Karlsruhe Institute of Technology

Customizable Contraction Hierarchies with Turn Costs

Bachelor's Thesis of

Michael Zündorf

At the Department of Informatics
Institute of Theoretical Informatics

Reviewer:	Prof. Dr. Dorothea Wagner
Second reviewer:	Prof. Dr. Peter Sanders
Advisors:	Valentin Buchhold, M.Sc.
	Tim Zeitz, M.Sc.

March 1, 2019 – June 30, 2019

KIT – The Research University in the Helmholtz Association www.kit.edu

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text. I also declare that I have read and observed the *Satzung zur Sicherung guter wissenschaftlicher Praxis am Karlsruher Institut für Technologie (KIT).*

Karlsruhe, June 30, 2019

. . . .

(Michael Zündorf)

Abstract

Many algorithms are known to solve the shortest path problem on road networks without turn costs. This work focuses on Customizable Contraction Hierarchies and extends this technique to handle networks with turn costs efficiently. We compare different ways to model turn costs in road networks and evaluate which impact those models have on the performance of Customizable Contraction Hierarchies. We also propose some improvements to handle turn costs faster. Albeit most of these improvements were designed for turn costs, some do also apply to Customizable Contraction Hierarchies in general. In addition to that, we introduce a new algorithm to build a Customizable Contraction Hierarchie which is not only simpler but also more efficient than the original algorithm.

Zusammenfassung

Es gibt verschiedenste Algorithmen, die das kürzeste Wege Problem auf Straßengraphen lösen, jedoch ohne Abbiegekosten zu betrachten. In dieser Arbeit untersuchen wir Customizable Contraction Hierarchies und erweitern diese, um auch Abbiegekosten korrekt zu behandeln. Darüber hinaus vergleichen wir verschiedene Möglichkeiten die Abbiegekosten zu modellieren und gehen auf die Auswirkungen der Modellierung auf Customizable Contraction Hierarchies näher ein. Außerdem stellen wir Verbesserungen vor, um Abbiegekosten effizienter zu handhaben. Obwohl diese Verbesserungen für Abbiegekosten entwickelt wurden, können sie teilweise auch auf Customizable Contraction Hierarchies im Algemeinen angewandt werden. Darüber hinaus stellen wir einen neuen Algorithmus für den Aufbau von Customizable Contraction Hierarchies vor. Dieser Algorithmus ist nicht nur simpler, sondern auch effizienter, als der ursprüngliche Algorithmus.

Contents

1 Introduction

1.1 Motivation

The single-source shortest-path problem is one of the best-known problems in computer science. Dijkstra's Algorithm was the first algorithm to solve this problem in almost linear time [Dij59]. There is a broad spectrum of problems which are related to the single source-shortest path problem. Those problems include real-time traffic updates, time-dependent routing, public transit, electric vehicle routing and many more. In the past decades, many new techniques have been developed to deal with these problems. A comprehensive summary of the state of the art route planning algorithms is given by Bast et al. [Bas+16]. One of the problems related to the single source-shortest path is the turn cost problem. Here the problem is to find the shortest path while considering costs for turns and respecting turn restrictions. It is important to distinguish between different types of turns since it could be faster to make a right turn instead of a left turn. For example, some mail delivery services try to eliminate as many left turns as possible while planning their routes [UPS]. One of the state-of-the-art speed-up techniques is the Customizable Contraction Hierarchie (CCH). It is a three-stage approach allowing real-time traffic updates in a few seconds while being able to process shortest path queries in under a second. The combination of CCHs with the turn cost problem has not been examined yet. Therefore, we study this combination in our work.

1.2 Related Work

Turn costs are important for real-world routing applications. Therefore many speed-up techniques have already been studied in combination with turn costs. Among those techniques are Customizable Route Planning (CRP) [DGPW11] and Contraction Hierarchies (CH) [GV11]. Those earlier works already introduced two models which allow the incorporation of turn costs into graphs, one which embeds the turn costs directly into the graph and another one which adds turn cost tables to each node. An earlier version of the CCH paper already includes a chapter about turn costs. The paper gives a prediction for the performance of CCHs with turn costs in the first model. In addition to that, some ideas for improvements are mentioned. However nothing of this has been evaluated yet, and the corresponding chapter has been removed from the final version [DSW14 | DSW16]. In this work, we pick up some of the ideas mentioned there and evaluate them.

1.3 Outline

In the next chapter, we cover the basic concepts of graph theory and route planning. Moreover, we define the conventions we use in this work. In the third chapter, we describe how we incorporate turn costs into our graph and elaborate on the basic properties of the two models we use. In Chapters 4 to 6, we show how the turn cost models can be used with CCHs. In addition to that, we propose some improvements for CCHs in context of turn costs. Our primary focus lies on the speed-up of the customization stage, which suffers the most if we include turn costs. To achieve a speed-up in this stage, we modify the metric-independent preprocessing stage, to reduce the number of lower triangles which are enumerated in the customization stage. We also introduce the fast and simple triangulation[1] algorithm (FastAlgorithm). This algorithm and some of the modifications we introduce in Sections 5.3 and 6.1 can also be applied to speed-up CCHs in general. After that, we present an experimental evaluation of our proposed algorithms on some real world road networks.

[1]In our context, triangulated is equivalent to chordal, but triangulation results in the better acronym. However, this is not a planar triangulation.

2 Preliminaries

In the following sections we introduce basic concepts of graph theory and route planning. We also introduce notations that are used throughout this work. In particular, we define graphs and cost functions to model routing networks. In addition to that we give a brief introduction to CCHs.

2.1 Graph Theory

An *undirected graph* $G = (V, E)$ consists of a set of vertices $V = \{1, 2, \ldots, n\}$ and a set of edges $E \subseteq \big\{\{u, v\} \mid u, v \in V\big\}$. In contrast, a *directed graph* $G = (V, E)$ consists of V and a set of edges $E \subseteq \big\{(u, v) \mid u, v \in V\big\}$. The only difference between the two definitions is that in a directed graph an edge has a direction. This means that the edge (u, v) has a source u and a target v. In particular, this implies that the edges (u, v) and (v, u) are not equal. If not stated otherwise, we only consider simple directed loop-free graphs in this work.

The *degree* $deg(v)$ of a vertex v is the number of edges that contain v. If the graph G is directed we differentiate between *indegree* $deg_{in}(v)$ and *outdegree* $deg_{out}(v)$ of a vertex v. The indegree is defined as the number of edges that end in v. Similarly the outdegree is defined as the number of edges that start in v.

$$\begin{aligned}
deg(v) &:= \#\big\{e \mid e \in E \text{ and } v \in e\big\} \\
deg_{in}(v) &:= \#\big\{(u, v) \mid (u, v) \in E\big\} \\
deg_{out}(v) &:= \#\big\{(v, u) \mid (v, u) \in E\big\}
\end{aligned}$$

The *line graph* $\mathcal{L}(G)$ of a graph G represents the adjacency of edges in G. For this the vertices of $\mathcal{L}(G)$ correspond to the edges of G. Two vertices in $\mathcal{L}(G)$ are connected if and only if the corresponding edges in G are adjacent. The concept of line graphs can be extended to directed graphs. In this case, the resulting graph is also directed and we call it *line digraph*. In this graph, all vertices correspond to directed edges in G. A directed edge in $\mathcal{L}(G)$ is induced between (a, b) and (c, d) if b equals c. We also denote the directed line graph with $\mathcal{L}(G)$. Thus $\mathcal{L}(G)$ is directed if and only if G is directed.

A graph G is called *chordal* or *triangulated* if it contains no holes. This means that there exists no induced cycle on more than three vertices in G. An alternative characterization is that a vertex order *ord* exists such that the neighbors of each vertex v which come after v in *ord* form a clique [FG65]. Such an order is called a *perfect elimination scheme*.

2.2 Route Planning

In the context of route planning, a road network is modeled as a directed graph. A vertex in this graph corresponds to a intersection and a directed edge corresponds to a road segment. A bidirectional road is modeled as two directed edges. Additionally we use a *weight function* c, sometimes referred to as *metric*. Such a metric is used to model the incurring costs for taking a road. In our work, we assume that the values of c are non-negative. Moreover, if not stated otherwise, the metric we use describes the travel time needed to traverse a road. For this metric it should be clear that all values are non-negative.

Similarly we define a *turn cost function* c^t which describes the time required to take a turn and a *turn restriction function* c^r, which is *true* if it is forbidden to take a turn. We say that c^t and c^r are consistent if, $c^r(e, g) = \mathit{true}$ implies that $c^t(e, g)$ is infinity.

$$\begin{aligned} c\ &: E \to \mathbb{R}_0^+ \cup \{\infty\} \\ c^t &: E^2 \to \mathbb{R}_0^+ \cup \{\infty\} \\ c^r &: E^2 \to \mathbb{B} \end{aligned}$$

A *path* from v_1 to v_n is an n-tuple of vertices $P = (v_1, v_2, \ldots, v_n)$. We call P valid, if $(v_i, v_{i+1}) \in E$ and $c^r\big((v_i, v_{i+1}), (v_{i+1}, v_{i+2})\big) = \mathit{false}$ for all i. Accordingly, we define the length of a path P as the sum of costs for all edges and all turns that the path consists of and denote it with $c(P)$. A shortest path between u and v is then defined as a path for which $c(P)$ is minimal among all possible paths from u to v. Note that a shortest path is not necessarily unique. There may exist multiple shortest paths.

$$c(P) := \sum_{i=1}^{n-1} c\big((v_i, v_{i+1})\big) + \sum_{i=1}^{n-2} c^t\big((v_i, v_{i+1}), (v_{i+1}, v_{i+2})\big)$$

In addition to this, we define the position *pos* for each vertex. Positions correspond to the coordinates of the intersection that the vertex represents. For simplicity, we assume that all positions are on an infinite Euclidean plane and not on the surface of a sphere.

$$\mathit{pos}(v)\colon V \to \mathbb{R}^2$$

2.3 Customizable Contractiron Hierarchies

Customizable Contraction Hierarchies are a speed-up technique for point to point shortest path queries. The technique is similar to CHs but uses a metric-independent order and therefore allows a three-stage approach as introduced by Delling et al. [DGPW11].

Stage 1: metric-independent preprocessing
Stage 2: metric customization
Stage 3: query stage

In the metric-independent preprocessing some *shortcuts* are added depending on a permutation *ord* of the vertices. In our model, the turn restriction function c^r is already available at this stage, whereas c^t is not available. After this preprocessing stage, the resulting graph G_c is chordal [DSW16]. In the second stage, the metric c is customized. This means that the costs for shortcuts are calculated. In this stage c^t is available. The time required for the customization heavily depends on the density of the graph because we enumerate triangles in G_c. In the query stage, the customized metric c and G_c are used to find the shortest path between two vertices. In this stage, we use a bidirectional query algorithm to find an up-down path.

We define the *rank* of v in *ord* as $ord^{-1}(v)$. Thus it is equal to the number of vertices which are before v in *ord*. Based on this order, we define two types of neighbors of v. The neighbors of v with a lower rank are denoted by $N^{-}(v)$. Accordingly, the neighbors with a higher rank are denoted by $N^{+}(v)$. Since all ranks are unique, each neighbor of v is in either $N^{-}(v)$ or $N^{+}(v)$.

$$N^{-}(v) := \left\{u \,\middle|\, \{u,v\} \in E \text{ and } rank[u] < rank[v]\right\}$$
$$N^{+}(v) := \left\{u \,\middle|\, \{u,v\} \in E \text{ and } rank[u] > rank[v]\right\}$$

Additionally, we define the *elimination tree* based on *ord* and denote it by G_e. In G_e each vertex has an edge to its next neighbor in the perfect elimination scheme *ord*. That is there is an edge from v to the lowest ranked neighbor in $N^{+}(v)$. In combination with this we assign each vertex v a level $l(v)$. We define the level of v depending on the level of the lower neighbors of v. The level of v is as small as possible while still being greater than the level of any vertex in $N^{-}(v)$. The levels defined like this are equal for G_c and G_e.

$$l(v) := \begin{cases} 0 & \text{if } N^{-}(v) = \emptyset \\ 1 + \max\left\{l(u) \,\middle|\, \{u,v\} \in E \text{ and } u \in N^{-}(v)\right\} & \text{else} \end{cases}$$

3 Modeling Turn Costs

There are multiple ways of modeling turn costs in a directed graph. The two models we consider for this work are the expanded graph model and the table based model. These models have already been introduced and tested in the context of CHs by Geisberger et al. [GV11]. Since CCHs and CHs are closely related, some of their insights can also be applied to CCHs.

3.1 Expanded Graph

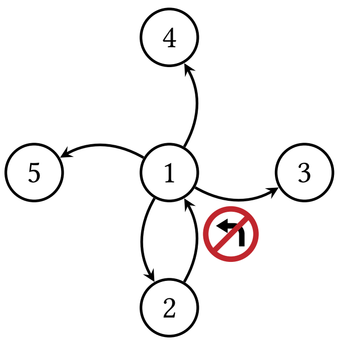
In the *expanded graph model* (edge-based [GV11]) we use a Graph G_{exp} to incorporate turn costs. In G_{exp} a vertex corresponds to a directed edge in the original graph and an edge corresponds to an edge and a turn in the original graph. Therefore the expanded graph is directed and there exists an edge between two vertices if and only if the turn at the intersection corresponding to the edges is allowed. An example of this is given in Figure 3.1. The costs c_{exp} of an expanded edge is defined as the sum of the cost of the starting edge in the original graph and the cost of the turn it implies.

$$\begin{aligned} G = (V, E) \rightarrow G_{exp} &= (V_{exp}, E_{exp}) \\ V_{exp} &\coloneqq E \\ E_{exp} &\coloneqq \left\{(e_i, e_j) \;\middle|\; e_i, e_j \in E \text{ and } \neg c^r(e_i, e_j)\right\} \end{aligned}$$

$$\begin{aligned} c_{exp} \colon E_{exp} &\rightarrow \mathbb{R}_0^+ \\ (e_i, e_j) &\mapsto c(e_i) + c^t(e_i, e_j) \end{aligned}$$

In terms of graph theory, the expanded graph created without any turn restrictions in G is $\mathcal{L}(G)$. The graph G_{exp} is, therefore, a subgraph of $\mathcal{L}(G)$. The advantage of this model is that a path in G_{exp} corresponds to a path in G with turn costs, and thus turn costs are handled implicitly. For a path P in G_{exp}, the definition of c_{exp} results in alternating weights of edges and turns in G. Therefore any speed-up technique can be applied to this graph in order to incorporate turn costs. No further modifications are required. However, this transformation increases the average vertex degree and density of the graph. Yet the maximum degree cannot increase. The reason behind both is that a single vertex v in G results in $deg_{out}(v)$ vertices u_i in G_{exp} with $deg_{in}(u_i) \leq deg_{in}(v)$. Without turn restrictions, those values are equal. Thus a vertex with high degree results in many vertices with high degree, whereas a vertex with low degree results in few vertices with low degree. This also gives a bound on the number of edges in the expanded graph.

$$\left|E_{exp}\right| \leq \sum_{v \in V} deg_{out}(v) \cdot deg_{in}(v)$$



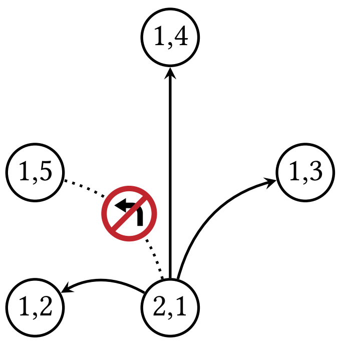


Figure 3.1: On the left, a graph G with a restricted left turn is shown. It is forbidden to take the turn form $(2, 1)$ to $(1, 5)$. On the right, the corresponding edges starting at $(2, 1)$ of the expanded graph G_{exp} are shown. The vertices on the right correspond to directed edges on the left.

This is a drawback for calculations which are heavily based on the sparseness of a graph. For example, the enumeration of triangles in the customization phase of a CCH becomes slower since the number of triangles grows more than linear with the number of edges. Another disadvantage of this model is the required space to store the graph. Overall, the number of edges is at least two to three times larger due to the expansion.

For some algorithms, a geographic embedding of the vertices of a graph is required. We use the center of the edge as the position for the corresponding vertex in the expanded graph.

$$pos(e)\colon E \longrightarrow \mathbb{R}^2, (u,v) \mapsto \frac{1}{2}\big(pos(u) + pos(v)\big)$$

3.2 Table Graph

In the *table graph model* (node-based [GV11]), the topology of G stays the same and the costs for a turn are stored in a table for each node. When taking a turn over a node v the needed turn costs can be looked up in this table. To allow this, we define two new functions $id_{in}, id_{out}\colon E \longrightarrow \mathbb{N}_0$. The function id_{in} assigns to each outgoing edge of a vertex v a unique integer and id_{out} assigns to each incoming edge a unique integer.

$$\begin{aligned} id_{in}\big((v,x)\big) &= id_{in}\big((v,y)\big) &\iff x = y \\ id_{out}\big((x,v)\big) &= id_{out}\big((y,v)\big) &\iff x = y \end{aligned}$$

We also define the function t with $t(v) \in \mathbb{R}^{m\times n}$, where $m := deg_{in}(v)$ and $n := deg_{out}(v)$. This function assigns a 2D-matrix to each vertex v which stores the turn costs for each possible turn at vertex v. This 2D-matrix is the turn table of v.

$$t\big(v_1\big)\Big[id_{in}\big((v_0,v_1)\big)\Big]\Big[id_{out}\big((v_1,v_2)\big)\Big] := c^t\big((v_0,v_1),(v_1,v_2)\big)$$

This model allows us to store all turn costs efficiently. Geisberger et al. [GV11] already mentioned that the number of different turn tables is limited, and duplicates need to be stored only once. Furthermore, since there is some degree of freedom in the choice of id_{in}, id_{out} and t we can choose it in such a way that as many tables as possible are equal.

Note that if both id_{in} and id_{out} are given, t is already completely defined. Therefore, it is sufficient to define id_{in} and id_{out} in such a way that all tables than can be equal are equal. For example it is possible to define a *lexicographic* order on matrices and chose id_{in} and id_{out} so that all tables of t are lexicographic minimal. This is always possible since the image of id_{in} can be permutated freely for each start vertex, and the image of id_{out} can be permutated freely for each end vertex. This is also described in the following formula where $a \perp\!\!\!\perp b$ denotes that a and b can be chosen independently.

$$\begin{aligned} u \neq v &\implies id_{in}\big((u,x)\big) \perp\!\!\!\perp id_{in}\big((v,y)\big) \\ u \neq v &\implies id_{out}\big((x,u)\big) \perp\!\!\!\perp id_{out}\big((y,v)\big) \end{aligned}$$

The disadvantage of this model is that the turn costs are handled explicitly. Thus a shortest-path algorithm needs to be modified to use t and to respect turn costs.

4 Nested Dissection Order

For the construction of a CCH a permutation *ord* is required. It has been shown that a nested dissection order (ND-order) is an appropriate choice for a CCH [BCRW16]. An ND-order is created by successively removing small balanced separators until the whole graph is empty. A possible way to construct a small separator is to derive it from a small balanced cut, which can be found by a graph partitioning algorithm. In the context of this work, we use the InertialFlow partitioning algorithm [SS15]. To derive a separator from a given cut, we pick the vertices from one side of the cut. We always choose the smaller side. This choice is not optimal, but in practice, this works well on road networks. Due to the sparseness of road networks, this choice is not worse than a minimal vertex cover on the subgraph induced by the cut. Also, the actual order of those separator nodes in the ND-order makes no big difference because the vertices often form a clique in G_c [DSW16].

4.1 Expanded Graphs

There exists a simple way to derive an ND-order for G_{exp} from an ND-order for G. We can replace each vertex in the order of G by all its outgoing edges. This results in a permutation of the vertices of G_{exp} since every edge of G occurs exactly once and is, therefore, a valid order. This order can be calculated in roughly the same time as an order for G.

Alternatively, we can calculate an ND-order on G_{exp} directly. This is slower but results in a better quality of the order. As we already mentioned earlier, the expanded graph is denser than the original one. Therefore, the order can be improved further by a better separator derivation. A minimal separator can be derived from a cut by taking a vertex cover on the graph induced by the cut edges. Since this induced graph is bipartite, there are efficient algorithms to compute such a minimal vertex cover due to Kőnig's theorem and efficient matching algorithms on bipartite graphs [Kőn36 | HK73].

However, we propose an even better and easier algorithm to construct an ND-order with InertialFlow for the expanded graph. For this, we use the fact that the undirected graph of G_{exp} is a subgraph of the line graph of G. This results in a duality between a cut in G and a separator in G_{exp}. Since the graph partitioning algorithm actually calculates a cut, we can use the edges in this cut to form an ND-order for G_{exp}. This yields a simpler implementation of InertialFlow for this case.

In the expanded graph model, another optimization can be used to increase the quality of an ND-order. Previously we mentioned that the order of the vertices in a separator makes no difference. This statement is only true for the number of edges in G_c, but we can increase the number edges e for which $c(e)$ will always be infinite. We will call those edges *always-infinity edges.* Since we take a cut in G where every edge has a direction, there are two different types of edges, and we need one edge of both directions to form a

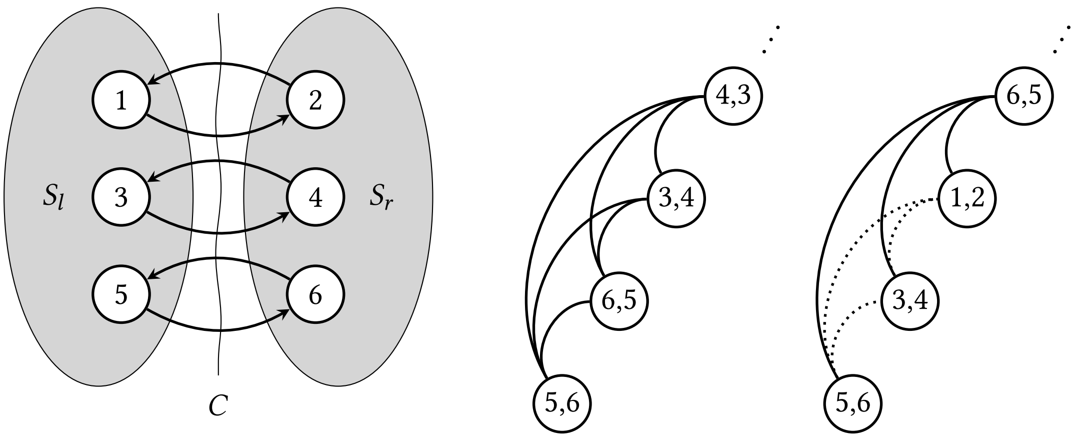


Figure 4.1: On the left is a visualization of a cut in G. In the middle is an arbitrary contraction order which results in no infinity edges after the first four contractions. On the right the edges in the order are grouped which results in three infinity edges after the first four contractions (shown by the dotted edges).

cycle. If we group these edges by their directions, all shortcuts created between edges in the first group are always-infinity edges, since it is impossible to form a cycle without any edge of the opposite direction. This is shown in Figure 4.1.

To be more precise, let the cut C split G into two parts S_l and S_r. The vertices induced by the edges in C often form a clique after the metric-independent preprocessing. If all vertices induced by edges from S_l to S_r are contracted first, the edges between those vertices have to be infinity edges. The reason for this is that an edge from S_r to S_l is required to form a path between the vertices, but these edges are not available since they are contracted later. Suppose there are e_l edges from S_l to S_r, and e_r edges from S_r to S_l, and without loss of generality let $e_l \geq e_r$. Then up to a quarter of the edges in the separator are always infinity. It can be even more if the difference between e_l and e_r is larger. The approximations is derived from the following formula.

$$\lim_{\substack{e_l \to \infty \\ e_r \to e_l}} \frac{\binom{e_l}{2}}{\binom{e_l+e_r}{2}} \geq \lim_{e_l \to \infty} \frac{\binom{e_l}{2}}{\binom{2e_l}{2}} = \lim_{e_l \to \infty} \frac{e_l - 1}{2(2e_l - 1)} = \frac{1}{4}$$

4.2 Table Graph

Since the topology of G stays untouched in this model, we can use the same ND-order as without turn costs. This is actually a drawback for the customization stage. Compared with the expanded model we have less degrees of freedom to choose an ND-order but the graph still encodes the same information. For example, an intersection without turn restrictions results in a clique inside the vertex. This means that inside the intersection all pairs of turns are allowed and there is a shortcut for each of those pairs.

4.3 Small Parts

As mentioned before, calculating an ND-order is based on balanced graph partitioning. However, graph partitioning is an $\mathcal{NP}$-hard problem. Therefore the partitioning algorithm, in our case InertialFlow, is not necessarily optimal. While successively removing balanced separators, at some point, the remaining parts are small enough to make a *brute-force* approach feasible. Thus if a part becomes smaller than a limit *lim*, it is possible to test all possible ND-orders and choose the solution which results in the fewest number of triangles. To further improve the result we count all created triangles and not only those created in the partition. To do this we simulate the contraction on all vertices in the part but also create shortcuts with vertices adjacent to any vertex in the part. After this we simulate the customization and count the triangles.

5 Metric-Independent Preprocessing

In this chapter, we deal with stage one of the CCH computation. We create shortcuts in G which decrease the search-space in the query stage. The key idea is that if there is a path between two vertices u and v, there should also be a path from u to v, which only uses vertices w_i with $rank[w_i] \geq \min\{rank[u], rank[v]\}$. To simplify this process the operations are performed on the undirected version of the graph G. In this case it is sufficient to iterate over all vertices ordered by their rank and add a shortcut between all pairs of neighbors with a higher rank. The resulting graph is the minimal chordal supergraph G_c of G for which *ord* is a perfect elimination scheme. The graph G_c is fully defined by G and *ord*. Dibbelt et al. [DSW16] specify a rather complex algorithm based on a quotient graph [GL79]. Their algorithm is able to compute G_c in $\mathcal{O}\big(|E_c| \cdot \alpha(|V|)\big)$ where α denotes the inverse Ackermann function. We now propose a much simpler and also more efficient algorithm which accomplishes the same task in $\mathcal{O}\big(|V| + |E_c|\big)$. This algorithm is heavily based on the work of Habib et al. [HMPV00] for the recognition of chordal graphs in linear time. To the best of our knowledge, this algorithm has not yet been used in the context of CCH computation.

5.1 Contraction

The fast and simple triangulation algorithm iterates through all vertices in the order defined by their rank. For each vertex u it determines the lowest ranked neighbor v in *ord* which is not yet processed. It removes all duplicated edges starting at u and then adds the neighbors of u with a higher rank to v. This means that three vertices u, v and w with $rank[u] < rank[v] < rank[w]$ result in a shortcut $\{v, w\}$ if u is adjacent to both v and w.

Algorithm 1: FastAlgorithm

```
  input :  undirected graph $G = (V, E)$, permutation $ord$
  output : minimal chordal supergraph $G_{res} = (V, E_{res})$
1 $G_{res} \longleftarrow G$
2 $rank \longleftarrow ord^{-1}$
3 for each $u \in V$ ordered by $rank$ do
4     if $N^+(u) \neq \emptyset$ then
5         $v \longleftarrow \text{argmin}\{rank[w] \mid w \in N^+(u)\}$
6         for each $\{u, w\} \in E_{res}$ do
7             if $rank[u] < rank[w]$ then
8                 $E_{res} \longleftarrow E_{res} \cup \{\{v, w\}\}$
```

Theorem 5.1: *FastAlgorithm calculates the minimal chordal supergraph of G induced by ord, that is $E_{res} = E_c$.*

Proof. We first need to show that *ord* is a perfect elimination scheme for G_{res}. It is sufficient to show that each vertex is simplicial, which means that all neighbors of v which follow after v in *ord* form a clique [FG65]. Suppose this is not true. Then there must be a counterexample consisting of three vertices x, y and z with $rank[x] < rank[y] < rank[z]$ and $\{x, y\}, \{x, z\} \in E_{res}$ and $\{y, z\} \notin E_{res}$. Without loss of generality we choose a counterexample such that x has the highest possible rank among all counterexamples.
In Line 5 we find the closest neighbor u after x. If $u \neq y$, then the edges $\{u, y\}$ and $\{u, z\}$ are be added in Line 8. Therefore u would be a counterexample which comes after x. This conflicts with our assumption that x has the highest possible rank. Thus $u = y$ holds. In this case, we add the edge $\{y, z\}$ in Line 8 and this is not a counterexample at all. Because of this, there exists no counterexample and G_{res} is chordal.
To show that $G_{res} = G_c$ we now only need to show that the graph is minimal. This is true since every time we add edges to the neighbor of v those edges are required to make v simplicial. ■

Theorem 5.2: *FastAlgorithm has a runtime of $\mathcal{O}(|V| + |E_c|)$.*

Proof. To show this, we use the accounting method. We assign to each edge in E one token and to each edge in E_c one token. A token can either be spent to remove a duplicate edge or to create a new shortcut. We assume that each edge $\{u, w\}$ in Line 6 is unique and has two tokens. In Line 8, one of those tokens is spent to create a new shortcut $\{v, w\}$ and the second token is given to this shortcut. Now there are two cases. Either $\{v, w\}$ is a new edge, then this edge gains a second token because it is in E_c. Also all edges are still unique. In the other case $\{v, w\}$ already existed so we can use its token to remove it. After this all edges are unique again.
To efficiently recognize and remove duplicated edges, we do this lazily. This means that we will remove edges with the same destination while searching the lowest ranked neighbor in Line 5. This can be done with an array where we mark the destination vertex of the edge we process. If the destination is already marked we need to remove the current edge. This allows us to recognize duplicates in $\mathcal{O}(1)$ and still guarantees that all edges processed in line 6 are unique. Since the tokens cover all operations except the enumeration of all vertices and we only provide $E + E_c$ tokens, the algorithm runs in $\mathcal{O}(|V| + |E_c|)$. ■

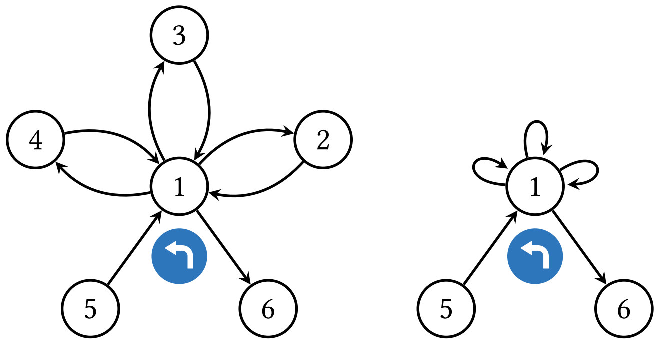


Figure 5.1: On the left, a graph G with only left turns allowed. Thus the only path from 5 to 6 is $(5, 1, 4, 1, 3, 1, 2, 1, 6)$. On the right, the same graph after contracting $\{2, 3, 4\}$. It is important that three shortcuts are added since their values of id are different. In this example all pairs of self-loops create a shortcut when 1 is contracted.

5.2 Contracting Table Graphs

Contracting vertices and adding shortcuts is also possible in the table graph model. For the contraction on the undirected graph we define a generalized id function.

$$id\colon E_c \longrightarrow \mathbb{N}_0^2, \{u, v\} \mapsto \big(id_{in}(u), id_{out}(v)\big)$$

Two edges e_1 and e_2 are considered equal if and only if $e_1 = e_2$ and $id(e_1) = id(e_2)$. This is necessary to distinguish between multi-edges and between shortcuts which are produced by the contraction. To contract a vertex v we actually contract all its edges. In our case we ordered the edges by id. Thus we behave as if we temporarily expanded the vertex we contract and use an order for G_{exp}, where each vertex v is replaced by all edges which contain v. This is done to avoid any sort of conflicts which could occur in the contraction. For example in Figure 5.1 we see that the contraction can create self-loops but multi-edges are also possible. We also need to define id for created shortcut. To be more precise, if we contract a vertex b and there are edges $\{a, b\}$ and $\{b, c\}$ we add two shortcuts $\{a, c\}$. One with $id\big(\{a, c\}\big) = \big(id_{in}(a), id_{out}(c)\big)$ and another one with $id\big(\{a, c\}\big) = \big(id_{out}(a), id_{in}(c)\big)$. Note that a can be equal to c, which results in self-loops.

5.3 Infinity Edges

Since the contraction is performed on the undirected graph of G_c, some directed edges in G_c always have an infinite weight, independent of the used metric c. This has already been mentioned in Section 4.1. There even can be edges which are always infinite in both directions. An example of such a graph is given in Figure 5.2. We therefore suggest replacing each undirected edge $\{u, v\}$ in G_c by two directed edges (u, v) and (v, u). After this, we remove the always infinity edges to reduce the number of edges for the following stages. Note that the elimination tree and the levels of the vertices have to be calculated before this step. We also propose a simple way to detect those edges.

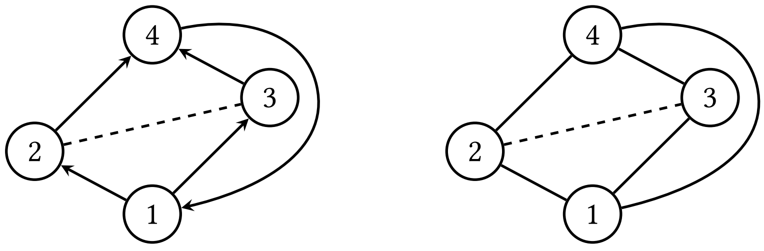


Figure 5.2: A graph G on three vertices, where $ord = (1, 2, 3, 4)$ produces an always infinity edge between the nodes 2 and 3. The edge between 2 and 3 is always-infinity in both directions since there is no path in G between 2 and 3 in the subgraph induced by the vertices 1, 2 and 3.

Theorem 5.3: *The edge $e = (u, v)$ is an always infinity edge if and only if $c(e) = \infty$ after customization of the zero metric c [DSW14].*

$$c(e) := \begin{cases} 0 & \text{if } e \in E \\ \infty & \text{if } e \notin E \end{cases}$$

Proof. Obviously, $c(e)$ is infinity, since it is infinity for every metric. It is clear that for each other metric c' it holds that $c'(e) \geq c(e)$. □

Since $c(e) \in \{0, \infty\}$, it is possible to modify the customization to stop enumerating lower triangles of e once $c(e) = 0$. It is also possible to use weights in $\mathbb{B}$ instead of $\mathbb{R}$.

As the most time of the preprocessing phase is spent on finding a good ND-order, the additional time of a single basic customization makes no significant difference for the overall preprocessing time. Since this optimization is not based on the metric or turn costs, it works with all metrics for all graphs, and the directed graph can still be customized with different metrics.

6 Customization

In the previous chapter, we explained how to compute G_c and how to remove infinity edges. In this chapter we want to incorporate the costs c and turn costs c^t into G_c. To do this we define c for the shortcuts. we also ensure that the path which only uses higher ranked vertices is always a shortest path. For this, it is sufficient to ensure that the costs for each edge $\{u, v\}$ is not greater than for any path from u to v. We call a tuple $\big(\{u, v\}, w\big)$ a *lower triangle* of $\{u, v\}$ if w is adjacent to both u and v and w has a lower rank than both u and v. Dibbelt et al. [DSW16] showed that it is sufficient to ensure that $c\big(\{u, v\}\big) \leq c\big(\{u, w\}\big) + c\big(\{w, v\}\big)$ holds for every lower triangle and if it holds we call the metric c *customized*. To customize c we enumerate all vertices by their rank. For each vertex v we enumerate all upward edges $\{u, v\}$ and for each edge all lower triangles. If the lower triangle inequality does not hold we modify c to make it hold. This customization also works if we remove edges which are infinite in both directions. We do not need to direct any edges for this since c is infinity for both directions. Therefore those edges are never used for a shortest path. Thus we only need to enumerate lower triangles and adjust the weights of the edges.

6.1 Directed Customization

We now explain a modification to handle directed graphs. In this case, there are two types of lower triangles depending on the direction of the edge $\{u, v\}$. The first approach is customizing each edge (u, v) and (v, u) with $rank[u] < rank[v]$, while processing u. This is the same as with undirected edges, but it requires storing the incoming downward edges for each vertex. However, it is also possible to customize G_c without this additional information. We need to customize two types of edges. The first type is an edge (u, v) which is directed upwards, thus $rank[v] > rank[u]$. The second type is directed downwards, and thus $rank[v] < rank[u]$. The corresponding triangles are shown in Figure 6.1. Our approach also handles vertices by their rank, but for each vertex u we first customize all downwards edges (u, v) by ascending value of $rank[v]$. After this we customize the upwards edges starting at u in any order.

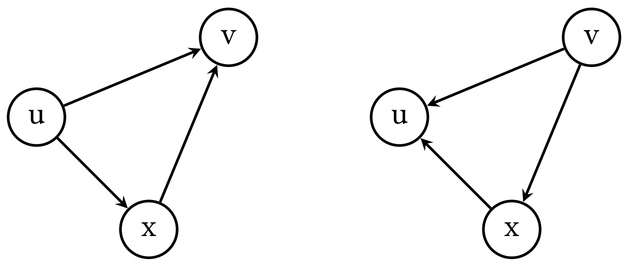


Figure 6.1: A lower triangle induced by an edge of type one on the left, and one induced by an edge of type two on the right. The order is $ord = (x, u, v)$.

Theorem 6.1: *The customization is correct if all vertices are handled in order of their rank and for each vertex u all edges (u, v) of type two are customized by ascending value of $rank[v]$ and before any edge of type one starting at u.*

Proof. To show this, we prove that the other two edges, which are required to form a triangle, are already customized. If the edge (u, v) is upward the edge (u, x) is customized earlier, since (u, x) is directed downwards. However the edge (x, v) is also directed upwards, but the vertex x has a lower rank and therefore (x, v) is customized first. Alternatively (u, v) is directed downwards. In this case (v, x) is also directed downwards, but x has a lower rank then u, and therefore (v, x) is customized earlier. The other edge is (x, u). This edge is directed upwards and is customized first because x has the lowest rank. ■

Corollary 6.2: *The customization is correct if the vertices are handled in the order of their rank and for each vertex u the edges (u, v) are handled by ascending value of $rank[v]$.*

Proof. This follows from Theorem 6.1 since the destination rank of an upwards edge is higher than the destination rank of a downwards edge. ■

6.2 Table Customization

Table customization behaves like customization described above. In Section 5.2, we already see that G_c can contain self-loops and multi-edges. Therefore a triangle can also be formed by three self-loops or by two edges and one self-loop. This can be handled in the same way as for the contraction. For each vertex, we enumerate the edges by increasing value of *id*. This ensures that the other two edges which are needed to form a triangle are customized before the currently processed edge.

Note that it would technically be possible to handle self-loops by modifying the turn cost table. Thus we would not only customize c but also c^t. However this conflicts with other optimizations for this model since one turn table is shared across multiple vertices. Thus if we modify c^t we loose one of the main advantages of this model.

7 Evaluation

In this chapter, we evaluate the algorithms that we proposed in the previous chapters. We have a particular focus on the customization of the CCH computation since turn costs have the most impact on this stage. We apply our algorithms to three different graphs, and we always use travel time as metric. The first one is a road network of the American city *Chicago* [Res]. We use this graph since it is quite dense for a road network, which we expect to have a bad impact on our algorithms. However, we have neither turn restrictions nor turn costs for this graph. Therefore, we allow all turns and set U-turn costs to 100s and all other turn costs to zero. This is the same approach as chosen by Delling et al. [DGPW11]. The second graph we use consists of the region around the German city *Stuttgart* and was provided to us by the PTV Group[1]. We use this graph since we have actual turn cost data for this graph. The third graph is the DIMACS graph of *Europe* [DGJ09]. This graph has already been used in related works to test algorithms. However, for this graph, we only have turn restrictions. Therefore, we define the turn cost the same way as for Chicago. Thus it is exactly the same instance as used by Delling et al. [DGPW11]. We do not expect a significant difference to real turn costs, since CCH operations are mostly metric independent. An overview of the used test instances is given in Table 7.1.

In the work of Geisberger et al. [GV11] the largest strongly connected component (SCC) of a graph was used for evaluation. For our tests, we do this too but with a further restriction. We want the edges of our graph to be strongly connected as well. This means that every edge is part of a cycle, and for each pair of edges a and b, there exists a path that starts with a and ends with b. This ensures that the expanded graph also forms an SCC.

Such a maximal subgraph of a graph G can be found by calculating the largest SCC of G_{exp}. As long as this SCC is not trivial, the subgraph of G which is induced by the vertices of this SCC is the required graph. Thus we take the corresponding edges in G and all induced vertices.

All tests were performed on a dual 8-core Intel Xeon E5-2670 processor clocked at 2.6 GHz, with 64 GiB of DDR3-1600 RAM, 20 MiB of L3 and 256 KiB of L2 cache. Our code was written in c++ and was compiled with g++ 8.2.1 and -O3.

[1] *https://www.ptvgroup.com/*

Table 7.1: Overview over our test graphs.

	Vertices	Edges	avg *deg*	max deg_{in}	max deg_{out}
Chicago	12 978	39 017	3.01	7	7
Chicago$_{exp}$	39 017	135 297	3.47	7	7
Stuttgart	109 695	252 487	2.30	6	7
Stuttgart$_{exp}$	252 487	395 003	1.56	5	6
Europe	17 349 990	39 936 473	1.56	12	12
Europe$_{exp}$	39 936 473	105 381 970	2.64	12	12

7.1 Nested Dissection Order

In this section, we compare the algorithms we proposed in Chapter 4. Since our primary focus is on the optimization of the customization phase, we count the number of lower triangles in the resulting CCH to compare the ND-orders because the running time of the basic customization mostly depends on the number of lower triangles. If we use precalculated triangles for that step, it only depends on this. The algorithms we compare are given in the following list.

InertialFlow is the order created by the InertialFlow which chooses the smaller side of a cut as separator.

Expanded order is the same order as above, but each vertex is replaced by its outgoing edges and thus forms an order for the expanded graph.

InertialFlow with vertex cover is the order created by the InertialFlow which chooses a minimum vertex cover as separator.

InertialFlow with cuts is the order created by the InertialFlow which uses cuts on the unexpanded graph as a separators for the expanded graph.

In Figures 7.1 to 7.3 we see the number of lower triangles in G_c created with the ND-order produced by our algorithms. We see that most triangles are on the lowest levels. Those are also the levels with the most vertices. We also see that the InertialFlow which uses cuts in the unexpanded graph is the best algorithm in terms of created triangles, but it is still far worse than the number of triangles in the unexpanded graph. The results of the *expanded order* also represents the number of triangles which are created in the table model. We see that the number of triangles is much higher as with the best order which can be used with the expanded model. Overall the number of triangles can increase by a factor up to ten. This can be seen in Figure 7.1 by comparing the *green* and *red* area or by looking at the total number of lower triangles in Table 7.2.

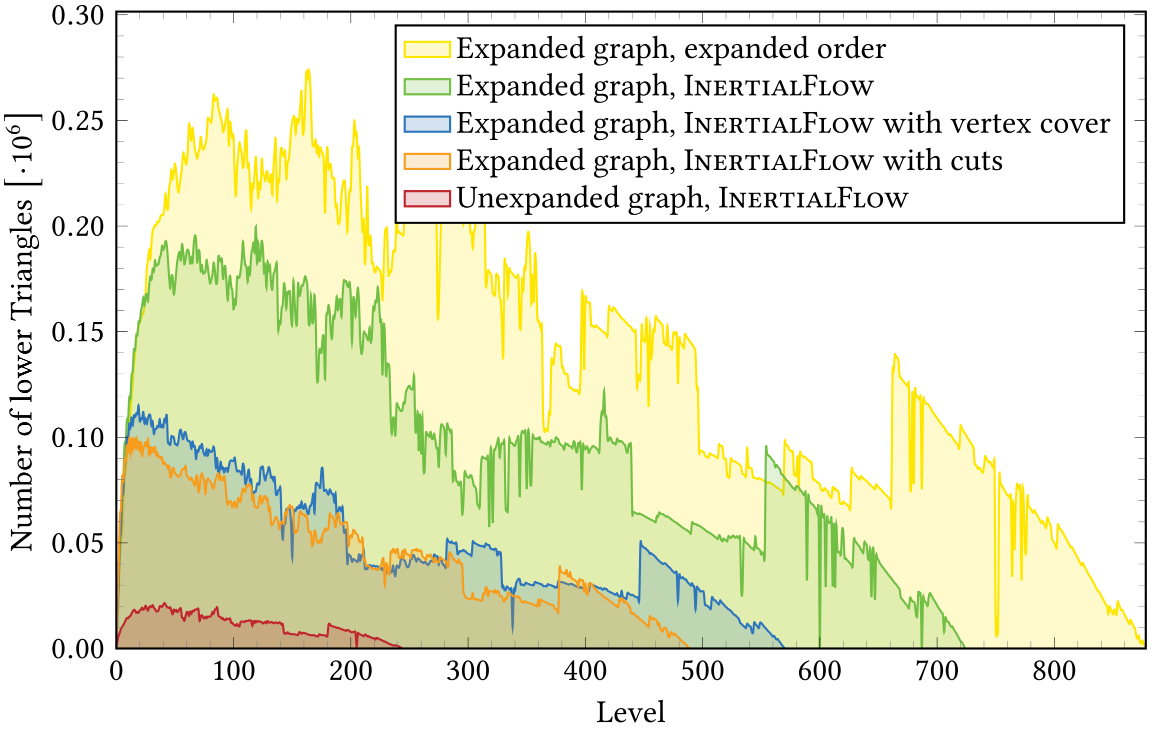


Figure 7.1: Triangles in Chicago.

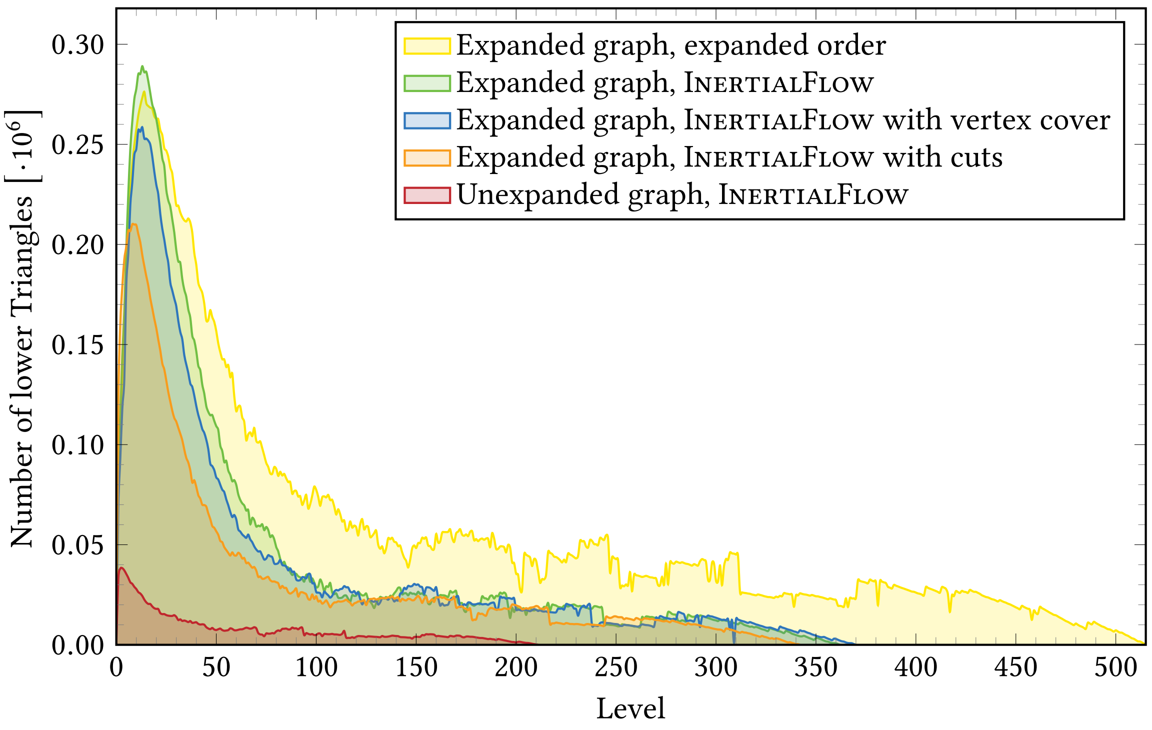


Figure 7.2: Triangles in Stuttgart.

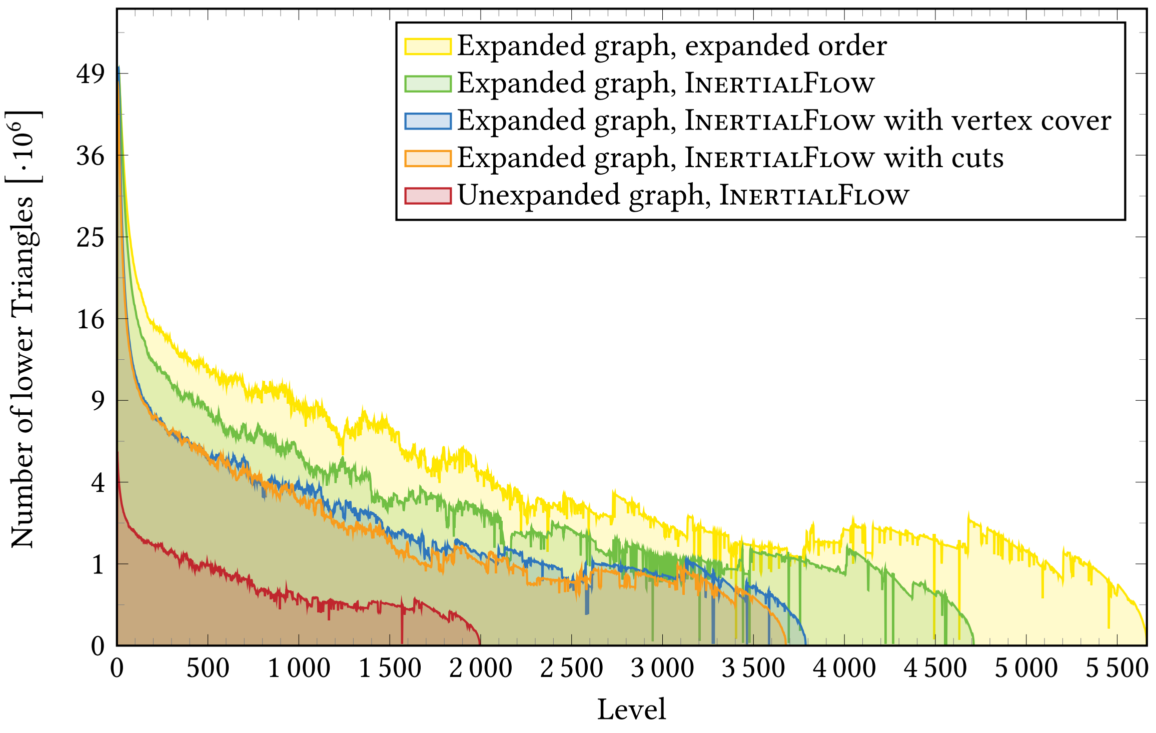


Figure 7.3: Triangles in Europe. Note that the y axis is scaled with $\sqrt{x}$.

7.2 Infinity Edges

In this section, we use InertialFlow, which uses cuts to test the impact of always infinity edges on the number of triangles. This was the best InertialFlow variation as seen in the previous section. To show how good this optimization is we mainly look at the number of lower triangles since this has the most impact on the time required for the customization. In the directed case the lower triangles consisting of $\big((x, y), z\big)$ and $\big((y, x), z\big)$ are considered as two different triangles and therefore counted twice, whereas $\big(\{x, y\}, z\big)$ is counted only once. However those numbers better fit to the actual time required for the customization and are therefore displayed like this.

The number of lower triangles can be seen in Figures 7.4 to 7.6. And the time required for the customization can be seen in Table 7.3. We see that this optimization is so efficient that the number of lower triangles is only two to three times larger as in the unexpanded case and the runtime grows by roughly the same factor. In Tables 7.2 and 7.3 we also see that this optimization can be applied in the unexpanded case. However the impact in that case is much smaller.

Table 7.2: Overview over lower triangles in our graphs.

	Both direction infinity edges	Single direction infinity edges	Triangles with infinity edges	Triangles w/o infinity edges
Chicago	1 643	22 250	2 472 267	2 064 625
Chicago$_{exp}$	195 793	600 336	22 838 460	4 104 031
Stuttgart	8 743	83 757	1 812 437	1 362 426
Stuttgart$_{exp}$	350 713	1 247 204	11 843 125	2 158 633
Europe	1 509 723	12 667 920	1 271 694 088	993 233 990
Europe$_{exp}$	83 623 967	266 426 169	9 839 173 866	1 615 075 840

Table 7.3: Time required to customize the metric. All times are given in milliseconds.

	with infinity arcs	w/o undirected infinity edges	w/o all infinity edges
Chicago	41 625	40 669	40 838
Chicago$_{exp}$	425 514	266 725	77 732
Stuttgart	39 606	38 802	37 732
Stuttgart$_{exp}$	309 267	212 364	109 557
Europe	23 864 191	23 300 575	19 893 854
Europe$_{exp}$	239 833 848	123 387 347	46 823 088

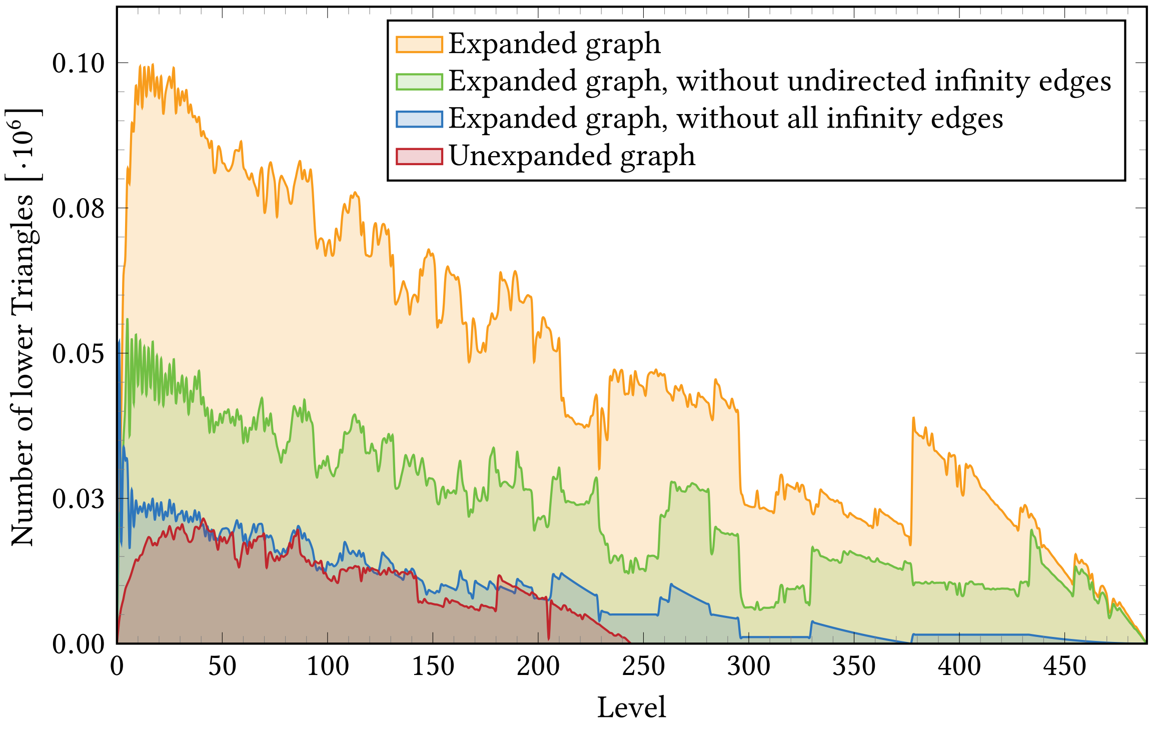


Figure 7.4: Triangles in Chicago without infinity arcs.

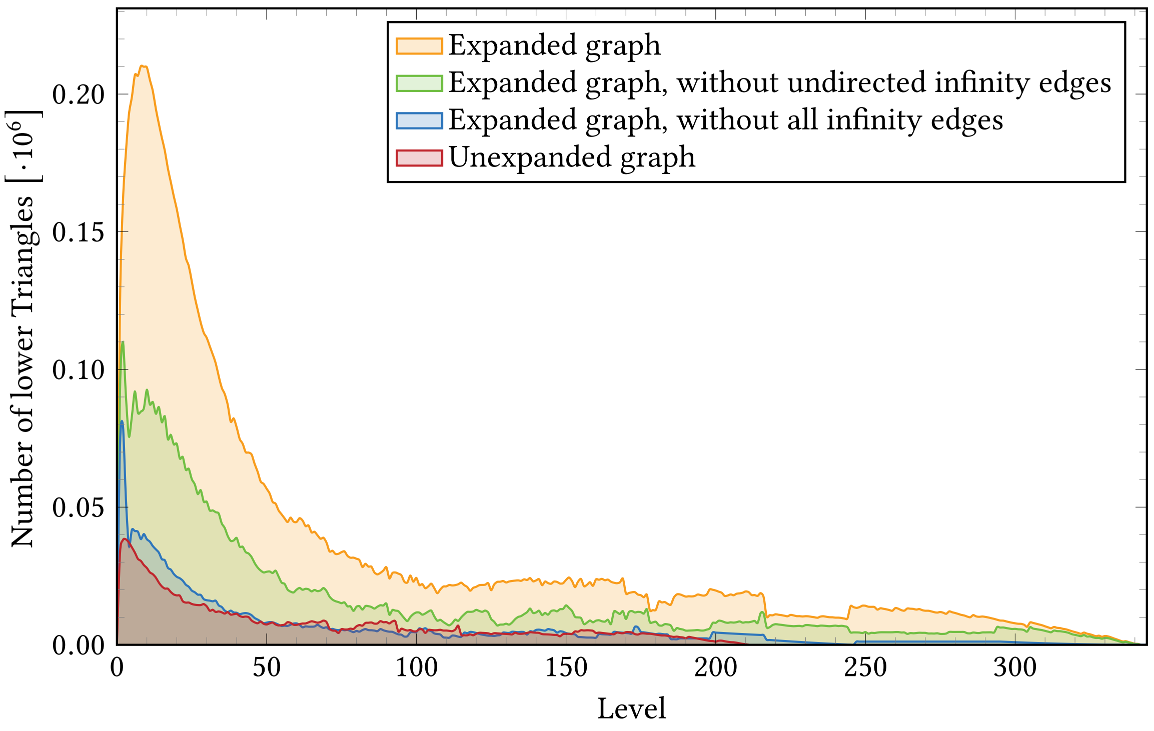


Figure 7.5: Triangles in Stuttgart without infinity arcs.

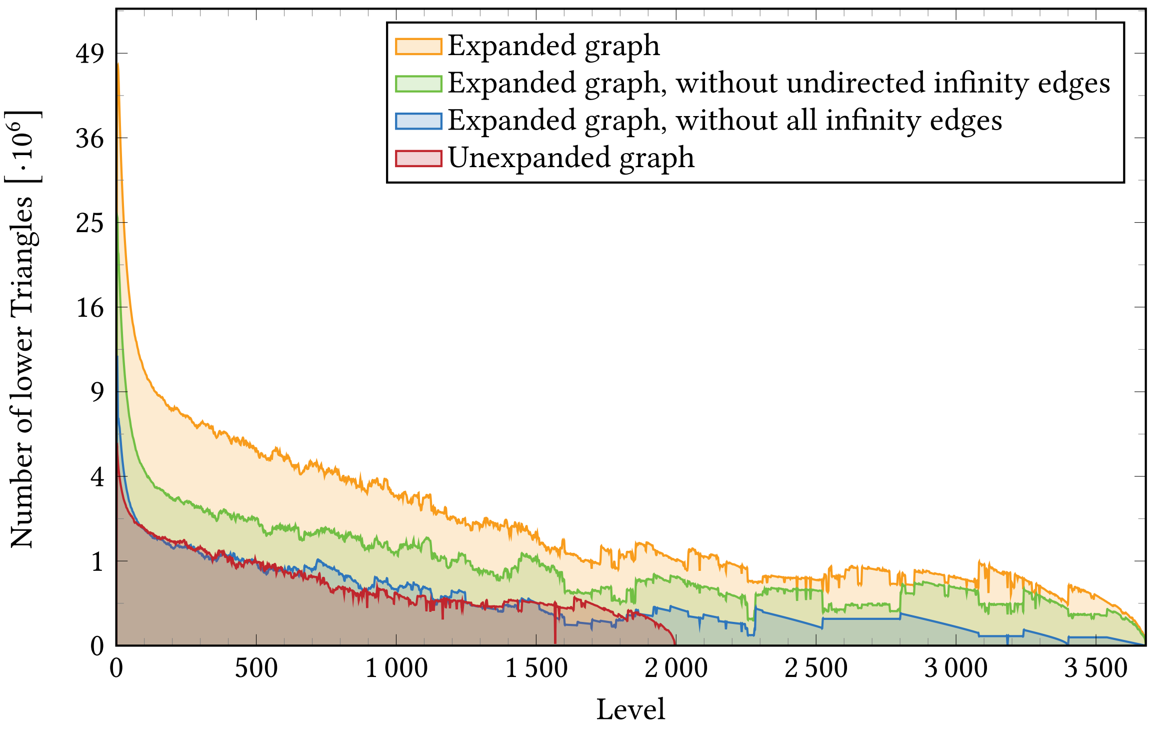


Figure 7.6: Triangles in Europe without infinity arcs. Note that the y axis is scaled with $\sqrt{x}$.

7.3 Small Parts

In Section 4.3 we mentioned a further optimization for ND-orders. This optimization should decrease the number of triangles on the lowest levels. We tested this optimization with $lim = 5$ and $lim = 10$. However, we aborted the tests with $lim = 10$ after several days since the algorithm did not finish on the Europe network. For $lim = 5$ the result was almost equal to the InertialFlow without the optimization. Overall this optimization does not seem to bring any benefit. If we also consider the runtime, it is better not to use it at all. It should also be mentioned that other partitioning algorithms do the opposite. The Metis ND-order computation algorithm switches to a greedy partitioning if the part is small enough [KK98].

7.4 Contraction

In this section, we compare the FastAlgorithm with the quotient graph based contraction algorithm described by Dibbelt et al. [DSW16] which is implemented in RoutingKit [RK]. However, the implementation in RoutingKit uses an additional optimization. The permutation *ord* is applied to the graph G before calculating G_c and is then contracted with the *identity* as order. This is done to improve cache-locality and is described in [DSW16]. This optimization can also be applied to our algorithm and has been implemented for these tests. In Table 7.4 we can see that our algorithm is not only simpler and asymptotically faster but also more than three times faster than the algorithm implemented in RoutingKit.

Table 7.4: Time required to compute G_c in milliseconds and speed-up as $\frac{QG}{\text{FAST}}$. The measured time is only for the contraction, not for the reordering process, or any other computations.

	Quotient Graph	FastAlgorithm	Speed-up
Chicago	7.4	2.2	3.41
Chicago$_{exp}$	39.3	10.8	3.63
Stuttgart	32.2	9.8	3.29
Stuttgart$_{exp}$	106.6	33.3	3.19
Europe	8 162.3	1 828.4	4.46
Europe$_{exp}$	38 966.1	10 373.9	3.76

7.5 Queries

The last point we evaluate is the speed of point-to-point shortest-path queries. The query algorithm used in this section is the elimination-tree-based query algorithm. For this algorithm we traverse the reachable vertices by their rank while updating the shortest distance. Thus we just follow the parent link of each vertex. Our queries work without the stalling described by Dibbelt et al. [DSW14] and without the optimization for local queries described by Buchhold et al. [BSW18]. Without those improvements the query time is mostly independent from the distance between the origin and the destination of the query. Therefore we test our algorithm with random queries. We choose the origin o and the destination d independently and uniformly at random from the set of vertices. For the expanded model we use the vertices corresponding to outgoing edges of o and d as origins and destinations. Thus we have multiple origins and destinations in this case. We make one million of those queries and use the average query time to compare our algorithms. This is the same approach chosen by Dibbelt et al. [DSW16] to compare CCHs with other algorithms. The algorithms we compare are given in the following list.

CCH is the CCH algorithm without turn costs and an ND-order produced by InertialFlow.

CCH$_t$ is the CCH algorithm which uses tables to incorporate turn costs and an ND-order produced by InertialFlow.

CCH$_{exp}$ is the CCH algorithm which performs on the expanded graph to incorporate turn costs and an ND-order produced by InertialFlow.

CCH$_{exp^+}$ is the CCH algorithm which performs on the expanded graph, without infinity edges and with an ND-order produced by InertialFlow with cuts.

Table 7.5: Minimum, maximum and average query time on one million random queries. All times are given in microseconds.

	Chicago			Stuttgart			Europe		
	min	max	avg	min	max	avg	min	max	avg
CCH	1	694	64	7	90	40	91	8 928	1 127
CCH_t	2	1 218	719	34	1 073	222	614	30 185	9 278
CCH_{exp}	127	2 857	518	46	251	108	357	25 652	5 380
CCH_{exp^+}	2	1 641	176	33	205	83	209	29 035	2 786

Dibbelt et al. [DSW14] already mentioned that they expect queries to be two times slower in the expanded graph model. In our tests we see that this is true for Stuttgart, whereas for Europe and Chicago it seems to be more like a factor of three. This is probably due to the fact that those road networks are more dense then the Stuttgart network. Note that for this tests the expanded graph has an additional slow-down, since we use multi-origin and multi-destination queries to query equivalent paths. However in real applications this is probably not needed and would further speed-up the expanded model.

8 Conclusion

Based on the expanded graph model we saw that turn costs can be incorporated into CCHs without any extra work. In our evaluation we see that this approach only has a slow-down of a factor two to three in the metric-independent preprocessing and query stage. However the time required for the customization increased by up to a factor of ten. We also saw that some small modifications to the ND-order calculation can significantly speed-up the second stage. With some more improvements it is possible to make stage two only three times slower compared to the non turn cost version. We also saw that the turn-table-based approach requires much more work and is slower compared to the expanded model.

8.1 Future Work

In our tests the table model was already worse than the expanded model without any optimizations. This and the fact that the expanded model is easier to use and has more space for improvements lead to the fact that our improvements were designed to work especially well with the expanded model. Are there any table model specific optimizations which make this approach competitive with the expanded model?

In the evaluation of the query time for our algorithms we already mentioned that we did not implement some known improvements. Which impact do stalling techniques have on the query time in combination with the removal of infinity edges?

For this work we only evaluated the InertialFlow partitioning algorithm. However in practice more advanced partitioning algorithms like FlowCutter are used [HS18]. How does the cut based ND-order approach for the expanded model perform with other cut based partitioning algorithms?

Bibliography

[Bas+16] Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. “Route Planning in Transportation Networks”. In: *Algorithm Engineering: Selected Results and Surveys* (2016). Edited by Lasse Kliemann and Peter Sanders, pp. 19–80. URL: *https://doi.org/10.1007/978-3-319-49487-6_2*.

[BCRW16] Reinhard Bauer, Tobias Columbus, Ignaz Rutter, and Dorothea Wagner. “Search-space size in contraction hierarchies”. In: *Theoretical Computer Science* Volume 645 (2016), pp. 112–127. URL: *https://doi.org/10.1016/j.tcs.2016.07.003*.

[BSW18] Valentin Buchhold, Peter Sanders, and Dorothea Wagner. “Real-Time Traffic Assignment Using Fast Queries in Customizable Contraction Hierarchies”. In: *17th International Symposium on Experimental Algorithms (SEA 2018)* Volume 103 (2018). Edited by Gianlorenzo D’Angelo, 27:1–27:15. ISSN: 1868-8969. DOI: *https://doi.org/10.4230/LIPIcs.SEA.2018.27*.

[DGJ09] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson. “The Shortest Path Problem: Ninth DIMACS Implementation Challenge”. In: *Proceedings. DIMACS Book* Volume 74 (2009).

[DGPW11] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. “Customizable Route Planning”. In: *Experimental Algorithms* (2011). Edited by Panos M. Pardalos and Steffen Rebennack, pp. 376–387. URL: *https://doi.org/10.1007/978-3-642-20662-7_32*.

[Dij59] Edsger W. Dijkstra. “A note on two problems in connexion with graphs”. In: *Numerische Mathematik* Volume 1.1 (1959), pp. 269–271. ISSN: 0945-3245. URL: *https://doi.org/10.1007/BF01386390*.

[DSW14] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. “Customizable Contraction Hierarchies”. Version 4. In: *Computing Research Repository* (2014). URL: *https://arxiv.org/abs/1402.0402v4*.

[DSW16] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. “Customizable Contraction Hierarchies”. In: *Experimental Algorithms* (2016), 1.5:1–1.5:49. URL: *https://doi.org/10.1145/2886843*.

[FG65] Delbert R. Fulkerson and Oliver Gross. “Incidence matrices and interval graphs”. In: *Pacific Journal of Mathematics* Volume 15 (1965), pp. 835–855. URL: *https://doi.org/10.2140/pjm.1965.15.835*.

[GL79] Alan George and Joseph W. Liu. “A quotient graph model for symmetric factorization”. In: *Sparse matrix proceedings* (1979), pp. 154–175. URL: *https://doi.org/10.1137/0209044*.

[GV11] Robert Geisberger and Christian Vetter. “Efficient Routing in Road Networks with Turn Costs”. In: *Experimental Algorithms* (2011). Edited by Panos M. Pardalos and Steffen Rebennack, pp. 100–111. URL: *https://doi.org/10.1007/978-3-642-20662-7_9*.

[HK73] John E. Hopcroft and Richard M. Karp. “An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs”. In: *SIAM Journal on Computing* Volume 2.4 (1973), pp. 225–231. URL: *https://doi.org/10.1137/0202019*.

[HMPV00] Michel Habib, Ross McConnell, Christophe Paul, and Laurent Viennot. “Lex-BFS and partition refinement, with applications to transitive orientation, interval graph recognition and consecutive ones testing”. In: *Theoretical Computer Science* Volume 234.1 (2000), pp. 59–84. URL: *https://doi.org/10.1016/S0304-3975(97)00241-7*.

[HS18] Michael Hamann and Ben Strasser. “Graph Bisection with Pareto Optimization”. In: *Experimental Algorithms* Volume 23 (2018), 1.2:1–1.2:34. ISSN: 1084-6654. URL: *https://doi.org/10.1145/3173045*.

[KK98] George Karypis and Vipin Kumar. “A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs”. In: *SIAM Journal on Computing* (1998). ISSN: 1064-8275. URL: *https://doi.org/10.1137/S1064827595287997*.

[Kőn36] Dénes Kőnig. “Theorie der endlichen und unendlichen Graphen: Kombinatorische Topologie der Streckenkomplexe”. In: *Akademische Verlagsgesellschaft mbh* Volume 16 (1936).

[Res] Transportation Networks for Research Core Team. “Transportation Networks for Research”. URL: *https://github.com/bstabler/TransportationNetworks* (visited on 05/29/2019).

[RK] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. “RoutingKit”. URL: *https://github.com/RoutingKit/RoutingKit* (visited on 05/29/2019).

[SS15] Aaron Schild and Christian Sommer. “On Balanced Separators in Road Networks”. In: *Experimental Algorithms* (2015). Edited by Evripidis Bampis, pp. 286–297. URL: *https://doi.org/10.1007/978-3-319-20086-6_22*.

[UPS] United Parcel Service. “ORION: The algorithm proving that left isn’t right”. 2016. URL: *https://www.ups.com/us/en/services/knowledge-center/article.page?kid=aa3710c2* (visited on 05/29/2019).