then splits the range (l, r) in two parts: $(l, m-1)$ and $(m+1, r)$ and, recursively, computes the maximum scoring node into each of them: $m' = \mathsf{RMQ}(l, m-1)$ and $m'' = \mathsf{RMQ}(m+1, r)$; (iii) finally, it inserts the two triples $\langle l, m', m-1\rangle$ and $\langle m+1, m'', r\rangle$ in the heap with priorities given by the scores of nodes $N_1(u)[m']$ and $N_1(u)[m'']$, respectively. Hence, each step extracts one triple and inserts two triples in the heap, overall taking $O(k \log k)$ time.

It is clear that the latter method is better than the former one whenever the range becomes slightly larger than k.

A top-k prefix-search query over the FoF of a user u can be obviously computed by running the above algorithm over all the $d(u)$ friends of u and inserting the (at most) top-k results from each friend of u in the max-heap mentioned above. This algorithm has to manage at most $k \times d(u)$ candidates and, thus, it requires $O(k \times d(u) \log(k \times d(u)))$ time in the worst case. This worst-case analysis is quite precise: it suffices that a fraction of u's friends reports $\Theta(k)$ results to match this time complexity.

However a smarter approach is possible that offers the same worst-case time complexity but with a possibly better performance in real scenarios. The idea is to initialize the Max-Heap with the top-1 result from each friend of u (not the top-k), and then repeat the three steps above until k *distinct* nodes have been extracted from the max-heap. We notice that these nodes may come from different adjacency lists and in multiple copies from them. So from the one hand this approach guarantees that an adjacency list is examined only if it may potentially include one of the top-k final results; but, from the other hand, this algorithm may subtly induce more than k steps because of the presence of duplicates in the adjacency lists of the friends of u. These duplicates are clearly at most $k \times d(u)$ but they may be significantly less depending on the graph's structure. This is the algorithmic approach to top-k over FoF we will experiment in the following sections.

5. EXPERIMENTAL RESULTS

All the algorithms were implemented in C++11 and compiled with GCC 4.9.1 with the highest optimization settings. The tests were performed on a machine with 24 Intel Xeon E5-2697 Ivy Bridge cores (48 threads) clocked at 2.70Ghz, with 64GiB RAM, running Linux 3.12.7.

The data structures were saved to disk after construction, and memory-mapped to perform the queries. The timings for each query are derived by averaging the last three measurements out of four total measurements. In the following all reported query times are in microseconds (μs) and spaces are in megabytes (MBytes).

The source code is available at `http://github.com/nopper/compressed-indexes-string/tree/www15` for the reader interested in replicating the experiments.

Datasets. We used the following three networks.

- Dblp. The co-authorship network built from a DBLP data downloaded from `http://dblp.uni-trier.de/xml` in October 2014. The graph is available at `http://zola.di.unipi.it/rossano/dblp.tgz`.
- LiveJournal is a snapshot of the friendship network of LiveJournal blogging community crawled in 2006 [3]. In this graph, there exists a directed edge from u to v when u is a friend of v.
- Twitter is a snapshot crawled starting on June 6th and lasting until June 31st, 2009 [23]. In this graph, there exists a directed edge from node u to node v when u *follows* v in Twitter.

We preferred these datasets among other freely available ones because they were the most complete. In particular, we excluded a Facebook snapshot [21] because about 51 million nodes, out of its about 59 million nodes, have degree 1, which is very far from being a realistic characteristic of this social network.

Table 1 reports some basic statistics on our datasets. The column $|\mathcal{D}|$ indicates the size in characters of the dictionary of string names of graph nodes. We notice that the size of N_2 makes unfeasible any approach which attempts its indexing directly. Indeed, the number of elements in N_2's lists w.r.t. N_1's ones grows by a factor ≈ 32 on Dblp, a factor ≈ 49 on LiveJournal, and a factor ≈ 4380 on Twitter, which in turn induce similar increases in the index space.

Implementation of the key step. It is the one that searches the pattern P in the adjacency list of a given node u. We used this step to solve prefix-search for P over Friends of u, and to solve FoF-query by iterating it over all the adjacency lists of the friends of u (for details see Section 4).

According to what we discussed in the previous sections, we provide the following three implementations.

- Intersect answers the query via standard information retrieval approaches by weakly intersecting V_P and the adjacency list of u.
- Scan answers the query by decompressing and scanning the adjacency list of u and by checking whether the queried prefix P prefixes the name of each processed node. Instead of comparing the pattern and each name character-by-character, we store in a global array the rank of each name in the alphabetic ordering. Thus, it suffices to report any node whose rank belongs to the interval $\langle l_P, r_P\rangle$. This is a necessary and sufficient condition for a name to be prefixed by P (see beginning of Subsection 4.1).
- Range answers the query by executing two NextGEQ operations (one for l_P and one for $r_P + 1$) over the adjacency list of u to identify the contiguous range of nodes which are prefixed by P. Reporting those nodes requires the scan of this range.

In all solutions we represent the adjacency lists with any of the compression schemes for integer sequences described in Section 3. In our experiments we tried four of them (namely, Elias-Fano, Interpolative, OptPFD, and Varint-G8IU) since they offer various space/time trade-offs[2].

We remark that there exist compression schemes specifically designed to achieve higher compression on graphs, especially Web graphs [6, 8]. But these representations support basic operations onto the link-structure of the input graph, such as the retrieval of the whole adjacency list of a

[2]Code is available at `http://github.com/ot/partitioned_elias_fano`. See [30] for more details.

Dataset	$\|V\|$	$\|E\|$	Avg. Degree	$\|N_2\|$	$\|N_2\|/\|V\|$	$\|\mathcal{D}\|$ in chars
Dblp	1,420,763	12,005,120	8.5	383,444,104	269.9	20,054,749
LiveJournal	4,846,608	68,475,391	14.1	3,370,481,580	695.4	56,269,582
Twitter	41,652,229	1,468,365,182	35.3	6,391,337,859,405	153,445.3	406,349,035

Table 1: Basic statistics on our three datasets.

		Pattern length $\|P\|$				
Dataset	Space	1	2	3	4	5
Dblp	14.43	0.41	0.50	0.51	0.54	0.56
LiveJournal	42.74	0.42	0.52	0.54	0.57	0.60
Twitter	316.63	0.50	0.58	0.59	0.63	0.64

Table 2: Space occupancy (in MBytes) and average query time (in μs) of CPI.

given node. Therefore the only way to solve our queries over these representations would be to scan the whole list and check for possible results. Experiments in [8] on LiveJournal show that the fastest representations scan an adjacency list by taking 2 μs per node; this is much higher than what we aim for in our setting where baselines achieve timings in the order of tens of nanoseconds per node on the same dataset. For this reason we do not experiment with them.

Prefix Search in $\mathcal{D}$. The first step of any solution is to perform a prefix-search for a given pattern P in the dictionary $\mathcal{D}$. As we mentioned in Section 3 this is a very well-studied problem and several solutions exist. Comparing the plethora of solutions known for this problem is out of the scope of this paper. We only report the time/space efficiency of the compressed permuterm index (CPI) described in [17][3]. This is a compressed solution that is competitive in query time w.r.t. uncompressed ones (e.g., tries) [17]. Anyway, we remark that any other prefix-search data structure could be plugged in without jeopardizing the conclusions of this paper.

Table 2 reports CPI's space occupancy and its average query time by varying the pattern length from 1 to 5 characters. Experiments were performed by searching 1000 patterns for each length which were randomly selected from $\mathcal{D}$ (i.e., succesful searches). In the subsequent experiments we do not account for the prefix-search time and space occupancy.

Space occupancy. In Table 3 we report the space occupancy to represent the adjacency lists of each graph with different compression schemes. Note that both Intersect and Scan work on lists encoding the original nodeIDs; conversely, Range renames nodeIDs accordingly to the alphabetic rank of the corresponding names.

Interpolative is always the most performant encoder, with a gain between $11\% - 21\%$ on original nodeIDs, and $4\% - 28\%$ on alphabetic nodeIDs.

We point out that all integer encoders require more space when alphabetic nodeIDs are used, except for Elias-Fano which is independent on the renumbering of nodeIDs because of its properties. This is a very important outcome of this experiment because we have shown theoretically (and we will prove experimentally) that alphabetic nodeIDs are crucial to achieve efficient time performance on our prefix-queries on friends and FoF. Consequently Elias-Fano will be the winning encoding choice in terms of time efficiency in our solutions, and this will come at the cost of loosing a small percentage (i.e., $2\% - 6\%$) in space occupancy wrt Interpolative which however may be significantly slower (i.e., $2.2 \div 4.0$ times).

[3]An implementation of CPI is available at `http://code.google.com/p/cpi00`

	Dblp		LiveJournal		Twitter	
Encoding	Space	bpe	Space	bpe	Space	bpe
Varint-G8IU	33.86	22.87	163.49	19.57	3836.72	21.70
Interpolative	29.77	20.04	139.40	16.63	3067.45	17.32
OptPFD	33.70	22.76	160.26	19.17	3440.63	19.44
Elias-Fano	33.94	22.93	177.35	21.26	3513.07	19.86
Varint-G8IU	38.43	26.04	208.02	25.00	4271.63	24.18
Interpolative	32.06	21.63	170.50	20.42	3441.41	19.45
OptPFD	38.23	25.90	204.57	24.58	3856.98	21.82
Elias-Fano	33.94	22.93	177.35	21.26	3513.07	19.86

Table 3: Index space occupancies in MBs and bits per edge (bpe) for each dataset by varying the compression scheme. The top part refers to the compression of original nodeIDs, the bottom part refers to the compression of *alphabetic* nodeIDs, according to our renumbering scheme at the beginning of Section 4.1.

Prefix-search over friends. Our first experiment reports the query time of each solution in solving prefix-search over friends (space has been given in Table 3). In order to run our experiment we need a pattern P and a node u. We selected 1000 patterns of length from 1 to 5 at random from $\mathcal{D}$, and we selected 10,000 nodes u as follows.

For each graph, we identified the 11-quantiles in its degree distribution. For each pair of consecutive 11-quantiles, we selected uniformly at random a group of 1000 nodes having degree between two consecutive quantiles. This way, we have 10 groups of 1000 nodes each, hence 10,000 nodes in total. This selection is very close to a uniform selection of 10,000 nodes from V, and the subdivision in groups will be useful in the subsequent experiments where our goal will be to study the impact of node degree on query time.

We report in Table 4 the average query time in microseconds of the three solutions mentioned at the beginning of this section, and study how this time depends onto the pattern length and the compression scheme in use. Due to space limitations, we only report results for LiveJournal and Twitter. We observed similar results on Dblp.

These results lead us to three main conclusions which are inspired by the three main horizontal bands in which the Table 4 can be divided according to the proposed approaches.

The first conclusion concerns with Intersect. Since this solution has to process each element in V_P, its poor performance is not surprising when answering queries for short patterns because they induce a large V_P. Interestingly, even if its

		LiveJournal					Twitter				
	Encoding	$\|P\|=1$	$\|P\|=2$	$\|P\|=3$	$\|P\|=4$	$\|P\|=5$	$\|P\|=1$	$\|P\|=2$	$\|P\|=3$	$\|P\|=4$	$\|P\|=5$
Intersect	Varint-G8IU	26,350.37	2166.34	425.54	101.27	41.54	219,029.26	20,280.51	3436.84	776.84	295.10
	Interpolative	26,314.67	2164.79	426.07	102.26	42.61	220,805.02	20,366.81	3472.17	799.64	313.35
	OptPFD	26,318.48	2164.56	425.81	101.97	42.30	220,816.15	20,356.54	3460.55	788.96	304.41
	Elias-Fano	26,307.03	2161.47	424.05	100.58	40.94	220,466.91	20,333.70	3452.83	782.82	298.50
Scan	Varint-G8IU	1.77	1.54	1.52	1.52	1.51	29.27	27.59	27.59	27.41	27.48
	Interpolative	2.75	2.51	2.49	2.49	2.49	39.25	37.93	37.80	37.76	37.86
	OptPFD	2.68	2.44	2.41	2.40	2.40	28.60	27.21	26.93	26.74	26.77
	Elias-Fano	1.75	1.51	1.48	1.48	1.48	43.95	42.21	42.20	42.09	42.19
Range	Varint-G8IU	1.14	0.99	0.97	0.97	0.98	2.20	1.42	1.31	1.27	1.27
	Interpolative	1.95	1.79	1.77	1.77	1.77	5.72	3.57	3.36	3.30	3.30
	OptPFD	1.82	1.67	1.65	1.64	1.64	3.77	2.47	2.31	2.26	2.25
	Elias-Fano	0.87	0.70	0.68	0.67	0.67	2.04	1.02	0.88	0.84	0.83
	Avg. $\|V_P\|$	431,055	41,869	8896	2326	975	3,135,085	330,600	60,828	14,810	5769
	Avg. # results	9.68	1.76	1.15	1.04	1.02	131.09	14.38	4.43	2.12	1.59

Table 4: Average query time (in μs) to answer a prefix-search over the friends of a node u in either dataset LiveJournal or Twitter, by varying the length of the pattern P.

performance improves rapidly, Intersect is noncompetitive for longer patterns as well: when $|P| = 5$, Intersect is always slower than Range by at least 62 times on LiveJournal, and 360 times slower on Twitter. One may consider to precompute sets V_P and augment them to support fast NextGEQ operations. This would certainly speed up intersections and, thus, the overall query processing of this solution. However, due to the space overhead introduced by augmentation, this approach can be applied only on short patterns, thus inheriting the slow performance on long patterns. This leads us to exclude Intersect from the next experiments because such limitations cannot be reverted on the next, more difficult, types of queries involving FoF and top-k.

The second conclusion is about Scan, for which Varint-G8IU, except for few exceptions, is the fastest encoding. This leads us to choose Varint-G8IU as integer compressor for the approach Scan in the next experiments.

The third and last conclusion is about Range. Elias-Fano exhibits the best performance on any dataset. This leads us to use Elias-Fano as integer compressor for the approach Range in the next experiments.

In conclusion we can observe that Range (with Elias-Fano) is faster than Scan (with Varint-G8IU) by a factor ranging from 2 to 2.25 on LiveJournal, and from 14.3 to 33.1 on Twitter. Observe that this gap increases as the pattern length increases too. This is a virtue of Range which requires just two NextGEQ operations to identify the range in N_1 that contains all the query results, then it needs to decode only values within this range. This implies that its query time decreases as the number of query results decreases too (which occurs when P's length increases). This favorably compares with Scan that always needs to decode and scan the whole list N_1, regardless the number of query results.

As far as space occupancy is concerned, Range with Elias-Fano (8th row in Table 3) is $\approx 10\%$ better than Scan with Varint-G8IU (1st row) on Twitter, but it is $\approx 8\%$ worse than Scan (with Varint-G8IU) on LiveJournal.

Prefix-search over FoF. In the next experiments we compare Range and Scan in answering prefix-search over FoF, given their best performance above. Recall that in this type of query, given a prefix P and a node u, we use one of these two solutions to process the adjacency lists of each friend of u to identify those nodes which are prefixed by P. By the considerations above, we expect that the performance gap between Range and Scan will be amplified because each solution is run over the adjacency lists of the, potentially many, friends of u. We report in Table 5 the query time to answer prefix-search query on FoF over our datasets, and we also specify the average number of results returned for each query. We run this experiment by using the same patterns (1000) and nodes (10,000) of the previous paragraph, now prefix-querying over their FoF. The results adhere to our expectations: Range is significantly faster than Scan on any dataset for any pattern-length. The space occupancy remains the one discussed in the previous paragraph. The smallest gain is on Dblp where Range is faster than Scan by a factor from 1.9 to 2.2, the largest gain is on Twitter where the improvement is by a factor from 4 to 38 depending on the pattern length. Again, the gain factor increases with the pattern length and on the network size.

In Figure 1 we plot the average query time of Range and Scan on Twitter for patterns of length 1 and 5. Nodes are divided into 10 groups accordingly to the 11-quantiles as described at the beginning of the previous paragraph. The largest gain is on the second group (a factor 5.4 for $|P| = 1$ and a factor 201 for $|P| = 5$) while the smallest gain is on the last group (a factor 3.8 for $|P| = 1$ and a factor 35 for $|P| = 5$).

A more careful analysis of the query time of the two solutions provides a precise and clear explanation of all our experiments. Given a pattern P and a queried-node u, the query time $T_{\mathsf{Range}}(u, P)$ of Range is approximately equal to the sum of three cost terms. The first one is the cost of identifying the range on nodes prefixed by P in each u's friend lists, which is equal to $2\ T_{\mathsf{EFNextGEQ}} \times d(u)$ time, where $T_{\mathsf{EFNextGEQ}}$ is the time of a NextGEQ with Elias-Fano. Second, the cost of decoding nodes within each of these ranges, which equals $T_{\mathsf{EFAccess}} \times |R_u(P)|$, where T_{EFAccess} is the cost of an Access operation with Elias-Fano and $|R_u(P)|$ is total length of these

Dblp

Solution	Pattern length $\|P\|$ 1	2	3	4	5
Scan (Varint-G8IU)	48	43	42	41	40
Range (Elias-Fano)	25	21	20	19	18
Avg. # results	98.80	45.37	25.06	14.65	3.38

LiveJournal

Solution	Pattern length $\|P\|$ 1	2	3	4	5
Scan (Varint-G8IU)	206	147	146	145	145
Range (Elias-Fano)	123	68	66	65	66
Avg. # results	551.81	21.13	4.41	2.14	1.70

Twitter

Solution	Pattern length $\|P\|$ 1	2	3	4	5
Scan (Varint-G8IU)	133,006	103,560	99,895	99,666	99,540
Range (Elias-Fano)	33,019	5924	2998	2783	2635
Avg. # results	90,017	12,573	1925	896	385

Table 5: Average query time (in μs) to answer a prefix-search over the FoF of a node u in all datasets, by varying the length of the pattern P.

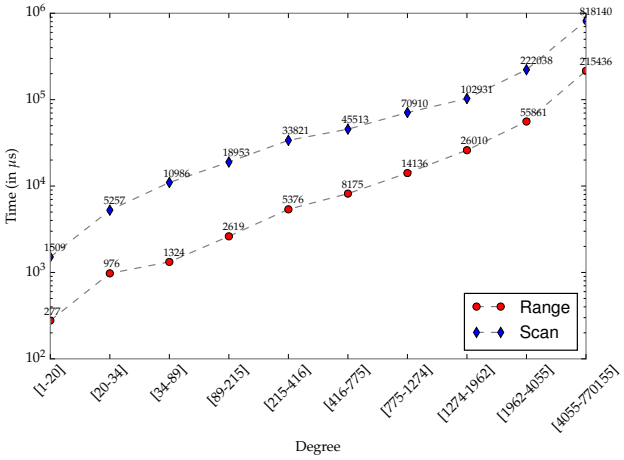
ranges (i.e., the number nodes in u's friends lists which are prefixed by P). Third, the time $T_{\mathsf{Report}}(u, P)$ to manage and report query-results (i.e., sort them and remove duplicates), which is independent on the particular solution in use, and depends only on size of the candidate-list of results.

The query time of Scan is approximately $T_{\mathsf{Scan}}(u, P) = T_{\mathsf{VIAccess}} \times \hat{N}_2(u) + T_{\mathsf{Report}}(u, P)$, where T_{VIAccess} is the time of a Access operation with Varint-G8IU and $\hat{N}_2(u)$ is the number nodes in u's friends lists. The number $\hat{N}_2(u)$ is always at least the size of the FoF network of u (i.e., $N_2(u)$), and it may be larger because it accounts also for duplicates.

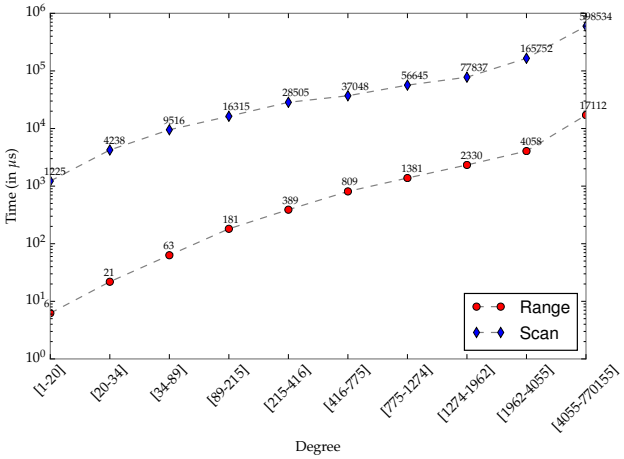
Even if T_{VIAccess} is smaller than T_{EFAccess}, which, in turn, is smaller than $T_{\mathsf{EFNextGEQ}}$ [30], the large difference in size between $R_u(P)$ and $\hat{N}_2(u)$ significantly favours Range query time, especially in larger graphs.

By increasing the pattern length, $T_{\mathsf{Range}}(u, P)$ decreases because both $R_u(P)$ and $T_{\mathsf{Report}}(u, P)$ decrease too. Instead, the first term in $T_{\mathsf{Scan}}(u, P)$ remains fixed and only $T_{\mathsf{Report}}(u, P)$ decreases. Thus, T_{Range} decreases more rapidly than T_{Scan} by increasing the pattern length and, in fact, its gain improves as shown in Table 5.

Finally, we observed a significant drop in the gain of $T_{\mathsf{Range}}(u, P)$ versus $T_{\mathsf{Scan}}(u, P)$ when the pattern length is fixed but u's degree grows. At a first glance, one would be temped to think that this is due to the first term 2 $T_{\mathsf{EFNextGEQ}} \times d(u)$ in $T_{\mathsf{Range}}(u, P)$ that, obviously, increases with u's degree. However this is not the case, just look at the query times of Range for the last bins in Figure 1: in average Range requires 215,436 μs for $|P| = 1$ and 17,122 μs for $|P| = 5$. Thus, since Range executes exactly the same number of NextGEQ operations in solving queries with different pattern lengths on the same node u, the time cost of all these operations cannot be larger than 17,122 μs. Surprisingly, experiments show that more than half of the query time is spent in managing and reporting query-results (i.e., the term $T_{\mathsf{Report}}(u, P)$). For example, $T_{\mathsf{Report}}(u, P)$ is roughly 110,000 μs in the last bin of Figure 1 for $|P| = 1$, where queries have 3,227,675 results on average. These considerations explain the drop in the gain



(a) $|P| = 1$



(b) $|P| = 5$

Figure 1: Average query time (in μs) to answer a prefix-search over FoF on Twitter for $|P| = 1$ and $|P| = 5$. The queried nodes are divided into 10 groups based on their degrees. The y-axis is in log-scale.

achieved by Range over Scan when querying higher degree nodes, because of an increase in the number of candidate results to process and report; and, furthermore, they motivate the interest for efficient solutions to the top-k variant of the problem because the reporting step is absent there since the reporting is confined to just k nodes (and thus it is independent of the range size).

Top-k prefix-search over FoF. The previous experiments established that Range is the clear winner. In this paragraph we make a step forward by experimenting over the algorithms proposed in Subsection 4.2 for retrieving top-k results. Due to space limitations, we focus on answering top-k prefix-search over FoF. We experiment the following solutions.

- Range+Score implements the simple strategy which retrieves and scores all the results identified by applying Range over the friends of the queried node u. A Max-Heap is used to keep only the k largest scores. This

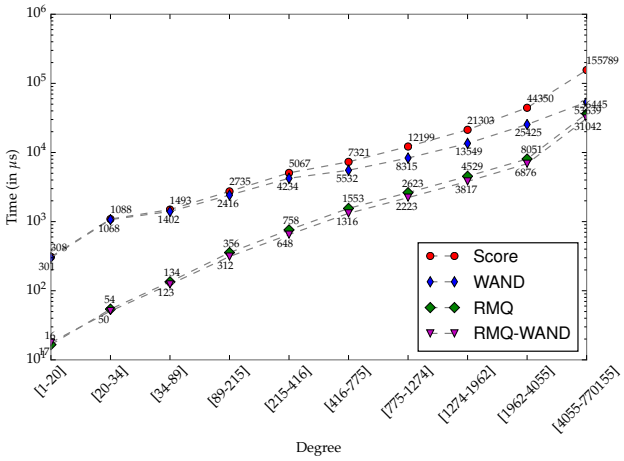


Figure 2: Average query time (in μs) to answer a Top-k ($k = 10$) prefix-search over the FoF on Twitter for the nodes divided into 10 groups based on their degrees and patterns of length 1. The y-axis is in log-scale. All solutions are based on Range which is then dropped from the legend.

solution serves as a baseline to show the improvement induced by strategies targeted to the top-k problem.

- Range+WAND implements a solution inspired by early termination strategies in Search Engines. The solution keeps a global array M, entry $M[v]$ stores the maximum score of a node in $N_1(v)$. This way, we process the friends of u by starting from the most promising ones (i.e., the nodes v with largest values of $M[v]$). This gives the chance of early stopping the processing of u's friends.

- Range+RMQ implements the solution described in Subsection 4.2, which uses an intermingled execution of RMQ-queries to speed up the detection of top-k nodes.

- Range+RMQ+WAND is an hybrid approach that has been inspired by experimental results, which show the complementary characteristics of the previous two approaches. First, the solution runs Range+RMQ on nodes with a degree larger than a threshold D, thus computing and inserting their top-k in the Max-Heap; then, it processes the remaining low-degree nodes with the WAND strategy. In our experiments the threshold D has been fixed to 1000.

In our experiments we use the degree of a node as its score. We remark that our solutions are indenpendent of the scoring functions. Indeed, we obtained the same results with a random scoring generation. We run our experiments by using the same set of patterns and nodes as in the previous paragraphs. In Figure 2 we plot the query time on Twitter by varying the node degree and by fixing $k = 10$ and the pattern length to 1. Due to space limitations, we do not show pictures for other experiments which are, nevertheless, commented below.

We notice that, albeit Range+Score is the same solution as in Figure 1, the performance here are improved on nodes with larger degrees. This is because in solving top-k there is no need to sort and deduplicate all the matching results (as observed at the end of the previous paragraph). However, all these matching results need to be scored, which is a non-negligible task. This explains the limited improvement on high-degree nodes and the slightly worse performance on low-degree ones.

In any case our algorithm based on RMQ gives a significant improvement over Range+Score. The gain is a factor ranging from 2.9 to 20.2. The gain decreases with the increasing of the queried-node degree.

A comment is in order now. Even if Range+WAND seems to be noncompetitive w.r.t. Range+RMQ, intuition suggests that these two strategies complement each other. Indeed, Range+RMQ is very efficient on high-degree friends of u because they are likely to have much more than k matching results and, thus, RMQ operations allow us to skip most of them. Range+WAND, instead, is more likely to exclude low-degree friends of u which will have lower maxima on average. Experiments confirm this intuition: Range+RMQ+WAND improves Range+RMQ by a factor up to 1.7. The largest improvements are on large degree nodes where Range+RMQ has the smallest gain with respect to the baseline Range+Score. This way, our solutions improve the gain over the baseline Range+Score, which is now at least a factor 5.0 instead of 2.9.

We finally report that the time gain of the best solutions decreases on longer patterns, vanishing with patterns of length $|P| = 5$. This is due to the fact that the pattern's occurrences are few enough that Range+Score is sufficiently fast to identify the top-k nodes among them. We remark that the reduced time gain of Range+RMQ over long patterns does not occur for graphs of average degree smaller than Twitter and, we foresee that, this will not occur for a larger snapshot of Twitter where we expect more pattern's results to be scanned, so that Range+Score will turn out to be slower than Range+RMQ on par of what happens on our snapshot for shorter patterns.

6. FUTURE WORK

First of all we would like to experiment our solutions on much larger networks. As shown in our experiments, the gain of our solutions increase with the social network size.

Second, we would like to extend our solutions to the case of queries over *multi-attribute* nodes, such as age, geographic location, preferences, and so on. The simplest approach to deal with this scenario would be to apply our node-renumbering scheme onto each attribute, thus blowing-up the space by a factor proportional to the number of indexed attributes. We foresee to design solutions which do not pass through an *explicit replication* of the graph, but rather exploit information-theoretic ideas to encode succinctly different *permutations* of the same adjacency lists.

Finally we remark that our graphs were *static* so that every node/edge change would require a reconstruction of the corresponding part of the index. In a more realistic setting, we would need to update efficiently the index by e.g. any known dynamization technique [26,31]. An experimental evaluation on real graph-update traces is foreseen.

Acknowledgments

We wish to warmly thank Domenico Dato and Daniele Vitale (Istella, Tiscali) for exposing us to some of the problems we

addressed in this paper and for fruitful discussions. This work was partially supported by MIUR PRIN ARS-Technomedia.

7. REFERENCES

[1] S. Alstrup, G. S. Brodal, and T. Rauhe. Optimal static range reporting in one dimension. In *STOC*, pages 476–482, 2001.

[2] A. Amir, M. Lewenstein, and N. Lewenstein. Pattern matching in hypertext. In *WADS*, LNCS 1272, pages 160–173, 1997.

[3] L. Backstrom, D. P. Huttenlocher, J. M. Kleinberg, and X. Lan. Group formation in large social networks: membership, growth, and evolution. In *KDD*, pages 44–54, 2006.

[4] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *LATIN*, pages 88–94, 2000.

[5] P. Boldi, M. Santini, and S. Vigna. Permuting web and social graphs. *Internet Mathematics*, 6(3):257–283, 2009.

[6] P. Boldi and S. Vigna. The webgraph framework I: compression techniques. In *WWW*, pages 595–602, 2004.

[7] N. Brisaboa, R. Cánovas, F. Claude, M. Martínez-Prieto, and G. Navarro. Compressed string dictionaries. In *SEA*, LNCS 6630, pages 136–147, 2011.

[8] N. R. Brisaboa, S. Ladra, and G. Navarro. k2-trees for compact web graph representation. In *SPIRE*, pages 18–30, 2009.

[9] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Y. Zien. Efficient query evaluation using a two-level retrieval process. In *CIKM*, pages 426–434, 2003.

[10] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. On compressing social networks. In *ACM SIGKDD*, pages 219–228, 2009.

[11] M. Curtiss and *et al.* Unicorn: A system for searching the social graph. *VLDB*, 6(11):1150–1161, Aug. 2013.

[12] B. B. Dalvi, M. Kshirsagar, and S. Sudarshan. Keyword search on external memory data graphs. *VLDB Endow.*, 1(1):1189–1204, 2008.

[13] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.

[14] R. M. Fano. On the number of bits required to implement an associative memory. *Memorandum 61, Computer Structures Group, MIT, Cambridge, MA*, 1971.

[15] P. Ferragina and R. Grossi. The String B-tree: A new data structure for string search in external memory and its applications. *J. ACM*, 46(2):236–280, 1999.

[16] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *J. ACM*, 57(1), 2009.

[17] P. Ferragina and R. Venturini. The compressed permuterm index. *ACM Transactions on Algorithms*, 7(1):10, 2010.

[18] P. Ferragina and R. Venturini. Compressed cache-oblivious String B-tree. In *ESA*, pages 469–480, 2013.

[19] J. Fischer. Optimal succinctness for range minimum queries. In *LATIN*, pages 158–169, 2010.

[20] E. Fredkin. Trie memory. *Communication of the ACM*, 3(9):490–499, Sept. 1960.

[21] M. Gjoka, M. Kurant, C. Butts, and A. Markopoulou. Walking in facebook: a case study of unbiased sampling of osns. In *IEEE Conference on computer communications*, 2010.

[22] B. P. Hsu and G. Ottaviano. Space-efficient data structures for top-k completion. In *WWW*, pages 583–594, 2013.

[23] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW*, pages 591–600, 2010.

[24] U. Manber and S. Wu. Approximate string matching with arbitrary costs for text and hypertext. In *Proc. IAPR Workshop on Structural and Syntactic Pattern Recognition*, pages 22–33, 1992.

[25] C. D. Manning, P. Raghavan, and H. Schülze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[26] K. Mehlhorn and M. H. Overmars. Optimal dynamization of decomposable searching problems. *Inf. Process. Lett.*, 12(2):93–98, 1981.

[27] A. Moffat and L. Stuiver. Binary interpolative coding for effective index compression. *Inf. Retr.*, 3(1):25–47, 2000.

[28] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *SODA*, pages 657–666, 2002.

[29] G. Navarro. Improved approximate pattern matching on hypertext. In *LATIN*, Lecture Notes in Computer Science, Vol. 1380, pages 352–357, 1998.

[30] G. Ottaviano and R. Venturini. Partitioned Elias-Fano indexes. In *SIGIR*, pages 273–282, 2014.

[31] M. H. Overmars. *The Design of Dynamic Data Structures*. Lecture Notes in Computer Science #156, Springer, 1983.

[32] D. Salomon. *Variable-length Codes for Data Compression*. Springer, 2007.

[33] P. Sarkar and A. W. Moore. Fast nearest-neighbor search in disk-resident graphs. In *ACM SIGKDD*, pages 513–522, 2010.

[34] A. A. Stepanov, A. R. Gangolli, D. E. Rose, R. J. Ernst, and P. S. Oberoi. Simd-based decoding of posting lists. In *CIKM*, pages 317–326, 2011.

[35] F. M. Suchanek and G. Weikum. Knowledge bases in the age of big data analytics. *PVLDB*, 7(13):1713–1714, 2014.

[36] J. Ugander and L. Backstrom. Balanced label propagation for partitioning massive graphs. In *WSDM*, pages 507–516, 2013.

[37] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow. The anatomy of the facebook social graph. In *Preprint* `arXiv:1111.4503v1`, 2011.

[38] S. Vigna. Quasi-succinct indices. In *WSDM*, pages 83–92, 2013.

[39] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with imized document ordering. In *WWW*, pages 401–410, 2009.

[40] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. In *ICDE*, pages 59–70, 2006.