Robust Distance Queries on Massive Networks

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

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

Abstract. We present a versatile and scalable algorithm for computing exact distances on real-world networks with tens of millions of arcs in real time. Unlike existing approaches, preprocessing and queries are practical on a wide variety of inputs, such as social, communication, sensor, and road networks. We achieve this by providing a unified approach based on the concept of 2-hop labels, improving upon existing methods. In particular, we introduce a fast sampling-based algorithm to order vertices by importance, as well as effective compression techniques.

1 Introduction

Answering point-to-point distance queries in graphs is a fundamental building block for many applications [21] in social networks, search, computational biology, computer networks, and road networks. Dijkstra's algorithm [22] can answer such queries in almost linear time, but this can take several seconds on large graphs. This motivates two-phase algorithms, in which auxiliary data computed during *preprocessing* is used to accelerate on-line *queries*. Although there are practical exact algorithms for road networks [2, 8, 13] and some social and communication graphs [3–5, 17, 18, 23], none is robust on a wide range of inputs.

We propose an exact algorithm that is much more robust to network structure, scales to large networks, and improves (or at least is competitive with) existing specialized solutions. Our method is based on *hierarchical hub labeling* (HHL) [3], a special kind of 2-hop labeling [11]. HHL preprocessing first *orders* vertices by importance, then transforms this ordering into *labels* that enable fast exact shortest-path distance queries (either in RAM or in external memory [1, 17, 20]). Labels can be optionally compressed with no loss in correctness.

While there are fast algorithms to transform an ordering into the corresponding labeling [3, 4], finding a good ordering is challenging. Heuristics that are effective on road networks [3, 13] or on unweighted, undirected small-diameter networks [4] are not robust on other inputs. Compression strategies are similarly specialized to particular networks [4, 10]. We close both gaps by introducing efficient algorithms to find good orders (Section 3) and compress the resulting labels (Section 4) on a wide variety of inputs, including some which no other known method can handle. Our experiments (Section 5) show that our methods are robust, scaling to graphs with tens of millions of arcs. We answer queries to optimality within microseconds using significantly less auxiliary data than previous approaches, effectively widening the range of inputs that can be dealt with efficiently. Details omitted from this extended abstract can be found in the full version [9].

2 Background

The input to the distance query problem is a directed graph $G = (V, A)$ with a positive length function $\ell : A \to \mathbb{Z}_{>0}$. Let $n = |V|$ and $m = |A|$. We denote the length of a shortest path (or the *distance*) from vertex v to vertex w by $\text{dist}(v, w)$. A *distance query* takes a pair of vertices (s, t) as input and outputs $\text{dist}(s, t)$.

A *labeling algorithm* [19] preprocesses the graph to compute a *label* for every vertex such that an s–t query can be answered using only the labels of s and t. The *2-hop labeling* or *hub labeling* (HL) algorithm [11] is a special case with a two-part label $L(v)$ for every vertex v: a *forward label* $L_f(v)$ and a *backward label* $L_b(v)$. (For undirected graphs, each vertex stores a single label that acts as both forward and backward.) The forward label $L_f(v)$ is a sequence of pairs $(w, \text{dist}(v, w))$, with $w \in V$; similarly, $L_b(v)$ has pairs $(u, \text{dist}(u, v))$. Vertices w and u are said to be *hubs* of v. To simplify notation, we often interpret labels as sets of hubs; $v \in L_f(u)$ thus means label $L_f(u)$ contains a pair $(v, \text{dist}(u, v))$. The size $|L(v)|$ of a forward or backward label is its number of hubs. A *labeling* is the set of labels for all $v \in V$ and its size is $\sum_v (|L_f(v)| + |L_b(v)|)$. The *average label size* is the size of the labeling divided by $2n$. Labels must obey the *cover property*: for any 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. We do not assume that shortest paths are unique; to avoid confusion, we mostly refer to (ordered) *pairs* $[u, w]$ instead of paths u–w. We say that a vertex v *covers* (or *hits*) a pair $[u, w]$ if $\text{dist}(u, v) + \text{dist}(v, w) = \text{dist}(u, w)$, i.e., if at least one shortest u–w path contains v.

To find $\text{dist}(s, t)$, an HL query finds the hub $v \in L_f(s) \cap L_b(t)$ that minimizes $\text{dist}(s, v) + \text{dist}(v, t)$. If the entries in each label are sorted by hub ID, this takes linear time by a coordinated sweep over both labels, as in mergesort.

Our focus is on *hierarchical hub labelings* (HHL). Given a labeling, let $v \lesssim w$ if w is a hub of $L(v)$. Abraham et al. [3] define a hub labeling as hierarchical if $\lesssim$ is a partial order. (Intuitively, $v \lesssim w$ if w is "more important" than v.) Natural heuristics for finding labelings produce hierarchical ones [3, 4, 13, 17].

Abraham et al. [3] show that one can compute the smallest HHL consistent with a given ordering $rank(\cdot)$ on the vertices in polynomial time. In this *canonical labeling*, vertex v belongs to $L_f(u)$ if and only if there exists w such that v is the highest-ranked vertex that hits $[u, w]$. Similarly, v belongs to $L_b(w)$ if and only if there exists u such that v is the highest-ranked vertex that hits $[u, w]$. Although canonical labelings were originally defined under the assumption that shortest paths are unique [3], the same definition holds when they are not [14]. The algorithms by Abraham et al. [2, 3] to compute a labeling from a given order are polynomial, but impractical for most graph classes.

More recently, Akiba et al. [4] proposed the *Pruned Labeling* (PL) algorithm, which efficiently computes a labeling from a given vertex order. (We will use it as a subroutine.) Starting from empty labels, PL processes vertices from most to least important (higher to lower *rank*). The iteration that processes vertex v adds v to all relevant labels. To process v, it runs two pruned versions of Dijkstra's algorithm [22] from v. The first works on the forward graph (out of v) as follows. Before scanning a vertex w (with distance label $d(w)$ within Dijkstra's algorithm),

it computes a v–w distance estimate q by performing an HL query with the current partial labels. (If the labels do not intersect, set $q = \infty$.) If $q \leq d(w)$, the $[v, w]$ pair is already covered by previous hubs and the algorithm prunes the search (ignores w). Otherwise (if $q > d(w)$), it adds $(v, \text{dist}(v, w))$ to $L_b(w)$ and scans w as usual. The second Dijkstra computation uses the reverse graph and is pruned similarly; it adds $(v, \text{dist}(w, v))$ to $L_f(w)$ for all scanned vertices w. Note that the number of Dijkstra scans equals the size of the labeling. Also, rather than assuming shortest paths are unique, PL breaks ties on-line in favor of more important (higher-ranked) vertices. Akiba et al. show that PL is correct and produces a minimal labeling (deleting any hub violates the cover property). It is easy to show that it is also hierarchical, and thus canonical.

3 Computing Orderings

Knowing how to efficiently compute a hierarchical labeling from an order, we now consider how to find orders that lead to small labelings. (Recall that any order produces correct labels.) One can sidestep this issue by using the order implied by vertex degrees [4, 17]; using degree as a proxy for importance works well for some unweighted and undirected small-diameter networks, but is not robust. *Contraction Hierarchies* (CH) [3] orders vertices bottom-up, using only local information that is carefully updated as decisions are made. This often leads to small labels [3], but can be costly because the number of updates may be superlinear and the updates themselves may be expensive.

For better results, Abraham et al. [3] propose a greedy top-down algorithm. It finds good labels for a wide range of graph classes, but is too expensive (in both time and space) for large instances. In this section, we recap this *basic algorithm* and then show that using on-line tie breaking leads to even better orders, but with greater preprocessing effort. Finally, we propose a sampling technique that makes the basic algorithm much faster while still finding good solutions.

Basic Algorithm. The basic algorithm [3] defines the order greedily: the i-th highest ranked vertex (hub) is the one that hits the most previously uncovered shortest paths (i.e., not covered by the $i - 1$ hubs already picked). To implement this rule efficiently, the basic algorithm starts by building n full shortest path trees, one rooted at each vertex of the graph. The tree T_s rooted at s represents all uncovered shortest paths starting at s. This effectively makes shortest paths unique: the algorithm assumes that only vertices on the s–t path in T_s can hit the pair $[s, t]$. The number of descendants of v in T_s is thus the number of uncovered shortest paths that start at s and contain v. The total number of descendants of v over all trees, denoted by $\sigma(v)$, is the number of shortest paths that would be hit if v were picked as the next most important hub.

Each iteration of the algorithm picks as the next hub the vertex v^* for which $\sigma(v^*)$ is maximum. To prepare for the next iteration, it removes the subtree rooted at v^* from each tree (as the paths they represent are now covered by v^*) and updates the $\sigma(\cdot)$ values of all descendants and ancestors of v^*. This

algorithm is *path-greedy*: it maximizes the number of new paths hit in each iteration. Abraham et al. also propose a *label-greedy* variant, which picks the vertex v^* that maximizes the ratio between $\sigma(v^*)$ and the number of labels to which v^* will be added; this leads to slightly better labels. Note that the basic algorithm breaks ties off-line (a-priori) while computing the initial trees; the resulting paths determine not only which hub to select next, but also to which labels this hub is added. Our experiments thus refer to it as OffPG (path-greedy) or OffLG (label-greedy). Both variants run in $O(mn \log n)$ time [3].

Better Tie-breaking. When shortest paths are far from unique (as in some unweighted small-diameter networks), the basic algorithm underestimates the number of pairs hit by each hub it picks. Since it breaks ties a-priori, it produces bigger labels than needed. For better results, we propose a simple hybrid algorithm: first compute a vertex order using the basic algorithm, then use PL to find the labeling. Since PL breaks ties in favor of paths with the highest maximum vertex rank, this can lead to substantially smaller labels with little overhead. We refer to this algorithm as HybPG (path-greedy) or HybLG (label-greedy).

We also propose an algorithm that breaks ties *on-line* while selecting the order. Although impractical for large instances, it finds the smallest hierarchical labels we are aware of (on moderate-sized inputs). It follows the same approach as the basic algorithm, picking in each iteration the hub v^* (not picked before) that covers the most uncovered pairs $[u, w]$. The challenge is finding v^* efficiently in every iteration: since ties are broken on-line, we must implicitly maintain the numbers of descendants in all shortest path DAGs, which is harder than in trees.

Thus, during initialization, we compute an $n \times n$ distance table between all n vertices and create an $n \times n$ boolean matrix in which entry (u, w) indicates whether the pair $[u, w]$ is already covered by a previously selected hub. All entries $[u, w]$ with finite $\text{dist}(u, w)$ are initially false. For each vertex v, we maintain $\sigma(v)$, the number of new pairs that would be covered if v were selected as the next hub. Initially, this is the total number of pairs $[u, w]$ hit by v. For a fixed v, this value can be found in $O(n^2)$ time: check for each $[u, w]$ if $\text{dist}(u, v) + \text{dist}(v, w) = \text{dist}(u, w)$.

Each iteration of the algorithm is as follows. First, pick a vertex v for which $\sigma(v)$ is maximum (in $O(n)$ time). Then find the set Q of uncovered pairs hit by v (note that $|Q| = \sigma(v)$); this takes $O(n^2)$ time using the distance table and the boolean matrix. For each pair $[u, w] \in Q$, mark $[u, w]$ as covered and add v to both $L_f(u)$ and $L_b(w)$ (if not there already). Finally, update the other σ values: for each pair $[u, w] \in Q$ and vertex $x \in V$, decrease $\sigma(x)$ by one if x hits $[u, w]$. This step takes $O(|Q|n)$ time. Since any pair appears in some Q at most once during the algorithm, the combined size of all Q lists is $O(n^2)$. Altogether, the algorithm runs in $O(n^3)$ worst-case time and $\Theta(n^2)$ space. This *path-greedy* algorithm can be extended to be *label-greedy* with the same bounds. We call these variants OnPG and OnLG, respectively.

Finding Good Orderings Faster. All methods considered so far are impractical for large graphs, since they use $\Omega(n^2)$ space and time. We thus propose

an improvement of the path-greedy hybrid algorithm (HybPG) that uses sampling to compute *estimates* $\tilde{\sigma}(\cdot)$ on the $\sigma(\cdot)$ values. The estimates need only be precise enough to distinguish important vertices (those for which $\sigma(\cdot)$ is large) from unimportant ones. We can tolerate fairly large errors on the estimate of unimportant vertices. Intuitively, if $\sigma(v) \gg \sigma(w)$, we want to have $\tilde{\sigma}(v) > \tilde{\sigma}(w)$.

A natural approach is to build $k \ll n$ trees from random roots and set $\tilde{\sigma}(v)$ to be the total number of descendants of v in all trees of the sample [7]. (Sampling paths uniformly would be ideal, but too costly.) Once a vertex v^* is picked (from a priority queue), we update counters as in the basic algorithm, with all descendants of v^* removed from the sampled trees. Unfortunately, when k is small (as required for good performance), such $\tilde{\sigma}(v)$ estimates are only accurate for very important vertices (with many descendants in most trees); as sampled trees get smaller, we have insufficient information to assess less important vertices.

We deal with this by generating more trees (from new roots) as the algorithm progresses. We grow them using Dijkstra's algorithm, but pruning vertices already covered by previously picked hubs (like in PL). Newly added trees thus only contain uncovered paths and get smaller as the algorithm progresses, keeping space and time under control. Since we need partial labels for pruning, we add v^* to all relevant labels (running one PL iteration from v^*) right after v^* is selected as next hub. We balance the work spent growing trees and constructing (adding hubs to) labels. Let c_t be the total number of arcs and hubs touched so far while building new trees (the k original trees are free); define c_l similarly, for operations during label construction. We generate trees from random new roots until either $c_t > c_l$ or the total number of vertices in existing trees exceeds $10kn$. To bound the space usage, we represent small trees as hash tables.

Although the total number of descendants in the sample is a natural estimator for the total over all n trees, its variance is very high. In particular, it overestimates the importance of vertices that are at (or near) the root of a sampled tree [12]. Replacing the sum (or average) by a more robust measure (such as the median) would remedy this, but is costly to maintain as trees (and counters) are updated. We achieve both robustness and speed as follows. Instead of keeping a single counter $\tilde{\sigma}(v)$ for each vertex v, we keep c counters $\tilde{\sigma}_1(v), \tilde{\sigma}_2(v), \ldots, \tilde{\sigma}_c(v)$, for some constant c. Counter $\tilde{\sigma}_i(v)$ is the total number of descendants of v over all trees t_j such that $i = (j \bmod c)$. (Here t_j is the j-th tree in the sample, not the tree rooted at j.) These counters are easy to maintain and allow us to eliminate outliers when evaluating v, for instance by discarding the counter i that maximizes $\sigma_i(v)$ and taking as estimator the average value of the remaining counters. (Intuitively, if v is close to the root of one tree, only one counter will be affected.) In general, increasing c improves accuracy, but can be costly because the priority of a vertex depends on all its c counters. We found that using $c = 16$ and discarding the two highest counters gives good results with negligible overhead. In case of ties, we prefer vertices maximizing $\tilde{\sigma}(v) = \sum_{i=1}^{c} \tilde{\sigma}_i(v)$. Moreover, we ensure at least c trees are live during the execution. We call this ordering algorithm SamPG. We have no label-greedy variant of this algorithm, as it is unclear how to obtain good estimates on the number of labels a hub is added to.

4 Compression

Representing labels compactly is crucial for large graphs. We first show how to represent distances or IDs with fewer bits without sacrificing query times, then propose a more elaborate technique that exploits similarities across labels, trading higher compression for slower (but still exact and fast enough) queries.

Basic Compression. Recall that a label $L_f(u)$ can be seen as an array of pairs $(v, \text{dist}(u,v))$ sorted by hub ID v. In practice [2], it pays to first represent all hubs, then the corresponding distances (in the same order). Since distances are only read when hubs match, queries have fewer cache misses. We represent distances with as few bits (8, 16, or 32) as needed for the largest distance stored in any label. (For unweighted small-diameter networks, 8 bits are enough [4].)

Less trivially, one can use fewer bits to represent hub IDs. Abraham et al. [2] *rename* the hubs so that IDs 0 to 255 are assigned to the most important (higher-ranked) vertices, and use only 8 bits to represent them (and 32 bits otherwise). On road networks, space is reduced by around 10% (and queries become faster), since many hubs in each label are in this set. For greater effectiveness on more inputs, we propose two improvements: delta representation and advanced reordering.

Delta representation stores hub IDs in difference form. Let the hub IDs in a label be $h_1 < h_2 < h_3 < \ldots$ We store h_1 explicitly, but for every $i > 1$ we store $\Delta_i = h_i - h_{i-1} - 1$. A label with hubs `(0 16 29 189 299 446 529)` is thus represented as `(0 15 12 159 109 146 82)`. Because queries always traverse labels in order, we can retrieve h_i as $\Delta_i + h_{i-1} + 1$. Since $\Delta_i < h_i$, this increases the range of entries that can be represented with fewer bits. (In the example above, 8 bits suffice for all entries.) To keep queries simple, we avoid variable-length encoding. Instead, we divide the label into two blocks: we start with 8 bits per entry, and switch to 32 bits when needed.

Our second technique is to rename vertices to increase the number of 8-bit hub entries. We could reorder hubs by rank (as in Abraham et al. [2]) or by frequency, with smaller IDs assigned to hubs that appear in more labels, but we can do even better (by about 10%) with *advanced reordering*. We assign ID 0 to the most frequent vertex and allocate additional IDs (up to $n-1$) to one vertex at a time. For each vertex v that is yet unassigned, let $s(v)$ be the number of labels in which v could be represented with 8 bits if v were given the smallest available ID. Initially, $s(v)$ is the number of labels containing v, but its value may decrease as the algorithm progresses. Each iteration of our method picks the vertex v with maximum $s(v)$ value and assigns an ID to it. If multiple available IDs are equally good (i.e., realize $s(v)$), we assign v the *maximum* ID among those, saving smaller IDs for other vertices. In particular, the second most frequent vertex could have any ID between 1 and 256 and still be represented as 8 bits, so it gets ID 256.

The main challenge for advanced reordering is efficiently updating the $s(\cdot)$ values. Our lazy implementation keeps a priority queue with estimated $\tilde{s}(\cdot)$ values. Each iteration picks the maximum such element $\tilde{s}(v)$ and computes the actual $s(v)$ value. If the estimate is approximately correct, we assign an ID to v; otherwise, we reinsert v into the queue with $\tilde{s}(v) \leftarrow s(v)$.

Token-Based Compression. We now present a novel scheme to achieve even higher compression. It extends *hub label compression* (HLC) [10], which interprets each label as a tree and represents each unique subtree (which may occur in many labels) only once. We explain HLC first, then our improvements.

HLC represents the hubs of a forward label $L_f(u)$ as a tree rooted at u. For canonical hierarchical labels, the parent of $w \in L_f(u) \setminus \{u\}$ in the tree is the highest-ranked vertex $v \in L_f(u) \setminus \{w\}$ that hits $[u, w]$ (the tree representing $L_b(u)$ is defined analogously). The key insight is that the same subtree often appears in the labels of several different vertices. HLC represents each unique subtree as a *token* consisting of (1) a *root vertex* r; (2) the number k of *child tokens*; (3) a list of k pairs (i, d_i) indicating that the root of the child token with ID i is within distance d_i from r. A token with no children ($k = 0$) is a *trivial token*, and is represented implicitly. Each nontrivial unique token is stored only once. The data structure also maintains an index mapping each vertex v to its two *anchor tokens*, the roots of the trees representing $L_f(v)$ and $L_b(v)$.

An s–t query works in two phases. The first reconstructs the labels $L_f(s)$ and $L_b(t)$ by traversing the corresponding trees in BFS order and aggregating distances appropriately. The second phase finds the vertex $v \in L_f(s) \cap L_b(t)$ that minimizes $\text{dist}(s, v) + \text{dist}(v, t)$. Since the label entries produced by the first phase are not sorted by hub ID, the second phase uses hashing rather than merging [10].

Although HLC compresses road network labelings by an order of magnitude [10], it is much less effective on small-diameter inputs: high-degree vertices are costlier to represent and there are fewer exact matches between subtrees.

To make HLC effective on a wider range of inputs, we now propose *mask tokens*. A mask token t represents a unique subtree, but not directly: it contains the ID of another token t' (its *reference token*), as well as an incidence vector (bitmask) indicating which children of t' should be taken as children of t. Note that both t and t' must have the same root. This avoids the need to represent the same children multiple times. To exploit this further, we use *supertokens*. A supertoken has the same structure as a standard token (with a root and a list of children), but represents the union of several tokens, defined as the union of their children. For each vertex v, we create a supertoken representing the union of *all* standard tokens rooted at v. Subtrees that actually appear in the labeling can be represented as mask tokens using the supertoken as reference.

Since a mask that refers to a supertoken with k children needs k bits, space usage can be large. But most mask entries are zero (original tokens tend to have few children), motivating the use of *mask compression*. We propose a *two-level approach*. Conceptually, we split a k-bit mask into $b = \lceil k/8 \rceil$ *buckets*, each representing up to 8 consecutive bits. For example, a label with $k = 45$ has six 8-bit buckets: bucket 0 refers to bits 0 to 7, bucket 1 to bits 8 to 15, and so on. Only nonempty buckets are stored explicitly: an *index array* indicates which q buckets (with $1 \leq q \leq b$) are nonempty, and is followed by q 8-bit incidence arrays representing the nonempty buckets. The index takes $\lceil \lceil k/8 \rceil / 8 \rceil$ bytes.

In general, there will be fewer nonempty buckets if the "1" entries in each bit mask are clustered. Since correctness does not depend on the order in which

children appear in a supertoken, we can permute them to make the "1" entries more concentrated. Therefore, for each child x of v, we count the number $c_v(x)$ of standard tokens rooted at v in which x appears, then sort the children of the supertoken rooted at v in decreasing order of $c_v(x)$.

Token-based compression must transform labels into trees, which requires finding parents for all vertices in the label. Delling et al. [10] compute such parents in $O(nM^3)$ time, where M is the maximum label size. We use a much faster (and novel) $O(nM^2)$-time algorithm tailored to hierarchical labels. It augments PL to maintain tentative parent pointers as it goes, using the fact that, by the time a hub is added to a label, its final children are already present.

5 Experiments

We implemented all algorithms in C++ using Visual Studio 2013 with full optimization. All experiments were conducted on a machine with two Intel Xeon E5-2690 CPUs and 384 GiB of DDR3-1066 RAM, running Windows 2008R2 Server. Each CPU has 8 cores (2.90 GHz, 8×64 kiB L1, 8×256 kiB, and 20 MiB L3 cache), but all runs are sequential. We use at most 32 bits for distances.

We test *social networks* (Epinions, Slashdot, Flickr, Hollywood, WikiTalk), *computer networks* (Gnutella, Skitter, MetroSec), *web graphs* (NotreDame, Indo, Indochina, uk2002), *road networks* (ber-t, fla-t, eur-t, eur-d), and *3D triangular meshes* (buddha), available from snap.stanford.edu, webgraph.di.unimi.it, www.dis.uniroma1.it/challenge9, and socialnetworks.mpi-sws.org/datasets.html. We also test unweighted grid graphs with holes from VLSI applications (alue7065; steinlib.zib.de) and grids with obstacles built from *computer games* (FrozenSea, AR0503SR; movingai.com). For the latter we set edge lengths to 408 for axis-aligned moves and 577 for diagonal moves. (Note that $577/408 \approx \sqrt{2}$.) We also test synthetic inputs: square *grids* (grid$i$), *Delaunay triangulations* of random points on the unit square (del$i$), random geometric graphs, often used to model *sensor networks* (rgg$i$) [16], random *preferential attachment* graphs (rba$i$), and random *small-world* networks (rws$i$) [15], with $i = \log n$. Some instances are unweighted, while in others (with suffix -w) edge lengths correspond to Euclidean distances (scaled appropriately and rounded up).

Table 1 summarizes our main results. For each instance, we show its type, average number of vertices (n), average out-degree (m/n), and whether it is directed (d) and weighted (w). We then show the preprocessing time and average number of hubs per label if we run PL with vertices ordered by degree (with ties in the order broken at random) or if we run SamPG, our new ordering algorithm. We then show the space and average time for random queries for the two main label representations we propose: RXL (*Robust eXact Labeling*) uses delta compression and CRXL (*Compressed RXL*) uses two-level mask compression. Both use SamPG. The additional preprocessing time for RXL (over SamPG) is very small (delta compression is fast), but CRXL increases the preprocessing times by 20%–50% (due to parent pointer computation and token generation).

Table 1. Key values for inputs, ordering quality of degree and SamPG, and performance of RXL and CRXL.

instance						degree		SamPG		RXL		CRXL	
type	name	n	m/n	d	w	prep [s]	lab	prep [s]	lab	[MiB]	[µs]	[MiB]	[µs]
sensor	rgg20	1048576	13.1	○	○	2804	1135.7	977	220.0	806.5	2.0	167.3	23.4
	rgg20-w	1048576	13.1	○	●	52962	5502.7	3608	588.8	3154.3	4.9	436.4	76.1
roads	fla-t	1070376	2.5	○	●	1321	791.8	103	41.4	260.9	0.5	55.0	3.4
	eur-t	18010173	2.3	●	●	–	–	8364	82.4	17202.8	0.8	1589.3	13.3
	eur-d	18010173	2.3	●	●	–	–	18664	163.1	33059.5	1.5	2184.2	32.1
grid	alue7065	34046	3.2	○	○	1	98.2	3	55.9	6.1	0.5	2.8	3.5
	grid20	1048576	4.0	○	○	92	144.8	364	126.6	526.5	1.3	127.0	14.8
triang	buddha	543524	6.0	○	○	119	289.5	122	91.5	179.8	0.9	62.6	9.0
	buddha-w	543524	6.0	○	●	1424	1164.7	678	336.0	952.9	2.9	176.6	41.5
	del20	1048576	6.0	○	○	241	286.8	306	117.5	452.1	1.1	134.1	13.2
	del20-w	1048576	6.0	○	●	4606	1598.9	2449	575.3	3077.1	4.8	426.6	115.6
game	FrozenSea	754304	7.6	○	●	160	241.4	214	92.1	429.3	0.9	133.0	10.9
web	NotreDame	325729	4.5	●	○	4	21.1	17	11.3	25.9	0.1	19.5	0.4
	Indo	1382908	12.0	●	○	253	171.7	241	27.4	217.5	0.4	127.9	1.3
	Indochina	7414866	25.8	●	○	12028	539.8	14824	65.5	3916.5	0.7	1322.9	3.2
	uk2002	18520486	15.8	●	○	–	–	43090	278.5	34140.5	1.8	2533.1	25.2
comp	Gnutella	62586	2.4	●	○	37	240.9	60	157.1	39.4	0.9	17.8	7.4
	Skitter	1696415	13.1	○	○	1905	456.5	2813	273.5	1074.6	2.3	316.7	20.6
	MetrocSec	2250498	19.2	○	○	356	132.0	2276	116.5	592.8	0.8	207.7	3.6
social	Epinions	75888	6.7	●	○	12	94.2	50	91.3	29.2	0.6	13.3	3.6
	Slashdot	82168	10.6	●	○	40	188.3	140	190.7	65.3	1.5	31.2	7.4
	rws17	131072	6.0	○	○	5827	4264.4	9224	3597.7	901.2	27.5	1102.9	327.8
	rba20	1048576	12.0	○	○	8006	1485.6	26238	1541.6	4918.0	11.0	2517.6	131.8
	Hollywood	1139905	98.9	○	○	38412	2921.3	61411	2114.3	5934.3	13.9	2050.0	204.0
	Flickr	1861232	12.2	●	○	3353	423.3	10332	322.4	3093.8	2.5	603.8	17.2
	WikiTalk	2394385	2.1	●	○	281	68.0	999	60.2	625.8	0.5	127.3	2.1

We confirm Akiba et al.'s observation that ordering by degree works well on some inputs. SamPG is much more robust, however, often finding much smaller labels (as in Indo, rgg20-w, buddha-w, or fla-t). Because both algorithms have superlinear dependence on label size, SamPG is much faster when it finds better labels. However, since SamPG spends about two-thirds of its time maintaining sampled trees, it is slower when label sizes are similar.

RXL can handle instances with up to tens of millions of arcs and supports queries in microseconds. Compared to RXL, CRXL reduces space usage by up to an order of magnitude (as in eur-t and uk2002). Query times increase mainly due to worse locality, but still take only microseconds. On uk2002, with almost 300 million arcs, it uses only 2.5 GiB and answers queries in 25 µs.

Fig. 1 shows, for the ordering algorithms discussed in Section 3, their average label sizes *relative to SamPG*; shorter bars are better. As expected, degree is the least robust order. Differences between the other approaches are much smaller, but still significant. When ties are numerous, OffLG [3], the label-greedy algorithm that breaks ties in advance (off-line), is much worse than other methods. HybLG, which uses the same order but breaks ties on-line with PL when building the labels, is much better, as is its path-greedy variant (HybPG). Adding sampling to HybPG yields SamPG, with almost no loss in quality. In fact, SamPG can be better (as in the game graph AR0503SR), since tie-breaking is partially on-line, with new trees representing only uncovered pairs. Most importantly, SamPG is asymptotically faster: even on such small instances, the median time (not shown)

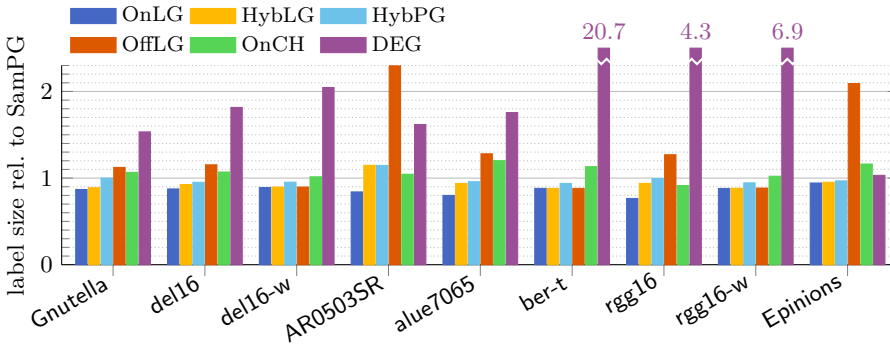

Fig. 1. Label sizes of various orderings relative to SamPG.

is less than half a minute for SamPG, about half an hour for HybPG, HybLG, and OffLG, and days for OnLG. The median time for the CH-based order (OnCH) is only a minute, and it is twice as fast as SamPG on ber-t (Berlin). Although it is not robust, taking hours on Epinions and Gnutella, it finds remarkably small labels, considering that it picks the order based only on local information.

Fig. 2 (left) shows the asymptotic behavior of SamPG on road (square-shaped subgraphs of eur-t) and various synthetic graph classes. Label sizes increase relatively fast for small-world (rws) graphs, and less so for preferential attachment (rba) problems. Higher-diameter inputs have much better behavior. The degree order is asymptotically worse than SamPG for Delaunay triangulations, random geometric graphs, and road networks.

Fig. 2 (right) analyzes the trade-off between space usage and query times for various compression techniques (cf. Section 4). We consider five different representations of the same (SamPG) labels; from left to right, these are *CRXL*, *CRXL*$_1$, *HLC*, *RXL*, and *plain*. The *plain* method represents all hub IDs as 32-bit integers and distances with as few bits as needed (8, 16, or 32) in each case. By incorporating delta compression for hub IDs, RXL uses as little as half as much space as the plain representation, and often has faster queries due to better

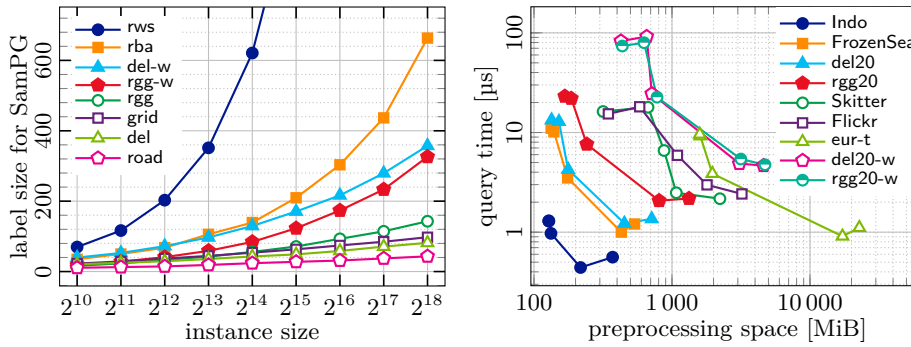

Fig. 2. Left: label sizes for SamPG. Right: space and time tradeoffs; from left to right, the curves are CRXL, CRXL$_1$, HLC, RXL, plain (CRXL and CRXL$_1$ may coincide).

Table 2. Average label size (superhubs for PLL, hubs for RXL), preprocessing time, space, and query times for various methods.

	label size		preprocessing [s]				space [MiB]				query [µs]			
instance	PLL	RXL	PLL	Tree	RXL	CRXL	PLL	Tree	RXL	CRXL	PLL	Tree	RXL	CRXL
Gnutella*	644×16	791	54	209	307	451	209	68	95.7	49.1	5.2	19.0	7.1	45.9
Epinions*	33×16	118	2	128	31	39	32	42	19.1	7.7	0.5	11.0	1.1	4.1
Slashdot*	68×16	219	6	343	85	110	48	83	37.4	17.8	0.8	12.0	1.7	8.0
NotreDame*	34×16	25	5	243	18	22	138	120	22.9	11.9	0.5	39.0	0.2	1.0
WikiTalk*	34×16	113	61	2459	1076	1278	1000	416	560.8	86.5	0.6	1.8	1.0	3.4
Skitter	123×64	273	359	–	2862	3511	2700	–	1074.6	316.7	2.3	–	2.3	20.6
Indo*	133×64	43	173	–	173	201	2300	–	158.6	90.2	1.6	–	0.5	1.8
MetroSec	19×64	116	108	–	2300	2573	2500	–	592.8	207.7	0.7	–	0.8	3.6
Flickr*	247×64	360	866	–	5888	7110	4000	–	1794.6	345.9	2.6	–	2.8	19.9
Hollywood	2098×64	2114	15164	–	61736	75539	12000	–	5934.3	2050.0	15.6	–	13.9	204.0
Indochina*	415×64	91	6068	–	8390	8973	22000	–	1978.8	876.8	4.1	–	0.9	3.9

locality. HLC is Delling et al.'s *hub label compression* [10], but using as few bits as needed (8, 16, or 32) for all distances; it has good compression ratio for road and other high-diameter networks, but is less effective for small-diameter graphs (such as Skitter). CRXL_1 and CRXL use supertokens and bitmasks; while CRXL_1 uses only one level, CRXL may use two. Both are most effective on small-diameter networks. The extra level often helps, but not always (as in Indo). Queries take a few microseconds, fast enough for most applications.

Table 2 compares RXL and CRXL to two state-of-the-art algorithms. PLL is a restricted variant of PL by Akiba et al. [4] tailored to unweighted and undirected networks. This extended PL algorithm joins each new hub v in the order with a small set $S(v)$ of neighboring vertices, then adds all vertices in the "superhub" $\{v\} \cup S(v)$ to all labels that would benefit from at least one vertex in the set. It stores $\text{dist}(u, v)$ explicitly, but for $w \in S(v)$ it stores $\text{dist}(u, w) - \text{dist}(u, v)$, which is in $\{-1, 0, 1\}$ on unweighted, undirected graphs. The resulting labeling is not hierarchical (any two vertices u, w in $S(v)$ will be in each other's labels), but uses less space and has faster preprocessing (all $|\{v\} \cup S(v)|$ searches run simultaneously). The second algorithm, *Tree Decomposition* [5] (Tree), is not label-based. We report preprocessing time (including SamPG for our methods), space, and average query time, as well as the average number of hubs for RXL and superhubs for PLL (×16 and ×64 indicate superhub sizes). Tree and PLL were run (sequentially) on a 2.93 GHz Intel Xeon X5670 [4], a machine similar to ours. For consistency with previous work [4, 5], all inputs in Table 2 are undirected; those obtained from directed ones are marked by asterisks.

Superhubs are quite effective in accelerating PLL preprocessing, which is generally faster than for RXL (notably for MetroSec or WikiTalk). Even so, RXL (which does not use superhubs and is more general) has comparable query times and uses less space, sometimes by a large margin, as in Indo and Indochina. In fact, RXL often has fewer hubs than PLL has superhubs, indicating that SamPG indeed finds good orders. Tree is slower than RXL and sometimes uses much more space. CRXL requires less space than any other method.

We conclude that our approach is quite robust. By combining a new sampling-based order (leveraging both HHL [3] and PL [4]) and a novel label representation,

RXL is competitive with any other technique, each specialized in different graph classes (such as road networks or social graphs).

References

1. Abraham, I., Delling, D., Fiat, A., Goldberg, A.V., Werneck, R.F.: HLDB: Location-based services in databases. In: GIS. pp. 339–348. ACM Press (2012)
2. Abraham, I., Delling, D., Goldberg, A.V., Werneck, R.F.: A hub-based labeling algorithm for shortest paths on road networks. In: SEA, pp. 230–241. (2011)
3. Abraham, I., Delling, D., Goldberg, A.V., Werneck, R.F.: Hierarchical hub labelings for shortest paths. In: ESA. LNCS 7501, pp. 24–35. Springer (2012)
4. Akiba, T., Iwata, Y., Yoshida, Y.: Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In: SIGMOD, pp. 349–360. ACM (2013)
5. Akiba, T., Sommer, C., Kawarabayashi, K.-I.: Shortest-path queries for complex networks: Exploiting low tree-width outside the core. In: EBDT, pp. 144–155 (2012)
6. Boldi, P., Rosa, M., Santini, M., Vigna, S.: Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In: WWW, pp. 587–596 (2011).
7. Brandes, U.: A faster algorithm for betweenness centrality. Journal of Mathematical Sociology 25(2), pp. 163–177 (2001)
8. Delling, D., Goldberg, A.V., Pajor, T., Werneck, R.F.: Customizable route planning. In: SEA. LNCS 6630, pp. 376–387. Springer (2011)
9. Delling, D., Goldberg, A.V., Pajor, T., Werneck, R.F.: Robust Exact Distance Queries on Massive Networks. MSR-TR-2014-12, Microsoft Research (2014)
10. Delling, D., Goldberg, A.V., Werneck, R.F.: Hub label compression. In: SEA. LNCS 7933, pp. 18–29. Springer (2013)
11. Gavoille, C., Peleg, D., Pérennes, S., Raz, R.: Distance Labeling in Graphs. Journal of Algorithms 53, pp. 85–112 (2004)
12. Geisberger, R., Sanders, P., Schultes, D.: Better approximation of betweenness centrality. In: ALENEX. pp. 90–100 (2008)
13. Geisberger, R., Sanders, P., Schultes, D., Vetter, C.: Exact Routing in Large Road Networks Using Contraction Hierarchies. Trans. Science 46(3), pp. 388–404 (2012)
14. Goldberg, A.V., Razenshteyn, I., Savchenko, R.: Separating hierarchical and general hub labelings. In: MFCS, LNCS 8087, pp. 469–479. Springer (2013)
15. Hagberg, A.A., Schult, D.A., Swart, P.J.: Exploring network structure, dynamics, and function using NetworkX. In: SciPy, pp. 11–15 (2008)
16. Holtgrewe, M., Sanders, P., Schulz, C.: Engineering a scalable high quality graph partitioner. In: IPDPS, pp. 1–12, IEEE (2010)
17. Jiang, M., Fu, A.W.C., Wong, R.C.W., Cheng, J., Xu, Y.: Hop doubling label indexing for point-to-point distance querying on scale-free networks, coRR (2014)
18. Jin, R., Ruan, N., Xiang, Y., Lee, V.: A highway-centric labeling approach for answering distance queries on large sparse graphs. In: SIGMOD, pp. 445–456 (2012)
19. Peleg, D.: Proximity-preserving labeling schemes. Journal of Graph Theory 33(3), 167–176 (2000)
20. Schenkel, R., Theobald, A., Weikum, G.: HOPI: An efficient connection index for complex XML document collections. In: EDBT, pp. 237–255. Springer (2004)
21. Sommer, C.: Shortest-path queries in static networks. ACM Computing Surveys 46, 547–560 (2014)
22. Tarjan, R.: Data Structures and Network Algorithms. SIAM (1983)
23. Wei, F.: TEDI: Efficient shortest path query answering on graphs. In: SIGMOD, pp. 99–110. ACM (2010)