A Hub-Based Labeling Algorithm for Shortest Paths in Road Networks

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

Microsoft Research Silicon Valley
{ittaia,dadellin,goldberg,renatow}@microsoft.com

Abstract. Abraham et al. [SODA 2010] have recently presented a theoretical analysis of several practical point-to-point shortest path algorithms based on modeling road networks as graphs with low highway dimension. They also analyze a *labeling algorithm*. While no practical implementation of this algorithm existed, it has the best time bounds. This paper describes an implementation of the labeling algorithm that is faster than any existing method on continental road networks.

1 Introduction

Motivated by computing driving directions, the problem of finding point-to-point shortest paths in road networks has received significant attention in recent years. Even though Dijkstra's algorithm [11] solves it in almost linear time [15], continent-sized road networks require something faster. Preprocessing makes sublinear-time algorithms possible; see [5] for a survey of existing methods.

In particular, goal-directed methods, such as *arc flags* (AF) [16], direct the search towards the target. Hierarchical methods, such as *contraction hierarchies* (CH) [14], sparsify the search space by visiting only important vertices when far from the source or target. *Transit node routing* (TNR) [3, 4] reduces long-range queries to a few table lookups, using the fact that on road networks a small set of vertices is enough to hit all long shortest paths out of a region. TNR+AF [5] (combining TNR, CH, and arc flags) is the fastest algorithm for random queries, six orders of magnitude faster than Dijkstra. For local and mid-range queries, CH and High-Performance Multi-Level Routing (HPML) [9] are the fastest.

Although algorithms such as these are known to work well in practice, a theoretical analysis has been given only recently, by Abraham et al. [2]. The method with the best time bounds is a *labeling algorithm*. Labeling algorithms have been studied before in more theoretical settings [6, 12, 21].

The preprocessing stage of the labeling algorithm computes, for each vertex v, a *forward label* $L_f(v)$ and a *reverse label* $L_r(v)$. Each consists of a set of vertices w, together with their respective distances from (in $L_f(v)$) or to (in $L_r(v)$) v. A labeling is *valid* if it has the *cover property*: for every pair of vertices s and t, $L_f(s) \cap L_r(t)$ contains a vertex u on a shortest path from s to t. An s–t query finds the vertex $u \in L_f(s) \cap L_r(t)$ that minimizes $\text{dist}(s,u) + \text{dist}(u,t)$ and

returns the corresponding path. Intuitively, a label for v is a set of *hubs* to which v has a direct connection, and any two vertices s and t share at least one hub on the shortest s–t path. Although efficient in theory, the algorithm as described by Abraham et al. [2] is impractical for continent-sized road networks: preprocessing would be too slow, and the worst-case memory usage is prohibitive.

Motivated by theory, we develop HL (Hub-based Labeling algorithm), a practical implementation of the labeling algorithm for road networks. We start from the fact that the sets of vertices visited by the forward and reverse searches of hierarchical algorithms (such as CH) contain the corresponding labels. Similar observations have been made implicitly for graphs of bounded tree-width [12] and road networks [17, 14]; we make it explicit and take advantage of it. We then propose several techniques to make our method truly practical. First, we show how to obtain much smaller labels by efficiently pruning the CH search space and applying ideas from the theoretical preprocessing algorithm [2]. Second, we describe how to compress each label. Finally, we show how to implement preprocessing and queries efficiently.

Our main contribution is to show that the labeling algorithm is practical. In fact, our experiments show that HL is currently the fastest algorithm for the problem. When optimized for speed, it answers a random query in as much time as five random accesses to main memory. This is faster than TNR+AF by a factor of more than three, and than HPML by more than an order of magnitude. For local queries, HL is about three times faster than HPML and an order of magnitude faster than TNR+AF. Using compression, we obtain a version of HL with a memory footprint that is comparable to the other two algorithms, but is still faster for all types of queries.

This paper is organized as follows. Section 2 reviews relevant previous work and describes our experimental setup. Section 3 presents the basic version of HL. Section 4 describes several improvements that make it truly practical. Section 5 compares HL with other algorithms experimentally. We conclude in Section 6. The full version of this paper [1] contains details omitted due to space limitations.

2 Preliminaries

The preprocessing stage of a point-to-point shortest path algorithm takes a graph $G = (V, A)$ as input, with $|V| = n$, $|A| = m$, and length $\ell(a) > 0$ for each arc a. The length of a path P in G is the sum of its arc lengths. The query stage takes a source s and a target t as input and returns the distance $\text{dist}(s, t)$ between them.

Dijkstra's algorithm. The standard solution to this problem is Dijkstra's algorithm [11], which processes vertices in increasing order of distance from s. For every vertex v, it maintains the length $d(v)$ of the shortest s–v path found so far, as well as the predecessor $p(v)$ of v on the path. Initially $d(s) = 0$, $d(v) = \infty$ for all other vertices, and $p(v) = \textit{null}$ for all v. At each step, a vertex v with minimum $d(v)$ value is extracted from a priority queue and *scanned*: for each arc $(v, w) \in A$, if $d(v) + \ell(v, w) < d(w)$, we set $d(w) = d(v) + \ell(v, w)$ and $p(v) = w$. The algorithm terminates when the target t is extracted.

Contraction hierarchies. Preprocessing enables much faster exact queries on road networks. The *contraction hierarchies* (CH) algorithm [14], in particular, is based on the notion of *shortcuts* [19]. The *shortcut operation* deletes (temporarily) a vertex v from the graph; then, for any neighbors u, w such that $(u, v) \cdot (v, w)$ is the only shortest path between u and w, it adds a *shortcut arc* (u, w) with $\ell(u, w) = \ell(u, v) + \ell(v, w)$, thus preserving the shortest path information.

The CH preprocessing routine defines a total order among the vertices and shortcuts them sequentially in this order, until a single vertex remains. It outputs a graph $G^+ = (V, A \cup A^+)$ (where A^+ is the set of shortcut arcs created), as well as the vertex order itself. We denote the position of a vertex v in the order by $rank(v)$. Define $G^\uparrow = (V, A^\uparrow)$ by $A^\uparrow = \{(v, w) \in A \cup A^+ : rank(v) < rank(w)\}$. Similarly, $A^\downarrow = \{(v, w) \in A \cup A^+ : rank(v) > rank(w)\}$ and $G^\downarrow = (V, A \cup A^\downarrow)$.

During an s–t query, the *forward CH search* runs Dijkstra from s in $G^\uparrow$, and the *reverse CH search* runs reverse Dijkstra from t in $G^\downarrow$. For every $v \in V$, these searches lead to upper bounds $d_s(v)$ and $d_t(v)$ on distances from s to v and from v to t. For some vertices, these estimates may be greater than the actual distances (and even infinite for unvisited vertices). However, as shown by Geisberger et al. [14], the maximum-rank vertex u on the shortest s–t path is guaranteed to be visited, and $v = u$ will minimize $d_s(v) + d_t(v) = \text{dist}(s, t)$.

Queries are correct regardless of the contraction order, but query times and the number of shortcuts added may vary greatly. For best results, on-line heuristics are used to select which vertex to shortcut next [14]. Our implementation [7] sets the priority of a vertex u to $2ED(u) + CN(u) + H(u) + 5L(u)$, where $ED(u)$ is the difference between the number of arcs added and removed (if u were shortcut), $CN(u)$ is the number of previously contracted neighbors, $H(u)$ is the number of arcs represented by the shortcuts added, and $L(u)$ is the *level* u would be assigned to. We define $L(u)$ as $L(v) + 1$, where v is the highest-level vertex among all lower-ranked neighbors of u in G^+; if there is no such v, $L(u) = 0$.

Labeling algorithm. The preprocessing of the theoretical labeling algorithm of Abraham et al. [2] is based on *shortest path covers* (SPCs). Intuitively, an (r, k)-SPC S is a set of vertices that (1) hits every shortest path of length between r and $2r$ and (2) is *sparse*, in the sense that every ball of radius $2r$ has at most k elements from S. For a fixed parameter h (the *highway dimension* of the graph), (r, h)-SPCs exist for all r, and the greedy algorithm finds an $O(r, O(h \log n))$-SPC. The value of h is believed to be small for road networks.

The preprocessing routine computes greedy SPCs C_i for $r = 2^i$, $0 \leq i \leq \log D$, where D is the graph diameter. For each v, it takes as a label the union over i of C_i intersected with the ball of radius $2 \cdot 2^i$ around v. As stated, the algorithm is impractical for continent-sized road networks. Greedy SPCs require many all-pairs shortest paths computations, which would take months. Furthermore, the theoretical bound on the label size ($O(k \log n \log D)$) could be in the thousands, leading to unrealistic space requirements and uncompetitive queries.

Experimental setup. Since we use actual measurements to justify our design decisions, we describe our experimental setup in advance. We implemented our

algorithm in C++ and compiled it with Microsoft Visual C++ 2010. We ran our tests on a machine with two Intel Xeon X5680 processors and 96 GB of DDR3-1333 RAM, running Windows 2008R2 Server. Each CPU has 6 cores clocked at 3.33 GHz, 6 x 64 kB L1, 6 x 256 kB L2, and 12 MB L3 cache. Preprocessing is parallelized (with OpenMP), but queries are sequential and pinned to one core.

In most experiments we report the (parallel) preprocessing time (excluding the CH preprocessing) and total space consumption in GB. For most of the paper, we evaluate queries by running 100 000 000 s–t queries (with s and t picked uniformly at random in advance) and reporting the average time. We focus on computing the *length* of shortest paths; for full path descriptions, one could apply the expansion techniques used for TNR [4], for example.

We use two input graphs taken from the 9th DIMACS Implementation Challenge [10]. The *Europe* instance, representing Western Europe, has 18 million vertices and 44 million arcs. The *USA* road network has 24 million vertices and 58 million arcs. In both cases, arc costs are 32-bit integers representing travel times. Unless otherwise mentioned, we use the Europe instance as default.

3 HL Overview

Preprocessing. Geisberger et al. [14] suggest implementing many-to-many queries by precomputing and storing the sets of vertices of the forward CH searches for a set of sources and of the reverse CH searches for a set of targets, along with the corresponding distance estimates. A query from a source to a target is done by intersecting the corresponding sets. They have not pursued this approach for point-to-point queries, probably because it looked impractical. Indeed, our sampling-based estimates for Europe show that one would need about 154 GB to store all labels (whose average size is 536). The time estimates are encouraging, however: 321 seconds to compute all labels and 3 μs for queries. To make the algorithm truly practical, however, we need several additional ingredients.

In particular, the sets visited by CH are not *strict labels*: a bound $d(w)$ stored within a label for v may actually be greater than $\text{dist}(v, w)$. As Section 4 will show, we can efficiently *prune* each label by eliminating entries with wrong distance estimates. A simple heuristic (based on stall-on-demand [14]) reduces the label size to about 133, which is already much more practical. As Section 4 will show, we can go further and remove *all* vertices whose distance estimates are not tight, making the labels strict. By combining this with ideas from the theoretical algorithm [2], we achieve labels with fewer than 85 entries on average.

Query. We now consider how to represent labels to allow efficient queries. We describe the L_f labels; the L_r labels are symmetric. A forward label $L_f(v)$ is represented as the concatenation of three elements: (1) a 32-bit integer N_v representing the number of vertices in the label; (2) a zero-based array I_v with the (32-bit) IDs of all vertices in the label, in ascending order; and (3) an array D_v with the (32-bit) distances from v to each vertex in the label. Note that vertices appear in the same order in I_v and D_v: $D_v[i] = \text{dist}(v, I_v[i])$.

Given s and t, the query algorithm must pick, among all vertices $w \in L_f(s) \cap L_r(t)$, the one minimizing $d_s(w) + d_t(w) = \text{dist}(s, w) + \text{dist}(w, t)$. Because the I_v arrays are sorted, this can be done with a single sweep through the labels, similar to mergesort. We maintain array indices i_s and i_t (initially zero) and a tentative distance μ (initially infinite). At each step we compare $I_s[i_s]$ and $I_t[i_t]$. If these IDs are equal, we found a new w in the intersection of the labels, so we compute a new tentative distance $D_s[i_s] + D_t[i_t]$, update μ if necessary, then increment both i_s and i_t. If the IDs differ, we increment either i_s (if $I_s[i_s] < I_t[i_t]$) or i_t (if $I_s[i_s] > I_t[i_t]$). We stop when either $i_s = N_s$ or $i_t = N_t$, and return μ.

Low-level details. Implementation details are important because the fastest version of our query is less than five times slower than a random memory access.

A key aspect of the algorithm is that it accesses each array sequentially, thus minimizing the number of cache misses. Avoiding cache misses is also the motivation for having I_v and D_v as separate arrays: while we must access almost all IDs in a label, distances are only needed when IDs match. We also align each label to a cache line, which has 64 bytes in our machine.

Another practical improvement is to use the highest-ranked vertex as a sentinel by assigning ID n to it. Because this vertex must belong to all labels, it will lead to a match in every query; it therefore suffices to test for termination only after a match. In addition, we store the distance to the sentinel at the beginning of the label; this enables us to obtain a quick upper bound on the s–t distance.

We forced procedure inlining whenever appropriate (a function call takes about 150 ns, roughly the time of 3 memory accesses), and prefetch data to the L1 cache whenever appropriate. Finally, we use pointer arithmetic (instead of maintaining indices) to traverse the labels during queries.

4 Efficient HL Implementation

This section introduces techniques that make HL efficient by reducing the average label size, speeding up long-distance queries, and using compression. We also describe several lower-level improvements.

4.1 Label pruning

We can use a fast heuristic modification (similar to *stall-on-demand* [20]) to the CH search to identify most vertices with incorrect distance bounds. Suppose we are performing a forward CH search (the reverse case is similar) from v and we are about to scan w, with distance bound $d(w)$. We examine all incoming arcs $(u, w) \in A^{\downarrow}$. If $d(w) > d(u) + \ell(u, w)$, then $d(w)$ is provably incorrect. We can safely remove w from the label, and we do not scan its outgoing arcs. This technique significantly decreases the average label size (to 133.0) and query time (to 937 ns).

We use *bootstrapping* (i.e., HL itself) to prune the labels further. We compute labels in descending level order. Suppose we have just computed the partially

pruned label $L_f(v)$. We know that $d(v) = 0$ and that all other vertices w in $L_f(v)$ have higher level than v, which means $L_r(w)$ must have already been computed. We can therefore compute $\text{dist}(v, w)$ by running a v–w HL query, using $L_f(v)$ itself and the precomputed label $L_r(w)$. We remove w from $L_f(v)$ if $d(w) > \text{dist}(v, w)$. Bootstrapping reduces the average label size to 109.6 (30.6 GB in total), and improves average queries to 812 ns. Preprocessing is slightly slower, at 580 s. The resulting labeling algorithm is strict and practical, but substantial further improvements are possible.

Note that, without bootstrapping, labels can be trivially computed in parallel, since they are independent. Bootstrapping requires greater care. We can process vertices of the same level in parallel, but must synchronize after each level, since computing the label of a level-i vertex requires access to labels at higher levels. Fortunately, road networks have only about 150 levels [7].

4.2 Label Ordering

We can assign new *internal IDs* to the vertices to change the order in which they appear in the labels; this may speed up queries or improve compression rates.

For most vertices, keeping the original *input* order seems to be a good idea. Rearranging vertices by rank or level (either ascending or descending) actually increases query times on Europe from 812 ns to more than 1100 ns. This happens because nearby vertices in the graph tend to have similar original IDs. During an s–t query, a large portion of the corresponding labels represents vertices in small regions around s and t; it is often the case that all vertex IDs in one region are larger than all IDs in the other. As a result, the query algorithm may reach the end of one label (thus stopping the search) while visiting a fraction of the other. Rearranging vertices destroys this locality and decreases query performance.

For faster queries, it is often better to keep the input order for all but the topmost (highest-ranked) k vertices, which are assigned internal IDs from 0 to $k - 1$. In particular, the *top k input* order (in which the input order among the top k vertices is preserved), achieves query times of 769 ns with $k = 256$. The *top k level* order (which sorts the top k vertices by level), is slightly worse: query times are about the same as keeping the original input order (for $k = 256$). Unless otherwise stated, we use the top 256 input order.

As already mentioned, one optimization we apply to all label orderings is to assign ID $n = |V|$ to the highest-ranked vertex, which is used as a sentinel.

4.3 Shortest Path Covers

The CH preprocessing algorithm tends to contract the least important vertices (those on few shortest paths) first, and the more important vertices (those on more shortest paths) later. The heuristic used to choose the next vertex to contract works poorly near the end of preprocessing, when it must order important vertices relative to one another. This has been observed before [14]: a variant of TNR based on CH yielded worse locality filters than previous versions. We use shortest path covers to improve the ordering of important vertices. We do this

near the end of CH preprocessing, when most vertices have been contracted, the graph is small, and the greedy SPC algorithm becomes feasible.

More precisely, we start by running the CH preprocessing with our original selection rule, but pause it as soon as the remaining graph G_t has only t vertices left (we use $t = 25\,000$). We then run a greedy algorithm to find a set C of good cover vertices, i.e., vertices that hit a large fraction of all shortest paths of G_t, with $|C| < t$ (we use $|C| = 2048$). Starting with $C = \emptyset$, at each step we add to C the vertex v that hits the most uncovered (by C) shortest paths in G_t. After C is computed, we continue the CH preprocessing, but forbid the contraction of the vertices in C until they are the only ones left. This ensures the top $|C|$ vertices of the hierarchy will be exactly those in C, which are then contracted in reverse greedy order (i.e., the first vertex found by the greedy algorithm is the last one remaining).

Setting $t = 25\,000$ and $|C| = 2048$ decreases the average label size on Europe by about 20%, from 109.62 to 84.74. Query times are reduced accordingly, from 769 ns to 594 ns. Given our emphasis on query times, we use the SPC-augmented preprocessing with these parameters as default. The time to build the hierarchy increases from 3 minutes to 151 minutes, however. If this is an issue, a good compromise is to use $t = 10\,000$ and $|C| = 512$: preprocessing takes only 25 minutes, but queries are almost as fast (598 ns) and labels almost as small (85.79 entries) as with the original parameters.

4.4 Label Compression

Even after reducing the average label size from 536 to 85, we still need 23.9 GB to store all labels if we represent every vertex ID and distance as a separate 32-bit integer. For low-ID vertices, we can use an *8/24 compression* scheme: we represent each of the first 256 vertices as a single 32-bit word, with 8 bits allocated to the ID and 24 bits to the distance. (This could obviously be generalized for different numbers of bits.) For effectiveness, it pays to reorder vertices so that the important ones (which appear in most labels) have the lowest IDs. With top 256 input ordering, the space usage decreases from 23.9 GB to 20.1 GB. Because of better locality, queries also improve, from 594 ns to 572 ns.

Another compression technique we considered exploits the fact that the forward (or reverse) CH trees of two nearby vertices in a road network are different near the roots, but are often the same when sufficiently away from them, where the most important vertices appear. By reordering vertices in reverse rank order, for example, the labels of nearby vertices will often share long common prefixes, with the same sets of vertices (but usually different distances). Our compression scheme computes a dictionary of the common label prefixes and reuses them.

Given a parameter k, the *k-prefix compression* scheme decomposes each forward label $L_f(v)$ (reverse labels are similar) into a *prefix* $P_k(v)$ (with the vertices with internal ID lower than k) and a *suffix* $S_k(v)$ (with the remaining vertices).

Take the forward (pruned) CH search tree T_v from v: $S_k(v)$ induces a subtree containing v (unless $S_k(v)$ is empty), and $P_k(v)$ induces a forest F. The *base* $b(w)$ of a vertex $w \in P_k(v)$ is the parent of the root of w's tree in F; by definition,

$b(w) \in S_k(v)$. (If $S_k(v)$ is empty, let $b(v) = v$.) Each prefix $P_k(v)$ is represented as a list of triples $(w, \delta(w), \pi(w))$, where $\delta(w)$ is the distance between $b(w)$ and w, and $\pi(w)$ is the position of $b(w)$ in $S_k(v)$. Two prefixes are *equal* only if they consist of the exact same triples. We build a dictionary (an array) consisting of all *distinct* prefixes. Each triple uses 64 consecutive bits: 32 for the ID, 24 for $\delta(\cdot)$, and 8 for $\pi(\cdot)$. A forward label $L_f(v)$ has three elements: the position of its prefix $P_k(v)$ in the dictionary, the number of vertices in the suffix $S_k(v)$, and $S_k(v)$ itself (represented as before). To save space, labels are not cache-aligned.

During a query from v, suppose w is in $P_k(v)$. We have $\text{dist}(b(w), w) = \delta(w)$ and we know the position $\pi(w)$ of $b(w)$ in $S_k(v)$, where $\text{dist}(v, b(w))$ is stored explicitly. We can therefore compute $\text{dist}(v, w) = \text{dist}(v, b(w)) + \text{dist}(b(w), w)$.

On Europe, this approach reduces the space usage from 20.1 GB to 8.0 GB (with $k = 2^{16}$), for the price of a slightly longer preprocessing (502 s instead of 489 s). At 1172 ns, queries become about twice as slow.

To save even more, we use a *flexible prefix compression* scheme. Instead of using the same threshold k for all labels, it may split each label L in two arbitrarily. As before, common prefixes are represented once and shared among labels. Deciding which prefixes to keep is no longer straightforward. To minimize the total space usage, including all n suffixes and the (up to n) prefixes we actually keep, we model this as a *facility location* [18] problem. Each label is a *customer* that must be represented (served) by a suitable prefix (facility). The opening cost of a facility is the size of the corresponding prefix. The cost of serving a customer L by a prefix P is the size of the corresponding suffix ($|L| - |P|$). Each label L is served by the available prefix that minimizes the service cost. We use local search [18] to find a good heuristic solution.

The flexible approach reduces the space usage to 5.6 GB with the same query time (1170 ns), but the preprocessing time increases from 502 s to 2002 s.

4.5 Partition Oracle

We now describe an acceleration technique for long-range HL queries. If the source and the target are far apart, the HL searches tend to meet at very important (high-rank) vertices. If we rearrange the labels such that more important vertices appear before less important ones, long-range queries can stop traversing the labels when sufficiently unimportant vertices are reached.

During preprocessing, we first find a good partition of the graph into cells of bounded size, while trying to minimize the total number b of boundary vertices.

Second, we perform CH preprocessing as usual, but delay the contraction of boundary vertices until the contracted graph has at most $2b$ vertices. Let B^+ be the set of all vertices with rank at least as high as that of the lowest-ranked boundary vertex. This set includes all boundary vertices and has size $|B^+| \leq 2b$.

Third, we compute labels in normal fashion, but we also store at the beginning of a label for v the ID of the cell v belongs to.

Fourth, for every pair (C_i, C_j) of cells, we run HL queries between each vertex in $B^+ \cap C_i$ and each vertex in $B^+ \cap C_j$, and keep track of the internal ID of their meeting vertex. Let m_{ij} be the maximum such ID over all queries made for this

pair of cells. We then build a $k \times k$ matrix, with entry (i, j) corresponding to m_{ij} and represented with 32 bits. Building the matrix requires up to $4b^2$ queries and concludes the preprocessing stage.

An s–t query (with $s \in C_a$ and $t \in C_b$) looks at vertices in increasing order of internal ID (as usual), but now it stops as soon as it reaches (in either label) a vertex with internal ID higher than m_{ab}—we know no query from C_a to C_b meets at a vertex higher than m_{ab}. Although this strategy needs one extra memory access to retrieve m_{ab}, long-range queries only look at a fraction of each label.

In practice, we use the PUNCH algorithm [8] to partition Europe into cells with up to $U = 20\,000$ vertices. It takes less than 3 minutes to find the partition, and 4 minutes to compute the oracle (matrix). Building the contraction hierarchy (with 2048/25K SPCs) takes about 2.5 hours. We use a top 2048 level order and 8/24 compression. Using the oracle reduces average query times from 572 ns to 357 ns. Local queries get slightly worse, mainly due to the different label ordering.

4.6 Index-free Labels

To perform an s–t query, HL must bring two labels, $L_f(s)$ and $L_r(t)$, from memory. To locate these labels in memory, it must access the entries for s and t in an *index array*. When applying all the speed-oriented optimizations described above, these two accesses can be a significant fraction of the query time.

We can eliminate the index array as follows. We reserve c bytes in each label array (forward and reverse) for each label. We store the first c bytes of $L_f(v)$ at position $v \cdot c$ in the forward label array (the reverse case is similar); the remaining entries—if any—are stored in a third array (the *escape array*). Each label (in the label array) also stores an index to the escape array. An s–t query starts reading the label arrays directly (with no index), and continues reading from the escape array if necessary. This approach increases the memory footprint of HL (since it allocates too much space for short labels), but accelerates queries that do not access the escape array. The choice of c determines the trade-off between memory and query times.

On Europe with the oracle, queries are fastest (276 ns, from 357 ns) with $c = 512$. The total space increases very little (20.1 GB to 21.3 GB), since almost two-thirds of the labels are split. The oracle ensures we rarely have to access the escape array. Indeed, using $c = 1024$ (when only 0.2% of the labels are split) requires much more space (34.4 GB) but query times are similar (280 ns). With no oracle, query times vary from 650 ns ($c = 512$) to 479 ns ($c = 1024$).

5 Experimental Results

We consider three variants of HL. The *prefix* variant is optimized for space: it uses the flexible prefix compression scheme (with inverse rank order), an index, and the oracle. The *global* variant is optimized for random and long-range queries: it uses the oracle (with top 2048 level order), no index, and 8/24 compression. The *local* version is optimized for fast short- and mid-range queries,

which are more common in practice; it uses an index but no oracle, 8/24 compression, and top 256 input order.

Table 1 compares preprocessing and random queries for all three HL variants and five previously known fast algorithms. The first is CH [14]. The second, CHASE, is a combination of CH and arc flags [5]. The third algorithm is High-Performance Multi-Level Routing (HPML) [9]: its preprocessing uses separators to build a large number of small auxiliary graphs, and each query composes some of them appropriately to create an acyclic search graph. The fourth algorithm is transit node routing [3, 4]. Long-range TNR queries consist basically of table lookups of distances between important (transit) nodes; for short-range queries, it uses CH. Finally, we consider TNR+AF [5], a combination of TNR and arc flags that reduces the average number of table lookups to less than four. Since these algorithms were tested on an older AMD machine [5, 9], Table 1 shows *scaled* running times, obtained by dividing the best published times by 1.915, the factor by which our Xeon CPU is faster (based on our calibration experiments).

The table includes a (hypothetical) implementation of a Table Lookup algorithm: it precomputes all pairs of distances, reducing queries to a single lookup. Preprocessing would be fast enough on a GPU [7], but space usage is prohibitive. We use a random memory access as an estimate of its query time.

To analyze local queries, Figure 1 plots median query times against Dijkstra rank [19]. For a search from s, the Dijkstra rank of v is i if v is the i-th vertex scanned when Dijkstra's algorithm is run from s. For HL, we run 10 000 queries per rank. All times for non-HL algorithms are taken from [5, 9] and scaled.

Although practical, HL preprocessing is slower than existing algorithms, considering they could be easily parallelized. TNR, in particular, is at least an order of magnitude faster in this regard (and can be improved even further [13]). This gap in preprocessing time between HL and other methods can be much smaller if slightly slower queries are acceptable, but our emphasis is on query times.

Table 1. Results on random queries. HL preprocessing is parallelized (others are not) with the times for building the hierarchy and computing the labels reported separately. Table Lookup preprocessing excludes copying distances from GPU to main memory.

	Europe			USA		
method	preprocessing time [h:m]	space [GB]	query [ns]	preprocessing time [h:m]	space [GB]	query [ns]
CH [5]	0:13	0.4	93 995	0:14	0.5	67 885
CHASE [5]	0:52	0.6	9 034	1:59	0.7	9 922
HPML [9]	≈12:00	3.0	9 817	≈12:00	5.1	10 078
TNR [5]	0:58	3.7	1 775	0:47	5.4	1 566
TNR+AF [5]	2:00	5.7	992	1:22	6.3	888
HL prefix	2:31 + 0:45	5.7	527	2:17 + 0:40	6.4	542
HL local	2:31 + 0:08	20.1	572	2:17 + 0:07	22.7	627
HL global	2:31 + 0:14	21.3	276	2:17 + 0:18	25.4	266
Table Lookup	> 11:03	1 208 358.7	56	> 22:44	2 293 902.1	56

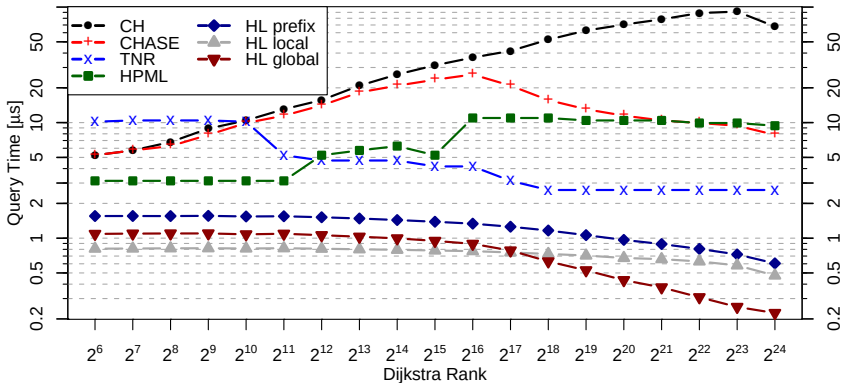

Fig. 1. Median query times on Europe for various ranges.

All variants of HL have faster queries than previous techniques. For random queries, HL global is about 3.5 times faster than TNR+AF and 6 times faster than TNR. Figure 1 shows that TNR is slower on short- or mid-range queries, taking 4 μs to 10 μs; HL local is an order of magnitude faster. This should also hold for TNR+AF, since arc flags only accelerate long-range TNR queries. (Unfortunately, there are no published values for short- and mid-range TNR+AF queries.) Although HL performs more operations than TNR+AF, better locality leads to fewer accesses to the main memory, which explains why it is faster. For short-range queries, the fastest previous algorithm (HPML) is four times slower than HL local and almost three times slower than HL global. (The published implementation of HPML [9] cannot handle some short-range queries, though it could easily be composed with CH.) In fact, HL global is only five times slower than Table Lookup (i.e., one random memory access) on average. For short- and mid-range queries, HL local is about 13 times slower than a random access.

Finally, we note that HL prefix needs a quarter of the space of HL global, but is only twice as slow, which is fast enough to outperform previous methods.

6 Concluding Remarks

We presented Hub Labels (HL), a labeling algorithm to compute exact point-to-point shortest paths in road networks. HL combines elements from a theoretical algorithm with contraction hierarchies. With careful engineering, HL is significantly faster than the best previous approaches for queries of all ranges. Some of our techniques may help accelerate other methods as well; in particular, a variant of our partition oracle could be used as a locality filter for TNR.

Our results show that road networks admit smaller labelings than the bounds of [2] suggest. It would be interesting to prove better bounds. Finding better SPCs or CH orderings, or faster algorithms to compute them, could improve HL even further by reducing the average label size. In particular, one would like a fast algorithm to approximate the smallest labeling (the method in [6] is

impractical for large networks). Reducing the space usage of HL is also desirable, as are extensions to time-dependent and other augmented networks.

References

1. I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. A Hub-Based Labeling Algorithm for Shortest Paths on Road Networks. Technical Report MSR-TR-2010-165, Microsoft Research, 2010.
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, 2010.
3. H. Bast, S. Funke, and D. Matijevic. Ultrafast shortest-path queries via transit nodes. In Demetrescu et al. [10], pp. 175–192.
4. 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.
5. R. Bauer, D. Delling, P. Sanders, D. Schieferdecker, D. Schultes, and D. Wagner. Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm. *ACM Journal of Experimental Algorithmics*, 15(2.3):1–31, 2010.
6. E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.*, 32, 2003.
7. D. Delling, A. V. Goldberg, A. Nowatzyk, and R. F. Werneck. PHAST: Hardware-Accelerated Shortest Path Trees. In *IPDPS*. IEEE, 2011. To appear.
8. D. Delling, A. V. Goldberg, I. Razenshteyn, and R. F. Werneck. Graph Partitioning with Natural Cuts. In *IPDPS*. IEEE, 2011. To appear.
9. D. Delling, M. Holzer, K. Müller, F. Schulz, and D. Wagner. High-Performance Multi-Level Routing. In Demetrescu et al. [10], pp. 73–92.
10. C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, *DIMACS* vol 74. AMS, 2009.
11. E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.
12. C. Gavoille, D. Peleg, S. Pérennes, and R. Raz. Distance labeling in graphs. *J. Algorithms*, 53(1):85–112, 2004.
13. R. Geisberger. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. Master's thesis, Karlsruhe University, 2008.
14. R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *WEA*, volume 5038 of *LNCS*, pp. 319–333. Springer, June 2008.
15. A. V. Goldberg. A Practical Shortest Path Algorithm with Linear Expected Time. *SIAM Journal on Computing*, 37:1637–1655, 2008.
16. M. Hilger, E. Köhler, R. H. Möhring, and H. Schilling. Fast Point-to-Point Shortest Path Computations with Arc-Flags. In Demetrescu et al. [10], pp. 41–72.
17. 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, 2007.
18. M. Resende and R. F. Werneck. A Fast Swap-based Local Search Procedure for Location Problems. *Annals of Operations Research*, 150:205–230, 2007.
19. P. Sanders and D. Schultes. Highway Hierarchies Hasten Exact Shortest Path Queries. In *ESA*, volume 3669 of *LNCS*, pp 568–579. Springer, 2005.
20. D. Schultes and P. Sanders. Dynamic Highway-Node Routing. In *WEA*, volume 4525 of *LNCS*, pp 66–79. Springer, June 2007.
21. M. Thorup and U. Zwick. Approximate Distance Oracles. *Journal of the ACM*, 52(1):1–24, 2005.