COMPARATIVE STUDY OF SPEED-UP ROUTING ALGORITHMS IN ROAD NETWORKS

Raheleh Zarei Chamgordani

A thesis
in
The Department
of
Software Engineering and Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montréal, Québec, Canada

May 2021

Concordia University
School of Graduate Studies

This is to certify that the thesis prepared

By: **Raheleh Zarei Chamgordani**

Entitled: **Comparative Study of Speed-Up Routing Algorithms in Road Networks**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

Dr. D. Pankratov — Chair

Dr. J. Opatrny — Examiner

Dr. D. Pankratov — Examiner

_____ — Examiner

Dr. L. Narayanan — Supervisor

Approved Dr. H. Harutyunyan
Chair of Department or Graduate Program Director

May 13th 20 21

Mourad Debbabi, Dean
Gina Cody School of Engineering and Computer Science

Abstract

Comparative Study of Speed-Up Routing Algorithms in Road Networks

Raheleh Zarei Chamgordani

We study the problem of finding the shortest distance and the shortest path from one node to another in graphs modeling large road networks. Classical algorithms like *Dijkstra* and *Astar* do not have good performance in such networks [10, 22, 54]. In recent years, two new approaches called *Contraction Hierarchy* [26] and *Hub Labeling* [10] which use preprocessing to generate auxiliary data to improve the query time performance were proposed, and many variants have followed. These algorithms are very efficient on large networks when a large number of queries is expected. In the literature, these algorithms are called *speed-up* algorithms. More recently, *dynamic* routing algorithms have been proposed, such as *Customizable Contraction Hierarchy* [21] and *Dynamic Hierarchical Hub Labeling* [18]. These are designed to respond efficiently to edge weight changes resulting from changes in traffic.

In this thesis, we present an experimental study of the performance of the above static and dynamic routing algorithms on two different road networks, in terms of travel time and query processing time. Our results show that Customizable Contraction Hierarchy is the best for shortest path query in both the static and dynamic settings, while Hub Labeling is the most efficient in answering shortest distance queries in the static setting. We also show that Dynamic Hub Labeling's edge weight update operations are inefficient in practice.

Acknowledgments

I would like to offer my special thanks to my supervisor Dr. Lata Naraynan for her continuous support and patience during my research work. I took great advantage of her plentiful experience and received brilliant advice throughout my thesis. In addition, I would like to thank my thesis committee.

Finally, I would like to express my heartfelt gratitude to my parents for their continuous motivation and support. A very special thanks to my amazing sisters my supportive brother for their support and encouragement. I would also like to thank my friends for their encouragement and helpful advice.

Contents

List of Figures viii

List of Tables xi

List of Algorithms xiii

1 Introduction 1

1.1 Motivation . . . 1
1.2 Problem Definition . . . 3
1.3 Thesis Contributions . . . 4
1.4 Thesis Organization . . . 5

2 Static Routing Algorithms 6

2.1 Basic Techniques . . . 6
2.2 Goal-Directed Techniques . . . 7
2.3 Other Techniques with Additional Preprocessing Phase . . . 8
2.4 Hierarchical Techniques . . . 9
2.4.1 Contraction Hierarchy(*CH*) . . . 11
2.4.1.1 Contraction . . . 11
2.4.1.2 Node Ordering . . . 13
2.4.1.3 An Example of Contraction Hierarchy . . . 16
2.4.1.4 Query . . . 19
2.5 Routing algorithms based on Labeling Techniques . . . 22
2.5.1 Hierarchical Hub Labeling Using Pruned Landmark Labeling . 25
2.5.1.1 *HHL* for Directed Weighted Graphs . . . 25
2.5.1.2 Example of *HHL* for a Directed Weighted Graph . . 29

2.5.1.3 Different Node Ordering 33
2.5.1.4 Query 33
2.6 Empirical Studies of Static Routing Algorithms 34

3 Dynamic Routing Algorithms 36
3.1 Customizable Contraction Hierarchy (*CCH*) 36
3.1.1 Building a Contraction Hierarchy 38
3.1.1.1 Different Graph Partitioning Algorithms 39
3.1.2 Customization 43
3.1.2.1 Perfect Customization and the *CCH+P* Algorithm . 44
3.1.3 Query 46
3.2 Dynamic Hierarchical Hub Labeling 48
3.2.1 Incremental Update 49
3.2.2 Decremental Update 53
3.2.2.1 Detecting Affected Nodes 54
3.2.2.2 Removing Affected Hubs 58
3.2.2.3 Computing New Hubs 60
3.3 Empirical Studies of Comparison of Dynamic Routing Algorithm . . . 64

4 Experimental Analysis 65
4.1 Experimental Settings 65
4.2 Analysis of Static Routing Algorithms 70
4.2.1 Contraction Hierarchy 70
4.2.1.1 Two Different Node Ordering Heuristics 71
4.2.1.2 CH Performance Analysis 72
4.2.2 Customizable Contraction Hierarchies 73
4.2.2.1 *CCHB* Performance Analysis 74
4.2.2.2 *CCH* with Perfect Witness Search Performance Analysis 75
4.2.2.3 *CCHB* or *CCH+P* ? 77
4.2.3 Hierarchical Hub Labeling 77
4.2.3.1 *HHL* Performance Analysis 79
4.2.4 Comparison with Other Routing Algorithms 80
4.3 Dynamic Routing Algorithms 83
4.3.1 Effect of Update Interval on Customization Time 87

4.3.2 Effect of Update Interval on Travel Time 89
4.3.3 Improving the Performance of Dynamic *HHL* 91

5 Conclusion and Future Work **95**

List of Figures

1 Shortcut added between u, w. Note that edge (u, v) and (v, w) do not belong to the graph any more. 12

2 No shortcut added, path $P = \langle u, x, y, z, w \rangle$ already exist with $d(\langle u, v, w \rangle) >= d(P)$ 12

3 1-Hop search starting from node v leads to find the edge (u, w) as witness path shown in red. 13

4 No shortcut added for the contraction of nodes in the path graph. Numbers in the nodes show the priority of each node for contraction and the number below each nodes shows the edge difference calculated for each node. 15

5 Before the contraction of node u, the edge difference of contracting node v is $3 - 3 = 0$. 15

6 After the contraction of node u, the edge difference of contracting node v is $1 - 2 = -1$. 15

7 Two shortcuts added in this node ordering. As we see the search space is maximum two edges. 16

8 Directed graph $G = (V, E)$. 17

9 Overlay graph G' (Left) and graph G (Right) after the contraction of node a. 18

10 Overlay graph G' (Left) and graph G (Right) after the contraction of node b. 18

11 Overlay graph G' (Left) and graph G (Right) after the contraction of node d. 19

12 Final overlay graph. Overlay graph together with the order of nodes lead to a contraction hierarchy. On the right, the same graph is drawn but nodes of lower priority are placed below nodes of higher priority. 20

13 Query in contraction hierarchy of graph G. 21
14 Efficiency of contraction hierarchy compared to *Dijkstra*, bidirectional *Dijkstra* and *Astar* 23
15 Directed weighted graph $G = (V, E)$ with the order of nodes for label contraction 29
16 Priority queue status after visiting node a of our example graph in *HHL* construction. 30
17 Label of nodes of graph G after the forward iteration (on the left) and after the backward iteration (on the right) of node a. 30
18 Label of nodes of graph G after the forward iteration of node d. . . . 31
19 Final label of nodes of example graph G. 32
20 Chordal graph of graph G and its upward directed graph with respect to a ND-order O. 39
21 Lower Triangle, Intermediate Triangle and Upper Triangle of three nodes x, y and z. Rank of each node is shown next to it. 44
22 Respected metric of graph G. 45
23 Customized metric of graph G. 45
24 Perfect metric of graph G. 47
25 Elimination tree construction of graph G with respect the order O. . 48
26 Edge weight decrease of edge (c, a) of graph G. 53
27 Removed outdated entries from labeling L of graph G after weight decrease of the edge (c, a). 54
28 Edge weight increase of edge (b, c) of graph G. 57
29 Removed outdated entries from labeling L of graph G after the weight increase of the edge (b, c) shown in red. 60
30 New hubs added to the labeling L of graph G. 61
31 Montreal downtown road network 69
32 Eichstaett road network 69
33 Improvement of $Avg\ TT$ with respect to different update interval for 10000 queries using *CCH+P* algorithm for Montreal downtown network (Left) and Eichstaett (Right). 84

34 *Avg CT* as a percentage of *Est.PP* with respect to different update interval for different number of queries using *CCH+P* algorithm for Montreal downtown network (Left) and Eichstaett (Right). 84
35 *Avg CT* as a percentage of *Est.PP* with respect to different update interval for different number of queries (1000 queries for Montreal and 1000, 5000 and 10000 queries) using dynamic *HHL* algorithm for Montreal downtown network (Left) and Eichstaett (Right). 86
36 *Avg CT* (ms) for different weight update intervals for 1000 queries for Montreal Downtown road network. Numbers are the log-scaled. . . . 88
37 *Avg CT* (ms) for different weight update intervals for different number of queries for Eichstaett road network. Numbers are the log-scaled. . 88
38 *Avg TT* (s) for different weight update intervals for 1000 queries for Montreal Downtown road network. 89
39 *Avg TT* (s) for different weight update intervals and different number of queries for Eichstaett road network. 89
40 Sum of all the edges whose weight has changed during the simulation for 1000 queries and 200 as weight update interval for Montreal downtown Network. 92
41 Sum of all the edges whose weight has changed during the simulation for 1000 queries and 200 as weight update interval for Eichstaett Network. 92
42 X–axis shows the percentage of edges whose weight has increased in the second interval of weight update. The Y–axis shows the fraction of affected nodes for the Montreal downtown network. (update interval is 200). 94
43 X–axis shows the percentage of edges whose weight has increased in the second interval of weight update. The Y–axis shows the fraction of affected nodes for Eichstaett network. (Update interval is 200). . . 94

List of Tables

1 Initializing the priority of each node. 17
2 Priority of the remaining nodes after the contraction of node a. . . . 18
3 Priority of the remaining nodes after the contraction of node b. 19
4 Algorithms used in our experiments. 68
5 Benchmark road networks 68
6 Dijkstra and Astar for Montreal downtown road network. 70
7 Dijkstra and Astar for Eichstaett road network. 71
8 Contraction Hierarchies performance for 1000 queries for two different node order for Montreal downtown road network 73
9 Contraction Hierarchies performance for two different node order for Eichstaett road network. 73
10 *CCHB* performance for different node ordering, averaged over 1000 queries for Montreal downtown road network. 75
11 *CCHB* performance for different node ordering, averaged over 1000 queries for Eichstaett road network. 75
12 *CCH+P* performance with perfect witness search for different graph partitioning for Montreal downtown road network. 76
13 *CCH+P* performance with perfect witness search for different graph partitioning for Eichstaett road network. 76
14 Comparison of *CCHB* and *CCH+P* for Montreal downtown road network. 77
15 Comparison of *CCHB* and *CCH+P* for Eichstaett road network. . . . 78
16 Hub Labeling performance with different node ordering for Montreal downtown road network. 79
17 Hub Labeling performance with different node ordering for Eichstaett road network. 80

18 Comparing static routing algorithms for 1000 queries for Montreal downtown road network 82
19 Comparing static routing algorithms for 1000 queries for Eichstaett road network 82
20 Travel time changes for different interval for 10000 queries for Montreal downtown network (Left) and Eischtaett (Right) for *CCH+P* algorithm. 83
21 Comparison of *Avg CT* and *Est.PP* after each edge update interval for different number of queries for Montreal downtown network. 85
22 Comparison of *Avg CT* and *Est.PP* after each edge update interval for different number of queries for Eichstaett network. 85
23 *CCH+P* and *HHL* with different edge weight update interval and different number of queries for Montreal downtown road network. 90
24 *CCH+P* and *HHL* with different edge weight update interval and different number of queries for Eichstaett road network. 90
25 *Avg CT* (ms) and *Avg TT* (s) for $c = 0.1$ and $c = 0.2$ for 1000 queries and 200 as weight update interval for Montreal downtown road network. 91
26 *Avg CT* (ms) and *Avg TT* (s) for $c = 0.1$ and $c = 0.2$ for 1000 queries and 200 as weight update interval for Eichstaett network. 91
27 Average time (ms) of running Incremental and Decremental update for 1000 queries and 200 as weight update interval for Montreal downtown and Eichstaett network. 93

List of Algorithms

1 CHPreprocessing($G = (V, E)$, Order O) 14
2 PrunedBFS(G, v_i, L_{i-1}) 26
3 Compute Labels_BFS(G, V, Order O) 26
4 PrunedDijkstra(G, v_i, L_{i-1}, dir) 27
5 Compute Labels_Dijkstra(G, V, Order O) 28
6 SPQuery(u, v, L) 34
7 Inc_PLL($a, b, l(a, b), L$) 51
8 ResumePrunedBFS(G, v_k, u, δ, L) 51
9 Inc_PLL($a, b, l(a, b), L$) 52
10 ResumePrunedDij($G, v_k, u, \delta, p, L, dir$) 52
11 AFF_a In undirected unweighted graph 57
12 AFF_a In directed weighted graph 58
13 RemoveOutdatedHubs(AFF_a, AFF_b) In undirected unweighted graph . 59
14 RemoveOutdatedHubs(AFF_a, AFF_b) In directed weighted graph . . . 59
15 ComputeNewLabels() In undiredcted unweighted graph 62
16 ComputeNewLabels() In diredcted weighted graph 63

Chapter 1

Introduction

1.1 Motivation

Finding a shortest and quickest path from a given source to a given target in road networks is an important component of real-world applications such as navigation systems (e.g., Google Maps and Bing Maps), route planning apps, and traffic simulation software. According to the data reported in *Statistics Canada* [64], the number of Canadian commuters, defined as those who spend at least 60 minutes travelling to work, has increased by 30 percent or by 3.7 million to 15.9 million between 1996 and 2016. In Canada, long commuting times are known as a big-city phenomenon. In 2016, about 60 percent of workers in Toronto, Montréal or Vancouver had a long commute to work by car. Among those, 37 percent worked in Toronto, 15 worked in Montréal, and another 8 percent worked in Vancouver [64]. Although the proportion of Canadian commuters using public transit has increased slightly over the past 20 years, the vast majority of commuters in Canada prefer travelling by car as their commuting mode [64].

During 2016 and 2017, the American Driving Survey found that on average, drivers spent 51 minutes driving approximately 31.5 miles each day, making an average of 2.2 driving trips. Countrywide in the USA, drivers made 183 billion trips, driving 2.6 trillion miles, in 2016 and 2017. In 2016-2017, all driving metrics increased when

comparing statistics with the previous period measured, 2014-2015 [40].

It is therefore not surprising that maps and navigation apps are indispensable tools for smartphone users. According to the [34], 66.8 percent of smartphone users use these apps monthly, with a growth of 5.8 percent from 2017 to 2018. They also anticipated that more than 155 million people would use maps and navigation apps by the end of 2019 and expected that growth will remain steady in the coming years, and this extensive usage will reach 67.6 percent by 2021. The most popular navigation app, Google Maps, was released in 2005 as a standard digital mapping service and later that year launched driving and public transport directions. It is used by over two-thirds (67 percent) of smartphone owners. It is used for driving routes, cycle routes, as well as walking and public transport directions [45]. Waze, which launched in 2009 by a startup based in suburban Tel Aviv with the motivation to save drivers five minutes on every trip [1] comes in second at 12 percent as an app for driving routes only [45].

In addition to this overwhelming use of navigation apps, there are other reasons that justify the necessity of finding the shortest and fastest routes in today's life such as the expansion of road networks. Rapid road network expansion in a short period may bring urban planning problems related to routing. This will cause bigger maps and the resulting network will be a graph with more junctions and streets. As road networks expand, the need for more efficient methods to respond to direction queries of users will increase.

Another motivation for studying efficient routing is the extensive global phenomenon which is traffic congestion. Researchers have defined congestion from different points of view. The most common definition of congestion in the state of traffic flow is when the travel demand exceeds road capacity [44]. Traffic congestion might be a result of high population density, growth in the number of motor vehicles, and rise of delivery services [48]. Congestion that occurs by a high density of vehicles results in excess travel time [59]. Based on studies of TomTom navigation systems, nine out of the 12 Canadian cities experienced more traffic congestion in 2019 than they did in 2018. Also according to the INRIX Roadway Analytics [2] in 2017, over the next 10 years, the most congested 25 cities of the U.S. are estimated to cost the drivers 480 billion due to lost time, wasted fuel, and carbon emitted during congestion [47]. It is

clear that road networks are not static and the travel time will frequently change due to traffic jams and traffic congestion. This evokes the importance and necessity for real-time navigation apps that are capable of responding to the changes in a timely fashion.

1.2 Problem Definition

In this thesis, we study the classical shortest path problem in large road networks. A road network is modelled by a graph, where the nodes represent intersections between roads, and the edges represent roads. The weight of an edge can either be the length/distance (in meters) of the corresponding road, or the estimated travel time on the road. Given a source node and destination node in the road network, the goal is to find a shortest path from the source to the destination, and to return the answer quickly, that is, minimize the so-called *query time*.

The above problem can be studied in both the static setting, where edge weights do not change, such as when the edge weight is the distance metric, or in the more realistic dynamic setting, such as when the edge weight is the travel time on the corresponding road. Classical algorithms like *Dijkstra* and *Astar* work very well in small networks, but don't have good query time performance on large networks. In recent years, a number of researchers have proposed new and more efficient algorithms for the shortest path problem in road networks.

In this thesis, we study the query performance of some of these new algorithms in finding the shortest path and compare them with each other and with classical approaches. We also study the query time performance of the dynamic variants of these algorithms. Besides the efficiency of the query, we study the preprocessing time, and the quality of the route in different algorithms.

1.3 Thesis Contributions

We provide a comprehensive empirical study of recent shortest path algorithms in road networks in both static and dynamic settings. We run our experiments using the well-known simulator SUMO [3] on the Montreal downtown road network as well as the Eichstaett road network. In the static setting, we evaluate and compare the performance of Contraction Hierarchy [26] (*CCH*), Customizable Contraction Hierarchy [22] (*CCH*), and Hierarchical Hub Labeling [10] (*HHL*). In the dynamic setting, we evaluate and compare the performance of Customizable Contraction Hierarchy and Dynamic Hub Labeling [18].

- We compare the effect of different node orderings on the Contraction Hierarchy routing algorithm in both road networks with respect to the query performance. Our results show that performance of contraction hierarchy changes significantly considering our different node ordering heuristic.
- We compare the effect of different node orderings extracted from different graph partitioning algorithms on two different versions of the Customizable Contraction Hierarchy routing algorithm in both road networks with respect to the query performance. Our results show that using Customizable Contraction Hierarchy, to make a contraction hierarchy is very fast and leads to better query performance.
- We compare the effect of different node orderings techniques on the Hierarchical Hub Labeling routing algorithm in both road networks with respect to the query performance. Our results show that size of the labels and query performance depends on the node ordering method.
- We compare the best versions of Contraction Hierarchy, Customizable Contraction Hierarchy, and Hub Labelling with each other and also with *Dijkstra* and *Astar* algorithms. Our results show that *HHL* is the best with respect to shortest distance query performance and *CCH* is the best with respect to shortest path query performance.
- In the dynamic setting, we implemented the fully dynamic Hub Labeling algorithm that also returns the shortest distance and shortest path of a given source

and destination.

- We ran experiments with two dynamic speed-up algorithms, *HHL* and *CCH* and compare their query performance in two road networks instances for different number of queries. Our results show that the dynamic hub labelling operations for responding to edge weight changes are too expensive in a highly dynamic network. In contrast, the customization operation in *CCH* allows for an efficient response to edge weight changes, which further ensures a better route quality.

1.4 Thesis Organization

In Chapter 2, we give a literature review about static routing algorithms and explain the main speed-up algorithms in detail. In Chapter 3, we describe dynamic routing algorithms in detail and provide a brief explanation of graph partitioning methods used in the preprocessing phase. In Chapter 4, we present our detailed experiments on comparing static routing algorithms and dynamic routing algorithms. The final chapter concludes our thesis and gives some ideas for future work.

Chapter 2

Static Routing Algorithms

In this chapter, we study the background of static routing algorithms in road networks. In the point-to-point shortest path problem, the input is a directed or undirected weighted graph $G = (V, E)$, a source $s \in V$, and a target $t \in V$, and one must compute the length of the shortest path from s to t in G which is denoted by $d(s, t)$, the distance between s and t. We call this the Shortest-Distance problem. If we are also required to provide the shortest path itself, then we call it the Shortest-Path problem. We start with basic techniques and continue with routing algorithms that were designed for large road networks. In this thesis we use some terms precisely, some of which are quite standard graph-theoretic terms. Let $G = (V, E)$ be a directed graph with weight function $l : E \longrightarrow \mathrm{R}$ that gives each edge (x, y) of G a positive weight denoted by $l(x, y)$. A path in G is a sequence of nodes $\langle v_1, ..., v_k \rangle$ such that $(v_i, v_{i+1}) \in E$. The weight of a path P is defined as the sum of weights of all edges in the path. We use n to refer to the number of nodes of the graph ($|V|$) and m to refer to the number of edges of the graph ($|E|$).

2.1 Basic Techniques

The classic and best-known algorithm to compute the shortest distance (path) in a graph is *Dijkstra*'s algorithm [23]. It uses a priority queue to scan (settle) the nodes

during the search. Nodes are prioritized based on their distance to the source node. At first, all the nodes have the distance set to infinity except the source that has the distance zero. In every iteration, the algorithm chooses the node with the smallest distance from the source from the priority queue, and relaxes [1] its outgoing edges. The algorithm continues until it meets the target node. The number of nodes that are scanned during this search is called the *search space* of the *Dijkstra* algorithm. In practice, a faster version of point-to-point search which is *bidirectional search*, was proposed in [19]. Two *Dijkstra* algorithms start simultaneously from source and target nodes in forward and backward direction respectively. A stopping criterion as explained by Goldberg et al. [29] could be when a node v is about to be scanned a second time in each direction. Another stopping criterion is $d(s,v)+l(v,w)+d(w,t) < \mu$ where $l(v,w)$ is the length of edge (v,w) and μ is the length of the best path seen so far. In road networks, *bidirectional Dijkstra* visits about half as many nodes as the standard *Dijkstra* algorithm.

2.2 Goal-Directed Techniques

Goal-directed techniques guide the search toward the target by reducing the number of nodes in the search space. It prunes the nodes that are not in the direction of the target node. They use a heuristic for this purpose. Heuristics must be admissible, that is, they should never *overestimate* the cost of reaching the target node. *Astar* is a classic goal-directed technique for finding the shortest path. It uses a heuristic that assigns a value to each node of the graph. It is denoted by the function $h : V \longrightarrow R$. This value is a lower bound on the distance from a node v to target node t. The *Astar* algorithm runs a modified version of *Dijkstra* that uses the $d(u,v) + h(v)$ as the value of nodes in the priority queue. This results in scanning the nodes that are closer to the target t earlier than other nodes during the algorithm.

[1] The operation of relaxation is standard in shortest path algorithms and is described in Chapter 24 of the CLRS textboook [4].

2.3 Other Techniques with Additional Preprocessing Phase

Many types of networks, including road networks, are considered big for *Dijkstra*'s algorithm to answer the shortest distance (path) in a reasonable time especially in the situation that the number of shortest distance queries is huge. This is the motivation to construct *speed-up* techniques for these type of networks and queries [61]. In this situation, a speed-up technique is considered as a technique that uses precomputed information to reduce the search space of Dijkstra's algorithm. Precomputed information is some auxiliary data that guides the search or prunes it. One such algorithm is *ALT (Astar, landmark, triangle inequality)* [30] algorithm. It creates a better lower bound than *Astar* during the preprocessing phase. It picks a small set $L \subseteq V$ of landmarks and stores the distances between them and all nodes in the graph. During an *s-t* query, it uses triangle inequalities involving the landmarks to compute a valid lower bound on $d(u,t)$ for any node u. More precisely, for any landmark l, both inequality, $d(u,t) \geq d(u,l) - d(t,l)$ and $d(u,t) \geq dist(l,t) - dist(l,u)$ hold.

Arc Flags [35, 41], *Geometric Containers* [57, 62] and *Precomputed Cluster Distances* [43] algorithms are other speed-up algorithms which create labels that encode additional information on the edges of the network during a preprocessing phase. *Geometric Containers* precomputes an edge label $L(a)$ for each edge $a = (u,v) \in E$, that encodes the set of nodes to which a shortest path from u begins with the edge a. Instead of storing original node Ids in $L(a)$, it approximates this set of nodes by using their geometric information (coordinates). During a query, if the target node t is not in $L(a)$, the search can be pruned at a. In *Arc Flag* during the preprocessing, instead of storing geometric information, the graph is partitioned to K cells with roughly equal size with a small number of boundary nodes, then every edge contains a vector of K bits such that bit i is set to one if the edge is on the shortest path to some nodes of cell i and zero otherwise. Boundary nodes are used to compute the arc flag of a cell i by running a backward shortest path tree from each boundary node of cell i and setting the i^{th} flag for all arcs of the tree. During the *s-t* query, the edge with no bit set to one for the cell containing the node t can be pruned. *Precomputed Cluster Distances* is similar to *Arc Flags* but in addition to the boundary nodes, it stores the

shortest path distances between all pairs of cells in labels. The query algorithm is a pruned version of *Dijkstra*'s algorithm. For any node u visited by the search, a valid lower bound on its distance to the target is $d(s,u) + d(C(u), C(t)) + d(v,t)$, where $C(u)$ is the cell containing u and v is the boundary vertex of $C(t)$ that is closest to t.

Separator-based techniques are another category of speed-up algorithms based on the observation that road networks also have small separators (although they are not considered as planar graphs). Separators are a subset of the nodes or edges whose removal disconnect the graph. *Node separators* [57, 60] and *Arc Separators* [38], are the separator-based techniques whose preprocessing phase builds an *overlay graph* which contains additional information that is used during the query. This additional information is the shortcuts added to the original graph to preserve the shortest path between every two nodes. In *Node separators*, shortcuts are between the nodes of S where $S \in V$ are the nodes whose removal from the graph G, decompose it into several partitions. In *Arc Separators*, at first, graph G is decomposed to K roughly balanced blocks and then shortcuts are added between the boundary nodes of each cell.

2.4 Hierarchical Techniques

Hierarchical methods are a category of speed-up routing algorithms that exploit the inherent hierarchy of road networks [26]. The authors say that sufficiently long paths in a road network pass through a small network of important nodes and edges, such as highways [49]. Thus, in a preprocessing phase, they compute these important nodes and generate some auxiliary data that will be used during the query.

Reach-Based Routing [32] introduced by Gutman, is a shortest path algorithm optimized for road networks. Reach of a node u denoted by $r(u,P)$ is defined as $\min\{d(s,u), d(u,t)\}$ for a shortest path P from s-t that contains node u. Reach captures the importance of the node and use it to prune a Dijkstra-based search by checking two conditions $d(s,u) > r(u)$ and $d(u,t) > r(u)$ which means u is not in the s-t shortest path. Computing exact reach requires computing shortest path for all pairs of nodes which is too expensive in large road networks.

Another speed-up technique proposed was *Highway Hierarchy* (*HH*) [49]. The basic idea of the *HH* approach is that, outside some local areas around the source and the target nodes, only a subset of 'important' nodes has to be considered [49]. It has two subroutines. Edge reduction and node reduction. Node reduction removes (bypass) low degree nodes (nodes of degree one and two). Edge reduction removes non highway edges. Highway edges are those belong to some shortest path P from node s to node t such that they are not fully contained in the neighborhood of s nor in the neighborhood of t. Neighbors of a node v are those that are in a user defined radius of v.

Another speed-up technique which combines the idea of Highway Hierarchy and labeling algorithms (to be described later), is *Transit Node Routing* [13] (TNR). *TNR* is based on a simple observation intuitively used by humans. "When you drive to somewhere 'far away', you will leave your current location via one of only a few 'important' traffic junctions [transit nodes]". During preprocessing, it creates a distance table in which for every node, it stores its distance to all neighboring transit nodes and distance between all transit nodes as well. So a non-local shortest-path query can be reduced to a small number of table lookups. Although fast in the query, *TNR* needs considerably higher preprocessing time and space than previous approaches.

Highway-Node Routing (*HNR*) [55] , extends the idea from Highway Hierarchy and Multi-Level Overlay Graphs. Multi-Level Overlay Graph [36] is a separator based approach in the category of speed-up algorithms to find the shortest path. It finds a subset $S \subset V$ of important nodes and for each pair of nodes $u, v \in S$, an edge (u, v) is added to the overlay if the shortest path from u to v in G does not contain any other node w from S. In addition to edges between separator nodes of the same level, the overlay contains, for each cell on level i, edges between the reduced level i separator nodes and the interior level $(i-1)$ separator nodes. *HNR*, uses highway hierarchies to classify nodes by 'importance' and construct multi-level overlay graphs based on these nodes.

2.4.1 Contraction Hierarchy(*CH*)

In this section, we describe in detail the Contraction Hierarchy algorithm, as it is known to be one of the best algorithms in practice [12], and is also one of the algorithms that we implement and evaluate in this thesis.

Contraction Hierarchy [26] is a successor of *HH* and *HNR*, and uses a new approach regarding the node ordering and hierarchy construction. The algorithm obtains a faster query time using two queues containing so-called *forward* and *backward* edges of graph G that are denoted by $G_{\downarrow}$ and $G_{\uparrow}$. *CH* was invented with the motivation to optimize *HNR*, and has three phases: node ordering, node contraction, and query.

Before explaining each phase in detail, we define the term *overlay graph.*

An **overlay graph** [54] is a subgraph that preserves the shortest path distance of G. In other words, if $G' = (V', E')$ is an overlay graph of G, then $d_G(u, v) = d_{G'}(u, v)$ where $V' \subseteq V$ and E' is a set of edges whose weights preserve the shortest path distance.

2.4.1.1 Contraction

We construct a hierarchy by contracting the nodes in a specific order O, which we will describe in the next section. A node v is contracted by removing it from the network. The remaining graph after this contraction is an *overlay graph* that preserves all shortest paths as in the original graph. The combination of resulting overlay graph of Algorithm 1 and Order O, is called a Contraction Hierarchy (*CH*).

Contraction of a node v is achieved by replacing two-edge paths of the form $\langle u, v, w\rangle$ with a shortcut edge (u, w). See Figure 1 for an illustration. Note that this shortcut is added only if $\langle u, v, w\rangle$ is the only shortest path from u to w. So, before contracting the node v, and adding a shortcut to the graph, the algorithm searches for the shortest path from u to w; if different from $\langle u, v, w\rangle$, this shortest path is a *witness* that adding a shortcut is unnecessary. In other words, a path $P = \langle u, ..., w\rangle \neq \langle u, v, w\rangle$ such that $d(P) \leq d(\langle u, v, w\rangle)$ is a witness path. See Figure 2 for an example. The search for

finding this path is called *witness search*. The process of finding witnesses and adding shortcuts for a node u, if no witness is found, is called the contraction of u. See lines 3-5 of Algorithm 1.

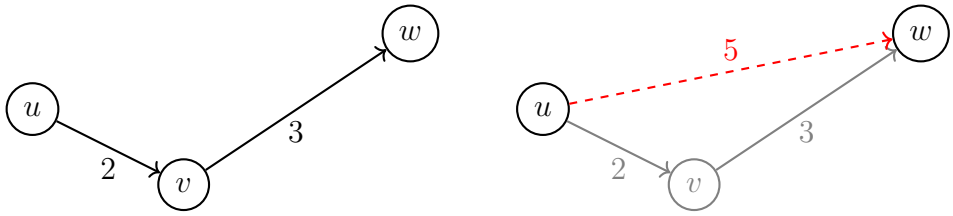


Figure 1: Shortcut added between u, w. Note that edge (u, v) and (v, w) do not belong to the graph any more.

Witness search for contracting of a node v is actually a problem of finding many-to-many shortest paths between u and w in G' for all incoming neighbors u and outgoing neighbors w of v. We want to know if there exists a shortest path whose $d(u, w)$ is equal to the shortcut length $l(u, v) + l(v, w)$. To do this search, we can simply perform a forward shortest path search in the current overlay graph from source nodes $\{u|(u, v) \in E'\}$ to all target nodes $\{w|(v, w) \in E'\}$. We can stop the search from u when it has reached distance $l(u, v) + max\{l(v, w) : (v, w) \in E'\}$. Such a limited search is called *local search*.

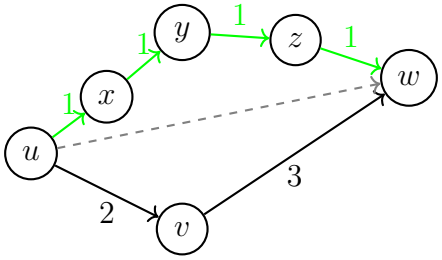


Figure 2: No shortcut added, path $P = \langle u, x, y, z, w\rangle$ already exist with $d(\langle u, v, w\rangle) >= d(P)$

Since this local search could be very costly, there are two ways to limit the search: *settled nodes limit* and a *hop limit*. In running *Dijkstra*'s algorithm, from a source node to a target node, we actually build the shortest path tree from source to target. During this process, nodes are either *unreached*, *reached* or *settled*. Settled nodes are the nodes that are already in the tree, while reached nodes are those that were added to the queue but are not yet in the shortest path tree. In settled nodes limit

approach, we do the same as *Dijkstra* but we limit the number of settled nodes to a certain maximum. This approach leads to a denser remaining graph and does not speed up the contraction a lot [27].

Hop limit approach, limits the number of edges of witness paths. It provides a better contraction speedup. As an example, in 1-Hop local search, we just scan all the incident edges of start node u to find the target node w. As we see in Figure 3, for 1-hope limit, we start scanning all the forward edges from node u and only edge u, w drawn in red is a 1-hop witness path. If we run the witness search without considering the mentioned limitations, it is called *Perfect witness search*. We should notice that these limitations do not influence the correctness of the contraction hierarchy as long as we add a shortcut (u, w) when we have not found a witness path, but they will cause a denser overlay graph and therefore slower query time.

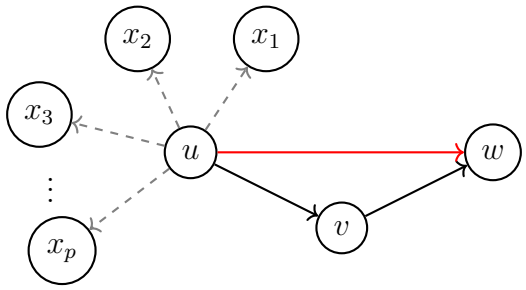


Figure 3: 1-Hop search starting from node v leads to find the edge (u, w) as witness path shown in red.

2.4.1.2 Node Ordering

As mentioned in the previous section, we contract nodes according to a node ordering. Any node order will result in a correct procedure. However, the selected node order has a significant impact on the running time performance of *CH*. Although finding an optimal node ordering was shown to be an NP-hard problem [14], already simple heuristics turn out to work quite well in practice.

A priority function can have several terms with different coefficients, forming a

Algorithm 1: CHPreprocessing($G = (V, E)$, Order O)

```
1: for all v ∈ V in increasing importance O do
2:    for all (u, w) s.t. u ∈ incoming(v), w ∈ outgoing(v) do
3:       dist ← length of u-w shortest paths in G' − {v}
4:       if dist > l(u, v) + l(v, w) then
5:          Add shortcut (u, w) to G' with l(u, v) + l(v, w)
6:       end if
7:    end for
8:    Contract(v)
9: end for
```

linear heuristic function. The main element of ordering heuristics proposed in the literature is the so-called *edge difference*, which is defined as the number of shortcuts introduced when contracting node v minus the number of incident edges of v. The aim of this term is to keep the edges in overlay graph as small as possible otherwise, the overlay graph will mostly look like a complete graph.

Note that Contraction Hierarchy not only aims to reduce the number of shortcuts but also as a second criterion, it aims to reduce the search space when performing a query. These two criteria are competing criteria. To take this into account, *uniformity* is introduced as another element in the ordering heuristic [26]. For example, as Figure 4 shows, if the graph is a path graph and we choose the illustrated node order, then no shortcut will be added to the graph, but at query time, we need to visit all the nodes. However if we contract every other node (non adjacent nodes), as shown in Figure 7, then the contracted graph will have more shortcuts but with smaller search space during the query time. So the idea is to contract nodes everywhere in the graph uniformly rather than to keep contracting nodes in a small area. For this purpose, Geisberger et al. proposed to add other terms to the formula [26], such as *contracted neighbors*, which is a counter that is incremented when a neighbor is contracted, and *original edge term* which is the number of original edges underlying shortcuts. This last term increases the space requirements but is also beneficial for path unpacking that we will talk about later in this section.

In order to contract nodes based on a good approximation of their priority, the

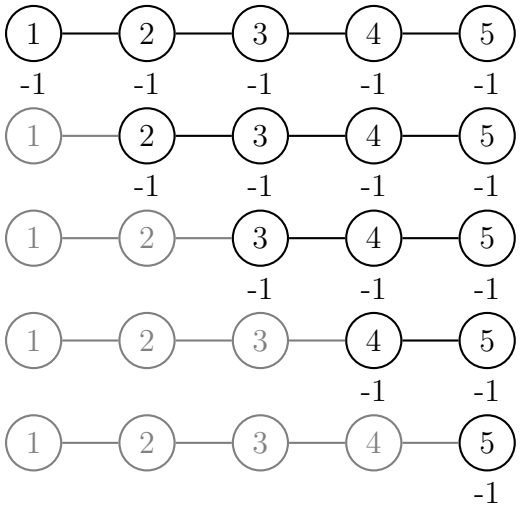


Figure 4: No shortcut added for the contraction of nodes in the path graph. Numbers in the nodes show the priority of each node for contraction and the number below each nodes shows the edge difference calculated for each node.

algorithm needs to update the priority of nodes. This is because the contraction of a node u can change the priority of other nodes especially nodes in its neighbors. See Figures 5 and 6. In [54], it's shown that updating all the neighbors of u is a good compromise between accuracy and performance.

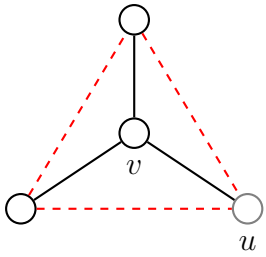


Figure 5: Before the contraction of node u, the edge difference of contracting node v is $3 - 3 = 0$.

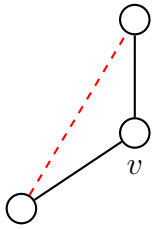


Figure 6: After the contraction of node u, the edge difference of contracting node v is $1 - 2 = -1$.

There exist three methods for this purpose:

1. Lazy update that updates the priority of a node before contracting it. If the priority of node v exceeds the priority of the next node $v\prime$ that is in the queue,

then we reinsert v into the queue and proceed with $v\prime$. This process continues until a consistent minimum is found.

2. Update only the priority of the neighbors of node v

3. Periodically reevaluate all the priorities and rebuild the priority queue.

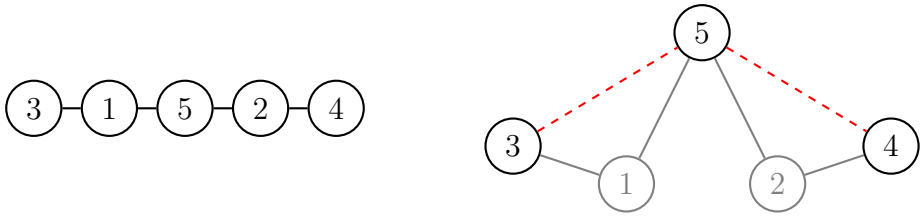


Figure 7: Two shortcuts added in this node ordering. As we see the search space is maximum two edges.

2.4.1.3 An Example of Contraction Hierarchy

Before explaining the query phase of Contraction Hierarchy, we illustrate all the steps of the preprocessing phase in an example. Figure 8 shows our example graph G. In this example for simplicity the priority heuristic is calculated as below

$$Priority = 2 * edgeDifference + contractedNeighbors$$

The edge difference term of a node is calculated as

$$edgeDifference = shortcuts - (IncomingEdges + OutgoingEdges)$$

Table 1 illustrates the initial priority of each node before contracting any node based on the given priority formula. To calculate the edge difference term of each node we need to know the number of shortcuts that will be added to the overlay graph after the contraction of a specific node. For example, to calculate the initial priority of node a, note that it has incoming edges from b, c, e and outgoing edges to b, c, d, e. To calculate the number of shortcuts, if node a gets selected for the contraction, the number of shortcuts is 2, one for $\langle e, b \rangle$ pair and another one for $\langle c, b \rangle$ pair. So the edge difference is $2 - (3 + 4) = -5$. At this point, a has no contracted neighbors, and

therefore the priority of a is -10. After calculation of the priority of each node of the graph, we select the node with the smallest priority number to contract. As Table 1 represents the first node that needs to be contracted is node a.

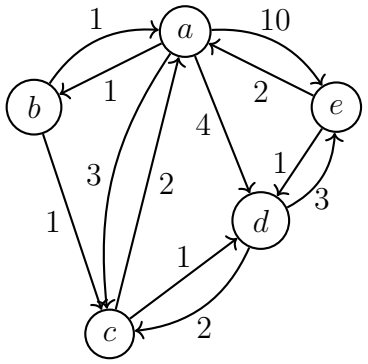


Figure 8: Directed graph $G = (V, E)$.

	a	b	c	d	e
$2 \times edgeDifference$	-10	-4	-6	-4	-8
$1 \times contractedNeighbors$	0	0	0	0	0
$priority$	-10	-4	-6	-4	-8

Table 1: Initializing the priority of each node.

When we contract node a, all of its incident edges will be removed from graph G and the new shortcuts will be added to G'. See Figure 9 for an illustration of graph G and $G\prime$ after contracting node a. As we see, two shortcuts $\langle e, b\rangle$ with weight 3 and $\langle c, b\rangle$ with weight 3 which are depicted with red dashed arrows have been added to the overlay graph.

Table 2 shows the priority of each node that is not yet contracted, after the contraction of node a. Note that now, the contractedNeighbors term has changed for all the nodes. For example, node c now has 4 incident edges, and if we contract it, two shortcuts $\langle b, d\rangle$ and $\langle d, b\rangle$ will be added , so its edge difference is -2, and since it has a contracted neighbor (namely a), its priority is $2 \times -2 + 1 = -3$. Based on Table 2, the next candidate for contraction is either node b or node e. We break ties in

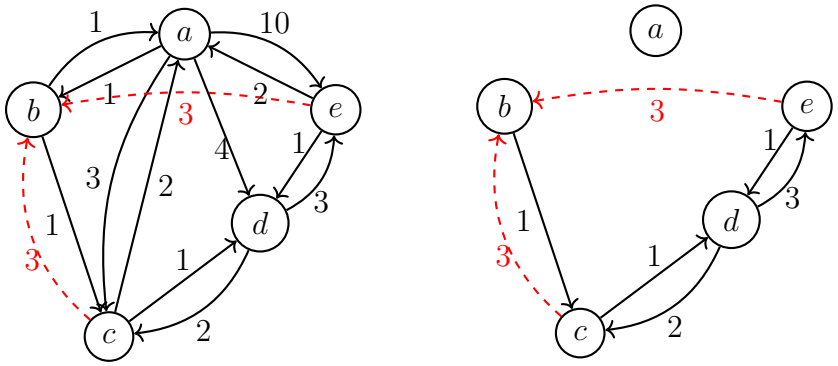


Figure 9: Overlay graph G' (Left) and graph G (Right) after the contraction of node a.

random and continue with node b. Contraction of node b, will not add any shortcut. See Figure 10 for an illustration.

	b	c	d	e
$2\times$	-6	-4	-4	-6
$1\times$	1	1	1	1
priority	-5	-3	-3	-5

Table 2: Priority of the remaining nodes after the contraction of node a.

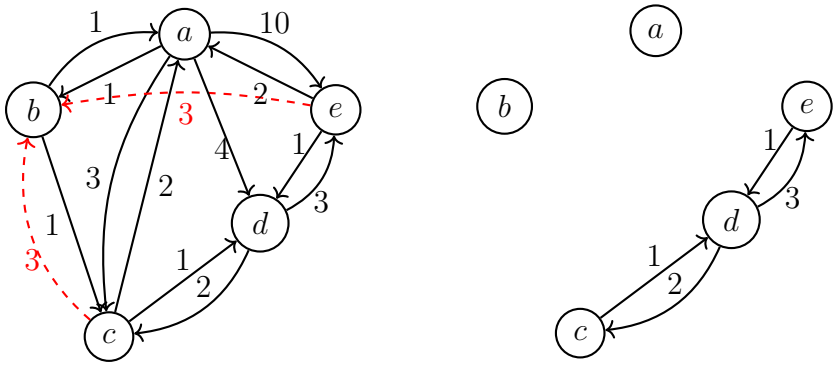


Figure 10: Overlay graph G' (Left) and graph G (Right) after the contraction of node b.

After the contraction of node b, as Table 3 shows, the next candidate is node d. See Figure 11 for an illustration of overlay graph and graph G after the contraction of node d. Note that contracting d adds two shortcuts $\langle e, c\rangle$ and $\langle c, e\rangle$ to the overlay graph. We continue with contracting first node c, and then node e, neither of which will add any shortcuts.

	c	d	e
$2\times$	-4	-4	-4
$1\times$	2	1	2
priority	-2	-3	-2

Table 3: Priority of the remaining nodes after the contraction of node b.

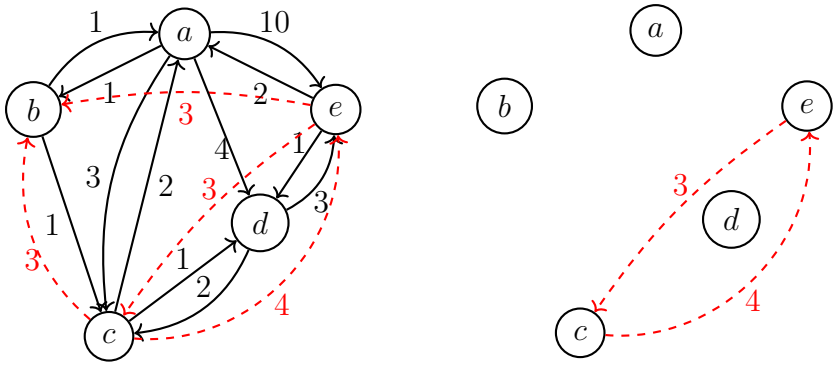


Figure 11: Overlay graph G' (Left) and graph G (Right) after the contraction of node d.

After contracting all the nodes of G, the final overlay graph will be as Figure 12. When the contraction is done, we have the order of nodes based on the order of contracting nodes. Having both overlay graph and order of nodes, result in a contraction hierarchy showed on the right of Figure 12. In our contraction hierarchy, we show the original edges with solid arrows and shortcuts with red dashed arrows.

2.4.1.4 Query

The contraction hierarchy splits the overlay graph G' into two graphs considering the node ordering $v_1, v_2, ..., v_n$: the *Upward graph* which is $G_\uparrow := (V, E_\uparrow)$ where $E_\uparrow := \{(u, v) \in E : O(u) < O(v)\}$ and the *Downward graph* which is $G_\downarrow := (V, E_\downarrow)$ where $E_\downarrow := \{(u, v) \in E : O(u) > O(v)\}$. Note that in Figure 13, the edges in $G_\uparrow$ and $G_\downarrow$ point upwards and downwards respectively. For the shortest path query from s to t, we perform a modified bidirectional *Dijkstra* shortest path search, consisting of a forward search in $G_\uparrow$ and a backward search in $G_\downarrow$. If there is a path from s to t in the original graph, then both forward and backward search will meet at a node v that

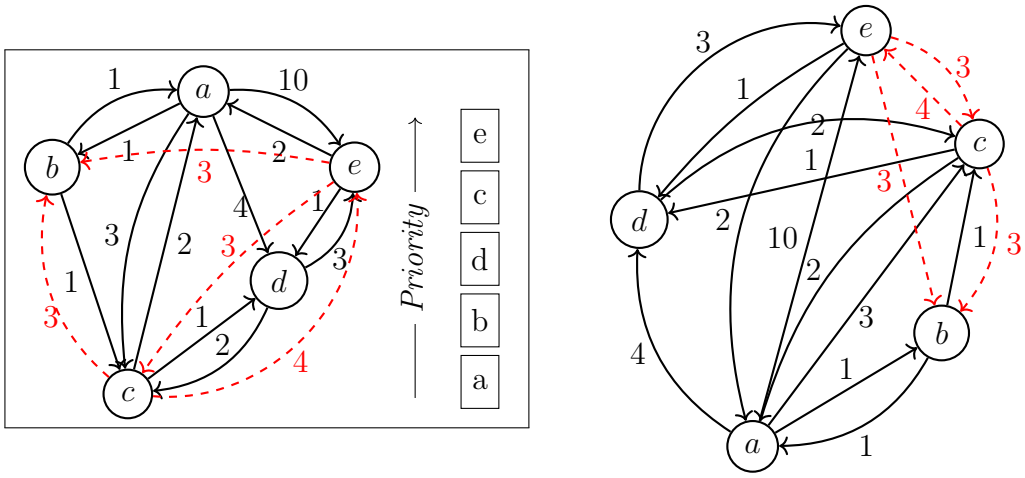


Figure 12: Final overlay graph. Overlay graph together with the order of nodes lead to a contraction hierarchy. On the right, the same graph is drawn but nodes of lower priority are placed below nodes of higher priority.

has the highest rank among all the other nodes in this shortest path. An *up-down* path P with respect to order O is a path that can be split into an upward path P_u and a downward path P_d. The upward path is a sequence of nodes of increasing rank, and the downward path is a sequence of nodes of decreasing rank. The upward and downward paths meet at the node with the maximum rank on the path which is called the *meeting node*. We have

$$d(s,t) = \min\{d(s,v) + d(v,t) : v \text{ is settled in both searches}\}$$

The query alternates between forward and backward search. Whenever we settle a node in one direction that is already settled in the other search, the path will be counted as a new candidate for the shortest path. Search is stopped in one direction if the smallest element in the priority queue is at least as large as the best candidate path found so far or if the queue is empty. Since additional settled nodes in this direction cannot possibly contribute to better solutions, it does not affect the correctness of the query. To illustrate the query phase in an example, we use the contraction hierarchy we built in the previous section.

Assume the query asks to find the shortest path from node d to node b. For the forward search, we need to only look at nodes with higher priority than d, outgoing from d, and for the backward search only look at nodes with higher priority than b,

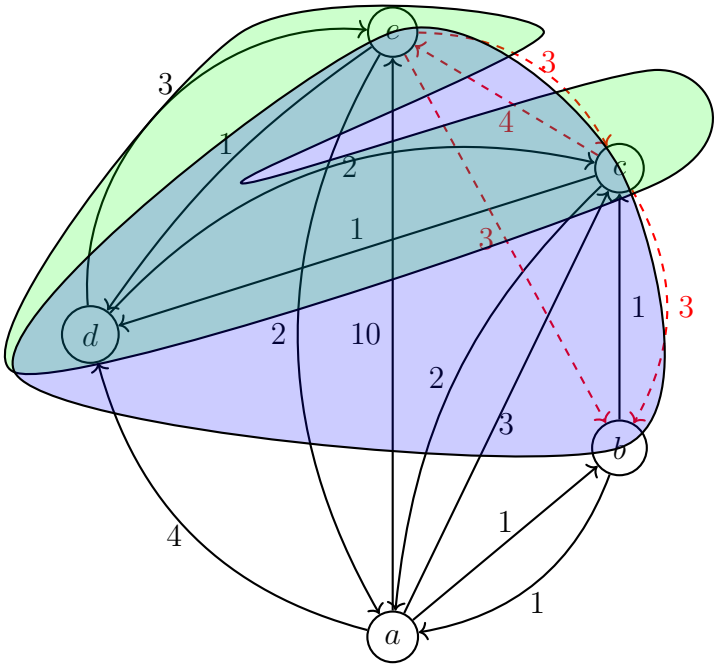


Figure 13: Query in contraction hierarchy of graph G.

incoming to node b. As we see in Figure 13, the search space from d is shown with green area and it contains node e and c which are higher than d. Also, search space to node b is shown with blue area which contains nodes d,e and c incoming to b. Among two possible candidate for shortest paths, $\langle d, c, b\rangle$ with weight 5 and $\langle d, e, b\rangle$ with weight 6, the first one has lower cost, and is therefore selected. Here, the meeting point is the node c.

There is an optimized variant of the query which is called *stall-on-demand* technique. Before settling a node v with distance $d(v)$ in forward search, we check the information in $G_{\downarrow}$ to see if there is downward edges (w, v) with $w > v$ such that $d(w) + l(w, v) < d(v)$. In this way, we can conclude that search can be stopped at node v with distance $d(w) + l(w, v)$ because the continuation of the search will be futile. It means we won't get a shortest path from v since distance to v is not optimal.

There is an additional step called *Path unpacking* if we want to obtain the actual shortest path. In this case, it is required to store a contracted node v together with its shortcut. In *CH*, since each shortcut (u, w) bypasses exactly one node v, therefore it is

sufficient to obtain a simple recursive unpacking routine. For the example above, the computed shortest path from e to c used the single edge (e, c), which was a shortcut edge. When outputting the actual path, we would unpack the shortcut edge to its corresponding two edge path, by retrieving the contracted node a.

The following theorems state the main results known about the complexity of Contraction Hierarchy.

Theorem 1 *[17] The number of nodes that get settled during a shortest path query in Contraction Hierarchy in a path graph is $O(n \log n)$ and in a tree graph is $O(\Delta \log \Delta + n)$ where Δ is the diameter of the tree graph.*

Theorem 2 *[17] For an arbitrary graph, deciding if there is an order that adds at most k shortcuts in Contraction Hierarchy is NP-complete. In fact, it is APX-hard to find an order that minimizes the number of shortcuts, and for all $\epsilon > 0$ it is NP-hard to approximate the optimal number of shortcuts within $7/6-\epsilon$.*

The Figure 14 obtained from [5] shows the number of visited nodes in *Dijkstra*, bidirectional *Dijkstra* and *Astar* algorithms and shows why contraction hierarchy works much better in practice.

2.5 Routing algorithms based on Labeling Techniques

The idea behind routing algorithms based on labeling technique, is to precompute labels for nodes, so that distances between pairs of nodes can be computed simply by examining the labels, and not the input graph. The idea of such graph labeling schemes proposed in [25, 46] is to compute a label $L(u)$ for every node u in the graph such that for every pair of nodes u, v, their distance can be obtained by only looking at $L(u)$ and $L(v)$ without using any additional information.

A special case of this approach is Hub Labeling (*HL*) [16], in which the label $L(u)$ contains a set of nodes, the so-called *hubs* of u, together with their distances from

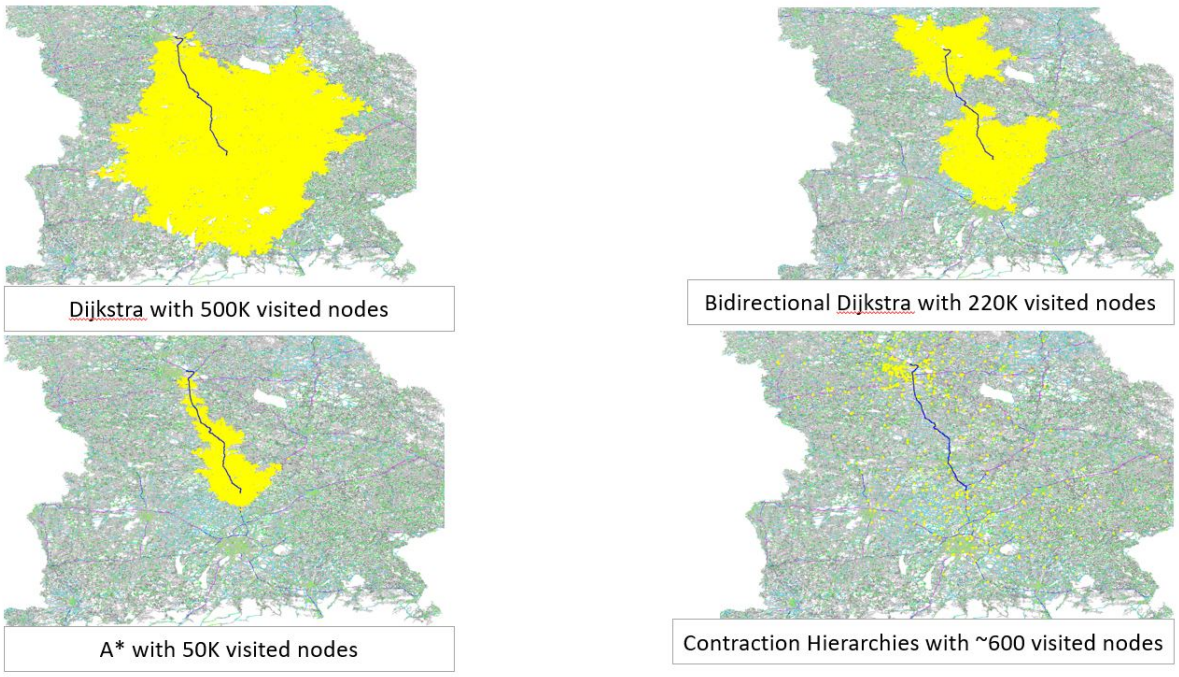


Figure 14: Efficiency of contraction hierarchy compared to *Dijkstra*, bidirectional *Dijkstra* and *Astar*

u. In order to find an existing s-t shortest distance for any source s and target t, $L(s) \cap L(t)$ must contain at least one node of this path. This property is called *cover property*. Then, the distance $d(s,t)$ can be determined in linear time that is proportional to the size of the inputs, i.e., to the length of the longest label [25]. It is computed by evaluating

$$d(s,t) = \min\{d(s,u) + d(u,t) | u \in L(s) \text{ and } u \in L(t)\}$$

For directed graphs, the label associated with every node u is split into two separate labels: the forward label $L_{OUT}(u)$ that has distances from u to the hubs, while the backward label $L_{IN}(u)$ has distances from the hubs to u. In this case, to answer the s-t shortest distance query it is enough to look for the common hub in $L_{OUT}(s) \cap L_{IN}(t)$. We say that the forward (backward) label size of v, $|L_{OUT}(v)|$ ($|L_{IN}(v)|$), is the number of hubs in $L_{OUT}(v)(L_{IN}(v))$. In Hub Labeling the aim is to reduce the average label size while preserving the cover property to answer the correct shortest distance query. See Figure 19 for an example of the labels constructed by a hub labelling algorithm.

In [8] Abraham et al. provided a practical implementation of Hub Labeling. They used the fact that in hierarchical algorithms such as *CH*, the set of nodes visited in forward search and backward search contains the corresponding labels. For example, labels of node v can be defined as forward (backward) search space of a *CH* query from v. In their label construction paradigm, they use a fast heuristic similar to stall-on-demand technique to decreases the average label size. In this method, when it performs a forward *CH* search (or backward) from a node v and want to scan a node w, with distance bound $d(w)$, it first checks all incoming edge to v considering $G_{\downarrow}$, if $d(w) > d(u) + (u, w)$, then $d(w)$ is incorrect and as a result it can safely remove w from the label of v, and it does not scan its outgoing arcs.

A special case of 2-hop cover labeling that is called *Hierarchical Hub Labeling* (*HH*) is proposed in [9]. In *HHL*, we first order nodes by importance and then transform this ordering to labels. In [9], a hub labeling is defined as a hierarchical hub labeling if, for two distinct nodes v, w, the relation $\precsim$ defined by $v \precsim w$ if and only if $L(v)$(forward or backward) contains w, is a partial order.

A hierarchical labeling L respects a total order O if the implied partial order is consistent with O. Based on this definition, given a total order O, a *canonical labeling* is the labeling that contains only the highest ranked node v in the forward label of s and in the backward label of t for the s-t shortest distance where v belongs to the s-t shortest path. Therefore $O(v) \geq O(s)$ and $O(v) \geq O(t)$. This also implies that the canonical labeling respects O. Canonical hub labeling is minimal since removing any hub w from a label set will break the cover property. In [9], author proposed to use *CH* orders, in order to make canonical labeling. They also proposed a technique that makes even smaller labels than *CH*-induced labels for a wide range of networks by choosing the most important nodes greedily, based on how many shortest paths they hit.

2.5.1 Hierarchical Hub Labeling Using Pruned Landmark Labeling

To construct a hierarchical hub labeling given a total order O, one can push the hub information from a higher ranked node to its reachable lower ranked nodes such that all shortest paths in input graph are covered. However, the labels created in this method are not minimal. More recently, Akiba et al. in [10] proposed a fast exact method referred to by *Pruned landmark Labeling (PLL)* algorithm, which efficiently computes a canonical labeling (also called as *index*) from a given node order. Starting from empty labels, *PLL* processes nodes based on a given order O, from most to least important node. In the iteration corresponding to node v, it runs Breath First Search (BFS) in an undirected unweighted graph from node v. Assuming that v is the i-th node in the ordering, we construct the labelling L_i by adding to the previous labeling L_{i-1}. Algorithms 2 and 3 show the pseudocode for label construction in an undirected unweighted graph in the *PLL* algorithm. We describe the algorithm for directed weighted graphs in more detail because that is the algorithm implemented in this thesis.

Akiba et al. proved that *PLL* is correct and produces a minimal labeling. In [20], it is proved that *PLL* is also hierarchical and canonical.

2.5.1.1 *HHL* for Directed Weighted Graphs

In this section, we show how to modify the algorithm for undirected graphs described in the previous section for directed and weighted graphs. The pseudocode is given in Algorithms 4 and 5. Note that we denote the directed by $G = (V, E)$ the incoming neighbors of u by $N_{IN}(u)$ and its outgoing neighbors by $N_{OUT}(u)$.

As in the undirected case, starting from empty labels, *PLL* processes nodes based on a given order O, from most to least important node. In the iteration corresponding to node v, it runs *two* pruned versions of *Dijkstra*'s algorithm from the node v. Next we describe the modified *Dijkstra* procedure in the forward graph corresponding to a node v. To decide if the labeling needs to be modified, when processing the next

Algorithm 2: PrunedBFS(G, v_i, L_{i-1})

1: $distance_i[v_i] \leftarrow 0$ and $distance_i[v] \leftarrow \infty$ for all $v \in V \backslash \{v_i\}$
2: $L_i[v] \leftarrow L_{i-1}[v]$ for all $v \in V$
3: $Q \leftarrow v_i$
4: **while** $Q \neq \emptyset$ **do**
5: $\quad u \leftarrow Q.Dequeue()$
6: $\quad$ **if** $Query(v_i, u, L) > distance_i[u]$ **then**
7: $\quad\quad L_i[u] \leftarrow L_{i-1}[u] \cup \{(v_i, distance_i[u])\}$
8: $\quad$ **end if**
9: $\quad$ **for all** $w \in N(u)$ s.t. $distance_i[w] = \infty$ **do**
10: $\quad\quad distance_i[w] = distance_i[u] + 1$
11: $\quad\quad Q.Enqueue(w)$
12: $\quad$ **end for**
13: **end while**
14: Return L_n

Algorithm 3: Compute Labels_BFS(G, V, Order O)

1: $L_0 \leftarrow \emptyset$ for all $v \in V$
2: **for** $i = 0, 1, 2.., n$ **do**
3: $\quad L_i \leftarrow PrunedBFS(G, v_i, L_{i-1})$
4: **end for**
5: Return L_n

node, say u with least estimated distance d from v, it first computes a v-u distance by performing an *HL* query with the current partial labels. We represent this query by $\delta = Query(v_i, u, L_{i-1})$. Here L_0 is the empty label, and L_i is created from L_{i-1} using the information obtained by the i-th pruned *Dijkstra* from node v_i in the order of v_1, v_2, . . . , v_n, as already mentioned. If the labels of u and v do not intersect in L_{i-1}, the query distance will be infinity ($\delta = \infty$). If $\delta \leq d$, the v-u pair is already covered by previous hubs and the algorithm prunes the search (ignores u). In the case $\delta > d$, it adds $(v, d(v, u))$ to the labelling $L_{i,IN}(u)$ and relaxes the outgoing edges of u as usual. The *Dijkstra* procedure in the backward graph works analogously. In the same situation, it adds $(v, d(u, v))$ to $L_{i,OUT}(u)$ for all scanned nodes u.

In order to respond to the shortest path query, we need to keep track of the node's predecessor and save it together with other information in the labels. So, labels would be a set of triples instead of pairs $((v, \delta(v,u), w))$ where the third element,w, would be the parent of node u, in the pruned *Dijkstra* search from v toward u. In a directed graph like a road network, this needs to be done for both forward and backward labels of each node. The Algorithm 6 in Section 2.5.1.4 shows the shortest path query function in *HHL*. Since in this thesis our focus in on road networks which are directed weighted graphs, and we are interested to know the shortest path, in Algorithm 4, we show the label construction paradigm that enables us to retrieve the shortest path from the labels after the construction. We also provide an example for *PLL* for our directed weighted example graph introduced in previous section.

Algorithm 4: PrunedDijkstra(G, v_i, L_{i-1}, dir)

$distance_i[v_i] \leftarrow 0$ and $distance_i[v] \leftarrow \infty$ for all $v \in V \backslash \{v_i\}$
$L_{(dir)i}[v] \leftarrow L_{(dir)i-1}[v]$ for all $v \in V$
$Q \leftarrow (v_i, distance_i[v_i], none)$
while $Q \neq \emptyset$ **do**
  $u \leftarrow Q.ExtractMin()$
  $L_{(!dir)i}[u] \leftarrow L_{(!dir)i-1}[u] \cup \{(v_i, distance_i[u], parent[u])\}$
  for all $w \in N_{dir}(u)$ **do**
    $dist = distance_i[u] + l(u,w)$
    if $dist < distance_i[w]$ and $dist < Query(v_i, w, L, dir)$ **then**
      $distance_i[w] = dist$ and $parent[w] = u$
      if $\{w, distance_i[w], parent[w])\}$ is in Q **then**
        $Q.Update(\{w, distance_i[w], parent[w])\})$
      end if
    else
      $Q.Enqueue(\{(w, distance_i[w], parent[w])\})$
    end if
  end for
end while
Return $L_{(dir)n}$

Algorithm 5: Compute Labels_Dijkstra(G, V, Order O)

1: $L_0 \leftarrow \emptyset$ for all $v \in V$
2: **for** $i = 0, 1, 2.., n$ **do**
3: $\quad L_{(OUT)i} \leftarrow PrunedDijkstra(G, v_i, L_{(OUT)i-1}, forward)$
4: $\quad L_{(IN)i} \leftarrow PrunedDijkstra(G, v_i, L_{(IN)i-1}, backward)$
5: **end for**
6: Return L_n

The theorem below gives upper bounds on the label size and preprocessing time of *PLL* method.

Theorem 3 *[10] Label construction needs $O(nm)$ time and $O(n^2)$ space.*

A more precise bound can be proved for graphs of bounded treewidth:

Theorem 4 *[10] Let w be the tree-width of G. There is an order of nodes with which the PLL takes $O(wmlogn + w^2nlog^2n)$ time for preprocessing, stores an index(Labeling) with $O(wnlogn)$ space, and answers each query in $O(wlogn)$ time.*

2.5.1.2 Example of *HHL* for a Directed Weighted Graph

Here we use the same example graph as the one used in Section 2.4.1.3 to illustrate *CH*. We order nodes based on their degree, that is, the sum of the number of incoming edges and the outgoing edges to that node. In this case, the priority of nodes to construct labels for our example graph is as shown in Figure 15. The node a with 7 incident edges is the first node to start constructing labels with, and node b with 3 incident edges is the least important and last node to construct labels with.

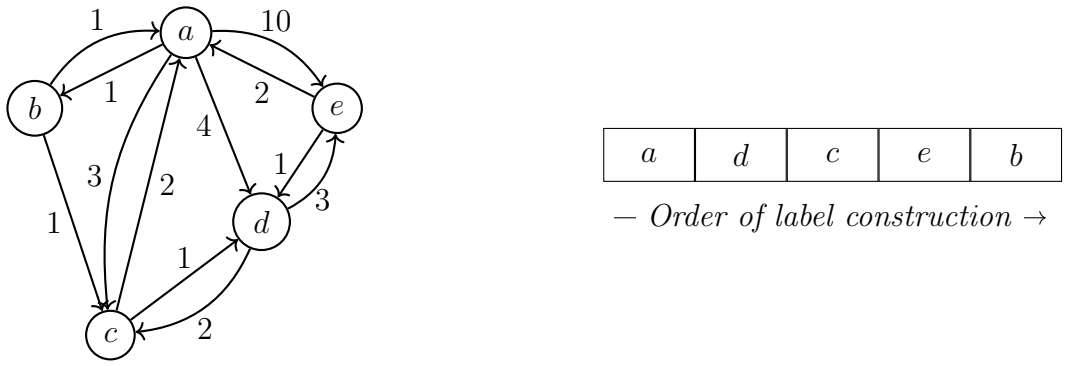


Figure 15: Directed weighted graph $G = (V, E)$ with the order of nodes for label contraction

Starting from node a, and in forward direction, initially $(a, 0, none)$ is in the queue. This will cause the addition of triple $(a, 0, none)$ to be added to the backward label of a (Line 6 of Algorithm 4). In the next step we check all the outgoing edges of a and put the corresponding neighbors in the queue only if their distance to a via the edge is less than the previous estimate of distance to a (held in $distance_a$ array and initialized to ∞) and is also less than the distance from a as given by $Query(a, u, L, forward)$ which is the distance given by the current labeling (Line 9). Figure 16 shows the priority queue after reaching each outgoing edges of node a.

As the algorithm progresses it chooses triple $(b, 1, a)$ from the queue and adds a to the backward label of b and checks the outgoing edges of b to either insert new nodes in the queue or update the weight of the existing ones. For example, c is the neighbor of node b and its distance from a through b will be 2 while $distance_a(c)$ is 3, and using the existing hub labeling, the query returns ∞, so we replace $distance_a(c)$ by 2, change the parent to b, and update the queue entry for c accordingly. In this

$(b, 1, a)$
$(c, 3, a)$
$(d, 4, a)$
$(e, 10, a)$

Figure 16: Priority queue status after visiting node a of our example graph in *HHL* construction.

case the next node that get extracted from the queue will be c with weight 2. Figure 17 shows the label of each node after visiting all the remaining nodes in the queue (d and e).

a_{IN}	$(a, 0, -)$
a_{OUT}	
b_{IN}	$(a, 1, a)$
b_{OUT}	
c_{IN}	$(a, 2, b)$
c_{OUT}	
d_{IN}	$(a, 3, c)$
d_{OUT}	
e_{IN}	$(a, 6, d)$
e_{OUT}	

a_{IN}	$(a, 0, -)$
a_{OUT}	$(a, 0, -)$
b_{IN}	$(a, 1, a)$
b_{OUT}	$(a, 1, a)$
c_{IN}	$(a, 2, b)$
c_{OUT}	$(a, 2, a)$
d_{IN}	$(a, 3, c)$
d_{OUT}	$(a, 4, c)$
e_{IN}	$(a, 6, d)$
e_{OUT}	$(a, 2, a)$

Figure 17: Label of nodes of graph G after the forward iteration (on the left) and after the backward iteration (on the right) of node a.

This finishes the call to $PrunedDijkstra(G, a, L_{(OUT)}a, forward)$. Next we perform the call to $PrunedDijkstra(G, a, L_{(IN)}a, backward)$. Here we scan the incoming edges to a and add an entry to the backward label of the nodes if necessary.

We now describe the forward iteration for node d, that is, the execution of $PrunedDijkstra(G, d, L_a, forward)$. After processing the first node in the priority queue which is d itself, we will have c and e in the queue with weight 2 and 3

respectively. The algorithm continues with c, and it inserts d in backward label of c since there is no intersection node in forward label of d and backward label of c before adding this new entry ($Query(d, c, L, forward) = \infty$). Also the $distance_d[c] = \infty$. After processing node e as the node with minimum distance in the queue, next we process the outgoing neighbors of c, namely a. As an outgoing arc of c, node a with the distance 4 is the next node that we extract from the queue. However, since the intersection of forward label of d and backward label of a have an intersection with the distance of 4, so we do not need to add any new entry to cover $\langle d, a \rangle$ path.

a_{IN}	$(a, 0, -)$	
a_{OUT}	$(a, 0, -)$	
b_{IN}	$(a, 1, a)$	
b_{OUT}	$(a, 1, a)$	
c_{IN}	$(a, 2, b)$	$(d, 2, d)$
c_{OUT}	$(a, 2, a)$	
d_{IN}	$(a, 3, c)$	
d_{OUT}	$(a, 4, c)$	$(d, 0, -)$
e_{IN}	$(a, 6, d)$	$(d, 3, d)$
e_{OUT}	$(a, 2, a)$	

Figure 18: Label of nodes of graph G after the forward iteration of node d.

After iterating forward and backward through all the nodes based on their order in decreasing importance, the final labels of all nodes in the graph G will be as shown in Figure 19.

a_{IN}	$(a, 0, -)$			
a_{OUT}	$(a, 0, -)$			
b_{IN}	$(a, 1, a)$	$(b, 0, -)$		
b_{OUT}	$(a, 1, a)$	$(d, 2, c)$	$(c, 1, c)$	$(b, 0, -)$
c_{IN}	$(a, 2, b)$	$(d, 2, d)$	$(c, 0, -)$	
c_{OUT}	$(a, 2, a)$	$(d, 1, d)$	$(c, 0, -)$	
d_{IN}	$(a, 3, c)$	$(d, 0, -)$		
d_{OUT}	$(a, 4, c)$	$(d, 0, -)$		
e_{IN}	$(a, 6, d)$	$(d, 3, d)$	$(e, 0, -)$	
e_{OUT}	$(a, 2, a)$	$(d, 1, d)$	$(e, 0, -)$	

Figure 19: Final label of nodes of example graph G.

2.5.1.3 Different Node Ordering

In this section we explain three heuristics for node ordering that have been used to construct labels in *HHL* algorithm.

Degree. In a graph, the *Degree* of a node could be a sign of the strength of that node. The idea of using the degree for the ordering is that nodes with a higher degree have more connections to many other nodes and therefore many shortest paths would pass through them [10].

Betweenness. The *betweenness* of a node is defined as the number of shortest paths that pass through that node. Abraham et al. [9] use shortest-path trees to calculate the betweenness. The betweenness of a node is the sum of the descendant sizes of that node in all shortest-path trees. To implement this, they build n full shortest path trees, each rooted at one of the nodes in the graph. The tree rooted at node v contains all the uncovered shortest path starting at v. Therefore, the number of descendants of node w is the number of uncovered shortest paths that start at v and contain w. So, it is the number of the shortest path that will be hit if we pick w as the next most important node in label construction. This is denoted by $\sigma(w)$. There are two variants of this approach. *Path-greedy*, at each iteration, picks the node as the next most important one, which covers the most uncovered paths by previous nodes. In [51], they assign the value $1/\sigma(w)$ as the value of the importance. We denote it by *BTW-PG*. *Label-greedy*, at each iteration, picks the node as the next most important one, which maximize $\sigma(w)/x_w$, where x_w is the number of nodes such that w goes into their labels. We denote it by *BTW-LG*.

2.5.1.4 Query

To answer distance queries, we apply merge-join-like algorithms. For this purpose, pairs in labels need to be sorted by nodes. In [10], it does not sort labels explicitly by storing ranks of nodes, but instead it stores them by construction based on their rank. That is, instead of adding the pair (v_k, δ) to the label of a node u in the k-th pruned *Dijkstra* from node v_k, it adds a pair (k, δ). Thus, because pairs are added from higher rank nodes to those with lower rank, all the labels are automatically

sorted.

Algorithm 6 shows the pseudocode for the Shortest Path Query function, which returns the shortest path from u to v. Recall that $P[u]$ is the predecessor of node u.

Algorithm 6: SPQuery(u, v, L)

1: **if** $(u == v)$ **then**
2: Return u or v
3: **end if**
4: **if** $(u == None)$ **then**
5: Return v
6: **end if**
7: **if** $(v == None)$ **then**
8: Return u
9: **end if**
10: $MP = L_f(u) \cap L_b(v)$ `// Finds the Meeting Point of` (u, v)
11: Return Concatenate(SPQuery$(P[u], MP, L)$, SPQuery$(MP, P[v], L)$)

Referring to the example in Section 2.5.1.2, and the labeling given in Figure 19, in order to find the shortest path from node a and d, it is enough to intersect the forward label of a and the backward label of d. We see that the meeting point is node a and the shortest path distance is 0+3=3. To get the actual path, we need to recursively intersect the label of the parent of a and d with the label of meeting. In this example, since the parent of a is empty it means that a is the end node. So, we only need to intersect forward label of a and backward label of c which is the parent node of d. In the next step, similarly, we need to intersect the forward label of a and backward label of b until we find the complete path.

2.6 Empirical Studies of Static Routing Algorithms

In [26], the efficiency of *CH* has been examined with respect to the effect of different priority terms in the ordering heuristic and it has been shown that for a good heuristic

to lead to an efficient query performance we need a combination of terms. They compare this efficiency with *HNR* algorithm which is mentioned as the fastest previous algorithm. In [27], Geisberger at al again have shown that *CH* algorithm and its variants are more effective than previous speed-up and non speed-up algorithms from both preprocessing and query time point of view. They also show the robustness of *CH* for distance as edge weight. In [21], the authors introduce a new node ordering for *CH*, which we call *R-CH* and empirically study its performance versus other node ordering heuristics. In [8], another node ordering heuristic which we call it *S-CH* was proposed for *CH*. In our experiments, we compare (for the first time) the efficiency of *CH* using *R-CH* and *S-CH* proposed in [8] and [21].

A comparison of a distance query time of *CH* algorithm with *HL* [8] algorithm proved that *HL* algorithm is faster than any existing method on continental road networks. A similar result was reported in [9] for *HHL* and *CH* comparison on a variety of networks, such as social networks, artificially constructed networks, and some road networks. The authors of [10] compare the performance of *PLL* with *HHL*. Their experimental results show that for a variety of networks including road networks, the label construction of *PLL* method is significantly more efficient than label construction using path greedy and label greedy method for building *HHL*. It also examines the size of the label of different networks using *PLL* and three different node ordering (Random node ordering, Degree and Closeness) and shows that Random strategy is much worse than other two strategies. Degree and Closeness can find central vertices successfully. In our experiments, we use *PLL* and four different node orderings (*Degree*, *BTW-LG*, *BTW-PG* and *GP-FC20*).

The authors of [42] compare *HHL* with different ordering schemes (Degree, Betweenness and a new ordering scheme called significant path) with *CH* on the different networks such as undirected road networks with respect to response time and index size. They show that the shortest distance query is more efficient in *HHL* and the shortest path query is more efficient in *CH*. In our experiment we run *HHL* with *PLL* implementation and compare it with *CH* on directed road networks.

Chapter 3

Dynamic Routing Algorithms

In some types of networks, it may happen that edge weight often changes, or edges might be added or removed from the network. In fact, this is likely to happen in road networks in the real world because of changes in traffic. So, it is not enough to know about the shortest path between nodes in the static network. We also want to efficiently find the shortest path and shortest distance route in a dynamic network. One obvious solution to respond to the shortest path query in a dynamic scenario using speed-up algorithms, is to run the preprocessing phase every time that a modification happens in the network. Although this approach guarantees that queries are as fast as queries in static conditions, re-running of the preprocessing phase can be costly. As a consequence, other approaches have been developed to solve this problem. In this section, we study the dynamic variants of two speed-up algorithms for which we already explained their static versions in Sections 2.4.1 and 2.5.1.

3.1 Customizable Contraction Hierarchy (*CCH*)

As we noted in Section 2.4.1, a limitation of *CH*, is that in its preprocessing phase, the weight of edges of the network needs to be known. For this reason, *CH* is called *metric-dependent*. In other words, a significant change in the edge weights of the

network may require a costly computation of the preprocessing phase. One way that is proposed in [27], is to keep the original node order and instead of re-contracting nodes from scratch, update existing shortcuts to comply with the changes and then identify the subset U of nodes whose contraction has to be repeated to add new shortcuts. In t in V that are not in U is not necessary because no new shortcuts need to be added. In the mobile scenario of *CH* in [27], for dynamic edge weight changes, it does not even re-contract a set of affected nodes, but it adapts these changes at query time by bypassing affected shortcuts and going down the hierarchy. The same technique is used in [55] that adapts the query algorithms to answer weight change updates.

An interesting technique suggested in [24] is to make the contraction phase completely *metric-independent*. The authors suggest that the shortest path query might not only consist of source and destination but also of a set of parameters which determine under which metric the optimal path in the road network is to be computed (These parameters are actually edge weights or other metrics). Thus, it moves all the necessary metric-dependent work to the query phase. However, this makes the query phase significantly slow.

Most recent approaches divide the preprocessing phase into two separate phases [21,22], the metric-independent phase and the metric-dependent phase. In the metric-independent phase, it only considers the topology of the input network which is stable most of the time, and builds a contraction hierarchy based on that. In the second phase, which is quite fast, it applies the user-defined metrics to the contraction hierarchy built in the first phase. This also makes the query phase very fast. In these techniques, when edge weights change, they only run the fast metric-dependent phase without requiring the execution of the whole preprocessing phase. In this section, we will describe the approach of [22] and in Chapter 4 we will run experiments using different node ordering techniques.

3.1.1 Building a Contraction Hierarchy

In this section, we describe how to build a metric-independent contraction hierarchy. The main building block for this, is an algorithm called metric-independent *Nested Dissection Orders* [28] (ND-orders). Given a partition of the network, ND-orders produces an ordering of the nodes. (In Section 3.1.1.1, we describe several ways of partitioning the network that we use in our experiments in this thesis.) We now use this ordering to create a chordal graph by examining nodes in order from lowest to highest and adding edges to make the set of neighbors of the current node that have higher rank a clique. This chordal graph constitutes our metric-independent contraction hierarchy.

In *CCH* context, a chordal graph is constructed by adding these extra edges as shortcuts to the graph by first dropping all edge directions and edge weights of G. More precisely, an unweighted node contraction of node v in G consists of removing v and inserting edges between all neighbors $N(v)$ of higher order than v, if not already present. Given the order O, the *core graph* $G_{O,i}$ is obtained by contracting all nodes $O(1)...O(i-1)$ in order. The original graph G augmented by the set of shortcuts is denoted by $G^*_O = \bigcup_i G_{O,i}$. Note that the order O for the final graph G^*_O is a perfect elimination ordering [1]

Figure 20, represent an example of creating a chordal graph and its upward directed representation of it for graph G that we used in previous sections. In this figure, the rank of each node is shown next to it. We start by node d with rank 0. Since e and c, are neighbors of it with higher rank, and there is no edge between them, we add a new edge (dashed, red line). Note that no other shortcuts need to be added.

The created hierarchy corresponding to G^*_O can be represented by an *upward directed graph* which is denoted by $\hat{G_O}$. By upward directed graph, we mean a graph such that all its edges are directed upward. See Figure 20. Given a node order O, every undirected graph can be represented by an upward directed graph. i.e., every edge $(O(i), O(j))$ with $i < j$ is replaced by a directed edge $\{O(i), O(j)\}$.

[1] A perfect elimination ordering of a graph is an ordering of the nodes of the graph such that for each node v, the neighbors of v that come after v in the ordering as well as v form a clique. A graph is chordal if and only if it has a perfect elimination ordering.

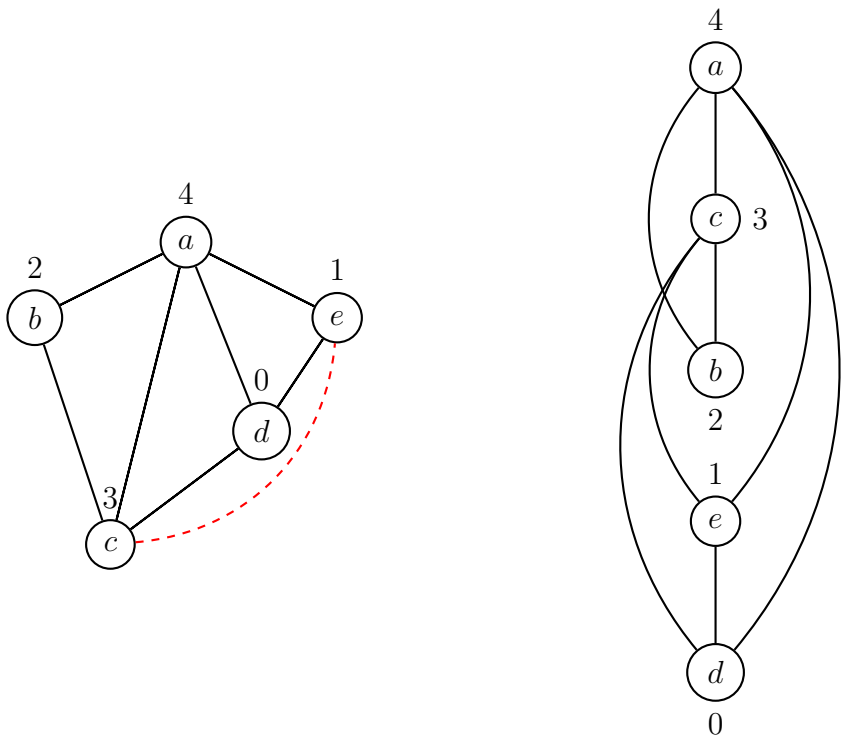


Figure 20: Chordal graph of graph G and its upward directed graph with respect to a ND-order O.

Theorem 5 *[21] Running time complexity of contracting all nodes in metric-independent Contraction Hierarchy is* $O(\tilde{m}\alpha(n))$ *where* $\tilde{m}$ *is* $\sum d(x)$ *and* $d(x)$ *is the degree of* x *just before* x *is contracted.*

3.1.1.1 Different Graph Partitioning Algorithms

Graph Partitioning is the problem of cutting a graph into separate regions of roughly equal size while the number of edges between regions is as small as possible. A *cut* is a set of edges whose removal from the graph partitions the set of nodes into two subsets V_1 and V_2. It turns out that good graph separators are the ones that contribute to many shortest paths. After finding good separators we need to create an ordering for the nodes. There should be an algorithm to build contraction order from graph partitioning. Finding a good order of nodes has a direct influence on the time required for the customization and the shortest distance queries. The approach used in all graph partitioning techniques mentioned below is nested dissection.

In [22], metric-independent ND-orders need good graph bisectors which are NP-hard in theory. However, there are previous studies that apply graph partitioning techniques with nested dissections and show that it works well even for continental-size road networks [31,50]. Nested dissection is an approach that recursively computes order for V_1, V_2 and Q where V_1 and V_2 constitute the partition determined by the cut Q resulting from a graph partitioning tool. We would like to study the effect of different graph partitioning algorithms on the query time of *CCH* and in upcoming sections, compare this experiment's result with other basic and speed-up routing algorithms.

In this section, we describe several node partitioning algorithms that can be used by ND-order to create the ordering to be used by *CCH*. Implementations of all the algorithms described in this section were available [31], and we have used them in our evaluations described in Chapter 4. We give brief descriptions here for completeness.

FlowCutter [33] is an algorithm that heuristically finds balanced connected cuts in a flow network. In FlowCutter, *s-t* cuts are computed in increasing size. It uses unit capacity for every edge ($c : E \longrightarrow \{0, 1\}$). In the beginning, s is the only source node and t is the only target node that is arbitrarily chosen. The algorithm starts by computing a maximum s-t flow. With respect to this flow, there is a source-side cut C_S and a target-side cut C_T. If the cut C_S is balanced (it can also be C_T) then the algorithm finishes, otherwise, one of the sides, source-side or target-side is smaller. Assuming the source side is small, the algorithm now transforms non-source nodes into additional source nodes to invalidate existing cut and computes a new, more balanced s-t cut, which will be the second cut in the sequence. This is done in two steps, first mark all the nodes on the source side as a source node and secondly pick one node on the target side as the source node. This one node is called *piercing node*. It ensures that we will find a different cut in the next iteration. Choosing a good piercing node influences the quality of the cut. As the algorithm iterates, it maintains a maximum flow. Based on this flow, there is a source-side cut and a target-side cut. It computes a set of cuts or separators that trade off cut and separator size, respectively, for a parameter ϵ called *imbalance*. The imbalance is a parameter that represents the maximum acceptable unbalance of cuts.

To combine FlowCutter with nested dissection to generate node order for graph G,

we bisect G along with the node separator Q created by FlowCutter into subgraphs with V_1 and V_2 the sets of nodes. Then it recursively computes orders for V_1 and V_2. The order of G is the order of V_1 followed by the order of V_2 followed by the nodes in Q in an arbitrary order. It continues the recursion until it reaches a clique or tree. For a clique, any order is optimal. For trees, an order can be derived from an optimal node ranking which is a ranking that the largest rank assigned to some node is as small as possible among all rankings [52]. In our experiments, we denote the combination of ND-order and Flowcutter as obtained from [6] by *FC* followed by the number of source-target pairs.

InertialFlow [53] combines sorting and maximum flow. In the first step, it projects the geographical coordination of nodes to their closest point on a line $l \in \mathbb{R}_2$ and then sorts them by their appearance on l. In the next step, it defines S and T such that if a node is in first $\lfloor \alpha.|\mathrm{V}| \rfloor$ it belongs to S and if a node is in the last $\lfloor \alpha.|\mathrm{V}| \rfloor$, it belongs to T for a parameter $\alpha \in [0, 0.5]$. Then it computes the maximum flow between S and T and returns a corresponding minimum s-t cut. In the implementation of the [31], they use 0.2 for α, and instead of a line, they use four directions West-East, South-North, Southwest–Northeast, and Southeast–Northwest as suggested in [33] since their experiments have shown this approach works well for the road networks. The nested dissection technique that is used for node ordering is the same as FlowCutter and it only parallelizes the partitioning algorithm. We denote the combination of ND-order and Inertial Flow as obtained from [6] by *IF*.

RoutingKit also implements InertialFlow followed by a nested dissection approach to generate orders for *CCH*. It executes InertialFlow with three different α, [0.25, 0.33, 0.4] and then picks the one that creates more balanced cuts. We denote this technique by *IFR*.

InertialFlowCutter [31]. It combines the idea of InertialFlow to use geographic coordinates with the incremental cut computations of FlowCutter. It improves FlowCutter by initializing S and T in the same way as Inertial Flow, but with a smaller α. In addition, it chooses multiple nodes as piercing nodes at once. This is called *bulk piercing*. In this way, it enumerates multiple InertialFlow cuts simultaneously, without having to restart the flow computations. In [31], it is suggested to run multiple InertialFlowCutter to improve solution quality. Therefore, they run $q \in \mathrm{N}$ different

instances of InertialFlowCutter with different directions. In [31] implementation, [4, 8, 12, 16] are used as instances with different directions and we use the same setup. We denote InertialFlowCutter by *IFC* followed by the number of q as instance number in our experiments.

KaHiP [50]. It is a multi-level evolutionary and a general-purpose graph partitioning technique. It creates the most balanced cuts compared to other graph partitioning techniques. KaHiP, uses a multi-level approach for partitioning large graphs. A partitioned graph consists of blocks which are nodes and connectivity between blocks which are edges. Given a partition, the gain of a node v in block A with respect to a block B is defined as the reduction in the cut when v is moved from block A to block B. It consists of two components. The first component is local searches on pairs of blocks that share a non-empty boundary in partitioned graphs(i.e. edges in the partitioned graph). Local search finds an eligible boundary node v in block A having a maximum gain. In another word, a node v that maximizes the reduction in cut size when moving it from block A to block B. If there is more than one such node, it breaks ties randomly. These local searches are not restricted to the balance constraint of the graph partitioning problem and are undone after they have been performed. The second component uses the information gathered in the first component. That means it builds a model using the node movements performed in the first step enabling the algorithm to find combinations of those node movements that maintain balance. There are different mode of KaHiP implementation such as *fast* which creates *Fast Partitioning*, *eco* which creates *Good Partitioning* and *Strong* which creates *Very Good Partitioning* [56]. We use strong mode. We also use version 2.11 for our experiment and denote it by *K2.11*. The same nested dissection technique as InertialFlow is used for KaHiP.

Metis [39]. Like KaHiP this is a general-purpose graph partitioning tool but faster than KaHiP. It works in three phases. In the *coarsening phase,* the graph G which is shown by G_0, is transformed into a sequence of smaller graphs, each with fewer nodes $G_1, G_2, ..., G_m$ such that $|V_1| > |V_2| > |V_3| > \cdots > |V_m|$. In the *partitioning phase*, a 2-way partition $|P_m|$ of the graph $G_m = (V_m, E_m)$ is computed that partitions V_m into two parts, each containing half the nodes of G_0. Finally, in the *uncoarsening phase*, the partition P_m of G_m is projected back to G_0 by going through intermediate

partitions P_{m-1}, P_{m-2},...,P_1, P_0. We used Metis 5.1.0, available from the authors' website. Metis has its own node ordering tool called *ndmetis* which we used. It also uses a nested dissection approach.

3.1.2 Customization

In the previous section, we described how to build a metric-independent contraction hierarchy for the undirected unweighted graph corresponding to the original network. The direction of edges, as well as edge weights, are introduced in the customization phase. To start with, for each edge in the metric-independent contraction hierarchy G^*, we assign two weights. If in the original road network graph G, both directed edges were present, we simply use the same weights. If the edge is a one-way street in the original road network, then either upward or downward weights are set to ∞.

We discuss three types of metrics: *respecting* metric, *customized* metric denoted by m_c and *perfect customized* metric denoted by m_p.

Next we describe how to create these three types of metrics. To make a respecting metric, it is enough to assign to all edges of $G^{\hat{}}_O$ that already exist in G their input weight and ∞ to all other edges (as described above for the initialization). Figure 22 illustrates the respecting metric for the directed weighted graph G that we already used for our examples. For the red line between nodes c and e, since we do not have equivalent edge in original graph G, we set both forward and backward weights to infinity.

Before describing the procedure to make customized and perfect customized metrics, we need to define three types of triangles related to an edge. Consider an edge (x, y) with $O(x) < O(y)$ and let z be a node adjacent to both x and y. If $O(z) < O(x)$ then $\triangle xyz$ is called a *lower triangle*; if $O(z) > O(y)$, then $\triangle xyz$ is called an *upper triangle*; otherwise it is called an *intermediate triangle*. Figure 21 illustrates the lower triangle of edge (y, z), intermediate triangle of edge (z, x) and upper triangle of edge (x, y).

Now we are ready to describe how to make a customized metric. The main idea

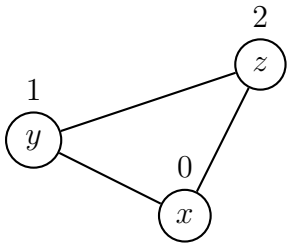


Figure 21: Lower Triangle, Intermediate Triangle and Upper Triangle of three nodes x, y and z. Rank of each node is shown next to it.

is to make all lower triangles observe the triangle inequality. In these triangles they need to check whether the path $\langle x, y, z\rangle$ is shorter than $\langle x, y\rangle$. In the affirmative case, $m_c(x, y)$ is decreased to the distance of this path. We process edges by starting with incident edges of the node with lowest rank and proceed in order of rank. When the algorithm finishes, for all the edges, the triangle inequality holds.

Since we are customizing a directed weighted graph, we consider both m_d for downward weight of an edge and m_u for upward weight of an edge. Thus for every lower triangle $\{x, y, z\}$ of (x, y), we calculate

$$m_u(x, y) = \min\{m_u(x, y), m_d(x, z) + m_u(z, y)\}$$

$$m_d(x, y) = \min\{m_d(x, y), m_u(x, z) + m_d(z, y)\}$$

Figure 23 shows an example of building customized metric. We start with all the edges of the node with the smallest rank which is node d and check for each edge their lower triangle and based on the minimum value of all these lower triangle we set their forward and backward weight. Then we continue the same process with nodes e, b, c and a. When processing the edges incident on e, we observe that for edge (c, e), when considering its lower triangle $\triangle cde$, we obtain a shorter forward distance of 3 and backward distance of 4 (from c), and we set the metric appropriately.

3.1.2.1 Perfect Customization and the *CCH+P* Algorithm

For a perfect customization metric, we need to ensure that *all triangles* (upper, intermediate, and lower) observe the triangle inequality. In a perfect customization, we

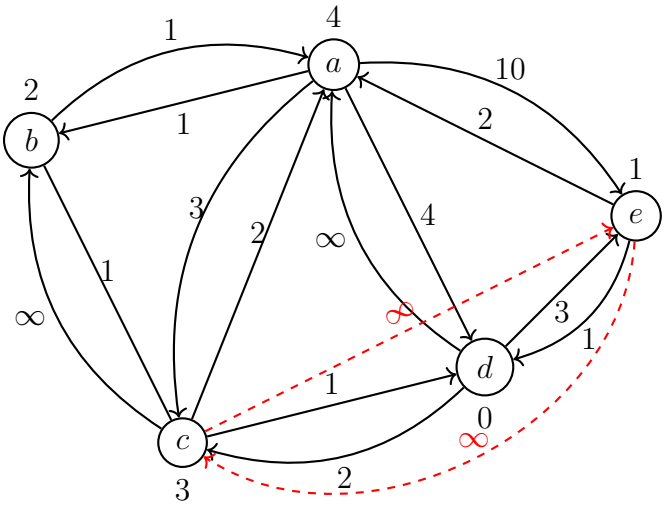


Figure 22: Respected metric of graph G.

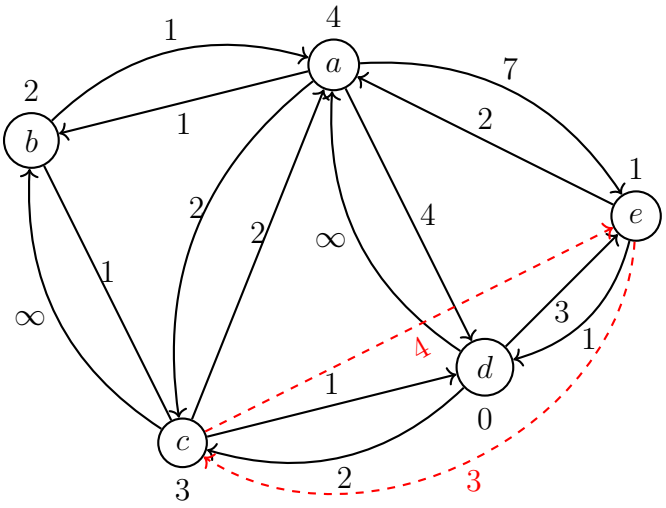


Figure 23: Customized metric of graph G.

want to compute m_p given the customized metric m_c. At first it copies all the m_c values into m_p. Then in increasing rank of node x for all (x, y) edges, it enumerates all its intermediate and upper triangles $\triangle xyz$ and checks whether the path via z is shorter or not. If it is shorter then it changes the value of m_p to that path. In other words, it computes $m_p(x, y) = \min\{m_p(x, y), m_p(x, z)+m_p(z, y)\}$. After perfect customization, the $m_p(x, y)$ is equal to the shortest path distance of (x, y). In the

directed weighted graph, for intermediate triangles it computes

$$m_u(x, y) = \min\{m_u(x, y), m_u(x, z) + m_u(z, y)\}$$

$$m_u(x, y) = \min\{m_d(x, y), m_d(x, z) + m_d(z, y)\}$$

and for upper triangles in computes

$$m_u(x, y) = \min\{m_u(x, y), m_u(x, z) + m_d(z, y)\}$$

$$m_d(x, y) = \min\{m_d(x, y), m_d(x, z) + m_u(z, y)\}$$

.

Note that after a customized metric or perfect customized metric is created, we can optimize the contraction hierarchy by removing some edges. There are two variants of this optimization. The first variant removes all the edges (x, y) whose weight after basic customization is not equal to the shortest path distance between x and y. The second version removes these edges and additionally, removes other (x, y) edges if and only if an upper or intermediate triangle $\{x, y, z\}$ exists such that the shortest path from x to y over z is not longer than the x-y shortest distance. In a directed weighted graph, a perfect witness search may remove edges only in one direction. In this condition, like the original *CH*, it runs two searches: upward search and downward search. The edge (x, y) is removed from the upward search graph if and only if an intermediate with $m_u(x, y) = m_u(x, z) + m_u(z, y)$ exists or an upper triangle with $m_u(x, y) = m_u(x, z) + m_d(z, y)$ exists. The same operation needs to be done analogously for the downward graph. In our implementation of *CCH+P*, we use the second optimization. Figure 24 illustrate the graph G after performing perfect customization. Dashed gray arrows are edges removed from the graph.

3.1.3 Query

There are three algorithms to compute the shortest path distance in *CCH* between two nodes s and t in a directed weighted graph G. The first algorithm is the same as the bidirectional *Dijkstra* that is explained in Section 2.4.1.4. The second one which is also explained in that section is the stall-on-demand version of the first algorithm that is also applicable in *CCH*.

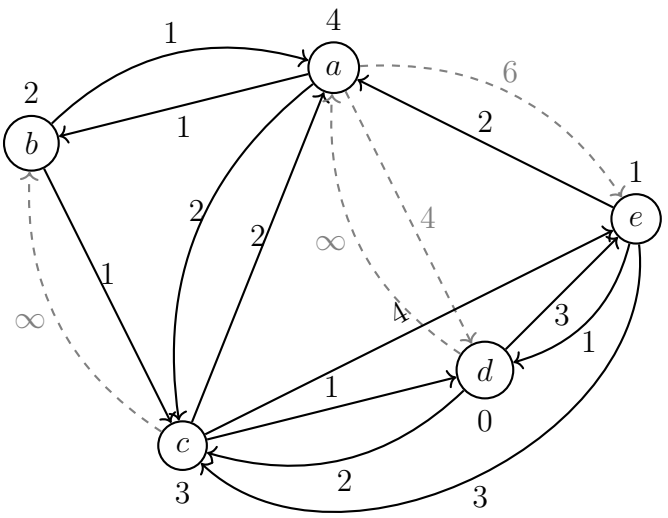


Figure 24: Perfect metric of graph G.

The third algorithm is an elimination tree search. Starting from the upward directed graph, we now build an elimination tree $T_{G,O}$, which is a tree directed toward its root, to be used in the query phase. The parent of node u is its upward neighbor v in the upward directed graph of minimum rank. The set of nodes on the path from v to the root is the set of nodes in the search space of v denoted by $SS(v)$ [15]. $SS(s)$, is all the ancestors of s by increasing rank.

Figure 25 shows the construction of the elimination tree of graph G^* for the graph G that we used in the previous chapter.

In the elimination tree search, for every node v we have two tentative distances $d_f(v)$ and $d_b(v)$ that are initially set to ∞. It has four steps. In the first step, it finds the Lowest Common Ancestor (LCA) of s and t. To do this, it checks all the ancestors of s increasing by their rank and analogously for t until it finds node x as LCA. In the second step, it iterates over all nodes y on the tree path from s to x and relaxes all forward edges of such y. In the third step, it does a similar operation from t to x. In the fourth step, it iterates over all nodes y from x to the root n of the tree and relaxes all forward and backward edges. During the fourth step, it finds a node z for which $d_f(z) + d_b(z)$ is the minimum. It means that the up-down shortest path from s to t goes through z.

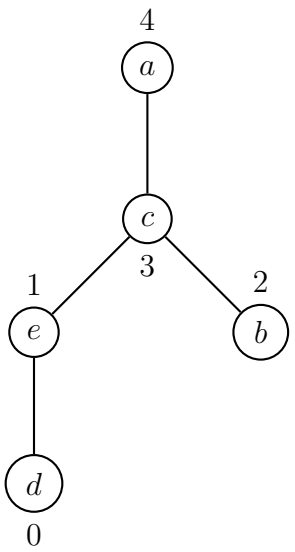


Figure 25: Elimination tree construction of graph G with respect the order O.

3.2 Dynamic Hierarchical Hub Labeling

The hub labeling algorithm was adapted to dynamic graphs or large evolving networks in [11] for the first time. In this study, Akiba et al. have shown how to update the labels in order to answer correct shortest distance where edges have been added to the network. They prove the correctness of their algorithm in undirected unweighted graphs and then very briefly describe how it can be generalized to cover not only edge insertion but also the weight decrease of the edges, in both undirected graphs and directed graphs. They call these label updates the *incremental index updates*. In their approach, they do not remove outdated labels since these labels never underestimate the shortest path distance and removing them is a costly process.

Recently the authors of [18] expanded this approach to cover edge removal or edge weight increase in a network to make a fully dynamic hub labeling algorithm; their algorithm is called a *decremental index update*. In Sections 3.2.1 and 3.2.2, we will explain the incremental and decremental update algorithms respectively.

3.2.1 Incremental Update

The key idea of the Incremental update algorithm is explained by [11, 18] as follows. Suppose the edge weight of a directed edge (a, b) decreases, and the shortest path distance from v to u decreases as a result, where v and u are any arbitrary nodes in the graph. Then we can conclude that

1. All shortest paths from v to u must pass through the edge (a, b) and
2. The shortest path distance from v to every vertex w on the sub-path from b to u must also decrease and
3. All such shortest paths must pass through a node in $L(a) \cup L(b)$, except for the path from a to b.

These observations imply that it is enough to run *Dijkstra* starting at node b but rooted at nodes in $L(a) \cup L(b)$ to see if labels need to be updated. We now describe the algorithms in detail.

Let us assume that, in an undirected unweighted graph, the edge insertion happened between two nodes a and b in graph G. As stated above, instead of starting BFS from every node in the graph (inserting $(v_k, 0)$ into the initial queue of the BFS), we start the search from node b rooted at node v_k where $v_k \in L(a)$. That is, we insert $(b, d(v_k, a)+1)$ into the queue. Algorithms 7 and 8 give the pesudocode of incremental update in undirected unweighted networks.

Next we describe the algorithm for directed weighted graphs. Note that while it is mentioned in [11] that the approach can be extended to weighted undirected and directed graphs, no details are given. In [18], the extensions are described in detail for directed weighted graphs, though no pseuodocode is given. Essentially, when the weight of the edge between a and b decreases, we need to run both forward and backward *Dijkstra*-like searches. The first type of search starts at node b and is rooted at a node v_k where $v_k \in L_{IN}(a)$. That is, it inserts $(b, d_{IN}(a, v_k) + l(a, b))$ into the queue where $d_{IN}(a, v_k)$ is the shortest distance between node a and v_k and $l(a, b)$ is the weight of the edge (a, b) that is decreased. Note that $d_{IN}(a, v_k)$ can

be obtained by running *Dijkstra* from v_k to a. This search continues from b in the forward direction and first meets all the forward neighbors of b. When the algorithm meets a node u, it checks the distance d from v_k to u as obtained by a query to the original labelling (i.e. by running $Query(v_k, u, L, forward)$), and compares it with the newly computed *Dijkstra* distance from v_k to u that is denoted by δ. If $d \leq \delta$, then the shortest path distance from v_k and u has not changed, and no labels need to be changed. Otherwise, it either needs to add an appropriate hub with distance δ to the label of u if it doesn't already exist or update labels. Note that, since we need to preserve the canonical labeling property after any change in the labels, the algorithm needs to check the rank of v_k and u. If $O(v_k) < O(u)$, then we need to check the backward label of u ($L_{IN}(u)$) for the appropriate operation (adding or modify label) and if $O(v_k) > O(u)$, then algorithm needs to check the forward label of v_k ($L_{OUT}(v_k)$) and either add to its label or modify it.

The second search, starts at node a and is rooted at v_k where $v_k \in L_{OUT}(b)$. In other words, it inserts $(a, d_{OUT}(b, v_k) + l(a, b))$ into the queue where $d_{OUT}(b, v_k)$ can be obtained by $Query(b, v_k, L, forward)$. The search continues from a in backward direction and meets all the backward neighbors (and neighbors of these neighbors as well) of a. When the algorithm meets a node u, it checks the distance from v_k to u in backward direction and computes $d = Query(v_k, u, L, backward)$. The rest of algorithm is analogous to the first search and the only difference is the direction in which the search runs.

In dynamic *HHL*, we also want to retrieve the actual shortest path. So we need to follow the tuple structure of labels instead of pairs as explained in Section 2.5.1.4. So, the first search algorithm starts with adding the triple $(b, d_{IN}(a, v_k) + l(a, b), a)$ to the queue, and for backward search we start with adding $(a, d_{OUT}(b, v_k) + l(a, b), b)$.

Algorithms 9 and 10 represent the pesudocode of incremental update in directed weighted networks, and also illustrate how the the actual shortest path is retrieved in incremental updates.

Algorithm 7: Inc_PLL$(a, b, l(a, b), L)$

for all $v_k \in L(a) \cup L(b)$ from a lower i **do**
  if $v_k \in L(a)$ **then**
    ResumePrunedBFS$(G, v_k, b, d(v_k, a) + 1, L)$
  end if
  if $v_k \in L(b)$ **then**
    ResumePrunedBFS$(G, v_k, a, d(v_k, b) + 1, L)$
  end if
end for

Algorithm 8: ResumePrunedBFS(G, v_k, u, δ, L)

$Q.Enqueue((u, \delta))$
while $Q \notin \emptyset$ **do**
  $(v, \delta) \leftarrow Q.Dequeue()$
  if $Query(v_k, v, L) \leq \delta$ **then**
    continue
  end if
  $L(v) \leftarrow L(v) \cup \{(v_k, \delta)\}$
  for all $w \in N(u)$ **do**
    $Q.Enqueue((w, \delta + 1))$
  end for
end while

Algorithm 9: Inc_PLL$(a, b, l(a, b), L)$

for all $v_k \in L_{IN}(a) \cup L_{OUT}(b)$ from a lower k **do**
  if $v_k \in L_{IN}(a)$ **then**
    ResumePrunedDij$(G, v_k, b, d(v_k, a) + l(a, b), a, L, forward)$
  end if
  if $v_k \in L_{OUT}(b)$ **then**
    ResumePrunedDij$(G, v_k, a, d(b, v_k) + l(a, b), b, L, backward)$
  end if
end for

Algorithm 10: ResumePrunedDij$(G, v_k, u, \delta, p, L, dir)$

$Q.Enqueue((u, \delta, p))$
while $Q \notin \emptyset$ **do**
  $(v, \delta, P[v]) \leftarrow Q.ExtractMin()$
  if $Query(v_k, v, L, dir) \leq \delta$ **then**
    continue
  end if
  if $O(v_k) < O(v)$ **then**
    $L_{IN}(v) \leftarrow L_{IN}(v) \cup \{(v_k, \delta, P[v])\}$
  end if
  if $O(v) < O(v_k)$ **then**
    $L_{OUT}(v_k) \leftarrow L_{OUT}(v_k) \cup \{(v, \delta, P[v])\}$
  end if
  for all $w \in N_{dir}(v)$ **do**
    $Q.Enqueue((w, \delta + l(u, w), v))$
  end for
end while

An example for Incremental update is shown in Figure 26 where the weight of edge (c, a) has decreased from 2 to 1. In this case we need to run the Algorithm 9 for nodes $\{a, c, d\} \in L_{IN}(c)$ and $\{a\} \in L_{OUT}(a)$. Since $d \in L_{IN}(c)$, we start the *Dijkstra* search rooted at node d in forward direction, starting from node a. The tentative distance of this search results in distance 3 for the path $\langle d, c, a \rangle$, while the distance resulting from a query using the current labeling is 4. Thus, we need to update the forward label of d, since $O(a) < O(d)$. First we check if a is already one of the hubs of node d, which is true in our example, so we only need to change the distance from 4 to 3.

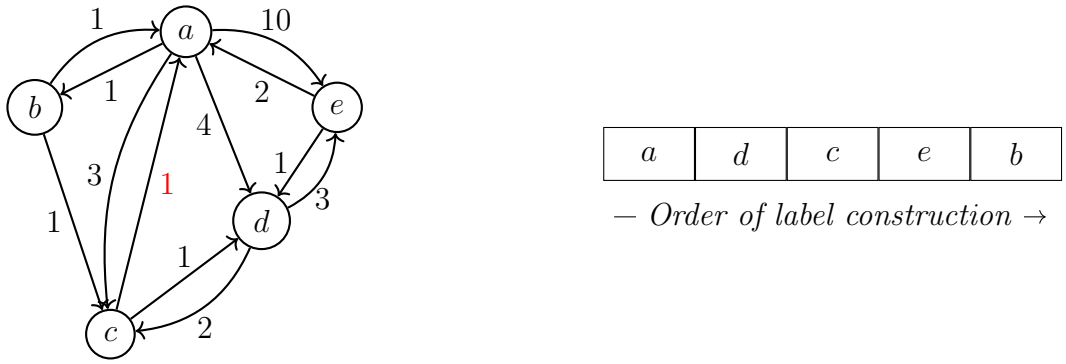


Figure 26: Edge weight decrease of edge (c, a) of graph G.

After running the Algorithm 10, the labeling of graph G will look like as the Figure 27.

The following theorem states the complexity of Incremental Update.

Theorem 6 *[11] The incremental update operation for edge (a, b) needs $O(l^2 s)$ time, where ℓ is $|L(a) \cup L(b)|$ and s is the maximum number of nodes visited during each ResumedPrunedBfs. In the worst case, it takes time $O(n^3)$.*

3.2.2 Decremental Update

In this section, we describe how to update the labeling in the case when an edge is removed, or the weight of an edge is increased, as given in [18]. This operation is called Decremental update. A real world example this situation is when a traffic jams

a_{IN}	$(a, 0, -)$			
a_{OUT}	$(a, 0, -)$			
b_{IN}	$(a, 1, a)$	$(b, 0, -)$		
b_{OUT}	$(a, 1, a)$	$(d, 2, c)$	$(c, 1, c)$	$(b, 0, -)$
c_{IN}	$(a, 2, b)$	$(d, 2, d)$	$(c, 0, -)$	
c_{OUT}	$(a, 1, a)$	$(d, 1, d)$	$(c, 0, -)$	
d_{IN}	$(a, 3, c)$	$(d, 0, -)$		
d_{OUT}	$(a, 3, c)$	$(d, 0, -)$		
e_{IN}	$(a, 6, d)$	$(d, 3, d)$	$(e, 0, -)$	
e_{OUT}	$(a, 2, a)$	$(d, 1, d)$	$(e, 0, -)$	

Figure 27: Removed outdated entries from labeling L of graph G after weight decrease of the edge (c, a).

happens in the road network. In the following, we assume that we had a graph G_{i-1} and a labelling for it, and the edge (a, b) has either been removed, or its weight has increased to give graph G_i. The decremental update operation can be broken into 3 phases. The first phase is detecting the nodes that are affected by this decremental change. In the second phase, the algorithm removes the outdated labels using the set of affected nodes that are found in the previous phase. In the third phase, the appropriate hubs with their correct distance need to be added to the labels.

3.2.2.1 Detecting Affected Nodes

We say that a node v is *affected* if the removal of an edge (a, b) from graph G_{i-1} has affected a shortest path between v and any other node u that is induced by labeling L and passes through edge (a, b). The set of such affected nodes is denoted by AFF_{ab}, and it can be partitioned into two disjoint subsets AFF_a and AFF_b as the nodes closer to a and b respectively. In addition, in [18], they observed that if u and v are two affected nodes such that $u \in AFF_a$ and $v \in AFF_b$, then a hub h of pair (u, v) is

also affected and either $h \in AFF_a$ or $h \in AFF_b$.

The algorithm to find AFF_a mimics a BFS search rooted at node b and starts at a and it prunes the search when it reaches a node that is not affected. We put a in AFF_a as the first affected node and continue the search by visiting the neighbors of a. In this search let u be each neighbor of a, the algorithm needs to detect whether u is affected or not. For this purpose it needs to checks two conditions. Let h be the hub of pair (u, b) in L of G_{i-1}. The first condition (Condition 1) checks whether the hub between this pair is already in the set of affected nodes or not. If h is already in the list of affected nodes then u is counted as an affected node as well. Note that since we run BFS from a then the hub h of pair (u, b) is analyzed by the algorithm before u. The second condition (condition 2) checks whether h is either u or b. In this case, u is affected if a shortest path between u and b passes through edge (a, b). To do this, it is enough two run two BFS searches one from u to b to compute the distance $d_{i-1}(u, b)$ and another one from u to a to compute the distance $d_{i-1}(u, a)$ in G_{i-1}.

1. $h \in AFF_a$
2. $(h = u \vee h = b) \wedge (d_{i-1}(u, b) = d_{i-1}(u, a) + 1)$

Algorithm 11 gives the pseudo code of this algorithm for AFF_a in undirected unweighted networks. At the end of running the algorithm 11, AFF_a will contain all the affected nodes of node a. An analogous algorithm can be used to calculate AFF_b. In [18] it is assumed that the shortest path is not unique and different shortest paths of (u, b) might contribute different hubs. So the algorithm repeats conditions 1 and 2 for the set of hubs of shortest paths of (u, b) (lines 11 to 15). However, in our implementation of detecting the affected nodes, since we use the Akiba label construction approach which breaks ties on-line in favour of the most important nodes, there is always a single hub for the pair (u, b).

For the case of a directed weighted graph G, for the edge (a, b) which is removed or whose weight has increased, we have either $d_{i-1}(u, a) < d_{i-1}(u, b)$ or $d_{i-1}(a, u) > d_{i-1}(b, u)$ or both. In this case, we use a different method to divide AFF_{ab} into sets AFF_a and AFF_b. For labeling L and removal of the edge (a, b) (or increase in its

weight), if there exists a shortest path (v, u) induced by L from node v to node u that passes through the edge (a, b) then we say v belongs to AFF_a and u belongs to AFF_b. Note that v might be the endpoint of a shortest path passing through (a, b) from a different origin node, in other words, it is possible that $v \in AFF_b$ as well. That is, AFF_a and AFF_b are not necessarily disjoint.

Both algorithms explained for undirected unweighted graphs are applicable for directed weighted graphs. The only modification is the direction of searches and instead of BFS, we need to run a *Dijkstra*-like search. For instance, to calculate the AFF_a, at the beginning, only node a is in the AFF_a. As the algorithm progresses in the backward direction and meets the backward neighbors of a, it checks two conditions below:

1. $h \in AFF_a$
2. $(h = u \vee h = b) \wedge (d_i(u, b) = d_i(u, a) + l(a, b))$

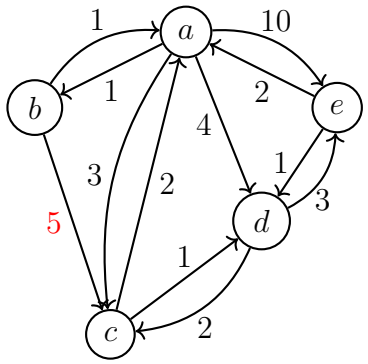
The affected nodes of b can be computed analogously to affected nodes of a . In this case, the *Dijkstra* search starts at b and is rooted in a in the forward direction. It checks the forward neighbors of b and for each neighbor we check the similar conditions as 1 and 2. Algorithm 12 shows the pseudo code of our algorithm to compute AFF_a.

Figure 28 shows an example of detecting the affected nodes in our example graph G. In this example we assume that the weight of edge (b, c) has increased from 1 to 5. To detect affected nodes of node b, that is, AFF_b, we run Algorithm 12 to execute a backward search (*Dijkstra*) rooted at c, and starting at b. At the beginning, b is in the queue, so we iterate over all the incoming neighbors of b and for each such neighbor u, we find its hub with c, and check the condition on line 11. If this condition holds, it means that the path from neighbor u of b to node c has changed due to the weight increase of edge (b, c), then u is an affected node; we set its status as visited and enqueue it. In this example, when we consider the incoming neighbor node a, we see from Figure 19 that the hub to go from a to c is a itself, and since the condition in line 11 of Algorithm 12 is met (because the shortest path from a to c before the edge weight increase went through the edge (b, c), that is, $d_{i-1}(a, c) = d_{i-1}(a, b) + l(b, c)$) , we conclude that node a is affected. Indeed, we can see that the weight of the path

Algorithm 11: AFF_a In undirected unweighted graph

```
1: AFF_a ← ∅
2: for all v ∈ V do
3:    visited[v] ← false
4: end for
5: visited[a] ← true
6: Q.Enqueue(a)
7: while Q ≠ ∅ do
8:    v ← Q.Dequeue()
9:    AFF_a ← AFF_a ∪ v
10:   for all u ∈ N(v) s.t !visited[u] do
11:      H ← Find the set of smallest–distance hub of (u, b)
12:      for all h ∈ H do
13:         if h ∈ AFF_a ∨ ((h == u ∨ h == b) ∧ d_{i-1}(u, b) == d_{i-1}(u, a) + 1) then
14:            visited[u] ← true
15:            Q.Enqueue(u)
16:         end if
17:      end for
18:   end for
19: end while
```

$\langle a, b, c\rangle$ has changed from 2 to 6. At the end of this process, the $AFF_b = \{a, b\}$. We run the same process for node c running a forward search rooted at b, and starting at c and we get $AFF_c = \{c, d, e\}$.



a	d	c	e	b

— Order of label construction →

Figure 28: Edge weight increase of edge (b, c) of graph G.

Algorithm 12: AFF_a In directed weighted graph

```
1: AFF_a ← ∅
2: for all v ∈ V do
3:    visited[v] ← false
4: end for
5: Q.Enqueue(a)
6: while Q ≠ ∅ do
7:    v ← Q.Dequeue()
8:    AFF_a ← AFF_a ∪ v
9:    for all u ∈ N_IN(v) s.t !visited[u] do
10:      h ← Find the smallest–distance hub of (u, b)
11:      if h ∈ AFF_a ∨ ((h == u ∨ h == b) ∧ d_{i−1}(u, b) == d_{i−1}(u, a) + l(a, b)) then
12:         visited[u] ← true
13:         Q.Enqueue(u)
14:      end if
15:   end for
16: end while
```

Theorem 7 *[18] Algorithm 11 for the edge (a, b) requires $O(m_{AFF_a}\Lambda log|AFF_a| + m_{AFF_b}\Lambda log|AFF_b|)$ worst-case computational time, where $m(S)$ is the number of edges incident on elements of a set of nodes and Λ is the maximum label size.*

3.2.2.2 Removing Affected Hubs

In decremental updates, it is important to remove the outdated labels because if we ignore such entries then we will underestimate some shortest path distances, and give inaccurate answers. For this reason, removal of such entries can not be avoided. The algorithm that removes these entries, checks all the (v, u) pairs such that $v \in AFF_a$ and $u \in AFF_b$. Algorithm 13 gives the pseudo code of this algorithm in undirected unweighted networks. The modification of this algorithm in directed weighted graphs is as follows: In the case that $O(u) < O(v)$ if $(u, \delta_{uv}) \in l_{OUT}(v)$ then the algorithm removes (u, δ_{uv}, w) entry from $l_{OUT}(v)$. Similarly, for $O(u) > O(v)$ if $(v, \delta_{uv}, w) \in l_{IN}(u)$ then algorithm removes the (v, δ_{uv}, w) entry from $l_{IN}(u)$. Algorithm 14 gives

the pseudo code of this algorithm.

Algorithm 13: RemoveOutdatedHubs(AFF_a, AFF_b) In undirected unweighted graph

```
1: for all (v, u) where (v ∈ AFF_a ∧ u ∈ AFF_b) ∨ (v ∈ AFF_b ∧ u ∈ AFF_a) do
2:     Remove (v, δ_uv) from L(u) if v ∈ L(u)
3: end for
```

Algorithm 14: RemoveOutdatedHubs(AFF_a, AFF_b) In directed weighted graph

```
1: for v ∈ AFF_a do
2:     for u ∈ AFF_b do
3:         if O(v) < O(u) then
4:             Remove (v, δ_vu, w) from L_IN(u) if v ∈ L_IN(u)
5:         else if O(u) < O(v) then
6:             Remove (u, δ_vu, w) from L_OUT(v) if u ∈ L_OUT(v)
7:         end if
8:     end for
9: end for
```

To remove the outdated hubs in the example of Figure 28, we check each pair (u, v) whose starting node u belongs to AFF_b and whose end node v belongs to AFF_c. Recall that $AFF_b = \{a, b\}$ and $AFF_c = \{c, d, e\}$. Consider the pair (a, c) where $a \in AFF_b$ and $c \in AFF_c$. In this case, since a is the smallest distance hub for the pair (a, c) and $O(a) < O(c)$, we need to remove a from the c_{IN} label. When Algorithm 14 finishes, the labeling L of graph G is as shown in Figure 29. In this figure, the entries shown in red are the removed hubs. Notice that the tuple $(a, 2, b)$ has been removed from C_{IN} in Figure 29.

The time complexity to remove the outdated labels in an undirected and unweighted graph is as follows.

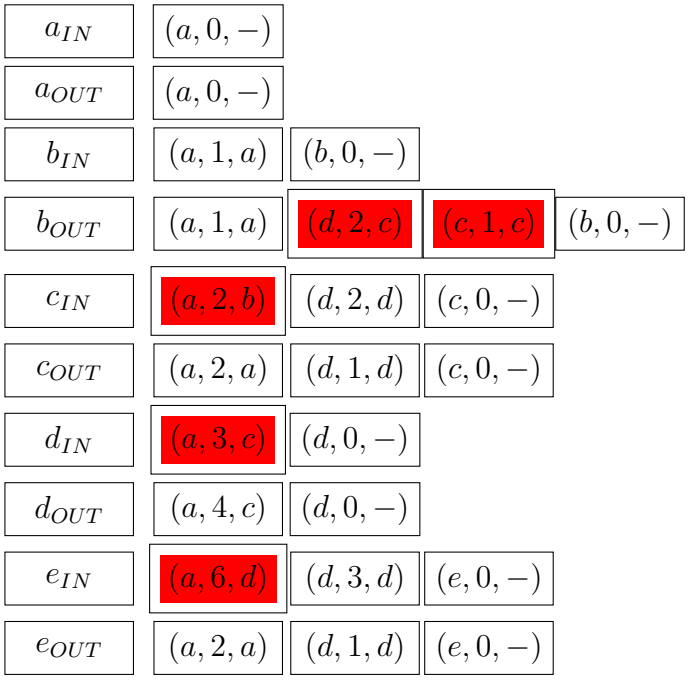


Figure 29: Removed outdated entries from labeling L of graph G after the weight increase of the edge (b, c) shown in red.

Theorem 8 *[18] Algorithm 13 requires* $O(|AFF_a|\Lambda log|AFF_b| + |AFF_b|\Lambda log|AFF_a|)$ *worst-case computational time where* Λ *is the maximum label size.*

3.2.2.3 Computing New Hubs

In this phase, in order for labeling L to be correct, the algorithm needs to add the necessary entries for the pairs whose shortest path is no longer correct due to removing outdated hubs. This phase is similar to label construction that we explained in Section 2.5.1 but limited to the pairs (s, t) such that $s, t \in AFF_{ab}$. For such pairs, the algorithm first checks all the s-t shortest paths. Then it tests if the s-t path is covered or not. If it is covered then it proceeds to the next path and if not, it needs to add the proper label entries.

In [18], the authors describe an algorithm called *Order Restore* for restoring the labeling. We describe it below for directed weighted graphs. Essentially, we perform a *Dijkstra*-like visit rooted at v, for each affected node $v \in AFF_{ab}$. Searches are

performed according to the node ordering. If $v \in AFF_a$, then the direction of search is forward, while if $v \in AFF_b$, then the direction of the search will be backward.

We also change the label entries to a tuple data structure instead of pairs, in order to be able to retrieve the shortest path. Algorithm 16 shows the pseudo code of our implementation. The algorithm can also be used in undirected unweighted networks and the only modification is removing the direction of search, replacing edge weights with 1 and *Dijkstra*-like search with BFS-like search in the network. We omit the pseudo code for this case.

Returning to our example in Figure 29, to add new hubs to the labeling, we need to iterate over all affected nodes AFF_{bc} and for each node, run a *Dijkstra*-like search to compare the tentative distance with the distance resulting from current labeling L and add necessary hubs. In our example, when we scan node $a \in AFF_{bc}$, we check its distance to the nodes b, c, d, e. The tentative distance from a to c resulting from *Dijkstra* search is 3 through the path $\langle a, c \rangle$ and the distance resulting from $Query(a, c, L)$ is ∞, so we will add tuple $(a, 3, a)$ into the backward label of c. At the end of running the Algorithm 16, the labeling is as shown in Figure 30.

a_{IN}	$(a, 0, -)$		
a_{OUT}	$(a, 0, -)$		
b_{IN}	$(a, 1, a)$	$(b, 0, -)$	
b_{OUT}	$(a, 1, a)$	$(b, 0, -)$	
c_{IN}	$(a, 3, a)$	$(d, 2, d)$	$(c, 0, -)$
c_{OUT}	$(a, 2, a)$	$(d, 1, d)$	$(c, 0, -)$
d_{IN}	$(a, 4, a)$	$(d, 0, -)$	
d_{OUT}	$(a, 4, c)$	$(d, 0, -)$	
e_{IN}	$(a, 7, d)$	$(d, 3, d)$	$(e, 0, -)$
e_{OUT}	$(a, 2, a)$	$(d, 1, d)$	$(e, 0, -)$

Figure 30: New hubs added to the labeling L of graph G.

Algorithm 15: ComputeNewLabels() In undiredcted unweighted graph

Sort AFF_{ab} by the node ordering
for all $x \in AFF_{ab}$ in increasing order **do**
  for all $v \in V \backslash \{x\}$ **do**
    $distance[v] \leftarrow \infty$
  end for
  $distance[x] \leftarrow 0$
  $Q.Enqueue(\{(x, distance[x])\})$
  while $Q \neq \emptyset$ **do**
    $(v, distance[v]) \leftarrow Q.Dequeue()$
    if $O(v) < O(x)$ **then**
      Continue
    end if
    if $(x \in AFF_a \wedge v \in AFF_b) \vee (x \in AFF_b \wedge v \in AFF_a)$ **then**
      if $distance[v] < Query(x, v, L)$ **then**
        $L_{(v)} \leftarrow \{(x, distance[v])\}$
      end if
      for all $u \in N(v)$ **do**
        if $distance[u] > distance[v] + l(v, u)$ **then**
          $distance[u] \leftarrow distance[v] + l(v, u)$
          $Q.Enqueue\{(u, distance[u])\}$
        end if
      end for
    end if
  end while
end for

Algorithm 16: ComputeNewLabels() In diredcted weighted graph

Sort AFF_{ab} by the node ordering
for all $x \in AFF_{ab}$ in increasing order **do**
  for all $v \in V \backslash \{x\}$ **do**
    $distance[v] \leftarrow \infty$
  end for
  $distance[x] \leftarrow 0$, $Q.Enqueue(\{(x, distance[x])\})$
  while $Q \neq \emptyset$ **do**
    $(v, distance[v]) \leftarrow Q.Dequeue()$
    if $O(v) < O(x)$ **then**
      Continue
    end if
    if $x \in AFF_a \wedge v \in AFF_b$ **then**
      if $distance[v] < Query(x, v, L)$ **then**
        $L_{IN}(v) \leftarrow \{(x, distance[v])\}$
      end if
      for all $u \in N_{OUT}(v)$ **do**
        if $distance[u] > distance[v] + l(v, u)$ **then**
          $distance[u] \leftarrow distance[v] + l(v, u)$, $Q.Enqueue(\{(u, distance[u])\})$
        end if
      end for
    end if
    if $v \in AFF_a \wedge x \in AFF_b$ **then**
      if $distance[v] < Query(v, x, L)$ **then**
        $L_{OUT}(v) \leftarrow \{(x, distance[v])\}$
      end if
      for all $u \in N_{IN}(v)$ **do**
        if $distance[u] > distance[v] + l(u, v)$ **then**
          $distance[u] \leftarrow distance[v] + l(v, u)$, $Q.Enqueue(\{(u, distance[u])\})$
        end if
      end for
    end if
  end while
end for

Theorem 9 *[18] Algorithm 15 requires* $O(|AFF_{ab}|(m + nlog|AFF_{ab}| + n\Lambda))$ *worst-case computational time.*

3.3 Empirical Studies of Comparison of Dynamic Routing Algorithm

Previous studies have compared the performance of dynamic routing algorithms with each other and with static routing algorithms with respect to customization time and query performance. In [21] *CCH* has been compared with two variants of *CH*([21,27]) and with another metric-independent algorithm called *CRP* [22] to show the efficiency of query in *CCH* for travel time and distance metric. In addition, the effect of three different orders, *Metdep*, the order used in [27], which we refer to as *R-CH*, and two graph partitioning algorithms (viz. Metis, KaHip) on query performance is studied. No attempt to study the dynamic behavior of *CCH*, in terms of running customization after a certain number of edge weight changes or after a certain amount of time.

To reach a better query and preprocessing performance Gottesbüren et al combined the idea of InertialFlow and FlowCutter - two flow-based graph bipartitioning algorithms- and created InertialFlowCutter algorithms [31]. To prove the efficiency of their algorithm, they use *CCHB* from RoutingKit and run the experiment on different setups of these partitioning algorithms on the Colorado, California and Nevada, USA and Europe road networks. In our experiments described in Section 4.2.2, we use the same partitioning algorithms as [31] in combination with *CCH+P*, on the Montreal and Eichstaett road networks.

In the experimental analysis of dynamic *HHL* in [18], they compare the efficiency of running only Decremental, only Incremental, and both Incremental and Decremental update in the the case of weight change with running the label construction from scratch. Their experiments were conducted on many types of networks, such as communication networks, social networks, and one road network in the Netherlands. Moreover, no comparison with other dynamic routing algorithms has been done.

Chapter 4

Experimental Analysis

4.1 Experimental Settings

In this Chapter, we present an extensive experimental analysis of the algorithms introduced and described earlier. We use the setup explained below.

Compiler and machine. All the experiments were run on an Intel i7-8550U CPU @ 1.80GHz, 4 Cores, 8 Logical Processors of a personal computer. The necessary implementation and integration are done in C++.

FrameWork. SUMO [3] is an open-source, traffic simulation package designed to handle large networks implemented in C++. It can import real city maps and generate random traffic. We use version 1.6.0 of SUMO to run our experiments and compare the algorithms.

Instances. We run experiments in two different real city road networks. The map of Eichstaett, a city in Germany which is our smaller instance is available in SUMO website. Our second road network is based on the Montreal downtown map which is extracted from OpenStreetMap(OSM) [7] with [45.45122,-73.6830711,45.5305231,-73.5390472] as the latitude and longitude coordinates.

Figures 31 and 32 show these two road networks. Table 5 describes the properties

of each road network.

Queries. For our experiments, we use a tool in SUMO, called *randomTrips.py* that generates a set of trips/queries, choosing source and destination nodes uniformly at random.

The vehicle arrival rate, ie. the number of steps after which the next trip/query starts, is randomized based on a binomial distribution whose parameters are set to generate the desired number of queries. [1]

Edge Weights. We study two different weight functions for the edges in the network.

- *Travel time*: Moving average is used to calculate the travel time of each edge. The option, *–device.rerouting.adaptation-steps* for period N is used to obtain the average. For static experiments, we use the default value of N that is 180 simulation steps and for dynamic experiments, we set N to 10 simulation steps.
- *Distance*: Another weight function we use for the weight of edges is the geographical distance of corresponding roads in the map.

Performance Measures: We use the following two main performance measures to analyze our experiments.

- *Avg QT* (millisecond): Average running time of shortest distance query
- *Avg QT-WP* (millisecond): Average running time of shortest path query (the result gives the shortest path in addition to shortest distance)

We also use the following performance measures to better analyze the algorithms.

- *Avg TT* (second): Average travel time of trips
- *Avg RL* (meter): Average route length of trips when edge weight is distance
- *CHE*: Number of edges in the contraction hierarchy in *CCH* and *CCH+P*

[1] However as mentioned in SUMO documentation, the actual number of trips may be lower if the road capacity is insufficient to accommodate that number of vehicles.

- *Avg E*: Average explored edges during the distance query in *CCH*. In the elimination tree search, this is the average number of forward and backward edges from ancestors of s to LCA and t to LCA and from LCA to the root.
- *Avg PP* (millisecond): Average preprocessing time. In *CH*, this time is the time required to make contraction hierarchies. In *CCH*, is the time necessary to make a metric-independent contraction hierarchy plus the time required for the initial basic customization. Finally, in *CCH+P* it is the time required to make perfect customization following by a perfect witness search. In *HHL*, this is the time required to make the labels.
- *Avg CT* (millisecond): Average customization time in dynamic experiments.
- *Avg LS*: Average label size for *HHL* algorithm.

Algorithm Implementations. We compared the performance of *Dijkstra*, *Astar*, two variants of *CH*, called *S-CH* and *R-CH*, and *HHL*. *Dijkstra*, *Astar* and *S-CH* are integrated into SUMO, and we used these implementations in our experiments. For *R-CH*, *CCHB*, *CCH+P* and *HHL* that are not implemented in SUMO, we use the code available in their Github repository and integrated them in SUMO. For dynamic *HHL*, the code was not available and we implemented it. For all the graph partitioning algorithms, we use the source code available at [6] and create the different node orderings as separate files and used them in *CCHB* and *CCH+P*.

To ensure the fairness of our comparisons, we make sure that all the implementations are done in C++ and complied with the same compilation flags. No external libraries are used and there is no parallelization in any of the algorithms.

Experiments. We run two classes of experiments. In the *static* experiments, we run the static algorithms given in Sections 2.4.1 and 2.5.1 as well as *Dijkstra* and *Astar*. In these experiments, the preprocessing is done once at the beginning, and queries are answered based on this preprocessing. Edge weights which might change as a result of the queries are not updated during the experiments. For static experiments, we created 1000 random trips/queries for both networks.

In the *dynamic* experiments, we run the algorithms given in Sections 3.1 and 3.2.

Algorithm	Source Code
Dijkstra	SUMO
Astar	SUMO
S-CH	SUMO
R-CH	[58]
CCHB	[58]
CCH+P	[58]
HHL	[51]
All graph partitioning	[6]
Dynamic *HHL*	Our implementation

Table 4: Algorithms used in our experiments.

After an update interval, edge weights are updated. We considered update intervals of 100, 200, 500, and 1000 simulation steps. We ran experiments for 1000, 5000, and 10000 random trips.

The *Avg QT*, *Avg QT-WP* and *Avg CT* for static experiment is the averaged over 10 runs and for dynamic experiment is the average over 3 runs. For *HHL*, we used the code available at [51] For dynamic *HHL*, the code was not available and we implemented it.

Road Network	Nodes	Edges
Montreal	1735	3854
Eichstaett	248	598

Table 5: Benchmark road networks

Figure 31: Montreal downtown road network

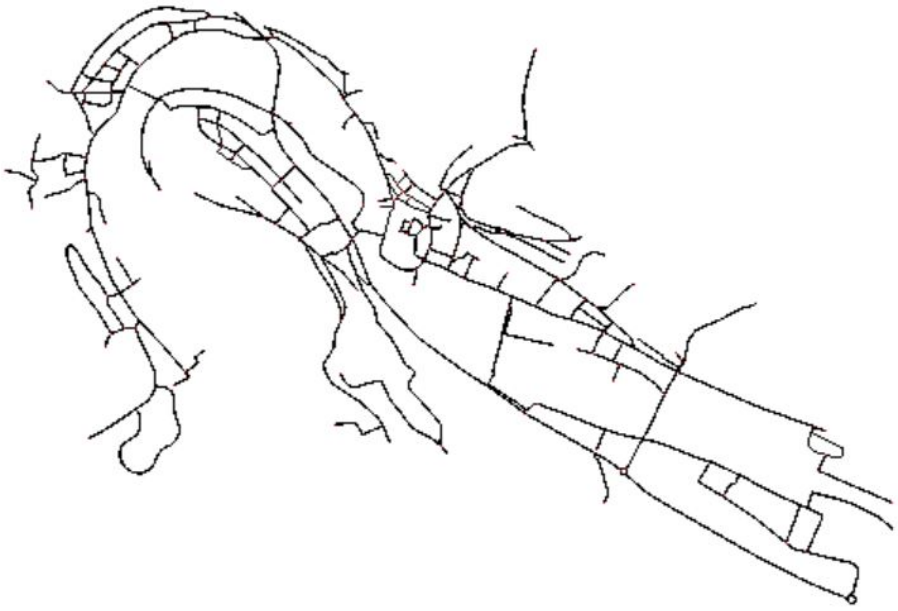


Figure 32: Eichstaett road network

4.2 Analysis of Static Routing Algorithms

In this section, we present the results of our static experiments, *ie.* experiments in which edge weights are not updated during the simulation; all queries are answered based on the initial edge weights. In Sections 4.2.1, 4.2.2, and 4.2.3, we give the results for *CH*, *CCH*, and *HHL* respectively, for different node orderings, in an attempt to find the ordering that gives the best query performance for each algorithm. Finally in Section 4.2.4, we compare the results of these three algorithms. In Tables 6 and 7 below, we give the performance of *Dijkstra* and *Astar* to serve as a benchmark. There is no special setup for *Dijkstra*. For *Astar*, $euclideandistance/\ maximumVehicleSpeed$ is chosen as the heuristic to bound travel time. For distance edge weight it is only the euclidean distance.

Instance	Edge Weight	Measures	Dijkstra	Astar
Montreal	Travel Time	*Avg TT*	477.16	477.16
		Avg E	2719.81	1876.37
		Avg QT	23370.5	**17197.2**
		Avg QT-WP	26600	**21023.5**
	Distance	*Avg RL*	3590.19	3590.19
		Avg QT	**18166.7**	21580.4
		Avg QT-WP	**24097.9**	26663.9

Table 6: Dijkstra and Astar for Montreal downtown road network.

4.2.1 Contraction Hierarchy

Recall that constructing a contraction hierarchy is based on contracting nodes in a certain order. The strategy of node ordering has a direct impact on the efficiency of *CH*. Although optimal node ordering is an NP-hard problem [14], there are heuristics that work well in practice. In this section, we describe two ordering heuristics and then compare their effect on query performance in *CH*.

Instance	Edge Weight	Measures	Dijkstra	Astar
Eichstaett	Travel Time	*Avg TT*	284.43	284.43
		Avg E	418.25	319.19
		Avg QT	1929.9	1640.1
		Avg QT-WP	1964.2	1535.6
	Distance	*Avg RL*	2049.70	2049.70
		Avg E	422.81	415.92
		Avg QT	3049.2	3422.2
		Avg QT-WP	3084.9	3622.3

Table 7: Dijkstra and Astar for Eichstaett road network.

4.2.1.1 Two Different Node Ordering Heuristics

The first ordering is based on a node ordering introduced in [8]. We denote it by *S-CH*. The priority of a node to contract, is the linear combination of several terms. It is an on-line heuristic that is used to select which node to shortcut next. The equation below, shows the priority heuristic used to contract nodes. $P(u)$ is the priority.

$$P(u) = 2 * ED(u) - CN(u) - H(u) - 5 * L(u)$$

Where $CN(u)$ is the number of previously contracted neighbors of u; $H(u)$ is the number of edges represented by the shortcuts added; $L(u)$ is the level of u that is defined as $L(u) = L(v) + 1$ where v is the highest level node among all lower ranked neighbors of u in G' (if there is no such v, then $L(u) = 0$); and $ED(u)$ is calculated as below:

$$ED(u) = Incoming(u) + Outgoing(u) - 2 * Shortcuts$$

It should be mentioned that formula above is the one implemented in SUMO, but is slightly different from the original formula in [8]. *S-CH* uses lazy update to keep the priority queue updated. The witness search stopping criteria is the settled nodes limit which is implemented by creating shortest path trees with depth 4.

The second heuristic, we denote by *R-CH*, and has a different way to define the priority of each node for contraction. It uses the formula below:

$$I(u) = L(u) + \frac{\mid A(u) \mid}{\mid D(u) \mid} + \frac{\sum_{a \in A(u)} h(a)}{\sum_{a \in D(u)} h(a)}$$

[21]. In this expression, $I(u)$ is the priority or importance of u; $A(u)$ is the number of edges that are inserted when we contract node u; $D(u)$ is the number of edges removed after contracting node u; $h(a)$ is the number of edges that shortcut a represents if fully unpacked and $L(u)$ is the approximate level of u. Initially, the level of all nodes is zero. If a node x is contracted, then the level of every incident node y, is the maximum of level y or $L(x) + 1$. This setup dose not use lazy update or periodic queue rebuilding. It updates the priority of the neighbors of node being contracted and uses the limited hop search as the stopping criteria which is set to $p = 1500$

4.2.1.2 CH Performance Analysis

As shown in Tables 8 and 9 for both travel time and distance as edge weights, and regardless of the size of the network, *R-CH* has better shortest distance query time and shortest path query time and explores fewer edges than *S-CH*. In addition, *R-CH* has approximately 10 orders of magnitude faster preprocessing time which is a result of ordering heuristics and witness search stopping criteria that we already discussed. We also observed that the size of the backward search and forward search queues in *S-CH*, is approximately 2.5 times bigger than *R-CH* for both networks. For example for the Eichstaett road network, there are 430 edges in forward search for *R-CH* and 1094 Edges for *S-CH*.

Another interesting observation is the sensitivity of *CH* to the edge weight used: for distance edge weight, the preprocessing time is slower compared to travel time (This is more evident in Table 8, which is a larger road network). Regarding the quality of the route, *S-CH*, appears to have a slightly better quality of the route, when using travel time as edge weight.

Instance	Edge Weight	Measures	CH	
			R-CH	*S-CH*
Montreal	Travel Time	*Avg PP*	3419.5	39006.8
		Avg TT	479.77	477.16
		Avg E	23.28	182.5
		Avg QT	**355**	3170.5
		Avg QT-WP	**470.1**	3361.2
	Distance	*Avg PP*	5073.9	40452.1
		Avg RL	3590.19	3590.19
		Avg E	25.39	173.5
		Avg QT	**437.9**	2834.6
		Avg QT-WP	**535.9**	3458.2

Table 8: Contraction Hierarchies performance for 1000 queries for two different node order for Montreal downtown road network

Instance	Edge Weight	Measures	CH	
			R-CH	*S-CH*
Eichstaett	Travel Time	*Avg PP*	262.4	2836.2
		Avg TT	293.96	294.17
		Avg E	8.36	51.34
		Avg QT	**88.1**	753.7
		Avg QT-WP	**154.4**	929.2
	Distance	*Avg PP*	265.6	2863.3
		Avg RL	2182.44	2182.44
		Avg E	8.1	50.44
		Avg QT	**90**	727.5
		Avg QT-WP	**168.2**	939.6

Table 9: Contraction Hierarchies performance for two different node order for Eichstaett road network.

4.2.2 Customizable Contraction Hierarchies

In this section, we compare the performance measures in *CCH* with respect to different node ordering. We study two different variants of *CCH*. In the first version, which we call it *CCHB*, elimination tree search for queries, and ***b**asic* customization with an upper triangle instead of the lower triangle has been used. The second variant, which we denote *CCH+P*, uses ***p**erfect* customization. In the following sub-sections,

we measure the query performance and preprocessing time of *CCHB* and *CCH+P* according to the graph partitioning algorithms described in Section 3.1.1.1.

4.2.2.1 *CCHB* Performance Analysis

Tables 10 and 11 show the query performance as well as preprocessing time for travel time and distance as edge weights for *CCHB* corresponding to all graph partitioning algorithms. As we see, a graph partitioning algorithm that creates a smaller chordal graph (smaller *CHE*) has smaller search space in elimination tree. Thus, a partitioner which leads to sparser chordal graph (fewer nodes and edges in the search space), always has the fastest query. For example, in Montreal downtown network, we observed that the average number of edges in the search space is 481 for *FC20* which has the fastest query time and 27431 for *IF* which has the slowest query times respectively. In this road network, for travel time as edge weight, *FC20* outperforms the other partitioners for both shortest distance query and shortest path query. For distance as edge weight, however, *FC100* performs better for distance query time. Both *FC20* and *FC100* have the closest trip duration to *Dijkstra* and *Astar* as shown in Table 6. In this network, after all variants of FlowCutter, next best orders are *Metis* and *K2.11* for both edge weights.

In Table 11 we see the same pattern as Montreal road network for Eichstaett for relation between the *CHE*, average number of edges in search space and other performance measures. In this network, *IFR* outperforms other graph partitioning algorithms for both shortest distance query and shortest path query. It also has the closest trip duration to *Dijkstra* and *Astar* as shown in Table 7. InertialFlowCutters are better configurations after *IFR* in Eichstaett network.

Observing Tables 10 and 11, all variants of InertialFlowCutter (IFC) are substantially better than all variants of InertialFlow (*IF* and *IFR*). For Montreal downtown, all variants of InertialFlowCutter, fall behind FlowCutter, *Metis* and *K2.11*. This is correct for Eichstaett as well, except *IFC4* which is marginally better than *K2.11*.

Instance	Edge Weight	Measures	*CCHB*										
			FC3	*FC20*	*FC100*	*IFC16*	*IFC12*	*IFC8*	*IFC4*	*IF*	*IFR*	*Metis*	*K2.11*
Montreal	Travel Time	*CHE*	8697	8429	8480	16035	15689	16263	17218	52037	54245	9243	10467
		Avg PP	343.1	337.7	352.2	468.8	456.5	471.8	451.5	940.9	1067.0	359.4	379.2
		Avg TT	476.88	477.08	477.16	479.01	479.64	479.56	479.01	479.36	480.5	476.91	478.17
		Avg E	38.55	38.09	37.9	118.59	118.4	119.6	123.05	267.63	282.46	42.56	44.77
		Avg QT	343.4	**328.1**	332.6	3191.5	3121.2	3178.6	3231.5	15243.5	13819.3	419.3	477.9
		Avg QT-WP	607.6	**575.7**	592.2	3345.4	3506.9	3506.5	3673.1	16724.4	14380.5	621.3	721.3
	Distance	*CHE*	8697	8429	8480	16035	15689	16263	17218	52037	54245	9243	10467
		Avg PP	333.0	325.4	318.1	442.6	416.1	427.1	439.8	902.8	1013.3	358.1	375.1
		Avg RL	3601.19	3601.19	3601.19	3601.19	3601.19	3601.19	3601.19	3601.19	3601.2	3601.19	3601.19
		Avg E	38.55	38.07	37.92	118.55	118.41	119.6	123.01	267.59	282.56	42.56	44.77
		Avg QT	321.9	328.6	**308.7**	2981.0	2863.0	2923.7	3009.4	14324.3	13784.8	405.8	453.7
		Avg QT-WP	**622.7**	638.9	624.8	3555.3	3485.0	3526.0	3714.9	16960.6	14640.4	669.5	803.3

Table 10: *CCHB* performance for different node ordering, averaged over 1000 queries for Montreal downtown road network.

Instance	Metric	Measures	*CCHB*										
			FC3	*FC20*	*FC100*	*IFC16*	*IFC12*	*IFC8*	*IFC4*	*IF*	*IFR*	*Metis*	*K2.11*
Eichstaett	Travel Time	*CHE*	1878	1748	1748	1645	1652	1645	1680	2241	994	1801	1612
		Avg PP	102.8	111.7	99.7	105.3	104.8	99.9	105.6	109.1	95.6	101.1	100.1
		Avg TT	281.51	282.97	282.97	286.33	289.55	286.33	286.33	295.68	291.57	281.36	281.31
		Avg E	43.84	38.26	38.26	37.24	37.81	37.24	37.74	53.07	24.86	38.74	38.06
		Avg QT	444.3	377.0	381.2	369.5	366.2	342.9	341.5	619.7	**158.5**	378.0	654.4
		Avg QT-WP	596.9	513.2	530.6	460.3	464.0	467.1	365.9	754.0	**258.7**	453.6	423.4
	Distance	*CHE*	1878	1748	1748	1645	1652	1645	1680	2241	994	1801	1612
		Avg PP	95.9	96.9	96.0	93.6	94.6	96.3	90.4	104.3	88.9	98.4	90.7
		Avg RL	2182.44	2182.44	2182.44	2182.44	2182.44	2182.44	2182.44	2182.44	2182.44	2182.44	2182.44
		Avg E	43.57	37.95	37.95	37.0	37.54	37.0	37.5	53.08	24.81	38.52	38.31
		Avg QT	438.7	357.6	359.6	325.0	326.5	317.7	322.3	585.7	**178.4**	351.3	651.9
		Avg QT-WP	628.1	553.5	568.2	497.8	477.9	476.5	495.1	793.9	**282.9**	505.1	440.6

Table 11: *CCHB* performance for different node ordering, averaged over 1000 queries for Eichstaett road network.

4.2.2.2 *CCH* with Perfect Witness Search Performance Analysis

In this section, we study the performance of *CCH+P*. For the query, we use a stall-on-demand type of bidirectional search.

As we see in Tables 12 and 13, for the Montreal downtown network, *FC20* is the

preferred configuration for both *CCH* and *CCH+P* considering distance query time and shortest path query time (Difference between *FC20* and *FC3* in distance query time for travel time edge weight is negligible). But for Eichstaett, *Metis* is the best one for *CCH+P* while *IFR* is the best algorithm for *CCH*. From Tables 12 and 13 we see that the relative performance of graph partitioning algorithms is different when used with *CCHB* and *CCH+P*.

Instance	Edge Weight	Measures	*CCH+P*										
			FC3	*FC20*	*FC100*	*IFC16*	*IFC12*	*IFC8*	*IFC4*	*IF*	*IFR*	*Metis*	*K2.11*
Montreal	Travel Time	*CHE*	5302	5165	5165	5680	5635	5622	5707	7466	7557	5456	5712
		Avg PP	572.7	539.6	560.4	1272.2	1259.5	1309.6	1404.4	8701.0	7725.9	544.4	635.4
		Avg TT	481.75	475.15	478.49	478.95	478.71	481.32	479.46	478.26	477.49	476.22	477.23
		Avg E	28.13	27.32	27.71	48.31	48.45	49.12	48.77	75.66	65.52	29.7	31.4
		Avg QT	**365.8**	366.1	398.9	815.5	788.7	787.2	769.8	1420.0	1058.2	422.0	450.2
		Avg QT-WP	443.7	**442.1**	457.4	741.7	754.4	798.3	906.6	1433.9	1037.2	444.1	462.7
	Distance	*CHE*	5302	5165	5165	5680	5635	5622	5707	7466	7557	5456	5712
		Avg PP	652.7	626.4	652.4	1456.6	1569.6	1484.5	1607.8	9270.4	9161.1	674.3	722.3
		Avg RL	3601.19	3601.13	3601.19	3601.19	3601.2	3601.19	3601.2	3601.19	3601.19	3601.19	3601.13
		Avg E	28.25	27.38	27.81	47.77	48.18	48.61	48.32	77.04	68.01	29.7	31.57
		Avg QT	446.8	**429.5**	488.0	943.1	901.7	847.8	873.9	1501.2	1345.8	502.1	522.6
		Avg QT-WP	520.8	**492.1**	539.9	941.5	921.9	963.5	982.9	1645.6	1390.5	590.1	597.5

Table 12: *CCH+P* performance with perfect witness search for different graph partitioning for Montreal downtown road network.

Instance	Edge Weight	Measures	*CCH+P*										
			FC3	*FC20*	*FC100*	*IFC16*	*IFC12*	*IFC8*	*IFC4*	*IF*	*IFR*	*Metis*	*K2.11*
Eichstaett	Travel Time	*CHE*	951	1022	1022	918	899	918	916	888	665	819	1069
		Avg PP	161.2	150.0	165.0	171.3	160.4	149.3	144.7	198.1	141.4	145.1	225.7
		Avg TT	288.59	284.17	284.17	286.67	286.67	286.67	286.67	290.73	290.48	292.42	296.54
		Avg E	16.91	18.85	18.85	16.74	16.51	16.74	16.72	15.39	14.3	12.36	20.32
		Avg QT	238.8	275.9	286.8	209.2	189.2	193	203.7	164.5	151.3	**129.9**	374.2
		Avg QT-WP	333.5	389.8	384.4	282.9	266.9	280.3	282	241.5	224.7	**150.9**	462.7
	Distance	*CHE*	951	1022	1022	918	899	918	916	888	665	819	1069
		Avg PP	174.1	189.1	199.6	190.8	200.4	205.4	187.5	225.6	143.8	185.1	258.6
		Avg RL	2182.44	2182.44	2182.44	2182.44	2182.44	2182.44	2182.44	2182.44	2182.44	2182.44	2182.44
		Avg E	17.45	19.51	19.51	18.17	17.75	18.17	18.14	16.6	14.72	12.83	20.46
		Avg QT	234.8	287.5	265.9	220.9	196.8	213.9	207	177.9	137	**125.8**	322.7
		Avg QT-WP	321.2	373.9	385.1	344.6	301.7	311	309.1	267.5	**182.9**	215.6	489.1

Table 13: *CCH+P* performance with perfect witness search for different graph partitioning for Eichstaett road network.

4.2.2.3 *CCHB* or *CCH+P* ?

In Tables 14 and 15, we compare the best variant of *CCHB* and *CCH+P* with respect to shortest path query for both travel time and distance edge weight. We see that *CCH+P* leads to a significantly better query performance than *CCHB* for all graph partitioning algorithms, regardless of edge weight (travel time or distance) and the road network used (Montreal or Eichstaett). However this is at the cost of higher preprocessing time compared to *CCHB*. Therefore, depending on the application and the specific trade-off between preprocessing and query time, either of them could be the better algorithm. Since *CCH+P* creates a sparser contraction hierarchy than *CCHB*, the shortest path query is faster in *CCH+P* than *CCHB*.

Going forward in this thesis, assuming that the query time is the important metric to optimize, we will only use *CCH+P* when comparing with other algorithms.

Instance	Edge Weight	Measures	*CCHB*	*CCH+P*
Montreal	Travel Time	*CHE*	8429	5165
		Avg PP	337.7	539.6
		Avg TT	477.08	475.15
		Avg E	38.09	27.32
		Avg QT	328.1	366.1
		Avg QT-WP	575.7	**442.1**
	Distance	*CHE*	8697	5165
		Avg PP	333	626.4
		Avg RL	3601.19	3601.19
		Avg E	38.55	27.38
		Avg QT	321.9	429.5
		Avg QT-WP	622.7	**492.1**

Table 14: Comparison of *CCHB* and *CCH+P* for Montreal downtown road network.

4.2.3 Hierarchical Hub Labeling

Since finding a good node ordering to construct effective labels in *HHL* is challenging, we would like to study the effect of different node orderings on label construction in

Instance	Edge Weight	Measures	*CCHB*	*CCH+P*
Eichstaett	Travel Time	*CHE*	994	819
		Avg PP	95.06	145.1
		Avg TT	291.57	292.42
		Avg E	24.86	12.36
		Avg QT	158.5	129.9
		Avg QT-WP	258.7	**150.9**
	Distance	*CHE*	994	665
		Avg PP	88.9	143.8
		Avg RL	2182.44	2182.44
		Avg E	24.81	14.72
		Avg QT	178.4	137
		Avg QT-WP	282.9	**182.9**

Table 15: Comparison of *CCHB* and *CCH+P* for Eichstaett road network.

road networks. For our experiments, we use the label construction paradigm that was proposed by Akiba (as explained in Section 2.5.1).

Previous studies use different node ordering approaches, such as the degree of nodes and betweenness of nodes. In our experiments, we use *Degree*, two versions of betweenness, *BTW-PG* and *BTW-LG* as node ordering techniques. We are also interested to see how ordering based on graph partitioning works, compared to these approaches. The reason for this selection is the fast order construction of graph partitioning compared to variants of betweenness. For *Degree* as node ordering, we use the product of the number of outgoing edges and incoming edges instead of their sum since it results in better performance in practice [37,63]. Since label construction using both variants of betweenness, is very slow, so in our experiments, we only use the order extracted from this approach and give to Akiba algorithm as a file of ordered nodes. By looking at the experiments done for *CCH+P*, we use the graph partitioning algorithms which have the best performance. Thus, we use *FC20* for the Montreal downtown network and *Metis* for Eichstaett. We denote them by *GP-FC20* and *GP-Metis*.

4.2.3.1 *HHL* Performance Analysis

As we see in Tables 16 and 17, regardless of the label size, the best configuration is *BTW-PG* for the query time. *BTW-LG* and *Degree* are second and third best options and the *GP-FC20* has the slowest query time. This result holds for all the edge weight/network size combinations. The quality of the route doesn't change for different node ordering strategies. *Degree* doesn't create a good ordering compared to betweenness, since in the road networks, the degree of nodes is small. *GP-FC20* has the slowest query time and it has the slowest preprocessing time after *Degree*. Betweenness strategies have the lowest preprocessing time.

The time to create the ordering is not included as part of the preprocessing time. We note that in our experiments, for the Montreal downtown network, *Degree* took 1ms, *GP-FC20*, 191ms, *BTW-LG*, 295729ms and *BTW-PG*, 320833ms, to create the ordering. The same relative time to create the ordering holds for Eichstaett as well.

Instance	Edge Weight	Measures	*HHL*			
			Degree	*BTW-LG*	*BTW-PG*	*GP-FC20*
Montreal	Travel Time	*Avg PP*	10314.2	5888.3	5031.6	37659.2
		Avg TT	476.96	476.96	476.96	476.96
		Avg LS	36.34	34.72	39.89	71.64
		Avg QT	100.7	92.4	**82**	182.1
		Avg QT-WP	1030.6	1139	**1027.3**	3979
	Distance	*Avg PP*	10765.5	7506.8	4947.3	37564.3
		Avg RL	3601.19	3601.19	3601.19	3601.19
		Avg LS	38.53	35.74	43.49	71.32
		Avg QT	100.6	102.1	**81.8**	181.6
		Avg QT-WP	1085	1262.9	**1006.1**	4010.2

Table 16: Hub Labeling performance with different node ordering for Montreal downtown road network.

Instance	Metric	Measures	*HHL*			
			Degree	*BTW-LG*	*BTW-PG*	*Metis*
Eichstaett	Travel Time	*Avg PP*	264.1	94.4	87.7	586
		Avg TT	294.17	294.17	294.17	294.17
		Avg LS	13.51	10.70	11.32	32.54
		Avg QT	34.9	33.2	**31.1**	37.5
		Avg QT-WP	367.7	341.5	**337.5**	637.2
	Distance	*Avg PP*	251.3	120.2	111.7	575.6
		Avg RL	2182.44	2182.44	2182.44	2182.44
		Avg LS	13.06	12.51	12.84	32.29
		Avg QT	34.4	35.5	**30.9**	35.8
		Avg QT-WP	378.1	379.1	**371.4**	631.6

Table 17: Hub Labeling performance with different node ordering for Eichstaett road network.

4.2.4 Comparison with Other Routing Algorithms

In this section, we compare the speed-up algorithms (*CH*, *CCH+P* and *HHL*) with each other. In Tables 18 and 19 we provided the results of the best version of each algorithm in order to compare them.

Regarding the shortest distance query, the best algorithm in our road networks is *HHL*. This is because the labels already represent shortest path information, and the algorithm only needs to perform merge-join-like operations to answer distance queries, without needing to explore the graph. However, the preprocessing time to construct the labels is much higher than the preprocessing time for the other algorithms for the Montreal network. The preprocessing time for all algorithms is comparable in the Eichstaett network. *CCH+P* and *R-CH* stands as the second and third best algorithm for this type of query. Note that the preprocessing time is higher for *CH* than *CCH+P*, but query time of *CH* is generally better or comparable to that of *CCH+P*.

Next we discuss the shortest path query. Although *HHL* is efficient for shortest distance query, we see that it is less efficient than *CH* and *CCH+P* in returning the actual shortest path, which is likely due to its recursive implementation. Note that

HHL is still a better option than *Dijkstra* and *Astar* in terms of query performance. *CCH+P* can be seen to be better that *R-CH* for shortest path query performance for both edge weights for the Montreal network, and for the travel time edge weight in the Eichstaett network.

We noted already that we can answer the shortest path query as fast as the shortest distance query in *CCH+P*; this is not the case for *CH* or for *HHL*. In fact for *HHL* as already noted, the time to answer the shortest path query is much worse than the time to answer the shortest distance query.

Comparing the preprocessing time of algorithms in the Montreal downtown network and Eichstaett network shows that label construction time is very sensitive to the size of the network. In Eichstaett, *HHL* has the fastest preprocessing time compared to other speed-up algorithms. In Montreal downtown network, *HHL* has the slowest preprocessing time among other speed-up algorithms. Our experiments show that contraction hierarchy construction can be done faster in *CCH+P* which has perfect customization followed by a perfect witness search compared to *CH*. Thus *Avg PP* is faster in *CCH+P* than *CH*; this difference will even more if we consider *CCHB* which has faster preprocessing time than *CCH+P*. In addition, *CCH+P* is not very sensitive to the edge weight which is in contrast to *CH*.

For both networks, considering the overall performance (query time and preprocessing time) of each algorithm, it is obvious that speed-up algorithms overtake *Dijkstra* and *Astar*.

Instance	Edge Weight	Measures	*CCH+P* (*FC20*)	*CH* *R-CH*	*HHL* *BTW-PG*	*Dijkstra*	*Astar*
Montreal	Travel Time	*Avg PP*	539.6	3419.5	5031.6	-	-
		Avg TT	475.15	479.7	476.96	477.16	477.16
		Avg E	27.32	23.28	-	2719.81	1879.37
		Avg QT	366.1	355	**82**	23370.5	17197.2
		Avg QT-WP	**442.1**	470.1	1027.3	26600	21023.5
	Distance	*Avg PP*	508	5073.9	4947.3	-	-
		Avg RL	3590.19	3590.19	3590.19	3590.19	3590.19
		Avg E	27.38	25.39	-	2717.23	2654.56
		Avg QT	429.5	437.9	**81.8**	18166.7	21580.4
		Avg QT-WP	**492.1**	535.9	1006.1	24097.9	26663.9

Table 18: Comparing static routing algorithms for 1000 queries for Montreal downtown road network

Instance	Edge Weight	Measures	*CCH+P* (*Metis*)	*CH* *R-CH*	*HHL* *BTW-PG*	*Dijkstra*	*Astar*
Eichstaett	Travel Time	*Avg PP*	145.1	262.4	87.7	-	-
		Avg TT	281.36	293.96	294.17	294.17	294.17
		Avg E	12.36	8.36	-	434.31	332.8
		Avg QT	129.9	88.1	**31.1**	3534.2	2984
		Avg QT-WP	**150.9**	158.2	337.5	4588.5	3840
	Distance	*Avg PP*	110.2	265.6	111.7	-	-
		Avg RL	2182.44	2182.44	2182.44	2182.44	2182.44
		Avg E	12.83	8.1	-	436.3	428.08
		Avg QT	125.8	90	**30.9**	2961.8	3304.8
		Avg QT-WP	215.6	**162.1**	371.4	3801.1	4347.8

Table 19: Comparing static routing algorithms for 1000 queries for Eichstaett road network

4.3 Dynamic Routing Algorithms

In this section, we compare the performance of dynamic speed-up algorithms in road networks in which the edge weights often change. In this case, new queries (vehicles) and even those who are already running in the network, might need to reroute in order to find the quickest path. Table 20 shows the improvement in travel time that can be achieved by updating the edge weights after different time intervals for both Montreal downtown and Eichstaett networks. Figure 33 shows that for the Montreal downtown network, up to 35 % improvement in travel time can be obtained by updating the edge weight after every 100 simulation steps. Similarly for the Eichstaett network, up to 21 % improvement in travel time can be obtained by updating the edge weight after every 100 simulation steps.

Interval	*CCH+P*
100	1672.31
200	1867.19
500	2394.20
1000	2509.37
No update interval	2509.37

Interval	*CCH+P*
100	4850.84
200	4733.02
500	4937.03
1000	5444.56
No update interval	5962.07

Table 20: Travel time changes for different interval for 10000 queries for Montreal downtown network (Left) and Eischtaett (Right) for *CCH+P* algorithm.

In SUMO we use parameter *–device.rerouting.period* to set the period after which vehicles might be rerouted. We set the value of this parameter to be equal to the value of *–device.rerouting.adaptation-interval* which is the time interval after which the edge weights of the network are updated.

To respond to the edge weight changes, our static speed-up algorithms, such as *CH* and *HHL* would have to repeat the preprocessing stage, which incurs a large extra cost. Since it would be too expensive to run the preprocessing after every update interval, we instead estimate it by multiplying the preprocessing cost with the number of customization runs and denote it by *Est.PP*. In Tables 21 and 22, we show the total customization cost and estimated preprocessing cost for the Montreal

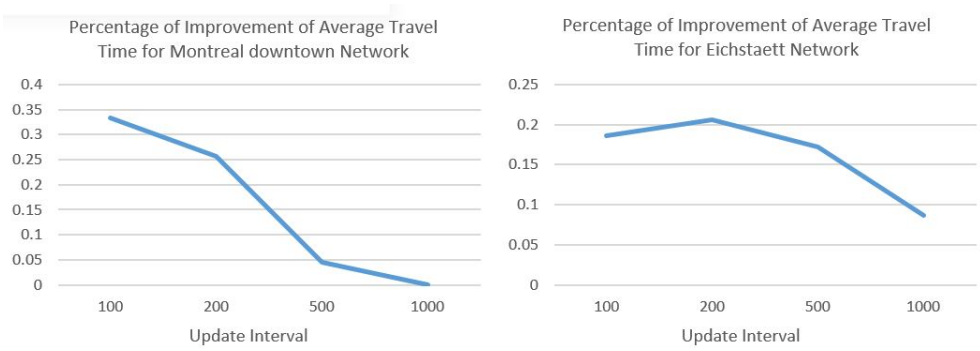


Figure 33: Improvement of *Avg TT* with respect to different update interval for 10000 queries using *CCH+P* algorithm for Montreal downtown network (Left) and Eichstaett (Right).

downtown and Eichstaett networks respectively for different combinations of number of queries and update intervals.

In Figures 34 and 35, we show the total customization cost as a fraction of the total estimated preprocessing cost for the Montreal and Eichstaett networks. We see that in general, for smaller number of queries (1000 queries), and smaller update intervals, the customization cost is a smaller fraction of the estimated preprocessing cost. For 5000 queries, the customization cost stays the same fraction of the estimated preprocessing cost, and for larger number of queries, the ratio of the customization cost to the preprocessing cost decreases with the length of the update interval.

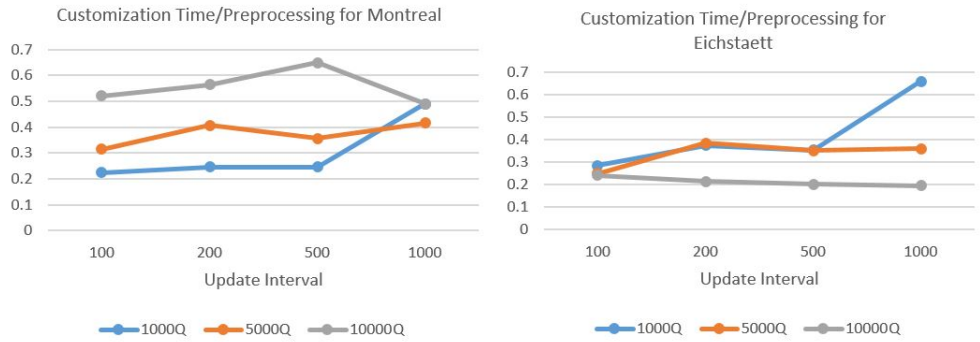


Figure 34: *Avg CT* as a percentage of *Est.PP* with respect to different update interval for different number of queries using *CCH+P* algorithm for Montreal downtown network (Left) and Eichstaett (Right).

Interval	Measures	1000Q		5000Q		10000Q	
		CCH+P	*HHL*	*CCH+P*	*HHL*	*CCH+P*	*HHL*
100	*Avg CT*	2791	133519856	4577	-	11800.6	-
	Est.PP	12410.8	115726.8	14569.2	-	22663.2	-
200	*Avg CT*	1596	52274734	3509	-	7607	-
	Est.PP	6475.2	60379.2	8633.6	-	13490	-
500	*Avg CT*	532	27001860.5	1731	-	3855	-
	Est.PP	2158.4	20126.4	4856.4	-	5935.6	-
1000	*Avg CT*	264	17857173	1124	-	2114	-
	Est.PP	539.6	5031.6	2698	-	4316.8	-

Table 21: Comparison of *Avg CT* and *Est.PP* after each edge update interval for different number of queries for Montreal downtown network.

Interval	Measures	1000Q		5000Q		10000Q	
		CCH+P	*HHL*	*CCH+P*	*HHL*	*CCH+P*	*HHL*
100	*Avg CT*	582.3	650891	4476	5371200	8672.6	13684369
	Est.PP	2031.4	1227.8	17847.3	10787.1	35694.6	21574.2
200	*Avg CT*	382.3	439089	2636.6	3968182	4518	8177463.6
	Est.PP	1015.7	613.9	6819.7	4121.9	21039.5	12716.5
500	*Avg CT*	154.3	173783	1229	1731928.3	2145.3	3724850.3
	Est.PP	435.3	263.1	3482.4	2104.8	10592.3	6402.1
1000	*Avg CT*	96	86075	680	865964.15	936	1582816.3
	Est.PP	145.1	87.7	1886.3	1140.1	4788.3	2894.1

Table 22: Comparison of *Avg CT* and *Est.PP* after each edge update interval for different number of queries for Eichstaett network.

To conclude our remarks above, dynamic speed-up algorithms achieve better travel times than static speed-up algorithms if the latter only perform pre-processing once or very rarely. On the other hand, a dynamic speed-up algorithm like *CCH* can be much more efficient at answering queries than a static speed-up algorithm that repeats the preprocessing step often.

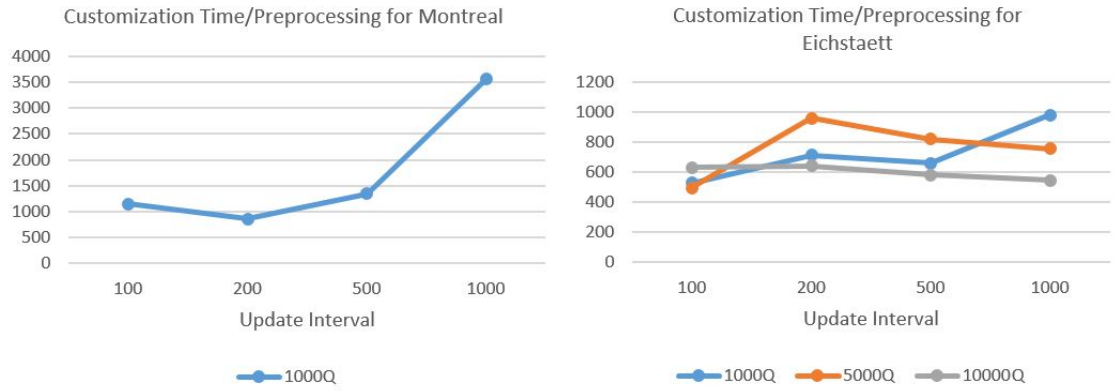


Figure 35: *Avg CT* as a percentage of *Est.PP* with respect to different update interval for different number of queries (1000 queries for Montreal and 1000, 5000 and 10000 queries) using dynamic *HHL* algorithm for Montreal downtown network (Left) and Eichstaett (Right).

In this section, we therefore only consider the dynamic algorithms - *CCH* and dynamic *HHL* - since we are able to customize them to respond to edge weight changes without performing the entire preprocessing phase. We use the best version of these algorithms with respect to the query time for this experiment. Thus, for the Montreal downtown network, we use *CCH+P* with *FC20* and *CCH+P* with *Metis* for Eichstaett; For *CCH+P* after every edge weight change operation, we run only the perfect customization phase. We run dynamic *HHL* with *BTW-PG* for both networks; after each weight change update, we scan all the edges whose weight has changed since the last update. If the weight of an edge increased, we run Decremental update for that edge and if it decreased we run Incremental update for the edge. We explained these functions in detail in Sections 3.2.1 and 3.2.2.

Since travel time is an edge weight whose value may change over time, we run dynamic experiments considering travel time change in our road networks. We run the experiment in three query sizes (1000, 5000, and 10000 queries) and four weight change intervals (100, 200, 500, 1000 simulation steps). When the edge weights are updated, we also run the customization phase of both algorithms. We are interested in investigating which algorithm is the most efficient one to use for dynamic scenarios with respect to query time and customization time.

4.3.1 Effect of Update Interval on Customization Time

As we see in Tables 23 and 24, *CCH+P* is very efficient to respond to weight updates in road networks compared to dynamic *HHL*. Its light customization phase makes it very efficient to answer dynamic conditions without running a heavy preprocessing phase. This holds regardless of the weight update frequency and the number of queries. For instance, for the Montreal downtown network in 1000 queries experiment, for 100 simulation steps as update interval, the customization runs about 23 times which takes 2791ms in all on average, and 121.3ms per customization run on average. In contrast, the preprocessing run which consists of building the metric-independent contraction hierarchy followed by perfect customization takes 539.6ms.

For Dynamic *HHL* however, our experiments show that for more frequent weight updates, we get very slow customization time to the point that it is impractical for large networks, and for large numbers of queries. In fact, we had to stop the simulation for the Montreal downtown network for 5000 and 10000 queries after some hours of simulation. In this situation, it may be beneficial to reconstruct the labels for the entire network from scratch, as the whole preprocessing time would be less than the customization time. This is also true for the Eichstaett, the smaller network (See Tables 21 and 22).

We conclude that *CCH+P* is much more efficient than dynamic *HHL* in responding to weight updates in road networks compared to dynamic *HHL*. This holds for both networks, regardless of the number of queries and the length of the update interval. In fact, for large networks and large number of queries (i.e. heavy traffic), dynamic *HHL* proves to be impractical. Figures 36 and 37 which has log-scaled y-axis illustrate that for both networks and for all the number of queries, customization time decreases when the time interval for edge weight update increases.

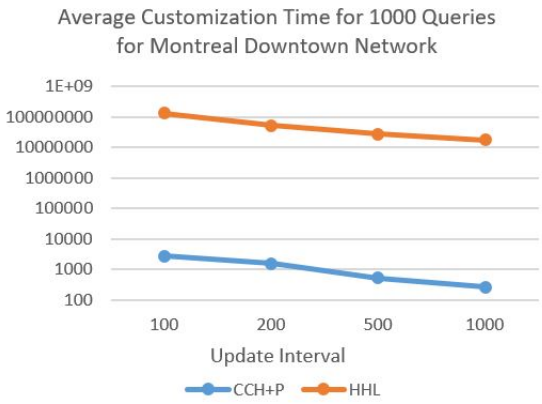


Figure 36: *Avg CT* (ms) for different weight update intervals for 1000 queries for Montreal Downtown road network. Numbers are the log-scaled.

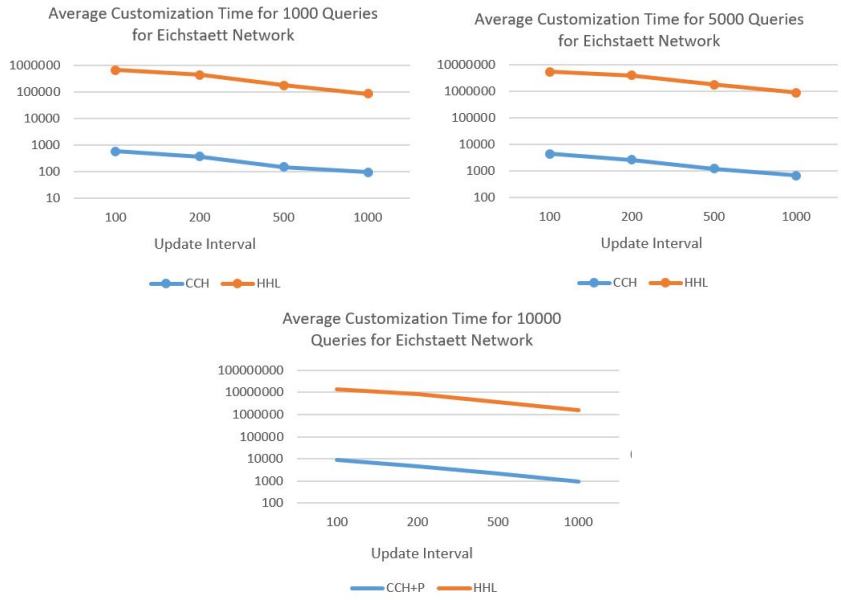


Figure 37: *Avg CT* (ms) for different weight update intervals for different number of queries for Eichstaett road network. Numbers are the log-scaled.

4.3.2 Effect of Update Interval on Travel Time

As expected, in Figures 38 and 39 for both approaches and both networks, average travel time increases as we do the updates less frequently.

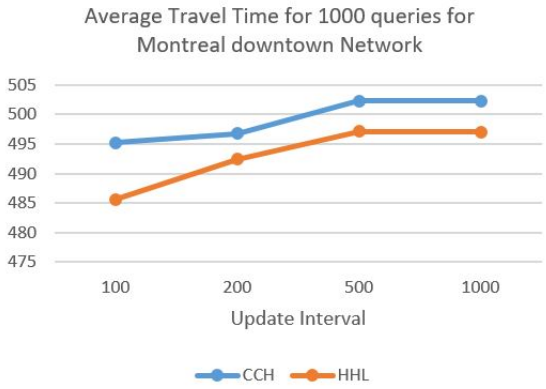


Figure 38: *Avg TT* (s) for different weight update intervals for 1000 queries for Montreal Downtown road network.

Figure 39: *Avg TT* (s) for different weight update intervals and different number of queries for Eichstaett road network.

Interval	Measures	1000Q		5000Q		10000Q	
		CCH+P	*HHL*	*CCH+P*	*HHL*	*CCH+P*	*HHL*
100	*Avg CT*	2791	133519856	4577	-	11800.6	-
	Avg TT	495.24	485.63	774.33	-	1619.99	-
200	*Avg CT*	1596	52274734	3509	-	7607	-
	Avg TT	496.76	492.44	865.26	-	1866.74	-
500	Avg CT	532	27001860.5	1731	-	3855	-
	Avg TT	502.26	497.12	891.75	-	2275.73	-
1000	*Avg CT*	264	17857173	1124	-	2114	-
	Avg TT	502.26	497.02	878.04	-	2358.81	-

Table 23: *CCH+P* and *HHL* with different edge weight update interval and different number of queries for Montreal downtown road network.

Interval	Measures	1000Q		5000Q		10000Q	
		CCH+P	*HHL*	*CCH+P*	*HHL*	*CCH+P*	*HHL*
100	*Avg CT*	582.3	650891	4476	5371200	8672.6	13684369
	Avg TT	249.3	241.93	3034.71	3015.6	4850.84	4833.41
200	*Avg CT*	382.3	439089	2636.6	3968182	4518	8177463.6
	Avg TT	273.66	260.35	3173.27	3161.52	4733.02	4961.10
500	*Avg CT*	154.3	173783	1229	1731928.3	2145.3	3724850.3
	Avg TT	272.22	267.48	3325.49	3321.27	4937.03	5355.41
1000	*Avg CT*	96	86075	680	865964.15	936	1582816.3
	Avg TT	271.07	284.43	3558.98	3528.3	5444.56	5535.6

Table 24: *CCH+P* and *HHL* with different edge weight update interval and different number of queries for Eichstaett road network.

4.3.3 Improving the Performance of Dynamic *HHL*

In this section, we investigate if we can make a better trade-off between the quality of the route and customization time for dynamic *HHL*. For this purpose, we consider both road network instances of our study with 1000 queries and weight update interval of 200 simulation steps. The key idea is to only perform Incremental and Decremental update operations for edge weight changes that change the weight significantly. Figures 40 and 41 show the percentage of edges whose weight has increased or decreased over all the edge weight update intervals.

We run two experiments. Let parameter c represent the minimum percentage of weight change that has to occur for an edge to be considered for an Increment or Decrement operation. For example, $c = 0$ means that any weight change will cause an edge to be considered, and $c = 0.1$ means any edge whose weight changed by more than 10% will be considered.

The first experiment considers the edges whose weight has changed by more than 10 percent. The second experiment only consider the edges whose weight has changed by more than 20 percents. As we see in Tables 23 and 24, even though the customization time has been improved significantly, yet it doesn't compete with the customization time using *CCH+P* algorithm and *CCH* algorithm stands as the best algorithm to use for dynamic scenario.

Measures	$c = 0.1$	$c = 0.2$	All the changes
Avg CT	28838672	25389141	52274734
Avg TT	491.34	490.31	492.44

Table 25: *Avg CT* (ms) and *Avg TT* (s) for $c = 0.1$ and $c = 0.2$ for 1000 queries and 200 as weight update interval for Montreal downtown road network.

Measures	$c = 0.1$	$c = 0.2$	All the changes
Avg CT	112364	52617	439086
Avg TT	256.81	255.47	260.35

Table 26: *Avg CT* (ms) and *Avg TT* (s) for $c = 0.1$ and $c = 0.2$ for 1000 queries and 200 as weight update interval for Eichstaett network.

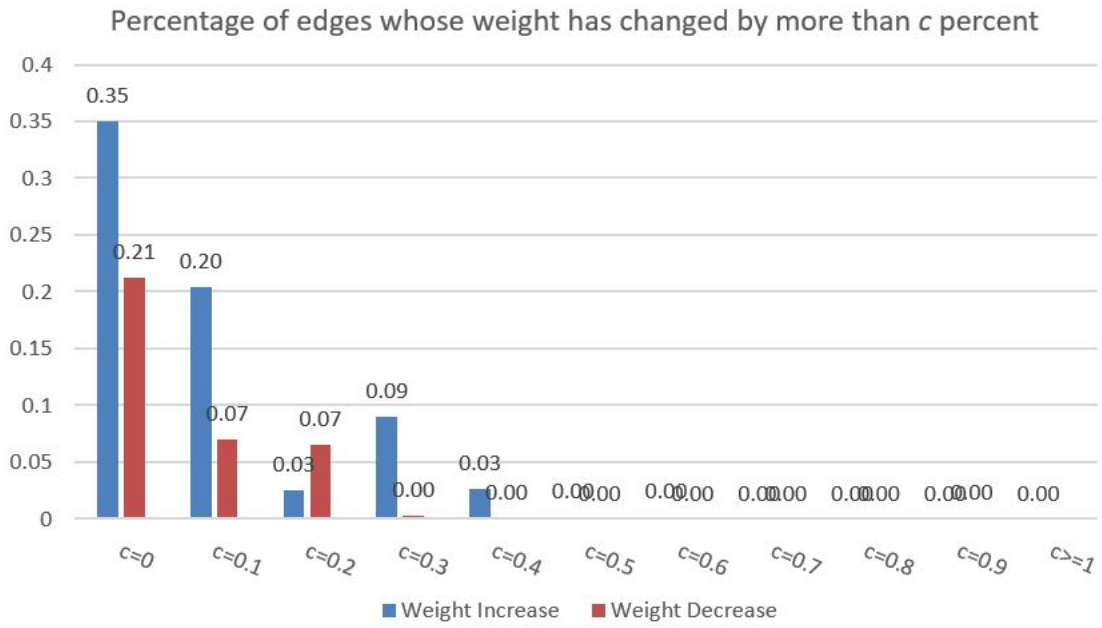


Figure 40: Sum of all the edges whose weight has changed during the simulation for 1000 queries and 200 as weight update interval for Montreal downtown Network.

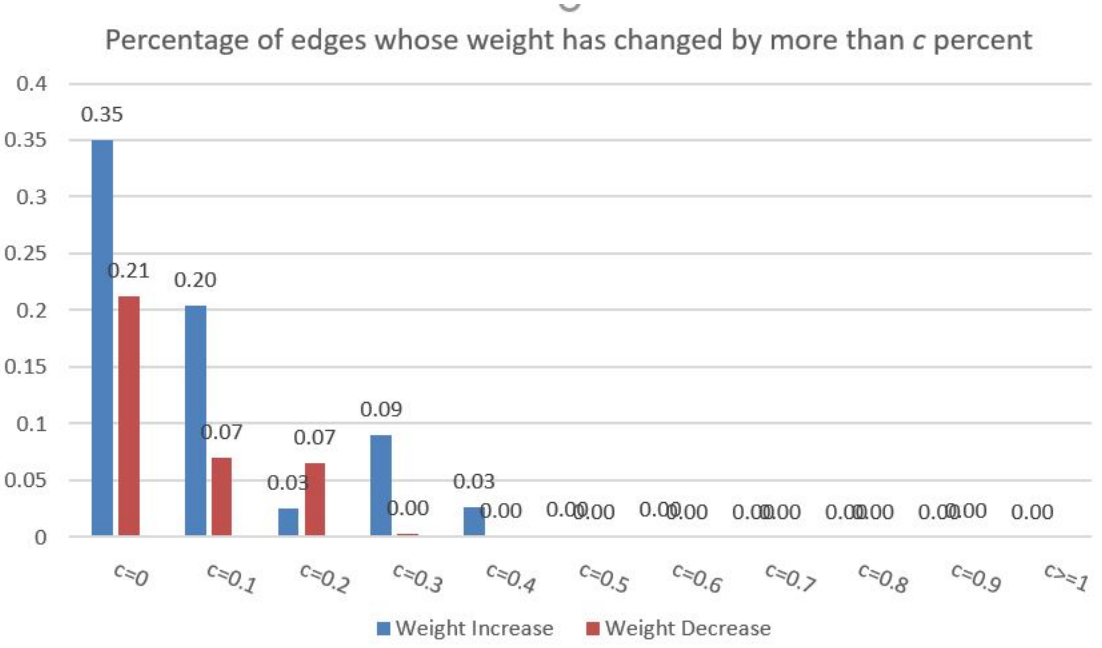


Figure 41: Sum of all the edges whose weight has changed during the simulation for 1000 queries and 200 as weight update interval for Eichstaett Network.

To investigate the reason for the inefficiency of dynamic *HHL*, we measured the average time taken for Incremental updates versus the average time taken for Decremental updates. The results are shown in Table 27 for Montreal and Eichstaett networks for 1000 queries and update interval 200 simulation steps. We see that the average time for Incremental updates is much less than the average time for Decremental updates in both networks.

	Montreal	Eichstaett
Incremental Update	9015.8	239.18
Decremental Update	56917.6	995.6

Table 27: Average time (ms) of running Incremental and Decremental update for 1000 queries and 200 as weight update interval for Montreal downtown and Eichstaett network.

The reason for the inefficiency of Decremental updates, is the process of finding the affected nodes separately for each edge whose weight has increased which requires to run *Dijkstra* (See Algorithm 12). In addition, during the phase in which we add new hubs, we need to run *Dijkstra* for all (u, v) pairs of affected nodes. To show this, we calculated for each edge whose weight has increased, the fraction of nodes in the network that was found by our Algorithm 12 to be "affected", i.e. the fraction of affected nodes. Then we ordered the edges in increasing order of the fraction of affected nodes and plotted it. As we see in Figures 42 and 43, the fraction of affected nodes for each edge whose weight has increased, is about 10% at minimum and about 200% at maximum for the Montreal downtown network and about 30% at minimum and about 200% at maximum for the Eichstaett network. Note that when a node belongs to AFF_a as well as AFF_b for an edge (a, b), then it is counted twice, which is why in some cases, we show that $> 100\%$ of nodes are affected. We can see that for 10 % of the edges, at least 50 % of nodes are affected nodes in the Montreal downtown network, while in the Eichstaett network, for at least 7 % of edges, at least 50 % of the nodes are affected. The high number/percentage of affected nodes illustrates the workload that Decremental update needs to do in dynamic *HHL*.

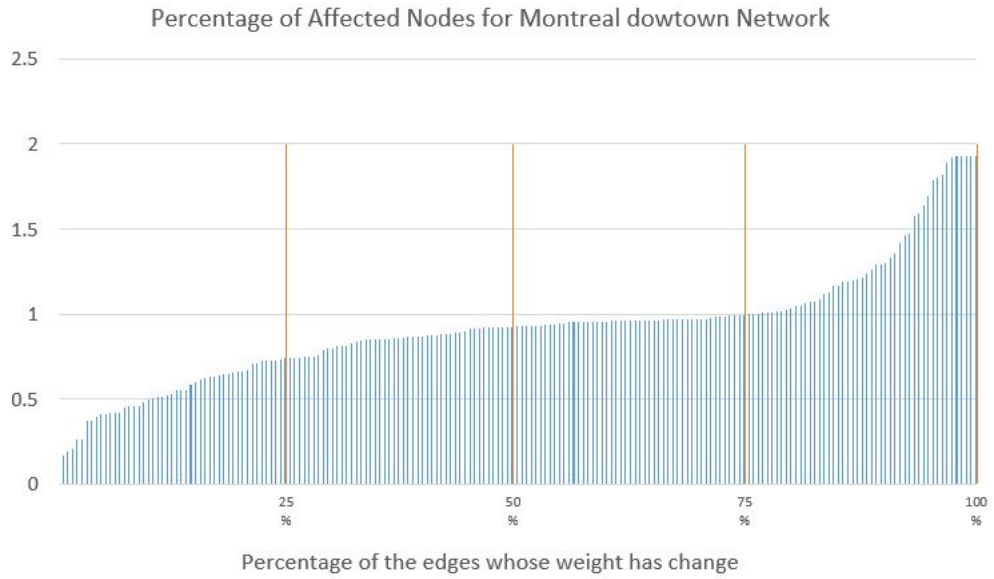


Figure 42: X–axis shows the percentage of edges whose weight has increased in the second interval of weight update. The Y–axis shows the fraction of affected nodes for the Montreal downtown network. (update interval is 200).

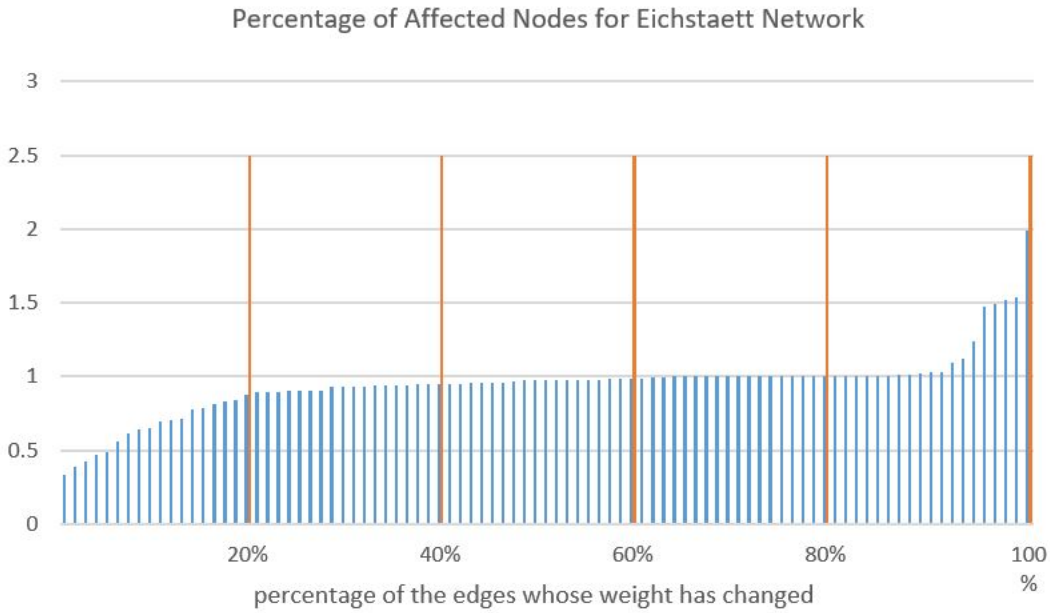


Figure 43: X–axis shows the percentage of edges whose weight has increased in the second interval of weight update. The Y–axis shows the fraction of affected nodes for Eichstaett network. (Update interval is 200).

Chapter 5

Conclusion and Future Work

The use of navigation systems in road networks is increasing every day. In this thesis, we run experiments to compare state of the art routing algorithms that are used to respond to point-to-point queries in a graph. The algorithms in our study are Contraction Hierarchy, Hierarchical Hub Labeling, Customizable Contraction Hierarchy, Customizable Contraction Hierarchy with perfect customization and dynamic Hierarchical Hub Labeling. We run experiments on the Montreal and Eicstaett road networks to measure the performance of each algorithm with respect to the query time and travel time in static road networks and also considering the customization time in dynamic road networks.

Regarding the static routing algorithms, we compared two variants - *R-CH* and *S-CH* of Contraction Hierarchy with respect to query and preprocessing performance and showed that *R-CH* node ordering heuristic, together with stall-on-demand query leads to a better performance. We were interested to see the effect of perfect customization on *CCH*, thus we used two variants of *CCH*, one named as *CCHB* which uses basic customization and the other one, *CCH+P*, which uses using different graph partitioning algorithms and ND-orders. We illustrated that using perfect customization results in a significant improvement in query efficiency. However this is at the cost of slower preprocessing time.

For the Hub Labeling algorithm, we ran experiments with four different node orderings and showed that label construction using *PLL* algorithm together with *BTW-PG* as node ordering results in an efficient shortest distance query performance for both networks. However, label construction is very sensitive to the size of the network. By comparing all these five algorithms together, we showed that, although in terms of distance query times *HHL* algorithm was the fastest in both networks, it is not a good approach to answer shortest path query. For shortest path queries, *CCH+P* is significantly better than *HHL*. We observed that both variants of *CCH+P*, have a significant improvement in preprocessing time compared to *R-CH*. Regarding the query time, *CCH+P* with *FC20* as graph partitioning algorithm for Montreal network, and *Metis* for Eichstaett, results in a better query performance compared to *R-CH*. All the speed-up routing algorithms have better performance than *Dijkstra* and *Astar*.

Another interesting experiment was to see how dynamic speed-up algorithms react to changes of the network. We ran this experiment with *CCH* and dynamic *HHL* since we can respond to edge weight changes without running the entire preprocessing phase. We use three sets of queries 1000, 5000 and 10000 queries. We observed that *CCH+P* outperforms *HHL* for any size of network and queries in terms of the customization time. For dynamic *HHL* our results showed that running the preprocessing is more efficient than updating labels after each update change interval. We investigated the reason for low efficiency of dynamic *HHL* and observed that for both networks, for about 10% of the edges whose weight has changed already contains half of the nodes as affected nodes which in consequence leads to a very expensive decremental update operation.

It would be interesting to do our experiments on other city networks, and also larger road networks. However in larger networks, the time taken is prohibitive even for road networks, and paralleization and more powerful computers would be needed. Implementations based on real-time traffic data from road networks, rather than simulated traffic, would also be interesting.

Finally, our experiments make it clear that dynamic hub labelling, particularly the decremental update operation, is not efficient in practice. A new approach to handle several edge weight changes together, rather than individually, would seem to be needed. Another algorithmic contribution could be to find an ordering for nodes

that leads to better query performance.

Bibliography

[1] https://www.bloomberg.com/news/articles/2019-11-12/navigation-apps-changed-the-politics-of-traffic.

[2] https://inrix.com/products/roadway-analytics/.

[3] https://www.eclipse.org/sumo/.

[4] https://mitpress.mit.edu/books/introduction-algorithms.

[5] https://www.graphhopper.com/blog/2017/08/14/flexible-routing-15-times-faster/.

[6] https://github.com/kit-algo/InertialFlowCutter.

[7] http://www.openstreetmap.org.

[8] Ittai Abraham, Daniel Delling, Andrew V Goldberg, and Renato F Werneck. A hub-based labeling algorithm for shortest paths in road networks. In *International Symposium on Experimental Algorithms*, pages 230–241. Springer, 2011.

[9] Ittai Abraham, Daniel Delling, Andrew V Goldberg, and Renato F Werneck. Hierarchical hub labelings for shortest paths. In *European Symposium on Algorithms*, pages 24–35. Springer, 2012.

[10] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 349–360, 2013.

[11] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling. In *Proceedings of the 23rd international conference on World wide web*, pages 237–248, 2014.

[12] Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F Werneck. Route planning in transportation networks. In *Algorithm engineering*, pages 19–80. Springer, 2016.

[13] Holger Bast, Stefan Funke, Domagoj Matijevic, Peter Sanders, and Dominik Schultes. In transit to constant time shortest-path queries in road networks. In *2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 46–59. SIAM, 2007.

[14] Reinhard Bauer, Tobias Columbus, Bastian Katz, Marcus Krug, and Dorothea Wagner. Preprocessing speed-up techniques is hard. In *International Conference on Algorithms and Complexity*, pages 359–370. Springer, 2010.

[15] Reinhard Bauer, Tobias Columbus, Ignaz Rutter, and Dorothea Wagner. Search-space size in contraction hierarchies. *Theoretical Computer Science*, 645:112–127, 2016.

[16] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 32(5):1338–1355, 2003.

[17] Tobias Columbus. On the complexity of contraction hierarchies. *Student Thesis, Karlsruhe Inst. Technol., Karlsruhe, Germany*, 2009.

[18] Gianlorenzo D'angelo, Mattia D'emidio, and Daniele Frigioni. Fully dynamic 2-hop cover labeling. *Journal of Experimental Algorithmics (JEA)*, 24(1):1–36, 2019.

[19] George Bernard Dantzig. *Linear programming and extensions*, volume 48. Princeton university press, 1998.

[20] Daniel Delling, Andrew V Goldberg, Thomas Pajor, and Renato F Werneck. Robust exact distance queries on massive networks. *Microsoft Research, USA, Tech. Rep*, 2, 2014.

[21] Daniel Delling, Andrew V Goldberg, Thomas Pajor, and Renato F Werneck. Customizable route planning in road networks. *Transportation Science*, 51(2):566–591, 2017.

[22] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Customizable contraction hierarchies. *Journal of Experimental Algorithmics (JEA)*, 21:1–49, 2016.

[23] Edsger W Dijkstra et al. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[24] Stefan Funke, André Nusser, and Sabine Storandt. On k-path covers and their applications. *Proceedings of the VLDB Endowment*, 7(10):893–902, 2014.

[25] Cyril Gavoille, David Peleg, Stéphane Pérennes, and Ran Raz. Distance labeling in graphs. *Journal of Algorithms*, 53(1):85–112, 2004.

[26] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *International Workshop on Experimental and Efficient Algorithms*, pages 319–333. Springer, 2008.

[27] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact routing in large road networks using contraction hierarchies. *Transportation Science*, 46(3):388–404, 2012.

[28] Alan George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, 1973.

[29] Andrew V Goldberg. Point-to-point shortest path algorithms with preprocessing. In *International Conference on Current Trends in Theory and Practice of Computer Science*, pages 88–102. Springer, 2007.

[30] Andrew V Goldberg and Chris Harrelson. Computing the shortest path: A search meets graph theory. In *SODA*, volume 5, pages 156–165, 2005.

[31] Lars Gottesbüren, Michael Hamann, Tim Niklas Uhl, and Dorothea Wagner. Faster and better nested dissection orders for customizable contraction hierarchies. *Algorithms*, 12(9):196, 2019. `https://arxiv.org/abs/1906.11811`.

[32] Ronald J Gutman. Reach-based routing: A new approach to shortest path algorithms optimized for road networks. *ALENEX/ANALC*, 4:100–111, 2004.

[33] Michael Hamann and Ben Strasser. Graph bisection with pareto optimization. *Journal of Experimental Algorithmics (JEA)*, 23:1–34, 2018.

[34] Amy He. People continue to rely on maps and navigational apps. `https://www.emarketer.com`.

[35] Moritz Hilger, Ekkehard Köhler, Rolf H Möhring, and Heiko Schilling. Fast point-to-point shortest path computations with arc-flags. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, 74:41–72, 2009.

[36] Martin Holzer, Frank Schulz, and Dorothea Wagner. Engineering multilevel overlay graphs for shortest-path queries. *Journal of Experimental Algorithmics (JEA)*, 13:2–5, 2009.

[37] Minhao Jiang, Ada Wai-Chee Fu, Raymond Chi-Wing Wong, and Yanyan Xu. Hop doubling label indexing for point-to-point distance querying on scale-free networks. *arXiv preprint arXiv:1403.0779*, 2014.

[38] Sungwon Jung and Sakti Pramanik. An efficient path computation model for hierarchically structured topographical road maps. *IEEE Transactions on Knowledge and Data Engineering*, 14(5):1029–1046, 2002.

[39] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998. `http://glaros.dtc.umn.edu/gkhome/metis/metis/download`.

[40] W Kim, V Anorve, and BC Tefft. American driving survey, 2014–2017. 2019.

[41] Ulrich Lauther. An experimental evaluation of point-to-point shortest path calculation on road networks with precalculated edge-flags. In *The Shortest Path Problem*, pages 19–39, 2006.

[42] Ye Li, Leong Hou U, Man Lung Yiu, and Ngai Meng Kou. An experimental study on hub labeling based shortest path algorithms. *Proceedings of the VLDB Endowment*, 11(4):445–457, 2017.

[43] Jens Maue, Peter Sanders, and Domagoj Matijevic. Goal-directed shortest-path queries using precomputed cluster distances. *Journal of Experimental Algorithmics (JEA)*, 14:3–2, 2010.

[44] United Nations. 2018 revision of world urbanization prospects, 2018.

[45] RILEY PANKO. The Popularity of Google Maps: Trends in Navigation Apps in 2018, url = https://themanifest.com/, urldate = 10 JULY 2018.

[46] David Peleg. Proximity-preserving labeling schemes. *Journal of Graph Theory*, 33(3):167–176, 2000.

[47] Bob Pishue. Us traffic hot spots: Measuring the impact of congestion in the united states. 2017.

[48] Trevor Reed. Inrix global traffic scorecard. 2019.

[49] Peter Sanders and Dominik Schultes. Engineering highway hierarchies. In *European Symposium on Algorithms*, pages 804–816. Springer, 2006.

[50] Peter Sanders and Christian Schulz. Think locally, act globally: Highly balanced graph partitioning. In *International Symposium on Experimental Algorithms*, pages 164–175. Springer, 2013.

[51] Ruslan Savchenko. Hub labeling algorithms. `https://github.com/savrus/hl`.

[52] Alejandro A Schäffer. Optimal node ranking of trees in linear time. *Information Processing Letters*, 33(2):91–96, 1989.

[53] Aaron Schild and Christian Sommer. On balanced separators in road networks. In *International Symposium on Experimental Algorithms*, pages 286–297. Springer, 2015.

[54] Dominik Schultes. Route planning in road networks. In *Ausgezeichnete Informatikdissertationen*, pages 271–280, 2008.

[55] Dominik Schultes and Peter Sanders. Dynamic highway-node routing. In *International Workshop on Experimental and Efficient Algorithms*, pages 66–79. Springer, 2007.

[56] Christian Schulz. Karlsruhe high quality partitioning. `https://github.com/KaHIP/KaHIP`.

[57] Frank Schulz, Dorothea Wagner, and Karsten Weihe. Dijkstra's algorithm online: an empirical case study from public railroad transport. *Journal of Experimental Algorithmics (JEA)*, 5:12–es, 2000.

[58] Ben strasser. Routingkit. `https://github.com/RoutingKit/RoutingKit`.

[59] Cambridge Systematics. Traffic congestion and reliability: Trends and advanced strategies for congestion mitigation. Technical report, United States. Federal Highway Administration, 2005.

[60] Dirck Van Vliet. Improved shortest path algorithms for transport networks. *Transportation Research*, 12(1):7–20, 1978.

[61] Dorothea Wagner and Thomas Willhalm. Speed-up techniques for shortest-path computations. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 23–36. Springer, 2007.

[62] Dorothea Wagner, Thomas Willhalm, and Christos Zaroliagis. Geometric containers for efficient shortest-path computation. *Journal of Experimental Algorithmics (JEA)*, 10:1–3, 2005.

[63] Yosuke Yano, Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. Fast and scalable reachability queries on graphs by pruned labeling with landmarks and paths. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 1601–1606, 2013.

[64] Tetyana Yaropud, Jason Gilmore, and Sébastien LaRochelle-Côté. *Results from the 2016 Census: Long commutes to work by car*. Statistics Canada= Statistique Canada, 2019.