Hierarchical Hub Labelings for Shortest Paths

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

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

Abstract. We study hierarchical hub labelings for computing shortest paths. Our new theoretical insights into the structure of hierarchical labels lead to faster preprocessing algorithms, making the labeling approach practical for a wider class of graphs. We also find smaller labels for road networks, improving the query speed.

1 Introduction

Computing point-to-point shortest paths in a graph is a fundamental problem with many applications. Dijkstra's algorithm [14] solves this problem in near-linear time [18], but for some graph classes sublinear-time queries are possible if preprocessing is allowed (e.g., [12, 16]). In particular, Gavoille et al.'s *distance labeling* algorithm [16] precomputes a *label* for each vertex such that the distance between any two vertices s and t can be computed given only their labels. A special case is *hub labeling* (HL), where the label of each vertex u consists of a collection of vertices (the *hubs* of u) with their distances to or from u. Labels obey the *cover property*: for any two vertices s and t, there exists a vertex w on the shortest s–t path that belongs to both labels (of s and t). Cohen at al. [9] give a polynomial-time algorithm to approximate the smallest labeling within a factor of $O(\log n)$, where n is the number of vertices. The average label size can be quite large ($\Omega(\sqrt[3]{n})$ even for planar graphs [16]), but not always. On real-world DAG-like graphs, for instances, labels are quite small in practice [8].

Abraham et al. [4] conjecture that road networks have a small *highway dimension*, and show that if the highway dimension is polylogarithmic, so is the label size. This motivated the experimental study of labels for road networks. Unfortunately, preprocessing algorithms with theoretical guarantees on the label size run in $\Omega(n^4)$ time [9, 4, 1], which is impractical for all but small graphs. In previous work [2], we showed how to compute labels for road networks based on *contraction hierarchies* (CH) [17], an existing preprocessing-based shortest path algorithm. The resulting labels are unexpectedly small [2], with average size 85 on a graph of Western Europe [13] with 18 million vertices. The corresponding query algorithm is the fastest currently known for road networks.

In this paper we study *hierarchical* hub labelings, a natural special case where the relationship "vertex v is in the label of vertex w" defines a partial order on the vertices. We obtain theoretical insights into the structure of hierarchical labelings and their relationship to vertex orderings. In particular, we show that for every total order there is a minimum hierarchical labeling. We use the theory to develop efficient algorithms for computing the minimum labeling from an ordering, and for computing orderings which yield small labelings. We also show that CH and hierarchical labelings are closely related and obtain new top-down CH preprocessing algorithms that lead to faster CH queries.

Our experimental study shows that our new label-generation algorithms are more efficient and compute labels for graphs that previous algorithms could not handle. For several graph classes (not only road networks), the labels are small enough to make HL the fastest distance oracle in practice. Furthermore, the new algorithms compute smaller labels; in particular, we reduce the average label size for Western Europe from 85 to 69, accelerating the fastest method by 8%.

This paper is organized as follows. After settling preliminaries in Section 2, Section 3 considers the relationship between vertex orderings and hierarchical labelings. Section 4 presents several techniques for computing vertex orderings, and Section 5 shows how to compute labels from orderings. Section 6 studies the relationship between labelings and CH, while Section 7 presents our experimental evaluation. Section 8 contains concluding remarks. Details omitted due to space constraints can be found in the full version of this paper [3].

2 Preliminaries

The input to the shortest path problem is a graph $G = (V, A)$, 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 point-to-point problem (*a query*) is, given a source s and a target t, to find the distance $\mathrm{dist}(s, t)$ between them, i.e., the length of the shortest path P_{st} between s and t in G. We assume that shortest paths are unique, which we can enforce by breaking ties consistently.

The standard solution to this problem is Dijkstra's algorithm [14]. It builds a shortest path tree by processing 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 $p(v) = \mathit{null}$ 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 all vertices have been processed. Dijkstra's worst-case running time, denoted by $\mathrm{Dij}(G)$, is $O(m + n \log n)$ [15] in the comparison model, and even better in weaker models [18].

For some applications, even linear time is too slow. For faster queries, *labeling* algorithms preprocess the graph and store a *label* with each vertex; the s–t distance can be computed from the labels of s and t. Our focus is on *hub labeling* (HL), a special case. For each vertex $v \in V$, HL builds a forward label $L_f(v)$ and a backward label $L_b(v)$. The forward label $L_f(v)$ consists of a sequence of pairs $(u, \mathrm{dist}(v, u))$, where u is a vertex (a *hub* in this context). Similarly, $L_b(v)$ consists of pairs $(u, \mathrm{dist}(u, v))$. Note that the hubs in the forward and backward labels of u may differ. Collectively, the labels obey the *cover property*: for any two vertices s and t, $L_f(s) \cap L_b(t)$ must contain at least one vertex on the shortest s–t path. For an s–t query, among all vertices $u \in L_f(s) \cap L_b(t)$ we pick the one minimizing $\text{dist}(s, u) + \text{dist}(u, t)$ and return this sum. If the entries in each label are sorted by hub ID, this can be done with a coordinated sweep over the two labels, as in mergesort.

We say that the forward (backward) *label size of* v, $|L_f(v)|$ ($|L_b(v)|$), is the number of hubs in $L_f(v)$ ($L_b(v)$). The time for an s, t query is $O(|L_f(s)|+|L_b(t)|)$. The *maximum label size* (denoted by M) is the size of the biggest label. The *labeling* $\mathcal{L}$ is the set of all forward and backward labels. We denote its *size* by $|\mathcal{L}| = \sum_v (|L_f(v)| + |L_b(v)|)$, while $L_a = |\mathcal{L}|/(2n)$ denotes the average label size.

Cohen et al. [9] show how to generate in $O(n^4)$ time labels whose average size is within a factor $O(\log n)$ of the optimum. Their algorithm maintains a set of U of uncovered shortest paths (initially all the paths) and labels $L_f(v)$ and $L_b(v)$ (initially empty) for every vertex v. Each iteration of the algorithm selects a vertex v and a set of labels to add v to so as to maximize the ratio between the number of paths covered and the increase in total label size.

Given two distinct vertices v, w, we say that $v \preceq w$ if $L_f(v) \cup L_b(v)$ contains w. A labeling is *hierarchical* if $\preceq$ is a partial order. We say that this order is *implied* by the labeling. Cohen et al.'s labels are not necessarily hierarchical.

Given a total order on vertices, the *rank function* $r : V \to [1 \ldots n]$ ranks the vertices according to the order. We will call the corresponding order r.

3 Canonical Labelings

We say that a hierarchical labeling $\mathcal{L}$ respects a total order r if the implied partial order is consistent with r. Given a total order r, a *canonical labeling* is the labeling that contains only the following hubs. For every shortest path P_{st}, the highest ranked vertex $v \in P_{st}$ is in the forward label of s and in the backward label of t (with the corresponding distances). A canonical labeling is hierarchical by construction: the vertex v that we add to the labels of s and t has the highest rank on P_{st}, and therefore $r(v) \geq r(s)$ and $r(v) \geq r(t)$. This also implies that the canonical labeling respects r.

Lemma 1. *Let $\mathcal{L}$ be a hierarchical labeling. Then the set of vertices on any shortest path has a unique maximum element with respect to the partial order implied by $\mathcal{L}$.*

Proof. The proof is by induction on the number of vertices on the path. The result is trivial for paths with a single vertex. Consider a path $v_1 \ldots v_k$ with $k > 1$. The subpath $v_2 \ldots v_k$ has a maximum vertex v_i by the inductive assumption. Consider the subpath $v_1 \ldots v_i$. The internal vertices v_j are not in $L_b(v_i)$ by the choice of v_i. Therefore either $v_i \in L_f(v_1)$ (and v_i is the maximum vertex on the path), or $v_1 \in L_b(v_i)$ (and v_1 is the maximum vertex). □

Given a hierarchical labeling $\mathcal{L}$, the lemma implies that all total orders r that $\mathcal{L}$ respects have the same canonical labeling $\mathcal{L}'$.

Theorem 1. *Let $\mathcal{L}$ be a hierarchical labeling, r any total order such that $\mathcal{L}$ respects r, and $\mathcal{L}'$ the canonical labeling for r. Then $\mathcal{L}'$ is contained in $\mathcal{L}$.*

Proof. Consider the shortest path P from s to t, and let v be the maximum rank vertex on P. Let $\mathcal{L} = (L_f, L_b)$. We show that $v \in L_f(s)$; the case $v \in L_b(t)$ is similar. Consider the shortest path P' from s to v. Since shortest paths are unique, $P' \subseteq P$, and therefore v is the maximum rank vertex on P'. It follows that the only vertex of P' that is in $L_b(v)$ is v. Thus v must be in $L_f(s)$. □

We now consider how to extract the canonical labeling $\mathcal{L}'$ from a hierarchical labeling $\mathcal{L}$. One approach is to first extract a total order r from $\mathcal{L}'$, then build the canonical label from the order. We can find r with a topological sort of the DAG representing the partial order induced by $\mathcal{L}$. We can then easily build canonical labels from r in $O(n\text{Dij}(G))$ time, as Section 5 will show.

We can often do better by simply *pruning* the labels in $\mathcal{L}$. We explain how to prune forward labels; backward labels can be dealt with similarly. Consider a vertex $w \in L_f(v)$. We must keep w in $L_f(v)$ if and only if it is the maximum-rank vertex on the shortest v–w path P_{vw}. Since we have not computed the ranks, we must test for maximality indirectly. We use the following observation: if the highest ranked vertex in P_{vw} is $u \neq w$, then u must be in both $L_f(v)$ and $L_b(w)$ (since it belongs to the canonical label). By running what is essentially a v–w HL query, we can determine if such a vertex u exists. If so, we delete w from $L_f(v)$. The algorithm takes $O(M)$ time to process each of the $|\mathcal{L}|$ vertex-hub pairs (v, w), for a total time of $O(|\mathcal{L}|M) = O(nM^2)$. (Recall that M is the maximum label size.) When labels are not too big ($M = O(\sqrt{m})$), this is faster than the $O(n\text{Dij}(G))$ approach mentioned above.

4 Vertex Orderings

Canonical labelings are the smallest hierarchical labelings defined by a vertex ordering. By the *quality* of an ordering we mean the size of its implied labeling. In this section, we discuss several ways of computing vertex orderings. We review a known approach, improve existing algorithms, and introduce new ones.

Contraction Hierarchies. CH [17] has been heavily studied in the context of point-to-point shortest paths in road networks. It is a preprocessing-based algorithm, but not a labeling algorithm. During preprocessing, CH orders all vertices and applies the *shortcut* operation to each vertex in that order. Intuitively, the ordering is from the least to the most important vertex. When applied to a vertex v, the shortcut operation temporarily removes v from the graph and adds as few arcs as necessary to preserve the distances between the remaining vertices. More precisely, for any two neighbors u and w of v, it runs a *witness search* (Dijkstra) to compute $\text{dist}(u, w)$ in the current graph (without v). If $\ell(u, v) + \ell(v, w) < \text{dist}(u, v)$, it adds a *shortcut arc* (u, w) with $\ell(u, w) = \ell(u, v) + \ell(v, w)$. The output of CH preprocessing is a graph $G^+ = (V, A \cup A^+)$, where A^+ denotes the set of shortcuts, as well as the order in which vertices were shortcut. The CH query algorithm runs bidirectional Dijkstra on G^+, but considers only upward arcs.

The CH query performance and $|A^+|$ highly depend on the order in which vertices are shortcut. The best known orderings [17, 19] use online heuristics to estimate how "important" each vertex is based on local graph properties (such as the net number of shortcuts added if the vertex were contracted).

The running time of CH preprocessing depends on the number of shortcuts. In the best case, when both G and G^+ have constant degree, it uses $O(n)$ space and runs in $O(nW)$ time, where W denotes the time for a witness search. In road networks, witness searches are local Dijkstra searches, which makes preprocessing quite fast. If G^+ is dense, however, shortcuts may need $O(n^2)$ witness searches, causing the standard implementation of CH preprocessing to run in $O(n^3W)$ time and $O(n^2)$ space.

Even for road networks, previous experiments [2] showed that the ordering computed by CH preprocessing can be improved. Next, we discuss ways to compute orderings with better worst-case time bounds that yield smaller labels.

Top-Down. We turn to an algorithm (which we call TD) that selects vertices top-down, from most to least important (as opposed to CH, which is bottom-up). Conceptually, each unpicked vertex v maintains the set U_v of all *uncovered paths*, i.e., shortest paths that contain v but do not contain any previously selected vertex. Each iteration selects the next most important vertex v^* according to some criterion that depends on U_{v^*}, then updates all sets $U_v = U_v \setminus U_{v^*}$. This is repeated until all paths are covered.

We consider two versions of this algorithm, with different selection criteria. The *covering* version (TDc) always picks the vertex v which maximizes $|U_v|$. The *weighted covering* version (TDwc) selects the v maximizing $|U_v|/(s_v + t_v)$, where s_v and t_v are the sizes of the sets consisting of the first and last vertices on the paths in U_v, respectively. TDwc is inspired by Cohen et al.'s algorithm [9].

An obvious implementation of TD is to compute every U_v from scratch in each round. This takes $O(n)$ space but $O(n^2\mathrm{Dij}(G))$ time, which is impractical even for mid-sized graphs. We therefore propose an *incremental* implementation of TDc that runs in $O(n\mathrm{Dij}(G))$ time. It can be extended to other TD algorithms as long as each iteration can pick the vertex v^* (given the updated U_v sets) in $O(\mathrm{Dij}(G))$ time. In particular, this holds for TDwc.

The incremental TDc implementation maintains a shortest path tree T_r for every vertex r, representing all uncovered shortest paths starting at r. Initially, these are full shortest path trees, computed in $O(n\mathrm{Dij}(G))$ time. Moreover, each vertex v also explicitly maintains a count $c(v) = |U_v|$ of the uncovered shortest paths containing v. For $r, v \in V$, the number of shortest paths starting at r and containing v is equal to the size of the subtree of T_r rooted at v. We can initialize all $c(\cdot)$ in $O(n^2)$ total time by adding the subtree sizes for all r.

Now consider an iteration in which a vertex v is selected. It hits several previously uncovered paths, and we must update the data structures accordingly. Consider a tree T_r. In the beginning of the iteration, it represents all uncovered paths that start at r. The ones that contain v are exactly those represented in the subtree of T_r rooted at v. We delete this subtree by setting to *null* the parent pointers of all of its vertices. Since each vertex can be removed from each tree once, the total cost of all subtree deletions over the entire algorithm is $O(nm)$.

While traversing the subtree, we also compute its size δ_r. Then we traverse the path in T_r from v to r and for every vertex w on the path, we subtract δ_r from $c(w)$. The total traversal cost (over all iterations of the algorithm) is $O(n^3)$ in the worst case (although much faster in practice). For provably better bounds, we need a more careful implementation of these *path updates*.

Lemma 2. *A* TD *iteration can update the* $c(\cdot)$ *values in* $O(\mathrm{Dij}(G))$ *time.*

Proof. Consider an iteration in which a vertex v is selected as the most important. For a vertex w, let Δ_w be the amount by which $c(w)$ must eventually be decreased. From the trivial algorithm above, Δ_w is the sum of δ_u over all u such that w is an ancestor of v in T_u. (Recall that δ_u is the size of the subtree of T_u rooted at v.) We claim we can compute all Δ_w's without explicitly traversing these paths. Consider the set S of all vertices whose $c(\cdot)$ values have to be updated; S is exactly the set of vertices in the union of the u–v paths on all trees T_u that contain v. Since all these paths end at v (and shortest paths are unique), their union is a shortest path tree I_v rooted at v. Also, it is easy to see that Δ_w is the sum of δ_u over all descendants u of w in I_v (including w itself). These values can thus be computed by a bottom-up traversal of I_v (from the leaves to the root v). The overall bottleneck is building I_v, which takes $O(\mathrm{Dij}(G))$ time. $\square$

The last aspect we need to consider is the time to select the next vertex in each iteration. For TDc, a simple $O(n)$ traversal of all $c(v)$ values suffices. For TDwc, we also need to know the values s_v and t_v for each candidate vertex v. The value of s_v is $|T_v|$, which we can maintain explicitly and update in $O(n)$ time per iteration. We keep t_v by maintaining the in-trees T'_v analogous to the out-trees T_v. Then $t_v = |T'_v|$. Maintaining the out-trees has no effect on the asymptotic complexity of the algorithm. We have the following result:

Theorem 2. *The* TDc *and* TDwc *algorithms can be implemented to run in* $\Theta(n\mathrm{Dij}(G))$ *time and* $\Theta(n^2)$ *space.*

The $\Theta(n^2)$ time and space requirements limit the size of the problems one can solve in practice. We are careful to minimize constant factors in space in our implementation. This is why we recompute subtree sizes instead of maintaining them explicitly. Moreover, we do not store distances within the trees (which we do not need); we only need the topology, as defined by parent pointers. We store the parent pointer of a node with in-degree $\deg(v)$ with only $\lceil \log_2(\deg(v)+1)\rceil$ bits, which is enough to decide which incoming arc—if any—is the parent. To find the children of a vertex v, we examine its neighbors w and check if v is the parent of w. Since we only look for the children of each vertex in each tree once (right before the vertex is deleted), the total time spent on traversals is $O(mn)$. Note that the input size (n and m) fully determines the running time and space consumption of TD. This is not the case for CH, which can be much faster or slower than TD depending on the graph topology and cost function.

Range Optimization. Although TD yields better orderings then CH, its $\Theta(n^2)$ space and time requirements limit its applicability. We now discuss how to combine ideas from TD and CH to obtain better orderings for large graphs.

An obvious approach is to run CH preprocessing until the contracted graph is small enough, then use TD to order the remaining vertices. This idea has been used with the non-incremental implementation of TDc, and indeed improves the ordering [2], even though it can only optimize the most important vertices.

To improve the ordering among other vertices, we propose a *range optimization* algorithm. It takes an ordering r and parameters X and Y as input, and reorders all vertices v with $X < r(v) \leq Y$. It first shortcuts all vertices v with $r(v) \leq X$, creating a graph G' with $n - X$ vertices. It then runs Dijkstra's algorithm from each v in G' to compute all shortest paths U that are not covered by vertices w with $r(w) > Y$. The search from v is responsible for paths starting at v, and can stop as soon as all vertices in the priority queue have at least one vertex w with $r(w) > Y$ as an ancestor. Finally, we run TDwc with G' and U as input. Note that we only need to store a *partial* tree for each vertex v. If X and Y are chosen appropriately, the trees are small and can be stored explicitly.

This algorithm reoptimizes a range within a given vertex ordering. Intuitively, more important vertices in the range move up and less important ones move down. To allow a vertex to move between arbitrary positions, we use an *iterative range optimization algorithm.* It covers the interval $[1, n]$ by k overlapping intervals $[X_i, Y_i]$ with $X_1 = 0$, $Y_k = n$, $X_i < Y_i$, and $X_{i+1} \leq Y_i$. The algorithm starts with some vertex ordering, e.g., the one given by CH. It then proceeds in k steps, each reordering a different interval. In practice, it pays to process the intervals in decreasing order of importance; this is faster than processing them in increasing order, and the resulting labels are at least as small.

5 Computing the Labels

We now discuss how to build a canonical labeling from an ordering efficiently. As defined in Section 3, we must add the maximum-rank vertex on each shortest path P_{st} to the labels $L_f(s)$ and $L_b(t)$.

The most straightforward approach is to use Dijkstra to build a shortest path tree out of each vertex s. We then traverse the tree from s to the leaves, computing for each vertex v the maximum-rank vertex w on the s–v path. We add $(w, \text{dist}(s, w))$ to $L_f(s)$ and $(w, \text{dist}(s, v) - \text{dist}(s, w))$ to $L_b(v)$, if not already in the labels. Note that, if we use TD to order vertices, we can incorporate this approach and compute labels on the fly. However, the $O(n\text{Dij}(G))$ running time of this approach makes it infeasible for large networks.

A previous approach [2] is based on CH. Given G^+, we construct $L_f(s)$ for a given vertex s as follows ($L_b(s)$ is computed analogously). Run Dijkstra's algorithm from s in G^+ pruning arcs (u, v) with $r(u) > r(v)$ and add all scanned vertices to $L_f(s)$. To make the labeling canonical we apply label pruning (Section 3). Note that when pruning the label of v, we need labels of higher-ranked vertices, which we achieve by computing labels from high to low rank vertices.

We now introduce a *recursive* label-generation procedure, which is more efficient. It borrows from CH the shortcut operation, but not the query. Given a graph G_i where all vertices u with $r(u) < i$ are shortcut, and an ordering r, we pick the lowest-rank vertex v (with $r(v) = i$) and shortcut it, obtaining the graph G_{i+1}. Then we recursively compute the labels in G_{i+1}. (The basis of the recursion is G_n, with a single vertex s, when we just set $L_f(s) = L_b(s) = \{(s, 0)\}$.) To extend the labeling to v, we *merge* the labels of its neighbors. We show how to construct $L_f(v)$; the construction of $L_b(v)$ is symmetric. We initialize $L_f(v)$ with $(v, 0)$ and then, for every arc (v, w) in G_i and for every pair $(x, d_w(x)) \in L_f(w)$, we add to $L_f(v)$ a pair $(x, d_w(x)+\ell(v, w))$. If the same hub x appears in the labels of multiple neighbors w, we keep only the pair that minimizes $d_w(x) + \ell(v, w)$. Since labels are sorted by hub ID, we build the merged label by traversing all neighboring labels in tandem, as in mergesort.

Theorem 3. *The recursive algorithm computes a correct hierarchical labeling for an ordering r.*

We can make the labeling computed by this procedure canonical by pruning each label immediately after it is generated, as in the CH-based approach.

6 Building Contraction Hierarchies

A vertex ordering determines a canonical labeling. It also determines a contraction hierarchy (G^+): simply shortcut the vertices according to this order, which may take up to $O(n^3W)$ time. We give an $O(n\mathrm{Dij}(G))$ algorithm that does not use shortcutting. The key observation is that a shortcut (v, w) is added by CH preprocessing if and only if v and w are the two highest-ranked vertices on P_{vw}.

Given an ordering r, we first compute all shortcuts (u, v) with $r(u) < r(v)$. To do so, we run Dijkstra's algorithm from each vertex u of the graph. Whenever we scan a vertex v, we check whether v is the first vertex on the path from u to v with $r(v) > r(u)$. If so, we add (u, v) to G^+. We can stop the search as soon as all vertices v in the priority queue have an ancestor w with $r(w) > r(u)$. The shortcuts (v, u) with $r(v) > r(u)$ can be computed analogously. This algorithm builds G^+ in $O(n\mathrm{Dij}(G))$ time.

Given a small canonical labeling, we can compute the shortcuts for G^+ even faster. We use $L_f(u)$ to compute all shortcuts (u, v) with $r(u) < r(v)$ as follows. For each pair $(v, d) \in L_f(u)$, we check whether there is at least one other pair $(v', d') \in L_f(u)$ such that $d = d' + \mathrm{dist}(v', v)$. If there is none, we add (u, v) to G^+. Note that we need to run a HL query for $\mathrm{dist}(v', v)$. We compute (v, u) with $r(v) < r(u)$ from $L_b(u)$ analogously. The overall running time of this approach is $O(nM^3)$, since for each label we must run $O(M^2)$ queries, each in $O(M)$ time. In practice, using the HL one-to-many query [11] may accelerate this approach.

7 Experiments

We implemented CH, HL, and TD in C++ and compiled them with Visual C++ 2010, using OpenMP for parallelization. We use a 4-ary heap as priority queue.

The experiments were conducted on a machine with two Intel Xeon X5680 CPUs and 96 GB of DDR3-133 RAM, running Windows 2008R2 Server. Each CPU has 6 cores (3.33 GHz, 6 x 64 kB L1, 6 x 256 kB L2, and 12 MB L3 cache). Preprocessing uses all 12 cores, but queries are sequential. Our implementation of CH follows [19] and uses $E_q(u) + O_q(u) + lev(u)$ as priority function. $E_q(u)$ is the *edge quotient* (number of shortcuts added divided by the number of elements removed if u were shortcut); $O_q(u)$ is the *original edges quotient* (number of original arcs in the shortcuts added divided by the number of original arcs in the shortcuts removed); and $lev(u)$ is the level of u. Initially, $lev(u) = 0$ for each vertex u. Whenever a vertex v is shortcut, the level of all its neighbors v in the graph with the shortcuts added so far is set to $\max\{lev(v), lev(u) + 1\}$. We implement HL queries as in [2], with no compression unless mentioned otherwise.

We first consider the orderings produced by TDwc on various inputs: road networks [13], meshes [21], communication and collaboration networks [5], and artificial inputs (Delaunay triangulations, random geometric graphs, small-world graphs [20, 22]). For each graph, Table 1 shows its size ($|V|$), its density ($|A|/|V|$), the TDwc running time (TIME), the average resulting label size (L_a), and HL query time (QT). We then report the ratio of shortcuts to original arcs ($|A^+|/|A|$) in the graph G^+ produced from this order. Finally, we report query times and number of scanned vertices for both CH and bidirectional Dijkstra queries (BD).

The results show that TDwc is very efficient for many graphs. Moreover, HL is practical for a wide variety graphs, with small labels and queries in microseconds or less. In fact, HL queries are always faster than BD, even when

Table 1. TDwc labels on five groups of instances: meshes, road networks, artificial graphs, communication networks, and collaboration graphs. QT is the query time.

				HL		CH			BD	
INSTANCE	$\lvert V\rvert$	$\lvert A\rvert/\lvert V\rvert$	TIME [s]	SIZE L_a	QT [μs]	$\lvert A^+\rvert/\lvert A\rvert$	VERT SCANS	QT [μs]	VERT SCANS	QT [μs]
face	12530	5.8	20	48.8	0.3	2.2	143	55	3653	598
feline	20629	6.0	70	66.8	0.6	2.6	211	104	4529	810
horse	48485	6.0	362	108.4	0.8	2.8	355	250	13371	2154
bay-d	321270	2.5	14274	44.5	0.4	1.1	119	33	107813	13097
bay-t	321270	2.5	12739	29.3	0.3	1.0	78	17	93829	12352
G_pin_pout	99995	10.0	1833	8021.6	66.5	196.4	13166	1165430	354	160
smallworld	100000	10.0	2008	5975.0	49.7	102.8	9440	612682	523	203
rgg_18	262085	10.0	10694	225.6	2.2	1.0	723	668	85186	31612
del_18	262144	6.0	12209	97.1	0.7	2.0	304	183	49826	10818
klein_18	262144	6.0	10565	1718.7	12.8	13.6	4135	50844	4644	1590
as-22july06	22963	4.2	86	22.5	0.2	1.2	70	30	63	125
caidaRouter	190914	6.4	7399	343.8	3.1	2.7	2081	6527	377	298
astro-ph	14845	16.1	40	231.2	2.1	2.4	798	1529	151	112
cond-mat	36458	9.4	249	293.3	2.7	2.9	1145	2661	220	134
preferential	100000	10.0	1727	586.4	4.5	11.3	2265	10941	198	215
coAuthors	299067	6.5	19488	789.3	6.9	3.7	4047	26127	530	378

labels are large. Due to better locality (it just merges two arrays), HL can still be slightly faster even when BD scans much fewer vertices than there are hubs in the labels, as in smallworld and G_n_pin_pout. The fact that labels are large for some graphs is not surprising, given known lower bounds [16].

CH queries are always slower than HL, and usually faster than BD. In several cases, however, CH queries are worse than BD in terms of both running times and number of scanned vertices. CH has a weaker termination condition, cannot balance the two sides of the search space, and works on a denser graph.

As predicted, TDwc preprocessing time depends mostly on the network size and not on its structure (unlike CH preprocessing). TDwc preprocessing uses $\Theta(n^2)$ space, limiting the size of the graphs we can handle in memory to a few hundred thousand vertices; at under six hours, running times are still acceptable.

Table 2 compares the TDwc orderings to those computed by TDc and CH preprocessing. For the latter, we give ordering time, number of shortcuts in G^+, CH query search space, and label size. All values in the table are relative to those obtained from the TDwc ordering. Compared to TDwc, TDc produces larger labels (by 1% to 8%) and more shortcuts; since TDwc is not much slower, it is usually a better choice. Compared to TDwc, the CH ordering produces bigger labels (by 16% to 44%) and the CH query search space increases by 8% to 31%. Interestingly, this happens despite the fact that CH preprocessing adds fewer shortcuts (up to 30%). CH preprocessing is much faster than TDwc when G^+ is sparse, but much slower otherwise. In several cases, it did not finish in six hours.

We now consider the effects of range optimization on the road network of Western Europe [13] (18 million vertices and 42 million arcs) with travel times.

Table 2. Performance of orderings obtained from CH and TDc, relative to TDwc.

	CH-preprocessing				TDc			
instance	time	$\|A^+\|$	#sc	L_a	time	$\|A^+\|$	#sc	L_a
face	0.082	0.93	1.11	1.17	0.785	1.06	1.05	1.04
feline	0.115	0.95	1.11	1.16	0.651	1.07	1.05	1.05
horse	0.135	0.97	1.16	1.23	0.644	1.07	1.05	1.05
bay-d	0.000	0.88	1.31	1.36	0.589	1.04	1.06	1.06
bay-t	0.000	0.88	1.29	1.30	0.849	1.05	1.06	1.05
G_pin_pout	DNF	–	–	–	0.808	1.12	0.99	1.02
smallworld	DNF	–	–	–	0.752	1.09	1.00	1.02
rgg_18	0.045	0.96	1.16	1.21	0.948	1.07	1.05	1.08
del_18	0.003	0.94	1.08	1.16	0.521	1.06	1.04	1.05
klein_18	DNF	–	–	–	0.989	1.06	1.00	1.02
as-22july06	31.021	0.96	1.29	1.44	0.668	1.00	0.98	1.01
caidaRouter	3.623	0.70	1.17	1.23	0.798	1.01	0.97	1.02
astro-ph	56.873	0.79	1.25	1.20	0.689	1.04	1.00	1.03
cond-mat	24.771	0.81	1.20	1.19	0.658	1.05	1.00	1.03
preferential	DNF	–	–	–	0.826	1.15	0.98	1.06
coAuthors	DNF	–	–	–	0.761	1.09	0.99	1.05

Table 3 reports the label size after applying range optimization multiple (i) times, using the ranges $[n - 2^{17}, n]$, $[n - 2^{20}, n - 2^{15}]$, $[n - 2^{22}, n - 2^{17}]$, and $[0, n - 2^{20}]$. As initial ordering ($i = 0$), we use the one given by CH. We report the *total* time (in seconds) to obtain this ordering and the performance of CH queries. Each iteration takes about 70 minutes. The first one reduces label sizes by 25%, but later ones have a smaller impact. The effect on CH queries is similar. After five iterations the label size is stable. Experiments on smaller inputs indicate that the final ordering is close to the one obtained by TDwc.

Table 3. Range opt.

i	TIME	L_a	#SC	[μs]
0	108	95.52	288	96.4
1	4546	73.17	213	80.1
2	8925	70.32	206	77.8
3	12737	69.58	203	75.5
4	16730	69.17	201	74.2
5	20606	69.01	200	73.4
6	24512	69.01	200	73.4

For Western Europe, we also compare various versions of our algorithm with our previous HL implementations [2], Contraction Hiearchies, transit-node routing (TNR) [6], and its combination with arc-flags (TNR+AF) [7]. HL-0 uses pure CH preprocessing (label size 97.6), HL-15 uses TDwc on the topmost 32768 vertices (size 78.3), HL-17 optimizes the top 131072 (size 75.0), while HL-∞ uses five iterations of range optimization (size 69.0). In all cases, building G^+ takes 105 s, generating the labels takes 55 s, and the remaining time is spent on improving the ordering. As explained in [2], the "local" version uses 8/24 compression and an index, whereas the "global" version is index free with a partition oracle (cell size 20 000).

Table 4. Performance of various algorithms. HL and CH preprocessing use 12 cores, others are sequential.

method	prepro [h:m]	space [GB]	query [μs]
CH [17]	0:02	0.4	96.381
TNR [7]	0:58	3.7	1.775
TNR+AF [7]	2:00	5.7	0.992
HL local [2]	2:39	20.1	0.572
HL global [2]	2:45	21.3	0.276
HL-0 local	0:03	22.5	0.700
HL-15 local	0:05	18.8	0.556
HL-17 local	0:25	18.0	0.545
HL-∞ local	5:43	16.8	0.508
HL-∞ global	6:12	17.7	0.254

We observe that our new techniques (faster label generation, incremental TDwc) accelerate preprocessing by two orders of magnitude. With longer preprocessing, we improve the best previous query times (HL global [2]) by 8%. Preprocessing for HL-0 and HL-15 is fast enough for real-time traffic updates on large metropolitan areas, even when taking turn costs into account [10].

8 Conclusion

Our study of hierarchical labelings makes HL practical on a wider class of problems. The work raises several natural questions. It would be interesting to study the relationship between hierarchical labelings, which are more practical, and general hub labelings, which have theoretical guarantees on the label size. In preliminary experiments, we ran Cohen et al.'s algorithm [9] on significantly smaller graphs (given its $O(n^4)$ running time). It produces labels that are about as good as those created by TDwc for some graph classes (such as road networks), but asymptotically smaller for others (like small-world graphs). Another open question is how to efficiently compute (or approximate) optimal hierarchical labelings, or a good lower bound on their size. Finally, we would like to reduce the space consumption of HL further, beyond existing compression schemes [2].

Acknowledgments. We thank Ruslan Savchenko for implementing [9].

References

1. I. Abraham, D. Delling, A. Fiat, A. V. Goldberg, and R. F. Werneck. VC-Dimension and Shortest Path Algorithms. *ICALP*, LNCS 6755, 690–699, 2011.
2. I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. A Hub-Based Labeling Algorithm for Shortest Paths on Road Networks. *SEA*, LNCS 6630, 230–241, 2011.
3. I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. Hierarchical Hub Labelings for Shortest Paths. MSR-TR-2012-46, Microsoft Research, 2012.
4. 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.
5. D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner. 10th DIMACS Implementation Challenge – Graph Partitioning and Graph Clustering, 2011.
6. H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes. In Transit to Constant Shortest-Path Queries in Road Networks. In *ALENEX*, pp. 46–59M, 2007.
7. 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 J. of Exp. Algo.*, 15(2.3):1–31, January 2010.
8. J. Cheng and J. X. Yu. On-line Exact Shortest Distance Query Processing. In *EDBT*, pp. 481–492. ACM Press, 2009.
9. E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and Distance Queries via 2-hop Labels. *SIAM J. Comput.*, 32, 2003.
10. D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Customizable Route Planning. In *SEA*, LNCS 6630, 376–387, 2011.
11. D. Delling, A. V. Goldberg, and R. F. Werneck. Faster Batched Shortest Paths in Road Networks. In *ATMOS*, OASIcs 20, pp. 52–63, 2011.
12. D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering Route Planning Algorithms. In *Algo. of Large and Complex Networks*, LNCS 5515, 117–139, 2009.
13. C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, DIMACS 74. AMS, 2009.
14. E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.
15. M. L. Fredman and R. E. Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *J. Assoc. Comput. Mach.*, 34:596–615, 1987.
16. C. Gavoille, D. Peleg, S. Pérennes, and R. Raz. Distance Labeling in Graphs. *Journal of Algorithms*, 53:85–112, 2004.
17. R. Geisberger, P. Sanders, D. Schultes, and C. Vetter. Exact Routing in Large Road Networks Using Contraction Hierarchies. *Transportation Science*, 2012.
18. A. V. Goldberg. A Practical Shortest Path Algorithm with Linear Expected Time. *SIAM Journal on Computing*, 37:1637–1655, 2008.
19. T. Kieritz, D. Luxen, P. Sanders, and C. Vetter. Distributed Time-Dependent Contraction Hierarchies. In *SEA*, LNCS 6049, pp. 83–93, 2010.
20. J. Kleinberg. Navigation in a Small World. *Nature*, 406:845, 2000.
21. P. V. Sander, D. Nehab, E. Chlamtac, and H. Hoppe. Efficient Traversal of Mesh Edges Using Adjacency Primitives. *ACM Trans. on Graphics*, 27:144:1–144:9, 2008.
22. D. Watts and S. Strogatz. Collective Dynamics of "Small-World" Networks. *Nature*, 393:409–410, 1998.