Using Incremental Many-to-One Queries to Build a Fast and Tight Heuristic for A* in Road Networks

BEN STRASSER and TIM ZEITZ, Karlsruhe Institute of Technology

We study exact, efficient, and practical algorithms for route planning applications in large road networks. On the one hand, such algorithms should be able to answer shortest path queries within milliseconds. On the other hand, routing applications often require integrating the current traffic situation, planning ahead with predictions for future traffic, respecting forbidden turns, and many other features depending on the specific application. Therefore, such algorithms must be flexible and able to support a variety of problem variants. In this work, we revisit the A* algorithm to build a simple, extensible, and unified algorithmic framework applicable to many route planning problems. A* has been previously used for routing in road networks. However, its performance was not competitive because no sufficiently fast and tight distance estimation function was available. We present a novel, efficient, and accurate A* heuristic using Contraction Hierarchies, another popular speedup technique. The core of our heuristic is a new Contraction Hierarchies query algorithm called *Lazy RPHAST*, which can efficiently compute shortest distances from many incrementally provided sources toward a common target. Additionally, we describe A* optimizations to accelerate the processing of low-degree vertices, which are typical in road networks, and present a new pruning criterion for symmetrical bidirectional A*. An extensive experimental study confirms the practicality of our approach for many applications.

CCS Concepts: • **Theory of computation** → **Shortest paths** • **Mathematics of computing** → *Graph algorithms* • **Applied computing** → *Transportation*;

Additional Key Words and Phrases: Route planning, shortest paths, realistic road networks

ACM Reference format:
Ben Strasser and Tim Zeitz. 2023. Using Incremental Many-to-One Queries to Build a Fast and Tight Heuristic for A* in Road Networks. *ACM J. Exp. Algor.* 27, 4, Article 4.6 (February 2023), 28 pages.
https://doi.org/10.1145/3571282

1 INTRODUCTION

The past two decades have seen a plethora of research works on route planning in large road networks [4]. Routing a user through a road network can be formalized as the shortest path problem in weighted graphs. Vertices represent intersections. Roads are modeled using edges. Edges are weighted by their traversal times. The problem can be solved with Dijkstra's algorithm [31]. Unfortunately, on continental-sized networks, it is too slow for many applications. Therefore, speedup techniques have been developed. These techniques employ an offline *preprocessing* phase where auxiliary data is precomputed. This auxiliary data is then utilized to accelerate shortest path

This article is the extended version of our conference paper "A Fast and Tight Heuristic for A* in Road Networks" [54].
Authors' address: B. Strasser and T. Zeitz, Am Fasanengarten 5, Geb. 50.34, Raum 321, 76131 Karlsruhe; emails: academia@ben-strasser.net, tim.zeitz@kit.edu.

1084-6654/2022/02-ART4.6
https://doi.org/10.1145/3571282

computations in the online *query* phase. With this approach, speedups of more than three orders of magnitude have been achieved. This allows interactive query times of milliseconds or less on continental-sized road networks.

However, for many real-world applications, this basic model is too simplistic. For realistic routing, many additional features need to be considered. This includes turn costs and restrictions, live traffic, user preferences, and traffic predictions. Some applications may have additional application-specific requirements. Further, it is insufficient to handle each of these features independently. Instead, all requirements must be supported in combination.

Extending Dijkstra's algorithm to support these features is comparatively easy. In contrast, extending speedup techniques is vastly more complex. The results on adopting specific speedup techniques to specific extended problem fill many research papers [8, 18, 22, 30, 37, 53] and sometimes entire dissertations [5, 11, 20]. We will present a brief overview in Section 1.1. Techniques achieving fast query times have been successfully developed for many extended scenarios; however, we observe two problems: the resulting techniques are complex to implement and hard to extend further. For example, although there exist speedup techniques achieving fast queries for each of the features mentioned earlier, we are unaware of any work supporting the combination of these features. We believe a different trade-off between running time and extensibility is necessary for practical applications. Therefore, we prioritize a simple and extensible approach over the fastest possible query times in this work.

1.1 Related Work

A considerable amount of research effort has been put into accelerating shortest path computations in transportation networks. For a comprehensive overview, we refer to the work of Bast et al. [4]. Here, we first highlight a few key results for the classical shortest path problem on road networks that are specifically relevant to our work. In the second part of this section, we will discuss adaptations of these techniques to extended problem models.

Speedup Techniques for Shortest Paths in Road Networks. A* [43] is a well-known approach to accelerate shortest path queries by directing the search toward the target through heuristic distance estimates. It has been successfully employed in many problems beyond route planning in transportation networks [15, 19, 52]. It is one of the algorithms fundamental to our work. The performance of A* depends on the quality of the heuristic estimates. For example, although geographic distances may seem like a natural choice for distance estimates in road networks, the resulting heuristic is relatively ineffective: it performs worse than Dijkstra's algorithm [38]. A much more accurate heuristic can be obtained with ALT [38, 40]. ALT was one of the early speedup techniques for routing in road networks. It uses precomputed distances to specific landmark vertices combined with the triangle inequality to compute distance estimates. On road networks, this achieves speedups of around two orders of magnitude over Dijkstra's algorithm. However, the speedups obtained by ALT are still limited by the quality of the distance estimates. To achieve faster queries, Core-ALT has been developed [10]. Here, parts of the network are *contracted* during the preprocessing, leaving a much smaller remaining core graph to run the goal-directed search. Core-ALT yields speedups of about three orders of magnitude compared to Dijkstra's algorithm. Since then, there has been little development on A*-based techniques for routing in road networks. That is because hierarchical techniques have proven more effective.

Hierarchical speedup techniques utilize the inherent hierarchy in road networks, usually in combination with the *contraction* of less important vertices. One popular example is **Contraction Hierarchies (CH)** [36]. In a preprocessing step, additional shortcut edges are inserted, which allow skipping unimportant parts of the network at query time. The "importance" of vertices is

determined heuristically during preprocessing. CH queries take less than a tenth of a millisecond, which is a speedup of more than four orders of magnitude over Dijkstra's algorithm. The preprocessing takes only a few minutes and produces little more auxiliary data than the input graph. CH has been used successfully in many real-world applications, and quite a few open source implementations exist.[1] CH is another fundamental algorithm for our work. More specifically, we extend on PHAST [21] and RPHAST [23]. These algorithms reuse the CH preprocessing but modify the query phase. PHAST supports *one-to-all* queries from one source vertex to *all* other vertices. RPHAST is a variant for *one-to-many* queries to a subset of vertices known in advance. Another popular hierarchical speedup technique also used in practice [48] is **Multilevel Dijkstra (MLD)** [51]. MLD is based on a multi-level partition of the road network computed during preprocessing. Queries are slightly slower than CH queries but still three to four orders of magnitude faster than Dijkstra's algorithm. The memory consumption is even smaller than with CH. Hierarchical speedup techniques have been combined with goal-directed search and A* in other works [10, 12, 39]. However, those works primarily focused on obtaining faster queries rather than simple and flexible approaches.

The speedup technique known to achieve the fastest queries is **Hub Labels (HL)** [2, 24]. With HL, query times in less than a microsecond are possible. This is only a few times as much as an uncached memory access. This speed comes at the cost of an expensive preprocessing phase and a tremendous memory footprint, more than an order of magnitude larger than the input graph. Despite the extremely fast query times, it appears that the trade-off offered by HL is, in practice, often not very attractive. We are not aware of any works adapting HL to extended problem models. Additionally, simpler techniques like CH already offer queries fast enough to completely disappear behind other parts of practical applications (e.g., the network latency).

Speedup Technique Adaptions for Extended Problems. One of the earliest approaches adapting speedup techniques to extended problems was built on A*. In the work of Delling and Wagner [27], ALT is used for routing on dynamic and time-dependent graphs. The approach is conceptually similar to ours. However, because the heuristic is ALT based, the accuracy of the estimates is limited, and the resulting query performance is not competitive. Further, the evaluation was performed with only synthetic traffic data. With production-grade real-world traffic data, the problem becomes significantly harder [53]. Similar to the development for the classical problem, the approach was combined with contraction [25, 49]. Even though this improves query running times, the obtained speedups are not competitive with purely hierarchical techniques.

Customizable Route Planning (CRP) [22] is an engineered variant of MLD [51] that was developed to allow updating weights without invalidating the entire preprocessing. For this, a second, faster preprocessing phase is introduced, which takes at most a few seconds. This phase is called the *customization*. It can be run regularly to update weights. This enables the integration of live traffic and user preferences. CRP was designed to support turn costs without any additional modifications. Queries in CRP take at most a few milliseconds. CRP is one of the few examples of a speedup technique designed for flexibility rather than maximum query performance. However, its flexibility also has limits. For example, integrating traffic predictions into CRP was studied in the work of Baum et al. [14]. However, achieving reasonable memory consumption and query times was only possible by giving up exactness. Further, TD-CRP can only compute approximate shortest distances rather than paths.

There are many works extending CH to more complex problem models. In the work of Geisberger and Vetter [37], turn information is integrated into CH. The proposed approach achieves

[1] https://gist.github.com/PayasR/bc46af938195a827e42006c3f5544e4a. Although this list is certainly not exhaustive and possibly not representative of what algorithms are used in practice, it still paints a relatively clear picture of which techniques have reached popularity.

fast queries, but preprocessing becomes an order of magnitude slower than classical CH. In the work of Dibbelt et al. [30], CH is extended to **Customizable Contraction Hierarchies (CCH)**. CCH also has a second preprocessing (customization) phase where weights can be altered. This allows supporting user preferences and live traffic with queries an order of magnitude faster than CRP. Supporting turn costs in CCH was studied by Bucchold et al. [18]. However, even with all improvements proposed in their work [18], turn integration still causes a slowdown of at least factor three to the customization phase. A considerable amount of research and engineering effort has been put into studying the combination of traffic predictions with CH. Several papers [6, 7, 8, 44] and an entire dissertation [5] have been published on the subject. Different variants with trade-offs regarding exactness, query speed, and space consumption were proposed [8]. Recently, a new approach has been published [53], which simultaneously achieves competitive results in all three aspects but only at the cost of considerable implementation complexity.[2]

Finally, note that there are many other extended problem models. Although they are beyond the scope of this work, they highlight the need for extensible speedup techniques. Examples for this are electric vehicle routing [13, 33], multi-criteria optimization [34, 35], computing alternative routes [1, 3], routing with incomplete and noisy traffic data [26], and routing for trucks [45].

1.2 Contribution and Outline

Clearly, there is an enormous amount of efficient speedup techniques for routing in road networks. Further, for many extended problems, there are dedicated research results on extensions of specific techniques. Although many problems can be solved efficiently, we still believe this situation is unsatisfactory. For practical applications, a unified and extensible approach with manageable implementation complexity is often more important than the fastest query performance. Therefore, we revisit the A* algorithm and propose a flexible, unified framework for many routing problems. It can be applied to any extended routing problem where tight lower bounds are available at preprocessing time. The core of our approach is a new CH-based A* heuristic. It allows for much tighter estimates than previous A* heuristics and thus significantly faster queries. The query running times of our technique are not competitive with approaches tailored to specific problems (i.e., typically an order of magnitude slower); however, they are often sufficient for practical applications. Further, our approach only requires the classical CH preprocessing regardless of the specific extended routing problem. Therefore, preprocessing time and memory consumption are typically significantly better than approaches aiming for competitive query times in specific problems.

The remainder of this work is organized as follows. We discuss notation and fundamental algorithms in Section 2. In Section 3, we introduce Lazy RPHAST, a simple and efficient CH-based algorithm for the incremental many-to-one shortest path problem. It is the first algorithm to efficiently support accelerated shortest path computations from *dynamic* source sets to a fixed target. Section 4 contains several optimizations for A* in road networks accelerating the processing of low-degree vertices (Section 4.1) and an improved variant of bidirectional A* (Section 4.2). Our main contribution, the CH-Potentials framework, is presented in Section 5. We first show how to use Lazy RPHAST to build a fast and tight heuristic for A* in road networks. Based on this heuristic, we provide a unified framework (Sections 5.1 and 5.2) for a variety of practical route planning problems (Section 5.3). CH-Potentials can be applied to all of these problems without any modifications to the preprocessing. In Section 6, we provide an extensive experimental evaluation analyzing the performance characteristics of our algorithms. It shows that CH-Potentials achieves decent running times when tight lower bounds are available at preprocessing time. However, the

[2] In https://github.com/kit-algo/rust_road_router, the preprocessing alone has more than 10,000 lines of code. In contrast, the classical CH can typically be realized within a few hundred lines of code.

strength of our approach is not in query running times but its flexibility: problem extensions can be supported without adjustments to the preprocessing. The price for each extension is a query slowdown depending on how tight the lower bounds used during preprocessing remain.

2 PRELIMINARIES

We consider directed graphs $G = (V, E)$ with $E \subseteq V \times V$ with $n = |V|$ vertices and $m = |E|$ edges with weight functions $w : E \to \mathbb{R}^{\geq 0}$. We use uv as a short notation for the edge (u, v). The *reversed* graph $\overleftarrow{G} := (V, \{vu \mid uv \in E\})$ contains all edges in the reverse direction. The corresponding reversed weight function is $\overleftarrow{w}(vu) := w(uv)$. We denote by $N(u) = \{v \mid uv \in E \vee vu \in E\}$ the *undirected neighborhood* of a vertex u and refer to the numbers of distinct neighbors $|N(u)|$ as the *degree* of u.

Given vertices s and t, we want to obtain a st-path $P = (s = v_0, \ldots, t = v_k)$ of minimum weight $w(P) := \sum_i w(v_{i-1}v_i)$. We denote this shortest path weight as the distance $\text{dist}_w(s, t)$ between s and t. If there is no path, we set $\text{dist}_w(s, t) = \infty$.

2.1 Dijkstra's Algorithm

In this section, we recall the central aspects of Dijkstra's classical shortest path algorithm [31] and introduce the notation used throughout this article. Dijkstra's algorithm maintains an array of tentative distances D and a priority queue Q of vertices ordered by increasing distance from s. We denote by k the minimum key in Q (i.e., the tentative distance of the closest remaining vertex). Initially, all distances are set to ∞. The start vertex distance $D[s]$ is set to zero, and the queue is initialized with s. In each iteration, the closest remaining vertex u is popped from the queue and *settled*. Outgoing edges uv are relaxed—that is, $D[v]$ is set to $\min(D[v], D[u] + w(uv))$. If the tentative distance at the head vertex $D[v]$ is improved, it is added to the queue or its position in the queue is adjusted. Once t is settled, $D[t]$ contains the shortest path distance between s and t. The search can stop when t is settled. The vertices visited by Dijkstra's algorithm throughout the search are denoted as the *search space*.

Dijkstra's algorithm can also be run from the target t on the reversed graph $\overleftarrow{G}$. We call this a *backward search*. Running two Dijkstra searches simultaneously, one from s and a backward search from t, until the searches meet is called *bidirectional* search. In this case, we denote by $\overrightarrow{D}$, $\overrightarrow{Q}$, and $\overrightarrow{k}$ the distances, queue, and minimum queue key of the forward search and by $\overleftarrow{D}$, $\overleftarrow{Q}$, and $\overleftarrow{k}$ the respective data of the backward search. Typically, the searches are interleaved by alternating settling a vertex from each direction. Another common approach is to settle a vertex from the direction with the smaller minimum queue key. Theoretically, any direction interleaving strategy is possible. When the sum of the minimum keys in both queues $\overrightarrow{k} + \overleftarrow{k}$ is greater than the best-known tentative total distance μ, the algorithm can terminate.

2.2 A* Algorithm

A* is a goal-directed variant of Dijkstra's algorithm. It uses a heuristic function $h_t(v)$ that maps a vertex v onto an estimate of the distance from v to the target t. A* orders the vertices in the priority queue by $D[v] + h_t(v)$ instead of $D[v]$ as Dijkstra's algorithm does it. Thus, vertices closer to the target are explored earlier. Figure 1 depicts an example of the search space of the A* algorithm. With Dijkstra's algorithm, all vertices closer to s than the target would have been explored. Thus, the search space would be something resembling a circle around s. With A*, the search is directed toward the target and fewer vertices are explored. Dijkstra's algorithm is a special case of A* with $h_t(v) = 0$.

Fig. 1. Vertices explored by A*. The color indicates the vertex removal order from the queue. Blue was removed first. Next is green. Red was removed last. Background by OpenStreetMap.

A* is equivalent to running Dijkstra's algorithm on the graph with *reduced weights* [43]. This reduced weight function is defined as $w_{h_t}(uv) := w(uv) - h_t(u) + h_t(v)$. A heuristic function is called *feasible*, if $w_{h_t}(e) \geq 0$ for all edges e. If the employed heuristic is feasible, A* is guaranteed to have found the optimal distance after settling t. In this article, we limit our discussion to feasible heuristics.

2.3 Contraction Hierarchies

CH is a two-phase speedup technique to accelerate shortest path computations on road networks through precomputation. For a detailed discussion, we refer to the work of Geisberger et al. [36]. In a preprocessing phase, vertices are ordered totally by "importance" where more important vertices should lie on more shortest paths. Intuitively, vertices on highways are more important than vertices on rural streets. For CH, such an ordering is obtained heuristically. The position of a vertex in the order is also denoted as its *rank*. Vertices of higher rank are informally referred to and visualized as "higher up" in the hierarchy. Therefore, an edge where the tail has a lower rank than the head is an *upward* edge. Analogously, when the head vertex has the lower rank, the edge is said to go *downward*. Once such an importance order was obtained, all vertices are contracted successively by ascending importance. To *contract* a vertex means temporarily removing it from the graph while inserting *shortcut edges* between more important neighbors to preserve shortest distances among them. The result is an *augmented graph* G^+ of original edges and shortcuts with weights w^+. We often refer to the augmented graph split into an upward graph $G^\uparrow = (V, E^\uparrow)$ containing only upward edges and a downward graph $G^\downarrow = (V, E^\downarrow)$ containing only downward edges.

The augmented graph has the property that for any two vertices s and t, there always exists an *up-down st*-path—that is, a path that first uses only edges from $E^\uparrow$ and then only edges from $E^\downarrow$, with the same length as a shortest path in G. Figure 2 presents an illustration. From every shortest path (red) in G, an up-down path of equal length in G^+ (blue) can be constructed. Such an up-down path can be found by running Dijkstra's algorithm from s on $G^\uparrow$ and t on the reversed downward graph $\overleftarrow{G^\downarrow}$. By construction, at least the highest-ranked vertex on the up-down path must be in the intersection of the forward and backward search spaces. Pseudocode for the backward search is

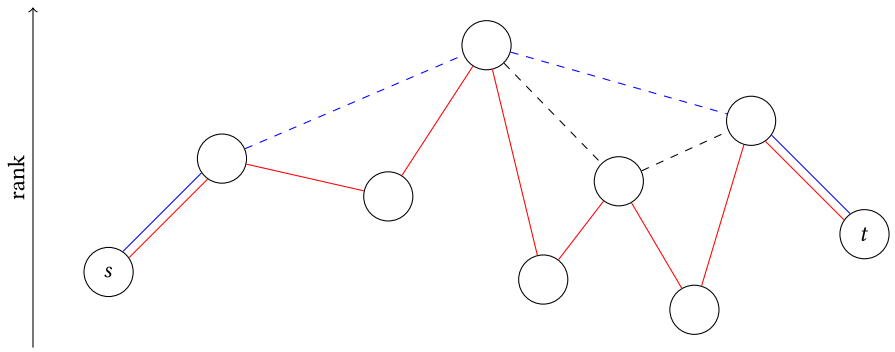

Fig. 2. Solid lines are edges in G. Dotted lines are shortcuts. Red is a shortest st-path in G. Blue is an equally long up-down path in G^+.

ALGORITHM 1: CH backward search.

Data: $D^{\downarrow}[v]$: tentative distance from any vertex $v \in V$ to target t in $G^{\downarrow}$
Data: Minimum priority queue Q, ordered by tentative distances
$D^{\downarrow}[v] \leftarrow +\infty$ for all $v \neq t$;
$D^{\downarrow}[t] \leftarrow 0$;
Make Q only contain t with tentative distance 0;
while Q *not empty* **do**
  $u \leftarrow$ pop minimum element from Q;
  for *all reversed downward edges* $uv \in \overleftarrow{E^{\downarrow}}$ **do**
    if $D^{\downarrow}[u] + \overleftarrow{w}^{+}(uv) < D^{\downarrow}[v]$ **then**
      $D^{\downarrow}[v] \leftarrow D^{\downarrow}[u] + \overleftarrow{w}^{+}(uv)$;
      Add v or decrease v's key in Q to $D^{\downarrow}[v]$;

presented in Algorithm 1. The set of vertices reachable in $G^{\uparrow}$ and $\overleftarrow{G^{\downarrow}}$ is called the *CH search space* of a vertex. The CH query performance strongly correlates with the size of this CH search space and the number of edges between the vertices in the search space. Luckily, on road networks, the CH search space is small [22, 36].

2.4 Customizable Contraction Hierarchies

CCH [30] is a variant of CH extended to a three-phase setup. The phases consist of a slow *preprocessing* phase, a faster *customization* phase, and the accelerated *query* phase. The preprocessing phase typically takes hours, the customization seconds, and the query fractions of a millisecond. The preprocessing phase is independent of the weights and only uses the graph topology. Weights are introduced in the customization phase. Shortest paths are computed in the query phase.

The key idea for CCH is to use a vertex importance order that is independent of the weight function. CCH use *nested dissection* orders for this. A small balanced vertex separator is computed, and the separator vertices get the highest importance. The remaining vertices are ordered by recursing on the independent cells obtained by removing the separator vertices. This results in orders that can be used as CH orders. Regardless of the weight function, any shortest path between cells has to use some of the separator vertices. Therefore, these vertices will always be "important." Such orders can be obtained with partitioning algorithms tailored to road networks [41, 42] in less than an hour, even for continental-sized road networks. After obtaining the order, the augmented

ALGORITHM 2: PHAST basic all-to-one search.

```
Data: D[v]: tentative distance from any v ∈ V to t
Execute Algorithm 1, filling D;
for all CH levels L from most to least important do
    for all up-edges uv ∈ E↑ with u in L do
        if D[u] > D[v] + w+(uv) then
            D[u] ← D[v] + w+(uv);
```

graph G^+ is computed. The standard CH preprocessing algorithms could be used for this, but specialized, significantly faster CCH variants exist. However, discussing them in detail is beyond the scope of this work. See the work of Bucchold et al. [16] and Dibbelt et al. [30] for details. A CCH augmented graph fulfills all necessary properties for CH query algorithms. The CH query algorithms can therefore be applied without modifications. However, the CCH preprocessing grants some useful additional properties.

During preprocessing, an *elimination tree* can be constructed. The root of the elimination tree is the most important vertex. For every other vertex, its parent is its least important upward neighbor. In the work of Bauer et al. [9], it was proven that the set of ancestors of a vertex in the elimination tree is equal to its CH search space. This makes it possible to explore the CCH search space with a more efficient shortest path algorithm: for the forward search, start at the source and follow the elimination tree upward until the root is reached. All outgoing edges of encountered vertices are relaxed. The backward search works analogously but starts at the target. Because this elimination tree-based algorithm does not require a queue, it is faster than a Dijkstra-based CCH query. With this algorithm, CCH can consistently answer shortest path queries on continental-sized road networks in fractions of a millisecond. CH queries are still marginally faster for weight functions with a strong hierarchy, such as travel times. However, on weight functions with a less pronounced hierarchy, CH performance degrades significantly, and CCH queries turn out to be the faster variant. Note that only nested dissection orderings admit an elimination tree of low height. Thus, the elimination tree query cannot be used with classical CH.

2.5 PHAST

PHAST [21] is a CH extension that computes distances from all vertices to one target vertex (or vice versa, the reverse case works analogously). This is sometimes denoted as a *all-to-one* (or *one-to-all*) problem. The preprocessing phase remains the same as for CH. The query phase is split into two steps. The first step is analogue to the CH query: from t, all reachable vertices via reversed down-edges are explored as shown in Algorithm 1. For the second step, PHAST utilizes an assignment of vertices into *levels*. These levels correspond to the importance ordering but allow multiple vertices in the same level. However, no edge must connect two vertices within the same level. Such a level assignment can be obtained by assigning all vertices without lower-ranked neighbors to the lowest level. All other vertices are then iteratively assigned to the next level above the highest level of any downward neighbor.

The main work of the second step consists of iterating over all CH levels from top to bottom. In each iteration, all up-edges starting within the current level are relaxed in reverse. Once all levels are processed, the shortest distances from all vertices to t are computed. Pseudocode is provided in Algorithm 2. PHAST is faster than Dijkstra's algorithm on road graphs because it is a better fit for modern processor architectures, better at utilizing data locality, and, most importantly, can be parallelized very effectively. See the work of Delling et al. [21] for an in-depth experimental performance analysis.

ALGORITHM 3: Lazy RPHAST algorithm.

Data: $D^{\downarrow}[v]$: tentative distance from any vertex $v \in V$ to t as computed by Algorithm 1
Data: $D[v]$: memoized distance from vertex $v \in V$ to t, shared between invocations, initialized to $\perp$ during the selection
Execute Algorithm 1, filling $D^{\downarrow}$;
$D[v] \leftarrow \perp$ for all $v \in V$;
Function `ComputeAndMemoizeDist(`u`)`:
  if $D[u] = \perp$ **then**
    $D[u] \leftarrow D^{\downarrow}[u]$;
    for *all up-edges* $uv \in E^{\uparrow}$ **do**
      $D[u] \leftarrow \min(D[u], w^{+}(uv) +$ `ComputeAndMemoizeDist(`v`))`;
  return $D[u]$;

2.6 RPHAST

RPHAST [23], short for Restricted PHAST, is a PHAST extension for efficiently computing distances from a smaller set of source vertices to one target vertex (again, the reverse case works analogously), solving the *many-to-one* problem. Given a set of source vertices S, the first step is to copy the combined search space of all sources into a *restricted subgraph*. Let V_{restr} be the set of vertices reachable in $G^{\uparrow}$ from any $s_i \in S$. The restricted subgraph $G^{\uparrow}_{\text{restr}}$ is a subgraph of $G^{\uparrow}$ induced by V_{rest}. Finding and copying the relevant edges into this restricted subgraph is called the *selection* step. In the query step, a target t is given, and the PHAST algorithm is applied, but the downward sweep (the second step of the PHAST algorithm) is performed only on the restricted subgraph. RPHAST is particularly effective when many targets are queried for the same source set S. If the source set changes often, selection times may become problematic.

3 THE INCREMENTAL MANY-TO-ONE PROBLEM

This section discusses a variant of the many-to-one shortest path problem, which naturally arises from A* heuristics. Here, the target vertex t is known in the selection step, but the source vertices $s_1, \ldots, s_k$ are queried one after another. We denote this problem as the *incremental many-to-one* problem. The first step is the *target selection* where the target vertex t is provided. Then, an arbitrary number of source vertices are given one after another. For each source s_i, the distance $\text{dist}(s_i, t)$ has to be computed before the next source s_{i+1} is provided.

We consider the combined running time of the target selection and each incremental query as the total running time to answer an incremental many-to-one query. Since the target selection time is included, computing the distances to all vertices with Dijkstra or PHAST is too slow. Additionally, since the source set is provided incrementally, RPHAST in its basic form is not well suited to our problem. Fortunately, we can do better.

3.1 Lazy RPHAST on CH

The core idea of our algorithm is to do the RPHAST computation lazily using memoization. In the target selection, we first run the backward CH search from t to obtain an array $D^{\downarrow}$. $D^{\downarrow}[v]$ is the minimum down vt-path distance or $+\infty$, if there is no such path. $D^{\downarrow}$ is computed as shown in Algorithm 1. Further, the distances $D[v]$ are initialized to a sentinel value $\perp$, where $\perp$ is some particular value distinct from any other valid distance (including ∞). This value indicates that the distance from v to t has not yet been computed.

Now, distances from many sources s_i to t can be computed incrementally with the ComputeAndMemoizeDist function as shown in Algorithm 3. The key to doing this efficiently is reusing the

ALGORITHM 4: Elimination tree-based Lazy RPHAST algorithm.

Data: $D^{\downarrow}[v]$: tentative distance from any vertex $v \in V$ to t as computed by Algorithm 1
Data: $D[v]$: memoized distance from any vertex $v \in V$ to t, shared between invocations, initialized to $\perp$ during the selection
Data: $P[v]$: parent of a vertex $v \in V$ in the elimination tree, a parent of $\perp$ indicates the root vertex
Data: S: stack with vertices to compute distances, empty initially, only used to store intermediate data
Execute Algorithm 1, filling $D^{\downarrow}$;
$D[v] \leftarrow \perp$ for all $v \in V$;
Function `ComputeAndMemoizeDist(`u`)`**:**
    `// Determine the vertices` v `for which` $D[v]$ `needs to be computed`
    $v \leftarrow u$;
    while $D[v] = \perp$ **do**
        Push v onto S;
        if $P[v] = \perp$ **then**
            break;
        $v \leftarrow P[v]$;
    `// Compute` D `for those vertices`
    while S *not empty* **do**
        $v \leftarrow$ pop top element from S;
        $D[v] \leftarrow D^{\downarrow}[v]$;
        for *all up-edges* $vx \in E^{\uparrow}$ **do**
            $D[v] \leftarrow \min(D[v], w^{+}(vx) + D[x])$;
    return $D[u]$;

distance information D across invocations through memoization. Thus, the first step of the algorithm is always to check if the distance for the requested vertex has already been computed. If this is the case, it immediately returns this distance. If not, the distance $D[s_i]$ is initialized to the shortest down-path distance $D^{\downarrow}[s_i]$ obtained by the backward search. Then, the algorithm iterates over all up-edges $(s_i v)$ and checks if the up-down path through this neighbor can improve the distance. The algorithm is invoked recursively to obtain the shortest distance from a neighbor v to t.

Correctness. Due to Geisberger et al. [36], an up-down path of shortest distance must exist in G^{+}. Further, G^{+} can be decomposed into two directed acyclic graphs $G^{\uparrow}$ and $G^{\downarrow}$ and the up path can be found in $G^{\uparrow}$ and the down path in $G^{\downarrow}$. The Lazy RPHAST selection finds the shortest down path in $G^{\downarrow}$. Now observe that the ComputeAndMemoizeDist function is, in fact, a recursive **Depth-First Search (DFS)** on $G^{\uparrow}$. Edges uv are relaxed once all upward neighbors of u have been processed (i.e., in DFS post order). A DFS post order also is a reverse topological order for $G^{\uparrow}$. Therefore, the algorithm relaxes edges uv in reverse topological order of the tail vertices u. Since $G^{\uparrow}$ is a directed acyclic graph, this yields shortest up distances in $G^{\uparrow}$. Concatenated with the down paths obtained by the backward search, this yields optimal shortest distances.

3.2 Lazy RPHAST on CCH

Algorithm 3 can be applied to CCH without any modifications. However, we can also utilize the elimination tree in the Lazy RPHAST algorithm. To compute the distance $D[v]$ of a vertex v, the distances of all upward neighbors must be final. In Algorithm 3, these upward neighbor distances are computed recursively. Thus, the search space is explored in a DFS-like fashion and distances are finalized in DFS post order. However, the path from a vertex to the root in the elimination

tree also is a topological order for the upward search space. This is because the ancestors in the elimination tree contain the entire upward search space [9]. Therefore, iterating over the vertices on the elimination tree path from the root to v while relaxing outgoing upward edges of each vertex also yields shortest distances. Further, with this approach, when a vertex v already has a distance $D[v] \neq \perp$, all ancestors of v must already have their final distance. Thus, as soon as the algorithm encounters a vertex with a final distance, the remaining search space is known to have final distances.

We obtain the procedure described Algorithm 4, which utilizes the elimination tree. For any vertex that already has a memoized distance $D[v] \neq \perp$, the algorithm immediately returns this distance. Otherwise, the algorithm follows the elimination tree upward until a vertex with a finalized distance is encountered. The elimination tree is represented as parent pointer array P. The visited vertices are pushed onto a stack S. This enables the algorithm to enumerate the vertices in reversed order by popping them from the stack. While popping the vertices, all outgoing upward edges are relaxed. This finalizes the shortest distances for all vertices on the elimination tree path, including the desired query vertex v.

4 OPTIMIZATIONS FOR A* IN ROAD NETWORKS

In this section, we propose optimizations for A* in road networks. First, we present several low-degree vertex optimizations that exploit road network characteristics to reduce the overhead of queue operations and heuristic evaluations. Second, we discuss the bidirectional A* algorithm and propose an improved pruning criterion for symmetric bidirectional potentials. These optimizations can be used with *any* A* heuristic.

4.1 Low-Degree A* Improvements

Preliminary experiments showed that a significant amount of query running time is spent in heuristic evaluations and queue operations. We can reduce both by keeping some vertices out of the queue, as the heuristic only needs to be evaluated when a vertex is pushed into the queue. For example, consider a chain of vertices with precisely two neighbors. Traversing this chain by successively pushing each vertex into the queue, evaluating the heuristic, and popping the vertex again from the queue appears quite wasteful. Therefore, we now explore techniques to process such vertices consecutively without using the queue. The techniques discussed here are essentially a lazy variant of the ideas used in TopoCore [29].

4.1.1 Skip Degree Two Vertices. Recall that $|N(u)|$ is the number of neighbors v such that $vu \in E$ or $uv \in E$. Our algorithm differs from classical A* when removing a vertex u from the queue. A* iterates over the outgoing edges uv of u and tries to reduce $D[v]$ by relaxing uv. If A* succeeds, v's weight in the queue is set to $D[v] + h_t(v)$. Our algorithm, however, behaves differently, if $|N(v)| \leq 2$. Our algorithm determines the longest degree two chain of vertices $u, v_1, \ldots, v_k, w$ such that $|N(v_i)| = 2$ and $|N(w)| > 2$. If our algorithm succeeds in reducing $D[v_1]$, it does not push v_1 into the queue. Instead, it iteratively tries to reduce all $D[v_i]$. It stops if a $D[v_i]$ cannot be reduced. If it does not reach w, then only D is modified, but no queue action is performed. If $D[w]$ is modified and $|N(w)| > 2$, w's weight in the queue is set to $D[w] + h_t(w)$.

As the target vertex t might have degree two, our algorithm cannot rely on stopping when t is removed from the queue. Instead, our algorithm stops as soon as $D[t]$ is less than the minimum weight in the queue.

4.1.2 Skip Degree Three Vertices. We can also skip some degree three vertices. Denote by $u, v_1, \ldots, v_k, w$ a degree two chain as described in the previous section. If $|N(w)| > 3$ or w is in the queue, our algorithm proceeds as in the previous section. Otherwise, there exist up to two

degree chains $w, x_1, \ldots, x_p, y$ and $w, a_1, \ldots, a_q, b$ such that $x_1 \neq v_k \neq a_1$. Our algorithm iteratively tries to reduce all $D[x_i]$ and $D[a_i]$. If it reaches b, b's weight in the queue is set to $D[b] + h_t(b)$. Analogously, if y is reached, y's weight is set to $D[y] + h_t(y)$. If neither y nor b are reached, no queue operation is performed. Using this method, we avoid pushing every other degree three vertex into the queue.

4.1.3 Stay in Largest Biconnected Component. Many vertices in road networks lead to dead ends. Unless the source or target is in this dead end, it is unnecessary to explore these vertices.

In the preprocessing phase, we compute the subgraph G_C, called *core*. G_C is induced by the largest biconnected component of the undirected graph underlying G. We compute the core using Tarjan's algorithm [56]. For every vertex v in the input graph G, we store an *attachment vertex* a_v to the core. For vertices in the core, $a_v = v$. For every vertex v outside of the core, the attachment vertex a_v is the cut vertex in the core that separates v's component from the core (or a sentinel value $\perp$ for vertices in components not connected to the core).

The query phase is divided into two steps. First, we run A* on the subgraph induced by the core and s's component. Formulated differently, we only consider vertices that are in the core or have the same attachment vertex as s. We achieve this implicitly by not following edges to vertices without this property. If t is part of G_C or in the same component as s, this A* search finds it. Otherwise, we find a_t. In that case, we continue by searching a path from a_t toward t restricted to t's biconnected component. The final result is the concatenation of both paths. When t is not connected to the core ($a_t = \perp$) but s is ($a_s \neq \perp$), we immediately return a distance of ∞.

4.2 Bidirectional A*

On road graphs, bidirectional search provides a simple way to halve the practical running time of Dijkstra's algorithm. Thus, a bidirectional variant of A* also seems desirable. However, as shown in the work of Goldberg and Harrelson [38], the necessary modifications are not straightforward. We revisit bidirectional A* and propose an alternative approach. Our experiments show that it is competitive with the solution described by Goldberg and Harrelson [38].

The straightforward idea is to use two heuristics $\overrightarrow{h}_t(v)$ and $\overleftarrow{h}_s(v)$. The forward search has its queue ordered by $\overrightarrow{D}[v] + \overrightarrow{h}_t(v)$, where $\overrightarrow{h}_t(v)$ estimates the distance $\text{dist}(v, t)$ from v to t. Similarly, the backward search has its queue ordered by $\overleftarrow{D}[v] + \overleftarrow{h}_s(v)$, where $\overleftarrow{h}_s(v)$ is an estimate of the distance $\text{dist}(s, v)$ from s to v.

The problem with this straightforward approach is that these two potentials induce different reduced graphs (see Section 2.2). Thus, each direction would run on a different graph in the equivalent bidirectional Dijkstra search. This breaks the usual bidirectional Dijkstra stopping criterion. To the best of our knowledge, no better stopping criterion exists than running *both* searches until the unidirectional stopping criterion is met for one direction. The forward search can skip vertices already settled by the backward search and vice versa. Unfortunately, this straightforward bidirectional A* still performs more work than a unidirectional A* [38]. Thus, on its own, it is not a useful algorithm. However, it can serve as a basis for further algorithmic refinements. Goldberg and Harrelson[38] refer to this as *symmetric* bidirectional A*.

To obtain a bidirectional stopping criterion, the *average potential* is proposed [38]. It combines a forward and a backward heuristic $\overrightarrow{h}_t$ and $\overleftarrow{h}_s$ into a combined *average heuristic* $h_{\overline{st}}$. The idea is to use a common reduced graph, whose weights are the average weights of the individual reduced graphs. Both searches run on the same common reduced graph. This allows stopping the searches when $\overrightarrow{k} + \overleftarrow{k} \geq \mu$. Formally, $h_{\overline{st}}(v)$ is defined as $(\overrightarrow{h}_t(v) - \overleftarrow{h}_s(v))/2$. The forward search uses $h_{\overline{st}}(v)$ as its heuristic. The backward search uses $-h_{\overline{st}}(v)$. Unfortunately, average potentials have two

downsides. First, evaluating the average potential requires evaluating both $\overrightarrow{h}_t$ and $\overleftarrow{h}_s$. Evaluating the average heuristic is therefore slower than evaluating just one of the composing heuristics. Second, the bidirectional stopping criterion comes at the cost of worse estimates for each direction on its own. $h_{\overline{st}}(v)$ is a worse estimate for $\text{dist}(v,t)$ than $\overrightarrow{h}_t$. Similarly, $-h_{\overline{st}}(v)$ is a worse estimate for $\text{dist}(s,v)$ than $\overleftarrow{h}_s$.[3] The second downside can be partially mitigated through pruning with the composing heuristics. When the forward search scans an edge uv where $\overrightarrow{D}[u] + w(uv) + \overrightarrow{h}_s(v) > \mu$ holds (i.e., the distance plus the estimate of the remaining distance is already greater than the currently known tentative distance), it is not necessary to push v into the queue. The pruning rule for the backward search is analogous.

To avoid the downsides of the average potential, we revisit symmetric bidirectional A* and propose a new pruning criterion. We describe the idea for the forward search. The pruning rule for the backward search is analogous. The central idea consists of using information from the backward search to prune edges in the forward search. We do not use the average heuristic. Instead of a strong stopping criterion, we use a pruning rule that gets stronger the longer the search runs. Such a pruning rule will eventually prune all remaining branches and stop the search. The stopping criteria for each direction remain the same (unidirectional) as before. However, usually, the search stops early because the queues are empty.

Let uv be an edge that we relax in the forward search. Before pushing v into the queue, we apply the new pruning rule. If we can prove that every path using uv is at least as long as the shortest known path length μ, then we do not have to push v. We therefore want to obtain a lower bound for $\text{dist}(s,u) + w(uv) + \text{dist}(v,t)$. As u is settled, $\overrightarrow{D}[u]$ contains the shortest path length $\text{dist}(s,u)$ (i.e., $\text{dist}(s,u) = \overrightarrow{D}[u]$). $w(uv)$ is also known as it is just an edge weight. It remains to lower bound $\text{dist}(v,t)$. Vertices are removed from the backward queue ordered by $\overleftarrow{h}(v) + \text{dist}(v,t)$. If v is not yet removed from the backward queue, we know that $\overleftarrow{h}(v) + \text{dist}(v,t) \geq \overleftarrow{k}$. This gives us the required lower bound (i.e., $\text{dist}(v,t) \geq \overleftarrow{k} - \overleftarrow{h}(v)$). Thus, v does not have to be pushed if $\overrightarrow{D}[u] + w(uv) + \overleftarrow{k} - \overleftarrow{h}(v) \geq \mu$. The vertex v might still be pushed into the queue when there is another edge wv for which pruning is impossible. Checking the pruning rule requires evaluating the backward heuristic. However, pruning is only possible once the searches have met (i.e., $\mu < \infty$). Before that, each direction only has to evaluate its own heuristic. Thus, our pruning improves on both downsides of the average potential.

Unless stated differently, for bidirectional A*, we always alternate between removing a vertex from the forward and the backward queues. We also evaluate expanding the search with the smaller minimum queue key in our experiments. Although this may sound sensible, our experiments show later in Table 4 that it is never beneficial in terms of running time.

5 THE CH-POTENTIALS FRAMEWORK

In this section, we introduce an algorithmic framework to apply A* with a Lazy RPHAST based heuristic to various practical route planning problems. We call our approach *CH-Potentials*. The core idea is to compute a CH augmented graph during preprocessing and use A* with a straightforward application of Lazy RPHAST as the heuristic to answer queries. When the CH preprocessing and the A* algorithm are performed on the same graph with the same weight function, this yields a *perfect* heuristic. However, this case is, of course, not particularly interesting. One could just

[3] To obtain an actual lower bound from this average heuristic, one has to add $\overleftarrow{h}_s(t)/2$ in the forward case and $\overrightarrow{h}_t(s)/2$ in the backward. Adding any constant to a heuristic function does not change the reduced graph.

answer the shortest path query directly with a CH query. The approach becomes useful when the query runs on a *different but related* graph or weight function than the preprocessing. Therefore, we start by establishing a common formal framework for the use of CH-Potentials. Then, we exemplarily describe some extended routing problems and how to apply the CH-Potentials framework to them.

5.1 Formal Problem Setup: Inputs, Outputs, and Phases

We consider a variety of different applications with slightly different problem models. The goal is always to answer many shortest path queries quickly. To describe our framework, we establish a shared notation: input to each query are vertices s and t, and a graph G_q with query weights w_q. However, the precise formal inputs of the query and what exactly w_q represents depends on the application. In the simplest case, w_q will be scalar edge weights. Live traffic is an example of this. The challenge in this scenario is that values of w_q might change between queries. However, w_q can also represent something more complex than scalar numbers. It can be any function that computes a weight for an edge. This function can also take additional parameters from the state of the search. In the case of traffic predictions, w_q is a function mapping the edge entry time to the traversal time, and the query takes an additional departure time parameter.

To enable quick shortest path computations, we consider a two-phase setup with an additional offline preprocessing phase before the online query phase. The input to the preprocessing phase is a lower bound graph $G_\ell = (V_\ell, E_\ell)$ with lower bound weights w_ℓ where $w_\ell(e)$ must be a scalar value for every edge e of G_ℓ. The preprocessing output is auxiliary data that allows to quickly compute distances on G_ℓ with respect to w_ℓ. In the applications considered in this article, w_ℓ is always the *free-flow* travel time.

The query phase may use this auxiliary data to answer shortest path queries between vertices s and t on $G_q = (V_q, E_q)$ with weights w_q. Let $\phi : V_q \to V_\ell$ denote a function mapping vertices in the query graph to vertices in the lower bound graph. The only requirement for a routing problem to fit into our problem framework is that the query weight of an edge $w_q(uv)$ is greater or equal to the shortest distance $\text{dist}_\ell(\phi(u), \phi(v))$ between the corresponding vertices $\phi(u)$ and $\phi(v)$ in the lower bound graph G_ℓ with respect to w_ℓ. Unless stated otherwise, G_q and G_ℓ are the same graph, ϕ is the identity function, and only w_q changes for the queries.

5.2 CH-Potentials

CH-Potentials can be used to solve any problem in this setup. The preprocessing is always the computation of the CH augmented graph G_ℓ^+ and remains the same regardless of the specific routing problem. The query consists of A* with the heuristic function $h_t(v) = \text{dist}_{w_\ell}(\phi(v), \phi(t))$ computed using Lazy RPHAST. At the beginning of each query, we perform the target selection (i.e., a backward CH search) from the target t. The heuristic function $h_t(v)$ is implemented by a call to the `ComputeAndMemoizeDist` for vertex $\phi(v)$ (see Algorithm 3). In contrast to the preprocessing phase, the exact implementation of the A* search depends on the application. Our approach only provides the heuristic h_t for the A* search. As the performance of A* depends on the accuracy of the heuristic estimates, the smaller the difference between query weights and lower bound distances, the better CH-Potentials will perform.

Correctness. Our heuristic is always *feasible*—that is, $w_q(uv) - h_t(u) + h_t(v) \geq 0$ holds for all edges. By requirement and because of the triangle inequality, the following must hold:

$$w_q(uv) - h_t(u) + h_t(v) \geq \text{dist}_\ell(\phi(u), \phi(v)) - \text{dist}_\ell(\phi(u), \phi(t)) + \text{dist}_\ell(\phi(v), \phi(t)) \geq 0.$$

Thus, A* will always determine the optimal shortest distance.

5.3 Applications

5.3.1 Avoiding Tunnels and/or Highways. Avoiding tunnels or highways is a common feature of navigation devices. Implementing this feature with CH-Potentials is easy. We set w_ℓ to the free-flow travel time. If an edge is a tunnel or a highway, we set w_q to $+\infty$. Otherwise, w_q is set to the free-flow travel time.

5.3.2 Forbidden Turns and Turn Costs. The classical shortest path problem allows changing edges at vertices freely. However, in the real world, turn restrictions, such as a forbidden left or right turn, exist. Additionally, taking a left turn might take longer than going straight. This can be modeled using turn weights [18, 22, 37]. A *turn weight* $w_t : E \times E \to \mathbb{R}^{\geq 0}$ maps a pair of incident edges onto the turning time or $+\infty$ for forbidden turns. For CH-Potentials, we use zero as the lower bound for every turn weight in the heuristic. Thus, the graph G_ℓ and weights w_ℓ for preprocessing is the unmodified input graph without turn weights.

A path with vertices $v_1, v_2, \dots v_k$ has the following *turn-aware weight*:

$$w_\ell(v_1v_2) + \sum_{i=2}^{k-1} w_t(v_{i-1}v_i, v_iv_{i+1}) + w_\ell(v_iv_{i+1}).$$

The objective is to find a path between two given edges with minimum turn-aware weight. The first term $w_\ell(v_1, v_2)$ is the same for all paths, as it only depends on the source edge. It can thus be ignored during optimization.

We solve this problem by constructing a *turn-expanded* graph as G_q. Edges in the input graph G_ℓ correspond to *expanded vertices* in G_q. For every pair of incident edges xy and yz in G_ℓ, there is an *expanded edge* in G_q with expanded weight $w_t(xy, yz) + w_\ell(yz)$. A sequence of expanded vertices in the expanded graph G_q corresponds to a sequence of edges in the input graph G_ℓ. The weight of a path in G_q is equal to the turn-aware weight of the corresponding path in G_ℓ minus the irrelevant $w_\ell(v_1v_2)$ term. Thus, the turn-aware routing problem can be solved by searching for shortest paths in G_q.

In this scenario, preprocessing and query use different graphs G_ℓ and G_q. We define the vertex mapping function ϕ as $\phi(xy) = y$. Obviously, $w_q((xy, yz)) = w_t(xy, yz) + w_\ell(yz) \geq \text{dist}_\ell(\phi(xy), \phi(yz))$ and this approach yields a feasible heuristic. Sadly, the undirected graph underlying G_q is always biconnected, if the input graph is strongly connected. The optimization described in Section 4.1.3 is therefore ineffective. With this setup, CH-Potentials supports turn costs without requiring turn information in the CH.

5.3.3 Predicted Traffic or Time-Dependent Routing. The classical shortest path problem assumes that edge weights are scalars. However, in practice, travel times vary along an edge due to traffic. Recurring traffic can be predicted by observing the traffic in the past. It is common [8, 14, 53] to represent these predictions as *travel time functions*. An edge weight is no longer a scalar value but a function that maps the entry time onto the traversal time.

In this setting, the query weight w_q is a function from $E \times \mathbb{R}$ to $\mathbb{R}^+$. $w_q(e, \tau)$ is the travel time through edge e when entering it at moment τ. The input to the extended problem consists of a source vertex s and a target vertex t, as in the classical problem formulation. Additionally, the input contains a source time τ_s. A path with edges $e_1, e_2 \dots e_k$ is weighted using α_k, which is defined recursively as follows:

$$\begin{aligned} \alpha_1 &= w_q(e_1, \tau_s) \\ \alpha_k &= \alpha_{k-1} + w_q(e_1, \alpha_{k-1}). \end{aligned}$$

The objective is to find a path to t that minimizes α_k.

If all travel time functions fulfill the *FIFO property*, this problem can be solved using a straightforward extension of Dijkstra's algorithm [32]. The necessary modification to A* is analogous. Without the FIFO property, the problem becomes NP-hard [50, 58]. The FIFO property states that arriving earlier by departing later is impossible. Formally stated, the following must hold $\forall e \in E, \tau \in \mathbb{R}, \varepsilon \in \mathbb{R}^{>0} : w_q(e, \tau) \leq w_q(e, \tau + \varepsilon) + \varepsilon$. Our implementation stores edge travel times using piecewise linear functions. The A* search uses the tentative distance τ at a vertex u when evaluating the travel time of outgoing edges uv. This strategy is quite similar to TD-ALT [27, 49].

For the preprocessing, we set $w_\ell(e) = \min_\tau w_q(e, \tau)$, which is the minimum travel time. By keeping travel time functions out of the CH preprocessing, we avoid a lot of algorithmic complications compared to other works [8, 14, 25, 53] that have to create shortcuts of travel time functions.

5.3.4 Live and Predicted Traffic. Besides predicted traffic, we also consider live traffic. Live traffic refers to the current traffic situation. It is essential to distinguish between predicted and live traffic. Live traffic data is more accurate for the current moment than predicted data and may differ significantly from predicted traffic if unexpected events like accidents happen. However, only using live traffic data is problematic for long routes as traffic changes while driving. At some point, one wants to switch from live traffic to predicted traffic. In this section, we first describe a setup with only live traffic and then combine it with predicted traffic.

To support only live traffic, we set w_ℓ to the free-flow travel time. w_q is set to the travel time accounting for current traffic. As traffic only increases travel times, w_ℓ is a valid lower bound for w_q. In a real-world application, values from w_q could be updated between queries. This is all that is necessary to apply CH-Potentials in a live traffic scenario.

To combine live traffic with predicted traffic, we define a modified travel time function w_q that is then used as query weights. Denote by $w_p(e, \tau)$ the predicted travel time along edge e at moment τ. Further, $w_c(e)$ is the travel time according to current live traffic. Finally, we denote by τ_{soon} the moment we switch to predicted traffic. In our experiments, we set τ_{soon} to 1 hour in the future. We must ensure that the modified travel time function fulfills the FIFO property. For this reason, we cannot make a hard switch at τ_{soon}. Our modified travel time function linearly approaches the predicted travel time. Formally, we set $w_q(e, \tau)$ to $w_c(e)$, if $\tau \leq \tau_{\text{soon}}$. Otherwise, we check whether $w_p(e, \tau_{\text{soon}}) < w_c(e)$. If it is the case, we set $w_q(e, \tau)$ to $\max\{w_c(e) + (\tau_{\text{soon}} - \tau), w_p(e, \tau)\}$. Otherwise, we set $w_q(e, \tau)$ to $\min\{w_c(e) - (\tau_{\text{soon}} - \tau), w_p(e, \tau)\}$. In our implementation, we do not modify the representation of w_p but evaluate the preceding formulas at each travel time evaluation. We set w_ℓ to the free-flow travel time.

With this setup, CH-Potentials supports a combination of real-time and predicted traffic. We did not make any modifications that would hinder a combination with other extensions. Further adding tunnel or highway avoidance or turn-aware routing is simple. This straightforward integration of complex routing problems is the strength of CH-Potentials.

5.3.5 Live Traffic with CCH-Potentials. Alternatively, live traffic can be supported using a three-phase setup as described in Section 2.4. In this case, we use the CCH instead of the CH preprocessing to build the augmented graph. This allows us to introduce updates to w_ℓ and the augmented graph within a few seconds using the CCH customization phase. This setup provides an alternative way of integrating live traffic compared to the approach described in Section 5.3.4 where live traffic was represented only in w_q. In contrast, with CCH-Potentials, changes to the live traffic situation are handled by changing w_ℓ via customization. Additional features, such as turn restrictions, can be implemented by adjusting w_q. For severe traffic events, the three-phase approach will have significantly faster query running times than the one described in Section 5.3.4. However, it leads to a more complex setup and updating w_ℓ takes a few seconds.

Table 1. Instances Used in the Evaluation with Sequential Preprocessing Running Times to Construct (C)CH-Potentials

	Vertices [$\cdot 10^6$]	Edges [$\cdot 10^6$]	Preprocessing [s]			Aux. Data [MiB]	
				CCH			
			CH	Phase 1	Phase 2	CH	CCH
OSM Germany	16.2	35.4	298.7	1,467.4	10.1	645	616
DIMACS Europe	18.0	42.2	276.2	2,480.9	12.4	742	765
TDGer06	4.7	10.8	59.2	331.7	2.7	196	169
TDEur17	25.8	55.5	293.9	2,102.3	14.1	1,030	881
TDEur20	28.5	60.9	311.9	2,219.5	15.2	1,130	959

With CCH-Potentials, w_ℓ can be updated by rerunning Phase 2.

6 EVALUATION

In this section, we present our experimental evaluation. Our benchmark machine runs openSUSE Leap 15.3 (kernel 5.3.18), and has 128 GiB of DDR4-2133 RAM and an Intel Xeon E5-1630 v3 CPU, which has four cores clocked at 3.7 Ghz and 4 × 32 KiB of L1, 8 × 256 KiB of L2, and 10 MiB of shared L3 cache. All experiments were performed sequentially. Our code is written in Rust and compiled with `rustc 1.57.0-nightly` in the release profile with the `target-cpu=native` option. The source code of our implementation and the experimental evaluation can be found on GitHub.[4]

6.1 Inputs and Methodology

Our primary benchmark instance is a graph of the road network of Germany obtained from OpenStreetMap.[5] To obtain the routing graph, we use the import from RoutingKit.[6] The graph has 16M vertices and 35M edges. For this instance, we have proprietary traffic data provided by Mapbox.[7] The data includes a live traffic snapshot from Friday 2019/08/02 in the afternoon and comes in the form of 320K OSM node pairs and live speeds for the edge between the vertices. It also includes traffic predictions for 38% of the edges as predicted speeds for all 5-minute periods over a week. We exclude speed values faster than the free-flow speed computed by RoutingKit. We also perform experiments on the Europe instance provided by PTV[8] for the 9th DIMACS implementation challenge [28]. Additionally, we have three graphs with proprietary traffic predictions also provided by PTV. The PTV instances are not OSM based. One is an old instance of Germany with traffic predictions from 2006 for 7% of the edges, which was used to evaluate many competing time-dependent algorithms. The other two are newer instances of Europe with predictions for 27% (TDEur17) and 76% (TDEur20) of the edges. Table 1 contains an overview over the instances. In this table, we further report the sequential preprocessing running time to construct the augmented graphs and the space consumption. We use the CH preprocessing from RoutingKit.[9] For CCH, we use our own implementation with nested dissection orders obtained by InertialFlowCutter [41]. We report sequential preprocessing running times as averages over 10 runs. Note that CCH preprocessing can be parallelized efficiently. Thus, practical running times can be even faster. To evaluate point-to-point queries, we generate 10,000 queries where both source and target are vertices drawn uniformly at random and report average results.

[4] https://github.com/kit-algo/ch_potentials.
[5] https://download.geofabrik.de/europe/germany-200101.osm.pbf.
[6] https://github.com/RoutingKit/RoutingKit.
[7] https://mapbox.com.
[8] https://ptvgroup.com.
[9] https://github.com/RoutingKit/RoutingKit.

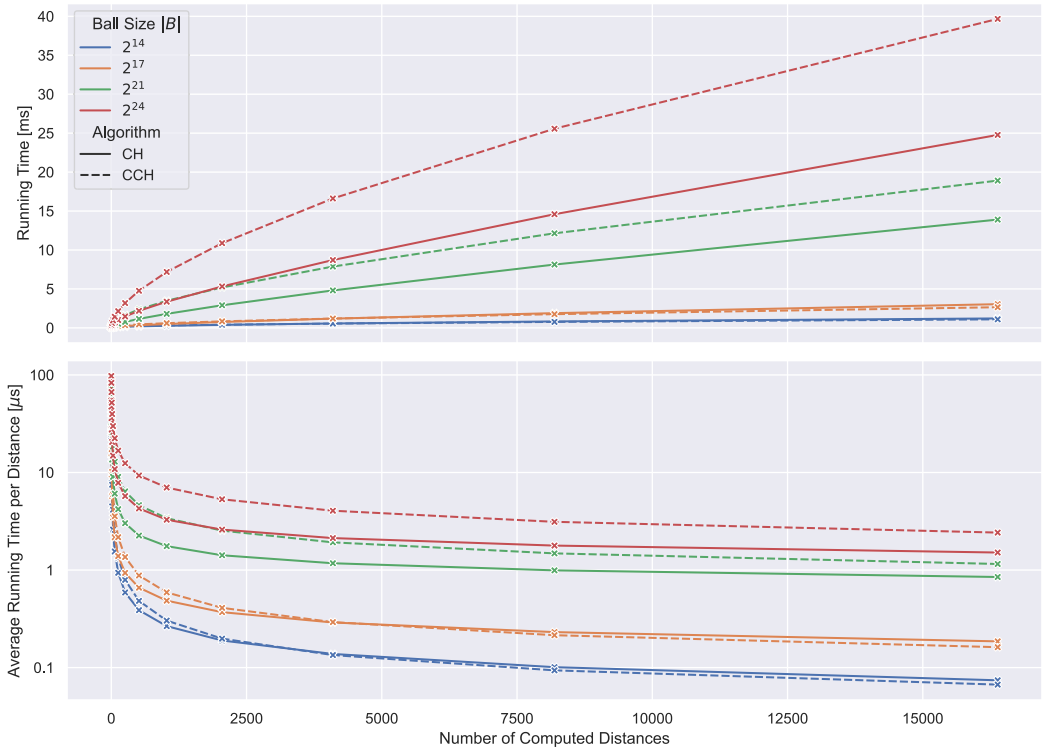

Fig. 3. Average running times of incremental Lazy RPHAST while querying $|S| = 2^{14}$ from a ball of varying size $|B|$ on OSM Germany excluding selection times. The upper figure contains the total elapsed running time. The lower figure contains the averaged running time per source (i.e., y/x from the upper figure). Note the different y-axis scales and units.

Lazy RPHAST is evaluated with many-to-one queries where each query consists of a source set S with 2^{14} sources and one target vertex t. However, instead of picking sources and targets from the full vertex set V, we draw them from local subsets of vertices B of varying size $|B|$ called *balls*. A ball B is generated by picking a center uniformly at random and running Dijkstra's algorithm from it until the desired number of vertices $|B|$ is settled. This allows us to evaluate the performance depending on the distribution of the vertices. Since we use a fixed number of sources per query, a small ball size means that the vertices are densely clustered in the same region, whereas large ball sizes mean that the vertices are distributed over large parts of the network. For each ball size, we generate 100 balls. We pick one set of sources from each ball to which we compute distances from 100 different targets selected uniformly at random from the same ball. Therefore, each reported running time is the mean over 10,000 queries. With this, we follow the methodology from Delling et al. [23].

6.2 Lazy RPHAST

To evaluate Lazy RPHAST in the incremental setting, we measure the elapsed running time after distances from 2^i sources were queried. Figure 3 depicts the total elapsed time and the average running time to compute a single distance. The first few distances are the most expensive since much of the CH search space has not been explored yet. With around 100 μs, the running times are comparable to standard CH queries. For later queries, little work remains to be done, and the overhead per distance becomes almost constant depending on the ball size.

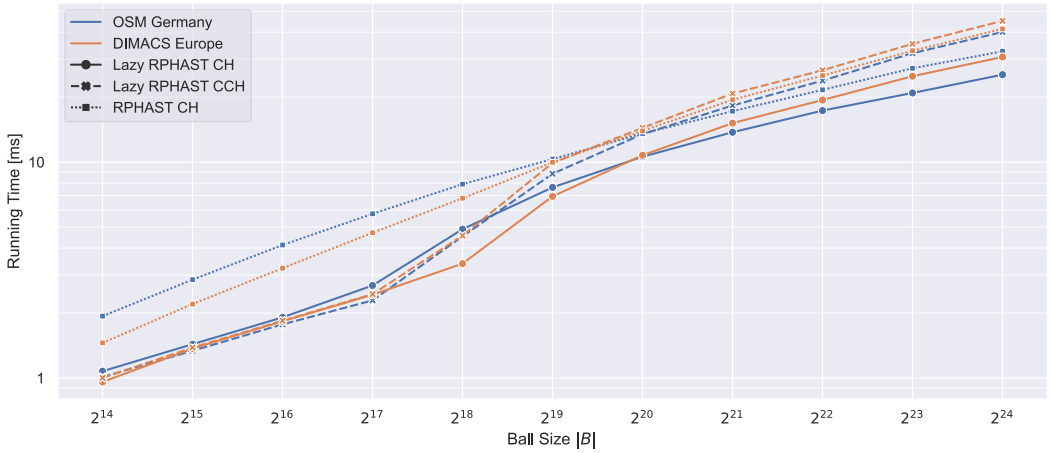

Fig. 4. Running times of (C)CH-based Lazy RPHAST and CH-based RPHAST for many-to-one queries with $|S| = 2^{14}$ sources picked from a ball of varying size $|B|$. The running time includes the selection and the time to compute all distances.

The performance difference between the CH and CCH-based variants is interesting. The CCH-based variant can utilize the elimination tree for a more efficient implementation, but the search space is denser. This makes earlier queries more expensive and later queries cheaper. Therefore, it depends on the ball size which variant is faster in terms of total running time.

Despite laziness being the distinguishing feature in Lazy RPHAST, the algorithm can still be used for nonincremental many-to-one queries. Figure 4 depicts average running times to compute distances between one target vertex and 2^{14} sources for different ball sizes for Lazy RPHAST and our own implementation of RPHAST [23]. The query generation methodology is the same as in the previous experiment. As this is the same methodology also used in the work of Delling et al. [23], we can also roughly relate our results to the performance of other one-to-many algorithms. Keep in mind that algorithms like RPHAST optimize for a different setting than we do. We can compare the performance for fixed S-t terminal sets. The difference is that with RPHAST, one can efficiently compute distances from a different t' to the same S set, whereas with Lazy RPHAST, one can efficiently extend S while t stays the same.

When looking at total running times for a single many-to-one query, including selection times, Lazy RPHAST yields performance competitive to RPHAST. The CH-based variant is consistently faster than RPHAST. Further, even the Lazy RPHAST CCH-based variant is faster than CH-based RPHAST on ball sizes up to $|B| = 2^{19}$ despite the larger search space. Unsurprisingly, the absolute running times of RPHAST and Lazy RPHAST are similar overall: both algorithms run on the same CH search space. Our RPHAST running times roughly reproduce the results reported by Delling et al. [23]: our running times are between 20% (on small ball sizes) and 60% (on larger ball sizes) faster, which is likely due to differences in benchmark machine performance. We conclude that Lazy RPHAST is a very valuable extension of RPHAST. It allows for efficiently handling dynamic source sets and is even competitive in the setting where both the target and the source set change between queries.

6.3 (C)CH-Potentials Heuristic

The performance of A* depends on the tightness of the heuristic and the overhead of evaluating the heuristic. CH-Potentials computes optimal distance estimates with respect to w_ℓ. However, for

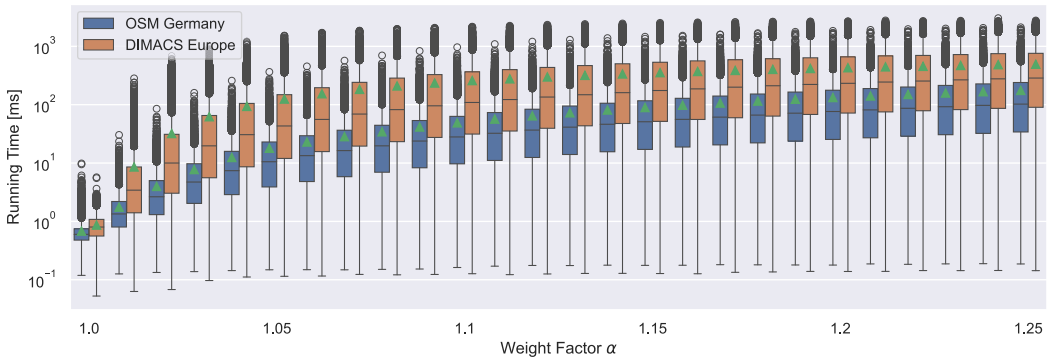

Fig. 5. Running times on a logarithmic scale for queries on OSM Ger with scaled edge weights $w_q = \alpha \cdot w_\ell$. The boxes cover the range between the first and third quartile. The band in the box indicates the median and the diamond the mean. The whiskers cover 1.5 times the interquartile range. All other running times are indicated as outliers.

Table 2. Average Query Running Times and Number of Queue Pushes with Different Heuristics and Low-Degree Optimizations on OSM Ger with $w_q = 1.05 \cdot w_\ell$

	BCC	Deg2	Deg3	Zero	ALT	CH	CCH	Oracle
Running time [ms]	○	○	○	2,133.0	317.9	47.9	54.4	34.3
	●	○	○	1,355.3	233.9	36.3	38.5	24.8
	●	●	○	753.4	122.6	19.5	22.1	12.7
	●	●	●	580.7	90.8	15.9	18.1	10.1
Queue [$\cdot 10^3$]	○	○	○	8,087.1	863.1	137.1	137.1	137.1
	●	○	○	6,298.2	685.7	112.7	112.7	112.7
	●	●	○	2,901.4	303.4	43.3	43.3	43.3
	●	●	●	1,681.4	179.7	26.8	26.8	26.8

The BCC column indicates if the search stays within the largest biconnected component (see Section 4.1.3), Deg2 if vertices of degree two are skipped (Section 4.1.1), and Deg3 if vertices of degree three are skipped (Section 4.1.2).

most applications, there will be a gap between w_q and w_ℓ (otherwise, one could use CH without A*). We evaluate the impact of the difference between w_q and w_ℓ on the performance of A*. The lower bound w_ℓ is set to the free-flow travel time. The query weights w_q are set to $\alpha \cdot w_\ell$, where $\alpha \geq 1$. Increasing α degrades the heuristic's quality. Figure 5 depicts the results. Clearly, α has a significant influence on the running time. Average running times range from below a millisecond to a few hundred milliseconds depending on α. Up to around $\alpha = 1.1$, the running time grows quickly. For $\alpha > 1.1$, the growth slows down. This illustrates the strengths and limits of our approach and goal-directed search in general. CH-Potentials can only achieve competitive running times if the application allows tight lower bounds at preprocessing time.

We observe that the running times for a fixed α fluctuate heavily. This is an interesting observation, as with uniform source and target sampling, nearly all queries are long-distance. The query distance is thus not the reason. After some investigation, we concluded that this is due to nonuniform road network density. Some regions have more roads per area than others. The number of vertices explored by A* depends on the density of the search space area. As the density varies, the running times vary.

Table 2 depicts the performance of unidirectional A* with different heuristics and optimizations on OSM Ger with $w_q = 1.05 \cdot w_\ell$. The factor 1.05 was chosen to resemble realistic problem settings

Table 3. Performance of Different Variants of Bidirectional A* on OSM Ger with $w_q = 1.05 \cdot w_\ell$

Low Deg. Opt.	Bidirectional Potential	New Pruning	Running Time [ms] Zero	ALT	CH	CCH	Oracle	Queue Pushes [$\cdot 10^3$] Zero	ALT	(C)CH/ Oracle
○	Average	○	1,441.41	126.46	62.61	53.91	37.29	4,493.97	292.01	125.16
○	Average	●	1,451.96	128.20	62.48	54.28	38.89	4,491.56	290.92	125.08
○	Symmetric	○	5,779.64	795.56	122.70	111.78	88.66	16,042.82	1,688.60	259.78
○	Symmetric	●	1,453.58	261.80	59.22	51.97	37.37	4,491.56	624.25	116.71
●	Average	○	365.82	33.22	19.34	18.66	9.96	916.15	57.27	23.60
●	Average	●	369.51	33.37	19.54	18.88	9.98	908.55	56.09	23.25
●	Symmetric	○	1,512.48	241.27	40.98	38.99	26.36	3,317.81	334.90	44.67
●	Symmetric	●	368.94	72.67	21.54	20.39	11.22	908.55	123.77	20.72

All variants alternate between the forward and the backward search.

where goal-directed search can achieve reasonable speedups (compare to Table 6). We compare (C)CH-Potentials to three other heuristics. The first heuristic is the *Zero* heuristic where $h(v)$ is 0 for all vertices v. This corresponds to using Dijkstra's algorithm. Second, we compare against our implementation of ALT [40]. We use 16 landmarks generated with the avoid strategy [40] and activate all during every query. Finally, we compare against a hypothetical *Oracle-A** heuristic. This heuristic has instant access to a shortest distance array with respect to w_ℓ (i.e., it is faster than the fastest heuristic possible in our model). We fill this array before each query using a reverse Dijkstra search from the target vertex but do not include the running time for this step. Thus, the reported running times of Oracle-A* do *not* account for any heuristic evaluation. (C)CH-Potentials computes the same distance estimates, but the heuristic evaluation has some overhead. Comparing against Oracle-A* allows us to measure this overhead. No other heuristic, which only has access to the preprocessing weights, can be faster than Oracle-A*.

We observe that the number of queue pushes roughly correlates with running time. Each optimization reduces both queue pushes and running times. All optimizations yield a combined speedup of around 3. (C)CH-Potentials outperforms ALT by a factor of between six and seven and settle correspondingly fewer vertices. This is not surprising since ALT computes worse distance estimates. In contrast, CH-Potentials already computes exact distances with respect to w_ℓ. As (C)CH-Potentials and Oracle-A* have the same heuristic values, the number of queue pushes are by construction equal. The only difference between (C)CH-Potentials and Oracle-A* is the overhead of the heuristic evaluation. This overhead leads to a slowdown of around 1.6. Thus, CH-Potentials is already quite close to the best possible heuristic in this model. No competing algorithm such as ALT or CPD-Heuristics can be significantly faster. CCH-Potentials is slightly slower than CH-Potentials because they use a weight-independent vertex importance order.

6.4 Bidirectional A*

In this section, we investigate the performance of bidirectional A*. We first evaluate different variants of bidirectional A* in Tables 3 and 4, then compare the best ones against unidirectional A* in Table 5. Table 3 studies the impact of our improved pruning and the low-degree optimizations. As observed in the previous section, enabling the low-degree optimizations achieves a speedup of roughly three. Symmetric bidirectional A* without our improved pruning has the worst performance. Enabling the improved pruning improves the performance of symmetric bidirectional A* significantly. For all heuristics except ALT, symmetric A* with improved pruning has smaller search spaces than the average potential and similar running times. Without the low-degree

Table 4. Performance of Different Direction Selection Criteria of Bidirectional A* on OSM Ger with Different Query Weights

w_q	Bidirectional Potential	Choose Direction	Running Time [ms]					Queue Pushes [$\cdot 10^3$]		
			Zero	ALT	CH	CCH	Oracle	Zero	ALT	(C)CH/ Oracle
w_ℓ	Average	Alternating	373.18	12.83	0.79	1.13	0.18	916.15	23.08	0.60
	Average	Min. Key	406.35	13.68	1.44	1.75	0.56	986.40	26.39	1.15
	Symmetric	Alternating	376.72	40.19	0.69	0.92	0.19	908.55	76.61	0.57
	Symmetric	Min. Key	427.51	50.46	1.77	1.99	0.83	978.62	99.62	1.44
$w_\ell \cdot 1.05$	Average	Alternating	365.82	33.22	19.34	18.66	9.96	916.15	57.27	23.60
	Average	Min. Key	391.70	38.06	21.76	20.44	11.30	986.41	67.65	26.42
	Symmetric	Alternating	368.94	72.67	21.54	20.39	11.22	908.55	123.77	20.72
	Symmetric	Min. Key	394.38	84.84	27.28	24.64	14.53	978.63	145.28	24.82
$w_\ell \cdot 1.5$ if speed is <80 kph	Average	Alternating	361.83	19.50	10.92	10.94	5.34	845.06	34.03	13.25
	Average	Min. Key	391.47	31.65	21.05	20.10	11.00	917.13	52.23	23.78
	Symmetric	Alternating	364.55	37.33	11.89	11.75	6.00	836.44	57.93	11.53
	Symmetric	Min. Key	395.04	54.90	23.36	22.48	12.54	908.12	84.33	22.01

The symmetric variant uses the improved pruning, whereas the average variant does not. All variants use all low-degree optimizations.

Table 5. Performance of Bidirectional and Unidirectional A* on OSM Ger with Different Query Weights

w_q		Running Time [ms]					Queue Pushes [$\cdot 10^3$]		
		Zero	ALT	CH	CCH	Oracle	Zero	ALT	(C)CH/ Oracle
w_ℓ	Unidirectional	584.87	43.02	0.47	0.64	0.16	1,674.35	96.21	0.66
	Average	373.18	12.83	0.79	1.13	0.18	916.15	23.08	0.60
	Symmetric	376.72	40.19	0.69	0.92	0.19	908.55	76.61	0.57
$w_\ell \cdot 1.05$	Unidirectional	580.66	90.79	15.91	18.09	10.06	1,681.39	179.66	26.78
	Average	365.82	33.22	19.34	18.66	9.96	916.15	57.27	23.60
	Symmetric	368.94	72.67	21.54	20.39	11.22	908.55	123.77	20.72
$w_\ell \cdot 1.5$ if speed is <80 kph	Unidirectional	637.24	96.62	21.78	21.37	14.62	1,674.26	171.02	36.54
	Average	361.83	19.50	10.92	10.94	5.34	845.06	34.03	13.25
	Symmetric	364.55	37.33	11.89	11.75	6.00	836.44	57.93	11.53

The symmetric variant uses the improved pruning, whereas the average variant does not. All variants use all low-degree optimizations.

improvements, the improved symmetric variant is marginally faster. With the low-degree improvements, the average potential remains slightly faster. This is due to the reduced impact of the heuristic evaluation overhead with the low-degree optimizations. Enabling the improved pruning for the average potential reduces the search space size marginally and slightly increases running times.

Table 4 shows the performance of bidirectional A* with different strategies to decide whether to advance the forward or the backward search next. The results clearly show that alternating the directions is always superior. Selecting the direction by minimum queue key may lead to huge imbalances in the progress of the searches. This causes the searches to meet later and the total search space to grow significantly.

In Table 5, we investigate the effectiveness of bidirectional search compared to unidirectional search depending on the query weights. Interestingly, only the zero heuristic and ALT consistently achieve speedups through bidirectional search. With $w_q = w_\ell$, unidirectional CH-Potentials is

Table 6. CH-Potentials Performance for Different Route Planning Applications

			Running Time [ms]	Queue [$\cdot 10^3$]	Length Incr. [%]	Dijkstra [ms]	Speedup
DIMACS Eur	Unmodified $w_q = w_\ell$	CH U	0.9	1.1	0.0	2,106.0	2,405.8
OSM Ger	Unmodified $w_q = w_\ell$	CH U	0.6	0.5	0.0	2,182.6	3,795.4
	No Tunnels	CH U	29.2	46.8	5.2	2,198.0	75.2
		CH B	33.4	35.7	5.2	2,198.0	65.8
	No Highways	CH U	378.7	583.8	42.5	1,992.5	5.3
		CH B	433.1	481.6	42.5	1,992.5	4.6
	Live	CH U	129.4	193.9	15.0	2,119.3	16.4
		CH B	193.6	188.8	15.0	2,119.3	10.9
		CCH U	1.1	0.8	0.0	2,119.3	1,920.4
	Turns	CH U	3.0	5.7	1.1	4,708.2	1,556.0
		CH B	1.1	0.8	1.1	4,708.2	4,223.8
	Live + Turns	CCH U	4.8	8.8	1.0	4,621.8	959.7
		CCH B	2.1	1.6	1.1	4,621.8	2,168.1
	TD	CH U	120.8	104.4	12.3	3,133.7	25.9
	TD + Live	CH U	198.3	170.3	20.7	3,436.5	17.3
	TD + Live + Turns	CH U	474.2	657.8	21.7	6,420.5	13.5
TDEur17	TD	CH U	80.4	79.8	3.9	3,454.3	43.0
TDEur20	TD	CH U	97.7	72.8	4.2	5,060.2	51.8
TDGer06	TD	CH U	4.2	6.4	3.1	603.5	144.2

Depending on the problem, we apply unidirectional or bidirectional CH-Potentials (CH U or CH B) or CCH-Potentials (CCH U/B). We report average running times and the number of queue pushes. We also report the average length increase—that is, how much longer the final shortest distance is compared to the lower bound. Finally, we report the average running time of Dijkstra's algorithm as a baseline and the speedup over this baseline.

already optimal and only traverses the shortest path. Here, the bidirectional search will only introduce unnecessary overhead. When query weights are scaled up uniformly, bidirectional search achieves some search space reduction. However, it is not enough to significantly reduce running times due to the overhead of running a second search. This changes drastically when the query weight increases are applied nonuniformly in the third scenario. Here only weights with speed less than 80 kph were scaled up. This touches only the beginning and end of most shortest paths between randomly chosen vertices. The middle part of the shortest paths will typically use faster edges like highways. In this case, bidirectional (C)CH-Potentials is a factor of two faster than the unidirectional variant. This is because the search space of a unidirectional search expands greatly while exploring the end of the path to the target where the reduced weights are bad. In contrast, the bidirectional searches meet in the middle of the shortest path where the reduced weights are close to zero, thus avoiding this expansion. This is also why ALT behaves like this for all query weights. By construction, the ALT heuristic has better reduced weights for edges that lie on many shortest paths like highways. Conversely, unimportant edges have bad reduced weights. This makes bidirectional search so critical for the ALT performance. In contrast, a potential as tight as CH-Potentials makes bidirectional search in many scenarios unnecessary. Bidirectional search for CH-Potentials only pays off when the reduced weights are bad around the terminals.

6.5 Applications

Table 6 depicts the running times of (C)CH-Potentials in various applications, such as those described in Section 5.3. We report speedups compared to extensions of Dijkstra's algorithm for

each application. We start with the base case where $w_q = w_\ell$. This is the problem variant solved by the basic CH algorithm. CH achieves average query running times of 0.16 ms on OSM Ger. CH-Potentials is roughly four times slower but still achieves a speedup of 3,795 over Dijkstra. Such significant speedups are typical for CH. This shows that CH-Potentials gracefully converges toward CH in the $w_q = w_\ell$ special case. On DIMACS Europe, the average degree is somewhat higher, making the low-degree optimizations less effective and leading to more queue operations and a slightly slower running time.

In the other scenarios, the performance of CH-Potentials strongly depends on the quality of the heuristic. We measure this quality using the length increase of w_q compared to w_ℓ. Avoiding highways results in the most significant length increase and the smallest speedup. The other extreme is turn restrictions. They have little impact on the length increase. The achieved speedups are therefore comparable to CH speedups. Since turn costs and restrictions appear primarily in the beginning and end of shortest paths and not in the middle on highways, utilizing bidirectional search results in even better speedups. Mapbox live traffic has a length increase of around 15%, which yields running times of 130 ms. Applying bidirectional CH-Potentials, in this case, is in fact detrimental to the performance because the bad reduced edge weights appear in the middle of shortest paths. The same is the case for forbidden tunnels or highways. Applying CCH-Potentials customized to the current traffic (or the metric with forbidden edges) to these problems yields running times similar to the unmodified case and speedups of more than three orders of magnitude. With bidirectional CCH-Potentials, one can easily additionally support turn costs and still have running times of a few milliseconds without any adjustments to the customization.

The length increase of Mapbox traffic predictions is about 12.3%, and results in a running time of 120 ms. The speedup in the predicted scenario is larger than in the live setting, as the travel time function evaluations slow down Dijkstra's algorithm. Combining predicted and live traffic results in a running time of around 200 ms. Adding turn restrictions additionally increases the running times significantly. This increase is primarily due to the BCC optimization of Section 4.1.3 becoming ineffective when considering turns. It is not due to the length increase of using turns. With everything activated, our algorithm still has a speedup of 13.5 over the baseline. Interestingly, the PTV traffic predictions have a much smaller length increase than the Mapbox predictions. This results in somewhat faster running times of our algorithm.

Comparison with Related Work. While the query running times reported in Table 6 are decent in many settings, they are not competitive with techniques tailored to specific applications. In the simple $w_q = w_\ell$ setting, HL can be used to answer queries in less than a microsecond [24], more than three orders of magnitude faster than with CH-Potentials. Live traffic and arbitrary weight functions can be handled with CCH resulting in query times of around 0.1 ms [16]. CCH can also be adjusted to turn costs. With some algorithmic adjustments to the preprocessing [18], query times around an order of magnitude faster than CCH-Potentials are possible (0.3 ms compared to 2.1 ms with CCH-Potentials). For time-dependent routing, CATCHUp [53] achieves query times more than an order of magnitude faster than CH-Potentials (6.3 ms compared to our 97.7 ms). These numbers are not perfectly comparable due to different benchmark machines, but the overall picture is clear enough. However, the advantage of the CH-Potentials framework over these fine-tuned techniques is that it is a unified and flexible approach that can handle all of these applications without any adjustments to the preprocessing. Further, CH-Potentials preprocessing times are at least an order of magnitude faster than approaches like CATCHUp (sequentially more than 4 hours on TDEur20 compared to 5 minutes with CH-Potentials) and require significantly less memory (e.g., HL needs 20 GiB on DIMACS Europe compared to around 750 MiB with CH-Potentials, see Table 1). Finally, to the best of our knowledge, for problem settings such as the combination of

predicted and live traffic, there does not exist any exact technique to handle this setting, let alone to integrate turn costs additionally. The key achievement of CH-Potentials is that problem extensions can be integrated by trading query performance rather than developing new algorithms.

7 CONCLUSION

In this article, we introduced CH-Potentials, a fast, exact, and flexible two-phase routing framework based on A* and CH. The approach can handle complex, integrated routing scenarios with little implementation complexity and no changes to the preprocessing algorithms. CH-Potentials provides *exact* distances for lower bound weights known at preprocessing time as an A* heuristic. Thus, the query performance of CH-Potentials crucially depends on the availability of reasonable lower bounds in the preprocessing phase. Our experiments show that this availability highly depends on the application. We also show that the overhead of our heuristic is within a factor 1.6 of a hypothetical A*-heuristic that can instantly access lower bound distances. Achieving significantly faster running times could still be possible in variations of the problem setting. The core building block of our approach is Lazy RPHAST, a new CH query variant for the incremental many-to-one problem. We showed that it also delivers competitive performance for many-to-one problems. This leads to multiple avenues for future research.

Dropping the provable exactness requirement using a setup similar to anytime A* [47, 59] would be interesting. Another promising research avenue would be to investigate graphs other than road networks. Much research for grid maps exists, including a series of competitions called *GPPC* [55]. Hierarchical techniques have been shown to work well on these graphs [57]. It might also be worthwhile to apply CH-Potentials to even more routing applications, such as to the shortest ϵ-smooth path problem [26] or the problem of computing alternative routes [1, 3, 46]. Many-to-one problems also appear as subproblems in other routing algorithms, such as in nearest neighbor computations [17]. Investigating whether Lazy RPHAST can be used to improve these algorithms appears to be a worthwhile direction for future research. Studying the performance of Lazy RPHAST in a many-to-many context would also be interesting.

REFERENCES

[1] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. 2013. Alternative routes in road networks. *ACM Journal of Experimental Algorithmics* 18, 1 (2013), 1–17.

[2] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato Fonseca F. Werneck. 2012. Hierarchical hub labelings for shortest paths. In *Algorithms—ESA 2012*. Lecture Notes in Computer Science, Vol. 7501. Springer, 24–35. DOI: https://doi.org/10.1007/978-3-642-33090-2_4

[3] Roland Bader, Jonathan Dees, Robert Geisberger, and Peter Sanders. 2011. Alternative route graphs in road networks. In *Theory and Practice of Algorithms in (Computer) Systems*. Lecture Notes in Computer Science, Vol. 6595. Springer, 21–32.

[4] Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller–Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. 2016. Route planning in transportation networks. In *Algorithm Engineering—Selected Results and Surveys*. Lecture Notes in Computer Science, Vol. 9220. Springer, 19–80.

[5] Gernot Veit Batz. 2014. *Time-Dependent Route Planning with Contraction Hierarchies.* Karlsruhe Institute of Technology (KIT).

[6] Gernot Veit Batz, Daniel Delling, Peter Sanders, and Christian Vetter. 2009. Time-dependent contraction hierarchies. In *Proceedings of the 11th Workshop on Algorithm Engineering and Experiments (ALENEX'09)*. 97–105.

[7] Gernot Veit Batz, Robert Geisberger, Sabine Neubauer, and Peter Sanders. 2010. Time-dependent contraction hierarchies and approximation. In *Experimental Algorithms*. Lecture Notes in Computer Science, Vol. 6049. Springer, 166–177. http://www.springerlink.com/content/u787292691813526/.

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

[9] Reinhard Bauer, Tobias Columbus, Ignaz Rutter, and Dorothea Wagner. 2016. Search-space size in contraction hierarchies. *Theoretical Computer Science* 645 (2016), 112–127.

[10] Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. 2010. Combining hierarchical and goal-directed speed-up techniques for Dijkstra's algorithm. *ACM Journal of Experimental Algorithmics* 15, 2.3 (Jan. 2010), 1–31.

[11] Moritz Baum. 2018. *Engineering Route Planning Algorithms for Battery Electric Vehicles.* Karlsruher Institut für Technologie (KIT). DOI: https://doi.org/10.5445/IR/1000082225

[12] Moritz Baum, Julian Dibbelt, Andreas Gemsa, Dorothea Wagner, and Tobias Zündorf. 2019. Shortest feasible paths with charging stops for battery electric vehicles. *Transportation Science* 53, 6 (2019), 1501–1799.

[13] Moritz Baum, Julian Dibbelt, Thomas Pajor, Jonas Sauer, Dorothea Wagner, and Tobias Zündorf. 2020. Energy-optimal routes for battery electric vehicles. *Algorithmica* 82, 5 (2020), 1490–1546. DOI: https://doi.org/10.1007/s00453-019-00655-9

[14] Moritz Baum, Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. 2016. Dynamic time-dependent route planning in road networks with user preferences. In *Experimental Algorithms*. Lecture Notes in Computer Science, Vol. 9685. Springer, 33–49.

[15] Massimo Bono, Alfonso Emilio Gerevini, Daniel Damir Harabor, and Peter J. Stuckey. 2019. Path planning with CPD heuristics. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI'19)*. 1199–1205. DOI: https://doi.org/10.24963/ijcai.2019/167

[16] Valentin Buchhold, Peter Sanders, and Dorothea Wagner. 2019. Real-time traffic assignment using engineered customizable contraction hierarchies. *ACM Journal of Experimental Algorithmics* 24, 1 (2019), Article 2.4, 28 pages. DOI: https://doi.org/10.1145/3362693

[17] Valentin Buchhold and Dorothea Wagner. 2021. Nearest-neighbor queries in customizable contraction hierarchies and applications. In *Proceedings of the 19th International Symposium on Experimental Algorithms (SEA'21) (Leibniz International Proceedings in Informatics (LIPIcs))*, David Coudert and Emanuele Natale (Eds.), Vol. 190. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, Article 18, 18 pages. DOI: https://doi.org/10.4230/LIPIcs.SEA.2021.18

[18] Valentin Buchhold, Dorothea Wagner, Tim Zeitz, and Michael Zündorf. 2020. Customizable contraction hierarchies with turn costs. In *Proceedings of the 20th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'20) (OpenAccess Series in Informatics (OASIcs))*, Dennis Huisman and Christos Zaroliagis (Eds.). Accepted for publication.

[19] Liron Cohen, Tansel Uras, Shiva Jahangiri, Aliyah Arunasalam, Sven Koenig, and T. K. Satish Kumar. 2018. The FastMap algorithm for shortest path computations. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI'18)*. 1427–1433. DOI: https://doi.org/10.24963/ijcai.2018/198

[20] Daniel Delling. 2009. *Engineering and Augmenting Route Planning Algorithms.* Universitat Fridericiana zu Karlsruhe (TH). DOI: https://doi.org/10.5445/IR/1000011046

[21] Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck. 2013. PHAST: Hardware-accelerated shortest path trees. *Journal of Parallel and Distributed Computing* 73, 7 (2013), 940–952.

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

[23] Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. 2011. Faster batched shortest paths in road networks. In *Proceedings of the 11th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (OpenAccess Series in Informatics (OASIcs))*, Alberto Caprara and Spyros Kontogiannis (Eds.), Vol. 20. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 52–63. DOI: https://doi.org/10.4230/OASIcs.ATMOS.2011.52

[24] Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. 2013. Hub label compression. In *Experimental Algorithms*. Lecture Notes in Computer Science, Vol. 7933. Springer, 18–29. DOI: https://doi.org/10.1007/978-3-642-38527-8_4

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

[26] Daniel Delling, Dennis Schieferdecker, and Christian Sommer. 2018. Traffic-aware routing in road networks. In *Proceedings of the 34rd International Conference on Data Engineering*. IEEE, Los Alamitos, CA. https://doi.org/10.1109/ICDE.2018.00172

[27] Daniel Delling and Dorothea Wagner. 2007. Landmark-based routing in dynamic graphs. In *Experimental Algorithms*. Lecture Notes in Computer Science, Vol. 4525. Springer, 52–65.

[28] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson (Eds.). 2009. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*. DIMACS Book, Vol. 74. American Mathematical Society.

[29] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. 2015. Fast exact shortest path and distance queries on road networks with parametrized costs. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, New York, NY, Article 66, 4 pages. DOI: https://doi.org/10.1145/2820783.2820856

[30] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. 2016. Customizable contraction hierarchies. *ACM Journal of Experimental Algorithmics* 21, 1 (April 2016), Article 1.5, 49 pages.

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

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

[33] Jochen Eisner, Stefan Funke, and Sabine Storandt. 2011. Optimal route planning for electric vehicles in large networks. In *Proceedings of the 25th AAAI Conference on Artificial Intelligence (AAAI'11)*.

[34] Stefan Funke, André Nusser, and Sabine Storandt. 2014. On k-path covers and their applications. In *Proceedings of the 40th International Conference on Very Large Databases (VLDB'14)*. 893–902.

[35] Robert Geisberger, Moritz Kobitzsch, and Peter Sanders. 2010. Route planning with flexible objective functions. In *Proceedings of the 12th Workshop on Algorithm Engineering and Experiments (ALENEX'10)*. 124–137.

[36] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. 2012. Exact routing in large road networks using contraction hierarchies. *Transportation Science* 46, 3 (Aug. 2012), 388–404.

[37] Robert Geisberger and Christian Vetter. 2011. Efficient routing in road networks with turn costs. In *Experimental Algorithms*. Lecture Notes in Computer Science, Vol. 6630. Springer, 100–111.

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

[39] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. 2007. Better landmarks within reach. In *Experimental Algorithms*. Lecture Notes in Computer Science, Vol. 4525. Springer, 38–51.

[40] Andrew V. Goldberg and Renato F. Werneck. 2005. Computing point-to-point shortest paths from external memory. In *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments (ALENEX'05)*. 26–40.

[41] Lars Gottesbüren, Michael Hamann, Tim Niklas Uhl, and Dorothea Wagner. 2019. Faster and better nested dissection orders for customizable contraction hierarchies. *Algorithms* 12, 9 (2019), 1–20. DOI: https://doi.org/10.3390/a12090196

[42] Michael Hamann and Ben Strasser. 2018. Graph bisection with Pareto optimization. *ACM Journal of Experimental Algorithmics* 23, 1 (2018), Article 1.2, 34 pages. http://doi.acm.org/10.1145/3173045

[43] Peter E. Hart, Nils Nilsson, and Bertram Raphael. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* 4 (1968), 100–107.

[44] Tim Kieritz, Dennis Luxen, Peter Sanders, and Christian Vetter. 2010. Distributed time-dependent contraction hierarchies. In *Experimental Algorithms*. Lecture Notes in Computer Science, Vol. 6049. Springer, 83–93.

[45] Alexander Kleff, Frank Schulz, Jakob Wagenblatt, and Tim Zeitz. 2020. Efficient route planning with temporary driving bans, road closures, and rated parking areas. In *Proceedings of the 18th International Symposium on Experimental Algorithms (SEA'20) (Leibniz International Proceedings in Informatics)*, Simone Faro and Domenico Cantone (Eds.), Vol. 160. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, Article 17, 13 pages. https://doi.org/10.4230/LIPIcs.SEA.2020.17

[46] Moritz Helge Kobitzsch. 2015. *Alternative Route Techniques and Their Applications to the Stochastics On-Time Arrival Problem*. Karlsruhe Institute of Technology (KIT).

[47] Maxim Likhachev, Geoffrey J. Gordon, and Sebastian Thrun. 2003. ARA*: Anytime A* with provable bounds on suboptimality. In *Advances in Neural Information Processing Systems 16 (Neural Information Processing Systems, NIPS 2003, December 8–13, 2003, Vancouver and Whistler, British Columbia, Canada)*, Sebastian Thrun, Lawrence K. Saul, and Bernhard Schölkopf (Eds.). MIT Press, Cambridge, MA, 767–774.

[48] Microsoft Bing Blogs. Bing Maps New Routing Engine. Retrieved January 25, 2020 from https://blogs.bing.com/maps/2012/01/05/bing-maps-new-routing-engine/.

[49] Giacomo Nannicini, Daniel Delling, Leo Liberti, and Dominik Schultes. 2012. Bidirectional A* search on time-dependent road networks. *Networks* 59 (2012), 240–251.

[50] Ariel Orda and Raphael Rom. 1989. *Traveling Without Waiting in Time-Dependent Networks Is NP-hard*. Technical Report. Department of Electrical Engineering, Technion-Israel Institute of Technology.

[51] Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. 2002. Using multi-level graphs for timetable information in railway systems. In *Algorithm Engineering and Experiments*. Lecture Notes in Computer Science, Vol. 2409. Springer, 43–59.

[52] Ben Strasser, Daniel Harabor, and Adi Botea. 2014. Fast first-move queries through run-length encoding. In *Proceedings of the 7th Annual Symposium on Combinatorial Search (SOCS'14)*.

[53] Ben Strasser, Dorothea Wagner, and Tim Zeitz. 2021. Space-efficient, fast and exact routing in time-dependent road networks. *Algorithms* 14, 3 (Jan. 2021), 90. https://www.mdpi.com/1999-4893/14/3/90.

[54] Ben Strasser and Tim Zeitz. 2021. A fast and tight heuristic for A* in road networks. In *Proceedings of the 19th International Symposium on Experimental Algorithms (SEA'21) (Leibniz International Proceedings in Informatics (LIPIcs))*, David Coudert and Emanuele Natale (Eds.), Vol. 190. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, Article 6, 16 pages. DOI: https://doi.org/10.4230/LIPIcs.SEA.2021.6

[55] Nathan R. Sturtevant, Jason M. Traish, James R. Tulip, Tansel Uras, Sven Koenig, Ben Strasser, Adi Botea, Daniel Harabor, and Steve Rabin. 2015. The grid-based path planning competition: 2014 entries and results. In *Proceedings of the 8th Annual Symposium on Combinatorial Search (SOCS'15)*. 241.

[56] Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM Journal on Computing* 1, 2 (1972), 146–160.

[57] Tansel Uras and Sven Koenig. 2014. Identifying hierarchies for fast optimal search. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI'14)*. 878–884.

[58] Tim Zeitz. 2022. NP-hardness of shortest path problems in networks with non-FIFO time-dependent travel times. *Information Processing Letters* 179 (May 2022), 106287. https://doi.org/10.1016/j.ipl.2022.106287

[59] Rong Zhou and Eric A. Hansen. 2002. Multiple sequence alignment using anytime A*. In *Proceedings of the 18th National Conference on Artificial Intelligence and the 14th Conference on Innovative Applications of Artificial Intelligence*. 975–977.

Received 15 December 2021; revised 17 October 2022; accepted 29 October 2022