Real-Time Traffic Assignment Using Fast Queries in Customizable Contraction Hierarchies

Valentin Buchhold
Karlsruhe Institute of Technology, Germany

Peter Sanders
Karlsruhe Institute of Technology, Germany

Dorothea Wagner
Karlsruhe Institute of Technology, Germany

Abstract

Given an urban road network and a set of origin-destination (OD) pairs, the traffic assignment problem asks for the traffic flow on each road segment. A common solution employs a feasible-direction method, where the direction-finding step requires many shortest-path computations. In this paper, we significantly accelerate the computation of flow patterns, enabling interactive transportation and urban planning applications. We achieve this by revisiting and carefully engineering known speedup techniques for shortest paths, and combining them with customizable contraction hierarchies. In particular, our accelerated elimination tree search is more than an order of magnitude faster for local queries than the original algorithm, and our centralized search speeds up batched point-to-point shortest paths by a factor of up to 6. These optimizations are independent of traffic assignment and can be generally used for (batched) point-to-point queries. In contrast to prior work, our evaluation uses real-world data for all parts of the problem. On a metropolitan area encompassing more than 2.7 million inhabitants, we reduce the flow-pattern computation for a typical two-hour morning peak from 76.5 to 10.5 seconds on one core, and 4.3 seconds on four cores. This represents a speedup of 18 over the state of the art, and three orders of magnitude over the Dijkstra-based baseline.

2012 ACM Subject Classification Theory of computation → Shortest paths

Keywords and phrases traffic assignment, equilibrium flow pattern, customizable contraction hierarchies, batched shortest paths

Digital Object Identifier 10.4230/LIPIcs.SEA.2018.27

Acknowledgements We thank Peter Vortisch for providing the Stuttgart instance, and Lukas Barth and Ben Strasser for interesting discussions.

1 Introduction

The number of drivers traveling along a road segment within a given period is the result of many individual decisions. The common behavioral assumption in practice is that motorists driving between a given origin and destination choose the path with the minimum travel time (known as *Wardrop's first rule* [39]). This seems natural, since travel is usually not a goal in itself, but entails disutility. However, the travel time on a path depends on the route choice of all other drivers, who themselves are trying to choose minimum travel time routes. Due to congestion, the travel time on a road segment increases with the traffic flow on it. As a result, some drivers choose at some point alternative routes, which can also get congested, and so on. When no driver can improve his travel time by unilaterally changing routes, each

© Valentin Buchhold, Peter Sanders, and Dorothea Wagner;
licensed under Creative Commons License CC-BY
17th International Symposium on Experimental Algorithms (SEA 2018).
Editor: Gianlorenzo D'Angelo; Article No. 27; pp. 27:1–27:15

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

route used between a given origin and destination has the same travel time. This condition is known as the *user equilibrium*, and the flow pattern is called the *equilibrium flow pattern* [37].

We study the efficient computation of equilibrium flow patterns in road networks. More formally, given an urban road network and a set of origin-destination (OD) pairs, we want to compute the traffic flow on each road segment at equilibrium. This is known as *traffic assignment*, and is one of the major problems facing transportation engineers and urban planners [37]. In this paper, we accelerate the process of traffic assignment significantly (by a factor of 18). Our goal is twofold. The short-term objective is to enable *interactive* transportation and urban planning applications. The long-term aim is to develop a real-time demand-responsive public transit system, which makes use of a traffic assignment procedure as a subroutine. There, we decrease (rather than increase) the travel cost per individual as the flow increases, since increased flow makes public transit more cost-effective.

Related Work. The traffic assignment problem has been studied for over 60 years, and has motivated extensive research in the operations research community. The formulation as a mathematical program first appeared in 1956 [5]. A common solution employs the Frank-Wolfe algorithm [17], a feasible-direction method for solving quadratic programs with linear constraints. The application of the Frank-Wolfe method to the traffic assignment problem was first proposed in the late 1960s [6, 31], and the solution was implemented and experimentally evaluated for the first time in 1975 [25]. The textbook by Sheffi [37] offers a comprehensive introduction to the traffic assignment problem and considers research published until 1985. More recent developments are covered by Patriksson [32]. Peeta et al. [33] survey recent advances in *dynamic* traffic assignment, recognizing time variations in traffic flows and conditions during the period of analysis. Finally, Babonneau et al. [1] collect benchmark instances previously used in the literature, however, their largest instance is still an order of magnitude smaller than the benchmark instance used in this paper.

The performance of the Frank-Wolfe algorithm is clearly dominated by the direction-finding step, which requires a large number of shortest-path computations. The past decade has seen intense research on speedup techniques [2] for Dijkstra's algorithm [13], which rely on a slow preprocessing phase to enable fast queries. One of the most prominent and versatile techniques among these are contraction hierarchies (CH) [18], which exploit the inherent hierarchy of the network. A fairly recent development are customizable speedup techniques [8, 12, 14], which split preprocessing into a slow metric-independent part, taking only the graph structure into account, and a fast metric-dependent part (the *customization*), incorporating new edge weights (the *metric*). CRP [8] and customizable CHs [12] are the most prominent among them. A common approach to accelerate one-to-all queries is to bundle together multiple shortest-path computations in a single search [20, 8, 7, 4, 40].

To the best of our knowledge, there is only a single paper [26] that solves the traffic assignment problem using state-of-the-art shortest-path algorithms (plain CHs in this case).

Contribution and Outline. The contribution of this work is twofold. First, we accelerate the state of the art in the area of traffic assignment. On our main benchmark instance, we see a speedup of 18. This is more than three orders of magnitude faster than the Dijkstra-based baseline. However, the building blocks to achieve this are also independent contributions, not restricted to traffic assignment, but generally applicable in the area of route planning. Our two main building blocks are as follows. (1) Currently, there are two CCH query algorithms, one based on Dijkstra's algorithm and one based on elimination trees (a structure encoding the CH search space of each vertex). We thoroughly reengineer the elimination tree search (Section 3), providing a unified query algorithm that combines the good local-query

performance of the Dijkstra-based search with the good global-query performance of the elimination tree search. (2) We introduce a *centralized* elimination tree search for computing batched point-to-point shortest paths fast (Section 4). While there is a large amount of work on one-to-all, one-to-many, many-to-many, and point-of-interest queries [7, 9, 15, 16, 22, 10], we are the first that accelerate batched *point-to-point* shortest paths. All building blocks are extensively experimentally evaluated using solely real-world data (Section 5), whereas previous work fell back on synthetic OD-pairs [26].

2 Preliminaries

We now briefly review the three main algorithms we build upon. First, we describe the Frank-Wolfe algorithm for solving the traffic assignment problem. Then, we discuss two speedup techniques for Dijkstra's algorithm, CHs and their customizable counterpart CCHs.

2.1 Traffic Assignment

Popular approaches [37] to the traffic assignment problem formulate a mathematical program, known as *Beckmann's transformation* [5], whose solution is the equilibrium flow pattern. It is a convex minimization program with linear constraints. The *Frank-Wolfe algorithm* [17, 37], a feasible-direction method, is especially suitable for solving Beckmann's transformation, since the direction-finding step can be implemented relatively efficiently. Being a feasible-direction method, it iteratively finds a feasible descent direction and advances by an optimal move size in the direction. An important subroutine of the Frank-Wolfe algorithm (when solving Beckmann's transformation) is the *all-or-nothing assignment* procedure, which processes each OD-pair in turn and assigns one flow unit to each edge on the shortest travel time path.

The algorithm can be summarized as follows. (1) Perform an all-or-nothing assignment using free-flow travel times, yielding an initial solution. (2) Update the travel time on each edge according to the current solution (recall that the travel time increases with the traffic flow). (3) Perform an all-or-nothing assignment using the current travel times, yielding a set of auxiliary flows. (4) Perform a *line search* to determine the optimal move size α. (5) Set the new solution to a convex combination of the current solution and the auxiliary flows (according to α). (6) Check the stopping criterion, and terminate or go to step (2). In this paper we update travel times according to the modified Davidson function [30], perform the bisection method of Bolzano as line search, and stop after a predefined number of iterations.

2.2 Contraction Hierarchies

Contraction hierarchies (CH) [18] are a two-phase speedup technique to accelerate point-to-point shortest-path computations, which exploits the inherent hierarchy of road networks. The preprocessing phase heuristically orders the vertices by importance, and *contracts* them from least to most important. Intuitively, vertices that hit many shortest paths are considered more important, such as vertices on highways. To contract a vertex v, it is temporarily removed from the graph, and *shortcut* edges are added between its neighbors to preserve distances in the remaining graph (without v). Note that a shortcut is only needed if it represents the only shortest path between its endpoints, which can be checked by running a *witness search* (local Dijkstra) between its endpoints. The query phase runs a bidirectional Dijkstra on the augmented graph that only relaxes edges leading to vertices of higher ranks (importance). The stall-on-demand [18] optimization prunes the search at any vertex v with a suboptimal distance label, which can be checked by looking at upward edges coming into v.

Traffic Assignment Using CHs. The shortest-path computations are by far the most time-consuming part of the Frank-Wolfe algorithm. Carrying them out with the use of CHs (instead of Dijkstra's algorithm) accelerates traffic assignments by two orders of magnitude [26]. Since the weight of each edge changes between two iterations, the CH is rebuilt from scratch in each iteration. Queries do not unpack shortcuts, but assign one flow unit to each (shortcut) edge on the packed path. After computing all paths, the shortcuts are unpacked in top-down fashion, with cumulated flow units propagated from shortcut to original edges.

2.3 Customizable CHs

Customizable contraction hierarchies (CCH) [12] are a three-phase technique, splitting CH preprocessing into a relatively slow metric-independent phase and a much faster customization phase. The metric-independent phase computes a *nested dissection order* [3, 19] on the vertices of the unweighted graph, and contracts them in this order without running witness searches (as they depend on the metric). As a result, it adds every potential shortcut. The customization phase computes the weights of the shortcuts by processing them in bottom-up fashion. To process a shortcut (u, v), it enumerates all triangles $\langle u, v, w\rangle$ where w has lower rank than u and v, and checks if the path (u, w, v) improves the weight of (u, v). After *basic customization*, one can optionally run *perfect witness searches* to remove superfluous edges.

There are two different query algorithms possible. First, one can run a standard CH search without modification. In addition, Dibbelt et al. [12] describe a query algorithm based on the *elimination tree* of the augmented graph. The parent of a vertex in the elimination tree is its lowest-ranked upward neighbor in the augmented graph. Bauer et al. [3] prove that the ancestors of a vertex v in the elimination tree are exactly the set of vertices in the CH search space of v. Hence, the elimination tree query algorithm explores the search space by traversing the elimination tree, avoiding a priority queue completely.

3 Accelerating Elimination Tree Searches

In the next section, we will devise a fast traffic assignment procedure based on customizable contraction hierarchies. While Dibbelt et al. [12] observe that the CCH query algorithm based on elimination trees achieves fastest query times for random queries (which tend to be long-range), it is slower by more than an order of magnitude than the Dijkstra-based query algorithm for local queries (see Section 5). However, the input of the traffic assignment problem consists of both local and long-range OD-pairs, requiring a query algorithm that can handle both types of queries well. Therefore, we review and carefully engineer the elimination tree search in this section. The result is a fast, unified CCH query algorithm, combining good performance for both local and long-range queries.

Given a source vertex s and a target vertex t, the original elimination tree search [12] works in five phases. First, we compute the lowest common ancestor (LCA) x of s and t in the elimination tree T rooted at the highest-ranked vertex r. This is done by enumerating the ancestors of s and t in increasing rank order until a common ancestor is found. Second, we revisit each vertex v on the s–x path in T, relaxing all *outgoing* upward edges of v. Third, we do the same for each vertex v on the t–x path in T, relaxing all *incoming* upward edges of v. Fourth, we visit each vertex v on the x–r path in T, relaxing all outgoing *and* incoming upward edges of v. Moreover, we check at each such vertex v whether the s–t path via v improves the tentative s–t distance. Fifth, we again revisit each vertex on the s–r and t–r path to reset its distance labels for the next shortest-path computation.

Phase Reduction. Our first optimization reduces the number of phases of the elimination tree search. We refrain from computing the LCA first, and then visiting each vertex from the source (target) to the LCA again. Instead, while we enumerate the ancestors of s and t in the same fashion as before, we immediately relax their edges. Moreover, we observe that the resetting phase is unnecessary. After relaxing the edges of a vertex, its distance labels are never read again. Therefore, we can safely reset them to ∞ right after relaxing the edges, avoiding the fifth phase completely. Note that we cannot reset parent pointers, since they may be needed afterwards. However, this is not an issue because resetting the distance labels suffices to decide whether a vertex has been visited before during the next query. With this optimization, each vertex is visited at most once, instead of up to three times as before.

Pruning Rule. The basic elimination tree search does not make use of pruning. Only when combined with the perfect witness search, Dibbelt et al. [12] employ the following basic pruning rule. Due to the removal of superfluous edges, a vertex may have an ancestor in the elimination tree that is not in its perfect search space. Such an ancestor will have a distance label of ∞ when visited during the search. To accelerate queries, Dibbelt et al. do not relax the edges of a vertex with a distance label of ∞. We observe that a stricter pruning rule is possible. We do not relax edges of a vertex whose distance label exceeds the current tentative shortest-path distance, since those edges cannot possibly contribute to a shorter path. Despite its simplicity, this optimization accelerates the search quite drastically, by a factor of 15 for short-range queries (see Section 5). Moreover, our pruning rule does not require the perfect witness search, but can also be combined with the basic customization.

4 Accelerating Traffic Assignments by Fast Batched Shortest Paths

Previous work [26] applying speedup techniques to traffic assignment observed that the performance bottleneck depends on the traffic scenario under study. For short or off-peak periods, where there are few OD-pairs, preprocessing dominates the total running time. When there are many OD-pairs, as for long or peak periods, queries become the main bottleneck.

To decrease the preprocessing time, we apply the concept of customization to traffic assignment. Customizable speedup techniques [8, 12, 14] split preprocessing into a metric-independent part, taking only the graph structure into account, and a metric-dependent part (the *customization*), incorporating new edge weights (the *metric*). Since the graph topology does not change in all iterations of the traffic assignment procedure and only edge weights change, it suffices to run a fast customization in each iteration instead of an entire preprocessing. We build our accelerated traffic assignment upon customizable contraction hierarchies [12], which allows us to employ the hierarchy decomposition optimization from [26]. As basic query algorithm, we use the engineered elimination tree search from the previous section. To reduce the query time, the following sections introduce several optimization techniques for computing batched point-to-point shortest paths fast.

4.1 Reordering OD-pairs to Exploit Locality

Previous work processed the OD-pairs in no particular order. However, reordering the OD-pairs so that pairs with similar forward and reverse search spaces tend to be processed in succession improves memory locality and cache efficiency. We call two search spaces similar if their symmetric difference is small. Note that the size of the symmetric difference between the search spaces of u and v is equal to the distance between u and v in the elimination tree. Hence, we partition the elimination tree into as few cells with bounded diameter as possible,

assign IDs to cells according to the order in which they are reached during a DFS [29] on the elimination tree, and reorder OD-pairs lexicographically by the origin and destination cells.

We use a simple yet optimal greedy algorithm to partition the elimination tree into as few cells with diameter at most U as possible. Our algorithm repeatedly cuts out a subtree (with diameter at most U) and makes it a cell of its own. In order to do so, it maintains for each vertex v the height $h(v)$ of the remaining subtree T_v rooted at v, initialized to zero, and processes vertices in ascending rank order. To process v, we examine its children w_i in order of increasing height of T_{w_i}. If $h(v) + 1 + h(w_i) \leq U$, we set $h(v) = 1 + h(w_i)$. Otherwise, we cut out T_{w_i}, making it a cell of its own. We use $U = 40$ in our experiments.

4.2 Centralized Searches

Instead of processing similar OD-pairs *in succession*, processing them *at once* in a single search achieves additional speedup. The idea of bundling together multiple shortest-path computations was introduced in [20] and later used in [8, 7, 9, 4, 40]. However, in each case, centralized searches were only used for one-to-all and -many queries, and only combined with plain Dijkstra (and Bellman-Ford in [8]). Even (R)PHAST [7, 9] performs the CH searches sequentially, and bundles only the linear sweeps. We extend the idea to point-to-point queries, and combine it with CH searches, including appropriate stopping and pruning criteria.

The basic idea of centralized searches is to maintain k distance labels for each vertex u, laid out consecutively in memory. The i-th distance label represents the tentative distance from the i-th source to u. Initially, the i-th distance label of the i-th source is set to zero, and all remaining $kn - k$ distance labels to ∞. When relaxing an edge (u, v), we try to improve all k distance labels of v at once. Increasing k allows us to compute more shortest paths at once, however, it also evicts useful data from caches.

Dijkstra-Based Search. Initially, we insert all k sources (targets) into the queue of the forward (reverse) search. As keys, we can use many different values, for example the minimum over all k entries in a distance label or the minimum over the entries that were improved by the last edge relaxation. However, a preliminary experiment showed that using the minimum over *all* k entries clearly dominates the others, which is consistent with previous observations on related techniques [20]. We can stop the forward (reverse) search as soon as its queue is empty or the smallest queue entry exceeds the maximum over all k tentative shortest-path distances. When using stall-on-demand [18], we prune the forward (reverse) search at a vertex v when each of the k distance labels of v is suboptimal.

Elimination Tree Search. Computing multiple shortest paths in a single elimination tree search is more involved, since it uses no queues that can easily be initialized with multiple sources and targets. Instead, we equip the forward and reverse search each with a tournament tree [23]. Suppose we have k sorted sequences that are to be merged into a single output sequence, as in k-way mergesort. To do so, we have to repeatedly find the smallest from the leading elements in the k sequences. This can be done efficiently with a tournament tree.

In our case, the k sorted sequences are the paths in the elimination tree T from each source (target) to the root, and the single output sequence is the order in which we want to process the vertices during the search. More precisely, we initialize the tournament tree with all k sources (targets). Then, we extract a lowest-ranked vertex from the tournament tree, process it, and insert its parent in T into the tournament tree. We continue with a next-lowest-ranked vertex, until we reach the root of T. Note that in our case, unlike in mergesort, the sequences are implicit, and never stored explicitly.

As soon as two (or more) of the k paths in T converge at a common vertex, there are duplicates in the single output sequence. However, we want to process each vertex at most once. Therefore, whenever two or more paths converge, we block all but one of them, so that only one continues to move through the tournament tree. To do so, we insert for each path to be blocked a vertex with infinite rank into the tournament tree (instead of the next vertex on the path). We know that some paths converged, when we extract the very same vertex several times in succession from the tournament tree.

A clear advantage of the centralized elimination tree search is that it retains the *label-setting* property, i.e., each vertex and each edge is processed at most once. In contrast, the centralized Dijkstra-based search is a *label-correcting* algorithm. Note that one centralized elimination tree search is slower than k elimination tree searches by a factor of $\log k$ in O-notation (due to k-way merging), but outperforms them in practice (see Section 5).

4.3 Instruction-Level Parallelism

Modern CPUs have special registers and instructions that enable single-instruction multiple-data (SIMD) computations performing basic operations (e.g., additions, subtractions, shifts, compares, and data conversions) on multiple data items simultaneously [24]. We implemented versions of the centralized searches using SSE instructions (working with 128-bit registers), and additionally versions using AVX(2) instructions (manipulating 256-bit registers), requiring a processor based on Intel's Haswell or AMD's Excavator microarchitecture.

As an example, we describe how an AVX-accelerated edge relaxation (used in Dijkstra-based and elimination tree searches) works, assuming $k = 8$. Since we use 32-bit distance labels, all k labels of a vertex fit in a single 256-bit register. To relax an edge (u, v), we copy all k distance labels of u to an AVX register, and broadcast the edge weight to all elements of another register. Then, we add both registers with a single instruction, and check with an AVX comparison whether any tentative distance improves the corresponding distance label of v. If so, we compute the packed minimum of the tentative distances and v's distance labels. In the same fashion, we implement the other aspects (stopping and pruning criteria).

4.4 Core-Level Parallelism

Dibbelt et al. [12] introduce parallelization techniques for the triangle enumeration during customization. However, we observed that the perfect witness search building the upward and downward search graphs (which is difficult to parallelize) actually takes up 60 % of the customization time. Hence, the speedup obtainable by parallelizing the customization phase is limited (less than a factor of 1.5). For simplicity, we stick to sequential customization.

In contrast, the shortest-path computations are easy to parallelize. Since the centralized computations are independent from one another, we can assign contiguous subsets of OD-pairs to distinct cores. We distribute the OD-pairs to cores in chunks of size 64. This maintains some locality even between centralized computations. Each core executes a chunk, then requesting another chunk until no chunk remains. Flow units on the (shortcut) edges are cumulated locally and aggregated after computing all paths. We observe an almost perfect speedup for the time spent on queries (see Section 5).

Table 1 Traffic scenarios used for the evaluation of our traffic assignment procedures. We report for each scenario the period of analysis and the number of OD-pairs departing within that period.

scenario	analysis period	# OD-pairs
Tue30m	Tue., 7:00–7:30	118 933
Tue01h	Tue., 7:00–8:00	246 089
Tue02h	Tue., 7:00–9:00	478 098
Tue24h	a whole Tuesday	3 355 442
MonSun	a whole week	21 248 278

5 Experiments

Our publicly available code[1] is written in C++14 and compiled with the GNU compiler 7.3 using optimization level 3. We use 4-heaps [21] as priority queues. To ensure a correct implementation, we make extensive use of assertions (disabled during measurements), and check results against reference implementations such as Dijkstra's algorithm. Our benchmark machine runs openSUSE Leap 42.3 (kernel 4.4.114), and has 128 GiB of DDR4-2133 RAM and an Intel Xeon E5-1630 v3 CPU, which has four cores clocked at 3.70 Ghz.

5.1 Inputs and Methodology

Our main instance is the metropolitan area of Stuttgart [36], Germany, encompassing more than 2.7 million inhabitants. The experiments were performed on the largest strongly connected component, consisting of 134 663 vertices and 307 759 edges. While this instance is significantly smaller than road networks studied before for evaluating point-to-point queries [2], it is the largest available to us that provides real-world capacities and OD-pairs, and is still an order of magnitude larger than the instances collected in [1]. Moreover, urban planners are usually interested in traffic assignments on metropolitan areas, not continents.

The OD-pairs were obtained from [27, 28], which was calibrated from a household travel survey [38] conducted in 2009/2010. The raw data contains about 51.8 million trips between 1174 traffic zones for a whole week, encompassing various modes of transportation such as pedestrian, bicycle, public transit and car. For our experiments we only considered car trips, and extracted five different traffic scenarios, as shown in Table 1. We chose a typical two-hour morning peak on a working day (Tuesday), and also included two smaller and two larger scenarios. While it may be unrealistic to compute a traffic assignment for a whole week (as the period of analysis would be too inhomogeneous), it shows the scalability of our procedures for tens of millions of OD-pairs. Note that we assume the actual origins and destinations to be uniformly distributed in the zone, and obtain OD-pairs by picking the origins and destinations uniformly at random from the zones according to the predicted trips.

Since our engineered elimination tree search is not restricted to traffic assignment, we evaluate it on the European road network, which is the standard benchmark instance for point-to-point queries [2]. It has 18 017 748 vertices and 42 560 275 edges, and was made available by PTV AG for the 9th DIMACS Implementation Challenge [11].

The CH preprocessing is borrowed from the open-source library RoutingKit[2]. We compute nested dissection orders for CCHs using Inertial Flow [35], setting the balance parameter

[1] `https://github.com/vbuchhold/routing-framework`
[2] `https://github.com/RoutingKit/RoutingKit`

$b = 0.3$. The reported running times do no include partitioning time, as it suffices to partition a network only once, and reuse the resulting order for all traffic assignments on the same network. Partitioning the metropolitan area of Stuttgart takes less than two seconds (even on a single core). We always use perfect witness searches in combination with CCHs.

5.2 Elimination Tree Search

First we evaluate our engineered elimination tree search on its own. As most queries in the real world tend to be local, we use the established Dijkstra rank methodology [34], which considers local and long-range queries equally. In contrast, random queries tend to be long-range. The Dijkstra rank (with respect to a source s) of a vertex v is r if v is the r-th vertex settled by a Dijkstra search from s. We run 1000 point-to-point queries (without path unpacking) for each Dijkstra rank tested, with s picked uniformly at random. Figure 1 compares the performance of our accelerated elimination tree search (CCH-tree-fast), the original CCH query algorithms (CCH-Dij and CCH-tree), and the plain CH search on Europe with travel times. Note that CCH-tree is not really the original algorithm, but already uses our phase reduction optimization. CCH-tree-fast additionally uses our stricter pruning rule.

We observe that CCH-tree, while outperforming CCH-Dij on random queries [12], is actually much slower for most Dijkstra ranks, especially for the realistic ones. The reason is that the performance of CCH-tree is independent of the Dijkstra rank, since it always processes each vertex in the search space. However, our stricter pruning rule makes the algorithm sensitive to the Dijkstra rank, drastically speeding up short- and mid-range queries (by up to a factor of 15). As a result, CCH-tree-fast combines the good local-query performance of CCH-Dij with the good global-query performance of CCH-tree, and is faster than both on mid-range queries. It can be seen as a unified CCH query algorithm, replacing both original ones. Moreover, for many (realistic) Dijkstra ranks, it is about as fast as the non-customizable CH search. When optimizing travel *distances* (not shown in the figure), CCH-tree-fast even outperforms the CH search.

5.3 Traffic Assignment

We now evaluate the impact of our building blocks (customization, reordering OD-pairs, centralized searches, and parallelism at multi-core and instruction levels) on the performance of the traffic assignment procedure. As already mentioned, we run a predefined number of iterations for each scenario: twelve on Tue30m, Tue01h and Tue02h, six on Tue24h and MonSun. This choice is consistent with [37] and also justified by subsequent experiments.

Customization and Centralized Searches. Table 2 considers the influence of customization and of the centralized searches on the performance of the traffic assignment. For now, we use only a single core. The CCH-based procedures use the engineered elimination tree search.

Switching from plain to customizable CHs reduces the running time for all traffic scenarios. As expected, we obtain larger speedups for smaller scenarios (a factor of 3 on Tue30m), since preprocessing time dominates more in such scenarios. In contrast, reordering the OD-pairs so that similar OD-pairs are processed successively works better for larger scenarios, improving the running time on MonSun by about 20 %.

The impact of computing multiple shortest paths at once without exploiting instruction-level parallelism is limited. However, when using SIMD instructions, centralized searches decrease the running time by up to another factor of 5.2. Increasing k allows us to compute more shortest paths at once, but it also evicts useful data from caches. Setting $k = 32$ seems

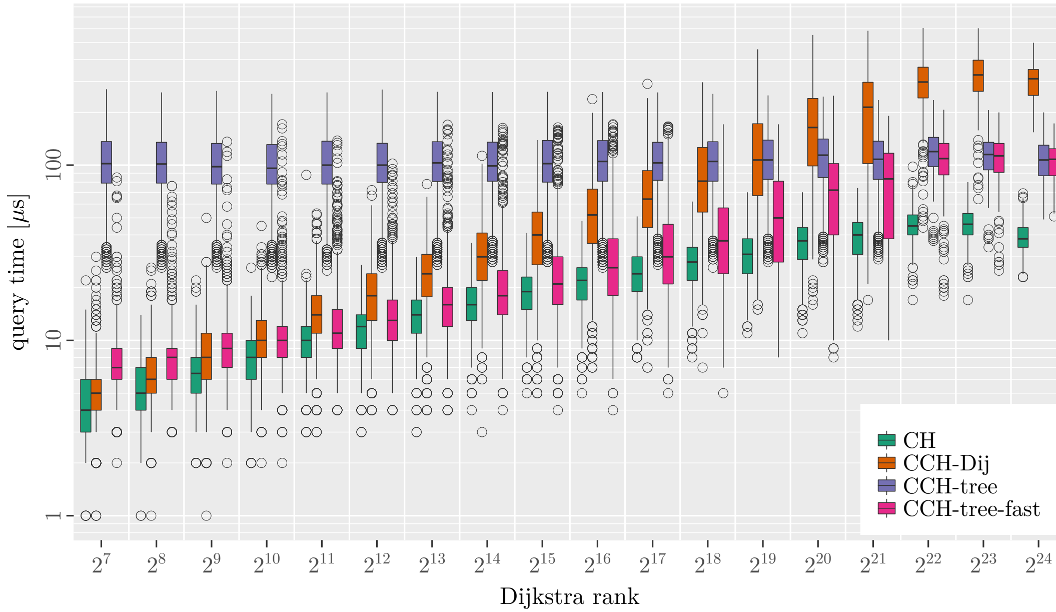


Figure 1 Performance of our engineered elimination-tree search (CCH-tree-fast), the original CCH query algorithms (CCH-Dij and CCH-tree), and a CH. The input is Europe with travel times.

to be a good choice. Moreover, we observe that the centralized elimination searches achieve greater speedups than the Dijkstra-based ones, since they are label-setting. (Although the clustering approach described in Section 4 is tailored to the elimination tree search, preliminary experiments with unbiased clustering approaches not building upon the elimination tree showed a quite similar performance difference.)

Combining the optimizations, the traffic assignment procedure based on AVX-accelerated centralized elimination tree searches with $k = 32$ gives the best overall performance. It speeds up the state of the art by a factor of about 8 on all of our traffic scenarios. Compared to the Dijkstra-based baseline, this configuration is several hundred times faster.

Core-Level Parallelism. Table 3 shows how the traffic assignment procedure scales as the number of cores increases. We observe that the time spent on queries scales very well. With 4 cores, we gain a speedup of 3.5 for Tue24h and MonSun, and even our smallest scenario is accelerated by a factor of 2.7. In total, our multi-threaded centralized traffic assignment procedure decreases the running time on our main benchmark instance Tue02h from 76.5 to 4.3 seconds, a speedup of 18 over the state of the art.

For comparison, we also run the state of the art on four cores, parallelizing the shortest-path computations as described in Section 4. We observe that even on a single core, our procedure is always more than twice as fast as the parallelized state of the art. The difference between both parallelized versions is again a factor of about 8.

Convergence. Next, we evaluate how long the traffic assignment procedure takes to converge. For that we run a very large number of iterations of the procedure (300 in our case) and take the resulting flow pattern as the equilibrium situation. Figure 2 shows the average deviation of the OD-travel-costs in each iteration from the OD-travel-costs at equilibrium.

We observe that Tue30m, Tue01h, and Tue02h require more iterations to converge, since they are periods during the morning peak. In contrast, Tue24h and MonSun behave like

Table 2 Impact of the centralized searches on the running time (in seconds) of the traffic assignment procedure for our scenarios. We evaluate the influence of using customizable CHs, reordering the OD-pairs (sorted), computing k shortest paths simultaneously, and using SSE and AVX instructions. The prior state of the art and our default configuration are highlighted in bold.

algo	sorted	k	SSE	AVX	Tue30m	Tue01h	Tue02h	Tue24h	MonSun
Dij	○	1	○	○	1857.27	3582.67	6028.48	20128.24	128993.50
CH	○	**1**	○	○	**35.85**	**52.81**	**76.54**	**183.59**	**1082.20**
CH	•	1	○	○	35.06	48.72	67.83	153.90	851.63
CH	•	4	○	○	34.47	45.75	62.20	130.07	656.66
CH	•	4	•	○	30.27	39.85	52.89	99.22	507.49
CH	•	8	○	○	38.04	50.80	71.11	151.24	763.45
CH	•	8	•	○	29.41	36.23	46.79	85.89	410.03
CH	•	8	○	•	29.64	36.33	45.51	81.98	397.91
CH	•	16	○	•	29.33	35.08	44.34	75.45	352.90
CH	•	32	○	•	29.87	37.30	46.71	72.85	322.04
CH	•	64	○	•	34.79	43.98	54.75	85.28	354.15
CCH	○	1	○	○	11.99	22.75	40.02	132.54	803.48
CCH	•	1	○	○	10.62	19.59	34.12	108.08	659.57
CCH	•	4	○	○	9.71	17.28	29.50	87.70	499.25
CCH	•	4	•	○	6.28	10.60	17.82	51.91	298.91
CCH	•	8	○	○	11.71	20.72	35.14	102.89	581.38
CCH	•	8	•	○	5.31	8.61	14.02	39.25	218.79
CCH	•	8	○	•	4.86	7.84	12.66	35.11	195.63
CCH	•	16	○	•	4.58	6.98	10.83	27.48	144.81
CCH	•	**32**	○	•	**4.60**	**6.89**	**10.54**	**25.14**	**126.70**
CCH	•	64	○	•	5.91	8.92	13.55	29.78	145.33

Table 3 Impact of core-level parallelization on the performance of the traffic assignment procedure. We report for each scenario the time spent on queries and the total running time (both in seconds).

	#	Tue30m		Tue01h		Tue02h		Tue24h		MonSun	
algo	cores	qry	total	qry	total	qry	total	qry	total	qry	total
CH	4	4.14	24.76	8.21	28.33	14.85	34.41	46.28	54.97	295.39	304.33
CCH	1	3.12	4.60	5.41	6.89	9.05	10.54	24.42	25.14	125.97	126.70
CCH	2	1.85	3.32	3.18	4.66	5.23	6.71	13.46	14.18	70.18	70.91
CCH	3	1.38	2.86	2.29	3.76	3.63	5.10	9.09	9.82	47.12	47.85
CCH	4	1.17	2.65	1.82	3.31	2.86	4.33	6.95	7.67	35.90	36.64

off-peak periods, since the traffic is considered to be uniformly distributed over the period of analysis. In relatively uncongested networks, the edge flows are in the range where the travel time functions are almost flat, the updated travel times are closer to the initial ones, and the equilibrium flow pattern is more similar to the initial solution [37]. The peak scenarios are close to equilibrium after about twelve iterations, the off-peak scenarios after about six iterations.

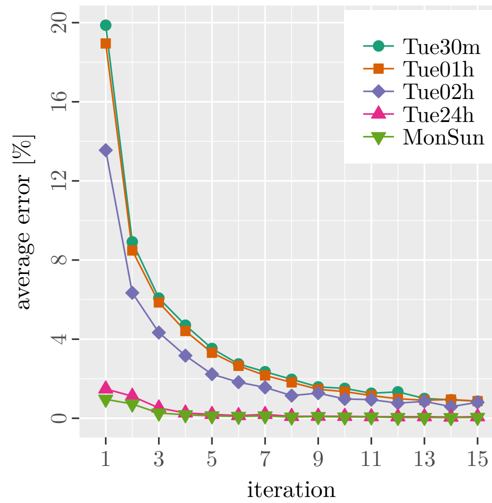
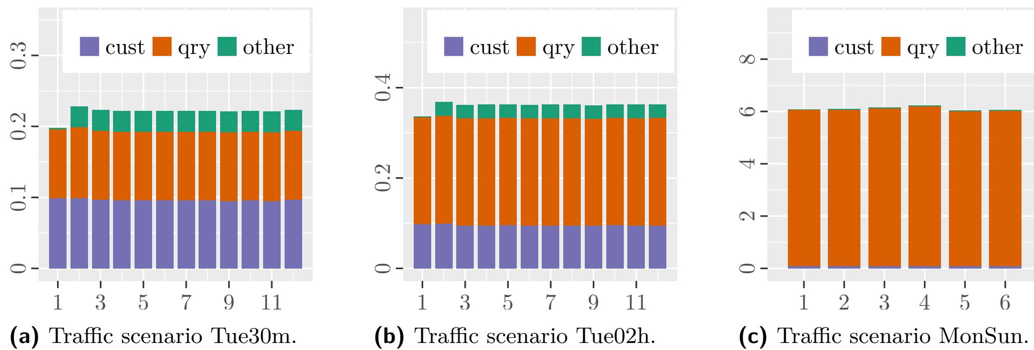


Figure 2 Convergence of the traffic assignment procedure. The plot shows the average deviation of the OD-travel-costs in each iteration from the OD-travel-costs at equilibrium.



(a) Traffic scenario Tue30m. **(b)** Traffic scenario Tue02h. **(c)** Traffic scenario MonSun.

Figure 3 Time in seconds (vertical) spent in each iteration (horizontal) for the multi-threaded traffic assignment procedure (using all 4 cores). For MonSun, customization and other work are hardly visible, since they take only 1.59 % and 0.41 % of the total time, respectively.

Time per Iteration. Figure 3 plots the running time (per phase) that our multi-threaded traffic assignment spends in each iteration. First, we observe that the procedure spends the same amount of time in each iteration. Although the inherent hierarchy of the network is weakened while computing an equilibrium flow pattern [26], this is expected since the performance of both CCH customization and queries is mostly metric-independent [12]. For our smallest scenario, customization takes 44 % of the total time. This decreases to 27 % for Tue02h, and to 2 % for our largest scenario. All other work, such as the line search, the edge updates, and the convergence checks, is negligible (only 12 % even for the smallest scenario).

6 Conclusion

We accelerated the computation of equilibrium flow patterns significantly. This was achieved by carefully engineering a number of building blocks, including customization, an improved CCH query algorithm, centralized searches, and parallelism at multi-core and instruction levels. Moreover, the improved, unified CCH query algorithm (replacing both original query algorithms) and the centralized elimination tree search are not restricted to the traffic

assignment problem, but generally applicable to (batched) point-to-point shortest paths. All building blocks were evaluated on real-world data used in production systems. On a metropolitan area encompassing more than 2.7 million inhabitants, we compute the flow pattern for a typical two-hour morning peak in merely 4.3 seconds, 18 times faster than the state of the art, and 1390 times faster than the Dijkstra-based baseline. This makes interactive urban transportation planning applications practical.

For traffic scenarios where the shortest-path computations are still the performance bottleneck of the traffic assignment procedure, it would be interesting to process only a sample of the demand in early iterations, and add more and more OD-pairs in subsequent iterations. In addition, we are interested in testing our traffic assignment procedure on benchmark instances that are even an order of magnitude larger than the one used in this work. Since we are not aware of any such real-world instances, we plan to work on *realistic* generators for synthetic OD-pairs. Finally, it would be interesting to study the efficient computation of time-dependent traffic flow profiles.

References

1. Frédéric Babonneau and Jean-Philippe Vial. Test instances for the traffic assignment problem. Technical report, Ordecsys, 2008.
2. Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route planning in transportation networks. In *Algorithm Engineering: Selected Results and Surveys*, pages 19–80. Springer, 2016. `doi:10.1007/978-3-319-49487-6_2`.
3. Reinhard Bauer, Tobias Columbus, Ignaz Rutter, and Dorothea Wagner. Search-space size in contraction hierarchies. *Theoretical Computer Science*, 645:112–127, 2016. `doi:10.1016/j.tcs.2016.07.003`.
4. Reinhard Bauer and Daniel Delling. SHARC: Fast and robust unidirectional routing. *ACM Journal of Experimental Algorithmics*, 14:2.4:1–2.4:29, 2009. `doi:10.1145/1498698.1537599`.
5. Martin Beckmann, C. Bart McGuire, and Christopher B. Winsten. *Studies in the Economics of Transportation*. Yale University Press, 1956.
6. M. Bruynooghe, A. Gilbert, and M. Sakarovich. Une méthode d'affectation du traffic. In *Proceedings of the 4th International Symposium on the Theory of Road Traffic Flow*, 1968.
7. Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck. PHAST: Hardware-accelerated shortest path trees. *Journal of Parallel and Distributed Computing*, 73(7):940–952, 2013. `doi:10.1016/j.jpdc.2012.02.007`.
8. Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable route planning in road networks. *Transportation Science*, 51(2):566–591, 2017. `doi:10.1287/trsc.2014.0579`.
9. Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Faster batched shortest paths in road networks. In *Proceedings of the 11th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'11)*, pages 52–63, 2011. `doi:10.4230/OASIcs.ATMOS.2011.52`.
10. Daniel Delling and Renato F. Werneck. Customizable point-of-interest queries in road networks. *IEEE Transactions on Knowledge and Data Engineering*, 27(3):686–698, 2015. `doi:10.1109/TKDE.2014.2345386`.
11. Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*. American Mathematical Society, 2009.

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

13 Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

14 Alexandros Efentakis and Dieter Pfoser. Optimizing landmark-based routing and preprocessing. In *Proceedings of the 6th ACM SIGSPATIAL International Workshop on Computational Transportation Science (IWCTS'13)*, pages 25–30, 2013. doi:10.1145/2533828.2533838.

15 Alexandros Efentakis and Dieter Pfoser. GRASP. Extending graph separators for the single-source shortest-path problem. In *Proceedings of the 22th Annual European Symposium on Algorithms (ESA'14)*, pages 358–370, 2014. doi:10.1007/978-3-662-44777-2_30.

16 Alexandros Efentakis, Dieter Pfoser, and Yannis Vassiliou. SALT. A unified framework for all shortest-path query variants on road networks. In *Proceedings of the 14th International Symposium on Experimental Algorithms (SEA'15)*, pages 298–311, 2015. doi:10.1007/978-3-319-20086-6_23.

17 Marguerite Frank and Philip Wolfe. An algorithm for quadratic programming. *Naval Research Logistics Quarterly*, 3(1-2):95–110, 1956. doi:10.1002/nav.3800030109.

18 Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact routing in large road networks using contraction hierarchies. *Transportation Science*, 46(3):388–404, 2012. doi:10.1287/trsc.1110.0401.

19 Alan George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, 1973. doi:10.1137/0710032.

20 Moritz Hilger, Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Fast point-to-point shortest path computations with arc-flags. In *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, pages 41–72. American Mathematical Society, 2009.

21 Donald B. Johnson. Priority queues with update and finding minimum spanning trees. *Information Processing Letters*, 4(3):53–57, 1975. doi:10.1016/0020-0190(75)90001-0.

22 Sebastian Knopp, Peter Sanders, Dominik Schultes, Frank Schulz, and Dorothea Wagner. Computing many-to-many shortest paths using highway hierarchies. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX'07)*, pages 36–45, 2007. doi:10.1137/1.9781611972870.4.

23 Donald E. Knuth. *The Art of Computer Programming: Sorting and Searching.* Addison-Wesley, 1998.

24 Daniel Kusswurm. *Modern X86 Assembly Language Programming: 32-bit, 64-bit, SSE, and AVX.* Apress, 2014.

25 Larry J. LeBlanc, Edward K. Morlok, and William P. Pierskalla. An efficient approach to solving the road network equilibrium traffic assignment problem. *Transportation Research*, 9(5):309–318, 1975. doi:10.1016/0041-1647(75)90030-1.

26 Dennis Luxen and Peter Sanders. Hierarchy decomposition for faster user equilibria on road networks. In *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*, pages 242–253, 2011. doi:10.1007/978-3-642-20662-7_21.

27 Nicolai Mallig, Martin Kagerbauer, and Peter Vortisch. mobitopp - A modular agent-based travel demand modelling framework. In *Proceedings of the 2nd International Workshop on Agent-based Mobility, Traffic and Transportation Models, Methodologies and Applications (ABMTRANS'13)*, pages 854–859, 2013. doi:10.1016/j.procs.2013.06.114.

28 Nicolai Mallig and Peter Vortisch. Modeling car passenger trips in mobitopp. In *Proceedings of the 4nd International Workshop on Agent-based Mobility, Traffic and Transportation Models, Methodologies and Applications (ABMTRANS'15)*, pages 938–943, 2015. doi:10.1016/j.procs.2015.05.169.

29 Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer, 2008. doi:10.1007/978-3-540-77978-0.

30 Enock T. Mtoi and Ren Moses. Calibration and evaluation of link congestion functions: Applying intrinsic sensitivity of link speed as a practical consideration to heterogeneous facility types within urban network. *Journal of Transportation Technologies*, 4:141–149, 2014. doi:10.4236/jtts.2014.42014.

31 John D. Murchland. Road traffic distribution in equilibrium. In *Proceedings of Mathematical Methods in the Economic Sciences*, 1969.

32 Michael Patriksson. *The Traffic Assignment Problem: Models and Methods*. Topics in Transportation. VSP, 1994.

33 Srinivas Peeta and Athanasios K. Ziliaskopoulos. Foundations of dynamic traffic assignment: The past, the present and the future. *Networks and Spatial Economics*, 1(3-4):233–265, 2001. doi:10.1023/A:1012827724856.

34 Peter Sanders and Dominik Schultes. Highway hierarchies hasten exact shortest path queries. In *Proceedings of the 13th Annual European Symposium on Algorithms (ESA'05)*, pages 568–579, 2005. doi:10.1007/11561071_51.

35 Aaron Schild and Christian Sommer. On balanced separators in road networks. In *Proceedings of the 14th International Symposium on Experimental Algorithms (SEA'15)*, pages 286–297, 2015. doi:10.1007/978-3-319-20086-6_22.

36 Johannes Schlaich, Udo Heidl, and R. Pohlner. Verkehrsmodellierung für die Region Stuttgart: Schlussbericht. unpublished, 2011.

37 Yosef Sheffi. *Urban Transportation Networks: Equilibrium Analysis with Mathematical Programming Methods*. Prentice Hall, 1985.

38 Verband Region Stuttgart. Mobilität und Verkehr in der Region Stuttgart 2009/2010: Regionale Haushaltsbefragung zum Verkehrsverhalten. *Schriftenreihe Verband Region Stuttgart*, 29:1–138, 2011.

39 John G. Wardrop. Some theoretical aspects of road traffic research. In *Proceedings of the Institution of Civil Engineers*, pages 325–362, 1952. doi:10.1680/ipeds.1952.11259.

40 Hiroki Yanagisawa. A multi-source label-correcting algorithm for the all-pairs shortest paths problem. In *24th IEEE International Symposium on Parallel and Distributed Processing (IPDPS'10)*, pages 1–10, 2010. doi:10.1109/IPDPS.2010.5470362.