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

A R T I C L E   I N F O

Article history:
Received 2 November 2020
Received in revised form 15 December 2021
Accepted 22 December 2021
Available online 29 December 2021

Keywords:
Compact graph representations
Dynamic graphs
k^2-tree
Graph library

A B S T R A C T

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.

© 2021 Elsevier Inc. All rights reserved.

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 recompressions.

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 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 $\lceil \log_{k^2} n^2 \rceil = \lceil \log_2 n \rceil$.

An example of this tree-shaped representation is shown in Fig. 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 $k^2m(\log_{k^2} \frac{m^2}{n^2} + O(1))$. 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 $O(\log_2 n)$ time by traversing the $k^2$ down to the desired matrix cell, until an empty submatrix (! 0) is found or we reach the 1 of the cell in the last level. Obtaining the neighbours of a node is done in $O(\sqrt{m})$ average time (for uniformly distributed adjacency matrices), and in $O(n)$ worst-case time, by reaching all the 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 [2]. In this paper, $k$ is a fixed value across the trees and remains constant a long 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 $C = \{E_0, \ldots, E_r\}$. Each static edge set $E_j$ is then represented using a static $k^2$-tree, except $E_0$ which is represented through a dynamic and uncompressed adjacency list. Fig. 2 depicts a link query over the different $E_j$ sets of the data structure.

Let $m_j$ be the number of edges in each set $E_j$. As discussed by Munro et al. [5], we must control both the number of edges $m_j$ in each set $E_j$ 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 $m_0$ is at most $m/\log^{2-\epsilon} n$, for some constant $\epsilon > 0$. We must also have that $r \leq 2/\epsilon$, so when $\epsilon$ is a fixed constant so is $r$. For example when $\epsilon = 1/4$ 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 $E_j$ is static (except for $E_0$) and has a maximum allowed size $m/\log^{2-\epsilon} n$. Whenever we reach the maximum size for $E_0$ (overflow), we find a set $E_j$, with $i < j \leq r$ such that $\sum_{i=0}^{j-1} m_i \leq m/\log^{2-\epsilon} n$ and (re)build $E_j$ with all edges in it and in the previous sets, and reset all previous sets to empty. By construction, $E_j$ has a maximum capacity enabling it to store the content of sets $E_0$ through $E_{j-1}$. We detail this process below.

\(^1\) https://github.com/aplf/sdk2tree.
\(^2\) https://github.com/joo956/dynamic_k2tree.
\(^3\) https://github.com/simongog/sdsl-lite.
Fig. 1. Example of adjacency matrix (left) and its corresponding \( k^2 \)-tree with \( k = 2 \) (right).

Fig. 2. \texttt{check\_link} query going through \( E_i \) sets to find an edge (noted in black).

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 \( O(m_0 \log n) \) 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 \( k^2 m_i \left( \log_{k^2} \left( n^2/m_i \right) + O(1) \right) \) bits (plus sublinear-order terms to support rank), where \( m_i \leq m/\log^{2-i} n \). Hence, overall, the space required is

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

(1)

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

\[
\sum_{i=1}^{r} k^2 m_i \log_{k^2} \frac{n^2}{m_i} \leq k^2 \sum_{i=1}^{r} \frac{m}{\log^{2-i} n} \log_{k^2} \left( \frac{n^2}{m} \log^{2-i} n \right)
\]

\[
= \frac{k^2 m}{\log^2 n} \sum_{i=1}^{r} \log^i n \left( \log_{k^2} \frac{n^2}{m} + (2 - i \epsilon) \log_{k^2} \log n \right). \quad (2)
\]

We can set \( r = 2/\epsilon \) because the sum is monotonic on \( r \). Then, because
\[ \sum_{i=1}^{r} \log^j n = \log^r n \left(1 + O(\log^{-\epsilon} n)\right) = \log^2 n \left(1 + O(\log^{-\epsilon} n)\right), \quad \text{and} \]
\[ \sum_{i=1}^{r} i \log^k n = r \log^r n \left(1 + O(\log^{-\epsilon} n)\right) = r \log^2 n \left(1 + O(\log^{-\epsilon} n)\right). \]

Equation (2) is
\[ k^{2m} \left(\log_{k^2} \frac{n^2}{m} + (2 - r + O(\log^{-\epsilon} n)) \log_k \log n\right) \]
\[ = k^{2m} \log_{k^2} \left(\frac{n^2}{m} \right) (1 + o(1)). \]

The whole Equation (1), since \(\sum_{i=1}^{r} m_i \leq m\), is then upper bounded by
\[ k^{2m} (\log_{k^2} (n^2/m) + O(1))(1 + o(1)), \]
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 \(O(\log_k n)\) 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 \(\log_k n\) and the bit is set to 0, then we set it to 1 and we are done. We note that this departure from the construction by Munro et al. [5], opportunistically inserting \((u, v)\) in our data structure in \(O(\log_k n)\) 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 \(\sum_{i=0}^{j} m_i \leq m / \log^2/j n\), and rebuild \(E_j\) with all edges in \(E_0, \ldots, E_j\) by successive unions of \(k^2\)-trees. Fig. 3 illustrates the process.

If opportunistic insertion occurs, then it takes \(O(\log_k n)/\epsilon\). If not and \(|E_0| \neq m_0\), 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 \(0 \leq i < j\). Note that the construction of the \(k^2\)-tree for \(E_0\) takes \(O(m_0 \log_k n)\) time [2], and the pairwise union of at most \(j \) \(k^2\)-trees representing collections \(E_0, \ldots, E_{j-1}\) takes \(O(m_j \log_k n)\) 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 \(O(\log_k n)\) 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 \(m_j / \log^2(1 - \varepsilon)\) \(\geq m_j / \log^2 n\). Therefore, the amortized cost of inserting an edge in \(E_j\) is \(O(\log_m n \log^2 n)\) and, since each edge can be moved once to each \(E_j\), with \(0 < j \leq r = \lceil 2 / \varepsilon \rceil\), the amortized cost of inserting an edge is \(O((1/\varepsilon) \log_m n \log^2 n)\). 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 \(E_j\) to 2-tree. Note that there is a bit set to 1 for each edge in a k2-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 k2-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 \(O((\log_m n) / \varepsilon)\), since checking if an edge exists in a given k2-tree takes \(O(\log_k n)\) [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 k2-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 \(C\). The full rebuild after \(m / \log \log n\) edges are deleted costs \(O(m \log_k n)\), i.e., it has an amortized cost of \(O(\log_m n \log \log n)\) per deleted edge. Overall, deleting an edge has then an amortized cost of \(O((1/\varepsilon) + \log \log n) \log_k n)\).

Querying (checking) for the presence of an edge works just as in k2-trees with the difference that we need to query all sets in the collection. Therefore, the querying cost increases by a factor of \(O(1/\varepsilon)\).

### 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 adk2tree project, but the underlying k2-tree implementation is based on the adal-lite k2-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. 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 \(E_i\) k2-tree.

#### 3.3.1. Edge iterator

As previously mentioned, the edge iterator iterates over the container \(E_0\) and then over the k2-tree for each \(E_i\). In the uncompressed container \(E_0\), it takes linear time to retrieve all its edges. However, for each k2-tree, it relies on a k2-tree edge iterator and it takes time proportional to the size of the k2-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 k2-tree. Thus, this queue has at most size \(O((\log(V)/\log(k)))\) which is the maximum level of the k2-tree. Iterating over edges in a k2-tree is performed then by visiting internal nodes and, if there are any children, then we check in all the k2 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 k2-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 k2-tree, however it saves the state from all the incomplete searched states. The running time for iterating all neighbours of a given vertex is the same of the neighbour-listing operation (see the end of Section 2), i.e., \(O(\sqrt{m})\) average time and \(O(n)\) worst-case time, as seen before for k2-trees [2].

#### 3.4. Comparison with other constructions

Given a graph \(G\), for a fixed \(\varepsilon\), the presented data structure uses essentially the same space as a static k2-tree, and supports insertions and deletions in \(O(\log_m n \log^2 n)\) and \(O(\log_m n \log \log n)\) amortized time, respectively. The implementation of dynamic k2-trees using dynamic bit vectors [4] requires a small space overhead, and it supports insertions and deletions in \(O(\log_m n \log n)\) time, which implies a slowdown factor of \(\Theta(\log^{-1/2} n)\) 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 $O(m \log n^2 / m + m \log k)$ bits, and supports edge queries in $O(\log n)$ time and updates in $O(\log_k n)$ 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 $sdk2dsdl$ which is based on the $sdsdl$-lite data structure library. We present an experimental analysis comparing different implementations: our new $sdk2dsdl$ 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/apfl/sdk2tree. Our new $sdk2dsdl$ 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 @ 2.04 GHz machine with 32K L1d cache, 64K L1i cache, 512 KB 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 $dk2tree$, $sdk2tree$, $sdk2dsdl$ 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 $sdk2dsdl$ 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.$^9$ 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 $sdk2tree$, $sdk2dsdl$ 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 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. Fig. 4 shows the average running time for adding an edge. As mentioned before, we include $k2tree$ in this comparison to observe the

---

$^5$ http://law.di.unimi.it/datasets.php.
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 sdk2dsdl is faster or as fast as sdk2tree for this operation.

Fig. 5 shows the average running time for removing an edge. For both dataset types, dk2tree was slower than others. The sdk2tree and sdk2dsdl implementations achieved close execution times and similar behaviour among datasets, with the library implementation sdk2dsdl being faster. We note that costs seem to correlate well with the predicted bounds.

Figs. 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 $O(\sqrt{m})$. The results on the duplication model show good correlation with previous average-case analyses [2]. We note that, in the worst case, it takes $O(n)$ time to obtain the neighbours of a vertex with the $k^2$-tree, even if there are none to return. 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 sdk2dsdl 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 operations. 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 (≈ 20%) bits/edge ratio compared to sdk2tree. This seems to be related to using a 64-bit index for the data structures.

Fig. 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.

Fig. 9 shows the maximum resident memory while removing edges. Since we are adding all edges before removing about 50% of them, the memory requirements for sdk2tree are exactly the same as in Fig. 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.
Fig. 9. Maximum resident memory while deleting edges in real Web graphs and in synthetic graphs (generated from a duplication model), respectively.

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

Fig. 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 sdk2dssl, sdk2dsdl and dk2tree are identical to those observed in Figs. 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 k²-tree structure stored on disk uses 22.66 bits per edge, i.e., a total of 701 MB. Although we are using the exact same implementation of k²-trees for representing the static collections within our sdk2tree and sdk2dssl 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 sdk2dssl, 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 sdk2dssl breadth-first search algorithm (applied to dataset dm50K) divided by its theoretical temporal complexity of $O(n \sqrt{m} + m)$. We omit dataset indochina-2004 from the graph algorithm tests for sdk2dssl as its topology does not allow for an adequate assessment of algorithms expected efficiency.

The first column of Table 2 shows the behaviour 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 behaviour of depth-first search (DFS). It has a behaviour similar to BFS for all dataset graph types (for both 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 \(O(m \sqrt{m})\). 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 \(O(\log n \log m)\) 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 \(O(m \sqrt{m} \log n \log m)\) 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 \(\text{fifth column, CT neighbour}\) is around an order of magnitude greater than 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 \(\text{which would be decompressed before the union operation taking place}\). 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 Fig. 11, a detailed analysis of heap memory usage. The analysis was performed using valgrind, with parameters \text{-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 Figs. 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 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 docu-

\(^8\) https://github.com/KDE/massif-visualizer.
ment. 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 envisage 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.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by the European Union’s Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie [grant agreement 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/CP0-CP0/28495/2017, CMUP-ERJ/TPC/0046/2014, UIDB/50021/2020], by MCINAEI/10.13039/501100011033 (PGE and ERDF) [grants RTC-2017-5908-7, PID2019-105221RB-C41, PID2020-114635RB-I00], by MCIN-AEI "NextGenerationEU"/PRTR [grants PDC2021-120917-C21, PDC2021-121239-C31], by Xunta de Galicia/GAIN (co-funded by ERDF) [grant ED431C 2021/53], 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, FEDER Galicia 2014-2020 80%, SXU 20% [grant ED431G 2019/01].

References

