Customizable Contraction Hierarchies

Patrick Steil
Karlsruhe Institute of Technology

Daniel-Delong Zhang
Karlsruhe Institute of Technology

Abstract

This project focuses on Customizable Contraction Hierarchies (CCH) to enhance route planning efficiency. Implemented in C++, we conducted experiments similar to the original publications to evaluate CCH's effectiveness in solving the shortest path problem on large, continental-sized networks. Our findings emphasize CCH's adaptability and potential for revolutionizing route navigation systems.

2012 ACM Subject Classification Theory of computation → Shortest paths

Keywords and phrases Route Planning, Shortest Path, Graph Algorithms

Supplementary Material Code: `https://github.com/PatrickSteil/CCH`

1 Introduction

Efficiently determining the shortest path in large-scale networks remains a pivotal challenge across diverse domains. Whether optimizing transportation logistics or digital mapping services, the shortest path problem entails identifying the most optimal route between two points within a network. Traditional approaches, like the Contraction Hierarchies (CH) [3] algorithm, excel in preprocessing road networks and swiftly computing shortest paths. However, CH confronts limitations when dealing with metrics such as geographical distances and the lack of embedding more inherent node hierarchies in real-world networks. Moreover, its inability to accommodate dynamic edge-weight updates without recomputing the entire hierarchy restricts its practicality.

Customizable Contraction Hierarchies (CCH) [1] emerge as a solution addressing these drawbacks. CCH resolves the limitations of CH by introducing adaptability and enhanced customization. Its preprocessing time falls within the minute range, allowing for rapid construction of a dynamic hierarchy. Customization, achievable within seconds, facilitates edge weight updates without necessitating a complete hierarchy rebuild. Most notably, CCH offers exceptional query efficiency, taking only microseconds to compute regardless of the metric used, effectively overcoming the shortcomings encountered by CH.

This project delves into evaluating and implementing Customizable Contraction Hierarchies, focusing on their effectiveness in addressing the deficiencies of traditional CH algorithms. Through empirical evaluations and experimentation, this study aims to showcase the superior performance and versatility of CCH in tackling various shortest-path scenarios within networks.

2 Notation

A *graph* $G = (V, E)$ consists of *nodes* V and *edges* E, where each edge $e = (u, v)$ represents a connection between the nodes u and v. Edges can be directed or undirected. Additionally, we call a graph *weighted* if there is a function that assigns the nodes/edges a real number. A *path* P in a graph is represented as a sequence of nodes $(v_1, v_2, \ldots, v_n)$ where $(v_i, v_{i+1}) \in E$ for all $i = 1, 2, \ldots, n-1$. The *shortest path problem* on a weighted graph asks for a path P between two given nodes, such that there exists no other path P' with a smaller weight than

P. Node separators are sets of nodes, the removal of which divides a graph into disconnected components. Additionally, nested dissection, a graph partitioning technique, recursively divides a graph by finding node separators. We also define an ordering on all vertices, defined by the rank from a nested dissection. We define $e = (u, v) \in E^{\uparrow}(u), v \in V$ as an upward edge, meaning $u < v$. Similarly, we define $E^{\downarrow}(u)$. With respect to this order, we define the upweight of an edge $e \in E$ as $w^{\uparrow}(e)$. Analogously, we define $w^{\downarrow}(e)$.

3 Related Work

In the realm of graph theory and route planning, various algorithms have been developed to efficiently find the shortest paths between nodes in large graphs. This section explores the key algorithms necessary for building the foundations for Customizable Contraction Hierarchies.

3.1 Dijkstra's Algorithm

Dijkstra's algorithm [2] is a classic method for finding the shortest paths between nodes in a graph. Named after Dutch computer scientist Edsger W. Dijkstra, this algorithm greedily explores the graph, iteratively selecting the node with the smallest tentative distance from the source. It maintains a priority queue to efficiently explore the graph and update the distances until the shortest path to all nodes is found. While Dijkstra's algorithm guarantees the accuracy of the shortest paths, its computational complexity can become a bottleneck for large graphs.

3.2 Contraction Hierarchies

Contraction Hierarchies (CH) [3] is an enhancement to traditional routing algorithms, designed to accelerate the process of finding shortest paths in road networks. The key idea behind CH is to preprocess the graph by contracting all nodes in an "importance" order. By creating this hierarchy of nodes, where shortcuts between nodes skip unnecessary parts of the network, the algorithm can quickly identify the shortest path. Although CH significantly improves query times, the preprocessing step can be computationally intensive. The query itself consists of two Dijkstra searches, starting from the source and the target, and only relaxing upward edges.

4 Customizable Contraction Hierarchies

Customizable Contraction Hierarchies (CCH) [1] builds upon the foundation laid by Contraction Hierarchies but introduces a customizable aspect to better handle changing edge weights. The main idea behind the CCH is a *metric-independent* preprocessing, where nodes are contracted in an order computed using a nested dissection. The *metric-dependent* processing (called customization) then assigns and adapts the edge weights of the hierarchy. Just like the CH, two Dijkstra searches can be run in the hierarchy. But there is another, even simpler method, which scans the entire graph starting from the source and target in a predefined manner. This method is called *Elimination tree query* and is not only very fast, but can also handle negative edge weights.

In this section, we discuss the different steps of CCH and look at the implementation details of the data structures and algorithms. Our code is available here `https://github.`

`com/PatrickSteil/CCH`. We first explain some detail on the different data structures used, and afterwards, we go into more detail on the actual algorithms.

4.1 Data Structures

4.1.1 Graph

Within our underlying graph data structures, we have implemented two distinct types of graphs: *StaticGraph* and *DynamicGraph*. As their names imply, the dynamic graph facilitates rapid edge insertions and various manipulation techniques. In contrast, the static graph employs a compact and cache-efficient representation, optimizing it for rapid access and iteration over edges.

The dynamic graph is realized through the utilization of a `std::vector<Edge>` for each node, enabling the storage of a list of adjacent edges associated with that node. To accommodate directed graphs, incoming edges are also stored. The choice of a `std::vector<Edge>` proves advantageous for efficient addition or removal of edges. Furthermore, a practical strategy involves preallocating space, or *reserving*, before bulk insertions, especially when introducing new edges.

Conversely, the static graph relies on several fixed-sized `std::vectors`. Some of these vectors store information about vertices, such as *toAdj*, while others contain details about an edge identified by its index, such as *toVertex*, *fromVertex*, *upWeight*, or *downWeight*. To account for directed edges, a list of incoming edge IDs is maintained for each node.

4.1.2 CCH Data

The central component is the CCH Data class, responsible for housing the chordal graph, edge mapping, and information regarding removed flags. In addition to its storage role, the class offers valuable methods, such as *doForAllLowerTriangles* and *doForAllVerticesBottomToTop*. These methods take a function as a parameter, subsequently executing it on all lower triangles or all vertices from bottom to top, and many more.

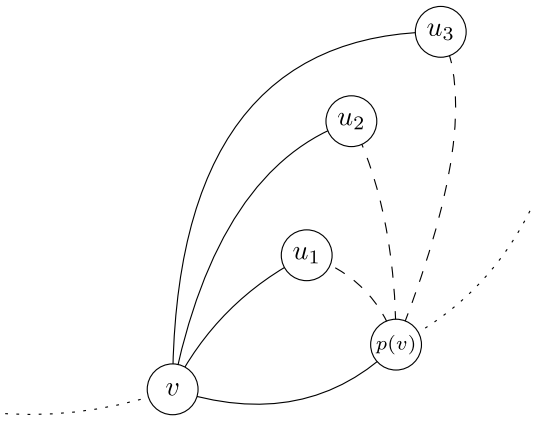
Upon completing the meticulous customization phase, rather than outright removing edges, we opt to *mark* them as removed based on their directions. This approach facilitates swift verification through a logical AND operation and expeditious marking using a logical OR operation (refer to the code snippet below).

```
1     // Enum to indicate the direction an edge should be removed
2     enum RemovalFlag { NOTHING = 0, UPWARD = 1, DOWNWARD = 2 };
3     // ...
4     // Vector keeps track of which edges should be removed
5     std::vector<uint8_t> removalFlags;
6     // ...
7     // Sets the edge for removal
8     void setForRemoval(Index edge, RemovalFlag flag) {
9       removalFlags[edge] |= flag;
10    }
```

4.2 Order

The metric-independent order in which the chordal graph is built is defined by the rank from a nested dissection. We will not go into detail on how this is achieved since it is not in the scope of our project, but we will evaluate the performance of orders computed by

Figure 1 Contraction of vertex v. The dashed edges are added onto $p(v)$, the parent of v. Note that only outgoing edges from v are added.



FlowCutter [6], Metis [4] and KaHIP [5]. Further details and results can be found in the original work for CCH [1].

4.3 Chordal Graph Building

Since the metric is irrelevant to the contraction process, we can model our graph as undirected. Missing directed edges can simply have infinity assigned as their weight in the customization phase. The chordal graph-building process is orchestrated by the preprocessing class, which generates the chordal graph based on a CCH Data object. The underlying idea is straightforward: for each vertex, starting from the bottom and moving to the top, we perform vertex contraction. This process is captured in the following code snippet:

```
1    data.doForAllVerticesBottomToTop([&](VertexID vertex) {
2      contractVertex(vertex);
3    });
```

The *contractVertex* method is different to the traditional "contraction", namely no shortcuts between all possible vertices inside a neighbourhood are enumerated and inserted. For the current vertex $v \in V$, we determine the lowest upward-neighbour (called *parent* $p(v)$) and perform

$$E^{\uparrow}(p(v)) = E^{\uparrow}(p(v)) \cup \left\{(p(v), u) \mid (v, u) \in E^{\uparrow}(v)\right\}$$

Put in simpler terms: we find the parent of v, and take all upgoing edges starting at v and copy them to the parent of v. This is illustrated in Figure 1. Note: This approach is very similar to some other algorithms used in graph theory when operating on chordal graphs. The order is called *elimination sequence*, and hence the name elimination tree query.

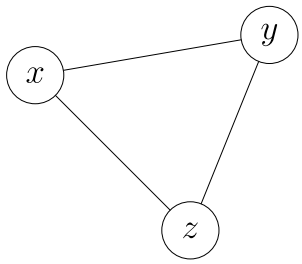
After contracting all vertices, we sort all edges in the chordal graph and save the chordal graph as a static graph (since we won't change its structure).

This "elimination sequence" approach is not only very fast (as we will show in Section 5) but also much simpler to implement than the traditional shortcut approach.

4.4 Customization Process

For the customization, we additionally require the notation of a *triangle* $\{x, y, z\}$, which is a set of three adjacent vertices. A triangle can be a lower, a middle or an upper triangle in

Figure 2 Rank of $z < x < y$. The triangle $\{x, y, z\}$ is a lower triangle for the edge (x, y), a middle triangle for the edge (z, y) and an upper triangle for the edge (z, x).



regards to an edge depending on the rank of the vertices. The triangle $\{x, y, z\}$ is a lower one for the edge (x, y) if the rank of z is lower than the rank of x and y. Analogously, if the rank of z is between the rank of x and y, we have a middle triangle and if it is larger we have an upper triangle. This is further illustrated in Figure 2.

The customization process encompasses three key stages. Initially, edge weights from the original graph are systematically mapped to their counterparts in the chordal graph. Subsequently, the *basic customization* phase is initiated to ensure the adherence of all triangles within the chordal graph to the triangle inequality. Finally, the *perfect customization* is executed, selectively removing edges that do not contribute to any shortest path.

4.4.1 Edge Mapping

The edge mapping process is straightforward. For each edge in the original graph, the corresponding edge in the chordal graph is identified. Depending on the direction of this edge, the up- or down-weight is then set accordingly. This can be trivially done in parallel.

4.4.2 Basic Customization

The basic customization process is more intricate. It involves iterating over all vertices from bottom to top. For each outgoing edge of the current vertex, the algorithm enumerates all lower triangles associated with the edge.

When looking at an edge $e = (u, v), u < v$ and the lower common neighbour x, we need to ensure that the current edge e is a *real shortcut*, meaning it preserves the shortest path distances. This means the shortcut weight must reflect the fact that it could be faster to use the detour over x, rather than traversing the edge. Hence, we only need to ensure that this equation holds for every edge and every according lower triangle:

$$w^{\uparrow}(e) \leq w^{\downarrow}((u, x)) + w^{\uparrow}((x, v))$$
$$w^{\downarrow}(e) \leq w^{\uparrow}((u, x)) + w^{\downarrow}((x, v))$$

Note: This seems inherently sequential, but we again can take advantage of the elimination sequence and the parent relation defined above. We can partition all vertices into levels, such that no edge lies inside one level. This can be computed by performing a simple BFS starting from the root (with respect to the parent relation). It has been shown [1], that two nodes on the same level can be "customized" at the same time, independently. This means we can parallelize the basic customization *per level* pretty easily without the use of expensive locks and thus reduce the running time drastically.

4.4.3 Perfect Customization

The perfect customization phase shares similarities with the basic customization but differs in its approach. Here, the algorithm iterates over vertices from top to bottom, enumerating all upper and middle triangles. If an edge weight changes, the algorithm marks the edge in the appropriate direction for removal. An important optimization is applied: only upper triangles are enumerated, as middle triangles can be obtained by a simple "rotation". This optimization ensures efficient triangle enumeration for a given edge. Similarly to the basic customization, we again can perform this perfect customization in parallel *per level*.

4.5 Query Algorithms

For our shortest path queries, we discuss two different algorithms. In the first algorithm, we use a similar approach to the original Contraction Hierarchies [3] namely a bidirectional Dijkstra search. The second algorithm utilizes a so-called elimination tree to scan the graph from the source and target in a predefined manner. This approach eliminates the necessity for a priority queue contrary to the Dijkstra-based ones.

4.5.1 Bidirectional Dijkstra

The Bidirectional Dijkstra algorithm uses two independent Dijkstra instances. For that, we first build two *StaticGraphs*, one with all upward edges $E^{\uparrow}$ and the corresponding upweights $w^{\uparrow}(e)$ for the search starting from the source s and one with the corresponding downweights $w^{\downarrow}(e)$ for the search from the target t. Because we only need to enumerate outgoing neighbours, incoming edges do not need to be maintained in the static graph. For the query itself, we alternate between the forward- and the backwards Dijkstra until both of the priority queues are empty or the min key of one of the Dijkstra searches surpasses the weight of the shortest path found so far.

4.5.2 Elimination Tree Query

In the Chordal Graph Building step, we already introduced the parent function, with which the so-called Elimination Tree can be built by linking each vertex with its parent. This gives us a predefined order in which we can relax outgoing edges from the source s and the target t respectively, instead of having to use a priority queue. To do so efficiently we build two *StaticGraphs*, like for the bidirectional Dijkstra algorithm. Additionally, we use two arrays $d_f(v)$ and $d_b(v)$ to store the tentative distances for the forward and backward relaxations respectively. These are initialized with ∞. First, we compute the Lowest Common Ancestor (LCA) of s and t by repeatedly updating the lower-ranked vertex by their respective parents until they coincide. Simultaneously, we relax all outgoing forward edges for s and its ancestors and relax backward edges for t and its ancestors. After finding the LCA, we can continue to relax all edges (both forward and backward) until we reach the root of our elimination tree. At the same time, we check for the minimal distance $d_f(z) + d_b(z)$ to determine the vertex z through which the shortest up-down path will lead. This vertex is important for the path extraction.

4.5.3 Path Extraction

For the Path Extraction, we are given the shortest up-down path from s to t via z. We then recursively unpack all upward and downward shortcuts (p_i, p_{i+1}). This is done by enumerating

Figure 3 A simple query schema. The vertex x is the LCA of s and t. The dashed arcs represent the edges that are relaxed.

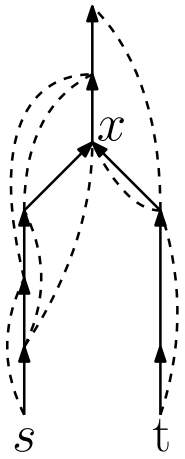


Table 1 An overview of the networks on which we performed our experiments.

	Karlsruhe	Germany	Europe
# Vertices	120 412	5 763 063	18 010 173
# Edges	302 605	13 984 846	42 188 664

over all lower triangles p_i, p_{i+1}, x until we find a triangle for which $w^{\uparrow}(p_i, p_{i+1}) = w^{\downarrow}(x, p_i) + w^{\uparrow}(x, p_{i+1})$ holds if it was an upward shortcut or $w^{\downarrow}(p_i, p_{i+1}) = w^{\uparrow}(x, p_i) + w^{\downarrow}(x, p_{i+1})$ if it was a downward shortcut. The shortcut is then updated by (p_i, x, p_{i+1}). In case no such triangle was found, (p_i, p_{i+1}) is already a fully unpacked edge and the recursion step ends.

5 Experiments

We evaluate our implementation on three real-life road networks. See Table 1 for a more detailed overview of the networks used. We are not allowed to publish the networks used in these experiments.

Since we did not need to implement our nested dissection algorithm, we were given nested dissection orders computed by FlowCutter [6] and by two standard graph partitioning tools KaHIP [5] and Metis [4]. Note that not for all networks, all three orders were available. We ran all of the experiments on a `Supermicro SuperServer SYS 6029UZ-TR4+ MB X11DPU-Z+(LGA 3647) 192GB(12x16GB) DDR4 2666MHz ECC`. The parallel algorithms were run with 16 threads.

5.1 Building the Chordal Graph

In Table 2, we report the results of the building process for each network and each (available) order. As the results show, FlowCutter consistently produced the best orders, regarding running time and the resulting number of edges. Remarkably, we can construct the chordal graph for the Europe instance in little over four seconds. Note that the "traditional" approach (enumerating shortcuts between neighbours and inserting them) took well over a minute. It

Table 2 Building the chordal graph based on the used order. We report the total time, the time needed to build the chordal graph, the time to sort the edges and the time to convert the graph into our static graph. Additionally, we report the number of edges in the resulting chordal graph.

Network	Order	Total [ms]	Building [ms]	Sorting [ms]	Convert. [ms]	# Edges
Karlsruhe	FlowCutter	24.405	4.855	3.010	16.540	422 530
	Metis	28.717	6.184	4.111	18.422	478 282
Germany	FlowCutter	1 407.902	310.585	165.209	932.108	19 862 528
Europe	FlowCutter	4 066.808	938.655	509.503	2 618.650	60 300 841
	KaHIP	5 024.307	1 341.430	754.997	2 927.880	74 077 603
	Metis	5 024.198	1 463.430	746.928	2 813.840	70 070 274

Table 3 Shown are the running times of the different kinds of customizations; basic and perfect. All the parallel variants were performed with 16 threads. Additionally, we report the time it takes to apply the given weight to the respective edge in the chordal graph. The customization behaved metric-independent, meaning there were no noticeable differences between travel time and geo-distance as metrics. Here, we chose the travel time metric.

		Sequential		Parallel		
Network	Order	Basic [s]	Perfect [s]	Basic [s]	Perfect [s]	Apply [*s*]
Karlsruhe	FlowCutter	0.065	0.037	0.018	0.011	0.005
	Metis	0.094	0.049	0.025	0.012	0.005
Germany	FlowCutter	6.446	2.516	1.709	0.657	0.226
Europe	FlowCutter	18.554	7.507	5.435	1.892	0.723
	KaHIP	22.301	9.038	7.283	1.689	0.739
	Metis	60.303	15.347	29.772	3.071	0.699

is also worth noting, that our algorithm spends most of the time converting the collected chordal graph into our compact static graph data structure.

In Figure 4, we also show the distribution of the nodes over the calculated levels per order. You can see that FlowCutter and KaHIP have significantly fewer levels, which conversely means that the longest path from a node to the root node is relatively short (in terms of the parent relation).

5.2 Customizations

As our results show, the metric of the graph can be adjusted in just a few seconds. Two things are striking: FlowCutter also consistently delivers the best orders here compared to the black box partitioners. In addition, parallel approaches show great success, especially on Europe we achieve a speedup of about three compared to the sequential variant. This allows us to adapt the new metric on all networks in under seven seconds. It is also astonishing that Metis appears to deliver worse results (in terms of customization runtime) than its black box competitor KaHIP. This is surprising given that Metis generates almost 4 million fewer new edges (see Table 2).

Figure 4 The distribution of the nodes to the respective levels is shown computed from an order. The underlying graph is the Europe instance, and the three orders are FlowCutter, KaHIP and Metis. A node at level 0 is a node that is furthest away from the root (at the highest level). It can be seen that FlowCutter and KaHIP both induce significantly fewer levels (both deliver just under 1200 level), whereas Metis is significantly worse (at just over 2000 level). As a result, there are significantly fewer nodes per level on average, which is a disadvantage for parallelism.

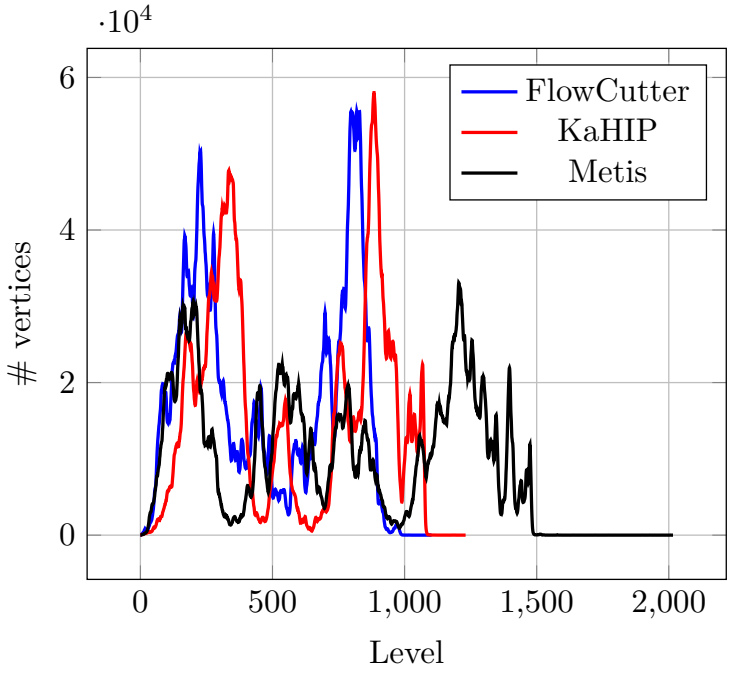


Table 4 Shown are the running times of the different kinds of query algorithms on the respective network using the travel time metric: plain Dijkstra, CCH-Dijkstra and Elimination Tree (after basic– and perfect customization). The running times are averaged over 1000 random source-target shortest-path queries. Note that the plain Dijkstra is order-independent, since it is executed on the original graph.

Network	Order	Dijkstra [ms]	CCH-Dijkstra [ms]	Speedup	Elimination tree [ms]	Speedup
Karlsruhe	FlowCutter	16.636	0.018	924.222	0.024	693.167
	Metis	16.636	0.023	723.304	0.031	536.645
Germany	FlowCutter	1 865.130	0.165	11 303.818	0.181	10 304.586
Europe	FlowCutter	6 058.010	0.179	33 843.631	0.187	32 395.775
	KaHIP	6 058.010	0.214	28 308.458	0.229	26 454.192
	Metis	6 058.010	0.537	11 281.210	0.556	10 895.701

5.3 Query Performance

We show the results of our query experiments in several tables. Firstly, we compare the normal Dijkstra with the bidirectional Dijkstra CCH variant and with the elimination tree query. On the other hand, we take a closer look at the Elimination Tree Query in detail, namely the runtime and other interesting statistics regarding different edge metrics.

As can be seen in Table 4, both CCH queries provide speedups of three to five orders of magnitude over the simple Dijkstra, especially favourable for large networks. On all networks, we can answer random shortest path queries in well under a millisecond, which is more than fast enough for any real-world application. The CCH-Dijkstra variant beats the elimination tree query, but only marginally. On the other hand, the elimination tree query can also work with negative edge weights, which speaks in its favour. And once again it is clear that FlowCutter delivers very good orders, as it beats Metis and KaHIP on every network. The comparison between KaHIP and Metis also shows that KaHIP computes significantly better orders. To be fair, it must be said that Metis is designed for speed and KaHIP for quality (which is also confirmed in our experiments).

If you compare the runtime of the query using the geo-distance metric with the travel time metric (see Table 5), you can see a slight decrease in performance of just around $8-27\%$. However, the query has to do a lot more work overall (when using geo-distance as a metric), which can be seen both in the number of relaxed edges and in the resulting path length. The realization that geo-distance is a rather “bad” metric is nothing new since CH already achieves significantly worse results with this metric than with the travel time metric. This can also be seen here, as the perfect customization cannot remove as many edges for the geo-distance metric compared to the travel time metric. This means, much more edges are in the resulting query graph, which explains why the query takes slightly longer. However, the elimination tree query is very effective and robust for both metrics used here, and can even be used with the relatively expensive path unpacking for real-time applications.

6 Conclusion

Our implementation of the CCH shows good results, given an order by e.g., FlowCutter or KaHIP, the entire chordal graph can be built within seconds and edge weights be adapted

Table 5 The following table shows different runtimes and other measures of the elimination tree query. The order used was calculated by FlowCutter, and both basic and perfect customization were calculated. The reported times are the total runtime of the query, the time to find the lowest common ancestor (LCA) and the (separately executed) time to extract the path. In addition, the number of edge relaxations (both upwards and downwards), the average path length and the number of removed edges (after the perfect customization) are also reported. The "removed edges" count each edge direction, meaning an edge that is removed in both direction is counted twice.

Network	Metric	Query [ms]	LCA [ms]	# ↑ relax	# ↓ relax	Path Ext. [ms]	Length	# Remov.
Karlsruhe	Travelt.	0.024	0.015	2 020.59	2 022.12	0.021	134.848	336 129
	GeoDist.	0.026	0.018	2 579.94	2 582.13	0.030	174.194	187 270
Germany	Travelt.	0.185	0.117	33 736.10	34 375.70	0.157	495.860	17 015 570
	GeoDist.	0.230	0.151	47 431.50	47 756.10	0.404	1 032.960	10 508 830
Europe	Travelt.	0.187	0.141	35 288.30	34 993.30	0.467	1 209.980	54 115 792
	GeoDist.	0.237	0.173	49 407.50	48 908.30	0.813	2 100.320	35 967 083

given a metric. Parallel customization is simple, but very effective, especially on large networks. Our queries are very fast, achieving a speedup of more than 5 orders of magnitude compared to Dijkstra. What could be done in the future: better customization and use of SIMD intrinsics to adjust several metrics (geo-distance, travel time, ...) at the same time. It would also be interesting to test the simple nested dissection algorithm "InertialFlow" against FlowCutter and KaHIP.

References

1 Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Customizable contraction hierarchies. *ACM J. Exp. Algorithmics*, 21(1):1.5:1–1.5:49, 2016. doi:10.1145/2886843.

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

3 Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In Catherine C. McGeoch, editor, *Experimental Algorithms, 7th International Workshop, WEA 2008, Provincetown, MA, USA, May 30-June 1, 2008, Proceedings*, volume 5038 of *Lecture Notes in Computer Science*, pages 319–333. Springer, 2008. doi:10.1007/978-3-540-68552-4_24.

4 George Karypis and Vipin Kumar. *METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices*, September 1998.

5 Wolfgang Ost, Christian Schulz, and Darren Strash. Engineering data reduction for nested dissection. *CoRR*, abs/2004.11315, 2020. URL: https://arxiv.org/abs/2004.11315, arXiv:2004.11315.

6 Ben Strasser. PACE Solver Description: Tree Depth with FlowCutter. In Yixin Cao and Marcin Pilipczuk, editors, *15th International Symposium on Parameterized and Exact Computation (IPEC 2020)*, volume 180 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 32:1–32:4, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.IPEC.2020.32, doi:10.4230/LIPIcs.IPEC.2020.32.