Robust Exact Distance Queries on Massive Networks

DANIEL DELLING
Microsoft Research
dadellin@microsoft.com

ANDREW V. GOLDBERG
Microsoft Research
goldberg@microsoft.com

THOMAS PAJOR
Microsoft Research
tpajor@microsoft.com

RENATO F. WERNECK
Microsoft Research
renatow@microsoft.com

July 2014
Technical Report
MSR-TR-2014-12

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.

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
http://www.research.microsoft.com

1. Introduction

Answering point-to-point distance queries in graphs is a fundamental building block for many applications [37] in social networks, search, computational biology, computer networks, and road networks. Dijkstra's algorithm [39] 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 [2,12,18] and some social and communication graphs [3–5,22,23,41], 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 [16]. 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 [1,22,36]). Labels can be optionally compressed with no loss in correctness. See Figure 1 for an illustration.

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 [3,18] or on unweighted, undirected small-diameter networks [4] are not robust on other inputs. Compression strategies are similarly specialized to particular networks [4,14]. 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. This is the full version of an extended abstract [13] presented at the 22nd European Symposium on Algorithms (ESA 2014).

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* [33] 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 [16] is a special case with a two-part label $L(v)$ for every vertex v: a *forward label* $L_f(v)$

Graph
Ordering
Labeling
Compressed Labeling

Figure 1: Overview of the hierarchical labeling algorithm framework. An ordering is computed for the input graph and used to compute a labeling for fast exact distance queries. An optional compression step reduces the space consumption of this labeling.

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 $(w, \mathrm{dist}(v, w))$, with $w \in V$; similarly, $L_b(v)$ has pairs $(u, \mathrm{dist}(u, v))$. 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 $(v, \mathrm{dist}(u, v))$. 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 $\mathrm{dist}(u, v) + \mathrm{dist}(v, w) = \mathrm{dist}(u, w)$, i.e., if at least one shortest u–w path contains v.

To find $\mathrm{dist}(s, t)$, an HL query finds the hub $v \in L_f(s) \cap L_b(t)$ that minimizes $\mathrm{dist}(s, v) + \mathrm{dist}(v, t)$. 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, 18, 22].

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 [19]. 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 [39] 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 $(v, \mathrm{dist}(v, w))$ to $L_b(w)$ and scans w as usual. The second Dijkstra computation uses the reverse graph and is pruned similarly; it adds $(v, \mathrm{dist}(w, v))$ 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). We show that it is also hierarchical and thus canonical.

Lemma 2.1. PL *finds a hierarchical labeling.*

Proof. We show that if $rank(v) > rank(u)$, then u is not a backward hub of v (forward hubs can be analyzed in a similar way). The algorithm can add u to $L_b(v)$ only if v is scanned during the forward pruned Dijkstra computation from u. Consider the maximum ranked vertex x that hits $[u, v]$. We process x before u and v; since x is the maximum vertex to hit $[u, x]$, when processing x we add it to $L_f(u)$ and to $L_b(v)$. Therefore, during the forward Dijkstra computation from u, the u–v query using the current labels returns $\text{dist}(u, v)$. This implies that we will not scan v. $\square$

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, 22]; 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.

3.1. 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].

3.2. 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. (See Appendix A.1.) We call these variants OnPG and OnLG, respectively.

3.3. 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 [11]. (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 (see Appendix A.2 for details).

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 [17]. 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 (see Appendix B.3). 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.

4.1. Basic Compression

Recall that a label $L_f(u)$ can be seen as an array of pairs $(v, \mathrm{dist}(u, v))$ 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 $h_1 < h_2 < h_3 < \dots$ 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 $\tilde{s}(v) \leftarrow s(v)$. See Appendix A.3 for details.

4.2. Token-Based Compression

We now present a novel scheme to achieve even higher compression. It extends *hub label compression* (HLC) [14], 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 [14].

Although HLC compresses road network labelings by an order of magnitude [14], 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. [14] 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. See Appendix A.4.

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). The data is available from `snap.stanford.edu`,

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

`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$i$) [21], random *preferential attachment* graphs (rba$i$), and random *small-world* networks (rws$i$) [20], 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). See Appendix B.1 for details.

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*)

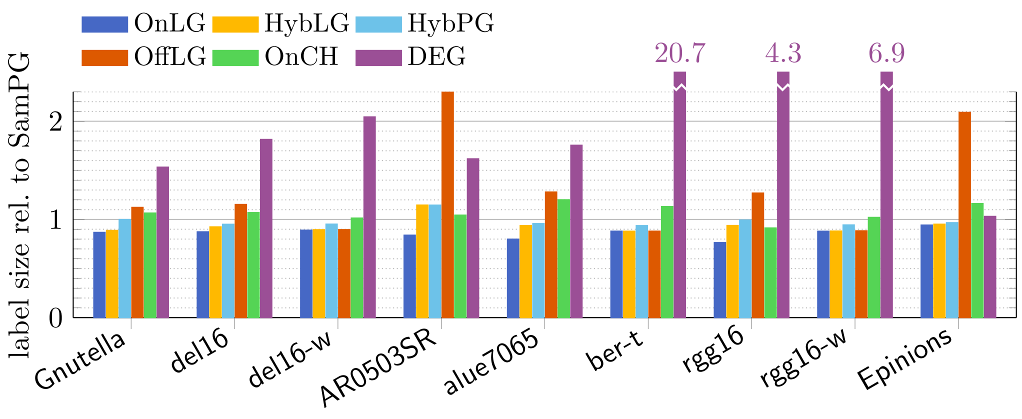


Figure 2: Label sizes of various orderings relative to SamPG.

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).

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.

Figure 2 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) 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.

Figure 3 (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,

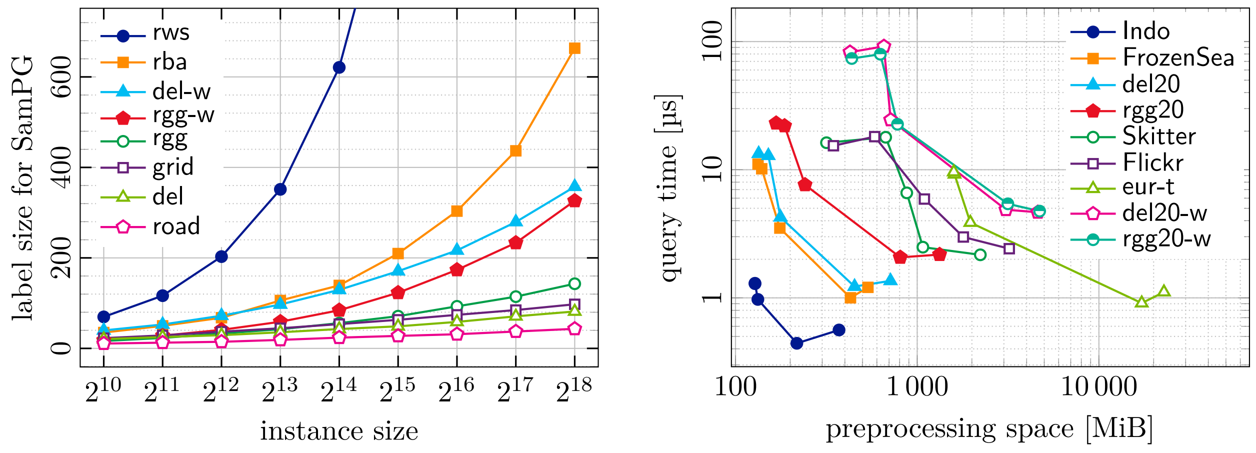


Figure 3: Left: label sizes for SamPG. Right: space and time tradeoffs; from left to right, the curves are CRXL, CRXL$_1$, HLC, RXL, plain (CRXL and CRXL$_1$ may coincide).

random geometric graphs, and road networks (see Appendix B.4).

Figure 3 (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 locality. HLC is Delling et al.'s *hub label compression* [14], 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

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

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).

Acknowledgments. We thank Takuya Akiba, Alan Mislove, and Christian Schulz for making benchmark instances available.

References

[1] Ittai Abraham, Daniel Delling, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck. HLDB: Location-based services in databases. In *Proceedings of the 20th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems (GIS'12)*, pages 339–348. ACM Press, 2012.

[2] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. A hub-based labeling algorithm for shortest paths on road networks. In *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*, volume 6630 of *Lecture Notes in Computer Science*, pages 230–241. Springer, 2011.

[3] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Hierarchical hub labelings for shortest paths. In *Proceedings of the 20th Annual European Symposium on Algorithms (ESA'12)*, volume 7501 of *Lecture Notes in Computer Science*, pages 24–35. Springer, 2012.

[4] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD'13, pages 349–360. ACM, 2013.

[5] Takuya Akiba, Christian Sommer, and Ken-ichi Kawarabayashi. Shortest-path queries for complex networks: Exploiting low tree-width outside the core. In *Proceedings of the 15th International Conference on Extending Database Technology*, EDBT'12, pages 144–155. ACM, 2012.

[6] Réka Albert, Hawoong Jeong, and Albert-László Barabási. Diameter of the world wide web. *Nature*, 401:130–131, September 1999.

[7] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[8] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice & Experience*, 34(8):711–726, 2004.

[9] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World Wide Web*, pages 587–596. 2011.

[10] Paolo Boldi and Sebastiano Vigna. The WebgGaph framework I: compression techniques. In *Proceedings of the 13th international conference on World Wide Web*, pages 595–602. 2004.

[11] Ulrik Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25(2):163–177, 2001.

[12] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable route planning. In *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*, volume 6630 of *Lecture Notes in Computer Science*, pages 376–387. Springer, 2011.

[13] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Robust Distance Queries on Massive Networks. In *Proceedings of the 22nd Annual European Symposium on Algorithms (ESA'14)*, volume 8737 of *Lecture Notes in Computer Science*, pages 321–333. Springer, September 2014.

[14] Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Hub label compression. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *Lecture Notes in Computer Science*, pages 18–29. Springer, 2013.

[15] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors. *9th DIMACS Implementation Challenge - Shortest Paths*, 2006.

[16] Cyril Gavoille, David Peleg, Stéphane Pérennes, and Ran Raz. Distance Labeling in Graphs. *Journal of Algorithms*, 53:85–112, 2004.

[17] Robert Geisberger, Peter Sanders, and Dominik Schultes. Better approximation of betweenness centrality. In Ian Munro and Dorothea Wagner, editors, *Proceedings of the 10th Workshop on Algorithm Engineering and Experiments (ALENEX'08)*, pages 90–100. SIAM, April 2008.

[18] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact Routing in Large Road Networks Using Contraction Hierarchies. *Transportation Science*, 46(3):388–404, August 2012.

[19] Andrew V. Goldberg, Ilya Razenshteyn, and Ruslan Savchenko. Separating hierarchical and general hub labelings. In *Mathematical Foundations of Computer Science 2013*, volume 8087 of *Lecture Notes in Computer Science*, pages 469–479. Springer, 2013.

[20] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy2008)*, pages 11–15, Pasadena, CA USA, August 2008.

[21] Manuel Holtgrewe, Peter Sanders, and Christian Schulz. Engineering a scalable high quality graph partitioner. In *24th International Parallel and Distributed Processing Symposium (IPDPS'10)*, pages 1–12. IEEE Computer Society, 2010.

[22] Minhao Jiang, Ada Wai-Chee Fu, Raymond Chi-Wing Wong, James Cheng, and Yanyan Xu. Hop doubling label indexing for point-to-point distance querying on scale-free networks, 2014. CoRR.

[23] Ruoming Jin, Ning Ruan, Yang Xiang, and Victor Lee. A highway-centric labeling approach for answering distance queries on large sparse graphs. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 445–456. ACM, 2012.

[24] Thorsten Koch, Alexander Martin, and Stefan Voß. SteinLib: An updated library on Steiner tree problems in graphs. Technical Report ZIB-Report 00-37, Konrad-Zuse-Zentrum für Informationstechnik Berlin, 2000.

[25] Jure Leskovec. Stanford large network dataset collection. `http://snap.stanford.edu/`, 2014.

[26] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. Predicting positive and negative links in online social networks. In *Proceedings of the 19th international conference on World wide web*, pages 641–650. ACM, 2010.

[27] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. Signed networks in social media. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1361–1370. ACM, 2010.

[28] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 177–187. ACM, 2005.

[29] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[30] Clémence Magnien, Matthieu Latapy, and Michel Habib. Fast computation of empirically tight bounds for the diameter of massive graphs. *Journal of Experimental Algorithmics (JEA)*, 13:10:1–10:9, 2009.

[31] Ripeanu Matei, Adriana Iamnitchi, and Ian Foster. Mapping the Gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. *IEEE Internet Computing Journal*, 2002.

[32] Alan Mislove, Massimiliano Marcon, Krishna P Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and analysis of online social networks. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 29–42. 2007.

[33] David Peleg. Proximity-preserving labeling schemes. *Journal of Graph Theory*, 33(3):167–176, 2000.

[34] Matthew Richardson, Rakesh Agrawal, and Pedro Domingos. Trust management for the semantic web. In *The Semantic Web – ISWC 2003*, pages 351–368. Springer, 2003.

[35] Pedro V. Sander, Diego Nehab, Eden Chlamtac, and Hugues Hoppe. Efficient traversal of mesh edges using adjacency primitives. *ACM Transactions on Graphics (TOG)*, 27(5):144, 2008.

[36] Ralf Schenkel, Anja Theobald, and Gerhard Weikum. HOPI: An efficient connection index for complex XML document collections. In *Advances in Database Technology – EDBT 2004*, pages 237–255. Springer, 2004.

[37] Christian Sommer. Shortest-path queries in static networks. *ACM Computing Surveys*, 46:547–560, 2014.

[38] Nathan R. Sturtevant. Benchmarks for grid-based pathfinding. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2):144–148, 2012.

[39] Robert Tarjan. *Data Structures and Network Algorithms.* SIAM, 1983.

[40] Duncan J Watts and Steven H Strogatz. Collective dynamics of ‘small-world’ networks. *Nature*, 393(6684):440–442, 1998.

[41] Fang Wei. TEDI: Efficient shortest path query answering on graphs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD’10, pages 99–110. ACM, 2010.

APPENDIX

This appendix contains information omitted from the main text. It is structured as follows. Appendix A contains algorithmic details: the label-greedy variant of the on-line algorithm, implementation details for SamPG, how to implement our advanced reordering routine efficiently for RXL, and how to build trees from the labels. Appendix B explains our instances in more detail and presents further experiments.

A. Detailed Algorithms

A.1. Label-Greedy Algorithm

This section presents details for OnLG, the label-greedy variant of the on-line algorithm for finding orders introduced in Section 3. In each iteration, it picks the vertex v for which the ratio $\sigma_v/(f_v + b_v)$ is maximized. Here σ_v is defined as before (number of uncovered pairs hit by v), f_v is the number of forward labels v would be added to (if picked next), and b_v is the number of backward labels v would be added to.

Our label-greedy algorithm uses all data structures from the path-greedy variant (OnPG) described in Section 3, as well as new ones to keep track of label counts. We maintain n-sized arrays representing f_v and b_v explicitly, as well as two $n \times n$ matrices, F and B. Entry $F[v, u]$ indicates how many uncovered pairs $[u, \cdot]$ (i.e., starting at u) are hit by v, and $B[v, w]$ represents how many uncovered pairs $[\cdot, w]$ are hit by v. These matrices can be initialized in $O(n^3)$ total time. We start with zero matrices and, for each pair $[u, w]$, check every vertex v; if $\text{dist}(u, v) + \text{dist}(v, w) = \text{dist}(u, w)$, we increment both $F[v, u]$ and $B[v, w]$. We can then initialize f_v as the number of nonzero entries matching $F[v, \cdot]$; similarly, b_v is the number of nonzero entries matching $B[v, \cdot]$.

The algorithm then proceeds as in the path-greedy version, but picking the vertex v that maximizes $\sigma_v/(f_v + b_v)$. Moreover, the third step of the main loop must update not only the σ values, but also F, B, f, and b. If a vertex $x \in V$ hits a pair $[u, w] \in Q$, we decrement both $F[x, u]$ and $B[x, w]$. If $F[u, x]$ becomes zero, we decrement f_x; if $B[x, w]$ becomes zero, we decrement b_x. The total running time is still $O(n^3)$.

As described, both variants (path- and label-greedy) run in $\Theta(n^3)$ time regardless of the graph topology. Our actual implementation has several optimizations that make it faster on sparse graphs (for example, the distance table can be built with n calls to Dijkstra's algorithm rather than Floyd-Warshall). Our implementation still runs in $\omega(n^2)$ time in practice, however.

A.2. Implementation Details for SamPG

This section explains implementation details for SamPG that are crucial to achieve sub-quadratic space and time.

Recall from Section 3 that we sample new trees as the algorithm progresses, and that trees created later in the algorithm are born smaller, as we use PL-based pruning to build them. Thus, we use different representations for large and small trees. A large tree (with at least $n/8$ vertices) is represented as an n-sized array; the i-th position contains the parent of vertex i, or `null` if the vertex is not in the tree. Each smaller tree is represented as a hash table that associates each vertex with its parent; vertices not in the tree do not appear in the table. Note that the same tree may switch representations as it shrinks during the algorithm.

Also, recall that each iteration of our algorithm involves the following two steps. First, determine the vertex v^* to be selected as hub; then remove its descendants from all active trees. To support these operations efficiently, we use the following hybrid approach. In the beginning, we linearly scan over all vertices to determine v^* and then iterate over all active trees T_u, removing v^*'s descendants from T_u as we go. As soon as we reach an interation that picks a hub v^* with fewer than $n/8$ descendants in total (over all sampled trees), we start maintaining a *reverse index* that stores with each vertex v pointers to the trees that contain it. From this point on, we use this reverse index to access the relevant trees directly. At the same time, we build a *max-heap* of vertices (with priorities as keys), which allows us to efficiently determine the next hub v^* without looking at all n vertices in subsequent iterations.

A.3. Implementation Details for Advanced Reordering

This section shows how the advanced reordering routine can be implemented efficiently. To do so, we need some definitions. For every label L (forward or backward), let its *horizon* $h(L)$ be the maximum ID that a hub in L could be assigned and still be represented in 8 bits (this is the ID of the last 8-bit entry already in L plus 256). We say that L is *open* if there exists at least one free ID i such that $i \leq h(L)$, and *closed* otherwise. Note that $s(v)$ is the number of open labels that contain v. Let $\mu(v)$ be the minimum $h(L)$ over all open labels that contain v. If v^* has the highest $s(v^*)$ value, we assign to v^* the maximum free ID $i \leq \mu(v^*)$.

Assigning ID i to v^* may increase $h(L)$ for some of the labels that contain v^*. Moreover, it may cause some open labels to become closed (even if they do not contain v^*). Although the first issue is easy to handle (the $h(L)$ values can be updated efficiently), the second is more challenging, since it may affect the $s(\cdot)$ values of many other vertices.

To avoid recomputing $s(\cdot)$ values too often, we use a lazy version of this algorithm. Instead of maintaining accurate $s(v)$ values for all vertices, we keep upper bounds $\tilde{s}(v)$ on their value in a priority queue (max-heap). Each step of the algorithm removes the top element v from the heap (with value $\tilde{s}(v)$), then explicitly computes $s(v)$ by looking at the $h(\cdot)$ values of all labels containing v. If $\tilde{s}(v)$ is accurate enough, we assign v an ID. Otherwise, we set $\tilde{s}(v) \leftarrow s(v)$ and insert v back into the heap. One could guarantee that a vertex v is removed from the heap no more than $O(\log n)$ times by “accepting” a vertex as long as $s(v) \geq (1-\epsilon)\tilde{s}(v)$, for some constant $\epsilon > 0$, but we found that setting $\epsilon = 0$ works well enough in practice.

We keep an n-sized boolean array that indicates if each ID (from 0 to $n-1$) is still available. To check if a label L is still open (while computing $s(v)$), we traverse this array backwards starting at position $h(L)$ until we find an available ID (or hit the beginning). To speed this process up, each used ID i keeps a *predecessor* ID $\mathit{pred}[i] < i$ with the following property: all IDs j such that $\mathit{pred}[i] < j < i$ are used. Note that $\mathit{pred}[i]$ itself may or may not be used. When looking for an empty position in the array, we can use the *pred* pointers to skip over blocks of used IDs. When we finally reach an unused ID j, we use path compression [39] to accelerate future searches: we traverse the same path again and set to j the *pred* pointers of all entries visited.

A.4. Parents

An important step of token-based compression (with or without masks) is transforming labels into trees, which in turn requires finding the parents of all vertices in the label. Consider the forward label $L_f(u)$ (backward labels can be processed similarly). Recall that the parent $p(w)$ of $w \in L_f(u) \setminus \{u\}$ is the highest-ranked vertex $v \in L_f(u) \setminus \{w\}$ that covers $[u, w]$. Delling et al. [14] compute such parents in $O(nM^3)$ time, where M is the maximum label size. While this is acceptable when labels are small (as in road networks), it can be prohibitive when labels are bigger, as in some social and communication networks.

We propose a more sophisticated approach that is faster on hierarchical labelings. It works by augmenting PL to maintain tentative parent pointers. Recall that PL adds hubs to labels in decreasing order of rank (importance). As the algorithm progresses, we ensure that all vertices w in a (partial) label $L_f(u)$ obey the following *parent condition*: $p(w)$ is the highest-ranked vertex $v \in L_f(u) \setminus \{w\}$ that hits $[u, w]$; if no such vertex v exists, $p(w)$ is *null* (undefined). (For simplicity, we focus on forward labels, but backward labels can be dealt with similarly.) Note that some vertices may not have parents, since vertex u itself may not be in $L_f(u)$ yet. The partial label can thus be seen as a forest of rooted trees. As more vertices are added to the label, this forest may change, with new vertices being added and existing vertices acquiring parents. Once a vertex has a parent, however, it is final.

Lemma A.1. *Consider any vertex w in a partial label $L_f(u)$. If $p(w)$ is defined, then it will remain the same even after other hubs are added to the label.*

Proof. Let v be the original $p(w)$; it hits $[u, w]$. Although a new vertex v' may also hit $[u, w]$, it will have $\mathit{rank}(v') < \mathit{rank}(v)$, since vertices are added to labels in decreasing order of rank. So v will remain the parent. □

Accordingly, a newly-added hub v may become the parent only of the current roots in the forest representing $L_f(u)$. For each root $w \in L_f(u)$, we must test if $\mathrm{dist}(u, v) + \mathrm{dist}(v, w) = \mathrm{dist}(u, w)$. We know $\mathrm{dist}(u, w)$ (from $L_f(u)$) and $\mathrm{dist}(u, v)$ (the test happens during a backward Dijkstra search from v). To get $\mathrm{dist}(v, w)$, we need the following lemma.

Lemma A.2. *If v becomes the new parent of a vertex w in $L_f(u)$, then $w \in L_f(v)$.*

Proof. Assume by contradiction that $w \notin L_f(v)$. Note that $\mathit{rank}(w) > \mathit{rank}(v)$, since PL processes vertices in decreasing order of rank. In particular, this means that w has been considered for insertion into $L_f(v)$. Since it was not inserted (by assumption), there must exist a vertex x with $\mathit{rank}(x) > \mathit{rank}(w)$ that hits $[v, w]$. Vertex x has been considered for insertion into $L_f(u)$ before w was. Regardless of whether x was actually inserted into $L_f(u)$ or not, the pair $[u, w]$ must already be covered (either by x itself or by the vertex that covered $[u, x]$ and prevented the insertion of x). In either case, w should not belong to $L_f(u)$, a contradiction. □

The algorithm to find the parents is as follows. We add hubs to labels in decreasing order of rank. Consider the PL iteration that processes hub v. We first add v itself to $L_f(v)$. We then fill an $O(n)$-sized array *vdist*, with $\mathit{vdist}[x] = \mathrm{dist}(v, x)$ if $x \in L_f(v)$ and ∞ otherwise. This array can be initialized once with ∞ and updated in $O(|L_f(v)|)$ time for each new hub v. Then we run a pruned (reverse) Dijkstra search from v. Before scanning a vertex u, we traverse $L_f(u)$. For every hub $w \in L_f(u)$

whose parent is undefined, we set $p(w) \leftarrow v$ if and only if $\text{dist}(u,v) + \text{dist}(v,w) = \text{dist}(u,w)$. The first term comes from Dijkstra's algorithm, the second is $vdist[w]$, and the third comes from $L_f(u)$.

The algorithm takes $O(nM^2)$ time in total, which is a factor of $O(M)$ faster than the original approach of Delling et al. [14]. Moreover, it only adds a small constant overhead to the original PL algorithm, since every label it traverses is first traversed by PL. Since finding parent pointers is only necessary if we do advanced token-based compression, in our experiments we actually compute parents separately, after the labels are computed in full (using a similar procedure).

B. Additional Experiments

In this section we describe our instances in more detail and present further experiments that were omitted from the main part of the paper.

B.1. Instances

In the following we give more detail on the inputs we use in our experiments. Figures for the number of vertices and arcs of each instances are given in Table 1 (Section 5), or are mentioned here explicitly, otherwise.

Social Networks

- Epinions is a who-trusts-whom social network from the general consumer review website `epinions.com`. Vertices represent users and (directed) arcs represent trust relationships [25,34].
- Slashdot is a social network of the user community from the technology-related news website Slashdot. Vertices represent Slashdot users and a directed arc from u to v indicates that user u tagged user v as friend or foe [25,29].
- Flickr is a social network based on a photo sharing website. Vertices represent users and a directed arc from u to v indicates that user u is linked to (follows) user v [32].
- Hollywood is an undirected social graph of movie actors. Vertices represent actors, and two actors are joined by an edge whenever they appear in the same movie [9,10].
- WikiTalk is a discussion network obtained from Wikipedia's talk pages from its inception until 2008. Vertices represent Wikipedia users, and a directed arc from u to v indicates that user u posted on the talk page of user v [25–27].

Computer Networks

- Gnutella is a snapshot of a peer-to-peer file sharing network. Vertices represent hosts in the Gnutella network topology, and directed arcs represent connections between hosts [25,31].
- Skitter is an undirected Internet topology network. Vertices represent entities in the Internet and edges communication between them [25,28].

- MetroSec is an undirected Internet IP traffic graph. Vertices represent hosts and an edge between u and v indicates that the hosts u and v appeared as sender/receiver in an IP packet [30].

Web Graphs

- NotreDame is a web graph of the University of Notre Dame (domain `nd.edu`) from 1999. Vertices represent web pages and (directed) arcs hyperlinks between them [6, 25].
- Indo is a small crawl of the `.in` domain. Vertices represent web pages and (directed) arcs hyperlinks between them [9, 10].
- Indochina is a fairly large crawl of the country domains of Indochina. Vertices represent web pages and (directed) arcs hyperlinks between them [9, 10].
- uk2002 is a web graph obtained from a crawl of the `.uk` domain in 2002 performed by UbiCrawler [8]. Again, vertices represent web pages and (directed) arcs hyperlinks between them [9, 10].

Game Maps. These undirected graphs represent computer game maps where agents can move horizontally, vertically, and diagonally in a grid-like game world with obstacles. The graphs are available from `movingai.com`, and we weight the edges by 408 for axis-aligned moves and by 577 for diagonal moves.

- AR0503SR is a smaller map from the game Baldur's Gate II [38]. The graph is undirected, and it has 38 215 vertices and 296 348 arcs.
- FrozenSea, a map from the game Starcraft [38], is the largest available instance.

VLSI Instances and 3D Meshes

- alue7065 represents a circuit from VLSI applications [24].
- buddha is a fairly detailed 3D computer graphics surface mesh obtained from a scan of a small Buddha statue [35].

Road Networks. These graphs represent simple road networks. Vertices depict intersections and arcs represent street segments in direction of traffic flow. Arcs are weighted either according to travel time (-t) or geographic distance (-d).

- bay-t is a medium-sized road network of the San Francisco Bay Area in California, USA [15]. The graph is undirected, and it has 321 270 vertices and 794 830 arcs.
- fla-t is a larger road network of the state of Florida, USA [15].
- eur-t and eur-d represent the continental road network of Western Europe [15].
- ber-t is a small road network of Berlin (Germany) extracted from eur-t. The graph has 32 413 vertices and 80 845 directed arcs.

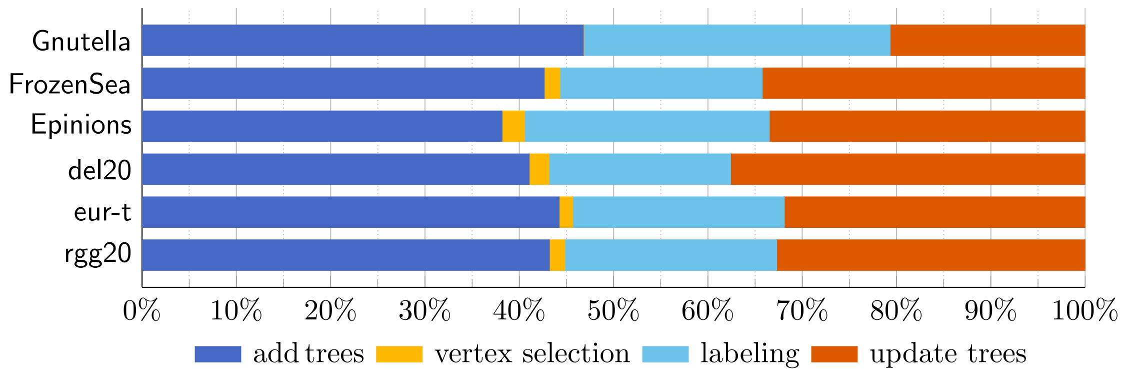


Figure 4: Stacked running times for various instances. We show the running time for adding trees, vertex selection, pruned labeling, and updating trees.

Generated Networks. For the following graph classes (available upon request) we vary the number n of vertices by setting $n = 2^i$.

- road$i$ are square-shaped subgraphs of the Western European road network (eur-t), centered at a random (but fixed for varying i) location.
- del$i$ are Delaunay triangulations over random points in the unit square [21], with unit edge lengths. The del$i$-w instances have the same topology, but lengths represent Euclidean distances (scaled so that the average length is 1000, then rounded up).
- rgg$i$ are geometric graphs modeling sensor networks: vertices represent random points in the unit square, with two vertices connected by an (unweighted) edge if the corresponding Euclidean distance is at most $0.55\sqrt{(\ln n)/n}$; these graphs are almost connected with high probability [21]. We use our own generator for these instances. The rgg$i$-w graphs have the same topology, but edge lengths represent Euclidean distances (scaled so that the maximum length is 1000, then rounded up).
- rba$i$ are random preferential attachment graphs based on the Barabási-Albert model [7, 20]: Vertices are added one at a time, thereby connecting each newly-added vertex to k existing vertices with probability proportional to their (current) degrees. We set $k = 6$ in our experiments.
- rws$i$ are random small-world graphs based on the Watts-Strogatz model [20, 40]. It first creates a ring of n vertices and connects each vertex with its k closest neighbors of the ring. With probability p, each edge $\{u, v\}$ is then rewired to $\{u, w\}$, with w chosen uniformly at random. We set $k = 6$ and $p = 0.2$ in our experiments.

B.2. Preprocessing Time

Figure 4 presents a detailed analysis of the running time of SamPG for some instances. It shows how the total running time is split among the main subroutines: generating new sample trees (add trees), choosing the best vertex (vertex selection), adding hubs to the labels with a PL step (labeling), and

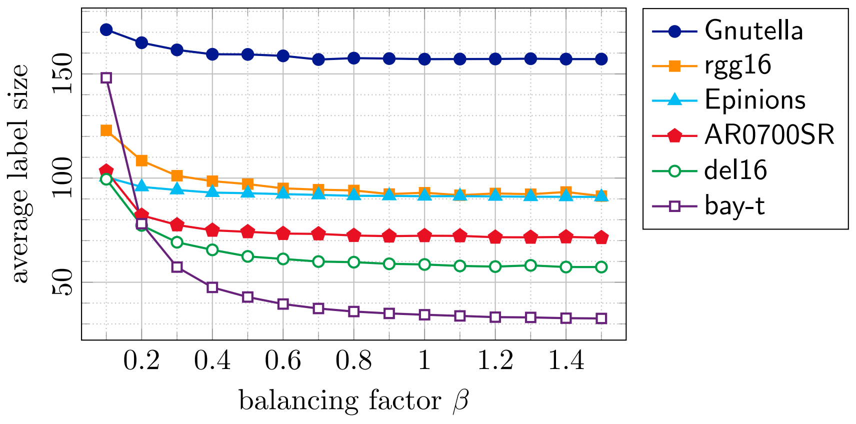


Figure 5: Effect of balancing on label quality.

updating sample trees (update trees). The plots show that picking the best vertex takes negligible time. The time to generate trees is not much greater than the time to update trees. The time spent adding hubs to labels varies from 1/5 (for del20) to 1/3 of the total time (for Gnutella).

B.3. Parameters

We now evaluate our choice of parameters for SamPG, starting with balancing the work of maintaining trees and adding new hubs to labels. Recall from Section 3 that we keep two counters (c_t and c_l) that account for the number of operations in these respective parts of the algorithm. To control balance, we introduce a parameter β: new trees are generated until c_t exceeds βc_l. (We still stop growing trees if the total number of vertices in existing trees exceeds $10kn$; cf. Section 3.) Figure 5 shows that RXL behaves poorly when β is very small, and improves as β increases. On all instances tested, most gains are obtained before β reaches 1, justifying our choice of 1 as default balancing parameter.

Figure 6 analyzes the choice of priority function we use to select the best vertex in each iteration of SamPG. For each instance and each function, it shows how the average label size evolves during the execution of the algorithm. Let $\lambda(F, i)$ be the average label size after the i-th iteration of the algorithm (i.e., after i hubs are selected) with priority function F. For clarity, we do not plot this value directly; instead, for each method P we plot $\lambda(F, i) - \lambda(\mathsf{deg}, i)$, i.e., the difference to the baseline degree-based method. Negative values favor F over deg. Besides deg, we consider three functions F. Function avg simply picks the vertex with the most descendants in the sampled trees. The other two functions use 16 counters (as described in Section 3): cnt16-med picks their median value, while cnt16-2max discards the two highest counters and takes the average of the remaining ones.

As expected, deg works well on some small-diameter networks (such as Epinions), but is worse than counter-based methods for high-diameter inputs. Moreover, avg is much less robust than the counter-based approaches, since it is heavily biased towards the roots of the trees in the sample. Picking the median (as in cnt16-med) helps eliminate such outliers, generally leading to smaller labels. In some cases, however, completely discarding non-median buckets may be too extreme. The

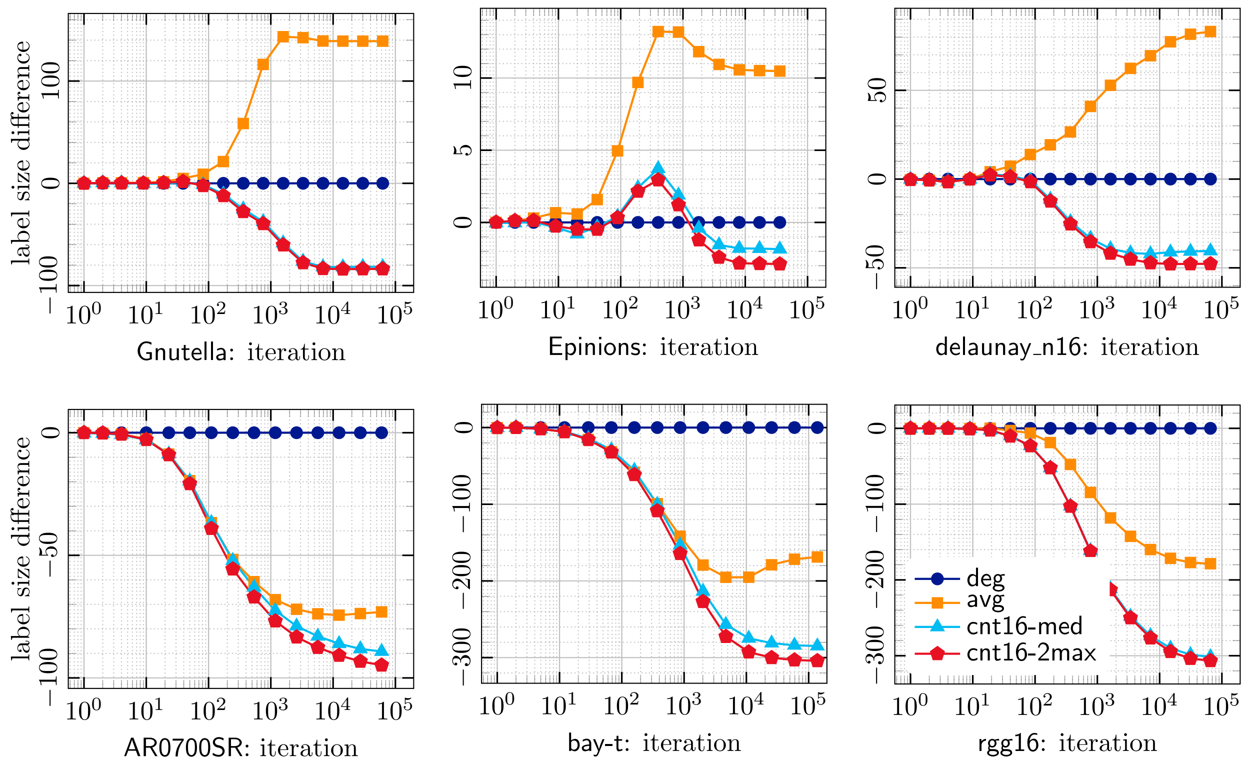
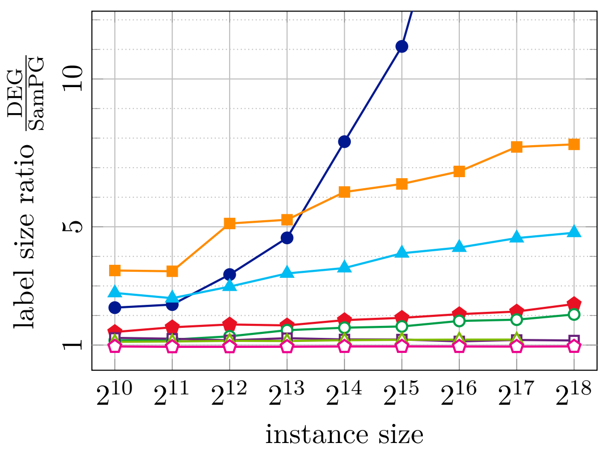


Figure 6: Average label size difference from degree (deg) for different priority functions.

most robust method is cnt16-2max, which discards only 1/8 of the buckets and makes effective use of the remaining ones.

B.4. More Asymptotics

Figure 7 shows further details regarding the asymptotic behavior of our algorithm on several graph classes. We evaluate performance by plotting the ratio of the degree-based method over SamPG for label size (left) and preprocessing time (right). We observe that on road networks (road), geometric graphs (rgg), and (somewhat less obviously) on triangulations (del), SamPG performs asymptotically better than degree, producing smaller labels in less preprocessing time. For the remaining instances degree and SamPG exhibit the same asymptotic behavior. This is even the case for regular grids (grid), where degree essentially corresponds to a random ordering (most vertices have degree 4). Here the algorithm benefits from on-line tie-breaking; ties are numerous, with most paths hit by a large number of vertices. Note that while the preprocessing for SamPG can be slower than degree for some graph classes, it is never off by more than a constant factor, indicating that SamPG is indeed robust to the input.



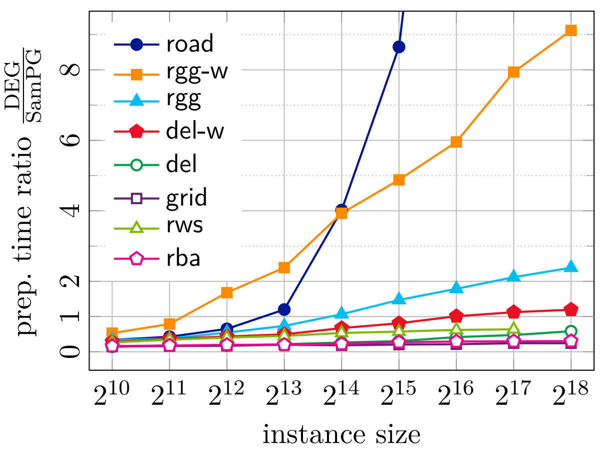


Figure 7: Asymptotic behavior of SamPG compared to degree (DEG) on several graph classes. The left plot shows the label size ratio of DEG over SamPG, while the right plot shows the same ratio for preprocessing time. The legend of the right plot also applies to the left one.