K^2-trees for Compact Web Graph Representation

Nieves R. Brisaboa
Database Laboratory
University of A Coruña
A Coruña, Spain
brisaboa@udc.es

Susana Ladra
Database Laboratory
University of A Coruña
A Coruña, Spain
sladra@udc.es

Gonzalo Navarro
Dept. of Computer Science
University of Chile
Santiago, Chile
gnavarro@dcc.uchile.cl

ABSTRACT

The directed graph representation of the World Wide Web has been extensively used to analyze the Web structure, behavior and evolution. However, those graphs are huge and do not fit in main memory, whereas the required graph algorithms are inefficient in secondary memory. Compressed graph representations reduce their space while allowing efficient navigation in compressed form. As such, they allow running main-memory graph algorithms on much larger Web subgraphs. In this paper we present a Web graph representation based on a very compact tree structure that takes advantage of large empty areas of the adjacency matrix of the graph. Our results show that our method is competitive with the best alternatives in the literature, offering an interesting space/time tradeoff. It gives a very good compression ratio (3.3–5.3 bits per link) while permitting fast navigation on the graph to obtain direct as well as reverse neighbors (2–15 microseconds per neighbor delivered). Moreover, we show that our representation allows for extended functionality not usually considered in compressed graph representations.

Categories and Subject Descriptors

E.1 [**Data structures**]; E.2 [**Data storage representations**]; E.4 [**Coding and information theory**]: Data compaction and compression; H.3.2 [**Information storage and retrieval**]: Information storage—*File organization*

General Terms

Algorithms

Keywords

Web graphs, Compact data structures

1. INTRODUCTION

The World Wide Web structure can be regarded as a directed graph at several levels, the finest grained one being pages that point to pages. Many algorithms of interest to obtain information from the Web structure are essentially basic algorithms applied over the Web graph. One of the classical references on this topic [17] shows how the HITS algorithm to find hubs and authorities on the Web starts by selecting random pages and finding the induced subgraphs, which are the pages that point to or are pointed from the selected pages. Donato et al. [12] show how several common Web mining techniques used to discover the structure and evolution of the Web graph build on classical graph algorithms such as depth-first search, breath-first-search, reachability, and weakly and strongly connected components. A more recent example [23] presents a technique for Web spam detection that boils down to algorithms for finding strongly connected components, for clique enumeration, and for minimum cuts. There are entire conferences devoted to graph algorithms for the Web (e.g. *WAW: Workshop on Algorithms and Models for the Web-Graph*).

The problem of how to run typical graph algorithms over those huge Web graphs is always present in those approaches. Even the simplest external memory graph algorithms, such as graph traversals, are usually non disk-friendly [25]. This has pushed several authors to consider *compressed graph representations*, which aim to offer memory-efficient graph representations that still allow for fast navigation without decompressing the graph. The aim of this research is to allow classical graph algorithms to be run in main memory over much larger graphs than those affordable with a plain representation.

The most famous representative of this trend is surely the *WebGraph Framework*, by Boldi and Vigna [6]. It is associated to the site `http://webgraph.dsi.unimi.it`, which by itself witnesses the level of maturity and sophistication that this research area has reached.

The WebGraph compression method is indeed the most successful member of a family of approaches to compress Web graphs based on their statistical properties [5, 7, 1, 24, 22, 21]. Boldi and Vigna's representation allows fast extraction of the neighbors of a page while spending just a few bits per link (about 2 to 6, depending on the desired navigation performance). Their representation explicitly exploits Web graph properties such as: (1) the power-law distribution of indegrees and outdegrees, (2) the locality of reference, (3) the "copy property" (the set of neighbors of a page is usually very similar to that of some other page).

More recently, Claude and Navarro [10] showed that most of those properties are elegantly captured by applying Re-Pair compression [18] on the adjacency lists, and that *reverse navigation* (that is, finding the pages that point to a given page) could be achieved by representing the output of Re-Pair using some more sophisticated data structures [9]. Reverse navigation is useful to compute several relevance ranking on pages, such as HITS, PageRank, and others. Their technique offers better space/time tradeoffs than WebGraph, that is, they offer faster navigation than

.

WebGraph when both structures use the same space. Yet, WebGraph is able of using less space if slower navigation can be tolerated.

Asano et al. [2] achieve even less than 2 bits per link by explicitly exploiting regularity properties of the adjacency matrix of the Web graphs, such as horizontal, vertical, and diagonal runs. In exchange for achieving much better compression, their navigation time is substantially higher, as they need to uncompress full domains in order to find the neighbors of a single page.

In this paper we also aim at exploiting the properties of the adjacency matrix, yet with a general technique to take advange of clustering rather than a technique tailored to particular Web graphs. We introduce a compact tree representation of the matrix that not only is very efficient to represent large empty areas of the matrix, but at the same time allows efficient forward and backward navigation of the graph. An elegant feature of our solution is that it is symmetric, in the sense that forward and backward navigation are carried out by similar means and achieve similar times.

We show experimentally that our technique offers a relevant space/time tradeoff to represent Web graphs, that is, it is much faster than those that take less space, and much smaller than those that offer faster navigation. Thus our representation can become the preferred choice for many Web graph traversal applications: Whenever the compression it offers is sufficient to fit the Web graph in main memory, it achieves the best traversal time within that space. Furthermore, we show that our representation allows for other queries on the graph not usually considered in compressed graph representations.

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 in the methods, as it is 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. We follow this assumption in this paper.

2. OUR PROPOSAL

2.1 Conceptual description

The adjacency matrix of a Web graph of n pages is a square matrix $\{a_{ij}\}$ of $n \times n$, 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 on average there are about 15 links per Web page, this matrix is extremely sparse. Due to 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 colums have many 1s, but most have very few. More specific properties of Web graphs have been already studied [2].

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.

We represent the adjacency matrix by a k^2-ary tree, which we call 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 of the tree, where all are leaves and represent some 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 internal nodes in the tree (i.e., with value 1) have exactly k^2 children, whereas leaves (with value 0 or at the last tree level) have no children.

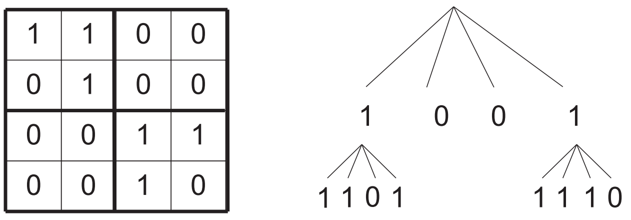


Figure 1: Example of the representation of the adjacency matrix of a Web graph.

Assume for simplicity that n is a power of k; we will soon remove this assumption. Conceptually, we start dividing the adjacency matrix 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 in the cells of the submatrix there is at least one 1. 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, with the children of root, has been built, the method proceeds recursively for each child with value 1, until we reach submatrices full of 0s, or we reach the cells of the original adjacency matrix. In the last level of the tree, the bits of the nodes correspond to the matrix cell values. Figure 1 illustrates a 2^2-tree for a 4×4 matrix.

A larger k induces a shorter tree, with fewer levels, but with more children per internal node. 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 $n' = k^{\lceil \log_k n \rceil}$. This does not cause a significant overhead as our technique is efficient to handle large areas of 0s.

Figure 2 shows an example of the adjacency matrix of a Web graph (we use the first 11×11 submatrix of graph CNR [6]), and how it is expanded to an $n' \times n'$ matrix for n' a power of $k = 2$ (at the left) and of $k = 4$ (at the right). The figure also shows the trees corresponding to those k values.

Notice that the last level of the tree represents cells in the original adjacency matrix, but most empty cells in the original adjacency matrix are not represented in this level because, where a large area with 0s is found, it is represented by a single 0 in a smaller level of the tree.

2.2 Navigating with a k^2-tree

To obtain the pages pointed by a specific page p, that is, to find direct neighbors of page p, we need to find the 1s in row p of the matrix. We start at the root and travel down the tree, choosing exactly k children of each node. We start with an example and then generalize in the next section.

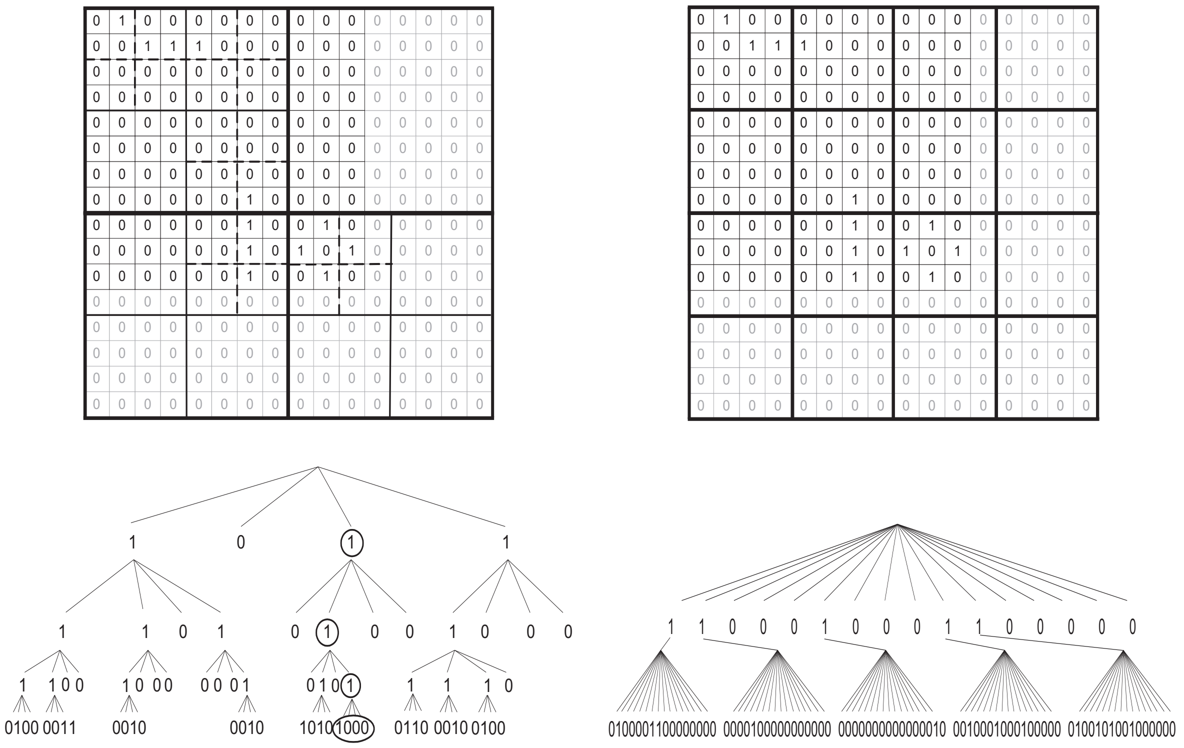


Figure 2: Expansion and subdivision of the adjacency matrix (top) and resulting trees (bottom) for $k = 2$ (left) and $k = 4$ (right).

Example. We find the pages pointed by the first page in the example of Figure 1, that is, find the 1s of the first matrix row. We start at the root of the 2^2-tree and compute which children of the root overlap the first row of the matrix. These are the first two children, to which we move:

- The first child is a 1, thus it has children. To figure out which of its children are useful we repeat the same procedure. We compute in the corresponding submatrix (the one at the top left corner) which of its children represent cells overlapping the first row of the original matrix. These are the first and the second children. They are leaf nodes and their values are 1 and 1.
- The second child of the root represents the second submatrix, but its value is 0. This means that all the cells in the adjacency matrix in this area are 0.

Now we know that the Web page represented by this first row has a link to itself and another to page 2.

Reverse neighbors. An analogous procedure retrieves the list of reverse neighbors. To obtain which pages point to page q, we need to locate which cells have a 1 in column q of the matrix. Thus, we carry out a symmetric algorithm, using columns instead of rows. For instance, to know the pages pointing to the last page (placed at the rightmost column) we compute the children of the root representing submatrices that overlap that column. These are the second and the fourth children. The second child has value 0, therefore no pages in those rows point to the last page. The fourth child has a 1, therefore we compute which of its children represent submatrix overlapping cells in the last columns; these are the second and the fourth. Now we can conclude that the last page is only pointed by page 3.

Summarizing, the use of the k^2-tree is completely symmetric to search for direct or for reverse neighbors. The only difference is the formula used to compute the children of each node that will be used in the next step. In either case we perform a top-down traversal of the tree. If we want to search for direct(reverse) neighbors in a k^2-tree, we go down through k children forming a row(column) inside the matrix.

3. DATA STRUCTURE AND ALGORITHMS

Our data structure is essentially a compact tree of N nodes. There exist several such representations for general trees [15, 20, 4, 13], 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 is achieved by a so-called "ultra-succinct" representation [16] for general trees. Our representation is much simpler, and close to the so-called LOUDS (level-ordered unary degree sequence) tree representation [15, 11] (which would not achieve $N + o(N)$ bits if directly applied to our trees).

Our data structure can be regarded as a simplified variant of LOUDS for the case where arities are just k^2 and 0, which achieves the information-theoretic minimum of $N + o(N)$ bits, provides the traversal operations we require (basically move to the i-th child, although also parent is easily supported) in constant time, and is simple and practical.

3.1 Data structure

We represent the whole adjacency matrix via the k^2-tree using two bit arrays we call T (tree) and L (leaves):

T: This bit array stores all the bits of the k^2-tree except those in the last level. The bits of the k^2-tree are placed following a levelwise traversal of the tree. We will represent first the k^2 binary values of the children of the root node, then the values of the second level, and so on.

L: This bit array stores the last level of the tree. Thus it represents the value of (some) original cells of the adjacency matrix.

We create over T an auxiliary structure that enables us to compute *rank* queries efficiently. Given an offset i inside a sequence T of bits, $rank(T, i)$ counts the number of times the bit 1 appears in $T[1, i]$. This can be supported in constant time and fast in practice using sublinear space on top of the bit sequence [15, 19]. In practice we use an implementation that uses 5% of extra space on top of the bit sequence and provides fast queries, as well as another that uses 37.5% extra space and is much faster [14].

We do not need to perform *rank* over the bits in the last level of the tree; that is the practical reason to store them in a different bitmap (L). Thus the space overhead for *rank* is paid only over T.

3.1.1 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 $\lceil \log_{k^2}(n^2) \rceil$ nodes. Each such (internal) node costs k^2 bits, for a total of $k^2 m \lceil \log_k n \rceil$ bits. However, especially 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 the total space is $\sum_{\ell=1}^{\lfloor \log_{k^2} m \rfloor} k^{2\ell} + k^2 m (\lceil \log_{k^2} n^2 \rceil - \lfloor \log_{k^2} m \rfloor) = k^2 m (\log_{k^2} \frac{n^2}{m} + O(1))$ bits in the worst case.

This shows that, at least in a worst-case analysis, a smaller k yields less space occupancy. For $k = 2$ the space is $4m(\log_4 \frac{n^2}{m} + O(1)) = 2m \log_2 \frac{n^2}{m} + O(m)$ 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 the worst case, as Web graphs are far from uniformly distributed.

Finally, the expansion of n to the next power of k can, in the horizontal direction, force the creation of at most k^ℓ new children of internal nodes at level $\ell \geq 1$ (level $\ell = 1$ is always fully expanded unless the matrix is all zeros). Each such child will cost k^2 extra bits. The total excess is $O(k^2 \cdot k^{\lceil \log_k n \rceil - 1}) = O(k^2 n)$ bits, which is usually negligible. The vertical expansion is similar.

3.2 Finding a child of a node

Our levelwise traversal satisfies the following property, which permits fast navigation to the i-th child of node x, $child_i(x)$ (for $0 \leq i < k^2$):

LEMMA 1. *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 $T : L$*

PROOF. $T : L$ is formed by traversing the tree levelwise and appending the bits of the tree. We can likewise regard this as traversing the tree levelwise and appending the k^2 bits of the childred of the 1s found at internal tree nodes. By the time node x is found in this traversal, we have already appended k^2 bits per 1 in $T[1, x-1]$, plus the k^2 children of the root. As $T[x] = 1$, the children of x are appended at positions $rank(T, x) \cdot k^2$ to $rank(T, x) \cdot k^2 + (k^2 - 1)$. $\square$

Example. To represent the 2^2-tree of Figure 2, arrays T and L have the following values:

$$T = 1011\ 1101\ 0100\ 1000\ 1100\ 1000\ 0001\ 0101\ 1110,$$
$$L = 0100\ 0011\ 0010\ 0010\ 1010\ 1000\ 0110\ 0010\ 0100.$$

Remember that in T each bit represents a node. The first four bits represent the 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 the children of node 3 start at position 12. The bit in position 4, that is the fifth bit of T, represents the first child of node 0, and so on.

For the following, we mark with a circle the involved nodes in Figure 2. We compute the second child of the third node is, that is, child 1 of node 2. If we compute *rank* until the position of the bit representing node 2, $rank(T, 2) = 2$, we obtain that there are 2 nodes with children until that position because each bit 1 represents a node with children. As each node has 4 children, we multiply by 4 the number of nodes to know where it starts. As we need the second child, this is $child_1(2) = rank(T, 2) * 2^2 + 1 = 2 * 4 + 1 = 9$. In position 9 there is a 1, thus it represents a node with children and its fourth child can be found at $child_3(9) = rank(T, 9) * 2^2 + 3 = 7 * 4 + 3 = 31$. Again it is a 1, therefore we can repeat the process to find its children, $child_0(31) = rank(T, 31) * 2^2 + 0 = 14 * 4 + 0 = 56$. As $56 \geq |T|$, we know the position belongs to the last level, corresponding to offset $56 - |T| = 56 - 36 = 20$ (to 23) in L.

3.3 Navigation

To find the direct(reverse) neighbors 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. We have already explained that these are obtained by a top-down tree traversal that chooses k out of the k^2 children of a node, and also gave the way to obtain the i-th child of a node in our representation. The only missing piece is the formula that maps global row numbers to the children number at each level.

Recall $h = \lceil \log_k n \rceil$ is the height of the tree. Then the nodes at level ℓ represent square submatrices of width $k^{h-\ell}$, and these are divided into k^2 submatrices of width $k^{h-\ell-1}$. Cell (p_ℓ, q_ℓ) at a matrix of level ℓ belongs to the submatrix at row $\lfloor p_\ell / k^{h-\ell-1} \rfloor$ and column $\lfloor q_\ell / k^{h-\ell-1} \rfloor$.

Let us call p_ℓ the relative row position of interest at level ℓ. Clearly $p_0 = p$, and row p_ℓ of the submatrix of level ℓ corresponds to children number $k \cdot \lfloor p_\ell / k^{h-\ell-1} \rfloor + j$, for $0 \leq j < k$. The relative position in those children is $p_{\ell+1} = p_\ell \bmod k^{h-\ell-1}$. Similarly, column q corresponds $q_0 = q$ and, in level ℓ, to children number $j \cdot k + \lfloor q_\ell / k^{h-\ell-1} \rfloor$, for

$0 \leq j < k$. The relative position at those children is $q_{\ell+1} = q_\ell \bmod k^{h-\ell-1}$.

The algorithm for extracting direct and reverse neighbors is described in Figure 3. The one for direct neighbors is called **Direct**$(k^h, p, 0, -1)$, where the parameters are: current submatrix size, row of interest in current submatrix, column offset of the current submatrix in the global matrix, 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. The one for reverse neighbors is called **Reverse**$(k^h, q, 0, -1)$, where the parameters are the same except that the second is the column of interest and the third 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 turned out to be slightly more efficient in practice.

```
Direct($n, p, q, x$)
 1.  If $x \geq |T|$ Then // leaf
 2.    If $L[x - |T|] = 1$ Then output $q$
 3.  Else // internal node
 4.    If $x = -1$ or $T[x] = 1$ Then
 5.      $y = rank(T, x) \cdot k^2 + k \cdot \lfloor p/(n/k) \rfloor$
 6.      For $j = 0 \ldots k-1$ Do
 7.        Direct($n/k, p \bmod (n/k), q + (n/k) \cdot j,\ y + j$)
```

```
Reverse($n, q, p, x$)
 1.  If $x \geq |T|$ Then // leaf
 2.    If $L[x - |T|] = 1$ Then output $p$
 3.  Else // internal node
 4.    If $x = -1$ or $T[x] = 1$ Then
 5.      $y = rank(T, x) \cdot k^2 + \lfloor q/(n/k) \rfloor$
 6.      For $j = 0 \ldots k-1$ Do
 7.        Reverse($n/k, q \bmod (n/k), p + (n/k) \cdot j, y + j \cdot k$)
```

Figure 3: Returning direct(reverse) neighbors of page $p(q)$. It is assumed that n is a power of k and that $rank(T, -1) = 0$.

3.3.1 Analysis

Our navigation time has no worst-case guarantees better than $O(n)$, as a row $p-1$ full of 1s followed by p full of 0s could force a **Direct** 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^ℓ submatrices at each level of the tree, and therefore the total work is on 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 $O(k^t + m \cdot \sum_{\ell=t+1}^{h-1} k^\ell / k^{2\ell}) = O(k^t + m/k^t) = O(\sqrt{m})$ time. This is not the ideal $O(m/n)$ (average output size), but much better than $O(n)$ or $O(m)$.

Again, if the matrix is clustered, the average performance is indeed better than under 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.

3.4 Construction

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

Our procedure builds the tree recursively. If we are at the last level, we read the k^2 corresponding matrix cells. If all are zero, we return zero, otherwise we output their k^2 values and return 1. If we are not at the last level, we make the k^2 recursive calls for the children. If all return zero, we return zero, otherwise we output the k^2 answers of the children and return 1.

We have separate arrays for each level, so that the k^2 bits that are output at each level are appended to the corresponding array. 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.

Figure 4 shows the construction process. It is invoked as **Build**$(n, 1, 0, 0)$, where the first parameter is the submatrix size, the second is the current level, the third is the row offset of the current submatrix, and the fourth is the column offset. After running it we must carry out $T = T_1 : T_2 : \ldots : T_{h-1}$ and $L = T_h$.

```
Build($n, \ell, p, q$)
 1.  $C$ = empty sequence
 2.  For $i = 0 \ldots k-1$ Do
 3.    For $j = 0 \ldots k-1$ Do
 4.      If $\ell = \lceil \log_k n \rceil$ Then // leaf level
 5.        $C = C : a_{p+i, q+j}$
 6.      Else // internal node
 7.        $C = C$ : Build($n/k, \ell + 1,$
                  $p + i \cdot (n/k), q + j \cdot (n/k)$)
 8.  If $C = 0^{k^2}$ Then Return 0
 9.  $T_\ell = T_\ell : C$
10.  Return 1
```

Figure 4: Building the tree representation.

The total time is clearly linear in the matrix size, that is, $O(n^2)$. If, instead, we have an adjacency list representation of the matrix, 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 for 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 could try to achieve time proportional to m, the number of 1s in the matrix. For this sake we could 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. After the tree is built we can traverse it levelwise to build the final representation, or recursively to output the bits to different sequences, one per level, as before. The space could still be $O(k^2 m (1 + \log_{k^2} \frac{n^2}{m}))$, that is, proportional to the final tree size, if we used some dynamic compressed parentheses representation of trees [8]. The total time would be $O(\log m)$ per bit of the tree.

Note that, as we produce each tree level sequentially, and also traverse each matrix row (or adjacency list) sequentially, we can construct the tree on disk in optimal I/O time provided we have main memory to maintain $\log_k n$ disk blocks to output the tree, plus B disk blocks (where B is the disk page size in bits) for reading the matrix. The reason we do not need the n row buffers for reading is that we can cache the rows by chunks of B only. If later we have to read again from those rows, it will be after having processed a submatrix of $B \times B$ (given the way the algorithm traverses the matrix), and thus the new reads will be amortized by the parts already processed. This argument does not work on the adjacency list representation, where we need the n disk page buffers.

4. A HYBRID APPROACH

As we can observe in the examples of the previous section, the greater k is, the more space L needs, because even though there are fewer submatrices in the last level, they are larger. Hence we may spend k^2 bits to represent very few 1s. Notice for example that when $k = 4$ in Figure 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 leaf level.

We can improve our structure if we use a larger k for the first levels of the tree and a small 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.

Figure 5 illustrates this hybrid solution, where we perform a first subdivision with $k = 4$ and a second subdivision with $k = 2$. We store the first level of the tree in T_1, where the subdivision uses $k = 4$ and the second level of the tree in T_2, where the subdivision uses $k = 2$. In addition, we store the 2×2 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 algorithms used to obtain the direct and reverse neighbors are similar to those explained for fixed k. Now we have a different sequence T_ℓ for each level, and L for the last level. There is a different k_ℓ per level, so Lemma 1 and algorithms **Direct** and **Reverse** for navigation in Section 3.3 must be modified accordingly. We must also extend n to $n' = \Pi_{\ell=0}^{h-1} k_\ell$, which plays the role of k^h in the uniform case.

5. EXPERIMENTAL EVALUATION

We ran several experiments over some Web crawls from the *WebGraph* project. Table 1 gives the main characteristics of the graphs used. The first column indicates the name of the graph (and the WebGraph version used). Second and

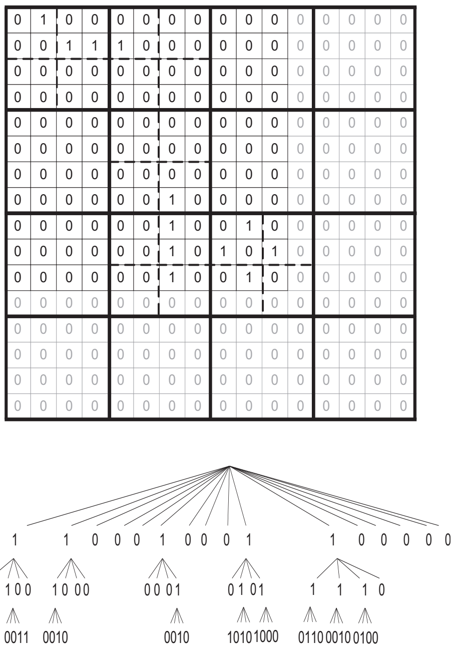


Figure 5: Expansion, subdivision, and final example tree using different values of k.

Table 1: Description of the graphs used.

File	Pages	Links	Size (MB)
CNR (2000)	325,577	3,216,152	14
EU (2005)	862,664	19,235,140	77
Indochina (2002)	7,414,866	194,109,311	769
UK (2002)	18,520,486	298,113,762	1,208

third columns show the number of pages and links, respectively. The last column gives the size of a plain adjacency list representation of of the graphs (using 4-byte integers).

The machine used in our tests is a 2GHz Intel® Xeon® (8 cores) with 16 GB RAM. It ran Ubuntu GNU/Linux with kernel version 2.4.22-15-generic SMP (64 bits). The compiler was gcc version 4.1.3 and `-O9` compiler optimizations were set. 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.

5.1 Comparison between different alternatives

We first study our approach with different values of k. Table 2 shows 12 different alternatives of our method over the EU graph using different values of k. All build on the *rank* structure that uses 5% of extra space [14]. The first column names the approaches as follows: $'2 \times 2'$, $'3 \times 3'$ and $'4 \times 4'$ stand for the alternatives where we subdivide the matrix into 2×2, 3×3 and 4×4 submatrices, respectively, in every level of the tree. On the other hand, we denote $'H - i'$ the hybrid approach where we use $k = 4$ up to level i of the tree, and then we use $k = 2$ for the rest of the levels. The second and third columns indicate the size,

Variant	Tree (bytes)	Leaves (bytes)	Space (bpe)	Direct (μs/e)	Reverse (μs/e)
2×2	6,860,436	5,583,076	5.21076	2.56	2.47
3×3	5,368,744	9,032,928	6.02309	1.78	1.71
4×4	4,813,692	12,546,092	7.22260	1.47	1.42
$H-1$	6,860,432	5,583,100	5.21077	2.78	2.62
$H-2$	6,860,436	5,583,100	5.21077	2.76	2.59
$H-3$	6,860,412	5,583,100	5.21076	2.67	2.49
$H-4$	6,861,004	5,583,100	5.21100	2.53	2.39
$H-5$	6,864,404	5,583,100	5.21242	2.39	2.25
$H-6$	6,876,860	5,583,100	5.21760	2.25	2.11
$H-7$	6,927,924	5,583,100	5.23884	2.10	1.96
$H-8$	7,159,112	5,583,100	5.33499	1.97	1.81
$H-9$	8,107,036	5,583,100	5.72924	1.79	1.67

Table 2: Comparison of our different approaches over graph EU.

in bytes, used to store the tree T and the leaves L, respectively. The fourth column shows the space needed in main memory by the structures (e.g., including the extra space for *rank*), in bits per edge. Finally, the last two columns show the times to retrieve the direct (fifth column) and reverse (sixth) neighbors, measured in microseconds per link retrieved (μs/e).

We observe that, when we use a fixed k, we obtain better times when k is greater, because we are shortening the height of the tree, but the compression ratio worsens, as the space for L becomes dominant and many 0s are stored in there.

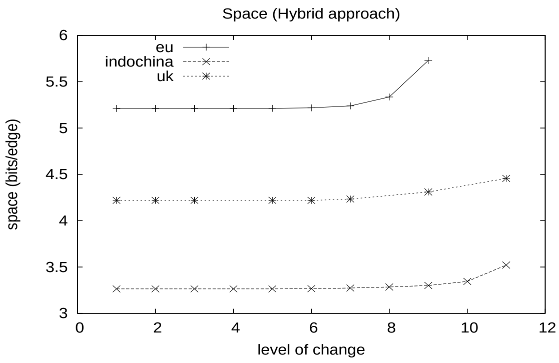
If we use a hybrid approach, we can maintain a compression ratio close to that obtained by the $'2 \times 2'$ alternative while improving the time, until we get close to the $'4 \times 4'$ alternative. The best compression is obtained for $'H-3'$, even better than $'2 \times 2'$. Figure 6 shows similar results graphically, for the three larger graphs, space on top and time to retrieve direct neighbors on the bottom. It can be seen that the space does not worsen much if we keep $k = 4$ up to a moderate level, whereas times improve consistently. A medium value, say switching to $k = 2$ at level 7, looks as a good compromise.

5.2 Comparison with other methods

We first compare graph representations that allow retrieving both direct and reverse neighbors. Figures 7 and 8 show the space/time tradeoff for retrieving direct and reverse neighbors, respectively. 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.

We compare our compact representations with the proposal in [9, Chapter 7] that computes both direct and reverse neighbors (*RePair_both*), as well as the simpler representation in [10] (as improved in [9, Chapter 6], *RePair*) that retrieves just direct neigbors. In this case we represent both the graph and its transpose, in order to achieve reverse navigation as well (*RePair × 2*). We do the same with Boldi and Vigna's technique [6] (*WebGraph*), as it also allows for direct neighbors retrieval only (we call it *WebGraph × 2* when we add both graphs). As this technique uses less space on disk than what the process needs to run, we show in *WebGraph (RAM)* the minimum space needed to run (yet we keep the best time it achieves with sufficient RAM space).

We include our alternatives 2×2, 3×3, 4×4, and *Hybrid5*, all of which use the slower solution for *rank* that uses just 5% of extra space [14], and *Hybrid37*, which uses the faster



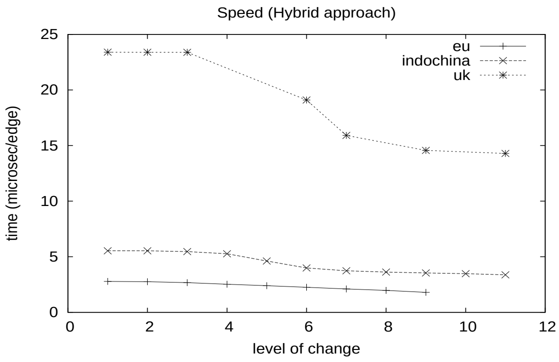


Figure 6: Space/time behavior of the hybrid approach when we vary the level where we change the value of k.

rank method that uses 37.5% extra space on top of T.

As we can see, our representations (particularly *Hybrid5* and 2×2) achieve the best compression (3.3 to 5.3 bpe, depending on the graph) among all the techniques that provide direct and reverse neighbor queries. The only alternative that gets somewhat close is *RePair_both*, but it is much slower to retrieve direct neighbors. For reverse neighbors, instead, it is an interesting alternative[1]. The alternative *Hybrid37* offers relevant tradeoffs in some cases, particularly on graph UK. Finally, *WebGraph × 2* and *RePair × 2* offer very attractive time performance, but they need significantly more space. As explained, using less space may make the difference between being able of fitting a large Web graph in main memory or not.

If, instead, we wished only to carry out forward navigation, alternatives *RePair* and *WebGraph* become preferable (smaller and faster than ours) in most cases. Figure 9, however, shows graph EU, where we still achieve significantly less space than *WebGraph*.

We also compare our proposal with the method in [2] (*Smaller*). As we do not have their code, we ran new experiments on a Pentium IV of 3.0 GHz with 4 GB of RAM, which resembles better the machine used in their experiments. We used the smaller graphs, on which they have reported experiments. Table 3 shows the space and average time needed to retrieve the whole adjacency list of a page, in milliseconds

[1]It is tempting to apply the technique over the transposed graph in order to achieve better time for direct neighbors, yet this does not work due to the statistical nature of Web graphs [9, Chapter 7].

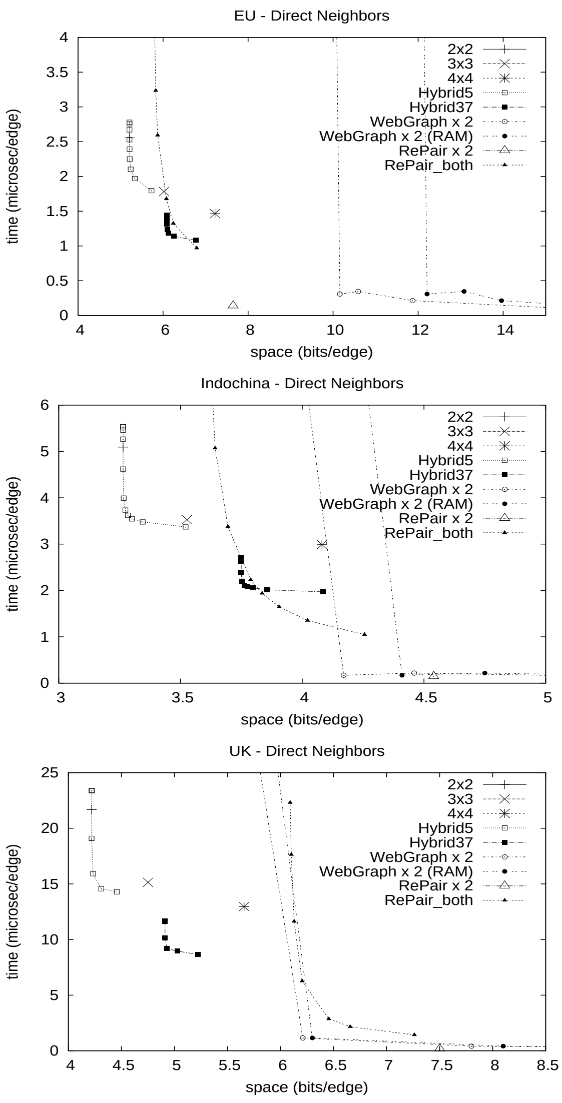


Figure 7: Space/time tradeoff to retrieve direct neighbors for different representations over graphs EU (top), Indochina (center) and UK (bottom).

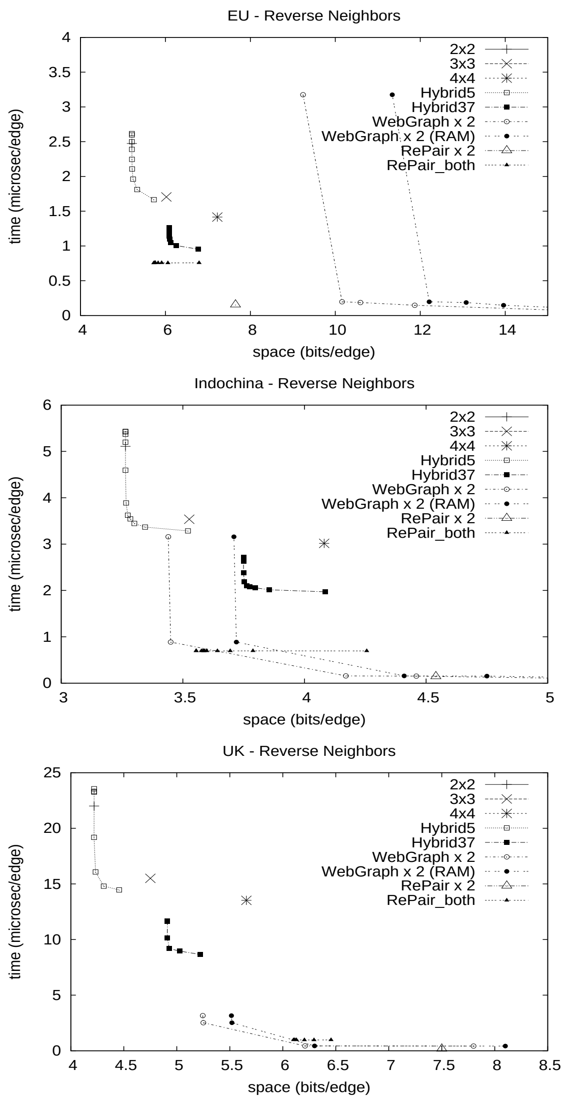


Figure 8: Space/time tradeoff to retrieve reverse neighbors for different representations over graphs EU (top), Indochina (center) and UK (bottom).

per page. As, again, their representation cannot retrieve reverse neighbors, *Smaller* × *2* is *an estimation*, obtained by multiplying their space by 2, of the space they would need to represent both the normal and transposed graphs[2].

We observe our method is orders of magnitude faster to retrieve an adjacency list, while the space is similar to *Smaller* × *2*. The difference is so large that it could be possible to be competitive even if part of our structure (e.g. L) was in secondary memory (in which case our main memory space would be similar to just *Smaller*). Yet we have not carried out this experiment.

[2]This is probably slightly overestimated, as transposed Web graphs compress slightly better than the original ones. Indeed it could be that their method can be extended to retrieve reverse neighbors using much less than twice the space. The reason is that they store the intra-domain links (which are the major part) in a way that they have to uncompress a full domain to answer direct neighbor queries, and answering reverse neighbors is probably possible with the same amount of work. They would have to duplicate only the inter-domain links, which account for a minor part of the total space. Yet, this is speculative.

6. EXTENDED FUNCTIONALITY

While alternative compressed graph representations [6, 10, 2] are limited to retrieving the direct, and sometimes the reverse, neighbors of a given page, and we have compared our technique with those in these terms, we show now that our representation allows for more sophisticated forms of retrieval than extracting direct and reverse neighbors.

First, in order to determine whether a given page p points to a given page q, most compresssed (and even some classical) graph representations have no choice but to extract all the neighbors of p (or a significant part of them) and see if q is in the set. We can answer such query in $O(\log_k n)$

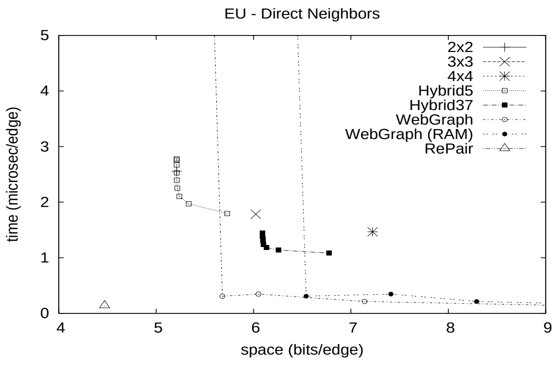


Figure 9: Space/time tradeoff for graph representations that retrieve only direct neighbors (and ours) over graph EU.

Space (bpe)	*Smaller*	*Smaller* × *2*	*Hybrid5*
CNR	1.99	3.98	4.46
EU	2.78	5.56	5.21
Time (msec/page)			
CNR	2.34		0.048
EU	28.72		0.099

Table 3: Comparison with approach *Smaller* on small graphs. The last column is an estimation.

time, by descending to exactly one child at each level of the tree. More precisely, at level ℓ 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 just as in Section 3.3. If we arrive at the last level and find a 1 at cell (p, q), then there is a link, otherwise there is not.

A second interesting operation is to find the direct neighbors of page p that are within a *range* of pages $[q_1, q_2]$ (similarly, the reverse neighbors 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. The algorithm is similar to **Direct** and **Reverse** in Section 3.3, except that we do not enter all the children $0 \le j < k$ of a row (or column), but only from $\lfloor q_1/k^{h-\ell-1} \rfloor \le j \le \lfloor q_2/k^{h-\ell-1} \rfloor$ (similarly for p_1 to p_2).

Yet a third operation of interest 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. The algorithm to solve this query indeed generalizes all of the others we have seen: extract direct neighbors of p ($p_1 = p_2 = p$, $q_1 = 0$, $q_2 = n-1$), extract reverse neighbors 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 direct neighbors of p within range $[q_1, q_2]$ ($p_1 = p_2 = p$), and find the reverse neighbors of q within range $[p_1, p_2]$ ($q_1 = q_2 = q$). Figure 10 gives the algorithm. It is invoked as **Range** $(n, p_1, p_2, q_1, q_2, 0, 0, -1)$.

The total number of nodes of level ℓ that can overlap area $[p_1, p_2] \times [q_1, q_2]$ is $(\lfloor p_2/k^{h-\ell-1} \rfloor - \lfloor p_1/k^{h-\ell-1} \rfloor + 1) \cdot (\lfloor q_2/k^{h-\ell-1} \rfloor - \lfloor q_1/k^{h-\ell-1} \rfloor + 1) \le ((p_2 - p_1 + 1)/k^{h-\ell-1} + 1) \cdot ((q_2 - q_1 + 1)/k^{h-\ell-1} + 1) = A/(k^2)^{h-\ell-1} + P/k^{h-\ell-1} + 1$, where $A = (p_2 - p_1 + 1) \cdot (q_2 - q_1 + 1)$ is the area to retrieve and $P = (p_2 - p_1 + 1) + (q_2 - q_1 + 1)$ is half the perimeter. Added

```
**Range**$(n, p_1, p_2, q_1, q_2, d_p, d_q, x)$
 1.  **If** $x \ge |T|$ **Then** // leaf
 2.     **If** $L[x - |T|] = 1$ **Then** output $(d_p, d_q)$
 3.  **Else** // internal node
 4.     **If** $x = -1$ **or** $T[x] = 1$ **Then**
 5.        $y = rank(T, x) \cdot k^2$
 6.        **For** $i = \lfloor p_1/(n/k) \rfloor \ldots \lfloor p_2/(n/k) \rfloor$ **Do**
 7.           **If** $i = \lfloor p_1/(n/k) \rfloor$ **Then** $p'_1 = p_1 \bmod (n/k)$
 8.           **Else** $p'_1 = 0$
 9.           **If** $i = \lfloor p_2/(n/k) \rfloor$ **Then** $p'_2 = p_2 \bmod (n/k)$
10.           **Else** $p'_2 = (n/k) - 1$
11.           **For** $j = \lfloor q_1/(n/k) \rfloor \ldots \lfloor q_2/(n/k) \rfloor$ **Do**
12.              **If** $j = \lfloor q_1/(n/k) \rfloor$ **Then** $q'_1 = q_1 \bmod (n/k)$
13.              **Else** $q'_1 = 0$
14.              **If** $j = \lfloor q_2/(n/k) \rfloor$ **Then** $q'_2 = q_2 \bmod (n/k)$
15.              **Else** $q'_2 = (n/k) - 1$
16.              **Range**$(n/k, p'_1, p'_2, q'_1, q'_2,$
                       $d_p + (n/k) \cdot i, d_q + (n/k) \cdot j,$
                       $y + k \cdot i + j)$
```

Figure 10: Returning the pairs (p, q) of pages p pointing to q in a range. It is assumed that n is a power of k and that $rank(T, -1) = 0$.

over all the levels $0 \le \ell < \lceil \log_k n \rceil$, the time complexity adds up to $O(A + P + \log_k n) = O(A + \log_k n)$. This gives $O(n)$ for retrieving direct and reverse neighbors (we made a finer average-case analysis in Section 3.3.1), $O(p_2 - p_1 + \log_k n)$ or $O(q_2 - q_1 + \log_k n)$ for ranges of direct or reverse neighbors, and $O(\log_k n)$ for queries on single links.

7. CONCLUSIONS

Compressed graph representations allow running graph algorithms, which are essential to extract information from the Web structure, on much larger subsets of the Web than classical graph representations, in main memory. We have introduced a compact representation for Web graphs that takes advantage of the sparseness and clustering of their adjacency matrix. Our representation is a particular type of tree, which we call the k^2-tree, that enables efficient forward and backward navigation in the graph (a few microseconds per neighbor found) within compact space (about 3 to 5 bits per link). Our experimental results show that our technique offers an attractive space/time tradeoff compared to the state of the art. Moreover, we support queries on the graph that extend the basic forward and reverse navigation.

More exhaustive experimentation and tuning is needed to exploit the full potential of our data structure, in particular regarding the space/time tradeoffs that can be obtained from the hybrid approach. We plan also to study more in depth the clustering properties of the Web graph, and how to improve them by reordering pages. In particular, locality of reference can be improved by the folklore idea of sorting the domains in reverse order, as then `aaa.bbb.com` will stay close to `zzz.bbb.com`. We also plan to work on achieving better analytical predictions of the behaviour of our technique, ideally considering the statistical laws that are known to govern Web graphs. Finally, we plan to research and experiment more in depth on the extended functionality supported by our representation, such as retrieving a range of neighbors.

We believe the structure we have introduced is of more general interest. It could be fruitful, for example, to generalize it to binary relations, such as the one relating keywords with the Web pages (or more generally, documents) where they appear. Then one could extract not only the Web pages that contain a keyword, but also the set of keywords present in a Web page, and thus have access to important summarization data without accessing the page itself. Our range searching could permit searching within subcollections or subdirectories. Our structure could become a relevant alternative to the current state of the art in this direction, e.g. [3, 9]. Another example is the representation of discrete grids of points, for computational geometry applications or geographic information systems.

8. REFERENCES

[1] M. Adler and M. Mitzenmacher. Towards compressing Web graphs. In *Proc. 11th Data Compression Conference (DCC)*, pages 203–212, 2001.

[2] Y. Asano, Y. Miyawaki, and T. Nishizeki. Efficient compression of web graphs. In *Proc. 14th Annual International Conference on Computing and Combinatorics (COCOON)*, LNCS 5092, pages 1–11, 2008.

[3] J. Barbay, M. He, I. Munro, and S. S. Rao. Succinct indexes for strings, binary relations and multi-labeled trees. In *Proc. 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 680–689, 2007.

[4] D. Benoit, E. Demaine, I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

[5] K. Bharat, A. Broder, M. Henzinger, P. Kumar, and S. Venkatasubramanian. The Connectivity Server: Fast access to linkage information on the Web. In *Proc. 7th World Wide Web Conference (WWW)*, pages 469–477, 1998.

[6] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *Proc. 13th International World Wide Web Conference (WWW)*, pages 595–601. ACM Press, 2004.

[7] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the Web. *Journal of Computer Networks*, 33(1–6):309–320, 2000. Also in *Proc. 9th World Wide Web Conference (WWW)*.

[8] H.-L. Chan, W.-K. Hon, T.-W. Lam, and K. Sadakane. Compressed indexes for dynamic text collections. *ACM Transactions on Algorithms*, 3(2):article 21, 2007.

[9] F. Claude. Compressed data structures for Web graphs. Master's thesis, Department of Computer Science, University of Chile, August 2008. Advisor: G. Navarro. Technical Report TR/DCC-2008-12. Available at `http://www.cs.uwaterloo.ca/~fclaude/docs/msc_thesis.pdf`.

[10] F. Claude and G. Navarro. A fast and compact Web graph representation. In *Proc. 14th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 4726, pages 105–116, 2007.

[11] O. Delpratt, N. Rahman, and R. Raman. Engineering the louds succinct tree representation. In *Proc. 5th International Workshop on Experimental Algorithms (WEA)*, LNCS 4007, pages 134–145, 2006.

[12] D. Donato, S. Millozzi, S. Leonardi, and P. Tsaparas. Mining the inner structure of the Web graph. In *Proc 8th Workshop on the Web and Databases (WebDB)*, pages 145–150, 2005.

[13] R. Geary, N. Rahman, R. Raman, and V. Raman. A simple optimal representation for balanced parentheses. *Theoretical Computer Science*, 368(3):231–246, 2006.

[14] R. González, S. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Poster Proc. Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*, pages 27–38, Greece, 2005. CTI Press and Ellinika Grammata.

[15] G. Jacobson. Space-efficient static trees and graphs. In *Proc. 30th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989.

[16] J. Jansson, K. Sadakane, and W.-K. Sung. Ultra-succinct representation of ordered trees. In *Proc. 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 575–584, 2007.

[17] J. Kleinberg, R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. The Web as a graph: Measurements, models, and methods. In *Proc. 5th Annual International Conference on Computing and Combinatorics (COCOON)*, LNCS 1627, pages 1–17, 1999.

[18] J. Larsson and A. Moffat. Off-line dictionary-based compression. *Proc. IEEE*, 88(11):1722–1732, 2000.

[19] I. Munro. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, LNCS v. 1180, pages 37–42, 1996.

[20] I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

[21] S. Raghavan and H. Garcia-Molina. Representing Web graphs. In *Proc. 19th International Conference on Data Engineering (ICDE)*, page 405, 2003.

[22] K. Randall, R. Stata, R. Wickremesinghe, and J. Wiener. The LINK database: Fast access to graphs of the Web. Technical Report 175, Compaq Systems Research Center, Palo Alto, CA, 2001.

[23] H. Saito, M. Toyoda, M. Kitsuregawa, and K. Aihara. A large-scale study of link spam detection by graph algorithms. In *Proc. 3rd International Workshop on Adversarial Information Retrieval on the Web (AIRWeb)*. ACM Press, 2007.

[24] T. Suel and J. Yuan. Compressing the graph structure of the Web. In *Proc. 11th Data Compression Conference (DCC)*, pages 213–222, 2001.

[25] J. Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001. Version revised at 2007 from `http://www.cs.duke.edu/~jsv/Papers/Vit.IO_survey.pdf`.