Compressed Indexes for String Searching in Labeled Graphs

Paolo Ferragina Francesco Piccinno Rossano Venturini
Dipartimento di Informatica, University of Pisa
{ferragina, piccinno, rossano}@di.unipi.it

ABSTRACT

Storing and searching large labeled graphs is indeed becoming a key issue in the design of space/time efficient online platforms indexing modern social networks or knowledge graphs. But, as far as we know, all these results are limited to design compressed graph indexes which support basic access operations onto the *link structure* of the input graph, such as: *given a node u, return the adjacency list of u.*

This paper takes inspiration from the Facebook Unicorn's platform and proposes some *compressed-indexing* schemes for large graphs whose *nodes are labeled with strings of variable length*— i.e., node's attributes such as user's (nick-)name— that support sophisticated *search operations* which involve both the linked structure of the graph and the string content of its nodes.

An extensive experimental evaluation over real social networks will show the time and space efficiency of the proposed indexing schemes and their query processing algorithms.

Categories and Subject Descriptors

H.3.2 [**Information Storage and Retrieval**]: Information Storage; E.4 [**Coding and Information Theory**]: Data Compaction and Compression

Keywords

Compression; Data structures; Social Networks

1. INTRODUCTION

In a recent paper [11], Facebook's engineers introduced the Unicorn system as an online, in-memory social graph-aware indexing platform designed to search huge graphs distributed over many commodity servers. In that paper the authors claimed that *"Unicorn is based on standard concepts in information retrieval, but it includes features to promote results with good social proximity".*

Storing and searching large labeled graphs is indeed becoming a key issue in the design of space/time efficient online platforms indexing modern social networks or knowledge graphs [35]. However, as far as we know, all these results are limited to design compressed graph-indexes (see e.g., [5, 6, 8, 10, 30]) which support only basic access operations onto the *link-structure* of the input graph, such as: *given a node u, return the adjacency list of u.* Other results consider more sophisticated access-patterns to the graph structure, but do not deal with compression issues or labeled nodes [12, 33].

In this paper we take inspiration from the Unicorn's platform and make one step forward in the design of *compressed-indexing* schemes for large graphs whose *nodes are labeled with strings of variable length*, i.e., node's attributes such as user's (nick-)name or country or any other information the user has input for its profile or the system has drawn from user behavior, and support (i) basic access operations into the link-structure of the graph (as above), plus (ii) novel sophisticated *search operations* which involve both the linked structure of the graph and the string content of its nodes.

As an example, we aim at designing a compressed graph index that efficiently supports the Facebook's typeahead search (Section 6 of [11]). Typeahead enables Facebook users to find other users by typing the first few characters of the person's name. For example, if a user is typing in the name of "Jon Jones" the typeahead backend sequentially receives queries for "J", "Jo", "Jon", "Jon ", "Jon J", etc. For each prefix, the system has to return a list of individuals for whom the user might be searching for. Some of these individuals will be within the user's explicit circle of friends or they will be Friends-of-Friends (shortly, FoF).

A simple implementation for the typeahead query among the FoF of a given user could be based on standard information retrieval algorithms in which users' adjacency lists are treated as posting lists. The idea is to solve the query into two main steps: (1) the former step identifies a set of candidate users in the graph having the queried string as a prefix of their names; (2) the latter step reports those candidates which are in the FoF's network of the given user. Step 1 is implemented by searching the queried-prefix in a dictionary containing all users' names; step 2 is implemented by scanning the adjacency list of the given user u and, for each friend of u, looking at her friends (FoF of u) by checking whether they occur also in the candidate set. This solution is consistent with the proposal in [11], where the authors claim that *"The index server stores adjacency lists and performs set operations on those lists, the results of which are returned to the aggregators. In the index server, intersection, union,*

WWW 2015, May 18–22, 2015, Florence, Italy.
ACM 978-1-4503-3469-3/15/05.
http://dx.doi.org/10.1145/2736277.2741140.

and difference operations proceed as is described in standard information retrieval texts".

But unfortunately this simple approach may be computationally expensive because both the candidate set to be generated in step 1 and the FoF's network searched in step 2 could turn out to be very large. A strategy to partially mitigate the cost of step 1 consists in precomputing and storing as posting lists the results of step 1 for short or frequently searched prefixes. For example, the solution in [11] precomputes results for prefixes of few characters. Despite this simple trick, we are not aware of any other solution which is able to answer a query without processing the whole FoF network.

In [11] the authors also discuss variants of this FoF query, by introducing a *social relevance* score (Section 6.1 in [11]) which allows to *rank the FoF users* whose names are prefixes by the queried string. In this case too, the query is answered by scanning the FoF for prefix-match and then ranking the results, thus inheriting the limitations above.

The problem of answering FoF queries has been addressed also in a distributed setting in which a large graph has to be distributed among several servers, with the goal of reducing the traffic among them. Authors in [36] proposed a graph partitioning approach to relocate nodes while respecting strict shard balancing constraints. Even in this distributed setting, we are faced with the problem of answering queries on a single in-memory shard whose inefficient solution may become the dominant cost and may impact on the traffic induced by the partial results sent among machines.

Notation and problems definition. In the following we will refer to a graph $G = (V, E)$ with $n = |V|$ nodes and $m = |E|$ edges. Without loss of generality we focus on social graphs made up of users $u \in V$ which have associated a name $\mathsf{name}(u)$ which is either the user's *screen name* or her *real name*. The dictionary of strings $\mathcal{D}$ contains the names associated with all users in the graph. Graphs representing a (social) network may have types for both nodes — e.g., users, pages, photos, posts, etc. – and edges — e.g., friends, followers, live-in, likers, etc.. Even if, in the rest of the paper, we will restrict our attention to the case in which nodes are labeled with users' names and edges represent their friendships, our solutions can be easily adapted to deal with several other node or edge types, simultaneously.

In this kind of graphs (either directed or undirected) a link between two users u and v exists if both users know each other or are in some sort of relationship (e.g., user u is a follower of v, u liked the posts of v, u is a friend of v, and so on). For the sake of clarity we will refer to this situation as *friendship*: u is friend of v.

For each user u, we use $N_1(u)$ to denote the adjacency list of node u in G, namely, the list of u's friends, and use $d(u)$ to denote the number of nodes in this list or, equivalently, the degree of u. A friend-of-friend (FoF) of a user u is any user which is either a direct friend of u or a friend of a friend of u. In graph terminology, a FoF of u is a node at distance at most 2 from u in G. In the following we will use $N_2(u)$ to denote the list of FoF of u (i.e., $N_2(u) = N_1(u) \cup (\cup_{v \in N_1(u)} N_1(v))$).

Finally, we will assume that each user u is associated with a *relevance score*, denoted $\mathsf{score}(u)$, which quantifies the importance of u in the social network (e.g., the PageRank of u, the number of likes she got, etc.). This score will be used to restrict our graph-search operations on the k neighbors of a given node (e.g., $k = 10$) that match the prefix-query and have the highest scores.

Given Table 1 in [11] (see also Table 1 in our paper), the size of $N_1(u)$ can go from hundreds (`friends`) to thousands or millions (`likers` or `live-in`); and the size of $N_2(u)$ may even blow up to three or more order of magnitudes than the size of $N_1(u)$. This poses challenging efficiency concerns about the storage and brute-force scanning/intersection of those lists for the queries above, even in the in-memory storage of the graph or its parts. Let us then introduce the problems we attack in our paper.

Prefix-search over friends: Given a node u and a string P, search for all friends v of u (i.e., $v \in N_1(u)$) such that $\mathsf{name}(v)$ is prefixed by P.

Prefix-search over FoF: Given a node u and a string P, search for all FoF v of u (i.e., $v \in N_2(u)$) such that $\mathsf{name}(v)$ is prefixed by P.

Top-k prefix-search over friends (or FoF): Given a node u, a string P and a positive integer k, return the top-k *scoring* nodes which are friends (or FoF) of u and are prefixed by P.

In our paper we wish to design a compressed-indexing scheme for labeled graphs which guarantees efficient time and compressed space for all queries mentioned above. Our proposal combines, in a principled way, string searching and graph storage solutions, so that it can be looked at as a basic block upon which one could design more sophisticated queries and compressed graph-indexing platforms.

Our Contributions. In this paper we introduce and efficiently solve the problems above both theoretically and in practice via an extensive experimental analysis over (a large part) of three real networks of increasing size: namely, Dblp of about 1.4 mln nodes and 12 mln edges, LiveJournal of about 4.8 mln nodes and 68.5 mln edges, and Twitter of about 41.6 mln nodes and 1.4 billion edges.

Prefix-search over friends. We propose a simple renumbering of the nodeIDs which takes into account the alphabetic ordering of node's strings and allows to turn the implementation of this query-type from a series of list intersections over spread out integers (a là [11]) to a single (thus fast) range query over an adjacency list. Our experiments on prefix-search over friends show that our solution significantly outperforms baselines based on standard IR approaches. In particular, it improves the approach based on intersections by at least a factor 62, and the approach based on scanning by a factor up to 33 depending on the dataset and the queried-pattern length.

Prefix-search over FoF. Building upon the best approaches for the problem on friends, we turn to range-queries also the problem on FoF and show an improvement of a factor up to 38 over the best IR-based approach. Even if the gain of our solution varies depending on the pattern length and network size, we provide a careful analysis which gives a precise and clear explanation of all our experiments.

Top-k prefix-search over friends and FoF. We propose to augment the previous solutions with a Range-Maximum-Query data structure built over the scores of the friends of every node u in the graph, at the cost of $2 + o(1)$ bits per edge [19]. Then, taking inspiration from [28], we turn the top-k prefix search over friends and FoF of u into a specifically designed intermingled execution of RMQ-queries. We also investigate an hybrid solution based on a principled combination of this RMQ-approach with *early termination* approaches, which are typically used in IR search engines. This allows us to devise a pretty clear solution which significantly beats a strong baseline hinging on an approach that scans and scores the query results. Experiments show that our approach improves the query time by a factor up to 20.

Before proceeding further a comment is in order. The string-matching literature is full of results regarding the *prefix-search* problem on dictionary of strings and labeled trees, possibly compressed (see e.g., [15, 16, 18], and refs therein). About thirty years ago some (theoretical) work has been done also on the topic *pattern-matching on labeled DAGs* (see e.g., [2, 24, 29]), in which queries asked to search for a labeled path in those DAGs.

The problem we propose in our paper is the *first one* addressing the string-matching issue with *some neighbor constraints in labeled and general graphs, with significant practical applications*. Additionally, our extensive experimental tests on large and real graphs will show that our algorithms are not only theoretically valid but also efficient in practice.

2. RELATED WORK

Because of space limitations we refer the reader to the Introduction for other related works on this subject and for the notation we adopt in this paper.

Here we dig a little bit more on the Unicorn system [11]. Since a Unicorn instance (or vertical) may store an index that is too large to fit into the memory of a single machine, the index is broken up into multiple shards such that each shard can fit in a single index server. Queries to Unicorn are broadcast to all index servers, each index server retrieves entities from its shard and returns it to an Aggregator server, which then combines all these entities and returns them to the caller. As a result, queries may require multiple round-trips in order to retrieve objects that are more than one edge away from source nodes.

In order to reduce the round-trips over thousands of machines, and thus optimize the performance, the authors of [36] presented a label-propagation algorithm for finding 'joints' in graphs of particularly massive proportions, with an emphasis on the Facebook social graph. Specifically the goal was to perform graph sharding such that, as often as possible, whenever $v \in N_1(u)$ both u and v belong to the same shard. By doing this well, they reduced the number of other machines that need to be queried, and also reduced the total amount of data that need to be transferred over the network, consequently increasing overall system throughput and latency.

In this paper we introduce a *new re-assignment algorithm for node-IDs and proper compressed data structure and searching algorithms* that may be used either on the FB-graph as a whole, or inside every single FB-shard (as computed by [36]) with the goal of offering guaranteed performance in query time, reduced hops among machines, and compressed space occupancy.

3. BACKGROUND

Social networks (e.g., Facebook [21, 37]) have hubs and a surprisingly large number of FoF per individual. This neighbor structure has substantial algorithmic implications for graph traversals. In particular BFS out to distance two (i.e., touching $N_2(u)$'s neighbors) will potentially query a large number of individuals who may match a prefix-query especially if either the degree of u is large or if P is short. This is the reason why services on social networks show only few (e.g., five or ten) most promising users that match a prefix-query (such as the Typeahead tool in Facebook).

It is therefore obvious that compression issues and properly structured (string-matching) computations come into play if we wish to solve efficiently (in time and space) the problems we stated in the introduction. For the sake of space, we recall here few algorithmic techniques on which our solutions will hinge upon. For each of them, we will make use of state-of-the-art algorithms and their implementations which are available around the Web.

Prefix-search on a dictionary of strings. This is undoubtedly the most well-known problem in data-structural design for strings. It asks for preprocessing a set $\mathcal{D}$ of n strings, having total length L, in such a way that, given a query-pattern P, all strings in $\mathcal{D}$ having P as a prefix can be returned efficiently in time and space. Efficiently means in time proportional to P's length, and in space proportional to L or to the empirical entropy of $\mathcal{D}$'s strings. As far as the output size (i.e., number and length of returned strings) is concerned, we notice that returned strings are contiguous (if dictionary is alphabetically sorted) so they can be represented via a *range* $\langle l_P, r_P \rangle$ of their IDs. Just two integers suffice to encode the occ strings prefixed by P, where $occ = r_P - l_P + 1$. If the explicit strings are required by the underlying application, then a scan of that interval suffices, taking $O(occ)$ time.

The first solution to the prefix-search problem dates back to Fredkin ('60s) [20], who introduced the notion of (compacted) trie to solve it. After this seminal paper, a plethora of different solutions have been introduced to address different issues, namely, compression, I/O-efficiency, cache-obliviousness, and so on (see e.g., [7, 17, 18] and refs therein). We will not dig into these algorithmic details, we content ourselves, for the purposes of this paper, to remind the reader that the prefix-search problem on a dictionary of strings can be solved *optimally in time (I/Os) and in compressed space* [18].

Our solutions will be oblivious w.r.t. the particular data structure used to solve prefix-search queries over $\mathcal{D}$; in the experiments, we adopt the Compressed Permuterm index of [17] because of its compressed space occupancy (up to h-th order entropy of $\mathcal{D}$), very efficient prefix-query time complexity (i.e., $O(|P|)$ time), and appealing practical performance.

Accessing compressed sequences of integers. Representing strictly monotone sequences of integers in compressed space is a crucial problem, studied since the '50s with its most important application in inverted indexes [25]. Here the goal is to minimize the space occupancy of the posting lists with a compression scheme that guarantees the efficient processing of users' queries. Of course, the obvious sequen-

tial scan may be slow, so that engineers resort to *skipping* strategies. The basic idea is to divide a posting list in small blocks that are compressed independently, and to store the maximum integer present in each block. This allows to find and decode only the block that possibly contains the sought integer by scanning the sublist of maxima, thus skipping a potentially large number of useless blocks.

Compression is achieved by delta-encoding the integers within each block, i.e., representing the differences between consecutive integers. These differences may be encoded by any of the known variable-length binary codes: such as *unary codes*, *Elias Gamma/Delta codes*, *Golomb/Rice codes* [32], or the faster *Variable byte codes* (Vbyte) [32, 34], *PForDelta* (PFD) [40] or OptPFD [39]. A completely different integer-encoding approach is taken by *Binary Interpolative Coding* [27], but experiments [30] have shown that, although this produces the most compressed sequence, when the sequence is highly clustered, it results very slow in decoding and thus it is less appealing for posting-list storage and in our setting too.

In the present paper we will concentrate on the *Elias-Fano* representation of monotone sequences [13, 14], which has been applied recently and successfully to the compression of inverted indexes [30, 38], showing both excellent space occupancy and query performance thanks to its efficient random access into a compressed posting list. More precisely, given a monotonically increasing sequence of n integers drawn from an *universe* $[m] = \{0, 1, \ldots, m-1\}$, Elias-Fano can be used to represent the sequence by using at most $n\lceil \log \frac{m}{n} \rceil + 2n + o(n)$ bits and to solve efficiently two key operations Access and NextGEQ by decoding only a small portion of the indexed sequence (see e.g., [30] and refs therein for more details). The Access(i) operation returns the ith element of the sequence. The NextGEQ(d) operation returns the smallest integer in the sequence that is greater than or equal to a given value d. Elias-Fano representation supports constant-time Access and logarithmic-time NextGEQ operations.

These compressed encodings will be at the core of the storage and access to the adjacency lists N_1 of the following known/new solutions.

Range Maximum Queries over integer sequences. Given an integer array $S[1, n]$ the Range-Maximum-Query problem asks to build a data structure that supports efficiently the following query: Given a range $[l, r]$, return the position p of the maximum value in $S[l, r]$. Notice that the value of the maximum can then be computed by accessing $S[p]$. There exist data structures that solve the RMQ-problem in constant-query time and $O(n)$ space in addition to S's storage [4]. By distinguishing the maximum value $S[p]$ from its position p, some authors were able to achieve the optimality in space occupancy without sacrificing the constant query-time [19]. The net result is a solution that takes $2+o(1)$ bits per integer, and still supports RMQ in constant time.

Our solution to the top-k query will deploy this data structure upon the scores of the nodes in the adjacency list N_1. By combining the Elias-Fano storage scheme for adjacency lists and a global array of nodes' scores, we will be able to plug RMQ-query support over NextGEQ operation, by paying only 2 additional bits per integer. Details below.

4. PROBLEMS AND ALGORITHMS

We start by presenting a possible solution to prefix searching over friends (or FoF) that uses standard information retrieval techniques. This approach is consistent with the solution described in [11], and makes use of the following two data structures.

1. The dictionary $\mathcal{D}$ of users' names is indexed with any data structure solving string-prefix searches.
2. The adjacency lists $N_1(u)$, of every node u in the graph, are indexed with compressed representation for integer sequences, supporting efficient Access and NextGEQ operations.

Given a pattern P and a target node u, let us start with solving the *prefix-search over friends* problem. This requires just to scan the friends of u in the graph checking whether their string name(u) is prefixed by P, thus taking $O(|P| \times |N_1(u)|)$ time.

Alternatively, one could resort standard list-intersection operations as follows.

1. Prefix search for P in $\mathcal{D}$ in order to obtain the set $V_P \subseteq V$ of all the nodes v in the graph whose name is prefixed by P.
2. Nodes in V_P are sorted by their identifiers, and the resulting list is intersected with $N_1(u)$.

Step 1 takes $O(|P|)$ time and step 2 requires $O(|V_P| \log |V_P|)$ time to sort V_P and $\min(|V_P|, |N_1(u)|)$ calls to the NextGEQ-operations to intersect nodes. The sorting of V_P is required to speed up the series of NextGEQ-operations over $N_1(u)$, which are faster when querying non-decreasing values. Obviously, the larger is the candidate set V_P, the worse is this solution which, nevertheless, may be appealing because of its simplicity.

More costly is to scale this solution to the *prefix-search over FoF* problem because $N_1(u)$ is substituted by $N_2(u)$ with a consequent significant increase in the total number of nodes to be intersected in step 2. In particular, we observe that checking whether a node of V_P occurs in $N_2(u)$ needs either to materialize $N_2(u)$ and execute $\min(V_P, |N_2(u)|)$ calls to the NextGEQ-operation over it, or it needs to perform $d(u)$ calls to NextGEQ, one call per adjacency list of a friend of u (without thus materializing $N_2(u)$). The former approach is faster, but it incurs in a huge space occupancy (see comments above on FB FoF's network, and our Table 1); the latter approach saves space but it incurs in a total of $O(|V_P| \times d(u))$ executions of NextGEQ.

As far as the *top-k* variant of the two problems above is concerned, it could be solved by keeping only the k nodes with the highest scores during the list intersections. An alternative approach could mimic the *early termination* strategies in inverted indexes (e.g., WAND [9]) and thus compute these top-k nodes without generating the score of all of them. Indeed, the idea could be to store for each node v the maximum score $M(v)$ among the nodes in its adjacency list. At query time, instead of processing all the neighbors of u at once, we can start from the most promising ones (i.e., the ones with the largest values of M), and then immediately exclude the adjacency list of a node v as soon as we recognize that the kth largest score discovered so far is larger than $M(v)$ and, thus, none of v's neighbors may enter in the top-k ones.

Overall these solutions have some serious efficiency drawbacks. The first solution has to decode and scan $N_1(u)$ (and possibly $N_2(u)$) regardless of the size of the final output and pattern length. This may be a problem because of u's degree and the typical size of FoF. The second solution incurs into two other drawbacks. One concerns with the need of materializing (and, possibly, sorting) the set V_P of candidate nodes. This list may be huge, especially if the pattern P is short.[1] This has been recognized as a problem also in [11] and, indeed, they precompute and store the sets V_P for all the patterns P of length at most few characters (e.g., two). Of course, this strategy trades space occupancy for time efficiency but it does not scale with the pattern's length. The other drawback is the need to check, for each element in the (sorted) set V_P, its presence within the friend- or the FoF-list of u. This task is computational demanding whenever the set V_P or the degree of u are, even moderately, large. In fact, although we discussed the similarity with traditional query-processing approaches in posting lists, their adaptation on our problems is slower for two main reasons: (i) a SE-query is usually solved by processing the posting lists of the *few terms* which compose the query, while in our case we have to process the *possibly many* adjacency lists of the neighbors of nodes in $N_1(u)$ (in case of query over $N_2(u)$); (ii) our query can be re-phrased in terms of AND or OR operators in SE-query processing terminology— i.e., V_P AND ($N_1(u)$ OR $N_1(v_1)$ OR ... OR $N_1(v_{d(u)})$), where $v_i \in N_1(u)$— but unfortunately the OR operator is more than 20 times slower than AND [30].

In Section 5 we will investigate the time and space efficiency of all these proposals that will constitute the baselines against which we will compare our novel algorithmic solutions (described below) to the four problems in this paper.

4.1 Prefix-search over friends (or FoF)

Our novel compressed scheme hinges on a simple yet crucial step, that is the *renumbering* of the nodesIDs: *Each node $u \in V$ has assigned a new identifier $\mathsf{rank}(u)$, which is the rank of $\mathsf{name}(u)$ within the alphabetic ordering of all nodes' names in $\mathcal{D}$.* This renumbering bridges nodeIDs with their names so that a node u is prefixed by a pattern P if and only if $\mathsf{rank}(u)$ is within the range $\langle l_P, r_P \rangle$ (see Section 3).

We store any data structure that solves prefix-search queries over the dictionary $\mathcal{D}$ of users' names; and we adopt the new nodeIDs for storing the adjacency list $N_1(u)$ of every node u in the graph. This way, we guarantee the following two key properties.

- For any pattern P, all the nodes prefixed by P in $N_1(u)$ will form a consecutive range, if any: i.e., $\langle l_P, r_P \rangle$. This range can be identified by searching for P in the prefix-search data structure built on the $\mathcal{D}$'s strings, taking $O(|P|)$ time.

- The range can be identified in $N_1(u)$ with only two NextGEQ-operations: it starts at position $l = \mathsf{NextGEQ}(l_P)$ and ends at position $r = \mathsf{NextGEQ}(r_P + 1) - 1$. The range is empty whenever $r < l$.

[1]Table 1 shows that the average size of $N_2(u)$ is from 269 (Dblp) to about 153,445 (Twitter). This means that patterns of few characters will likely occur more often than these numbers.

This simple solution can be generalized to solve FoF queries in either $2d(u)$ NextGEQ-operations, by querying the adjacency list $N_1(v)$ of each friend v of u, or only 2 NextGEQ-operations by materializing and indexing the whole $N_2(u)$.

As a final remark, we notice that, once we renamed the nodes, the query we are solving becomes a well-known *range reporting query*. The problem of designing efficient data structures to answer these types of queries has been studied in the literature where the best (theoretical) data structure [1] is able to retrieve all the, say *occ*, elements within any range in optimal $O(occ)$ time and linear space. Our use of Elias-Fano representation solves that query in slightly suboptimal time (i.e., $O(\log |N_1(u)| + occ)$ worst-case time) but has the great advantage of being very efficient in practice and offer compressed space, as we will show in the experiments.

4.2 Top-k prefix-search over friends (or FoF)

We start by considering the top-k prefix-search queries over friends, and assume that the scores of the nodes are stored in a global table indexed by the new nodeIDs.

For each user u, we define the array $S_1(u)$ that stores the scores of u's friends according to their order in $N_1(u)$ (i.e., for any $1 \le i \le d(u)$, $S_1(u)[i] = \mathsf{score}(N_1(u)[i])$). On top of $S_1(u)$ we build a RMQ data structure that takes $2 + o(1)$ bits per edge, hence almost 2 bits per node in $N_1(u)$ (see Section 3). The crucial observation here is that $S_1(u)$ does not need to be stored because, if the RMQ-query returns position p in $N_1(u)$, we can retrieve the corresponding nodeID by accessing $N_1(u)$ and then using it to get its score from the global table above.

Given these data structures, answering a top-k prefix-search P over the friends of u proceeds as follows.

- We search for P in the data structure built over $\mathcal{D}$, and thus identify the range $\langle l_P, r_P \rangle$ of nodes in the graph that are prefixed by P.

- We map this range onto $N_1(u)$ by computing the positions $l = \mathsf{NextGEQ}(l_P)$ and $r = \mathsf{NextGEQ}(r_P + 1) - 1$.

- We finally identify the k nodes with the largest score in $N_1(u)[l, r]$ by adopting two different approaches. The former is trivial: it scans and scores all the nodes in that range taking time proportional to its size (i.e., $O(r - l + 1)$ time). The latter approach is more sophisticated and computes the k results in $O(k \log k)$ time, hence independent of the range size, by adapting an algorithm originally introduced by Muthukrishnan [28] to solve a different problem, and then extended by various authors to solve top-k strings in a dictionary (see e.g., [22] and refs therein). Here we adapt these approaches to our node-scoring setting as follows:

  (1) The algorithm is recursive and uses an auxiliary max-heap of at most $2k$ elements, each being a triple of integers. The heap is initialized by computing in $O(1)$ time the value $m = \mathsf{RMQ}(l, r)$, which is the position (hence node) in $N_1(u)$ storing the largest score within the range $N_1(u)[l, r]$. Then the triple $\langle l, m, r \rangle$ is inserted in the heap, with priority equal to the score of node $N_1(u)[m]$.

  (2) The algorithm proceeds by repeating three main steps for k times: (i) It picks from the heap the triple with maximum priority (score), say $\langle l, m, r \rangle$; (ii) it

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.

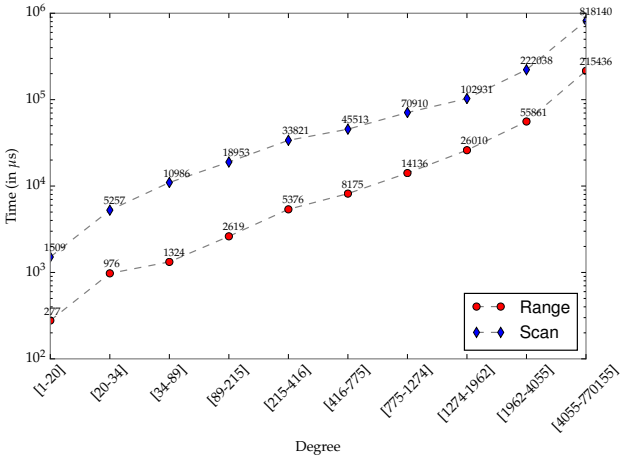
ranges (i.e., the number nodes in u's friends lists which are prefixed by P). Third, the time $T_{\mathsf{Report}}(u, P)$ to manage and report query-results (i.e., sort them and remove duplicates), which is independent on the particular solution in use, and depends only on size of the candidate-list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

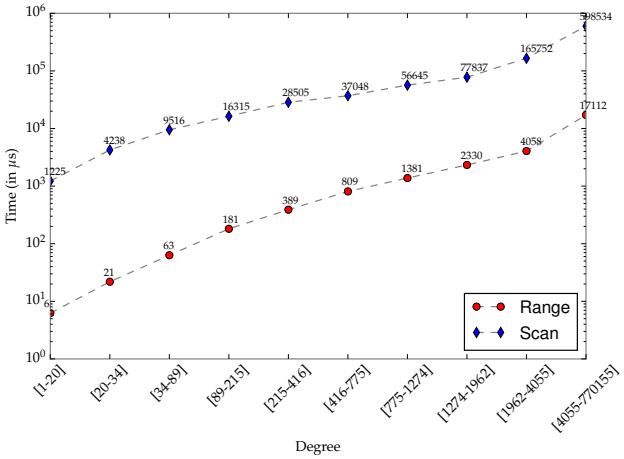
Even if T_{VIAccess} is smaller than T_{EFAccess}, which, in turn, is smaller than $T_{\mathsf{EFNextGEQ}}$ [30], the large difference in size between $R_u(P)$ and $\hat{N}_2(u)$ significantly favours Range query time, especially in larger graphs.

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

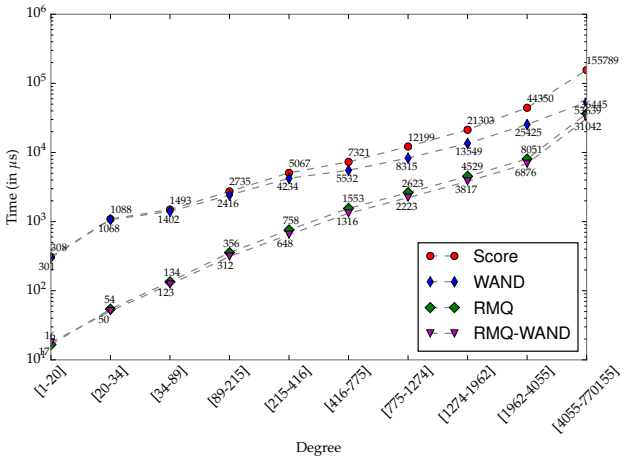


Figure 2: Average query time (in μs) to answer a Top-k ($k = 10$) prefix-search over the FoF on Twitter for the nodes divided into 10 groups based on their degrees and patterns of length 1. The y-axis is in log-scale. All solutions are based on Range which is then dropped from the legend.

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.