On dynamic succinct graph representations

Miguel E. Coimbra*, Alexandre P. Francisco*, Luís M. S. Russo*,
Guillermo de Bernardo†,§, Susana Ladra†, Gonzalo Navarro‡

*INESC-ID / IST
Universidade de Lisboa
Portugal
miguel.e.coimbra@tecnico.ulisboa.pt
aplf@tecnico.ulisboa.pt
luis.russo@tecnico.ulisboa.pt

†Universidade da Coruña
Centro de Investigación CITIC
Facultad de Informática
Spain
gdebernardo@udc.es
sladra@udc.es

§Enxenio SL
Spain
gdebernardo@enxenio.es

‡IMFD, Dept. of Computer Science
University of Chile
Chile
gnavarro@dcc.uchile.cl

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. In this paper we follow instead the ideas by Munro *et al.*, and we present an alternative implementation for representing dynamic graphs using k^2-trees. Our experimental results show that this new implementation is competitive in practice.

Introduction

Graphs are ubiquitous among many complex systems, where we find large and dynamic complex networks. It is therefore important to be able to not only store such graphs in compressed form, but also to update and query them efficiently while compressed. Most succinct data structures for representing graphs are however static [1, 2]. And only recently, by relying on compact representations of dynamic bit vectors, succinct representations for dynamic graphs were presented [3]. These representations suffer however from a well known bottleneck in compressed dynamic indexing [4, 5]. In this paper we adopt the ideas proposed by Munro *et al.* [4] to represent dynamic graphs through collections of static and compact graph representations.

Our approach relies on k^2-trees to represent static graphs, but it supports edge insertions and removals. The edge insertion time is almost the same as the average construction time per edge of static k^2-trees. We provide an implementation and an extensive experimental evaluation.

From static k^2-trees to dynamic graphs

Let $G = (V, E)$ be a graph where V is the set of vertices, with size n, and $E \subseteq V \times V$ is the set of edges, with size m. The main idea is to represent G dynamically, supporting edge insertions and deletions, as well as common operations over graphs, through 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.

As discussed by Munro *et al.* [4], 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 $m/\log^2 m$ elements. In general we require that m_i is at most $m/\log^{2-i\varepsilon} m$, for some constant $\varepsilon > 0$. We must also have that $m_r = m/\log^{2-r\varepsilon} m \leq m$, which implies that $r \leq 2/\varepsilon$, when m is at least 3. 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. Whenever we reach the maximum for a given set E_i, we find a set E_j, with $i < j \leq r$ such that $\sum_{\ell=0}^{j} m_\ell \leq m/\log^{2-j\varepsilon} m$ and (re)build E_j with all edges in it and in the previous sets, and reset all previous sets. We detail this process below.

Space

Let us analyse 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 expected constant time. If we use also a hash table to store the adjacency lists, then we need $\mathcal{O}(m_0 \log m_0 + m_0 \log n)$ bits, where $m_0 \leq m/\log^2 m$ is the number of edges in E_0. Each set E_i, for $1 \leq i \leq r$, is represented in a static k^2-tree and it requires $k^2 m_i \left(\log_{k^2}(n^2/m_i) + \mathcal{O}(1)\right)$ bits [2], where $m_i \leq m/\log^{2-i\varepsilon} m$. Hence, overall, the space required is

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

bits. The first term in Equation 1 can be written as $\mathcal{O}\left((m/\log^2 m)(\log m + \log n)\right) = \mathcal{O}\left(m/\log n\right)$. To bound the second term we essentially need to sum a geometric sequence, we will assume $m_i = m/\log^{2-i\varepsilon} m$ as this is the case that requires more space. First let us sum the m_i values,

$$\sum_{i=1}^{r} m_i = m_1 \sum_{i=1}^{r} (\log^{\varepsilon} m)^{i-1} = m_1 \frac{(\log^{r\varepsilon} m) - 1}{(\log^{\varepsilon} m) - 1}. \tag{2}$$

Notice that as m grows the logarithms dominate the values in the fraction which therefore approximates $(\log^{r\varepsilon} m)/\log^{\varepsilon} m$. This expression can be further upper bounded by $(\log^2 m)/\log^{\varepsilon} m$, because of the relation between r and ε. Hence the overall bound is the relation $\sum_{i=1}^{r} m_i \leq m_1(\log^2 m)/\log^{\varepsilon} m = m$

Now for the complete formula we obtain an upper bound by noticing that $m_i \geq m_0$ for all i. The deduction is the following:

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

$$\leq k^2 \left(m \left((2 \log\log n) + \log_{k^2}(n^2/m) \right) + \mathcal{O}(1/\varepsilon) \right) .$$

A tighter bound can be obtained by noticing that the largest terms in the sum are the last ones. Hence essentially m_r takes the role of m_0 in the previous expression, yielding an $\varepsilon \log\log n$ term instead of a $2 \log\log n$ term, which is expected to be reasonably small.

Therefore, the overall space in bits is bounded by $k^2 m \left(\log_k(n^2/m) + 2\log\log n\right) + \mathcal{O}(k^2/\varepsilon) + o(m)$.

Insertion, deletion and queries

We rely on efficient set operations over k^2-trees [6]. 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 uncompressing C and C', with only some negligible extra space being used.

Insertion works as follows. Given a new edge (u, v), if $|E_0| < m_0$, then just add (u, v) to E_0 and we are done; otherwise, build a k^2-tree for E_0, find $0 < j \leq r$ such that $\sum_{i=0}^{j} m_i \leq m_j$, and rebuild E_j with all edges in $E_0, \ldots, E_j$ by successive unions of k^2-trees.

If $|E_0| < m_0$, then insertion takes expected constant 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.* [4] 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 imputed to the new edges added to E_j, which are at least $m_{j-1} = m_j/log^{\varepsilon}m$. Therefore, the amortized cost of inserting an edge in E_j is $\mathcal{O}(\log_k n \log^{\varepsilon} m)$ 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 $\mathcal{O}(\log_k n \log^{\varepsilon} m(1/\varepsilon))$. And 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 the corresponding bit to zero in E_j k^2-tree, update the number m' of deleted edges, and if $m' > m/\log\log m$, rebuild $\mathcal{C}$.

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. Note that there is a bit set to 1 for each edge in a k^2-tree. We are just exploiting that fact when we delete an edge and, hence, we do not use extra space. The full rebuild after $m/\log\log m$ edges are deleted costs

$\mathcal{O}(m \log_k n)$, *i.e.*, it has an amortized cost of $\mathcal{O}(\log_k n \log\log m)$ per deleted edge. Overall deleting an edge has then an amortized cost of $\mathcal{O}((\log_k n)/\varepsilon + \log_k n \log\log m)$.

Querying 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 $\mathcal{O}(1/\varepsilon)$.

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 it supports insertions and deletions in $\mathcal{O}(\log_k n \log^\varepsilon m)$ and $\mathcal{O}(\log_k n \log\log m)$ amortized time, respectively. The implementation of dynamic k^2-trees using dynamic bit vectors [3] requires a small space overhead, and it supports insertions and deletions in $\mathcal{O}(\log_k n \log n)$ time. Hence, since m is $\mathcal{O}(n^2)$, it has a slowdown by a factor of $o(\log n/\log\log n)$ 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 [3] work similarly to static k^2-trees – in practice they replace static bit vectors for dynamic ones – they suffer a slowdown by a factor of $\Omega(\log n/\log\log n)$ [5, Chapter 12].

We compare also with a new representation, k^2-tries, proposed recently [7]. This data structure uses $\mathcal{O}(m\log(n^2/m) + m\log k)$ bits, and it supports edge queries $O(\log_k n)$ time and updates in $O(\log_k n)$ amortized time. The implementation provided by k^2-tries authors supports only edge additions and queries, with slightly worse time complexities.

Experimental analysis

We compare the dynamic k^2-tree implementation proposed in this paper, henceforth named `sdk2tree` for static dynamic k^2-trees, with the dynamic implementation `dk2tree` based on dynamic bit vectors [3], a static implementation `k2tree` [2], and also with two versions of k^2-tries, `k2trie{1,2}`, that differ only on the parametrization (trading compression for speed) [7]. All other implementations were provided by their authors, and all code is available at `https://github.com/aplf/sdk2tree`.

All tested implementations are written in *C* and compiled with `gcc 6.3.0 2017-05-16` using the `-O3` optimization flag. Experiments were performed on an SMP machine with 256GB of RAM and four Intel(R) Xeon(R) CPU E7-4830 @ 2.13GHz, each one with 512KB in L1 cache, 2MB in L2 cache, 24MB in L3 cache and eight cores, 64 threads in total. All implementations are single-threaded.

We implemented a common interface to test each implementation. All dynamic data structures `dk2tree`, `sdk2tree` 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.

Datasets and methodology

We used 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 $|V|$

Table 1: The first four datasets were synthetically generated using a duplication model. The last five datasets are real-world Web graphs made available by the Laboratory for Web Algorithmics (LAW) [8, 9] (uk-2007-05 is actually uk-2007-05-100000 in the LAW website). Serialized space is reported for each data structure.

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

and $|E|$, respectively, and bits per edge (after serialization) for each implementation.

Real-world graphs were obtained from the Laboratory of Web Algorithmics[1] [8, 9]. Synthetic datasets were generated from the partial duplication model [10]. 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 [11]. We generated random graphs with selection probability $p = 0.5$, which is within the range of interesting selection probabilities [10]. The number of edges for those graphs is approximately 25 times the number of vertices.

We consider four major operations: edge additions, removals, querying/checking and vertex neighborhood listing. Elapsed time was measured using the `clock()` function[2]. Although the k2tree implementation does not support additions, we included it in the comparison. For that we build a k2tree for each dataset and we divided the time it took by the number of edges, obtaining then the average construction time per edge. This allowed us to evaluate the overhead introduced by dynamic data structures. The removal operation is compared only between sdk2tree and dk2tree. This operation was evaluated by adding all edges and removing a sample of 50% of them. All three *k2tree implementations were directly compared for the listing operation. After adding all edges, we evaluated this operation by asking for the neighborhoods 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[3]), and the disk space taken by data structures serialization.

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

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

[3] https://www.gnu.org/software/time/

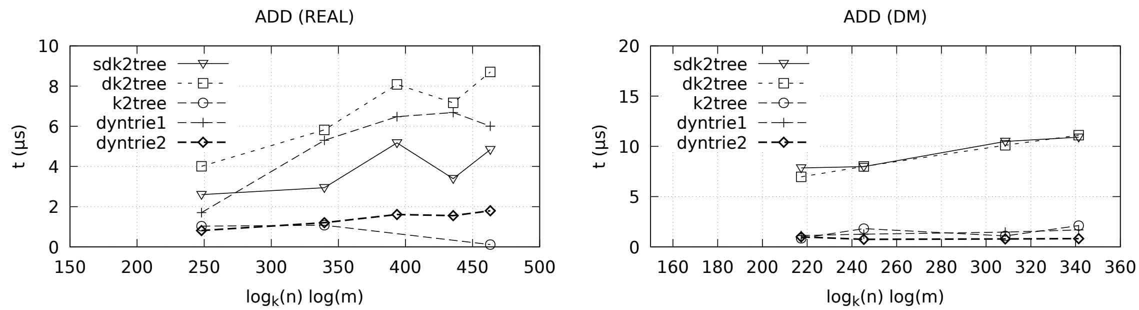

Figure 1: Average time taken for adding an edge in real Web graphs and in synthetic graphs (generated from a duplication model), respectively.

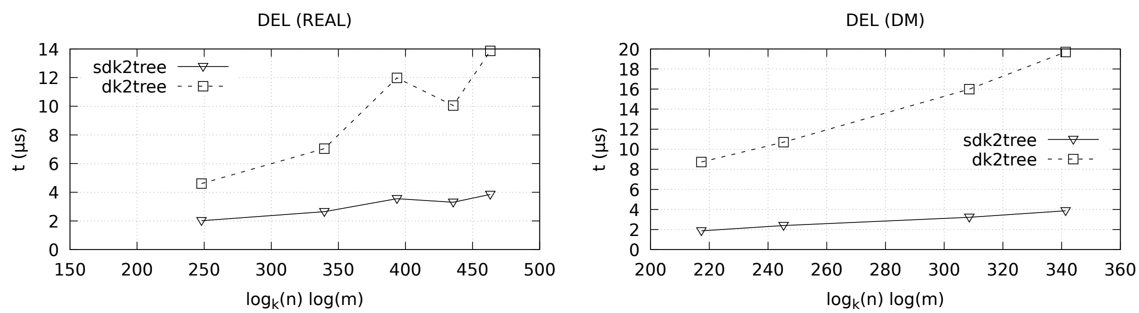

Figure 2: Average time taken for deleting an edge in real Web graphs and in synthetic graphs (generated from a duplication model), respectively.

Cost analysis

Let us analyse the cost of each operation over the different datasets and for the different implementations. Figure 1 shows the average running time for adding an edge. As mentioned before, we included `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`. We can observe that `k2trie`{`1,2`} are sometimes slightly faster 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. Figure 2 shows the average running time for removing an edge. Across all datasets, `sdk2tree` was consistently faster than `dk2tree`. We note that costs seem to correlate well with the predicted bounds.

Figures 3 and 4 show the average running time for listing vertex neighborhoods 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 average-case bound on the cost of listing vertex neighborhoods with `k2tree` [2]. This bound is valid also for `sdk2tree` and `dk2tree` as discussed previously in the theoretical analysis.

Let us now analyse how much memory is used by each implementation. In this analysis we will consider resident memory while we are performing operations. For the

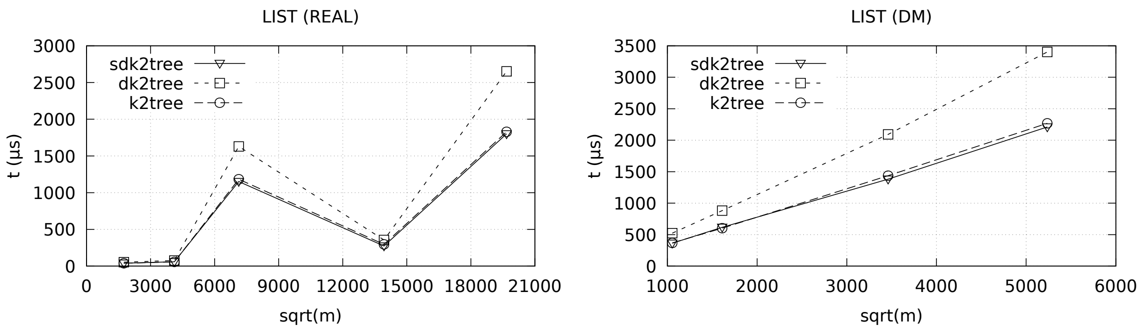

Figure 3: Average time taken for listing neighbors of random vertices in real Web graphs and in synthetic graphs (generated from a duplication model), respectively.

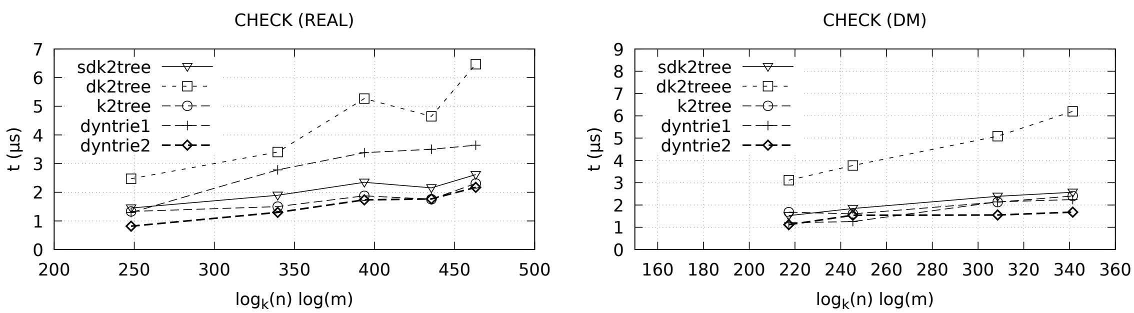

Figure 4: Average time taken for querying edges in real Web graphs and in synthetic graphs (generated from a duplication model), respectively.

space that each data structure takes once serialized on secondary memory, we refer the reader to Table 1. Figure 5 shows the max 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. We will analyse this in more detail below.

Figure 6 shows the max 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 Figure 5. This also means that the operation remove does not increase the space requirements in this implementation. On the other hand, the memory requirements are now higher for `dk2tree`, being more close to those of `sdk2tree`.

Figure 7 shows the max resident memory while adding edges and listing vertex neighborhoods. Since we are adding all edges as before, the memory requirements for `sdk2tree` and `dk2tree` are identical to those observed in Figures 5 and 6. We included now also the `k2tree` in our analysis. Given that this last implementation requires much more space for constructing the data structure, we had to use log scale in Figure 7. We should note however that once constructed, `k2tree` requires much

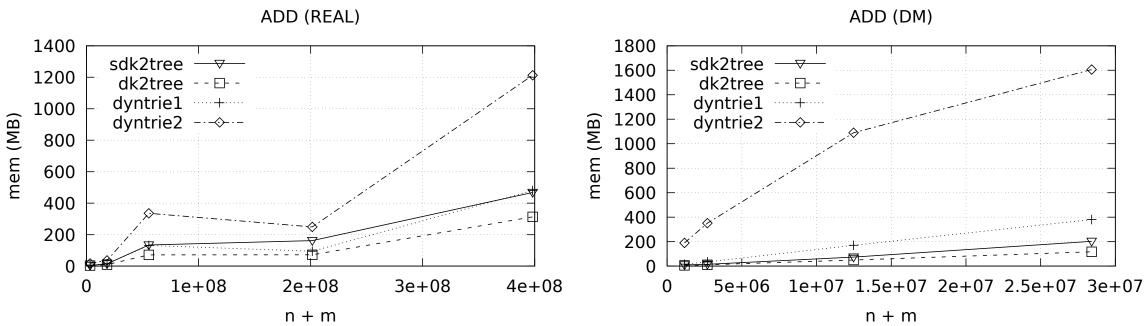

Figure 5: Max resident memory while adding edges in real Web graphs and in synthetic graphs (generated from a duplication model), respectively.

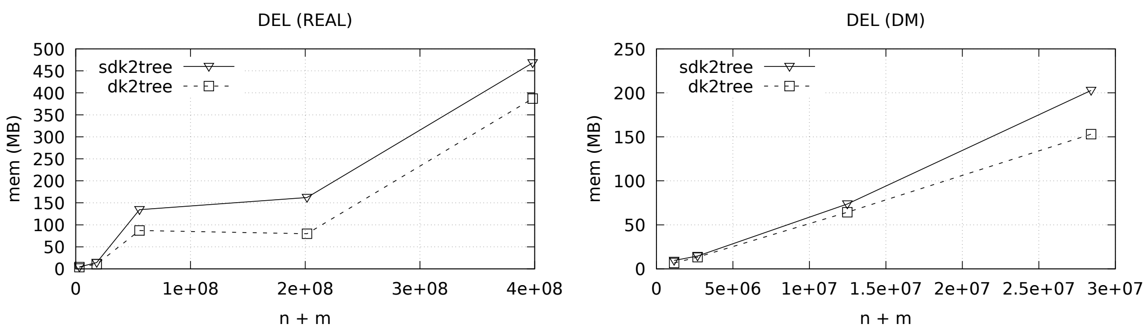

Figure 6: Max resident memory while deleting edges in real Web graphs and in synthetic graphs (generated from a duplication model), respectively.

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

Memory allocation analysis

Our implementation of the dynamic k^2-tree is based on the technique presented in [4], 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 8, 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`[4].

It can be observed that during execution where edges are continuously added,

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

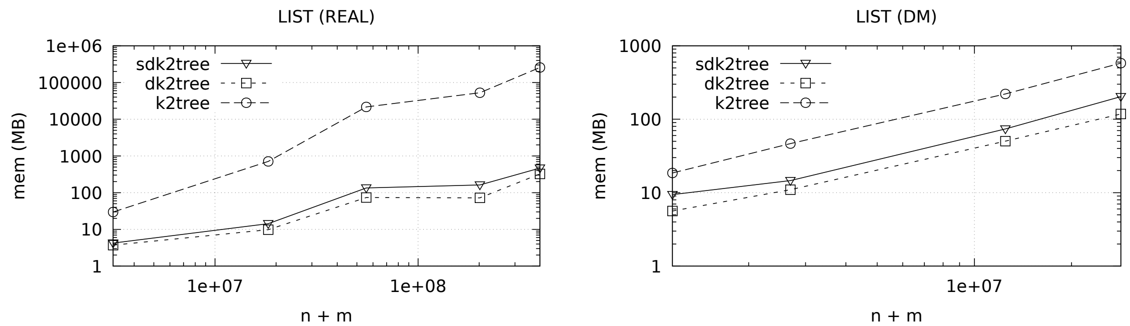

Figure 7: Max resident memory while listing neighbors of random vertices in real Web graphs and in synthetic graphs (generated from a duplication model), respectively.

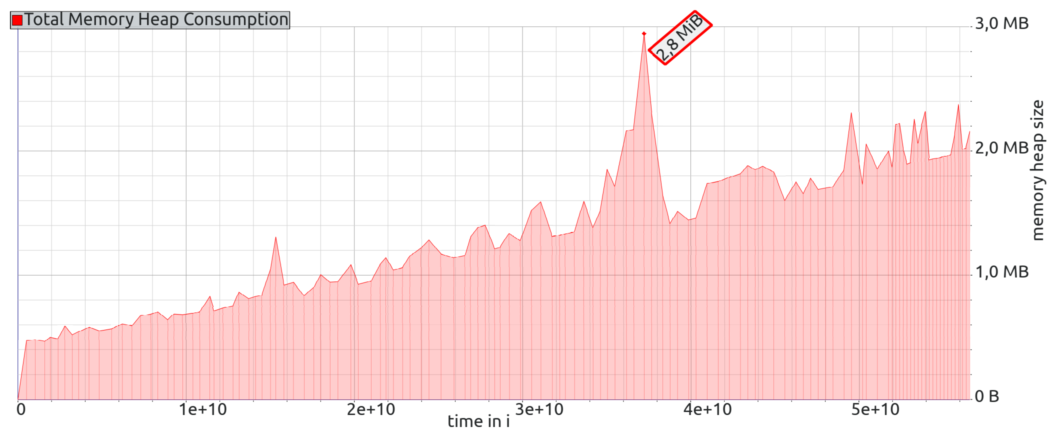

Figure 8: `valgrind` heap allocation profile for dataset `uk-2007-05`. The label `time in i` in the x axis denotes the number of instructions executed.

there are memory peaks associated with the union operation, increasing temporarily the heap usage by at most a factor of 2. This explains also the difference in maximum resident memory between `sdk2tree` and `dk2tree` observed before in Figures 5 and 6. 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.

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. This makes `sdk2tree` a semi-dynamic data structure. Nevertheless, it 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.

Implementations like those analysed 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 neighborhoods, in order to produce usable libraries for the analyses of large evolving graphs. We are aiming also 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.

Acknowledgments

This research has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie [grant agreement No 690941], namely while the first three 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 FCT TUBITAK/0004/2014, CMUP-ERI/TIC/0046/2014, PTDC/CCI-BIO/29676/2017, UID/CEC/50021/2019], by MICINN-AEI (PGE and ERDF) [grants TIN2016-77158-C4-3-R,RTC-2017-5908-7], by Xunta de Galicia (co-founded by ERDF) [grants ED431G/01, ED431C 2017/58], and by the Millennium Institute for Foundational Research on Data (IMFD).

References

[1] Paolo Boldi and Sebastiano Vigna, "The webgraph framework I: compression techniques," in *World Wide Web Conference (WWW)*, 2004, pp. 595–602.

[2] Nieves R. Brisaboa, Susana Ladra, and Gonzalo Navarro, "Compact representation of web graphs with extended functionality," *Information Systems*, vol. 39, pp. 152–174, 2014.

[3] Nieves R. Brisaboa, Ana Cerdeira-Pena, Guillermo de Bernardo, and Gonzalo Navarro, "Compressed representation of dynamic binary relations with applications," *Information Systems*, vol. 69, pp. 106–123, 2017.

[4] J. Ian Munro, Yakov Nekrich, and Jeffrey Scott Vitter, "Dynamic data structures for document collections and graphs," in *ACM Symposium on Principles of Database Systems (PODS)*, 2015, pp. 277–289.

[5] Gonzalo Navarro, *Compact data structures: A practical approach*, Cambridge University Press, 2016.

[6] Carlos Quijada-Fuentes, Miguel R. Penabad, Susana Ladra, and Gilberto Gutiérrez, "Set operations over compressed binary relations," *Information Systems*, vol. 80, pp. 76–90, 2019.

[7] Diego Arroyuelo, Guillermo de Bernardo, Travis Gagie, and Gonzalo Navarro, "Faster dynamic compressed d-ary relations," in *String Processing and Information Retrieval (SPIRE)*, 2019, pp. 419–433.

[8] Paolo Boldi and Sebastiano Vigna, "The WebGraph framework I: Compression techniques," in *World Wide Web Conference (WWW)*, 2004, pp. 595–601.

[9] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," in *World Wide Web Conference (WWW)*, 2011, pp. 587–596.

[10] Fan Chung, Linyuan Lu, T Gregory Dewey, and David J Galas, "Duplication models for biological networks," *Journal of Computational Biology*, vol. 10, no. 5, pp. 677–687, 2003.

[11] Ashish Bhan, David J Galas, and T Gregory Dewey, "A duplication growth model of gene expression networks," *Bioinformatics*, vol. 18, no. 11, pp. 1486–1493, 2002.