Information Systems 39 (2014) 152–174

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/infosys

# Compact representation of Web graphs with extended functionality ☆

Nieves R. Brisaboa [a], Susana Ladra [a,*], Gonzalo Navarro [b]

[a] Database Laboratory, University of A Coruña, Spain
[b] Department of Computer Science, University of Chile, Chile

ARTICLE INFO

*Article history:*
Received 27 January 2013
Received in revised form
22 June 2013
Accepted 4 August 2013
Recommended by: Xifeng Yan
Available online 21 August 2013

*Keywords:*
Web graphs
Compact data structures

ABSTRACT

The representation of large subsets of the World Wide Web in the form of a directed graph has been extensively used to analyze structure, behavior, and evolution of those so-called Web graphs. However, interesting Web graphs are very large and their classical representations do not fit into the main memory of typical computers, whereas the required graph algorithms perform inefficiently on secondary memory. Compressed graph representations drastically reduce their space requirements while allowing their efficient navigation in compressed form. While the most basic navigation operation is to retrieve the successors of a node, several important Web graph algorithms require support for extended queries, such as finding the predecessors of a node, checking the presence of a link, or retrieving links between ranges of nodes. Those are seldom supported by compressed graph representations.

This paper presents the $k^2$-tree, a novel Web graph representation based on a compact tree structure that takes advantage of large empty areas of the adjacency matrix of the graph. The representation not only retrieves successors and predecessors in symmetric fashion, but also it is particularly efficient to check for specific links between nodes, or between ranges of nodes, or to list the links between ranges. Compared to the best representations in the literature supporting successor and predecessor queries, our technique offers the least space usage (1–3 bits per link) while supporting fast navigation to predecessors and successors (2–8 μs per neighbor retrieved) and sharply outperforming the others on the extended queries. The representation is also of general interest and can be used to compress other kinds of graphs and data structures.



## 1. Introduction

Coherent subsets of the World Wide Web, such as crawls over large domains (e.g., comprising whole countries and other geographic areas, languages, cultures, or organizations) are called *Web graphs*. The representation of Web graphs as directed graphs is extremely useful to analyze their structure, behavior, and evolution. Typically, the graph is constructed by defining each Web page as a graph node, and each hyperlink as a graph edge, but it is also possible to build coarser graphs by considering that nodes are, for example, domains or servers. Various analyses of Web graphs then boil down to running (often classical) graph algorithms on their directed graph representation.

For example, Donato et al. [1] show that several common Web mining techniques, used to discover the structure and evolution of Web graphs, build on classical graph algorithms such as depth-first search, breath-first-search, reachability, and weakly and strongly connected components.

☆ A preliminary partial version of this paper appeared in Proceedings of SPIRE 2009, pp. 18–30.

* Corresponding author: Facultad de Informática, Campus de Elviña s/n, 15071 A Coruña, Spain. Tel.: +34 981167000; fax: +34 981167160.
*E-mail addresses:* brisaboa@udc.es (N.R. Brisaboa), sladra@udc.es (S. Ladra), gnavarro@dcc.uchile.cl (G. Navarro).

0306-4379/$ - see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.is.2013.08.003

Probably the most important analysis task on Web graphs is related to measuring the importance of Web pages, as it lies at the heart of successful search engines. For example, one of the most important algorithms to find hubs and authorities on the Web, HITS [2], starts by selecting random pages and finding the induced subgraphs, which are the pages that point to or are pointed from the selected pages. Forward and backward navigation is also inherent to the definition of the well-known PageRank algorithm [3,4], as well as variants such as Truncated PageRank [5] (especially if one wishes to estimate PageRank for isolated pages rather than for the whole graph [6]).

Detecting Web spam is another important problem that can be addressed via Web graph analysis. Saito et al. [7] present a technique that boils down to algorithms for finding strongly connected components, for clique enumeration, and for minimum cuts. Becchetti et al. [5] show that detecting deceiving link structures is intimately tied to the problem of estimating the number of "supporters" of a node, which are those that can be found by navigating backwards a small distance. They develop alternative forward-navigation techniques to avoid the need of backward navigation. They also mention various measures useful to detect Web spam that require forward and backward navigation capabilities, such as "edge-reciprocity", PageRank distribution of iterated predecessors at small distance [8], and backward propagation of "spamicity" from known spam nodes [9].

Yet another important target of Web graph analyses is the detection of communities, which is interesting both to understand social processes and to better target searches and advertisement. Kumar et al. [10] are strong supporters of the idea that communities can be automatically detected by analyzing the Web graph structure. In particular, they argue that the most important measure is "co-citation", that is, two pages that point to a third one, and that Web communities can be found by looking for dense bicliques (two sets of nodes, most of the first set pointing to most of the second). Those structures are most naturally found via forward and reverse navigation, lacking which the authors focus on ways to detect the bicliques using forward navigation only. Bicliques are also studied by Buehrer and Chellapilla [11] to detect communities and to compress Web graphs. As another example, Gibson et al. [12] propose algorithms to find communities that build on the HITS algorithm already mentioned. To measure the quality of the communities detected, there exist several metrics, such as their conductance, expansion, or internal density [13]. To efficiently compute these metrics, it is necessary to support single-link queries, so as to answer whether there is a link from one Web page to another, and also range queries, which give the number of edges that connect nodes from a community with nodes outside the community or within the community.

Range queries are also useful to regard Web graphs at different levels of granularity, for example determining whether there are links between two domains, sites, or subdirectories, or listing those links.

This brief recollection reveals, first, the importance of running complex graph algorithms on very large Web graphs, and second, the importance of supporting both forward and backward navigation on those Web graphs, as well as extra functionality such as single-link or range queries, as these provide a relevant tool for many Web graph analysis activities.

The problem of analyzing large Web graphs has been addressed in two ways. One is to resort to external memory and streaming graph algorithms. While various graph algorithms have external memory versions [14], many others are not disk-friendly. The second solution, which is the focus of this paper, is to use *compact graph representations*. These are memory-efficient data structures that represent large graphs and support fast navigation without decompressing them. Their aim is to support complex algorithms on large graphs within main memory, over much larger graphs than those that could be fit using classical graph representations.

In this paper we introduce the $k^2$-tree, a compact tree representation of the adjacency matrix of Web graphs that supports forward and backward navigation in a symmetric fashion. It also supports other more sophisticated functionality, such as querying for individual links, listing the links between ranges of nodes, and determining the existence of links between ranges of nodes. The $k^2$-tree exploits the clustering that arises in the adjacency matrix of Web graphs, by factoring out large empty areas. As such, it is a more general technique that can be used in other scenarios as well.

Our experimental results show that the $k^2$-tree is the most compact Web graph representation in the literature when both forward and backward navigation is to be supported, reaching 1–3 bits per link. It is also efficient to support forward and backward navigation, taking 2–8 μs per retrieved neighbor. There exist faster alternatives in the literature but they require significantly more space. Thus the $k^2$-tree extends the range of Web graph sizes that can be handled in main memory with an efficient support for various Web graph analysis tasks.

A conference version of this work appeared in SPIRE [15]. This paper extends that work by incorporating details and experiments of extended functionality of our proposal, that is, single-link and range queries, and adding various improvements to the method, which have given significantly better space/time tradeoffs (actually, the $k^2$-tree is up to date the most space-efficient representation thanks to these improvements, as the original version is not anymore competitive). It also includes an extensive experimental study, where we analyze the effects of the node ordering, and also compare the results achieved by our technique at compression and query time with other methods, including techniques that have appeared since the publication of the preliminary version, and that are much more competitive than those existing by then.

## 2. Related work

Most of the research in Web graph compression focuses on obtaining a compact representation that permits the efficient extraction of the successors of any Web page, that is, they emulate a classical adjacency list representation. The space requirements for these methods are commonly

measured in bits per edge (bpe), that is, the number of bits that are necessary to operate with the graph in main memory divided by the number of edges of the Web graph.

The most famous representative of this trend is surely Boldi and Vigna's *WebGraph Framework* [16]. The WebGraph compression method is indeed the most successful member of a family of approaches to compress Web graphs based on their statistical properties [17–22]. It allows fast extraction of the neighbors of a page while spending just a few bpe (about 1–6, depending on the desired navigation performance). Their representation explicitly exploits Web graph properties such as: (1) the locality of reference, and (2) the "copy property" (i.e., the set of neighbors of a page is usually very similar to that of some other page). They effectively exploit the power-law distribution of the gaps between successors. More recent variants of the WebGraph framework keep improving compression via better encodings and node orderings [23,24].

Claude and Navarro [25] showed that most of those properties are elegantly captured by applying Re-Pair compression [26] on the adjacency lists, and that reverse navigation can be achieved by representing the output of Re-Pair using sophisticated compact data structures for sequences [27]. While they cannot reach the best compression ratios of WebGraph, their representation is faster when both use the same space.

Asano et al. [28] go further in explicitly exploiting the particularities of Web graph adjacency matrices, by looking for known regularities (rows, columns, diagonal, L-shapes, etc.) and compressing them. While they achieve good compression results on small graphs (the approach does not seem to scale up easily), their navigation time is considerably high, as they need to uncompress full domains in order to find the neighbors of a single page.

Apostolico and Drovandi [29] presented a method that orders the nodes of the Web graph following a Breadth First Search (BFS) strategy, instead of using the popular lexicographic order. This retains the main properties of Web graphs outlined above, actually improving locality. They achieve little space (about 1–4 bpe), less than Asano et al. [28], while maintaining a retrieval time comparable to WebGraph variants. In addition, they introduce an efficient technique to determine whether two given nodes are connected, that is, to find out whether a page $p$ has a link to a page $q$ within about 60% of the time needed to extract the whole adjacency list of $p$.

Buehrer and Chellapilla [11] propose a Web graph compression technique based on finding bicliques on the directed graph, and representing them in a compact way. Their method, called "Virtual Node Miner (VNM)", is highly scalable and supports incremental updates. While its authors do propose some encodings, VNM is more properly described as a *graph preprocessing* method, as its output is still a graph that can be in turn compressed with other techniques. In particular, Hernández and Navarro [30] combined VNM with various techniques, including the $k^2$-trees we present in this paper (considering the conference version [15]). We will discuss this and other followups of our work in the Conclusions.

A recent proposal by Grabowski and Bieniecki [31] obtains very competitive time and space results. Their method, based on list merging, is called "LM" and consists in representing blocks of $h$ adjacency lists. It converts each block into two streams: in the first one they store compactly all the different nodes that appear at those lists, and in the other stream they store flags necessary to reconstruct the original lists, such that the merge of the lists can be reversed. They use $h$ bits in the second stream for each element of the first stream, setting to 1 the bit for all the adjacency lists that contain that node. Both streams are then compressed.

Finally, we point out that any representation that offers forward navigation functionality can be converted into one that offers bidirectional navigation by simply storing the (compressed) graph and the (compressed) transposed graph. Boldi et al. [24] suggest a more clever approach: Separate the bidirectional subgraph $G_b$ (formed by the edges whose reciprocal is also in the graph $G$) from $G$, leaving $G' = G - G_b$. Then represent $G_b$, $G'$ and the transpose of $G'$. Then successors are queried in $G_b$ and $G'$, and predecessors in $G_b$ and the transpose of $G'$.

To conclude our revision, a note on the identity of Web graph nodes is in order. It is customary in compressed Web graph representations to assume that page identifiers are integers, which correspond to their position in an array of URLs. The space for that array is not accounted for, as it is assumed to be independent of the Web graph compression method. Moreover, it is assumed that URLs are alphabetically sorted, which naturally puts together the pages of the same domains, and thus locality of reference translates into closeness of page identifiers. It has been found, however, that other node orderings can be more convenient than lexicographic ordering [29,23,24]. We note that it is not totally fair to assume that those methods require the same space to represent the URLs as the lexicographic ordering, as the latter can be taken advantage of by prefix omission compression methods [32]. However, most Web graph analysis algorithms, including PageRank or HITS, are independent of the text of the URLs, and therefore one can maintain the URLs on disk and retrieve them only at the very end of the analysis algorithm. For these reasons, and following the literature, we will ignore the space to store the URLs of nodes when measuring the space in all the compression methods, and will assume that nodes are integers that can be reordered.

## 3. Our proposal

The adjacency matrix of a Web graph of $n$ pages is a square matrix $\{a_{ij}\}$ of $n \times n$ cells, where each row and each column represents a Web page. Cell $a_{ij}$ is 1 if there is a hyperlink in page $i$ towards page $j$, and 0 otherwise.

As there are a few tens of links per Web page, this matrix is extremely sparse. Assuming that we use a node reordering that favors the locality of reference, many 1s are placed around the main diagonal (that is, page $i$ has many pointers to pages nearby $i$). Due to the copy property, similar rows are common in the matrix. Finally, due to skewness of distribution, some rows and columns have many 1s, but most have very few.

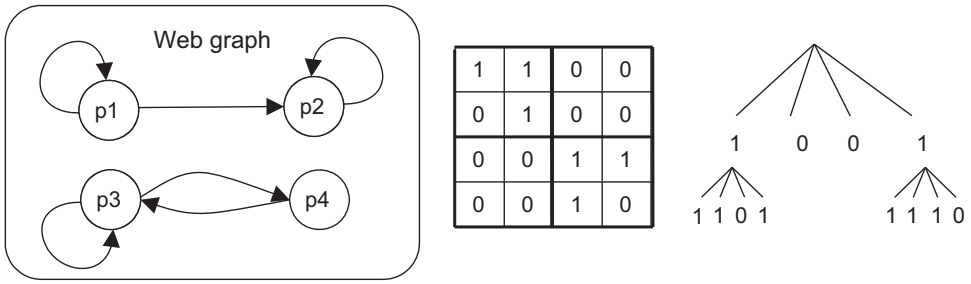


**Fig. 1.** Representation of a Web graph (left) by its adjacency matrix (center), and the corresponding $k^2$-tree (right).

We propose a compact representation of the adjacency matrix that exploits its sparseness and clustering properties. The representation is designed to compress large matrix areas with all 0s into very few bits.

### 3.1. Conceptual description

We represent the adjacency matrix by a $k^2$-ary tree, which we call a *$k^2$-tree*, of height $h = \lceil \log_k n \rceil$. Each node contains a single bit of data: 1 for the internal nodes and 0 for the leaves, except for the last level, where all are leaves and represent bit values of the matrix. The first level (numbered 0) corresponds to the root; its $k^2$ children are represented at level 1. Each child is a node and therefore it has a value 0 or 1. All the internal nodes (i.e., with value 1) have exactly $k^2$ children, whereas leaves (which have value 0 or lie at the last tree level) have no children.

The $k^2$-tree represents the adjacency matrix as follows. Assume for a moment that $n$ is a power of $k$; we will soon remove this assumption. Conceptually, we start dividing the adjacency matrix following a MX-Quadtree strategy [33, Section 1.4.2.1], into $k^2$ submatrices of the same size, that is, $k$ rows and $k$ columns of submatrices of size $n^2/k^2$. Each of the resulting $k^2$ submatrices will be a child of the root node and its value will be 1 iff there is at least one 1 in the cells of the submatrix. A 0 child means that the submatrix has all 0s and therefore the tree decomposition ends there; thus 0s are leaves in our tree. The children of a node are ordered in the tree starting with the submatrices in the first (top) row, from left to right, then the submatrices in the second row from left to right, and so on.

Once the level 1 of the tree, which contains the children of the root node, has been built, the tree is expanded recursively from each child with value 1. The expansion stops when we reach submatrices full of 0s, or when we reach a $k \times k$ submatrix of the original adjacency matrix, that is, we reach the last level of the tree. In this last level, the bits of the nodes correspond to the adjacency matrix cell values, following the node ordering we have previously defined. The height of the tree is $h = \lceil \log_{k^2} n^2 \rceil = \lceil \log_k n \rceil$.

Fig. 1 (left) illustrates a small Web graph consisting of four Web pages, p1 to p4. Its $4 \times 4$ adjacency matrix is shown in the center of the figure. On the right we illustrate the $2^2$-tree built for this example. Its height is $h = \lceil \log_2 4 \rceil = 2$, where level 1 corresponds to the children of the root node and level 2 contains the original cell values of the $2 \times 2$ adjacency submatrices that are not all 0s. Following the ordering previously defined, those submatrices containing at least one 1 are the first (top-left) and the fourth (bottom-right) of the adjacency matrix.

If $n$ is not a power of $k$, we conceptually extend our matrix to the right and to the bottom with 0s, making it of width $n' = k^h = k^{\lceil \log_k n \rceil}$, that is, we round $n$ up to the next power of $k$. This does not cause a significant overhead because our technique is efficient to handle large areas of 0s.

Note that, since the height of the tree is $h = \lceil \log_k n \rceil$, a larger $k$ induces a shorter tree, with fewer levels but with more children per internal node. Fig. 2 shows an example of the adjacency matrix of a small Web graph (the first $11 \times 11$ submatrix of graph CNR[1]), and how it is expanded to an $n' \times n'$ matrix for $n'$ a power of $k=2$ and of $k=4$. The figure also shows the $k^2$-trees corresponding to those $k$ values. Note that most empty cells in the original adjacency matrix are not explicitly represented, but rather grouped in higher-level leaves of the $k^2$-tree.

### 3.2. Data structures

As mentioned, the $k^2$-tree can be seen as a compact representation of an MX-Quadtree [33, Section 1.4.2.1], which is a Quadtree variant designed for sparse matrices. This usage of Quadtrees is not new, e.g. [34]. What is novel in the $k^2$-tree is its compact representation. As we show later, this representation which yields extremely good compression ratios and efficient navigation on Web graphs, due to the simplicity and compactness of the data structures we use to represent the $k$-ary tree, and the fact that they match well with the sparseness and clustering of the adjacency matrix of Web graphs.

Our data structure is essentially a compact tree of $N$ nodes. There exist several such representations for general trees [35–39], which asymptotically approach the information-theoretic minimum of $2N + o(N)$ bits. In our case, where there are only arities $k^2$ and 0, the information-theoretic minimum of $N + o(N)$ bits can be achieved with a so-called "ultra-succinct" representation [40] for general trees.

Our representation is much simpler and still uses $N + o(N)$ bits. We use a variant of the so-called LOUDS (level-ordered unary degree sequence) tree representation [35] tailored to the case where there are only arities $k^2$ and 0 (we note that just applying plain LOUDS on our tree would require $2N + o(N)$ bits). Our representation gives in constant time the

[1] From the Laboratory for Web Algorithmics, http://law.dsi.unimi.it.

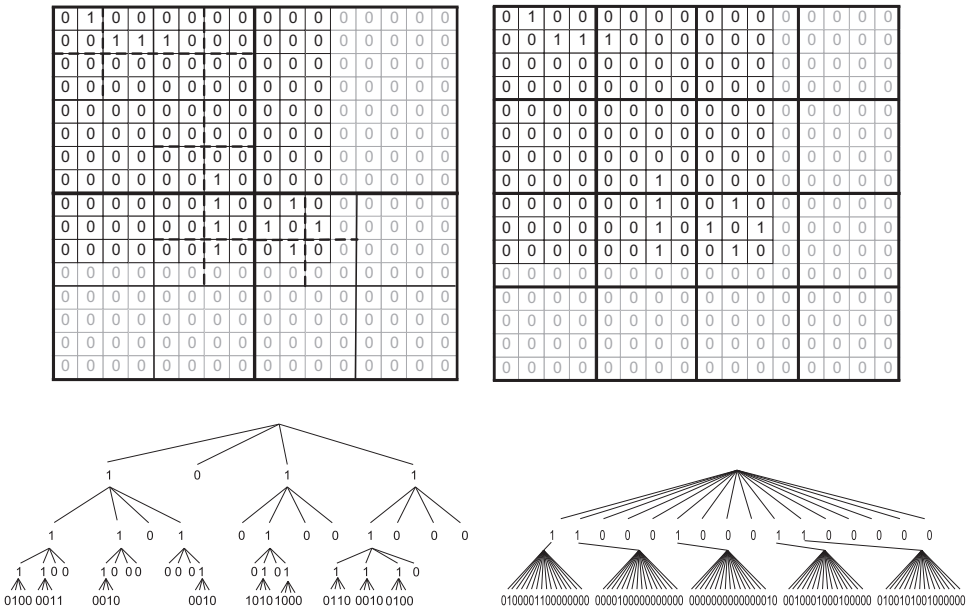


**Fig. 2.** Expansion and subdivision of the adjacency matrix (top) and resulting tree (bottom) for $k=2$ (left) and for $k=4$ (right).

$i$-th child of any node, which is the only query we need (it also gives the parent of a node, but we will not need that). We inherit the simplicity and practical efficiency of the basic LOUDS representation, which was found in recent experiments to be the most convenient one when its limited support of operations is sufficient [41].

### 3.2.1. Representation

We represent the whole adjacency matrix via the $k^2$-tree using two bit arrays:

- $T$ (*tree*): stores all the bits of the $k^2$-tree except those at depth $h$. The bits are placed following a levelwise traversal: first the $k^2$ binary values of the children of the root node, then the values of the second level, and so on.
- $L$ (*leaves*): stores the last level of the tree. Thus it represents the value of (some) original cells of the adjacency matrix.

A node in the $k^2$-tree is identified with the position $x$ of the corresponding bit in $T\|L$, where "$\|$" is the sequence concatenation operator. For example, to represent the $2^2$-tree of Fig. 2, arrays $T$ and $L$ are

$T = 1011\ 1101\ 0100\ 1000\ 1100\ 1000\ 0001\ 0101\ 1110,$

$L = 0100\ 0011\ 0010\ 0010\ 1010\ 1000\ 0110\ 0010\ 0100.$

In $T$ each bit represents a node. The first four bits represent nodes 0, 1, 2 and 3, which are the children of the root. The following four bits represent the children of node 0. There are no children for node 1 because it is a 0, then the children of node 2 start at position 8 and those of node 3 start at position 12. The bit in position 4, the fifth bit of $T$, represents the first child of node 0, and so on.

This representation permits fast navigation to the $i$-th child of node $x$, $child_i(x)$, for any $0 \le i < k^2$. Let $x$ be a position in $T$ (the first position being 0) such that $T[x] = 1$. Then $child_i(x)$ is at position $rank(T,x) \cdot k^2 + i$ of $T\|L$, where $rank(T,x)$ is the number of 1s in $T[0,x]$. To see this, consider the following. Two different traversals of the $k^2$-tree output the same $T\|L$ sequence. In traversal $A$, we traverse the $k^2$-tree levelwise starting at level 1 (i.e., skipping the root) and append the bits of the tree nodes visited. In traversal $B$, we traverse the $k^2$-tree levelwise starting at level 0 and append the $k^2$ bits of the children of the internal nodes visited. When traversal $A$ visits the node corresponding to $x$, we are filling $T\|L$ at position $x$. When traversal $B$ visits the same node, we are filling $T\|L$ at the position of the first child of $x$. To obtain this position we must add $k^2$ per 1 visited in traversal $A$ before position $x$. Since the 1 of the root is not represented, this is $(1+rank\ (T,x-1))k^2 = rank(T,x) \cdot k^2$ (the equality holds because $T[x] = 1$). Thus the $i$-th child of $x$ is at position $rank(T,x) \cdot k^2 + i$.

Note that internal nodes can only belong to the $T$ part. To carry out operation $child_i(x)$ efficiently, we need to support $rank(T,x)$ queries. This operation can be supported in constant time and fast in practice using sublinear space on top of the bit sequence [35,42]. In practice we use an implementation that uses 5% of extra space on top of the bit sequence and provides fast queries (if more space and less time is desired, one could replace the bitmap by another that uses 37.5% extra space and is much faster [43]; other more recent bitmap implementations are also competitive [44]). Bitmap $L$, instead, does not need any additional structure.

### 3.2.2. Space analysis

Assume the graph has $n$ pages and $m$ links. Each link is a 1 in the matrix, and in the worst case it induces the storage of one distinct node per level, for a total of $h = \lceil \log_{k^2} n^2 \rceil$ nodes. Each such (internal) node costs $k^2$

bits, for a total of $k^2 m\lceil \log_{k^2} n^2 \rceil$ bits. However, in the upper levels, not all the nodes in the path to each leaf can be different. In the worst case, all the nodes exist up to level $\lfloor \log_{k^2} m \rfloor$ (only since that level there can be $m$ different internal nodes at the same level). From that level, the worst case is that each of the $m$ paths to the leaves is unique. Thus, in the worst case, the total space in bits is

$$\left(\sum_{\ell=1}^{\lfloor \log_{k^2} m \rfloor} k^{2\ell}\right) + k^2 m(\lceil \log_{k^2} n^2 \rceil - \lfloor \log_{k^2} m \rfloor)$$
$$= k^2 m\left(\log_{k^2} \frac{n^2}{m} + O(1)\right).$$

This space formula is also valid when $n$ is not a power of $k$. The formula suggests that, at least in a worst-case analysis, a smaller $k$ yields less space occupancy. For $k=2$ the space is $4m(\log_4 n^2/m + O(1)) = 2m \log_2 n^2/m + O(m)$ bits, which is asymptotically twice the information-theoretic minimum necessary to represent all the matrices of $n \times n$ with $m$1s. In the experimental section we see that, on Web graphs, the space is much better than in the worst case, as Web graphs are far from uniformly distributed.

### 3.3. Construction

We give in this section various alternatives to build the representation $(T,L)$ of the $k^2$-tree given a Web graph. First we show in Section 3.3.1 that, if we start from the adjacency matrix, the time is $O(n^2(1/k+1/w))$, where $w$ is the length in bits of the computer word and $n$ is the number of graph nodes. The output $(T,L)$ is built within just $s = t+\ell$ bits of space, where $t = |T|$ and $\ell = |L|$, that is, the same space of the final output. Then, since representing the adjacency matrix is not realistic, we show in Section 3.3.2 how to simulate the same algorithm from the adjacency lists of the graph. This requires $O(n)$ words of extra construction space, and improves the time complexity to $O(n^2/k^2 + m + s/w)$, where $m$ is the number of graph edges. In Section 3.3.3 we improve the time to the output-sensitive $O(t/k^2 + m + s/w) = O(s)$, yet the construction space raises to $O(t)$ words. Subsequently, in Section 3.3.4, we reduce the construction space to just $s + o(s)$ bits, slightly raising the construction time to $O(s(1 + \log s/(k^2 \log\log s)))$. In Section 3.3.5 we obtain a practical result where $O(t)$ extra words and $O(m \log_k n)$ time is needed, but the output is built in order and the process has high locality of reference. Finally, in Section 3.3.6 we show how to build the structure with the adjacency list in secondary memory.

#### 3.3.1. From the adjacency matrix

Assume for a moment that our input is the $n \times n$ adjacency matrix. Construction of our tree is easily carried out bottom-up in linear time and optimal space (that is, using the same space as the final tree).

Our procedure builds the tree recursively. It consists in a depth-first traversal of the tree that outputs a bit array $T_\ell$ for each level $\ell$ of the tree. If we are at the last level $\ell = h$, we read the $k^2$ corresponding matrix cells. If all are 0, we return 0 and we do not output any bit string, as none of those 0s will be explicitly represented; otherwise we output the $k^2$ bits and return 1. If we are not at the last level, we make the $k^2$ recursive calls for the children. If all return 0, we return 0, otherwise we output the $k^2$ answers of the children and return 1.

The output for each call is stored separately for each level, so that the $k^2$ bits that are output at each level are appended to the corresponding bit array $T_\ell$. As we fill the values of each level left-to-right, the final $T$ is obtained by concatenating all levels but the last one, which is indeed $L$.

Algorithm 1 shows the construction process. It is invoked as **Build** $(n' = k^h, 1, 0, 0)$, where the first parameter is the (extended) submatrix width, the second is the current level, the third is the row offset of the current submatrix, and the fourth is its column offset. After running it we have $T = T_1 \| T_2 \| \ldots \| T_{h-1}$ and $L = T_h$.

**Algorithm 1. Build**$(n, \ell, p, q)$, builds the $k^2$-tree representation from the matrix $a_{ij}$.

```
C ← empty sequence
for i←0...k−1 do
  for j←0...k−1 do
    if ℓ = ⌈log_k n⌉ then/* last level */
      C←C‖a_{p+i,q+j}
    else/* internal node */
      C←C‖Build(n/k, ℓ+1, p+i·(n/k), q+j·(n/k))
    end
  end
end
if C = 0^{k^2} then return 0;
T_ℓ←T_ℓ‖C
return 1
```

The total time is clearly linear in the number of cells in the matrix, that is, $O(n^2)$. By accessing up to $w$ consecutive (say, horizontal) bits of the matrix in one operation, the time can be reduced to $O((n^2/k^2)k(1+k/w))$ to scan the matrix plus $O(n^2/k^2)$ for the recursive invocations, for a total complexity of $O(n^2(1/k+1/w))$.

#### 3.3.2. From the adjacency list

Representing the complete matrix for the construction process is not realistic on Web graphs, because the matrix is too sparse. We use instead the adjacency lists representation of the matrix, that is, for each Web page $p$ we have the list of Web pages $q$ such that $p$ has a link pointing to $q$. By using the adjacency lists we can still achieve the same time by setting up $n$ cursors, one per row, so that each time we have to access $a_{pq}$ we compare the current cursor of row $p$ with value $q$. If they are equal, we know $a_{pq}=1$ and move the cursor to the next node of the list of row $p$. Otherwise we know $a_{pq}=0$. This works because all of our queries to each matrix row $p$ are increasing in column value.

In this case we pay $O(n^2/k^2)$ in recursive invocations to reach each leaf, plus $O(t/w)$ to write down bitmap $T$, plus $O(1+k^2/w)$ to initialize each chunk of $k^2$ bits in $L$, plus $O(1)$ time to go over each entry of the adjacency list and write the corresponding 1 in the chunk of $L$.

This gives a total of $O(n^2/k^2+t/w+\ell(1/k^2+1/w)+m)=O(n^2/k^2+m+s/w)$. Note that we have improved the complexity as a function of $n^2$, and are more sensitive to the output size than in Section 3.3.1.

#### 3.3.3. Output-sensitive time

When the input consists of the adjacency list representation of the graph, we can achieve time proportional to the bit size of its compressed representation, $s$, instead of the bit size of the input matrix, $n^2$ (note $s \le n^2 \cdot k^2/(k^2-1)$, and it is much less when compression works well). For this sake we insert the 1s one by one into an initially empty tree, building the necessary part of the path from the root to the corresponding leaf (we create the $k^2$ children of each new internal node we build). After the tree is built we traverse it levelwise to build the final representation, or recursively to output the bits to different sequences, one per level, as in Section 3.3.1.

This builds the $k^2$-tree in time in $O(t/k^2)$, by initializing the array of children in constant time [45, Section III.8.1]. We then need $O(t/w)$ time to write it down to bitmap $T$. As for the leaves, we preserve the $O(m+\ell/w)$ time complexity. The sum gives $O(t(1/k^2+1/w)+m+\ell/w)=O(t/k^2+m+s/w)=O(s)$, which is fully independent of the matrix size $n^2$. However, the solution requires $O(t)$ words (not bits) of extra construction space, as we first build a classical tree, and this is considerable.

#### 3.3.4. Optimal space again

We can reduce the construction space to $s+o(s)$ bits by building the compact tree directly, via insertion of internal nodes (in $T$) and leaves (in $T$ and $L$). For this sake we use a dynamic bitmap representation for $T$ and $L$ [46,47]. This structure supports insertions and deletions of bits, as well as the *rank* operation, in time $O(\log s/\log\log s)$. Instead of traversing a classical tree and inserting the nodes on it as in Section 3.3.2, we traverse (the current) $T$ and $L$ with the formula previously given for computing the $i$-th child of a node. When we find a 0 in the path to a leaf that must be inserted in $L$, we turn it into a 1 and insert its $k^2$ 0s at its children positions, and continue descending to the right child (which will be recursively expanded until we reach $L$).

The time is now slowed down by the compact dynamic representation of $T$ and $L$. As the bitmap representations permit inserting $k^2$ contiguous bits in time $O((1+k^2/\log s)\log s/\log\log s)$, the total time is $O(t/k^2)$ bit flips, plus $O(s/k^2)$ insertions of chunks of $k^2$ bits, plus $O(m)$ to traverse the adjacency lists, which adds up to $O(m+s(1+\log(s)/k^2)/\log\log s)=O(s(1+\log s/(k^2\log\log s)))$.

#### 3.3.5. Inplace construction

We now show how to build $T$ and $L$ in left-to-right order (which allows, for example, storing them to disk immediately), within the same $O(t)$ extra space of Section 3.3.3.

We will maintain a priority queue with subproblems yet to solve. Each subproblem has associated a square matrix range $(x_1,x_2)\times(y_1,y_2)$ and the corresponding interval $[a, b]$ in the array of graph edges. Initially we enqueue the whole matrix $(0,n-1)\times(0,n-1)$ and the interval $[1, m]$. The general procedure is to process the next subproblem in the queue, append bits to $T$ and $L$, and possibly enqueue new subproblems, until the queue becomes empty.

A subproblem with matrix range $(x_1,x_2)\times(y_1,y_2)$, where $x_2-x_1+1=y_2-y_1+1=S$, where $S$ is a multiple of $k$, and interval $[a, b]$ is processed as follows. We subdivide the range into $k\times k$ subintervals and assign to each edge $(x,y)$ of the interval $[a, b]$ a value $i$, $0\le i<k$ with $i=\lfloor(x-x_1)/(S/k)\rfloor\cdot k+\lfloor(y-y_1)/(S/k)\rfloor$. Then we sort the interval using counting sort, delimiting $k^2$ subintervals $[a_i, b_i]$ of $[a, b]$.

Now, for each $0\le i<k^2$, we append to the bitmap $T$ a bit 1 if $a_i<b_i$, and a 0 otherwise. For each 1 appended, we also enqueue the corresponding submatrix range and the interval $[a_i, b_i]$.

When the matrix range extracted is of size $k\times k$, the corresponding interval $[a, b]$ is appended to $L$ in the form of $k^2$ bits that represents the small submatrix.

The time is $O(t+\ell(1/k^2+1/w)+m\log_k n)=O(m\log_k n)$. The construction is inplace, except for the queue, which holds $O(t)$ records in the worst case.

#### 3.3.6. With the input on disk

The algorithm of Section 3.3.4 is easily adapted to the case where the input graph does not fit in main memory (note we assume that the compressed $k^2$-tree does fit in main memory). Since the adjacency lists are traversed sequentially, we obtain $O(m/B)$ I/O time by reading them in buffered form and building the tree in compressed form in main memory. The variants of Sections 3.3.3 and 3.3.2 can also be easily implemented I/O-optimally if there is sufficient main memory to hold the extra construction space (in Section 3.3.2 we need the memory to hold $n$ buffers of $B$ words, as the lists are not read in synchronization).

Finally, if we have only $O(k^2B)$ integers of main memory available, we can use the algorithm of Section 3.3.5. The counting sort needs only $k^2$ integers in main memory, we use $k^2$ tree buffers for the sorting step, and the queue and the output $T$ and $L$ can be fully buffered. The I/O cost becomes $O((m/B)\log_k n)$. It is also possible to construct the $k^2$-tree in this way using $O(B)$ integers of main memory, but with $O((m/B)\log n)$ I/O time cost.

## 4. Navigation

In this section we detail the algorithms to navigate the $k^2$-tree. More specifically, Section 4.1 describes single-link queries, which determine whether or not one Web page has a link to another specific Web page. This can be seen as checking the value of a specific cell in the adjacency matrix. Section 4.2 describes the retrieval of the successors and the predecessors lists of a Web page, that is, which Web pages are pointed by or point to a specific Web page. Finally, Section 4.3 details how to proceed when we want to retrieve all the links among two ranges of pages, such as determining all the links between two different domains.

### 4.1. Checking individual links

In order to determine whether a given page $p$ points to a given page $q$, most compressed (and even some classical) graph representations have no choice but to extract all the successors of $p$ (or a significant part of them) and see whether $q$ is in the set. We can answer such query in $O(\log_k n)$ time, by descending to exactly one child at each level of the tree, such that we can determine if the cell $a_{pq}$ of the adjacency matrix is 1 (page $p$ points to page $q$) or 0 (page $p$ does not point to page $q$).

We start at the root node and descend recursively to the child node that represents the submatrix containing the cell $a_{pq}$ of the adjacency matrix. We give now the formula that maps the relative row and column numbers of the cell to the child number at each level and the formula that computes the new relative cell row and column numbers for the child submatrix.

Recall that $h = \lceil \log_k n \rceil$ is the height of the tree. Then the nodes at level $\ell$ represent square submatrices of size $k^{h-\ell}$, and these are divided into $k^2$ submatrices of size $k^{h-\ell-1}$. For instance, the root at level $\ell = 0$ represents the whole square matrix of width $k^h = n'$.

Let us call $p_\ell$ the relative row position of Web page $p$ at level $\ell$, and $q_\ell$ the relative column position of Web page $q$ at level $\ell$. Cell $(p_\ell, q_\ell)$ at a matrix of level $\ell$ belongs to the submatrix at row $\lfloor p_\ell / k^{h-\ell-1} \rfloor$ and column $\lfloor q_\ell / k^{h-\ell-1} \rfloor$. This corresponds to the child number $k \cdot \lfloor p/k^{h-\ell-1} \rfloor + \lfloor q/k^{h-\ell-1} \rfloor$ of the node that represents the matrix at level $\ell$. The relative row position for Web page $p$ in this child node is $p_{\ell+1} = p_\ell \bmod k^{h-\ell-1}$. The relative column position for Web page $q$ is $q_{\ell+1} = q_\ell \bmod k^{h-\ell-1}$.

Hence, starting from the root node at level $\ell = 0$, where $p_0 = p, q_0 = q$, at each level $\ell$ we descend to child $k \cdot \lfloor p/k^{h-\ell-1} \rfloor + \lfloor q/k^{h-\ell-1} \rfloor$, if it is not a zero, and compute the relative position of cell $(p,q)$ in the submatrix. If we reach the last level and find a 1 at cell $(p,q)$, then there is a link, else there is not.

For example, assume we want to know if page 2 points to page 3, that is, we want to know if there is a 1 at cell $a_{2,3}$ of the adjacency matrix of Fig. 1. We start at the root of the $2^2$-tree and descend to the second child of the root node, since the cell $a_{2,3}$ belongs to the second submatrix of the adjacency matrix. Since we find a 0, then page 2 does not point to page 3. If we want to know if page 3 has a link to itself, then we start from the root node and go down through the fourth child of the node, which represents the submatrix where the cell $a_{3,3}$ is located. There is a 1 there, indicating that this submatrix has at least one 1. Since the cell $a_{3,3}$ of the original adjacency matrix is the first cell of this submatrix, then we check the bit value contained in the first child of the node. It contains a 1, hence page 3 has a link pointing to itself.

Algorithm 2 checks whether a Web page $p$ points to a page $q$. It is invoked as **CheckLink**$(n', p, q, -1)$, where the parameters are: current submatrix size, row of interest in the current submatrix, column of interest in the current submatrix, and the position in $T \| L$ of the node to process (the initial $-1$ is an artifact because our trees do not represent the root node). Values $T$, $L$, and $k$ are global. It is assumed that $rank(T, -1) = 0$.

**Algorithm 2. CheckLink**$(n, p, q, z)$ returns 1 iff node $p$ points to node $q$ and 0 otherwise.

```
if z ≥ |T| then /* leaf    */
  return L[z−|T|]
else /* internal node    */
  if z = −1 or T[z] = 1 then
    y←rank(T, z) · k^2
    y←y + ⌊p/(n/k)⌋ · k + ⌊q/(n/k)⌋
    CheckLink(n/k, p mod (n/k), q mod (n/k), y)
  else
    return 0
  end
end
```

#### 4.1.1. Time analysis

The worst-case navigation time to check if a Web page $p$ points to another Web page $q$ is $O(\log_k n)$, since a full traversal from the root node to a leaf node is required for any pair of connected Web pages. The time can be even lower for nonexistent links.

### 4.2. Successors and predecessors

To find the successors(predecessors) of a page $p(q)$ we need to locate which cells in row $a_{p*}$ (column $a_{*q}$) of the adjacency matrix have a 1. Again, these are obtained by a top-down tree traversal, but instead of choosing just one child node, as for single-link queries, the algorithm must choose $k$ out of the $k^2$ children of a node.

As before, we give the formula that maps global row numbers to the children numbers at each level. Being $p_\ell$ the relative row position of interest at level $\ell$, row $p_\ell$ of the submatrix of level $\ell$ corresponds to children number $k \cdot \lfloor p_\ell / k^{h-\ell-1} \rfloor + j$, for $0 \le j < k$. Similarly, column $q$ corresponds to in level $\ell$ to children number $j \cdot k + \lfloor q_\ell / k^{h-\ell-1} \rfloor$, for $0 \le j < k$.

For instance, assume that we want to obtain the successors of Web page 10 of the Web graph represented in Fig. 2 (left). This Web page is represented at row $p_0 = 10$ at level $\ell = 0$, since the whole adjacency matrix is considered at this level. When $\ell = 1$ the relative position of Web page 10 inside the two submatrices of size $8 \times 8$ of the bottom of the matrix is $p_1 = 10 \bmod 8 = 2$. The relative row inside the submatrices of size $4 \times 4$ that overlap with row 10 at level $\ell = 2$ is $p_2 = 2 \bmod 4 = 2$, and finally, the relative position of row 10 inside the submatrices of size $2 \times 2$ that overlap with the row at level $\ell = 3$ is $p_3 = 2 \bmod 2 = 0$.

Algorithms 3 and 4 extract the successors and predecessors lists. The one for successors is invoked as **Successors**$(n', p, 0, -1)$, and the one for predecessors is invoked as **Predecessors**$(n', q, 0, -1)$, with the same meanings and assumptions as before. For the successors algorithm the second parameter is the row of interest and the third parameter is the column offset of the current submatrix. For the predecessors algorithm the second parameter is the column of interest and the third parameter is the row offset of the current submatrix

We note that the algorithms output the neighbors in order. Although we present them in recursive fashion for clarity, an iterative variant using a queue of nodes to process turns out to be slightly more efficient in practice, and what we use in our implementation.

**Algorithm 3. Successors**$(n,p,q,z)$ returns successors of page $p$.

```
if z ≥ |T| then /* last level */
  if L[z−|T|] = 1 then output q;
else /* internal node */
  if z = −1 or T[z] = 1 then
    y←rank(T,z)·k² + k·⌊p/(n/k)⌋
    for j←0…k−1 do
      Successors(n/k, p mod(n/k), q+(n/k)·j, y+j)
    end
  end
end
```

**Algorithm 4. Predecessors**$(n,q,p,z)$ returns predecessors of page $q$.

```
if z ≥ |T| then /* last level */
  if L[z−|T|] = 1 then output p;
else /* internal node */
  if z = −1 or T[z] = 1 then
    y←rank(T,z)·k² + ⌊q/(n/k)⌋
    for j←0…k−1 do
      Predecessors(n/k, q mod(n/k), p+(n/k)·j, y+j·k)
    end
  end
end
```

#### 4.2.1. Time analysis

The navigation time to retrieve a successors or predecessors list has no worst-case guarantees better than $k^h = O(n)$, as a row $p-1$ full of 1s followed by $p$ full of 0s could force a **Successors** query on $p$ to go until the leaves across all the row, to return nothing.

However, this is unlikely. Assume the $m$ 1s are uniformly distributed in the matrix. Then the probability that a given 1 is inside a submatrix of size $(n/k^\ell)\times(n/k^\ell)$ is $1/k^{2\ell}$. Thus, the probability of entering the children of such submatrix is (brutally) upper bounded by $m/k^{2\ell}$. We are interested in $k^\ell$ submatrices at each level of the tree, and therefore the total work is on an average upper bounded by $m\cdot\sum_{\ell=0}^{h-1}k^\ell/k^{2\ell}=O(m)$. This can be refined because there are not $m$ different submatrices in the first levels of the tree. Assume we enter all the $O(k^t)$ matrices of interest up to level $t=\lfloor\log_{k^2} m\rfloor$, and from then on the sum above applies. This is

$$O\left(k^t+m\cdot\sum_{\ell=t+1}^{h-1}\frac{k^\ell}{k^{2\ell}}\right)=O\left(k^t+\frac{m}{k^t}\right)=O(\sqrt{m})$$

time. This is not the ideal $O(m/n)$ (average output size), but better than $O(n)$ or $O(m)$.

Again, if the matrix is clustered, the average performance is indeed better than under a uniform distribution: whenever a cell close to row $p$ forces us to traverse the tree down to it, it is likely that there is a useful cell at row $p$ as well. This can be observed in the experimental evaluation described in Section 6.

### 4.3. Range queries

A more general operation is to find the successors of page $p$ that are within a *range* of pages $[q_1,q_2]$ (similarly, the predecessors of $q$ that are within a range $[p_1,p_2]$). This is interesting, for example, to find out whether $p$ points to a domain, or is pointed from a domain, in case we sort URLs in lexicographical order.[2] The algorithm is similar to **Successors** and **Predecessors** in Section 4.2, except that we do not enter all the children $0\le j<k$ of a row (or column), but only those $0\le j<k$ which also fulfil $\lfloor q_1/k^{h-\ell-1}\rfloor\le j\le\lfloor q_2/k^{h-\ell-1}\rfloor$ (similarly for $p_1$ to $p_2$).

An even more general operation is to find all the links from a range of pages $[p_1,p_2]$ to another $[q_1,q_2]$. This is useful, for example, to extract all the links between two domains, or to obtain significant measures for community detection. The algorithm to solve this query indeed generalizes all of the others we have seen: extract successors of $p$ ($p_1=p_2=p$, $q_1=0$, $q_2=n-1$), extract predecessors of $q$ ($q_1=q_2=q$, $p_1=0$, $p_2=n-1$), find whether a link from $p$ to $q$ exists ($p_1=p_2=p$, $q_1=q_2=q$), find the successors of $p$ within range $[q_1,q_2]$($p_1=p_2=p$), and find the predecessors of $q$ within range $[p_1,p_2]$ ($q_1=q_2=q$). Fig. 5 gives the algorithm. It is invoked as **Range**$(n',p_1,p_2,q_1,q_2,0,0,-1)$.

**Algorithm 5. Range** $(n,p_1,p_2,q_1,q_2,d_p,d_q,z)$ lists all links from nodes $[p_1,p_2]$ to $[q_1,q_2]$.

```
if z ≥ |T| then /* leaf */
  if L[z−|T|] = 1 then output (d_p, d_q);
else /* internal node */
  if z = −1 or T[z] = 1 then
    y←rank(T,z)·k²
    for i←⌊p1/(n/k)⌋…⌊p2/(n/k)⌋ do
      if i←⌊p1/(n/k)⌋ then p'1←p1 mod(n/k) ;
      else p'1←0;
      if i←⌊p2/(n/k)⌋then p'2←p2 mod(n/k);
      else p'2←(n/k)−1;
      for j←⌊q1/(n/k)⌋…⌊q2/(n/k)⌋ do
        if j = ⌊q1/(n/k)⌋ then q'1←q1 mod(n/k);
        else q'1←0;
        if j = ⌊q2/(n/k)⌋ then q'2←q2 mod(n/k);
        else q'2←(n/k)−1;
        Range (n/k, p'1, p'2, q'1, q'2, d_p+(n/k)·i, d_q+(n/k)·
        j, y+k·i+j)
      end
    end
  end
end
```

Moreover, we can check if there exists a link from a range of pages $[p_1,p_2]$ to another range $[q_1,q_2]$ in a more efficient way than finding all the links in that range. If we just want to know whether there is a link in the range, complete top-down traversals of the tree can be avoided if we reach an internal node that represents a submatrix of the original adjacency matrix that is entirely contained in the range sought and it is represented with a 1 bit in the $k^2$-tree. This means that there is at least one 1 inside that

[2] Range queries can also be useful over other reorderings. For instance, when using "Layered Label Propagation" (LLP) [24], which is described and used in Section 6, range queries can be used to support queries over communities.

submatrix, and thus there is a link in the range of the query. This operation is performed analogously to the range query described in Algorithm 5, except that it checks if the current submatrix is completely contained in the range sought. In such a case it finishes by returning 1, avoiding the traversal to the leaf level and any other extra top-down traversal of the $k^2$-tree. Algorithm 6 details this process. It is invoked as **LinkInRange**$(n', p_1, p_2, q_1, q_2, -1)$.

**Algorithm 6. LinkInRange**$(n, p_1, p_2, q_1, q_2, z)$ returns 1 iff there is a link from nodes $[p_1, p_2]$ to $[q_1, q_2]$.

```
if z ≥ |T| then /* leaf */
  return L[z−|T|]
else /* internal node */
  if z = −1 or T[z] = 1 then
    if p1 = 0 and q1 = 0 and p2 = n/k−1 and q2 = n/k−1 then
      return 1
    end
    y←rank(T, z) · k^2
    for i←⌊p1/(n/k)⌋ ... ⌊p2/(n/k)⌋ do
      if i = ⌊p1/(n/k)⌋ then p'1←p1 mod(n/k);
      else p'1←0;
      if i = ⌊p2/(n/k)⌋ then p'2←p2 mod(n/k);
      else p'2←(n/k)−1;
      for j←⌊q1/(n/k)⌋ ... ⌊q2/(n/k)⌋ do
        if j = ⌊q1/(n/k)⌋ then q'1←q1 mod(n/k);
        else q'1←0;
        if j = ⌊q2/(n/k)⌋ then q'2←q2 mod(n/k);
        else q'2←(n/k)−1;
        ifLinkInRange(n/k, p'1, p'2, q'1, q'2, y+k · i+j) then
          return 1
        end
      end
    end
  end
  return 0
end
```

We can speed up this method, at least in theory, as follows. Rows $r_1 = \lfloor p_1/(n/k) \rfloor$ and $r_2 = \lfloor p_2/(n/k) \rfloor$ and columns $c_1 = \lfloor q_1/(n/k) \rfloor$ and $c_2 = \lfloor q_2/(n/k) \rfloor$ are dealt with individually, whereas the sub-area of the $k^2$ children $[r_1+1, r_2-1] \times [c_1+1, c_2-1]$ is processed using $r_2-r_1-1$ *rank* queries on the children area in $T$. Any 1 found in the sub-area finishes the query with a positive answer. In practice $k^2$ is not so large to justify this complication, however.

#### 4.3.1. Time analysis

The total number of nodes of level $\ell$ that can overlap area $[p_1, p_2] \times [q_1, q_2]$ is

$$\begin{aligned}&(\lfloor p_2/k^{h-\ell-1} \rfloor - \lfloor p_1/k^{h-\ell-1} \rfloor + 1)\\ &\quad \cdot (\lfloor q_2/k^{h-\ell-1} \rfloor - \lfloor q_1/k^{h-\ell-1} \rfloor + 1)\\ &\quad \le ((p_2-p_1+1)/k^{h-\ell-1}+2) \cdot ((q_2-q_1+1)/k^{h-\ell-1}+2)\\ &\quad = A/(k^2)^{h-\ell-1} + P/k^{h-\ell-1} + 2,\end{aligned}$$

where $A = (p_2-p_1+1) \cdot (q_2-q_1+1)$ is the extension of the area to retrieve and $P = 2(p_2-p_1+1)+2(q_2-q_1+1)$ is the perimeter. Added over all the levels $0 \le \ell < \lceil \log_k n \rceil$, the time complexity of the range query that reports all the links adds up to $O(A+P+\log_k n) = O(A+\log_k n)$. This gives $O(n)$ for retrieving successors and predecessors (we made a finer average-case analysis in Section 4.2.1), $O(p_2-p_1+\log_k n)$ or $O(q_2-q_1+\log_k n)$ for ranges of successors or predecessors, and $O(\log_k n)$ for queries on single links.

To analyze the algorithm that checks the existence of a link we consider that, at level $\ell$, we only handle individually the $O(P/k^{h-\ell+1})$ cells that lie in the perimeter of the query area, whereas those that are fully contained in the query area can be checked using $O(P/k^{h-\ell+1})$ constant-time *rank* queries. This yields an overall time of $O(P+\log_k n)$ for this operation. This is indeed an unlikely worst case; we see in Section 6 that the times actually decrease for larger query areas, since nonempty subtrees contained in the query area are found closer to the $k^2$-tree root.

## 5. Improving the $k^2$-tree

We describe in this section various enhancements that improve the space and time performance of the basic $k^2$-tree.

### 5.1. Varying arities

As we can observe in the previous examples, the larger $k$ is, the more space $L$ needs, because its 1s have to be covered with coarser leaves. On very sparse matrices, we may spend $k^2$ bits to represent very few 1s. For example, when $k=4$ in Fig. 2, we store some last-level submatrices containing a unique 1, spending 15 more bits that are 0. On the contrary, when $k=2$, we use fewer bits for that last level of the tree.

On the other hand, the first levels of the $k^2$-tree are likely to be nonempty, and in that case using a smaller $k$ does not reduce the space, but it slows down the process of reaching the deeper levels.

A possible solution to perform best in both situations is to use a larger $k$ for the first levels of the tree and a smaller $k$ for the last levels. This strategy takes advantage of the strong points of both approaches:

- We use large values of $k$ for the first levels of subdivision: the tree is shorter, so we will be able to obtain the list of neighbors faster, as we have fewer levels to traverse.
- We use small values of $k$ for the last levels: we do not store too many bits for each 1 of the adjacency matrix, as the submatrices are smaller.

Fig. 3 illustrates a variable-arity $k^2$-tree, where we perform a first subdivision with $k = k_0 = 4$ and a second subdivision with $k = k_1 = 2$. We store the first level of the tree in $T_1$, where the subdivision uses $k_0=4$ and the second level of the tree in $T_2$, where the subdivision uses $k_1=2$. In addition, we store the $2 \times 2$ submatrices in $L$, as before:

$T_1 = 1100010001100000,$
$T_2 = 1100\ 1000\ 0001\ 0101\ 1110,$
$L = 0100\ 0011\ 0010\ 0010\ 1010\ 1000\ 0110\ 0010\ 0100.$

The navigation algorithms are similar to those explained for fixed $k$. Now we store a different sequence $T_\ell$ for each level, and $L$ for the last level. There is a different $k_\ell$ per level, which tells how many children have internal nodes of level $\ell$

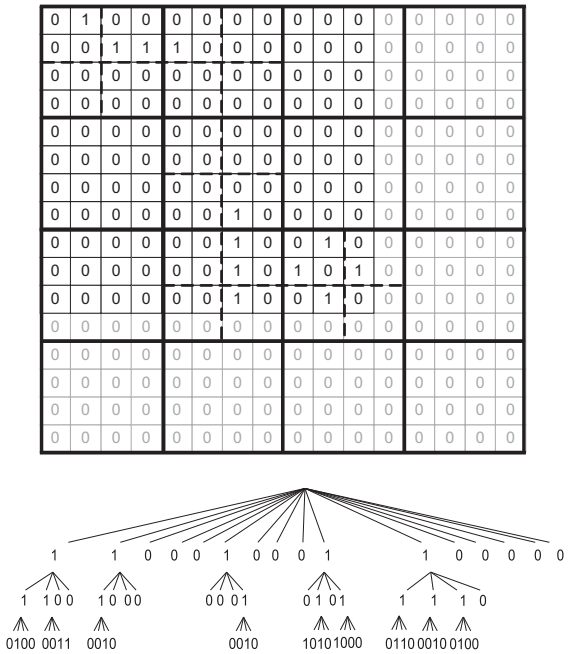


**Fig. 3.** Expansion, subdivision, and resulting $k^2$-tree using different values of $k$ at each level.

in level $\ell+1$. Thus, the formula for $child_i$ and the navigation algorithms in Section 4 must be modified accordingly. First, the $i$-th child of a node $T_\ell[x]$ will be found in $T_{\ell+1}$ at position $child_i^\ell(x) = rank(T_\ell, x-1)\cdot k_\ell^2 + i - 1$. Second, we must extend $n$ to $n' = \Pi_{\ell=0}^{h-1} k_\ell$, which plays the role of $k^h$ in the uniform case.

### 5.2. Partitioning the first level

The previous discussion suggests that it is a good idea to use a large value for $k_0$. It loses very little space (as finding very large empty submatrices is unlikely) and allows the navigation arrive faster to the deeper levels.

We go one step further and build separate $k^2$-subtrees for the $k_0^2$ first-level submatrices, of size $n^2/k_0^2$. This has several advantages. First, construction space decreases substantially, since we can build each $k^2$-tree separately and thus the space is a function of $n^2/k_0^2$ (this is important because the fastest construction algorithms in Section 3.3 require extra space proportional to the size of the tree they build), without affecting construction time. Second, navigation time improves because the bitmaps $T$ and $L$ are shorter and then we have more locality of reference. The limit is, of course, the point where we start to represent explicitly many submatrices of empty matrices.

In theoretical terms, one interesting choice is $k_0^2 = m$, and some fixed $k_\ell = k$ for the other levels. This spends $k_0^2 = m$ bits in the first level and $k^2 m(\log_{k^2}(n^2/m)+1)$ on the rest, which is almost the same space as in Section 3.2.2. The time to extract a single link is also improved to $\log_{k^2}(n^2/m)$, which becomes faster as the graph is denser. The gain in complexity for the other operations is less spectacular, from $O(A+\log_{k^2} n^2)$ to $O(A+\log_{k^2}(n^2/m))$. In practice, due to the clustering of the matrix, a much smaller value of $k_0$ turns out to be advisable.

### 5.3. Compressing the last level

The last level of the $k^2$-tree is stored as a bitmap $L$ that represents all the $k_L \times k_L$ submatrices of the original adjacency matrix containing at least one 1 (we are using $k_L = k_{h-1}$ for readability). These submatrices are represented consecutively following a depth-first traversal, composing a sequence of $k_L^2$−bit strings, each string representing a submatrix of the last level. As explained, we aim at using a small $k$ value at the last levels of the $k^2$-tree in order to reduce space, as a large $k$ value wastes many bits in these sparse submatrices. This, however, increases navigation time.

An alternative to reducing $k_L$ is to change the plain representation for these submatrices, so that instead of using a fixed ($k_L^2$) number of bits, they are compressed. We create a *vocabulary* with all the $k_L \times k_L$ nonempty submatrices that appear in the adjacency matrix, sort them by frequency, and represent each submatrix in $L$ with a pointer to its entry in the frequency-sorted array of distinct submatrices. Those pointers are integer numbers, with the property that the smaller ones are more frequent than the larger ones, and then a variable-length representation favoring small numbers is advisable. Now $L$ becomes a concatenation of those variable-length representations. To preserve efficient navigation over the compressed graph, we must guarantee fast direct access to any of those variable-length integers. For this purpose, we use Directly Addressable Codes (DACs) [48], which fit precisely these requirements. We use DACs in their "optimal space" configuration, that is, we use the variant of the codes where the optimal values for the configuration are obtained without any kind of restriction on the number of levels.

Using this technique not only allows us to use a larger $k_L$ at the leaf level, and consequently reduce navigation time and space in $T$. It also reduces considerably the space for $L$. When we increase the arity of the last level, the number of possible matrices grows fast (to $2^{k_L^2}$), so it seems we cannot go too far with $k_L$. However, only a few of these values appears in practice, due to sparseness, clustering and statistical properties of the matrix. In the experiments we have successfully used $k_L$ values from 4 to 16. In particular, this approach automatically exploits the different patterns described by Asano et al. [28], such as horizontal, vertical, and diagonal runs. Submatrices containing those patterns will appear more frequently and, consequently, fewer bits will be used to represent them.

## 6. Experimental evaluation

### 6.1. Experimental framework

We ran several experiments over some Web crawls from the Laboratory for Web Algorithmics[3] [49,16,24]. Table 1 gives the main characteristics of the graphs used: name (and version) of the graph, number of pages and links, average

[3] http://law.dsi.unimi.it

**Table 1**
Description of the Web graphs used.

| File | Pages ($n$) | Links ($m$) | Links/page | Size (MB) |
|---|---|---|---|---|
| CNR (2000) | 325,577 | 3,216,152 | 9.88 | 14 |
| EU (2005) | 862,664 | 19,235,140 | 22.30 | 77 |
| Indochina (2002) | 7,414,866 | 194,109,311 | 26.18 | 769 |
| UK (2002) | 18,520,486 | 298,113,762 | 16.10 | 1208 |
| Arabic (2005) | 22,744,080 | 639,999,458 | 28.14 | 2528 |
| UK-2007-05 (2007) | 105,896,555 | 3,738,733,648 | 35.31 | 14,666 |

links per page, and the size of a plain adjacency list representation of the graphs (using 4-byte integers).

The machine used in our tests is a 2 GHz Intel® Xeon® (8 cores) with 16 GB RAM. It ran Ubuntu GNU/Linux with kernel version 2.6.32-33-server (64 bits). The compilers used were gcc version 4.4.3, with `-O9` compiler optimizations, and javac 1.6.0_25. All tests use just one CPU core. Space is measured in bits per edge (bpe), by dividing the total space of the structure by the number of edges (i.e., links) in the graph. Time results measure average CPU user time per neighbor retrieved: We compute the time to search for the neighbors of all the pages (in random order) and divide by the total number of edges in the graph.

### 6.2. Influence of node ordering

In this section we show that none of the properties exploited by other Web graph compression methods, that is, the similarity of the adjacency lists, the skewed distribution, or the locality of reference, determine the good performance of $k^2$-trees. Rather, we show that this depends only on the clustering of the adjacency matrix (which is a consequence of, but not equivalent to, the locality of reference).

In addition, we show that $k^2$-trees perform better on other orderings than the lexicographic one, in particular with BFS [29], and thus we use this ordering in the sequel.

We will also show that the space and time analyses of Sections 3.3.2 and 4.2.1 (calculated for uniformly distributed graphs) are overly pessimistic for Web graphs.

In order to test these hypotheses, we create four graphs called `RandReord`, `DomainReord`, `BFSReord`, and `LLPReord`, which are equal to the original graph `Indochina` (called `Original` in this subsection), yet with a reordering of the identifiers of the Web pages. This reordering of the identifiers causes a reordering of the rows and columns in the adjacency matrix, since each row and each column represents one Web page according to its identifier. Web pages are alphabetically sorted by URL in graph `Original` (i.e., `Indochina`), so locality of reference translates into closeness of 1s in the adjacency matrix. `DomainReord`, `BFSReord`, `LLPReord` and `RandReord` synthetic graphs are created as follows:

- Graph `DomainReord` tries to improve locality of reference by the folklore idea of sorting the domains in reverse order, as then `aaa.bbb.com` will stay close to `zzz.bbb.com`. Within each full domain, paths are sorted in the usual lexicographic order.
- Graph `BFSReord` permutes the nodes of graph `Indochina` in a breadth-first search fashion, following the idea of Apostolico and Drovandi [29].
- Graph `LLPReord` permutes the nodes of graph `Indochina` using the "Layered Label Propagation" (LLP) algorithm, a solution that mixes clusterings and orders and it was successfully applied in the WebGraph framework, obtaining significant improvements in compression ratio for both social and Web graphs [24].
- Graph `RandReord` is obtained by a random permutation of the Web page identifiers of `Indochina`. This reordering eliminates the locality of reference, since the 1s in a row are no longer together but distributed along all the row. Yet, note this graph retains row similarity and skewed degree distributions.

We create two more graphs that eliminate the locality of reference by displacing the domains randomly in the adjacency matrix without overlaps. The first graph, `DomainsDisplace`, conserves the internal structure of each domain, whereas for the second, `DomainsDisplaceAndRand`, we uniformly distributed the intra-domain links inside each domain. For both graphs, the inter-domain links are then uniformly distributed outside the domains.

In addition, we also create a uniformly distributed graph with the same number of nodes and edges of graph `Indochina`. We call this graph `Uniform`. This graph is sparse but it does not preserve any of the properties of Web graphs: outdegrees of pages no longer follow a skewed distribution and 1s are spread along all the adjacency matrix, so there is no locality of reference nor similarity of adjacency lists.

Fig. 4 shows the effect of these transformations for the smallest Web graph of the dataset (graph `CNR`).

Table 2 compares the behavior of the $k^2$-tree structure for these eight graphs. The first column indicates the graph and the configuration used to compress it. For all the graphs we use DACs at the leaf level, using their space-optimal configuration. We also partition the graph adjacency matrices into submatrices of size $2^{20} \times 2^{20}$, and then create one variable-arity $k^2$-tree for each submatrix. We use $k_\ell = 4$ for the first 5 levels of the tree and $k_\ell = 2$ for the rest, except for the last level, $k_L$. For each graph, we create three representations with $k_L = 4$, $k_L = 8$, and $k_L = 16$. Hence, the leaf submatrices are of size $4 \times 4$, $8 \times 8$, and $16 \times 16$, respectively.

The second column of the table shows the space (in bpe) obtained by each representation. According to the space analysis in Section 3.2.2, for $k = 2$ the space of the representation for uniformly distributed graphs is

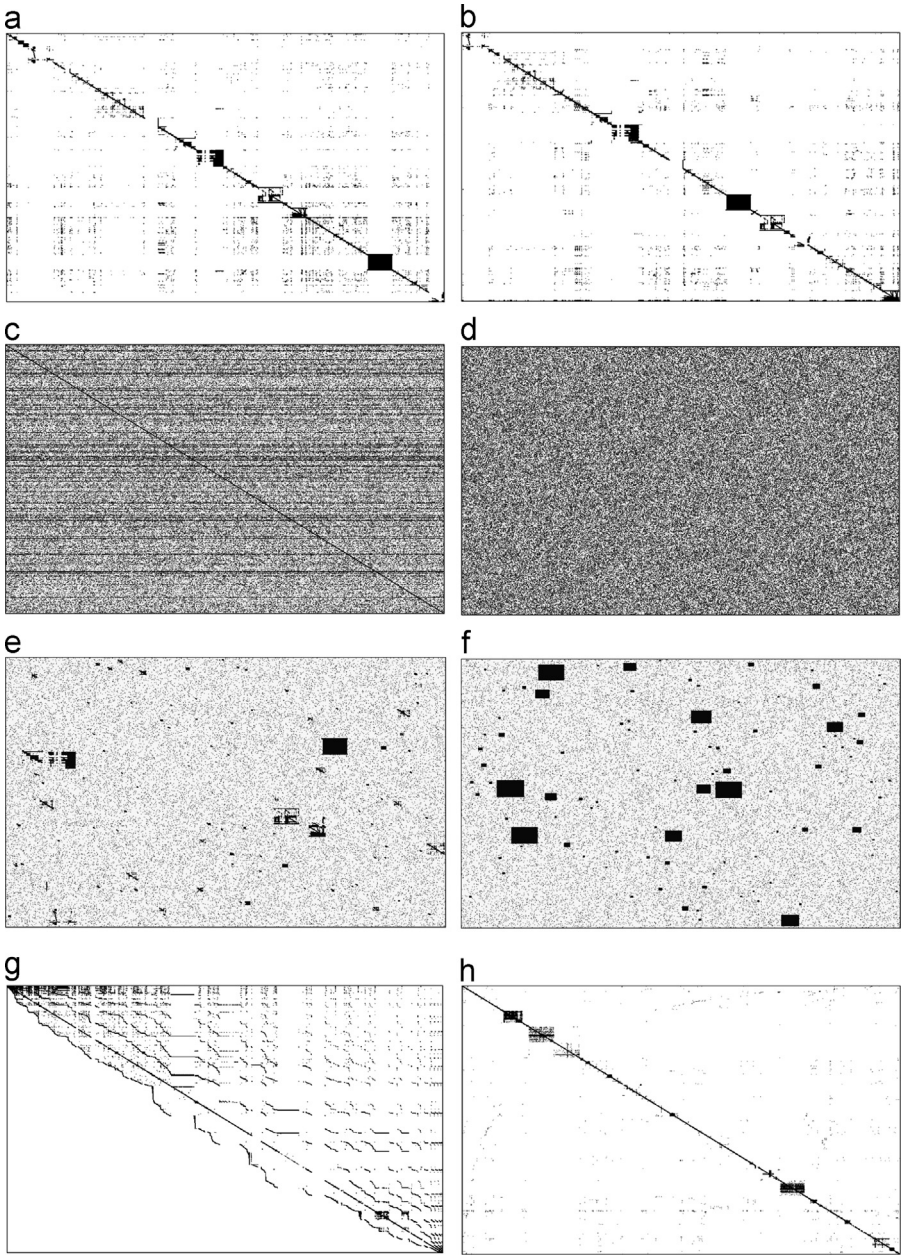


**Fig. 4.** Effect of different reorderings over the adjacency matrix of `CNR` graph: (a) `Original`, (b) `DomainReord`, (c) `RandReord`, (d) `Uniform`, (e) `DomainsDisplace`, (f) `DomainsDisplaceAndRand`, (g) `BFSReord`, (h) `LLPReord`.

asymptotically twice the information-theoretic minimum necessary to represent all the matrices of $n \times n$ with $m$ 1s, that is $\log_2\binom{n^2}{m}$, which is 17.95 for graph `Indochina`. As we can see in the table, the representation of the uniformly distributed graph occupies around 29–34 bpe, close to twice that value (and in fact not very dependent on $k$).

The results for `RandReord` show that, if we remove locality of reference, the performance of the $k^2$-tree is very close to that on a random graph. We remind that `RandReord` preserves the skewed degree distributions and row similarities, yet the compression performance of $k^2$-trees turns out to be blind to them. This seems to show that locality of reference is the important property that makes Web graphs compressible with $k^2$-trees. However, we show with `DomainsDisplace` and `DomainsDisplaceAndRand` that $k^2$-trees are indeed sensitive to the clustering of the adjacency matrix (in this case we consider the squares of intra-domain links as a rough clustering) and not necessarily to the locality of reference (note that locality of reference implies some clustering, but not vice versa). On `DomainsDisplace`, which conserves the intra-clustering structure but loses the locality of reference, we require only twice the space of the original graph. Even when inside those clusters the links are uniformly

**Table 2**
Effect of the reordering of the nodes for Indochina graph. $k_L$ is the value of $k$ used for the leaf level. The columns show, respectively, the space (in bpe); the size of the vocabulary of leaf submatrices, that is, the number of different non-zero $k_L \times k_L$ submatrices that appear in the adjacency matrix; the total number of elements of the sequence of submatrices of the last level; the required bits per leaf; and the average time per successor retrieved (in μs/e).

| Reordering | Space (bpe) | Leaves voc. | Num. leaves | $\frac{\lvert L\rvert}{Num.leaves}$ | Time (μs/e) |
|---|---|---|---|---|---|
| $k_L=4$ | | | | | |
| Original | 2.03 | 33,088 | 36,984,217 | 6.541 | 3.354 |
| DomainReord | 2.02 | 33,445 | 36,976,002 | 6.527 | 2.622 |
| BFSReord | 1.46 | 30,890 | 27,410,836 | 6.363 | 2.030 |
| LLPReord | 1.46 | 38,063 | 25,278,358 | 6.686 | 2.862 |
| RandReord | 32.43 | 218 | 192,861,682 | 5.062 | 141.211 |
| Uniform | 33.80 | 136 | 194,103,965 | 5.050 | 152.797 |
| DomainsDisplace | 4.22 | 35,151 | 44,684,209 | 6.551 | 27.193 |
| DomainsDisplaceAndRand | 10.77 | 65,528 | 136,516,474 | 5.697 | 30.933 |
| $k_L=8$ | | | | | |
| Original | 1.73 | 1,030,977 | 17,509,603 | 10.888 | 3.303 |
| DomainReord | 1.72 | 1,034,563 | 17,502,311 | 10.873 | 2.582 |
| BFSReord | **1.23** | 753,265 | 12,135,472 | 10.991 | 1.978 |
| LLPReord | 1.37 | 1,045,497 | 11,566,702 | 11.600 | 2.785 |
| RandReord | 30.25 | 3252 | 191,892,255 | 7.076 | 140.647 |
| Uniform | 31.60 | 2085 | 194,087,567 | 7.051 | 151.780 |
| DomainsDisplace | 3.83 | 1,085,770 | 26,317,833 | 9.836 | 27.183 |
| DomainsDisplaceAndRand | 10.12 | 2,441,503 | 114,991,387 | 8.523 | 30.676 |
| $k_L=16$ | | | | | |
| Original | 3.32 | 1,855,823 | 8,551,140 | 14.711 | 3.227 |
| DomainReord | 3.31 | 1,859,297 | 8,545,746 | 14.663 | 2.510 |
| BFSReord | 2.57 | 1,468,960 | 5,943,097 | 15.248 | **1.924** |
| LLPReord | 2.92 | 1,695,613 | 6,107,242 | 15.330 | 2.757 |
| RandReord | 28.18 | 85,164 | 190,628,173 | 9.105 | 139.505 |
| Uniform | 29.44 | 27,809 | 194,022,073 | 9.053 | 150.781 |
| DomainsDisplace | 5.31 | 1,874,531 | 17,930,554 | 11.825 | 26.928 |
| DomainsDisplaceAndRand | 20.49 | 9,431,141 | 79,369,216 | 8.000 | 30.756 |

distributed, the compression achieved is significantly better than in the Uniform graph.

On graphs Original, DomainReord, LLPReord, and BFSReord, the space is much better than on Uniform, as Web graphs are far from uniformly distributed and the $k^2$-tree technique takes advantage of this fact. The domain reordering slightly improves the compression ratio, since it is common that pages point to other pages inside the same domain, even if they do not share the same subdomain. These pages are distant in graph Original if their subdomains are not alphabetically close, but they are near to each other in graph DomainReord. LLPReord clusters the adjacency matrix and improves the compression ratio. Yet, BFSReord obtains the best compression ratio by a good margin (except for $k_L=4$, where it obtains the same compression ratio as LLPReord), as it creates larger areas of zeroes in the adjacency matrix.

The third column shows the size of the vocabulary of leaf submatrices, that is, the number of different non-zero $k_L \times k_L$ submatrices that appear in the adjacency matrix, as explained in Section 5.3. The fourth column shows the number of elements of the sequence of submatrices of the last level, that is, the total number of submatrices that are represented with DACs. We can observe that the vocabularies of leaves for Original, DomainReord, LLPReord, BFSReord and DomainsDisplace are larger than for RandReord and Uniform. This is because the adjacency matrices of these last two graphs have their 1s spread all along the matrix, such that it is rare that several 1s coincide in the same leaf submatrix. Since there are very few possible submatrices at the leaf level, the vocabulary of submatrices is small, but the sequence of submatrices is longer. In DomainsDisplaceAndRand, instead, the vocabulary is much larger because we have random and dense areas of the matrix. Moreover, due to the uniform distribution of the 1s in the adjacency matrix, the distribution of frequency of the submatrices of the vocabulary is also uniform. Consequently, DACs cannot obtain a very compact representation of the sequence. On the contrary, the vocabulary of leaves for a Web graph, such as Original, DomainReord, LLPReord, and BFSReord, is larger, but it follows a skewed distribution. Typical patterns such as horizontal, vertical, and diagonal runs [28] are, at some degree, captured inside those leaf submatrices. Some of them appear more frequently than others, so they are encoded using fewer bits than less frequent submatrices. Hence, the space required for the leaves representation is lower than for random graphs, taking into account the size of the leaves vocabulary. The fifth column shows this ratio. Note also that the compression of leaves makes the space non-monotonic on $k_L$.

The last column of the table shows the efficiency of successors retrieval by measuring the average time per neighbor retrieved in microseconds per edge (μs/e). In Section 4.2.1 we anticipated that navigation over Web graphs would be more efficient than over random graphs. This is because the $k^2$-tree has numerous leaves in the case of RandReord and Uniform graphs, as we can see in the table, and this implies that the retrieval of the neighbors must

traverse many more leaves to return usually just one neighbor per submatrix, or none. In the case of a Web graph, instead, the clustering makes that a visited leaf submatrix is likely to retrieve several neighbors, which reduces the navigation time. As a result, the navigation time is two orders of magnitude lower for Web graphs than for random graphs. It can also be seen that using a larger $k_L$ slightly improves navigation times.

More precisely, in Section 4.2.1 we predicted that the navigation time to retrieve a successors or predecessors list would be $O(\sqrt{m})$ on a random uniformly distributed matrix. We show now that this is indeed the case, whereas on a Web graph the time is basically linear in the size of the output list. Fig. 5a compares the time to extract an adjacency list on graphs `Original`, `RandReord` and `Uniform`. The times are separated by the length of the retrieved list, on the $x$-axis. The $y$-axis has been cut in order to properly visualize the curves, since results are considerably faster for graph `Original` than for `RandReord` and `Uniform`, as we have already seen in Table 2. We can observe that the time to retrieve an adjacency list is always linearly dependent on its length; however on the random graphs `RandReord` and `Uniform` there is a very large additive constant that shadows this effect. This dependence on the list length can be better seen in Fig. 5b, where the average query time to retrieve an adjacency list is divided by the length of the list.

Fig. 6 shows the effect of this additive constant $O(\sqrt{m})$. We show the average time per retrieved neighbor (in μs/e) for the original version of graphs `CNR`, `EU`, `Indochina` and `UK`, compared to their `RandReord` and `Uniform` versions. Note that the time per query is $O(\sqrt{m})$ and thus the time per retrieved neighbor is $O(\sqrt{m}/(m/n)) = O(n/\sqrt{m})$, so we represent the value $n/\sqrt{m}$ for each graph in the $x$-axis. We can observe that for `RandReord` and `Uniform` all the points are almost perfectly linearly aligned, thus confirming our theoretical analysis.

### 6.3. Space usage

Before comparing the $k^2$-tree with other techniques, we first show the effects of DACs on our method. We measure the space required by six different configurations of the $k^2$-tree for graph `Indochina`: `k=2 P` uses $2 \times 2$ subdivisions and the leaf level in plain form; `k=4 P` uses $4 \times 4$ subdivisions and the leaf level in plain form; `k=4 C` uses $4 \times 4$ subdivisions and the leaf level compressed with DACs; finally, `k_L=4 HC`, `k_L=8 HC`, and `k_L=16 HC` use the same configuration as in Table 2, that is, a hybrid approach and compressed leaf level. We consider separately the space for representing the tree ($T$), the leaves ($L$) and the vocabulary of leaf submatrices if needed. In addition, we also compute the successors time per edge, as in Section 6.2.

Fig. 7 shows the benefits of using DACs compared with using a plain representation of the leaf submatrices. Comparing `k=4 P` with `k=4 C` we can observe that we can achieve higher compression by just compressing the leaf submatrices with DACs, at the expense of increasing retrieval times. In addition, by using a hybrid approach as in Table 2, that is, `k_L=4 HC`, we can obtain even better spaces. If we use larger $8 \times 8$ leaf submatrices (`k_L=8 HC`), we decrease retrieval times and improve compression; with $16 \times 16$ leaf submatrices (`k_L=16 HC`) we obtain better times than with `k_L=8 HC`, but we double the space due to the overhead of representing the large resulting vocabulary. All the compressed alternatives outperform `k=2 P` both in space and in time.

Now we compare the space of our proposal with others. Table 3 compares our $k^2$-trees with the best alternatives in the literature that support direct and reverse navigation

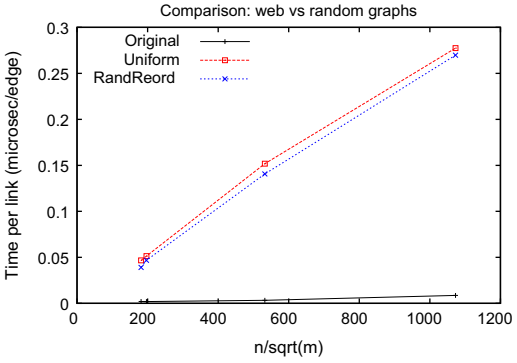


**Fig. 6.** Successors retrieval time (in μs/e) for Web graphs and random graphs compared to the corresponding constant $n/\sqrt{m}$ for different Web graphs.

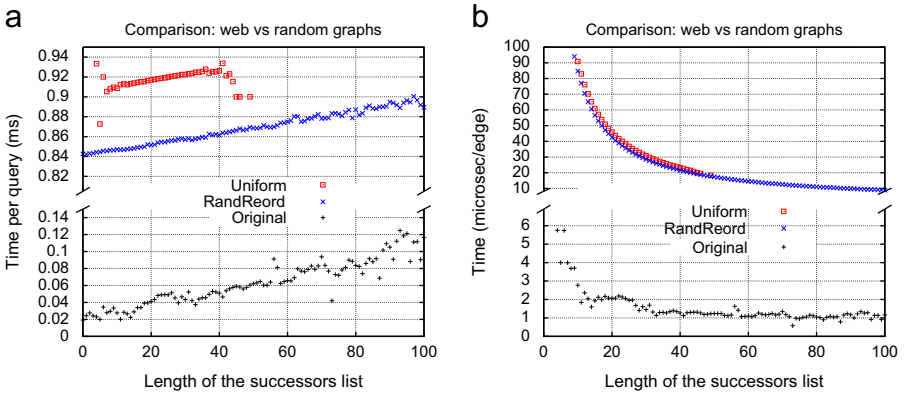


**Fig. 5.** Time performance for Web versus random graphs, all `Indochina` variants: (a) adjacency list retrieval time (in ms) for Web graphs and random graphs; (b) successor retrieval time (in μs/e) for Web graphs and random graphs.

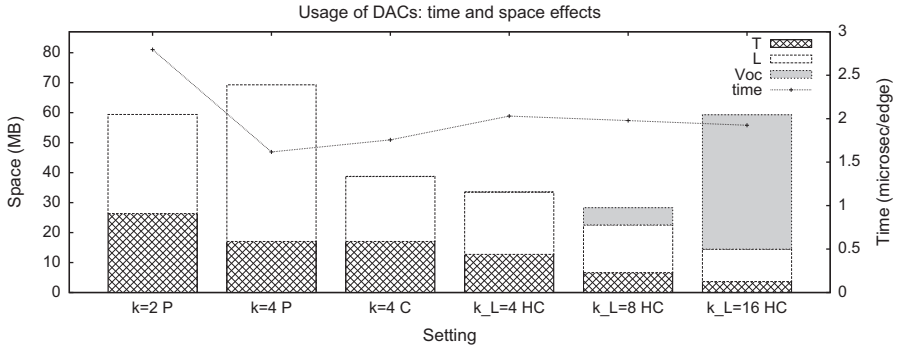


**Fig. 7.** Time and space results for different $k^2$-trees over graph `Indochina` using BFS ordering. We include three settings where no compression of the leaf levels is used (`k=2 P` and `k=4 P`) and four settings using DACs (`k=4 C`, `k_L=4 HC`, `k_L=8 HC`, and `k_L=16 HC`). `P` stands for plain, `C` for compressed with DACs, and `H` for hybrid. We represent the successor retrieval time (in μs/edge) as a line and the space (in MB) as bars. We separate the space requirements among the different parts of the structure (tree bitmap, leaves and vocabulary).

**Table 3**
Space consumption (in bpe) of the most compact representations of the $k^2$-tree and previous work, on different Web graphs. We include the alternative (`dir+rev` or `bi+dir+rev`) used to provide forward navigation on the methods that support only direct navigation. The exceptions when a particular graph uses a different alternative than the one indicated in the column are marked with an asterisk. The ordering used for each graph and technique is detailed in Section 6.3.

| Crawl | $k^2$-tree | Re-Pair WT | Re-Pair GMR | Re-Pair (bi+dir+rev) | WebGraph (dir+rev) | AD (dir+rev) | LM (dir+rev) |
|---|---|---|---|---|---|---|---|
| `EU` | **3.22** | 3.93 | 5.60 | 6.79 | 5.62* | 5.48 | 3.40* |
| `Indochina` | **1.23** | 2.20 | 3.44 | 4.15 | 2.04 | 2.00 | 2.14 |
| `UK` | **2.04** | 3.73 | 5.79 | 7.10 | 3.29 | 3.49 | 3.68 |
| `Arabic` | **1.67** | 2.51 | 3.80 | 4.92 | 2.47 | 3.02 | 2.67 |
| `UK-2007-05` | **1.69** | – | – | – | 2.11 | 2.68 | 2.26 |

within reasonable time, that is, much faster than decompressing the whole graph.[4]

The first column refers to our proposal. The space reported corresponds to the representation using BFS ordering and obtained with a partition of submatrices of size $2^{18} \times 2^{18}$ for graph `EU`, $2^{20} \times 2^{20}$ for graph `Indochina`, and $2^{22} \times 2^{22}$ for graphs `UK`, `Arabic` and `UK-2007-05`. For the next levels we use a variable-arity $k^2$-tree using $k=4$ for the first 5 levels for graphs `EU` and `Indochina`, and for the first 6 levels for graphs `UK` and `Arabic`, and then $k=2$ for the rest of the levels. Finally, we use $k_L=8$ at the last level of the tree and DACs with optimal-space configuration.

The second and third columns of Table 3 correspond to alternatives `Re-Pair WT` and `Re-Pair GMR` presented by Claude and Navarro [27] (code from the authors), which used BFS ordering, except for `Re-Pair WT` over graph `EU`, where lexicographic ordering was better. They apply Re-Pair compression to the adjacency lists and then encode the resulting sequences using compact data structures (`WT` and `GMR`) that offer direct and reverse navigation. We could not build those representations on the largest graph, `UK-2007-05`.

The rest of the columns correspond to techniques that only support direct navigation, where we represent the original and the transposed graph (denoted by `dir+rev`) or we use the technique proposed by Boldi et al. [24] to extract the "undirected" part of the graph, and transpose only the rest (denoted by `bi+dir+rev`). We apply both techniques to the Re-Pair compression of Claude and Navarro [25] (`Re-Pair`, code publicly available at http://webgraphs.recoded.cl), to the best representative of the WebGraph framework [16] (`WebGraph`, code publicly available at http://webgraph.dsi.unimi.it), and to the technique of Apostolico and Drovandi [29] (`AD`, version 0.3.2 of their software, publicly available at http://www.dia.uniroma3.it/drovandi/software.php). We also included the `LM` method by Grabowski and Bieniecki [31], using the source code provided by the authors.[5] Then we chose the most compact configuration of those methods for each graph: `bi+dir+rev` is used for `Re-Pair` and `dir+rev` for `WebGraph`, `AD`, and `LM`, except for graph `EU`, which achieves smaller representations for `WebGraph` and `LM` using alternative `bi+dir+rev` (those are marked with an asterisk in Table 3).

For WebGraph we have used their version 3.0.2, which uses the so-called "layered label propagation" (LLP) algorithm [24]. Only the space required by their

[4] A similar table reported before [27] did not include recent improvements on the $k^2$-tree obtained after the conference version [15], such as using DACs and BFS.

[5] $k^2$-tree is implemented in C, `Re-Pair` and `LM` are implemented in C++, whereas `WebGraph` and `AD` are implemented in Java.

structure on disk is reported, even if the process requires some extra memory to run.[6] For this comparison, WebGraph parameters are set to favor compression over speed (window size 10, maximum reference unlimited). With this parameterization, they retrieve successors in about 100 μs/e, which is much slower than all the others (we consider more time-efficient parameterizations when we measure time performance).

For `Re-Pair(bi+dir+rev)` we use BFS reordering except for `EU` graph, which obtains its best result with natural order. We could not build this representation on the largest graph, `UK-2007-05`. Although some specific construction techniques for very large graphs are proposed by Claude and Navarro [25], they were more difficult to run and the structure is too far from competitive with the other `(dir+rev)` alternatives.

For `LM(dir+rev)` we use LLP reordering, except for `EU` graph, which also obtains its best result with natural order.

As we can see, our proposal obtains the best space from all the alternatives of the literature, by a good margin. Previously, `Re-Pair WT` had been shown to achieve the smallest space when supporting the retrieval of successors and predecessors within microseconds [27], but our new $k^2$-tree outperforms it (and, in addition, it is much faster, as we see soon).

We have excluded from the table the technique of Asano et al. [28]. We do not have access to their code, but they report 1.99 bpe on graph `CNR` and 2.78 on graph `EU`. Their time performance, as we see later, is several orders of magnitude slower, because they need to decompress whole domains in order to extract adjacency lists. In this sense, their method is closer to a plain compression method than to a compact data structure. While they only support the successors functionality, it is possible that they could support extracting predecessors within little more space, because they have to decompress a whole domain anyway, and most of the predecessors can be found in the process. They would only need to represent inter-domain links separately, and these are not many.

### 6.4. Construction space and time requirements

We have detailed several methods for constructing the $k^2$-tree. The most practical ones in RAM are those described in Sections 3.3.3 and 3.3.5. Thus, we compare now the memory usage and construction time for compressing graph `UK-2007-05` using those two construction algorithms of the $k^2$-tree with the rest of the techniques configured to achieve high compression as in Section 6.3.

For this experiment, the machine used was an Intel® Core™ i7-3820 CPU at 3.60 GHz (8 cores) with 64 GB RAM. It ran Ubuntu GNU/Linux with kernel version 3.2.0-44-generic (64 bits). The compilers used were gcc version 4.6.3 and javac 1.6.0_25. All tests use just one CPU core. Time results measure CPU user time computed using the command `time` from Debian GNU/Linux distribution. The memory usage reported is the peak virtual memory size used by the program, which is read from the `/proc/[pid]/status` file, being `[pid]` the process identifier of the program.

[6] This conservative estimate is used because it is difficult to measure the actual memory footprint of their process.

**Table 4**
Memory usage and user time (in min) required for compressing graph `UK-2007-05` (just for successors or for successors and predecessors) with different techniques.

| Technique | Memory | Time dir | Time dir+rev |
|---|---|---|---|
| $k^2$-tree (Section 3.3.3) | 20.32 GB | 22 | 22 |
| $k^2$-tree (Section 3.3.5) | 16.73 GB | 19 | 19 |
| WebGraph | 4.29 MB | 82 | 183 |
| AD | 43.29 GB | 116 | 211 |
| LM | 31.42 MB | 29 | 58 |

**Table 5**
Time for checking individual links of Web graphs using $k^2$-trees.

| Time (μs) | Whole adjacency list | Average time per link | Single link |
|---|---|---|---|
| `EU` | 55.120 | 2.472 | 0.124 |
| `Indochina` | 51.781 | 1.978 | 0.106 |

Table 4 shows the results. The first column shows the memory requirements, the second column shows user time (expressed in minutes) to compress the graph to answer just successors, and the third column shows user time to compress the graph to answer successors and predecessors, that is, to compress the direct and the transpose graph. The two construction algorithms for the $k^2$-tree obtain similar results, but the inplace construction requires 20% less space and 15% less time. This algorithm uses a space similar to the input size of the graph (see Table 1), which shows that the space of the queue is not significant in practice. This is because the algorithm at Section 3.3.3 does require $O(t)$ words of extra space for building the tree, whereas the $O(t)$ words of extra space for the queue is an unusual worst case for the inplace algorithm. Also, the inplace algorithm obtains the lowest time, even compared to the time needed by the other techniques to compress just the direct graph. However, it requires much more space than `WebGraph` and `LM`. `AD`, on the other hand, obtains both higher space and time requirements.

### 6.5. Single link retrieval

We now study the efficiency of the navigation using the $k^2$-tree, and compare it with previous work. We start by analyzing its performance to determine if there is a link from a specific Web page to another.

In Section 4.1 we showed how to determine the existence of a link between two given pages $p$ and $q$. Most other techniques can answer this query only by extracting the adjacency list of $p$ until they can determine that $q$ is not present. The only proposals we know of that solve this query faster than via extracting the whole adjacency list are `AD`, which achieves on an average 60% of the time needed to retrieve the whole adjacency list, and `WebGraph`, which is able to retrieve successors lazily in increasing order, thus, on an average, only half of the list has to be retrieved.

Table 5 shows, on graphs `EU` and `Indochina`, the average time needed by the $k^2$-tree to retrieve a whole

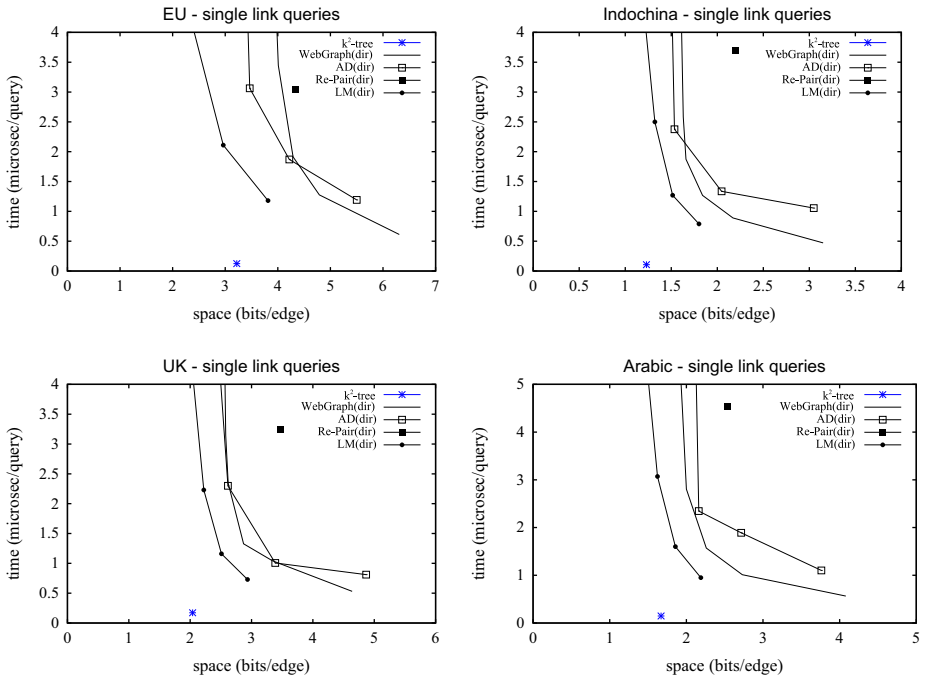


**Fig. 8.** Space/time tradeoff to solve single-link queries on graphs `EU`, `Indochina`, `UK`, and `Arabic` using different methods.

adjacency list, the corresponding time per link retrieved, and the time of a single-link query. We have used the smallest $k^2$-tree variants, as described in Section 6.3. We measure the average time to test links between random pairs of nodes (i.e., existing and non-existing links).

It can be seen that the time used by the $k^2$-tree to retrieve a single link is two orders of magnitude lower than the time to retrieve the whole list. What is more striking is that the single-link query time is also one order of magnitude faster than the per-link time of a successor query. This is because the latter needs to explore the whole row of the matrix (hopefully, but not always, avoiding to enter up to the $k^2$-tree leaves), whereas the single-link query goes directly to a single cell.

Fig. 8 shows the comparison with other methods of the state of the art. We measure the average time per single-link query over Web graphs `EU`, `Indochina`, `UK`, and `Arabic`. In addition to our $k^2$-tree, we include `AD(dir)`, `WebGraph(dir)`, `LM(dir)`, and `Re-Pair(dir)`, all of them using only the space to support successor queries. `AD(dir)` was configured with parameter $\ell = 4, 8, 16, 100$, and we used the specific query for checking single-links. `WebGraph(dir)` was configured with different values for parameters $w$ and $m$ and we used the lazy successors retrieval until the link can be confirmed or discarded. Finally, for `LM(dir)` and `Re-Pair(dir)` we estimate the time to answer a single-link query as half of the time required for retrieving the whole list of successors.[7] `LM` was used with their parameter $h$ set at power-of-two values between 8 and 64. Finally, we configured `Re-Pair(dir)` with the same parameters used by the authors in their experiments [25].

As we can see from the figures, $k^2$-tree is faster than the rest of the alternatives (5–10 times), even when only the space for supporting successors is taken into account and those methods are configured to be as efficient as they can. When those alternatives are configured to achieve the smaller space as possible, our proposal becomes one order of magnitude faster (20–50 times). Hence, $k^2$-tree outperforms the rest of the methods by a large margin when retrieving single links.

### 6.6. Successors and predecessors retrieval times

The technique `Re-Pair WT` [27], which had obtained the smallest space reported in the literature (and is superseded by the new $k^2$-tree in Table 3), can navigate the graph in compressed form, but it requires around 35 µs/e for the successors list and 55 for the predecessors list. For example, this would require more than one hour to completely traverse a small/medium-sized Web graph, such as `Indochina`. This technique will be excluded from our time comparisons, because all the other techniques, including $k^2$-tree, offer navigation times (usually well) below 10 µs/e. To be concrete, `Re-Pair WT` loses by a wide margin, in space and time, to the new $k^2$-tree. Instead, we include a variant of `Re-Pair GMR` labeled `Re-Pair GMR(2)`, which uses a complementary form of the `GMR` sequence representation that favors successors over predecessors retrieval [27].

Fig. 9 shows the space/time tradeoff for retrieving successors and predecessors over different graphs. We measure the average time efficiency in µs/e as before. Representations providing space/time tuning parameters appear as a line, whereas the others appear as a point.

[7] Notice that this is an optimistic estimation, thus, `WebGraph` and `AD` may obtain better times than those reported for `LM`.

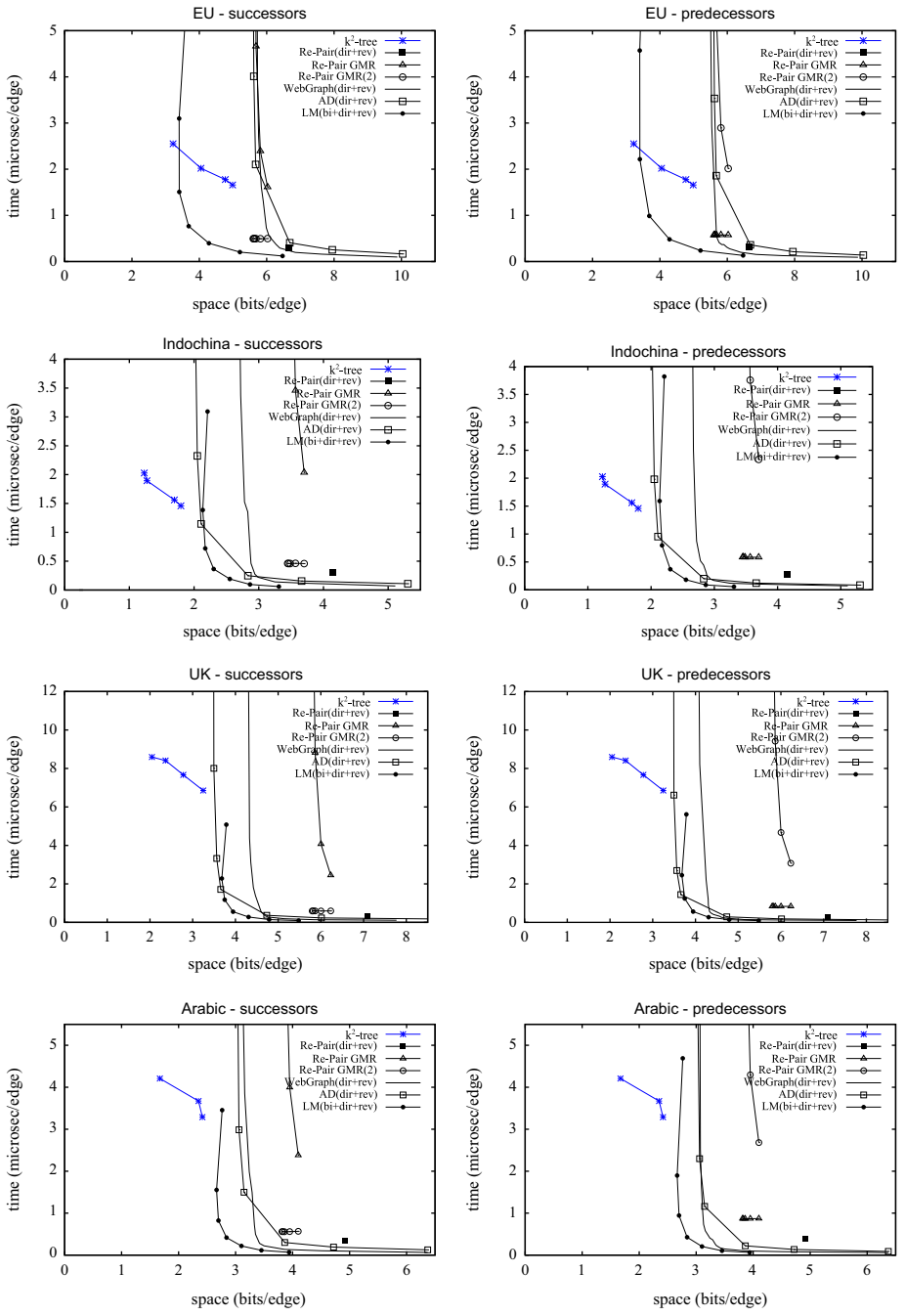


**Fig. 9.** Space/time tradeoff to retrieve successors (left) and predecessors (right) on graphs `EU`, `Indochina`, `UK`, and `Arabic`.

The space/time tradeoffs for `Re-Pair(bi+dir+rev)`, `Re-Pair GMR` and `Re-Pair GMR(2)` were obtained using the same parameters used by the authors in their experiments [25,27]. For `WebGraph (dir+rev)` we chose the dominant points among the results obtained by varying parameter $m$ between 0 and 5000 and parameter $w$ between 0 and 70. For `AD (dir+rev)`, we varied their parameter $\ell$ between 4 and 1500. We include the space for the offsets and indexes of the first node of each chunk (64 and 32 bits per chunk, respectively) to support random access to the graph. For `LM(dir+rev)`, we varied their parameter $h$ using power-of-two values between 8 and 512. Finally, we obtained several space/time tradeoffs for the $k^2$ tree by varying the $k$ parameter in different levels (combining alternatives $k=2$–8), using or not a large $k_0$ for the first level, using $k_L=4$–9 for the last level, and using either an optimal-space or a faster variant in DACs for the last level. We then show the dominant points.

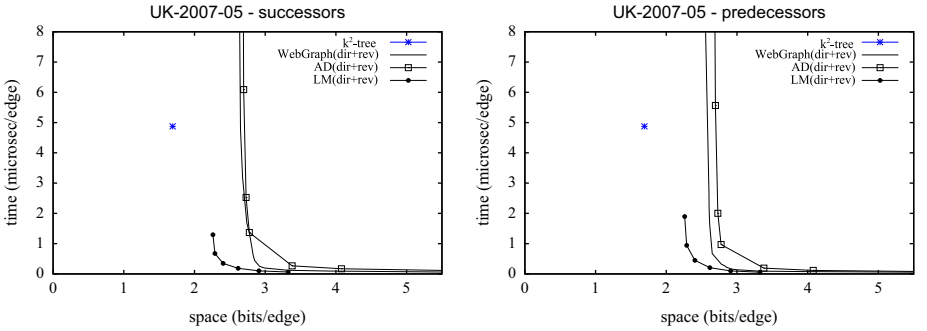


**Fig. 10.** Space/time tradeoff to retrieve successors (left) and predecessors (right) for graph `UK-2007-05`.

**Table 6**
Time per link for retrieving the successors list for different configurations of several methods over graph `UK-2007-05`.

| Configuration | Space (bpe) | Time (μs/e) |
|---|---|---|
| $k^2$-tree (as in Section 3) | 1.69 | 4.874 |
| `Webgraph` $w=70, m=1000$ | 2.61 | 16.771 |
| `Webgraph` $w=70, m=300$ | 2.65 | 4.801 |
| `Webgraph` $w=3, m=3$ | 3.32 | 0.121 |
| `AD` $\ell=1000$ | 2.68 | 12.666 |
| `AD` $\ell=200$ | 2.73 | 2.528 |
| `AD` $\ell=16$ | 3.38 | 0.267 |
| `LM` $h=256$ | 2.26 | 1.294 |
| `LM` $h=64$ | 2.41 | 0.449 |
| `LM` $h=8$ | 3.33 | 0.062 |

Fig. 10 shows the comparison on a much larger graph, `UK-2007-05`. As said before, here we do not include the representation `Re-Pair GMR` (nor `Re-Pair GMR (2)`), since we could not build them for such a large graph. We also discarded `Re-Pair(bi+dir+rev)`, as explained before. For the $k^2$-tree we show only a single point because other alternatives were larger and slower. Table 6 shows the figures for some of these points.

As we can see, our representation achieves the best compression (as low as 1.3–3 bpe, depending on the graph) among the techniques that provide successors and predecessors retrieval. The alternatives that get closest in space are `LM(dir+rev)` and `AD(dir+rev)`, depending on the graph. They achieve very fast navigation when occupying about twice the space of $k^2$-trees, but they get considerably slower when using less than that space. `WebGraph(dir+rev)` behaves similarly, yet the lowest space where it reaches very fast navigation is larger than that of `AD(dir+rev)` and `LM(dir+rev)`, especially on the smaller graphs (all these methods become very similar on the largest graphs). `Re-Pair GMR` obtains attractive space results on the smallest graph, but not on the larger ones. Moreover, it is much slower to retrieve successors (just like `Re-Pair GMR (2)` on predecessors; this technique is strongly asymmetric). `Re-Pair(bi+dir+rev)` also gives attractive times, but it is not competitive in space with techniques that are as fast as it. Finally, it is interesting to note that `LM(dir+rev)` performs particularly well on the smallest graph, `EU`, where it gets much closer than the others to our minimum-space configuration. Yet, on the larger graphs, it becomes very similar to `AD(dir+rev)` and `WebGraph(dir+rev)`.

**Table 7**
Space comparison between the $k^2$-tree and `VNM` on various graphs.

| Space (bpe) | `VNM` (∞) | `VNM` | $k^2$-`tree` |
|---|---|---|---|
| `EU` | 2.90 | 4.07 | 3.22 |
| `UK` | 1.95 | 3.75 | 2.04 |
| `Arabic` | 1.81 | 2.91 | 1.67 |

The $k^2$-tree obtains the best results when little space is available, and this is its main value when bidirectional navigation is required. However, if more main memory space is available, representations like `LM`, `AD`, and `WebGraph` obtain the list of predecessors or successors many times faster, using 2–4 times the space. Further, if we only require to carry out forward navigation, alternatives `Re-Pair`, `WebGraph`, `AD` and `LM` become preferable (smaller and faster than ours) in most cases.

We now consider proposals for which we do not have access to the code, but still can carry out some meaningful comparisons thanks to the shared WebGraph testbed. Buehrer and Chellapilla [11] proposed a technique we will label `VNM`, for "virtual node mining", which transforms the graph into a smaller one, and then encodes it.

Table 7 shows in the first column, `VNM` (∞), the space consumption they report for graphs `EU`, `UK`, and `Arabic`. This space does not include the storage of the offset per node, which is essential to provide direct access to the graph (and hence navigation). It is to be taken as a compression method. The second column, `VNM`, adds the space of their pointers array, such that the compressed graph can be randomly navigated. In the last column we repeat the space results of our smallest navigable representation of the $k^2$-tree (from Table 3). As we can observe, `VNM` obtains significantly worse compression ratio than the $k^2$-tree, even considering that `VNM` supports only successors retrieval (we would need to use a "`(dir+rev)`" variant, that could up to double the space, to have predecessors functionality as well). No traversal times are reported by Buehrer and Chellapilla [11].

However, the technique of Buehrer and Chellapilla [11] is still of interest in this scenario, since the output of the first part of the method is another graph, which can be encoded with any other technique. Hernández and Navarro [30] explore this path, and find in particular that the combination of the graph-reduction method with $k^2$-trees (considering the

conference version [15] with the DAC improvement, but not BFS) achieves a slight space reduction with respect to the plain $k^2$-tree. However, this comes at the price of a 10-fold increase in navigation times.

We also compare our proposal with the method by Asano et al. [28] (Asano). We used our smallest graphs, on which they have reported experiments. Table 8 shows the space and average time needed to retrieve the whole adjacency list of a node, in milliseconds per list. For this experiment, we represent graph EU with the configuration described in Section 6.3 and graph CNR using BFS ordering and a partition of submatrices of size $2^{16} \times 2^{16}$, $k=4$ for the first 4 levels, and then $k=2$ for the rest of the levels. Finally, we use $k_L=8$ at the last level of the tree and DACs with optimal-space configuration. The resulting compression ratio and time results are shown in the third column of the table (we remind that Asano does not support predecessors retrieval).

We observe that our method is two orders of magnitude faster to retrieve an adjacency list, while the space is slightly larger. The time difference is so large that it is possible to be more space-competitive and still much faster if part of our structure (e.g., $L$) is on disk. Our main memory space in this case, omitting bitmap $L$, is reduced to about half the space of the $k^2$-tree. The last column of the table shows the space needed by the $T$ part of our structure, and the navigation time (real time) required when the accesses to $L$ resort to disk. The structure is about 30% slower than the main $k^2$-tree but still two orders of magnitude faster than Asano.

### 6.7. Range searches

We now study the performance of range queries on the $k^2$-tree representation of Web graphs (recall Section 4.3). We compare the average time needed to retrieve $r>0$ consecutive lists of successors, of nodes $p$ to $p+r-1$, with the time spent to obtain all the links in the range $[p, p+r) \times [0, n]$ with a range query. Fig. 11a shows the results for $r=1$–20. For these experiments we used the $k^2$-tree of graph Indochina that chooses $k=2$ at each level of the tree and represents the leaves in plain form, to make the results of the experiments independent of the effects of the leaf vocabulary.

As it can be seen, except for $r=1$ (when the range includes just 1 node, and the query is equivalent to a successor query), retrieving consecutive lists of successors is slower than the corresponding range query (for 1 node, the range query adds a small overhead that is not present in neighbor queries). As $r$ grows, the time of the range query stays almost constant (which is much better than the worst-case theoretical result of $O(A+\log_k n)=O(rn)$ given in Section 4.3.1), while retrieving $r$ consecutive list of neighbors increases linearly with $r$, as expected.

Section 4.3 shows that checking the existence of a link in a range can, in turn, be done faster than listing all the links. Fig. 11b shows the average query time in milliseconds for this operation, on the same ranges as before. This is compared with the time of the range query that reports all the links in the same range (i.e., the same of Fig. 11a). Now that it is not dwarfed by the time to extract individual lists, it can be seen that the time to extract all the links in a range does increase with $r$, albeit rather slowly. Instead, checking the existence of a link inside the same range not only performs significantly faster, but its time decreases as the range width increases. This was already intuitively anticipated in Section 4.3.1.

These two operations can be useful to regard the graph at a higher level, for example formed by domains, while being able to list the domains some domain points to, the domains that point to some domain, whether two domains are linked, and so on. (For this sake we need to use the lexicographic ordering of nodes.)

**Table 8**
Comparison with approach Asano on small graphs.

| | Asano | $k^2$-tree | $k^2$-tree no-$L$ |
|---|---|---|---|
| Space (bpe) | | | |
| CNR | 1.99 | 3.11 | 0.80 |
| EU | 2.78 | 3.22 | 0.90 |
| Time (ms/list) | | | |
| CNR | 2.34 | 0.017 | 0.022 |
| EU | 28.72 | 0.055 | 0.073 |

## 7. Conclusions and future work

We have presented a new compact representation for Web graphs, the *$k^2$-tree*, that uses very little space (as little as 1.3–3 bits per edge, 60–80% of the best alternative representations), by taking advantage of the sparseness and clustering of the adjacency matrices. It supports various queries on the graph, such as checking the presence of single links (in about 0.1 μs,

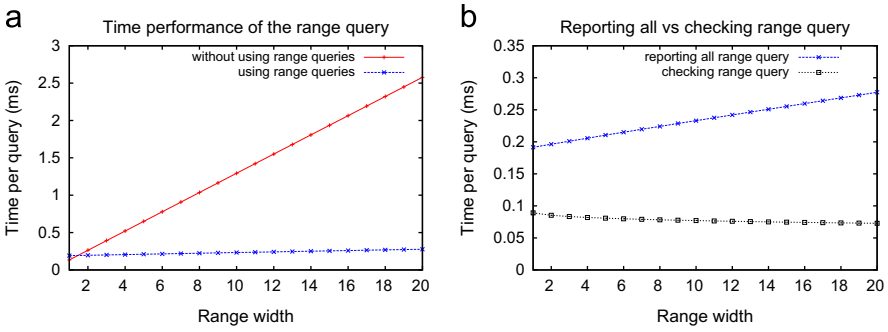


**Fig. 11.** Performance of range queries. (a) Range query performance compared to simple list retrieval queries, for different range widths. (b) Checking the existence of a link in a range compared to finding all the links in that range.

10 times faster than existing representations), bidirectional navigation towards successors and predecessors of a page (in 2–8 μs per neighbor retrieved, which is 10–20 times slower than the best alternatives), and even more general range queries such as recovering all the connected pairs of Web pages inside a range, or determining whether there exists a link between ranges of Web pages.

Our experiments show that the $k^2$-tree offers, by a significant margin, the smallest graph representation that efficiently supports direct and reverse navigation. No alternative can reach its space without being orders of magnitude slower. The most competitive alternatives [29,24,31] use about twice the space of the $k^2$-tree, in which case they do offer significantly faster navigation. On the other hand, to test individual links the $k^2$-tree is an order of magnitude faster than the other representations, even if they use much more space.

This makes the $k^2$-tree an attractive alternative to handle Web graphs that require very efficient compression to fit in the available main memory, and where we want to carry out specific queries such as testing individual links between nodes or ranges of nodes, or find predecessors or successors of specific nodes. We have described in the Introduction various Web graph analysis problems where such a functionality is of interest. On the other hand, when we only need limited functionality such as computing successors, or need bidirectional navigation but have enough main memory space to hold the graph and its transpose, alternative representations are much faster, especially if we aim at traversing a large portion of the graph.

Since the conference publication [15], the $k^2$-tree has been used in other scenarios, some of which have already been commented along the paper. Claude and Ladra [50] combined a $k^2$-tree representation for intra-node edges with a Re-Pair compression of inter-node edges, achieving more space but less time than plain $k^2$-trees (in many cases the result matches the most interesting space/time tradeoff of Apostolico and Drovandi [29]). Hernández and Navarro [30] combined $k^2$-trees with virtual node mining [11], achieving slightly less space but 10-fold slowdown. They have also combined $k^2$-trees with other structures and obtained good results on compressing social networks, particulary using dense subgraphs [51]. Shi et al. [52] also applied $k^2$-trees to social networks, and postulated that the "maturity" of the network can be estimated from the compression ratio achieved by the data structure. The $k^2$-tree has also been successfully applied to represent RDF graphs [53], whose labels have attributes, and then various $k^2$-trees must be used. Finally, $k^2$-trees have also been used to compress more general graphs [54].

The structure we have introduced can be of more general interest than graph compression. It could be fruitful, for example, to use it to represent general binary relations, as its range search capabilities are useful to handle many queries of interest [55]. For example, an application can be the representation of discrete point grids, for computational geometry applications or geographic information systems. The $k^2$-tree would take advantage when the data is clustered. Romero et al. [56] used the $k^2$-tree as a basis to create a succinct moving object structure for timestamp and interval queries. Dynamic variants of the $k^2$-tree have also been pursued [57], and these increase the applicability of this versatile structure.

## Acknowledgments

Funded by MICINN (PGE and FEDER) Grants TIN2009-14560-C03-02 and Xunta de Galicia (co-funded with FEDER) Grants 2010/17 and CN 2012/211 (for the first and second authors); and for the third author by Millennium Nucleus Information and Coordination in Networks ICM/FIC P10-024F. We also thank the reviewers for their comments, which improved the quality of the paper.

## References

[1] D. Donato, S. Millozzi, S. Leonardi, P. Tsaparas, Mining the inner structure of the Web graph, in: Proceedings of 8th Workshop on the Web and Databases (WebDB), 2005, pp. 145–150.
[2] J. Kleinberg, R. Kumar, P. Raghavan, S. Rajagopalan, A. Tomkins, The Web as a graph: measurements, models, and methods, in: Proceedings of 5th Annual International Conference on Computing and Combinatorics (COCOON), Lecture Notes in Computer Sciences, vol. 1627, 1999, pp. 1–17.
[3] S. Brin, L. Page, The anatomy of a large-scale hypertextual Web search engine, Computer Networks 30 (1998) 107–117.
[4] L. Page, S. Brin, R. Motwani, T. Winograd, The PageRank Citation Ranking: Bringing Order to the Web, Technical Report 1999–66, Stanford InfoLab, 1999.
[5] L. Becchetti, C. Castillo, D. Donato, R. Baeza-Yates, S. Leonardi, Link analysis for Web spam detection, ACM Transactions on the Web 2 (2008).
[6] M. Henzinger, A. Heydon, M. Mitzenmacheri, M. Najork, On near-uniform URL sampling, Computer Networks 33 (2000) 295–308.
[7] H. Saito, M. Toyoda, M. Kitsuregawa, K. Aihara, A large-scale study of link spam detection by graph algorithms, in: Proceedings of 3rd International Workshop on Adversarial Information Retrieval on the Web (AIRWeb), 2007, p. 48.
[8] A. Benczúr, K. Csalogány, T. Sarlós, M. Uher, Spamrank: fully automatic link spam detection, in: Proceedings of 1st International Workshop on Adversarial Information Retrieval on the Web (AIRWeb), 2005.
[9] B. Wu, B. Davison, Identifying link farm spam pages, in: Proceedings of 14th International World Wide Web Conference (WWW), Special Interest Tracks and Posters, 2005, pp. 820–829.
[10] R. Kumar, P. Raghavan, S. Rajagopalan, A. Tomkins, Trawling the Web for emerging cyber-communities, Computer Networks 31 (1999) 1481–1493.
[11] G. Buehrer, K. Chellapilla, A scalable pattern mining approach to Web graph compression with communities, in: Proceedings of 1st ACM International Conference on Web Search and Data Mining (WSDM), 2008, pp. 95–106.
[12] D. Gibson, J. Kleinberg, P. Raghavan, Inferring Web communities from link topology, in: Proceedings of 9th ACM Conference on Hypertext and Hypermedia, 1998, pp. 225–234.
[13] J. Leskovec, K.J. Lang, M. Mahoney, Empirical comparison of algorithms for network community detection, in: Proceedings of the 19th International Conference on World Wide Web, WWW'10, ACM, New York, NY, USA, 2010, pp. 631–640.
[14] J. Vitter, External memory algorithms and data structures: dealing with massive data, ACM Computing Surveys 33 (2001) 209–271.
[15] N. Brisaboa, S. Ladra, G. Navarro, K2-trees for compact web graph representation, in: Proceedings of 16th International Symposium on String Processing and Information Retrieval (SPIRE), Lecture Notes in Computer Sciences, vol. 5721, 2009, pp. 18–30.
[16] P. Boldi, S. Vigna, The WebGraph framework I: compression techniques, in: Proceedings of 13th International World Wide Web Conference (WWW), 2004, pp. 595–601.
[17] K. Bharat, A. Broder, M. Henzinger, P. Kumar, S. Venkatasubramanian, The connectivity server: fast access to linkage information on the Web, in: Proceedings of 7th International World Wide Web Conference (WWW), 1998, pp. 469–477.

[18] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, J. Wiener, Graph structure in the Web, Computer Networks 33 (2000) 309–320.
[19] M. Adler, M. Mitzenmacher, Towards compressing Web graphs, in: Proceedings of 11th Data Compression Conference (DCC), 2001, pp. 203–212.
[20] T. Suel, J. Yuan, Compressing the graph structure of the Web, in: Proceedings of 11th Data Compression Conference (DCC), 2001, pp. 213–222.
[21] K. Randall, R. Stata, R. Wickremesinghe, J. Wiener, The LINK Database: Fast Access to Graphs of the Web, Technical Report 175, Compaq Systems Research Center, Palo Alto, CA, 2001.
[22] S. Raghavan, H. Garcia-Molina, Representing Web graphs, in: Proceedings of 19th International Conference on Data Engineering (ICDE), 2003, p. 405.
[23] P. Boldi, M. Santini, S. Vigna, Permuting Web and social graphs, Internet Mathematics 6 (2009) 257–283.
[24] P. Boldi, M. Rosa, M. Santini, S. Vigna, Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks, in: Proceedings of 20th International World Wide Web Conference (WWW), 2011, pp. 587–596.
[25] F. Claude, G. Navarro, Fast and compact Web graph representations, ACM Transactions on the Web (TWEB) 4 (2010) article 16.
[26] J. Larsson, A. Moffat, Off-line dictionary-based compression, Proceedings of the IEEE 88 (2000) 1722–1732.
[27] F. Claude, G. Navarro, Extended compact Web graph representations, in: T. Elomaa, H. Mannila, P. Orponen (Eds.), Algorithms and Applications (Ukkonen Festschrift), Lecture Notes in Computer Sciences, vol. 6060, 2010, pp. 77–91.
[28] Y. Asano, Y. Miyawaki, T. Nishizeki, Efficient compression of Web graphs, in: Proceedings of 14th Annual International Conference on Computing and Combinatorics (COCOON), Lecture Notes in Computer Science, vol. 5092, 2008, pp. 1–11.
[29] A. Apostolico, G. Drovandi, Graph compression by BFS, Algorithms 2 (2009) 1031–1044.
[30] C. Hernández, G. Navarro, Compression of Web and social graphs supporting neighbor and community queries, in: Proceedings of 5th ACM Workshop on Social Network Mining and Analysis (SNA-KDD), 2011.
[31] S. Grabowski, W. Bieniecki, Merging adjacency lists for efficient web graph compression, in: Man–Machine Interactions 2 AISC 103, 2011, pp. 385–392.
[32] N. Brisaboa, R. Cánovas, F. Claude, M. Martínez-Prieto, G. Navarro, Compressed string dictionaries, in: Proceedings of 10th International Symposium on Experimental Algorithms (SEA), Lecture Notes in Computer Sciences, vol. 6630, 2011, pp. 136–147.
[33] H. Samet, Foundations of Multidimensional and Metric Data Structures, Morgan Kaufmann Publishers Inc., 2006.
[34] I. Simecek, Sparse matrix computations using the quadtree storage format, in: Proceedings of the 11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2009, pp. 168–173.
[35] G. Jacobson, Space-efficient static trees and graphs, in: Proceedings of 30th IEEE Symposium on Foundations of Computer Science (FOCS), 1989, pp. 549–554.
[36] I. Munro, V. Raman, Succinct representation of balanced parentheses and static trees, SIAM Journal on Computing 31 (2001) 762–776.
[37] D. Benoit, E. Demaine, I. Munro, R. Raman, V. Raman, S.S. Rao, Representing trees of higher degree, Algorithmica 43 (2005) 275–292.
[38] R. Geary, N. Rahman, R. Raman, V. Raman, A simple optimal representation for balanced parentheses, Theoretical Computer Science (TCS) 368 (2006) 231–246.
[39] K. Sadakane, G. Navarro, Fully-functional succinct trees, in: Proceedings of 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2010, pp. 134–149.
[40] J. Jansson, K. Sadakane, W.-K. Sung, Ultra-succinct representation of ordered trees, in: Proceedings of 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2007, pp. 575–584.
[41] D. Arroyuelo, R. Cánovas, G. Navarro, K. Sadakane, Succinct trees in practice, in: Proceedings of 11th Workshop on Algorithm Engineering and Experiments (ALENEX), 2010, pp. 84–97.
[42] I. Munro, Tables, in: Proceedings of 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS), Lecture Notes in Computer Sciences, vol. 1180, 1996, pp. 37–42.
[43] R. González, S. Grabowski, V. Mäkinen, G. Navarro, Practical implementation of rank and select queries, in: Poster Proceedings of Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA), 2005, pp. 27–38.
[44] S. Gog, M. Petri, Optimized succinct data structures for massive data, Software: Practice and Experience. DOI: http://dx.doi.org/10.1002/spe.2198, in press.
[45] K. Mehlhorn, Data Structures and Algorithms 1: Sorting and Searching, EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1984.
[46] M. He, I. Munro, Succinct representations of dynamic strings, in: Proceedings of 17th International Symposium on String Processing and Information Retrieval (SPIRE), Lecture Notes in Computer Sciences, vol. 6393, 2010, pp. 334–346.
[47] G. Navarro, K. Sadakane, Fully-functional static and dynamic succinct trees, ACM Transactions on Algorithms (TALG), in press.
[48] N. Brisaboa, S. Ladra, G. Navarro, DACs: bringing direct access to variable-length codes, Information Processing and Management 49 (2013) 392–404.
[49] P. Boldi, B. Codenotti, M. Santini, S. Vigna, Ubicrawler: a scalable fully distributed web crawler, Software: Practice and Experience 34 (2004) 711–726.
[50] F. Claude, S. Ladra, Practical representations for Web and social graphs, in: Proceedings of 20th ACM Conference on Information and Knowledge Management (CIKM), 2011, pp. 1185–1190.
[51] C. Hernández, G. Navarro, Compressed representation of web and social networks via dense subgraphs, in: Proceedings of 19th International Symposium on String Processing and Information Retrieval (SPIRE), Lecture Notes in Computer Sciences, vol. 7608, 2012, pp. 264–276.
[52] Q. Shi, Y. Xiao, N. Bessis, Y. Lu, Y. Chen, R. Hill, Optimizing K2-trees: a case for validating the maturity of network of practices, Computers and Mathematics with Applications 63 (2012) 427–436.
[53] S. Álvarez-García, N.R. Brisaboa, J. Fernández, M. Martínez-Prieto, Compressed k2-triples for full-in-memory RDF engines, in: Proceedings of 17th Americas Conference on Information Systems (AMCIS), 2011.
[54] S. Álvarez-García, N. Brisaboa, S. Ladra, O. Pedreira, A compact representation of graph databases, in: Proceedings of 8th Workshop on Mining and Learning with Graphs (MLG), ACM, 2010, pp. 18–25.
[55] J. Barbay, F. Claude, G. Navarro, Compact rich-functional binary relation representations, in: Proceedings of 9th Latin American Symposium on Theoretical Informatics (LATIN), Lecture Notes in Computer Sciences, vol. 6034, 2010, pp. 170–183.
[56] M. Romero, N. Brisaboa, M.A. Rodríguez, The smo-index: a succinct moving object structure for timestamp and interval queries, in: Proceedings of the 20th International Conference on Advances in Geographic Information Systems, SIGSPATIAL '12, ACM, 2012, pp. 498–501.
[57] N. Brisaboa, G. de Bernardo, G. Navarro, Compressed dynamic binary relations, in: Proceedings of 22th Data Compression Conference (DCC), 2012, pp. 52–61.