Dynamic Time-Dependent Routing in Road Networks Through Sampling*

Ben Strasser

Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
strasser@kit.edu

Abstract

We study the earliest arrival and profile problems in road networks with time-dependent functions as arc weights and dynamic updates. We present and experimentally evaluate simple, sampling-based, heuristic algorithms. Our evaluation is performed on large, current, production-grade road graph data with time-dependent arc weights. It clearly shows that the proposed algorithms are fast and compute paths with a sufficiently small error for most practical applications. We experimentally compare our algorithm against the current state-of-the-art. Our experiments reveal, that the memory consumption of existing algorithms is prohibitive on large instances. Our approach does not suffer from this limitation. Further, our algorithm is the only competitor able to answer profile queries on all test instances below 50ms. As our algorithm is simple to implement, we believe that it is a good fit for many realworld applications.

1998 ACM Subject Classification G.2.2 Graph Theory, E.1 Data Structures

Keywords and phrases shortest path, time-dependent, road graphs, preprocessing

Digital Object Identifier 10.4230/OASIcs.ATMOS.2017.3

1 Introduction

Routing in road networks is an important and well-researched topic [1]. It comes in many varieties. Nearly all formalize the road network as a weighted, directed graph. Nodes correspond to positions in the network. Edges correspond to road segments. The weight of an edge is the travel time a car needs to traverse the corresponding road segment. A common assumption is that these travel times are time-independent scalars that do not change throughout the day. With this assumption, the problem boils down to the classical shortest path problem: Given a weighted graph, a source node s and a target node t, find a shortest st-path.

A problem is that road networks are huge. Even near-linear-running-time algorithms, such as Dijkstra's algorithm [14], are too slow. Therefore, speedup based techniques were developped [1]: In a first slow preprocessing phase, auxiliary data is computed. In a second fast query phase, shortest path queries are answered in sublinear time using this auxiliary data. Contraction Hierarchies (CH) [18] are a popular technique following this setup.

We study extensions of this problem setting: the earliest arrival and profile problems [7]. The input of the earliest arrival problem consists of nodes s, t, and a departure time τ. The task consists of computing a st-path with minimum arrival time. The input of profile problem consists only of s and t. The output consists of a function that maps a departure time onto the corresponding minimum arrival time. Formulated differently, the profile problem solves

* Support by DFG grant FOR-2083.

the earliest arrival time problem for every departure time. We refer to the scenario without congestions as *time-independent*. Dijkstra's algorithm [14] can be augmented to solve the earliest arrival problem [15]. However, it remains too slow.

Predicted congestions are usually modeled using functions as edge weights. Every edge e is associated with a function $f_e(x)$ that maps the entry time x of a car into e onto the travel time $f_e(x)$. Following [26], we require that waiting must not be beneficial. This property is called the FIFO-property and formally states that $x + f_e(x) \leq y + f_e(y)$ for all $x \leq y$. Without this property, the problem is NP-hard. We further assume that the functions are periodic with a period length of one day. We refer to the lower bound of f_e as e's *freeflow weight*. The set of functions is the *congestion prediction*. The prediction is often done months in advance. All functions are available during preprocessing. Predictions are usually derived from past GPS traces or a traffic simulation. Routing with predicted congestions is a well-researched topic [11, 12, 8, 25, 10, 4, 17, 6, 22].

Predicted congestions give a very rough traffic estimation. Often the actual traffic situation differs significantly from the prediction. For this reason, we also consider *realtime congestions*. We assume that there is a system that measures the current realtime travel time for every edge. A realtime routing system computes shortest paths according to the realtime travel times. Such a routing system usually works in three phases. In the first preprocessing phase, only the road network but not the travel times are known. This phase can be slow. The second customization phase introducess the travel times. The third phase is the query phase and computes paths. The customization should run within a few seconds. It is rerun at regular intervals, such as for example every 10s, to incorporate the current traffic situation. Solutions to routing with only realtime congestions problem are MLD/CRP [28, 21, 9] and Customizable Contraction Hierarchies (CCH) [13].

Ideally, we want a routing system that accounts for predicted and realtime congestions. This scenario is also known as *dynamic time-dependent routing*. There are on this topic [10, 6]. In this paper, we propose an algorithm TD-S that solves the earliest arrival problem with predicted congestions. The acronym stands for Time-Dependent Sampling. We extend it to TD-S+P, which solves the profile problem, and to TD-S+D which additionally takes realtime congestions into account.

The routing problem with no congestion and with only realtime congestions can be solved efficiently and exactly, i.e., the computed paths are shortest paths. However, when taking predicted congestions into account, many proposed algorithms compute paths with an error. Let d denote the length of the computed path and d_{opt} the length of a shortest path. We define the absolute error as $|d - d_{\text{opt}}|$ and the relative error as $\frac{|d-d_{\text{opt}}|}{d_{\text{opt}}}$. Ideally, we would like to compute paths with no error but this is not an easy task.

Fortunately, for most applications a small error is acceptable as a traveler will not notice a slightly suboptimal path. Further, the employed predictions have associated uncertainties. Minimizing the error when it is below the uncertainty is not useful. A shortest path with respect to the prediction is not necessarily shortest with respect to reality. Paths with a small error are therefore indiscernible from "optimal" ones in terms of quality in practice. Unfortunately, the predictions that we have access to do not quantify these uncertainties. However, it is easy to see that the uncertainty is huge: The time a typical German traffic light needs to cycle though its program is between 30s and 120s [16]. Traffic lights often adapt to the actual realtime traffic. Predictions can therefore not take red light phases into account as they cannot be predicted months in advance. Every traffic light on a shortest path thus induces an uncertainty on the same scale as its program cycle length. Consider a 2h path with a relative error of 1%. The corresponding absolute error is 72s, i.e., one to three traffic lights. An error of 1% is thus small.

On sufficiently small instances, the routing problem with predicted congestions can be solved optimally using TCH [4]. Unfortunately, TCH requires a lot of memory. We compare against an open-source implementation by the primary author [3]. We cannot answer queries on a current Europe instance even with a 128GB machine – a show stopper if the program needs to run on a client's desktop machine, which usually has far less memory. A further downside of TCH is its complexity. It also makes extensions, for example realtime congestions, difficult. Further, TCHs are only optimally if one assumes arbitrary precise numbers with $O(1)$-operations. Making sure that numeric instabilities do not get out of hand is possible but not easy. The simplicity of an algorithm is a huge asset in applications. Unfortunately, it is difficult to formalize when an algorithm is "simple" and when not. For the context of this paper, we use the following crude approximation: An algorithm is complex, if it combines functions using linking and merging operations[1]. Numeric stability issues are typically caused by these operations. Avoiding these operations thus also avoids these issues.

Even though a lot of research into time-dependent routing exists, in-depth experimental studies of the very simple approaches are, to the best of our knowledge, lacking. How good is an optimal solution to the time-independent routing problem with respect to the routing problem with predicted congestions? We refer to this simple approach as *Freeflow* heuristic. A further interesting question is, how good is an optimal solution to the earliest arrival problem with predicted congestions with respect to the earliest arrival problem with both congestion types? We refer to this approach as *Predicted-Path* heuristic. Both heuristics obviously compute paths with an error. However, how bad are these errors in practice? To the best of our knowledge, this fundamental question has not been investigated in existing papers. One of the objectives of our work is to fill this gap.

Our proposed algorithms are extensions of the Freeflow heuristic that trade an increased query running time for a significant decrease in error. TD-S and TD-S+P can be implemented with minimum effort assuming that a blackbox solution to the routing problem with no congestions is available. TD-S+D additionally needs a blackbox that can handle realtime congestions. Our program is build on top of the open-source CH and CCH implementations of RoutingKit [30]. Fortunately, one can swap them out for any other algorithms that fulfills the requirements. An open-source TD-S implementation is available [31].

An experimental error evaluation crucially depends on high quality realworld data. Fortunately, PTV [27] has given us access to their current production-grade congestion data for 2017 specifically to evaluate TD-S. As this data has a significant commercial value, we are not allowed to freely share it without explicitly consent of PTV. We are not aware of high-quality open congestion prediction data. OSM does **not** include predictions.

1.1 Related Work

There exist a lot of papers beside the already mentioned ones. We do not build upon them nor rerun their experiments. We therefore limit our description to a brief overview. For a detailed survey, we refer to [2].

The authors of [25] observed that ALT [19], a time-independent speedup technique, can be applied to the graph weighted with the freeflow weights. This yields the simple algorithm named TD-ALT. In [10], the technique was extended to TD-CALT by first coarsening the graph and then applying TD-ALT to the core. TD-CALT was evaluated

[1] Definitions: Merge respectively link of f and g is h such that $h(x) = \min\{f(x), g(x)\}$ respectively $h(x) = f(x) + g(f(x) + x)$

with respect to realtime congestion. Coarsening, and thus TD-CALT, requires linking and merging operations. In [8], SHARC [5] (a combination of Arc-Flags [24] with shortcuts and coarsening) was extended to the time-dependent setting. SHARC was combined with ALT yielding L-SHARC [8]. Another technique is FLAT [22]. Here the idea is to precompute the solutions from a set of landmarks to every node and during the query phase to route all sufficiently long paths through a landmark. FLAT also works without linking- and merging operations. In [23], it was extended to CFLAT. In [6], MLD/CRP was combined with travel time functions yielding TD-CRP, which can also be used in the dynamic scenario.

A deep structural insight was shown in [17]. Graphs and source and target nodes s and t exist, such that the st-profile contains a superpolynomial number of paths. This implies that the profile problem cannot be solved on every instance with polynomial running time. Fortunately, realworld graphs do not have this worst-case structure.

1.2 Outline

We start by describing our implementation of the Freeflow heuristic. We further present a slight variation called Avgflow heuristic. Afterwards, we describe TD-S and the profile extension TD-S+P. In the next step, we introduce the dynamic extension TD-S+D. Finally, we experimentally evaluate all three algorithms on production-grade instances and compare our results with TCH.

2 The Freeflow and Avgflow Heuristic

The freeflow travel time along an edge assumes that there is no congestion. Formally, the freeflow travel time of an edge e is the minimum value of e's travel time function f_e. The Freeflow heuristic works in two steps:

1. Find shortest time-independent path H with respect to the freeflow travel time.
2. Compute the time-dependent travel time along H for the given departure time.

The first step is independent of the edge function weights. The functions are only used in the second step. The running time is dominated by the first step. Fortunately, this step can be accelerated using any time-independent speedup technique. In our implementation, we use a CH. In a preprocessing step, we compute a CH for the road graph weighted by freeflow travel times. The first step of the Freeflow heuristic computes H using a CH-query.

A slight variation is the Avgflow heuristic. It is exactly the same as the Freeflow heuristic but uses the average travel time instead of the minimum travel time for every edge.

3 Time-Dependent-Sampling: TD-S

The Freeflow heuristic never reroutes based on the current traffic situation. TD-S tries to alleviate this problem. Similarly to the Freeflow heuristic, TD+S's query algorithm works in two steps:

1. Compute a subgraph H.
2. Run the time-dependent extension of Dijkstra's algorithm on the subgraph.

If a shortest time-dependent path P is part of H, then P is found by TD-S and the computed arrival time is exact. Otherwise, TD-S's solution has an error.

We compute the subgraph using a sampling approach. We define a constant number k of time-intervals. Within each time-interval, we average the time-dependent travel times. For every interval, we thus obtain a time-independent graph. For every graph, we compute a shortest time-independent path. The union of these paths is the subgraph H.

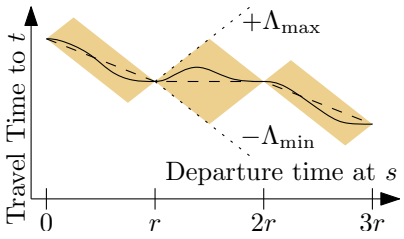

Figure 1 The st-profile is the solid line. It must be in the orange area. The dashed function is our approximation.

Similarly to the Freeflow heuristic, the shortest time-independent path computations can be accelerated using existing speedup techniques. In our implementation, we compute for each time-interval a time-independent CH. The first step of TD+S executes k CH-queries. The second step uses the time-dependent extension of Dijkstra's algorithm restricted to the subgraph H.

The Freeflow heuristic can be seen as a special case of TD+S, where subgraph consists of a shortest freeflow path. The number of time-intervals k is a trade-off between query and preprocessing running times, and space consumption on the one hand, and solution error on the other hand. We recommend using small numbers below 10 for k. The chosen interval boundaries should reflect rush hours in the input data.

4 Computing Profiles: TD-S+P

The query algorithm of TD-S+P also works in two steps:

1. Compute a subgraph H.
2. Sample at regular intervals the travel time by running the time-dependent extension of Dijkstra's algorithm restricted to the subgraph H.

The subgraph computation step is the same as for TD-S.

We interpolate linearly between the sampled travel times. For a sampling rate of 10 min, our algorithm first computes the subgraph H, then runs Dijkstra's algorithm with the departure times 0:00, 0:10, 0:20...23:50 restricted to H. Denote by $a_1, a_2, a_3 \dots a_{144}$ the computed arrival times. For example, the computed arrival time for departure time 0:07 is $0.3a_1 + 0.7a_2$.

We can bound the error that the profile algorithm induces on top of the error of TD-S using a theoretical argument: Denote by $\Lambda_{\max}$ the maximum and by $-\Lambda_{\min}$ the minimum slope of every linear piece in every profile function and by r the sample rate. As shown in the appendix B, the maximum absolute error is bounded by $r(\Lambda_{\max} + \Lambda_{\min})/4$. The proof idea is illustrated in Figure 1. Following [22], we assume that $\Lambda_{\max}$ and $-\Lambda_{\min}$ are bounded by a small constant. In the same paper, the values $\Lambda_{\max} = 0.19$ and $\Lambda_{\min} = 0.15$ were experimentally estimated for the Berlin instance. With $r = 10$min this gives a maximum error of 51 seconds. Our analysis is very similar to the TRAP oracle from [22]. Unfortunately, this bounds on the error induced by the profile algorithm. The base algorithm TD-S induces additional error, which were not able to bound using theoretical arguments.

TD-S+P is faster than iteratively running TD-S as the subgraph is only computed once. Further, Dijkstra's algorithm iteratively runs on the same small subgraph H. The first run loads H into the cache and thus all subsequent runs incur nearly no cache misses.

Figure 2 illustrates a typical profile computed with TD-S+P. The source node of the example is near the inner city of Stuttgart, a city notoriously known for its large daily traffic

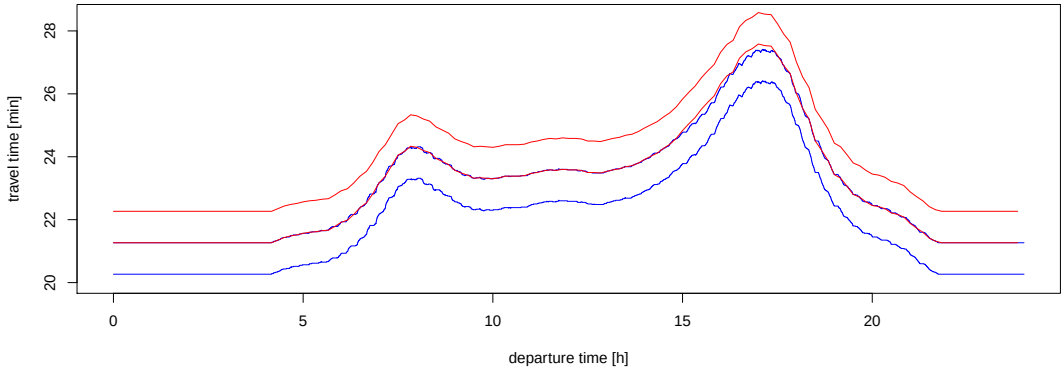

Figure 2 Example profile over 24h. The red curve (top) was computed with TD-S+P, while the blue one (bottom) is the exact solution. The middle overlapping curves are the actual profiles. To improve readability, we plot both curves a second time shifted by one unit on the y-axis.

congestions, in front of the central train station. The target node lies in Denkendorf, a village about 20min south-east outside of Stuttgart. The computed profile is smoother than the "exact" profile, which has a lot of small fluctuations. However, this does not mean that the "exact" profile is more accurate. Most of the small fluctuations are within only a few seconds, i.e., well below the accuracy of the input data. Formulated differently, these fluctuations are imprecisions in the input that are propagated to the output. Fortunately, the general form of both curves is very similar, which is the important information. The largest absolute error is 19s, respectively 1.17% relative error, at the peak of the evening spike. Recall, that crossing a single red traffic light can induce a delay of more than 19s.

5 Dynamic Traffic: TD-S+D

TD-S and TD-S+P work with predicted congestions. However, in many applications we must also take realtime congestions into account. We adapt TD-S by modifying the computation of the subgraph H yielding TD-S+D. The second step is left unchanged.

The subgraph H is the union of k paths. TD-S+D adds a shortest st-path according to the current realtime traffic as $k+1$-th path to the union. To efficiently determine this path, a efficient solution to the routing problem with realtime congestions is needed. We use the CCH algorithm [13] with a FlowCutter contraction order [20].

In the preprocessing step, TD-S+D computes a CCH in addition to the k CHs of TD-S. At regular time intervals, such as every 10s, TD-S+D updates the CCH edge weights to reflect the realtime traffic situation. This update involves a CCH customization, which runs within at most a few seconds. A TD-S+D query consists of running k CH queries and one CCH query. The subgraph H is the union of the $k+1$ shortest paths. Finally, the time-dependent extension of Dijkstra's algorithm is run restricted to the subgraph H.

5.1 Simulating Traffic

We have access to production-grade time-dependent edge data. Unfortunately, we do not have access to good measured realtime traffic. We therefore simulate realtime congestions to study the performance of TD-S+D.

For an earliest arrival time query from s to t with departure time τ, we first compute the shortest time-dependent path P with respect to the historic travel times. On P we generate three traffic congestions by picking three random start edges. From each of these edges we

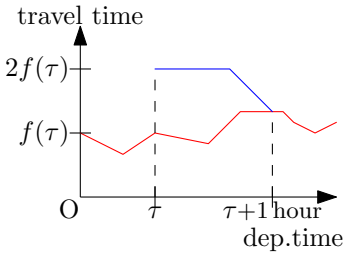

Figure 3 Travel-time weight function with simulated congestion.

Table 1 Node count, edge count, percentage of time dependent edges, and number of break points per time-dependent weight function for all instances.

	Nodes [K]	Directed Edges [K]	TD-Edges [%]	avg. Break Points per TD-Edge
Lux	54	116	34	30.9
Ger	7 248	15 752	29	29.6
OGer	4 688	10 796	7	17.6
CEur	25 758	55 504	27	27.5

follow P for 4min, yielding three subpaths. For every edge e in a subpath, we generate a congestion according to the construction illustrated in Figure 3. The red line is the original travel time function f. The blue line is the congested function.

Denote by f the travel time function of e. We modify f by doubling the travel time at $f(\tau)$ and assume that it remains constant for some time. The congestion should be gone at $\tau + 1h$. To maintain the FIFO-property, the modified function must have slope of -1 before $\tau + 1$ before joining the predicted travel time. In the awkward and rare situation where $2f(\tau) < f(\tau + 1h)$, we do not generate a congestion.

6 Experimental Results

6.1 Setup

We use three production-grade instances (Lux, Ger, CEur) with traffic predictions for the first half of 2017. To compare with related work, we further include an a decade-old Germany instance (OGer) in our study. We thank PTV for giving us access to this data. All instances model a car on a typical Tuesday. The instance sizes are depicted in Table 1. The European graph contains several central European countries as illustrated in Figure 4. Experiments on further instances are reported in an older ArXiv version of our work [29]. The results are essentially comparable to the experiments presented in this paper on all non-synthetic, realworld instances. Additional experiments that evaluate the achieved errors in detail are in the appendix.

Our experiments were run on a machine with a E5-2670 processor and 64GB of DDR3-1600. All reported running times are sequential. For every instance, we generated 10^5 queries with source stop, target stop and source time picked uniformly at random. All experiments use the same set of test queries.

The large instances (Ger, CEur) are useful to investigate scaling behavior. The old instance (OGer) is useful to compare with related work. The city with periphery instance (Lux) is useful to investigate errors in urban contexts.

Figure 4 Central Europe.

Table 2 Number of exact time-dependent queries and absolute and relative errors for Freeflow, TD-S+4, and TD-S+9. "Q99" refers to the 99%-quantile and "Q99.9" the 99.9%-quantile.

Graph	Algo	Exact [%]	Relative Error [%]				Absolute Error [s]			
			Avg	Q99	Q99.9	Max	Avg	Q99	Q99.9	Max
Lux	Freeflow	80.0	0.244	5.1	11.5	28.1	5.0	106	235	356
Lux	Avgflow	81.0	0.123	2.5	6.4	19.4	2.5	49	143	329
Lux	TD-S+4	97.7	0.008	0.2	1.5	4.9	0.2	4	30	141
Lux	TD-S+9	99.6	<0.001	0.0	0.1	1.7	<0.1	0	3	27
Ger	Freeflow	67.9	0.085	1.5	3.1	12.4	11.1	200	417	825
Ger	Avgflow	69.2	0.044	0.8	1.9	10.3	5.9	113	284	587
Ger	TD-S+4	94.6	0.005	0.1	1.0	3.0	0.8	17	159	474
Ger	TD-S+9	98.2	0.001	<0.1	0.4	3.0	0.3	1	76	374
OGer	Freeflow	60.7	0.140	2.0	4.7	12.4	15.9	219	465	1 104
OGer	Avgflow	68.8	0.050	0.9	2.2	6.5	5.7	96	227	619
OGer	TD-S+4	96.4	0.002	0.1	0.4	2.0	0.3	6	47	333
OGer	TD-S+9	98.5	0.001	<0.1	0.2	2.0	0.1	1	24	276
CEur	Freeflow	54.9	0.089	1.4	2.7	10.8	26.4	428	833	1 477
CEur	Avgflow	55.8	0.048	0.8	1.7	6.6	14.2	235	507	1 069
CEur	TD-S+4	91.1	0.006	0.2	0.7	3.8	1.8	47	226	547
CEur	TD-S+9	96.8	0.001	<0.1	0.3	1.2	0.5	6	109	397

We evaluate TD-S with two selections of time windows. TD-S+4 uses the windows 0:00–5:00, 6:00–9:00, 11:00–14:00, and 16:00–19:00. TD-S+9 uses the windows 0:00–4:00, 5:50–6:10, 6:50–7:10, 7:50–8:10, 10:00–12:00, 12:00–14:00, 16:00–17:00, 17:00–18:00, and 19:00–21:00. These time windows reflect the rush hours in the dataset and were created using manual trial-and-error. Developing algorithms to automatically determine time windows bounds is an interesting avenue for future research.

To provide an in-depth comparison with related work, we run TCH on our test instances and test machine. The implementation we used is based on KaTCH [3], an open-source reimplementation of the algorithm of [4] by the primary author. Unfortunately, it only supports earliest arrival queries and no profile queries. We do not have access to implementations of other competing algorithms.

In Table 2, we report the errors for various algorithms and all instances. We report the percentage of queries that are solved exactly, the average, maximum, and quantiles[2] of the

[2] Definition x-quantile of n values: Sort n values increasingly, pick the $(n{\cdot}x)$-th value. 0-quantile is min. 0.5-quantile is median. 1-quantile is max.

Table 3 Average preprocessing and running times and memory consumption of various algorithms.

Graph	TD-Dijkstra	Freeflow	Avgflow	TD-S+4	TD-S+9	TCH
Average Query Running Time [ms]						
Lux	4	0.02	0.02	0.11	0.26	0.18
Ger	1 116	0.19	0.20	0.99	3.28	1.81
OGer	813	0.12	0.14	0.97	2.09	1.12
CEur	4 440	0.42	0.29	3.83	6.85	OOM
Max. Query Memory [MiB]						
Lux	13	17	17	29	47	328
Ger	1 550	2 132	2 130	3 630	6 127	42 857
OGer	461	855	854	1 880	3 589	8 153
CEur	4 980	7 058	7 053	12 411	21 336	>131 072
Total Preprocessing Running Time [min]						
Lux	—	<0.1	<0.1	<0.1	0.1	0.6
Ger	—	1.6	2.2	7.6	14.7	86.2
OGer	—	1.5	1.6	5.9	16.4	26.8
CEur	—	8.6	8.1	33.9	70.7	381.4

relative and absolute errors. The number of exactly solved queries decreases with instance size as the paths lengths grow with size. A longer path has more opportunities for errors. Similarly, the absolute errors grow with instance size as the paths get longer. The relative errors quantiles shrink with growing instance size, as individual errors have a smaller impact. The average errors are very small, as most queries are answered exactly. We report maximum error values as most related papers report them. However, these values are very sensitive to the random seed used during the query generation. The Freeflow heuristic achieves significantly lower error values than we expected at first. The Avgflow heuristic slightly beats the Freeflow heuristic. For some applications, these errors are low enough. However ideally, we want to have even lower error values. Fortunately, using TD-S significantly lower values are achievable. TD-S+9 answers 99.6% of the queries exactly in an urban scenario and 96.8% on a continental-sized instance. Even the 99.9%-quantiles are well below a relative error of 0.5%. TD-S+4 has larger errors as it uses fewer time windows. Fortunately, the query running times and the memory footprint of TD-S+4 are lower.

In Table 3, we compare query and preprocessing running times and memory consumption. We observe that the lower the solution error of an algorithm is, the more memory it requires. TCH is guaranteed to be exact. Unfortunately, its memory requirements are prohibitive on CEur. We tried to run it on a 128GB machine but got out-of-memory crashes while executing queries. The TCH preprocessing algorithm writes no longer needed data to the disk and evicts it from main memory. We were therefore able to run the preprocessing step. However, the whole data structure must be loaded into main memory to answer queries. TD-S+9 only needs 21GB on the same instance and TD-S+4 only 12GB. The 5GB memory consumption of TD-Dijkstra consists essentially of the input data. TD-S+4 only needs about 2.4 times the memory required by the input. TD-S+9 needs 4.1 times the memory. TD-S+4 has an about a factor 10 lower preprocessing time than TCH. The Freeflow heuristic has, for obvious reasons, the fastest query running time. It is followed by TD-S+4 which is about 33% to 50% faster than TCH. TD-S+9 is slightly slower than TCH.

Table 4 Average 24h-profile running times in milliseconds. "SubG" is the subgraph comp. time.

	Freeflow		Avgflow		TD-S+P4		TD-S+P9	
Graph	SubG.	Total	SubG.	Total	SubG.	Total	SubG.	Total
Lux	<0.1	2.6	<0.1	2.6	0.1	3.0	0.3	3.4
Ger	0.2	18.1	0.2	18.3	0.7	19.5	1.7	22.2
OGer	0.1	10.0	0.1	9.5	0.8	11.2	1.8	12.4
CEur	0.4	36.8	0.4	36.8	2.1	49.9	5.3	53.4

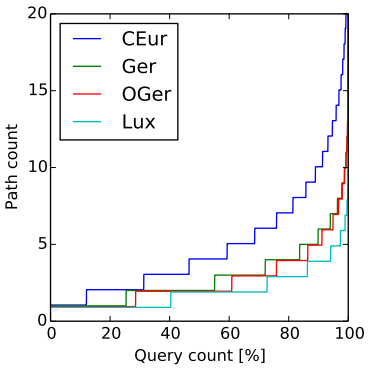

Figure 5 The number of optimal paths (y-axis) in function of number of queries (x-axis) for a 24h-profile of TD-S+P9.

In Table 4, we report profile query running times. In addition to the total running time, we report the amount of time needed to compute the subgraph. Even with TD-S+P9 on the large CEur graph the average running times are only slightly above 50ms for a 24h profile. The query running time of TD-S+P4 is only slightly lower than the running time of TD-S+P9. However, the later has a significantly lower error. TD-S+P9 is therefore superior to TD-S+P4 with respect to profile queries, if the larger memory footprint is not prohibitive.

A path can be optimal for several departure times throughout a day. For an *st*-query, we can count the number of paths that are optimal for at least one departure time. Two paths are the same if they have the same sequence of edges. It is not necessary that the travel times are the same. In Figure 5, we depict the number of optimal paths as function of the percentage of queries with at most that many paths. For most *st*-query there are only very few paths. However, there are outliers for which there can be a significant number of paths. The maximum number of observed paths on CEur is 34.

In theory, it is possible that the low errors we observe in our experiments are an artifact of the test query generation method. It is known that uniform random queries are with near certainty long-distance queries. If only short-distance queries were answered incorrectly, our evaluation method would not detect these errors. In Appendix A, we investigate this question using Dijkstra-rank plots and conclude that this effect luckily does not occure and that the presented results are representative.

6.2 Dynamic Time-Dependent Routing

In the dynamic scenario, we consider two types of congestions: (a) the predicted congestion, and (b) the realtime congestion. The predicted congestions are formalized as edge weight

Table 5 Average query running times.

	Lux	Ger	OGer	CEur
TD-S+D4	0.3	2.3	1.7	4.3
TD-S+D9	0.5	3.6	2.9	7.8

Table 6 Number of exact dynamic, time-dependent queries and absolute and relative errors for the predicted path, TD-S+D4, and TD-S+D9. "Q99" refers to the 99%-quantile and "Q99.9" the 99.9%-quantile.

Graph	Algo	Exact [%]	Relative Error [%]				Absolute Error [s]			
			Avg	Q99	Q99.9	Max	Avg	Q99	Q99.9	Max
Lux	Predict.P	1.6	17.228	56.1	75.0	93.8	323.0	739	826	997
Lux	TD-S+D4	94.7	0.017	0.5	2.3	6.2	0.6	15	93	231
Lux	TD-S+D9	95.0	0.016	0.5	2.2	6.2	0.5	14	89	231
Ger	Predict.P	55.1	1.2	17.9	36.9	79.3	78.5	552	741	1 001
Ger	TD-S+D4	90.9	0.032	1.0	2.6	7.0	3.5	116	233	474
Ger	TD-S+D9	93.4	0.026	0.9	2.5	6.2	2.8	99	216	469
OGer	Predict.P	52.3	1.352	18.7	38.3	65.8	84.9	563	738	934
OGer	TD-S+D4	91.5	0.031	1.0	2.6	5.4	3.2	108	224	462
OGer	TD-S+D9	92.9	0.028	0.9	2.5	5.4	2.9	102	219	462
CEur	Predict.P	72.6	0.392	7.0	25.9	81.9	41.0	443	653	1 870
CEur	TD-S+D4	89.5	0.015	0.5	1.6	5.2	3.3	106	244	547
CEur	TD-S+D9	94.0	0.011	0.3	1.4	5.2	1.9	69	205	397

functions. The predicted congestions used in our setup come from realworld production-grade data. The realtime congestions are randomly generated according to the scheme described in Section 5.1. In Table 6, we compare the errors induced by three approaches: The Predicted Path heuristic (Predict.P) as baseline, TD-S+D4, and TD-S+D9. The Predicted Path heuristic computes a shortest path P with respect to only the predicted congestion. P is then evaluated with respect to both congestion types. In Table 5, we report the query running times of TD-S+D4 and TD-S+D9. Freeflow and Predicted Path are similar in spirit. Freeflow solves the time-dependent routing problem with predicted congestions by ignoring predicted congestions. Similarly, Predicted Path solves the dynamic time-dependent routing problem by ignoring realtime congestion. The Freeflow heuristic produces surprisingly small errors. This contrasts with the predicted path heuristic, whose measured errors in Table 6 are very large. On the Luxembourg instance only 1.6% of the queries are solved optimally. Fortunately, TD-S+D significantly reduces these errors. Over all instances, the minimum number of optimally solved queries is 92.9%. This is a huge improvement compared to 1.6%. The errors induced by TD-S+D in the dynamic scenario are larger than those of TD-S in the static scenario. Fortunately, even the 99%-quantile of TD-S+D9 is well below 1% on all test instances, which is good enough for many applications.

Table 7 Comparison of earliest arrival query algorithms. "n/r" = not reported. We report the running times as published in the corresponding papers (ori) and scaled by processor clock speed.

	Numbers from		Link & Merge?	Relative Error [%]			Run T. [ms]	
				avg.	Q99.9	max.	ori	scaled
TDCALT-K1.00	[10]	OGer	•	0	0	0	5.36	3.77
TDCALT-K1.15	[10]	OGer	•	0.051	n/r	13.84	1.87	1.31
eco SHARC	[8]	OGer	•	0	0	0	25.06	19.7
eco L-SHARC	[8]	OGer	•	0	0	0	6.31	5.0
heu SHARC	[8]	OGer	•	n/r	n/r	0.61	0.69	0.54
heu L-SHARC	[8]	OGer	•	n/r	n/r	0.61	0.38	0.30
TCH	Tab. 3	OGer	•	0	0	0	1.12	
TDCRP (0.1)	[6]	OGer	•	0.05	n/r	0.25	1.92	
TDCRP (1.0)	[6]	OGer	•	0.68	n/r	2.85	1.66	
Freeflow	Tab. 2 & 3	OGer	○	0.140	4.7	12.4	0.12	
Avgflow	Tab. 2 & 3	OGer	○	0.050	2.2	6.5	0.14	
FLAT-SR_{2000}	[22]	OGer	○	1.444	n/r	n/r[3]	1.28	1.18
CFLAT-BC3K+R1K,N=6	[23]	OGer	○	0.031	n/r	19.154	4.10	4.73
TD-S+4	Tab. 2 & 3	OGer	○	0.002	0.4	2.0	0.97	
TD-S+9	Tab. 2 & 3	OGer	○	0.001	0.2	2.0	2.09	

Table 8 Comparison of profile query algorithms. We report the running times as originally reported in the corresponding publications. Further, we report running times scaled by processor clock speed.

	Numbers from	Run T. [ms]	
		ori	scaled
eco SHARC	[8]	60 147	47 388
heu SHARC	[8]	1 075	847
ATCH ϵ=2.5%	[4]	38.57	30
TD-S+P4	Tab. 4	19.5	19.5
TD-S+P9	Tab. 4	22.2	22.2

6.3 Comparison with Related Work

In Tables 7 and 8, we compare TD-S with related work on the OGer instance. Comparing the reported error values is very difficult. Many papers report maximum relative error over uniform 10^5 random queries. Unfortunately, this value heavily depends on the random seed used to generate the test queries. Comparing maximum error values across papers is therefore unfortunately not meaningful unless they differ by orders of magnitude. Average values are significantly less sensitive to this problem. To mitigate this problem, we also report quantiles.

Besides TD-S, Freeflow, Avgflow, only FLAT and CFLAT do not need link and merge operations. As the reported maximum error values are very similar, a detailed error comparison is not meaningful. A limitation of FLAT and CFLAT are is large memory consumption: For OGer 51 GB respectively 16.1 GB are reported [22] and [23]. TD-S4 only needs 1.8 GB. We

[3] It was clarified in [23] and in personal communication that the reported numbers are average values.

doubt that FLAT or CFLAT scales in terms of memory consumption to CEur. Compared with link- and merge-based techniques TD-S is highly competitive. TD-S4's average error is smaller than the average error of every non-exact competitor that reports average errors. TD-S4's query running time is only beat by Freeflow and heu L-SHARC. A major downside of L-SHARC is that it is complex. Not only is linking and merging needed. L-SHARC combines A*/ALT, Arc-Flags, and contraction. None of these components is easy to implement. TCH has, in addition to being complex, a large memory consumption.

Profile queries have not been described and evaluated for all competitors. Only [8] and [4] report experiments. We present an overview in Table 8. ATCH is a TCH variant. We do not expect ATCH to scale to CEur because of memory restrictions. We believe that SHARC would run on CEur but the query running times could significantly increase. On OGer, TD-S+P clearly wins in terms of query running time. Eco SHARC and ATCH, but not heu SHARC, are exact and therefore the comparison with TD-S+P is not completely fair. However, to compute profiles on CEur, ATCH is not an option because of memory constraints. Further, eco SHARC likely has query running times above a minute and is therefore too slow for many applications. There are thus no alternatives to TD-S+P.

7 Conclusion

We introduce TD-S, a simple and efficient solution to the earliest arrival problem with predicted congestions on road graphs. We extend it to TD-S+P which is the only algorithm to solve the profile variant in at most 50ms on all test instances. Further, we demonstratd with TD-S+D that additional realtime congestions can easily be incorporated into TD-S.

Acknowledgments. I thank Julian Dibbelt, Michael Hamann, Stefan Hug, Alexander Kleff, and Frank Schultz for fruitful discussions.

References

1 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. Technical Report abs/1504.05140, ArXiv e-prints, 2016. To appear in LNCS Volume on Algorithm Engineering, Lasse Kliemann and Peter Sanders (eds.).

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*, volume 9220 of *Lecture Notes in Computer Science*, pages 19–80. Springer, 2016.

3 Gernot Veit Batz. Katch open source code. Uploaded to github at `https://github.com/GVeitBatz/KaTCH`, 2016.

4 Gernot Veit Batz, Robert Geisberger, Peter Sanders, and Christian Vetter. Minimum time-dependent travel times with contraction hierarchies. *ACM Journal of Experimental Algorithmics*, 18(1.4):1–43, April 2013.

5 Reinhard Bauer and Daniel Delling. SHARC: Fast and robust unidirectional routing. *ACM Journal of Experimental Algorithmics*, 14(2.4):1–29, August 2009. Special Section on Selected Papers from ALENEX 2008.

6 Moritz Baum, Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. Dynamic time-dependent route planning in road networks with user preferences. In *Proceedings of the 15th International Symposium on Experimental Algorithms (SEA'16)*, volume 9685 of *Lecture Notes in Computer Science*, pages 33–49. Springer, 2016.

7 K. Cooke and E. Halsey. The shortest route through a network with time-dependent internodal transit times. *Journal of Mathematical Analysis and Applications*, 14(3):493–498, 1966.

8 Daniel Delling. Time-dependent SHARC-routing. *Algorithmica*, 60(1):60–94, May 2011.

9 Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable route planning in road networks. *Transportation Science*, 51(2):566–591, 2017.

10 Daniel Delling and Giacomo Nannicini. Core routing on dynamic time-dependent road networks. *Informs Journal on Computing*, 24(2):187–201, 2012.

11 Daniel Delling and Dorothea Wagner. Time-dependent route planning. In *Robust and Online Large-Scale Optimization*, volume 5868 of *Lecture Notes in Computer Science*, pages 207–230. Springer, 2009.

12 Ugur Demiryurek, Farnoush Banaei-Kashani, and Cyrus Shahabi. A case for time-dependent shortest path computation in spatial networks. In *Proceedings of the 18th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS'10)*, pages 474–477, 2010.

13 Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Customizable contraction hierarchies. *ACM Journal of Experimental Algorithmics*, 21(1):1.5:1–1.5:49, April 2016.

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

15 Stuart E. Dreyfus. An appraisal of some shortest-path algorithms. *Operations Research*, 17(3):395–412, 1969.

16 Forschungsgesellschaft für Verkehrswesen. Richtlinien für Lichtsignalanlagen (RiLSA), 2010.

17 Luca Foschini, John Hershberger, and Subhash Suri. On the complexity of time-dependent shortest paths. *Algorithmica*, 68(4):1075–1097, April 2014.

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, August 2012.

19 Andrew V. Goldberg and Chris Harrelson. Computing the shortest path: A* search meets graph theory. In *Proceedings of the 16th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'05)*, pages 156–165. SIAM, 2005.

20 Michael Hamann and Ben Strasser. Graph bisection with pareto-optimization. In *Proceedings of the 18th Meeting on Algorithm Engineering and Experiments (ALENEX'16)*, pages 90–102. SIAM, 2016.

21 Martin Holzer, Frank Schulz, and Dorothea Wagner. Engineering multilevel overlay graphs for shortest-path queries. *ACM Journal of Experimental Algorithmics*, 13(2.5):1–26, December 2008.

22 Spyros Kontogiannis, George Michalopoulos, Georgia Papastavrou, Andreas Paraskevopoulos, Dorothea Wagner, and Christos Zaroliagis. Engineering oracles for time-dependent road networks. In *Proceedings of the 18th Meeting on Algorithm Engineering and Experiments (ALENEX'16)*. SIAM, 2016.

23 Spyros Kontogiannis, Georgia Papastavrou, Andreas Paraskevopoulos, Dorothea Wagner, and Christos Zaroliagis. Improved oracles for time-dependent road networks. Technical report, ArXiv, 2017.

24 Ulrich Lauther. An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background. In *Geoinformation und Mobilität - von der Forschung zur praktischen Anwendung*, volume 22, pages 219–230. IfGI prints, 2004.

25 Giacomo Nannicini, Daniel Delling, Leo Liberti, and Dominik Schultes. Bidirectional A* search on time-dependent road networks. *Networks*, 59:240–251, 2012. Best Paper Award.

26 Ariel Orda and Raphael Rom. Shortest-path and minimum delay algorithms in networks with time-dependent edge-length. *Journal of the ACM*, 37(3):607–625, 1990.

27 PTV AG – Planung Transport Verkehr. http://www.ptv.de, 1979.

28 Frank Schulz, Dorothea Wagner, and Karsten Weihe. Dijkstra's algorithm on-line: An empirical case study from public railroad transport. *ACM Journal of Experimental Algorithmics*, 5(12):1–23, 2000.

29 Ben Strasser. Intriguingly simple and efficient time-dependent routing in road networks. Technical report, ArXiv e-prints, 2016.

30 Ben Strasser. Source code of routingkit. Uploaded to github at `https://github.com/RoutingKit/RoutingKit`, 2016.

31 Ben Strasser. Td-s experimental open source code. Uploaded to github at `https://github.com/ben-strasser/td_p`, 2016.

A Dijkstra Rank Plots

In theory, it is possible that the low errors we observe in our experiments are an artifact of our test query generation method. The employed generation method is the current state-of-the-art and is used in all competitor papers. However, it has a bias and maybe this bias is exploited by TD-S. Picking the source and the target nodes uniformly at random nearly always yields a long-distance query. This means that, on the Europe instance, it is very unlikely that a test query is generated, such that the resulting path has a running time of for example 30min, which is short compared to the diameter of the instance. In theory, it is possible that only short-distance queries have a significant error but we do not generate such test queries. We demonstrate in this section that this effect does not occur.

That this effect does not occur can be seen from our experiments in the main part. All queries generated on the Luxembourg instance are local compared to the diameter of Europe or Germany. If short-distance queries were harder to solve than long-distance queries, the achieved errors on the Luxembourg instance should be higher than those achieved on the Europe or Germany instance. Fortunately, this is not the case and we can therefore conclude that short-distance queries are not inherently harder than long-distance queries. It is possible though that Luxembourg is in some sense special and short-distance queries are only easy on the Luxembourg instance. To show that this also does not occur, we investigate short-distance queries in a more systematic setup in this section.

We compute Dijkstra-rank plots in Figure 6. Dijkstra-rank queries are generated as follows: We pick a random source node s uniformly at random. In the next step, we order all other nodes by freeflow distance from s using Dijkstra's algorithm. The query from s to the 2^i-th node in this order has Dijkstra-rank i. We generated 1 000 random source nodes s for our experiments. The intuition is that a query with a low Dijkstra-rank is more local than a query with a high Dijkstra-rank.

We plot the number of queries that were answered optimally by TD-S+4. We further plot the relative and absolute errors of TD-S+4. It is usual to use a box plot to visualize Dijkstra-rank results. However, because of the low errors of TD-S the boxes of our plots are degenerate and squashed onto the x-axis. As most errors are zero, zooming into the plots does not help. All observable dots are outliers. To emphasize the low achieved errors and because the squashed boxes do not cover the outliers, we decided to stick with a degenerate box plot representation.

One can observe that on every instance the number of correct queries shrinks with growing Dijkstra-rank. Similarly, the absolute error grows with a growing a Dijkstra-rank. This is

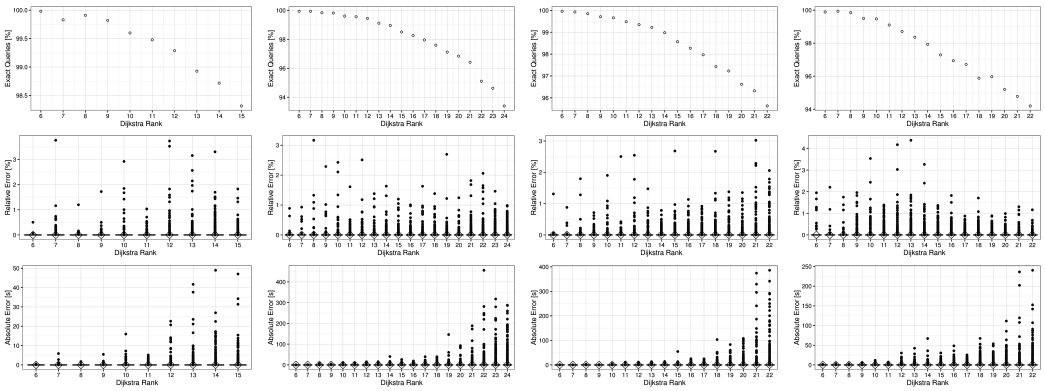

Figure 6 Percentage of correct queries (top), relative error (mid), and absolute error (bottom). Columns correspond to instances: Leftmost ist Lux, left middle is CEur, right middle ist Ger, and rightmost ist OGer.

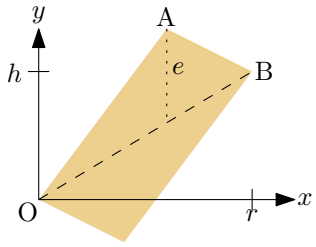

Figure 7 Proof Illustration. The optimal function must be in the orange area. e is the maximum error of our estimation.

non-surprising as the path length also grows with Dijkstra-ranks. The relative error seems to be independent of the Dijkstra-rank.

As the number of optimally solved queries is minimum with a large Dijkstra-rank, we conclude that TD-S is not exploiting the test query generation and our experimental results from the main part of this paper are representative.

B Profile Error Guarantee

The proof is sketched in Figure 1. The solid line is the unknown optimal function. The dashed function is the computed approximative function. As the slopes are bounded, we know that the unknown optimal function is inside of the orange area. The maximum difference between the approximative function and the boundary of the orange area therefore bounds the error induced by the profile algorithm.

Consider the situation depicted in Figure 7. Our objective is to compute the maximum vertical distance from the dashed line inside of the orange region. As the triangle below the dashed line is the same as the triangle above it rotated by 180°, we can focus solely on the upper triangle OAB. As the distance grows from O to A and decreases from A to B we know that the distance is maximum at $x = A_x$, i.e., the maximum vertical distance is the length e of the dotted segment. From the application we know that $B_x = r$, and that the slope of the line OA is $\Lambda_{\max}$, and that the slope of AB is $-\Lambda_{\min}$. We denote by h the y-position of B. Our objective is to compute the maximum value of e over all values of h. We start by computing e and then maximize the resulting expression over h.

The line OA is described by $y = \Lambda_{\max} x$, and the line OB is described by $y = \frac{h}{r}x$, and the line AB is described by $y = -\Lambda_{\min} x + (r\Lambda_{\min} + h)$. By intersecting OA and AB we get $\Lambda_{\max} A_x = -\Lambda_{\min} A_x + (r\Lambda_{\min} + h)$ which can be solved for A_x yielding $A_x = \frac{r\Lambda_{\min}+h}{\Lambda_{\max}+\Lambda_{\min}}$ which leads to

$$\begin{aligned} e &= \Lambda_{\max} A_x - \frac{h}{r} A_x \\ &= (\Lambda_{\max} - \frac{h}{r}) A_x \\ &= \frac{(\Lambda_{\max} - \frac{h}{r})(r\Lambda_{\min} + h)}{\Lambda_{\max} + \Lambda_{\min}} \end{aligned}$$

which is an expression for the desired vertical height. As $0 < \Lambda_{\max}$ and $0 < \Lambda_{\min}$ the value of e is maximum if and only if

$$(\Lambda_{\max} - \frac{h}{r})(r\Lambda_{\min} + h) = -\frac{h^2}{r} + (\Lambda_{\max} - \Lambda_{\min})h + r\Lambda_{\min}\Lambda_{\max}$$

is maximum. As $-h^2 < 0$ this parabola is maximum when its derivative is zero. We therefore compute the derivative $-\frac{2h}{r} + (\Lambda_{\max} - \Lambda_{\min})$ which is zero for $h = \frac{r(\Lambda_{\max}-\Lambda_{\min})}{2}$ which we can insert into the expression of e to obtain

$$\begin{aligned} \max_h e &= \frac{(\Lambda_{\max} - \frac{r(\Lambda_{\max}-\Lambda_{\min})}{2r})(r\Lambda_{\min} + \frac{r(\Lambda_{\max}-\Lambda_{\min})}{2})}{\Lambda_{\max} + \Lambda_{\min}} \\ &= \frac{(\frac{\Lambda_{\max}+\Lambda_{\min}}{2})(\frac{r(\Lambda_{\max}+\Lambda_{\min})}{2})}{\Lambda_{\max} + \Lambda_{\min}} \\ &= \frac{r(\Lambda_{\max} + \Lambda_{\min})}{4} \end{aligned}$$

which was the absolute error bound we needed to compute.