Hub Labels: Theory and Practice

Daniel Delling[1], Andrew V. Goldberg[1], Ruslan Savchenko[2]*, and Renato F. Werneck[1]

[1] Microsoft Research, 1065 La Avenida, Mountain View CA 94043, USA
{dadellin,goldberg,renatow}@microsoft.com
[2] Department of Mech. and Math., Moscow State University
1 Leninskiye Gory, Moscow 119991, Russia
ruslan.savchenko@gmail.com

Abstract. The Hub Labeling algorithm (HL) is an exact shortest path algorithm with excellent query performance on some classes of problems. It precomputes some auxiliary information (stored as a *label*) for each vertex, and its query performance depends *only* on the label size. While there are polynomial-time approximation algorithms to find labels of approximately optimal size, practical solutions use hierarchical hub labels (HHL), which are faster to compute but offer no guarantee on the label size. We improve the theoretical and practical performance of the HL approximation algorithms, enabling us to compute such labels for moderately large problems. Our comparison shows that HHL algorithms scale much better and find labels that usually are not much bigger than the theoretically justified HL labels.

1 Introduction

In this paper we study the *hub labeling* (HL) algorithm [12], a powerful technique to answer exact point-to-point shortest path queries with excellent performance on some real-world networks. Although HL has been studied both theoretically and experimentally [2–6, 8, 21], in practice one uses fast heuristics instead of the algorithms with theoretical solution quality guarantee. Since the heuristics are used on real-life networks [2–5], it is important to gauge their solution quality and their potential for improvement.

The theoretically justified algorithms have escaped extensive experimental studies in the past because, although polynomial, they do not scale well. Our goal is to speed up the algorithms without losing the theoretical guarantees. Even if the algorithms do not scale as well as the heuristics, experiments on moderate-size problems give a useful measure of solution quality. A small gap would justify the heuristics, while a large gap would motivate their improvement.

During preprocessing, labeling algorithms [19] compute a *label* for every vertex of the graph and answer s–t shortest path queries using only the labels of s and t (and not the graph itself). HL is a labeling algorithm where the label $L(v)$ of v is a collection of vertices (*hubs* of v) with distances between v and the hubs.

* Part of this work done while at Microsoft Research

The label $L(v)$ consists of two parts, the *forward label* $L_f(v)$ and the *backward label* $L_b(v)$. The labels must 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. Given the labels, HL queries are straightforward: to find $\text{dist}(s,t)$, simply find the hub $v \in L_f(s) \cap L_b(t)$ that minimizes $\text{dist}(s,v) + \text{dist}(v,t)$. The *size* of the label is its number of hubs. The memory footprint of the algorithm is dominated by the sum of all label sizes, while query times are determined by the maximum label size. To measure solution quality, we mostly use the average label size, which is equivalent to the sum of all forward and backward label sizes; in practice, the maximum label size is not much higher than the average.

Finding the smallest HL for general graphs is NP-hard. Cohen et al. [8] give an $O(n^5)$-time $O(\log n)$-approximation algorithm for minimizing the size of the labeling. Babenko et al. [6] generalize this result to an $O(\log n)$-approximation algorithm for general p-norms (as defined in Section 2); we refer to their algorithm as GHL^p.

A special case of HL is *hierarchical hub labeling* (HHL), where vertices are globally ranked by "importance" and the label for a vertex can only have more important hubs than itself. HHL labels can be polynomially bigger than HL labels for some graph classes [14], but small HHL labels exist for other classes, such as trees. For general graphs, finding the smallest HHL is NP-hard [15], and no polylog-approximation algorithm is known. Practical heuristics for computing HHL have been studied in [3, 5, 9].

Known approximation algorithms for HL [1, 6, 8], although polynomial, have high time bounds. Cohen et al. [8] describe an implementation of their algorithm with a speedup based on lazy evaluation, but it is still slow and can only handle small problems. For reachability queries (a special case with zero arc lengths), Schenkel et al. [21] implement this algorithm with a different variant of lazy evaluation, and use divide-and-conquer to handle larger problems. Other implementations focus on HHL and have no theoretical guarantees. The implementations of Abraham et al. [2, 3] work well on large road networks and some other networks of moderate size. The implementation of Akiba et al. [5] scales to large complex networks. Delling et al. [9] produce small labels for a wider range of inputs than either method.

Our Contributions. In this paper, we improve the GHL^p algorithm and study its practical performance. First, we propose a refinement of GHL^p that achieves an $O(n^3 \min(p, \log n) \log n)$ time bound and is more efficient in practice; for $p = 1$, this improves the bound of Cohen et al. from $O(n^5)$ to $O(n^3 \log n)$. Second, we investigate the tradeoff between average and maximum sizes for the labels computed by GHL^p for different values of p. Finally, our detailed experimental analysis confirms that GHL^1 produces smaller labels than HHL.

The preprocessing algorithms we study require $\Omega(n^2)$ memory and compute shortest path distances between all pairs of vertices. In principle, one could instead compute and store a distance table and answer queries by table lookup. For graphs for which HL queries work well, however, the labels are small. Thus one can run preprocessing on a large server, but run queries on a less powerful device. Furthermore, studying more sophisticated algorithms with large memory footprint allows us to judge the quality of more practical heuristics and potentially improve them.

2 Preliminaries

The input to the point-to-point shortest path problem is a directed graph $G = (V, A)$, a length function $\ell : A \to \mathcal{R}$, and a pair s, t of vertices. Let $n = |V|$ and $m = |A|$. The goal is to find $\mathrm{dist}(s, t)$, the length of the shortest s–t path in G, where the length of a path is the sum of the lengths of its arcs. We assume that the length function is non-negative and that there are no zero-length cycles.

The size of a forward (backward) label, $|L_f(v)|$ ($|L_b(v)|$), is the number of hubs it contains. We define the size of the full label $L(v) = (L_f(v), L_b(v))$ as $|L(v)| = (|L_f(v)| + |L_b(v)|)/2$. We generalize this definition as follows. Suppose vertex IDs are $1, 2, \ldots, n$. Define a $(2n)$-dimensional vector $\mathcal{L}$ by $\mathcal{L}_{2i-1} = |L_f(i)|$ and $\mathcal{L}_{2i} = |L_b(i)|$. We consider the ℓ_p norm of $\mathcal{L}$, defined as $\|\mathcal{L}\|_p = (\sum_{i=0}^{2n-1} \mathcal{L}_i^p)^{1/p}$, where p is a natural number or ∞; $\|\mathcal{L}\|_\infty = \max_i \mathcal{L}_i$. The hub labeling algorithm uses $O(\|\mathcal{L}\|_1)$ memory and has worst-case query time $O(\|\mathcal{L}\|_\infty)$.

3 Approximation Algorithms

We now discuss existing $O(\log n)$-approximation algorithms for $\|\mathcal{L}\|_p$. We start with Cohen et al.'s algorithm [8] for $p = 1$; Section 3.1 deals with arbitrary p.

Starting with an empty labeling, the algorithm in each iteration adds a vertex to some labels, until the labeling satisfies the cover property. The algorithm also maintains the set U of *uncovered* vertex pairs: $(u, w) \in U$ if $L_f(u) \cap L_b(w)$ does not contain a vertex on a shortest u–w path. Initially U contains all vertex pairs u, w such that w is reachable from u. The algorithm terminates when U becomes empty. In each iteration, the algorithm adds a vertex v to forward labels of $u \in S' \subseteq V$ and to backward labels of $w \in S'' \subseteq V$ such that ratio of the number of newly-covered paths over the total increase in the size of the labeling is (approximately) maximized. Formally, let $U(v, S', S'')$ be the set of pairs in U which become covered if v is added to $L_f(u) : u \in S'$ and $L_b(w) : w \in S''$. The algorithm maximizes $|U(v, S', S'')|/(|S'| + |S''|)$ over all $v \in V$ and $S', S'' \subseteq V$.

A *center graph* of v, $G_v = (X, Y, A_v)$, is a bipartite graph with $X = V$, $Y = V$, and an arc $(u, w) \in A_v$ if some shortest path from u to w goes through v. To do the maximization, the algorithm finds densest subgraphs of center graphs. The *density* of a graph $G = (V, A)$ is $|A|/|V|$. The *maximum density subgraph (MDS)* problem can be solved in polynomial time using maximum flows (e.g., [11]). For a vertex v, arcs of a subgraph of G_v induced by $S' \subseteq X$ and $S'' \subseteq Y$ correspond to the pairs of vertices in U that become covered if v is added to $L_f(u) : u \in S'$ and $L_b(w) : w \in S''$. Therefore, the MDS of G_v maximizes $|U(v, S', S'')|/(|S'| + |S''|)$ over all S', S''.

Cohen et al. [8] show that the use of a linear-time 2-approximation algorithm [18] instead of an exact MDS algorithm results in a faster algorithm while preserving the $O(\log n)$ approximation ratio. We refer to the *approximate* MDS problem as AMDS. The 2-AMDS algorithm works by iteratively deleting the minimum-degree vertex from the current graph, starting from the input graph and ending in a single-vertex graph. Kortsarz and Peleg [18] show that the subgraph of maximum density in the resulting sequence of subgraphs is a 2-AMDS. To find the desired triple (v, S', S''), Cohen et al. solve the AMDS problem for all $G_v : v \in V$, and take v that gives the approximately densest subgraph.

Zero-Weight Vertex Heuristic. Cohen et al. [8] note that G_v can have arcs (u, w) such that the label of one endpoint contains v but the other does not. In this case, if u is included in a subgraph, it contributes to the denominator of its density, even though we do not need to add v to $L_f(u)$. They propose an intuitive modification of the algorithm that maintains the $O(\log n)$ approximation guarantee but performs better in practice. They assign zero weight to a vertex $u \in X$ if $v \in L_f(u)$ and to a vertex $w \in Y$ if $v \in L_b(w)$ and unit weights to all other vertices. They also modify the 2-AMDS algorithm to repeatedly delete vertices minimizing the ratio of degree over weight, where $x/0 = \infty$. Zero-weight vertices will be removed last; in fact, the algorithm can stop as soon as it reaches such a vertex.

3.1 Optimizing Arbitrary Norms

The $\|\mathcal{L}\|_p$ algorithm for $p > 1$ by Babenko et al. [6] is similar to the case $p = 1$, but uses weighted MDS, with vertex weights determined by the current labeling. The *maximum weighted densest subgraph* problem, takes as input a graph with vertex weights $c(v) \geq 0$. The goal is to find a subgraph of maximum weighted density, defined as the number of arcs divided by the total weight of the vertices. The 2-AMDS algorithm generalizes to this case: instead of choosing a vertex with minimum degree, each iteration chooses a vertex v that minimizes the ratio between the degree and the weight $c(v)$.

Consider an iteration of GHL^p with labels L_f and L_b. For the center graphs, we define the weight of a vertex $u \in X$ by $c_p(u) := (|L_f(u)| + 1)^p - |L_f(u)|^p$ and for $w \in Y$, $c_p(w) := (|L_b(w)| + 1)^p - |L_b(w)|^p$. For $p = 1$ we get the standard (unweighted) MDS problem. The $O(\log n)$-approximation algorithm for $\|\mathcal{L}\|_p$ is similar to the algorithm for $\|\mathcal{L}\|_1$, except that each iteration finds the triple (v, S', S'') that maximizes $|U(v, S', S'')|/(c_p(S') + c_p(S''))$. A 2-approximation for such triple (v, S', S'') is sufficient, and can be found by running a weighted 2-AMDS algorithm for all center graphs G_v.

The observation that for $p \geq \log n$, the ℓ_∞ norm is within a constant factor of the ℓ_p norm yields a $\log(n)$-approximation algorithm for ℓ_∞.

4 Improved Time Bound

The time bound for the Cohen et al. [8] algorithm (GHL^1) is $O(n^5)$: each iteration computes n AMDSes on graphs with $O(n^2)$ arcs, and the number of iterations is bounded by $O(n^2)$, the worst-case size of the labeling. A naive implementation of GHL^1 maintains all center graphs and uses $O(n^3)$ space, but this can be reduced to $O(n^2)$. We propose an *eager-lazy* variant of GHL^1 that runs in $O(n^3 \log n)$ time and $O(n^2)$ space, and still achieves an $O(\log n)$ approximation ratio. It also extends to an $O(n^3 \min(p, \log n) \log n)$ implementation of GHL^p.

First we describe our data structures. We precompute, in $O(n\text{Dij}(n,m))$ time, a table of all pairs of shortest path distances. ($\text{Dij}(n,m)$ denotes the running time of Dijkstra's algorithm [10] on a network with n vertices and m arcs.) We maintain an $n \times n$ Boolean array, with bit (i,j) indicating whether the current labeling covers the pair i,j. These data structures use $O(n^2)$ space.

Eager evaluation is a variant of the 2-AMDS algorithm that guarantees that deleting the edges of an approximate densest subgraph reduces the MDS bound for the remaining graph. We use this fact to bound the number of times the algorithm selects the center graph of a vertex. Consider a graph G, let μ be an upper bound on its MDS value, and fix $\alpha \geq 1$. The *α-eager evaluation* repeatedly deletes the minimum degree vertex of G while the density of G is less than $\mu/(2\alpha)$. If we use the zero-weight heuristic, we do not remove zero-weight vertices. Let G' be the subgraph that remains at the end of this procedure. Let G_α be the subgraph of G induced by the vertices deleted during this process. Let $\tilde{G}$ be the graph G with all edges from G' deleted; if we use the zero-weight heuristic, we also change the weight of vertices of G' from one to zero.

Lemma 1. *During the algorithm, (1) the density of G' is at least $\mu/(2\alpha)$ and (2) the MDS value of $\tilde{G}$ is at most μ/α.*

Proof. By construction, we stop when the density of G is at least $\mu/(2\alpha)$, which implies (1). For a vertex v in G_α, let m_v be the number of edges adjacent to v when v was deleted from G. We have $m_v < \mu/\alpha$, since v is the smallest-degree vertex before the deletion and $m_v \geq \mu/\alpha$ would imply a $\mu/(2\alpha)$ lower bound on the graph density. Consider a subgraph H of $\tilde{G}$ and let H_α be H with G' deleted. Let m_h and n_h be the number of edges in H and vertices in H_α respectively. If we use zero-weight heuristic, m_h/n_h is the density of H; otherwise, it is an upper bound. We have $m_h \leq \sum_{v \in H_\alpha} m_v < n_h \cdot \mu/\alpha$. Therefore $m_h/n_h < \mu/\alpha$. □

Note that G' may be empty, in which case (μ/α) is an improved bound on the MDS value of $G = G_\alpha$.

Next we discuss *lazy evaluation.* Cohen et al. [8] observed that the MDS value of a center graph does not increase when edges are removed. They thus propose keeping a priority queue of AMDS values and processing the center graph corresponding to the maximum value. In addition, they maintain a marker for each center graph indicating whether it has changed since its last AMDS computation. Each iteration processes the subgraph with maximum value. If the AMDS value is invalid (outdated), they recompute it. Otherwise they augment the labeling according to this subgraph, delete the AMDS of the center graph used in this iteration, and mark the affected AMDS values as invalid.

Schenkel et al. [21] use a variant of lazy evaluation. Instead of maintaining the AMDSes, they maintain their density values. An iteration considers the maximum value, d_v, and recomputes an AMDS of G_v. If the density of this AMDS is at least d_v,[3] this subgraph is used to update the labels and the center graphs. Otherwise d_v is updated.

Combining the second variant of lazy evaluation with eager evaluation, we get our *eager-lazy labeling* version of GHL^1. Let $\alpha > 1$ be a constant. During initialization, the algorithm computes upper bounds μ_v on the MDS values of the center graphs using the 2-AMDS algorithm and keeps the values in a max-heap Q. At each iteration, the algorithm extracts the maximum μ_v from the heap and applies α-eager evaluation to G_v. If eager evaluation finds a non-empty subgraph G', it adds v to the labels of the vertices of G'. Then it updates U (the set of uncovered pairs) by iterating over all uncovered pairs of vertices (u, w) and, if $v \in L_f(u)$, $v \in L_b(w)$ and $\text{dist}(u, v) + \text{dist}(v, w) = \text{dist}(u, w)$, it marks (u, w) as covered. By Lemma 1 we know that after the update the MDS value of G_v is at most μ_v/α. The same bound holds if G' is empty. Finally, we set $\mu_v = \mu_v/\alpha$ and add μ_v back to Q. (One can use the bound found by the 2-AMDS algorithm, if it is smaller.)

Theorem 1. *The eager-lazy variant of GHL^1 is an $O(\log n)$ approximation algorithm running in $O(n^3 \log n)$ time and $O(n^2)$ space.*

Proof. The results of [8] imply that, for an $O(\log n)$ approximation, one can use constant factor approximations of the maximum density subgraph instead of the exact solutions. When the algorithm adds v to the labels, G' is a 2α approximation. The space bound is $O(n^2)$, since it is dominated by the distance table and the coverage matrix U. For the time bound, note that each iteration performs eager evaluation. In addition, if we find a non-empty G', we iterate over all vertex pairs to mark the newly covered ones, taking $O(n^2)$ time overall. To bound the number of iterations, note that each iteration that considers G_v decreases μ_v by a factor of α. For a graph with at least one edge, the maximum subgraph density is between $1/2$ and n. Therefore each G_v can be selected $O(\log n)$ times, yielding an $O(n \log n)$ bound on the number of iterations. $\square$

Now consider GHL^p for $p > 1$. The MDSes of the center graphs are monotonically non-increasing, so we can use lazy evaluation. To use eager evaluation, we generalize the α-eager evaluation algorithm to the weighted case. Let μ be an upper bound on the weighted density and fix $\alpha > 1$. Define the *score* of a vertex to be its degree divided by its weight. The algorithm repeatedly deletes the minimum-score vertex from the graph while the graph density if less than μ/α. As before, let G' be the graph left when the α-eager evaluation terminates, and let $\tilde{G}$ be G with the edges from G' deleted and the weights of vertices from G' adjusted if we use zero-weight heuristic. We generalize Lemma 1 as follows:

Lemma 2. *During the algorithm, (1) the weighed density of G' is at least $\mu/(2\alpha)$ and (2) the weighted MDS value of $\tilde{G}$ is at most μ/α.*

[3] Although the MDS value of G_v is monotonically decreasing, an approximate computation may find a subgraph with higher value than the previous one.

We defined the weight of a vertex $u \in X$ by $c_p(u) = (|L_f(u)|+1)^p - |L_f(u)|^p$ and for $w \in Y$, $c_p(w) = (|L_b(w)| + 1)^p - |L_b(w)|^p$. The maximum weighted subgraph density is $O(n)$ and the minimum non-zero density is $\Omega(1/n^p)$, so the density range is $O(n^{p+1})$. If an iteration of GHL^p chooses G_v, the density of G_v is reduced by a factor of α, yielding an $O(p \log n)$ bound on the number of times G_v is chosen. For $p > \log n$, $\|L\|_p = O(\|L\|_{\log n})$, and we get an $O(\log^2 n)$ bound.

Theorem 2. *The eager-lazy variant of GHL^p is an $O(\log n)$ approximation algorithm runs in $O(n^3 \min(p, \log n) \log n)$ time and $O(n^2)$ space.*

The implementation of GHL^p does not assume shortest path uniqueness. We could apply an a-priori length function perturbation to make the paths unique and center graphs less dense. We do not know how to improve theoretical bounds in this case; experimental results appear in Section 5.

4.1 Practical Improvement

The main bottlenecks of an iteration of our algorithm are updating the set U after adding a vertex to the labels and determining the set of center graph arcs adjacent to a vertex in the eager evaluation and AMDS subroutines. We propose a *shortest path graph* heuristic to speed up these operations in practice.

Let SPO_v be the graph induced by the arcs on shortest paths out of v. Similarly, SPI_v is induced by the arcs on shortest paths into v. Both SPO_v and SPI_v are acyclic (we assume no zero cycles). If shortest paths are unique, SPO_v and SPI_v are trees. We maintain these graphs implicitly: arc (u, w) is in SPO_v if $\text{dist}(v, u) + \ell(u, w) = \text{dist}(v, w)$, and in SPI_v if $\ell(u, w) + \text{dist}(w, v) = \text{dist}(u, v)$.

Suppose we add v to $L_f(u) : u \in S'$ and to $L_b(w) : w \in S''$. Then a pair $(u, w) : u \in S', w \in S''$ is covered if v is on a u–w shortest path. So we can iterate over all $u \in S'$ and perform a DFS of SPO_u starting at v. For each vertex w visited by the DFS, if $w \in S''$ we mark the pair (u, w) as covered.

We also use the *SPO* graphs to find the set of outgoing arcs from a vertex u in G_v (the case of incoming arcs is similar, using the *SPI* graphs). Again, we perform a DFS on SPO_u starting at v. When visiting a vertex w, we know that (u, w) is an arc of G_v if the pair u, w is not covered.

Although a DFS takes $O(m)$ time in the worst case, it tends to visit a small fraction of the graph, leading to a speedup in practice.

5 Experiments

We implemented all algorithms in this paper in C++ and compiled them with Microsoft Visual C++ 2012. We conducted our experiments on a machine running Windows Server 2008 R2. It has 96 GiB of DDR3-1333 RAM and two 6-core Intel Xeon X5680 3.33 GHz CPUs, each with 6×64 KB of L1, 6×256 KB of L2, and 12 MB of shared L3 cache. All our executions are single-threaded.

We test our algorithm on a wide range of realistic and synthetic instances. We test road networks from the 9th DIMACS Implementation Challenge [13]:

Table 1. Running times and label sizes for GHL[1] with $\alpha = 1.0, 1.1, 1.5$ and HHL.

			time (s)				average label			
instance	$\|V\|$	$\|A\|$	G1.0	G1.1	G1.5	HHL	G1.0	G1.1	G1.5	HHL
PGPgiant	10680	48632	19113.8	3338.9	1721.2	969.4	19.1	19.4	20.3	20.5
alue5067	3524	11120	2970.8	2486.1	1796.0	158.6	23.4	24.5	25.4	24.1
beethoven	2521	15090	607.5	336.4	219.8	39.3	25.4	26.0	27.9	26.4
berlin10k	10370	24789	16026.8	8648.8	6168.2	1102.7	20.5	21.3	23.4	25.7
berlin5k	5307	12640	2475.1	1494.6	1082.7	191.5	18.0	18.6	20.2	21.6
email	1133	10902	108.8	46.9	22.6	4.4	30.0	30.4	31.4	36.8
grid10	961	3720	77.5	68.2	46.7	3.9	18.6	19.2	20.0	18.2
grid12	3969	15624	7191.2	5861.7	4038.8	276.3	27.9	28.7	29.6	27.6
hep-th	5835	27630	6374.6	1479.4	716.1	344.3	38.7	39.2	41.2	47.3
ksw-32-1	1024	6118	47.9	26.1	14.8	2.8	42.8	43.5	45.3	58.6
ksw-45-1	2025	12090	320.3	154.7	87.9	17.7	59.1	59.6	62.0	84.9
ksw-64-1	4096	24482	2319.4	900.9	489.3	124.2	81.4	82.3	84.9	126.0
polblogs	1222	33428	375.5	144.9	74.4	8.2	25.2	25.5	26.4	29.1
power	4941	13188	696.2	387.8	318.3	69.8	13.7	13.9	14.9	14.0
rgg10u	993	6162	42.5	32.4	24.6	4.0	14.0	14.4	15.0	14.6
rgg10w	993	6162	27.6	18.6	14.1	3.5	15.4	15.9	17.4	17.2
rgg12u	4088	31746	3494.6	2599.5	1853.5	300.2	24.3	25.2	27.2	26.3
rgg12w	4088	31746	1959.9	1182.2	795.8	103.1	29.9	31.0	34.5	36.7
rome99	3353	8859	587.6	399.6	269.2	41.2	23.8	24.9	27.3	28.4
venus	2838	17016	977.9	557.7	365.8	57.9	27.3	28.0	29.9	28.1

rome99, berlin5k, and berlin10k (the latter two are subgraphs from the Western Europe network). From the 10th DIMACS Implementation Challenge [7], we consider various complex networks: PGPgiant (communication), email (social), hep-th (collaboration), polblogs (links), and power (power grid). From the SteinLib [17], we take a grid graph with holes from VLSI applications (alue5067). We also consider triangulations from Computer Graphics applications [20] (beethoven and venus). Finally, we generated synthetic instances representing random geometric graphs with unit (rgg-u) and Euclidean (rgg-w) edge lengths, square grids (grid), and Kleinberg's small world graphs [16] (ksw). A ksw graph with parameter N consists of an $N \times N$ toroidal grid with additional long-distance edges: for each vertex, we add an edge to a random vertex at Manhattan distance d in the grid, where the probability of picking a particular value of d is proportional to d^{-2}.

Road networks are weighted and directed; rgg-w are weighted and undirected; all others are unweighted and undirected. All instances are (strongly) connected, although all algorithms still work otherwise.

Table 1 gives preprocessing times and average label sizes for our GHL[1] algorithm for $\alpha = 1.0$, 1.1, 1.5 (denoted by Gα). For comparison, we also include the HHL implementation of Delling et al. [9] that produces the best label quality; it uses on-line tie-breaking and runs in $O(n^3)$ time. Although there are much faster variants of HHL [9, 3], they produce slightly worse labels. For example,

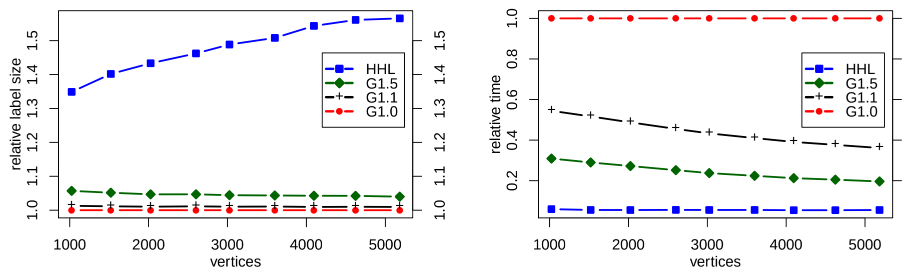

Fig. 1. Label size and running time (relative to G1.0) on small-world (ksw) problems as a function of input size.

the default algorithm of Delling et al. [9] takes less than 4 seconds to find labels for a road network with 32 413 vertices representing a bigger region of Berlin.

The table shows that increasing α from 1.0 (which produces the same results of Cohen et al.'s algorithm) to 1.1 barely degrades label quality and results in speedups ranging from less than two (for most graphs tested) to more than six (for complex networks such as hep-th). Increasing α to 1.5 leads to further speedups with more noticeable degradation of solution quality. In fact, G1.5 labels are sometimes bigger than HHL labels. Better approximations of the maximum density subgraph lead to smaller labels, as the theory predicts.

HHL is much faster than GHL^1, but usually produces worse labels. The difference is negligible for some graph classes (such as grids and triangulations), but HHL labels are larger by 20% for road networks and up to 50% for larger small-world instances (ksw).

Figure 1 analyzes the ksw family in more detail. It reports the average label sizes and running times of HHL, G1.1, and G1.5 relative to G1.0. Each data point represents five instances with different random seeds. The figure shows that HHL produces asymptotically worse labels than GHL^1. Similarly, G1.1 and G1.5 have slight asymptotic advantage over G1.0 in terms of running times. Although the version of HHL we test has very similar asymptotic complexity to G1.0, recall that much faster (often asymptotically so in practice) versions of HHL [9] (with slightly worse labels) exist.

Figure 2 examines the tradeoff between preprocessing time and quality for GHL^1 in more detail. Each instance is represented by 6 points, each corresponding (from left to right) to a distinct value of α: 1.0, 1.05, 1.1, 1.2, 1.5, and 2.0. For each instance, all results are relative to $\alpha = 1.0$: increasing α leads to higher speedups but larger labels. For complex networks, such as email, hep-th, and PGPgiant, the tradeoff is favorable: speedups of an order of magnitude lead to labels that are bigger by less than 15%. For other classes, the tradeoff is not as good, with even small speedups leading to non-trivial losses in quality. Given that, we use $\alpha = 1.1$ (a relatively small value) for the remainder of the experiments.

We now compare the quality of the labels computed by the GHL^p algorithm for different values of p: we use $p = 1, 2, 4, \lceil \log n \rceil$. Recall that GHL^p approxi-

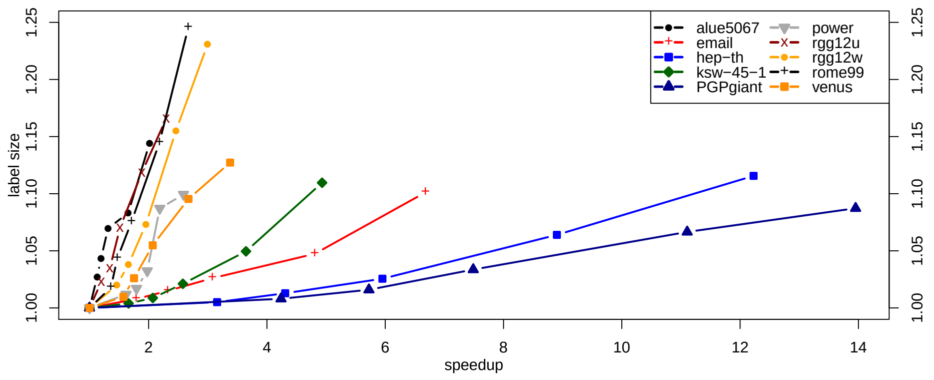

Fig. 2. Tradeoff between label size and running time. Each curve considers six values of α (1.0, 1.05, 1.1, 1.2, 1.5, and 2.0, from left to right); all values are relative to $\alpha = 1.0$.

mates the ℓ_p norm, and for $p = \lceil \log n \rceil$ it also approximates the ℓ_∞-norm. For select instances, Table 2 shows the average time as well as the average and maximum label sizes for each norm. The $\mathsf{L}_1\mathsf{L}_\infty$ problems were designed by Babenko et al. [6] to show that there can be a large gap between the best labelings according to ℓ_1 and ℓ_∞ norms, and indeed we see a big difference between GHL^1 and GHL$^{\lceil \log n \rceil}$ labels. For other problems, ℓ_1 and ℓ_2 labels have similar average and maximum sizes, with no clear winner. As p increases further, the average label size increases and the maximum size decreases. Intuitively, this is because large labels contribute more to the p-norm when p increases. Overall, however, the difference in either measure is not overwhelming. GHL^p is slower for larger p values, often by a large amount. These results suggest that, except in pathological cases, using $p > 1$ will increase running time but will not result in substantially different label norms.

Table 2. GHL^p performance for different p values ($\alpha = 1.1$)

	time (s)				average label				maximum label			
instance	L_1	L_2	L_4	L_∞	L_1	L_2	L_4	L_∞	L_1	L_2	L_4	L_∞
$\mathsf{L}_1\mathsf{L}_\infty$-13	8.8	21.6	30.7	28.6	3.8	7.5	13.7	14.1	171	171	33	21
$\mathsf{L}_1\mathsf{L}_\infty$-16	30.9	88.3	122.9	123.5	3.8	9.0	16.6	17.2	258	258	42	25
alue5067	2486.1	3930.6	6398.1	9415.8	24.5	24.4	24.4	25.9	41	39	37	37
berlin5k	1494.6	2661.6	4597.0	6697.7	18.6	18.6	19.0	20.1	40	37	36	34
email	46.9	93.3	159.3	222.5	30.4	30.5	31.6	32.9	59	52	49	46
grid10	68.2	147.9	191.1	298.9	19.2	19.3	19.7	20.2	29	27	29	27
hep-th	1479.4	2607.9	5939.2	7609.9	39.2	39.4	40.7	43.6	88	85	72	67
ksw-45-1	154.7	337.2	716.2	1110.4	59.6	59.4	60.1	61.0	82	76	74	70
polblogs	144.9	281.4	442.7	614.5	25.5	25.7	27.0	28.0	86	62	54	52
power	387.8	602.2	977.8	1491.9	13.9	14.0	14.3	15.1	28	30	29	29
rgg12u	2599.5	4440.9	7876.7	11976.3	25.2	25.1	25.7	26.9	45	45	42	42
rgg12w	1182.2	2234.8	3965.4	6226.2	31.0	30.8	31.0	32.2	56	50	51	48
rome99	399.6	769.5	1620.2	2534.8	24.9	24.7	25.4	26.8	50	47	43	44
venus	557.7	945.7	1725.4	2602.4	28.0	27.8	28.3	29.3	46	44	40	41

Table 3. Results on perturbed instances; all values are relative to the corresponding unperturbed instance.

	speedup				relative label size			
Instance	G1.0	G1.1	G1.5	HHL	G1.0	G1.1	G1.5	HHL
alue5067	4.63	5.27	5.51	3.17	1.30	1.29	1.34	1.38
berlin5k	0.99	1.01	1.00	1.00	1.00	1.00	1.00	1.00
email	3.12	2.96	2.84	1.77	1.61	1.60	1.61	1.61
grid10	4.93	5.61	5.50	2.21	1.25	1.24	1.28	1.35
hep-th	1.94	1.86	1.83	1.99	1.40	1.40	1.40	1.44
ksw-45-1	1.70	1.84	1.88	1.88	1.33	1.34	1.33	1.29
polblogs	4.98	5.00	5.41	1.94	1.89	1.89	1.88	2.06
power	1.03	1.16	1.29	1.13	1.08	1.09	1.06	1.09
rgg12u	1.91	2.36	2.51	2.35	1.19	1.19	1.22	1.32
rgg12w	1.01	1.02	1.01	1.01	1.00	1.00	1.00	1.00
rome99	1.01	1.00	1.00	0.99	1.00	1.00	1.00	1.00
venus	1.64	2.04	2.27	2.21	1.34	1.34	1.35	1.41

The algorithms we have tested so far perform on-line tie-breaking: they consider that a vertex v covers a pair s, t if there exists at least one shortest s–t path that contains v. To confirm that this leads to better labels than committing to one specific s–t path in advance, we ran the same algorithms on *perturbed* versions of some instances, with arc lengths increased by up to 1% at random, independently and uniformly. This mostly preserves the shortest path structure, but makes ties much less likely.

For each perturbed instance and algorithm, Table 3 shows the speedup (in terms of label generation time) and the average label size *relative to the original instance*. For graphs that were originally weighted (berlin5k, rgg12w, and rome99), all algorithms behave almost exactly as before. For the remaining inputs (in which ties are much more common), all algorithms become consistently faster, but produce worse labels. For example, all algorithms become about twice as fast on hep-th, but labels become 40% larger; for email, labels increase by as much as 50%. This confirms that, for best label quality, HL should be used with on-line tie-breaking.

6 Concluding Remarks

Our improvements to the running time and our compact data structures for the theoretically justified HL algorithm allow us to solve bigger problems than previously possible: instances with about 10 000 vertices can be solved in an hour. Our results provide some justification for using hierarchical labels (HHL) in practice: on real-world instances, HHL labels are not much bigger than those found by the theoretically justified (non-hierarchical) algorithms, and HHL is usually much faster. That said, the difference in quality (sometimes higher than 20%) is not negligible; faster approximation algorithms would still be quite useful.

References

1. Abraham, I., Delling, D., Fiat, A., Goldberg, A.V., Werneck, R.F.: VC-dimension and shortest path algorithms. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011. LNCS, vol. 6755, pp. 690–699. Springer, Heidelberg (2011)
2. Abraham, I., Delling, D., Goldberg, A.V., Werneck, R.F.: A hub-based labeling algorithm for shortest paths on road networks. In: Pardalos, P.M., Rebennack, S.(eds.) SEA 2011. LNCS, vol. 6630, pp. 230–241. Springer, Heidelberg (2011)
3. Abraham, I., Delling, D., Goldberg, A.V., Werneck, R.F.: Hierarchical hub labelings for shortest paths. In: Epstein, L., Ferragina, P. (eds.) ESA 2012. LNCS, vol. 7501, pp. 24–35. Springer, Heidelberg (2012)
4. Akiba, T., Iwata, Y., Kawarabayashi, K.I., Kawata, Y.: Fast shortest path distance queries on road networks by pruned highway labeling. In: ALENEX 2014, pp. 147–154, SIAM, Philadelphia (2014).
5. Akiba, T., Iwata, Y., Yoshida, Y.: Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In: SIGMOD 2013. pp. 349–360. ACM, New York (2013)
6. Babenko, M., Goldberg, A.V., Gupta, A., Nagarajan, V.: Angorithms for hub label optimization. In: Fomin, F., Freivalds, R., M., K., Peleg, D. (eds.) ICALP 2013. pp. 69–80. LNCS vol. 7965, Springer, Heidelberg (2013)
7. Bader, D., Meyerhenke, H., Sanders, P., Wagner, D. (eds.): Graph Partitioning and Graph Clustering. AMS, Boston (2013)
8. Cohen, E., Halperin, E., Kaplan, H., Zwick, U.: Reachability and distance queries via 2-hop labels. SIAM J. Comput. 32(5), 1338–1355 (2003)
9. Delling, D., Goldberg, A.V., Pajor, T., Werneck, R.F.: Robust exact distance queries on massive networks. TR MSR-TR-2014-12, Microsoft Research (2014)
10. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numer. Math. 1, 269–271 (1959)
11. Gallo, G., Grigoriadis, M.D., Tarjan, R.E.: A fast parametric maximum flow algorithm and applications. SIAM J. Comput. 18, 30–55 (1989)
12. Gavoille, C., Peleg, D., Pérennes, S., Raz, R.: Distance labeling in graphs. J. Algorithms 53(1), 85–112 (2004)
13. Goldberg, A.V., Kaplan, H., Werneck, R.F.: Reach for A*: shortest path algorithms with preprocessing. In: The Shortest Path Problem: Ninth DIMACS Implementation Challenge, DIMACS Book, vol. 74, pp. 93–139. AMS, Boston (2009)
14. Goldberg, A.V., Razenshteyn, I., Savchenko, R.: Separating hierarchical and general hub labelings. In: Chatterjee, K., Sgall, J. (eds.) MFCS 2013, LNCS, vol. 8087, pp. 469–479. Springer, Heidelberg (2013)
15. Kaplan, H.: Personal communication (2013)
16. Kleinberg, J.M.: The small-world phenomenon: An algorithmic perspective. In: STOC 2000. pp. 163–170, ACM, New York (2000)
17. Koch, T., Martin, A., Voß, S.: SteinLib: An updated library on Steiner tree problems in graphs. Tech. Rep. ZIB-Report 00-37, Konrad-Zuse-Zentrum für Informationstechnik Heidelberg (2000), http://elib.zib.de/steinlib
18. Kortsarz, G., Peleg, D.: Generating sparse 2-spanners. J. Alg. 17, 222–236 (1994)
19. Peleg, D.: Proximity-preserving labeling schemes. J. Gr. Th. 33(3), 167–176 (2000)
20. Sander, P.V., Nehab, D., Chlamtac, E., Hoppe, H.: Efficient traversal of mesh edges using adjacency primitives. ACM Trans. Graphics 27(5), 144:1–144:9 (2008)
21. Schenkel, R., Theobald, A., Weikum, G.: HOPI: An efficient connection index for complex XML document collections. In: Bertino, E. et al. (eds.) Advances in Database Technology — EDBT 2004. pp. 237–255. Springer, Heidelberg (2004)