Hub Label Compression

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

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

Abstract. The hub labels (HL) algorithm is the fastest known technique for computing driving times on road networks, but its practical applicability can be limited by high space requirements relative to the best competing methods. We develop compression techniques that substantially reduce HL space requirements with a small performance penalty.

1 Introduction

Computing the driving time between two points in a road network is the fundamental building block for location services, which are increasingly important in practice. Dijkstra's algorithm [14] can solve this problem in essentially linear time, but this is too slow for continental road networks. This motivates two-stage algorithms, which use a preprocessing phase to precompute some auxiliary data that is then used to accelerate queries. Several efficient algorithms have recently been developed following this approach, each offering a different tradeoff between preprocessing effort and query times [3–8, 11, 16–19].

This paper focuses on *hub labels* (HL), a labeling algorithm [9, 15] developed by Abraham et al. [3, 4] to work specifically with road networks. For each vertex v in the network, its preprocessing step computes a *label* consisting of a set of hubs (other vertices), together with the distances between v and these hubs. The construction is such that, for any two vertices s and t, there must be at least one hub on the shortest s–t path that belongs to the labels of *both* s and t. Queries can then be answered by simply intersecting the two relevant labels.

The HL algorithm has several attractive properties. First, it is the fastest point-to-point shortest-path algorithm for road networks, for both long-range and (more common) local queries. Second, its query algorithm is by far the simplest: it does not even need a graph data structure, allowing practitioners with no algorithm engineering expertise to implement fast queries. Finally, the concept of labels (and hubs) is intuitive and extremely powerful, naturally lending itself to the implementation of much more sophisticated queries, such as finding nearby points of interest, optimizing ride sharing schemes, or building distance tables [2].

One aspect of HL, however, severely limits its applicability: space usage. Although preprocessing time is in line with most other methods (a few minutes on a modern server [4]), in many settings it produces a prohibitive amount of data. Computing and storing all labels requires up to two orders of magnitude more space than storing the graph itself. For a standard benchmark instance representing Western Europe, labels require roughly 20 GiB of RAM, while a graph-based

algorithm such as contraction hierarchies (CH) [16] requires less than 0.5 GiB. For more realistic representations (with turn costs), space requirements are even higher, rendering HL impractical on most commodity machines.

One could use another algorithm instead, but this would mean sacrificing query speed, ease of use, or flexibility. Storing the labels in external memory is feasible [2], but makes queries orders of magnitude slower. Finally, Abraham et al. [3] propose a (RAM-based) compact label representation that reduces space usage by a factor of roughly 3, but the compression routine itself requires a large amount of time and space (including in-memory access to all labels).

We propose *hub label compression* (HLC), a technique that achieves high compression ratios and works in *on-line* fashion. Compressing labels as they are generated drastically reduces the amount of RAM used during preprocessing, which is only slightly slower than for plain HL. On continental road networks, HLC uses an order of magnitude less space than standard HL (1.8 GiB on Western Europe). Queries are somewhat slower, but still faster than almost all other known algorithms. Crucially, they are still easy to implement (requiring no graph or priority queue), and preserve the full generality of the HL framework.

The remainder of this paper is organized as follows. After a brief overview of the HL algorithm (in Section 2), Section 3 explains the basics of HLC: the data structure, query implementation, and justification for our design decisions. Section 4 proposes optimizations that enable faster queries and better compression. Section 5 explains how the compact data structure can be generated. Section 6 has an experimental analysis of our approach. We conclude in Section 7.

2 Hub Labels

We represent a road network as a directed graph $G = (V, A)$. A vertex $v \in V$ is an intersection, and an arc $(v, w) \in A$ is a road segment with a nonnegative *length* $\ell(v, w)$, typically reflecting driving times. Let $n = |V|$. In the *point-to-point shortest path problem*, we are given a *source* vertex s and a *target* vertex t, and our goal is to find $\text{dist}(s, t)$, the total length of the shortest s–t path in G.

The *hub labels* (HL) algorithm [3, 4] solves this problem in two stages. During *preprocessing*, HL creates a forward label $L_f(v)$ and a backward label $L_b(v)$ for each vertex $v \in V$. The forward label $L_f(v)$ consists of a sequence of pairs $(w, \text{dist}(v, w))$, in which $w \in V$ is a *hub* and $\text{dist}(v, w)$ is the distance (in G) from v to w. The backward label is similar, with pairs $(u, \text{dist}(u, v))$. By construction, labels obey the *cover property*: for any two vertices s and t, the set $L_f(s) \cap L_b(t)$ must contain at least one hub v that is on the shortest s–t path. Queries are straightforward: to find $\text{dist}(s, t)$, simply find the hub $v \in L_f(s) \cap L_b(t)$ that minimizes $\text{dist}(s, v) + \text{dist}(v, t)$. By storing the entries in each label sorted by hub ID, this can be done with sequential sweeps over both labels (as in merge sort), which is very simple and cache-friendly. To compute labels for road networks efficiently, Abraham et al. [3, 4] propose a two-step algorithm. First, as in CH [16], one computes the "importance" of each vertex, roughly measuring how many shortest paths it hits. The label for each vertex v is then built in a greedy fashion:

the label starts with only the most important vertex as a hub, with more vertices added as needed to ensure every shortest path originated at v is hit. The resulting labels are surprisingly small, with around 100 hubs on average on continental road networks. The resulting queries are fast, but memory requirements are high.

3 Compressed Labels

We now present our basic compression strategy. We describe it in terms of forward labels only, which we denote by $L(\cdot)$ to simplify notation; backward labels can be compressed independently using the same method. As Abraham et al. [4] observe, the forward label $L(u)$ of u can be represented as a tree T_u rooted at u and having the hubs in $L(u)$ as vertices. Given two vertices $v, w \in L(u)$, there is an arc (v, w) in T_u (with length $\text{dist}(v, w)$) if the shortest v–w path in G contains no other vertex of $L(u)$.

Our compression scheme exploits the fact that trees representing labels of nearby vertices in the graph often have many subtrees in common. We assign a unique ID to each distinct subtree and store it only once. Furthermore, each tree is stored using a space-saving recursive representation. More precisely, for any $v \in L(u)$, let $S_u(v)$ be the maximal subtree of T_u rooted at v. This subtree can be described by its root (v itself) together with a list of the IDs of its child subtrees, each paired with an *offset* representing the distance from v to the subtree's root. We call this structure (the root ID together with a list of pairs) a *token*. Common tokens can then be shared by different labels.

The remainder of this section details the actual data structure we use, as well as queries. Section 5 discusses how to actually build the data structure.

Data Structure. Our representation makes standard assumptions [8] for real-world road networks: (1) vertices have integral IDs from 0 to $n-1$ and (2) finite distances in the graph can be represented as unsigned 32-bit integers.

A token is fully defined by the following: (1) the ID r of the *root vertex* of the corresponding subtree; (2) the number k of *child tokens* (representing child subtrees of r); and (3) a list of k pairs (i, δ_i), where i is a token ID and δ_i is the distance from r to the root of the corresponding subtree. We thus represent a token as an array of $2k + 2$ unsigned 32-bit integers. We represent the collection of all subtrees by concatenating all tokens into a single *token array* of unsigned 32-bit integers. In addition, we store an *index*, an array of size n that maps each vertex in V to the ID of its *anchor token*, which represents its full label.

We still have to define how token IDs are chosen. We say that a token is *trivial* if it represents a subtree consisting of a single vertex v, with no child tokens. The ID of such a trivial token is v itself, which is in the range $[0, n)$. *Nontrivial* tokens (those with at least one child token) are assigned unique IDs in the range $[n, 2^{32})$. Such IDs are not necessarily consecutive, however. Instead, they are chosen so as to allow quick access to the corresponding entry in the token array. More precisely, a token that starts at position p in the array has ID $n + p/2$. (This is an integer, since all tokens have an even number of 32-bit

integers.) Conversely, the token whose ID is i starts at position $2(i-n)$ in the array. Trivial tokens are not represented in the token array, since the token ID fully defines the root vertex (the ID itself) and the number of children (zero).

Since all IDs must fit in 32 bits, the token array can only represent labelings whose (compressed) size is at most $8(2^{32}-n)$ bytes. For $n \ll 2^{32}$, as is the case in practice, this is slightly less than 32 GiB, and enough to handle all instances we test; bigger inputs could be handled by varying the sizes of each field.

Queries. Since a standard (uncompressed) HL label is stored as an array of hubs (and the corresponding offsets) sorted by ID, a query requires a simple linear scan. With the compact representation, queries require two steps: we first *retrieve* the two labels, then *intersect* them. We discuss each in turn.

Retrieving a label $L(v)$ means transforming its token-based representation T_v into an array of *pairs*, each containing the ID of a hub h and its distance $\text{dist}(v,h)$ from v. We can do this by traversing the tree T_v top-down, while keeping track of the appropriate offsets. For efficiency, we avoid recursion and perform a BFS traversal of the tree using the output array itself for temporary storage. More precisely, we do as follows. First, we use the index array to get t_v, the ID of v's anchor token, and initialize the output array with a single element, $(t_v, 0)$. We then process each element of this array in order. Let (t,d) be the element in position p (processed in the p-th step). If $t < n$ (i.e., it is a trivial token), there is nothing to do. Otherwise (if $t \geq n$), we read token t from the token array, starting at position $2(t-i)$. Let w be t's root vertex. We replace (t,d) by (w,d) in the p-th position of the output array and, for each pair (i,δ_i) in the token, append the pair $(i, d+\delta_i)$ to the output array. The algorithm stops when it reaches a position that has not been written to. At this point, each pair in the output array corresponds to a hub together with its distance from v.

The second query step is to *intersect* the two arrays (for source and target) produced by the first step. Since the arrays are *not* sorted by ID, it is not enough to do a linear sweep, as in the standard HL query. We could explicitly sort the labels by ID before sweeping, but this is slow. Instead, we propose using *indexing* to find common hubs without sorting. We first traverse one of the labels to build an index of its hubs (with associated distances), then traverse the second label checking if each hub is already in the index (and adding up the distances). The simplest such index is an array indexed by ID, but this takes $\Theta(n)$ space and may lead to many cache misses. A better alternative is to use a small hash table with a simple hash function (we use ID modulo 1024) and linear probing [10].

Discussion. Our data structure balances space usage, query performance, and simplicity. If compression ratios were our only concern, we could easily reduce space usage with various techniques. We could use fewer bits for some of the fields (notably the number of children). We could use relative (rather than absolute) references and variable-length encoding for the IDs [21]. We could avoid storing the length of each arc (v,w) multiple times in the token array (as offsets in tokens rooted at v) by representing labels as subtrees of the full CH graph [16], possibly

using techniques from succinct data structures [20]. Such measures would reduce space usage, but query times could suffer (due to worse locality) and simplicity, arguably the main attraction of HL, would be severely compromised.

4 Variants and Optimizations

We now consider optimizations to our basic compression scheme. They modify the preprocessing stage only, and require *no change* to the query algorithm.

The Token DAG. Conceptually, our compressed representation can be seen as a *token graph*. Each vertex of the graph corresponds to a *nontrivial* token x, and there is an arc (x, y) if and only if y is a child of x in some label. The length of the arc is the offset of y within x. The token graph has some useful properties. By definition, a token x that appears in multiple labels has the same children (in the corresponding trees) in all of them. This means x has the same set of descendants in all labels it belongs to, and by construction these are exactly the vertices in the subgraph reachable from x in the token graph. This implies that this subgraph is a tree, and that the token graph is a DAG. It also implies that the subgraph reachable from x by following only *reverse* arcs is a tree as well: if there were two distinct paths to some ancestor y of x, the direct subgraph reachable from y would not be a tree. We have thus proven the following.

Lemma 1. *The token graph is a DAG in which any two vertices are connected by at most one path.*

Note that all DAG vertices with in-degree zero are anchor tokens, and DAG vertices with out-degree zero (which we call *leaf tokens*) are nontrivial tokens that only have trivial tokens (which are not in the token DAG) as children.

Pruning the DAG. Retrieving a compressed label may require a nonsequential memory access for each internal node in the corresponding tree. To improve locality (and even space usage), we propose two operations. We can eliminate a non-anchor token t (rooted at a vertex v) with a single parent t' in the token DAG as follows. We replace each arc (t, t'') in the DAG by an arc (t', t'') with length equal to the sum of (t', t) and (t, t''). Moreover, in t', we replace the reference to t by a reference to trivial token v. This *1-parent elimination* operation potentially improves query time and space. Similarly, *1-child elimination* applies to a non-anchor token t that has exactly two parents in the DAG, a single nontrivial child t', and no nontrivial children. We can discard t and create direct arcs from each parent of t to t', saving nonsequential accesses with no increase in space.

Flattening. A more aggressive approach to speed up queries is to *flatten* subtrees that occur in many labels. Instead of describing the subtree recursively, we create a single token explicitly listing *all* descendants of its root vertex, with

appropriate offsets. We propose a greedy algorithm that in each step flattens the subtree (token) that reduces the expected query time the most, assuming all labels are equally likely to be accessed. Intuitively, our goal is to minimize the average number nonsequential accesses when reading the labels.

Let $\lambda(x)$ be the number of labels containing a nontrivial token x, and let $\alpha(x)$ be the number of proper descendants of x in the token DAG ($\alpha(x)$ is 0 if x is a leaf). The *total access cost* of the DAG is the total number of nonsequential accesses required to access all n labels. (This is n times the expected cost of reading a random label.) If H is the set of all anchor tokens, the total access cost is $\sum_{x \in H}(1+\alpha(x))$. The share of the access cost attributable to any token x is $\lambda(x) \cdot (1+\alpha(x))$. Flattening the corresponding subtree would reduce the total access cost by $v(x) = \lambda(x)\alpha(x)$, as a single access would suffice to retrieve x.

Our algorithm starts by traversing the token graph twice in topological order: a direct traversal initializes $\lambda(\cdot)$ and a reverse one initializes $\alpha(\cdot)$. It then keeps the $v(x) = \lambda(x)\alpha(x)$ values in a priority queue. Each step takes the token x with maximum $v(x)$ value, flattens x, then updates all $v(\cdot)$ values that are affected. For every ancestor z of x, we set $\alpha(z) \leftarrow \alpha(z) - \alpha(x)$; for every descendant y of x, we set $\lambda(y) \leftarrow \lambda(y) - \lambda(x)$. If $\lambda(y)$ becomes zero, we simply discard y. Finally, we remove the outgoing arcs from x (making x a leaf) and set $\alpha(x) \leftarrow 0$. We stop when the total size of the token array increases beyond a given threshold. Using Lemma 1, one can show that this algorithm runs in $O(\tau\mu)$ time, where τ is the initial number of tokens in the DAG and μ is the maximum label size.

Discussion. We implemented flattening as described above, but the concept is more general: one could flatten arbitrary subtrees (not just maximal ones), as long as unflattened portions are represented elsewhere with appropriate offsets. The 1-parent and 1-child elimination routines are special cases of this, as is Abraham et al.'s compression technique [3], which splits each label into a subtree containing its root and a (shared) token representing the remaining forest.

With no stopping criterion, the greedy flattening algorithm eventually leads to exactly n (flattened) tokens, each corresponding to a label in its entirety, as in the standard (uncompressed) HL representation. Conversely, we could have a "merge" operation that combines tokens rooted at the same vertex into a single token (not necessarily flattened) representing the union of the corresponding trees. This saves space, but tokens no longer represent minimal labels. Our BFS-based label retrieval technique is still correct (it access all relevant hubs), but it may visit each hub more than once, since Lemma 1 no longer holds. We could fix this by visiting tokens in increasing order of distance (with a heap), as in CH queries [16]. This similarity is not accidental: repeated application of the "merge" operation eventually leads to a single token rooted at each vertex (representing all subtrees rooted at it), with the token graph exactly matching the upward CH graph [16]. In this sense, HLC generalizes both CH and HL.

Reordering. If the tokens corresponding to the endpoints of a DAG arc are not stored consecutively in memory, traversing this arc usually results in a cache

miss. Tokens must be stored in increasing order of ID, but these IDs can be chosen to our advantage. By reordering tokens appropriately during preprocessing, we can potentially decrease the number of cache misses during queries.

We propose the following. First, we *mark* a subset M of the DAG arcs such that each token t in the DAG has at most one incoming arc and at most one outgoing arc in M. This creates a collection of vertex-disjoint paths. We then assign consecutive IDs to the vertices along each path (the order among paths is arbitrary). For random queries, this assignment avoids (compared to a random order) approximately $\lambda(t)/n$ cache misses for each marked arc (t, t'). We define the *gain* associated with a set M as the sum of $\lambda(t)$ over all marked arcs (t, t'). One can show that the set M^* with maximum gain can be found using minimum-cost flows. For efficiency, however, we use a greedy heuristic instead. We start with all arcs unmarked, then process the tokens in nonincreasing order of $\lambda(\cdot)$. To process a token t, we mark, among all outgoing arcs (t, t') such that t' has no marked incoming arc, the one maximizing $\lambda(t')$. If no such arc exists, we just skip t. Eventually, this leads to a maximal set M of marked arcs.

5 Label Generation

We now explain how the data structures described in Section 3 can actually be built. To create a compressed representation of an existing set of labels, we start with an empty token array and *tokenize* the labels (i.e., create their token-based representation) one at a time, in any order. To tokenize a label $L(v)$, we traverse the corresponding tree T_v bottom-up. To process a vertex $w \in T_v$, we first build the token t_w that represents it. This can be done because at this point we already know the IDs of the tokens representing the subtrees rooted at w's children. We then pick an ID i to assign to t_w. First, we use hashing to check if t_w already occurs in the token array. If it does, we take its existing ID. Otherwise, we append t_w to the token array and use its position p to compute the ID i, as described in Section 3 ($i = n + p/2$). When the bottom-up traversal of T_v ends, we store the ID of t_v (the anchor token of v) in the index array.

Note that label compression can be implemented in on-line fashion, as labels are generated. Asymptotically, it does not affect the running time: we can compress all labels in linear time. In practice, however, generating a label often requires access to other existing labels [3, 4]. If we are not careful, the extra cost of retrieving existing compressed labels may become the bottleneck.

With that in mind, we modify Abraham et al.'s recursive label generation [4] to compress labels as they are created. Building on the preprocessing algorithm for *contraction hierarchies* (CH) [16], they first find a heuristic *order* among all vertices, then *shortcut* them in this order. To process a vertex v, one (temporarily) deletes v and adds arcs as necessary to preserve distances among the remaining vertices. More precisely, for every pair of incoming and outgoing arcs (u, v) and (v, w) such that $(u, v) \cdot (v, w)$ is the only u–w shortest path, one adds a new shortcut arc (u, w) with $\ell(u, w) = \ell(u, v) + \ell(v, w)$. This procedure outputs the order itself (given by a rank function $r(\cdot)$) and a graph $G^+ = (V, A \cup A^+)$,

where A^+ is the set of shortcuts. The number of shortcuts depends on the order; intuitively, it is best to first shortcut vertices that belong to few shortest paths.

Labels are then generated one by one, in reverse contraction (or top-down) order, starting from the last contracted vertex. The first step to process a vertex v is to build an initial label $L(v)$ by combining the labels of v's upward neighbors $U_v = \{u_1, u_2, \ldots, u_k\}$ (u is an upward neighbor of v if $r(u) > r(v)$ and $(u, v) \in A \cup A^+$.) For each $u_i \in U_v$, let T_{u_i} be the (already computed) tree representing its label. We initialize T_v (the tree representing $L(v)$) by taking the first tree (T_{u_1}) in full, and making its root a child of v itself (with an arc of length $\ell(v, u_1)$). We then process the other trees T_{u_i} ($i \geq 2$) in top-down fashion. Consider a vertex $w \in T_{u_i}$ with parent p_w in T_{u_i}. If $w \notin T_v$, we add it—p_w must already be there, since we process vertices top-down. If $w \in T_v$ and its distance label $d_v(w)$ is higher than $\ell(v, u_i) + d_{u_i}(w)$, we update $d_v(w)$ and set w's parent in T_v to p_w.

Once the merged tree T_v is built, we eliminate any vertex $w \in T_v$ such that $d_v(w) > \text{dist}(v, w)$. The actual distance $\text{dist}(v, w)$ can be found by *bootstrapping*, i.e., running a v–w HL query using $L(v)$ itself (unpruned, obtained from T_v) and the label $L(w)$ (which must already exist, since labels are generated top-down).

Our algorithm differs from Abraham et al.'s [4] in that it stores labels in compressed form. To compute $L(v)$, we must retrieve (using the token array) the labels of its upward neighbors, taking care to preserve the parent pointer information that is implicit in the token-based representation. Similarly, bootstrapping requires retrieving the labels of all candidate hubs.

To reduce the cost of retrieving compressed labels during preprocessing, we keep an LRU cache of *uncompressed* labels. Whenever a label is needed, we first look it up in the cache, and only retrieve its compressed version if needed (and add it to the cache). Since labels used for bootstrapping do not need parent pointers and labels used for merging do, we keep an independent cache for each representation. To minimize cache misses, we do not generate labels in strict top-down order; instead, we process vertices in increasing order of ID, deviating from this order as necessary. If we try to process v and realize it has an unprocessed upward neighbor w, we process w first, then come back to v. (We use a stack to keep track of delayed vertices.) The cache hit ratio improves because nearby vertices (with similar labels) tend to have similar IDs in our test instances.

For additional acceleration, we also avoid unnecessary bootstrapping queries. If a vertex v has a single upward neighbor u, there is no need to bootstrap T_v (and u's token can be reused). If v has multiple upward neighbors, we bootstrap T_v in bottom-up order. If we determine that the distance label for a vertex $w \in T_v$ is correct, its ancestors in T_v must be as well, and need not be tested.

6 Experiments

We implemented our HLC algorithm in C++ and compiled it (using full optimization) with Microsoft Visual C++ 2010. We tested it on a machine running Windows Server 2008 R2 with 96 GiB of DDR3-1333 RAM and two 6-core Intel Xeon X5680 CPUs at 3.33 GHz (each CPU has 6×64 KB of L1, 6×256 KB L2,

Table 1. HLC and HL performance, aggregated over forward and backward labels.

	PREPROCESSING				QUERIES		
algorithm	time [h:m]	space [MiB]	tokens /vertex	reads /label	hash [ns]	linear [ns]	merge [ns]
basic HLC	00:47	2016.3	4.925	39.46	3338	3832	7035
+1-parents	00:48	1759.0	3.053	36.27	3313	3830	7007
+1-children	00:49	1759.0	2.912	35.78	3304	3809	6996
+5% flat	00:50	1840.1	2.912	12.96	2554	2999	6205
plain HL	00:38	21776.1	2.000	1.00	1208	1315	617

and 12 MB of shared L3 cache). For ease of comparison, all runs were sequential. Our default benchmark instance (made available by PTV AG for the 9th DIMACS Implementation Challenge [13]) represents (Western) Europe; it has $n = 18 \cdot 10^6$ vertices, $m = 42 \cdot 10^6$ arcs, and travel times as the cost function.

Our first experiment examines the effectiveness of all variants of HLC we considered. For reference, we also report the performance of a plain implementation of HL algorithm, where each label is an array of 32-bit integers (hubs and distances), sorted by hub ID. The complicated optimizations proposed by Abraham et al. [3] (such as ID reassignment, 8/24 compression, distance oracles, and index-free queries) will be considered later. We use the default contraction order proposed by Abraham et al. [4] (HL-15), with 78.24 hubs per label on average.

For each method, Table 1 shows the total preprocessing time (including finding the contraction order), the amount of data generated, the average number of (both trivial and nontrivial) tokens per vertex, and the average number of nontrivial tokens read to retrieve a label ("reads/label"). All values are aggregated over both forward and backward labels. We also show average times (over 10^7 random s–t pairs) for three query strategies: hashed index, linear index (an n-sized array), and merging (including a call to the STL sort function for HLC).

The basic version of HLC (described in Section 3) uses an order of magnitude less space than HL, with similar preprocessing time. This makes the label-based approach much more practical: while few current servers have 20 GiB of RAM available, most can easily handle 2 GiB data structures. (For comparison, labels compressed with `gzip` would take 7.7 GiB and would not support fast queries.) Space usage can be reduced by around 10% if 1-parent tokens are eliminated, with little effect on query times; 1-child elimination has a small but positive effect. Greedily flattening tokens until the token array increases by 5% reduces the number of tokens needed to represent each label by more than 70%, but queries are only about 25% faster ("popular" tokens end up in cache anyway, even without flattening). Flattening twice as much (10%) would further reduce query times by only 3%.

Regarding queries, hashing is slightly faster than linear indexing for HLC, and significantly faster than merging (which requires sorting). For HL, whose labels are already sorted by ID, merging is by far the best strategy, due to its favorable access pattern and simplicity. In the end, hashing and worse locality make HLC queries about four times slower than plain HL queries.

Table 2 compares the two versions of HLC (with all optimizations) with other fast algorithms. It includes CH [16] and three variants of HL [3, 4]: *HL-∞ global* is optimized for long-range queries, *HL-15 local* for short-range queries, and *HL prefix* minimizes space usage. HLC-0 and HLC-15 use the same vertex orders as HL-0 and HL-15, respectively [4]. We also include Transit Node Routing (TNR) [5, 6], which uses table lookups for long range queries, CH for local queries, and (optionally) hashing for midrange queries. TNR+AF [8] uses *arc flags* [19] to guide TNR towards the target. We report preprocessing time, space usage, and average time for random queries, considering the best available implementation of each method. All times are sequential and scaled to match our machine [3].

Table 2. Random queries on Europe.

method	prepro [h:m]	space [GB]	query [ns]
CH [16]	0:14	0.4	79373
TNR [5]	0:21	2.5	1711
TNR+AF [8]	2:00	5.7	992
HL prefix [3]	≈2:00	5.7	527
HL-15 local [4]	0:37	18.8	556
HL-∞ global [4]	≈60:00	17.7	254
HLC-0	0:30	1.8	2989
HLC-15	0:50	1.8	2554

While standard HL uses much more space than other methods, compression makes the hub labels approach less of an outlier. HLC uses only about 4 times as much space as CH (the most compact method), but random queries are 30 times faster. HLC is comparable to TNR in all three criteria. As we have argued before, however, HLC has advantages that can make it more practical: simplicity (no graphs or priority queues) and flexibility (natural extensions based on hubs).

Moreover, HLC is faster for local queries, which in practice are more common than long-range (or random) ones. Fig. 1 plots the median query times of several algorithms as a function of the *Dijkstra rank* [16]. (Vertex v has Dijkstra rank i with respect to s if v is the i-th vertex scanned by Dijkstra's algorithm from s.) Each point corresponds to 10 000 queries with a given rank. The numbers were taken from [3, 5, 8, 16] and scaled appropriately.

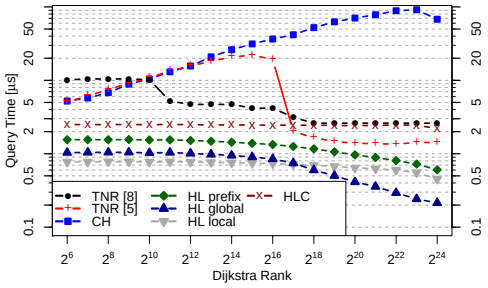

Fig. 1. Local queries.

Query times increase with Dijkstra rank for CH (since its search space is bigger), but decreases for TNR (since it can only do fast table lookups for more global queries). Standard HL can reorder vertices to allow long-range queries to skip some unimportant hubs [3]. In contrast, HLC must always visit nearby hubs before getting to more important ones, and its query times are largely independent of the Dijkstra rank. For local queries, HLC is only about three times slower than HL, and much faster than other methods.

Finally, Table 3 reports the performance of HLC on a variety of road networks. We start with Western Europe from PTV, taking both travel times (as before) and travel distances as cost functions. We also test an *expanded* version of this instance with turn costs, which we model by creating a vertex to represent

Table 3. Results for HLC on various instances: number n of vertices (in millions), average (out-)degree, preprocessing time, space usage, compression ratio (relative to plain HL), average number of hubs per label, and (hash-based) query times.

source	input	metric	turns	n [10^6]	deg	time [h:m]	space [MiB]	comp ratio	hubs /label	query [ns]
PTV	W. Europe	time	×	18.0	2.36	0:50	1840.1	11.83	78.24	2554
	W. Europe	time	✓	42.6	2.72	4:10	6363.4	11.02	107.03	3512
	W. Europe	dist	×	18.0	2.36	3:15	2973.9	15.91	171.25	5734
Tiger	US	time	×	23.9	2.41	0:53	2979.6	8.62	69.33	2486
	US	dist	×	23.9	2.41	2:32	4126.2	14.08	157.99	5294
OSM	Australia	time	×	4.9	1.97	0:13	408.8	9.48	51.30	1689
	S. America	time	×	11.3	2.18	0:29	1167.2	8.33	55.25	1865
	N. America	time	×	162.5	2.04	5:52	14560.6	18.25	106.18	3520
	Old World	time	×	188.7	2.02	6:14	17164.4	16.07	94.78	3232
Bing	Europe	default	×	47.9	2.23	1:53	4791.8	15.20	98.53	3264
	Europe	default	✓	107.0	2.26	8:01	13046.6	14.38	113.92	3854
	N. America	default	×	30.3	2.41	1:33	3461.2	14.52	107.74	3437
	N. America	default	✓	72.5	2.61	14:34	13403.8	11.04	132.87	4429

each original arc [11]; we follow Delling et al. [11] and set U-turn costs to 100 seconds and all other turn costs to zero. In addition, we test TIGER data [13] representing the USA, as well as four OpenStreetMap (OSM) instances (v. 121812) with realistic travel times (but no turn costs or restrictions), representing Australia, South America, North (and Central) America, and Old World (Africa, Asia, and Europe). Finally, we test the actual data used by Bing Maps (building on Navteq data) in production; the cost function is proprietary, but correlates well with travel times, as one would expect. We consider versions with turn costs (as used in production) and without. All instances are strongly connected.

Although the average number of hubs per label varies significantly, HLC always needs one order of magnitude less storage than plain HL. Without compression, some instances would require more than 200 GiB of RAM, which is hardly practical. This is an issue especially for OSM data, whose vertices represent both topology (intersections) and geometry, leading to sparse but large graphs. With compression, space usage is kept below 20 GiB in every case, which is much more manageable. Queries always remain below 6 μs.

7 Final Remarks

We presented compression techniques that substantially reduce the memory requirements for HL. Not only do they keep queries fast and simple, but they also preserve the flexibility and generality of labels. This makes the label-based approach practical in a wider range of applications. An open problem is to speed up its preprocessing to handle dynamic scenarios (such as real-time traffic) efficiently. Although techniques such as CRP [11, 12] can quickly adapt to changes in the length function, they have more complicated (and much slower) queries.

Acknowledgements. We thank D. Luxen for routable OSM instances [1].

References

1. Project OSRM, 2012. http://project-osrm.org/.
2. I. Abraham, D. Delling, A. Fiat, A. V. Goldberg, and R. F. Werneck. HLDB: Location-Based Services in Databases. In *Proc. SIGSPATIAL GIS*, pp. 339–348. ACM Press, 2012.
3. I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. A Hub-Based Labeling Algorithm for Shortest Paths on Road Networks. In *Proc. SEA*, LNCS 6630, pp. 230–241. Springer, 2011.
4. I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. Hierarchical Hub Labelings for Shortest Paths. In *Proc. ESA*, LNCS 7501, pp. 24–35. Springer, 2012.
5. J. Arz, D. Luxen, and P. Sanders. Transit Node Routing Reconsidered. In *Proc. SEA*, LNCS. Springer, 2013.
6. H. Bast, S. Funke, P. Sanders, and D. Schultes. Fast Routing in Road Networks with Transit Nodes. *Science*, 316(5824):566, 2007.
7. R. Bauer and D. Delling. SHARC: Fast and Robust Unidirectional Routing. *ACM JEA*, 14(2.4):1–29, 2009.
8. 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 JEA*, 15(2.3):1–31, 2010.
9. E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and Distance Queries via 2-Hop Labels. *SIAM J. Computing*, 32(5):1338–1355, 2003.
10. T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.
11. D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Customizable Route Planning. In *Proc. SEA*, LNCS 6630, pp. 376–387. Springer, 2011.
12. D. Delling and R. F. Werneck. Faster Customization of Road Networks. In *Proc. SEA*, LNCS. Springer, 2013.
13. C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors. *The Shortest Path Problem: 9th DIMACS Implementation Challenge*, DIMACS Book 74. AMS, 2009.
14. E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.
15. C. Gavoille, D. Peleg, S. Pérennes, and R. Raz. Distance Labeling in Graphs. *Journal of Algorithms*, 53:85–112, 2004.
16. R. Geisberger, P. Sanders, D. Schultes, and C. Vetter. Exact Routing in Large Road Networks Using Contraction Hierarchies. *Transportation Sci.*, 46(3):388–404, 2012.
17. A. V. Goldberg and C. Harrelson. Computing the Shortest Path: A* Search Meets Graph Theory. In *Proc. SODA'05*, pp. 156–165. SIAM, 2005.
18. A. V. Goldberg, H. Kaplan, and R. F. Werneck. Reach for A*: Shortest Path Algorithms with Preprocessing. In Demetrescu et al. [13], pp. 93–139.
19. U. Lauther. An Experimental Evaluation of Point-To-Point Shortest Path Calculation on Roadnetworks with Precalculated Edge-Flags. In Demetrescu et al. [13], pp. 19–40.
20. I. Munro and S. Rao. Succinct representation of data structures. In D. P. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*. CRC, 2004.
21. P. Sanders, D. Schultes, and C. Vetter. Mobile Route Planning. In *Proc. ESA'08*, LNCS 5193, pp. 732–743. Springer, 2008.