where $g(v)$ is the tentative distance just calculated, $h(v)$ is the heuristic which estimates the distance from the current neighboring node v to the target node t and ϵ is a factor which can improve the runtime if $\epsilon > 1$ at the cost of optimality.
- Replace the stored value of $f(v)$ with the new one.
- If the neighboring node v is not in the open set yet, insert it. Otherwise, update the position of v in the open set according to the new value of $f(v)$.

5. Move the active node u from the open set to the closed set. The node u is now *settled* and the final distance from the source node s to u is known.

6. *Expansion*: Get the node in the open set that has the shortest total distance f assigned and make it the new active node. If this node is the target node t, the shortest path between s and t has been found and the algorithm can be stopped. If the open set is empty, no path between s and t exists and the algorithm can also be stopped. Otherwise go back to step 4.

After the target node t has been found and the algorithm has finished, the assigned tentative distance $g(t)$ is the shortest distance between the source node s and t, if an admissible heuristic and $\epsilon = 1$ was used. Otherwise, $g(t)$ contains the length of the chosen path, which may not be the shortest available. With the predecessor nodes assigned in step 4, the chosen path leading from s to t can be reconstructed.

2.6.3 Runtime

The runtime of the A* algorithm depends on the quality of the heuristic, the value of ϵ, the runtime of the data structures used for the implementation and the data set. This means that the runtime can vary from polynomial to exponential. A detailed discussion can be found in [49].

2.7 Bidirectional A*

A bidirectional A* algorithm is based on the same ideas and assumptions as given for the bidirectional Dijkstra's algorithm in Section 2.5 on page 8. By executing the search in both directions from the start and target node, fewer nodes will be visited in most cases [51]. The difficult part of the bidirectional version is the termination. Even if a monotone and admissible heuristic is used and the forward and backward search areas

overlap, it is not guaranteed that the shortest path has been found. Different solutions for this problem have been found ([32, 51]), with the one from Ikeda et al. [32] working as follows:

To make the heuristics consistent between forward and backward search, they are combined, as shown in Equation 2.5. π_f is the distance estimate function in the forward and π_b in the backward search.

$$\begin{aligned} h_f(v) &= \tfrac{1}{2}(\pi_f(v) - \pi_b(v)) \\ h_b(v) &= \tfrac{1}{2}(\pi_b(v) - \pi_f(v)) = -h_f(v) \end{aligned} \tag{2.2}$$

Goldberg et al. proposed the following modification in [30] to make the new heuristic more intuitive:

$$\begin{aligned} h_f(v) &= \tfrac{1}{2}(\pi_f(v) - \pi_b(v)) + \tfrac{\pi_b(t)}{2} && t \text{ ... target vertex} \\ h_b(v) &= \tfrac{1}{2}(\pi_b(v) - \pi_f(v)) + \tfrac{\pi_f(s)}{2} && s \text{ ... source vertex} \end{aligned} \tag{2.3}$$

Using the new heuristic, a possible termination criterion for the bidirectional A* algorithm, also proposed by Goldberg et al. in [30], is: Let v_f and v_b be the node with the shortest distance $\delta(v) = g(v) + h(v)$ in the open set of the forward and backward search, respectively. By maintaining the length μ of the shortest path discovered so far (analogously to Section 2.5 on page 8), the search can be terminated if $\delta_f(v_f) + \delta_b(v_b) \geq \mu + h_b(t)$.

2.8 A*, Landmarks, Triangle Inequality (ALT)

The ALT (A*, Landmarks, Triangle Inequality) algorithm is an extended version of the A* algorithm, proposed by Andrew Goldberg and Chris Harrelson [27]. By utilizing a more sophisticated calculation method for the heuristic, based on the triangle inequality theorem and so-called *landmarks*, it achieves better runtimes for queries on road networks than a typical search using Euclidean distances. The drawback of this technique is that it requires a preprocessing phase. A description of the algorithm, summarized from [26], is given below.

2.8.1 Triangle Inequality

The triangle inequality states that the sum of the length of any two sides of a triangle is always greater or equal to the length of the remaining one. For vectors in the Euclidean space this relation is defined as shown in Equation 2.4 [1, p. 11].

$$|v_1| - |v_2| \leq |v_1 + v_2| \leq |v_1| + |v_2| \tag{2.4}$$

Another consequence of the triangle inequality is that the difference of any two sides is always smaller or equal to the remaining one. This means an upper and a lower bound

for the length of a side of a triangle can always be calculated from the other two sides.

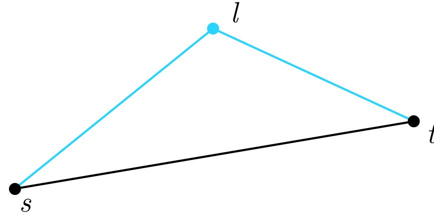


Figure 2.2: Triangle inequality in path finding example. The nodes s and t are the start and end of a path. If the distances from and to a third node l are known, the length of the path can be estimated.

Finding the shortest path with the A* algorithm requires an estimate of the remaining distance to the target (see Section 2.6 on page 9). Consider the simple example shown in Figure 2.2. The distance from node s to t can be estimated with the triangle inequality, if the distances from and to the node l are known (Equation 2.5).

$$\begin{aligned} d(s,l) \leq d(s,t) + d(t,l) \Leftrightarrow d(s,t) \geq d(s,l) - d(t,l) \\ d(l,t) \leq d(l,s) + d(s,t) \Leftrightarrow d(s,t) \geq d(l,t) - d(l,s) \end{aligned} \tag{2.5}$$

These estimates are admissible and monotone, which means if they are used as heuristic in the A* algorithm, the result will be optimal. Since only one value is required, the larger one should be taken, because it gives a tighter lower bound.

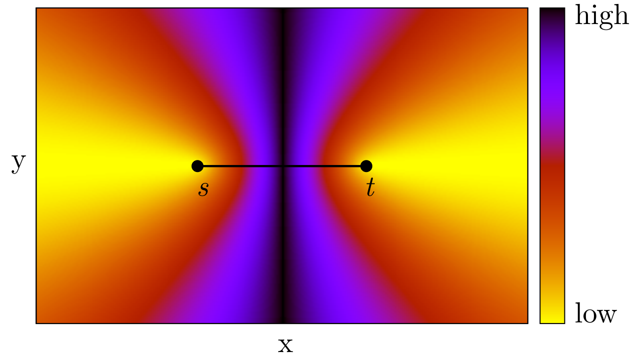


Figure 2.3: Deviation of the estimated lower bound from the real distance $d(s,t)$ depending on the relative position of l.

The quality of the bounds depends strongly on the position of the node l relative to s and t. As can be seen in Figure 2.3, the best results are achieved if l is located in front of s or behind t.

The triangle inequality also works for nodes not directly connected with each other, as long as the used distances from and to l are from the shortest paths.

2.8.2 Landmarks

Landmarks are a set of nodes from the network which are used for the estimation of the lower bounds with the triangle inequality. They are selected in a preprocessing phase, because the shortest paths between every node and landmark have to be calculated. As described above, the positions of the landmarks are responsible for the quality of the lower bounds and the resulting performance of search queries. Several different landmark selection methods have been proposed and compared in [17, 21, 27, 29], but no optimal method exists because of the inherent trade off between the time required for selection and the resulting query performance. A few of them are described in the following.

- *Random* [27]: The fastest and simplest method is the selection of nodes at random. While this method can give good results, its overall performance is poor compared to more sophisticated ones.

- *Partition-Corners* [17]: The network is divided into k cells and the four cornermost nodes of each cell are selected as landmarks. This method is fast and the authors claim that its performance can compete with advanced techniques.

- *Farthest* [27]: Start with a set containing a random node. Search the node that is the farthest away from the nodes in the set and add it to the set. Repeat this until the desired amount of nodes has been found. While this should give landmarks with a better distribution than the random method, it has been shown in [29] that its performance can be worse at certain networks, because it can favour nodes from sparse over dense regions.

- *Planar* [27]: Start with a node c from the center of the network. Divide the network into pie-slices containing about the same amount of nodes around c. Now find the node that is the farthest from c in every slice. If such a node is close to the border of two slices, ignore the neighboring nodes in the other slice to avoid having landmarks to close to each other.

  This method requires more time than *Farthest* but also has a consistently better performance.

- *Avoid* [29]: Consider an existing set S of landmarks. Start with a random node r. Calculate the shortest path tree T_r to all other nodes v. Now calculate the weight for every node v, which is defined as the difference between the distance $d(r, v)$ and the estimated lower bound for the same node pair and the given set S of landmarks. In the next step, calculate the size of every node v which is the sum of the weights of all nodes in T_v, the subtree from T_r rooted at v. If T_v contains a landmark from S, set the size to 0. In the final step, take the node with the largest size and traverse its subtree along the children with the largest size until a leaf node is reached. Add the leaf node to the set of landmarks.

The *Avoid* method improves the quality of an existing set of landmarks by adding additional ones from regions that are not well covered. Results can be improved by starting with nodes that are far from existing landmarks, but this adds to the processing time. Its performance is considerably better than the other methods mentioned with slightly more computing time required. The *maxCover* method [29] is an extended version of *Avoid*.

The number of landmarks influences the preprocessing time, query performance and storage requirements. According to Goldberg and Harrelson [27, 29], just one landmark can already deliver better performance than other methods. Performance gains start to diminish approximately between 10 and 20 landmarks, depending on the queried nodes, network topology and quality of the landmarks.

2.8.3 Algorithm

The ALT algorithm works exactly like the A* algorithm described in Section 2.6, using the triangle inequality with landmarks as heuristic. The landmarks and their distances from and to every node in the graph have to be calculated in a preprocessing phase. At the relaxation step during a query, the maximum of all lower bounds over all landmarks is used as heuristic. Not the whole set of landmarks is required for a query. Instead, a subset of the landmarks with the highest lower bounds for the path between the start and target node can be used. This subset can be updated with better landmarks during execution.

The algorithm can also be executed bidirectional by using the bidirectional version of A* (Section 2.7 on page 11). If the subset of landmarks is changed during a bidirectional search, the distances in the priority queue have to be recalculated. Otherwise the termination may not be correct.

2.9 Contraction Hierarchies (CH)

The Contraction Hierarchies algorithm, proposed by Geisberger et al. [23], is the continuation of Highway Hierarchies [54] and Highway Node Routing [58] to solve the single pair shortest path problem. As its name suggests, it is from the category of *hierarchical* techniques. The basic idea behind Contraction Hierarchies is, to reduce the runtime of queries by reducing the number of nodes that have to be visited to find the shortest path. This is achieved by the application of two concepts, Hierarchies and Shortcuts, during a preprocessing phase. A bidirectional version of Dijkstra's algorithm is used for shortest path queries. The following description of the algorithm and concepts is summarized from [22].

2.9.1 Hierarchies

The concept of Hierarchies is based on the assumption that not every edge in a graph has the same importance at certain stages during the pathfinding process. Applied to road networks this means that not every road is seen as equally important, as there are many different types of roads, e.g. residential area roads, primary roads and arterial roads (freeways[3]). When asking for the shortest route from one town to another, one may consider residential roads close to the start and target node, but will opt for freeways to cover the path between towns, assuming they are the shortest or fastest path. This approach separates the network in different layers or hierarchy levels. The lowest hierarchy level may consider only residential area roads and the highest only freeways. The search for the shortest path then starts at a low hierarchy level working upwards. Because only nodes in the current or higher hierarchy levels are considered, the search space is reduced.

The Contraction Hierarchies algorithm is an extreme case of the Hierarchies concept, because there are not just a few hierarchy levels, but every node has its own.

2.9.2 Shortcuts

In the concept of Shortcuts, the graph is enriched with additional edges to reduce the number of nodes that have to be visited during a search. Consider three nodes, u, v, w with two edges (u, v) and (v, w). By adding a new edge (u, w), a shortcut from node u to w is introduced. The edge weight c of the new edge is set to the sum of the other two edges: $c(u, w) = c(u, v) + c(v, w)$. The drawback of this concept is, that the graph gets denser with every additional edge, which increases the runtime of search queries again. This means that only good shortcuts (i.e. those that improve query runtimes the most) and as few as possible should be added, which can be achieved by processing the nodes in a beneficial order.

In Contraction Hierarchies, the process of adding shortcuts is called *node contraction* and works as follows. For a node v, consider all paths $p_{ij} = \langle u_i, v, w_j \rangle$ from a neighboring node u_i through v to a neighboring node w_j. For every path p_{ij} that is the shortest path in the graph from u_i to w_j, a shortcut (new edge) between u_i and w_j is created. The weight of the new edge is set to the total weight of the path p_{ij}. If there is already an edge between u_i and w_j, it is sufficient to update the edge weight only. After all paths p_{ij} have been processed, the node v and its incoming and outgoing edges are temporarily removed from the graph, before the next node is processed. The node contraction is repeated for all nodes in the graph in ascending order of their hierarchy level. The final graph used for queries consists of the initial graph and all created shortcuts.

[3]motorway, Autobahn

2.9.3 Node Ordering

The order in which nodes are contracted influences the size and quality of the resulting graph. By observing certain properties (so-called *priority terms*), the nodes can be arranged in a beneficial order for processing. The priority terms, order and hierarchy level of nodes is determined during preprocessing.

Initially, a minimum priority queue holds all nodes, with them being removed one by one for node contraction. The priority of a node in the queue is a linear combination of one or more terms (e.g. edge difference, number of neighbors contracted, Voronoi regions, cost of contraction)[4]. The main term is the *edge difference*: the number of shortcuts created at the node's hypothetical contraction minus the number of edges incident to this node in the current graph. The edge difference ensures that nodes with many shortest paths going through them are removed later, giving them a higher hierarchy level.

The priority is calculated once for all nodes at the initialization of the minimum priority queue. But after a node contraction the priority terms of the nodes remaining in the priority queue can change and have to be recalculated accordingly.

2.9.4 Preprocessing Algorithm

During preprocessing, the hierarchy levels of nodes are set and shortcuts are added to the graph, both of which will be used to improve the runtime of shortest path queries. For a given graph $G = (V, E)$ with a set V of vertices (nodes) and a set E of edges with nonnegative weights, preprocessing works as follows:

1. For all nodes $n \in V$, calculate the initial priority.
2. Create a minimum priority queue named Q and fill it with all nodes $n \in V$.
3. Remove the top node v from Q, set its hierarchy level to the number of nodes removed so far and mark it as active node.
4. *Node contraction*: For the active node v, consider all neighboring nodes $u_i \in Q$ with outgoing edges to v and all neighboring nodes $w_j \in Q$ with incoming edges from v:
   - *Local search for witness paths*: For every pair of nodes (u_i, w_j) with $u_i \neq w_j$, initiate a search for the shortest path p'_{ij} from u_i to w_j with all intermediate nodes from Q only.
   - If no path p'_{ij} is found or if it is longer than the path $p_{ij} = \langle u_i, v, w_j \rangle$, create a shortcut by adding a new edge from u_i to w_j with the edge weight set to the total weight of p_{ij}.
5. Update the priority of all nodes in Q and go back to step 3, until Q is empty.

[4]See [22, Chapter 3.2] for more information on priority terms.

2.9.5 Preprocessing Improvements

Searching for shortest paths, which is part of step 1, 4 and 5 of the preprocessing algorithm, can be very time consuming. Instead of processing every pair of nodes (u_i, w_j) separately, a Dijkstra's search can be started from u_i until all w_j nodes are settled or the tentative distance of the last node settled is larger than the longest path from $max_j(\langle u_i, v, w_j\rangle)$.

In step 5, the recomputation of the priorities, the following optimizations can be applied, with a periodic full update after every t contractions:

- neighbors only: Recompute the priorities for neighbors of the contracted node only.
- Lazy updates: After a node contraction, only recompute the priority for the top node of the queue. Repeat this until the top node stays the same.

The computation of the edge difference also requires a local search for shortest paths, but in contrast to step 4, no shortcuts are added to the graph. This means that search termination criterions can be less exact without adding a performance penalty during queries. Two possible criterions are:

- limit number of settled nodes
- limit number of maximum edges per shortest path (*hop limit*)

2.9.6 Query Algorithm

Queries are based on a bidirectional Dijkstra's algorithm (see Section 2.5 on page 8) which is executed on two versions of the resulting graph of the preprocessing phase, including hierarchy levels and shortcuts. For the forward search the *upward graph* $G_\uparrow$ is used and for the backward search the *downward graph* $G_\downarrow$, as defined in Equation 2.6.

$$\begin{aligned} G_\uparrow &= (V, E_\uparrow) \text{ with } E_\uparrow = (u, v) \in E | u < v) \\ G_\downarrow &= (V, E_\downarrow) \text{ with } E_\downarrow = (u, v) \in E | u > v) \end{aligned} \tag{2.6}$$

This means that in the relaxation step of the forward and backward search, only edges leading to or coming from neighboring nodes with a higher hierarchy level than the active node are processed, respectively.

Because of the edge restrictions in the upward and downward graph, the search can not terminate as soon as the first node has been settled in both searches. Instead, by maintaining the length of the shortest path discovered so far, the search in one direction can be stopped at the earliest if the shortest tentative distance in the open set is equal or larger. Let S_f contain all settled nodes in the forward search and S_b all settled nodes in the backward search. The shortest path is then defined as in Equation 2.7.

$$distance(s, t) = min(\delta(s, v) + \delta(v, t)) \text{ with } v \in \{S_f \cap S_b\}. \tag{2.7}$$

2.9.7 Query Improvements

The search space of a query can be reduced by a search pruning technique called *stall-on-demand*. Because not every edge is taken into consideration during relaxation, nodes may be settled with a non-optimal tentative distance. If the active node v about to be settled can be reached with a shorter tentative distance δ' through an incoming edge from a neighboring node u that was not considered because of the upward/downward graph restrictions, the active node gets *stalled* with *stalling distance* δ'. Stalled nodes are excluded from relaxation, because their tentative distance is not optimal. Stalling can be propagated (e.g. with a breadth-first search) to subsequent nodes w_i if the path over u and v to w_i is shorter than the current tentative distance of w_i. The propagation stops at nodes that are not being stalled. A stalled node gets unstalled if a shorter path is found later on.

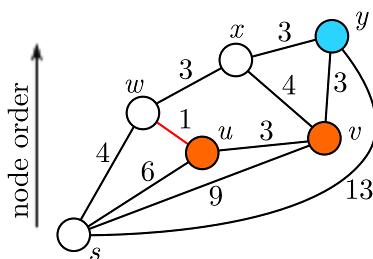


Figure 2.4: Stall-on-demand example. Numbers denote edge weights.

The example illustrated in Figure 2.4 works as follows. When node u is about to be settled, it gets stalled because its tentative distance δ is 6, but it could be reached over w with a distance δ' of 5. By propagation, node v with $\delta(v) = 9$ and node y with $\delta(y) = 13$ get stalled too, because they could be reached over w and u with $\delta'(v) = 8$ and $\delta'(y) = 11$. Node x is not stalled, because its tentative distance $\delta(x) = 7$, which is smaller than $\delta'(x) = 12$. Node y is unstalled when node x gets settled, because the new value for $\delta(y)$ is 10, which is smaller than the previous value of 13.

2.9.8 Path Reconstruction

The path returned from a successful query may contain shortcuts. By storing the node v that initiated the creation of the shortcut (u, w) during node contraction in the data structure of the shortcut, the original edges can be extracted recursively.

2.10 Customizable Route Planning (CRP)

The CRP algorithm is based on the graph separators approach (graph partitioning) in combination with several other techniques and optimizations. It is currently used by the search engine Bing from Microsoft [44] and was proposed by Delling et al. [9, 10]. By dividing the preprocessing into a metric-independent and a customization phase,

the algorithm can process new metrics (edge weights/costs) faster, which is important for e.g. real-time traffic updates. The primary steps of the preprocessing stage can be summarized as:

- simplifying the graph for faster processing
- breaking up the graph into fragments by finding small separators/cuts
- partitioning the graph by combining the fragments into cells of desired size
- creating a multilevel overlay from the partitions
- customization by applying the metrics and calculating the shortest path cliques of every cell

At the query stage, a bidirectional Dijkstra's search is run on the multilevel overlay. The graph partitioning and queries are not limited to certain algorithms (Delling et al. compare several different methods in [10]). A more detailed overview of the algorithm, summarized from [10], is given in the following.

2.10.1 Graph Partitioning

The use of partitioned graphs in pathfinding is based on the divide and conquer paradigm. It can have positive performance effects not only on queries, but also on required updates from changes to the network topology. The drawback is the time required for partitioning.

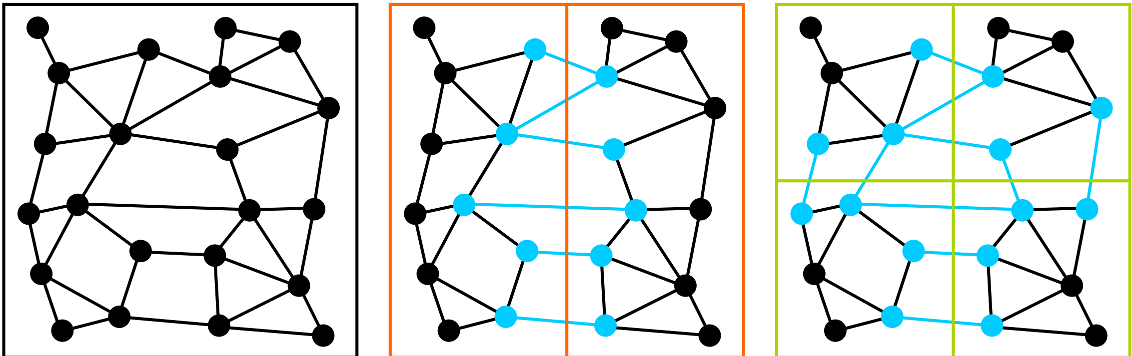


Figure 2.5: Graph partitioning: the original graph (left) is partitioned (center). The cells from the partitioned graph are divided again (right). Boundary nodes and edges are shown in blue.

Planar graphs can be divided with small separators [64]. Even though road networks are not strictly planar (because of e.g. bridges, tunnels, over-/underpasses), it has been shown that almost planar graphs also have small separators [37]. This means that partitioning leads to only a small amount of boundary nodes. A partition $P = \{C_1, ..., C_k\}$ of a graph $G = (V, E)$ with a set V of vertices (nodes) and a set E of edges

contains k cells (sets) with each $v \in V$ contained in exactly one cell C_i. Nodes with edges leading from one cell to another are boundary nodes and the edges are boundary (or cut) edges.

A multilevel partition is created by partitioning the cells from a partition again, which creates another partition containing all newly created cells (see Figure 2.5). Each partition is assigned to a distinct level in descending order, i.e. the last one created (which will be the one with the most cells) is assigned to level 1. The original graph is considered as base of a multilevel partition, placed at level 0, with every node having its own cell.

2.10.2 Overlay Graph

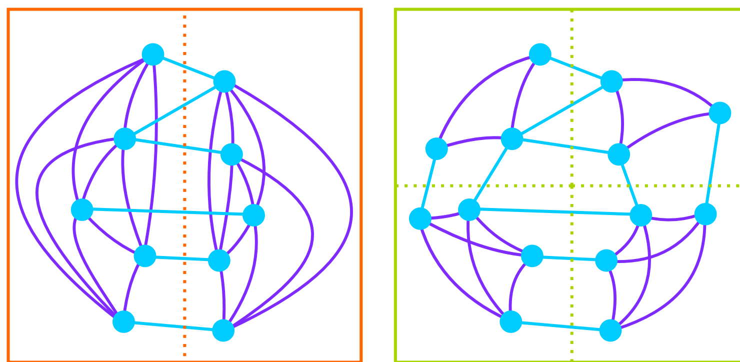


Figure 2.6: Overlay graph: the overlay graphs for the partitions from Figure 2.5 are created. Only boundary nodes and edges are inherited. Shortcuts (drawn in purple) between boundary nodes are added. The cell boundaries (shown in dashes) are not part of the overlay graph.

An overlay graph H is a contracted version of the original graph G. An overlay of a partitioned graph contains all boundary nodes and boundary edges of all cells (see Figure 2.6). To preserve the shortest paths between boundary nodes of a cell, a *clique* is build for every cell C_i of the partition P: Between every pair of boundary nodes (v, w) of a given cell C_i, an edge (shortcut) is added to H with the edge weight set to the total cost of the shortest path (restricted to C_i) between them. The cliques can be created with any method suitable to find the shortest path between the boundary nodes, which could be e.g. Dijkstra's or the Floyd-Warshall algorithm.

Because the overlay graph has fewer nodes than the original graph, shortest path queries can be answered faster between boundary nodes. Queries between any node s and t from G can be made by creating a *search graph*, which is the union of H and the cells containing s and t.

A multilevel overlay graph can be created from a multilevel partition. For every level i an overlay graph H_i is created from the partition P_i. By doing this in a bottom-up approach, with G on the lowest level, the cliques for H_i can be created from H_{i-1}, which is

faster than using G for every level. Queries in a multilevel overlay are more complicated. Nodes and edges are either scanned at the highest level where the corresponding cell does not contain s or t or otherwise in G at level 0. Level transitions can only happen at overlapping cell boundaries.

2.10.3 Preprocessing

The Customizable Route Planning algorithm splits preprocessing in a metric-independent phase, where a multilevel partition and multilevel overlay graph of the network are created, and a customization phase, where the metrics (edge weights/costs) are applied and the cliques/shortcuts of the cells in the overlays are calculated.

Because a good partition with as few separators (boundary edges between cells) as possible is crucial for the performance of queries, Delling et al. created PUNCH [11], which is a graph partitioning algorithm adjusted for road networks. A detailed description of PUNCH is given at the end of this section.

To enable multilevel queries a unique sequential identifier is assigned to every cell during partitioning and a list with the corresponding parent cell on each level is created for every cell on level 1. Also, with every node in G the identifier of the level 1 cell it belongs to is stored.

To speed up the customization phase, several acceleration techniques can be applied (for a complete description refer to [10, Chapter 5.2]):

- *Prepared data structures*: During partitioning, data structures required at customization are prepared.
- *Improving locality*: A temporary copy of the subgraph of a cell is created to increase locality during clique creation.
- *Pruning the search graph*: Internal boundary nodes of the underlying cells are contracted prior to clique creation.
- *Alternative algorithms*: The Bellman-Ford algorithm [4] instead of Dijkstra's algorithm is used for clique creation because it has better locality.
- *Parallelism*: Extended CPU instruction sets and multiple cores are used.
- *Phantom levels*: Additional partition levels are created which decreases the size of subgraphs and therefore customization time. They are not kept for queries because of the increased space consumption.

Another possibility that was not mentioned in [10] is the use of the Floyd-Warshall algorithm [19, 33, 69] in combination with a graphics card (GPU). The Floyd-Warshall algorithm conforms to the dynamic programming paradigm by breaking down the problem of shortest paths into simpler subproblems to solve it. Simpler subproblems can

have the advantage of requiring less memory or being solved in parallel. This can reduce the runtime by adapting the implementation to better utilize memory caches or multiple processing units. Venkataraman et al. [66] propose a blocked version of the Floyd-Warshall algorithm based on blocked matrix multiplication to improve the performance by better processor (CPU) cache utilization. In [42], this blocked version is adapted for a hybrid CPU-GPU system which takes advantage of the high number of processing units on a GPU and using the CPU to hide memory latency for large graphs exceeding the available GPU memory.

2.10.4 Query Algorithm

Queries use a bidirectional Dijkstra's algorithm (see Section 2.5 on page 8) which is executed on the original graph and the overlay graphs. A query for the nodes s and t starts at the original graph G in the forward and backward search. During the search, when a node v is made the active node, its *query level* $l_{st}(v)$ has to be determined. This is the highest level where v is neither in the same cell as s nor t. During relaxation, only edges from v in the overlay graph at the query level are scanned (i.e. boundary edges and shortcuts). By calculating $l_s(v)$ and $l_t(v)$, which is the highest level where v is not in the same cell as s or t respectively, $l_{st}(v)$ can be computed with $min(l_s(v), l_t(v))$. If v is in the same cell as s or t at level 1, the query level is 0 and the original graph is used. The search terminates under the same conditions as the normal bidirectional Dijkstra's algorithm.

2.10.5 Path Reconstruction

The shortest path that has been found between a pair of nodes may contain shortcuts from the cells (cliques) of the overlay graph. Several methods can be applied to reconstruct the path in the original graph:

- During the creation of the cliques in the overlay graph, the used nodes from the underlying level are stored with the shortcuts. The original edges can then be extracted recursively. This is the fastest method in the query phase but also comes with a considerable increase in memory usage.
- Instead of storing any information about the underlying nodes, a shortest path search is executed recursively between the nodes from shortcuts on the underlying levels. This has to be done after each query for all shortcuts in the shortest path. This method does not require additional space but is the slowest.
- The above method can be sped up by storing a flag with every edge indicating if it is used in a shortcut during clique creation. Unused edges can then be ignored at path reconstruction.

- Another possibility is a cache holding reconstructed paths with a least-recently used (LRU) update policy to limit the cache size.

2.11 Partitioning Using Natural Cut Heuristics (PUNCH)

While the concept of graph partitioning is straightforward, finding good partitions is not. Because this is a fundamental problem, various algorithms and solutions have been created (see [18, 68] for overviews). However, not every solution meets the requirements of different applications. Pathfinding in overlay graphs requires balanced cells with as few boundary edges as possible. For the partitioning in CRP, Delling et al. created PUNCH [11], which is a graph partitioning algorithm adjusted for road networks. It finds better partitions than general-purpose solutions such as METIS [35]. A description of how PUNCH works, summarized from [11], is given in the following.

2.11.1 Overview

Compared to other partitioning algorithms, PUNCH does not focus on balancing the size of the cells but on minimizing the number of cut edges (boundary edges). This is done by finding so called *natural cuts*, which can be e.g. rivers and mountains but also less natural things such as borders, railways or roads. These cuts separate areas of dense road networks which are connected by few boundary edges such as bridges, tunnels, mountain passes or border crossings. The algorithm is separated into two phases, *filtering* and *assembly*. During filtering, the size of the graph is reduced while preserving natural cuts. The graph is then split into fragments by solving the *maximum flow minimum cut* problem. Afterwards, the fragments are combined in the assembly phase to create cells of the desired size with as few boundary edges as possible.

2.11.2 Preliminaries

The input is an undirected graph $G = (V, E)$. Every vertex $v \in V$ has a size $s(v)$ and every edge $e \in E$ has a weight $w(e)$. Initially, every vertex has a size of 1 and every edge a weight of 2. If the input is created from a directed graph all directed edges are converted to undirected edges and initialized with a weight of 1. A vertex u that is directly connected with a vertex v can be contracted into a new vertex w with its size set to $s(w) = s(u) + s(v)$. The vertices u and v and the edges between them are removed from the graph, while the new vertex w is added. All edges incident to u or v from the remaining graph are changed to be incident to w instead. Possible parallel edges created during this process (multiple edges between w and the same neighbor n) are merged into one edge and the edge weights are combined. Contracting a set of vertices can be achieved by repeatedly applying this process.

2.11.3 Filtering

The main goal of the filtering phase is the reduction of the size (number of vertices and edges) of the graph (which reduces the running time significantly) and the preparation of fragments as input for the assembly phase. Fragments are small parts of the graph separated by natural cuts.

The size of the graph is reduced with the following steps. Every consecutive step takes the modified graph from the previous step as input:

- *Finding all bridges*: By executing a depth-first search (DFS), all articulation points and bridges can be found (see [62]). A bridge is an edge that disconnects the graph if removed (the same applies to an articulation point). In the following a subtree is a component of the graph that is only connected via a bridge with the rest of the graph.

- *Contracting small subtrees*: The previously found DFS tree is rotated to be rooted in the largest subtree. All other subtrees are then checked and contracted if their size (number of vertices) is below a given threshold.

- *Contracting paths*: Paths consist of neighboring nodes with degree 2 which are contracted into a single node.

- *Finding and contracting 2-edge-connected components*: The contracted graph is checked for 2-edge-connected components (see [52]), which are connected with exactly 2 edges with the remaining graph. All components with their size below a given threshold are contracted.

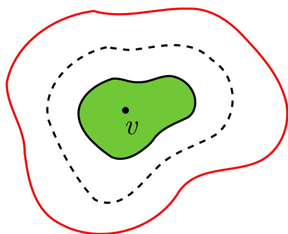


Figure 2.7: Finding a natural cut. A random vertex v is chosen as the start for a breadth-first search (BFS) to create a subgraph T with a given maximum size. The set of neighboring vertices (shown as red line) of T in $V \setminus T$ are called the *ring of v*. The *core of v* (the green area) consists of all vertices found by the BFS while the subgraph was smaller than a given threshold. The core and the ring are contracted and the minimum cut between them is calculated (dashed line).

The last step of the filtering phase is the detection of natural cuts and preparation of fragments. Natural cuts are found by computing the minimum cut (PUNCH uses the Push-Relabel algorithm from Goldberg et al. [28]) between a small set of vertices and works as illustrated in Figure 2.7. The cut edges from the minimum cut are stored and

the procedure is repeated for every vertex that does not belong to at least one core yet. After all minimum cuts are found, all cut edges C are removed from the graph. Every remaining connected component $G_C = (V, E \setminus C)$ is contracted and used as a fragment in the assembly phase. The fragments are connected with each other with the cut edges in C. Parallel edges are combined into one (see Figure 2.8).

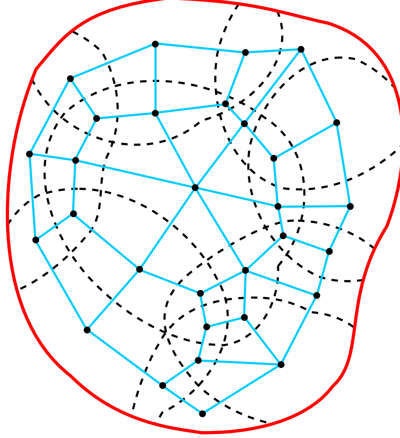


Figure 2.8: Creating a fragment graph. The original graph (encircled in red) is split into components by natural cuts (dashed lines). The components are contracted into fragments (black dots). The fragments are connected with each other by the combined cut edges (blue lines).

2.11.4 Assembly

In the assembly phase the final partition is created by combining and contracting the fragments created during filtering. The process is split in a greedy algorithm and a local search. In the following, fragments are called vertices and a vertex is synonymous with a cell of the partition. The contraction rules used above still apply.

During the greedy algorithm an initial partition is created. To find a good solution vertices are not contracted at random. Instead, a score (Equation 2.8) is assigned to every edge in the graph, where $w(u, v)$ is the edge weight and r is a biased random number between 0 and 1 (see [11, Chapter 4.1] for a full description).

$$score(u, v) = r * \left(\frac{w(u, v)}{\sqrt{s(u)}} + \frac{w(u, v)}{\sqrt{s(v)}} \right) \tag{2.8}$$

The edge with the highest score is then taken and the two vertices u and v associated with it are contracted. After contraction, the score has to be recomputed for all edges incident to the newly created vertex. This procedure is repeated until no further vertices can be combined without violating the size constrain set for a cell.

The initial partition G is then improved by local search. Two connected vertices $(r,\ s)$ are randomly chosen and a temporary graph G' consisting of r, s and all their

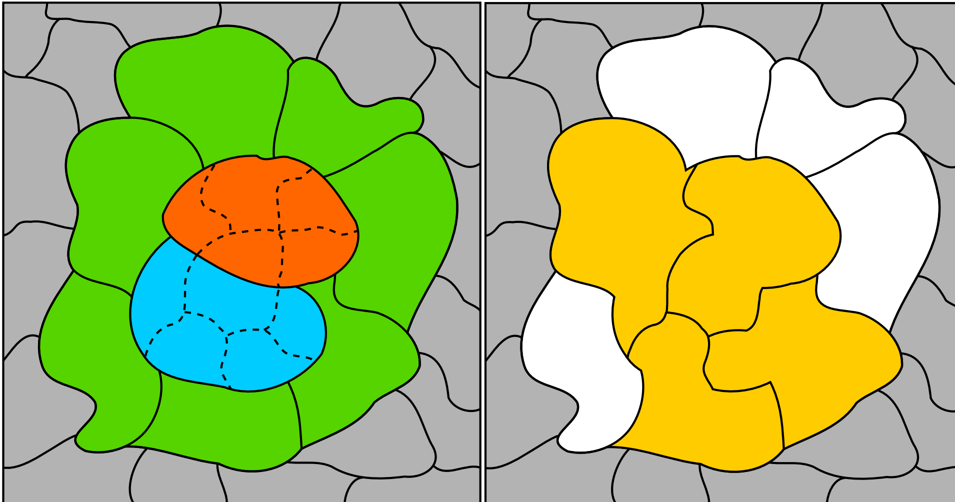


Figure 2.9: Temporary graph of local search. Left: a temporary graph is created with two randomly chosen, connected vertices (orange, blue) and their neighbors (green), with the former uncontracted into their fragments (dashed lines). The surrounding graph that is not part of the temporary graph is shown in gray. Right: a possible outcome after the greedy algorithm has been applied. Modified vertices are shown in yellow.

direct neighbors is created. Now, the vertices r and s are uncontracted, that is the initial fragments and edges from the start of the assembly phase are restored in G' (Figure 2.9). Then the greedy algorithm is applied on the temporary graph. When the algorithm stops, the solution quality (sum of the weights of the remaining edges) of G' is compared with the quality of the initial G'. If the quality has improved, the changes to vertices and edges in G' are transfered to G. A counter is maintained for every edge in G, which is increased for the edge incident to the chosen r and s every time the described procedure does not yield a better solution. When every counter has reached a given threshold, the local search is stopped and the final partition is finished.

Chapter 3

Framework and Dataset

In this chapter, the framework and the dataset used for the experimental analysis and performance comparison is presented.

3.1 Framework

A framework was created to provide the required functionality for the experimental evaluation. This includes loading, creating and manipulating networks, executing tests and recording the readings for evaluation. The framework also has various methods to process, output and visualize the results and provides common data structures and methods to the evaluated algorithms. For a fair comparison of the various pathfinding algorithms, they are all implemented in the framework. This means they are utilizing the same data structures and methods (e.g. the graph data structure and priority queue implementation) and the measurements can be taken at equivalent locations. In the following sections the design of important components in the framework are described and the configurations of the individual algorithms are given. It is assumed that the reader is familiar with the various concepts and algorithms as the following sections are only meant to give an overview of important implementation details for replicability and reproducibility.

3.1.1 Programming Language and Libraries

The whole framework is written in C++ (version 11) and uses no external libraries or code except for LodePNG [65], a lightweight PNG image encoder, which is used to write images to disk as PNG files.

3.1.2 Graph Data Structure

One of the most important parts is the data structure which stores the nodes and edges of a network. The requirements for it are fast access to the elements as well as the

possibility to dynamically add and remove nodes and edges at a low cost because some algorithms such as CH have to modify the graph during preprocessing. For this framework a modified version of the dynamic graph structure proposed by Mali et al. in [39] is used because it does satisfy the requirements and is actually designed for large dynamic transportation networks. Its core element is a *Packed Memory Array* (PMA) [5] in which the content is kept in a contiguous array intermixed with empty cells which are uniformly distributed. This allows for fast insertions at any position which is, for example, required to keep the edges of nodes close to each other for fast access. The PMA is self-managed and rebalances (i.e. redistributes the elements) or resizes itself if the specified upper and lower density bounds are reached. To confine these operations, the PMA uses a virtual binary tree that partitions the contiguous array and only those segments that are actually out of bounds are rebalanced.

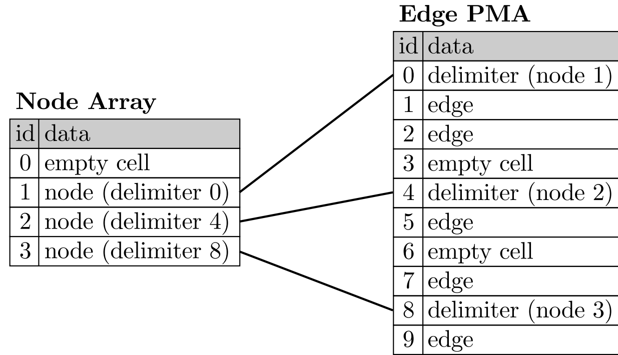


Figure 3.1: Illustration of the graph data structure.

The structure from Mali et al. uses three PMAs, one to store the nodes and two to store the incoming and outgoing edges, which allows for maximum flexibility in regard to insertion, deletion and order. Because not every feature is required for this framework, the following modified version is used. The PMA is replaced by a simple array for the nodes since the order of them does not matter and insertions just happen at the first free cell. Only one PMA is used for the edges with a flag indicating if an edge is incoming, outgoing or both. To associate nodes with edges and vice versa a delimiter is introduced which is placed in the PMA between the edges of different nodes. An index-pointer is assigned to every node and delimiter to access each other (see Figure 3.1). To keep the index-pointers synchronized, a callback function is linked to the delimiters which gets called when they are moved. The lower and upper density bounds used for the PMA, which have been determined after some tests, are $[0.3, 0.7]$ for the whole array and $[0.01, 1]$ for the leaves (smallest segments).

Node and Edge Data

The minimum data that is required to be stored with every node is the index-pointer to its delimiter in the edge array. The coordinates have to be included if the graph is going

to be visualized. They are also needed for the A* algorithm, which requires them for its heuristic, and for ALT, which requires them to partition the graph during preprocessing. For CH, a second index-pointer has to be stored to separate edges from the upward and downward graph and CRP requires the identifier of the cell a node belongs to.

With every edge its target node identifier, weight (e.g. geometric distance, travel time) and flags indicating its direction are stored. An additional identifier is required to match the corresponding edge entries between two nodes for advanced graph manipulations. For CH, a flag indicating if an edge is a shortcut and the identifier of the shortcut's intermediate node have to be stored.

3.1.3 Priority Queue

Priority queues (PQ) play an integral part in all presented algorithms and can be implemented in many different ways. The implementation of the priority queue in this framework is based on a binary heap and supports querying, removing and updating (also known as *decreaseKey*) any element. It uses a lookup table to associate external items with contained elements and provides the possibility to store additional data with them (e.g. the predecessor node during a search). Additionally, the code uses templates to allow any data types and to choose between a minimum or maximum queue. This means that any algorithm can use it without modification or providing storage for queue-related data.

Several optimizations are used to improve the performance. The internal heap starts at index one for faster parent/child calculation and the cell at index zero is used as temporary slot when elements are moved. Also, the *bottom up heuristic* [70] is applied: after the top element is removed, the empty element is first moved down into a leaf node and only then it is swapped with the last element which is then moved up.

4-ary Heap

A 4-ary heap [63, Chapter 3.2] instead of a binary heap was tested for its advantages of better memory layout and lower height. On the largest evaluated network using Dijkstra's algorithm, the maximum number of elements in the heap never exceeded 6 400 and stayed below 2 700 on average. An element consists of an identifier and a weight with a total size of eight bytes leading to a maximum memory requirement of only 50 kB for the whole heap. This means that the better memory layout of the 4-ary heap, which should reduce the number of cache misses, did not improve the performance, because modern processors offer far larger caches of several megabytes [34].

Over all evaluated networks using Dijkstra's algorithm, the function to move elements towards the root (decreaseKey) was called between 2.3 and 2.5 times more often than its counterpart which moves elements towards the leaves. This would indicate that a wider heap with a lower height should perform better. A closer inspection though revealed

that even on the largest network, elements are only moved 1.62 levels per call towards the root, on average, with a binary tree. Using a 4-ary heap reduced the number to 1.22. This means that its lower height only offers a small advantage. However, this advantage is lost by the more complex parent/child index calculations. The tests showed that neither of both heap variants has a measurable advantage over the other with Dijkstra's algorithm on road networks.

3.1.4 Customizable Route Planning (CRP) and Natural Cuts

The most complex part of the framework is the implementation of PUNCH [11], which partitions a graph along its natural cuts for CRP. An overview of the work flow and the most important parts is given in Figure 3.2. At first, smaller biconnected subtrees and paths in the input graph are contracted, followed by 2-edge-connected components which are found via cycle space sampling [52]. The contracted graph is broken up into fragments along its natural cuts. The final partition is then created by combining the fragments during the assembly phase. Finally, the graph is expanded again and the nodes are assigned to their cells.

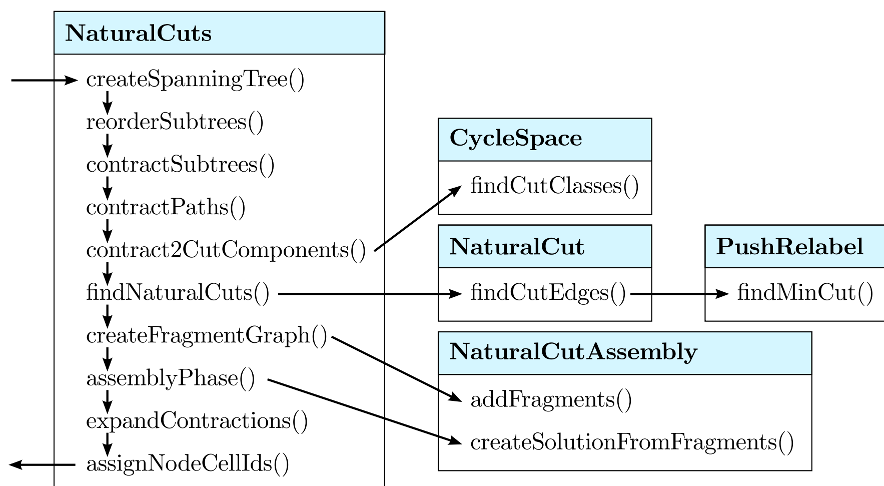


Figure 3.2: Natural cuts work flow overview with the most important classes and methods.

Push-Relabel

One of the core elements is the *Push-Relabel* algorithm, proposed by Sleator and Tarjan in [28], which finds minimum flow cuts in graphs. Although the algorithm itself is straightforward, several different configurations are possible. Excess flow can be pushed either directly between neighbouring nodes, or by the use of a *link-cut tree* [60] which is a forest of *splay trees* [59]. The link-cut tree has the advantage that it can push across several nodes at once, reducing the total number of pushes. Both methods were

implemented and tested and even though the link-cut tree required fewer pushes, its computational overhead made it significantly slower and is therefore not used.

Another choice that can be made with Push-Relabel is the order in which nodes are processed. The method that proved to be the fastest is a combination of first-in first-out (FIFO) and relabeling the active node until all excess is gone. Additionally, gaps are detected during relabeling and all labels are recalculated frequently (after each x relabels, where $x = |V|/10$).

Cliques

For every cell, the shortest paths between the boundary nodes have to be computed which is done with Dijkstra's algorithm. All intermediate nodes are stored for faster path unpacking during queries.

Settings

Six levels are used for the multilevel partition of CRP, where at level 0 every node has its own cell and level 6 consists of one cell containing all nodes. At the levels 1 to 5 the graph is partitioned with PUNCH and a cell size of $U = 2^{level*3+5}$. Also, no phantom levels are used. Several parameters can be configured for PUNCH. The settings used are shown in Table 3.1 and follow the recommendations from Delling et al. in [11].

Table 3.1: Settings used for PUNCH. The symbols used for each parameter are identical to those used by Delling et al. in [11].

parameter	description
$U = 2^{level*3+5}$	maximum size of nodes, fragments and subtrees to be contracted
$\tau = 5$	maximum size of subtrees that can get contracted and combined with its root node
$\zeta = 1$	*coverage*; counter determining how often the natural cut procedure is executed
$\alpha = 1$	factor for the BFS tree size during natural cut search
$f = 10$	factor for the core size during natural cut search
$\varphi = 16$ $\varphi = 4$ (at level 1)	maximum number of retries for a node pair from the solution to be refined during natural cut assembly
$a = 0.03$	randomization term (r) probability for score calculation during natural cut assembly
$b = 0.6$	randomization term (r) interval for score calculation during natural cut assembly

3.1.5 Contraction Hierarchies (CH)

During the preprocessing phase of CH, the order in which nodes are contracted is determined by the *priority terms*. The terms used in this framework are the *edge difference*, which is the difference between newly created shortcuts and the number of incident edges of a node and the *search space size*, which is the number of settled nodes during the local (*witness path*) search.

The chosen values for parameters follow the recommendations by Geisberger in [22]. The coefficient used for the edge difference is 190 and 1 for the search space size. The node limit during local searches is set at 1 000 and a full update of the priority of all remaining nodes is initiated after each 1 000 lazy updates. Other implementation choices are summarized in the following list:

- Local searches for witness paths are performed with Dijkstra's algorithm. The *stall-in-advance* technique and *hop limits* described in [22] are not implemented.
- Every shortcut stores its intermediate node for fast unpacking during queries.
- Queries are using the *stall-on-demand* technique [57], which prunes the search at nodes that have been reached over a suboptimal path.

3.1.6 A*, Landmarks, Triangle Inequality (ALT)

Three different methods are implemented for the selection of landmarks: *Random*, *Partition-Corners* [17] and a modified version of it. A detailed description of them is given in Section 4.6.2 on page 61. The distance from and to landmarks is calculated with Dijkstra's algorithm.

3.1.7 Other Components

The two remaining algorithms, Dijkstra and A*, do not have any special settings and are implemented as described earlier in the theoretical part. Furthermore, several tools/-classes are implemented which are summarized in no particular order in the following:

- *FlatSet*, *FlatMap*: For many tasks, short lists of unique elements are required (e.g. unique neighbors of a node). The containers provided by the Standard Template Library (STL) are designed for many entries. This requires a more complex internal representation which imposes a performance penalty. The implemented flat versions, however, only consist of a contiguous array and duplicates are identified by iterating over it, which is extremely fast with few entries.
- *ActiveList*: Consider a list of (active) items that is processed (in no particular order) but after every step one or more items are removed randomly. The active list solves this by keeping a pointer to the last active item on the list. Every time an

item has to be removed, it is swapped with the last active item and the pointer is decremented by one. This keeps the remaining active items in front of the pointer and the removed items behind it. The advantage of the active list is that it moves as few items as possible and the list can be reused endlessly by just resetting the pointer.

- *FastRandom*: For performance critical parts, a fast random number generator based on *Xorshift* from Marsaglia [41] is used.
- *SVG*: A lightweight class to create and save Scalable Vector Graphics (SVG) to visualize smaller graphs.
- *PNG*: A lightweight class to create bitmap images to visualize larger graphs. The bitmaps are saved as Portable Network Graphics (PNG) files with the LodePNG [65] library.
- *XML*: A simple Extensible Markup Language (XML) reader to import the OpenStreetMap data files.
- *Network*: The Network class provides all the functionality related to graphs. This includes:
  - importing/exporting graphs with different file formats
  - creating random graphs
  - adding and removing nodes and edges
  - accessing nodes and their corresponding edges and querying properties
  - visualizing graphs
- *OSMNetwork*: A class to filter and convert the XML data files from OpenStreetMap into compact graph data files for this framework.
- *ProcessingStatistics*: A class that helps collecting the measurements/statistical data taken by the tests.
- *Stopwatch*: With the Stopwatch class, the time taken by the tests is recorded.

3.1.8 Final Remarks

The complete framework (excluding the LodePNG library) has approximately 31 200 lines of code including comments in 100 files. Most of its functionality can be accessed from the command line and details how to compile and use it can be found in the appendix (Chapter C on page 132). The executable used for the experimental analysis has been compiled using g++ 4.8.4 with optimization level 4 (-O4), no debug information and no assertions under Linux Mint 17 (64 bit).

3.2 Dataset

A wide range of road networks from all over the world with different sizes, ranging from cities to continents, has been chosen to be used for evaluation and benchmarking. They are divided into the three main categories *Cities & Metropolitan Areas*, *Regions & Countries* and *Continental Sized*. The first two categories are further divided into *North America*, *Europe* and *Other World*.

The number of networks per category is shown in Table 3.2. The mean number of nodes for each category and separated for full and contracted networks is shown in Table 3.3. The difference between full and contracted networks is explained further below and a detailed list with key figures for all networks can be found in the appendix (Section A.1 on page 123).

Networks from the *Cities & Metropolitan Areas* category always designate the greater/metropolitan area if they are not specifically marked with the word *City*. To emphasize this, they will be decorated with the phrase *Greater Area* in some sections.

Table 3.2: Categories and number of networks. The detailed lists and key figures can be found in the appendix (Section A.1 on page 123).

category	short name	subcategory	short name	networks	detailed list
Cities & Metropolitan Areas	Cities	North America	C-NA	42	Table A.1
		Europe	C-EU	34	Table A.3
		Other World	C-OW	26	Table A.4
Regions & Countries	Regions	North America	R-NA	8	Table A.5
		Europe	R-EU	17	Table A.6
		Other World	R-OW	1	Table A.7
Continental Sized	Continental			6	Table A.8

Table 3.3: Mean number of nodes per category.

category	all networks mean nodes	full networks mean nodes	contracted networks mean nodes
C-NA	559 922	929 657	190 186
C-EU	280 621	457 489	103 754
C-OW	451 732	722 818	180 645
R-NA	5 470 332	10 652 044	3 743 095
R-EU	4 823 066	8 453 526	1 595 990
R-OW	4 834 762	-	4 834 762
Continental	14 926 827	23 947 347	13 122 723
all	2 004 233	2 459 156	1 612 494

All networks except one are based on current data from 2016 from **OpenStreetMap**[1]. The USA network data has been acquired from the **9th DIMACS Implementation Challenge - Shortest Paths** website[2] and has been included as a reference point for comparison, because it is used in the papers for Contraction Hierarchies [23] and Customizable Route Planning [9]. According to the source, the data for the USA network contains errors (e.g. gaps in freeways and bridges) and contains data from 2002 [14].

3.2.1 Metrics

Every network is strongly connected (which means every node can be reached from any other node) with directed and undirected edges. Two metrics are available for every edge/network: *geometric distance* and *travel time*. The geometric distance is the spherical distance between two nodes in road networks (and the Euclidean distance for synthetic networks). The travel time is the time required to traverse the given edge (road segment) at the given speed limit. Most road segments in the OpenStreetMap dataset are assigned to a category, but not each of them has a speed limit set. For those segments, the default speed limits listed in Table 3.4 are used, which are based on the averages over all networks.

Table 3.4: Default speed limits for road segments according to their OpenStreetMap category.

category	speed limit [km/h]
motorway	120
trunk	100
primary	100
secondary	70
tertiary	50
residential	30
others	50

Figure 3.3 shows the distribution of the edges for different speed limits. The difference between North America and the rest of the world is due to users, who are adding data to OpenStreetMap, having a different interpretation of the road categories in OpenStreetMap.

[1] http://www.openstreetmap.org
[2] http://www.dis.uniroma1.it/challenge9/download.shtml

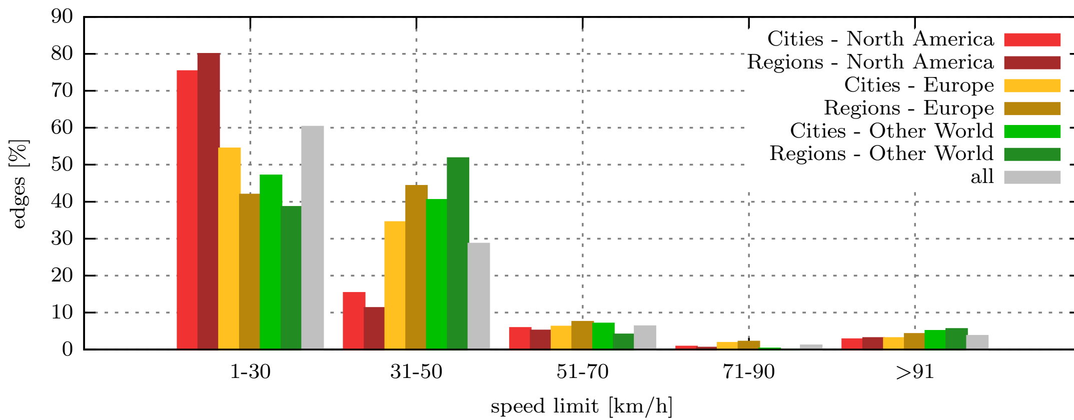


Figure 3.3: Distribution of edges for different speed limits.

3.2.2 Full and Contracted Networks

In the raw network data, nodes are used to represent intersections and end points but also geometry, which means that many paths consist of several nodes. This leads to the problem how such paths should be handled. By using the data without modification, more memory is consumed and the algorithms may take more time because of the higher number of nodes and edges. Instead, paths with several nodes could be contracted into a single edge. However, this means that some information about the geometry is lost and the question arises how exactly a path should be contracted.

An extreme version is shown in Figure 3.4 which keeps contracting all paths until none is left. While the size of the graph can be reduced by a large amount with this method, the extra work that has to be done to translate search queries that either start or end at removed nodes or edges may be significant.

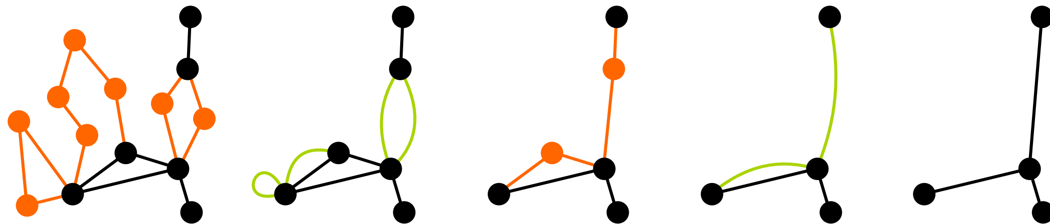


Figure 3.4: Unrestricted path contraction. From left to right: the paths (orange) are contracted. This leads to new edges (green). Resulting cycles are removed because they are not used in shortest paths and some algorithms do not even function with them. For parallel edges only the edge with the smallest weight is kept because the other (longer) edge(s) are never used during a search. After removing cycles and parallel edges, new paths have formed that can be contracted. This leads to a cascading effect that can collapse some graphs into a single vertex.

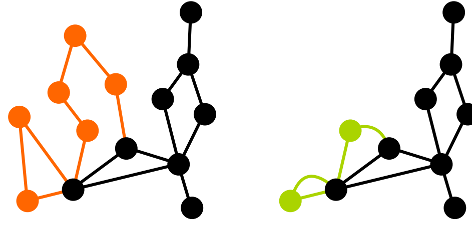
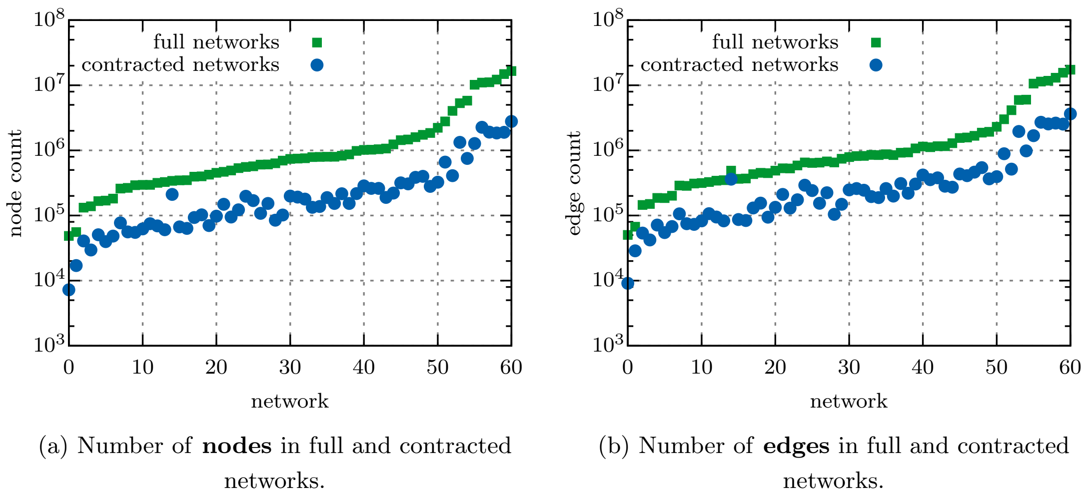


Figure 3.5: Restricted path contraction. Only paths with at least two nodes (orange) are contracted into a path with a single node (green). This has the advantage that no cycles or parallel edges are created and the paths of the original graph are still part of the final graph.

A more restrictive method, which is used in this work, is shown in Figure 3.5. Only paths with at least two nodes are contracted into a compact version with one node and two edges. This leads to some loss of geometric information but the paths themselves are conserved. To investigate the impact of contraction on the performance of the algorithms, two versions of every network are used: a full version, and a contracted one. For the largest networks though, only the contracted version is used because the test system does not have enough memory to fit the full ones which have tens of millions of nodes and edges.

In Figure 3.6, the number of nodes and edges for the full and contracted versions of all networks are shown. A logarithmic correlation between them can be seen, with two exceptions, the network of Graz and Buenos Aires.



(a) Number of **nodes** in full and contracted networks.

(b) Number of **edges** in full and contracted networks.

Figure 3.6: Comparison of number of nodes and edges in full and contracted networks.

3.2.3 Copyright and License

The USA network is based on data which has been downloaded from the **9th DIMACS Implementation Challenge - Shortest Paths** website[3] on the 20th July 2016 [14]. No copyright or license is mentioned, only the source is cited as ”UA Census 2000 TIGER/Line Files; U.S. Census Bureau, Washington, DC; Geography Division”[4].

All other networks (including roads, railways and waterways) are based on raw data from **OpenStreetMap**[5] which has been downloaded from **Mapzen**[6] and **Geofabrik**[7] on the 22nd September 2016 [24, 40, 48]. The data is under the copyright of **© OpenStreetMap contributors**[8] and provided through the **Open Database License ODbL 1.0**[9].

[3] http://www.dis.uniroma1.it/challenge9/download.shtml
[4] https://www.census.gov/geo/maps-data/data/tiger-line.html
[5] http://www.openstreetmap.org
[6] https://mapzen.com/data/metro-extracts/
[7] http://download.geofabrik.de/
[8] http://www.openstreetmap.org/copyright
[9] http://opendatacommons.org/licenses/odbl/, http://wiki.osmfoundation.org/wiki/Licence

Chapter 4

Experimental Analysis

In this chapter, the algorithms and concepts described in the theoretical part are applied to solve shortest path problems on various graphs. Their behavior, special characteristics and performance is analyzed in a practical setup through numbers and visually and compared with each other.

4.1 Preliminaries

Width and height specifications given for road networks apply to the center of the graphs and will differ widely from the real values on the edges. Also, illustration of networks are done based on their geographic coordinates without any transformation, which means they are distorted.

Different methods exist to count edges between nodes in directed graphs. In this work, edges between nodes are only counted once, i.e. if node u and v are connected by several directed edges with different edge weights, it still counts as only one edge.

4.2 Experimental Setup

For the evaluation on road networks, 10 000 pairs of nodes are selected uniformly at random for every network and the shortest path is computed for each pair with geometric distances and travel times. Because CH and CRP can answer queries much faster, 100 000 pairs are selected with them for accurate results.

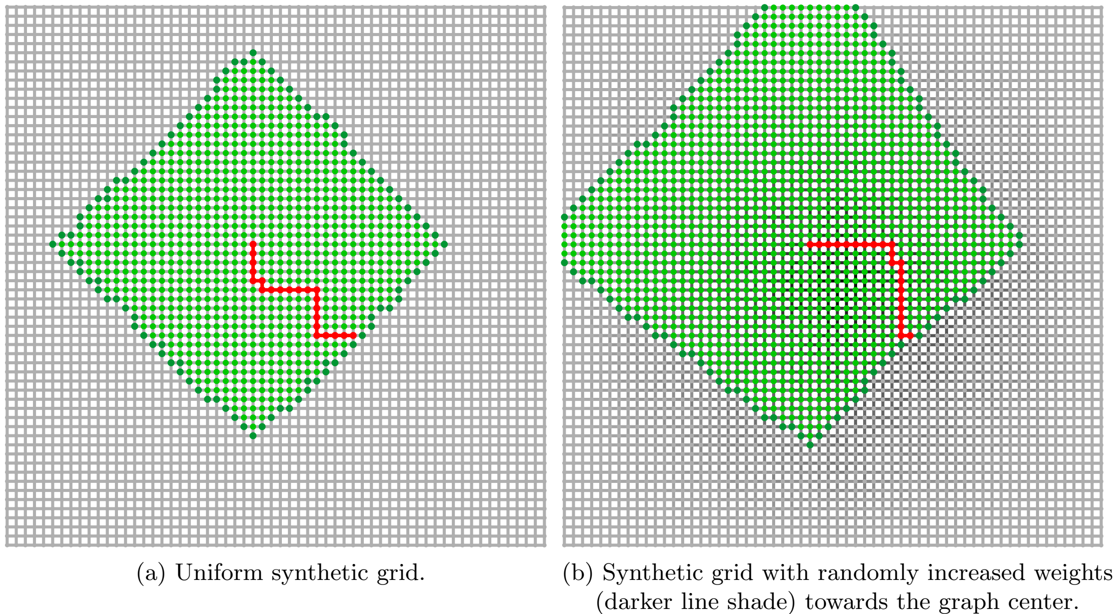
All tests are run in a single thread on an Intel Core i5-2500k processor running at 4.4 GHz with 16 GB of memory inside a virtual machine (Oracle VM VirtualBox 5.0.26) with Windows 10 Education (64 bit) as host and Linux Mint 17 (qiana 3.13.0-24-generic, 64 bit) as guest operating system. The performance penalty of the virtual machine is approximately 400 MHz.

Please refer to the previous Chapter 3 on page 29 for details about the framework, configuration of the evaluated algorithms and used dataset.

4.3 Dijkstra's Algorithm

Dijkstra's algorithm finds the shortest path between a source and one or more other nodes. The algorithm requires no preprocessing or modification of the input graph. It is a building block of many other algorithms, including those presented in this work, which means its behavior and performance has a direct impact on them. The basic, unidirectional version will be inspected in the following.

4.3.1 Synthetic Grids



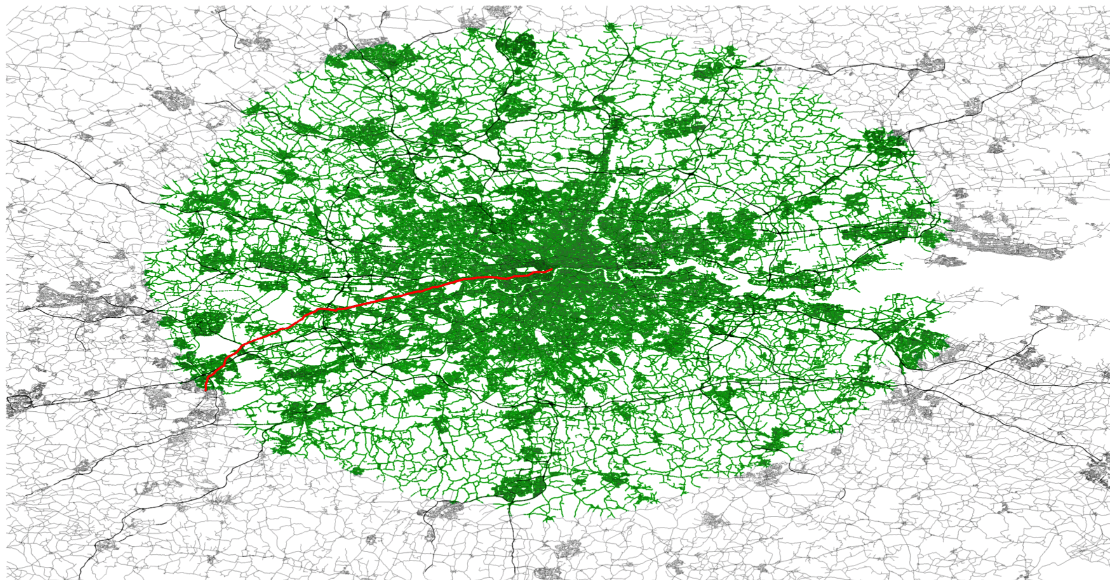
(a) Uniform synthetic grid.

(b) Synthetic grid with randomly increased weights (darker line shade) towards the graph center.

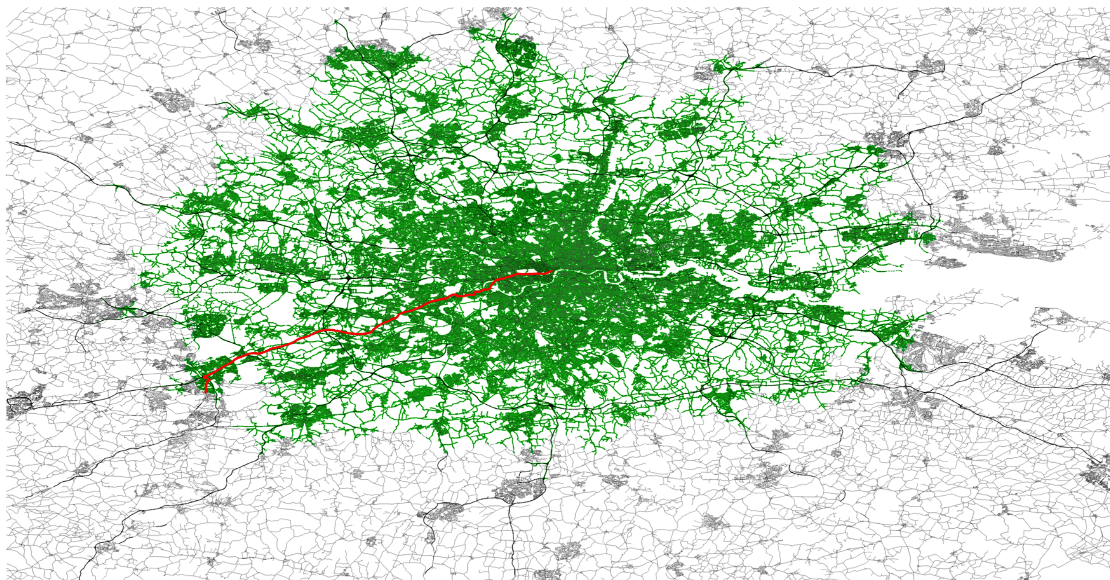
Figure 4.1: Dijkstra's search on synthetic grid graphs. The start is the upper left and the target the lower right end of the red line, which shows the shortest path found. Nodes in the closed set (settled nodes) are drawn in light green. Nodes in the open set are drawn in dark green.

Figure 4.1 shows a Dijkstra's search on a grid graph. The pattern of the nodes settled by the algorithm matches the expected circular expansion, which on a grid graph resembles a square (Figure 4.1a). By introducing increased edge weights towards the graph center (this could be e.g. a city center with increased congestion), the pattern changes, which in this case looks more like a trapezoid (Figure 4.1b). The algorithm settles more nodes at paths not leading to the target because it keeps expanding the nodes with the currently shortest distance, independent of the position of the target. This has the following consequence: For primarily planar graphs such as road networks, one can expect Dijkstra to expand in a circular fashion only if the edge weights are in proportion to the geometric distances. Other metrics result in different patterns which may impact the (expected) number of nodes settled.

4.3.2 Road Networks



(a) Search on Greater London Area with **geometric distances**.



(b) Search on Greater London Area with **travel times**.

Figure 4.2: Dijkstra's search on Greater London Area (224×116 km) with geometric distances and travel times. Settled nodes are drawn in green, the shortest path in red.

Figure 4.2a and 4.2b illustrate a search with geometric distances and travel times, respectively, for the Greater London Area, which has been chosen as a typical example of a European metropolitan area, with a dense city center surrounded by high speed/-capacity roads in ring and star formation. The search starts in the city center at the Trafalgar Square and stops at the Farnborough Airport south-west of it. Comparing both figures, the first one using geometric distances shows a circular expansion while in

the second one with travel times, it is clearly recognizable that the expansion happens more rapidly along the motorways (e.g. the search is almost cut off in the south along the horizontal motorway and spreading along the vertical ones in the north). Because the chosen target node is close to a motorway, the travel time search benefits from this behavior and settles fewer nodes (945 505) than the other search with geometric distances (1 074 016).

Table 4.1: Comparison of the average performance of Dijkstra's algorithm with geometric distances and travel times on the Greater London Area.

metric	nodes settled	node efficiency [%]	query time [ms]
geometric distances	804 485	0.204	136.1
travel times	803 247	0.151	148.8

Comparing the average performance of 10 000 random search queries on the Greater London Area (Table 4.1) reveals some interesting details. Even though the number of settled nodes is about the same for both metrics, the search with travel times is 26% less efficient. The reason for this lies in the fact that segments of certain road types, e.g. motorways, consist of fewer nodes than others and that the shortest paths prefer different road types, depending on the metric.

In Table 4.2, the average distribution of edges from shortest paths among different road types is compared between different metrics. The road types are based solely on the speed limit. While with geometric distances the shortest paths run mostly along secondary roads, with travel times there is a strong shift to motorways. Using secondary roads gives a more direct and shorter path to the target with geometric distances, while using the path via the motorway leads to a shorter travel time due to the higher speed limit. Because motorways consist of fewer junctions, which means fewer nodes in the graph, the shortest paths with travel times consist of fewer nodes, as the last column in the table shows. Switching from the full network, where nodes are also used to represent geometry, to the contracted network increases the difference between the shortest paths node count even more from a quarter to a half.

Table 4.2: Average distribution of edges from shortest paths among different road types for different metrics on the Greater London Area.

network	metric	motorway [%]	primary [%]	secondary [%]	residential [%]	total edge count
London	distance	7.0	22.5	62.5	8.0	1 641
(full)	time	44.5	25.3	28.5	1.8	1 215
London	distance	1.8	18.3	68.4	11.5	400
(contracted)	time	12.8	30.0	52.3	4.8	187

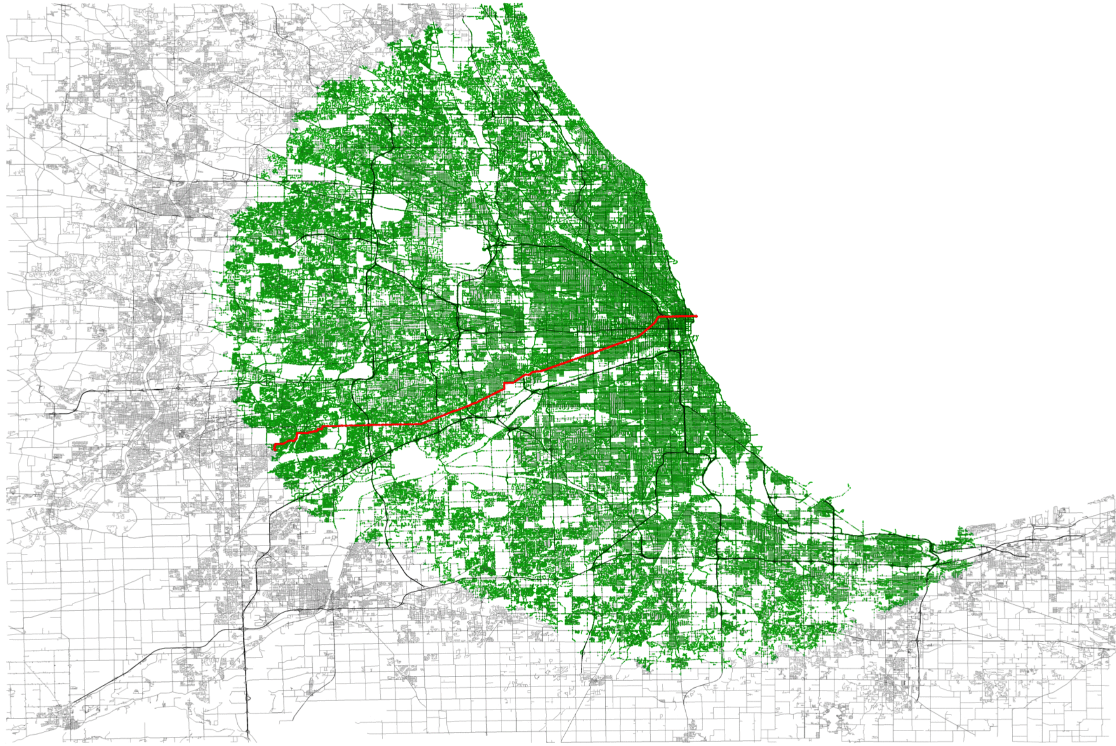
This means that the edge efficiency, which is often used as hardware independent metric to compare the performance between different graphs or algorithms, may not be used to compare different metrics. Instead, the number of settled nodes can be used to get a more accurate picture. Taking another look at Table 4.1, though, shows that a similar number of settled nodes does not equate to same query times. The reason for the difference in actual performance is that the search with geometric distances has a higher locality. The fringed search front of the search with travel times leads to a larger open set which requires more computations and makes it generally slower.

The next figures, 4.3a and 4.3b, show searches on the Greater Chicago Area, which has the typical grid layout of North American cities and high capacity roads leading straight into the city center. The search start is once again located at the city center (the Chicago Navy Pier), but the target is farther away from the next freeway in the suburbs south-west. In Figure 4.3a, with the search using geometric distances, the nodes are settled almost in a circular pattern. The grid layout does not have the same effect on the search pattern as seen on the uniform grid before (Figure 4.1). A closer look at the graph shows that there are enough diagonal roads to cancel out the square effect. Only in the south-south-east area those roads are missing and a slight tendency for the square pattern is visible. In Figure 4.3b, where travel times are used, the same effect as with the London graph can be seen. The search expands faster along paths with a lower travel time. While this had a positive effect on the number of settled nodes last time, the reverse happens this time because the target is farther off. The distance search settled fewer nodes (530 254) than the travel time search (633 620).

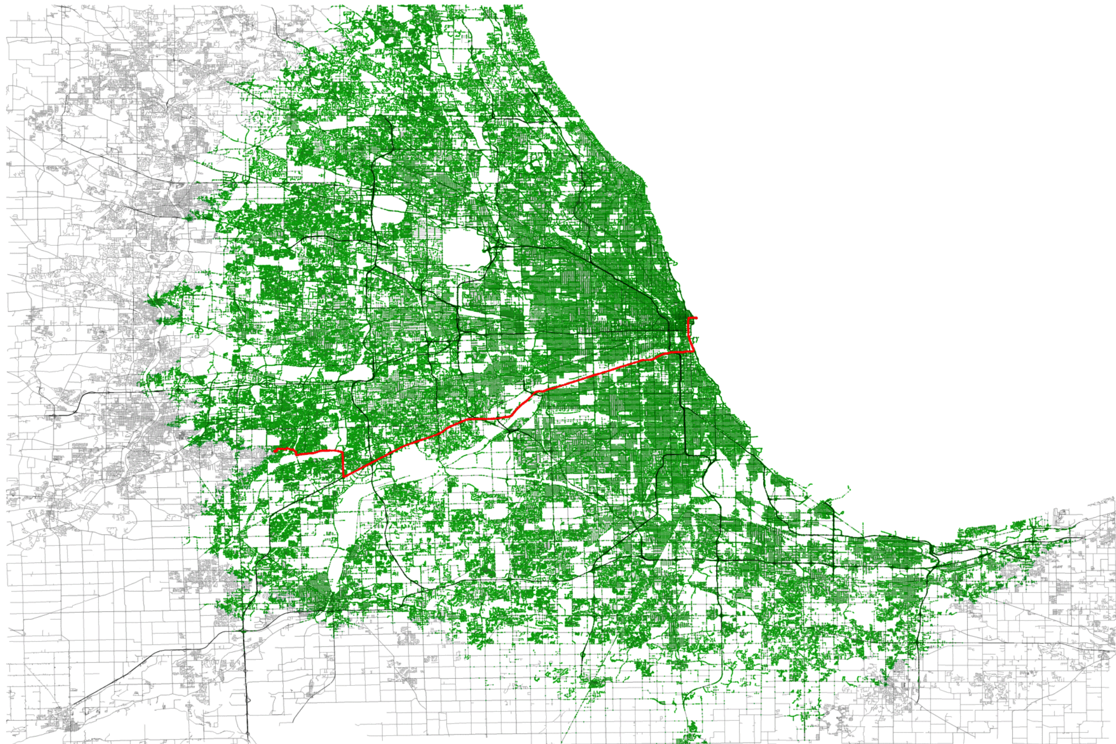
Table 4.3: Comparison of the average performance of Dijkstra's algorithm with geometric distances and travel times on the Greater Chicago Area.

metric	nodes settled	node efficiency [%]	query time [ms]
geometric distances	416 135	0.170	73.5
travel times	415 313	0.162	78.6

Table 4.3 shows that the difference in node efficiency between geometric distances and travel times is not as pronounced as it was on the London graph. The cause for this can be found in the average distribution of edges from the shortest paths (Table 4.4 on page 47). On the Chicago network, more edges from both metrics share the same road type which reduces the discrepancy. That primary roads are the predominant road type for both metrics indicates that Chicago has a better arterial road layout than London, where secondary roads prevail. Still, the query time suffers a similar performance hit caused by the fringed search front of the search with travel times, which can be seen in Figure 4.3b.



(a) Search on Greater Chicago Area with **geometric distances**.



(b) Search on Greater Chicago Area with **travel times**.

Figure 4.3: Dijkstra's search on Greater Chicago Area (160×107 km) with geometric distances and travel times. Settled nodes are drawn in green, the shortest path in red.

Table 4.4: Average distribution of edges from shortest paths among different road types for different metrics on the Greater Chicago Area.

network	metric	freeway [%]	primary [%]	secondary [%]	residential [%]	total edge count
Chicago (full)	distance	3.8	59.7	18.4	18.0	705
	time	17.7	71.1	7.4	3.6	672
Chicago (contracted)	distance	1.5	62.3	18.3	17.9	251
	time	7.8	83.1	5.7	3.4	209

4.3.3 Results & Conclusion

In Table 4.5, the average performance over all test cases is shown for Dijkstra's algorithm. Even though the search pattern differs between searches with geometric distances and travel times, about the same number of nodes is settled with both metrics. The node efficiency is not suited to compare the performance between metrics, because shortest paths with travel times prefer road types with fewer junctions (nodes) such as motorways which means they consist of fewer nodes. This leads to a lower efficiency, even when the same amount of nodes was settled compared to a search with geometric distances. Query times are generally faster with geometric distances because the search front is more compact which results in a smaller open set throughout the search. This increases the locality (e.g. during memory accesses) and reduces the number of computations. Overall, Dijkstra's algorithm only achieves a node efficiency of 0.32% and even on the network where it performed best (the contracted network of Graz) it was below two percent. On average, half of all nodes in a network are settled by it.

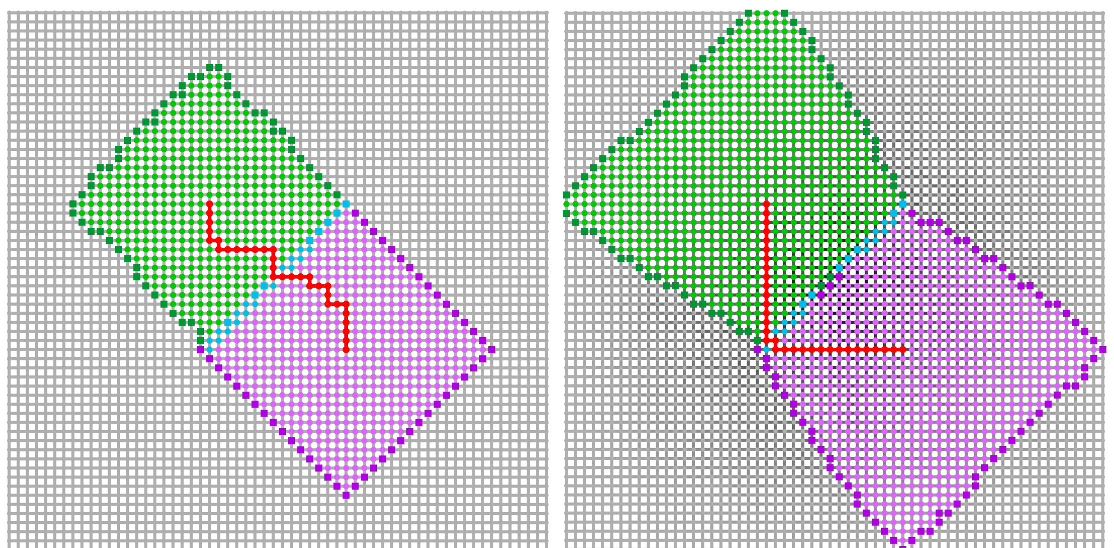
Table 4.5: Average performance of Dijkstra's algorithm with geometric distances and travel times.

	node efficiency [%]		nodes settled		query time [ms]	
category	distance	time	distance	time	distance	time
C-NA	0.294	0.242	280 045	280 055	48.4	52.3
C-EU	0.523	0.437	140 671	140 647	22.9	24.4
C-OW	0.241	0.208	294 931	294 916	53.71	56.75
R-NA	0.081	0.057	2 746 181	2 741 905	562.2	608.3
R-EU	0.199	0.132	2 413 498	2 417 147	480.9	530.9
R-OW	0.086	0.038	2 428 244	2 428 225	488.8	525.7
Continental	0.058	0.052	7 385 305	7 383 886	1 717.7	1 903.8
all	0.320	0.262	999 641	999 779	206.5	226.4

4.4 Bidirectional Dijkstra's Algorithm

In the bidirectional version, Dijkstra's algorithm is extended by a backward search with the goal of reducing the number of settled nodes. The algorithm alternates between the forward and backward search and stops when the combined shortest tentative distance to nodes in the open sets of both searches is larger than the shortest path discovered so far (see Section 2.5 on page 8). While the number of settled nodes may be lower with the bidirectional version, the actual time required for a search does not necessarily benefit to the same extend or may be worse even. The reason for this is that two searches also require maintaining two priority queues and querying the graph at two different positions leading to reduced locality during memory accesses which can have a severe performance impact because of cache misses (see [15] for a detailed analysis of memory and caches).

4.4.1 Synthetic Grids



(a) Uniform synthetic grid.

(b) Synthetic grid with randomly increased weights (darker line shade) towards the graph center.

Figure 4.4: Bidirectional Dijkstra's search on synthetic grid graphs. The start is the upper left and the target the lower right end of the red line, which shows the shortest path found. Nodes in the closed set (settled nodes) are drawn in light green for the forward search and in light purple for the backward search. Nodes in the open set are drawn in dark green and dark purple for the forward and backward search, respectively. Nodes touched by both searches are drawn in blue.

Figure 4.4a illustrates a search with the bidirectional Dijkstra's algorithm on a uniform grid. As expected, the search pattern resembles two unidirectional searches originating from the start and target node (as has been mentioned before, the circular expansion of Dijkstra's algorithm resembles a square on a grid graph). The grid graph

is well suited to demonstrate how the search can not stop just because the search fronts have met, because there could still be a shorter connection between other nodes (e.g. the nodes drawn in blue in the figure). Only after both searches have extended in all directions by the same distance as to the joint node, no shorter path can exist.

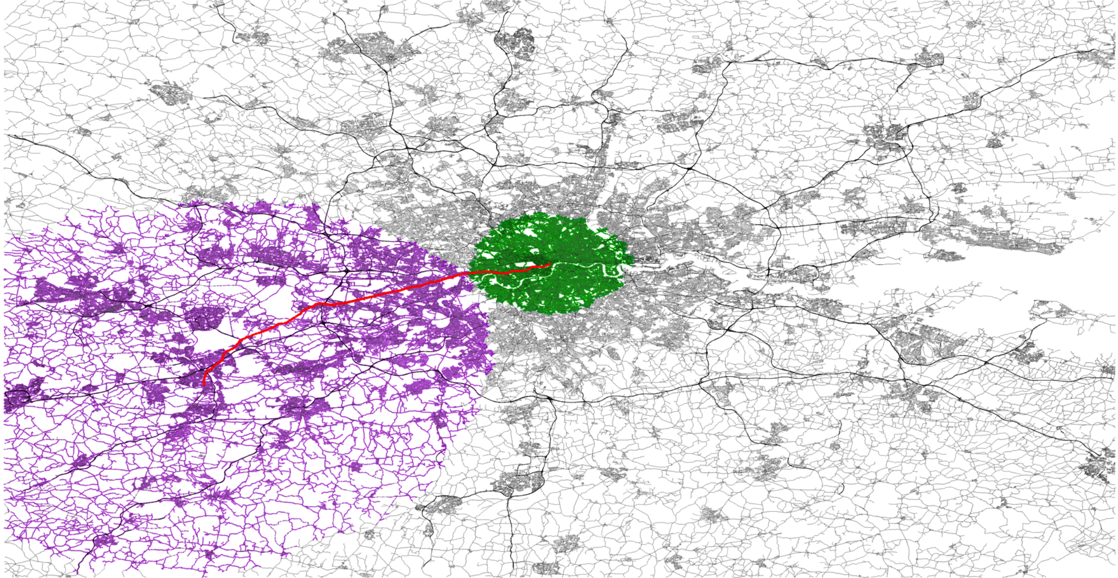
The search on the grid graph with increasing edge weights towards the center (Figure 4.4b) follows the same pattern. It should be noted that the start and target node of the search query have been chosen further apart compared to the demonstration for the unidirectional Dijkstra's algorithm in Figure 4.1 to emphasize the resemblance of the search patterns. The unidirectional version would have to scan 1.85 times more nodes to find the shortest path.

4.4.2 Road Networks

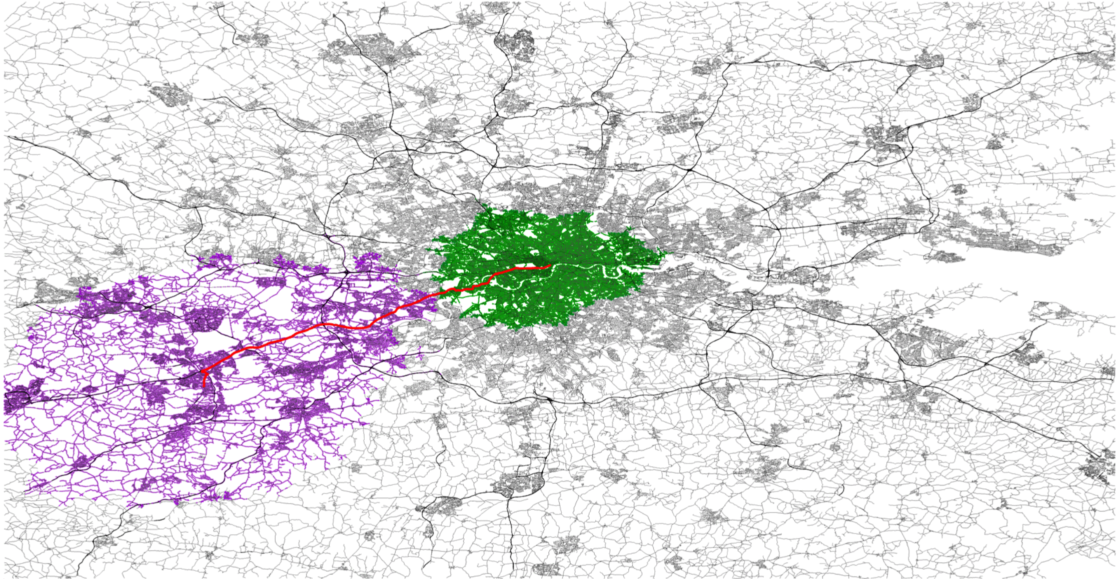
Figure 4.5 demonstrates a search with a bidirectional Dijkstra's algorithm on the Greater London Area with geometric distances and travel times. The forward search (shown in green) spans a much smaller area than the backward search (shown in purple) on the London graph for two reasons. The network at the city center, where the forward search starts, is denser which has the effect that the searched area looks smaller. The main reason, though, is that the algorithm does not just alternate between the forward and backward search every other iteration. Instead it keeps executing the search with fewer nodes on the open set which favors the search in the sparser area. By doing this, the search fronts can meet faster (see Section 2.5 on page 8). For example the backward search shown in Figure 4.5a has been executed 423 077 times and the forward search only 127 371 times, for a total of 550 448. If both searches had been performed equally often, each of them would have been called 309 291 times, for a total of 618 582, which would be 12% less efficient.

The bidirectional search on the Greater Chicago Area, shown in Figure 4.6, shows a different behavior. Even though the forward search starts in a denser area at the Chicago Navy Pier, its search space is more or less halved by Lake Michigan which cuts off the network. This and the fact that the difference in density at the start and target node is not as severe as on the London network leads to a more balanced alternation. The forward search shown in Figure 4.6a has been executed 184 099 and the backward search 142 594 times, for a total of 326 693. With an equal distribution, both searches would have been called 166 539 times, for a total of 333 078, which is only a difference of 2%. These results reveal that the size of the open set is not only dependent on the density of the network but also where a search starts.

Searching by travel times (Figure 4.5b and 4.6b) shows the same effect already seen with the unidirectional version of Dijkstra's algorithm: the expansion happens more rapidly along roads with a higher speed limit. While this had no impact on the average number of settled nodes before, the bidirectional search does benefit from it. The reason for this can be seen by comparing the unidirectional (Figure 4.3b on page 46) with the

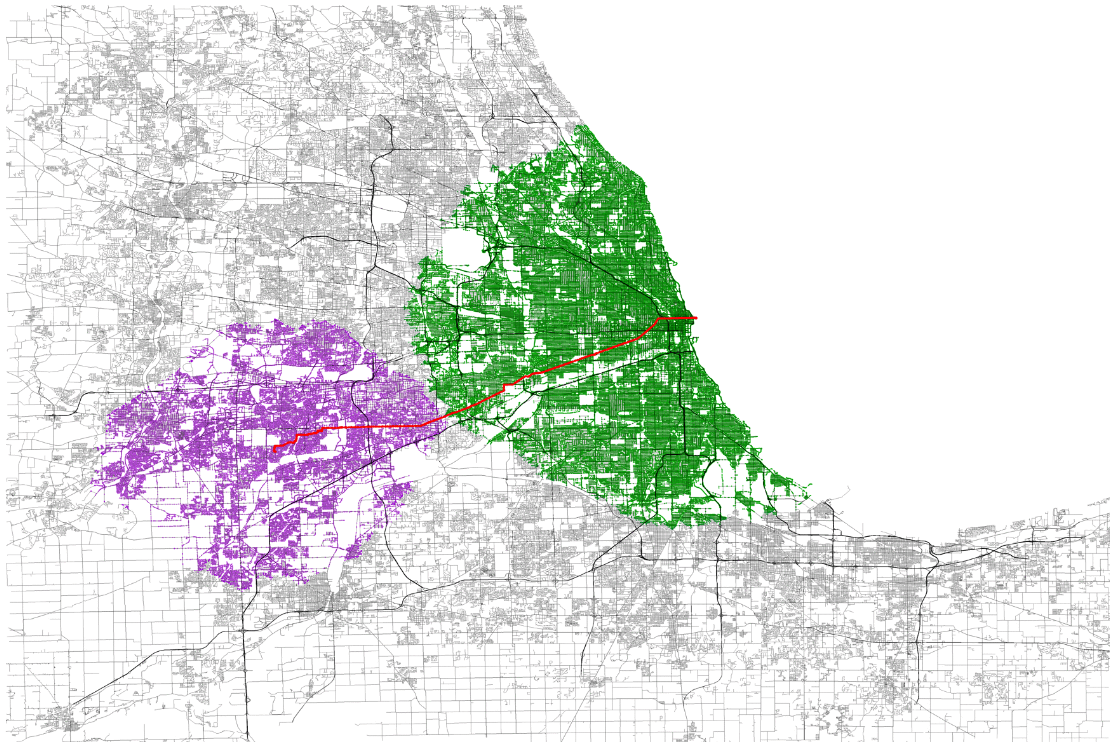


(a) Search on Greater London Area with **geometric distances**.

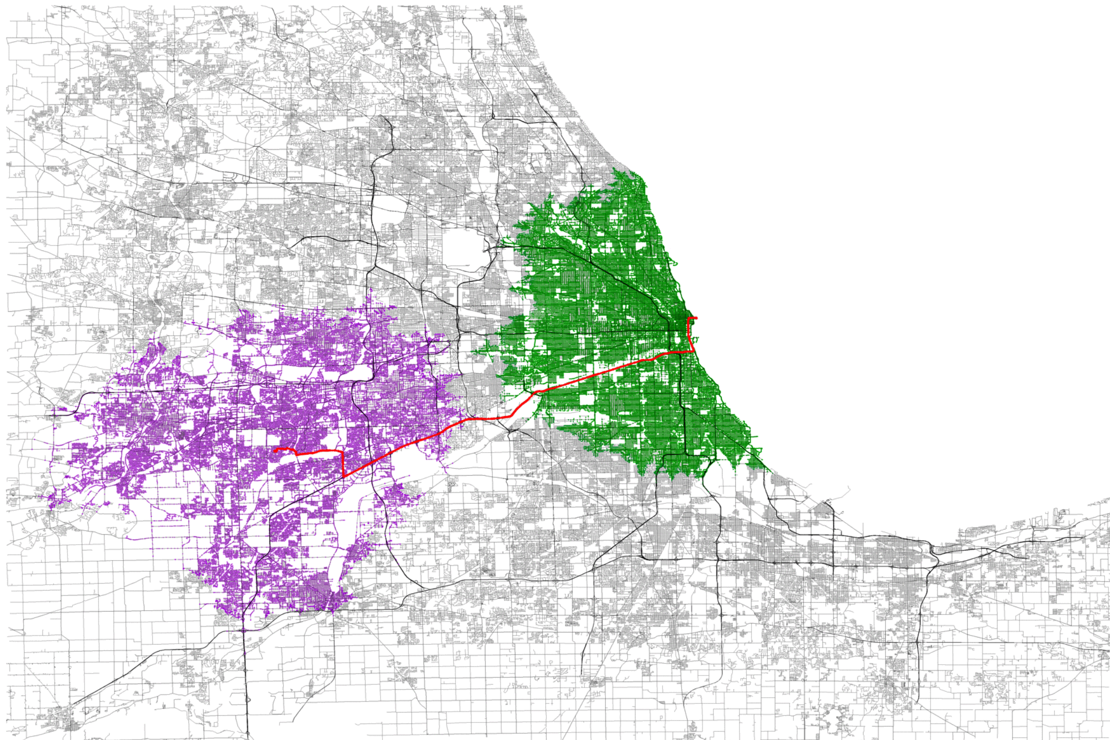


(b) Search on Greater London Area with **travel times**.

Figure 4.5: Bidirectional Dijkstra's search on Greater London Area (224 $\times$ 116 km) with geometric distances and travel times. Settled nodes are drawn in green for the forward search and in purple for the backward search. Nodes touched by both searches are drawn in blue. The shortest path is shown in red.



(a) Search on Greater Chicago Area with **geometric distances**.



(b) Search on Greater Chicago Area with **travel times**.

Figure 4.6: Bidirectional Dijkstra's search on Greater Chicago Area (160×107 km) with geometric distances and travel times. Settled nodes are drawn in green for the forward search and in purple for the backward search. Nodes touched by both searches are drawn in blue. The shortest path is shown in red.

bidirectional search (Figure 4.6b) on the Chicago network. Because the target node is not near a freeway or primary road, the unidirectional search expands past it for a significant amount. The bidirectional search solves this problem through the backward search which expands from the target node and creates a barrier for the forward search (and vice versa). This greatly reduces the chance that a search misses its target. Furthermore, as both searches extend faster along high speed roads, the search fronts may meet earlier on them, as can be seen in Figure 4.6b.

4.4.3 Results & Conclusion

Table 4.6 shows the average performance of a bidirectional Dijkstra's algorithm over all test cases, which is better in every aspect compared to the unidirectional version (Table 4.5 on page 47). Its node efficiency is almost twice as high and the average query takes 34% less time. While the unidirectional version had longer query times with travel times, the bidirectional one does benefit from the less compact search pattern. Fewer nodes are settled and query times are faster or similar compared to searches with geometric distances. The overhead of managing two searches instead of one only has a small impact. Comparing the average reduction in settled nodes and query times between the uni- and bidirectional version shows a performance loss below 8%. In conclusion, the bidirectional version should always be preferred over the unidirectional one.

Table 4.6: Average performance of a bidirectional Dijkstra's algorithm with geometric distances and travel times.

	node efficiency [%]		nodes settled		query time [ms]	
category	distance	time	distance	time	distance	time
C-NA	0.490	0.504	168 146	137 974	30.1	26.1
C-EU	0.993	1.001	77 524	62 739	13.0	11.2
C-OW	0.547	0.558	170 015	146 078	31.1	28.4
R-NA	0.111	0.085	1 975 105	1 815 230	423.1	427.8
R-EU	0.305	0.233	1 609 265	1 421 722	335.6	327.4
R-OW	0.113	0.051	1 853 564	1 804 496	392.6	413.6
Continental	0.083	0.073	4 937 186	4 692 264	1 178.1	1 243.2
all	0.561	0.559	662 339	599 815	142.3	142.4

4.5 A*, Bidirectional A*

The normal version of Dijkstra's algorithm, as presented before, does not take the target's position into consideration. This means that the search expands in all directions leading to many nodes being settled unnecessarily. A* tries to improve this by directing the search towards the goal which is done by utilizing a heuristic. The heuristic gives an estimate of the distance from the current node to the target. Different properties of the heuristic lead to different results, from fewer settled nodes to non-optimal results, as has been discussed in Section 2.6 on page 9. In this work, only the monotone and admissible heuristic based on the geometric distance will be used. The main problem with A* is finding a useful heuristic for a given metric, which is only trivial for geometric distances. A* can be executed as bidirectional search if the heuristic is consistent between the forward and backward search (see Section 2.7 on page 11).

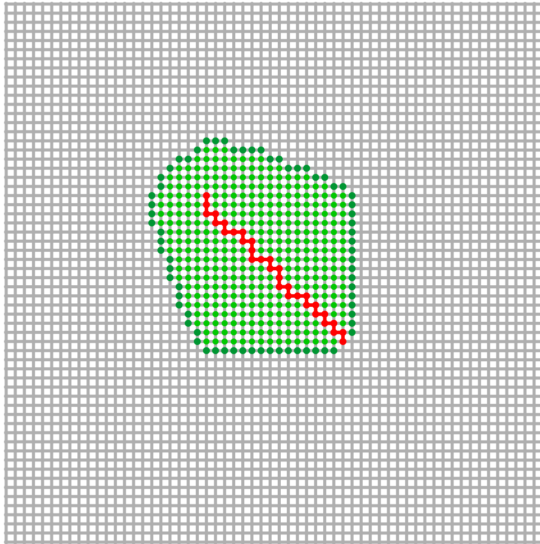
4.5.1 Synthetic Grids

In Figure 4.7, unidirectional and bidirectional searches with A* on synthetic grids are shown which demonstrate the advantage of a goal directed search compared to normal Dijkstra's algorithm, but also its problems. The unidirectional search on the uniform grid (Figure 4.7a) does not expand as much in the wrong directions near the start and target node and settles 4.55 times fewer nodes than the unidirectional and 2.45 times fewer nodes than the bidirectional version of Dijkstra's algorithm. The performance gains are more moderate on the graph with increasing edge weights (Figure 4.7b) and the problem with goal directed searches becomes visible. The used heuristic is based on the geometric distance and does not account for the geometry independent increases in edge weights which reduces its usefulness.

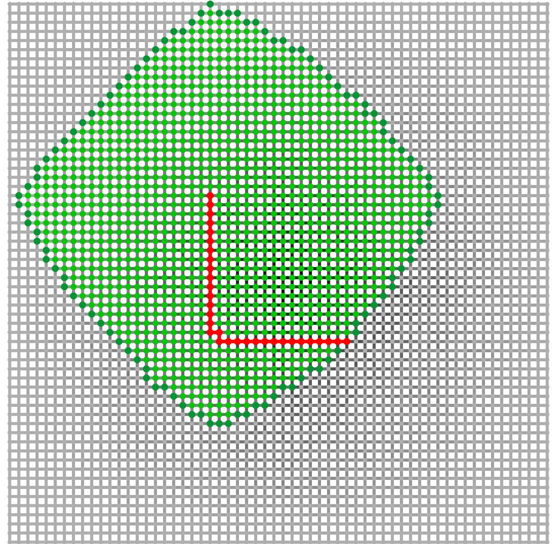
The bidirectional A* search is not the better choice on the uniform grid (Figure 4.7c) as it settles more nodes than the unidirectional version. On the graph with the increasing weights (Figure 4.7d), though, it outperforms all other algorithms used so far. Nonetheless, the same problem with the heuristic can be seen, leading to many nodes behind the start and target that are settled unnecessarily.

4.5.2 Road Networks

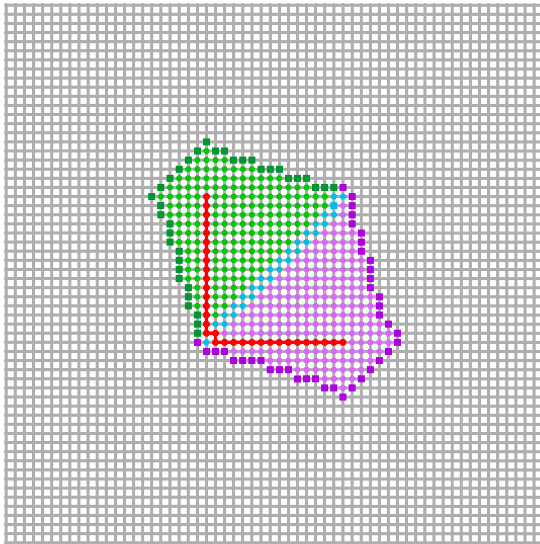
Unidirectional and bidirectional searches with A* and geometric distances on the Greater London and Chicago Area are shown in Figure 4.8. The networks used are the same as before but cropped and magnified to the affected area. The unidirectional search shows the typical drop shaped search pattern on both graphs (Figure 4.8a and 4.8c). The search, which starts on each network in the north-east, expands in a more circular fashion around the start but gets narrower towards the target. This pattern is a result of the accuracy of the heuristic which determines how much the search expands in the wrong directions. Because the heuristic underestimates the distance, the sum of the



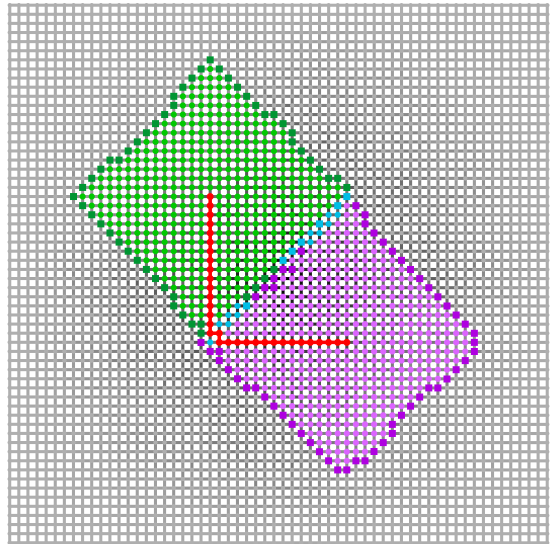
(a) Unidirectional A* on a uniform synthetic grid.



(b) Unidirectional A* on a synthetic grid with randomly increased weights (darker line shade) towards the graph center.



(c) Bidirectional A* on a uniform synthetic grid.

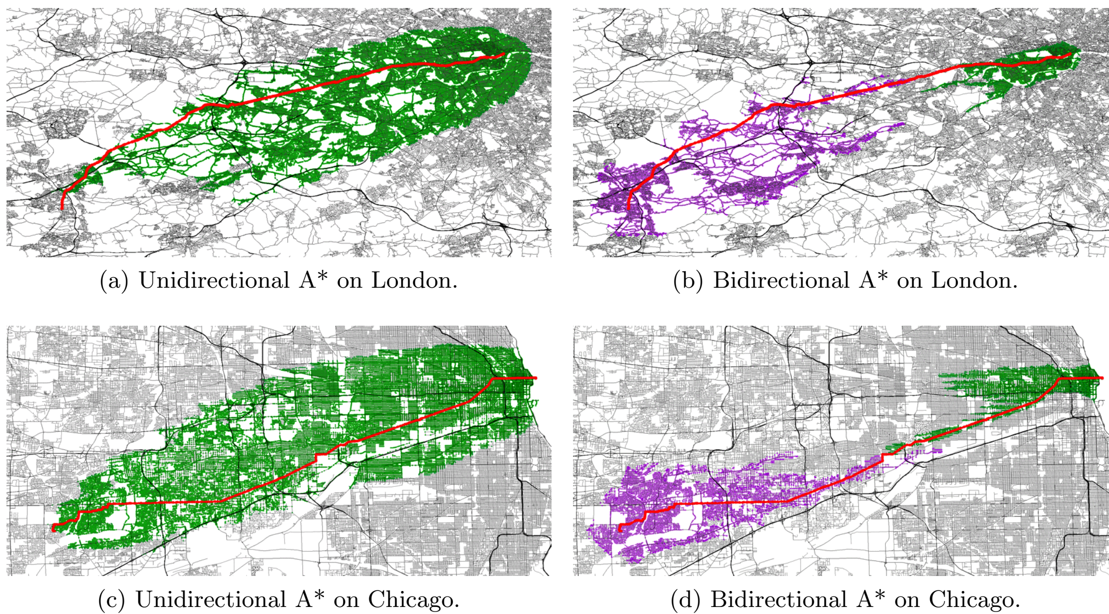


(d) Bidirectional A* on a synthetic grid with randomly increased weights (darker line shade) towards the graph center.

Figure 4.7: Unidirectional and Bidirectional A* search on synthetic grid graphs. The start is the upper left and the target the lower right end of the red line, which shows the shortest path found. Nodes in the closed set (settled nodes) are drawn in light green for the forward search and in light purple for the backward search. Nodes in the open set are drawn in dark green and dark purple for the forward and backward search, respectively. Nodes touched by both searches are drawn in blue.

actual distance from the start and the estimate of the remaining distance to the target gets larger towards the target compared to the area near the start.

The bidirectional search with A* (Figure 4.8b and 4.8d) works equally to the bidirectional search with Dijkstra's algorithm and finds the shortest path with fewer settled nodes. The drop shaped search pattern is not as distinct as with the unidirectional A* for two reasons: because the forward and backward search meet half-way the search fronts are not as narrow yet, resulting in a fringed cut, and the heuristic used with the bidirectional version is less accurate (see Section 2.7 on page 11), leading to more stray paths.



(a) Unidirectional A* on London.

(b) Bidirectional A* on London.

(c) Unidirectional A* on Chicago.

(d) Bidirectional A* on Chicago.

Figure 4.8: Unidirectional and Bidirectional A* search on the Greater London and Chicago Area with **geometric distances**. The network is cropped and magnified to the affected area. Settled nodes are drawn in green for the forward search and in purple for the backward search. Nodes touched by both searches are drawn in blue. The shortest path is shown in red.

While A* works well on road networks with geometric distances, no results will be given for searches with travel times because no trivial heuristic exists for it. It is possible to adapt the geometric heuristic by scaling it to the fastest road type but the results are poor.

The average performance over all test cases (see Table 4.7) reveals a problem. Even though 11% to 41% fewer nodes are settled with the bidirectional version, the query times do not improve equally and are even slower in some cases. While the bidirectional version of Dijkstra's algorithm saw only a small performance impact from the increased overhead of managing two searches, A* is affected much more. The reason for this is the heuristic, which requires the computation of the estimated distance for every node

that is added to the open set. This means the node has to be queried for its coordinates, which reduces the overall performance of A* and additionally impacts the bidirectional search, because of its lower locality.

4.5.3 Results & Conclusion

Table 4.7 shows that the A* algorithm significantly reduces the number of settled nodes compared to bidirectional Dijkstra's algorithm (Table 4.6 on page 52). The query times, however, do not decrease as much due to the increased number of computations and memory accesses caused by the heuristic. The bidirectional version executes only slightly faster and on several networks even slower than the unidirectional one. In case of the *Regions - Other World* category, A* is actually slower than bidirectional Dijkstra's algorithm despite 39% less nodes are settled and bidirectional A* is even slower. However, the main problem of A* is the lack of trivial heuristics for other metrics than geometric distances which limits its usage severely. Another disadvantage is the increased memory consumption to store each node's coordinates which may be a problem on computers with limited resources.

Table 4.7: Average performance of bidirectional Dijkstra's algorithm with geometric distances and travel times.

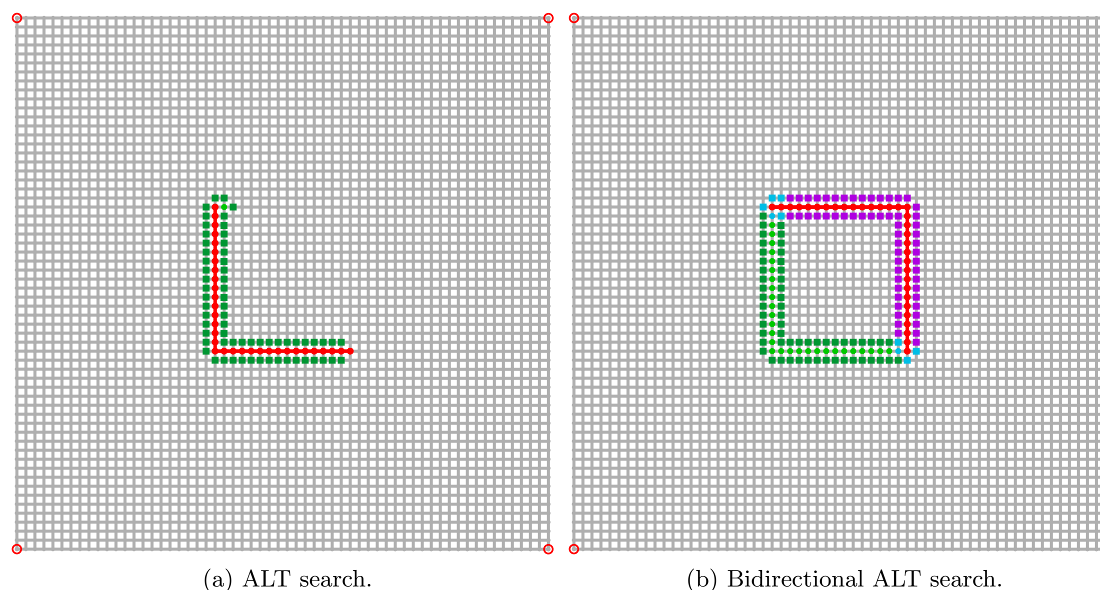
	node efficiency [%]		nodes settled		query time [ms]	
category	A*	Bidir. A*	A*	Bidir. A*	A*	Bidir A*
C-NA	1.195	1.800	70 189	46 195	23.9	21.5
C-EU	2.187	3.812	35 144	20 602	11.8	9.5
C-OW	1.082	1.718	76 187	47 713	26.6	22.2
R-NA	0.306	0.413	716 762	535 029	269.8	268.1
R-EU	0.668	0.880	648 071	498 364	244.7	253.0
R-OW	0.185	0.210	1 132 332	998 607	461.4	506.4
Continental	0.168	0.202	2 170 213	1 742 847	902.5	919.0
all	1.251	2.012	276 333	209 622	106.6	106.5

In conclusion, A* has a node efficiency that is two to four times better than bidirectional Dijkstra's algorithm on graphs with geometric distances. Its query times are lower in most cases and should further improve with an optimized implementation. As long as coordinates for nodes are available it is the recommended choice. For other metrics it should be compared to alternatives first.

4.6 A*, Landmarks, Triangle Inequality (ALT), Bidirectional ALT

ALT (A*, Landmarks, Triangle Inequality) can be seen as an evolution of the A* algorithm that solves its heuristic shortcomings. Through the application of the *triangle inequality* theorem and *landmarks*, ALT works with any metric without any further modifications. This is not possible with A*, which requires the implementation of a heuristic function for every metric. The drawbacks of ALT are that it requires preprocessing and the distances from and to landmarks have to be stored with every node, which increases the memory usage considerably. Also, after a change to edge weights, the landmark distances have to be recalculated. The performance of ALT relies strongly on the quality of the landmarks. For example, two landmarks, that are either identical with or behind the start and target nodes of a search, can be enough to achieve 100% node efficiency. Landmarks of lesser quality can be compensated to a certain degree by a higher quantity.

4.6.1 Synthetic Grids



(a) ALT search.

(b) Bidirectional ALT search.

Figure 4.9: ALT and Bidirectional ALT search on uniform synthetic grid graphs. The start is the upper left and the target the lower right end of the red line, which shows the shortest path found. Nodes in the closed set (settled nodes) are drawn in light green for the forward search and in light purple for the backward search. Nodes in the open set are drawn in dark green and dark purple for the forward and backward search, respectively. Nodes touched by both searches are drawn in blue. Landmarks are marked with red dots.

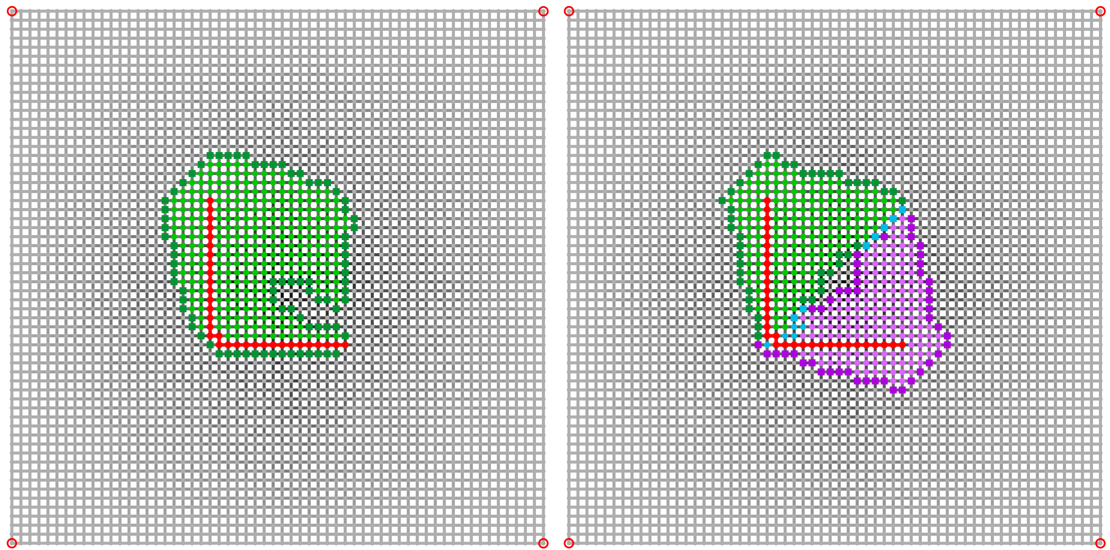
The ALT algorithm has no special preference for a certain graph type and can work exceptionally well with grid graphs, as can be seen in Figure 4.9a. With just four landmarks placed at the corners of the graph, the search achieves almost 100% node efficiency. Figure 4.9b illustrates an inherent problem of bidirectional algorithms using a heuristic. Usually, using a heuristic reduces the number of settled nodes by a significant amount because the search can, based on the extra information, follow possible best paths which also means the search front is much smaller. But a smaller search front also increases the chance that the forward and backward search miss each other, as the figure shows. The forward and backward search only meet each other the first time at their respective goals which means the bidirectional search was only half as efficient as the unidirectional one. This is an extreme case though and the bidirectional version performs on average two times better over all test cases (see Table 4.8).

Table 4.8: Average performance of ALT and Bidirectional ALT with 36 landmarks.

	node efficiency [%]		nodes settled		query time [ms]	
category	ALT	Bidir. ALT	ALT	Bidir. ALT	ALT	Bidir. ALT
C-NA	7.19	15.63	10 702	4 063	3.69	1.86
C-EU	12.42	22.43	5 091	2 543	1.75	1.12
C-OW	6.74	14.31	10 790	4 863	3.84	2.24
R-NA	1.63	4.84	114 773	37 796	41.08	18.39
R-EU	4.03	9.17	92 034	34 939	30.56	15.97
R-OW	1.19	4.51	125 807	32 666	53.54	18.34
Continental	1.05	2.96	325 373	115 606	120.27	59.10
all	7.38	14.98	40 776	14 971	14.46	7.21

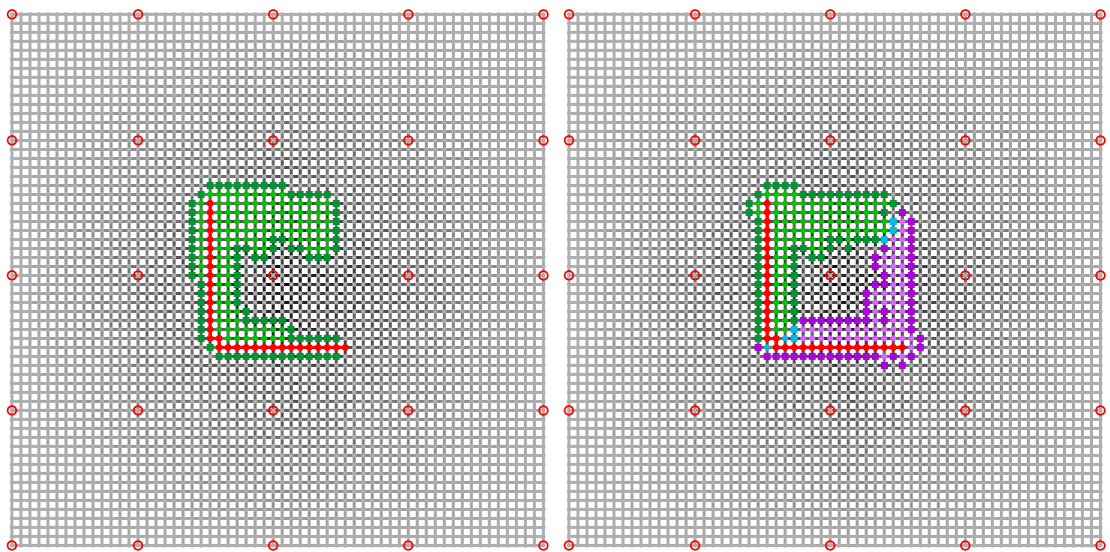
In Figure 4.10, a search with the uni- and bidirectional version is shown on a graph with randomly increasing edge weights towards the graph center. With only four landmarks (Figure 4.10a and 4.10b), much more nodes have to be settled and the performance drops significantly. The search patterns now look almost identically to the A* search on the uniform grid (Figure 4.7c on page 54). By increasing the number of landmarks by more than six times (Figure 4.10c and 4.10d), the number of settled nodes is halved and the search patterns starts resembling the one from the uniform grid. Because the positioning of the landmarks is not optimal for the given search, the algorithm has to visit many nodes for which the heuristic yields the same distance.

However, ALT can perform much better if the landmarks are in the right positions. In Figure 4.11, the start and target of the search query have been changed to line up with two landmarks. This brings the node efficiency up to 100% despite the random edge weights. The problem is that it is impossible to find a small set of landmarks that is optimal for every search query on most non-uniform road networks.



(a) ALT search with four landmarks.

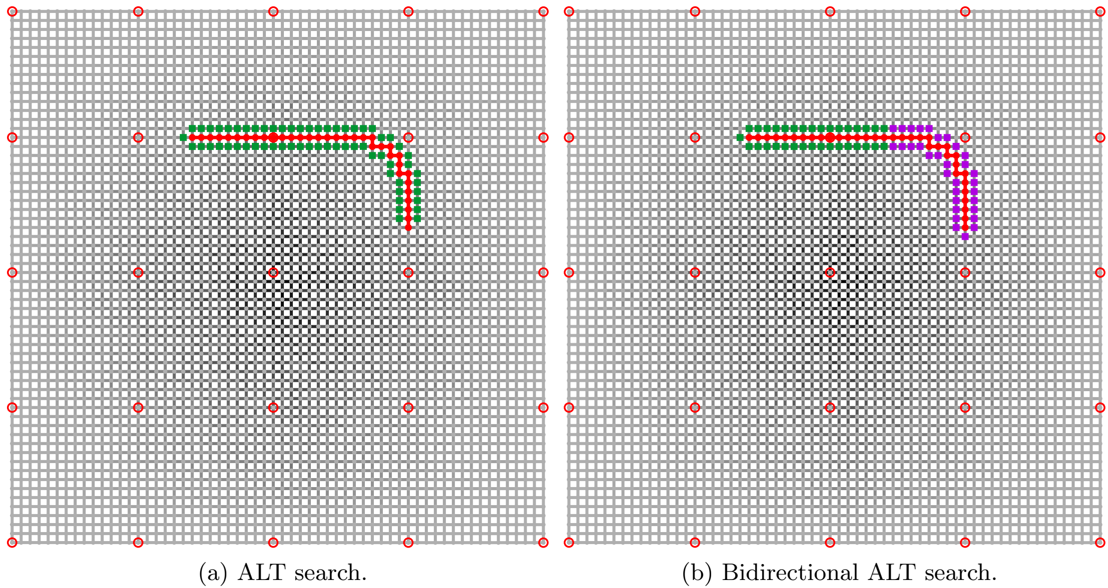
(b) Bidirectional ALT search with four landmarks.



(c) ALT search with 25 landmarks.

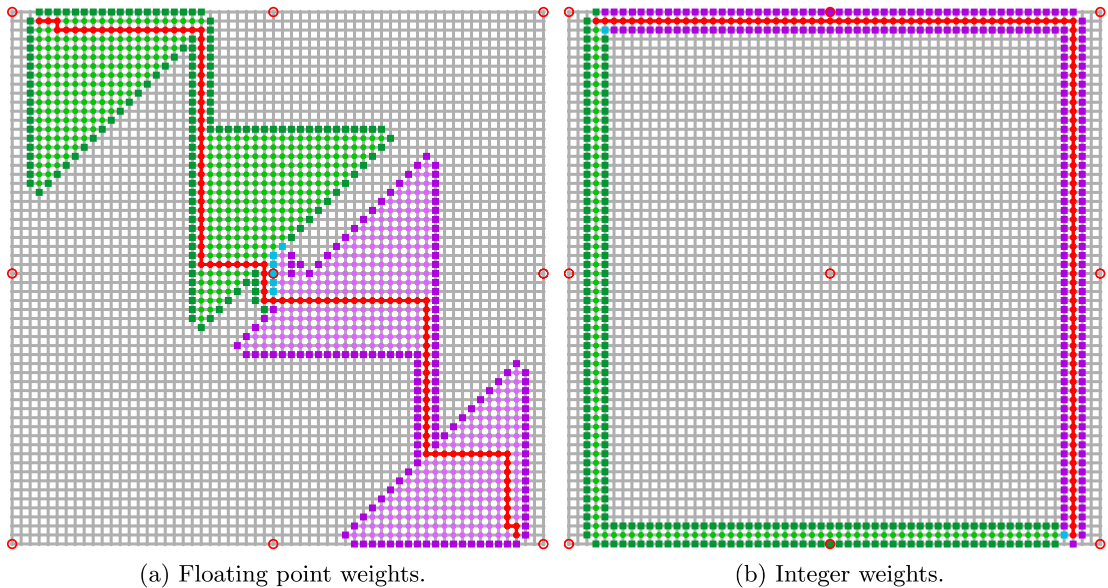
(d) Bidirectional ALT search with 25 landmarks.

Figure 4.10: ALT and Bidirectional ALT search with different landmark count on synthetic grids with randomly increased weights (darker line shade) towards the graph center. The start is the upper left and the target the lower right end of the red line, which shows the shortest path found. Nodes in the closed set (settled nodes) are drawn in light green for the forward search and in light purple for the backward search. Nodes in the open set are drawn in dark green and dark purple for the forward and backward search, respectively. Nodes touched by both searches are drawn in blue. Landmarks are marked with red dots.



(a) ALT search. (b) Bidirectional ALT search.

Figure 4.11: ALT and Bidirectional ALT search with the same setup as in Figure 4.10 but the start and target node of the query are changed to line up with landmarks. This brings the node efficiency up to 100% despite the random edge weights.



(a) Floating point weights. (b) Integer weights.

Figure 4.12: Bidirectional ALT search on a uniform synthetic grid with floating point and integer weights. Because the accuracy of floating point numbers is limited, small errors are introduced during distance calculations. These errors impact the search algorithm and lead to strange patterns and possibly false results (left figure). Switching to integer weights removes the errors (right figure).

Implementation Pitfalls

During the preparation of this chapter, a few graphs with strange search patterns showed up like the one in Figure 4.12a. Because the search is done on a uniform grid graph, the pattern should look like in Figure 4.12b, with the search following a direct path to the target. A thorough investigation of the framework's code revealed that the strange patterns are a result of the inherent accuracy problem of floating point numbers. The framework creates the synthetic grids with floating point weights which are used by the ALT algorithm to calculate the distances from and to landmarks. During a search, these distances are used by the heuristic function to calculate the estimates. Even though only addition and subtraction is used during these steps, small errors occur. Compared to the other algorithms, ALT is extremely sensitive to these errors. The problem has been fixed for this work by using integer weights for the uniform synthetic grid graphs, but this is not a solution if floating point weights have to be used.

4.6.2 Landmark Selection

Landmark selection is the key element in ALT's preprocessing duration and performance. Several different methods have been devised (Section 2.8.2 on page 14 gives an overview) but no optimal method exists because of the inherent trade off between time required for landmark selection and resulting query performance. As will be shown in the following sections, other algorithms with preprocessing such as CH and CRP have much better efficiency and query performance than ALT. This means that the preprocessing should be kept as short as possible to position ALT as feasible alternative.

In this work, the *Partition-Corners* algorithm from Efentakis et al. [17] will be used with some modifications. The basic idea of this algorithm is to partition the graph into k cells of about the same size (node wise) and select the four corner-most nodes of each cell as landmarks. Efentakis et al. are using *Buffoon* [55] to partition the graph which is similar to PUNCH and also works with natural cuts. The problem, though, is that partitioners based on natural cuts are slow and their specialty, finding cells with few boundary edges, serves no purpose for ALT. Instead, a general-purpose partitioner could be used, but these may not be fast enough on large graphs either. The issue with existing partitioners is that they create cells with certain features (e.g. connectivity, amount of boundary edges, etc.), which increases processing time, even though those features are not needed.

The only requirements for the partition are to have cells of the same size and compact shape containing neighboring nodes. To fulfill this requirement, the following simple method is proposed. Every node of a road network can be associated with x and y coordinates. Sorting nodes in a set by their x or y coordinate and then splitting the set in half creates two new sets according to the requirements. By repeating this process while alternating the sorting on the x and y coordinates, a set (graph) can be partitioned into

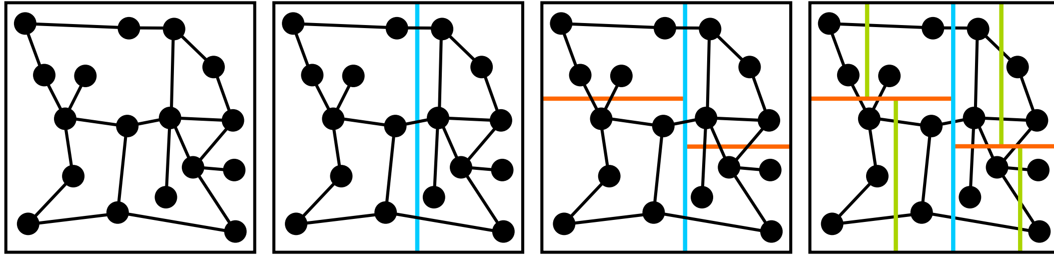


Figure 4.13: Partitioning a graph by coordinates. From left to right: The original graph is split in half based on the x coordinates (blue line). The two new cells are split in half again based on the y coordinates (orange lines). The four new cells are then split in half by the x coordinates (green lines). This process can be repeated until the cells have the desired size.

a given amount of cells of the same size with rectangular shape containing neighboring nodes (Figure 4.13). This simple method partitions the largest graph from the dataset (USA network) into 16 cells in less than 7 seconds. The required time can be further reduced by processing subsequent cells in parallel.

After the graph has been partitioned, the corner-most nodes of each cell are marked as landmark. This also means that several landmarks will be very close to each other, which is less useful for ALT's performance and should be avoided. Because Efentakis et al. do not give a solution to this problem in their paper, the following method will be used: During the distance calculation for a landmark l, the average distance from this landmark l to all other landmarks is recorded. If a landmark is less than a certain fraction of the average distance apart from l, it is removed.

Due to the removing of close landmarks, the final number can vary. To get a set amount of landmarks some have to be added or removed. To remove superfluous ones a list with the distances between every landmark is created. The landmark pair with the shortest distance is then taken from the list and from this pair the landmark with the shortest average distance to all other landmarks is removed. This process is repeated until the desired number of landmarks is left.

If additional landmarks are required, the center-most nodes of the densest cells are added. Because all cells contain the same amount of nodes and road networks are mostly planar, the density of a cell can be determined by the size of its area. If no center-most nodes are left, random nodes are chosen.

Algorithm Comparison

To assess the quality of the modified algorithm, it will be compared to randomly placed landmarks and the original Partition-Corners algorithm (using PUNCH instead of Buffoon, removing close landmarks and fixing the number of landmarks as described above). Because the randomly placed landmarks are different every time they are calculated, the comparison can only give an approximate overview.

Table 4.9 shows the increase in node efficiency between the modified algorithm and randomly placed landmarks. In most cases, the modified Partition-Corners algorithm works significantly better than randomly placed landmarks with an average increase in efficiency of 34%. There are a few networks from the *Cities - Other World* category though where it is worse. These networks have a small, dense center and are extremely sparse in the surroundings. While the random algorithm places most landmarks in the center, the modified one is "wasting" landmarks in the sparse areas. The original Partition-Corners algorithm suffers from the same problem on them.

Table 4.9: Comparison of the average node efficiency between the **modified Partition-Corners algorithm** and **randomly placed landmarks** with Bidirectional ALT and 36 landmarks. The numbers denote the efficiency increase in percent. Every column shows a different combination of the full and contracted networks with geometric distances and travel time metrics.

category	all [%]	full [%]	contr. [%]	all distance [%]	all time [%]	full distance [%]	full time [%]	contr. distance [%]	contr. time [%]
C-NA	35.7	37.0	34.5	41.1	30.1	42.4	31.8	40.0	28.5
C-EU	38.8	41.2	36.4	51.1	25.4	53.0	29.0	49.2	21.6
C-OW	17.9	26.0	8.8	27.2	8.6	40.2	13.4	14.4	2.5
R-NA	47.3	31.8	56.4	31.0	68.7	15.1	48.8	38.8	83.4
R-EU	38.7	35.2	43.3	35.7	42.8	35.0	35.5	36.6	54.6
R-OW	57.6	-	57.6	41.8	101.2	-	-	41.8	101.2
Continental	29.1	47.6	25.5	33.8	24.9	29.6	65.9	34.7	17.3
all	33.5	35.9	31.1	41.3	25.2	44.9	27.1	37.9	23.1

Table 4.10: Comparison of the average node efficiency between the **modified** and the **original Partition-Corners algorithm** with Bidirectional ALT and 36 landmarks. The numbers denote the efficiency increase in percent. Every column shows a different combination of the full and contracted networks with geometric distances and travel time metrics.

category	all [%]	full [%]	contr. [%]	all distance [%]	all time [%]	full distance [%]	full time [%]	contr. distance [%]	contr. time [%]
C-NA	14.0	12.7	15.1	14.9	12.9	14.1	11.4	15.6	14.5
C-EU	14.7	15.4	14.0	17.6	11.1	17.4	12.9	17.8	9.0
C-OW	3.2	4.3	1.8	5.1	1.0	6.0	2.4	4.1	-0.9
R-NA	9.6	4.5	12.2	7.7	11.5	-4.7	13.2	13.7	10.6
R-EU	13.5	14.5	12.3	13.0	14.2	14.7	14.3	11.1	14.1
R-OW	15.5	-	15.5	12.7	21.4	-	-	12.7	21.4
Continental	7.8	25.8	4.3	11.9	4.1	22.9	28.2	9.9	-0.8
all	11.9	11.9	11.9	13.7	9.8	13.5	10.2	13.9	9.3

In Table 4.10, the increase in node efficiency between the modified and the original Partition-Corners algorithm is shown. Even though both work after the same principal, the modified algorithm is better in most cases. Additionally, the preprocessing (including landmark distance calculation) of the modified algorithm only takes a few seconds more in the worst case compared to randomly placing landmarks, while the original version can take up to twice the time (i.e. several minutes) on larger graphs.

The performance gains with the unidirectional version of ALT are not as pronounced but similar. The results can be found in the appendix (Section B.1 on page 129, Table B.1, Table B.2).

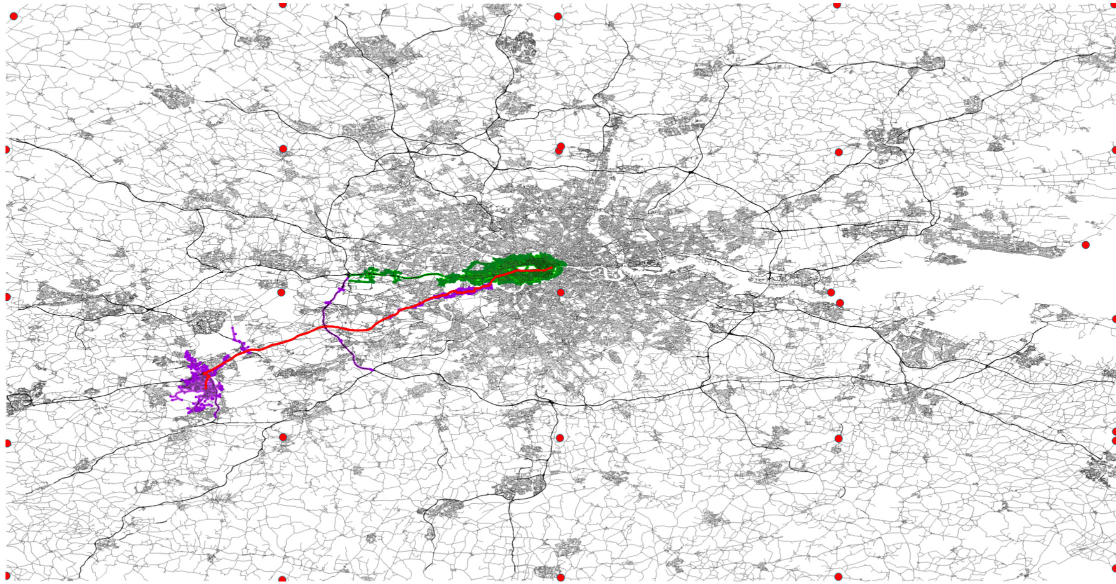
4.6.3 Search on Road Networks

Search queries with ALT and Bidirectional ALT on road networks are demonstrated on the Greater London Area in Figure 4.14 and Greater Chicago Area in Figure 4.15. Both figures consist of several sub-figures: the first sub-figure contains the complete network to show all landmarks, while the network is cropped and magnified to the area affected by the search in the other sub-figures, which illustrate all of the different search and metric combinations. The search starts in the city center in the east and ends in the suburbs in the west on all graphs.

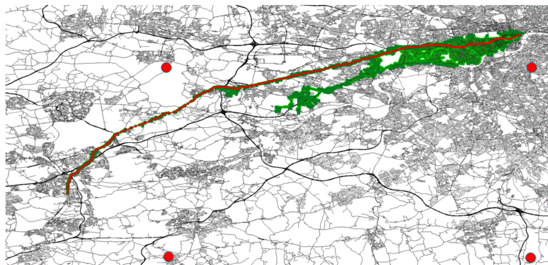
Using geometric distances on the Greater London Area, ALT performs much better than A*. While A* only knows the distance from the current node in a straight line to the target, the heuristic of ALT also contains information about the structure of the network ahead due to the landmarks. This means A* has to scan more nodes in the surroundings leading to the drop shaped search pattern (Figure 4.8 on page 55), while ALT can follow possible shortest paths more directly, which can be seen in Figure 4.14b and 4.14c. The backward search of the bidirectional search does exceptionally well, which also means that one of the landmarks is lined up (almost) perfectly.

In comparison, the performance is not as good with travel times (Figure 4.14d and 4.14e). More nodes are scanned near the start and target and the forward and backward searches branch out more, e.g. following several different motorway sections, even though the same amount of landmarks in the same locations are used. Generally, the unidirectional version of ALT performs worse with travel times than geometric distances. The results are not as conclusive for the bidirectional version, which takes a small hit on some graphs but performs equally or even better on others (see Table 4.11 on page 67).

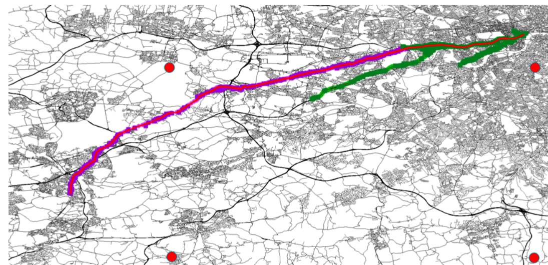
The search with geometric distances does not work as well on the Greater Chicago Area (Figure 4.15b and 4.15c). Even though ALT works great on the uniform grid graph with just four landmarks and the Chicago network is more grid like, the performance does not translate. The uni- and bidirectional searches follow several alternative paths and get stuck in the surroundings. The main problem is that there is no landmark behind (i.e. east of) the start and no distinct landmark behind (i.e. west of) the target, which is also the reoccurring issue with ALT. Because the road network of Chicago is



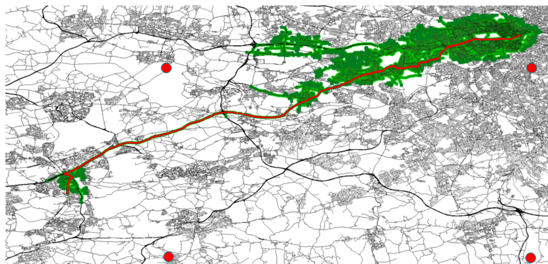
(a) Bidirectional ALT search with travel times on Greater London Area (224 × 116 km).



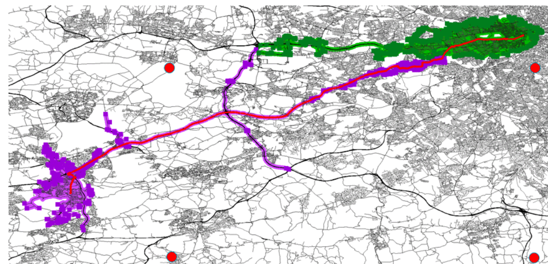
(b) ALT search with geometric distances.



(c) Bidir. ALT search with geometric distances.

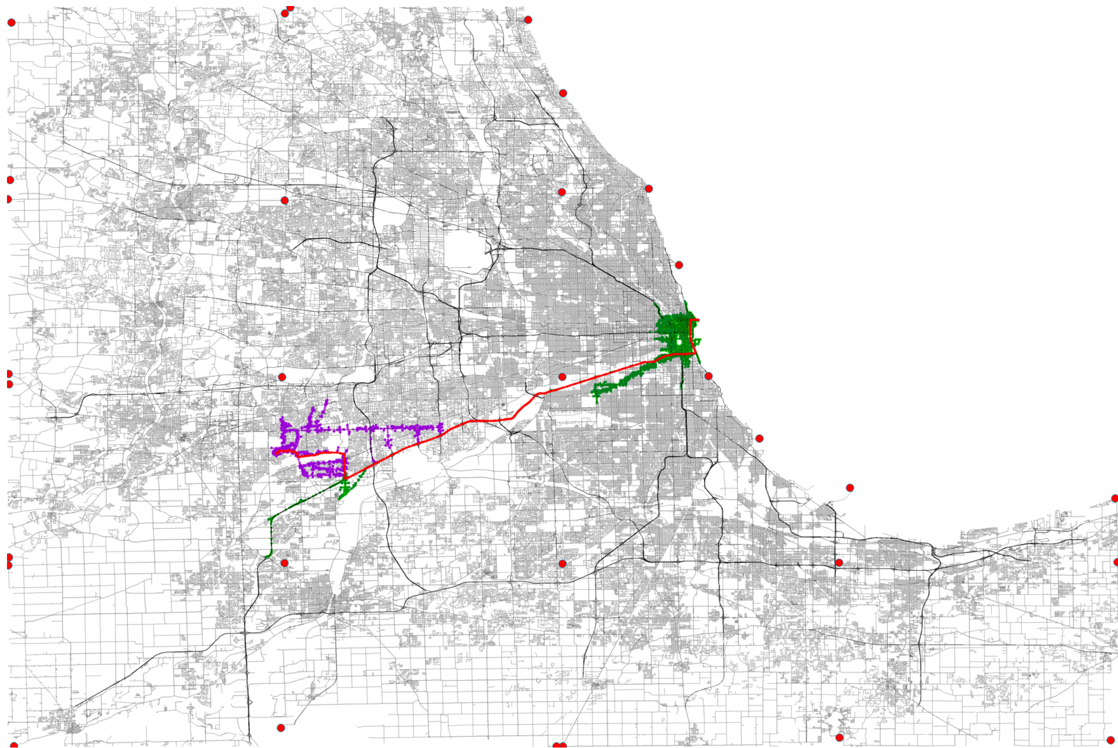


(d) ALT search with travel times.

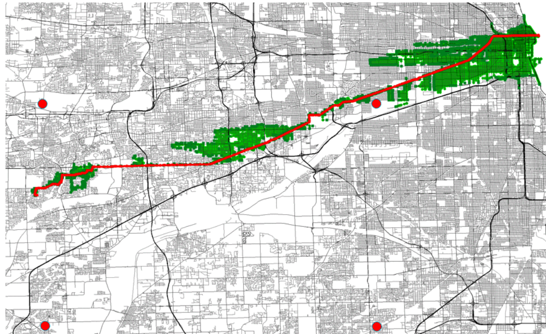


(e) Bidirectional ALT search with travel times.

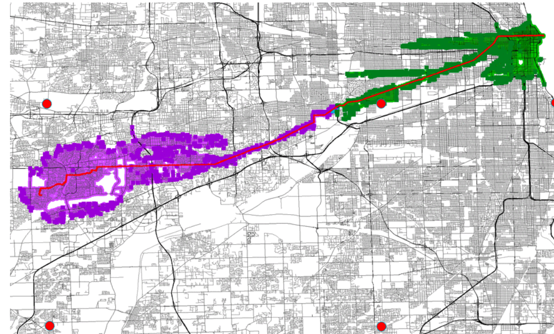
Figure 4.14: ALT search on Greater London Area. Settled nodes are drawn as circles in light green for the forward search and in light purple for the backward search. Nodes in the open set are drawn as squares in dark green and dark purple for the forward and backward search, respectively. Nodes touched by both searches are drawn in blue. The shortest path is shown in red. Landmarks are marked with red dots.



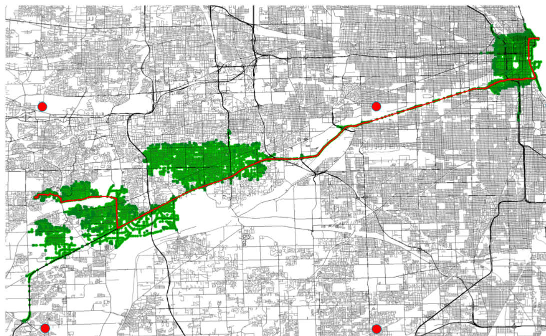
(a) Bidirectional ALT search with travel times on Greater Chicago Area (160 × 107 km).



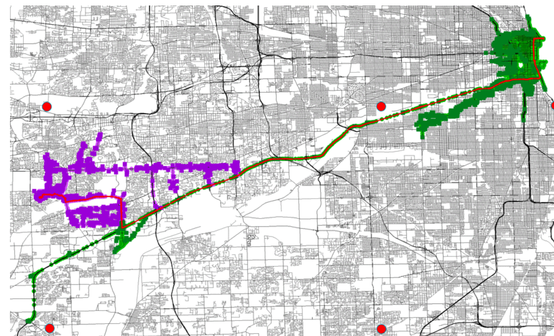
(b) ALT search with geometric distances.



(c) Bidir. ALT search with geometric distances.



(d) ALT search with travel times.



(e) Bidirectional ALT search with travel times.

Figure 4.15: ALT search on Greater Chicago Area. Settled nodes are drawn as circles in light green for the forward search and in light purple for the backward search. Nodes in the open set are drawn as squares in dark green and dark purple for the forward and backward search, respectively. Nodes touched by both searches are drawn in blue. The shortest path is shown in red. Landmarks are marked with red dots.

cut off at its densest part by Lake Michigan, it is impossible to set a few good landmarks for the large amount of endpoints, even if done by hand. The efficiency can still be increased by massively increasing the number of landmarks. For example using 224 instead of 26 landmarks, the node efficiency increases from 2.46% and 2.32% to 9% and 6.3% for the uni- and bidirectional version, respectively. However, the improvement is disproportionate to the extra cost in preprocessing and memory usage.

The search with travel times (Figure 4.15d and 4.15e) is plagued by similar problems. While the forward search has less problems leaving the city center, the uni- and bidirectional search struggle at the target, with the forward search overshooting the target by a large amount along the highway in the south-west. This is similar to the behavior that has already been seen on the London graph above.

Remark

The alert reader may have noticed that the search patterns of the forward search differ between the uni- and bidirectional searches in the figures. The reason for this is that a slightly different heuristic has to be used for a reliable termination of the bidirectional version, as has been explained in Section 2.7 on page 11.

4.6.4 Results & Conclusion

Table 4.11: Average performance of ALT and Bidirectional ALT with geometric distances and travel times with 36 landmarks.

	nodes settled				query time [ms]			
	ALT		Bidir. ALT		ALT		Bidir. ALT	
category	dist.	time	dist.	time	dist.	time	dist.	time
C-NA	10 314	11 090	4 329	3 797	3.53	3.84	1.98	1.74
C-EU	4 951	5 232	2 405	2 681	1.70	1.80	1.07	1.17
C-OW	10 348	11 233	4 846	4 879	3.68	3.99	2.24	2.25
R-NA	106 890	122 655	44 426	31 166	36.13	46.04	20.91	15.88
R-EU	85 597	98 472	39 309	30 569	27.68	33.44	17.94	14.00
R-OW	128 831	122 782	35 171	30 160	54.10	52.97	19.67	17.01
Continental	301 754	348 991	131 741	99 471	111.81	128.73	67.17	51.04
all	38 210	43 341	16 708	13 235	13.33	15.59	8.01	6.41

The results over all test cases (see Table 4.8 on page 58 and Table 4.11) show that the bidirectional version of ALT is on average two times faster and has a smaller performance gap between geometric distances and travel times metrics than the unidirectional version. While the bidirectional version of A* struggled to outperform the unidirectional one, this is not the case with ALT. Even though ALT is subject to a similar performance impact because for every node added to the open set the distance from and to landmarks has to

be retrieved, by switching from the uni- to the bidirectional version significantly fewer nodes have to be settled with ALT than it was the case with A*. This reduces the query times enough to be faster on almost all test cases and only on very small networks the benefits of the bidirectional version diminish.

The performance of ALT not only depends on the number of landmarks, but also on the size of the network which can be seen by comparing the efficiency in Table 4.8 on page 58 with the mean number of nodes in Table 3.3 on page 36. This is also the main cause why ALT performs so much better on European cities, which have only half as many nodes as American cities.

Even though the number of nodes is similar, European regions fare better than North American. The reason for this is that the European category contains more networks and some of them such as Austria, Cyprus and Poland achieve high efficiencies, but Europe and North America have the same overall trend/results for the majority of the networks.

The efficiency of ALT on contracted networks compared to their full versions gives a mixed picture (Table 4.12). While slightly better on the contracted networks of North American cities and about the same for European cities it is notable worse on the other contracted networks.

Table 4.12: Average node efficiency of ALT and Bidirectional ALT on full and contracted networks with 36 landmarks. The Continental and R-OW categories are not included because they consist of contracted networks only.

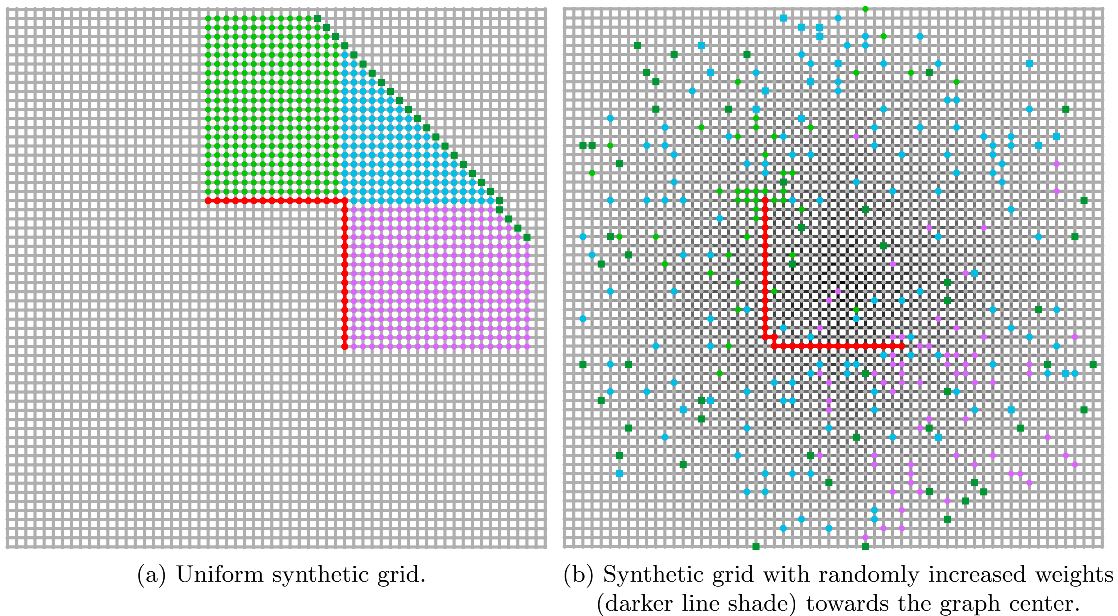
	ALT			Bidirectional ALT		
category	full [%]	contr. [%]	ratio [%]	full [%]	contr. [%]	ratio [%]
C-NA	6.46	7.93	123	15.12	16.14	107
C-EU	12.13	12.71	105	22.95	21.90	95
C-OW	7.20	6.27	87	16.15	12.47	77
R-NA	1.96	1.52	77	6.40	4.31	67
R-EU	4.68	3.45	74	10.80	7.72	71
all	7.71	7.09	92	16.45	13.72	83

In summary, the performance of bidirectional ALT is significantly better compared to the other algorithms presented so far, even though only a simple method for landmark selection was used. The only disadvantage, beside the preprocessing phase, is the increased memory consumption to store the distances from and to landmarks.

4.7 Contraction Hierarchies (CH)

The Contraction Hierarchies algorithm reduces the search space for queries by applying two concepts, Hierarchies and Shortcuts, during a preprocessing phase. Nodes are ordered and contracted based on their *priority terms*, which is basically the order of their importance as part of shortest paths. Shortcuts are added to retain shortest paths throughout the hierarchy (see Section 2.9 on page 15). One of the advantages of CH compared to other algorithms such as ALT or CRP is, that it does not require additional data structures because it only adds additional edges to the existing graph. The only modification required is an additional field at edges to store the middle node of shortcuts for path unpacking. This also means that the graph generated can be used as a base by other algorithms for further processing or search queries. The main disadvantage of CH is that the metric is required during preprocessing to create useful shortcuts. It is possible to keep the initial node ordering from a different metric and only repeat the hierarchy creation, but this has a severe impact on preprocessing and query times [22, Chapter 5.4.6]. Consequently, the preprocessing has to be repeated for every metric.

4.7.1 Synthetic Grids



(a) Uniform synthetic grid.

(b) Synthetic grid with randomly increased weights (darker line shade) towards the graph center.

Figure 4.16: CH search on synthetic grid graphs. The start is the upper left and the target the lower right end of the red line, which shows the shortest path found. Nodes in the closed set (settled nodes) are drawn in light green for the forward search and in light purple for the backward search. Nodes in the open set are drawn in dark green and dark purple for the forward and backward search, respectively. Nodes touched by both searches are drawn in blue.

The version of the algorithm proposed by Geisberger et al. [22] is not designed for uniform grids and thus performs poor on them. Uniform grid graphs contain no hierarchy and every node has the same importance. Only the degree of the nodes at the edges of the graph differs, with the corner nodes having the lowest degree. This means the algorithm keeps processing nodes at the corners of the graph. This has two effects: first of all, no shortcuts are created, because the path via the removed corner node is not shorter than the one through the opposite node (see Figure 4.17). Second, the created node order/hierarchy is worthless to improve search queries. Figure 4.16a shows a search with CH on a uniform grid graph. Without shortcuts, the search has to follow the grid. The search pattern reflects the order in which the nodes have been preprocessed, in this case from the lower left to the upper right (the forward/backward search only processes edges to/from nodes with a higher level in the hierarchy, respectively). CH can perform better on uniform grid graphs, though, by modifying the preprocessing to also contract nodes inside the graph, not only on the edges [61].

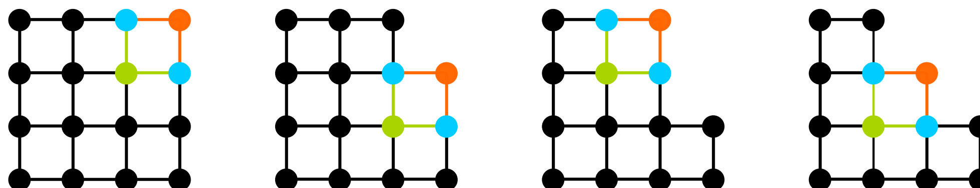


Figure 4.17: Contracting corner nodes. The orange node is contracted. The path between the two blue nodes via the orange node has the same length as the path via the opposite node in green and no shortcut has to be created. From left to right: the predicate holds for repeated contraction of corner nodes.

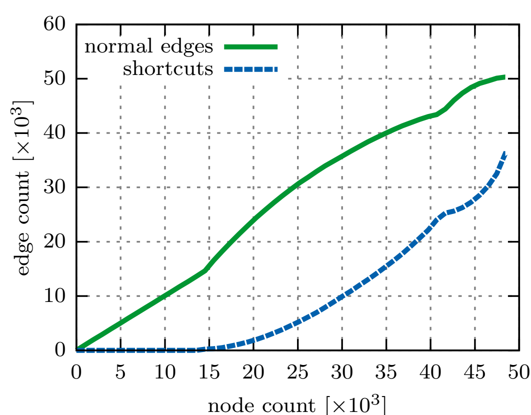
By introducing varying edge weights, the graph obtains structuring which can be used as hierarchy for CH. The preprocessing now adds shortcuts and the nodes are ordered in a hierarchical way. The typical search pattern, as shown in Figure 4.16b, emerges. The search is not bound to follow the grid in the original graph anymore, but can use shortcuts (which are not shown in the figure) to work its way up in the hierarchy to more important nodes until the forward and backward searches meet.

4.7.2 Preprocessing on Road Networks

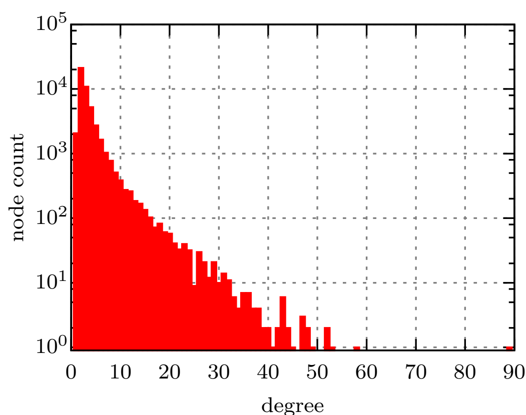
To better illustrate how nodes are ordered and shortcuts are created, the road network of Graz, a city in Austria, with travel times will be inspected. Graz has been chosen because a visual inspection of the shortcuts is much easier on a small graph. The full graph (48 437 nodes, 50 311 edges) and the contracted graph (7 241 nodes, 9 115 edges) will be used to see if they behave differently. As a reminder, paths consisting of neighboring nodes with degree two are contracted into a single node in the contracted graph. The degree distribution prior to preprocessing is shown in Table 4.13.

Table 4.13: Degree distribution for the road network of Graz.

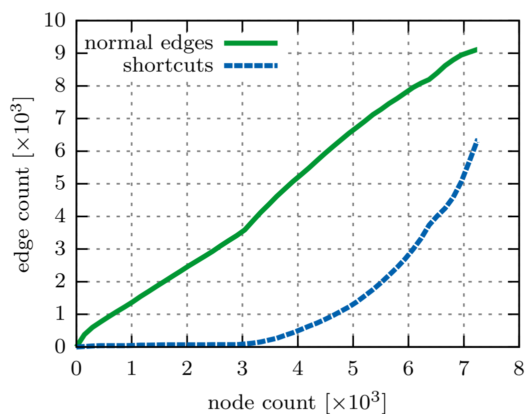
	full		contracted	
degree	nodes	[%]	nodes	[%]
1	2 067	4.3	2 009	27.7
2	41 277	85.2	195	2.7
3	4 402	9.1	4 348	60.0
4	661	1.4	659	9.1
5	29	0.1	29	0.4
6	1	0.0	1	0.0



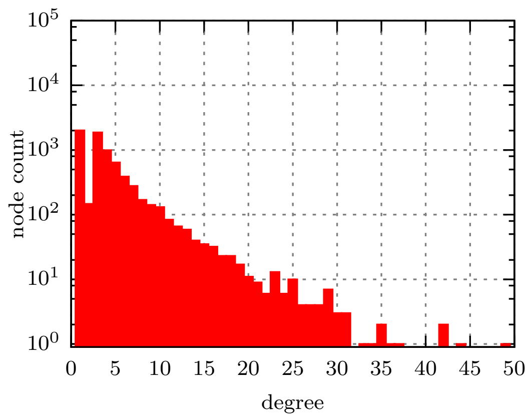
(a) Edge distribution (full network).



(b) Degree distribution (full network).



(c) Edge distribution (contracted network).



(d) Degree distribution (contracted network).

Figure 4.18: Edge and degree distribution for the road network of Graz with travel time metric. Left: Number of undirected edges for the given number of nodes, sorted by hierarchy level in ascending order. Right: Number of nodes with the given degree, including shortcuts.

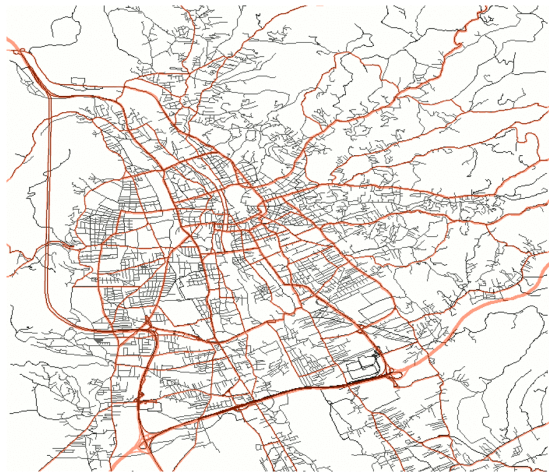
Applying the CH preprocessing to the full graph creates 36 228 shortcuts (72% increase in edges) and 6320 shortcuts (69% increase in edges) with the contracted graph.

In Figure 4.18, the edge and degree distribution after preprocessing is shown. The number of normal edges and shortcuts for the given number of nodes in 4.18a and 4.18c is sorted by the hierarchy level (in CH, every node has its own level, which means there are as many levels as nodes). The results for the full and contracted graph are similar. No shortcuts are created for the first 30-40% of levels, because unimportant nodes are contracted first. These are mostly leaf nodes and intermediate nodes from access roads (e.g. the contracted graph has 27.7% of all nodes as endpoints), which are similar to the corner nodes in the uniform grid graph. As the algorithm starts processing more important nodes, the number of shortcuts starts growing polynomial while the number of normal edges keeps growing linear. This is also the reason why the preprocessing phase is fast at the beginning but gets increasingly slower at the end. With every new shortcut, the graph gets denser and better connected, which increases the time required for *witness path* (shortest path) searches during contraction.

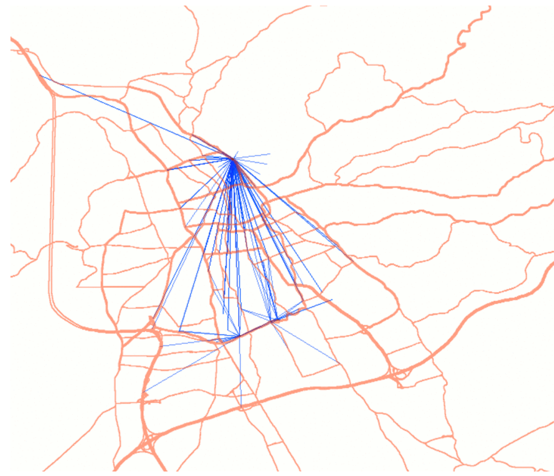
The degree distribution after preprocessing (Figure 4.18b and 4.18d) resembles a *power-law* distribution. By extending the graph with hierarchy depending shortcuts, it changes from a random to a *scale-free* network with high degree hubs. This enables search queries to take advantage of the *small-world* effect and find shortest paths with significantly fewer settled nodes. (For a detailed discussion of *power-law* distribution, *scale-free* and *small-world* networks, see [45].)

Figure 4.19 illustrates various hierarchy level ranges of the graph. All edges incident to the nodes from the specified hierarchy levels are shown, including those from other (lower) levels. For better orientation, major roads are drawn additionally in orange as overlay, but the overlay is not part of the respective graphs.

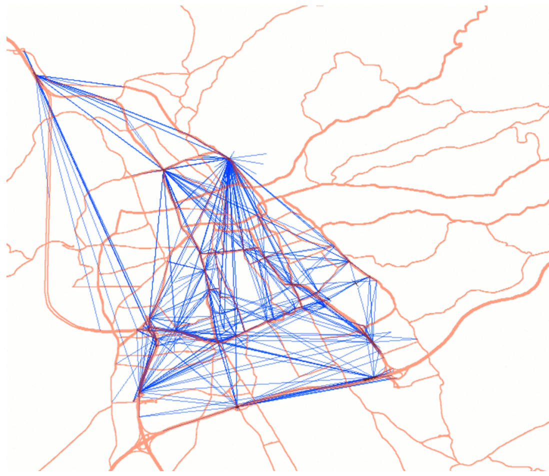
- Figure 4.19a: The initial road network. The area to the north and east is predominantly residential. The white spots in the south are the airport and wetlands around the River Mur. The city is confined by the Plabutsch mountain in the west.
- Figure 4.19b: The nodes and incident edges of the top 5 levels. Even though the top 5 levels consist of only 5 nodes and 17 normal edges, they are connected to many important hubs at the city center by 186 shortcuts.
- Figure 4.19c: The city center is already well connected by 1 096 shortcuts with just 50 levels.
- Figure 4.19d: Within the top 250 levels, the network spans the whole city with 4 087 shortcuts.
- Figure 4.19e: Reaching 10 000 levels, most of the major road segments are part of the graph. All hubs are connected and shortcuts for side roads in the outskirts show up. Only 32% of the shortcuts are left for the remaining 38 437 levels.
- Figure 4.19f: The full network with all shortcuts.



(a) Full network without shortcuts.

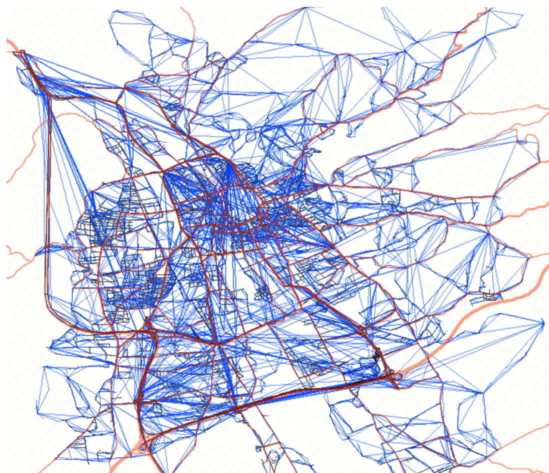


(b) Top 5 levels.

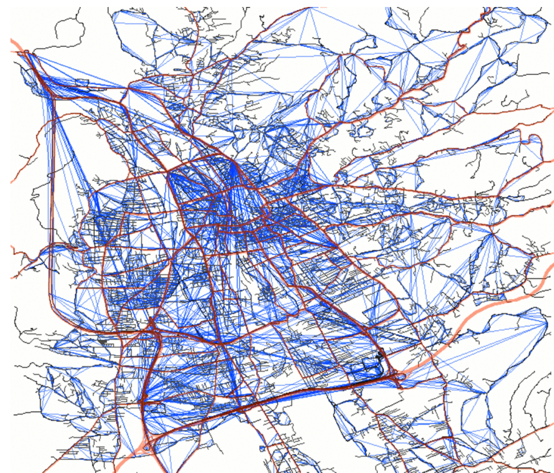


(c) Top 50 levels.

(d) Top 250 levels.



(e) Top 10 000 levels.



(f) Full network with all shortcuts.

Figure 4.19: CH network layout for various hierarchy level ranges for the road network of Graz (19 × 17 km). Edges from the original graph are drawn in black and shortcuts created by CH in blue. For better orientation, major roads are drawn additionally in orange as overlay, but the overlay is not part of the respective graphs.

Number of Shortcuts

In Figure 4.20, the number of shortcuts for all networks is shown, separated by network type and metrics. How pronounced the hierarchy of a network is depends on the metric used. As can be seen in the figure, fewer shortcuts are created with travel times which means the hierarchical structure is stronger. The reason for this is simple: shortest paths with travel times often run along roads with a higher speed limit. These roads are less often (over all networks, only 20% of the edges are motorways or primary roads) which means shortest paths often use the same edges. In comparison, using geometric distances leads to straighter paths utilizing more secondary roads which means the edges in a graph are used more evenly. The impact of this is significant with up to twice as many shortcuts created with geometric distances.

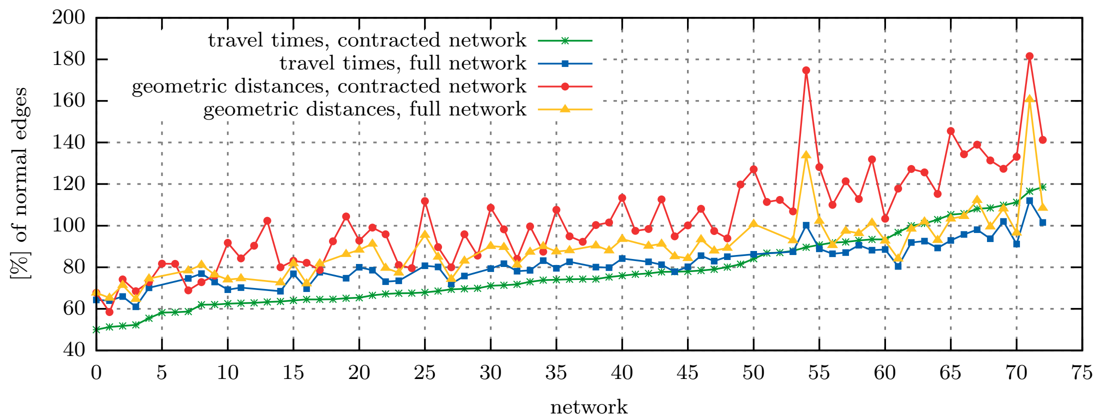
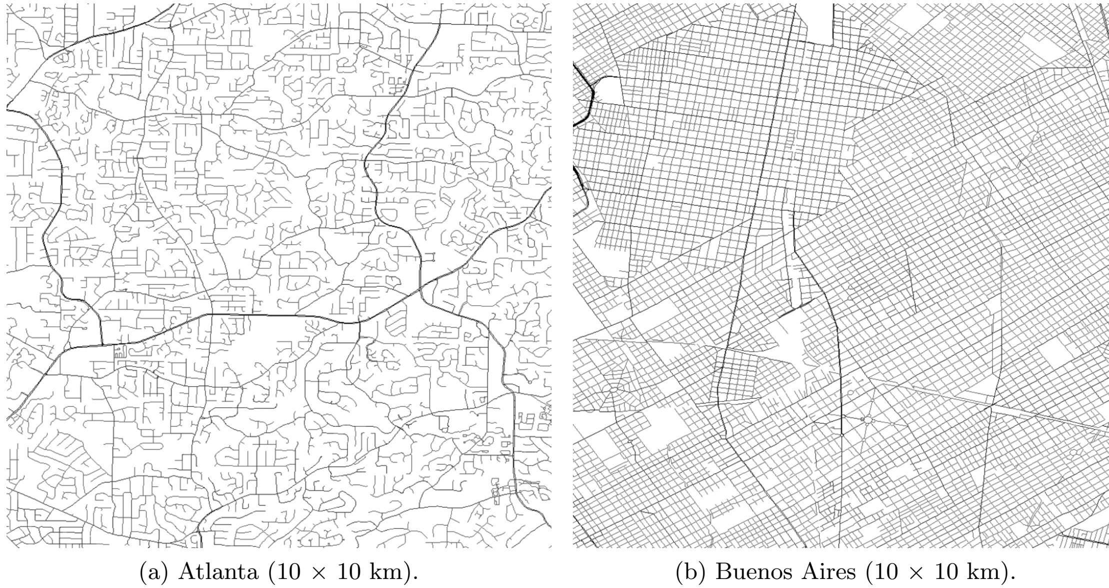


Figure 4.20: Number of shortcuts for all networks shown as percentage of the normal edge count. Sorted by the percentage of shortcuts with travel times on contracted networks in ascending order.

The figure also shows that switching between full and contracted networks does influence the percentage of created shortcuts to a varying degree. While using the contracted network is beneficial in many cases with travel times, the opposite is true with geometric distances.

How many shortcuts are created depends not only on the used metric but also how the graph is connected. Over all networks, the fewest shortcuts are created for the contracted network of Atlanta (50% increase in edges) and the most shortcuts are created for the contracted network of Buenos Aires (182% increase in edges). A section of both networks is shown in Figure 4.21. The network of Atlanta consists of many dead end roads (31% of all nodes have degree one) which do not contribute to the creation of shortcuts, as has been shown above on the network of Graz. Buenos Aires in contrast has a dominant grid layout (only 5% of all nodes have degree one). One may expect that a grid layout creates fewer shortcuts, as was the case with the uniform synthetic grid. But because the grid layout of Buenos Aires is not uniform and the network is traversed by many



(a) Atlanta (10 × 10 km).

(b) Buenos Aires (10 × 10 km).

Figure 4.21: Road network section of Atlanta and Buenos Aires.

arterial roads, a large number of shortcuts are created.

On average, 82% of the normal edge count are added as shortcuts. Generally, it is not trivial to predict how many shortcuts are created, because the *priority terms*, which are used to determine the order in which nodes are contracted, consist of several factors.

Preprocessing Time

The following figures and numbers are based on the sequential basic algorithm. Preprocessing times can be reduced by applying various optimizations (see the original paper on CH [23]) and by parallelizing (see [36, 67]).

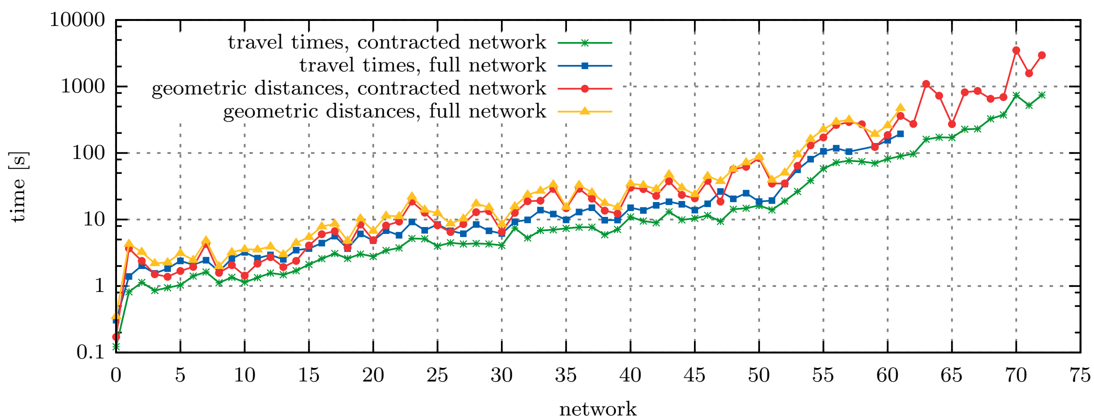


Figure 4.22: Time required for preprocessing for all networks, shown in seconds on a logarithmic scale. Sorted by the contracted networks' normal edge count in ascending order.

Figure 4.22 shows the total time required for preprocessing for all networks which is only a few seconds for small and less than a minute for medium sized graphs. Preprocessing on large graphs can take up to an hour, though. With travel times, significantly less time is required compared to geometric distances. For example the contracted network of North America requires 13 minutes with travel times and 59 minutes with geometric distances.

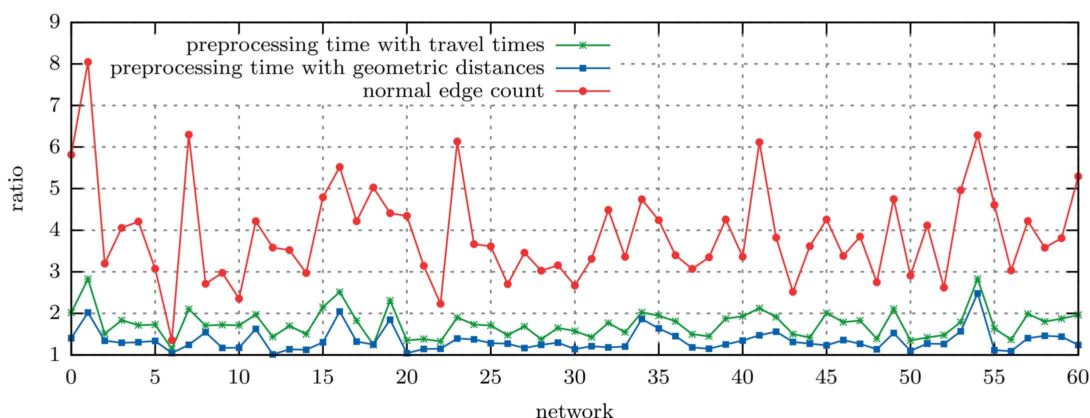


Figure 4.23: Ratios between the full and contracted version of a network for preprocessing time with travel times, preprocessing time with geometric distances and normal edge count. Sorted by the contracted networks' normal edge count in ascending order.

The ratios between full and contracted networks for the number of normal edges, preprocessing time with travel times and preprocessing time with geometric distances are shown in Figure 4.23. Most full networks have more than three times as many edges than their contracted version. Preprocessing times do not scale to the same extend, however. Switching from the full to the contracted network reduces the required average time by 43% with travel times and only by 20% with geometric distances.

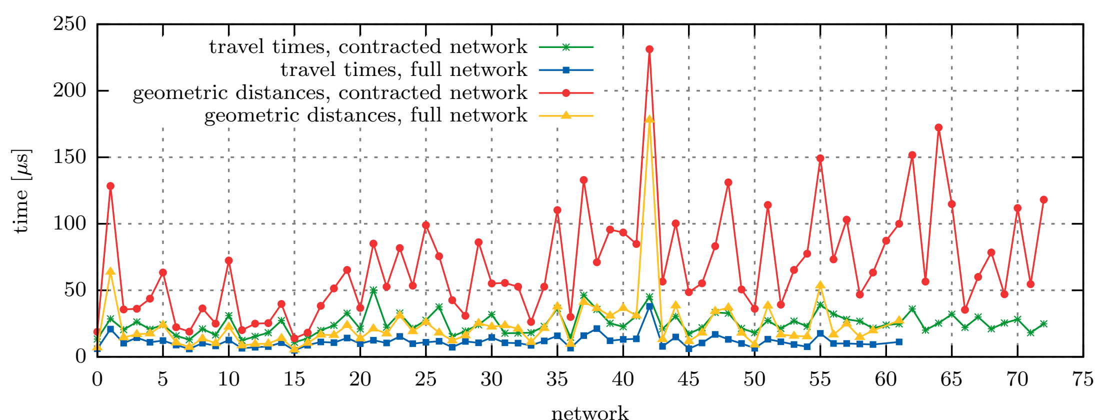


Figure 4.24: Average time required for preprocessing per normal edge for all networks. Sorted by the contracted networks' normal edge count in ascending order.

The average time required per normal edge, which is shown in Figure 4.24, does not increase with the size of the graph and is more stable/similar on full networks. Because the total preprocessing time improves only slightly by switching between the full and contracted network with geometric distances, the time spent per edge is noticeable worse with it. The outlier in the center is the network of Buenos Aires.

4.7.3 Search on Road Networks

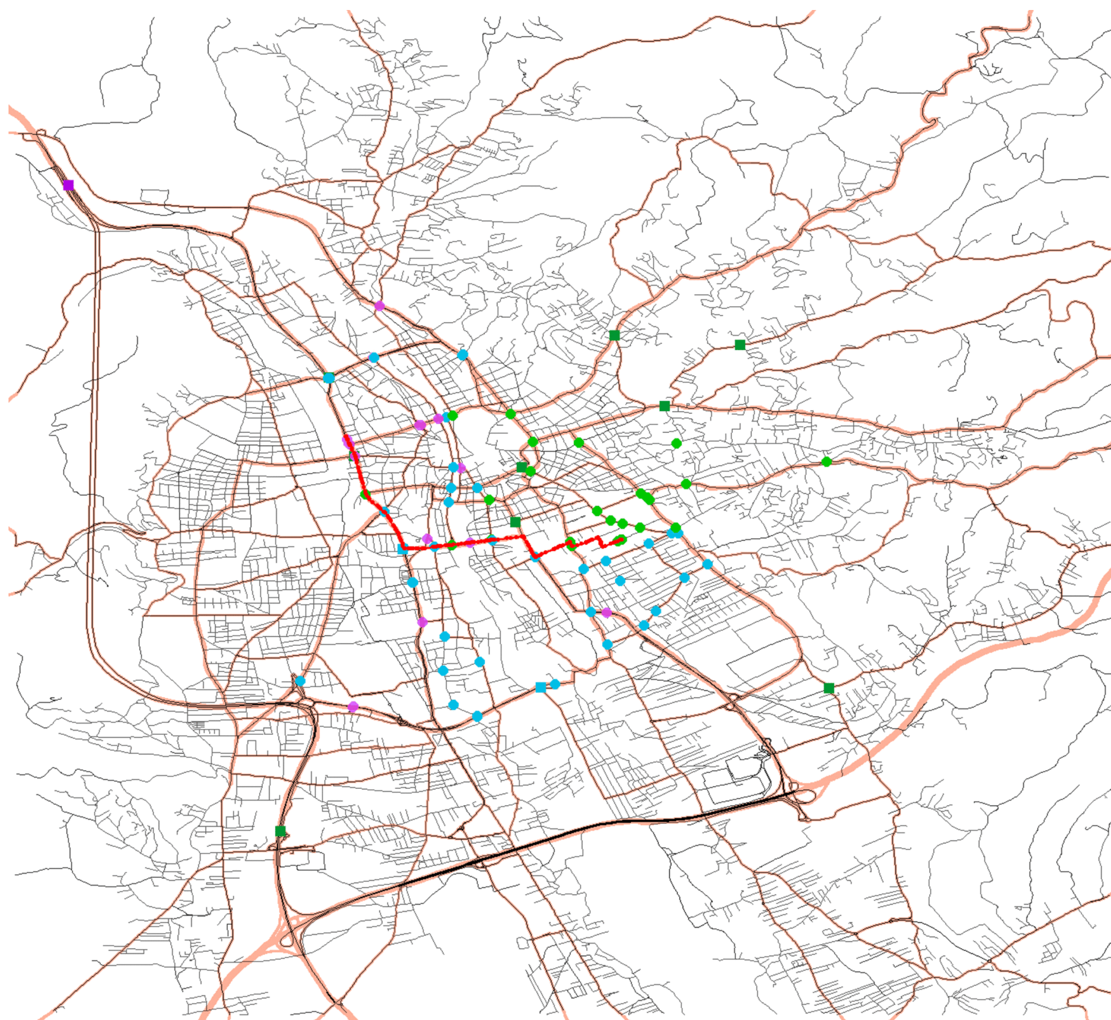


Figure 4.25: CH search on Graz (19 × 17 km) with **travel times**. Settled nodes are drawn as circles in light green for the forward search and in light purple for the backward search. Nodes in the open set are drawn as squares in dark green and dark purple for the forward and backward search, respectively. Nodes touched by both searches are drawn in blue. The shortest path is shown in red. Major roads are drawn in orange.

CH does not use additional or special data structures which means any search algorithm can work with the preprocessed network. In this work, the bidirectional Dijkstra's algorithm is used for search queries. There is one restriction that has to be obeyed

to take advantage of the hierarchy. The forward search may only consider edges to nodes at a higher hierarchy level and the backward search only edges from higher to lower level nodes.

A search with CH on the road network of Graz is shown in Figure 4.25. The queried path starts east, at the University of Technology and ends at the main railway station. CH only settles a few nodes near the start and target and quickly works its way up in the hierarchy through the shortcuts. Comparing the search in Figure 4.25 with the network layout previously shown in Figure 4.19 on page 73 reveals that most of the visited nodes match with hubs from the top levels. Compared to the other algorithms, CH achieves the highest efficiency (see Table 4.14). For this search, only 109 nodes have been settled, while the shortest path has 212 nodes, which results in an efficiency of 195%. This is possible because shortcuts created during preprocessing combine nodes.

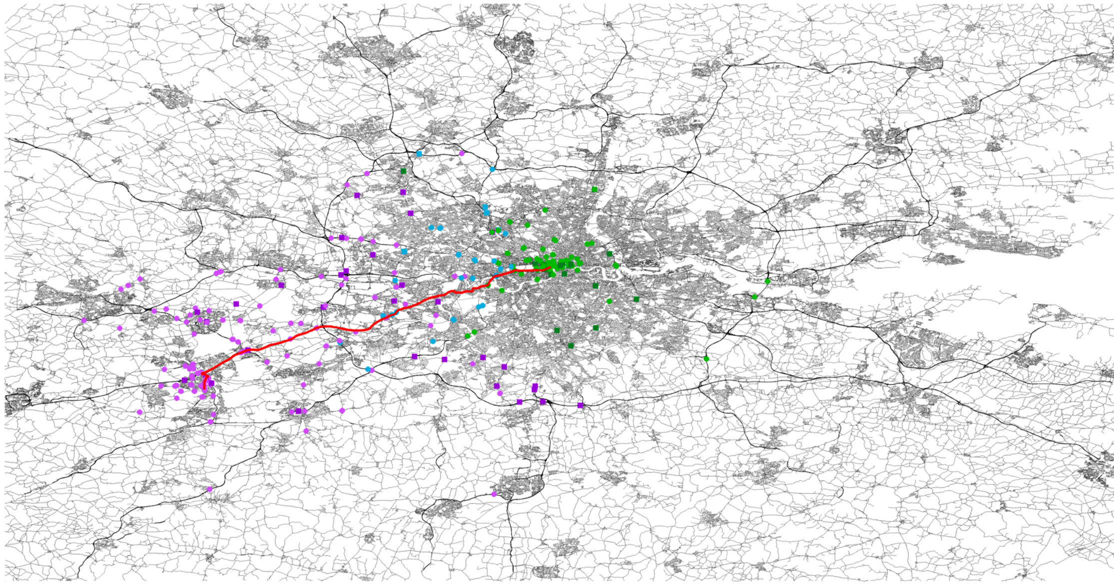
Table 4.14: Efficiency comparison between all algorithms.

	node efficiency [%]		edge efficiency [%]	
algorithm	mean	median	mean	median
Dijkstra	0.29	0.23	0.13	0.10
Bidir. Dijkstra	0.56	0.43	0.26	0.20
A*	1.25	0.98	0.57	0.46
Bidir. A*	2.01	1.60	0.91	0.71
ALT	7.38	5.98	3.36	2.74
Bidir. ALT	14.98	13.45	6.81	5.96
CH	150.64	99.93	15.81	8.29
CRP	79.40	50.73	4.85	2.47

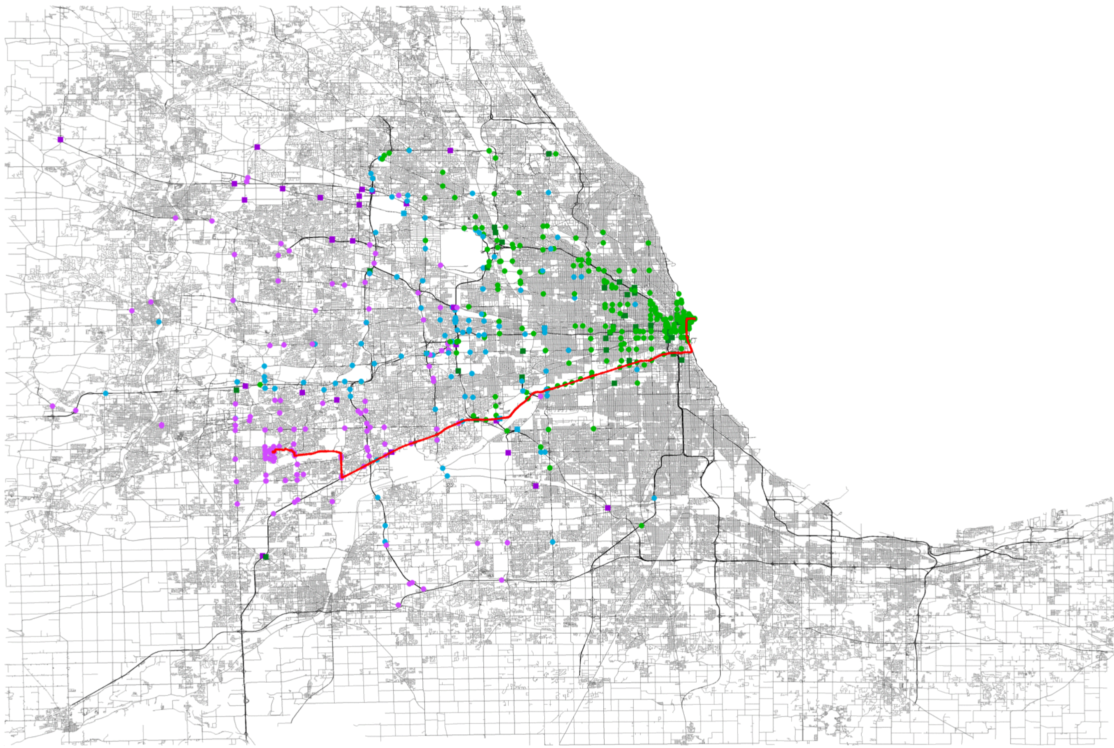
For comparison with the other algorithms, the same search as before on the Greater London and Chicago Area is shown in Figure 4.26. The search pattern is similar to the search on the network of Graz. Because the graph of London has 33 times and the graph of Chicago 17 times more nodes, there are much more important hubs compared to Graz and it is less obvious why each node was settled without closer inspection. The main difference between both searches is the efficiency, which is 377% on the London graph, while it is 89% on the Chicago graph. The discrepancy is especially high for this single search, while the average efficiency after 100 000 random searches is 254% for London and 139% for Chicago, which puts both above the median over all test cases that happens to be 100%. No definite reason could be found why the performance is so much better on the network of London.

4.7.4 Results & Conclusion

The average performance of CH over all test cases is shown in Table 4.15. Compared to the other algorithms presented so far, its performance is unparalleled. Even on the largest networks the average query only takes a few milliseconds. Due to the addition of



(a) Greater London Area (224 × 116 km).



(b) Greater Chicago Area (160 × 107 km).

Figure 4.26: CH search on Greater London and Chicago Area with **travel times**. Settled nodes are drawn as circles in light green for the forward search and in light purple for the backward search. Nodes in the open set are drawn as squares in dark green and dark purple for the forward and backward search, respectively. Nodes touched by both searches are drawn in blue. The shortest path is shown in red.

shortcuts to the graph the node efficiency is particularly high with an average of 150% over all networks and metrics and a top value of 870% on the network of California. The preprocessing of CH was faster and added fewer shortcuts with travel times than geometric distances and as expected the results show that queries with travel times are also significantly faster.

Table 4.15: Average performance of CH with geometric distances and travel times.

	node efficiency [%]		nodes settled		query time [ms]	
category	distance	time	distance	time	distance	time
C-NA	100	118	589	415	0.43	0.25
C-EU	118	125	436	344	0.31	0.21
C-OW	103	121	603	439	0.46	0.28
R-NA	204	225	1 148	775	1.40	0.72
R-EU	283	290	1 010	685	1.40	0.70
R-OW	114	92	1 842	996	2.08	0.75
Continental	261	363	1 390	824	2.39	1.24
all	142	159	685	480	0.69	0.38

While the efficiency of all previous algorithms dropped with increasing network size the opposite is true for CH. The reason for this is that on larger graphs the shortest paths found by random searches consist of more nodes which means more nodes are skipped by shortcuts.

Table 4.16 shows the impact of switching from full to contracted networks. Even though the number of settled nodes stays almost the same, the query times improve. This is due to the time required to reconstruct the shortest path by unpacking shortcuts. If the intermediate nodes are not required and the reconstruction is skipped, the query times are actually equal (see last 2 columns of Table 4.16).

Table 4.16: Average performance of CH on full and contracted networks. The Continental and R-OW categories are not included because they consist of contracted networks only. The query times without path reconstruction are only from networks that have a full and a contracted version.

	node efficiency [%]		nodes settled		query time [ms]		query time [ms] without path reconstruction	
category	full	contr.	full	contr.	full	contr.	full	contr.
C-NA	177	41	505	499	0.40	0.28	0.21	0.22
C-EU	201	42	394	386	0.32	0.20	0.15	0.15
C-OW	180	44	531	511	0.44	0.30	0.24	0.23
R-NA	576	94	864	994	1.36	0.96	0.45	0.45
R-EU	537	64	822	871	1.43	0.72	0.45	0.45
all	249	66	540	619	0.57	0.50	0.24	0.24

To see if the query performance is influenced by the number of shortcuts created, the node efficiency was compared with the percentage of shortcuts created over all test cases but no correlation was found between them.

The preprocessing phase only takes a few seconds on small and medium sized graphs (as has been shown in Figure 4.22) and can be improved further through optimizations and parallelization [23, 36, 67] which means CH is suitable for dynamic networks that update frequently. Furthermore, the low query times allow it to handle large number of queries. On larger countries/regions and continental sized networks the preprocessing starts taking minutes and without optimizations and multi-threading can even take up to an hour. While other algorithms such as ALT or CRP can reuse parts to speed up the preprocessing, this is not possible with CH without a severe impact on the performance [22]. CH can be parallelized, but not to the same degree/speedup as ALT and CRP due to its hierarchical nature [67]. This means that the only advantage that remains on large networks are the low query times.

4.8 Customizable Route Planning (CRP)

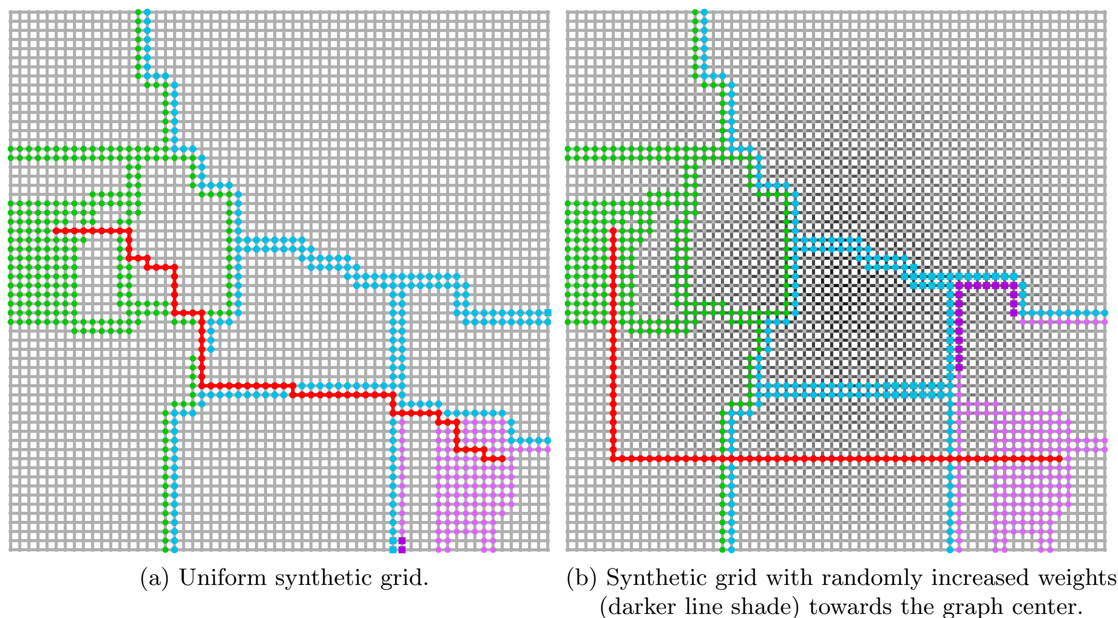
Customizable Route Planning is the most complicated of all presented algorithms. It has a preprocessing stage which partitions the graph in cells with as few boundary edges as possible and creates a multilevel overlay which is used during queries. The search for the shortest path is executed with a bidirectional Dijkstra's algorithm. Because the partitions and overlay can be created metric independently, customization (partial and complete edge weight changes) can be applied much faster compared to ALT or CH. Another advantage of CRP is that its preprocessing is highly parallelizable, including customization.

4.8.1 Synthetic Grids

Figure 4.27 shows a CRP search with 5 levels on a grid graph. The start and target node have been chosen farther apart (compared to previous examples) to better demonstrate how nodes are settled in a multilevel search. The partitioning in CRP is optimized for road networks with *natural cuts* (e.g. rivers, mountains; see Section 2.11 on page 24), but a grid graph as shown in the figure does not have such. While the algorithm still works with this kind of graphs, its advantage over general purpose partitioners is lost, i.e. it does not find partitions with considerably fewer boundary edges.

Figure 4.27a with the uniform grid will be analyzed first. The nodes shown as settled (drawn in green, purple and blue) are boundary nodes of cells on various levels, except those next to the start and target node, which are from the original graph. The search must scan the nodes in the original graph up to the first boundary, because only there a transition to a higher level is possible. Areas of unsettled nodes get larger farther away from the start and target node because the search uses the highest level where the

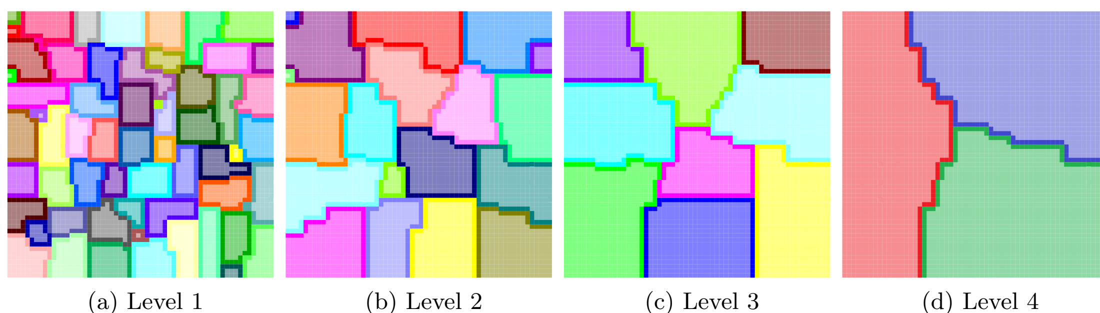
corresponding cell does not contain the start nor the target node. This means larger cells at higher levels can be used. For a better illustration, the partitions of all levels are shown in Figure 4.28, except level 0 and 5, because at level 0 every node is its own cell and at level 5 (the top level) all nodes are in the same cell.



(a) Uniform synthetic grid.

(b) Synthetic grid with randomly increased weights (darker line shade) towards the graph center.

Figure 4.27: CRP search on synthetic grid graphs. The start is the upper left and the target the lower right end of the red line, which shows the shortest path found. Nodes in the closed set (settled nodes) are drawn in light green for the forward search and in light purple for the backward search. Nodes in the open set are drawn in dark green and dark purple for the forward and backward search, respectively. Nodes touched by both searches are drawn in blue.



(a) Level 1

(b) Level 2

(c) Level 3

(d) Level 4

Figure 4.28: CRP partitions of the grid graph from Figure 4.27.

Comparing both figures (4.27, 4.28), the outlines of the cells from higher levels can be spotted easily in the pattern of the settled nodes. It is important to note that cells always share their boundaries with the cells from the next lower level, otherwise the transition between levels would not work. In the absence of natural cuts, the shape of

the cells is purely a result of the random factors used during partitioning (see Section 2.11 on page 24). At level 2 (Figure 4.28b), four cells with distinctly smaller sizes than the others can be spotted. This is a side effect of the grid graph and happens because the partitioner is tuned to optimize for fewer boundary nodes and not balanced cell sizes.

Even though CRP is not designed for grid graphs, the pattern shown by the settled nodes in Figure 4.27 is characteristic of it, as will be shown in further examples. The performance on this graph is poor though, because of the high number of boundary nodes. More than 40% of the nodes at level 1 are boundary nodes. For road networks the median is 4.62% over all test cases.

A last thing to note is the area where the forward and backward search meet each other which are the nodes shown in blue. As described in the theoretical part, a bidirectional search can not stop as soon as each search front meets the other the first time. This has the consequence that both searches may have to expand into more than one cell that has its boundary nodes already settled by the other. In Figure 4.27a there are four such cells.

Figure 4.27b shows the synthetic grid with increased edge weights towards the graph center. The partitioning for this graph is the same as for the other, because it is independent of the edge weights. The search itself though, which uses a bidirectional Dijkstra's algorithm, expands along the cells with the shortest distance. This can be seen on the right side in the figure, where a cell that had nodes settled by both searches (shown in blue) previously on the uniform grid now only has nodes settled by the backward search (shown in purple). The forward search expanded faster at the bottom of the graph and the shortest path was already found before it could reach through at the center.

4.8.2 Partitioning on Road Networks

How CRP works with real road networks will be analyzed now, starting with a close look at the partitioning, which is the key part for its performance.

Natural Cuts

Because it is clear from a theoretical standpoint what parts (those where the network flow is limited the most) qualify as natural cuts (see Section 2.11 on page 24), the following paragraphs will focus on what constitutes a natural cut in practice.

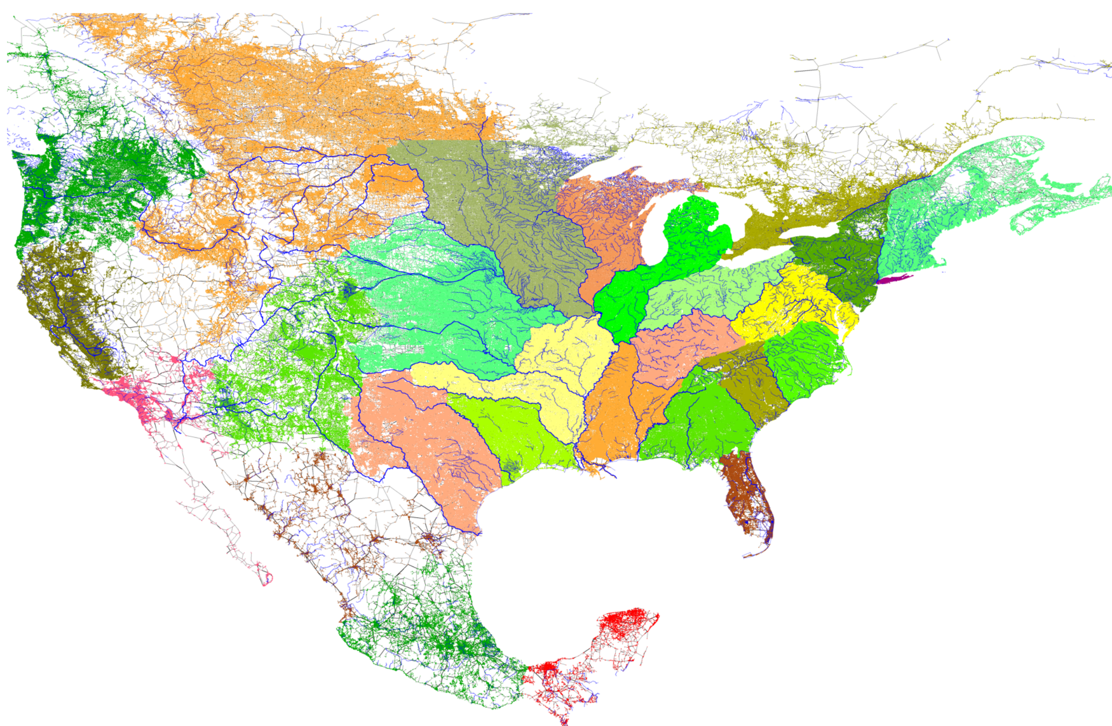


Figure 4.29: CRP partition of the North American road network with rivers shown on top of it in blue.

Continental Scale. In Figure 4.29 the partition of the North American road network with rivers on top of it is shown. The abrupt change in the density of the road network at the Mexican border happens because the used data from OpenStreetMap is less detailed for rural parts of Mexico. The first thing to notice is, how the major part of the cell boundaries run along rivers. Additional natural cuts are the mountain ranges in the west and the borders to Canada and Mexico are significant boundaries for two cells. The partition created contains 27 cells with only 1 636 boundary edges (126 per cell on median), which constitute less than 0.006% of all 29 743 795 edges in the graph. The partition of continental Europe, which is shown in Figure 4.30, follows the same pattern. Most cell boundaries run along rivers and mountain ranges. It contains 22 cells with 1 532 boundary edges (142 per cell on median), which also constitute less than 0.006% of all 26 372 551 edges. Comparing these results to other algorithms shows that CRP finds better partitions with fewer boundary edges on road networks because of its focus on natural cuts (see [11, p. 21] for a detailed comparison).

On a continental scale, rivers and mountain ranges can be seen as the dominating factor for partitioning, with borders only playing a minor part. While there are also huge desert areas on earth without rivers in abundance such as the Sahara, Central Asia or Australia, those are mostly uninhabited without vast road networks. Still it remains to be seen how the partitioning works on a smaller scale where rivers and mountains are more spread out.

Figure 4.30: CRP partition of the continental European road network with rivers shown on top of it in blue.

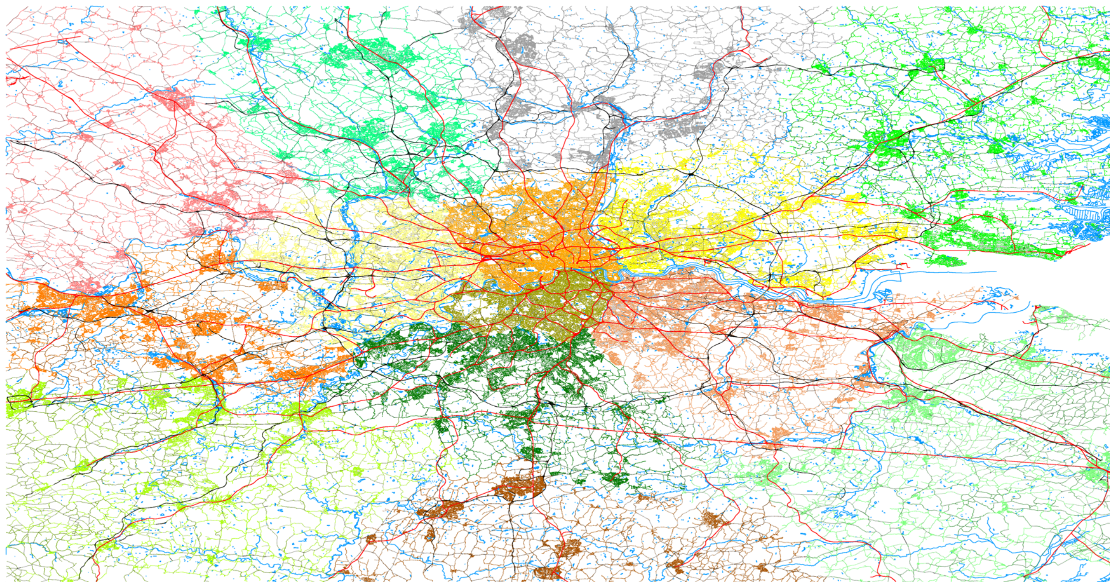


Figure 4.31: CRP partition of the Greater London Area (224 × 116 km) road network (black) with waters (blue) and railways (red) shown on top of it.

Greater Metropolitan Area Scale. Figure 4.31 shows the partition of the Greater London Area with waters and railways on top of it. Compared to the partitions of Europe and North America, rivers are not as important as boundaries anymore and there are no large mountains around London. Only the River Thames cuts right through the road

network, while its tributary rivers only matter at a few segments. The dense rail network is a main factor near the city center because the number of crossings is small compared to the amount of roads in the dense network next to it. Outside of the city center the opposite can be observed. While the rail tracks and motorways lead through towns surrounding London, the cell boundaries run between these towns in the rural areas with few roads (e.g. all boundaries in the northern part show this behavior). Motorways have almost no influence on the partitioning on this graph. Two main reasons for this circumstance can be found: There are no motorways that lead into the city center where they could have a similar effect to railways and the motorways outside of London are subject to the effect described just before. Another reason for the division of the road network can be seen in the city center as empty (white) spots, which are parks and large complexes of buildings such as shopping centers and industrial buildings (power plants, sewage plants etc.). The partition created contains 14 cells with 400 boundary edges (59 per cell on median), which constitute 0.024% of all 1 675 960 edges.

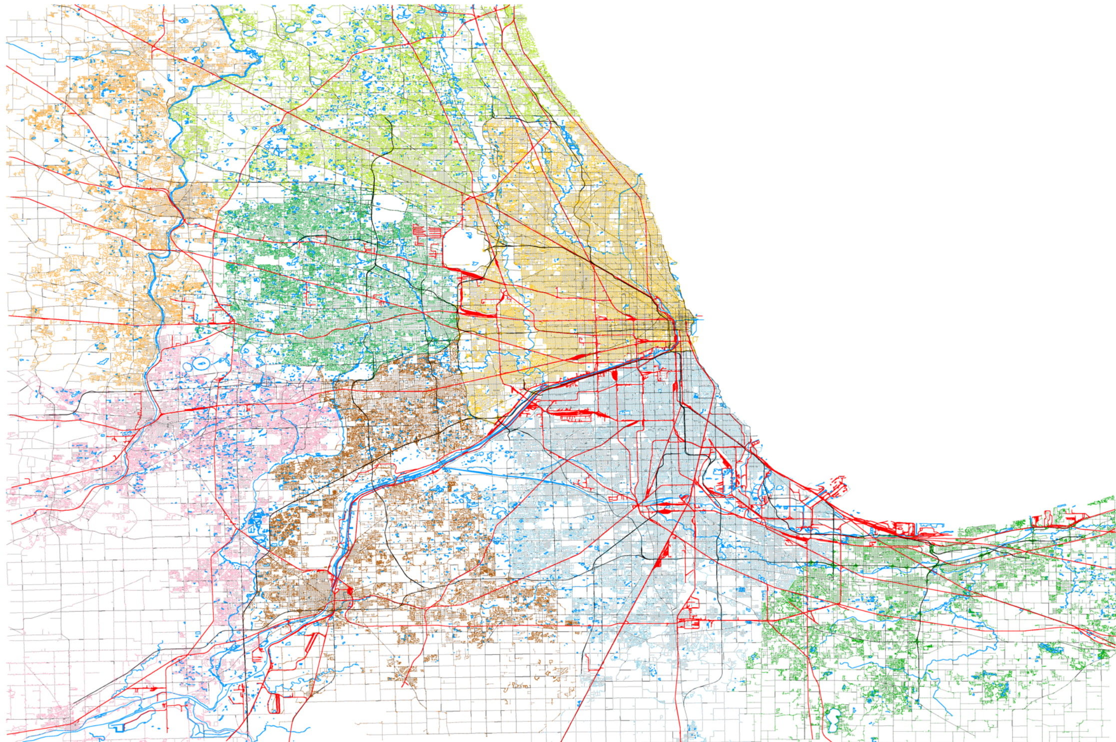


Figure 4.32: CRP partition of the Greater Chicago Area (160 × 107 km) road network (black) with waters (blue) and railways (red) shown on top of it.

In Figure 4.32 the partition of the Greater Chicago Area with waters and railways on top of it is shown. Compared to the Greater London Area, the freeways lead straight into the city center. Furthermore, the suburbs have a different layout. While London has many smaller towns, separated by rural areas, surrounding it, Chicago features a continuously urban sprawl only broken up by some remaining parks and recreational

areas. The graph only has half the number of nodes resulting in a partition with fewer cells. The city center is split by the Chicago River, which is encompassed by rail tracks and a freeway. Also, there are several industrial zones along the river which represent additional barriers. The northern part of the city center is cut off by freeways, cemeteries and the large O'Hare International Airport in the west and a river, freeways and golf courses in the north. On the contrary, the southern part of the city center is missing many features of the north and is mostly cut along parks and undeveloped zones. The remaining outer cells are for the most part separated by rivers and undeveloped land. Even though there are clear differences between the layouts of the London and Chicago area, the quality of the partition is, considering the reduced number of cells, similar, containing 8 cells with 283 boundary edges (80 per cell on median), which constitute 0.03% of all 920 466 edges.

Comparing the continental scale with the larger metropolitan area partitions, rivers are not as dominating as before. Railways and motorways/freeways only play a role at dense areas near or in the city center. A new important factor though are larger road free zones which disrupt the network. Those are mainly undeveloped land on the outskirts and parks, recreational areas and building complexes such as airports, shopping centers and industrial zones closer to the city center.

Figure 4.33: CRP partition of the London City Center (30 × 17 km) road network (black) with waters (blue) and railways (red) shown on top of it.

City Scale. Partitioning on a larger metropolitan area scale is still high up in the multilevel hierarchy, actually it is the second highest level with the settings used. To complete the analysis, the city center networks of London (Figure 4.33) and Chicago

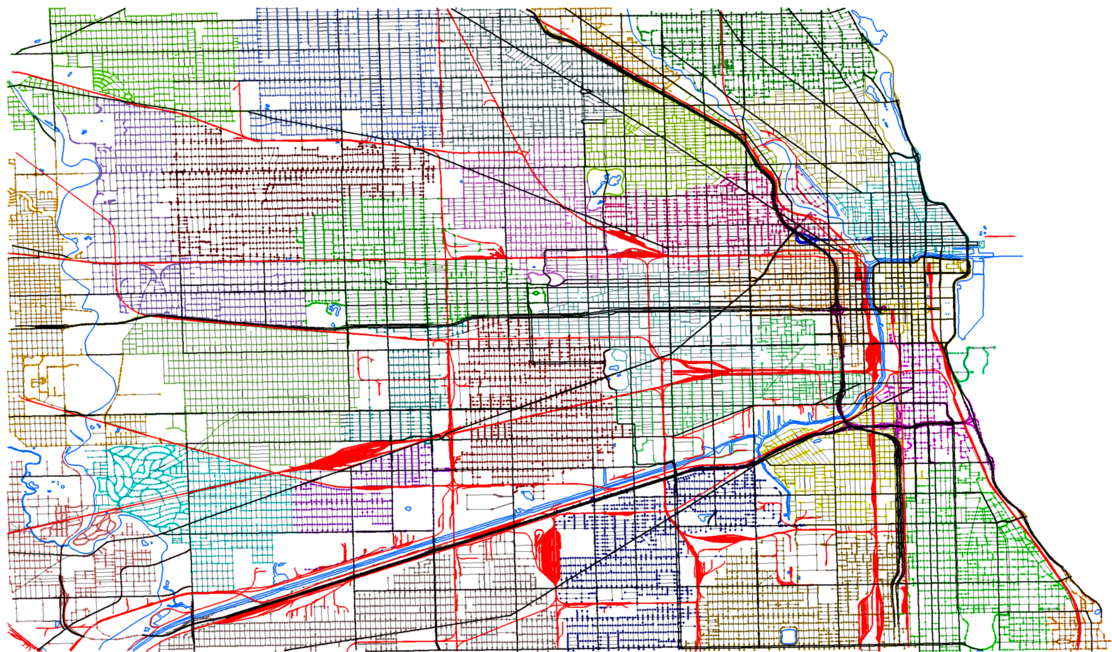


Figure 4.34: CRP partition of the Chicago City Center (30 × 17 km) road network (black) with waters (blue) and railways (red) shown on top of it.

(Figure 4.34) will be inspected now. As can be seen in both figures, all the factors mentioned before are parts of boundaries, even though railways and freeways seem to have no effect in some areas, which can be explained with the fact that they are elevated in the city center (the rapid transit system in Chicago is actually called *Chicago Elevated*). Because of the small cell size, boundaries also start running along residential roads at areas without any noticeable disruption in the network. This can be seen best in the northern part of the Chicago graph, with the negative effects of the grid graph beginning to show. While the cells in the London partition have 18 incident edges on median, the cells in the Chicago partition have almost twice as many with 35. One level below on level 1, the Chicago cells have 18 incident edges on median, while the median over all networks is 9.

Conclusion. As has been shown, it depends on the scale at which a network is partitioned to specify what is actually responsible for the cuts. On large scales, natural cuts are dominated by things one would associate with the term in the first place, such as mountain ranges and rivers. On smaller scales, other things come into play which are not that natural such as motorways and railways or larger zones (e.g. airports, shopping centers or industrial areas). While one may think that cuts require something that blocks the expansion of the road network, areas with a sparser network surrounding denser parts such as towns can also be a natural cut. On the smallest scales, the concept of natural cuts starts reaching its limits and its advantage over general purpose partitioners diminishes.

Boundary Sizes

All networks are partitioned using the same settings in accordance with the recommendations by Delling et al. (see [10, Chapter 7.2]) and a cell size of $s = 2^{level*3+5}$. The maximum and mean cell sizes for each level are shown in Table 4.17. The deviation increase between maximum and average values with rising level is due to networks not matching the size ratio (i.e. being too small) at the top level to fill whole cells.

Table 4.17: Cell sizes (node count) for the partition at the given level over all networks.

level	max cell size	mean cell size
1	256	206
2	2 048	1 688
3	16 384	13 355
4	131 072	94 130
5	1 048 576	461 091

In Figure 4.35, the percentage of boundary nodes per level is shown. While the numbers vary between individual networks (up to a factor of six at the first three levels), they stay constant over all networks, unaffected by their different sizes. For most networks, the percentage of boundary nodes follow the same trend between levels, e.g. if a network has a higher than average number of boundary nodes at level 1, this is also true for the other levels. This can be attributed to the fact that cells at a higher level share their boundary nodes with cells at lower levels, i.e. the partitions at different levels are not independent from each other. The results for level 4 and 5 are skewed at the start of the curves due to the networks being too small to fill whole cells. The networks where the partitioning performs the worst and best are similar to those from CH's preprocessing. Dense and highly connected graphs such as Buenos Aires (Figure 4.21b on page 75) or the Chicago City Center (Figure 4.34) create the most boundary nodes.

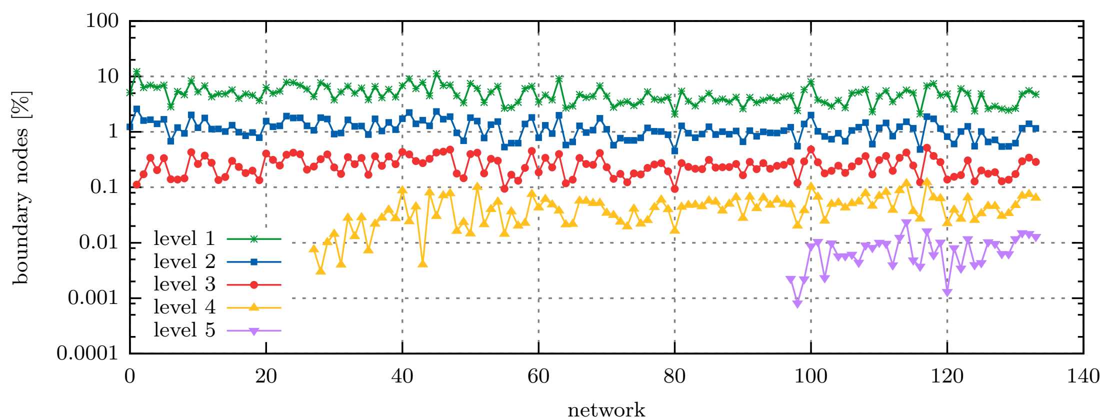
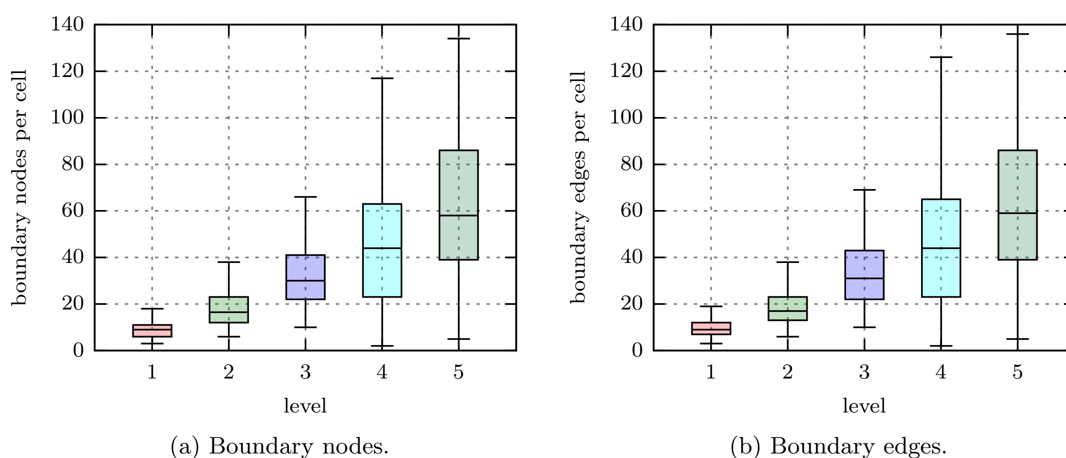


Figure 4.35: Percentage of nodes that are boundary nodes in the partition at the given level. Sorted by the networks' node count in ascending order.

The number of boundary nodes and edges per cell are shown in Figure 4.36. A boundary node can have more than one edge that connects it to other cells but as the sub-figures show this rarely happens as the results are almost identical. The number of boundary nodes per cell increases with the level but higher levels also have fewer cells and the total number of boundary nodes drops, as shown in Figure 4.35.



(a) Boundary nodes. (b) Boundary edges.

Figure 4.36: Number of boundary nodes and edges per cell over all networks at the given level.

Preprocessing Time

The following figure and numbers are based on the sequential algorithm. Preprocessing times can be significantly reduced by parallelization because many steps can be executed independently. For example switching from one to four threads makes partitioning 2.6 times faster [11, p. 15]. By using twelve threads, customization is almost ten times faster [9, p. 24].

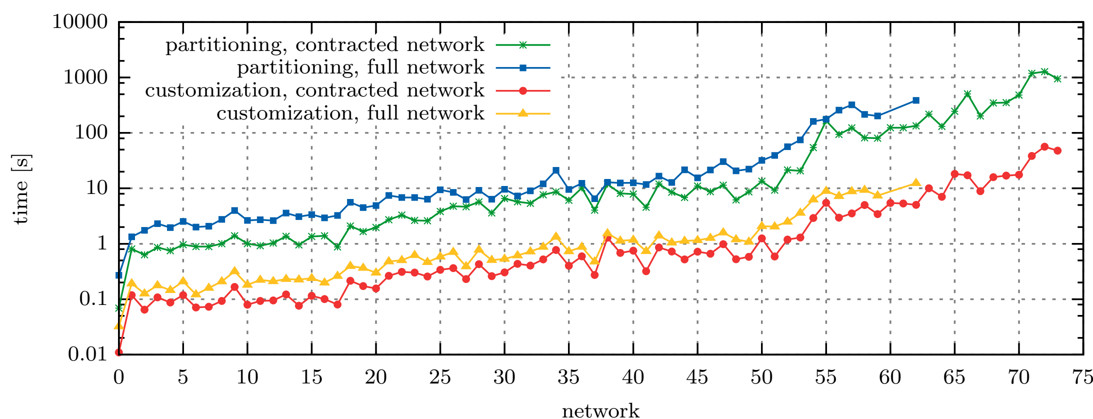
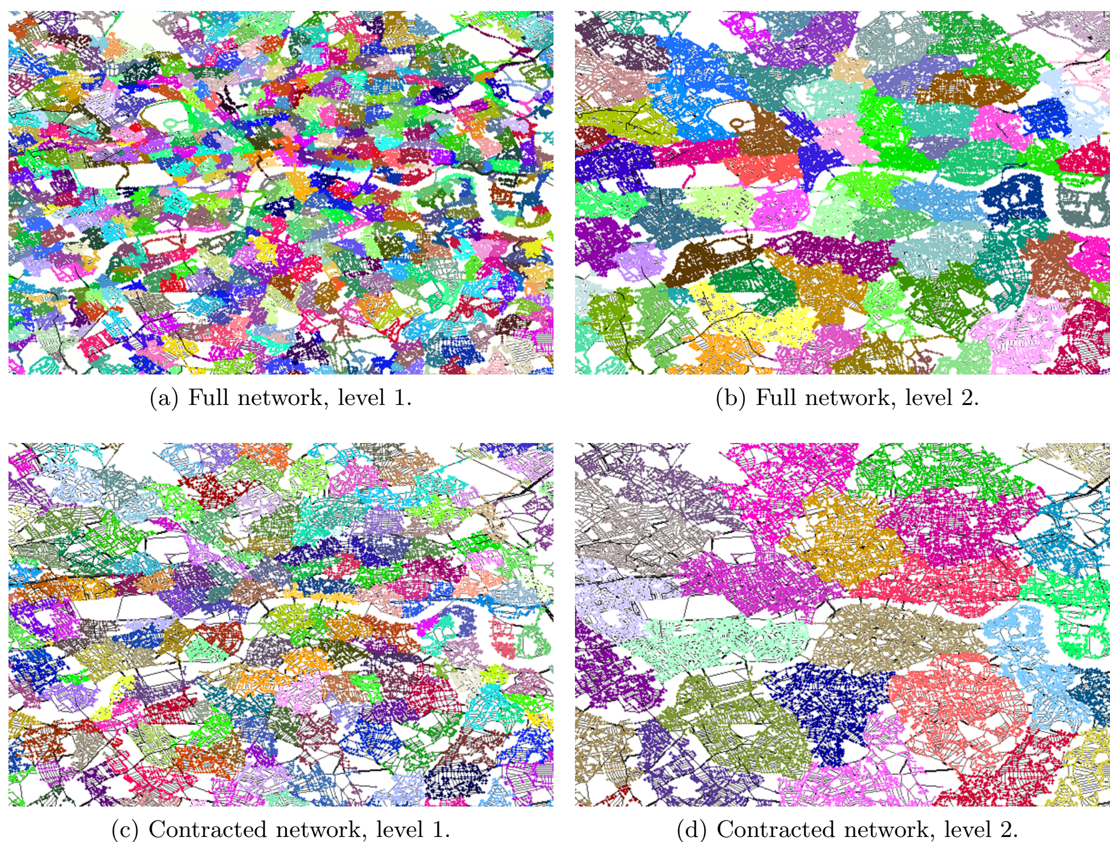


Figure 4.37: Time required for preprocessing for all networks, shown in seconds on a logarithmic scale. Sorted by the networks' normal edge count in ascending order.

Figure 4.37 shows the time required for partitioning a network and for customization. During customization, the cliques are created, i.e. the shortest paths between all boundary nodes of a cell are computed. Partitioning and customization is faster with contracted than with full networks in all cases. In contrast to CH, the partitioning is independent of the metric used. For small and medium sized graphs, the customization takes less than a second and even on larger ones it stays below ten seconds. Only on the largest graphs, namely the European and North American road networks, it can take up to a minute. However, as has been mentioned before, the customization is highly parallelizable. In [10], Delling et al. demonstrate that the customization of the European and North American network can be done in less than ten seconds by using twelve cores.

Contracted Networks



(a) Full network, level 1.

(b) Full network, level 2.

(c) Contracted network, level 1.

(d) Contracted network, level 2.

Figure 4.38: Partition comparison of Greater London Area's full and contracted network (magnified section of the center). Because of the different node densities, the size of the zones spanned by the cells do not match on the same level.

In contracted networks paths consisting of neighboring nodes with degree two are contracted into a single node which reduces the number and density of nodes in the network. This has the effect that the space spanned by cells gets larger, as shown in Figure 4.38. Even though the number of nodes per cell stays the same, the network is cut

differently because the cell boundary increases. Due to this the cells in the contracted networks that have been tested have approximately 1.7 times more boundary nodes.

Table 4.18: Comparison of the median number of boundary nodes per cell per level with different cell sizes over all networks with a full and contracted version. The first and second row show the results with the default size and the last row with a smaller size.

		boundary nodes				
network type	cell size (s)	level 1	level 2	level 3	level 4	level 5
full	$s = 2^{level*3+5}$	6	12	24	44	53.5
contracted	$s = 2^{level*3+5}$	11	22	41	42	58
contracted	$s = 2^{level*3+3}$	7	13	27	46	52

To see how this influences the preprocessing and performance of CRP, the tests have been repeated with a smaller cell size that better matches the partitioning of the full networks. In Table 4.18, the obtained median numbers of boundary nodes per cell are compared with the previous ones. Partitioning the contracted networks with the same cell size as the full networks increases the number of boundary nodes per cell as if a level was skipped. By reducing the cell size, the numbers resemble those of the full networks again. The results at higher levels, at least for the contracted networks with the default cell size, are not as representative because, as has been mentioned before, the networks are too small to fill whole cells or create more than one cell at the top level.

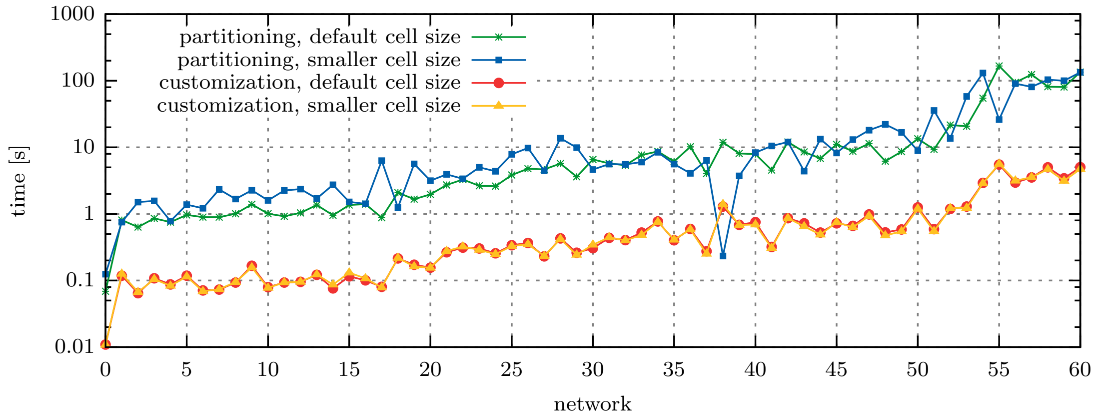


Figure 4.39: Time required for preprocessing for all contracted networks with different cell size. Sorted by the networks' node count in ascending order.

The impact of different cell sizes on the preprocessing time is shown in Figure 4.39. While the customization phase takes about the same time, partitioning with a smaller cell size is slower in most cases. The two outliers are the network of Buenos Aires and Tokyo. In Table 4.19, the average query times from contracted networks with different cell sizes are compared which shows that a slightly better query performance is achieved

with the default cell size. In conclusion, even though using the default cell size leads to more boundary nodes with contracted networks compared to their full version, reducing the cell size to match the number of boundary nodes does not improve the preprocessing nor the query times in most cases and should therefore not be done.

Table 4.19: Comparison of the query times on contracted networks with the default cell size $s = 2^{level*3+5}$ and a smaller size $s = 2^{level*3+3}$.

	query time [ms]			
	default size		smaller size	
category	distance	time	distance	time
C-NA	0.33	0.30	0.34	0.31
C-EU	0.20	0.19	0.20	0.18
C-OW	0.35	0.32	0.38	0.36
R-NA	0.80	0.75	0.87	0.82
R-EU	0.61	0.54	0.70	0.61
all	0.35	0.32	0.37	0.34

4.8.3 Search on Road Networks

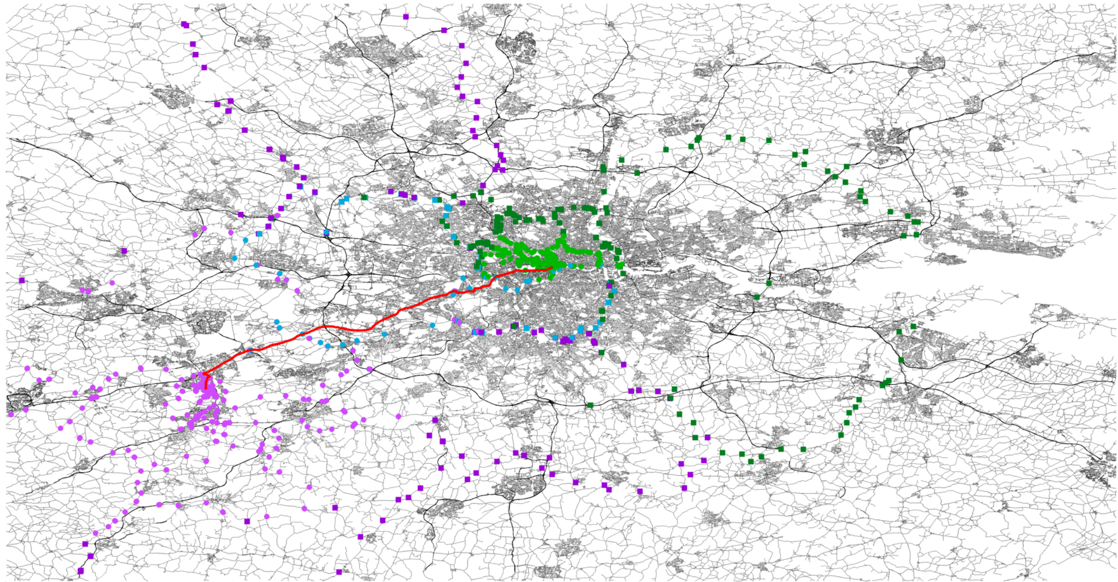
The search on road networks works equally to the synthetic grid. Because of the abstraction through the multilevel overlay the influence of the underlying network is small. Figure 4.40 illustrates a search with six levels on the Greater London and Chicago Area with travel times. The forward and backward search start scanning nodes at the lowest level at the start and target, working up the hierarchy at nodes farther away. The nodes touched by both searches run along the boundaries of the cells from the various partitions, as can be seen by comparing the search graphs with the partitions shown before (Figure 4.31 on page 85, Figure 4.32 on page 86).

As with the other bidirectional algorithms before, either the forward or backward search is executed, depending on which has fewer nodes in the open set (see Section 2.5 on page 8). Because the network around the start node in the city center is denser, which results in more boundary nodes, the backward search (purple) extends farther than the forward search (green).

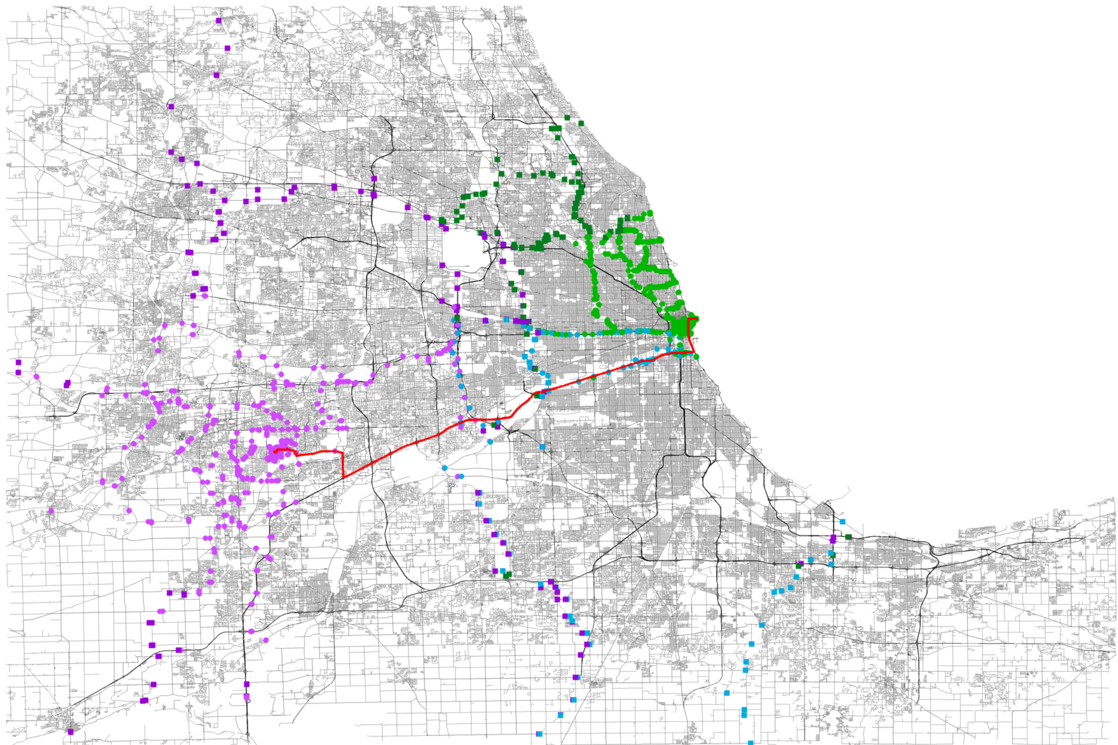
Due to the only small difference in the number of settled nodes and query times between travel times and geometric distances (see Table 4.20), the search pattern with geometric distances is almost identical to Figure 4.40 and therefore omitted.

4.8.4 Results & Conclusion

In Table 4.20, the average performance of CRP over all test cases is shown. Of the presented algorithms, CRP is the only one that can compete with the performance of CH. Due to the overlay network which only consists of the cells' boundary nodes, a high



(a) Greater London Area (224 × 116 km).



(b) Greater Chicago Area (160 × 107 km).

Figure 4.40: CRP search on Greater London and Chicago Area with **travel times**. Settled nodes are drawn as circles in light green for the forward search and in light purple for the backward search. Nodes in the open set are drawn as squares in dark green and dark purple for the forward and backward search, respectively. Nodes touched by both searches are drawn in blue. The shortest path is shown in red.

efficiency is achieved with an average of 79% over all networks and metrics and a top value of 569% on the network of Spain. The used metric only has a small influence on the performance. With travel times, slightly fewer nodes are settled and queries are a little bit faster. Compared to CH, the node efficiency is not as good but the query times are similar.

Table 4.20: Average performance of CRP with geometric distances and travel times.

	node efficiency [%]		nodes settled		query time [ms]	
category	distance	time	distance	time	distance	time
C-NA	57	50	996	943	0.34	0.31
C-EU	63	55	793	759	0.22	0.20
C-OW	59	54	1 014	969	0.36	0.33
R-NA	139	112	1 885	1 822	1.13	1.04
R-EU	198	141	1 456	1 391	0.78	0.68
R-OW	97	44	2 160	2 093	1.27	1.15
Continental	170	150	2 312	2 242	1.86	1.71
all	87	72	1 127	1 078	0.49	0.45

Table 4.21: Average performance of CRP on full and contracted networks. The Continental and R-OW categories are not included because they consist of contracted networks only. The query times without path reconstruction are only from networks that have a full and a contracted version.

	node efficiency [%]		nodes settled		query time [ms]		query time [ms] without path reconstruction	
category	full	contr.	full	contr.	full	contr.	full	contr.
C-NA	86.92	20.92	989	950	0.34	0.32	0.32	0.31
C-EU	96.17	21.71	808	743	0.23	0.19	0.21	0.19
C-OW	90.98	21.66	1 005	978	0.36	0.33	0.33	0.33
R-NA	354.64	48.67	1 449	1 988	0.77	1.19	0.65	0.73
R-EU	315.16	39.43	1 380	1 462	0.74	0.72	0.61	0.55
all	129.52	36.23	1 042	1 154	0.42	0.52	0.34	0.33

Switching from full to contracted networks only has a small effect (see Table 4.21), with contracted networks performing slightly better. The reason for the discrepancy at the *Regions - North America* category is that it consists of twelve contracted and only four full networks which skews the results.

Because of the overlay network, the intermediate nodes of shortest paths have to be reconstructed, similar to CH. There are different ways to manage the overlay network with a trade-off between increased processing time and memory usage. In this work all the intermediate nodes are kept in memory which has the smallest performance impact

on query times, as can be seen in Table 4.21. There is almost no impact from the reconstruction on the *Cities & Metropolitan Areas* networks and only on the *Regions & Countries* the performance starts dropping.

On small and medium sized graphs, even with the sequential algorithm the preprocessing of CRP only takes a few seconds and the customization stays below one or two seconds (as has been shown in Figure 4.37). While the preprocessing on larger graphs starts taking a significant amount of time, it is metric independent, which means the results can be reused for different metrics. Even on the largest graphs the customization itself takes less than a minute and can be reduced to a few seconds by parallelization [10]. This and the fast query times make CRP suitable for frequently updated dynamic networks of continental size or even larger. Furthermore, smaller changes to the network inside cells can also be handled by just repeating the customization phase.

Chapter 5

Performance Comparison

While each algorithm was reviewed separately in the previous chapter, their performance is now compared among each other. Furthermore, it is examined whether the many different road networks lead to similar results or if there are significant variations. To get a detailed image, the tests are performed separately for full and contracted networks with geometric distances and travel times.

5.1 Experimental Setup

The A* algorithm is only evaluated on networks with geometric distances due to its limitation with the heuristic with travel times. The query times for CH and CRP contain the costs for the path reconstruction, i.e. the unpacking of the intermediate nodes. 36 landmarks are used for ALT which are set with the customized version of the *Partition-Corners* algorithm explained in Section 4.6.2 on page 61. Other settings correspond to those specified in Section 3.1 on page 29. The same experimental setup as in the previous chapter is used (see Section 4.2 on page 41).

When the average performance of algorithms is compared in the following, the given percentage of the difference is calculated like this: the result (e.g. query time) from every network is added up and the total sum is compared. Furthermore, when numbers between full and contracted networks are compared directly, only those networks that have a full and contracted version are considered. This means the results will be slightly different to comparing the totals of all networks.

In the figures of the following sections, the networks are only referenced by a number on the x-axis. The corresponding names are listed in Table 5.1 for full networks and in Table 5.1 for contracted networks. A detailed list with key figures for each network can be found in the appendix (Section A.1 on page 123).

Table 5.1: Network number to name lookup table for full networks.

#	network	#	network	#	network	#	network
0	Graz	16	Hampton Roads	32	Phoenix	48	Los Angeles
1	Chicago City	17	Milan	33	Hong Kong	49	Washington
2	London City	18	Seoul	34	Minneapolis	50	Atlanta
3	Vienna	19	Moscow	35	Paris	51	Netherlands
4	Singapore	20	Lisbon	36	Saint Louis	52	Austria
5	San Diego	21	Rio de Janeiro	37	Chicago	53	Tokyo
6	Edmonton	22	Berlin	38	Philadelphia	54	Poland
7	Cape Town	23	Montreal	39	Detroit	55	California
8	Warsaw	24	Istanbul	40	Sao Paulo	56	Great Britain
9	Mumbai	25	Miami	41	Houston	57	Texas
10	Prague	26	Toronto	42	New York	58	Spain
11	Budapest	27	Madrid	43	Shanghai	59	Italy
12	Rome	28	Stockholm	44	Seattle	60	Germany
13	Brussels	29	Guangzhou	45	Dallas	61	USA
14	Buenos Aires	30	Bangkok	46	San Francisco Bay		
15	Cyprus	31	Melbourne	47	London		

Table 5.2: Network number to name lookup table for contracted networks.

#	network	#	network	#	network	#	network
0	Graz	18	Milan	36	Istanbul	54	California
1	Chicago City	19	Berlin	37	Bangkok	55	Tokyo
2	Vienna	20	Lisbon	38	Buenos Aires	56	Spain
3	San Diego	21	Guangzhou	39	Chicago	57	Italy
4	London City	22	Seoul	40	Detroit	58	Texas
5	Edmonton	23	Toronto	41	Seattle	59	Great Britain
6	Singapore	24	Montreal	42	New York	60	US-Northeast
7	Mumbai	25	Hong Kong	43	Houston	61	Germany
8	Warsaw	26	Minneapolis	44	Washington	62	France
9	Brussels	27	Rio de Janeiro	45	Sao Paulo	63	US-West
10	Prague	28	Philadelphia	46	San Francisco Bay	64	US-Midwest
11	Hampton Roads	29	Saint Louis	47	Dallas	65	Japan
12	Cyprus	30	Madrid	48	Atlanta	66	Africa
13	Rome	31	Miami	49	London	67	South America
14	Moscow	32	Phoenix	50	Los Angeles	68	US-South
15	Budapest	33	Paris	51	Austria	69	Asia
16	Cape Town	34	Shanghai	52	Netherlands	70	Europe
17	Stockholm	35	Melbourne	53	Poland	71	North America

5.2 Query Times

The most important performance metric for the application of the algorithms is the time required to execute a search query. To get comparable, undistorted results, all algorithms have been implemented in one framework using the same components (data structures, priority queues, etc.) and all tests are executed on the same system.

5.2.1 Geometric Distances

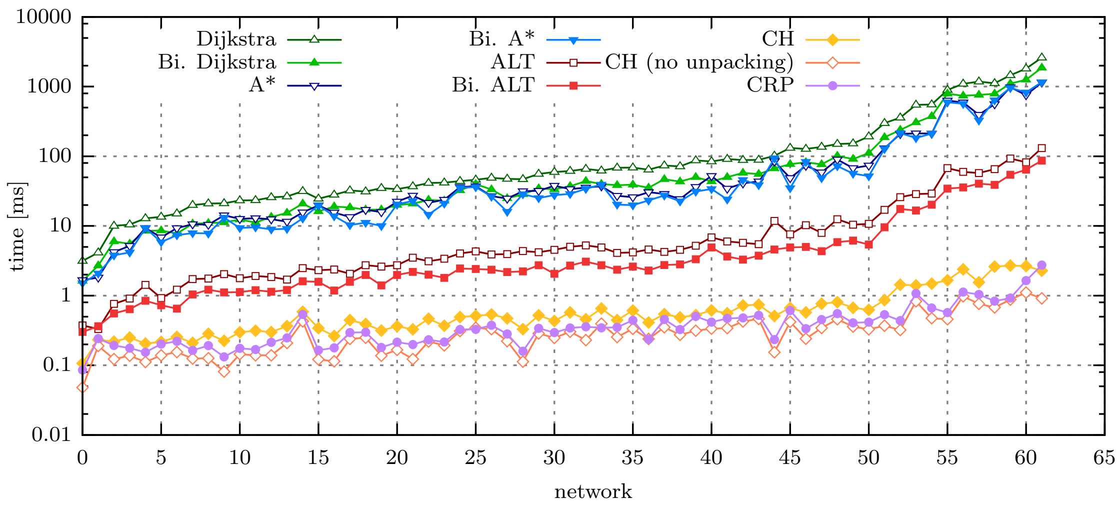


Figure 5.1: Average query times for all **full networks** with **geometric distances**. Sorted by the networks' node count in ascending order.

In Figure 5.1, the average query times for full networks with geometric distances are shown. All algorithms have in common that there is an upward trend in query times with increasing graph size, with CH and CRP being less affected. The bidirectional versions of Dijkstra and ALT are always faster than the unidirectional one except for very small graphs (less than 100 000 nodes), but the difference at those graphs is minimal.

The results are not as clear for A* where the bidirectional version is up to 12% slower on eleven cases which are spread independently of the networks size and affect graphs of distinctively different topologies. Furthermore, there are nine cases where the bidirectional version of Dijkstra is slightly faster than bidirectional A*. Eight of those nine graphs consist of several bays and coves that cut through the road network which renders the heuristic useless during many searches while the computational overhead remains (see Section 4.5 on page 53 for a detailed discussion about A*'s overhead). An example of such a search is shown in Figure 5.2 on the network of Seattle. The bidirectional search with A* settles about the same amount of nodes as with bidirectional Dijkstra (767 517 and 778 390 nodes respectively) but requires 347 ms compared to Dijkstra's 156 ms.

The query times for CRP are similar or slightly better than those for CH. Both algorithms follow the same trend on most graphs but there is a discrepancy on some.

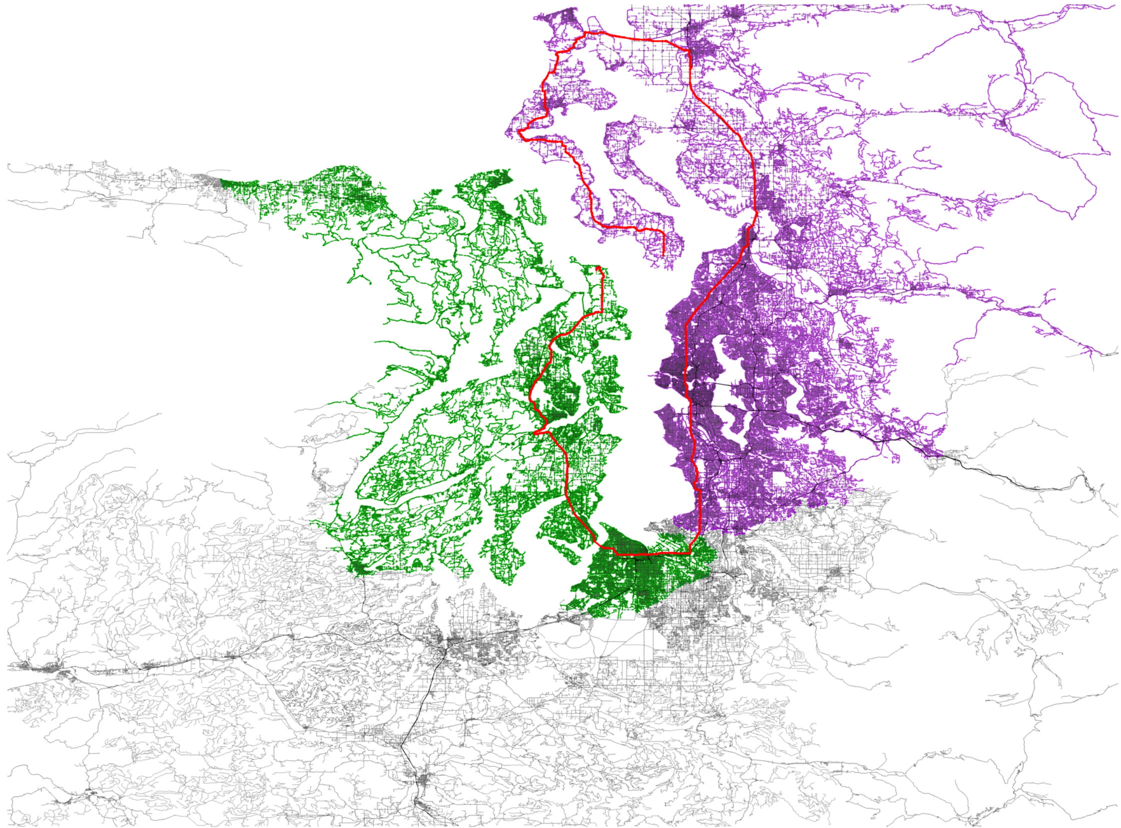


Figure 5.2: Bidirectional A* search on Seattle (289 × 214 km) with **geometric distances**. Settled nodes are drawn in green for the forward search and in purple for the backward search. Nodes touched by both searches are drawn in blue. The shortest path is shown in red.

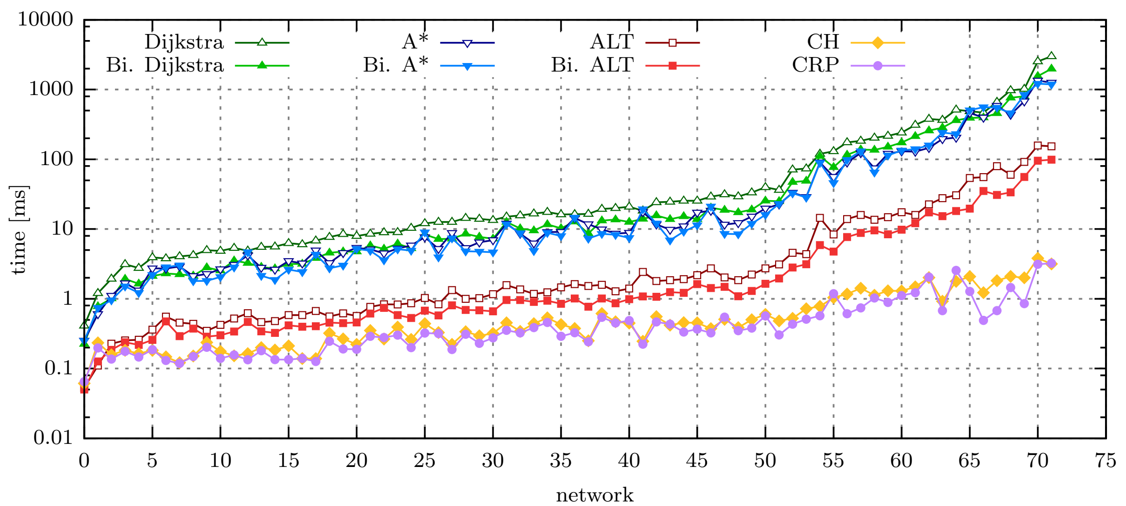


Figure 5.3: Average query times for all **contracted networks** with **geometric distances**. Sorted by the networks' node count in ascending order.

Disabling the path unpacking for CH leads to almost identical results for all networks, compared to CRP, and the discrepancies disappear. The query times for CRP without path unpacking, which are not shown, only differ by an insignificant amount.

In Figure 5.3, the average query times for contracted networks with geometric distances are shown. A contracted network has on average 4.8 times (80%) fewer nodes than its full version. The reduction in nodes translates very well to the performance of Dijkstra. The bidirectional version of Dijkstra is on average 82% faster than on full networks. A* also benefits from the reduced sizes but not as much. This leads to five more cases for a total of sixteen where the bidirectional version of A* is slower than bidirectional Dijkstra. The query times of ALT are 78% faster on average.

CH and CRP show a different picture. CH is only 37% faster and CRP performs almost the same with 12% faster queries on average. Due to the contraction, the graphs are smaller and fewer shortcuts have to be created which removes most of the cost of path unpacking. The results without path unpacking are therefore omitted because there is no noteworthy difference. That the performance of CH and CRP only increases by a small amount is not a disadvantage though. Actually it shows that they can deal with larger and uncontracted graphs much better than the other algorithms.

One last thing to mention is how close the query times of ALT and CH/CRP are on smaller graphs while the gap increases significantly with larger ones.

5.2.2 Travel Times

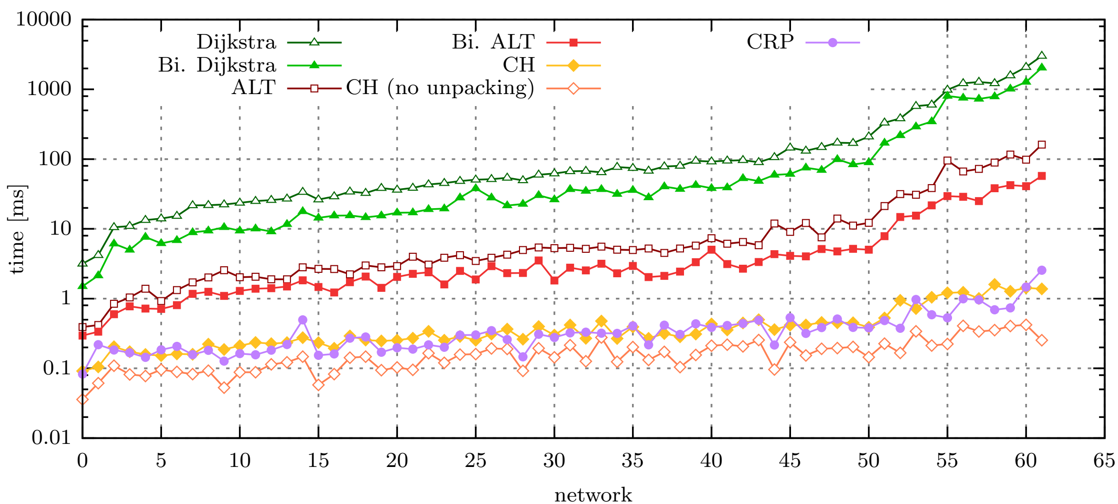


Figure 5.4: Average query times for all **full networks** with **travel times**. Sorted by the networks' node count in ascending order.

Figure 5.4 shows the average query times on full networks with travel times. The performance for each network is similar to the one with geometric distances as metric. The unidirectional algorithms are slightly slower, with Dijkstra losing 11% and ALT

21%. All other algorithms perform (slightly) better. Bidirectional Dijkstra is 2%, ALT 18% and CRP 9% faster. Only CH benefits significantly with 37% faster query times which brings its performance close to that of CRP. The query times for CH without path unpacking benefit equally which makes them faster than CRP's on all networks. The difference for CRP without path unpacking is insignificant and not shown.

A close inspection shows that bidirectional Dijkstra is slightly faster on small and medium sized graphs with travel times (compared to geometric distances) but the improvement vanishes at the larger ones. Bidirectional ALT on the contrary is actually slower on most smaller graphs and only starts pulling ahead on the larger ones.

The results on the contracted networks, which are shown in Figure 5.5, are likewise. Only CH is even faster with an improvement of 51% compared to using geometric distances which makes CH faster than CRP on most networks. Switching from the full to the contracted networks with travel times shows similar gains for all algorithms as with geometric distances. Only CH gains noticeable more with 48% faster queries.

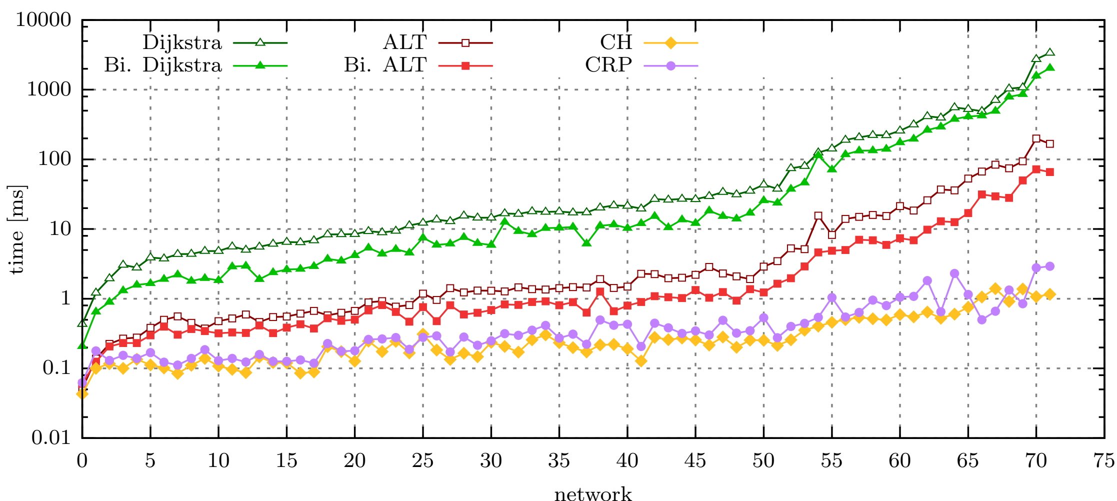
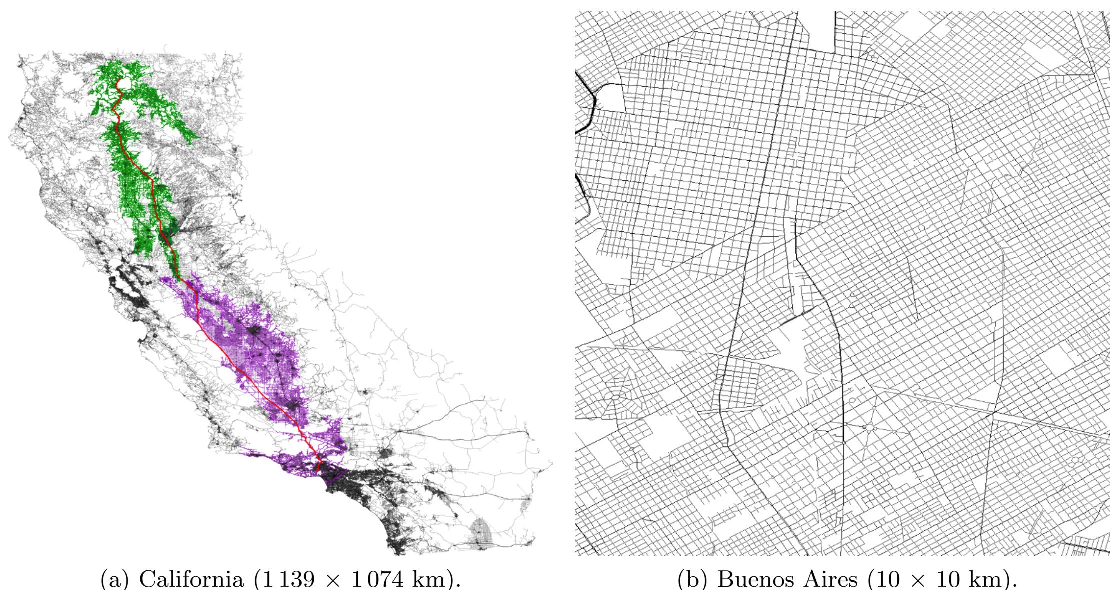


Figure 5.5: Average query times for all **contracted networks** with **travel times**. Sorted by the networks' node count in ascending order.

5.2.3 Conclusion

In general, the query times only correlate with the network size for all algorithms. This means that the topologies of the tested road networks from all over the world follow similar patterns. However, a few exceptions with special properties exist which affect the algorithms to a different degree. CRP and to some degree CH benefit from graphs with many bays and coves such as Seattle (Figure 5.2 on page 100), San Francisco and Stockholm because it increases the number of natural cuts. A* on the contrary performs worse on them because the heuristic is not designed for it. ALT also has problems with graphs that are too fragmented because the number of landmarks is too small to

cover all areas. Some algorithms (mainly bidirectional Dijkstra, A* and unidirectional ALT) have higher query times on narrow graphs such as the road network of California (Figure 5.6a), although the node efficiency is similar to other networks of equal size. The network of Buenos Aires, of which a section is shown in Figure 5.6b, has a dominant grid layout which has a notable impact on the performance of CH and CRP, which are not designed for grids.



(a) California (1 139 × 1 074 km).

(b) Buenos Aires (10 × 10 km).

Figure 5.6: Road network of California with a bidirectional A* search and a section from the road network of Buenos Aires.

CH and CRP dominate the other algorithms on full and contracted networks with geometric distances and travel times. The bidirectional versions of Dijkstra and ALT are always faster than the unidirectional ones. A* does not benefit as much from the bidirectional version and in some cases it is even slower. In most cases A* is faster than bidirectional Dijkstra, but the difference is small. On smaller, contracted networks the performance of bidirectional ALT comes close to CH/CRP with a smaller gap using geometric distances than travel times. Switching from contracted to full networks has a much smaller impact on CH and CRP than on the other algorithms. Only the performance of CH increases significantly by using travel times compared to geometric distances.

Only CH and CRP are fast enough to answer thousands of queries per second even on the largest graphs. On small and medium sized graphs, ALT may be an alternative if not as many queries are required. Dijkstra and A* are only fast enough on the smallest graphs or for single queries.

5.3 Node Efficiency

The node efficiency is a useful indicator for the theoretical performance and can be used to compare algorithms independently from implementation and hardware. It is calculated as the percentage of the number of nodes that have been settled in relation to the number of nodes in the shortest path.

5.3.1 Full Networks

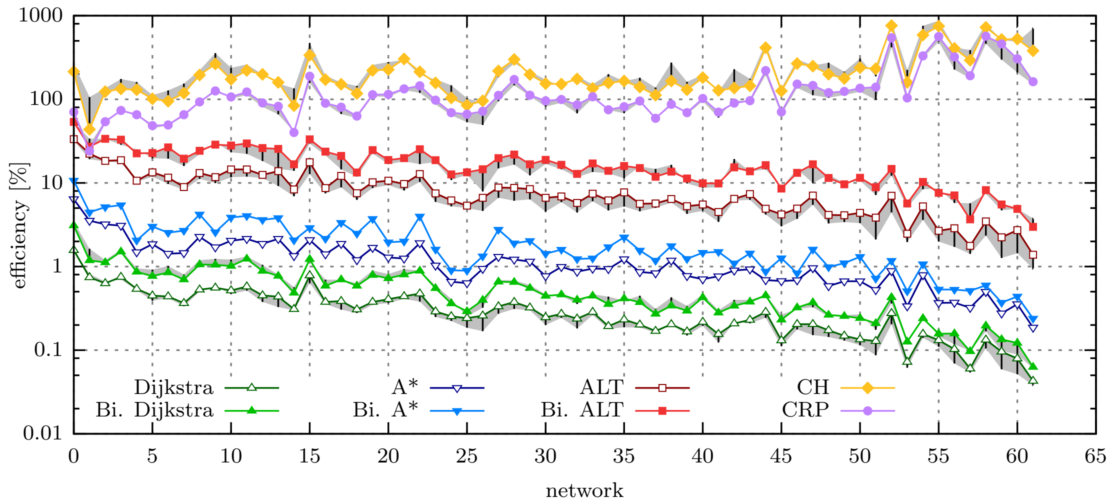
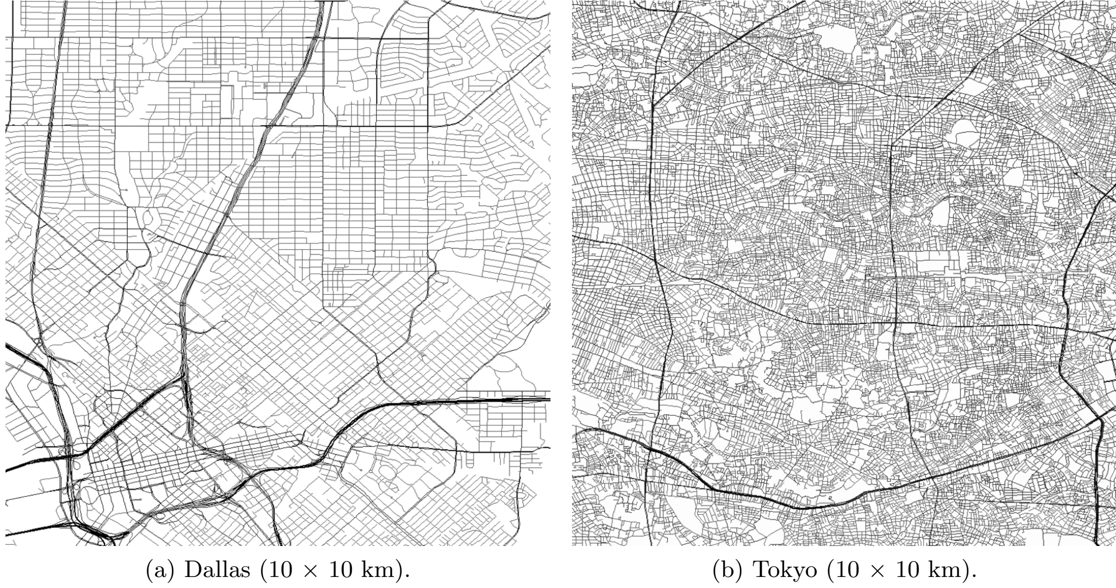


Figure 5.7: Average node efficiency for all **full networks** with **geometric distances** and **travel times**. The results with travel times are shown as offsets with black bars originating from the results with geometric distances. The gap between both results is additionally shaded in gray for better visibility. Sorted by the networks' node count in ascending order.

In Figure 5.7, the node efficiency for full networks with geometric distances is shown. Because the results with travel times are very similar, they are also shown as offsets in the same figure. CH has the highest efficiency followed closely by CRP. Both algorithms achieve very high percentages because they contract the underlying network. While the efficiency of all other algorithms drops with increasing network size, it grows with CH and CRP, reaching top values of 870% and 569%, respectively. All bidirectional versions of the algorithms perform better than the unidirectional ones, including A*. While the query times of A* are close to those of bidirectional Dijkstra, the node efficiency is noticeable better. On many networks the results of all algorithms follow the same trend, but there is some variation. The efficiency with travel times, compared to geometric distances, is in most cases slightly worse, except for CH, where it is slightly better.

There are some networks that stand out with worse efficiencies such as Buenos Aires (Figure 5.6b), Dallas (Figure 5.8a) and Tokyo (Figure 5.8b). All three networks are dense and well connected. Other networks such as Cyprus, Austria and Poland achieve



(a) Dallas (10 × 10 km). (b) Tokyo (10 × 10 km).

Figure 5.8: Road network section of Dallas and Tokyo.

comparatively higher efficiencies but this does not mean that the query times are better too. Using the example of Cyprus, the query times of CH and CRP are slightly faster on Cyprus and slower on Buenos Aires, while ALT performs equally on both and A* has slower queries on Cyprus compared to similar sized graphs.

5.3.2 Contracted Networks

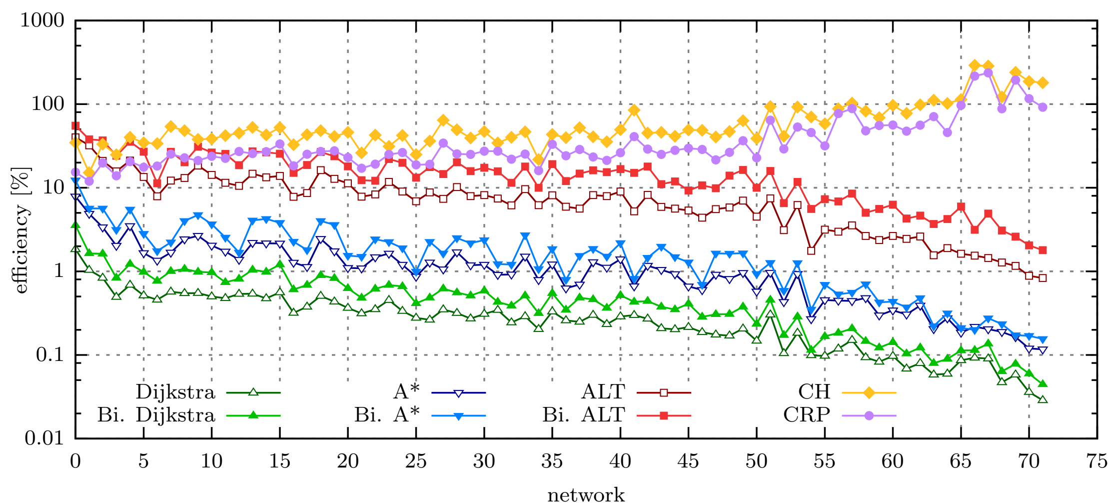
The node efficiency on contracted networks with geometric distances and travel times is shown in Figure 5.9. Compared to full networks, there are several differences. First of all, the efficiency of CH and CRP drops by 79% for geometric distances and 84% for travel times which correlates with the average network size reduction of 80%. The drop was to be expected because the advantage of contracting the network during preprocessing is gone. This brings the node efficiency of CH and CRP much closer to that of bidirectional ALT on small and medium sized graphs and in a few cases even below.

While there was only a small difference between geometric distances and travel times on full networks, it is more pronounced on the contracted ones. For a better illustration, the difference between both metrics and full and contracted networks is shown for some algorithms in Figure 5.10. The efficiency with travel times on contracted networks (drawn in blue) is below geometric distances (drawn in green) in most cases, even with CH. The figure also reveals that only CH and CRP see a significant efficiency drop between full and contracted networks while the others do not.

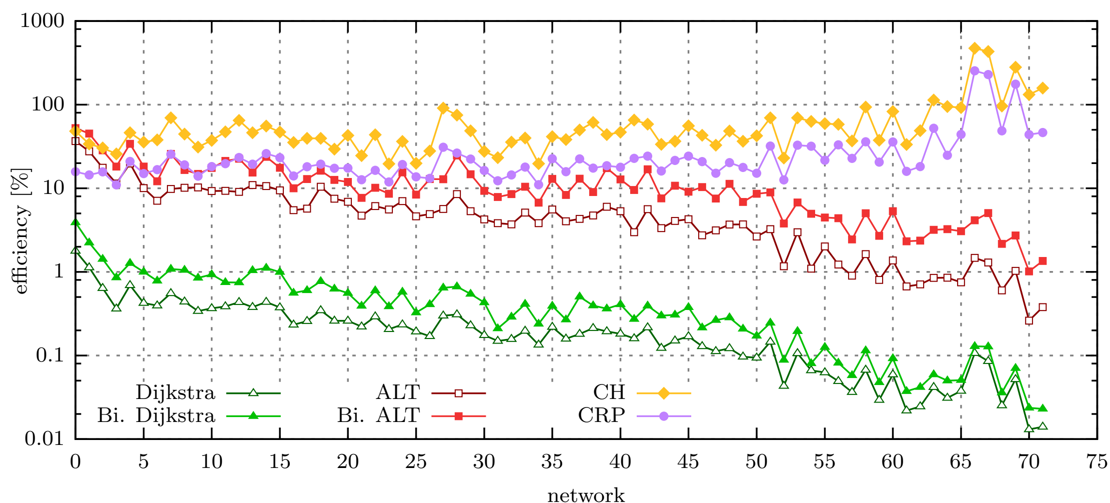
There is no clear picture how the contraction affects the efficiency. Some graphs that did not do as good as before, such as Buenos Aires, show a much better performance now, compared to other contracted networks. Others that had a good performance such

as Austria stayed the same while Cyprus dropped to average and others again that did not stick out before are now worse, such as Shanghai or Hong Kong.

Two things that did not change are the slight upward trend with increasing network size with CH and CRP, while it drops with the other algorithms, and that the bidirectional versions perform better than the unidirectional ones, albeit not as much.



(a) Average node efficiency on contracted networks with geometric distances.

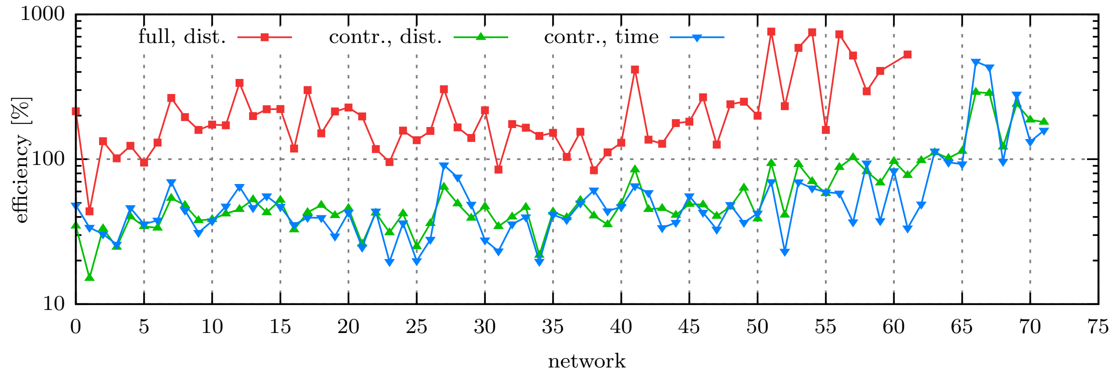


(b) Average node efficiency on contracted networks with travel times.

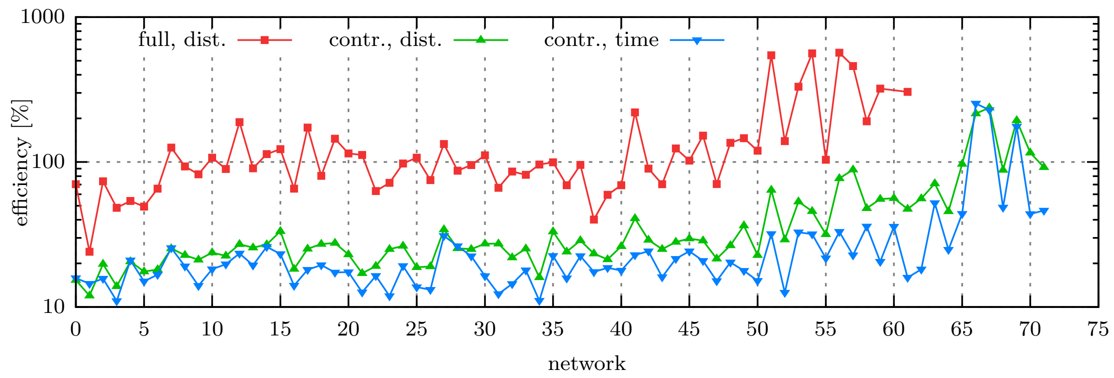
Figure 5.9: Average node efficiency for all **contracted networks** with **geometric distances** and **travel times**. Sorted by the networks' node count in ascending order.

5.3.3 Conclusion

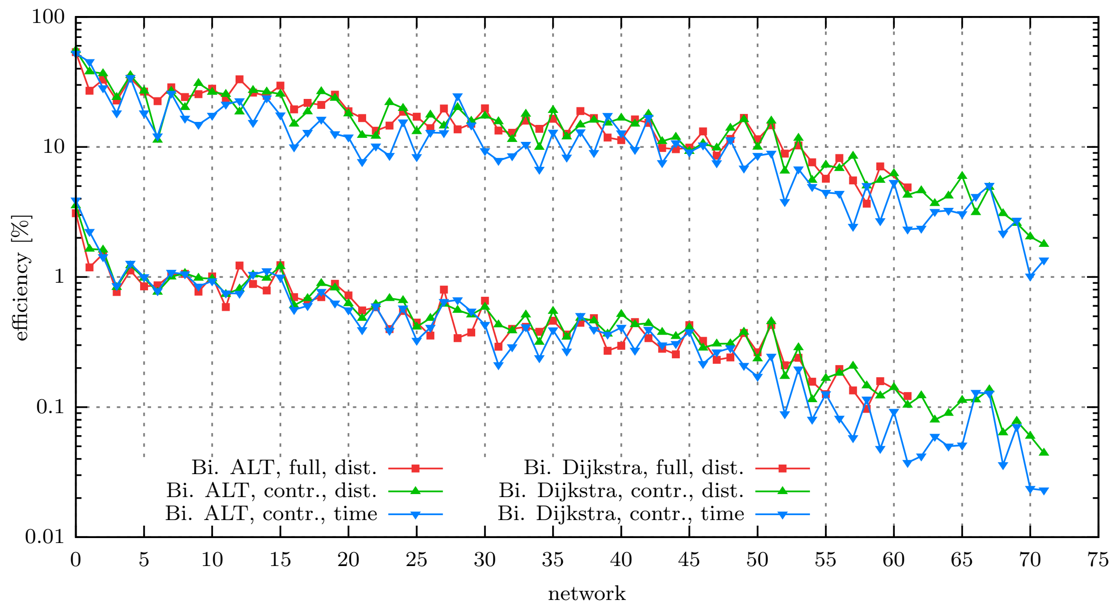
Compared to query times, there is more variation in the node efficiency with all algorithms. As has been shown, it is not possible to conclude from the efficiency to the actual query times. On some networks the query times may even be worse, relatively,



(a) Average node efficiency comparison with CH.



(b) Average node efficiency comparison with CRP.



(c) Average node efficiency comparison with bidirectional ALT (top) and bidirectional Dijkstra (bottom).

Figure 5.10: Average node efficiency comparison for all **contracted networks** between **geometric distances** and **travel times**. Sorted by the networks' node count in ascending order.

even though the efficiency is high. Switching between full and contracted networks affects algorithms that contract the graph in some way during preprocessing, such as CH and CRP, significantly, while other algorithms only see small differences. Using travel times leads to a lower efficiency in most cases, compared to geometric distances. Only CH can achieve better results with travel times that are noteworthy in some cases. Interestingly enough, the efficiency grows with CH and CRP, while it drops with increasing network size for all other algorithms. Also, the bidirectional algorithms always have a better efficiency than the unidirectional ones, although not to the same amount for each algorithm and metric (with the exception of the contracted network of Africa, where bidirectional A* is actually slightly worse).

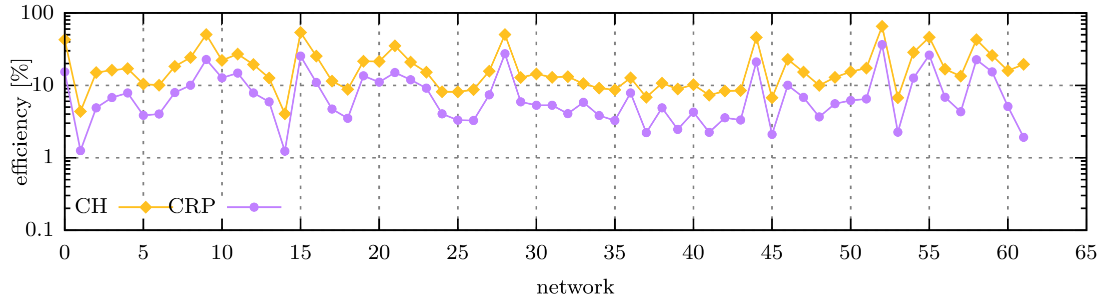
5.4 Edge Efficiency

The problem with the node efficiency is that it is biased towards CH and CRP. Both algorithms achieve high node efficiencies through the addition of new edges (shortcuts in CH and cell cliques in CRP) which reduce the number of settled nodes. A possible metric that reflects this is the edge efficiency. It is calculated as the percentage of the number of edges that have been scanned in relation to the number of edges in the shortest path. Please note the difference that the node efficiency counts every settled node only once, while the edge efficiency counts the same edge as often as it was accessed. Going through the boundary nodes/clique of a cell in CRP also counts as edge scans.

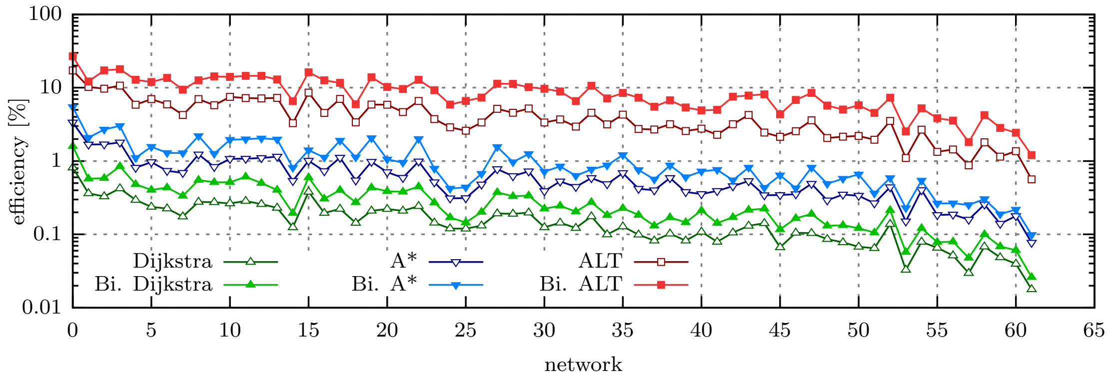
The edge efficiency for full and contracted networks with geometric distances is shown in Figure 5.11. As expected, the edge efficiency is lower than the node efficiency for all algorithms, with CH and CRP taking a significantly larger hit. Interestingly enough, the course of the curves is almost identical between node and edge efficiency for all algorithms except CH and CRP, where it is also similar but amplified. However, the slight upward trend in node efficiency with larger networks that was noticeable with CH and CRP is gone. On most contracted networks, bidirectional ALT is now more efficient than CH and CRP. The results with travel times follow the same pattern as with geometric distances and can be found in the appendix (Section B.2 on page 130).

5.4.1 Conclusion

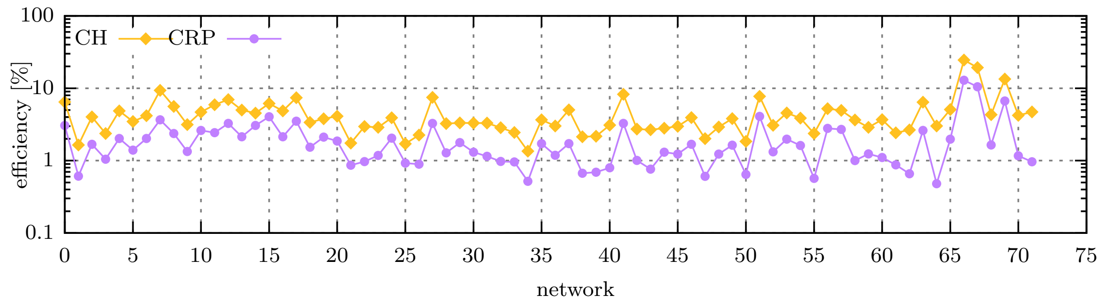
Algorithms such as CH and CRP increase the number of edges to reduce the number of nodes that have to be settled. By using the edge efficiency to compare the work that is done by the different algorithms, the cost that is hidden from the node efficiency is revealed, i.e. the increased amount of edges that have to be processed. However, the performance of the networks themselves stays the same, relatively. This includes switching between full and contracted networks or travel times and geometric distances, which also leads to similar results, whether viewed with node or edge efficiency.



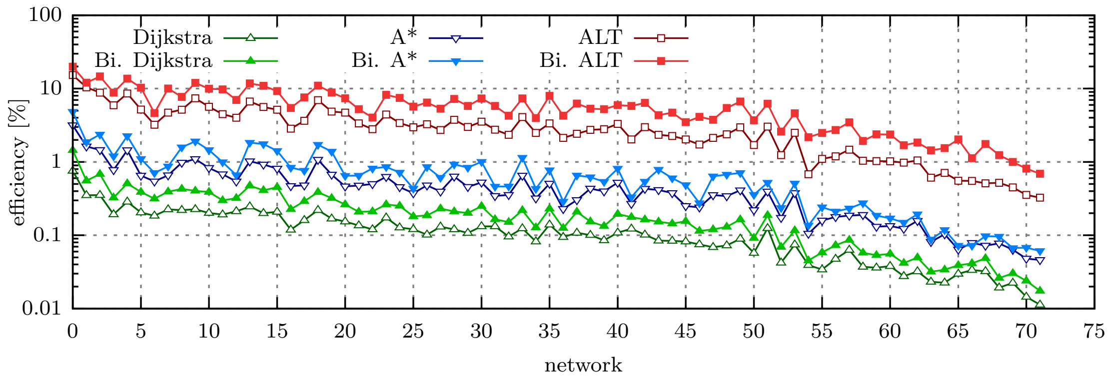
(a) Average edge efficiency on full networks with geometric distances.



(b) Average edge efficiency on full networks with geometric distances.



(c) Average edge efficiency on contracted networks with geometric distances.



(d) Average edge efficiency on contracted networks with geometric distances.

Figure 5.11: Average edge efficiency for all **full and contracted networks** with **geometric distances**. Sorted by the networks' node count in ascending order.

The results also unveil that the bidirectional ALT algorithm can compete with CH and CRP, at least theoretically. In practice it loses because CH and CRP do not have the computational overhead of the heuristic and their memory access pattern is much more cache friendly. The reason for this is that the edges/shortcuts of a node and the cliques of cells are stored next to each other in memory. ALT on the contrary has to scan more nodes and has to get the distances from and to landmarks for every node, which leads to more non-continuous memory accesses.

5.5 Preprocessing

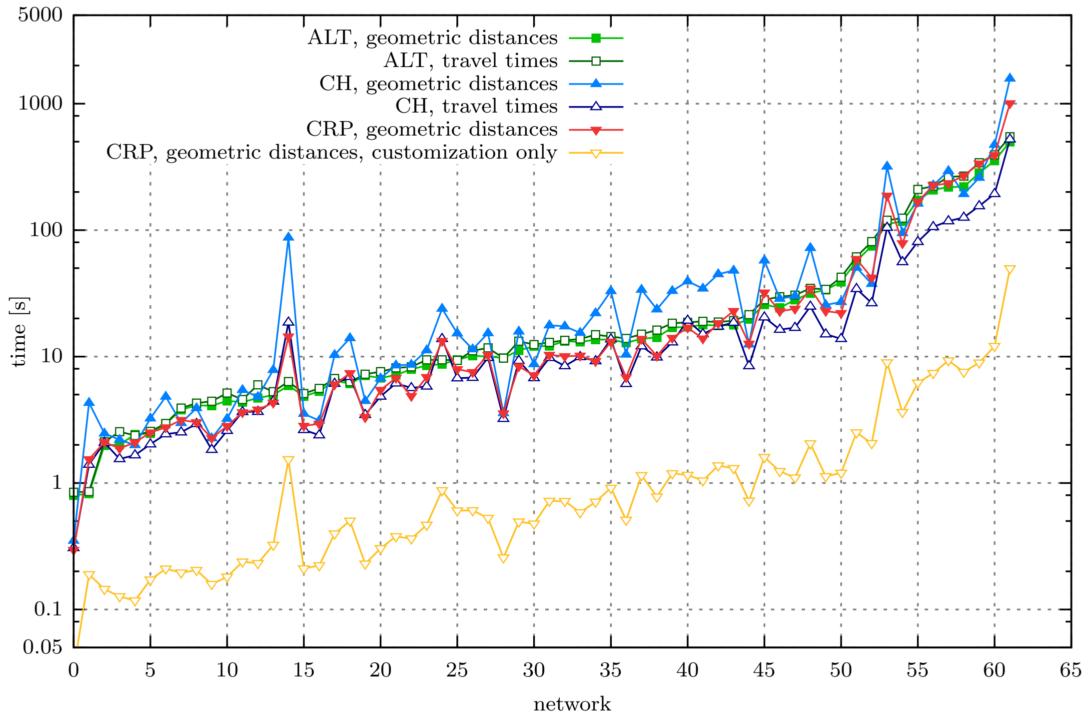
The time required for preprocessing plays an important part in assessing whether an algorithm is fast enough for dynamic networks. Dijkstra and A* have the advantage of not requiring preprocessing which means the network can change completely between each query. The other algorithms on the contrary require certain data or modifications which have to be prepared before a search query can be executed. During the preprocessing of ALT with the modified Partition-Corners algorithm the graph is partitioned, landmarks are set and the distance in both directions from every node to every landmark is computed. CH creates a hierarchical node ordering and the graph is contracted by adding shortcuts. And the preprocessing of CRP partitions the graph by natural cuts, creates an overlay and the cliques (shortest paths between all boundary nodes) for all cells are computed.

To give an undistorted overview of the actual work done by the different algorithms during preprocessing, the sequential performance without reusing any previously computed parts will be inspected. It should be noted that there are various additional optimizations available for all three algorithms which have not been implemented. These optimizations can lead to speedups of two to four times but even if implemented the overall outcome would stay the same.

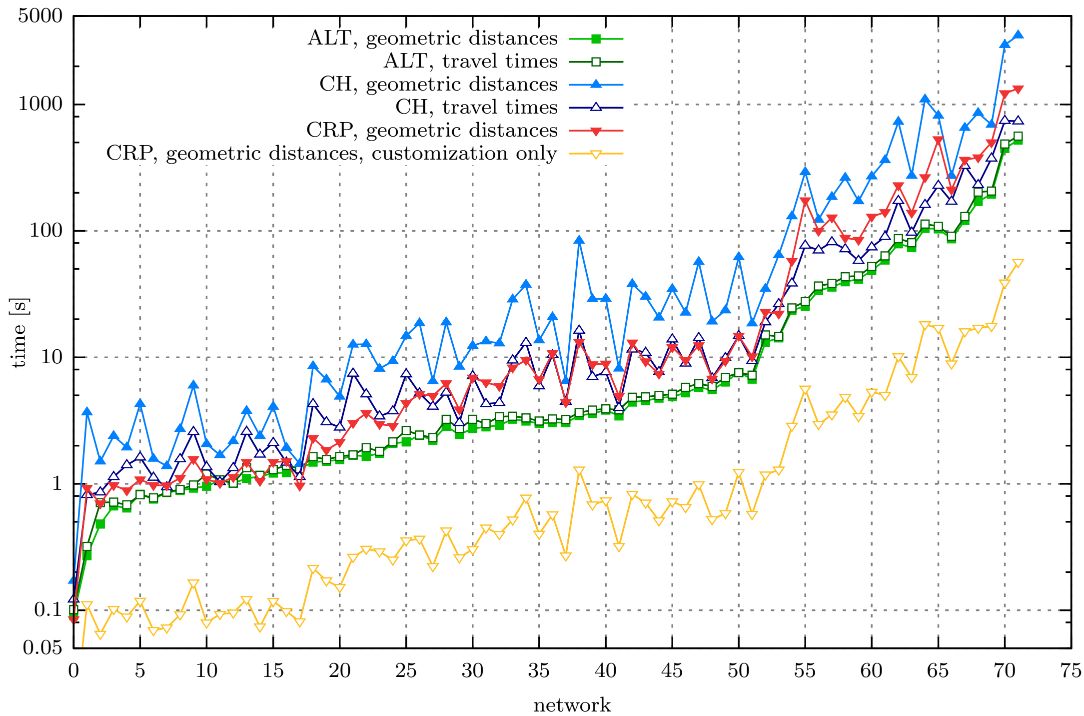
The time required for sequential preprocessing for all full networks with both metrics is shown in Figure 5.12a. For CRP, only the results with geometric distances are shown because they are almost identical with those using travel times.

The preprocessing time with ALT (with 36 landmarks) grows steadily with increasing network size, almost unaffected by the different topologies of the various networks. The difference between the metrics is minimal, with travel times taking 8.9% longer on average and 27% in the worst case. The partitioning phase with ALT (which is not shown separately) is negligible as it takes less than a second on most graphs and only a few seconds on the largest. The vast majority of time is spent with the landmark distance computations.

CH with geometric distances has the worst performance in most cases. It also has the highest sensitivity to the topology which leads to strongly varying results in regard to the network size. With travel times, the performance is significantly better and in many



(a) Sequential preprocessing time on full networks.



(b) Sequential preprocessing time on contracted networks.

Figure 5.12: Sequential preprocessing time for **all networks** with **geometric distances** and **travel times**. Sorted by the networks' node count in ascending order. The results for CRP with geometric distances and travel times are almost identical and therefore the latter are not shown.

cases even the fastest, especially on larger graphs. Furthermore, CH is less sensitive to the topology with travel times.

The graph partitioning with CRP is metric independent which means it does not have to be repeated after the graph has changed. Only during the customization phase the edge weights are used to create the cell cliques. Because of that the time required for the customization only is also shown in the figure. Even though the preprocessing of CRP and CH work differently, both are affected similarly by the topology. On most graphs CRP is faster than ALT and CH with geometric distances, except on the largest. On small and medium sized graphs CRP can keep up with CH with travel times but its performance drops significantly on larger graphs in comparison. However, the customization phase of CRP is an order of magnitude faster than any other algorithm. The time required for customization is equally affected by the topology because the number of boundary nodes created correlates with it, i.e. layouts with a stronger hierarchy result in fewer boundary nodes and therefore less computations during customization (see Figure 5.13). The metric has almost no influence on the customization with a difference between geometric distances and travel times of only ±4% at most.

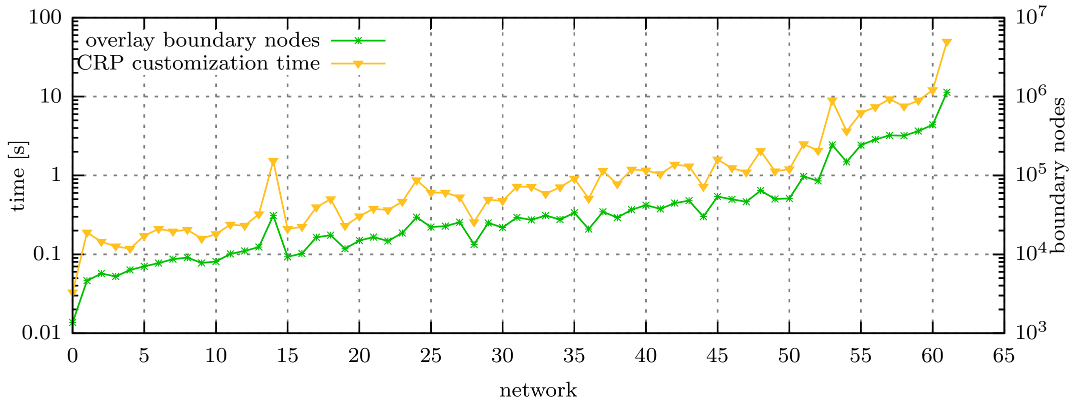


Figure 5.13: Comparison of the time required for the customization phase of CRP and the number of boundary nodes created. Sorted by the networks' node count in ascending order.

The networks where CH and CRP perform comparatively better or worse are similar to those mentioned before at the node efficiency comparison (see Section 5.3 on page 104). By far the worst performance is seen on Buenos Aires followed by other dense networks such as Tokyo and Dallas. Both algorithms are fast on networks with a strong hierarchy (which also tend to have many natural cuts) such as Atlanta, Austria and Seattle.

The results for sequential preprocessing on contracted networks, which are shown in Figure 5.12b, give a similar picture with only a few differences. On contracted networks, ALT is faster than CH and CRP in almost all cases. The time required with CH and geometric distances is more unsteady between networks and slower than any other algorithm. Also, CH with travel times is now slower than CRP on small networks.

5.5.1 Optimizations

Not every step of preprocessing has to be repeated after the edge weights of a network change. However, it depends on the algorithm which parts can be reused and thus how much time is saved. Preprocessing times can also be reduced at the cost of degraded query performance. For example an aggressive implementation of CH shows a speedup of 2.5 times on the contracted network of Europe, compared to the one in this framework.

The biggest gains though can be achieved by parallelization. For CH, Vetter achieves a speedup of seven times with sixteen threads in [67] but the gains start diminishing significantly around eight to ten threads. In [11], Delling et al. reduce the time required for partitioning with CRP by a factor of 3.7 with twelve threads and in [9], Delling et al. achieve a speedup of ten times with twelve threads for the customization phase. Similar to CH, the parallelization with CRP has diminishing returns with increasing thread count. The main part of ALT's preprocessing is the computation of the distances between nodes and landmarks which can be parallelized easily. Efentakis et al. achieve an improvement of 5.6 times with eight threads on a system with four cores and hyperthreading in [17].

5.5.2 Conclusion

Overall, the preprocessing of CH performs worst. It is the only algorithm that is influenced significantly by the chosen metric and it is affected the most by the topology of the networks. While CRP is also affected by the topology, its results are more steady and the influence by the metric is negligible. The preprocessing of ALT is the most stable and almost unaffected by the topology or metrics. Its preprocessing times are fast and ahead of the other algorithms on contracted networks.

For small networks the preprocessing of all algorithms stays clearly below ten seconds and is fast enough for dynamic networks. The same is true for most medium sized graphs if the improvements through parallelization, which have been mentioned above, are taken into account. However, on the largest graphs, where preprocessing starts taking minutes, even those improvements are not sufficient, except for CRP. Because only the customization has to be repeated with CRP, which is an order of magnitude faster than the partitioning, the required time can still be brought down to a few seconds. This makes CRP the only algorithm that is fast enough for continental sized dynamic networks where edge weights have to be updated every few seconds.

Several different preprocessing methods exist for ALT to find suitable landmarks (see Section 2.8.2 on page 14). While some of the best methods such as *maxCover* can find landmarks that improve the node efficiency by up to 2.5 times compared to the worst, the preprocessing time increases by a factor of ten (see [29, Section 8.2.1]). Furthermore, even with those better landmarks or highly optimized algorithms (like the one used by Efentakis in [17]) the query times with ALT are still at least an order of magnitude

slower compared to CH and CRP (see [17] for the best results achieved with ALT). The problem is that ALT is not a useful alternative to CH or CRP if its preprocessing and query times are significantly worse. For this reason a fast preprocessing method was chosen in this work and the results confirm this choice. On small and medium sized graphs, the preprocessing time of ALT is competitive and stays below or near ten seconds (and can be improved easily by parallelization). Another advantage of ALT, that has been revealed by the results, is that its preprocessing is extremely stable and almost unaffected by the topology. If the slower query times of ALT can be tolerated and the modified Partition-Corners algorithm for preprocessing is used, it is an alternative to CH and CRP on small and medium sized graphs. ALT can also be implemented and optimized much easier.

Chapter 6

Conclusion

Several algorithms from different categories and complexity were presented, analyzed and compared. Instead of just focusing on theoretical differences or final performance numbers, the actual behavior in a practical setup was investigated as well. It was shown that Dijkstra's algorithm expands differently with travel times than with geometric distances. It turned out that the travel time metric adds a hierarchy to the graph which affects Dijkstra's algorithm, even though it is not designed to utilize hierarchical information. Interestingly enough, though, the average number of nodes settled was the same between both metrics. The results also revealed that the node efficiency is not suited to compare the performance between different metrics. Even though the same work was done with both metrics, the results with travel times showed a lower efficiency. The reason for this is that quickest paths tend to have fewer nodes because they prefer road types with fewer junctions (nodes) such as motorways. It was seen that these behavioral patterns of Dijkstra's algorithm reflect themselves in the other algorithms which utilize it.

The hierarchical structure added by the travel time metric affected the search pattern but had no influence on the performance of the unidirectional version of Dijkstra's algorithm. On the contrary, the bidirectional version was able to benefit from the added hierarchy and settled fewer nodes with travel times. It was also shown that all algorithms benefit from using a forward and backward search, independently of the metric.

The preprocessing of CH and CRP was explained and illustrated in great detail. It was discovered that the graph created by CH has similar properties to a scale-free network with high degree hubs. This also turns out to be the reason why its query times are so fast, because search queries can take advantage of the small-world effect. With CRP it was investigated what constitutes as a natural cut from a practical standpoint. It was shown that natural cuts differ depending on the observed scale and on smaller scales are often man-made and not natural, such as railways, airports or industrial zones. Furthermore, it was seen that areas with a sparser network surrounding denser parts such as towns can also be a natural cut and that on the smallest scales, the concept of natural

cuts started reaching its limits.

Even though the preprocessing phases of CH and CRP work completely different, due to the large number of road networks tested, it was revealed that they are both equally affected by a network's topology, i.e. they both need more or less preprocessing time on the same graph. Also, the results showed that the preprocessing performance reflects itself in the query times.

A large amount of networks from all over the world and with different scales, ranging from cities to continents were used to see if there are notable differences. The evaluation showed that most road networks have a similar topology with only a few networks such as Buenos Aires standing out. The results where more influenced by natural constraints such as bays and coves and networks being forced into a narrow layout. Furthermore, the difference between full and contracted networks was investigated. Search queries with algorithms that contract the network in some form during preprocessing, such as CH and CRP, had a much smaller performance penalty on full networks.

The final performance comparison showed that only CRP is fast enough for dynamic networks of continental size with frequent updates and that ALT can be an alternative for small and medium sized networks, but only if a fast preprocessing method is used. While CH has one of the fastest query times, it suffers from its metric dependent preprocessing on large networks. The results also revealed that the influence of the hardware and implementation can not be disregarded. For example while A* settled significantly fewer nodes than Dijkstra's algorithm, its query times were only slightly better due to its computational overhead.

6.1 Future Work

There are many interesting aspects left that have not been touched by this work. The results from A* and ALT showed how additional memory accesses and computations can have a large impact on the performance. It would be interesting to have a closer look at possible optimizations for the graph data structure to reduce those impacts.

Also, the computation of cliques for CRP was done on the CPU, but as was mentioned during the theoretical overview, it is also possible to compute cliques with the Floyd-Warshall algorithm on a graphics card (GPU). The fastest graphics card currently available is the NVIDIA Titan X, which has 12 GB memory (with a bandwidth of 480 GB/s) and achieves 11 TFLOPS (11×10^{12} floating point operations per second) with FP32 (single precision floating point format) [47]. With such a large amount of memory and computational power, it should be possible to further reduce the time taken by the customization phase of CRP.

Bibliography

[1] Milton Abramowitz and Irene A. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. ninth Dover printing, tenth GPO printing. New York: Dover, 1964.

[2] Alfred V. Aho, J. D. Ullman and John E. Hopcroft. *The Design and Analysis of Computer Algorithms*. Reading, MA: Addison-Wesley, 1974. ISBN: 0-201-00029-6.

[3] Hannah Bast et al. 'Route Planning in Transportation Networks'. In: *Algorithm Engineering: Selected Results and Surveys*. Ed. by Lasse Kliemann and Peter Sanders. Springer International Publishing, 2016, pp. 19–80. ISBN: 978-3-319-49487-6.

[4] Richard Bellman. 'On a Routing Problem'. In: *Quarterly of Applied Mathematics* 16 (1958), pp. 87–90.

[5] Michael A. Bender, Erik D. Demaine and Martin Farach-Colton. 'Cache-Oblivious B-Trees'. In: *SIAM J. Comput.* 35.2 (Aug. 2005), pp. 341–358. ISSN: 0097-5397.

[6] M. Chen et al. *Priority Queues and Dijkstra's Algorithm*. Tech. rep. MSR-TR-2004-24. Austin, Texas: The University of Texas at Austin, Department of Computer Sciences, Oct. 2007.

[7] European Commission. *Galileo goes live!* 2016. URL: http://europa.eu/rapid/press-release_IP-16-4366_en.htm (visited on 21/12/2016).

[8] George B. Dantzig. 'On the shortest path route through a network'. In: *Management Science* 6 (1960), pp. 187–190.

[9] Daniel Delling et al. 'Customizable Route Planning.' In: *SEA*. Ed. by Panos M. Pardalos and Steffen Rebennack. Vol. 6630. Lecture Notes in Computer Science. Springer, 2011, pp. 376–387. ISBN: 978-3-642-20661-0.

[10] Daniel Delling et al. 'Customizable route planning in road networks.' submitted for publication. 2013. URL: http://research.microsoft.com/pubs/198358/crp_web_130724.pdf (visited on 24/09/2014).

[11] Daniel Delling et al. *Graph Partitioning with Natural Cuts*. Tech. rep. MSR-TR-2010-164. Dec. 2010. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=142349 (visited on 25/01/2015).

[12] Elena Deza and Michel Deza. *Dictionary of Distances.* North-Holland, 2006, pp. I–XV, 1–391. ISBN: 978-0-444-52087-6.

[13] E. W. Dijkstra. 'A note on two problems in connexion with graphs'. In: *Numerische Mathematik* 1 (1 1959). 10.1007/BF01386390, pp. 269–271. ISSN: 0029-599X.

[14] DIMACS. *9th DIMACS Implementation Challenge - Shortest Paths.* 2010. URL: http://www.dis.uniroma1.it/challenge9/download.shtml (visited on 20/07/2016).

[15] Ulrich Drepper. *What Every Programmer Should Know About Memory.* 2007. URL: https://www.akkadia.org/drepper/cpumemory.pdf (visited on 19/10/2016).

[16] Stuart E. Dreyfus. *An Appraisal of Some Shortest-Path Algorithms.* Tech. rep. RM-5433. Santa Monica, California: Rand Corporation, Aug. 1967.

[17] Alexandros Efentakis and Dieter Pfoser. 'Optimizing Landmark-Based Routing and Preprocessing'. In: *Proceedings of the Sixth ACM SIGSPATIAL International Workshop on Computational Transportation Science.* IWCTS '13. Orlando, FL, USA: ACM, 2013, 25:25–25:30. ISBN: 978-1-4503-2527-1.

[18] P.O. Fjallstrom. 'Algorithms for Graph Partitioning: A Survey'. In: *Linkoping Electronic Articles in Computer and Information Science* 3.10 (1998).

[19] Robert W. Floyd. 'Algorithm 97: Shortest path'. In: *Communications of the ACM* 5.6 (1962), p. 345.

[20] Michael L. Fredman and Robert Endre Tarjan. 'Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms'. In: *Journal of the ACM* 34.3 (July 1987), pp. 596–615. ISSN: 0004-5411.

[21] Fabian Fuchs. 'On Preprocessing the ALT-Algorithm'. Student Thesis. Faculty of Computer Science, Institute for Theoretical Informatics (ITI), Karlsruhe Institute of Technology, 2010.

[22] Robert Geisberger. 'Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks'. Diploma Thesis. Institut für Theoretische Informatik Universität Karlsruhe (TH), 2008.

[23] Robert Geisberger et al. 'Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks.' In: *WEA.* Ed. by Catherine C. McGeoch. Vol. 5038. Lecture Notes in Computer Science. Springer, 2008, pp. 319–333. ISBN: 978-3-540-68548-7.

[24] Geofabrik. *OpenStreetMap Data Extracts.* 2016. URL: http://download.geofabrik.de/ (visited on 09/22/2016).

[25] Andrew V. Goldberg, ed. *Summer School on Shortest Paths: Basic shortest path algorithms*. Microsoft Research, Silicon Valley, California. Department of Computer Science, University of Copenhagen, 2005. URL: http://www.diku.dk/PATH05/GoldbergSlides.pdf (visited on 17/09/2014).

[26] Andrew V. Goldberg and Chris Harrelson. *Computing the Shortest Path: A* Search Meets Graph Theory*. Tech. rep. MSR-TR-2004-24. Redmond, Washington: Microsoft Research, Mar. 2004.

[27] Andrew V. Goldberg and Chris Harrelson. 'Computing the shortest path: A* Search Meets Graph Theory'. In: *SODA*. SIAM, 2005, pp. 156–165. ISBN: 0-89871-585-7.

[28] Andrew V. Goldberg and Robert E. Tarjan. 'A New Approach to the Maximum-flow Problem'. In: *J. ACM* 35.4 (Oct. 1988), pp. 921–940. ISSN: 0004-5411.

[29] Andrew V. Goldberg and Renato F. Werneck. 'Computing Point-to-Point Shortest Paths from External Memory'. In: *ALENEX'05*. SIAM, 2005, pp. 26–40.

[30] Andrew V. Goldberg et al. *Efficient Point-to-Point Shortest Path Algorithms*. New Jersey, USA, Apr. 2006. URL: http://www.cs.princeton.edu/courses/archive/spr06/cos423/Handouts/EPP%20shortest%20path%20algorithms.pdf (visited on 02/09/2014).

[31] P.E. Hart, N.J. Nilsson and B. Raphael. 'A Formal Basis for the Heuristic Determination of Minimum Cost Paths'. In: *Systems Science and Cybernetics, IEEE Transactions on* 4.2 (July 1968), pp. 100–107. ISSN: 0536-1567.

[32] Takahiro Ikeda et al. 'A fast algorithm for finding better routes by AI search techniques'. In: *Vehicle Navigation and Information Systems Conference*. 1994, pp. 291–296.

[33] Peter Zilahy Ingerman. 'Algorithm 141: Path matrix'. In: *Communications of the ACM* 5.11 (1962), p. 556.

[34] Intel. *6th Generation Intel Core Processors Factsheet*. 2015. URL: http://download.intel.com/newsroom/kits/core/6thgen/pdfs/6th_Gen_Intel_Core-Intel_Xeon_Factsheet.pdf (visited on 26/11/2016).

[35] George Karypis and Vipin Kumar. 'A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs'. In: *SIAM J. Sci. Comput.* 20.1 (Dec. 1998), pp. 359–392. ISSN: 1064-8275.

[36] Tim Kieritz et al. 'Distributed Time-Dependent Contraction Hierarchies'. In: *Experimental Algorithms: 9th International Symposium, SEA 2010, Ischia Island, Naples, Italy, May 20-22, 2010. Proceedings*. Ed. by Paola Festa. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 83–93. ISBN: 978-3-642-13193-6.

[37] Richard J. Lipton and Robert E. Tarjan. 'A Separator Theorem for Planar Graphs'. In: *SIAM Journal on Applied Mathematics* 36.2 (1979), pp. 177–189.

[38] Dennis Luxen and Christian Vetter. 'Real-time Routing with OpenStreetMap Data'. In: *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. GIS '11. Chicago, Illinois: ACM, 2011, pp. 513–516. ISBN: 978-1-4503-1031-4.

[39] Georgia Mali et al. 'A New Dynamic Graph Structure for Large-Scale Transportation Networks'. In: *Algorithms and Complexity: 8th International Conference, CIAC 2013, Barcelona, Spain, May 22-24, 2013. Proceedings*. Ed. by Paul G. Spirakis and Maria" Serna. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 312–323. ISBN: 978-3-642-38233-8.

[40] Mapzen. *Metro Extracts*. 2016. URL: https://mapzen.com/data/metro-extracts/ (visited on 22/09/2016).

[41] G. Marsaglia. 'Random number generators'. In: *Journal of Modern Applied Statistical Methods* 2 (2003), pp. 2–13.

[42] Kazuya Matsumoto, Naohito Nakasato and Stanislav G. Sedukhin. 'Blocked All-Pairs Shortest Paths Algorithm for Hybrid CPU-GPU System.' In: *HPCC*. IEEE, 2011, pp. 145–152. ISBN: 978-1-4577-1564-8.

[43] G. Meyer and S. Beiker. *Road Vehicle Automation 2*. Lecture Notes in Mobility. Springer International Publishing, 2015. ISBN: 9783319190785.

[44] Microsoft. *Bing Maps New Routing Engine*. 2012. URL: https://blogs.bing.com/maps/2012/01/05/bing-maps-new-routing-engine (visited on 14/09/2016).

[45] M. E. J. Newman. 'The structure and function of complex networks'. In: *SIAM REVIEW* 45 (2003), pp. 167–256.

[46] T.A.J. Nicholson. 'Finding the shortest route between two points in a network'. In: *The Computer Journal* 6 (1966), pp. 275–280.

[47] NVIDIA. *Titan X*. 2016. URL: https://blogs.nvidia.com/blog/2016/07/21/titan-x/ (visited on 02/12/2016).

[48] OpenStreetMap. *OpenStreetMap Data Export*. 2016. URL: http://www.openstreetmap.org/export (visited on 22/09/2016).

[49] Judea Pearl. *Heuristics - intelligent search strategies for computer problem solving.* Addison-Wesley series in artificial intelligence. Addison-Wesley, 1984, pp. I–XVII, 1–382. ISBN: 978-0-201-05594-8.

[50] I. Pohl. *Bi-directional and Heuristic Search in Path Problems*. SLAC report. Department of Computer Science, Stanford University, 1969.

[51] I. Pohl. 'Bi-directional search'. In: *Machine Intelligence*. Ed. by B. Meltzer and D. Michie. Vol. 6. Edinburgh University Press, 1971. Chap. 9, pp. 127–140.

[52] David Pritchard and Ramakrishna Thurimella. 'Fast Computation of Small Cuts via Cycle Space Sampling'. In: *ACM Trans. Algorithms* 7.4 (Sept. 2011), 46:1–46:30. ISSN: 1549-6325.

[53] Peter Sanders. 'Fast Priority Queues for Cached Memory'. In: *J. Exp. Algorithmics* 5 (Dec. 2000). ISSN: 1084-6654.

[54] Peter Sanders and Dominik Schultes. 'Engineering Highway Hierarchies.' In: *ESA*. Ed. by Yossi Azar and Thomas Erlebach. Vol. 4168. Lecture Notes in Computer Science. Springer, 2006, pp. 804–816. ISBN: 3-540-38875-3.

[55] Peter Sanders and Christian Schulz. 'Distributed Evolutionary Graph Partitioning'. In: *Proceedings of the Meeting on Algorithm Engineering & Expermiments*. ALENEX '12. Kyoto, Japan: Society for Industrial and Applied Mathematics, 2012, pp. 16–29.

[56] Giuseppe M. L. Sarnè and Maria Nadia Postorino. 'Agents meet Traffic Simulation, Control and Management: A Review of Selected Recent Contributions'. In: *17th Workshop "From Objects to Agents"*. WOA, 2016.

[57] Dominik Schultes. 'Route Planning in Road Networks.' In: *Ausgezeichnete Informatikdissertationen*. 2008, pp. 271–280.

[58] Dominik Schultes and Peter Sanders. 'Dynamic Highway-Node Routing.' In: *WEA*. Ed. by Camil Demetrescu. Vol. 4525. Lecture Notes in Computer Science. Springer, 2007, pp. 66–79. ISBN: 978-3-540-72844-3.

[59] Daniel D. Sleator and Robert E. Tarjan. 'A data structure for dynamic trees'. In: *Journal of Computer and System Sciences* 26 (1983), pp. 362–391.

[60] Daniel D. Sleator and Robert E. Tarjan. 'Self-Adjusting Binary Search Trees'. In: *Journal of the ACM* 32.3 (1985), pp. 652–686.

[61] Sabine Storandt. 'Contraction Hierarchies on Grid Graphs.' In: *KI*. Ed. by Ingo J. Timm and Matthias Thimm. Vol. 8077. Lecture Notes in Computer Science. Springer, 2013, pp. 236–247. ISBN: 978-3-642-40941-7.

[62] Robert Tarjan. 'Depth first search and linear graph algorithms'. In: *SIAM Journal on Computing* (1972).

[63] Robert Endre Tarjan. *Data Structures and Network Algorithms*. Vol. 44. SIAM, 1983. ISBN: 0-89871-187-8.

[64] Peter Ungar. 'A Theorem on Planar Graphs'. In: *Journal of the London Mathematical Society* s1-26.4 (1951), pp. 256–262.

[65] Lode Vandevenne. *LodePNG (PNG image decoder and encoder)*. 2014. URL: http://lodev.org/lodepng/ (visited on 16/02/2015).

[66] Gayathri Venkataraman, Sartaj Sahni and Srabani Mukhopadhyaya. 'A Blocked All-Pairs Shortest-Path Algorithm'. In: *SWAT*. Springer, 2000, pp. 419–432.

[67] C. Vetter. *Parallel time-dependent Contraction Hierarchies*. 2009. URL: http://algo2.iti.kit.edu/download/vetter_sa.pdf (visited on 20/09/2014).

[68] C. Walshaw and M. Cross. 'JOSTLE: Parallel Multilevel Graph-Partitioning Software – An Overview'. In: *Mesh Partitioning Techniques and Domain Decomposition Techniques*. Ed. by F. Magoules. (Invited chapter). Civil-Comp Ltd., 2007, pp. 27–58. ISBN: 978-1-874672-29-6.

[69] Stephen Warshall. 'A Theorem on Boolean Matrices'. In: *Journal of the ACM* 9.1 (1962), pp. 11–12.

[70] Ingo Wegener. 'BOTTOM-UP-HEAPSORT, a new variant of HEAPSORT beating, on an average, QUICKSORT (if n is not very small)'. In: *Theoretical Computer Science* 118.1 (1993), pp. 81–98. ISSN: 0304-3975.

[71] J. W. J. Williams. 'Algorithm 232: Heapsort'. In: *Communications of the ACM* 7.6 (1964), pp. 347–348.

Appendix A

Additional Key Figures

A.1 Dataset

Given width and height specifications apply to the center of the graphs and will differ widely from the real values on the edges.

Table A.1: List of road networks and their key figures of North American cities and metropolitan areas.

name	nodes	edges	directed edges	width [km]	height [km]
Chicago City (contracted)	16 992	28 597	7 816	28	18
San Diego (contracted)	39 714	54 588	9 670	41	39
Edmonton (contracted)	48 309	67 545	13 363	234	142
Chicago City	55 559	67 268	21 143	30	18
Hampton Roads (contracted)	63 544	83 953	13 454	81	55
Toronto (contracted)	107 781	154 279	28 424	161	109
Montreal (contracted)	122 193	174 712	31 157	224	121
Minneapolis (contracted)	137 473	188 479	27 306	164	105
Philadelphia (contracted)	153 077	219 279	37 989	104	74
Saint Louis (contracted)	153 153	198 426	13 176	164	121
Miami (contracted)	168 967	240 989	44 090	73	180
San Diego	169 718	184 592	49 432	41	39
Phoenix (contracted)	178 570	244 510	38 389	179	142
Edmonton	181 384	200 620	44 024	234	142
Chicago (contracted)	216 083	309 617	34 825	160	107
Detroit (contracted)	216 614	304 410	37 731	198	161
Seattle (contracted)	220 614	272 279	14 626	289	214
New York (contracted)	260 348	379 083	67 825	142	84

Table A.2: List of road networks and their key figures of North American cities and metropolitan areas. (continuation)

name	nodes	edges	directed edges	width [km]	height [km]
Houston (contracted)	261 290	355 315	71 459	188	156
Washington (contracted)	280 497	364 925	60 824	172	122
San Francisco Bay (contracted)	306 758	409 051	58 810	271	215
Dallas (contracted)	316 966	433 649	75 499	187	141
Atlanta (contracted)	324 771	395 377	27 763	236	197
Hampton Roads	349 634	370 043	67 629	81	55
Los Angeles (contracted)	400 199	542 540	63 128	302	143
Montreal	534 526	587 045	97 674	224	121
Miami	570 942	642 964	136 257	73	180
Toronto	604 356	650 854	110 954	161	109
Phoenix	755 410	821 350	175 872	179	142
Minneapolis	795 087	846 093	138 772	164	105
Saint Louis	799 911	845 184	64 781	164	121
Chicago	826 934	920 466	131 253	160	107
Philadelphia	866 982	933 184	149 227	104	74
Detroit	984 444	1 072 240	142 771	198	161
Houston	1 023 386	1 117 411	214 202	188	156
New York	1 043 898	1 162 633	264 463	142	84
Seattle	1 240 937	1 292 601	89 038	289	214
Dallas	1 437 381	1 554 064	270 113	187	141
San Francisco Bay	1 471 627	1 573 920	269 840	271	215
Los Angeles	1 734 076	1 876 416	282 178	302	143
Washington	1 847 377	1 931 803	323 450	172	122
Atlanta	2 229 226	2 299 832	162 718	236	197

Table A.3: List of road networks and their key figures of European cities and metropolitan areas.

name	nodes	edges	directed edges	width [km]	height [km]
Graz (contracted)	7 241	9 115	1 200	19	17
Vienna (contracted)	29 629	42 083	13 443	64	38
London City (contracted)	40 671	53 626	10 073	30	17
Graz	48 437	50 311	6 876	19	17
Warsaw (contracted)	55 990	74 863	11 494	150	87
Brussels (contracted)	60 545	82 893	19 334	87	46
Prague (contracted)	62 109	82 633	10 682	130	72
Rome (contracted)	69 173	94 712	36 019	82	62
Moscow (contracted)	69 984	94 488	21 234	181	120
Budapest (contracted)	75 024	106 497	13 581	160	93
Stockholm (contracted)	84 756	104 050	10 941	248	170
Milan (contracted)	92 899	130 628	46 357	68	50
Berlin (contracted)	94 863	130 019	18 298	271	128
Lisbon (contracted)	97 529	133 473	34 337	116	88
London City	131 960	144 908	34 762	30	17
Vienna	138 124	150 578	48 791	64	38
Madrid (contracted)	153 319	223 650	81 519	138	137
Paris (contracted)	188 177	260 099	79 229	118	72
Istanbul (contracted)	197 116	292 700	46 084	152	74
Warsaw	266 151	285 023	54 944	150	87
Prague	295 136	315 658	45 769	130	72
Budapest	295 667	327 139	52 489	160	93
Rome	317 020	342 559	115 216	82	62
Brussels	326 325	348 672	71 788	87	46
London (contracted)	384 788	464 106	51 706	224	116
Milan	394 926	432 655	164 262	68	50
Moscow	423 708	448 212	111 173	181	120
Lisbon	453 438	489 381	133 431	116	88
Berlin	492 270	527 426	82 231	271	128
Istanbul	557 248	652 832	127 712	152	74
Madrid	606 894	677 224	273 786	138	137
Stockholm	634 723	654 017	54 316	248	170
Paris	798 642	870 556	271 189	118	72
London	1 596 644	1 675 960	224 107	224	116

Table A.4: List of road networks and their key figures of Other World cities and metropolitan areas.

name	nodes	edges	directed edges	width [km]	height [km]
Singapore (contracted)	50 719	71 169	18 736	136	66
Mumbai (contracted)	54 936	73 089	8 038	86	115
Cape Town (contracted)	77 091	106 231	10 068	77	99
Guangzhou (contracted)	101 723	148 294	62 195	311	161
Seoul (contracted)	102 211	155 347	16 435	144	107
Hong Kong (contracted)	133 626	195 585	87 391	261	181
Rio De Janeiro (contracted)	148 955	210 470	27 483	124	77
Singapore	165 955	186 405	67 327	137	66
Shanghai (contracted)	189 116	282 440	99 689	392	333
Melbourne (contracted)	191 575	258 694	66 661	172	132
Bangkok (contracted)	198 517	246 975	33 260	197	262
Buenos Aires (contracted)	211 348	361 303	89 834	159	103
Cape Town	258 944	288 084	39 798	77	99
Sao Paulo (contracted)	285 776	417 815	87 486	166	133
Mumbai	291 795	309 948	27 935	86	115
Buenos Aires	339 804	489 759	143 099	159	103
Seoul	398 583	451 719	74 667	145	107
Rio De Janeiro	468 003	529 518	83 362	124	77
Guangzhou	698 933	745 495	327 577	311	161
Bangkok	741 446	789 904	108 526	197	262
Melbourne	748 993	816 111	212 937	172	132
Hong Kong	787 956	849 907	406 546	261	182
Sao Paulo	1 016 997	1 149 036	248 666	166	133
Shanghai	1 068 688	1 162 008	434 116	392	333
Tokyo (contracted)	1 325 372	1 956 313	118 775	233	188
Tokyo	5 301 814	5 932 753	323 013	233	188

Table A.5: List of road networks and their key figures of North American states and regions.

name	nodes	edges	directed edges	width [km]	height [km]
California (contracted)	1 271 872	1 683 116	173 475	1 139	1 074
Texas (contracted)	1 904 475	2 552 576	252 825	1 472	1 205
US Northeast (contracted)	2 343 178	3 096 160	253 950	1 530	944
US West (contracted)	3 709 209	4 834 491	370 953	2 552	1 998
US Midwest (contracted)	4 683 720	6 355 110	369 762	2 668	1 475
US South (contracted)	8 546 116	10 934 235	890 168	3 482	1 793
California	10 188 049	10 599 292	836 070	1 139	1 074
Texas	11 116 039	11 764 140	908 488	1 472	1 205

Table A.6: List of road networks and their key figures of European countries and regions.

name	nodes	edges	directed edges	width [km]	height [km]
Cyprus (contracted)	66 979	86 792	5 625	257	126
Cyprus	346 085	365 898	23 065	257	126
Austria (contracted)	408 173	515 234	38 385	848	290
Netherlands (contracted)	660 352	887 830	132 398	439	303
Poland (contracted)	751 207	983 804	126 758	1 119	647
Spain (contracted)	1 855 021	2 626 188	615 158	1 400	865
Italy (contracted)	1 898 260	2 535 517	517 224	1 317	1 016
Great Britain (contracted)	2 266 835	2 704 174	226 199	947	969
Netherlands	2 788 354	3 015 824	502 326	439	303
Germany (contracted)	2 788 477	3 623 921	333 764	1 023	845
France (contracted)	3 668 602	4 801 377	561 744	1 445	972
Austria	4 037 973	4 145 033	204 390	848	290
Poland	5 785 865	6 018 451	619 386	1 119	647
Great Britain	10 968 833	11 406 167	987 652	947	969
Spain	12 256 439	13 027 583	2 108 392	1 400	865
Italy	14 911 334	15 548 577	1 722 523	1 317	1 016
Germany	16 533 327	17 368 755	1 624 559	1 023	845

Table A.7: List of road networks and their key figures of Asian countries and regions.

name	nodes	edges	directed edges	width [km]	height [km]
Japan (contracted)	4 834 762	7 112 501	292 433	1 406	1 173

Table A.8: List of road networks and their key figures of continental size.

name	nodes	edges	directed edges	width [km]	height [km]
Africa (contracted)	5 454 628	7 702 009	367 386	6 040	8 025
South America (contracted)	7 198 662	10 913 434	1 641 918	4 302	7 338
Asia (contracted)	10 681 715	14 706 424	1 881 412	11 449	7 915
Europe (contracted)	19 841 133	26 372 551	3 090 792	5 863	3 943
North America (contracted)	22 437 475	29 743 795	2 451 466	6 715	4 658
USA	23 947 347	28 854 312	0	6 269	2 682

Appendix B

Additional Results

B.1 ALT, Bidirectional ALT

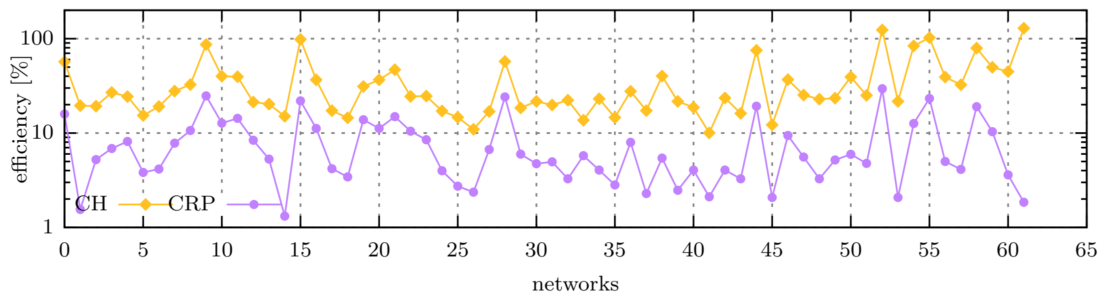
Table B.1: Comparison of the node efficiency between the **modified Partition-Corners algorithm** and **randomly placed landmarks** with Unidirectional ALT and 36 landmarks. The numbers denote the efficiency increase in percent. Every column shows a different combination of the full and contracted networks with geometric distances and travel time metrics.

category	all [%]	full [%]	contr. [%]	all distance [%]	all time [%]	full distance [%]	full time [%]	contr. distance [%]	contr. time [%]
C-NA	20.5	21.8	19.5	24.2	15.9	22.5	21.0	25.6	11.7
C-EU	21.7	22.3	21.2	25.1	17.6	23.6	20.8	26.5	14.5
C-OW	10.6	15.5	5.6	17.0	3.4	22.5	8.1	11.7	-1.9
R-NA	20.8	19.2	21.5	22.8	17.6	19.9	18.3	23.9	17.3
R-EU	20.2	24.6	15.3	23.2	15.3	28.7	18.7	17.7	10.8
R-OW	22.3	-	22.3	19.6	28.7	-	-	19.6	28.7
Continental	15.2	46.8	9.8	15.2	15.1	58.1	32.9	8.3	12.0
all	19.1	20.9	17.4	23.1	14.1	23.5	18.0	22.7	10.4

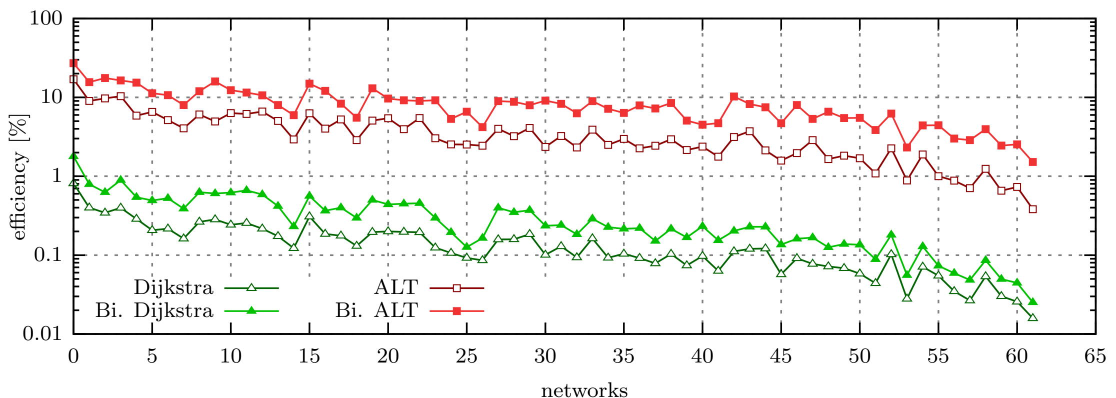
Table B.2: Comparison of the node efficiency between the **modified** and the **original Partition-Corners algorithm** with Unidirectional ALT and 36 landmarks. The numbers denote the efficiency increase in percent. Every column shows a different combination of the full and contracted networks with geometric distances and travel time metrics.

category	all [%]	full [%]	contr. [%]	all distance [%]	all time [%]	full distance [%]	full time [%]	contr. distance [%]	contr. time [%]
C-NA	12.3	11.2	13.1	11.5	13.3	10.2	12.5	12.4	14.1
C-EU	10.3	11.6	9.0	10.9	9.5	11.5	11.9	10.5	7.1
C-OW	2.9	3.7	1.9	3.1	2.6	3.9	3.5	2.2	1.5
R-NA	4.0	0.2	5.7	3.7	4.4	-2.7	4.3	6.4	4.5
R-EU	7.4	9.1	5.3	6.9	8.1	8.9	9.5	4.9	6.1
R-OW	8.7	-	8.7	7.9	10.4	-	-	7.9	10.4
Continental	3.2	10.1	1.8	4.5	1.4	9.7	10.7	3.4	-0.3
all	9.1	9.6	8.7	9.2	9.1	9.2	10.0	9.1	8.1

B.2 Edge Efficiency with Travel Times

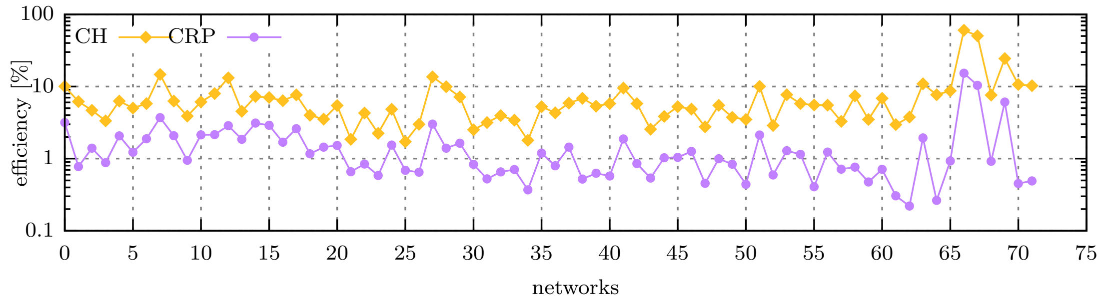


(a) Average edge efficiency on full networks with travel times.

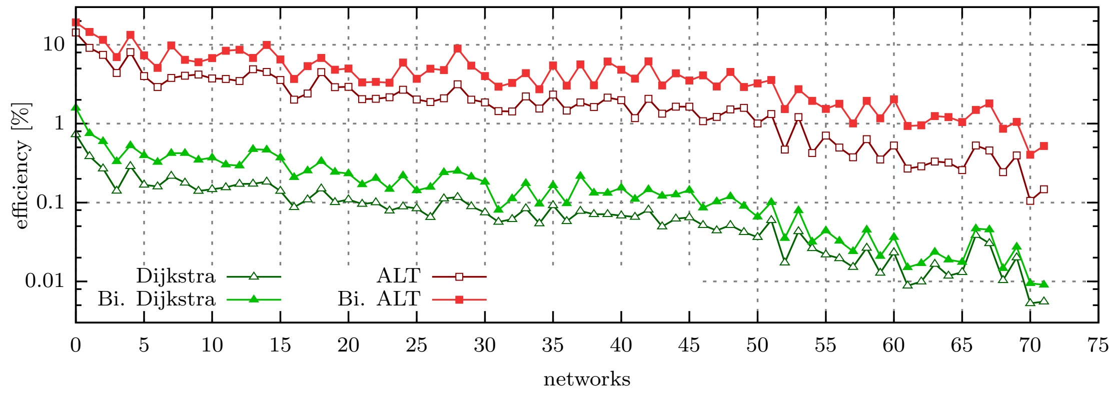


(b) Average edge efficiency on full networks with travel times.

Figure B.1: Average edge efficiency for all **full networks** with **travel times**. Sorted by the networks' node count in ascending order.



(a) Average edge efficiency on contracted networks with travel times.



(b) Average edge efficiency on contracted networks with travel times.

Figure B.2: Average edge efficiency for all **contracted networks** with **travel times**. Sorted by the networks' node count in ascending order.

Appendix C

Framework

C.1 Compiling

The whole framework is written in C++ (version 11) and uses no external libraries except for LodePNG [65], which is included. It has been tested with g++ 4.8.4 on Linux Mint 17 (qiana 3.13.0-24-generic, 64 bit). It can be compiled with the following commands:

make debug MT=1 Creates a build with debug information and assertions enabled.

make profile MT=1 Creates a build with debug information but assertions disabled for profiling.

make release MT=1 Creates a release build without debug information and assertions disabled.

The `MT=1` parameter enables multithreading support. To disable multithreading, set it to 0. Before using the framework, make sure that a subfolder `data` containing the graph files and a subfolder `output` exists.

C.2 Graph Files

The data for each graph is stored in two files, one for the node data and another for the edge data. The nodes file must contain the phrase `_nodes` and the edges file `_edges` in front of the file extension (e.g. `austria_nodes.bin` and `austria_edges.bin`). Four different graph file formats are supported:

- `CUSTOM_CSV_SIMPLE`
  - text format
  - uses file extension `.csv`
  - nodes file each line: node id, x, y, z

- edges file each line: start node id, target node id, cost

- `CUSTOM_CSV` and `CUSTOM_BINARY`
  - text format and binary format, respectively
  - uses file extension `.csv` and `.bin`, respectivley
  - nodes file each line: node id, x, y, z
  - edges file each line: start node id, target node id, distance cost (meter), time cost (seconds)

- `TIGER`
  - text format
  - uses file extension `.csv`
  - Tiger file format

The identifiers do not have to be consecutive and can start anywhere but have to be smaller than $2^{64} - 1$.

C.3 User Manual

The graph files have to reside in a subfolder called `data`. All output is written to the subfolder `output`. The supplied graphs with the file extension `.bin` have the format `CUSTOM_BINARY` except *usa*, which has the type `TIGER`. The framework (program) has the following command line parameters:

- The following applies to all commands:
  - valid strings for `<cost type>` are: `DISTANCE, TIME`; cost type only has an effect for `CUSTOM_CSV` and `CUSTOM_BINARY`
  - valid strings for `<data file format>` are: `CUSTOM_CSV_SIMPLE, CUSTOM_CSV, CUSTOM_BINARY, TIGER`

- Convert an OSM file to custom data files useable by this program:

  ```
  ./main convert <path + OSM filename> <path + output filename prefix> <contract paths 0|1>
  ```

  example usage: `./main convert data/filtered_austria.osm data/austria 0`

- Convert an OSM file to custom data files useable by this program:

  ```
  ./main convertcustom [<path + OSM filename>] <path + intermediate filename prefix> <path + output filename prefix> <contract paths 0|1> [<min x> <max x> <min y> <max y>]
  ```

- Omit the OSM filename to skip the conversion step in case the intermediate file has already been created.
- If min/max coordinates are specified, the network will be cropped accordingly.
- The min/max coordinates have to be in decimal degrees with x being East-West and y North-South!
- Use negative decimal degrees for West and South coordinates!

example usage: `./main convertcustom data/filtered_austria.osm data/austria data/austria 0 15.3668 15.5367 46.9849 47.1393`

- Run all algorithms on the same network and compare results to find mismatches:

```
./main debugunittest <data file prefix> <data file format> <cost type>
```

example usage: `./main debugunittest austria CUSTOM_BINARY DISTANCE`

- Print the given network:

```
./main printnetwork <data file prefix> <data file format> <scale> <max
dimension> <max type> <fat mode radius> <save statistics 0|1>
```

  - <scale>: upscales the coordinates in the graph; if longitude and latitude is used in the graph file, large values must be used (e.g. 10 000)
  - <max dimension>: max dimension is the maximum width/height in pixel the output image can have; recommended are the following values: 7 000 (= 1 200 dpi), 3 500 (= 600 dpi), 1 750 (= 300 dpi), 875 (= 150 dpi)
  - <max type>: only print roads up to the given type (0-3: primary road - residential road); use -1 for all roads
  - <fat mode radius>: use a value other than 0 to increase the size of every pixel printed

example usage: `./main printnetwork austria CUSTOM_BINARY 10000 7000 -1 0 0`

- Print the partitions created by CRP:

```
./main printcrppartitions <data file prefix> <data file format> <number of
levels> <with waterways 0|1> <with railways 0|1> <ways as separate file
0|1> <save statistic 0|1>
```

example usage: `./main printcrppartitions austria CUSTOM_BINARY 6 1 1 1 1`

- Print the partitions created by CRP:

```
./main printcrppartitions random <number of nodes> <with center traffic
0|1> <number of levels> <save statistic 0|1>
```

example usage: `./main printcrppartitions random 3600 1 4 1`

- Print the partition created by Natural Cuts:

  ```
  ./main printncpartition <data file prefix> <data file format> <max cell
  size>
  ```

  example usage: `./main printncpartitions austria CUSTOM_BINARY 65536`

- Print the network created by Contraction Hierarchies:

  ```
  ./main printchnetwork <data file prefix> <data file format> <cost type>
  <number of top level nodes>
  ```

  - valid values for `<number of top level nodes>`: use 0 for all nodes or a number between 1 and the maximum number of nodes in graph

  example usage: `./main printchnetwork austria CUSTOM_BINARY DISTANCE 10`

- Print the node degree and shortcut distribution of the network created by Contraction Hierarchies:

  ```
  ./main printchdistribution <data file prefix> <data file format> <cost
  type> <number of parts for shortcut distribution>
  ```

  example usage: `./main printchdistribution austria CUSTOM_BINARY DISTANCE 4`

- Find the shortest path with the given algorithms and network between the specified nodes:

  ```
  ./main search <data file prefix> <data file format> <cost type> <save
  graph 0|1> <save search statistic 0|1> <use Dijkstra 0|1> <use
  bidirectional Dijkstra 0|1> <use A-Star 0|1> <use bidirectional A-Star
  0|1> <use ALT 0|1> <use BidirectionalALT 0|1> <use CH 0|1> <use CRP 0|1>
  <start node number> <target node number>
  ```

  example usage: `./main search austria CUSTOM_BINARY DISTANCE 1 1 1 1 1 1 1 1 1 1 200 5000`

- Find the shortest path with the given algorithms and network between random nodes:

  ```
  ./main randomsearch <data file prefix> <data file format> <cost type>
  <save graph 0|1> <save search statistic 0|1> <use Dijkstra 0|1> <use
  bidirectional Dijkstra 0|1> <use A-Star 0|1> <use bidirectional A-Star
  0|1> <use ALT 0|1> <use BidirectionalALT 0|1> <use CH 0|1> <use CRP 0|1>
  <cost only 0 | reconstruct path 1> <search count>
  ```

  example usage: `./main randomsearch austria CUSTOM_BINARY DISTANCE 0 1 1 1 1 1 1 1 1 1 100000`

- Find the shortest path with the given algorithms between the specified nodes on a randomly generated grid graph:

  `./main search_randomgraph <number of nodes> <with center traffic 0|1> <save graph 0|1> <save search statistic 0|1> <use Dijkstra 0|1> <use bidirectional Dijkstra 0|1> <use A-Star 0|1> <use bidirectional A-Star 0|1> <use ALT 0|1> <use BidirectionalALT 0|1> <use CH 0|1> <use CRP 0|1> <start node number> <target node number>`

  example usage: `./main search_randomgraph 3600 1 1 1 1 1 1 1 1 1 200 1000`

- Get the number of the node nearest to the specified decimal coordinates in the given network:

  `./main nearestnode <data file prefix> <data file format> <x in decimal degrees (East-West!)> <y in decimal degrees (North-South!)>`

  - Use negative decimal degrees for West and South coordinates!

  example usage: `./main nearestnode austria CUSTOM_BINARY 15.459519 47.058519`

- Get the number of the node nearest to the specified sexagesimal coordinates in the given network:

  `./main nearestnode <data file prefix> <data file format> <x degree> <x minutes> <x seconds> <y degree> <y minutes> <y seconds>`

  - The x coordinates are East-West and the y coordinates are North-South!
  - Use negative degree, minutes and seconds for West and South coordinates!

  example usage: `./main nearestnode austria CUSTOM_BINARY 15 27 34.27 47 3 30.67`