A practical succinct dynamic graph representation

Miguel E. Coimbra[a,*], Joana Hrotkó[a,*], Alexandre P. Francisco[a,*], Luís M. S. Russo[a], Guillermo de Bernardo[b,c], Susana Ladra[b], Gonzalo Navarro[d]

[a] *INESC-ID / IST, Universidade de Lisboa, Portugal*
[b] *Universidade da Coruña, CITIC, Facultade de Informática, Spain*
[c] *Enxenio SL, Spain*
[d] *IMFD, Dept. of Computer Science, University of Chile, Chile*

Abstract

We address the problem of representing dynamic graphs using k^2-trees. The k^2-tree data structure is one of the succinct data structures proposed for representing static graphs, and binary relations in general. It relies on compact representations of bit vectors. Hence, by relying on compact representations of dynamic bit vectors, we can also represent dynamic graphs. However, this approach suffers of a well known bottleneck in compressed dynamic indexing. In this paper we present a k^2-tree based implementation which follows instead the ideas by Munro *et al.* (PODS 2015) to circumvent this bottleneck. We present two dynamic graph k^2-tree implementations, one as a standalone implementation and another as a `C++` library. The library includes efficient edge and neighbourhood iterators, as well as some illustrative algorithms. Our experimental results show that these implementations are competitive in practice.

Keywords: compact graph representations, dynamic graphs, k^2-tree, graph library

*Corresponding authors.
Email addresses: `miguel.e.coimbra@tecnico.ulisboa.pt` (Miguel E. Coimbra), `joana.hrotko@tecnico.ulisboa.pt` (Joana Hrotkó), `aplf@tecnico.ulisboa.pt` (Alexandre P. Francisco), `luis.russo@tecnico.ulisboa.pt` (Luís M. S. Russo), `gdebernardo@udc.es` (Guillermo de Bernardo), `sladra@udc.es` (Susana Ladra), `gnavarro@dcc.uchile.cl` (Gonzalo Navarro)

1. Introduction

Graphs are ubiquitous among many complex systems, where we find large and dynamic complex networks. It is therefore important to be able not only to store such graphs in compact form, but also to update and query them efficiently. Reusable graph libraries are then also indispensable to make these approaches and techniques readily available.

Most succinct data structures for representing graphs are however static [1, 2]. We focus on one such static succinct graph representation, called k^2-tree. The k^2-tree [2] was originally proposed for representing Web graphs, but was later proven efficient for representing other kinds of graphs and binary relations [3]. It consists of a succinct representation of the adjacency matrix of the graph that exploits large empty areas of the matrix, and obtains very compact representations while efficiently supporting neighbour queries, forward and backward navigation, and range searches.

A dynamic version of k^2-trees already exists [4]. Like many dynamization approaches of compact data structures, however, this introduces a significant slowdown, by an almost logarithmic factor, for all the queries.

In this paper we adopt the ideas proposed by Munro *et al.* [5] to represent dynamic graphs through collections of static and compact graph representations, and apply them to the static k^2-tree data structure, so that it supports edge insertions and removals. A good aspect of this dynamization is that query times are slowed down by a constant factor only. Further, the edge insertion time is almost the same as the average construction time per edge of static k^2-trees, and edge deletion time is even lower. In exchange, insertion and deletion times are amortized, not worst-case, as the structure undergoes periodic reconstructions.

We present a standalone implementation of these ideas, `sdk2tree`[1], as well a reusable one, `sdk2sdsl`[2], that is based on the `sdsl-lite` data structure library[3].

We briefly describe the original static k^2-trees in Section 2. In Section 3, we explain the technique [5] employed to implement dynamic graphs using collections of static k^2-trees and the details of our library implementation `sdk2sdsl`. We present extensive experimental analysis in Section 4 and final

[1] `https://github.com/aplf/sdk2tree`

[2] `https://github.com/joo95h/dynamic_k2tree`

[3] `https://github.com/simongog/sdsl-lite`

remarks in Section 5.

This paper is an extended version of a paper presented in the Data Compression Conference (DCC) [6]. It includes not only more details concerning the proposed construction, but also a graph library, including efficient edge and neighbourhood iterators. The experimental evaluation was also extended.

2. The static k^2-tree

Let $G = (V, E)$ be a graph where V is the set of vertices, of size n, and $E \subseteq V \times V$ is the set of edges, of size m. The k^2-tree data structure provides a static succinct representation of G [2]. At a high level, this data structure corresponds to an adjacency matrix representation, where a bit set to 1 indicates the existence of an edge and a bit set to 0 its absence. To reduce the space requirements for sparse graphs, a hierarchical decomposition of the matrix is used, where a sub-division consisting only of zeros is represented by a single 0 bit.

More concretely, the k^2-tree can be conceptually defined as a non-balanced k^2-ary tree that represents the recursive partition of the adjacency matrix into $k \times k$ submatrices. The root node contains k^2 children, each of them corresponding to one submatrix and sorted following a Z-order. The nodes of the tree store just one bit telling if the submatrix is non-empty (1) or it is all zeroes (0). Then, the non-empty submatrices are subdivided again until reaching an empty submatrix, or until no more subdivision is possible; thus, bits at last level of the tree correspond to cell values of the original adjacency matrix. The resulting tree is thus of height $\lceil \log_{k^2} n^2 \rceil = \lceil \log_k n \rceil$.

An example of this tree-shaped representation is shown in Figure 1. This conceptual tree is stored in one single bitmap following a level-wise traversal of the tree (i.e., concatenating the 4 bitmaps of the figure). Queries over the graph can be solved efficiently by performing top-down traversals over the tree representation. Those traversals are efficiently implemented thanks to the use of fast rank operations [3] over the bitmap.

The maximum length of this bitmap is $k^2 m(\log_{k^2} \frac{n^2}{m} + \mathcal{O}(1))$. A sub-linear number of extra bits are needed to enable constant-time rank operations on the bitmaps. Testing the existence of an edge is done in $\mathcal{O}(\log_k n)$ time by traversing the k^2 down to the desired matrix cell, until an empty submatrix (a 0) is found or we reach the 1 of the cell in the last level. Obtaining the neighbours of a node is done in $\mathcal{O}(n)$ worst-case time by reaching all the

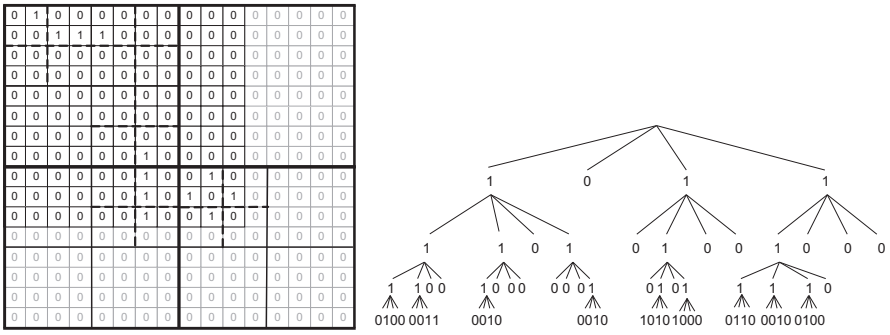

Figure 1: Example of adjacency matrix (left) and its corresponding k^2-tree with $k = 2$ (right).

cells in the corresponding matrix row, via entering all the children of each node that intersect the row; the reverse neighbours are obtained similarly by extracting the corresponding matrix column. In this paper, k is a fixed value across the trees and remains constant along time.

3. From static k^2-trees to dynamic graphs

The main idea to represent G dynamically, supporting edge insertions and deletions, as well as listing the neighbours of a given vertex v, is to use a collection of static edge sets $\mathcal{C} = \{E_0, \ldots, E_r\}$. Each static edge set E_i is then represented using a static k^2-tree, except E_0 which is represented through a dynamic and uncompressed adjacency list. Figure 2 depicts a link query over the different E_i sets of the data structure.

Let m_i be the number of edges in each set E_i. As discussed by Munro *et al.* [5], we must control both the number of edges m_i in each set E_i and the number r of such sets to achieve the optimal amortized cost for each operation. The first set (E_0) contains at most $m/\log^2 n$ elements. In general we require that m_i is at most $m/\log^{2-i\varepsilon} n$, for some constant $\varepsilon > 0$. We must also have that $r \leq 2/\varepsilon$, so when ε is a fixed constant so is r. For example when $\varepsilon = 1/4$ we get that r is at most $2/(1/4) = 8$. Hence the maximum number of edges per static set follows a geometric progression. Each set E_i is static (except for E_0) and has a maximum allowed size $m/\log^{2-i\varepsilon}$. Whenever we reach the maximum size for E_0 (overflow), we find a set E_j, with $i < j \leq r$ such that $\sum_{\ell=0}^{j} m_\ell \leq m/\log^{2-j\varepsilon} n$ and (re)build E_j with all edges in it and in the previous sets, and reset all previous sets to empty. By construction, E_j

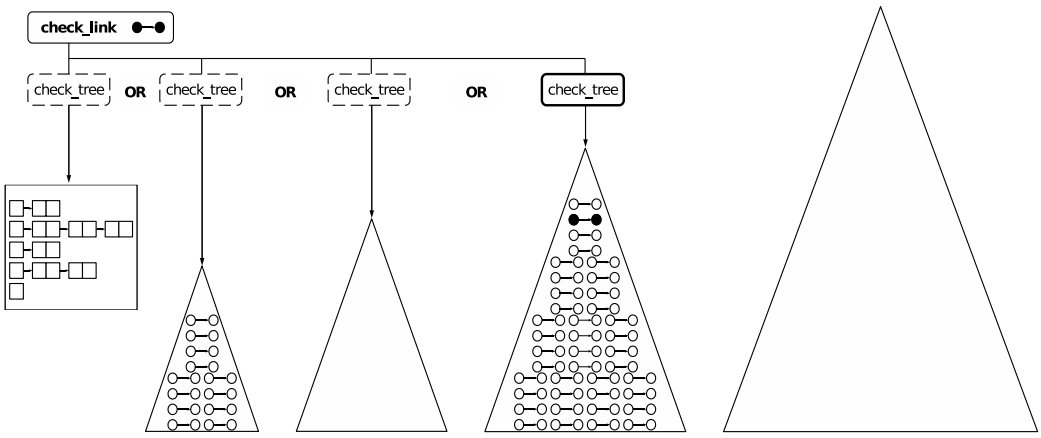

Figure 2: `check_link` query going through E_i sets to find an edge (noted in black).

has a maximum capacity enabling it to store the content of sets E_0 through E_{j-1}. We detail this process below.

3.1. Space

Let us analyze the required space to represent the data structure. The set E_0 is represented in an uncompressed adjacency list coupled with a hash table to allow answering queries on edge existence in constant expected time. This requires $\mathcal{O}(m_0 \log n)$ bits, where $m_0 \leq m/\log^2 n$ is the number of edges in E_0. Each set E_i, for $1 \leq i \leq r$, is represented with a static k^2-tree requiring $k^2 m_i \left(\log_{k^2}(n^2/m_i) + \mathcal{O}(1)\right)$ bits (plus sublinear-order terms to support rank), where $m_i \leq m/\log^{2-i\varepsilon} n$. Hence, overall, the space required is

$$\mathcal{O}(m_0 \log n) + \sum_{i=1}^{r} k^2 m_i \left(\log_{k^2}(n^2/m_i) + \mathcal{O}(1)\right) \ (1 + o(1)) \tag{1}$$

bits. The first term in Equation 1 is sublinear, $\mathcal{O}\left((m/\log^2 n)\log n\right) = \mathcal{O}\left(m/\log n\right)$. We now bound the main part of second term as follows, exploiting the fact that the formula is monotonic on every m_i:

$$\begin{aligned} \sum_{i=1}^{r} k^2 m_i \log_{k^2} \frac{n^2}{m_i} \ &\leq \ k^2 \sum_{i=1}^{r} \frac{m}{\log^{2-i\varepsilon} n} \log_{k^2} \left(\frac{n^2}{m} \log^{2-i\varepsilon} n\right) \\ &= \ \frac{k^2 m}{\log^2 n} \ \sum_{i=1}^{r} \log^{i\varepsilon} n \left(\log_{k^2} \frac{n^2}{m} + (2 - i\varepsilon) \log_{k^2} \log n\right). \end{aligned} \tag{2}$$

We can set $r = 2/\varepsilon$ because the sum is monotonic on r. Then, because

$$\sum_{i=1}^{r} \log^{i\varepsilon} n \;=\; \log^{r\varepsilon} n \left(1 + \mathcal{O}(\log^{-\varepsilon} n)\right) \;=\; \log^2 n \left(1 + \mathcal{O}(\log^{-\varepsilon} n)\right), \text{ and}$$

$$\sum_{i=1}^{r} i \log^{i\varepsilon} n \;=\; r \log^{r\varepsilon} n \left(1 + \mathcal{O}(\log^{-\varepsilon} n)\right) \;=\; r \log^2 n \left(1 + \mathcal{O}(\log^{-\varepsilon} n)\right),$$

Equation (2) is

$$k^2 m \left(\log_{k^2} \frac{n^2}{m} \left(1 + \mathcal{O}(\log^{-\varepsilon} n)\right) + (2 - r\varepsilon + \mathcal{O}(\log^{-\varepsilon} n)) \log_{k^2} \log n\right)$$

$$= \; k^2 m \log_{k^2}(n^2/m)(1 + o(1)).$$

The whole Equation (1), since $\sum_{i=1}^{r} m_i \leq m$, is then upper bounded by

$$k^2 m(\log_{k^2}(n^2/m) + \mathcal{O}(1))(1 + o(1)),$$

which asymptotically coincides with the space of the static k^2-tree representation, even considering the sublinear extra space to support rank operations.

3.2. Insertion, deletion and queries

We rely on efficient set operations over k^2-trees [7]. Given C and C' represented as two k^2-trees, we are able to compute k^2-trees representing $C \cup C'$, $C \cap C'$ and $C \setminus C'$ in linear time on the size $|C|$ and $|C'|$ of the representations. Moreover these operations are done without decompressing C and C', with only some negligible extra space being used.

Insertion works as follows. Given an edge (u, v), we check if it occurs in all sets in the collection in $\mathcal{O}(\log_k n)$ time, as described in Section 2. We only insert it if it does not exist in any set. While checking for the existence of (u, v) in sets $E_1, \ldots, E_r$, if we reach a leaf of a k^2-tree at depth $\lceil \log_k n \rceil$ and the bit is set to 0, then we set it to 1 and we are done. We note that this departs from the construction by Munro *et al.* [5], opportunistically inserting (u, v) in our data structure in $\mathcal{O}(\log_k n)$ time. Otherwise, if (u, v) does not exist in the collection and opportunistic insertion is not possible (i.e., we never reached a leaf while checking for (u, v) in the collection): if $|E_0| < m_0$, then just insert (u, v) in E_0 and we are done; otherwise, build a k^2-tree for E_0, find $0 < j \leq r$ such that $\sum_{i=0}^{j} m_i \leq m/\log^{2-j\varepsilon} n$, and rebuild E_j with

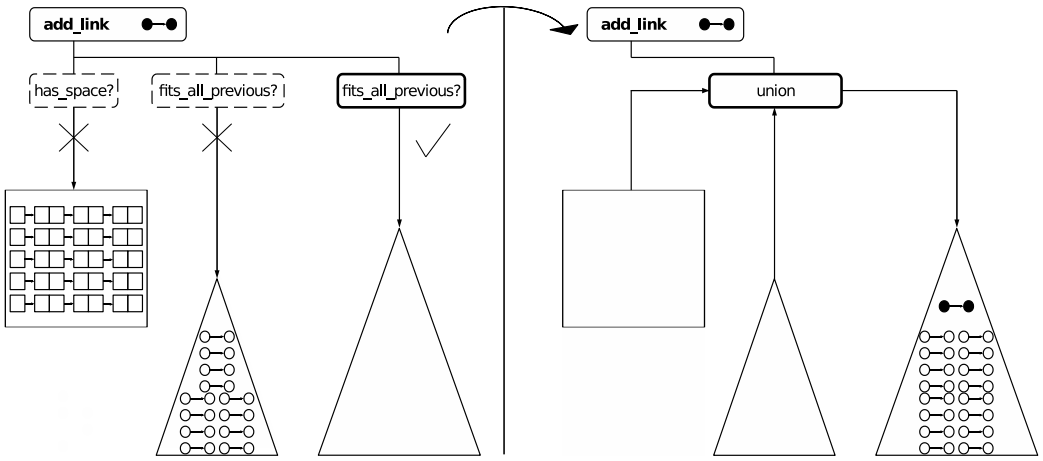

Figure 3: `add_link` searching for j such that E_j has enough space to hold all edges of sets $E_i, i < j$ because E_0 was full.

all edges in $E_0, \ldots, E_j$ by successive unions of k^2-trees. Figure 3 illustrates the process.

If opportunistic insertion occurs, then it takes $\mathcal{O}((\log_k n)/\varepsilon)$. If not and $|E_0| < m_0$, then insertion takes constant expected time since we are relying on an adjacency list coupled with a hash table to maintain adjacencies, as described before. Otherwise, we need to build a k^2-tree for E_0 and find some E_j to accommodate all previous collections E_i, for $0 \leq i \leq j$. Note that the construction of the k^2-tree for E_0 takes $\mathcal{O}(m_0 \log_k n)$ time [2], and the pairwise union of at most j k^2-trees representing collections $E_0 \ldots E_{j-1}$ takes $\mathcal{O}(m_j \log_k n)$ time, using only the required space to store a k^2-tree representing E_j. The amortized analysis of the insertion cost follows the argument presented by Munro *et al.* [5] for the general case. Either E_j is new and m has at least doubled, in which case the amortized cost is $\mathcal{O}(\log_k n)$ per edge insertion, or E_j is not new and we are adding to it all edges in collections $E_0, \ldots, E_{j-1}$. In this last case the building cost can be charged to the new edges added to E_j, which are at least $m/\log^{2-(j-1)\varepsilon} \geq m_j/\log^{\varepsilon} n$. Therefore, the amortized cost of inserting an edge in E_j is $\mathcal{O}(\log_k n \log^{\varepsilon} n)$ and, since each edge can be moved once to each E_j, with $0 < j \leq r = \lfloor 2/\varepsilon \rfloor$, the amortized cost of inserting an edge is $\mathcal{O}((1/\varepsilon) \log_k n \log^{\varepsilon} n)$. This is then the overall amortized cost of inserting an edge.

Deletion works as follows. Given an edge $(u, v) \in E$, if $(u, v) \in E_0$, then just remove it and we are done; otherwise, find $0 < j \leq r$ such that

$(u, v) \in E_j$ and, if there is such j, set to zero the bit corresponding to its leaf in E_j k^2-tree. Note that there is a bit set to 1 for each edge in a k^2-tree. While the bits of edges leading to internal nodes are used to navigate toward other nodes, the bits of edges leading to leaves do not affect the navigation. We exploit that fact by setting to zero that final edge, without having to restructure the k^2-tree in any way. This is somehow similar to how Munro et al. [5] mark deleted bits, but we do not need to allocate extra bits for marking the deleted nodes; we are able to mark them as deleted within the same data structure.

Deleting an edge in E_0 takes constant expected time. Checking and deleting an edge in our collections takes $\mathcal{O}((\log_k n)/\varepsilon)$, since checking if an edge exists in a given k^2-tree takes $\mathcal{O}(\log_k n)$ [2], and we might have to look in each collection E_i, with $0 < i \leq r = \lceil 2/\varepsilon \rceil$. Once an edge is found, setting the corresponding bit to 0 in the static k^2-tree takes constant time. We must rebuild the structure periodically in order to maintain the time and space bounds with respect to the actual number of edges in the graph. Whenever we delete an edge, we increase the number m' of deleted edges, and if $m' > m/\log\log n$, we rebuild $\mathcal{C}$. The full rebuild after $m/\log\log n$ edges are deleted costs $\mathcal{O}(m \log_k n)$, *i.e.*, it has an amortized cost of $\mathcal{O}(\log_k n \log\log n)$ per deleted edge. Overall, deleting an edge has then an amortized cost of $\mathcal{O}((1/\varepsilon + \log\log n) \log_k n)$.

Querying (checking) for the presence of an edge works just as in k^2-trees with the difference that we need to query all sets in the collection. Therefore, the querying cost increases by a factor of $\mathcal{O}(1/\varepsilon)$.

3.3. Graph library

The library proposed in this paper exposes an API supporting also edge and neighbourhood iterators. This API was built having in mind an easy and familiar interface, compared to other libraries such as: `SNAP`[4], `igraph` [8], among others. And the library was built after the `sdk2tree` project, but the underlying k^2-tree implementation is based on the `sdsl-lite` k^2-tree implementation, which uses static bit vectors, in `C++`.

The two types of iterators present a similar interface: `edge_begin`, `edge_end` for edges and `neighbour_begin(x)`, `neighbour_end()` for neighbours. The neighbour iterator receives the node whose neighbourhood is desired.

[4] `http://snap.stanford.edu`

The iterators may be used as other iterators in `C++` and follow its iterator pattern. If there are no edges to iterate then `edge_begin()`=`edge_end()`. The iterators do not return edges in any particular order, since they first iterate over the E_0 container and then over each E_i k^2-tree.

3.3.1. Edge iterator

As previously mentioned, the edge iterator iterates over the container E_0 and then over the k^2-tree for each E_i. In the uncompressed container E_0, it takes linear time to retrieve all its edges. However, for each k^2-tree, it relies on a k^2-tree edge iterator and it takes time proportional to the size of the k^2-tree. This iterator is implemented by saving in a queue all states where the search was still not finished in a depth-first approach over the k^2-tree. Thus, this queue has at most size $\mathcal{O}(\lfloor \log(V)/\log(k) \rfloor)$ which is the maximum level of the k^2-tree. Iterating over edges in a k^2-tree is performed then by visiting internal nodes and, if there are any children, then we check in all the k^2 children of that node; otherwise we backtrack. When we reach the last level, we check if the bit position is 1, which means we have found an edge and we return it.

3.3.2. Neighbourhood iterator

The neighbourhood iterator is very similar to the edge iterator in the sense that we keep a queue of the states from where we last evaluated each node of the tree. If there are neighbours of node x in the first container E_0, it iterates over them first. Once the uncompressed container is iterated, it goes to the k^2-tree collections. Similarly to the edge iterator, it follows a depth-first search keeping a queue with the incomplete searched states. The neighbour iterator follows the same algorithm from the neighbour-listing operation from the k^2-tree, however it saves the state from all the incomplete searched states. The running time for listing all neighbours of a given vertex is the same of the neighbour-listing operation.

3.4. Comparison with other constructions

Given a graph G, for a fixed ε, the presented data structure uses essentially the same space as a static k^2-tree, and supports insertions and deletions in $\mathcal{O}(\log_k n \log^{\varepsilon} n)$ and $\mathcal{O}(\log_k n \log\log n)$ amortized time, respectively. The implementation of dynamic k^2-trees using dynamic bit vectors [4] requires a small space overhead, and it supports insertions and deletions in

$\mathcal{O}(\log_k n \log n)$ time, which implies a slowdown factor of $\Theta(\log^{1-\varepsilon} n)$ with respect to the proposed data structure.

Edge queries over the proposed data structure take the same time as in static k^2-trees. Although dynamic k^2-trees using dynamic bit vectors [4] work similarly to static k^2-trees, they run on dynamic bit vectors, thereby having a slowdown of $\Omega(\log n / \log \log n)$ [3, Chapter 12].

We also compare our approach with a new representation, k^2-tries, proposed recently [9]. This data structure uses $\mathcal{O}(m \log(n^2/m) + m \log k)$ bits, and supports edge queries in $\mathcal{O}(\log_k n)$ time and updates in $\mathcal{O}(\log_k n)$ amortized time. The implementation provided by the k^2-tries authors supports only edge additions and queries, with slightly worse time complexities.

4. Experimental analysis

In the conference version of this work [6], we presented `sdk2tree`, our dynamic k^2-tree `C` implementation based on the techniques proposed by Ian Munro *et al.* [5]. We now also introduce a `C++` library version named `sdk2sdsl` which is based on the `sdsl-lite` data structure library. We present an experimental analysis comparing different implementations: our new `sdk2sdsl` library version; our previous dynamic graph `sdk2tree` implementation [6]; the dynamic graph `dk2tree` based on dynamic bit vectors [4]; two dynamic graph implementations (differing only on the parametrization to trading compression for speed) `k2trie{1,2}` based on dynamic tries [9]; the original static bit vector implementation `k2tree` [2]. We make available the source code for all implementations as well as usage instructions at `https://github.com/aplf/sdk2tree`. Our new `sdk2sdsl` implementation was written in `C++` and the others are in `C`, with every implementation single-threaded and compiled with `gcc 7.5.0` using the `-O3` optimization flag. Experiments were performed on an 8-core AMD Ryzen 7 2700X Eight-Core Processor @ 2.04GHz machine with 32K L1d cache, 64K L1i cache, 512KB L2 cache, 8192K L3 cache and system memory of 64 GB RAM. We implemented a common interface to test each implementation. All dynamic data structures `dk2tree`, `sdk2tree`, `sdk2sdsl` and `k2trie{1,2}` are initialized empty. The static `k2tree` is initialized by reading the whole graph from secondary storage. Once initialized, the interface starts a main loop which reads instructions from `stdin` representing all supported edge operations, with additions and deletions not available in `k2tree`, and `k2trie{1,2}` supporting only edge additions and queries. We also implemented and tested known graph

algorithms on our `sdk2sdsl` implementation: breadth-first search (BFS), depth-first search (DFS), global clustering coefficient and variants of triangle counting.

4.1. Datasets and methodology

We use both real and synthetic datasets. In Table 1 we identify the datasets and their properties. For each dataset, we present its vertex and edge counts written as $|V|$ and $|E|$, respectively, and bits per edge (after serialization) for each implementation.

Real-world graphs were obtained from the Laboratory of Web Algorithmics[5] [1, 10]. Synthetic datasets were generated from the partial duplication model [11]. Although the abstraction of real networks captured by the partial duplication model, and other generalizations, is rather simple, the global statistical properties of, for instance, biological networks and their topologies can be well represented by this kind of model [12]. We generated random graphs with selection probability $p = 0.5$, which is within the range of interesting selection probabilities [11]. The number of edges for those graphs is approximately 25 times the number of vertices.

We should note that bits per edge for real datasets in Table 1 are affected by the natural order of vertices, given in that case by URL lexicographic order, which favours Web graph compressibility. If ids were randomly assigned to vertices, then the bits per edge would be similar to those observed for random synthetic graphs.

We consider four major operations: edge additions, removals, querying/checking and vertex neighbourhood listing. Elapsed time was measured using the `clock()` function[6]. Each time and memory result is the average of 5 individual executions. Although the `k2tree` implementation does not support additions, we include it in the comparison. For that we build a `k2tree` for each dataset and divide the time it takes by the number of edges, obtaining the average construction time per edge. This allowed us to evaluate the overhead introduced by dynamic data structures. The removal operation is compared between `sdk2tree`, `sdk2sdsl` and `dk2tree`. This operation was evaluated by adding all edges and removing a sample of 50% of them. All implementations except the one based on dynamic tries were directly compared

[5] `http://law.di.unimi.it/datasets.php`

[6] `http://man7.org/linux/man-pages/man3/clock.3.html`

Table 1: Bit/edge ratio (post-serialization) is presented for each data structure. First four datasets were synthetically generated using a duplication model. Last four datasets are real-world Web graphs made available by the Laboratory for Web Algorithmics (LAW) [1, 10] (`uk-2007-05` is actually `uk-2007-05-100000` in the LAW website).

Dataset	$\lvert V\rvert$ (M)	$\lvert E\rvert$ (M)	`k2tree` (bit/edge)	`dk2tree` (bit/edge)	`sdk2tree` (bit/edge)	`sdk2sdsl` (bit/edge)	`k2trie1` (bit/edge)	`k2trie2` (bit/edge)
`dm50K`	0.05	1.11	21.10	23.64	21.26	25.26	43.16	298.99
`dm100K`	0.10	2.59	22.66	25.27	22.76	27.16	47.31	257.61
`dm500K`	0.50	11.98	27.87	30.85	27.97	33.31	57.92	187.91
`dm1M`	1.00	27.42	29.48	32.63	29.49	35.33	58.78	132.92
`uk-2007-05`	0.10	3.05	2.98	3.39	3.16	3.63	5.62	11.11
`in-2004`	1.38	16.92	2.99	3.40	3.14	3.64	3.90	6.97
`uk-2014-host`	4.77	50.83	9.47	10.55	9.58	11.42	13.07	21.88
`indochina-2004`	7.42	194.11	2.46	2.79	2.59	3.00	2.88	4.91
`eu-2015-host`	11.26	386.92	5.61	6.26	5.71	6.74	7.02	11.64

for the listing operation. After adding all edges, we evaluated this operation by asking for the neighbourhoods of a sample of 50% of the vertices. We measure for each implementation the average time per individual operation, the maximum resident set size (memory peak was obtained with `GNU time`[7]), and the disk space taken by the serialization of data structures.

4.2. Cost analysis

Let us analyze the cost of each operation over the different datasets and for the different implementations. Figure 4 shows the average running time for adding an edge. As mentioned before, we include `k2tree` in this comparison to observe the slowdown introduced by dynamic data structures. As expected, dynamic implementations take in general more time per add operation than `k2tree`. As expected also from the theoretical analysis, the add operation on `sdk2tree` is faster than on `dk2tree`, in particular for real Web graphs. The library version `sdk2sdsl` is faster or as fast as `sdk2tree` for this operation.

Figure 5 shows the average running time for removing an edge. For both dataset types, `dk2tree` was slower than others. The `sdk2tree` and `sdk2sdsl` implementations achieved close execution times and similar behavior among datasets, with the library implementation `sdk2sdsl` being faster. We note that costs seem to correlate well with the predicted bounds.

[7] `https://www.gnu.org/software/time/`

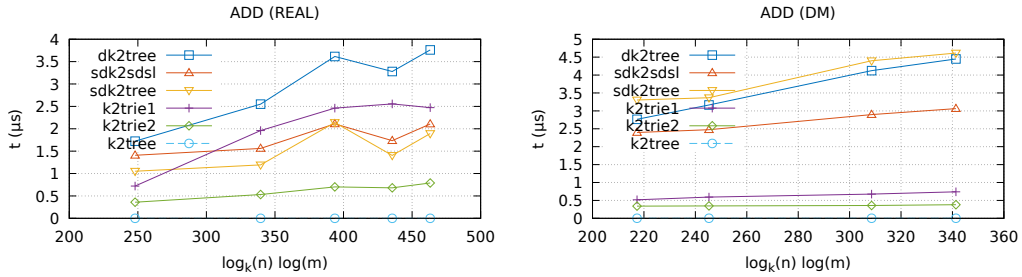

Figure 4: Average time taken for adding an edge in real Web graphs and in synthetic graphs (generated from a duplication model), respectively.

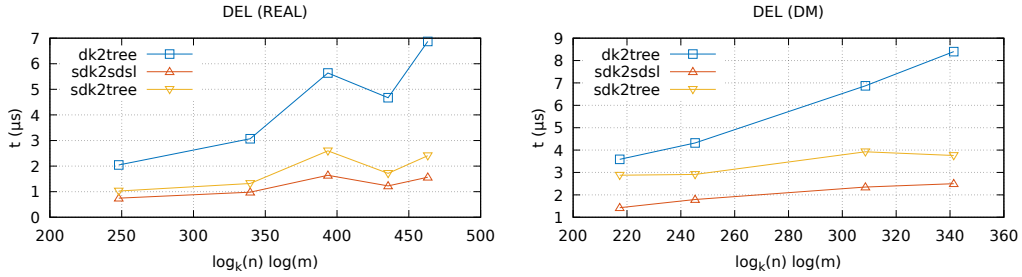

Figure 5: Average time taken for deleting an edge in real Web graphs and in synthetic graphs (generated from a duplication model), respectively.

Figures 6 and 7 show the average running time for listing vertex neighbourhoods and querying/checking edges. Across all datasets, `sdk2tree` was faster than `dk2tree` and on-par with `k2tree` and `k2trie{1,2}`. In the case of listing, we are plotting against $\mathcal{O}(\sqrt{m})$. The results on the duplication model show good correlation with previous average-case analyses [2]. We note that, in the worst case, it takes $\mathcal{O}(n)$ time to obtain the neighbours of a vertex with the k^2-tree, even if there are none to report. Those average and worst-case bounds are also valid for `sdk2tree` and `dk2tree` as discussed previously in the theoretical analysis. `dk2tree` (dynamic bit vectors) was slower than `sdk2tree`, which matched the static `k2tree` implementation. Our `sdk2sdsl` library version was consistently the fastest for the listing operation. For the edge query operation, `dk2tree` was consistently the slowest implementation, with the others coming very close.

Let us now analyze how much memory is used by each implementation. In this analysis we consider resident memory while we are performing op-

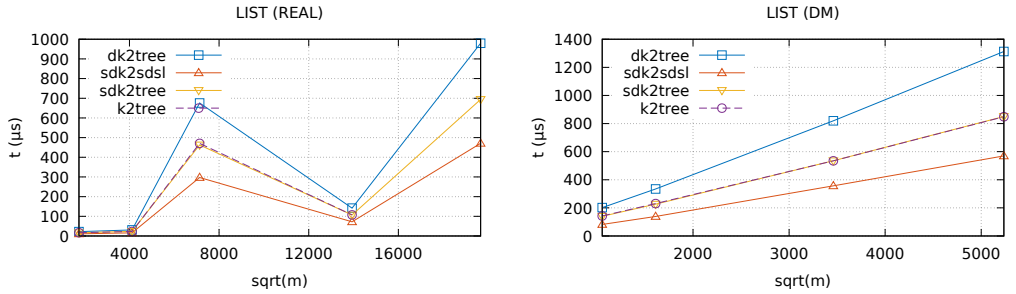

Figure 6: Average time taken for listing neighbours of random vertices in real Web graphs and in synthetic graphs (generated from a duplication model), respectively.

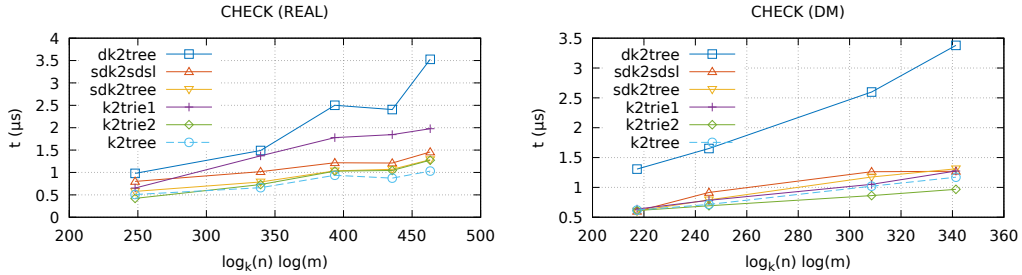

Figure 7: Average time taken for querying edges in real Web graphs and in synthetic graphs (generated from a duplication model), respectively.

erations. For the space that each data structure takes once serialized on secondary storage (i.e., the tree stored on disk as a sequence of nodes), we refer the reader to Table 1. We note that our `sdk2sdsl` library implementation serialized format has a higher ($\approx 20\%$) bits/edge ratio compared to `sdk2tree`. This seems to be related to using a 64-bit index for the data structures.

Figure 8 shows the maximum resident memory while adding edges in dynamic implementations. We can observe that `sdk2tree` requires more memory than `dk2tree`, although the growth rate is similar. This can look unexpected given the theoretical bounds derived previously, but we must recall that we are periodically merging together static collections in the `sdk2tree` implementation. The `sdk2sdsl` implementation followed the same pattern as `sdk2tree`, albeit consuming more memory than `sdk2tree`. Note that we use 64 bit integers in `sdk2sdsl` and 32 bit integers in `sdk2tree`.

Figure 9 shows the maximum resident memory while removing edges.

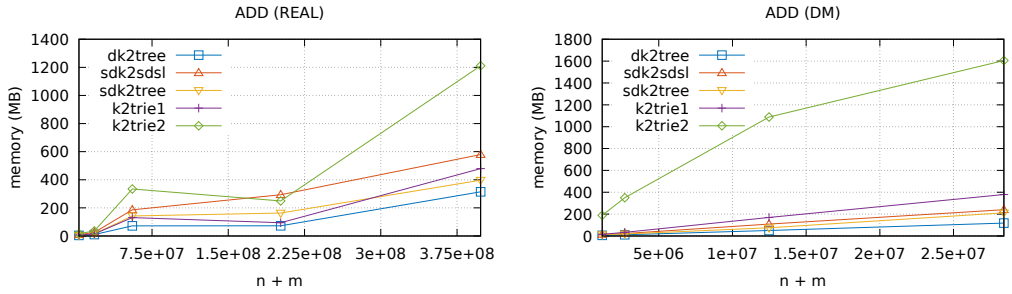

Figure 8: Maximum resident memory while adding edges in real Web graphs and in synthetic graphs (generated from a duplication model), respectively.

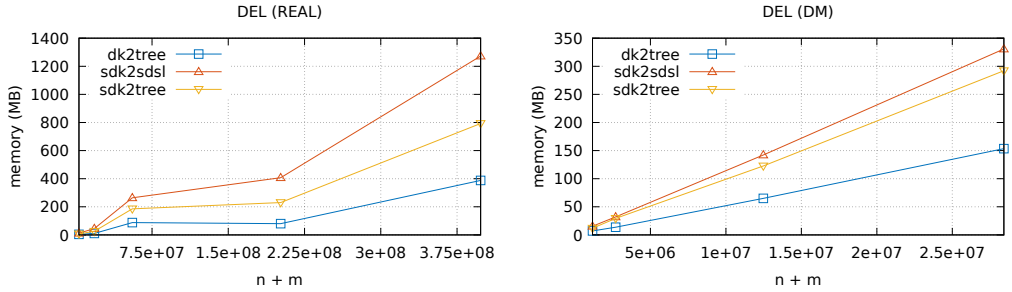

Figure 9: Maximum resident memory while deleting edges in real Web graphs and in synthetic graphs (generated from a duplication model), respectively.

Since we are adding all edges before removing about 50% of them, the memory requirements for `sdk2tree` are exactly the same as in Figure 8. This also means that the edge removal operation does not increase the space requirements in this implementation. `sdk2sdsl` consumed more memory than `sdk2tree`, with the memory requirements for `dk2tree` being the lowest on this operation.

Figure 10 shows the maximum resident memory while adding edges and listing vertex neighbourhoods. Since we are adding all edges as before, the memory requirements for `sdk2tree`, `sdk2sdsl` and `dk2tree` are identical to those observed in Figures 8 and 9. We include now also the static `k2tree` in our analysis. We should note however that once constructed, `k2tree` requires much less space as shown in Table 1. For instance, for the dataset `dm100K`, `k2tree` had a peak resident memory footprint of around 503.11 MB during construction, while its k^2-tree structure stored on disk uses 22.66 bits per edge, i.e, a total of 7.01 MB. Although we are using the exact same

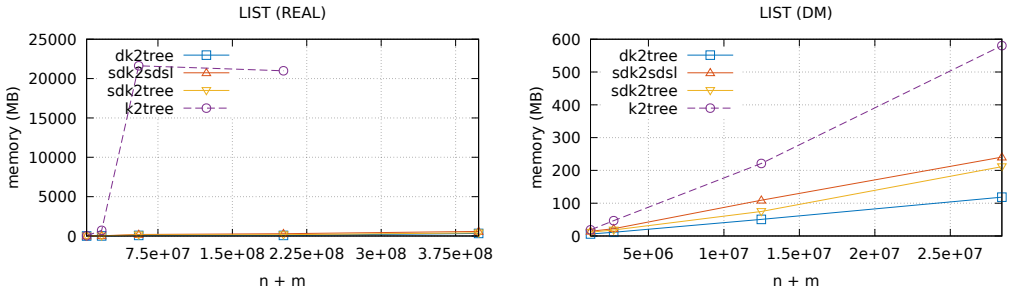

Figure 10: Maximum resident memory while listing neighbours of random vertices in real Web graphs and in synthetic graphs (generated from a duplication model), respectively.

implementation of k^2-trees for representing the static collections within our `sdk2tree` and `sdk2sdsl` implementations, we do not observe such a high memory footprint while adding edges in our implementations. This highlights the fact that we are merging those collections without decompressing them as mentioned before.

4.3. Graph library performance

We implemented some well known graph algorithms over `sdk2sdsl`, for which we compare consumed memory and execution time against expected theoretical results. For each algorithm, we present in Table 2 the running time and peak resident memory usage. Each cell holds the ratio of observed value to corresponding theoretical complexity. For example, the value of the cell of the first row and first column on the top represents the execution time (nanoseconds) of the `sdk2sdsl` breadth-first search algorithm (applied to dataset `dm50K`) divided by its theoretical temporal complexity of $\mathcal{O}(n\sqrt{m}+m)$. We omit dataset `indochina-2004` from the graph algorithm tests for `sdk2sdsl` as its topology does not allow for an adequate assessment of algorithms expected efficiency.

The first column of Table 2 shows the behavior of breadth-first search (BFS). For the time ratio, the implementation is such that the observed execution times increase by small amounts compared to the growing dataset sizes, with the peak resident memory values being more intimately connected to the topology of the datasets. Note the $\sqrt{m}$ due to the cost of listing of neighbourhoods.

The second column of Table 2 shows the behavior of depth-first search (DFS). It has a behavior similar to BFS for all dataset graph types (for both

Table 2: Ratios between observed values and corresponding theoretical graph algorithm complexities. The top part is the execution time ratio (nanoseconds) and the bottom part is the peak resident memory ratio (bytes).

Time ratio (ns)	BFS $n\sqrt{m}+m$	DFS $n\sqrt{m}+m$	CC $m\sqrt{m}$	CT (hash) $m\sqrt{m}$	CT (neighbour) $m\sqrt{m}\log_k n \log m$
dm50K	66.2884	67.3630	6.4365	0.0474	0.4566
dm100K	74.2812	73.8288	5.5574	0.0386	0.4350
dm500K	82.8276	84.9108	5.7892	0.0214	0.3033
dm1M	88.1807	91.0187	5.2030	0.0203	0.2773
uk-2007-05	5.0637	5.0362	0.8456	0.0173	N/A
in-2004	2.7018	2.6811	0.7807	0.0039	N/A
uk-2014-host	13.5329	13.2120	2.3418	0.0745	N/A
eu-2015-host	10.3745	10.1891	0.4817	0.0007	N/A
Memory ratio (B)	BFS $n+m$	DFS $n+m$	CC $n+m$	CT (hash) $n+m$	CT (neighbour) $n+m$
dm50K	16.465	16.526	78.771	75.848	14.581
dm100K	11.015	11.017	74.837	72.110	9.231
dm500K	12.744	12.748	72.662	70.569	10.561
dm1M	11.112	11.113	70.923	68.940	10.813
uk-2007-05	4.715	4.149	69.023	67.497	N/A
in-2004	6.119	4.970	74.324	70.670	N/A
uk-2014-host	7.686	6.596	72.019	67.283	N/A
eu-2015-host	3.174	2.625	13.585	12.442	N/A

time and memory), as expected.

For the (global) clustering coefficient (CC), the observed time ratios highlight the influence of graph density. This is shown with dataset `uk-2014-host`, whose ratio of 2.3418 (ns) is around 3x greater than the time ratio of the smaller dataset `in-2004` and close to 5x greater than the time ratio of the bigger `eu-2015-host`. The peak resident memory ratios for CC are more closely related to graph structure, with `eu-2015-host` (biggest of the tested web graphs) achieving a memory ratio 5x lower than `uk-2007-05` (smallest of the tested web graphs).

Note that we use a classic algorithm for computing both the clustering coefficient and counting triangles. This algorithm iterates over all edges (u, v) and, without loss of generality, it iterates over the neighbourhood of u, checking if each neighbour w of u is such that an edge (w, v) exists in the graph, where edge existence is checked against a hash table with all edges. Neglecting heavy hitters, *i.e.* vertices with more than $\sqrt{m}$, neighbours which are uncommon for large scale-free networks, the expected running time is $\mathcal{O}(m\sqrt{m})$. We can observe in Table 2 the third (CC) and fourth (CT hash) columns of the memory ratio section. Their memory ratio values are similar.

Since we can answer queries on edge existence with our proposed data structures in $\mathcal{O}(\log n \log m)$ time, we implemented an algorithm for counting triangles using edge queries directly against the data structure, without relying on a hash table. Note that the expected running time becomes now $\mathcal{O}(m\sqrt{m} \log n \log m)$ since we can no longer have edge queries in expected constant time. But now we need much less memory since we do not need a hash table to track edges, with memory usage essentially being the space required by the compact graph data structure. As observed in Table 2, the time ratio for this implementation (fifth column, CT neighbour) is around an order of magnitude greater than the the time ratio for CT hash.

4.4. Memory allocation analysis

Our implementation of the dynamic k^2-tree is based on the technique presented in [5], whose authors claim additional space is necessary to perform a union of two collections (which would be decompressed before the union operation taking place). The implementation we present is able to perform the union operation without decompressing the collections, effectively avoiding this pitfall. We show for dataset `uk-2007-05`, in Figure 11, a detailed analysis of heap memory usage. The analysis was performed using `valgrind`, with parameters `--tool=massif --max-snapshots=200 --detailed-freq=5`, and the visualizations using the `massif-visualizer`[8].

It can be observed that during execution where edges are continuously added, there are memory peaks associated with the union operation, temporarily increasing the heap usage by a factor of at most 2. This explains also the difference in maximum resident memory between `sdk2tree` and `dk2tree` observed before in Figures 8 and 9. The number of rebuilds/unions performed for dataset `uk-2007-05`, and for each static set in $\{E_1, \ldots, E_8\}$, is respectively 508, 127, 63, 32, 17, 9, 4 and 1.

5. Final remarks

We presented the `sdk2tree` implementation for representing dynamic graphs, based on the k^2-tree graph representation and relying on a collection of static k^2-trees. It is a dynamic data structure that supports edge additions and removals with competitive performance, showing faster execution times

[8] `https://github.com/KDE/massif-visualizer`

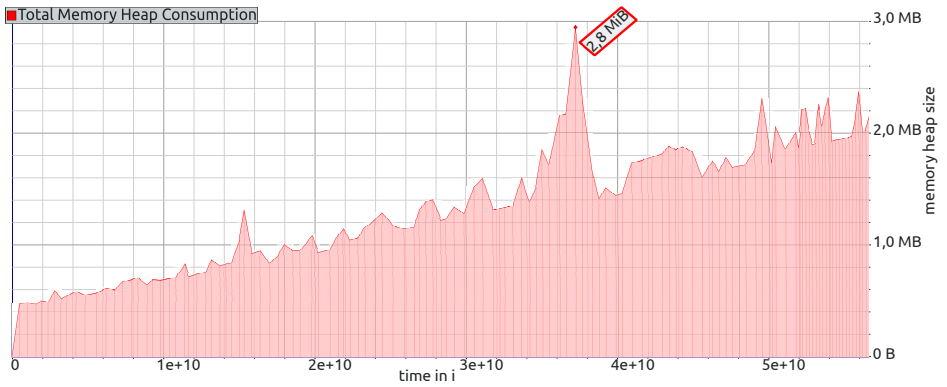

Figure 11: `valgrind` heap allocation profile for dataset `uk-2007-05`. The label `time in i` in the x axis denotes the number of instructions executed.

than the `dk2tree` implementation, a dynamic version of k^2-trees based on dynamic bit vectors, and on par with k^2-tries with respect to additions and queries.

We also present a `C++` implementation `sdk2sdsl`, a modular version which makes use of the succinct data structure library `sdsl-lite`. It achieves competitive performance compared to the other implementations analyzed in this document. `sdk2sdsl` also provides efficient implementations of edge and neighbourhood iterators, and of elementary graph algorithms, with empirical time and space complexity in tune with theoretical bounds.

Implementations like those analyzed in this paper, when implemented carefully, are of crucial importance for the efficient analysis and storage of evolving graphs, while drastically reducing the requirements of secondary storage compared to traditional dynamic graph representations. Hence, as future work, we envision further refinements to these data structures to achieve greater efficiency, namely in what concerns listing vertex neighbourhoods. `sdk2sdsl` is first step towards a reusable library for the analyses of large evolving graphs.

We are also aiming to research how these representations may be used within distributed graph processing systems in order to reduce the memory pressure observed often in these systems.

Acknowledgements

This work was supported by the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie [grant agree-

ment No 690941], namely while the first, third and fourth authors were visiting either the University of Chile or Enxenio SL. This work was partially funded by Fundação para a Ciência e a Tecnologia (FCT) [grants PTDC/-CCI-BIO/29676/2017, PTDC/CPO-CPO/28495/2017, CMUP-ERI/TIC/00-46/2014, UID/CEC/50021/2019], by MICINN-AEI (PGE and ERDF) [grants TIN2016-77158-C4-3-R, RTC-2017-5908-7,PID2019-105221RB-C41], by Xunta de Galicia (co-founded by ERDF) [grant ED431C 2017/58], and by ANID - Millennium Science Initiative Program - Code ICN17_002. We also wish to acknowledge the support received from the Centro de Investigación de Galicia "CITIC", funded by Xunta de Galicia and the European Union (European Regional Development Fund- Galicia 2014-2020 Program), by grant ED431G 2019/01.

References

[1] P. Boldi, S. Vigna, The WebGraph framework I: Compression techniques, in: S. I. Feldman, M. Uretsky, M. Najork, C. E. Wills (Eds.), Proceedings of the 13th international conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004, ACM, New York, NY, USA, 2004, pp. 595–602. doi:10.1145/988672.988752.
URL `https://doi.org/10.1145/988672.988752`

[2] N. R. Brisaboa, S. Ladra, G. Navarro, Compact representation of web graphs with extended functionality, Information Systems 39 (2014) 152–174.

[3] G. Navarro, Compact data structures: A practical approach, Cambridge University Press, 2016.

[4] N. R. Brisaboa, A. Cerdeira-Pena, G. de Bernardo, G. Navarro, Compressed representation of dynamic binary relations with applications, Information Systems 69 (2017) 106–123.

[5] J. I. Munro, Y. Nekrich, J. S. Vitter, Dynamic data structures for document collections and graphs, in: ACM Symposium on Principles of Database Systems (PODS), 2015, pp. 277–289.

[6] M. E. Coimbra, A. P. Francisco, L. M. Russo, G. De Bernardo, S. Ladra, G. Navarro, On dynamic succinct graph representations, in: 2020 Data Compression Conference (DCC), IEEE, 2020, pp. 213–222.

[7] C. Quijada-Fuentes, M. R. Penabad, S. Ladra, G. Gutiérrez, Set operations over compressed binary relations, Information Systems 80 (2019) 76–90.

[8] G. Csardi, T. Nepusz, et al., The igraph software package for complex network research, InterJournal, complex systems 1695 (5) (2006) 1–9.

[9] D. Arroyuelo, G. de Bernardo, T. Gagie, G. Navarro, Faster dynamic compressed d-ary relations, in: String Processing and Information Retrieval (SPIRE), 2019, pp. 419–433.

[10] P. Boldi, M. Rosa, M. Santini, S. Vigna, Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks, in: S. Srinivasan, K. Ramamritham, A. Kumar, M. P. Ravindra, E. Bertino, R. Kumar (Eds.), Proceedings of the 20th International Conference on World Wide Web, WWW '11, ACM, New York, NY, USA, 2011, pp. 587–596. doi:10.1145/1963405.1963488.
URL `http://doi.acm.org/10.1145/1963405.1963488`

[11] F. Chung, L. Lu, T. G. Dewey, D. J. Galas, Duplication models for biological networks, Journal of Computational Biology 10 (5) (2003) 677–687.

[12] A. Bhan, D. J. Galas, T. G. Dewey, A duplication growth model of gene expression networks, Bioinformatics 18 (11) (2002) 1486–1493.