Karlsruhe Institute of Technology

Parallelization Techniques for Customizable Contraction Hierarchies

Bachelor Thesis of

Max Göttlicher

At the Department of Informatics
Institute of Theoretical Informatics

Reviewers:	Prof. Dr. Dorothea Wagner
	Prof. Dr. Peter Sanders
Advisors:	Valentin Buchhold, M. Sc.
	Tobias Zündorf, M. Sc.

Time Period: June 16, 2018 – October 16, 2018

Statement of Authorship

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 the Satzung zur Sicherung guter wissenschaftlicher Praxis am Karlsruher Institut für Technologie (KIT).

Karlsruhe, October 16, 2018

Abstract

Customizable Contraction Hierarchies enable fast shortest path queries on continental routing graphs. They can be used to integrate real-time traffic data into online navigation. This is possible due to a two-phase precomputation. In the first, metric-independent phase shorcuts are added to the road graph. The second phase, called customization, is used to add a metric to the augmented graph. Customization can be done in seconds enabling frequent changes to the metric.

In this thesis we present several optimizations accelerating the customization phase. These optimizations make use of the structure of the vertex and are mainly focussed on triangle enumeration. We present several approaches to parallelize these optimizations. In the end we conduct experiments with these techniques and report the results.

Deutsche Zusammenfassung

Customizable Contraction Hierarchies ermöglichen sehr schnelle Kürzeste-Wege-Anfragen auf kontinentalen Straßengraphen. Darüber können sie genutzt werden, um die aktuelle Verkehrssituation in Echtzeit in die Navigation aufzunehmen. Dies ist durch eine zweigeteilte Vorberechnung möglich, deren erster Schritt, unabhängig von der der gewählten Metrik, neue Abkürzngskanten in den graphen einbaut. Erst im zweiten Schritt, der Anpassungsphase, wird die Metrik eingebaut. Die Anpassung kann auf einem Server in wenigen Sekunden durchgeführt werden, was häufige Änderungen der Metrik ermöglicht.

In dieser Arbeit stellen wir mehrere Optimierungen vor, die die Anpassungsphase beschleuningen. Die Optimierungen nutzen die Struktur der Knotenkontraktion und beziehen sich auf die Dreiecksaufzählung. Dazu stellen wir Ansätze vor, wie diese Optimierungen parallelisiert werden können. Zu beidem führen wir Ergebnisse durch und präsentieren die Ergebnisse.

Contents

1. Introduction

In the modern world computer assisted navigation has become more and more important. Dedicated navigation hardware can be found in many cars even though these devices are getting replaced by online mapping services accessed using smartphones. Several web-based services offer interactive real-time routing on a continental scale. Such systems also take into account real-time traffic data, providing a more realistic model of actual travel time. Online routing services have to deal with very high numbers of queries in the order of several thousand requests per second. This requires highly efficient routing algorithms with very low query time, which can easily be adapted to new traffic situations.

The classical approach to finding shortest paths is using Dijkstra's algorithm[6]. It can easily switch between different distance metrics without requiring additional computation. However, with query times of up to several seconds it is unsuitable for interactive online routing on large scale graphs.

Over the last few decades many speed-up techniques have been developed in order to improve query time. Most of them depend on additional precomputation with different requirements of processing time and additional memory. One approach employed in such techniques is to add additional edges called shortcuts to the graph. These shortcuts can help to reduce the number of edges visited. Other approaches use hierarchical structures in the road graph to divide it into smaller parts. Contraction Hierarchies[7] (CH) are a technique combine these concepts. They allow for very fast shortest-path queries. As Contraction Hierarchies are fully metric-dependent, they need to be rebuilt every time the metric changes. This requires a considerable amount of preprocessing, making CHs less useful with frequent metric updates.

Exploiting the fact that the basic road network rarely changes, some approaches split preprocessing into two phases. In the first phase a metric-independent structure is generated which is then populated with edge weights in a second phase. A two-phase approach allows using multiple metrics at once for performing queries on each of them. With a fast customization phase it is possible to provide personalized metrics for specific requirements of a single user.

The first speed-up technique to support two-phase with arbitrary metrics was Customizable Route Planning[3]. It supports very fast customization in the order of seconds using arbitrary metrics.

An approach similar to Contraction Hierarchies but with precomputation split into two phases are Customizable Contraction Hierarchies (CCH)[5].

With frequent traffic updates the performance of the customization phase is quite important. CCHs allow for the customization to be accelerated using parallelization and some sequential optimizations.

Basic methods for parallelizing CCH customization have already been presented in [5].

In this work we will first summarize the CCH algorithm as introduced in [5]. We will then introduce several optimizations of the customization phase. By exploiting the structure of the graph we can accelerate triangle enumeration, which is the basic operation of several steps in the CCH algorithm.

One more chapter will be about parallelizing the customization phase using our sequential optimizations. We will present several approaches to lock-free parallization of the customization phase as well as witness search.

We will also conduct experiments on how the modifications perform and present the results in this work.

2. Preliminaries

This chapter provides the formal definition and the basic concept of customizable contraction hierarchies.

We denote an *undirected Graph* by $G = (V, E)$ where V is the finite set of vertices and E is the finite (multi-)set of undirected edges. A *directed Graph* is denoted by $G = (V, A)$ where A is the (multi-)set of directed arcs. If E contains several instances of an edge e this edge is called a *multi-edge*. A graph not containing any multi-edges is called *simple*. An edge whose endpoints are equal is called a loop.

A *vertex order* $\pi : \{1 \dots n\} \to V$ is a permutation of the vertices of a graph. Its inverse π^{-1} is called *rank*.

With respect to the order we call a vertex v *higher* than another vertex w, if and only if it has a higher rank, i.e. $\pi^{-1}(v) > \pi^{-1}(w)$. Accordingly, w is called *lower* than v. An edge or arc can be traversed upwards and downwards, i.e. from the lower endpoint to the upper or from the upper to the lower endpoint respectively.

A directed graph G is *upward directed* with respect to an order π, if for every arc the tail vertex is below the head vertex, i.e. $\forall (v, w) \in A : \pi^{-1}(v) < \pi^{-1}(w)$. Every undirected graph can be transformed into an upward directed Graph $G^{\wedge}$ with respect to a vertex order π by removing loops and replacing each remaining edge $\{i, j\} \in E$ with an arc (i, j) if j is higher than i, otherwise we replace $\{i, j\}$ with an arc (j, i).

The *upward neighbourhood* of a vertex v in a graph G with respect to an order π consists of v's neighbours with a higher rank. It is denoted by $N_u(v) \coloneqq \{w\colon\ (v, w) \in A\}$. Similarly the downward neighbourhood is defined as $N_d(v) \coloneqq \{w\colon\ (w, v) \in A\}$ and contains the neighbours of v with lower rank. By $d_u(v) = |N_u(v)|$ we denote the upward degree of a vertex v and by $d_d(v) = |N_d(v)|$ we denote the downward degree of a vertex v.

An upward directed graph can be divided into *levels*. For each vertex the level is defined as

$$l(v) = \begin{cases} 0 & \text{if } N_d(v) = \emptyset \\ \max\{l(w)\colon\ w \in N_d(v)\} + 1 & \text{otherwise} \end{cases}$$

Each vertex is in the lowest level above the levels of its downward neighbours. We define the set of vertices on each level $L_V(i) \coloneqq \{v \in V\colon\ l(v) = i\}$. Similarly, we define the set of arcs starting at a vertex on each level $L_A(i) \coloneqq \{(u, v) \in A\colon\ u \in L_V(i)\}$.

Edge weights, also called *metric*, are a mapping $w : E \to \mathbb{R}_{>0}$ for undirected graphs or $w : A \to \mathbb{R}_{>0}$ for directed graphs. With respect to an order π we define an upward weight w_u for an edge or arc when traversed upwards and a downward weight w_d when traversed downwards. For an edge or arc (u, v) with asymmetric weights we denote by $m(u, v)$ the weight from u to v while $m(v, u)$ is the weight from v to u. This can also be used to model directed weights in undirected graphs.

A sequence of edges connecting a sequence of vertices is called a *path*. We denote a path P by $p_1 \to \ldots \to p_n$ where p_i is the i-th vertex in the path. We distinguish forward and backward arcs in a path. Arcs traversed in their forward direction are called *forward arcs*. Accordingly arcs traversed in reverse are called *backward arcs*. A path is called *simple* if all vertices in the path are distinct. A closed path, i.e. a path with equal start and end vertices is called a *cycle*.

Given weights w, the sum over all of a path's edges' weights is called *weight-length* with respect to w. The number of edges in a path is called its *hop-length*. We will refer to the *weight-length* as *length* unless noted otherwise. A shortest s-t-path is a path of minimum length connecting s and t. We call the length of such a path in G the *distance* of s and t $dist_G(s, t)$. If no path exists between s and t in G we set $dist_G(s, t) = \infty$.

An *up-down-path* with respect to an order π is a path that can be split into an upward path P_u and a downward path P_d meeting at a vertex z. This *meeting vertex* has the maximum rank in the path. P_u is the path from s to x. In P_u vertices appear by increasing rank, each arc is traversed forwards. P_d is the path from x to t containing downward arcs. The vertices in P_d appear by decreasing rank and the arcs in P_d are traversed backwards. The weight length of an up-down path P_d is the sum of the weight length of P_u with respect to w_u and the weight length of P_d with respect to w_d.

A *clique* is a subset of vertices $C \subseteq V$ such that all pairs of vertices in C are adjacent.

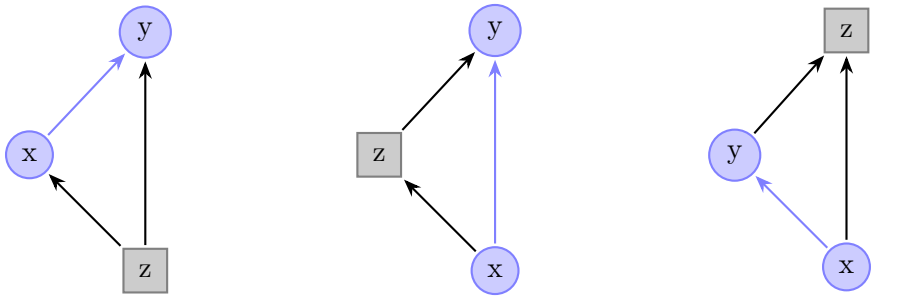
A Graph G is called *chordal* iff every cycle of at least four vertices contains a pair of vertices connected by an edge $e \in E$ not part of the cycle. Such an edge is called a *chord*. A *perfect elimination ordering* is a vertex order π so that the neighbourhood of $\pi(i)$ forms a clique in the remaining graph induced by $\pi(i), \ldots, \pi(n)$. A simple graph has a perfect elimination ordering if and only if it is a chordal graph[11].

An undirected vertex contraction G^*_π is a simple graph in which for each vertex v the upward neighbourhood $N_u(v)$ forms a clique. We denote by $G^\wedge_\pi$ the corresponding upward directed graph. The undirected vertex contraction is chordal. To prove this we show that every cycle of at least four vertices has a chord. The lowest vertex in every cycle is connected to two other vertices in the cycle. By construction the upward neighbourhood forms a clique, so an edge between those two neighbours must exist. If the cycle has four or more vertices this edge is a chord. Thus the graph is chordal. The vertex order π is the perfect elimination ordering of G^*_π. This follows directly from the definition of the elimination ordering.

The elimination tree $T_{G,\pi}$ of a chordal graph with elimination order π is a directed tree with root $\pi(n)$. In this tree each vertex's parent is its upward neighbour with the lowest rank with respect to the elimination order.

In a vertex contraction there are three types of triangles induced by an arc (x, y) or by the corresponding undirected edge $\{x, y\}$, illustrated in figure 2.1. These triangles differ in the order of the participating vertices. In a *lower triangle* $\{x, y, z\}$ induced by (x, y) the third point z has a lower rank than x and y, i.e. $\pi^{-1}(z) < \pi^{-1}(x) < \pi^{-1}(y)$. In an *intermediate triangle* z lies between x and y. In an *upper triangle* z is the highest point.

We call the path $x \to z \to y$ induced by a lower triangle $\{x, y, z\}$ of (x, y) the *lower triangle path*. The *intermediate triangle path* and *upper triangle path* are defined analogously.



(a) Lower triangle **(b)** Intermediate triangle **(c)** Upper triangle

Figure 2.1: An arc (x, y) induces three kinds of triangles. In a lower triangle as shown in (a) the third point is below the arc's endpoints. In the intermediate triangle as shown in (b) the arc's endpoints x and y lie above and below the third point z. The upper triangle has a third vertex higher than both of the arc's endpoints.

2.1 Graph Representation

We can employ different graph data structures to store a graph. These have different properties regarding the complexity of different operations such as enumerating the incident edges of a vertex or checking adjacency. For computing shortest paths we need to efficiently enumerate the incident edges and the respective neighbours of a vertex. We do not need a dynamic data structure as the graph remains mostly unchanged after construction. Routing graphs tend to have a high number of nodes in the order of tens of millions while they are sparse, meaning every vertex is only connected to very few other vertices. Not all data structures are suited for this.

Adjacency matrix: We store a quadratic matrix $M = (x_{ij}) \in \mathbb{R}^2$ in which (x_{ij}) represent the weight of the arc (i, j). If no arc exists connecting i and j we set (x_{ij}) to ∞. Using this approach it is easy to check whether two vertices are adjacent. Enumerating the neighbourhood of a vertex requires iterating all vertices and checking whether they are connected. In addition, storing the whole matrix uses a prohibitive amount of memory for a larger graph with several million vertices.

Adjacency list: Instead of storing all pairs of vertices, in adjacency list we only store the actual edges. For each vertex v we store a linked list or dynamic array containing all other vertices connected to v via an edge or outgoing arc of v. This way we can efficiently enumerate the neighbourhood of v while not using much more space than necessary to store all the edges. Storing the graph this way also allows for efficient addition of new arcs or edges. If a linked list is used arcs can also be deleted efficiently by just removing them from the list.

Adjacency Array: If there is no need for a dynamic graph data structure we can use an adjacency array. It consists of an array containing all the edge heads ordered by their tail. In additional array we store for each vertex v the range in which its outgoing arcs are stored.

Storing all arcs in the same array also improves locality of memory, which accelerates enumeration of the outgoing arcs of consecutive vertices. Packing arcs tightly in a single array also reduces memory consumption.

Unfortunately adding and removing arcs from and to the adjacency array is difficult and inefficient once it has been constructed. Edge weights are stored in a third array.

Edge list: Edges can also be stored explicitly by maintaining a list of all edges with their respective tail and head. This approach can be combined with an adjacency array by sorting the edges by tail.

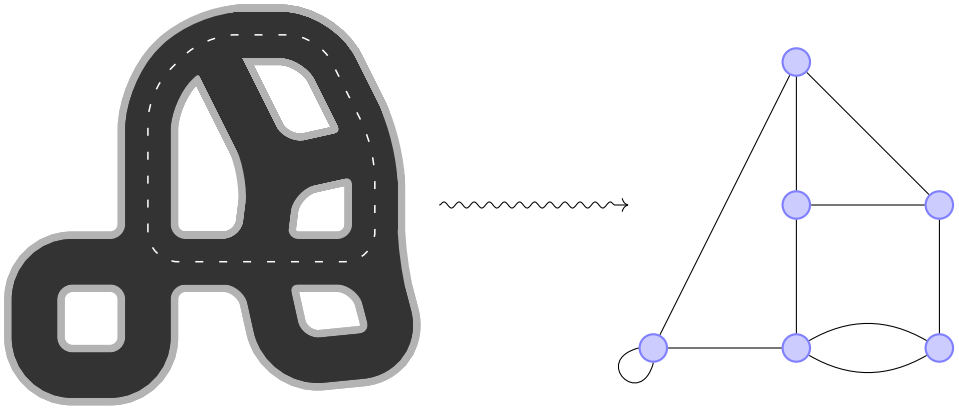


Figure 2.2: Transforming a road network to a routing graph can result in multi-edges and loops being formed. The road in the lower left corner is only connected to one junction. This results in a loop in the routing graph. The road in the lower right however is parallel to another road connecting the same junctions. The corresponding vertices are thus connected by a multi-edge.

2.2 Road Graphs

In real-world road networks not all streets are passable in both directions, such as one-way streets and highways. One-way streets can be represented by modelling the road network as a directed graph. This is also useful for direction dependent metrics such as travel time. A road network can be converted to a graph by using junctions as vertices and streets connecting them as edges or as arcs respectively. By modelling a road network this way some information such as the road's actual course is lost.

Another approach preserving the geographical course of the streets adds vertices dividing roads into small road segments. This second approach however adds a lot of vertices not necessary for larger scale navigation.

Road graphs obtained by the first approach may contain loops and multi-edges which need to be considered. Multi-edges can occur in places like residential areas which tend to have several roads connecting the same junctions, e.g. the situations in figure 2.2. Loops can occur in similar circumstances. When computing shortest paths loops can always safely be ignored. A path containing a loop will always be at least as long as the same path without the loop. Multi-edges however can be part of a shortest path. Dijkstra's algorithm can cope with multi-edges just fine. In a Customizable Contraction Hierarchy we need to take care of them because vertex contractions are simple graphs.

3. The CCH Algorithm

This chapter is about Customizable Contraction Hierarchies as described in [5]. The algorithm is included here to provide a self contained work.

By preprocessing the graph in two phases Customizable Contraction Hierarchies achieve a considerably faster query time than regular queries using Dijkstra's algorithm[6].

In the first, metric independent, phase the input graph $G = (V, E)$ is transformed into a vertex contraction by adding shortcut arcs as needed. The second phase is used to add a metric to the graph which can then be used for queries.

3.1 Building the Customizable Contraction Hierarchy

In this section we will describe how to construct the unweighted vertex contraction from an input graph using a vertex order.

We start by transforming the graph into an upward directed, simple and loop-free graph. If the input graph is directed we reverse all downward arcs and remove the resulting multiarcs.

We construct the contraction hierarchy by contracting vertices in the graph. A vertex is contracted by connecting all its upward neighbours to form a clique. We do this for every vertex in the graph with respect to the vertex order. See figure 3.1 for an example.

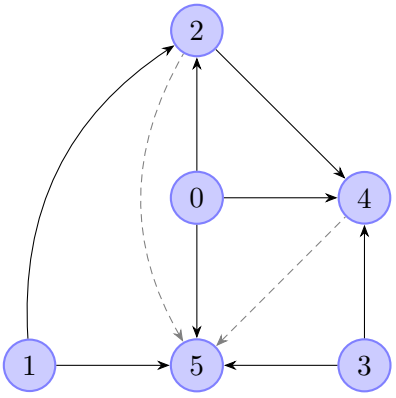


Figure 3.1: A vertex contraction of the graph from figure 2.2. Vertices have been relabelled with their respective ranks. The dashed arcs are shortcuts.

3.1.1 Obtaining the Vertex Order

As the construction of the contraction hierarchy does not rely on a metric, customizable contraction hierarchies work best with metric-independent orders[5]. Such an order can be computed using *nested dissection*[8] as suggested for the construction of Contraction Hierarchies by Bauer et al. in [1]. The order obtained using nested dissection is called *nested dissection order* (ND-order). Using an ND-order results in a low elimination tree height. This construction uses a balanced separator S splitting the graph into two vertex sets A and B. The vertices in the separator are assigned to a vertex order π_S in an arbitrary order. Then vertex orders π_A and π_B for the vertex sets A and B are computed recursively. Figure 3.2 illustrates the construction of an ND-order.

The final vertex order π is obtained by appending π_A, π_B and π_S, the resulting order is $\pi = \pi_A(1), \ldots, \pi_A(|A|), \pi_B(1), \ldots, \pi_B(|B|), \pi_S(1), \ldots, \pi_S(|S|)$.

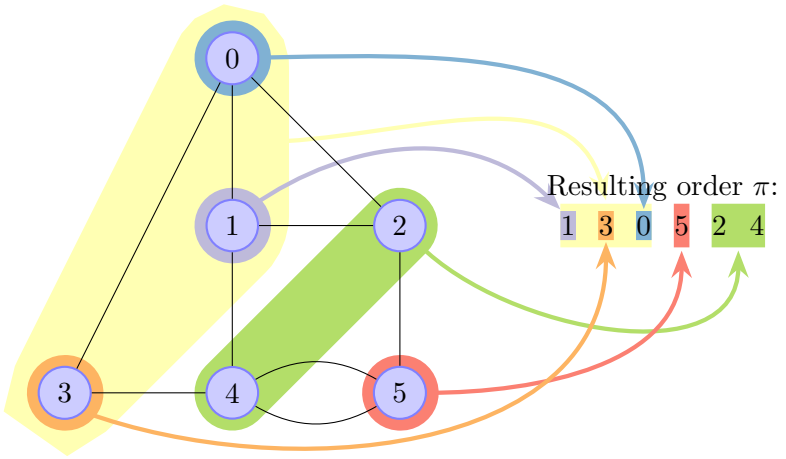


Figure 3.2: Computation of an ND-order for the graph from figure 2.2. This is the order used in figure 3.1. The first separator $S = \{2, 4\}$ splits the vertices into the sets $A = \{0, 1, 3\}$ and $B = \{5\}$. In the recursive step A is separated into $A' = \{3\}$ and $B' = \{1\}$ by $S' = \{0\}$. Note that S is not a very well-balanced separator in this example.

Computing a perfectly balanced separator is hard. There are however a few heuristic algorithms computing good separators and graph partitions on continental road graphs such as Inertial Flow[9] and PUNCH[4]. In our experiments we use the Inertial Flow graph partitioner[9]. It sorts the vertices along a line and computes the maximum flow from the first k nodes to the last k nodes. The corresponding minimum cut is used as separator.

3.2 Triangle Enumeration

For customization we will rely on the concept of triangles. In a vertex contraction with respect to a vertex order π, an arc induces three types of triangles, as described above. Enumerating the triangles induced by an arc is the fundamental operation of basic and perfect customization.

We can use two different approaches for enumerating the triangles differing in processing time and space consumption.

3.2.1 Basic Triangle Enumeration

The lower triangles of an arc (x, y) in $G_\pi^\wedge$ can be characterized by the intersection of the endpoints' downward neighbourhoods. The triangle $\{x, y, z\}$ is a lower triangle of (x, y), if

and only if $z \in N_d(x) \cap N_d(y)$. We can exploit this to efficiently enumerate all triangles. For this we need the adjacency arrays for the upward and downward neighbourhoods in $G_\pi^\wedge$ to have the same ordering, e.g. the head id or the head rank.

The neighbourhood intersection for lower triangles can then be computed by iterating the downward neighbourhoods of the arc's endpoints x and y simultaneously using two pointers. In each step we advance the iteration pointing to the lower vertex. If both iterations point to the same vertex, we found a triangle. In this case both pointers proceed to the next vertex.

The enumeration of upper and intermediate triangles works similarly. We enumerate intermediate triangles by intersecting the upward neighbourhood of the arc's tail and the downward neighbourhood of the arc's head. Upper triangles are enumerated by intersecting the upward neighbourhoods of both endpoints.

3.2.2 Triangle Preprocessing

Instead of iterating the entire neighbourhoods of all arcs we can precompute the triangles in an additional preprocessing step. This approach allows us to avoid computing the neighbourhood intersection while using much more space. Triangles are stored in an adjacency array structure mapping an arc id to the respective triangles. We need one such array for lower, intermediate and upper triangles each. For triangle enumeration to work we only need to store two of the arcs. The third arc is the one inducing the triangle. The endpoints are stored in the graph hence they do not need to be stored either.

3.2.3 Hybrid Triangle Enumeration

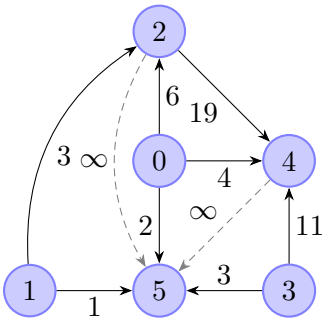
The number of triangles can be far higher than the number of arcs. This results in high memory usage for storing these triangles. In the worst case the graph contains $\Theta(n^3)$ compared to $\Theta(n^2)$ arcs. To avoid storing all triangles we can limit precomputation to those arcs (x, y) whose tail vertex x is below a certain level. The remaining triangles are still computed using the neighbourhood intersection.

3.3 Customization

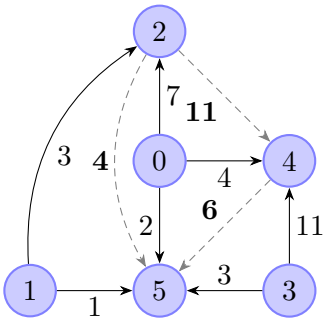
In the customization phase a metric m_G is extended from the input graph to the arcs generated in the metric independent preprocessing phase. We will refer to all edges and arcs not representing an edge in the input graph as *shortcuts*. Note that in a weighted vertex contraction this includes arcs with a weight different from that of their corresponding edge in the input graph. There are several kinds of metrics preserving shortest paths in the contraction hierarchy.

Let $\text{dist}_R(s, t)$ be the length of an arbitrary, not necessarily up-down st-path in G_π^* with respect to a metric m_R. A metric m_R fulfilling $\text{dist}_G(s, t) = \text{dist}_R(s, t)$ is called *respecting*. Such a metric can easily be constructed by copying the weights from the input graph and setting the remaining weights to infinity. This metric obviously fulfils the above constraints as all shortest paths from the input graph are kept. The shortcuts with infinite weight are not part of any shortest path.

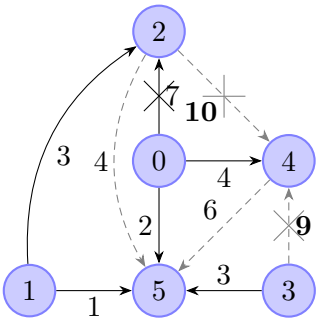
A respecting metric is not very useful on its own, as it does not make use of the shortcuts added in the first preprocessing phase. The query algorithms introduced in section 3.5 require the existence of a shortest up-down-path, which a respecting metric does not guarantee. See for example figure 3.3a in which no finitely long up-down path from vertex 2 to vertex 5 exists.



(a) Vertex contraction with respecting metric. Weights from the input graph have been copied to the corresponding arcs in the vertex contraction. The remaining weights are set to ∞.



(b) Customized metric after basic customization. Note that $(2, 4)$ became a shortcut deriving its weight from its lower triangle path $2 \to 0 \to 4$.



(c) Perfect metric. The arcs marked for deletion in witness search are striked out.

Figure 3.3: Computing customized and perfect metrics in a vertex contraction. The bold weights are the ones changed in the respective steps.

Let $dist_{ud}(s, t)$ be the length of the shortest up-down-path in $G^{\wedge}_{\pi}$ and let $dist_G(s, t)$ be the length of a shortest s-t-path in the graph G. In order for the query algorithms to be correct we need to preserve shortest paths from the input Graph G, i.e. for every (s, t) pair the shortest up-down path is exactly as long as the shortest s-t-path in the input graph or, more formally: $\forall s, t \in V : dist_{ud}(s, t) = dist_G(s, t)$. A metric m_C fulfilling these constraints is called *customized*. Note that in a customized metric arcs can have a lower weight than in the input graph. Such a case is illustrated in figures 3.3a and 3.3b where the arc $(2, 4)$ gets shorter because it has a lower triangle path $2 \to 0 \to 4$ with lower weight.

Customized metrics are not necessarily unique. Arcs not part of any shortest path can have arbitrary weights. A metric fulfilling the additional constraint of all arcs' weights being equal to the shortest path distance between their end points is called *perfect*. The perfect metric is unique for every input metric.

All customization steps use triangles in the graph in order to compute each arc's weight. If there is a triangle containing an arc, the other two arcs form a path between the endpoints of the arc. We can exploit this in order to compute the customized and the perfect metric.

3.3.1 Basic Customization

Basic customization makes use of the *lower triangle inequality*, i.e. for all lower triangles $\{x, y, z\}$ of an arc (x, y) the lower triangle path $x \to z \to y$ is at least as long as $x \to y$ or, more formally, $m(x, y) \leq m(x, z) + m(z, y)$. We will show that a respecting metric fulfilling the lower triangle inequality is customized[5].

Theorem 3.1. *A respecting metric fulfilling the lower triangle inequality is customized.*

Proof. A metric m is customized if for every *st*-pair a shortest up-down path with the same length as a shortest *st*-path in the input graph exists. If a shortest *st*-path exists in the input graph a shortest, not necessarily up-down *st*-path P exists. Note that a shortest path always exists if the input graph is connected. If P is not already up-down, it has a subpath $x \to y \to z$ with $\pi^{-1}(x) > \pi^{-1}(y)$ and $\pi^{-1}(z) > \pi^{-1}(y)$. By construction the vertex hierarchy contains an arc (x, z) or (z, x) as y is lower than both x and z. We know $m(x, z) \leq m(x, y) + m(y, z)$ because of the lower triangle inequality. The path thus does

get longer by replacing $x \to y \to z$ with $x \to z$. We can apply this argument iteratively for all such subpaths until the path is an up-down path. □

A customized metric m_C can be constructed from a respecting metric m_R by iterating all arcs in the vertex contraction ordered increasingly by their tail vertex. For each lower triangle $\{x, y, z\}$ of an arc (x, y) we check whether the lower triangle path through z is shorter than (x, y). If it is shorter, we update the metric to reflect the shorter weight: $m_C(x, y) \leftarrow m_C(x, z) + m_C(z, y)$. In this case we say (x, y) *derives* its weight from (z, x) and (z, y).

After this processing step m_C is customized. By definition z is below both x and y in every lower triangle $\{x, y, z\}$ of (x, y). The weights $m_C(z, x)$ and $m_C(z, y)$ have therefore already been computed and $m_C(x, y) \leq m_C(z, x) + m_C(z, y)$ holds. After all arcs on the level of z have been processed, $m_C(z, x)$ and $m_C(z, y)$ do not change until the algorithm terminates. Inductively we can conclude that after termination the lower triangle inequality holds for all lower triangles $\{x, y, z\}$ in the vertex contraction.

3.3.2 Perfect Customization

The perfect metric m_P can be obtained from a customized metric m_C. First, m_P is initialized with the weights from m_C. Then we iterate all arcs (u, v) ordered decreasingly by the level of u and enumerate the respective upper and intermediate triangles $\{u, v, z\}$. When the path through z is shorter than the current edge weight we set $m_P(u, v) \leftarrow m_P(u, z) + m_P(z, v)$. After this processing step the metric is the perfect metric as shown in [5]. See figure 3.3c for an example.

3.4 Witness Search

Not all edges in $G^{\wedge}_{\pi}$ are needed to preserve the shortest path distance for all pairs of vertices given a customized metric. Removing these arcs from the graph does not change the length of shortest paths. If shortest paths are unique, we can remove all arcs that are not part of a shortest path. If for an edge an upper or intermediate triangle with lower length exists, it can be deleted from the graph. The triangle induces a shorter up-down path which can be used instead of the arc. When shortest paths are not unique, we only need to preserve a single shortest path for every pair of vertices. In this case we also delete some arcs that are part of a shortest path. Examples of both can be found in figure 3.3c.

We can identify the unnecessary arcs during perfect customization. If an arc has a upper or intermediate triangle path which is at most as long as the arc itself, we mark the arc for deletion. After perfect customization we delete all marked arcs from the graph. This removes all arcs not necessary to preserve all shortest paths distances as shown in [5].

Note that this can result in two different graphs if upward and downward weights differ.

There might however be multiple possible up-down s-t-paths. The previous approach will not delete all edges that could be removed, as illustrated in figure 3.4.

Instead of deleting arcs from the graph it is possible to set their length to infinity in order to avoid these edges being considered in queries. This approach did not prove to be very useful in experiments as it did not produce any noticeable speedup compared to the perfect metric.

If we only set the weight to ∞ however, the arc still needs to be scanned. This results in more arcs being iterated than if we actually delete the arc.

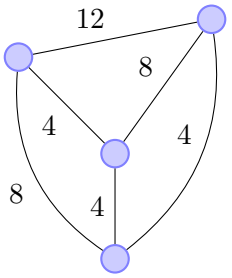


Figure 3.4: In this graph there are two shortest paths between the bottom vertex and the upper left vertex. The leftmost edge is not needed to preserve their distance.

3.5 Query

A distance query computes the distance between two vertices s and t in $G_\pi^\wedge$ given a customized metric. The query should also yield the corresponding shortest up-down-path in $G_\pi^\wedge$, which can be transformed to a shortest path in the input graph.

The subgraph of $G_\pi^\wedge$ reachable from a vertex v is called the *search space* $\mathrm{SS}(v)$. We can restrict shortest s-t-path queries to the respective search spaces. If an up-down-path from s to t exists, the forward part of that path lies within $\mathrm{SS}(s)$ and the downward part lies within $\mathrm{SS}(t)$. In a bidirectional search we can thus restrict the forward search from s to $\mathrm{SS}(s)$ and the backward search from t to $\mathrm{SS}(t)$.

3.5.1 Basic Query

The basic query performs a bidirectional Dijkstra search from s and t. We use the upward metric m_u for the forward search and the downward metric m_d for the backward search. The metrics are the same if the input metric is symmetric. Both searches operate on $G_\pi^\wedge$.

If the weights in the graph are not symmetric, the witness search yields different graphs for upward and downward weight. Hence we cannot use the same graph for both forward and backward search. Instead, we use the graph corresponding to the upward weight as forward search graph and the graph corresponding to the downward weight as backward search graph. This works because both graphs share the same vertices and only the arcs are different.

3.5.2 Elimination Tree Query

The dijkstra based query spends a lot of time updating the priority queue. We can use a different approach without a priority queue by taking advantage of the elimination tree.

This approach also uses a bidirectional search. We maintain an array of tentative forward and backward distances. In order to find a path between two points s and t we first determine their lowest common ancestor z in the elimination tree $T_{G_\pi^\wedge}$. Then we expand all outgoing arcs of vertices on the path between s and z. We repeat this process for the path from t to z. In a final step we expand all outgoing arcs of vertices on the path from z to the root of T. The last step is also used to find the vertex with the lowest distance between s and t. Note that a vertex higher up in the elimination tree can have a lower tentative distance than its children, e.g. in figure 3.5 vertex 7 has forward distance of 17 while its parent vertex 6 has forward distance of 18.

After each query we add an addition cleanup step to avoid the initialization time for the tentative distance arrays. For each vertex on the paths from s and t to the root we reset the tentative forward and backward distances to infinity.

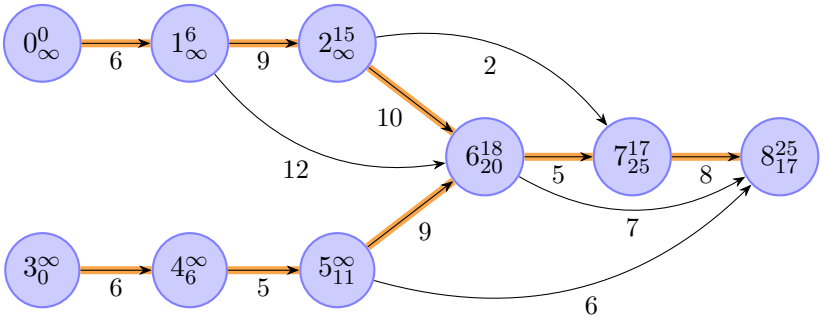


Figure 3.5: Elimination tree query from $s = 0$ to $t = 3$ on a graph with elimination tree highlighted in orange. The forward and backward search meet at $z = 6$. The nodes are labelled with their tentative forward (superscript number) and backward distances (subscript number).

The elimination tree query has the disadvantage of visiting the entire search spaces of s and t. However, it performs better on random queries than the Dijkstra-based query, as shown in [5] and in our own experiments in table 6.5.

We can perform an elimination tree query after deleting arcs in witness search. However, the resulting graph does not necessarily have an elimination tree as it is not necessarily chordal. Instead, we use the elimination tree of the vertex contraction like before.

After pruning arcs, a vertex v might have an ancestor in the tree that is not part of $\mathrm{SS}(v)$. The query is correct if all vertices in the search space of v are ancestors of v in the elimination tree because all vertices in the search space of v are visited if the are ancestors in the elimination tree. This is true because deleting arcs does not add vertices to the search space of v.

In figure 3.5 the arc $(2, 6)$ is not part of a shortest path because $2 \to 7 \to 6$ is shorter. Deleting it does not change any of the tentative distances. Figure 3.5 also illustrates why we cannot compute a new tree using the elimination tree rule for the pruned graph. In the new tree, vertex 6 would not be an ancestor of vertex 0 even though it is in its search space because the arc $(2, 6)$ is removed after witness search.

4. Sequential Optimizations

In this section we will present several optimizations and implementation details that improve the sequential performance of CCH customization.

4.1 Storing the Graph

The data structures used to store the graph have a great influence on the running time of the algorithms. Like Dibbelt et al. in [5], we reorder the vertices according to the vertex order in an adjacency array when constructing the vertex contraction. A vertex v in the input graph is mapped to a vertex $v' = \pi^{-1}(v)$ in the vertex contraction. The outgoing arcs of each vertex in the vertex contraction are ordered by their heads. We order the arcs in the adjacency array by their head's ids. The improvements of query time by reordering vertices has already been observed in [5]. Reordering the vertices and arcs in the graph data structure also raises customization performance because it improves cache locality. When performing a query we map the source and destination vertices in the input graph to their counterparts in the vertex contraction. This requires only a single additional memory access per vertex which is a negligible overhead.

4.2 Triangle enumeration

The outgoing arcs of each vertex are ordered by their head's ids, which, after reordering, is the same as the rank. In both basic and perfect customization sequential running time can be improved by exploiting this ordering to accelerate triangle enumeration.

When enumerating upper and intermediate triangles of an arc (x, y) we optimize the calculation of the neighbourhood intersection. In upper triangle enumeration we can skip those outgoing arcs of x whose head is below y. As outgoing arcs are ordered by head-id this can easily be achieved by starting enumeration at (x, y). This way on average only about half as many arcs have to be iterated to find the neighbourhood intersection.

In the intermediate triangle enumeration we can skip all outgoing arcs of x whose head is above y. We stop enumeration when we reach (x, y).

Computing the upward neighbourhood intersection only requires the upward directed graph for both endpoints. Using only one graph instead of two as when computing the intermediate neighbourhood intersection increases cache efficiency. Arcs in the downward graph need to be mapped to the corresponding upward arcs which requires additional memory acceses.

This is not necessary with the upward graph, hence upper triangle enumeration can be implemented more efficiently than both lower and intermediate triangle enumeration.

We can also make use of the fact that an upper triangle $\{x, y, z\}$ of an arc (x, y) is also an intermediate triangle of (x, z). Both (x, y) and (x, z) share the same tail vertex which means they start on the same level. Hence, we can combine enumeration of upper and intermediate triangles in perfect customization. Instead of enumerating upper and intermediate triangles separately, we only enumerate upper triangles. In addition to the metric update for (x, y), we also perform a metric update for (x, z) using the intermediate triangle path $x \to y \to z$.

Lower triangle enumeration cannot be accelerated by skipping arcs. During basic customization we can however replace lower triangle enumeration with upper triangle enumeration. We compute a customized metric m_C as follows: We iterate all arcs $(u, v) \in E$ ordered increasingly by the rank of u. For each arc (u, v) we enumerate all upper triangles $\{u, v, w\}$. The triangle $\{u, v, w\}$ is a lower triangle of (v, w). If the lower triangle path $u \leftarrow v \leftarrow w$ is shorter than (v, w) we update $m_C(v, w) \leftarrow m_C(v, u) + m_C(u, w)$ like we did before.

Theorem 4.1. *The metric m_C fulfils the lower triangle inequality after the last arc has been processed.*

Proof. We can show this by inducing over the levels. Our induction hypothesis is that when our algorithm reaches a level l, all lower triangles of arcs in or below that level fulfil the lower triangle inequality.

By definition arcs on the lowest level do not have lower triangles, hence they do not violate the lower triangle inequality.

A lower triangle $\{u, v, w\}$ of an arc (u, v) on level $l(u)$ has a third point w which by definition is on a level $l(w) < l(v)$. As our algorithm iterates levels in ascending order we must have already processed the arc (w, u) at some point which has the upper triangle $\{u, v, w\}$. The triangle has hence been processed before and fulfils the lower triangle inequality.

When we reach the highest level the lower triangle inequality holds for all levels. □

4.3 Path Unpacking

Retrieving the shortest path between two points in addition to the distance is essential for most applications of route planning. Neither the basic query nor the elimination tree query reveal the shortest path in the input graph directly, but both can be used to compute the shortest up-down-path by using a parent array for forward and backward search.

We call this up-down-path the *packed path*. It can be unpacked to obtain the shortest path in the input graph. In the original CCH paper[5] single arcs in the vertex contraction are unpacked by enumerating lower triangles in order to find the constituent arcs. A path is unpacked by recursively unpacking shortcuts until the original arcs have been found.

Instead of performing an additional triangle enumeration, we propose to store the constituent arcs during basic customization. This way we can avoid some overhead in the query phase while using more memory.

Arcs need to be unpacked in different ways depending on the direction of traversal. If an arc (u, v) is traversed forwards in the path, the first constituent arc (w, u) is traversed backwards. The second constituent arc (w, v) is traversed forwards again. When unpacking a forward arc (w, u) appears before (w, v)

Likewise, if (u, v) is traversed backward (w, u) is traversed forward and (w, v) is traversed backward. In this case (w, v) appears before (w, u).

4.3.1 Unpacking Upper and Intermediate Triangles

In the perfect metric some arcs derive their weight from an upper or intermediate triangle instead of a lower triangle. Using the original algorithm we would have to enumerate upper and intermediate triangles in addition to the lower triangles when unpacking arcs. With our approach we only need to update the constituent edges during perfect customization.

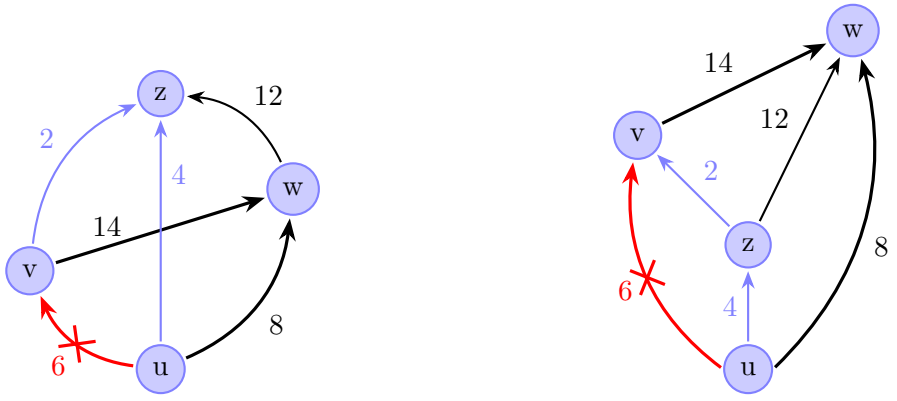
We cannot unpack upper and intermediate triangle paths the same way as lower triangle paths because the constituent arcs are traversed in different directions. A shortcut (u, v) deriving its weight from an intermediate triangle has two constituent arcs which are traversed in the same direction as (u, v). When the shortcut derives its weight from an upper triangle, the second constituent arc is traversed in the opposite direction.

Unpacking upper and intermediate triangles is only relevant with a perfect metric. The customized metric resulting from basic customization only depends on lower triangles. In witness search we delete all arcs depending on upper or intermediate triangles. In both cases we do not need to unpack upper or intermediate triangle paths.

4.3.2 Witness Search

Path unpacking based on lower triangles is still possible with perfect customization and witness search. Recall that an arc (x, y) is marked for deletion in witness search if there is an upper or intermediate triangle $\{x, y, z\}$ such that the shortest path from x to y over z is as most as long as (x, y). The arc (x, y) can however be part of a lower triangle of another arc (y, w) which dervies its weight from the corresponding lower triangle path.

We have to make sure no deleted triangles are stored to ensure the unpacked path exists. If we delete one of the lower arcs of a lower triangle and keep the upper arc, the respective lower triangle path is destroyed, as illustrated in figure 4.1. In the following we will show that a shortcut is always deleted if all lower triangle paths with the same length are destroyed in witness search.



(a) The arc (u, v) is deleted because its upper triangle path $u \to z \to v$ also has length 6. It is part of the lower triangle $\{u, v, w\}$ of the shortcut (v, w). By deleting (u, v) the only lower triangle of (v, w) is destroyed. However, the path $v \to z \to w$ has the same length as (v, w) so this arc is not necessary either.

(b) The path $u \to z \to v$ is part of an intermediate triangle of (u, v). Because it is no longer than (u, v), we can delete (u, v). However this destroys a lower triangle of (v, w) with the same length as (v, w). There is however another path with the same length in the lower triangle $\{v, z, w\}$.

Figure 4.1: Deleting arcs in the vertex contraction destroys the lower triangle $\{u, v, w\}$ of (v, w) drawn with thicker arcs. In both cases it does not matter which of v and w has higher rank.

Theorem 4.2. *After witness search every remaining shortcut (x, y) has a lower triangle path $x \to z \to y$ in the remaining graph with $m_P(x, y) = m_P(x, z) + m_P(z, y)$.*

Proof. Perfect customization processes arcs level-wise, starting at the highest level proceeding towards lower levels. We show that on no level we mark an arc for deletion that is part of the last remaining lower triangle of an arc not marked for deletion.

We will consider the two cases in which an arc is deleted separately. If an arc (u, v) is deleted because of an upper triangle path, we will show that all shortcuts deriving their weight from a lower triangle path containing (u, v) must have been marked for deletion. In case of an arc (u, v) deleted because of an intermediate triangle path we will prove for all shortcuts (v, w) and (w, v) depending on (u, v) the existence of another lower triangle path with the same length.

An upper or intermediate triangle $\{u, v, z\}$ can contain a shortcut (v, z) or (z, v) deriving its weight from the same triangle. Note that in this case neither (u, v) nor (u, z) will be deleted $\{u, v, z\}$ if all weights are positive.

We will first consider the upper triangle path.

If an arc (u, v) is deleted because of its upper triangle $\{u, v, z\}$, all arcs (v, w) and (w, v) deriving their weight from the lower triangle $\{v, w, u\}$ will also be deleted. In a contraction hierarchy w and z must be connected with an arc. This arc has the lower triangle $\{u, w, z\}$, hence $m_P(z, w) \leq m_P(z, u) + m_P(u, w)$. We can conclude that the paths $v \to u \to w$ is at least as long as $v \to z \to u \to w$. The path $v \to z \to w$ in turn is at most as long as the latter. If w is below z the path $v \to z \to w$ is an upper triangle path of (w, z). It is an intermediate triangle of (z, w) if w is above z. In both cases $v \to z \to w$ is a witness path. We can thus delete the arc connecting w and z.

An arc (u, v) is also marked for deletion if it has an intermediate triangle $\{u, v, z\}$ which induces a path of lower or equal length through z. In this case all arcs (v, w) and (w, v) that derive their weight from (u, v) must have another lower triangle of the same length. If such an arc exists the arc (z, w) must exist too as the vertex contraction is a chordal graph. It is part of the triangle $\{v, z, w\}$ which is a lower triangle of (v, w). The path $v \to z \to w$ can not be longer than $v \to u \to w$ because $v \to z \to u$ is at most as long as $v \to u$ and $\{z, w, u\}$ is a lower triangle of (z, w). So the arc (v, w) can derive its weight from $\{v, z, w\}$ instead of $\{v, u, w\}$.

□

We can avoid having to update stored lower triangles if an arc is deleted by choosing the highest lower triangle in basic customization.

The theorem holds true for asymmetric weights. The proof works analogously in that case.

5. Parallelization

Several steps in the CCH algorithm can be executed in parallel to increase performance. In this work we will concentrate on the customization phase. Some approaches can however easily be transferred to other steps as well.

5.1 Basic Customization

As described in [5] basic customization can be parallelized by processing the edges on each level in parallel. Each thread processes a block of edges and enumerates the respective arcs. Synchronization is only needed between levels.

This method is correct because lower triangles only depend on arcs in lower levels. As long as only arcs on the same level are processed at a time, no conflict can occur.

The respecting metric can be constructed in parallel by distributing blocks of arcs to the threads. Each thread then processes its respective block of arcs by copying weights from edges in the input graph to the respective arcs in the contraction hierarchy. To avoid data races in case of multi-edges in the input graph we enumerate constituting edges of arcs in the contraction hierarchy. If we enumerate edges in the input graph in parallel, we need to make sure multi-edges are processed by the same thread. Otherwise, two threads might access the same arc in the contraction hierarchy simultaneously, resulting in a data race if not synchronized.

Sequential basic customization can be accelerated by enumerating upper triangles instead of lower triangles. When enumerating upper triangles, a weight can be updated by two threads simultaneously possibly resulting in a data race. We could avoid these data races using locks, adding a considerable overhead.

Algorithm 5.1: atomicAssignMin(w: int32, other: int32) atomically assigns the minimum of w and other to w. The operator & is the C addres-of operator

```
1 expected ← w;
2 while w ≥ other and not atomicCompareExchange(&w, &expected, other) do
3 |   expected ← w;
```

We implement edge weights as an array of 32 bit integers. Most CPUs offer an atomic compare and exchange instruction for integral data types. It compares the contents of a

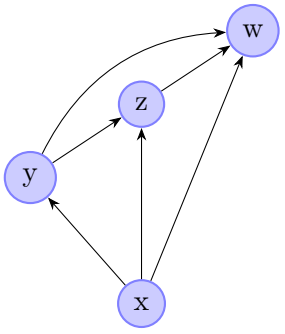


Figure 5.1: All outgoing arcs of x are on the same level. When processing them in parallel we might update (x, y) with a weight derived from (y, z) and (x, z) while writing a new weight to (x, z) at the same time.

destination pointer with an expected value and writes a new value if both are the same. We can use this instruction to atomically assign the minimum of two weights in basic customization using algorithm 5.1.

When storing path unpacking data we need to update the constituent arcs in addition to the edge weights. We need to write all of them atomically. Luckily most modern 64 bit processors have an atomic compare and exchange instruction for 128 bit fields. Before performing the triangle enumeration we pack the weight and the constituent arcs into a 128 bit integer and use the atomic exchange as before. To make sure we always use the highest lower triangle we can make use of the ordering of the graph. Outgoing arcs of higher arcs appear later in the adjacency array hence they have a higher index. As we want to find the highest lower triangle with lowest weight we reverse the indices by using the negative value when packing them to a 128 bit integer. The weight is written to the most significant 32 bits. The reversed indices of the constituent arcs are written to the least significant 32 bits.

Using this packing scheme allows to use upper triangle enumeration without locks for basic customization. Path unpacking information obtained here is ready for perfect customization and witness search.

5.2 Perfect Customization

The perfect customization can be parallelized similar to the basic customization. We iterate over the edges in each level in parallel, synchronizing threads between levels. This approach has been proposed by Dibbelt et al. in [5] who also showed its correctness. It relies on data being written atomically. We use an adjacency array to map the level directly to the respective arcs.

We can use a closely related approach for enumerating the arcs on each level. Instead of distributing the arcs on each level to the threads directly, we can distribute the vertices on each level to the threads and enumerate the outgoing arcs. In this case the arcs departing from each vertex are iterated in the same thread. No conflicts can occur here because no two vertices in upper and intermediate triangles can be on the same level. This has the advantage of being correct with metrics that cannot be written atomically. If the metric is not written atomically, data races can occur, as illustrated in section 5.2.

We can also parallelize the combined triangle enumeration. An upper triangle $\{x, y, z\}$ of (x, y) is an intermediate triangle of (x, z). If we distribute the vertices on each level to the threads this approach does not need any additional synchronization, because both arcs share the same tail vertex. In this case no data races can occur. In order to avoid data races when enumerating the arcs on each level directly, we make use of the atomic minimum operation described above. This allows parallelization by distributing the edges directly to the threads without additional locks. Pseudocode for this approach is given in algorithm 5.2.

Note that when performing a witness search, we do not need to update the constituent arcs. If we do not perform a witness search on top of perfect customization, we need to update the constituent arcs though. This requires the weight and the constituent arcs to be written atomically. We can use 128 bit operation to achieve atomicity, similar to the approach we presented for basic customization.

Our experiments in section 6.2 showed that parallelization by distributing vertices and distributing arcs perform similarly with a lower number of threads.

Algorithm 5.2: Perfect customization with combined triangle enumeration updating both upward and downward weights.

```
1 for l ← 0 to maxLevel do
2     for each (u, v) ∈ L_E(l) do in parallel
3         for each upper triangle {u, v, w} of (u, v) do
              /* Update weights for upper triangle                    */
4             atomicAssignMin(m_P(u, v), m_P(x, z) + m_P(z, y)) ;
5             atomicAssignMin(m_P(v, u), m_P(y, z) + m_P(z, x)) ;
              /* Update weights for intermediate triangle             */
6             atomicAssignMin(m_P(x, z), m_P(x, y) + m_P(y, z)) ;
7             atomicAssignMin(m_P(z, x), m_P(z, y) + m_P(y, x)) ;
```

5.3 Witness search

As witness search is done in combination with perfect customization, we can employ the techniques used there for witness search too. In addition to updating the weights, we mark arcs for deletion if we find a triangle inducing a shorter path. In order to avoid the more complex synchronization of bitsets, we use a full byte for each arc. No conflicts can occur here as the flags are only set and never reset. Hence the order in which threads write the flags does not matter as long as any of successfully flags the arc for deletion.

We can also use parallelization in order to build the new graphs for upward and downward weight. The number of edges deleted before a specific edge determines its location in the new adjacency array. This number can be calculated using the prefix sum of the deletion markers. For each vertex we compute the new edge ranges by subtracting the number of edges deleted up to the left and right borders respectively. The edges not marked for deletion are then be copied to their new locations in parallel. All steps except for the prefix sum can be parallelized trivially by dividing arcs and vertices into contiguous blocks which are then processed by the threads.

5.4 Prefix sums

Efficient parallelization of prefix sums is not trivial. They are usually implemented by scanning through the array sequentially. Some fundamental changes need to be made to enable parallelization.

The approach proposed by Blelloch in [2] achieves a theoretical running time of $O(n/p+\log n)$ on an array of length n with p threads. It divides the prefix sum calculation into an up-sweep and a down-sweep. In the up-sweep partial prefix sums are calculated which are combined in the down sweep. Unfortunately, on a modern CPU the overhead of synchronization and of steps introduced to make the algorithm parallelizable far exceeds the sequential running time.

We also tested another approach: We calculate the prefix sum on p contiguous blocks of equal size in parallel. Then we calculate a sequential prefix sum over the maximums of each block, i.e. the last elements of each block. The blocks are then incremented by this block-wise prefix sum. The pseudocode is presented in algorithm 5.3. This approach has been described in [10]

Algorithm 5.3: Parallel Prefix Sum

```
   Data: a: array[N ]
 1 prefix ← array[NumThreads()];
 2 begin parallel section
 3     id ← CurrentThreadID();
 4     left ← ⌊N · id/NumThreads()⌋;
 5     right ← ⌊N(id + 1)/NumThreads()⌋;
 6     PrefixSum(a, left, right);
 7     prefix [id + 1] ← a[right − 1];
 8     barrier begin sequential section
 9         PrefixSum(prefix, 0, NumThreads());
10     for i ← 1 to N do
11         a [i] ← a [i] + prefix [id]
```

Algorithm 5.3 has a complexity of $O(n/p + p)$, which is worse than the other approach. In practice we have $p \ll n$ so this is not really a problem. However, it has less parallelization overhead than the one above and is a lot faster in practice on a regular CPU. Unfortunately prefix sum calculation is mainly limited by memory bandwidth, so even with efficient parallelization we could achieve any speedup in our experiments in section 6.3

6. Experiments

Machine and Compiler: All algorithms have been implemented in C++ and compiled with clang 5.0.1 with compile option `-O3`. We ran our experiments on a dual-CPU machine with two Intel® Xeon® E5-2680 16 core processors with two threads per core, clocked at 2.7 GHz with a total of 256 GB of DDR3-RAM. It has 32 KB of L1 cache, 256 KB of L2 cache and 20.48 MB of L3 cache.

Instance	#Vertices	#Arcs	#Edges
Europe	18017748	42560275	22212039
Germany	4377307	10736200	5483012
Belgium	462843	1112155	591480
Karlsruhe	120481	304935	154856

Table 6.1: Comparison of the different test instances.

Graph Instances: In our experiments we used four instances of different sizes. Our main test instance is the Western European road network from the PTV Europe graph[1]. The other three instances are extracts from that graph, one of Germany, one of Belgium. The last one covers a greater region around the city of Karlsuhe with a latitude between 48.3° N and 49.2° N and a longitude between 8° N and 9° E. Table 6.1 gives an overview of the sizes of the different instances.

In our experiments we used nested dissection orders obtained using the Inertial Flow graph partitioner[9] with a balance of 0.3.

We tested the different parallelization approaches described in chapter 5. As this work is focussed on the customization phase we did not run experiments concerning the CCH construction.

6.1 Basic Customization

In figure 6.1 we compare the running times of basic customization using lower triangle enumeration and using upper triangle enumeration. In both cases we parallelized by distributing the arcs on each level directly to the threads. Even though using upper

[1] `https://i11www.iti.kit.edu/resources/roadgraphs.php`

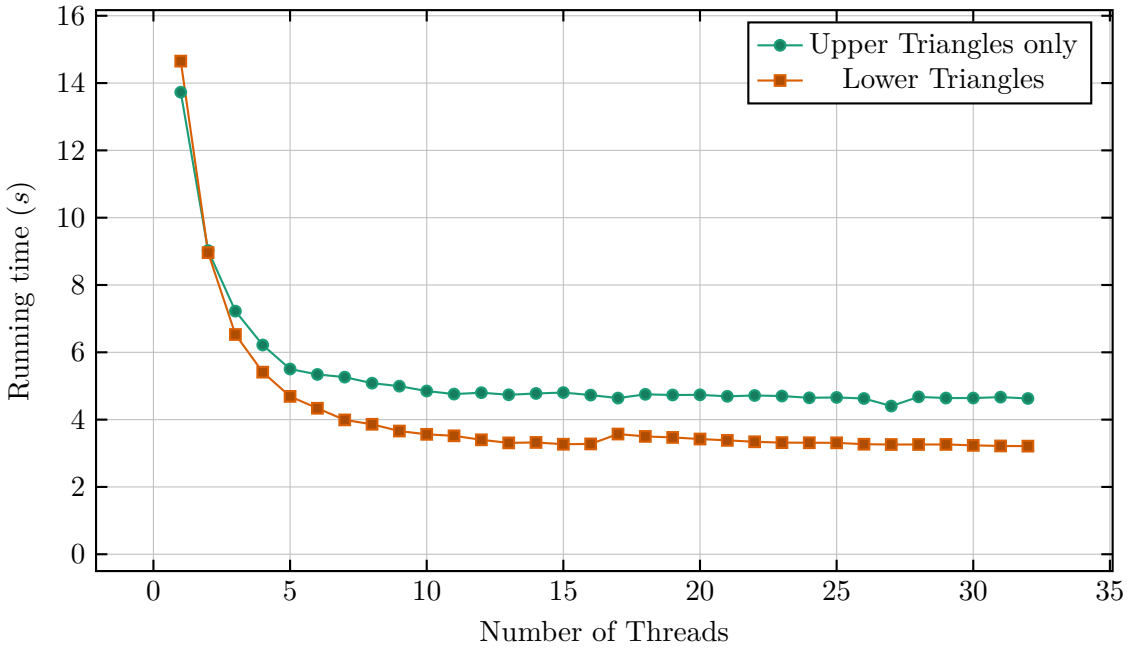


Figure 6.1: Running time of basic customization by enumerating upper and lower triangles on Europe.

triangle enumeration is slightly faster when using a single thread, enumerating lower triangles performs better when using more threads. This is most likely due to the atomic 128 bit operations which are not necessary with lower triangle enumeration. With a higher number of threads atomic memory accesses become more and more inefficient.

The small dent in the plot when using more than 16 threads can also be observed in other plots in this chapter. It corresponds to the number of physical cores of our test machine.

Graph	#Arcs in CCH	#Triangles	#Lower Tris (20% Levels)	#Upper Tris (20% Levels)
Europe	70362408	655640384	408303719	467071645
Germany	17841703	158081590	83172653	98960828
Belgium	2134434	18822408	9699709	12096320
Karlsruhe	497487	2291333	1210709	1462264

Table 6.2: Sizes of the vertex contractions of different graphs. All use an ND-order computed using the Inertial Flow partitioner. We counted the total number of triangles as well as the number of triangles up to a certain level. Note that the numbers of upper and intermediate triangles are the same for every level.

We also investigated the impact of triangle precomputation on basic customization. The results for all graphs are shown in table 6.3. We report the data for Europe additionally in figure 6.2. Interestingly precomputing triangles is considerably slower than performing the optimized triangle enumeration on all larger graphs. On Karlsruhe at least the sequential basic customization is slightly faster with triangle precomputation. When using a higher number of threads, basic customization using triangle precomputation performs about two times worse with all test instances. This might be due to more cache misses when iterating the array containing the triangles because this array is much larger than the graph's adjacency array as seen in table 6.2. Our implementation checks for each arc whether it is below the threshold level. Performance with precomputation might be better

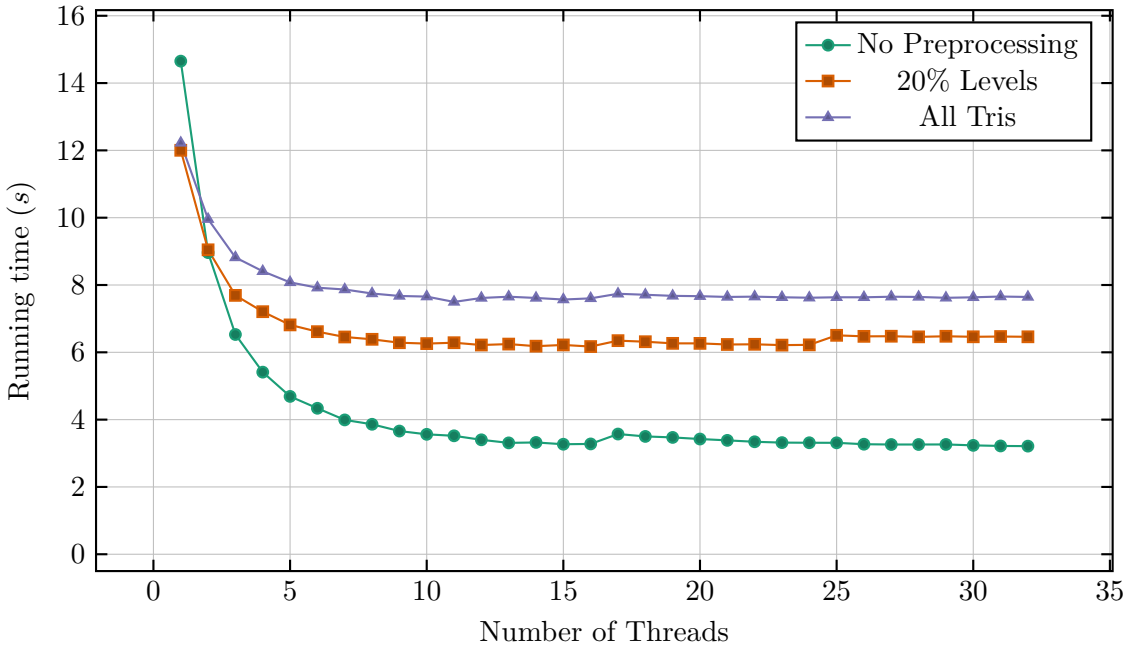


Figure 6.2: Running time for basic customization on Europe with travel time using different levels of triangle precomputation.

with an implementation which does not perform this additional check for each arc but only once for each level.

	Karlsruhe		Belgium		Germany		Europe	
#Thr.	none	all	none	all	none	all	none	all
1	85	131	406	704	3.483	5.721	14.652	24.160
2	51	73	227	377	2.081	3.361	8.962	14.534
4	26	38	144	226	1.284	1.990	5.411	8.533
8	20	32	107	170	0.898	1.384	3.862	5.887
16	27	31	137	148	1.131	1.212	4.922	4.781

Table 6.3: Running time in seconds for basic customization using upper triangle enumeration with and without triangle precomputation

6.2 Perfect Customization

In figure 6.3 we compare different optimizations of perfect customization with witness search. Parallelizing by distributing the arcs directly to the threads performed a bit better than distributing blocks of vertices when using more threads. When enumerating the incident arcs of a vertex we need more memory accesses than when enumerating the arcs so the memory bandwidth is saturated faster than when enumerating the arcs directly.

Combining upper and intermediate triangle enumeration accelerates perfect customization by about one third when executed sequentially. When enumerating intermediate triangles separately sequential perfect customization takes about 36.4 s compared to 25.7 s when combining upper and intermediate triangle enumeration. With a higher number of threads the advantage shrinks until vertex-wise parallelization is faster. This is most likely due to the atomic operations required in edge-wise parallelization.

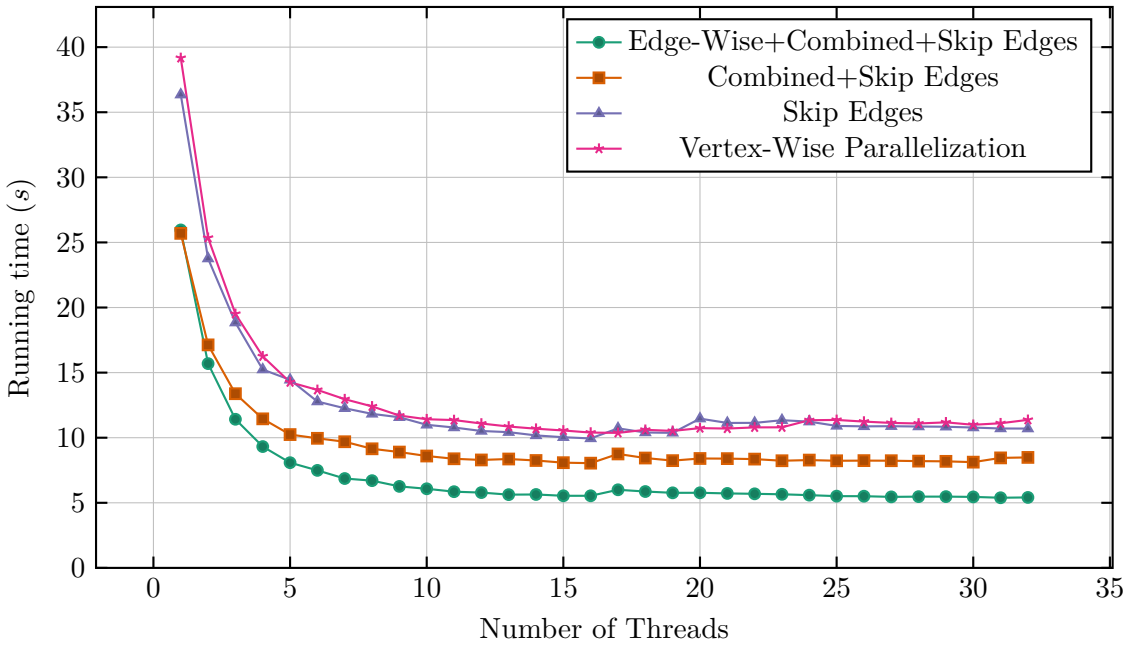


Figure 6.3: Performance of different optimizations of perfect customization with witness search on a varying number of threads with witness search. Note that perfect customization is based on basic customization and thus includes its running time.

In figure 6.4 we report the speedup corresponding to the running times in figure 6.3 compared to the respective sequential single threaded running times. The best speedup is achieved by edge-wise parallelization compared to vertex-wise parallelization. This approach does not only have the best running times it also achieves a higher speedup than all the other approaches. In all cases we do not benefit from using more than about 16 threads. This is most likely due to the memory bus being saturated, we can thus not read and write data faster.

	Karlsruhe (ms)		Belgium (ms)		Germany (s)		Europe (s)	
#Threads	perf.	p+w	perf.	p+w	perf.	p+w	perf.	p+w
1	122	152	659	763	5.348	6.223	22.550	25.960
2	82	78	358	409	3.242	3.664	14.032	15.691
4	37	38	209	245	1.939	2.183	8.337	9.320
8	29	30	156	173	1.392	1.554	5.982	6.701
16	27	30	137	152	1.131	1.288	4.922	5.539

Table 6.4: Running time for perfect customization (p) without and with witness search (+w). We report the customization times for all our test instances.

Performing a witness search in addition to perfect customization can increase query time as reported in table 6.5. Figure 6.5 shows a comparison of the running times of the different customization variants. Witness search setting the weights to infinity is faster than actually deleting the arcs. It is slightly faster than the regular perfect customization with four or more threads because it does not have to update the constituent arcs. On the other hand building the new graph takes less than a second. The perfect customization with witness search takes less than twice as long as basic customization.

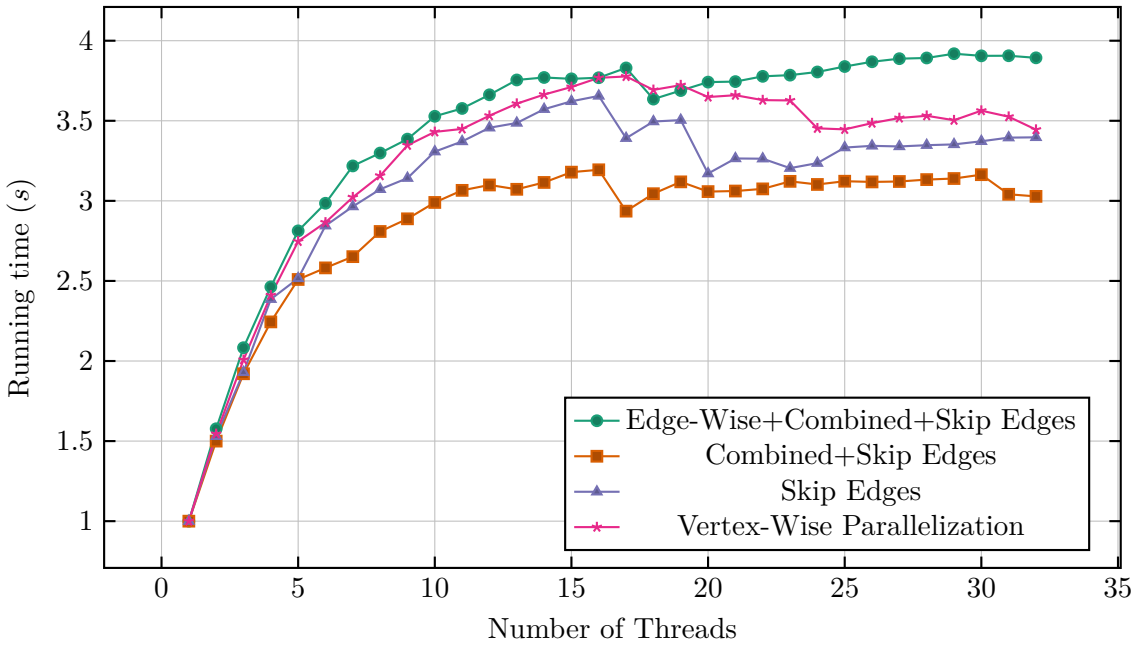


Figure 6.4: Speedup of perfect customization with witness search compared to the single threaded version.

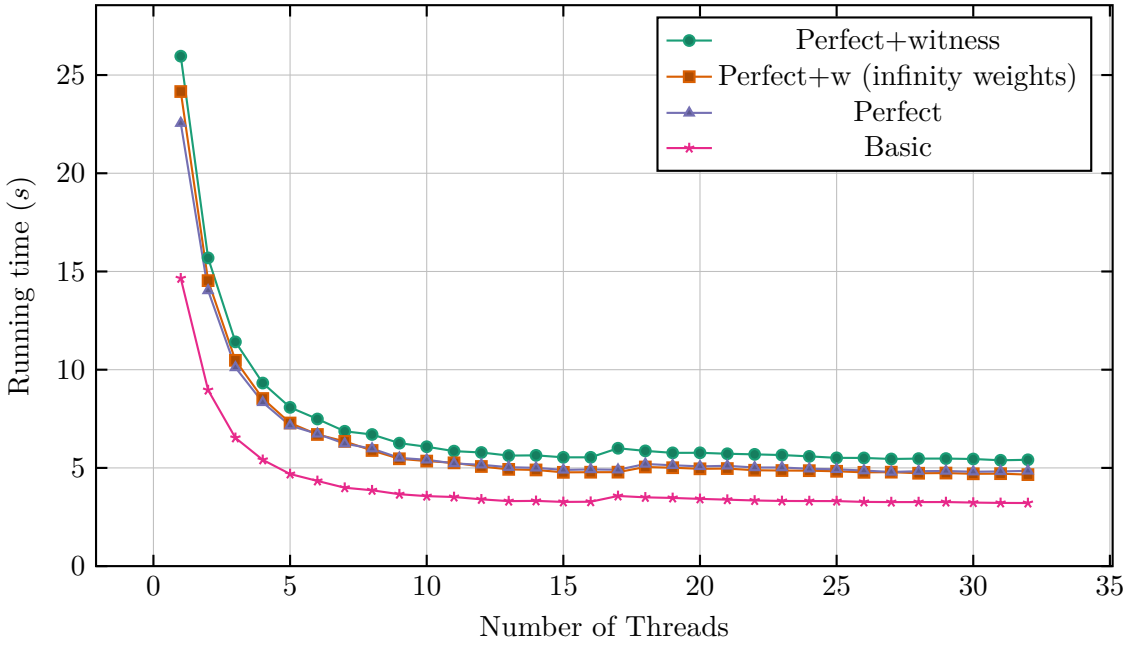


Figure 6.5: Comparison of the running times of different customization variants.

We report the times needed for perfect customization with and without witness search in table 6.4. If we compare this to the number of triangles from table 6.2 we see that customization on Europe takes about 48 ns per triangle. The time per triangle increases for the smaller graphs, Karlsruhe is as high as 83 ns per triangle, nearly twice as much.

We report the running times of queries using the different customization variants in table 6.5. The perfect customization itself does not have a big impact on query time. Setting the weight of unnecessary arcs to infinity does not really accelerate the query compared to the regular perfect metric. Interestingly using the perfect metric on Europe accelerates the elimination tree query a lot.

Instance	Customization	basic query time(ms)	basic query #visited vertices	basic query unpack(ms)	elimination tree query time(ms)	elimination tree query #visited vertices
Germany	Basic	1.617	1,064	–	1.282	1,074
	Perfect	1.587	1,083	–	0.839	1,074
	Perfect+w(∞)	1.541	983	–	0.977	1,074
	Perfect+w	1.114	983	–	0.534	1,074
Europe	Basic	3.403	1,311	0.198	1.297	1,199
	Perfect	3.418	1,328	0.198	1.266	1,199
	Perfect+w(∞)	3.372	1,218	0.198	1.266	1,199
	Perfect+w	3.037	1,218	0.214	0.702	1,199

Table 6.5: Average performance of the basic query with different variants of customization using the travel time metric. For each configuration we performed 100000 queries with start and destination chosen uniformly at random.

Actually deleting the arcs in witness search leads to a considerable acceleration of query time with both basic and elimination tree query in Europe and in Germany. The vertex search space is the same in both cases, but we do no need to touch the arcs when deleting them. This justifies the overhead of building the new graph in perfect customization. Setting the weights to infinty results in slightly faster queries than using the perfect metric, though. Hence setting weights to infinity performs better than the perfect metric in both customization and query. The elimination tree query profits much more from witness search than the basic query.

Unpacking paths is fast on both Europe and Germany. This is important for interactively drawing routes.

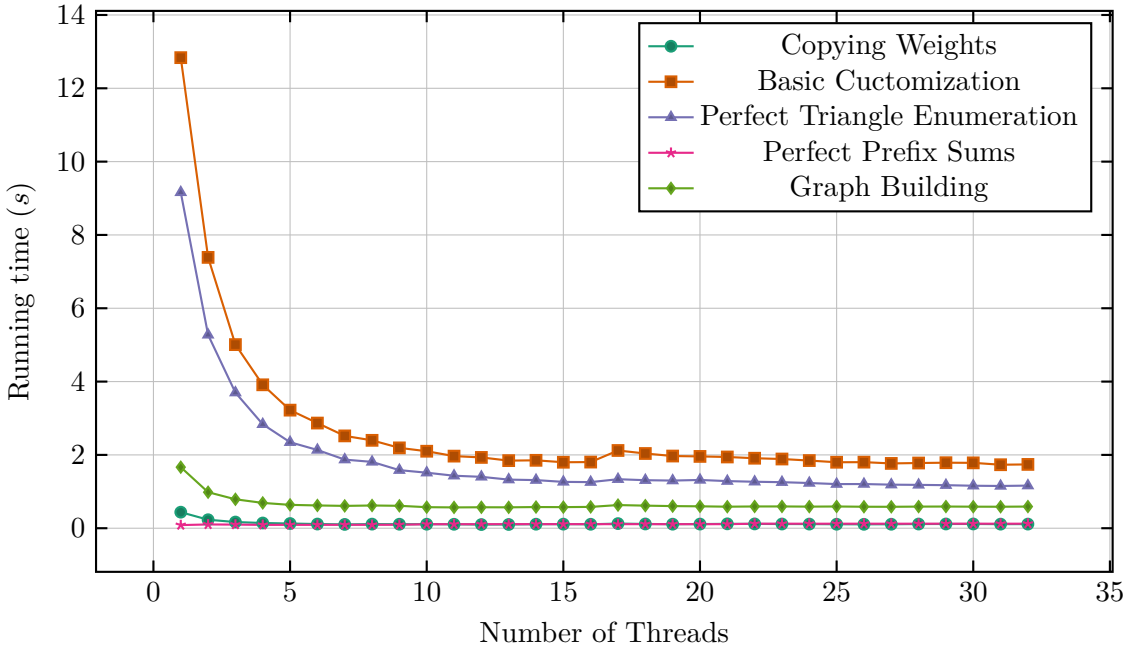


Figure 6.6: Detailed running time of perfect customization with witness search on Europe.

In figure 6.6 we report the running times of the different steps of perfect customization with witness search. All the steps can be accelerated using parallelization. Interestingly, triangle

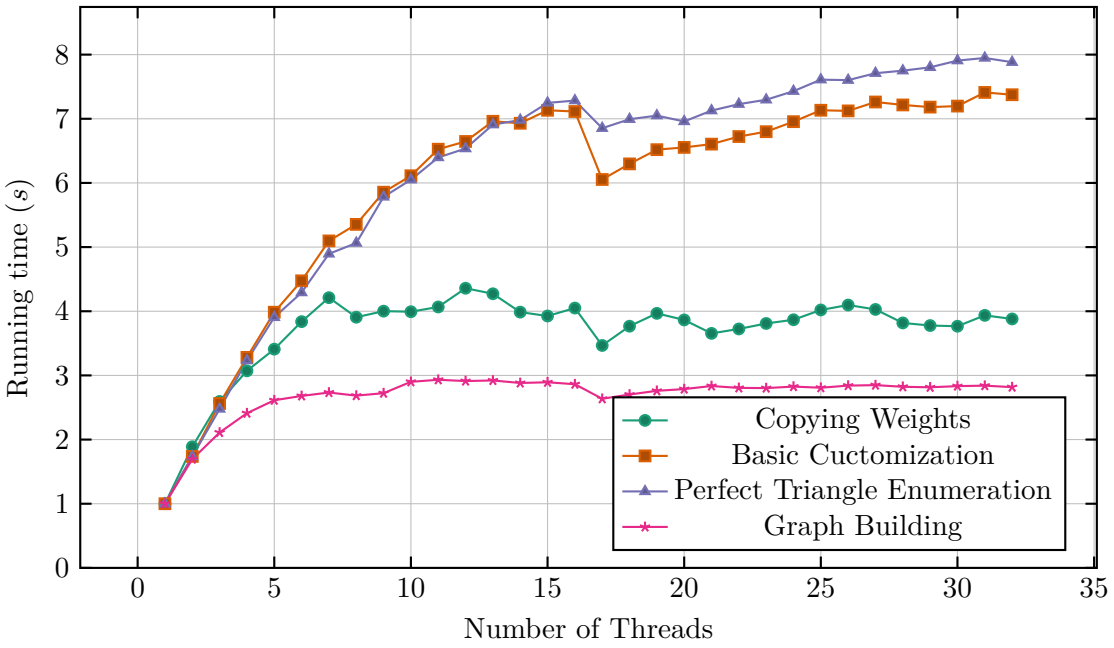


Figure 6.7: Speedup of the different steps of perfect customization and witness search on Europe.

enumeration in the perfect customization part is faster than basic customization. This might be due to the basic customization using lower triangle enumeration and constituent arcs being set in basic customization resulting in more memory accesses.

We report the corresponding speedup in figure 6.7. The construction of the respecting metric can be accelerated by a factor of 4 from about 0.4 s with a single thread to 0.1 s with six threads. It reaches its limits at six threads when memory bandwidth is exhausted. We achieved the highest speedup with upper triangle enumeration in perfect customization. It has a nearly linear speedup for up to 16 threads when it reaches its lower limit of 1.2 seconds.

Customization is accelerated a lot by reordering the vertices according to the contraction order. In figure 6.8 we report the impact of vertex reordering on perfect customization with witness search. Note that using all other optimizations the unordered version is still faster than the ordered approach using no optimizations and parallelizing vertex-wise.

In figure 6.9 we compare the running times of perfect customization with witness search using different levels of triangle precomputation. Unfortunately triangle precomputation did not accelerate the customization. When preocmputing all triangles, perfect customization takes about 1.5 times longer than without triangle precomputation when using eight or more threads. Precomputation is not very useful this way.

When precomputing all triangles for Europe we need about 3.2 GB for lower, intermediate and upper triangles each. Storing the graph uses another 2.3 GB. For every arc we store the head, the tail and two constituent arcs. In addition, we need the downward graph, as well as a mapping between the upward and the downward graph, and we need to store the metric. When performing a witness search we have two upward graphs, one for the forward search and one for the backward search. This requires about 3.4GB of additional memory. By combining upper and intermediate triangle enumeration we can avoid storing intermediate triangles. We do not need the lower triangles when we use upper triangles in the basic customization. So these modifications significantly reduce the total memory usage for precomputing triangles.

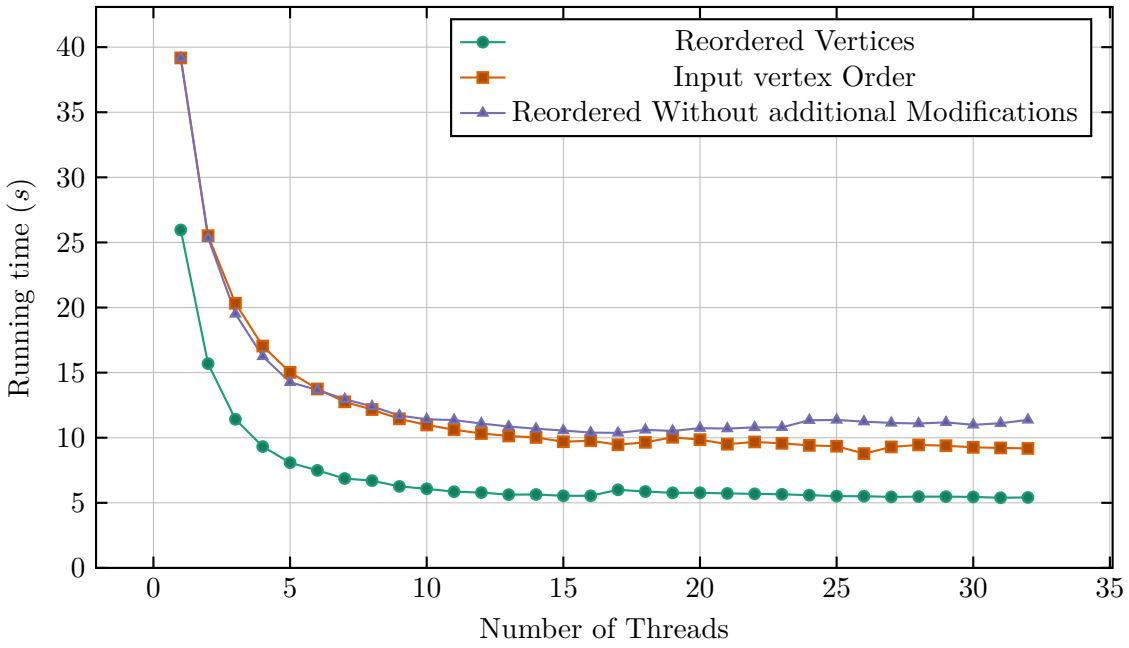


Figure 6.8: Impact of reordering the vertices on perfect customization with witness search.

# Treads	array size	Sequential	Block-Wise[10]	Two-Sweep[2]
1	10^6	$561\mu s$	$1028\mu s$	$4.3ms$
2	10^6	$561\mu s$	$974\mu s$	$4.3ms$
8	10^6	$561\mu s$	$990\mu s$	$4.3ms$
1	10^9	$2.1s$	$8s$	$19.1s$
2	10^9	$2.1s$	$4.1s$	$18.2s$
8	10^9	$2.1s$	$4.2s$	$19.0s$

Table 6.6: Running time of sequential and parallel prefix sum algorithms no an array a

6.3 Prefix Sums

Calculating the prefix sum is the only step of perfect customization with witness search we did not parallelize. The prefix sum is needed to find the indices of the remaining arcs. In our experiments we tested two different approaches to implement parallel prefix sums. Unfortunately none of them was able to beat sequential performance. With both approaches parallelization seemed not to have any effect. This indicates that the memory bandwidth is already exhausted by a single threaded prefix sum calculation.

The running time of the block-wise approach correspond to the number of additional memory accesses. When calculating the sequential sum every value is written once. Using the block-wise parallelization every value is written exactly twice.

The third approach is somewhat slower. It requires about four times more writes than the sequential variant and it is less cache efficient because it does not access memory sequentially.

Parallelizing prefix sums might be more efficient on modern machines with quad-channel DDR4-RAM which has a higher bandwidth of up to 100GB/s. On such a machine a single thread does not saturate the memory link when calculating a prefix sum and thus multithreading might accelerate the calculation. Without access to a machine with DDR4-RAM we could not verify this claim.

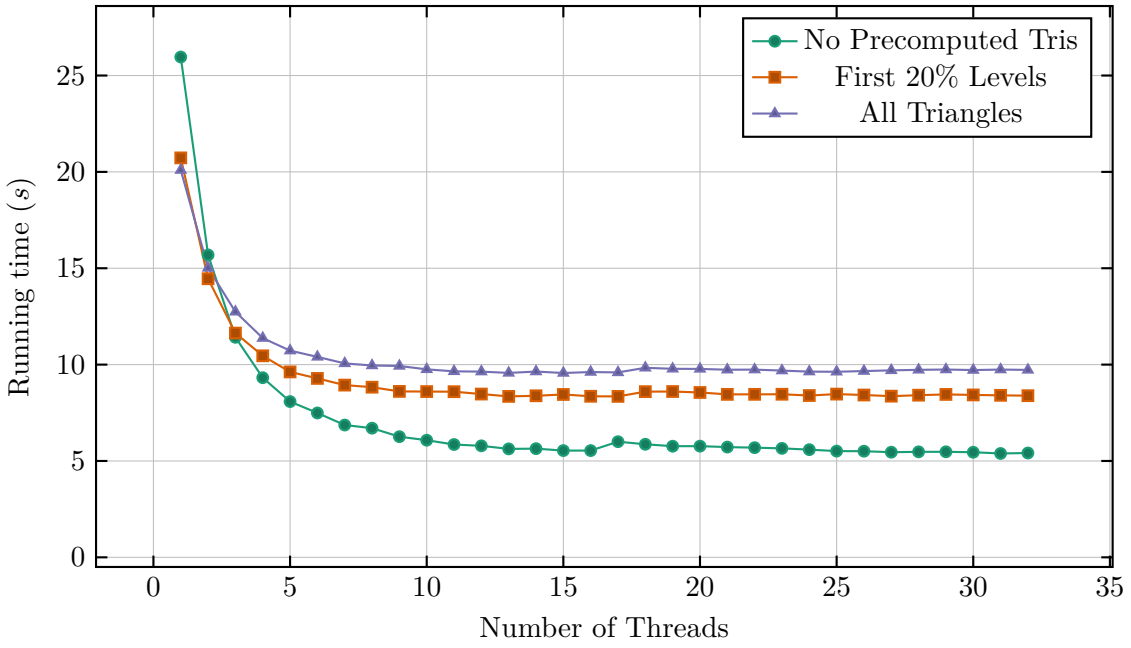


Figure 6.9: Running time for perfect customization with witness search on Europe with travel time metric using different levels of triangle precomputation.

7. Conclusion

In this thesis we designed and examined several techniques aimed at accelerating the customization phase of Customizable Contraction Hierarchies. We developed modifications for both sequential optimization and parallelization.

7.1 Summary

With our optimizations we were able to reduce the running time of perfect customization by a third. By exploiting several properties of the way the graph is stored we could accelerate triangle enumeration beyond the speed achieved with precomputed triangles. Triangle enumeration can be avoided in path unpacking by storing the constituent arcs of each shortcut. We showed that this approach works with witness search.

The sequential optimization we presented in this work can be used with parallel customization. Most of them work fine in parallel and accelerate customization even further. Only one technique did not work out as good as expected: Using upper triangles for basic customization can be parallelized but requires the use of atomic 128bit operations which makes it slower than using lower triangles.

Witness search can be done in parallel. The only step we did not parallelize is calculating prefix sums which is limited by memory bandwidth.

7.2 Future Work

A very promising method to further accelerate customization is to parallelize using the nested-dissection separator tree. The blocks induced by the separator tree have the advantage of being contigous blocks in memory which improves cache locality. In addition perfect customization can be parallelized lock free without atomic operations using the separator tree.

Parallelisation might also be used in other CCH phases. The bidirectional query can be parallelized using one thread for the forward search and one for the backward search.

Path unpacking can also be parallelized. By precomputing the unpacked length of an arc it is possible to use lock free parallelization to write the unpacked path to an array.

List of Figures and Tables

List of Figures

List of Tables

Bibliography

[1] Reinhard Bauer et al. "Search-space size in contraction hierarchies". In: *Theoretical Computer Science* 645 (2016), pp. 112–127. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2016.07.003. URL: http://www.sciencedirect.com/science/article/pii/S0304397516303176.

[2] Guy E. Blelloch. *Prefix Sums and Their Applications*. eng. Tech. rep. CMU-CS-90-190. School of Computer Science, Carnegie Mellon University, Nov. 1990.

[3] Daniel Delling et al. "Customizable Route Planning". In: *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*. Springer Verlag, May 2011. URL: https://www.microsoft.com/en-us/research/publication/customizable-route-planning/.

[4] Daniel Delling et al. "Graph partitioning with natural cuts". In: *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE. 2011, pp. 1135–1146.

[5] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. "Customizable contraction hierarchies". eng. In: *International Symposium on Experimental Algorithms*. Springer. 2014, pp. 271–282.

[6] Edsger W Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische mathematik* 1.1 (1959), pp. 269–271.

[7] Robert Geisberger et al. "Contraction hierarchies: Faster and simpler hierarchical routing in road networks". eng. In: *International Workshop on Experimental and Efficient Algorithms*. Springer. 2008, pp. 319–333.

[8] Alan George. "Nested dissection of a regular finite element mesh". In: *SIAM Journal on Numerical Analysis* 10.2 (1973), pp. 345–363.

[9] Aaron Schild and Christian Sommer. "On Balanced Separators in Road Networks". In: *Proceedings of the 14th International Symposium on Experimental Algorithms - Volume 9125*. New York, NY, USA: Springer-Verlag New York, Inc., 2015, pp. 286–297. ISBN: 978-3-319-20085-9. DOI: 10.1007/978-3-319-20086-6_22. URL: http://dx.doi.org/10.1007/978-3-319-20086-6_22.

[10] Johannes Singler. "Algorithm Libraries for Multi-Core Processors". PhD thesis. 2010.

[11] Douglas Brent West et al. *Introduction to graph theory*. Vol. 2. Prentice hall Upper Saddle River, 2001.