Institute of Formal Methods in Computer Science

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Performance Measurements for Personalizable Route Planning for uncorrelated Edge Costs

Felix Bühler

Course of Study: Informatik

Examiner: Prof. Dr. Stefan Funke

Supervisor: M. Sc. Florian Barth

Commenced: March 15, 2021

Completed: September 15, 2021

Abstract

Nowadays, ordinary route planners compute paths by choosing the shortest or fastest route. However, there exist additional metrics from which users with varying preferences could benefit. *Personalized route planning* offers the possibility to combine different metrics with personal preferences. Nevertheless, *personalized route planning* has mainly been tested with correlated metrics. But when including uncorrelated metrics, the computing time increases significantly. Previous work found that the speedup technique “Customizable Route Planning” can lead to feasible speedups for single metric calculations. Thus, in this work, we investigate how this speedup technique for Dijkstra improves the query performances of “Personalizable Route Planning” compared to “Personalizable Contraction Hierarchies”. Furthermore, we study the performances on uncorrelated metrics. We introduce a graph structure to compare the personalized speedup techniques “Personalizable Contraction Hierarchies”, “Personalizable Customizable Route Planning” and “Personalizable Route Planning”. Three graph partitioning algorithms have been implemented to realize “Customizable Route Planning”: K-means, Gonzales, and Merge. Our experiments show that Merge works well in combination with “Personalizable Contraction Hierarchies” preprocessing. We found that “Personalizable Customizable Route Planning” is a good alternative, as it uses much fewer edges for finding the costs of the shortest path. For uncorrelated metrics, “Personalizable Customizable Route Planning” and “Personalizable Route Planning” achieved speedups higher than “Personalizable Contraction Hierarchies”. Our contribution comprises a novel graph structure for comparing different Dijkstra variants. With our experiments, we provide a deeper understanding of the *personalized route planning* problem. Additionally, we propose improvements for “Personalizable Contraction Hierarchies” for less contracted graphs with uncorrelated metrics.

Kurzfassung

Heutzutage berechnen gewöhnliche Routenplaner Strecken, indem sie dabei die kürzeste oder schnellste Route auswählen. Es existieren jedoch zusätzliche Metriken, von welchen Nutzer mit unterschiedlichen Präferenzen profitieren könnten. Die *personalisierte Routenplanung* bietet die Möglichkeit, verschiedene Metriken anhand persönlicher Präferenzen zu kombinieren. Allerdings wurde die *personalisierte Routenplanung* bisher hauptsächlich auf korrelierten Metriken getestet. Werden jedoch unkorrelierte Metriken miteinbezogen, erhöht sich die Berechnungszeit erheblich. Frühere Arbeiten haben gezeigt, dass die Beschleunigungstechnik „Customizable Route Planning“ zu brauchbaren Beschleunigungen für einzelne Metrikberechnungen führen kann. Deshalb untersuchen wir in dieser Arbeit, wie diese Beschleunigungstechnik für Dijkstra die Abfrageleistungen von „Personalizable Route Planning“ im Vergleich zu „Personalizable Contraction Hierarchies“ verbessert. Zusätzlich untersuchen wir die Abfrageleistungen auf unkorrelierten Metriken. Wir führen eine Graphstruktur ein, um die personalisierten Beschleunigungstechniken „Personalizable Contraction Hierarchies“, „Personalizable Customizable Route Planning” und „Personalizable Route Planning“ zu vergleichen. Drei Graphpartitionierungsalgorithmen wurden implementiert um „Customizable Route Planning“ zu realisieren: K-means, Gonzalez, und Merge. Unsere Experimente zeigen, dass Merge in Kombination mit dem „Personalizable Contraction Hierarchies“-Preprocessing zu guten Ergebnissen führt. Wir haben herausgefunden, dass „Personalizable Customizable Route Planning“ eine gute Alternative ist, da weniger Kanten für die Ermittlung der Kosten des kürzesten Pfades verwendet werden. Für unkorrelierte Metriken erreichten „Personalizable Customizable Route Planning“ und „Personalizable Route Planning“ höhere Geschwindigkeitssteigerungen als „Personalizable Contraction Hierarchies“. Unser Beitrag umfasst eine neuartige Graphenstruktur, die es ermöglicht, verschiedene Dijkstra-Varianten zu vergleichen. In unseren Experimenten vermittlen wir damit ein tieferes Verständnis des Problems der *personalisierten Routenplanung*. Zusätzlich schlagen wir Verbesserungen für „Personalizable Contraction Hierarchies“ auf weniger kontrahierten Graphen mit unkorrelierten Metriken vor.

Contents

1 Introduction 15
1.1 Related Work . . . 17
1.2 Contribution . . . 18

2 Preliminaries 19
2.1 Dijkstra . . . 19
2.2 Contraction Hierarchies . . . 20
2.3 Customizable Route Planning . . . 22
2.4 Personalizable Route Planning . . . 25

3 Implementation 27
3.1 Multi-Level-Partitioning . . . 28
3.2 Preprocessing . . . 34
3.3 Queries . . . 37

4 Experimental Evaluation 43
4.1 Data and Settings . . . 43
4.2 Experimental Results . . . 43

5 Conclusion and Future Work 57

Bibliography 59

List of Figures

1.1 Possible paths with different user preferences but same start and target 16

2.1 Search space of Dijkstra and Bidirectional Dijkstra 20
2.2 Search space and termination condition of CH-Dijkstra 20
2.3 Search space of CRP-Dijkstra 23

3.1 The implemented pipeline of the data 27
3.2 Example why the largest-connected-component is extracted 28
3.3 Boundary nodes of MLP K-means 30
3.4 Gonzalez problems on directed graph 31
3.5 Boundary nodes of MLP Gonzalez 32
3.6 Boundary nodes of MLP Merge 33
3.7 Problem of contracting in parallel 34
3.8 Internal graph structure after contraction 35
3.9 Web-client for *personalized route planning* 37
3.10 Example query for Dijkstra and Bidirectional Dijkstra 38
3.11 PCH-Dijkstra 39
3.12 PCRP-Dijkstra 40
3.13 PRP-Dijkstra 41

4.1 MLPs number of boundary nodes 44
4.2 MLPs maximum partition size 45
4.3 MLPs comparison of maximum partition size and number of boundary nodes . . 45
4.4 MLPs run time 46
4.5 Preprocessing runtime of different MLP-methods 47
4.6 Dijkstra variants comparison with different MLPs 48
4.7 Heap-pops for single-level-partitioning 50
4.8 Relaxed-edges for single-level-partitioning 50
4.9 PCH preprocessing 51
4.10 PCH queries with MLP vs without MLP 52

List of Tables

3.1 Edge sorting for the different Dijkstra algorithms 36

4.1 Average nodes per partition for the state “Baden-Württemberg” 43
4.2 Preprocessing time and percentage of contracted nodes for each MLP 47
4.3 Dijkstra variants timings and resulting speedup factors 48
4.4 PCH query timings and resulting speedup factors 52
4.5 Used edges for an MLP of each Dijkstra variant 53
4.6 Query-times and speedups for different Merge partitionings 54
4.7 Timings and speedups for the car graph with additional metric *random* 55
4.8 Timings and speedups for the car graph with additional metric *height ascent* . . . 55
4.9 Timings and speedups for the bicycle graph 56

Acronyms

BFS Breadth-First Search. 28

CCH Customizable Contraction Hierarchies. 20

CH Contraction Hierarchies. 15

CRP Customizable Route Planning. 15

LP Linear Programming. 21

MLP Multi-Level-Partitioning. 18

OSM OpenStreetMap. 28

PCH Personalizable Contraction Hierarchies. 15

PCRP Personalizable Customizable Route Planning. 18

PRP Personalizable Route Planning. 15

PUNCH Partitioning Using Natural Cut Heuristics. 24

RNG Random Number Generator. 25

1 Introduction

Navigation systems have become an indispensable part of our daily lives. Most route planners, e.g., Google Maps, Bing Maps, and Apple Maps, efficiently compute an optimal path between a start and a destination. Currently, the queries can be computed by choosing a single metric, typically the fastest or shortest route. Metrics do not have to be static and might change. A traffic jam can occur if an accident happens, which influences the travel time needed for the road, thus affecting the travel-time metric. However, numerous other metrics exist than currently offered by route planners. Some less common metrics are, e.g., road bumpiness, how much nature can be seen, or how often accidents occur on the road. While most people are satisfied with using the fastest or shortest metrics, some might be interested in using less common metrics. However, popular route planners do not allow users to take advantage of such uncommon metrics. People with increased sensitivity to noise prefer to travel exclusively on quiet roads. Motorcyclists prefer curvy roads. Electric cars prefer not to have too much height ascent [12]. There exist route planners for specific use cases [4, 28]. Motorcyclists with electric bikes might want to combine the curviness metric with the height ascent metric, which is currently impossible. Every person has different personal preferences for their travel type. For providing a personalized, tailored query, a combination of many metrics could fix this issue. For example, an electric motorcyclist with increased sensitivity to noise might want to have a weighting of 60% for quiet roads, 30% for height ascents, and 10% for curvy roads. The currently used algorithms have to be extended in order to provide such personalized queries. Figure 1.1 depicts all possible routes of a graph with three metrics having the same starting and target position but different user preferences.

The classical solution for finding paths is the Dijkstra algorithm [11]. The Dijkstra algorithm is sufficient for federal-state-sized graphs, but the query times become impracticable for continental-sized graphs. Thus, speedup techniques have been developed for Dijkstra, which can reduce the query time from several seconds down to milliseconds or even microseconds [1]. Precomputing auxiliary data achieve speedups. Widespread speedup techniques with relatively small auxiliary data and acceptable query times are Customizable Route Planning (CRP) and Contraction Hierarchies (CH). Query algorithms have to be adapted to provide personalized queries with multiple metrics. Dijkstra can easily be extended to support personalized queries, but with the identical downside on more extensive graph networks for being slow. For CRP and CH, the preprocessing phase and query additionally have to be changed to support personalized queries. Funke and Storandt [15] introduced a personalized CH which is called Personalizable Contraction Hierarchies (PCH). Additionally, they proposed a Personalizable Route Planning (PRP) variant which is a combination of CRP and PCH. Barth et al. [2] improved preprocessing and query times of PCH and mentioned that if metrics are uncorrelated, the preprocessing time gets longer. Moreover, the amount of auxiliary data increases in contrast to using correlated metrics. To support a multitude of metrics, uncorrelated metrics should not introduce problems for preprocessing and memory. For example, if metric updates occur,

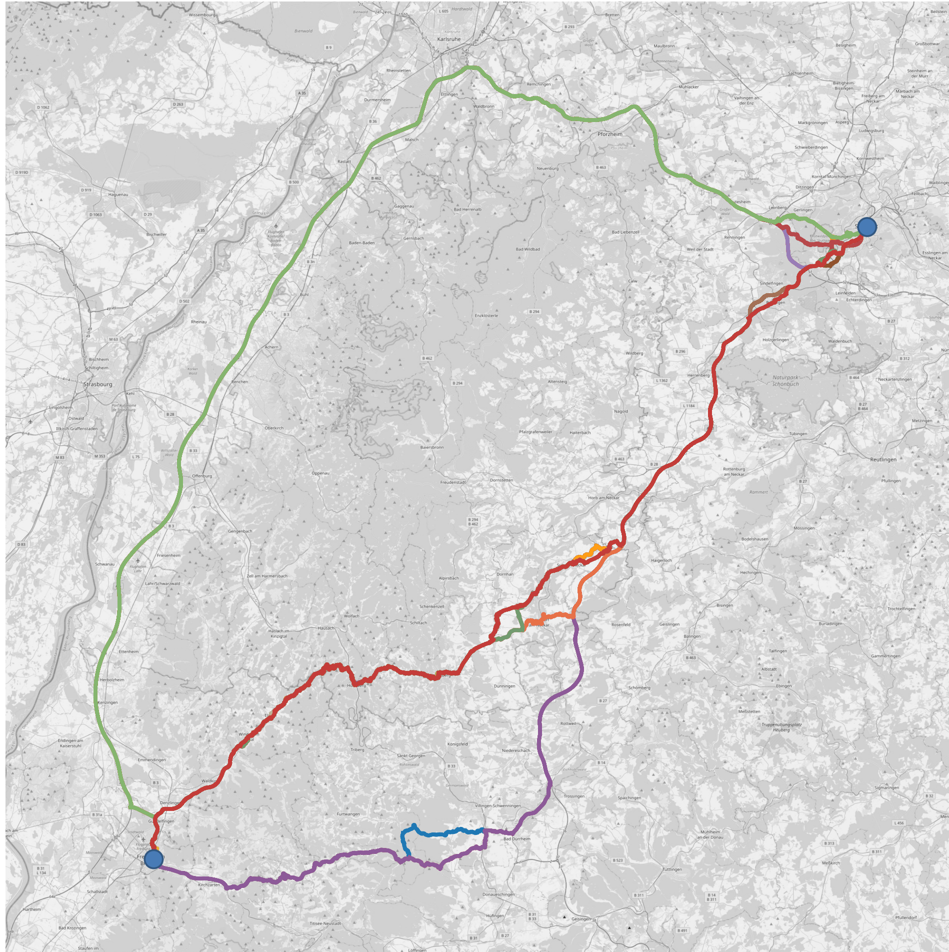


Figure 1.1: Different possible paths from "Stuttgart" to "Freiburg im Breisgau" using different user preferences for three metrics (*distance*, *uniform costs*, *car travel time*).

an accident happens, the auxiliary data may need recalculations. Therefore it is beneficial to ensure that these recalculations are fast, so users receive path updates fast and are not guided into a new traffic jam.

1.1 Related Work

Dijkstra [11] begins at the start node and expands to not yet visited nodes by continuously extending to the node with minimal costs. Through exploration of the graph, the Dijkstra algorithm can terminate if the goal is reached. A speedup technique that does not need any preprocessing is called Bidirectional Dijkstra [33]. As the name suggests, it searches from the start node and the target node. Both instances of Dijkstra will meet in the *middle* and can determine the shortest path. A more detailed explanation for both will be provided in Chapter 2. There are several speedup techniques available for the original Dijkstra algorithm. A good overview is provided by the survey paper from Bast et al. [3], aiming for a worldwide multi-modal journey planner.

Another speedup technique called table-lookup stores the paths from one node to all others in a table. A simple table-lookup has to be done when users request a route, and the results can be retrieved. This is considered to be the fastest possible speedup technique. The precalculation of such paths is often called preprocessing. For calculating all possible paths without personalization, a speedup technique has been proposed [6], which improves the preprocessing time. Table-lookup is infeasible for country-sized road networks in terms of memory. By adding personalization, the problem becomes worse because, for each pair of nodes, all possible personalized routes have to be stored.

Dijkstra itself has a slow query-time performance but does not need additional memory in terms of preprocessing data. In contrast, table-lookup provides a good query time but needs much space in memory. The speedup techniques mentioned below provide a more balanced trade-off between memory and query time. Generally, an overlay graph G' is generated from the original graph G by preserving the shortest path property. G' is a sparse graph because it consists of fewer nodes than G. The query starts on G, and when reaching a node that corresponds to a node in G', it can switch to G' and continues running Dijkstra. By doing so, the amount of visited nodes is reduced.

The speedup techniques can be generally split into four groups. The groups are called goal-directed, hierarchical, separator-based, and bounded-hop techniques:

- **Goal-directed** speedup techniques often use heuristics [18, 21]. It avoids scans of vertices that are not in the direction of the target. For each node, the spatial data of the road network is used for guidance. However, these heuristics for personalization are not trivial and have to be further investigated. Goal-directed techniques are not the objective of this work and have therefore not been considered.
- **Hierarchical** techniques exploit the inherent hierarchy of road networks, meaning longer paths might prefer the highway instead of dirt roads connecting villages. There are multiple hierarchical approaches [29, 31] of which CH [17] is considered the most popular. The preprocessing assigns each node a rank, which generates a hierarchy between these nodes. Shortcuts spanning *unimportant* nodes are created while generating the hierarchy. The bidirectional query starts the same as the Bidirectional Dijkstra, except it is only allowed to walk to nodes with an equal or higher hierarchy than the node itself. Both Dijkstras will meet at an *important* node. More details will be provided in Section 2.2.
- **Separator** techniques exploit the possibility to separate the graph into several cells. The generated cells are often called partitions. In the preprocessing step, all possible paths spanning the cell have to be calculated. These calculated paths can be reused and thereby

provide a speedup. To provide a speedup, partitioning is desired to have as few separators as possible and separate the graph at tunnels, overpasses, or bridges. Additionally, it is desired that the resulting cells are well balanced to provide steady query times. There are vertex-separator [34] and arc-separator-based methods [7]. Vertex-separators split the graph at specific nodes, whereas arc-based methods separate the partitions in-between connected nodes. The most popular separator technique for road networks is CRP [7], which is an arc separator speedup method. It will be further discussed in Section 2.3.

- **Bounded-hop** techniques yield the most speedups without calculating all possible paths [1]. The single-hop methods can be considered to be equivalent to table-lookup. Considering algorithms with more hops (two or three) has a much more reasonable trade-off. We assume that the combinatorial space is prohibitive for using this approach with large road networks for personalization. Therefore, we did not investigate these techniques further.

Speedup techniques are also frequently combined to use the advantages of multiple techniques. In this work, we utilize the speedup technique PRP, which we explain in detail in Section 2.4. It is a combination of Personalizable Customizable Route Planning (PCRP) and PCH.

1.2 Contribution

The focus of this work is on speedup techniques supporting one-to-one pathfinding with personalization. Compared to PCH, PRP provides promising results for personalization. Therefore, this work investigates how the CRP query algorithm performs. The preprocessing of PCH will be used to generate auxiliary data for CRP and PRP. We will show how uncorrelated cost metrics influence preprocessing and query performance. Due to the problems of contracting graphs with uncorrelated costs [2], there is an additional focus on how the Dijkstra-variants will perform on graphs that are not contracted as much as the ideal state for PCH. We developed a new graph structure to compare the different Dijkstra speedup variants. To support PCRP, three different Multi-Level-Partitioning (MLP) methods have been implemented: K-means, Gonzalez, and Merge. Performance measurements for PCRP, PCH, and PRP have been executed. For uncorrelated metrics, PCRP and PRP achieved speedups higher than PCH.

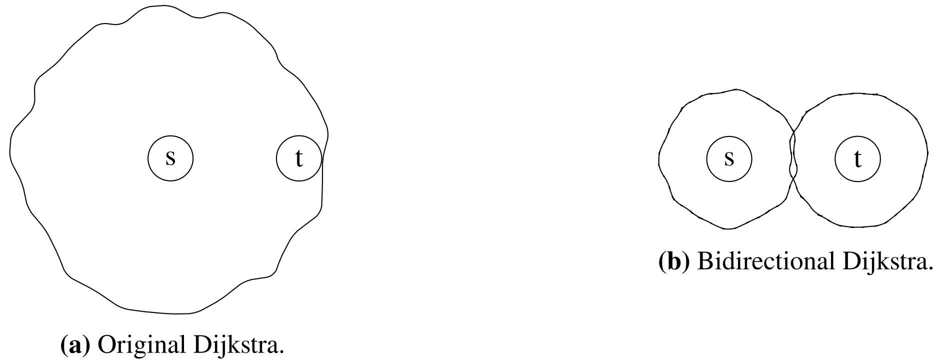
2 Preliminaries

This chapter formalizes *personalized route planning* and will have a deeper look into algorithms implemented in the next section. It is worth mentioning that *personalized route planning* is the problem and Personalizable Route Planning (PRP) is a speedup technique. For the problem, the definition from Funke and Storandt [15] is used: given the graph $G = (V, E)$ representing the street-network with edge $e(v_i, v_j) : e \in E, v \in V$ connecting exactly two vertices. Every edge has a dimensional non-negative cost vector $c(e) \in (\mathbb{R}_0^+)^d : e \in E$. Each entry reflects one metric, e.g., c_1 = distance, c_2 = travel time, c_3 = positive height ascent, etc. A path π is a list of vertices $(v_0, v_1, ...v_n) : v \in V$, where v_i and v_{i+1} have to be connected via an edge $e(v_i, v_{i+1})$. Given a personalized query users have to provide an start s and target t, where $s, t \in V$ and an $\alpha = (\mathbb{R}_0^+)^d$ vector. Each entry of α corresponds to the users personal preference according to $c(e)$. The goal is to find the path satisfying the following Equation (2.1), where P corresponds to all possible routes starting with s and ending with t.

$$\min_{\forall \pi \in P} \sum_{e \in \pi} \alpha^T c(e) \tag{2.1}$$

2.1 Dijkstra

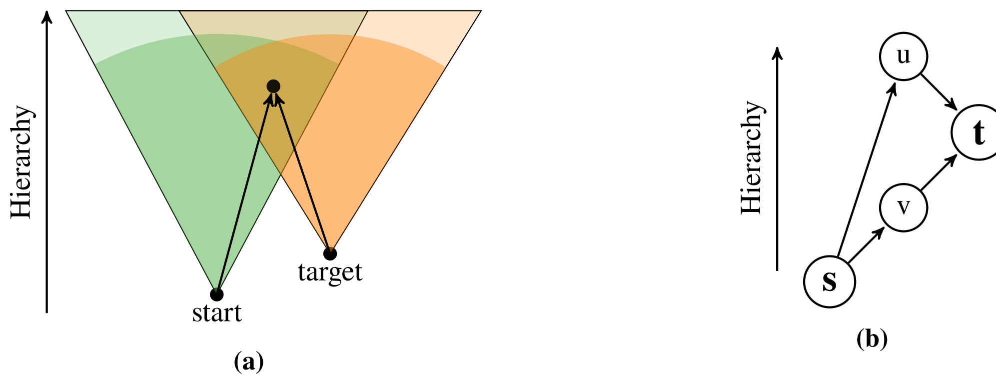
Dijkstra [11] is the most popular algorithm for solving the shortest-path-finding problem. It begins the search at the start node, iterates over the outgoing edges, and inserts all unvisited neighboring nodes with the costs into a min-heap. This min-heap is used to keep track of "what is the next least costly unvisited node". A separate array is used for keeping track of whether the node is visited or not. Theoretically, this array could only store a boolean, but instead, it uses "infinity" if the node is unvisited or the path costs if it has been visited. Dijkstra expands to the next node from the min-heap. It inserts all the unvisited neighboring nodes with outgoing edges of the node into the min-heap. By continuously doing that, Dijkstra extends the search while maintaining the optimal costs for all visited nodes. If the same node is inserted into the heap multiple times, only the first (due to the min-heap, the node with the lowest costs) is taken out. The other costs of the same node are ignored because there is another path with lower costs. This process continues until Dijkstra reaches the goal and terminates. The stored costs in the array are the minimal costs. Dijkstra can save minimal costs for each node and store the used edge on how it got here to reconstruct the full path. At the target node, the used edge can be used to look up its predecessor. By doing this recursively, the path to the start is found. An advantage of Dijkstra is the calculation of one-to-many shortest paths by not specifying a goal. However, this has the downside of calculating many paths, which are not needed in a one-to-one shortest path query. The search space of Dijkstra can be interpreted as an expanding circle around the start node, as can be seen in Figure 2.1a.



(a) Original Dijkstra.

(b) Bidirectional Dijkstra.

Figure 2.1: Visual interpretation of the search space of Dijkstra and Bidirectional Dijkstra.



(a)

(b)

Figure 2.2: (a) Search space of CH and (b) example why both heaps have to exceed the costs of the optimal meeting point for CH-Dijkstra.

The simplest speedup, which does not need any precomputation, is called Bidirectional Dijkstra [33]. Compared to the original Dijkstra, two instances of Dijkstra are started: one from the start (forward-Dijkstra) and one from the target node (backward-Dijkstra). A visual comparison of both search spaces can be seen in Figure 2.1b. The forward-Dijkstra begins at the start node and traverses the graph in the same way as the original Dijkstra. The backward-Dijkstra starts at the target node and traverses the graph backward. This requires the graph structure to be extended to support iterating over incoming edges of a node. A meeting point is the node both Dijkstra's have visited. Original Dijkstra itself can terminate if it reaches the goal. In contrast, the Bidirectional Dijkstra can terminate if the sum of the cost from both next heap elements exceeds the cost of reaching the meeting node [19]. For reconstructing the path, both Dijkstra arrays can be used to iterate from the meeting node to the start and target. This method typically has a speedup factor of about two.

2.2 Contraction Hierarchies

The CH is a hierarchical speedup technique proposed by Geisberger et al. [17]. An additional improvement for faster preprocessing is Customizable Contraction Hierarchies (CCH) [10], which is not part of this work. The core idea of CH is to start with the original graph and replace the edges of an *unimportant* node with shortcuts. Each new shortcut introduces an edge that skips a

single node. Creating shortcuts spanning the node is called "contracting the node". A shortcut can represent a pair of edges, a pair of shortcuts, or a combination of an edge and a shortcut. These shortcuts do not violate the correctness of the shortest-path property by summing the costs of the edges they are representing. For every shortcut, the connected nodes have a higher rank than the one skipped. This introduces a hierarchy for all nodes, which is called a "rank". By calculating these shortcuts, the number of edges and nodes the query has to visit will be reduced, thereby reducing the query time. To choose which nodes are getting contracted, a heuristic is used, often referred to as node-ordering. The query is a specialized bidirectional Dijkstra algorithm called CH-Dijkstra. CH-Dijkstra only works with bidirectional queries. The CH-Dijkstra starts from the start and the target, like the Bidirectional Dijkstra and is allowed to visit only nodes with the same or a higher rank than the previously visited node rank, as shown in Figure 2.2a. When both Dijkstra's have a meeting point, they have to continue until the first element of both heaps is larger than the cost of the meeting point. Figure 2.2b shows why the termination condition is different compared to the Bidirectional Dijkstra. If nodes s and t have been visited, path s-u-t can be more costly than path s-v-t. To make sure the path is the shortest, v has to be visited too.

Contracting the whole graph is not necessary. Instead, it is sufficient to contract until a certain threshold is reached. Example thresholds are a percentage of the number of nodes, maximum costs of edges, or a predefined set of nodes (specifying which nodes have to be contracted and which are left out). The not contracted nodes are called "core-nodes" and are assigned the same maximum rank. By allowing the CH-Dijkstra to visit neighbors with the same rank, the Dijkstra walking on the core-nodes can be considered an original Dijkstra.

The preprocessing has to be adapted to support personalization. This will be discussed in Section 2.2.1. The CH-Dijkstra has to be extended to PCH-Dijkstra by calculating the scalar product instead of using a single entry of the costs. The rest of CH-Dijkstra stays the same.

2.2.1 Personalizable Contraction Hierarchies

The contraction for multi-criteria is different from the original CH. PCH was introduced first by Funke and Storandt [15] with additional improvements [2, 14] using Linear Programming (LP).

For contracting a single node we consider the example of nodes $u - v - w$ where v is contracted. The rank of v would be lower than the rank of u and w. The calculation for one node is made for each in-edge and out-edge pair, which in our case is a single pair. Contracting a node with more neighbors is computational more expensive because there are more pairs to evaluate. Each of the incoming and outgoing pairs of edges has to be evaluated. To contract the node v an LP is created with the constraint $\sum_{i=1}^{d} \alpha_i = 1$ for all $\alpha_i \geq 0$.

We first check if there exists an α so that the path is optimal. The α can be computed by solving the LP.

$$\alpha^T (c(p) - c(p^*)) \leq 0 \tag{2.2}$$

If there exists an α for which the path p and the costs $c(p) * \alpha$ might be optimal, we test whether Dijkstra$(u, w, \alpha) \Rightarrow p^*$ has equal costs to p. If it is equal to the costs of p, the shortcut of the edge-pair is needed. This tells us that there is an α for which the edge-pair of the node is optimal.

Therefore, we need to create the shortcut and are finished. If the costs are not equal to p, we found a path with fewer costs from u to w. This path is added as a new constraint Equation (2.2) to our LP and thereby minimizing our feasible region. This process is repeated until the region of our LP is infeasible and thereby not adding a shortcut. If the region is infeasible, there is no such α for which the edge-pair would be optimal.

An improvement was proposed by Barth et al. [2] to add an objective function to maximize the gap between the costs seen in Equation (2.3). This reduces the number of created shortcuts and thereby the preprocessing time by comparing fewer edge pairs. Additionally, by reducing the shortcuts, the query times are improved by a reported factor of two.

$$\begin{aligned} &\max \delta \\ &\alpha^T \left(c(p) - c(p^*)\right) + \delta \leq 0 \end{aligned} \tag{2.3}$$

2.3 Customizable Route Planning

CRP [7] is an arc-separator-based approach to speed up Dijkstra. The preprocessing is split up into two phases, of which the first phase is metric independent, and the second phase is metric dependent. The first phase splits the graph into multiple cells. Applying the separation a single time is called single-level-partitioning, and doing so multiple times produces a hierarchy called MLP. The different separations of an MLP are therefore called levels. For generating an MLP, multiple algorithms can be used; see Section 2.3.1. The generated MLP specifies the overlay graph, stored in a matrix, where all possible paths passing through the cell are stored. The goal of the MLP is to have the least possible amount of interconnected nodes, which results in small matrices. In one row of a matrix the costs from a single entry node to all possible exit nodes are stored. The second preprocessing phase is used to fill the matrices with the correct cost values. Multiple methods can be used to do so [7, 15]. Interestingly, using CH-preprocessing works too [9]. To create a graph with edges spanning the cell, all the inner nodes should be contracted except for the boundary nodes. These edges can be used to get the optimal costs for each possible path. For CRP, the intermediate shortcuts of CH are removed, as they are not needed for calculating the costs due to the algorithm's design. The query calls a CRP-Dijkstra, which supports unidirectional queries. The search space can be interpreted similar to Dijkstra, however specific edges are skipped, see Figure 2.3. For each node to visit, Dijkstra evaluates the `query_level`. To calculate the `query_level`, the minimum level with matching `cell-IDs` for both start and target is collected. The minimum of both is the resulting `query_level`. Taking the minimum of both can be interpreted as "is the start or target node closer". The minimum level with matching `cell-IDs` tells the CRP-Dijkstra to go down in the hierarchy to not walk shortcuts over the cell containing the target node. The `query_level` defines what level of edges can be used. `query_level` = 0 is the original graph, where all above are the generated overlay graphs in the matrices. When the target node has been reached, the paths of the overlay graph have to be resolved. In contrast to CH, the edges do not store how they were created. All the paths for each shortcut of each cell have to be recalculated. This can be done because the entry and exit node of each cell is known. Using Bidirectional Dijkstra for all cells in parallel is possible. When using MLP, this has to be done recursively by using each intermediate level. There exists a bidirectional variant of CRP-Dijkstra. Same as for Bidirectional Dijkstra, it starts two CRP-Dijkstra instances at the start and target. The termination of both Dijkstra instances is the same as for the Bidirectional

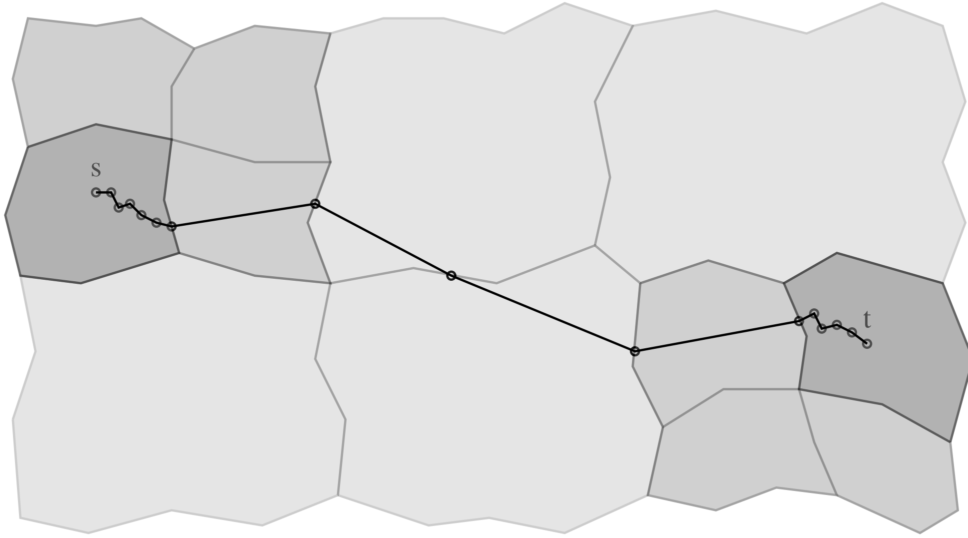


Figure 2.3: View of an CRP-Dijkstra. Darker cells indicate a lower level.

Dijkstra. The resulting path is a combination of all Bidirectional Dijkstra's in the correct order. To support personalization, the CRP has to be extended, which we call PCRP. We use the LP-oracle described in Section 2.2.1 as a second phase for this work.

2.3.1 Multi-Level-Partitioning

Partitioning a graph $G = (V, E)$ into many disjoint partitions is a fundamental problem in computer science. It is used in distributed computing, very-large-scale integration (VLSI) design, computer vision, image analysis, and route planning. The goal is to minimize a given objective function, subject to certain constraints. Most variants try to reduce the cut size, which corresponds to the number of edges linking vertices from different partitions.

Partitioning the graph into two partitions with a balanced amount of nodes and unit edge weights is NP-complete [16]. Getting an optimal solution for k partitions is not feasible and therefore approximating a solution is sufficient.

Partitioning the graph into k partitions is considered a single-level-partitioning. Let $P = \{C_1, ..., C_k\}$ be an ordered set of disjoint Cells $C_i \in V$ such that $\bigcup_{i=1}^{k} C_i = V$, meaning P is a partition of V. An MLP = $P^1, ..., P^L$ is an ordered family of partitions having L levels. $cell_l(v)$ provides the `cell-ID` of the node on level l. For MLP the following constraint has to be satisfied: $\forall u, v \in V, l \in 1, .., L-1, cell_l(u) = cell_l(v) \Rightarrow cell_{l+1}(u) = cell_{l+1}(v)$. This means if two nodes have a common partition on level l they both have to be in the same partition on the level above $(l + 1)$.

Older methods like the Kernighan Lin Algorithm [23] propose to solve this with matrices. The core idea is to start with random partitioning and improve the partitioning. Improving is done by exchanging the partition assignment of two nodes. However, one downside of this method is that the whole graph has to be stored in one big matrix. This is not feasible for continental road networks. Therefore other methods had to be developed, which work for road-network partitioning.

A prevalent method for partitioning is METIS [22]. This method is split into three phases. In the first phase, the graph is coarsed, so the size of the graph is reduced by summarizing multiple nodes into sets. In the second phase, partitioning takes place on the coarse graph by using matrices. In the last phase, the coarsed graph is uncoarsed back into its original size. Minor adjustments to the boundaries are made while uncoarsening the graph. The public implementation of Karypis and Kumar [22] supports MLP. This approach is not specialized for road networks. The most popular partitioning algorithm for road networks is Partitioning Using Natural Cut Heuristics (PUNCH) [8]. It uses the same approach as METIS [22] does but treating bridges, mountain passes, and ferries specially. Therefore, PUNCH is specialized on road networks.

A more recent, not very well evaluated approach is the Inertial Flow method [30]. It sorts the vertices by longitude or latitude (or some other linear combination). As an input, a balancing factor k has to be provided, which is by default $\frac{1}{4}$. It then computes the maximum flow from the first k nodes being the source to the last k nodes being the sink. The corresponding minimum cut is used as an edge separator. This process is applied recursively until the graph is separated into small partitions.

Another common way to get a solution for an NP-hard problem is to use heuristics to approximate partitioning [25]. Due to the complexity of these algorithms, the implementation of this work focused on more straightforward methods presented in the following.

K-means

K-means [24] is a popular clustering algorithm that does not support MLP. It operates on spatial data and does not take the road graph into account. The core algorithm is an iterative process, separating data points into k non-overlapping partitions. Only the parameter k has to be provided before the algorithm runs. It then initializes k points named centroids at random distinct points given by our graph data. Then the iterative process starts. All graph points are assigned to their nearest centroid point. After this, the centroids are moved to the mean of their assigned graph points. This process continues until the centroids do no longer move and the algorithm has converged. The separation of two cells drawn on a map is in a straight line. Each edge crossing this line has boundary nodes on both sides. For multiple separations, the centroids of K-means can be interpreted as points of a Voronoi diagram, where the separations can be extracted from.

Gonzalez

The Gonzalez algorithm [20] uses graph data and ignores metrics and spatial data. A random node is chosen in the beginning as a center. To determine how far away nodes are, a Dijkstra is started from this center. The most distant node then becomes the new center. The nodes adjacent to the new center then form one single cell. The algorithm continuously picks the node that is furthest away from the current sets of centers and lets this node be a center. If the wanted amount of k cells is reached, Gonzalez terminates.

Merge

Like the Gonzalez algorithm, this operates only on the graph data and ignores metrics and spatial data. However, this approach does not use a top-down approach. Instead, it uses a bottom-up approach. The core idea is that every node is a set and that highly connected sets get merged using a utility function (heuristic) until the wanted number of partitions or partition size is reached [13].

The heuristic used for merging two cells C_i and C_j is defined in Equation (2.4). $m_b(i, j)$ corresponds to the number of boundary edges between set C_i and set C_j. b_i and b_j are the number of boundary nodes of set C_i and set C_j. b_{ij} is the number of boundary nodes of the cell if both sets would be merged. For balancing the merging (and not merging only one set with all others), the number of nodes per set is also considered by n_i and n_j of set C_i and set C_j. To decide which merge comes first, the calculated value is multiplied by a random number of an Random Number Generator (RNG) between 1.0 and 1.01. To get satisfactory results, the algorithm has to be run multiple times with different seeds for the RNG. The result with the lowest sum of merged cells is the one being kept.

$$p(i, j) = \frac{m_B(i, j) * (1 + b_i + b_j - b_{ij})}{n_i * n_j} * \mathrm{RNG}(1.0, 1.01) \tag{2.4}$$

2.4 Personalizable Route Planning

However, if the MLP splits the graph into a small number of partitions, resulting in a high amount of nodes in those partitions, CRPs performance drops. The query speed up can be very low if s and t are not in the same cell and no boundaries are reached. Therefore no speedup will occur. The CRP-Dijkstra in on level zero behaves like the original Dijkstra. Having the right amount of partitions for CRP is crucial but not easy to determine. Funke and Storandt [15] proposed PRP, a combination of PCH (Section 2.2) and CRP (Section 2.3), to take advantage of both approaches. PCH-Query is used on the lowest level, thereby supporting to reach the borders faster. The CRP-Dijkstra continues after the boundary nodes are reached. CRP supports unidirectional Dijkstra, while CH does not. So a unidirectional PRP-Dijkstra would use the CH-edges to reach the borders fast, then switch to CRP-edges. When reaching the target cell, there is no way for Dijkstra to use the CH-edges. Therefore it has to rely on using the CRP-Dijkstra. This problem can be overcome by using a bidirectional approach to use the CH-edges from start and target. Both are expected to meet on a higher level.

3 Implementation

In this section, we provide a detailed description of the implementation process. Some details are implemented differently than the papers mentioned in Chapter 2 to provide a single implementation supporting: Dijkstra, Bidirectional Dijkstra, PCH-Dijkstra, PCRP-Dijkstra, and PRP-Dijkstra. Trade-offs had to be made, e.g., CRP-cells are not stored in matrices. The main goal is to provide an implementation to compare all the different Dijkstra variants. The code for this work can be found at the repository: https://github.com/Stunkymonkey/prp.

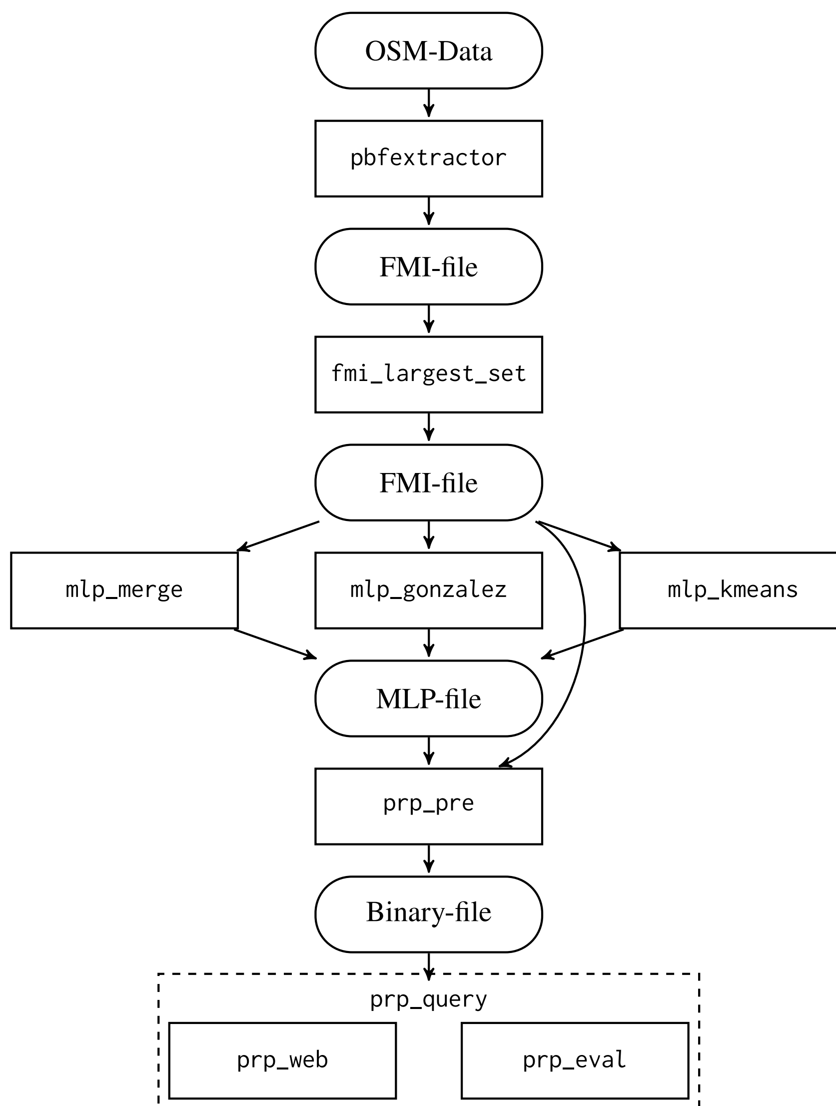


Figure 3.1: The implemented pipeline of the data.

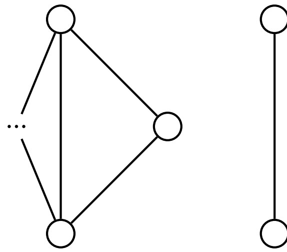


Figure 3.2: Example why the largest-connected-component is extracted.

Figure 3.1 depicts the data pipeline we used in this work. This section follows the pipeline and explains all the details of each component. The OpenStreetMap (OSM)-data is processed with the `pbfextractor` implemented by the supervisor. It supports two edge filters by which only car or bicycle edges can be exported. In addition, the metrics to be used must be specified. The code supports various amounts of metrics. However, the number and combination of metrics affect the time required for preprocessing, see Section 4.2.

The exported graph often has nodes or sets of isolated nodes that cannot be reached by nodes of the largest set of connected nodes. This will later affect the partitioning since a set of nodes that are not connected would always form its own partition. As stated before, the partition algorithms must be provided an k as a parameter specifying "how many partitions" are created. Creating partitions can be done by using spatial information, graph data, or a combination of both. When using only the graph data, sets of nodes, which are not connected to the largest connected component, form partitions of their own. This produces results that differ from the expected MLPs. Only the largest connected component is preserved, and all other nodes and edges are removed to prevent this. In Figure 3.2, it can be seen that the resulting partitions would produce unbalanced sizes but with a good cut size. For this example, it is best for the cut size to have the nodes on the right side form their own partition. Since the partitions are created based only on graph data, it is not easy to figure out which other nodes they should be joined to. Using the largest connected component, the parameter k creates balanced and connected partitions. This is implemented with the program `fmi_largest_set`, which converts the graph into an undirected graph and performs a Breadth-First Search (BFS). It iterates through all nodes and keeps the largest connected component stored. Using a non-recursive BFS is slightly faster than a recursive one. Using an undirected graph is not a problem because all our following MLP-methods will operate on an undirected graph or use spatial data. The export format of this program is the same as the `pbfextractor`, as it reduces the number of nodes and edges. Next up, an MLP is generated from the graph. Three different methods have been implemented: K-means, Gonzales, and Merge. Due to previous work with these approaches, We expect promising results from these simple algorithms.

3.1 Multi-Level-Partitioning

We designed a new file format to be able to store the MLP results efficiently. Our goal with this file format was to provide a generalized yet straightforward format, such that future work can also benefit from this when implementing different/novel/other partitioning algorithms. Therefore a similar format like the `fmi-file` was chosen.

Listing 3.1 First lines of an `mlp-file` example

```
3
20
13
226
384722
9381
9381
9381
9394
9394
9394
...
```

The file is structured as follows (Listing 3.1 depicts an example file with 3 levels): The first line contains the number of levels the MLP was created with, which specifies the amount of the following lines. These lines define the number of partitions and sub-partitions. It is important to note that the order is counterintuitive from bottom to top partitions. After these lines, the number of nodes is defined, followed by the number of nodes where each node is assigned a cell number. These numbers are the MLP-IDs used to calculate at which level and in which partition the nodes are.

The total amount of bottom partitions can easily be calculated by multiplying all the partitions together $20 * 13 * 226 = 58,760$. To get the `cell-ID` of the level l, use the Equation (3.1).

$$cell_{level}(node_partition_id) = \left\lfloor \frac{node_partition_id}{\prod_{i=0}^{level} mlp_levels[i]} \right\rfloor \tag{3.1}$$

When requesting $cell_0(node_partition_id)$, the product of nothing is its multiplicative identity which corresponds to 1, so the resulting number of partitions is the stored number. For example the first node from Listing 3.1 is on level 0 in the $9381/1 = 9381$th partition. On level 1 it is in the $9381/20 = \lfloor 469.05 \rfloor = 469$th partition. For level 2 it is in the $9381/(20 * 13) = \lfloor 36.08 \rfloor = 36$th partition. It is important to note that partitions can be empty, and no node is assigned to them. Aside from that, the `cell-ID`s do not represent any spatial arrangement. Meaning two partitions with `cell-ID` 1 and `cell-ID` 2 do not necessarily have to be beside each other.

To have a better visual understanding of the different MLP-algorithms, we generated images for visual inspection. The used MLP parameters are $3, 4, 5$ for K-Means and Gonzalez and $3, 12, 60$ for Merge. These images only show the nodes connected to a neighbor of a different cell, often called boundary nodes or border nodes. We provide further investigation in Section 4.2.1.

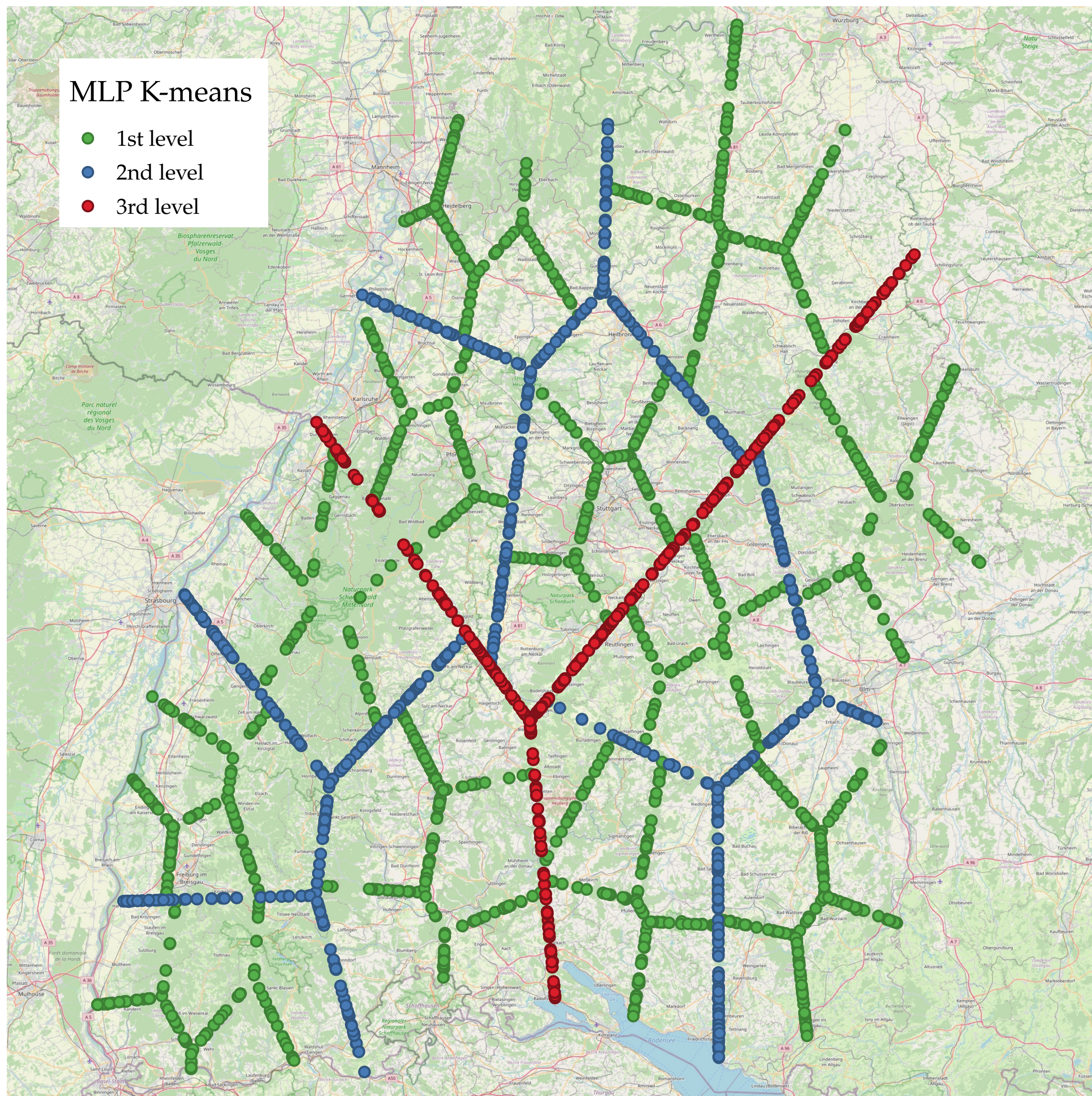


Figure 3.3: Example of MLP K-means showing all boundary nodes with their corresponding levels.

3.1.1 K-means

We chose to select the initial centroids with a seeded RNG instead of choosing them randomly. With this, our implementation yields deterministic results. The implementation uses a library[1] for the K-means algorithm and provides a wrapper for the MLP generation. For generating an MLP, the implementation makes a top-down approach by using all nodes from the graph for the topmost level. For the next level, only the nodes of a single cell are getting used for the generation process. To improve the generated MLP, the algorithm can run multiple times with different RNG-seeds and

[1] https://crates.io/crates/rkm

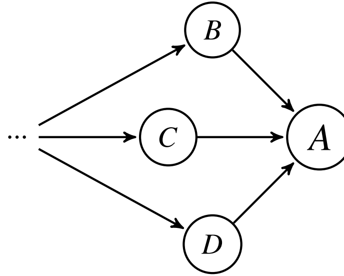


Figure 3.4: Example why Gonzalez can not operate on a directed graph.

only keep the one with the least number of boundary nodes. Only considering a mini-batch of nodes for recalculating the centroid is faster [27]. These improvements have to be further investigated but are not the primary goal of this work. The generated MLP can be seen in Figure 3.3.

3.1.2 Gonzalez

Initially, the graph is converted into an undirected graph. Without bidirectional edges, the resulting MLP could produce unwanted results. Figure 3.4 shows an example of this: If the next BFS started at node A, the neighboring nodes would not receive an update in hop-distance because they are unreachable. Therefore, not a single node of the entire graph except the node itself is updated by adding a zero (which can be interpreted as no updates at all). The next BFS-run could choose the node again. Using the same node twice as the source for two partitions would produce one empty partition and different results than parameter k. The MLP format would allow this but does not match the input parameters given.

Because the first phase of CRP should be metric independent, we chose to use the hop metric. When using the hop metric combined with Dijkstra, the result is the same as using a BFS. Therefore, we used a BFS implementation, which the main difference is the usage of a ring buffer (monotone increasingly distance) instead of a heap (with sorting). In the original implementation, if more than two nodes have the same sum of distances to the other set of centers, the new starting position is chosen randomly. Same as for K-means, a seeded RNG performs the random selection to have deterministic outputs. The implementation of the MLP structure is in line with our implementation of MLP for K-means, a top-down approach where the number of partitions must be specified.

While implementing, we discovered that if the number of partitions should be low, the partitions can be unbalanced if the initial Dijkstra starts at the center of all nodes. For example, if two partitions are requested, the center partition would have more nodes than the outer one. To prevent this, a single Dijkstra is run without assigning any nodes, thereby forcing the next Dijkstra to start at a node located at the edge of the graph. This produces a more balanced MLP for a low number of partitions. The resulting MLP of Gonzalez can be seen in Figure 3.5.

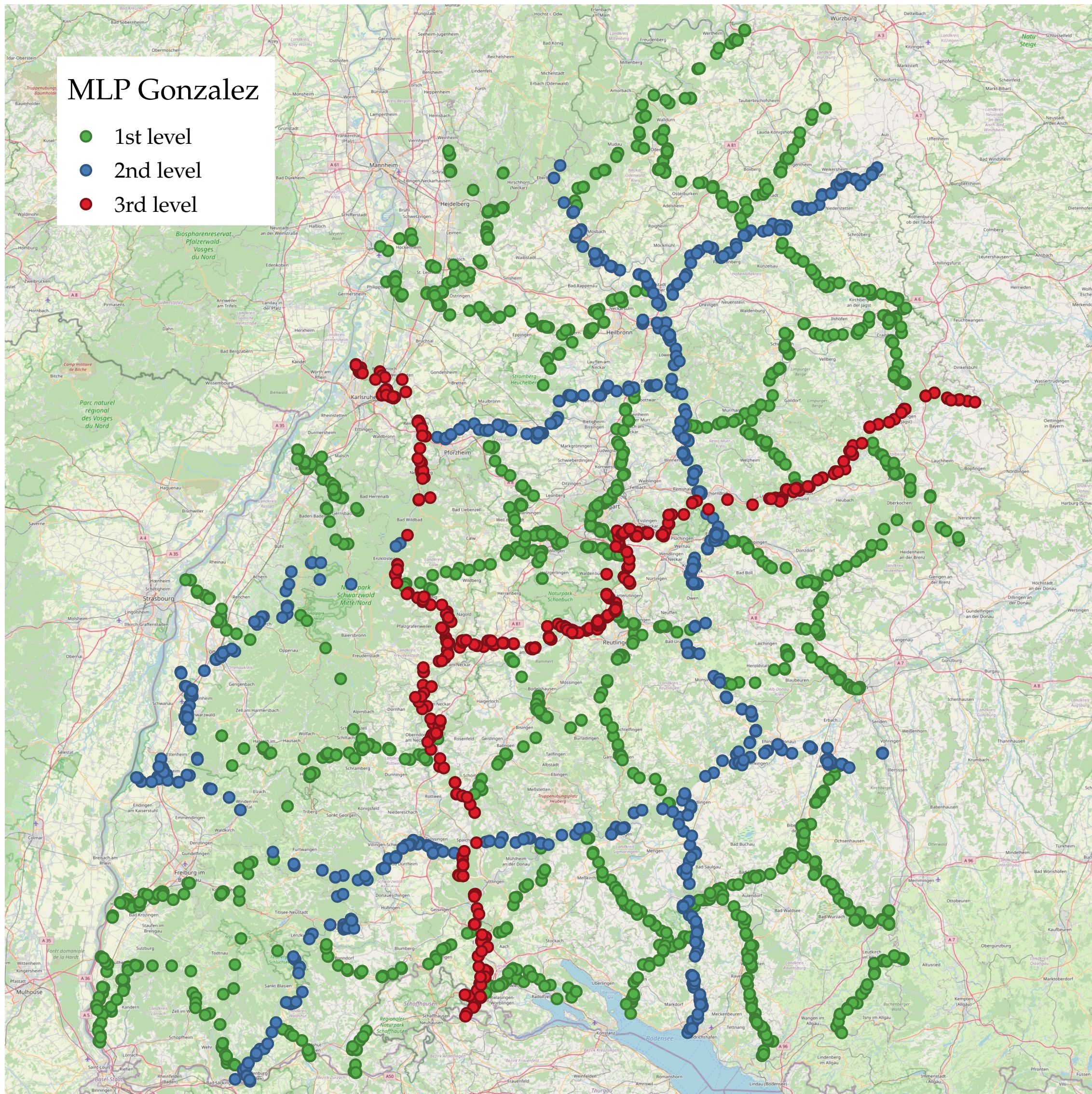


Figure 3.5: Example of MLP Gonzalez showing all boundary nodes with their corresponding levels.

3.1.3 Merge

We used an undirected graph that additionally verifies that only one edge exists between each pair of nodes. Equation (2.4) was used, but counting edges and nodes is done differently than proposed due to our own undirected graph structure. For example, counting the edges between both partitions can be done by iterating over all edges of nodes of one set and counting all the nodes in the other set. Due to the undirected graph, it is sufficient to count the edges between both sets.

The original version does not produce an MLP, but the implementation for this work provides it. To support generating an MLP, all sets are saved if a certain condition is met. The merging continues until the last termination condition is fulfilled.

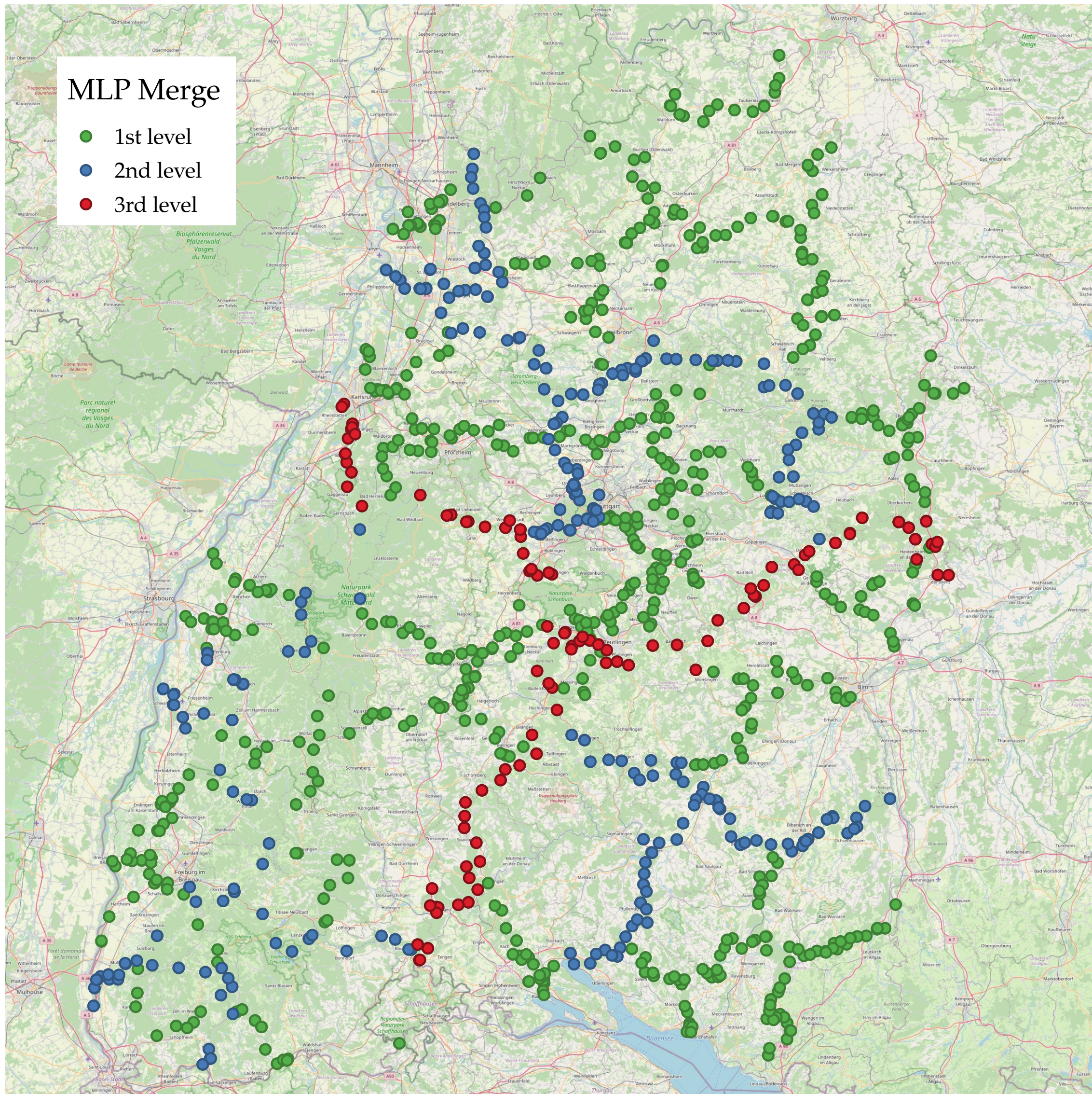


Figure 3.6: Example of MLP Merge showing all boundary nodes with their corresponding levels.

The official publication [13] proposes running the algorithm multiple times with different seeds for the RNG. Our implementation is run a single time for every parameter to keep the preprocessing time small.

The merging can be unbalanced in terms of cell sizes. Due to the bottom-up approach, it has only a local view and cannot act globally. Therefore, the algorithm could produce an unbalanced amount of partitions for multiple levels. By doing so, some cells in the MLP file are going to be empty. The example MLP can be seen in Figure 3.6, where the bottom right partition only contains three sub-cells instead of four like the other MLPs do. For Merge, the MLP format is not ideal because some partitions are empty, but this is due to the algorithm's heuristic. Improving the heuristic is not part of this work and has to be further investigated.

3.2 Preprocessing

The supervisor provided the implementation for the LP-oracle. The code was not customized to fit our particular graph structure Instead, the Dijkstra to find better paths is an anonymous function being called in our implementation. Delling and Werneck [9] proposed to use CH for CRP. Therefore, PCH preprocessing can be used for generating PCRP-edges. This can be done by fixating the boundary nodes and contracting all the inner nodes. The resulting graph spans the cell of which the boundary nodes are each cell's entry and exit nodes.

First, all data from the `fmi-file` is read. Whether the contraction has to stop after contracting a certain percentage of nodes or whether an MLP should be used must be specified. If a percentage is given, it can be interpreted as all nodes are in a single cell without any hierarchy. Next up, the graph is processed to support fast access of each node's incoming and outgoing edges. The outgoing edges are accessed with an offset array. The incoming edges have an additional offset array combined with an index array. Thereby storing the edges only a single time and providing fast access times for outgoing and a slightly slower access time for incoming edges. Then the `query_level` of all node neighbors is calculated for each node. This means that the `cell-ID` is compared on level 0. If they are equal, the result is 0 because they are in the same cell. Otherwise, it would compare the `cell-ID` on level 1 and so on until a common level is found. If no common level is found, then the resulting index is #level + 1. The comparison has to be done on all neighbors with incoming and outgoing edges. This result is used to determine which nodes have to be contracted. All nodes with the `query_level` of 0 are fully contracted, and then the following levels $1, \ldots, L$ are processed until the last level. The nodes with the `query_level` of #level + 1 are not contracted. They are the core-nodes known from the CH-contraction.

While contracting, there are two graphs in memory. There is a shortcut-graph containing all the not yet contracted nodes and shortcuts and a resulting graph containing all edges and shortcuts to resolve the shortcuts of the other graph. The contraction itself first calculates the node-ordering heuristic by `in_degree(node)` $*$ `out_degree(node)`, which is an upper bound for the number of shortcuts that could be created. Other implementations add the `amount_of_contracted_neighbors` to provide a more balanced way of contraction. In our case, this is already provided by the MLP. Without using `amount_of_contracted_neighbors`, it produces fewer shortcuts, even though the same nodes are contracted. For parallel contraction of nodes, a greedily independent set is calculated. This independent set is created from the lower $\frac{1}{10}$ (rounded up) of heuristics values. This is used until only 10 nodes are left for contraction. By rounding up, the number of nodes close to 10 gets a bit

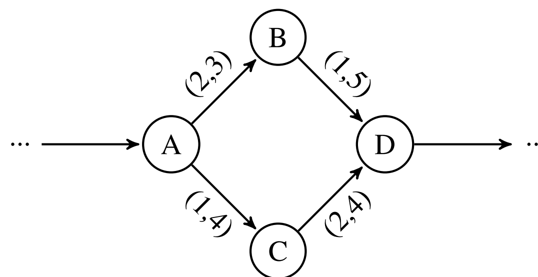


Figure 3.7: When contracting node B and C in parallel, shortcuts with identical costs $(3, 8)$ can be created.

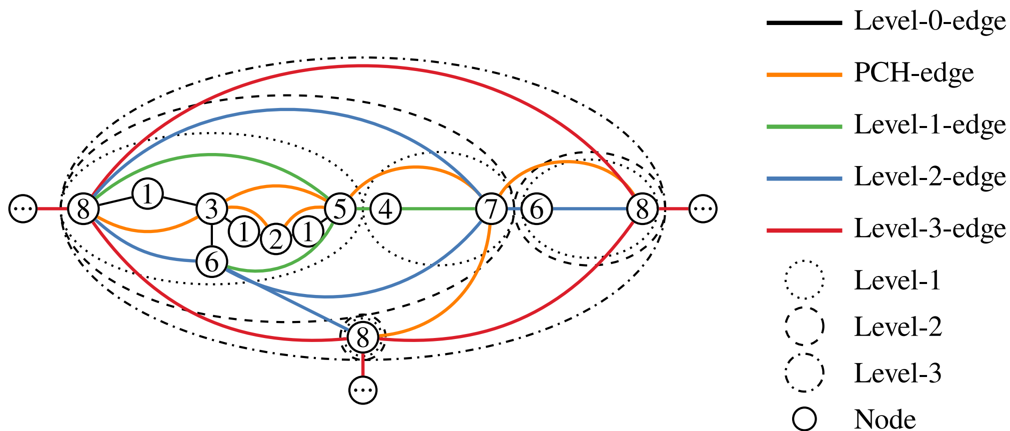


Figure 3.8: Excerpt how the internal graph structure looks after the contraction. For simplification, the graph in this example is treated as undirected, but the edges are all directed in the implementation. This graph supports queries of PCH-, PCRP-, and PRP-Dijkstra. The numbers inside the nodes represent the rank, of which the contraction order can be reconstructed. The maximum rank is 8, after which the contraction stops because no more nodes can be contracted.

bigger while not affecting the contraction in the beginning. By using a fraction of the heuristic, it will be a smaller independent set. To have a small independent set increases the contraction time because the maximum rank increases, and therefore the contraction follows the heuristic more optimally than by using $\frac{1}{2}$. The main advantage is that the number of edges created is decreased, thereby improving the query time.

After the independent set has been created, all nodes are collected, and all possible pairs of edges are checked. A single producer and multiple consumers system is used. The consumers have thread-local Dijkstras spawned combined with an LP-Oracle to check the given pairs. The LP-Oracle is then asked if the shortcut has to be created. Each shortcut stores the edge-ids of which it was created. The generated shortcuts are collected thread-local. This is done until there are no more pairs of edges. Then the resulting shortcuts of each thread are collected and combined. It could be argued that generating an array containing all the possible edge pairs and slicing them for each thread is faster. The problem is that if a single node has, e.g., 100 incoming edges and 100 outgoing edges, this will result in 10,000 combination pairs. Storing these pairs in an array is impracticable if the graph is dense/contracted very high because the array would be extensive. It could be argued that assigning each thread an equal amount of nodes is faster because the same start node can be used, and thus the costs could be cached. This is true for classical CH, but for PCH, the alphas are changed more often than the start node and thus needing complete recalculation without the possibility of caching the costs. Additionally, it turned out, if the graph is *dense*, the amount of time needed for contraction is very different for each node. When looking at Figure 3.7, it can be seen that contracting in B and C parallel can produce multiple shortcuts with identical costs from the same node to another same node. Therefore the shortcuts are sorted lexicographically and deduplicated before being stored in the shortcut-graph.

(Bi-/)Dijkstra		PCH		PCRP		PRP	
level 0	None	level 2	(982,754)	level 2	(982,754)	level 2	(982,754)
level 0	None	None	(583,491)	level 1	(301,54)	level 1	(301,54)
None	(231,83)	None	(462,605)	level 0	None	None	(231,91)
None	(231,91)	level 1	(301,54)	level 0	None	None	(231,83)
level 1	(301,54)	None	(231,91)	None	(583,491)	level 0	None
None	(583,491)	None	(231,83)	None	(462,605)	level 0	None
None	(462,605)	level 0	None	None	(231,91)	None	(583,491)
level 2	(982,754)	level 0	None	None	(231,83)	None	(462,605)

Table 3.1: Example of the edge-sorting of a single node for all outgoing edges. The same is applied to the incoming edges. The first column represents the assigned level and the second the contracted edges it is created from. A higher edge-id in this example implicates a higher rank.

Next up, the used edges for creating shortcuts are all moved to the resulting graph, and the shortcuts are added to the shortcut-graph. This is repeated until all nodes are contracted, or the percentage of the remaining nodes exceeds the given parameter. Only the topmost MLP-level is allowed not to be fully contracted, but the lower levels have to be fully contracted. Every MLP defines an upper bound for contraction. Even an MLP with two cells having a single separation has at least a fixed number of boundary nodes. For each MLP-level, the edges in the shortcut-graph are all assigned a level. All intermediate shortcuts that are used do not have this level-identifier. These level identifiers are later used for CRP. When the contraction is finished, both graphs are merged into a single graph. The nodes are ordered by rank because having an overlay graph is beneficial to forward iteration over an array. The edges are reordered for each query algorithm.

In this data structure, edges allowing to switch from one partition to another are edges from the original graph without modification except for an assigned level. This can be seen in Figure 3.8. Edges that span across an entire cell are always PCH-edges with an assigned level. From CRP for each node, the MLP-ID has to be saved and the level for each edge. In the original CRP implementation, the MLP-ID is converted into the `cell-ID` of each level. This is not done for our implementation because extra space is needed, and the cost for calculating the `cell-ID` again is expected to be negligible. For CH each node, the rank is needed, and each shortcut-edge has to store which edges they were created of.

While implementing the preprocessing, we found that using an MLP for fixing specific nodes allows for an easier debugging of newly developed CH/PCH contraction methods. If the contraction method is faulty, this results in long contraction times, or many shortcuts are created. By having the entry and exit nodes fixated, the cells with the specific problem can be tracked down to analyze the problem further. A smaller graph with the same problem has faster contraction times and an easier understanding of the problem.

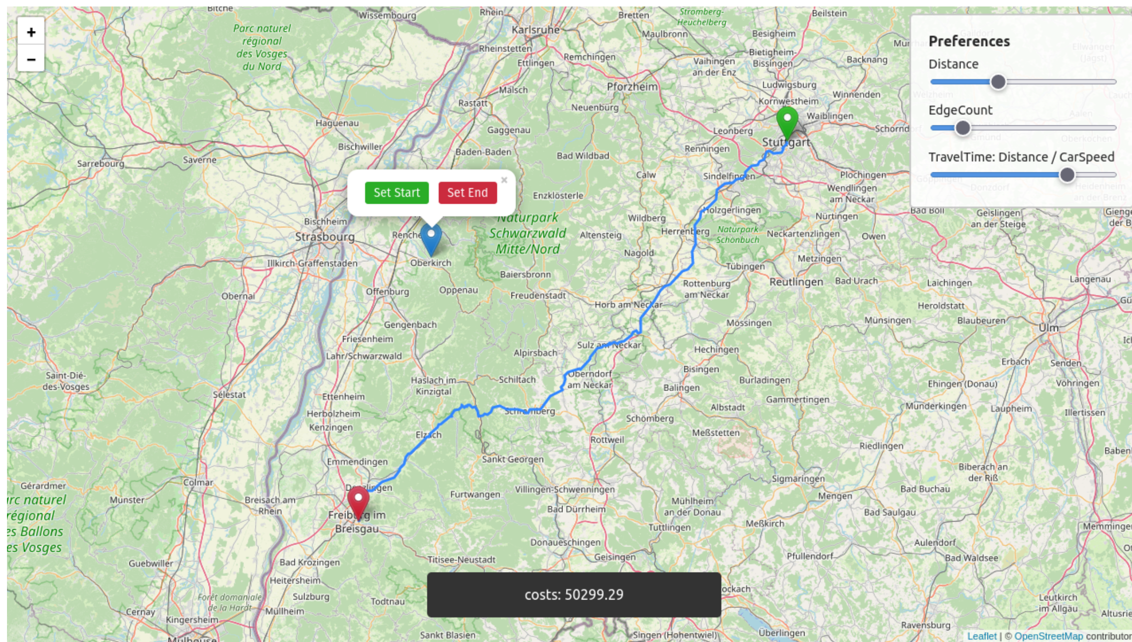


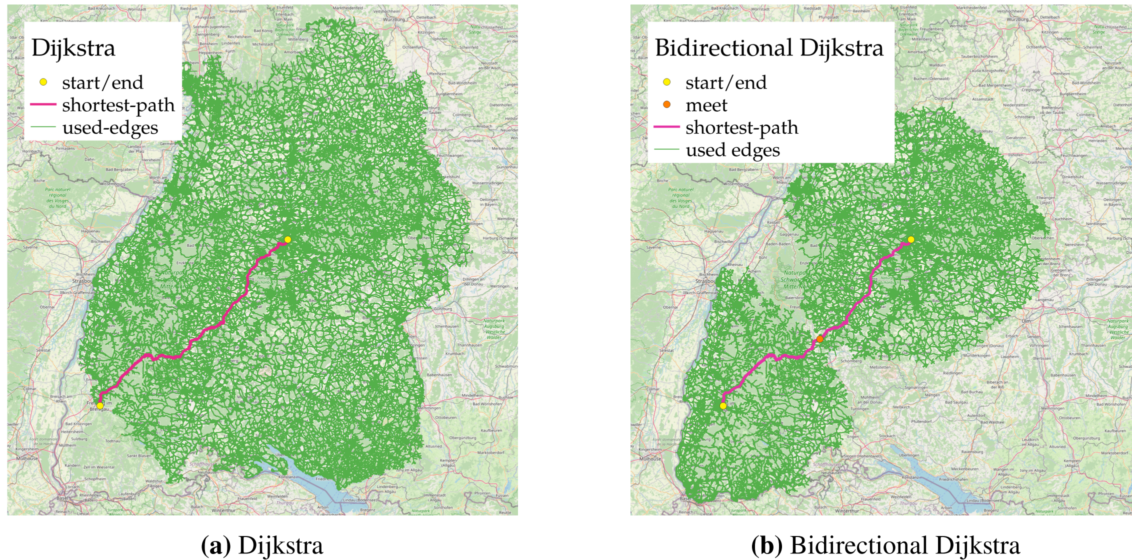
Figure 3.9: The web-client showing a personalized route from "Stuttgart" to "Freiburg im Breisgau".

3.3 Queries

When loading the graph from the `binary-file`, the query has to be specified. Based on the query algorithm, the edges are sorted before any query execution takes place. When iterating over the edges of one node, it is advantageous to have the edges sorted. Each query algorithm requires different edges and can skip certain edges. When the edges are sorted, the algorithms can terminate an iteration earlier if the sorting makes sure that the following edges are never needed. An example of edge sorting can be seen in Table 3.1, but it will be explained for each Dijkstra separately. Generally, it can be said that edges, which would never be considered, are moved to the bottom. With this, each Dijkstra can skip the iteration if the first edge would never be considered. The following five query algorithms are supported: original Dijkstra, Bidirectional Dijkstra, PCH-Dijkstra, PCRP-Dijkstra, and PRP-Dijkstra. Based on the example query shown in Figure 3.9, presenting the web-client, images have been created showing the exploration of each algorithm.

The original Dijkstra was developed first to have a base algorithm for correctness checks. Next up, the bidirectional was developed because it would be needed for PCH, PRP, and optionally by PCRP. Visual exploration of Dijkstra and Bidirectional Dijkstra can be seen in Figure 3.10a and respectively Figure 3.10b. It has been decided to do no alternation between the forward- and backward-Dijkstra for all bidirectional variants. Instead, the Dijkstra with the next lowest costs from its heap is taken. The edge-sorting for Dijkstra and Bidirectional Dijkstra is the same. All edges that were in the original graph and are no shortcuts are sorted to the front.

The PCH was implemented, and it is the only algorithm that could use all edges. The edges are reversed sorted based on the rank of the node it reaches when using the edge. When iterating the edges, the one connecting to the highest rank will be iterated first. The exploration can be seen in Figure 3.11. The particular termination condition has the disadvantage of exploring many nodes at



(a) Dijkstra

(b) Bidirectional Dijkstra

Figure 3.10: Example query for Dijkstra and Bidirectional Dijkstra.

the topmost level. This can be seen at the bottom of the image. The PCH-Dijkstra compares the rank of the node currently extracted from the heap and the node the edge is pointing to. If a node with a rank lower than the node itself exists, it can terminate the iteration because all following nodes will have a lower rank. The proposed speedup technique stall-on-demand for CH does slow down the PCH-query and is thereby not used.

The bidirectional variant of PCRP-Dijkstra was implemented by only using the level-identifiers of the edges combined with the partition-identifiers. Therefore, the edges are reverse sorted by their assigned levels. For every visited node, the `lowest_equal_level` has to be computed with the start and target. The minimum of both is taken and determines the `query_level`. The `query_level` defines which edges are allowed to be taken. The exploration can be seen in Figure 3.12. On the lowest level, the CRP-Dijkstra is the same as Dijkstra. When the boundary nodes are reached, the speedup occurs. For switching partitions, edges from the original graph (never shortcuts) with the correct `query_level` can be used. For CRP, the paper [7] proposed that edges of a higher level than the `query_level` should be ignored because the spanning edges would never be used. In this implementation, the exclusion was tested but checking if the edge is spanning the partition was significantly slower than including them. Originally CRP stores the edges for every level in a separate matrix where it can easily be extracted from. It can be interpreted that the edges without assigned levels are never used. For finding the costs, this is true, but for restoring the path of its upper edges, all edges are still needed.

PRP was implemented by combining PCH and PCRP. To implement this, two heaps for PCH and two heaps for PCRP are used. On the lowest level, it does the same as the PCH-Dijkstra. This can be seen in Figure 3.13. When reaching the boundary nodes, the Dijkstra switches to the PCRP-Variant. The sorting is not based on a single value for this algorithm, as before for the others. All edges are used except the intermediate PCH-edge without an assigned level of level one and above. For the PRP-Dijkstra, it was tested if extracting the lowest value of all four heaps is faster, but it turned out that running PCH-Dijkstra until both heaps are empty and switching to PCRP is faster.

PCH, PCRP and PRP use different Dijkstras for getting the costs. Nevertheless, the path resolution is made the same for all of them by using the PCH-path resolutions. In comparison, the CRP paper [7] proposed running local Bidirectional Dijkstras for resolving the paths. This has to be tested further on how big the performance draw-back is.

Two interfaces have been implemented, of which one is an asynchronous web-server that shares the graph data along with the threads, but on every thread, there is a reserved distance array for maximum performance. The web server communicates via `GeoJSON` [5] which, is a versatile format for encoding geographic data structures. The second interface is a simple command-line tool for evaluation.

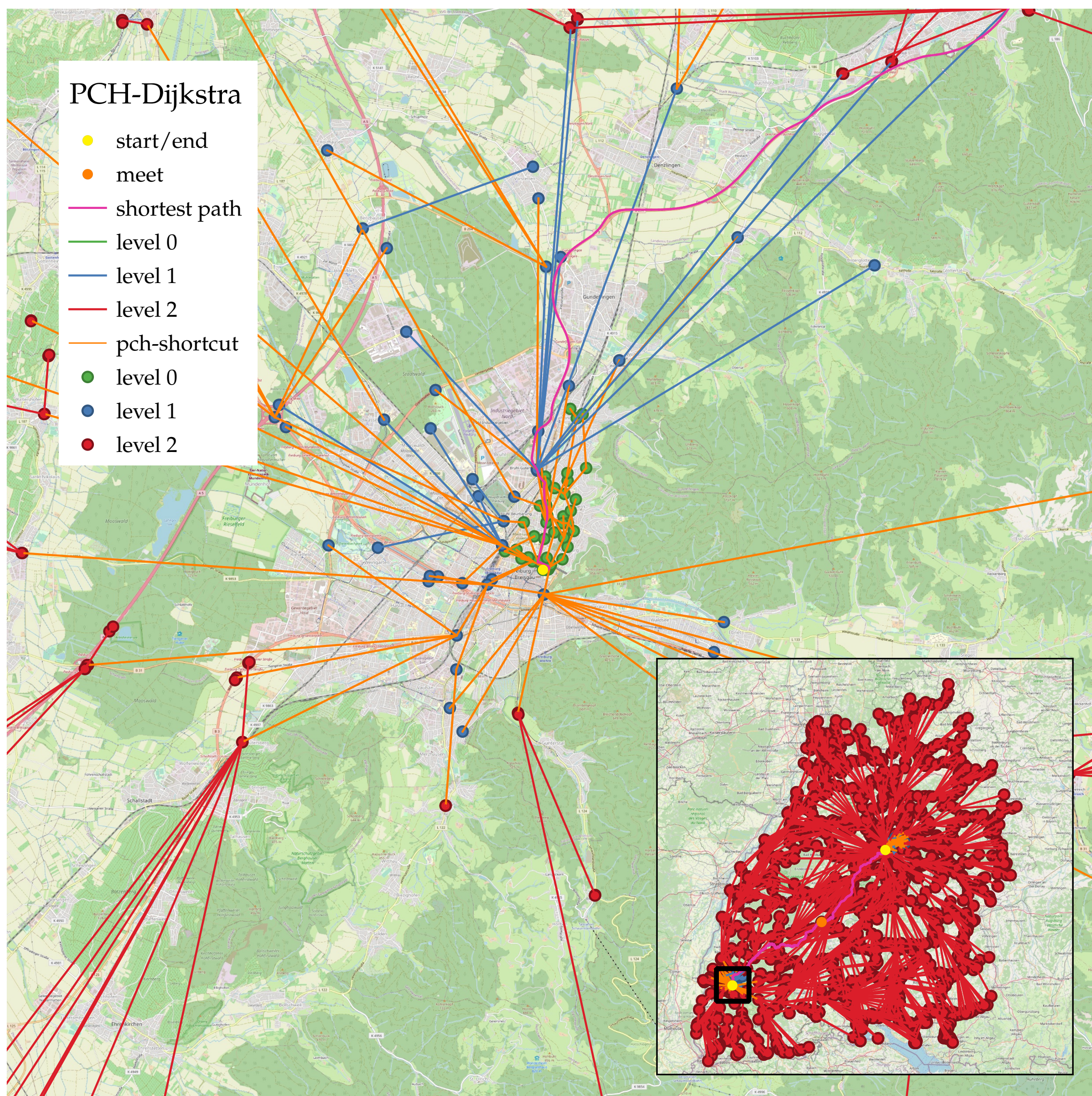


Figure 3.11: PCH-Dijkstra.

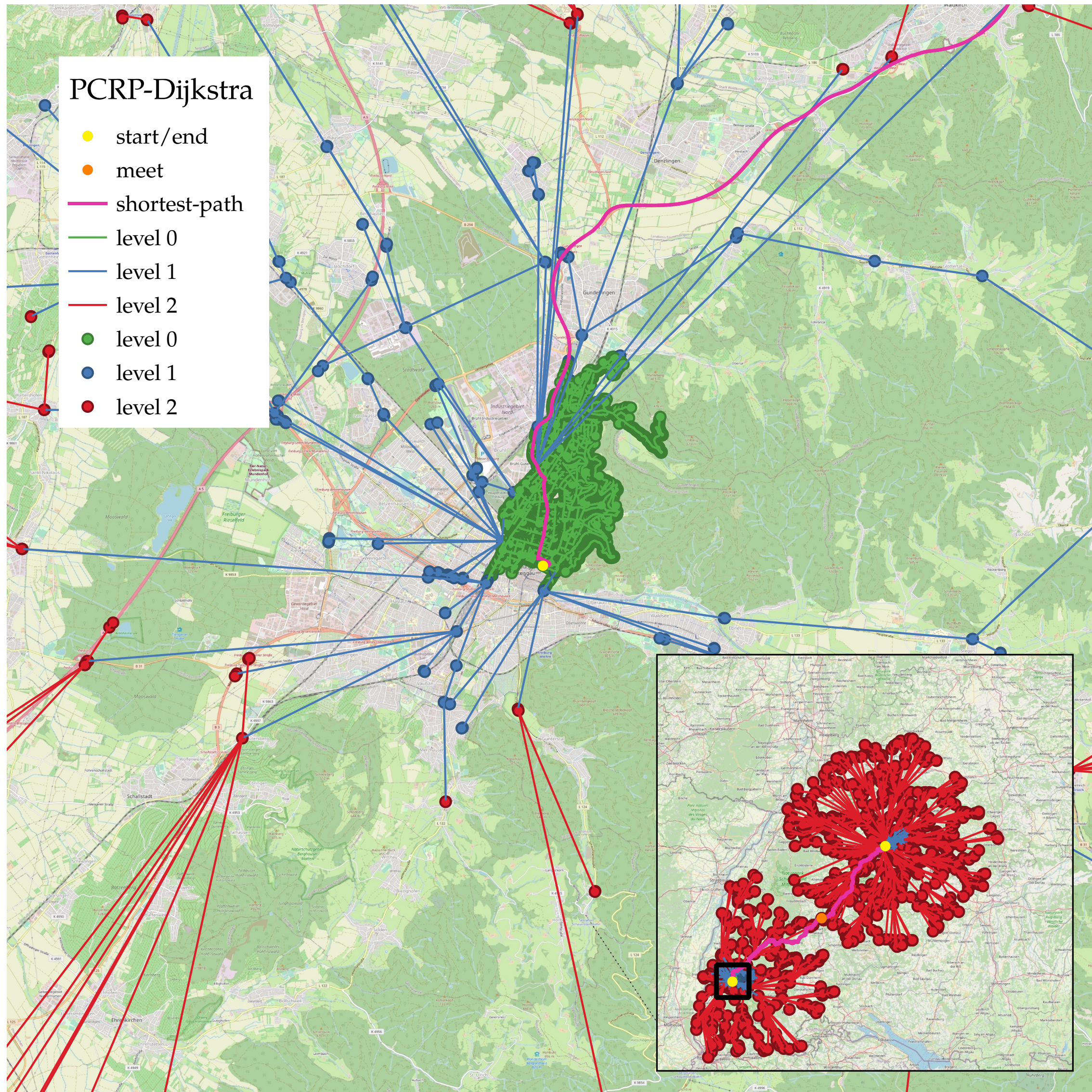


Figure 3.12: PCRP-Dijkstra.

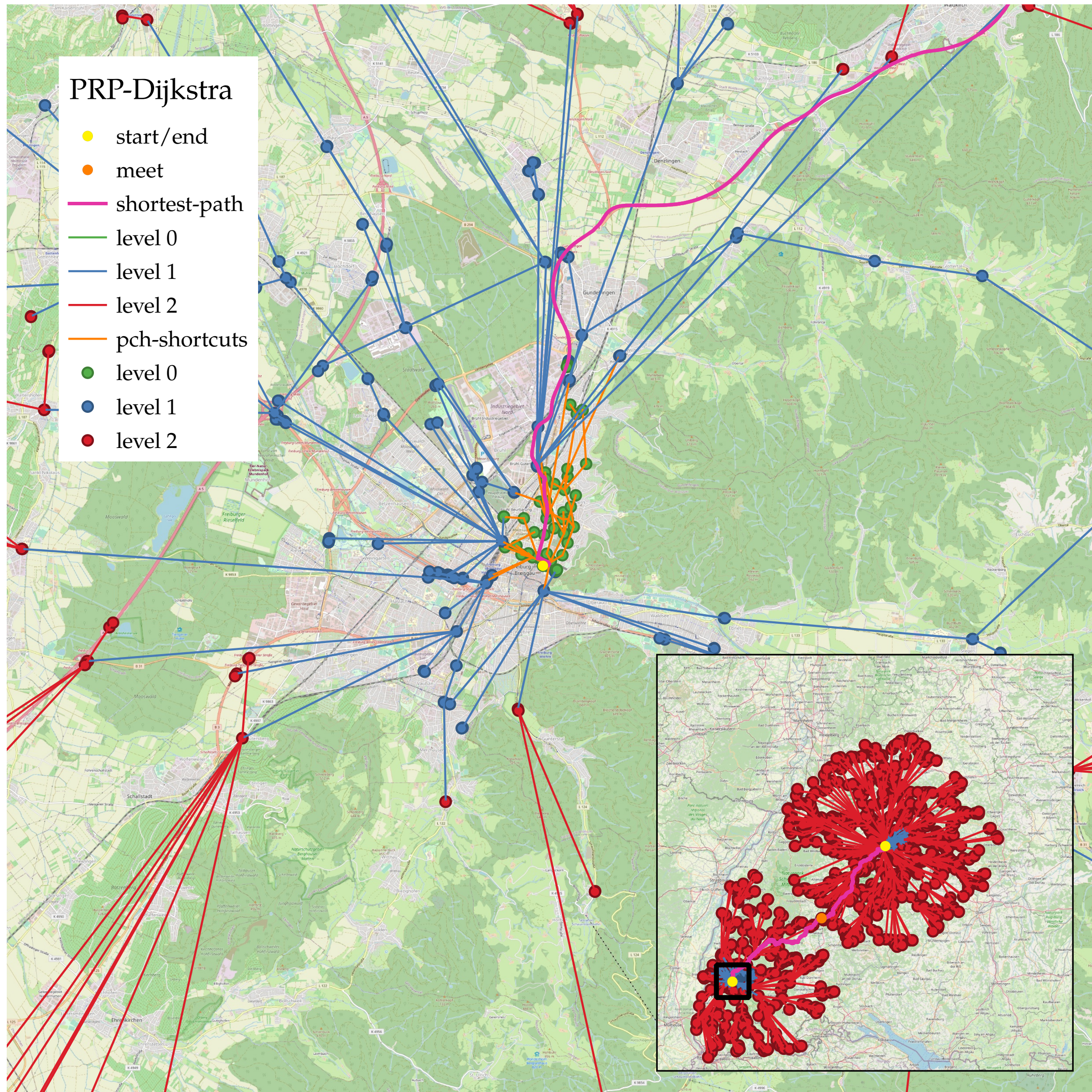


Figure 3.13: PRP-Dijkstra.

4 Experimental Evaluation

For our experiment, a virtual machine on a dedicated computer was provided. The virtual machine of an Intel(R) Core(TM) i7-2700K CPU @ 3.50 GHz CPU had 6 out of the 8 cores assigned. A total of 30 GB of RAM and a Samsung SSD 850 were installed. Ubuntu 20.04 was used with `GLPK` version 4.65-2 for solving LPs and `rustc` with version 1.53.0 for compilation of our implementation.

4.1 Data and Settings

The graph data used for our experiments were downloaded from OpenStreetMap [26] and NASA Shuttle Radar Topography Mission (SRTM) Version 3.0 Global 1 arc second [32]. The data was downloaded on 2021-08-01. It was decided to use the state "Baden-Württemberg" from Germany. The car graph consists of $4,357,213$ nodes and $8,757,596$ edges and the bicycle graph of $10,686,372$ nodes and $22,575,980$ edges.

Partition(s)	**1**	**512**	**1,024**	**2,048**	**4,098**	**8,196**	**16,384**	**32,768**
BW-car	4,357,213	8,510	4,255	2,128	1,064	532	266	133
BW-bicycle	10,686,372	20,872	10,436	5,218	2,609	1,304	652	326

Table 4.1: Average nodes per partition for the state "Baden-Württemberg".

The number of partitions is chosen by $2^i, i \in \{9, ..., 15\}$, producing from 512 up to $32,768$ partitions. The average nodes per partition can be seen in Table 4.1. By decreasing the number of partitions, the amount of boundary nodes is typically decreased as well. This results in higher preprocessing times because more nodes are contracted. All query-timings are averaged over 10,000 queries if not stated otherwise. The queries are taken from a stored JSON file from a disk to use the same queries for multiple graphs. This JSON file was created by randomly selecting nodes from the graph and storing their position in combination with a random α. The Dirichlet distribution was used to generate the α, which has the property of summing to one.

4.2 Experimental Results

In our experiments, Dijkstra is used as a baseline. All implemented Dijkstra-variants will be compared with the baseline, except the Bidirectional Dijkstra. The Bidirectional Dijkstra is well known and not the main focus of our work. The Dijkstra query times do not change by using different MLPs. Dijkstra needs only be calculated if the metrics change.

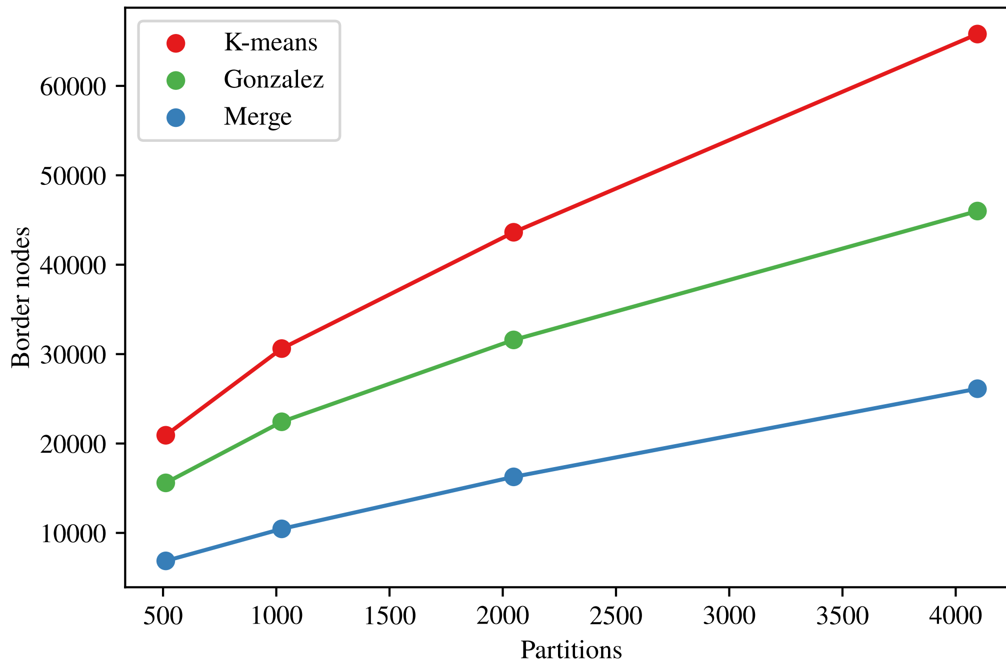


Figure 4.1: The number of boundary nodes determines how much nodes have to be contracted for each MLP.

4.2.1 Multi-Level-Partitioning

The MLPs are generated with a single level and will be evaluated for the car graph only. The generated MLPs are metric independent and are the same for the car graph with the same parameters, regardless of which metrics are used. The contraction, however, needs to be rerun if the metrics change. Run time, boundary nodes, and maximum partitions sizes are the most important criteria.

In Figure 4.1, the number of boundary nodes can be seen for each MLP algorithm of each number of partitions. It is depicted that the number of boundary nodes increases with the number of partitions. The Merge algorithm outperforms K-means and Gonzalez, of which K-means has the most boundary nodes. How well the partitions are balanced is shown in the following Figure 4.2. K-means has equal partitions, but Gonzalez has unbalanced partitions for a low amount of partitions. A combination of both properties can be seen in Figure 4.3. The depicted triangles use all the same parameters for each algorithm.

Regarding the number of boundary nodes, it is expected that K-means is not as good as the others because it relies on spatial data while the others rely on graph data. Instead, ignoring the graph data provides advantages in balanced partitions. From these figures, it can be seen that Merge is expected to perform consistently better than Gonzalez. K-means, in contrast, has more boundary nodes than the other algorithms but should result in steady query times. The higher number of boundary nodes can be advantageous because the number of nodes needed for contraction will be lower. Additional testing showed Merge to be more unbalanced than Gonzalez for a meager amount

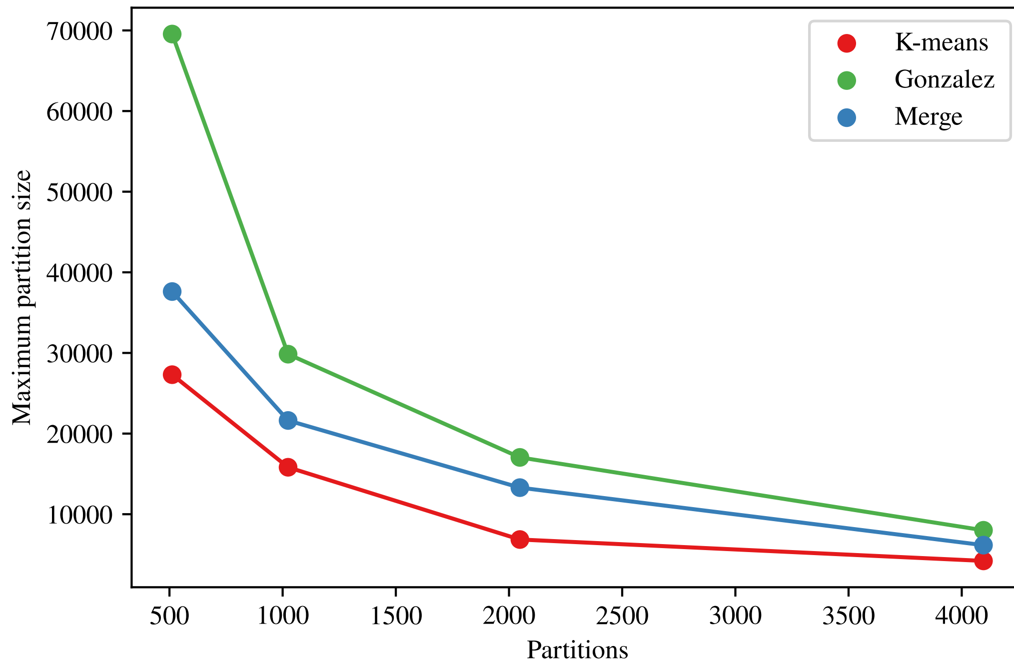


Figure 4.2: The maximum partition size shows how well the partitions are balanced for each MLP.

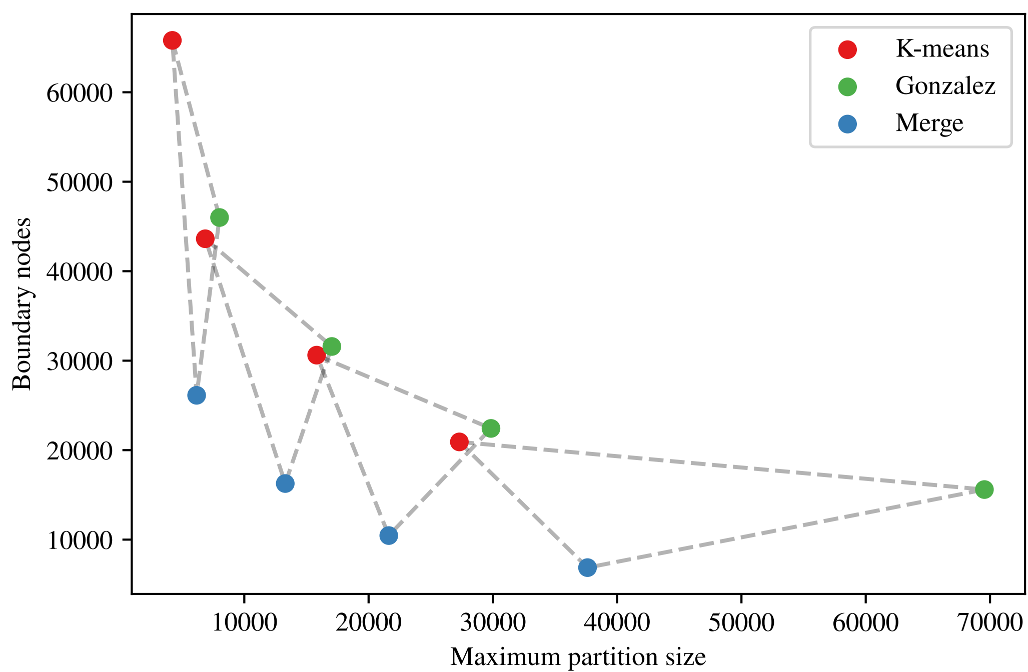


Figure 4.3: Comparison of maximum partition size and number of boundary nodes for each MLP

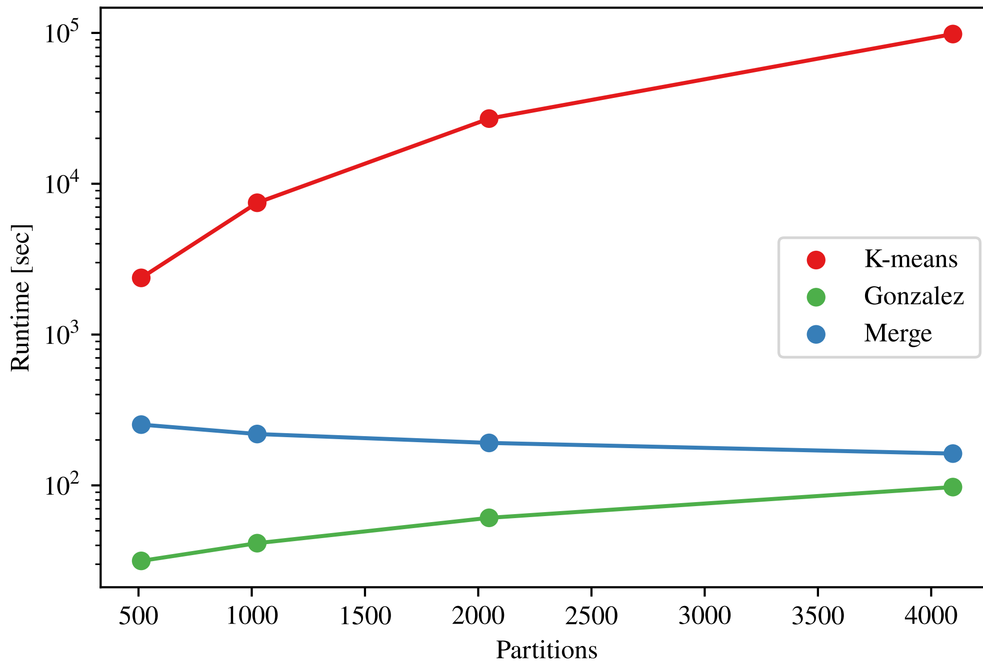


Figure 4.4: Run time of MLPs for each method.

of about ten partitions. Merge has the best properties based on the number of boundary nodes and maximum partition size. K-means and Gonzalez can be interpreted as equally good. These properties are highly dependent on the graph and should be seen as a general direction.

Next up, the runtime to generate the MLPs is investigated. It is important to note that the Y-scale is a logarithmic scale in Figure 4.4. K-means needs the most time for generating the MLP. For both K-means and Gonzalez, the time needed for generating the MLP increases by the number of partitions they have to create.

The runtime of Merge, on the other hand, is reduced due to the bottom-up approach. Long generation times are not bad for the real-world scenario because the MLP only needs recalculation if the graph network changes. For testing purposes, it was decided to use $2^{12} = 4,096$ partitions as the maximum for K-means with the car graph because it needs 27 hours and 18 minutes to generate the MLP. Running K-means for the bicycle graph would take even longer due to the higher number of nodes. For creating MLPs with more partitions, Merge and Gonzalez are better for evaluation.

4.2.2 Preprocessing

Next comes preprocessing with *distance*, *uniform cost* (each edge has a cost of one), and *car travel time* metrics. The most interesting criterion is the time needed for the contraction, as shown in Figure 4.5. The values drawn in Figure 4.5 can be seen in Table 4.2. Merge needs consistently less time for preprocessing than K-means and Gonzalez, even though the number of contracted nodes is

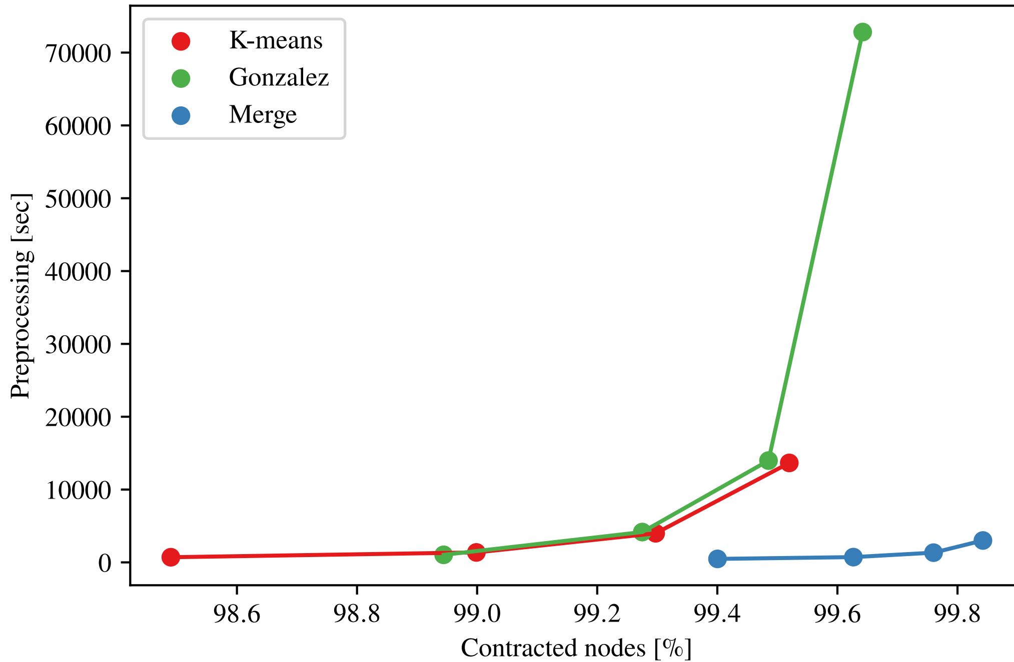


Figure 4.5: Preprocessing runtime with the same MLP parameter and different MLP-methods.

Partitions	K-means		Gonzalez		Merge	
	contracted	[min]	contracted	[min]	contracted	[min]
512	99.52 %	227	99.64 %	1213	99.84 %	50
1024	99.30 %	66	99.49 %	232	99.76 %	22
2048	99.00 %	22	99.28 %	69	99.63 %	12
4096	98.49 %	11	98.94 %	17	99.40 %	7

Table 4.2: Preprocessing time and percentage of contracted nodes for each MLP.

higher for each parameter than for the other methods. Gonzalez needs the most time (20 hours and 14 minutes in total) for contracting 99.64%. K-means performs better than Gonzalez but worse than Merge for the same parameters.

K-means was expected to need less time than the other methods due to its well-balanced partitions and high amount of boundary nodes. The high preprocessing times for Gonzalez can be explained due to its unbalanced partition sizes. Contrary to our expectations, Merge performs better than K-means and Gonzalez. The runtime of Merge can be explained because the node ordering heuristic starts the heuristic calculations locally in the same way as Merge by looking at its neighbors. Both are continuously contracting or merging nodes that follow a very similar scheme. Therefore, Merge generates MLPs that follow the node-ordering heuristic better than Gonzalez or K-Means. By following the heuristic better, the contraction can contract nodes inside the cell unhindered.

Query	MLP	512 [ms]	512 speedup	1024 [ms]	1024 speedup	2048 [ms]	2048 speedup	4096 [ms]	4096 speedup
PCRP	K-means	43.4	34.6	38.0	39.5	35.3	42.5	36.7	40.9
	Gonzalez	62.6	24.0	46.2	32.5	37.8	39.7	36.9	40.6
	Merge	27.9	53.8	18.0	83.6	16.4	91.5	16.4	91.5
PCH	K-means	88.9	16.9	89.1	16.8	96.0	15.6	112.4	13.4
	Gonzalez	109.7	13.7	105.0	14.3	99.3	15.1	104.9	14.3
	Merge	26.5	56.6	28.0	53.6	33.1	45.4	42.9	35.0
PRP	K-means	41.9	35.8	36.8	40.7	37.3	40.3	40.5	37.1
	Gonzalez	52.1	28.8	41.3	36.3	34.4	43.7	34.7	43.2
	Merge	22.2	67.6	15.9	94.6	14.2	105.6	15.7	95.8

Table 4.3: Query times and speedups for PCH, PCRP, and PRP in combination with K-means, Gonzalez, Merge.

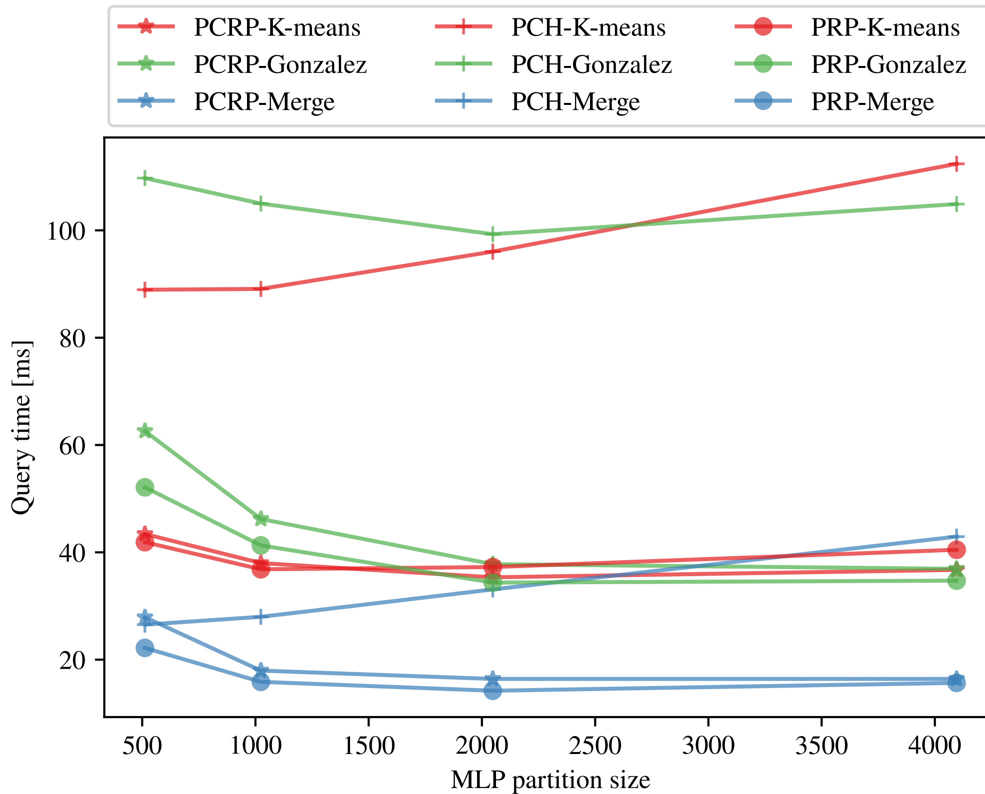


Figure 4.6: Dijkstra variants comparison with different MLPs (Visual representation of Table 4.3).

4.2.3 Single-Level Queries

The Figure 4.6 is the visualization of Table 4.3. When looking at the different MLP methods, it is shown that Merge outperforms K-means and Gonzalez for PCRP, PCH, and PRP. For a small number of contracted nodes, PCH has worse query times than PCRP and PRP. PCH achieves better query times by contracting more nodes. Both CRP and PRP have their best performance, around $2,048$ partitions with a single-level-partitioning. For CRP having less than $2,048$ nodes, the query time decreases.

As expected, Merge performs better than K-means and Gonzalez because the MLP properties of Merge are more promising than those of the others. The inferior PCH performance for less contracted nodes is due to the termination condition of CH. This problem for PCH becomes less significant the more nodes are contracted. CH itself gains the most speedups when contracting the graph densely. By doing so, the search space gets more and more limited. In contrast, PRP does not have this issue because of the usage of PCH-shortcuts for reaching the boundary nodes. The disadvantage of PRP is the overhead of switching between both internal PCH- and PCRP-heaps. For less contracted graphs, we propose for PCH to search in the topmost level with Bidirectional Dijkstra. This can be implemented by using four heaps, of two for PCH and two for Bidirectional Dijkstra on the topmost rank. In our tests with PRP, the performance was higher to explore all nodes below the maximum rank and then perform a bidirectional search. Contracting many nodes leads to larger partition sizes. For CRP, the time needed to reach a boundary is higher than the speedup provided by the pre-calculated shortcuts. Contracting the graph very densely is not wanted for PCRP and PRP because the speedup is minor. The time saved for preprocessing can provide faster updates for the users if a metric changes.

The implementation supports extracting the number of heap-pops and relaxed-edges in an additional run. Both can be seen in Figure 4.7 and Figure 4.8, respectively. In Figure 4.7, the number of heap-pops for PCH decreases with the number of contracted nodes. For PCRP, the number of heap-pops increases for having less than $2,048$ nodes. In Figure 4.8, the number of relaxed-edges increases with the size of the partitions. For MLPs generated with Gonzalez, the amount of relaxed-edges is consistently higher than K-means and Merge. PRP is always better than PCH and PRP in terms of heap-pops and relaxed-edges. Overall it can be seen that Merge outperforms both K-means and Gonzalez.

As previously explained, the increase of heap-pops for PCRP increases due to the time needed for reaching the boundary nodes. If unbalanced cells are contracted, typically, more edges will get created than having balanced ones. This can be observed for Gonzalez because more relaxed-edges are needed.

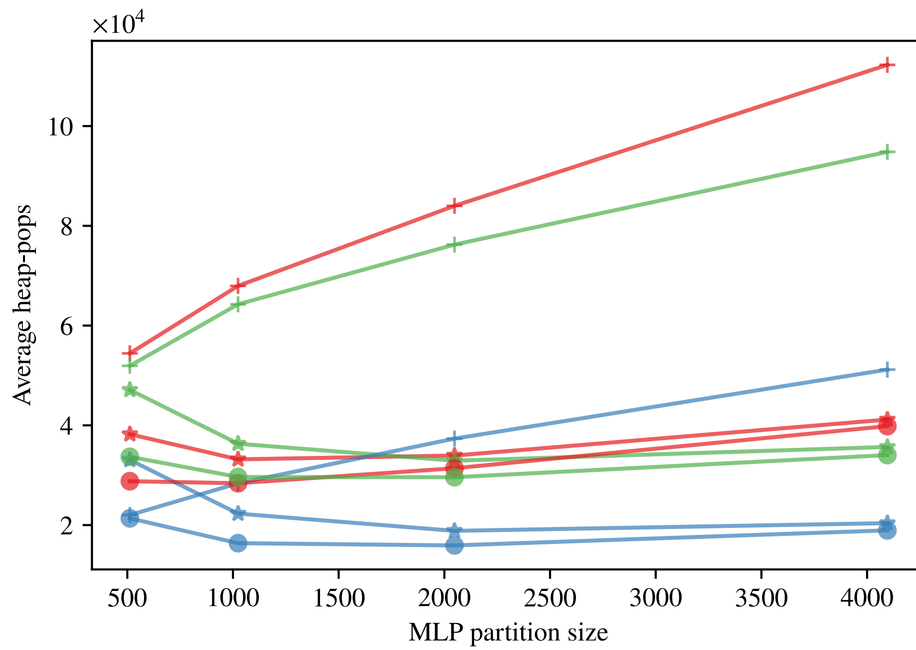


Figure 4.7: Heap-pops for single-level-partitioning.

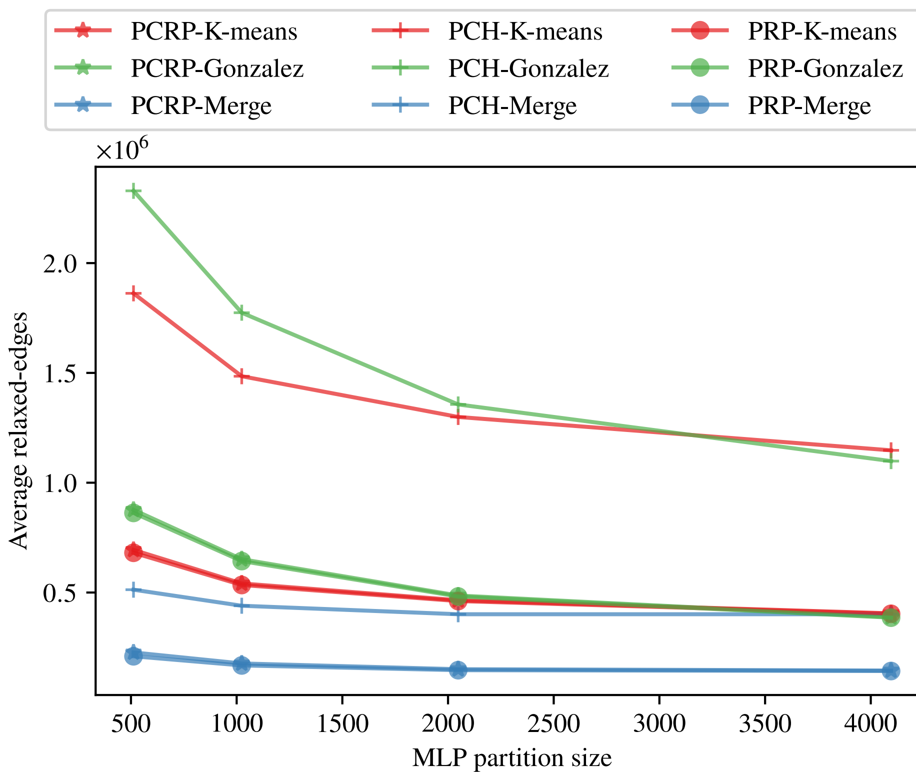


Figure 4.8: Relaxed-edges for single-level-partitioning.

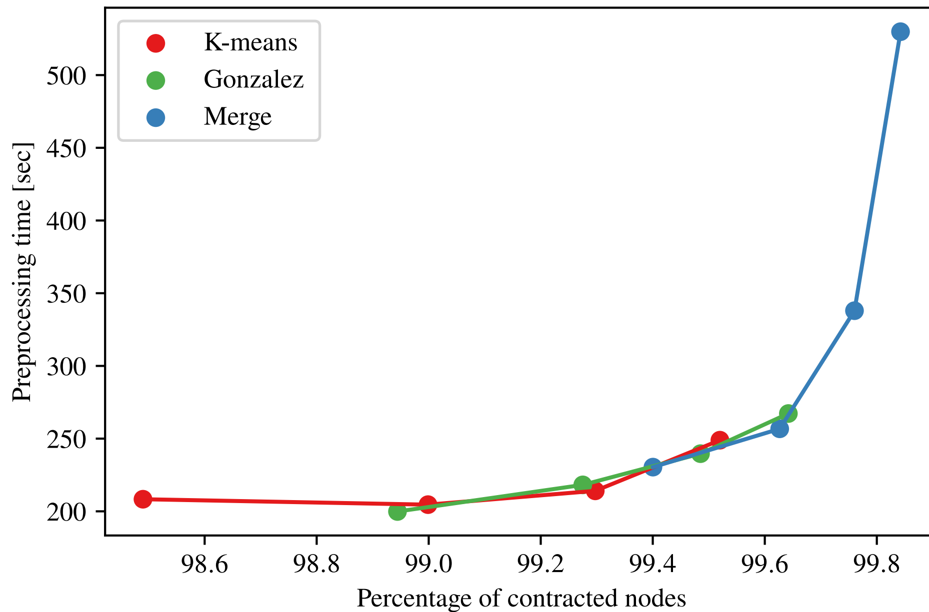


Figure 4.9: PCH preprocessing time without fixating nodes with the MLP (the termination percentage of nodes originates from the MLPs).

4.2.4 Personalizable Contraction Hierarchies

To compare the results with the work of others, a comparison of PCH with MLP and without MLP is made. From the generated MLPs, the percentages of nodes forming the boundaries are extracted. These percentages are then used to contract graphs without an MLP but with the same number of contracted nodes and the same three metrics: *distance*, *uniform costs*, and *car travel time*. First, the preprocessing run time and then the query times are evaluated. The resulting preprocessing time can be seen in Figure 4.9. All points should be interpreted as a single line because no restrictions are applied to the contraction in contrast to contracting with an MLP. It is depicted that more time is needed for PCH preprocessing if more nodes are contracted. All contractions are faster than using an MLP.

This is followed by comparing PCH with MLP and without MLP to show how much query performance drawbacks are made. Figure 4.10 is the visualization of Table 4.4, showing the different query times by using the same PCH-Dijkstra for measuring query times. All lines plotted with the triangles represent the query times of PCH without using an MLP. The lines depicted with the cross are the contracted graphs with an MLP. Contracting graphs using the MLPs K-means and Gonzalez perform worse than without MLP. In contrast to this, Merge has similar query times with or without using MLP. The blue line with crosses is nearly identical to the blue line with triangles.

Merge behaves very close to a natural contraction in terms of query time. As the heuristic itself is a bottom-up approach without a global view, Merge solves a similar problem. From the experiments, it appears that using Merge has advantages over K-means and Gonzalez. Section Section 4.2.1 and Figure 4.5 show that neither K-means for generating the MLP nor Gonzalez for contraction need the most time to preprocess a graph. Therefore, it was decided to run all further experiments using Merge only to reduce the time needed for evaluation.

MLP	Partitions	Contracted	with MLP time [ms]	with MLP speedup	no MLP time [ms]	no MLP speedup
K-means	512	99.52 %	88.94	16.87	33.49	44.81
	1024	99.30 %	89.09	16.85	41.81	35.89
	2048	99.00 %	96.04	15.63	54.14	27.72
	4096	98.49 %	112.40	13.35	74.49	20.15
Gonzalez	512	99.64 %	109.74	13.67	33.21	45.19
	1024	99.49 %	104.99	14.29	39.26	38.23
	2048	99.28 %	99.29	15.11	48.82	30.74
	4096	98.94 %	104.90	14.31	68.34	21.96
Merge	512	99.84 %	26.53	56.57	30.36	49.43
	1024	99.76 %	27.98	53.63	27.76	54.05
	2048	99.63 %	33.09	45.36	34.30	43.75
	4096	99.40 %	42.93	34.96	45.16	33.23

Table 4.4: PCH query timings and resulting speedup factors.

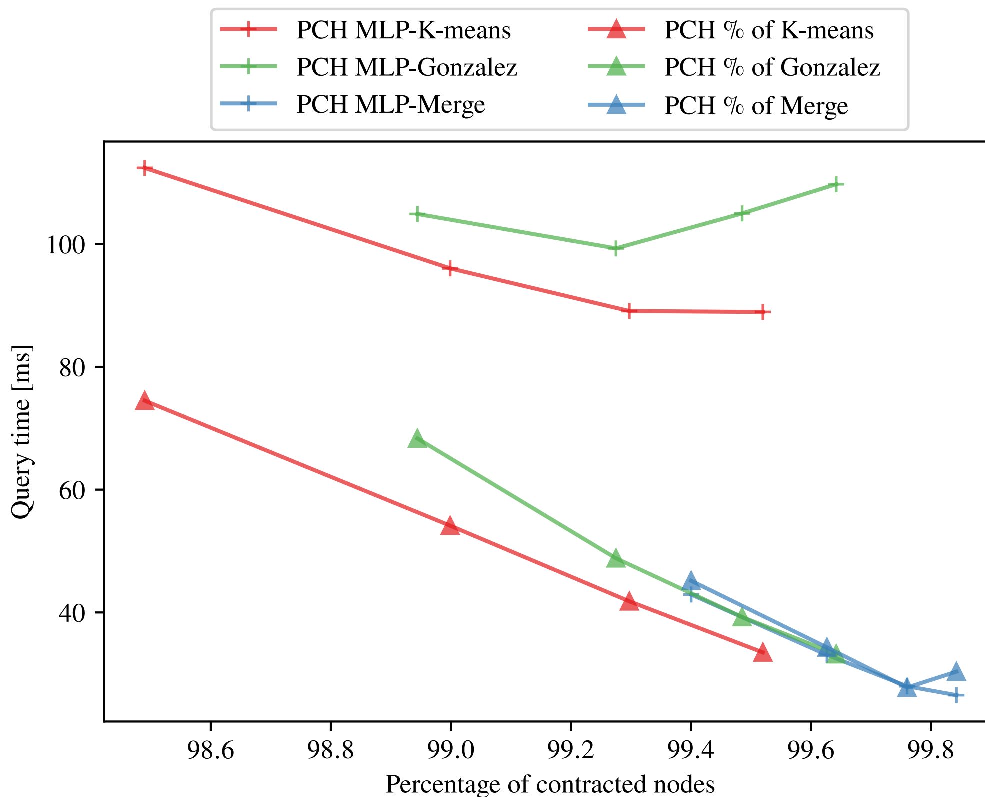


Figure 4.10: PCH queries with MLP vs without MLP (Visual representation of Table 4.4).

Partitions	Subpartitions	Max Edges [$\times 10^6$]		
		PCRP	PCH	PRP
512	None	9.2	17.8	17.8
	4	9.6	18.0	17.3
	4-4	9.9	18.4	17.2
	16	9.6	18.3	17.0
1024	None	9.2	17.5	17.5
	4	9.5	17.7	17.0
	4-4	9.9	18.2	16.9
	16	9.6	18.1	16.7
2048	None	9.1	17.2	17.2
	4	9.5	17.6	16.8
	4-4	10.0	18.0	16.6
	16	9.7	18.0	16.3

Table 4.5: Maximum number of edges each Dijkstra could use.

4.2.5 Multi-Level Queries

Next, we investigate how much an MLP with more than one level influences the query timings. The same car graph from the previous sections will be used with the same metrics. The only difference are the generated MLPs. The topmost level has 512, $1,024$, and $2,048$ partitions. The levels below are chosen with a single subdivision (4), (16), or with two subdivisions $(4,4)$. Funke and Storandt [15] had around $4,000$ nodes in a cell on the lowest level with the same structure of sublevels $(4,4)$. In contrast $1,024$ partitions have around $4,255$ nodes in the cell at the topmost level in our experiment.

We have not yet investigated how many edges each algorithm can use. This is especially interesting by using an MLP with more than one level. Table 4.5 shows the maximum number of edges each algorithm could use for calculating the path costs. $8,7 \times 10^6$ edges are given by the original graph and are not subtracted. Dijkstra and Bidirectional Dijkstra would use a maximum of $8,7 \times 10^6$ edges and no shortcuts. PCH is the only Dijkstra-variant that uses all edges because the generated edges originate from the contraction. For a single-level-partitioning, PRP uses precisely the same number as PCH of edges. PRP on a single-level-partitioning behaves like a PCH-Dijkstra exploring until boundary/core-nodes are reached. After all boundaries have been reached, a Bidirectional Dijkstra is used for walking only on core-nodes. For PRP, the number of edges decreases by using an MLP with multiple levels, which is expected. PCRP takes fewer edges than PCH and PRP, but it has to be kept in mind that these numbers are only for finding the costs. In our implementation, PCRP will use intermediate edges for path resolution that will never be used for finding the costs.

For query performance, Table 4.6 shows that for PCH, the speedup decreases with more subdivisions. In contrast to PCH, the query time improves for PCRP and PRP. Using multiple subdivisions $(4,4)$, is faster for preprocessing than using a single subdivision with (16) subpartitions. We observe that the preprocessing time increases for all subdivisions, especially for (16) subpartitions. For subdivisions $(4,4)$ it can be seen that having fewer top partitions improves the speedup.

Partitions	Subpartitions	Pre [min]	PCRP		PCH		PRP	
			[ms]	speedup	[ms]	speedup	[ms]	speedup
512	None	50	27.9	53.8	26.5	56.6	22.2	67.6
	4	58	15.1	99.6	27.0	55.6	12.8	117.6
	4-4	54	11.3	133.3	27.3	54.9	10.6	141.5
	16	72	11.3	133.4	28.8	52.1	10.7	140.4
1024	None	22	18.0	83.6	28.0	53.6	15.9	94.6
	4	21	12.8	117.7	29.1	51.6	13.8	109.0
	4-4	23	11.0	136.6	29.1	51.6	10.6	141.7
	16	28	11.1	135.2	32.6	46.1	10.7	139.9
2048	None	12	16.4	91.5	33.1	45.4	14.2	105.6
	4	16	12.9	116.6	33.6	44.6	12.2	122.7
	4-4	16	12.2	123.4	33.8	44.4	11.8	127.6
	16	16	12.3	122.1	34.0	44.1	11.9	126.0

Table 4.6: Query-times and speedups for different Merge partitionings.

Using an MLP with multiple levels while contracting prevents the node-ordering heuristic from following its optimal ordering. If the optimal node order is not followed, more edges are usually generated. Thereby the preprocessing time will increase. For PCRP, it is expected to have higher query times because the boundary nodes can be reached faster. These speedups also prove that storing the `cell-IDs` for each level and node is unnecessary due to our Equation (3.1). Recalculating the `cell-IDs` still provides a good speedup, but this could be improved for perfect optimization of PCRP.

4.2.6 Four metrics Queries

For uncorrelated metrics, the *height ascent* or *random* cost is added to the same graph from Section 4.2.3. The *random* metric has values between 1 and 100 and is equally distributed. *Height ascent* mainly consists of low values like 0 but reaches up to 295. This will first show that our algorithms are not custom-tailored to specific metrics and how this uncorrelated metric will affect the preprocessing and the resulting number of edges. The same MLPs have been reused because they are metric-independent. Query times are compared afterward.

For the additional *random* metric, the results can be seen in Table 4.7. The partitions range from 512 up to 8, 192. The contraction with 512 partitions took 17 hours and 9 minutes. For both PCRP and PRP, it can be seen that the query times are always better than PCH. The highest speedups for each Dijkstra-variant have been achieved with 4, 096 partitions for PCRP, 1, 024 partitions for PCH, and 2, 048 for PRP. It can be observed that the speedup is decreasing for all Dijkstra-variants with 512 partitions.

Partitions	Contracted	Pre [min]	PCRP		PCH		PRP	
			[ms]	speedup	[ms]	speedup	[ms]	speedup
512	99.84 %	1029	38.7	36.1	51.7	27.0	29.6	47.2
1024	99.76 %	178	26.0	53.7	47.8	29.2	21.4	65.2
2048	99.63 %	62	20.5	68.0	48.9	28.6	18.4	75.8
4096	99.40 %	28	20.2	69.3	56.7	24.6	19.0	73.5
8192	99.01 %	11	24.1	58.0	75.8	18.4	23.5	59.5

Table 4.7: Timings and speedups for the car graph with additional metric *random*.

Partitions	Contracted	Pre [min]	PCRP		PCH		PRP	
			[ms]	speedup	[ms]	speedup	[ms]	speedup
1024	99.76 %	939	37.1	56.9	67.7	31.2	31.1	67.8
2048	99.63 %	205	29.1	72.5	66.7	31.7	26.3	80.2
4096	99.40 %	38	28.6	73.9	77.8	27.1	27.1	77.8
8192	99.01 %	18	34.7	60.9	101.5	20.8	33.0	64.0

Table 4.8: Timings and speedups for the car graph with additional metric *height ascent*.

Contracting the graph further is expected to not reveal higher speedups than for 512 partitions. This is due to the higher number of edges created by the contraction. For the queries, the time needed to iterate the relaxed-edges exceeds the advantage of visiting fewer nodes, thus slowing down the queries.

The same effect can be observed by using the *height ascent* metric in Table 4.8. In comparison, we contracted to a minimum of $1,024$ partitions because the preprocessing time took 15 hours and 39 minutes. The preprocessing times are always higher than for the metric *random*. In contrast, the resulting speedups of *height ascent* are always higher than the speedups for the *random* metric. The graph with $1,024$ partitions with the same MLP and metric *random* has in total 19.26×10^6 edges and the metric *height ascent* 19.17×10^6 edges.

The higher preprocessing time for *height ascent* can be explained by the number of edges with the same costs for the respective metric. Having many edges with equal/zero costs has the advantage of creating a single optimal shortcut and covering other edges as well. Metric *random*, in contrast, has many edges with different values. Having edges with different costs simplifies the determination of the optimal paths for shortcuts. However, in combination with other metrics, multiple optimal paths exist, and thus more edges are needed to retain the shortest path property. A higher amount of edges results in lower speedups.

Partitions	Contracted	Pre [min]	PCRP		PCH		PRP	
			[ms]	speedup	[ms]	speedup	[ms]	speedup
8192	98.72 %	1078	168.3	32.1	552.3	9.8	164.6	32.8
16384	98.09 %	267	201.2	26.9	674.2	8.0	206.3	26.2
32768	97.14 %	113	246.3	21.9	913.0	5.9	244.9	22.1

Table 4.9: Timings and speedups for the bicycle graph.

4.2.7 Bicycle Queries

There exist more transportation methods than the car. Therefore, the last experiments use the bicycle graph with only two metrics, *height ascent*, and *bicycle unsuitability distance* metric. Bicycle unsuitability distance metric is a multiplication of "how well is the road suited for bicycles" and "distance". Both of these metrics are considered to be orthogonal to each other. The graph has around two times as many nodes and is more densely connected than the car graph. Therefore, the graph cannot be contracted as much as we have done so far with the car graph. Contracting the graph less is expected in lower speedups. A maximum time of about 18 hours for a contraction with 8, 192 partitions was chosen. This results in 98.72% nodes being contracted. Additionally, graphs with 16, 384 and 32, 768 partitions have been executed as well. In Table 4.9, the results are shown. Same as before, the preprocessing time increases for a more contracted graph. All query speeds increase with the number of contracted nodes. It can be seen that PCH is inferior to PCRP and PRP in terms of query performance.

The performance of our Dijkstra variants decreases due to the less contracted graph. It is challenging to contract more nodes as the time required increases. Having a faster contraction was not the primary goal of this work but can be achieved using CCH. The fact that PCH has the lowest speedup compared to PCRP and PRP is likely due to the termination condition of PCH. PCH explores more nodes on the topmost level of the MLP in comparison to PCRP and PRP.

5 Conclusion and Future Work

In this work, we developed a new graph structure to compare the different Dijkstra speedup variants. To support PCRP, three different MLP methods have been implemented: K-means, Gonzalez, and Merge. These methods were used to fixate the boundary nodes by not contracting them. The resulting graphs have been used for PCRP, PCH, and PRP. We executed performance tests for multiple graphs with different metrics that are less contracted than PCH usually is.

We found that the MLP method Merge works well in combination with PCH-contraction. Furthermore, contracting the graph very densely does not continuously improve the query performance, even for PCH. For less contracted graphs, we propose improving PCH by not using the traditional CH-termination on the topmost level. Instead, we recommend using a Bidirectional Dijkstra on the topmost level. We have shown that PCRP is a good alternative using fewer edges for finding the costs of the shortest path. For a specific number of contracted nodes, PCRP and PRP achieved higher speedups than we reached with PCH.

Our work is a contribution to find suitable algorithms for the *personalized route planning* problem. *Personalized route planning* can significantly enhance the usability and options for end-users of navigation systems. Such applications can be tailored to the user's preferences but also the needs of travel types. This is becoming increasingly important, especially with the current trend toward more and more modes of transportation, such as cargo bikes or electric scooters. The shortest route is no longer always the best. Sometimes, height ascent is a crucial factor. But every type of travel has its own preferences to determine an optimal route. This work contributes towards a generalized solution for *personalized route planning*. Applying our developed method, we show that the calculation times can be reduced compared to PCH.

In this work, simple methods were used to create an MLP. More sophisticated methods like PUNCH should be investigated with personalization. Due to the MLP file format, this can be tested with our implementation without modifications of the preprocessing and queries. Using more advanced methods produces fewer boundary nodes which typically increases the query performance for PCRP. A disadvantage of having a better MLP is the need to contract more inner nodes. It will be interesting how this will influence the preprocessing and query times.

For real-world applications with metric updates, the MLP algorithms have not yet been adopted. Combining data about traffic accidents can result in an MLP requiring only a few cells to be updated if an accident happens and therefore provide faster updates to the users.

In our implementation, PCRP uses the path resolution of PCH. If all the intermediate shortcuts are removed, the path resolution has to be done the original way as CRP does. Running Bidirectional Dijkstra between the entry and exit nodes will increase the query time and has to be further investigated.

CRP initially calculated all possible paths spanning from every entry to every exit node. Without personalization, this can be done for each metric separately by running a Dijkstra. With personalization, the task is more challenging. Iterating over all possible α is not possible. An algorithm that generates all possible personalized paths between two nodes with more than one hop has not yet been developed. Exclusively using such an algorithm is expected to be slow due to the number of possible paths. To speed up the CRP shortcuts calculation, this algorithm could be combined with PCH-contraction to produce the PCRP overlay graph. The graph is contracted by using the LP-Oracle. At a specific condition, it switches to the newly developed algorithm to calculate the edges for all entry and exit nodes on the intermediate graph. Operating on a denser contracted graph could be advantageous over the original graph.

Goal-directed algorithms are another area of research for *personalized route planning* because these algorithms provide promising query times without personalization.

This work helps to understand the problem of *personalized route planning* better. The graph structure developed in this work helps to have a better comparison of the different Dijkstra-variants. Especially, a deeper understanding of graphs with uncorrelated metrics and with proposed improvements for PCH is provided. It is currently not possible to have *personalized route planning* available in route planers. However, given the novelty and current progress, a solution may be closer than expected.

Bibliography

[1] I. Abraham, D. Delling, A. V. Goldberg, R. F. Werneck. “A Hub-Based Labeling Algorithm for Shortest Paths in Road Networks”. In: *International Symposium on Experimental Algorithms*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 230–241. ISBN: 978-3-642-20662-7. DOI: 10.1007/978-3-642-20662-7_20 (cit. on pp. 15, 18).

[2] F. Barth, S. Funke, S. Storandt. “Alternative Multicriteria Routes”. In: *2019 Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)*. Ed. by S. Kobourov, H. Meyerhenke. Philadelphia, PA: SIAM, 2019, pp. 66–80. ISBN: 978-1-61197-549-9. DOI: 10.1137/1.9781611975499.6 (cit. on pp. 15, 18, 21, 22).

[3] H. Bast, D. Delling, A. V. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, R. F. Werneck. “Route Planning in Transportation Networks”. In: *CoRR* abs/1504.05140 (2015). arXiv: 1504.05140 (cit. on p. 17).

[4] R. Boldt. *Kurviger - Dein Motorrad-Routenplaner*. 2021. URL: https://kurviger.de/ (visited on 09/15/2021) (cit. on p. 15).

[5] H. Butler, M. Daly, A. Doyle, S. Gillies, S. Hagen, T. Schaub. *The GeoJSON Format*. RFC 7946. RFC Editor, Aug. 2016. DOI: 10.17487/RFC7946 (cit. on p. 39).

[6] D. Delling, A. Goldberg, A. Nowatzyk, R. Werneck. *PHAST: Hardware-Accelerated Shortest Path Trees*. Tech. rep. MSR-TR-2010-125. Sept. 2010. DOI: 10.1109/IPDPS.2011.89 (cit. on p. 17).

[7] D. Delling, A. V. Goldberg, T. Pajor, R. F. Werneck. “Customizable Route Planning”. In: *Experimental Algorithms*. Ed. by P. M. Pardalos, S. Rebennack. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 376–387. ISBN: 978-3-642-20662-7. DOI: 10.1007/978-3-642-20662-7_32 (cit. on pp. 18, 22, 38, 39).

[8] D. Delling, A. V. Goldberg, I. Razenshteyn, R. F. Werneck. “Graph Partitioning with Natural Cuts”. In: *2011 IEEE International Parallel Distributed Processing Symposium*. 2011, pp. 1135–1146. DOI: 10.1109/IPDPS.2011.108 (cit. on p. 24).

[9] D. Delling, R. F. Werneck. “Faster Customization of Road Networks”. In: *Experimental Algorithms*. Ed. by V. Bonifaci, C. Demetrescu, A. Marchetti-Spaccamela. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 30–42. ISBN: 978-3-642-38527-8. DOI: 10.1007/978-3-642-38527-8_5 (cit. on pp. 22, 34).

[10] J. Dibbelt, B. Strasser, D. Wagner. “Customizable Contraction Hierarchies”. In: *CoRR* abs/1402.0402 (2014). arXiv: 1402.0402 (cit. on p. 20).

[11] E. W. Dijkstra et al. “A note on two problems in connexion with graphs”. In: *Numerische mathematik* 1.1 (1959), pp. 269–271. DOI: 10.1007/BF01386390 (cit. on pp. 15, 17, 19).

[12] J. Eisner, S. Funke, S. Storandt. "Optimal Route Planning for Electric Vehicles in Large Networks". In: *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*. AAAI'11. San Francisco, California: AAAI Press, 2011, pp. 1108–1113. ISBN: 978-1-57735-509-0 (cit. on p. 15).

[13] I. Flinsenberg, M. Van Der Horst, J. Lukkien, J. Verriet. "Creating graph partitions for fast optimum route planning". In: *WSEAS Transactions on Computers* 3.3 (2004), pp. 569–574. ISSN: 1109-2750 (cit. on pp. 25, 33).

[14] S. Funke, S. Laue, S. Storandt. "Personal Routes with High-Dimensional Costs and Dynamic Approximation Guarantees". In: *16th International Symposium on Experimental Algorithms (SEA 2017)*. Ed. by C. S. Iliopoulos, S. P. Pissis, S. J. Puglisi, R. Raman. Vol. 75. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 18:1–18:13. ISBN: 978-3-95977-036-1. DOI: 10.4230/LIPIcs.SEA.2017.18 (cit. on p. 21).

[15] S. Funke, S. Storandt. "Personalized Route Planning in Road Networks". In: *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*. SIGSPATIAL '15. Seattle, Washington: Association for Computing Machinery, 2015. ISBN: 9781450339674. DOI: 10.1145/2820783.2820830 (cit. on pp. 15, 19, 21, 22, 25, 53).

[16] M. R. Garey, D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. USA: W. H. Freeman & Co., 1979. ISBN: 0-7167-1045-5 (cit. on p. 23).

[17] R. Geisberger, P. Sanders, D. Schultes, D. Delling. "Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks". In: *Proceedings of the 7th International Conference on Experimental Algorithms*. WEA'08. Provincetown, MA, USA: Springer-Verlag, 2008, pp. 319–333. ISBN: 3540685480. DOI: 10.1007/978-3-540-68552-4_24 (cit. on pp. 17, 20).

[18] A. V. Goldberg. "A Practical Shortest Path Algorithm with Linear Expected Time". In: 37.5 (Feb. 2008), pp. 1637–1655. ISSN: 0097-5397. DOI: 10.1137/070698774 (cit. on p. 17).

[19] A. V. Goldberg. "Point-to-Point Shortest Path Algorithms with Preprocessing". In: *SOFSEM 2007: Theory and Practice of Computer Science*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 88–102. ISBN: 978-3-540-69507-3. DOI: 10.1007/978-3-540-69507-3_6 (cit. on p. 20).

[20] T. F. Gonzalez. "Clustering to minimize the maximum intercluster distance". In: *Theoretical Computer Science* 38 (1985), pp. 293–306. ISSN: 0304-3975. DOI: 10.1016/0304-3975(85)90224-5 (cit. on p. 24).

[21] P. E. Hart, N. J. Nilsson, B. Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. DOI: 10.1109/TSSC.1968.300136 (cit. on p. 17).

[22] G. Karypis, V. Kumar. "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs". In: *SIAM Journal on Scientific Computing* 20.1 (1998), pp. 359–392. DOI: 10.1137/S1064827595287997. eprint: https://doi.org/10.1137/S1064827595287997 (cit. on p. 24).

[23] B. W. Kernighan, S. Lin. "An efficient heuristic procedure for partitioning graphs". In: *The Bell System Technical Journal* 49.2 (1970), pp. 291–307. DOI: 10.1002/j.1538-7305.1970.tb01770.x (cit. on p. 23).

[24] S. Lloyd. "Least squares quantization in PCM". In: *IEEE Transactions on Information Theory* 28.2 (1982), pp. 129–137. DOI: 10.1109/TIT.1982.1056489 (cit. on p. 24).

[25] S. Martello, P. Toth. "Heuristic algorithms for the multiple knapsack problem". In: *Computing* 27.2 (1981), pp. 93–112. DOI: 10.1007/BF02243544 (cit. on p. 24).

[26] OpenStreetMap contributors. *Planet dump retrieved from https://planet.osm.org*. 2021. URL: https://www.openstreetmap.org (visited on 09/15/2021) (cit. on p. 43).

[27] K. Peng, V. C. M. Leung, Q. Huang. "Clustering Approach Based on Mini Batch Kmeans for Intrusion Detection System Over Big Data". In: *IEEE Access* 6 (2018), pp. 11897–11906. DOI: 10.1109/ACCESS.2018.2810267 (cit. on p. 31).

[28] *Powerful new route planner that prefers greenery and can generate round trip routes of a specified distance | Trail Router*. 2021. URL: https://trailrouter.com (visited on 09/15/2021) (cit. on p. 15).

[29] P. Sanders, D. Schultes. "Engineering Highway Hierarchies". In: *ACM J. Exp. Algorithmics* 17 (Sept. 2012). ISSN: 1084-6654. DOI: 10.1145/2133803.2330080 (cit. on p. 17).

[30] A. Schild, C. Sommer. "On Balanced Separators in Road Networks". In: *Experimental Algorithms*. Ed. by E. Bampis. Cham: Springer International Publishing, 2015, pp. 286–297. ISBN: 978-3-319-20086-6. DOI: 10.1007/978-3-319-20086-6_22 (cit. on p. 24).

[31] D. Schultes, P. Sanders. "Dynamic Highway-Node Routing". In: *Experimental Algorithms*. Ed. by C. Demetrescu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 66–79. ISBN: 978-3-540-72845-0. DOI: 10.1007/978-3-540-72845-0_6 (cit. on p. 17).

[32] *Shuttle Radar Topography Mission (SRTM)*. 2009. DOI: 10.3133/fs20093087 (cit. on p. 43).

[33] L. Sint, D. de Champeaux. "An Improved Bidirectional Heuristic Search Algorithm". In: *J. ACM* 24.2 (Apr. 1977), pp. 177–191. ISSN: 0004-5411. DOI: 10.1145/322003.322004 (cit. on pp. 17, 20).

[34] D. Van Vliet. "Improved shortest path algorithms for transport networks". In: *Transportation Research* 12.1 (1978), pp. 7–20. ISSN: 0041-1647. DOI: 10.1016/0041-1647(78)90102-8 (cit. on p. 18).

All links were last followed on September 15, 2021.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature