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
aplfr@tecnico.ulisboa.pt
luis.russo@tecnico.ulisboa.pt

†Universidade da Coruña
Centro de Investigación CITIC
Facultad de Informática
Spain
gdebernardoudees
sladra@udec.es

§Enxenio SL
Spain
gdebernardouenxenio.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 \( C = \{ E_0, \ldots, E_r \} \). 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 \( m/\log^{2-\varepsilon} m \) for some constant \( \varepsilon > 0 \). We must also have that \( m_r = m/\log^{2-\varepsilon} m \leq m \), which implies that \( r \leq 2/\varepsilon \), when \( m \) is at least 3. When \( \varepsilon \) 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 \( i < j \) such that \( \sum_{\ell=0}^{j} m_i \leq m/\log^{2-\varepsilon} m \) 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 \( O(m_0 \log m_0 + m_0 \log n) \) 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 \( k^2 m_i \left( \log_{k^2} (n^2/m_i) + O(1) \right) \) bits [2], where \( m_i \leq m/\log^{2-\varepsilon} m \). Hence, overall, the space required is

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

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

\[
\sum_{i=1}^{r} m_i = m_1 \sum_{i=1}^{r} (\log^\varepsilon m)^{i-1} = m_1 \frac{(\log^\varepsilon m)^r - 1}{(\log^\varepsilon m) - 1}.
\]

(2)

Notice that as \( m \) grows the logarithms dominate the values in the fraction which therefore approximates \( (\log^\varepsilon m)/\log^\varepsilon m \). This expression can be further upper bounded by \( (\log^2 m)/\log^\varepsilon m \), because of the relation between \( r \) and \( \varepsilon \). 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:

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

\[
= k^2 \left( \log_{k^2} (n^2/m_0) \left( \sum_{i=1}^{r} m_i \right) + O(r) \right) \leq k^2 \left( m \log_{k^2} (n^2/m_0) + O(1/\varepsilon) \right)
\]
A tighter bound can be obtained by noticing that the largest terms in the sum are the last ones. Hence essentially \( m_v \) 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 \( k^2 m (\log_k (n^2/m) + 2 \log \log n) + O(k^2/\varepsilon) + o(m) \).

**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 \( O(m_0 \log_k n) \) 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. [4] 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 \( E_0, \ldots, E_{j-1} \). In this case, the amortized cost of inserting an edge in \( E_j \) is \( O(\log_k n \log \varepsilon m) \) 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 \( O(\log_k n \log \varepsilon m(1/\varepsilon)) \). 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 \( O((\log_k n)/\varepsilon) \), since checking if an edge exists in a given \( k^2 \)-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 \( k^2 \)-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
\(O(m \log k n)\), i.e., it has an amortized cost of \(O(\log_k n \log \log m)\) per deleted edge. Overall deleting an edge has then an amortized cost of \(O((\log_k n)/\varepsilon + \log_k n \log \log m)\).

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 \(O(1/\varepsilon)\).

Comparison with other constructions

Given a graph \(G\), for a fixed \(\varepsilon\), the presented data structure uses essentially the same space as a static \(k^2\)-tree, and it supports insertions and deletions in \(O(\log_k n \log \varepsilon m)\) and \(O(\log_k n \log \log m)\) 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 \(O(\log_k n \log n)\) time. Hence, since \(m\) is \(O(n^2)\), 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 \(\Omega(\log n/\log \log n)\) \([5, \text{Chapter 12}]\).

We compare also with a new representation, \(k^2\)-tries, proposed recently \([7]\). This data structure uses \(O(m \log(n^2/m) + m \log k)\) 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 \texttt{sdk2tree} for static dynamic \(k^2\)-trees, with the dynamic implementation \texttt{dk2tree} based on dynamic bit vectors \([3]\), a static implementation \texttt{k2tree} \([2]\), and also with two versions of \(k^2\)-tries, \texttt{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 \url{https://github.com/aplf/sdk2tree}.

All tested implementations are written in \(C\) and compiled with \texttt{gcc 6.3.0 2017-05-16} using the \texttt{-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 \texttt{dk2tree}, \texttt{sdk2tree} and \texttt{k2trie}\{1,2\} are initialized empty. The static \texttt{k2tree} is initialized by reading the whole graph from secondary storage. Once initialized, the interface starts a main loop which reads instructions from \texttt{stdin} representing all supported edge operations, with additions and deletions not available in \texttt{k2tree}, and \texttt{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.

<table>
<thead>
<tr>
<th>Dataset</th>
<th></th>
<th></th>
<th>k2tree</th>
<th>dk2tree</th>
<th>sdk2tree</th>
<th>k2trie1</th>
<th>k2trie2</th>
</tr>
</thead>
<tbody>
<tr>
<td>dm50K</td>
<td>0.05</td>
<td>1.11</td>
<td>21.10</td>
<td>23.64</td>
<td>21.26</td>
<td>43.16</td>
<td>298.99</td>
</tr>
<tr>
<td>dm100K</td>
<td>0.10</td>
<td>2.59</td>
<td>22.66</td>
<td>25.27</td>
<td>22.76</td>
<td>47.31</td>
<td>257.61</td>
</tr>
<tr>
<td>dm500K</td>
<td>0.50</td>
<td>11.98</td>
<td>27.87</td>
<td>30.85</td>
<td>27.97</td>
<td>57.92</td>
<td>187.91</td>
</tr>
<tr>
<td>dm1M</td>
<td>1.00</td>
<td>27.42</td>
<td>29.48</td>
<td>32.63</td>
<td>29.49</td>
<td>58.78</td>
<td>132.92</td>
</tr>
<tr>
<td>uk-2007-05</td>
<td>0.10</td>
<td>3.05</td>
<td>2.98</td>
<td>3.39</td>
<td>3.16</td>
<td>5.62</td>
<td>11.11</td>
</tr>
<tr>
<td>indochina-2004</td>
<td>7.42</td>
<td>194.11</td>
<td>2.46</td>
<td>2.79</td>
<td>2.59</td>
<td>2.88</td>
<td>4.91</td>
</tr>
<tr>
<td>eu-2015-host</td>
<td>11.26</td>
<td>386.92</td>
<td>5.61</td>
<td>6.26</td>
<td>5.71</td>
<td>7.02</td>
<td>11.64</td>
</tr>
</tbody>
</table>

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

Real-world graphs were obtained from the Laboratory of Web Algorithmics [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/
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 $O(\sqrt{m})$, 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
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 \texttt{sdk2tree} requires more memory
than \texttt{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 \texttt{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
\texttt{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 \texttt{dk2tree}, being more close
to those of \texttt{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 \texttt{sdk2tree} and \texttt{dk2tree} are identical to those observed in Figures 5 and 6. We
included now also the \texttt{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, \texttt{k2tree} requires much
less space as shown in Table 1. For instance, for the dataset dm100K, $k^2$tree 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,
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 \texttt{sdk2tree} and \texttt{dk2tree} observed before in Figures 5 and 6. The number of rebuilds/unions performed for dataset \texttt{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 \texttt{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 \texttt{sdk2tree} a semi-dynamic data structure. Nevertheless, it supports edge additions and removals with competitive performance, showing faster execution times than the \texttt{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 refine-
ments 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