CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Dagstuhl Research Online Publication Server

Faster Batched Shortest Paths in Road Networks

Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck

Microsoft Research Silicon Valley
Mounatin View, CA, 94043, USA
{dadellin,goldberg,renatow}@microsoft.com

Abstract

We study the problem of computing batched shortest paths in road networks efficiently. Our focus is on computing paths from a single source to multiple targets (one-to-many queries). We perform a comprehensive experimental comparison of several approaches, including new ones. We conclude that a new extension of PHAST (a recent one-to-all algorithm), called RPHAST, has the best performance in most cases, often by orders of magnitude. When used to compute distance tables (many-to-many queries), RPHAST often outperforms all previous approaches.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Shortest Paths, Contraction Hierarchies, Many-to-Many, One-to-Many

Digital Object Identifier 10.4230/OASIcs.ATMOS.2011.52

1 Introduction

Motivated by web-based map services and autonomous navigation systems, the problem of finding shortest paths in road networks has received a great deal of attention recently. The main focus has been on the *point-to-point* variant of the problem: finding the best path from a single source s to a single target t. Years of research have led to very fast algorithms for this problem—see e.g. [5, 6] for overviews. Queries now need only a few memory accesses [1, 3].

Several applications, however, actually need the distances between *sets* of vertices. This has been formalized by Knopp et at. [16] as the *many-to-many* problem: given two sets of vertices, S and T, compute an $|S| \times |T|$ table with distances between them. They show that this can be solved much faster than simply running $|S| \cdot |T|$ point-to-point queries.

More recently, Delling et al. [4] considered another extended scenario: one-to-all queries. In this setting, one must compute the shortest paths from a source s to *all* other vertices in the graph. The method proposed by Delling et al. (called PHAST) can be orders of magnitude faster than Dijkstra's algorithm, and (unlike Dijkstra) can be easily parallelized.

This paper focuses on the *one-to-many* variant: given a set of targets T, compute the distances between a source s and all vertices in T. We assume the set T is given in advance and allow some extra processing to handle it. Queries (sources) then arrive in on-line fashion.

This version of the problem has several practical applications. It appears, for example, in algorithms for *path prediction*, which anticipate the trajectory of drivers using GPS locations [17, 18]. One must maintain a probability distribution over all possible destinations (typically any intersection within a metropolitan area). As the car moves, the distribution is updated accordingly. This is done under the assumption that, whatever the destination is, the driver wants to get there quickly—along a shortest path. Updating the probabilities requires computing shortest paths from the current location to all candidate destinations.

One-to-many shortest paths are also needed in *mobile opportunistic planning* [14]. At any point during a planned trip from s to t, the system must evaluate a set of potential intermediate goals (such as gas stations, coffee shops, or grocery stores) that may be suggested

© Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck;
licensed under Creative Commons License NC-ND
11th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems.
Editors: Alberto Caprara & Spyros Kontogiannis; pp. 52–63

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

to the driver. Deciding which waypoint to present depends on several factors, including the length of the modified route: compared to the original route to t, how much longer is the route that passes through the waypoint? This can be easily determined with two one-to-many computations, from s to the waypoints and from t to the waypoints (in the reverse graph).

A related application is ride sharing [11]. Here, one is given a set of offers (s, t), people driving from s to t who are willing to offer rides. When somebody searches for a ride from s' to t', it should be matched to the offer that requires the smallest detour. Taking the s'–t' path as a waypoint, this can be solved with one point-to-point and two one-to-many queries.

One-to-many queries also appear in some map matching algorithms [9]. In this case, one must find paths between clouds of points, each representing one (imprecise) GPS or cell tower reading. Assuming drivers drive efficiently, one can infer the most likely locations of a user by performing a series of shortest path computations between candidate points.

In this work, we study how existing approaches can be adapted to the one-to-many problem. More importantly, we introduce RPHAST (*restricted PHAST*), a new algorithm for this problem. An extensive experimental evaluation shows that RPHAST often yields the best running times for one-to-many computations. When the targets are close together, the speedup over existing algorithms is more than an order of magnitude. Besides being faster, RPHAST also uses less space than previous approaches. Moreover, our experiments show that RPHAST is often the best choice for solving the many-to-many problem.

This paper is organized as follows. Section 2 has background information, including notation, formal problem definitions, and related work. Section 3 is dedicated to the one-to-many problem; besides introducing our new algorithm, we discuss how existing techniques can be applied. Section 4 presents a thorough experimental evaluation of all techniques considered. Final remarks are made in Section 5.

2 Preliminaries and related work

We treat a road network as a graph $G = (V, A)$ where vertices represent intersections and arcs represent road segments. Let $|V| = n$ and $|A| = m$. Each arc $(v, w) \in A$ has a nonnegative *length* $\ell(v, w)$ representing the time to travel along the corresponding road segment.

The *many-to-many* shortest path problem takes as input the graph G, a nonempty set of *sources* $S \subseteq V$, and a nonempty set of *targets* $T \subseteq V$. Its output is an $|S| \times |T|$ table containing the distances $dist(s, t)$ from each source $s \in S$ to each target $t \in T$. Other variants of the problem are special cases. The standard *point-to-point* shortest path problem has a single source s ($S = \{s\}$) and a single target t ($T = \{t\}$). The *one-to-many* problem has a single source s, but multiple targets ($|T| \geq 1$). Finally, the *one-to-all* problem requires computing the distances from a single source to all vertices in the graph ($S = \{s\}$, $T = V$).

The remainder of this section reviews existing algorithms that are natural building blocks for the solution of the one-to-many problem: Dijkstra's algorithm [8], Contraction Hierarchies [12], Hub Labels [1], bucket-based many-to-many algorithms [16], and PHAST [4].

2.1 Dijkstra's algorithm

The standard approach to computing shortest paths in networks with nonnegative arc lengths is Dijkstra's algorithm [8]. For every vertex v, it maintains the length $d(v)$ of the shortest known path from the source s, as well as the predecessor (*parent*) $p(v)$ of v on the path. Initially $d(s) = 0$, $d(v) = \infty$ for all other vertices, and $p(v) = null$ for all v. The algorithm maintains a priority queue of *unscanned* vertices with finite d values. At each step, it removes from the queue a vertex v with minimum $d(v)$ value and *scans* it: for every arc $(v, w) \in A$

with $d(v) + \ell(v,w) < d(w)$, it sets $d(w) = d(v) + \ell(v,w)$ and $p(w) = v$. The algorithm terminates when the queue becomes empty or (for point-to-point or one-to-many queries) when all targets in T are scanned.

Dijkstra's running time is $O(m \log n)$ with binary heaps (or k-heaps [15]), and $O(m + n \log n)$ with Fibonacci heaps. When arc lengths are integers in the range $[0, C]$, the algorithm runs in $O(m + n \frac{\log C}{\log \log C})$ worst-case time with multi-level buckets [7]. A variant of multi-level buckets, smart queues [13], gives a linear expected time implementation if arc lengths are uniformly distributed. Furthermore, on many graph classes (including road networks), the smart queue implementation is no more than twice as slow as breadth-first search (BFS).

2.2 Contraction hierarchies

For point-to-point queries in road networks, several techniques can be much faster than Dijkstra (see [5, 6] for overviews). They work in two phases. The preprocessing phase, which is run offline, takes the graph as input and computes some auxiliary data. The *query phase* takes the source s and the target t as inputs, and uses the auxiliary data to speed up the computation of the shortest s–t path. We focus on *Contraction Hierarchies* (CH) [12], a two-phase algorithm that is a crucial building block for all methods we consider.

The preprocessing phase of CH picks a total order among the vertices and *shortcuts* them in this order. The *shortcut operation*, applied to a vertex v, temporarily deletes v from the graph and adds arcs between its neighbors to maintain the shortest path information. More precisely, for any pair $\{u, w\}$ of neighbors of v such that $(u, v) \cdot (v, w)$ is the only shortest u–w path in the current graph, we add a *shortcut* (u, w) with $\ell(u, w) = \ell(u, v) + \ell(v, w)$. The output of the preprocessing phase is the set A^+ of shortcut arcs, together with the position of each vertex v in the order (denoted by $rank(v)$). The algorithm is correct with any order, but query times and the size of A^+ may vary. In practice, the next vertex to shortcut is picked using on-line heuristics that try to keep the graph sparse by considering (among other factors) the number of arcs added and removed in each step [12].

The query phase of CH runs a bidirectional version of Dijkstra's algorithm on the graph $G^+ = (V, A \cup A^+)$, but only looking at *upward* arcs, i.e., those leading to neighbors with higher rank. More precisely, let $A^\uparrow = \{(v, w) \in A \cup A^+ : \; rank(v) < rank(w)\}$ and $A^\downarrow = \{(v, w) \in A \cup A^+ : \; rank(v) > rank(w)\}$. The forward search works on $G^\uparrow = (V, A^\uparrow)$, and the reverse search on $G^\downarrow = (V, A^\downarrow)$. Each vertex v maintains (possibly infinite) upper bounds $d_s(v)$ and $d_t(v)$ on its distances from s (found by the forward search) and to t (found by the reverse search). The algorithm keeps track of the vertex u minimizing $\mu = d_s(u) + d_t(u)$, and stops when the minimum value in either priority queue is at least as large as μ.

Consider the maximum-rank vertex u on the shortest s–t path. As shown by Geisberger et al. [12], u minimizes $d_s(u) + d_t(u)$ and the shortest s–t path is the concatenation of the s–u path (found by the forward search) and the u–t path (found by the backward search).

On continental road networks, CH visits only a few hundred vertices (out of tens of millions), making it four orders of magnitude faster than Dijkstra's algorithm [12].

2.3 Hub labels

Hub Labels (HL) [1] is a *labeling algorithm* [10] for the point-to-point problem. During preprocessing, it computes two labels for each vertex $v \in V$. The forward label $L_f(v)$ contains tuples $(u, d(v, u))$ (for several u), while the reverse label $L_r(v)$ contains tuples $(w, d(w, v))$ (for several w). (Here $d(x, y)$ denotes an upper bound on $dist(x, y)$.) These labels must have the *cover property*: for any pair $s, t \in V$, there is at least one vertex v (called the *hub*) in

both $L_f(s)$ and $L_r(t)$ such that $d(s, v) + d(v, t) = dist(s, t)$. An s–t query consists of simply traversing the labels and identifying the vertex v minimizing $d(s, v) + d(v, t)$.

HL uses CH to compute labels during preprocessing. $L_f(v)$ consists of all vertices scanned during an upward CH search in $G^{\uparrow}$, and $L_r(v)$ contains all vertices scanned by an upward CH search in $G^{\downarrow}$. The cover property follows from the correctness of CH.

Making HL practical on continental road networks requires many optimizations [1]. For example, removing from the labels all vertices whose distance bounds (given by the CH search) are too high reduces the average label size by 80%. One can also use *shortest path covers* (SPCs) [2] to identify the most important vertices of the graph and improve the CH order. This slows down preprocessing, but reduces the average label size to less than 100.

HL is the fastest technique for computing point-to-point shortest paths in road networks. With an efficient representation of the labels, queries need only a few memory accesses. This is orders of magnitude faster than CH. Unfortunately, preprocessing is one or two orders of magnitude more costly in terms of time (computing SPCs) and space (storing all labels).

2.4 Buckets

We now consider the computation of many-to-many shortest paths. As already mentioned, the goal is to fill an $|S| \times |T|$ *distance table* D with the distances between every source $s \in S$ and every target $t \in T$. This problem can be solved by computing $|S| \cdot |T|$ point-to-point shortest paths, but this can be wasteful. Knopp et al. [16] propose a *bucket-based approach* that extends any hierarchical speedup technique (such as CH) and can be much more efficient.

The algorithm starts by setting all entries in D to ∞ and creating an empty bucket $B(v)$ for each vertex $v \in V$. Then, for each $t \in T$, it runs a reverse CH search (an upward search in $G^{\downarrow}$) and adds a tuple $(t, d(v, t))$ to the bucket of each vertex v scanned. Finally, the algorithm fills the matrix by running a forward CH search (an upward search in $G^{\uparrow}$) from each $s \in S$. When scanning a vertex v, we process its bucket as follows. For every tuple $(t, d(v, t)) \in B(v)$, one checks if $d(s, v) + d(v, t)$ improves the value of $D(s, t)$ and updates the table accordingly. It is easy to see that the table will eventually have the correct distances.

Knopp et al. [16] observe that, in practice, it is faster to compute buckets in two steps during preprocessing. First, one runs reverse CH searches to generate an array of *triples* of the form $(v, t, d(v, t))$ (indicating v was reached during a search from t). This array is then sorted (using a standard comparison-based algorithm) according to the first element in each triple. Buckets can then be associated with contiguous segments of the array. Because sorting has good locality, it is not the bottleneck—the CH searches are more expensive.

2.4.1 HL buckets

Our experiments consider a straightforward application of the bucket method that uses HL as the underlying algorithm (instead of CH): a reverse HL label is a precomputed (and pruned) version of the reverse CH search space. Using HL leads to somewhat smaller buckets, but its main advantage is speed: building the array of triples is faster by an order of magnitude.

Sorting the array then becomes the main bottleneck. We make it faster by using bucket sort instead of a comparison-based algorithm. We cannot use the original vertex IDs as keys (bucket identifiers), however: since there are much fewer than n buckets (and elements) when T is small, most of the time would be spent visiting empty buckets. Instead, we temporarily assign (sequential) IDs to all vertices that appear in at least one of the labels scanned, ensuring that bucket sort runs in time proportional to the number of triples. With this approach, sorting the array of k triples takes only twice as much time as creating it—and is

one order of magnitude faster than using a standard $O(k \log k)$ comparison-based algorithm. While this technique could be applied to CH buckets, its effect would not be nearly as noticeable: the bottleneck is creating the array, not sorting it.

2.5 PHAST

We now discuss the one-to-all problem, in which we must find the distances from a single source s to all other vertices in the graph. At first sight, it seems one could not do much better than Dijkstra's algorithm, which is only twice as slow as a plain BFS [13]. Both Dijkstra and BFS, however, have very poor locality. Since they grow a ball of increasing radius around the source, the vertices in their working sets are usually spread over very different regions of the graph—and in memory, leading to many cache misses. Changing the graph layout in memory can help, but no single layout works well for all possible sources s.

For road networks, PHAST [4] offers a better solution. Unlike Dijkstra, it works in two phases. Preprocessing is the same as in CH: it defines a total order among the vertices and builds $G^{\uparrow}$ and $G^{\downarrow}$. A one-to-all query from s works as follows. Initially, set $d(s) = 0$ and $d(v) = \infty$ for all other $v \in V$. Then run an upward search from s in $G^{\uparrow}$(a forward CH search), updating $d(v)$ for all vertices v scanned. Finally, the *scanning phase* of the query processes all vertices in $G^{\downarrow}$ in reverse rank order (from most to least important). To process v, we check for each incoming arc $(u, v) \in A^{\downarrow}$ whether $d(u) + \ell(u, v)$ improves $d(v)$. If it does, we update the value. After all updates, $d(v)$ will represent the exact distance from s to v.

The correctness of PHAST follows from the correctness of CH [4]. Take any vertex v, and let w be the highest-ranked vertex on the shortest s–v path. PHAST finds the s–w subpath during the upward CH search from s, and the w–v subpath during the scanning phase.

The main advantage of PHAST over Dijkstra is that only the (cheap) upward CH search depends on the source s. The (more costly) scanning phase visits vertices and arcs in the same order for *any source*. Permuting vertices appropriately during preprocessing ensures the scanning phase accesses the lists of vertices and arcs sequentially, minimizing cache misses. This alone makes PHAST about 15 times faster than Dijsktra in large road networks.

Delling et al. obtain even greater speedups by computing trees from multiple sources at once (in applications that require it). To process a vertex v, the standard scanning phase updates $d(v)$ by looking at incoming arcs (u, v). Although the arc itself and $d(v)$ are accessed sequentially, reading $d(u)$ may lead to a cache miss. Keeping $k > 1$ distance labels for each vertex (one for each of k sources) allows us to amortize the cost of such a miss. When processing (u, v), we read an array of distances to u ($[d_1(u), d_2(u), \ldots, d_k(u)]$), and use it to update an array of distances to v ($[d_1(v), d_2(v), \ldots, d_k(v)]$). A cache miss will bring an entire cache line from memory; setting k appropriately (typically to 16) ensures this data is immediately useful. We can even use instruction-level parallelism (SSE instructions) to process up to four entries at once. This makes it two orders of magnitude faster than Dijkstra.

This speedup is still obtained on a single core. PHAST can easily be parallelized, and is orders of magnitude faster than Dijkstra for road networks when run on an NVIDIA GTX 580 GPU. For simplicity, however, this paper considers only single-core CPU implementations.

3 The one-to-many problem

We now study one-to-many shortest paths, the main focus of our paper. In its simplest version, this is the problem of computing the distances from a single source s to all vertices in a target set T. Given our motivating applications, however, we consider a slightly more involved scenario. We are given a fixed set of targets T in advance, and are then required to

answer multiple one-to-many queries for different sources s. Unlike in the many-to-many problem, the sources are revealed one at a time, and only after the set of targets. We therefore consider algorithms with three phases: preprocessing (same as before), target selection (run once T is known), and query (run for each s). We first consider straightforward applications of the methods presented so far, then introduce a novel approach.

3.1 Straightforward approaches

Section 2 suggests three natural approaches to the one-to-many problem.

First, one can perform a single one-to-many query (from s to T) as a series of $|T|$ point-to-point queries. For every target $t \in T$, we perform an independent s–t query. Being the fastest point-to-point algorithm, HL is the obvious candidate here. Note that there is no need for a target selection phase.

The second approach is to consider the one-to-many problem a special case of many-to-many, and use a bucket-based algorithm. The target selection phase builds the buckets from the reverse search spaces of all elements in T. The query phase looks at the forward search space from the source s, and processes the appropriate buckets. In our experiments, we test both CH and HL as the underlying methods.

The third basic approach is to consider one-to-many a special case of one-to-all. One can simply run a one-to-all algorithm from the source s to compute the distances to all vertices, then extract only the distances to vertices in T (and discard all others). If the underlying algorithm is Dijkstra's, it can stop as soon as all vertices in T are scanned, which can lead to significant speedups when s and T are restricted to a small area. In general, given its speed, one would prefer to use PHAST instead. Because it does not visit vertices in increasing order of distance, however, it is not clear how to have an early termination criterion.

3.2 Restricted PHAST

We now discuss a new algorithm that extends PHAST to handle the one-to-all scenario much more efficiently. We call this extension *restricted PHAST* (or RPHAST).

RPHAST leaves the preprocessing phase untouched: it assigns ranks to all vertices and builds the upward ($G^{\uparrow}$) and downward ($G^{\downarrow}$) graphs. Unlike PHAST, however, RPHAST has a target selection phase. Once T is known, it extracts from the contraction hierarchy only the information necessary to compute the distances from any source $s \in V$ to all targets T, creating a restricted downward graph $G_T^{\downarrow}$. RPHAST has the same query phase as PHAST, but using $G_T^{\downarrow}$ instead of $G^{\downarrow}$. It still uses $G^{\uparrow}$ for the forward searches from the source.

The challenging aspect of this algorithm is ensuring correctness: the graph built by the target selection phase must include all the information necessary to compute paths from any vertex in the graph to any vertex in T. Since the forward search is done on the full graph ($G^{\uparrow}$), we only need to ensure that $G_T^{\downarrow}$ contains the reverse search spaces of all vertices in T.

We could compute this explicitly by running a separate CH search on $G^{\downarrow}$ from each vertex in T and marking all vertices visited, but this would be slow. Instead, we perform a single search from *all vertices* in T at once. More precisely, the target selection phase builds a set T' of relevant vertices. It initializes both T' and a queue Q with T. While Q is not empty, we remove a vertex u from it and check for each downward incoming arc $(v, u) \in A^{\downarrow}$ whether $v \in T'$. If not, we add v to T' and Q. This process scans only vertices in T', and each only once. Finally, we build $G_T^{\downarrow}$ as the subgraph of $G^{\downarrow}$ induced by T'. This can also be done by scanning only the vertices in T'.

▶ **Lemma 1.** *RPHAST is correct.*

Proof. We must show that, for any pair of vertices $s \in V$ and $t \in T$, RPHAST will compute the shortest path from s to t. We claim that RPHAST will indeed find the same s–t path (in $G^+ = G^\uparrow \cup G^\downarrow$) as a standard CH query would. Let u be the highest-ranked vertex on this path. The shortest s–u path will be found by the forward CH search from s in $G^\uparrow$, which RPHAST runs during the query phase (this follows from the correctness of CH). We only need to show that the shortest u–t path in $G^\downarrow$ will be found by the scanning phase of the RPHAST query. By definition, vertices on this path have monotonically decreasing rank. Since the target selection phase of RPHAST works in this order (from most to least important arc), it will find the path. This concludes the proof. ◀

If T changes, we must rerun only the target selection phase, which is much faster than the full PHAST preprocessing. Although we only consider a single-core implementation in this paper, all acceleration techniques for PHAST [4] can be applied to RPHAST as well.

We note that Eisner et al. [9] propose a target selection phase similar to ours for performing one-to-many queries: they start a (backward) BFS in $G^\downarrow$ from all targets $t \in T$ at once, and mark all arcs scanned during this search. Queries are much different from RPHAST, however: they run a forward CH search from the source s, but allow it to go downward on marked arcs. We call this approach CH top-down (CTD). Because it is Dijkstra-based, CTD queries should be much slower than RPHAST. This is confirmed by our experiments in Section 4.

3.2.1 Full shortest paths

So far, we have assumed one only needs to compute the *distances* to T. RPHAST (and all other CH-based algorithms) could be trivially extended to maintain parent pointers, allowing easy retrieval of actual shortest paths in G^+ (which usually contain shortcuts). If the original graph edges are needed, one can use well-known *path unpacking* techniques [12] to expand the shortcuts. Since each shortcut is a concatenation of two arcs (or shortcuts), storing its "middle" vertex during preprocessing is enough for fast recursive unpacking during queries.

In certain applications, however, the set S of possible sources is known in advance—for example, when running path prediction algorithms within a single metropolitan area or state. In such cases, RPHAST does not need to keep the entire graph (and all shortcuts) in memory: we can modify its target selection phase to keep only the data needed for unpacking.

The main idea is to extend T' by all vertices that can be on shortest paths to T. We call this set T''. For simplicity, assume $S \subseteq T$ (if not, just extend T by S). The main issue here is that shortest paths between two vertices in T may actually contain vertices outside T.

We start by computing T' as in standard RPHAST. We must build the transitive shortest path hull of T, consisting of all vertices on shortest paths between all pairs $\{u, v\} \in T$. To do so, we first identify all *boundary vertices* B_T of T, i.e., all vertices in T with at least one neighbor $u \notin T$ in the original graph G. (If a shortest path ever leaves T, it must do so through a boundary vertex.) From each $b \in B_T$, we run an RPHAST query to compute all distances to T. We then mark all vertices and arcs in $G^\uparrow$ and $G^\downarrow$ that lie on a shortest path to any $t \in T$. This procedure marks the shortest path hull in G^+. We obtain T'' by unpacking all marked shortcuts and marking their internal vertices as well. This can be done by a linear top-down sweep over all marked vertices: for each vertex, we mark the middle vertex of each marked incident shortcut, as well as its two constituent arcs (or shortcuts). T'' is the set of all marked vertices at the end of this process.

The query phase performs the downward sweep on $G^\downarrow_{T''}$ (the subgraph of $G^\downarrow$ induced by T''). To query the parent vertex of a vertex $u \in T''$, we simply iterate over all incoming (original) arcs (v, u) and check whether $d(v) + \ell(v, u) = d(u)$.

Note that this approach is only practical when $S \cup T$ is a clustered set. In such cases, query times hardly increase because T'' is not much bigger than T'. The selection phase is about an order of magnitude slower than for standard RPHAST, however.

4 Experiments

We implemented all algorithms in C++ and compiled them with Microsoft Visual C++ 2010. We use smart queues [13] for Dijkstra's algorithm and binary heaps for all other methods (because the priority queues are small in such cases, they have little impact on performance).

Our evaluation was done on a dual Intel Xeon 5680 running Windows Server 2008 R2 with 96 GB of DDR3-1333 RAM. Each CPU has six cores clocked at 3.33 GHz. Preprocessing algorithms use multiple cores, but target selection and queries are sequential. Our benchmark instance is the European road network, with 18 million vertices and 42 million arcs, made available by PTV AG [19] for the 9th DIMACS Implementation Challenge [6]. To improve locality [4], we reorder the vertices of the graph in DFS order. The length of an arc represents the travel time between its endpoints. We represent lengths and distances as 32-bit integers. We do not test more road networks here due to space limitations, but note that RPHAST should work well on any input on which pure PHAST (or CH) works well [4].

We tested all algorithms mentioned in Section 3. We used the fastest single-core implementation of PHAST described by Delling et al. [4]. Our implementation of RPHAST is based on it, with the same ranking function. For HL, we used the "local" version proposed by Abraham et al. [1], as it is better suited for the application than the more complicated version optimized for long-range queries. CTD uses the same ranking function as PHAST. We implemented the bucket-based algorithm using both HL (also the "local" version) and CH as building blocks; we refer to the resulting algorithms as BHL and BCH. BCH uses the same ranking function as PHAST, and applies full stall-on-demand [20] when performing the upward searches to populate the buckets. As mentioned in Section 2.4.1, target selection uses bucket sort for BHL, but comparison-based sorting for BCH. All algorithms compute only distances, not the full shortest path descriptions.

4.1 One-to-many

In some applications of the one-to-many problem, such as finding nearby airports, the targets may be spread over the whole graph. In others, such as path prediction, the targets may be all vertices within a certain region (a city or metropolitan area). To model this, our experiments take into account both the number of targets and their distribution.

To generate our test problems, we pick a center c at random and run Dijkstra's algorithm from it until reaching a predetermined number of vertices. Let B be the set of vertices thus visited. We then pick our targets T as a random subset of B. The input parameters are $|B|$ (the size of the ball) and $|T|$ (the number of targets). Note that $|T| \leq |B|$. By varying these parameters, we can simulate the different scenarios described above.

In our first experiment, we fix the number of targets $|T|$ at 16 384 (2^{14}) and consider two different ball sizes, $|B| = |T|$ (the targets are all vertices in a contiguous region) and $|B| = 64 \cdot |T|$ (the targets are spread over a larger region). We pick sources uniformly at random from B. (Picking sources from the entire graph would hurt Dijkstra significantly, with almost no effect on the other algorithms.) For each set of parameters, we test 100 different sets of targets, each with 100 different sources.

Table 1 shows the results. For each algorithm, it first reports the total space required by the preprocessed data (including the original graph, if applicable) and the preprocessing

Table 1 Performance of various algorithms.

	preprocessing		$\lvert T\rvert = \lvert B\rvert = 2^{14}$ selection		query	$\lvert T\rvert = 2^{14}$, $\lvert B\rvert = 2^{20}$ selection		query
	space	time	space	time	time	space	time	time
algorithm	[GB]	[h:m]	[MB]	[ms]	[ms]	[MB]	[ms]	[ms]
Dijkstra	0.4	—	—	—	7.43	—	—	457.70
HL	20.1	2:39	—	—	6.27	—	—	6.94
BCH	0.4	0:05	15.65	943.3	1.72	15.44	991.0	2.81
BHL	20.1	2:39	11.21	50.9	1.40	11.35	80.0	1.85
CTD	0.4	0:05	0.43	2.4	2.39	3.36	37.3	23.95
PHAST	0.4	0:05	—	—	136.92	—	—	136.92
RPHAST	0.4	0:05	0.43	1.8	0.17	3.36	27.5	1.02

time (using all cores). Then, for each value of $|B|$, it shows the total target selection space (the amount of additional data it generates), the target selection time, and the average query time. Selection and query times for a wider range of $|B|$ are also shown in Figure 1.

The running time of Dijkstra's algorithm is proportional to $|B|$; it performs reasonably when $|B| = |T|$, but poorly when $|B| \gg |T|$. For $|B| = 2^{20}$ it is slower than PHAST, which solves the one-to-all problem.

RPHAST improves on PHAST by restricting the search space based on the target set. The improvement is bigger when the targets are close together, since there is less intersection between their search spaces. Changing $|B|$ from 2^{14} to 2^{20} increases the selection space by a factor of 7.3, from 0.43 MB to 3.36 MB. This corresponds to an increase in the average number of vertices in $G_T^{\downarrow}$ from 18 009 to 117 419. Query times increase slightly less (because the cost of the forward CH search does not depend on $|B|$), but selection times increase more (due to worse locality). Even for $|B| = 2^{20}$, however, RPHAST remains the fastest algorithm.

CTD and RPHAST share the same target selection phase, and work on the same graph during queries. CTD queries are much slower than RPHAST queries, however, due to poor locality: RPHAST performs a linear sweep, while CTD must run a Dijkstra-based search. For CTD, queries take about as much time as target selection, since both phases process about the same amount of data with similar access patterns.

HL and the bucket-based algorithms become only slightly slower as $|B|$ gets larger, mostly due to worse locality. The total number of elements in all buckets (and therefore the number of operations) does not depend on $|B|$. BHL has slightly smaller buckets than BCH, since HL labels correspond to smaller search spaces. Selection times do increase slightly with $|B|$, however, since elements are spread over more buckets. Because of its better initial locality, BHL selection is more affected, but is always an order of magnitude faster than BCH (thanks in part to the acceleration proposed in Section 2.4.1). Preprocessing, however, is much more expensive (in terms of both time and space) for HL than for CH.

The bucket-based methods have comparable query performance, with a slight advantage to BHL due to smaller buckets. For large $|B|$, however, BCH can be faster because it assigns vertex IDs in a more cache-friendly way.

HL queries are less than five times slower than the bucket-based approaches, which is surprisingly competitive for a simple application of a point-to-point algorithm. The fact that HL is slower than BCH is consistent with the findings of Geisberger et al. [12], who considered label-like structures to compute many-to-many queries, but ended up using buckets instead.

Figure 1 shows the results of varying $|B|$ with $|T|$ fixed at 2^{14}. When $|B|$ is small,

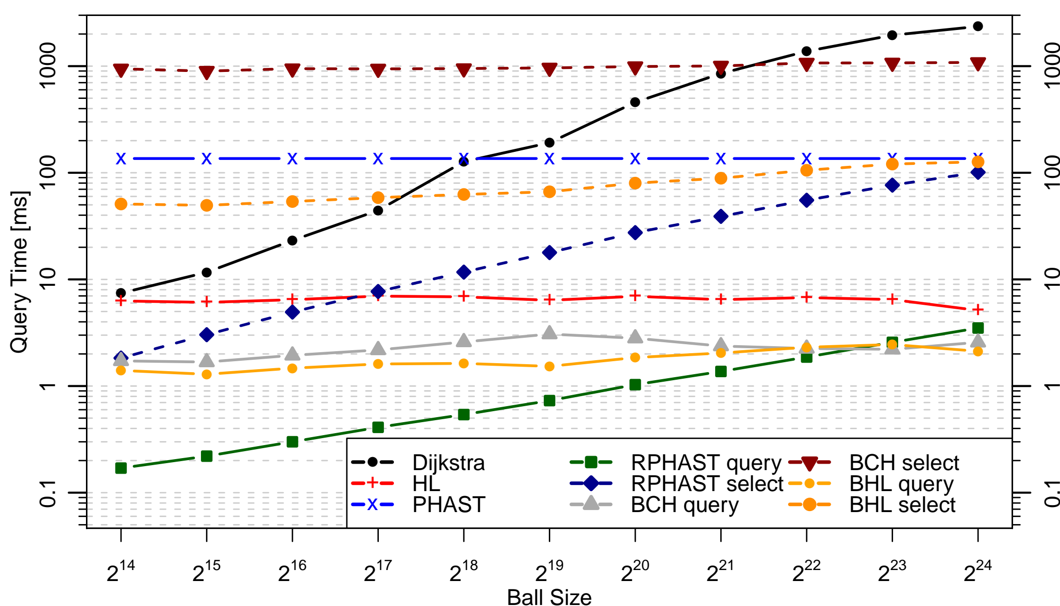

Figure 1 Running times of one-to-many algorithms with $|T| = 2^{14}$ targets picked from a ball of varying size $|B|$.

RPHAST outperforms other algorithms by an order of magnitude or more. (We omit CTD for clarity: both selection and query times would closely follow the RPHAST selection curve.) As $|B|$ increases, bucket-based algorithms become more competitive. The figure shows that, for $|T| = 2^{14}$, BHL and BCH become faster than RPHAST when $|B|$ is between 2^{22} and 2^{23}. The crossover point depends on $|T|$. Additional experiments show that it happens at $|B| \approx 2^{14}$ for $|T| = 2^{10}$, and not at all for $|T| = 2^{18}$, when RPHAST queries are always faster. Similarly, BHL can have faster target selection than RPHAST when $|T|$ is small and $|B|$ is large. Target selection is always faster for RPHAST than BCH, however.

4.2 Many-to-many

In light of these new results, we now reexamine the many-to-many problem. Recall that its input is a set S of sources and a set T of targets, and the goal is to compute an $|S| \times |T|$ table of distances between them. To solve this problem with one-to-many algorithms, we run the target selection phase on T followed by one-to-many queries from each vertex in S. In fact, this is exactly what the original many-to-many bucket-based algorithms do [16].

As in the previous experiment, we grow Dijkstra balls B from random points, and pick both S and T from B. For simplicity, we consider only the symmetric case, where $|S| = |T|$.

Figure 2 shows the running times for $|S| = |T| = 2^{14}$ and varying $|B|$. For all algorithms, running times include *both* target selection and the subsequent one-to-many queries. It excludes preprocessing times, which are the same as in Table 1. Each point is the average of 100 balls, each with a single $\{S, T\}$ pair. Note that we use a linear scale in the vertical axis (not logarithmic, as in Figure 1). The plot contains all one-to-many algorithms studied in the previous section, except Dijkstra, CTD, and PHAST, which are much worse. In addition, it includes results for SSE RPHAST, a variant of RPHAST that processes 16 trees at once and uses SSE instructions, as described in Section 2.5. Note that, like all other algorithms tested, RPHAST still uses a single CPU core.

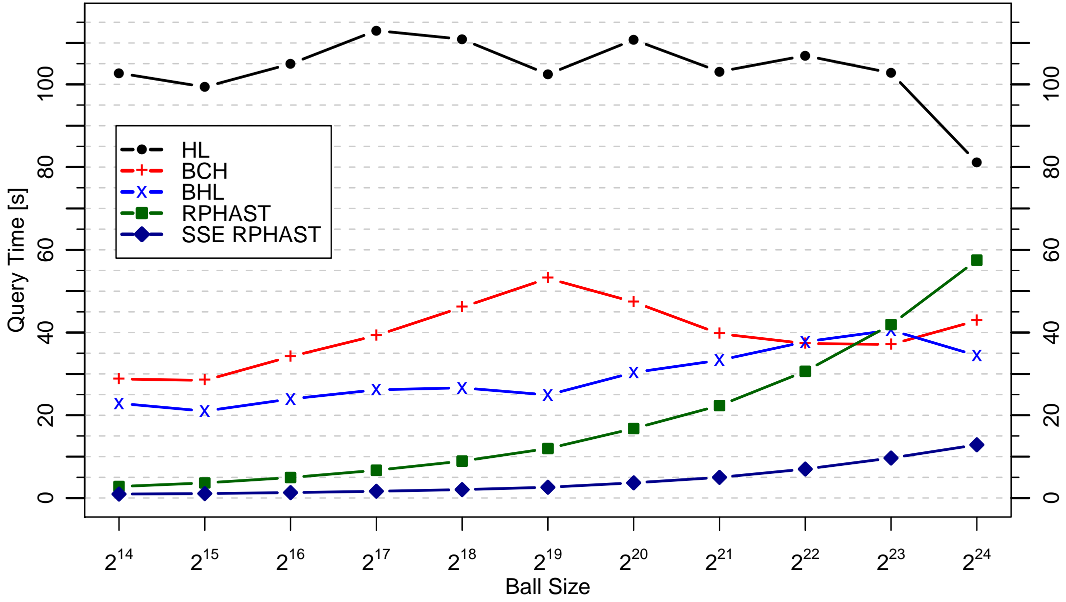

Figure 2 Running times of many-to-many algorithms with $|S| = |T| = 2^{14}$ picked from a ball of varying size $|B|$.

Because $|T|$ is rather large in the experiment, one-to-many queries are collectively much more expensive than target selection. As a result, the relative query times of HL, BCH, BHL, and RPHAST do not change much. HL is somewhat slower than the bucket-based methods; RPHAST is an order of magnitude faster when the sources are concentrated, but eventually becomes slower. Better cache utilization and instruction-level parallelism make SSE RPHAST about five times faster than RPHAST, and always faster than the bucket-based algorithms. If $|B| = |T|$, SSE RPHAST (0.95 ms) is 30 times faster than BCH (28.76 ms), the best previously known algorithm for this problem.

Of course, varying $|T|$ may change the relative order a bit. In particular, with $|T| = 2^{10}$, BHL eventually becomes faster than SSE RPHAST (for $|B| \approx 2^{22}$), but only slightly. In contrast, BCH never catches up, due to its much slower target selection phase.

5 Conclusion

We have studied the problem of computing one-to-many shortest paths on road networks with travel times. We observed that a new algorithm, RPHAST, can answer queries an order of magnitude faster than any previous solution when targets are numerous or concentrated in a restricted area. When targets are few and spread out, existing many-to-many algorithms can be faster, as long as there are enough queries to amortize their much higher cost of target selection. Because target selection is much cheaper than in previous solutions, our algorithm is particularly useful for applications such as path prediction, where there are many targets but relatively few queries (sources). When applied to the many-to-many problem, RPHAST usually outperforms existing bucket-based solutions, often by a significant margin.

Acknowledgements. We thank John Krumm for pointing us to applications of the one-to-many problem, and the anonymous referees for suggestions that helped improve the paper.

References

1. I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. A Hub-Based Labeling Algorithm for Shortest Paths on Road Networks. In *SEA*, LNCS 6630, pp. 230–241. Springer, 2011.
2. I. Abraham, A. Fiat, A. V. Goldberg, and R. F. Werneck. Highway Dimension, Shortest Paths, and Provably Efficient Algorithms. In *SODA*, pp. 782–793. SIAM, 2010.
3. H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes. In Transit to Constant Shortest-Path Queries in Road Networks. In *ALENEX*, pp. 46–59. SIAM, 2007.
4. D. Delling, A. V. Goldberg, A. Nowatzyk, and R. F. Werneck. PHAST: Hardware-Accelerated Shortest Path Trees. In *IPDPS*. IEEE Computer Society, 2011.
5. D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering Route Planning Algorithms. In *Algorithmics of Large and Complex Networks*, LNCS 5515, pp. 117–139. Springer, 2009.
6. C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, DIMACS Book 74. AMS, 2009.
7. E. V. Denardo and B. L. Fox. Shortest-Route Methods: 1. Reaching, Pruning, and Buckets. *Operations Research*, 27(1):161–186, 1979.
8. E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.
9. J. Eisner, S. Funke, A. Herbsty, A. Spillnery, and S. Storandt. Algorithms for Matching and Predicting Trajectories. In *ALENEX*, pp. 84–95. SIAM, 2011.
10. C. Gavoille, D. Peleg, S. Pérennes, and R. Raz. Distance Labeling in Graphs. *J. Algorithms*, 53(1):85–112, 2004.
11. R. Geisberger, D. Luxen, P. Sanders, S. Neubauer, and L. Volker. Fast Detour Computation for Ride Sharing. In *ATMOS*, OASIcs, 2010.
12. R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *WEA*, LNCS 5038, pp. 319–333. Springer, 2008.
13. A. V. Goldberg. A Practical Shortest Path Algorithm with Linear Expected Time. *SIAM Journal on Computing*, 37:1637–1655, 2008.
14. E. Horvitz, P. Koch, and M. Subramani. Mobile Opportunistic Planning: Methods and Models. In *User Modeling*, pp. 238–247, 2007.
15. D. B. Johnson. Efficient Algorithms for Shortest Paths in Sparse Networks. *Journal of the ACM*, 24(1):1–13, 1977.
16. S. Knopp, P. Sanders, D. Schultes, F. Schulz, and D. Wagner. Computing Many-to-Many Shortest Paths Using Highway Hierarchies. In *ALENEX*, pp. 36–45. SIAM, 2007.
17. J. Krumm and E. Horvitz. Predestination: Inferring destinations from partial trajectories. In *UbiComp*, pp. 243–260, 2006.
18. J. Krumm and E. Horvitz. Predestination: Where Do You Want to Go Today? *IEEE Computer Magazine*, 40(4):105–107, 2007.
19. PTV AG - Planung Transport Verkehr. `http://www.ptv.de`.
20. D. Schultes and P. Sanders. Dynamic Highway-Node Routing. In *WEA*, LNCS 4525, pp. 66–79. Springer, 2007.