A graph algorithm library based on compact data structures

Joana Hrotkó
Instituto Superior Técnico
Lisbon, Portugal
joana.hrotko@tecnico.ulisboa.pt

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 static bit vectors. By adding dynamism to the static compact data structures, we can also represent dynamic graphs. However, this approach suffers from a well known bottleneck in compressed dynamic indexing, the problem of maintaining a changing collection so that we can query the data structure efficiently. In this work we present a k^2-tree based implementation which follows instead the ideas by Munro to circumvent this bottleneck. We refactored and extended the work of Coimbra by building a C++ library. The library includes efficient edge and neighbourhood iterators, as well as some illustrative algorithms. We also included a study on the add operation first proposed by Munro. Our experimental results show that our implementation is competitive in practice.

1 INTRODUCTION

Graphs are a natural way of modeling connections in the World Wide Web and social networks [7]. In the World Wide Web, each web page corresponds to a graph node, and each link corresponds to a graph edge. Such a directed graph is called a web graph. In social networks, the population's behavior and attributes are typically represented by social network graphs. Analyzing the structure and data of the graph enables the in-depth mining of network characteristics. The most common representations of a graph are the adjacency matrix and list. For small scale graph data, these two approaches can provide efficient querying. However, interesting Web graphs are very large and their classical representations do not fit into the main memory of typical computers, whereas the required graph algorithms perform inefficiently on secondary memory. Compressed graph representations drastically reduce their space requirements while allowing their efficient navigation in compressed form. Most compression algorithms require decompressing all of the data from the beginning before we can access an element from the data structure. Compact data structures aim precisely at this challenge. According to Navarro et. al. [15] a compact data structure maintains the data, and the desired data structures over it, in a form that not only uses less space, but is also able to access and query the data in the compact form, that is, without decompressing it. Thus, a compact data structure allows us to fit and efficiently query, navigate and manipulate much larger datasets in main memory unlike if we used the data directly from its plain form and classical data structures on top.

2 GRAPH REPRESENTATIONS

A graph is a structure consisting of a set of vertices $V = \{v_1, v_2, ...\}$ and a set of edges $E = \{e_1, e_2, ...\}$, where each edge has two vertices and they are not necessarily distinct, so we have that $E \subseteq V \times V$. This data structure can be denoted as $G = (V, E)$.

2.1 Adjacency Matrix

An adjacency matrix representation of a graph is preferred when representing a dense graph where $|E|$ is close to V^2 or when we need to be able to query quickly if there is an edge connecting two given vertices which can be given in $\mathcal{O}(1)$. In this representation, we assume that the vertices are numbered $1, 2, ..., |V|$ in some arbitrary manner. Then the adjacency matrix M representation of G consists in a $|V| \times |V|$ matrix of Boolean values, with the entry in row v and column w defined to be 1 if there is an edge connecting vertex v and vertex w in the graph, and to be 0 otherwise [19]. So the time to retrieve an edge in this representation is $\mathcal{O}(1)$ and it takes $\mathcal{O}(|V|^2)$ memory.

2.2 Adjacency List

The standard representation for sparse graphs (where $|E|$ is much smaller than $|V|^2$) is the adjacency list representation. In this graph representation we keep track of all the vertices connected to each vertex on a linked list [19]. A graph G consists of an array Adj of $|V|$ lists, one for each vertex in V. For each $u, v \in V$, the adjacency list $Adj[u]$ contains all the vertices v such that the edge $(u, v) \in E$. There are variations to implement a adjacency list. Another possible implementation associates each vertex in a graph with an array of adjacent vertices using a hash table. In this case, there is an extra memory usage for the hash table, however it allows to search an edge in $\mathcal{O}(1)$ instead of $\mathcal{O}(|Adj[u]|)$.

2.3 Compressed Sparse Row and Column

These are widely known and most used formats of sparse data structures. Mainly, they are used for write-once-read-many tasks and these daa structures make absolutely no assumptions about the sparsity structure of the matrix. In this case most of the entries of the matrix representation are zeros. The Compressed Sparse Row or Yale format [10] are commonly used to compress this kind of matrices. However, this data structure is only memory efficient for matrices where the non-zero entries (nnz) are less than $m \times (n - 1) - 1/2$. Comparing with the direct array representation where the required memory is n^2 while the Compressed Sparse Row requires $2\times$ $nnz + m + 1$. Analogous to Compressed Sparse Row, we have the Compressed Sparse Column where the values are indexed first by column with a column-major order.

2.4 Compact Representations

Now we will continue our data structure analysis with the compact data structures. he Web Graph is relative to a certain set of Uniform Resource Locator (URL)s. It is represented as a directed graph where the URLs are nodes and edges represent the links from x to y whenever page x contains a hyperlink towards page y [1]. The features of the links of a Web Graph that are usually quoted are locality and similarity were originally exploited by LINK database [18]. The locality feature refers to most links contained in a page lead the user to some other pages within the same host ("home", "next"). All these links share the same prefix, which by ordering lexicographically, the index of the source and target are close to each other. Additionally, the similarity refers to the pages that occur close to each other (in lexiographic order) tend to have many common successors; this is because many navigational links are the same within the same local cluster of pages, and even non-navigational links are often copied from one page to another within the same host.

2.4.1 LINK Database and WebGraph. The latter approach, which can be referred to as Web Graph compression, can be traced back to the LINK database [18]. This work presented techniques to compress the links in order to accommodate larger graphs, where some of them presented around 6 billion edges. In the LINK database work they noticed the locality and similarity features leading them to conclude that URL-ids in the same adjacency list tend to be close together in the URL-id space. Thus, they presented reference compression techniques, where they represented the graph as an adjacency list. More recently, the LINK database [18] has led to the development in Java of the WebGraph framework [1] which still provides some of the best practical compression-versus-speed trade-offs. In similarity to the LINK database, this work also exploits both locality and similarity features by using the same techniques as previously mentioned. This work introduces a new technique where instead of compressing directly based on the delta technique, they first isolate subsequences corresponding to integer intervals whose length [1] is not below a certain threshold. Thus, each list of extra nodes will be compressed by using a list of integers interval and also a list of residuals which are only compressed using differences. This technique is named the differential compression.

2.4.2 k^2-tree. The k^2-tree [3] is a very efficient compressed data structure that competes directly with WebGraph framework. The k^2-tree, a novel Web graph representation based on a compact tree structure, takes advantage of large empty areas of the adjacency matrix of the graph offering the least space usage 1–3 bits per link. The k^2-tree is a k^2-ary tree where all nodes present k^2 child nodes or no children if they are a leaf node. In a graph with n nodes, the height of the k^2-tree is $h = \lfloor \log_k n \rfloor$. The k^2-tree will consist in two bit vectors based on the bit vectors [17]. T (tree) which stores all the bits of the k^2-tree except those at depth h. The bits are placed following a levelwise transversal: first the k^2 binary values of the children of the root node, the values of the second level, and so on. In addition, L (leaves) – stores the last level of the tree. This it represents the value of (some) original cells of the adjacency matrix. The representation $T||L$ permits fast navigation to get the i^{th} child of a node x in the tree, for any $0 < i < k^2$. Consider $child_i(x)$ where x is a position of T such that $T[x] = 1$. Then $child_i(x)$ is at position $rank_1(T, x) \cdot k^2 + i$ of $T||L$, where $rank_1(T, x)$ is the number of 1s in $T[0, x]$. In order to carry out the operation $child_i(x)$ efficiently, we need to support $rank_1(T, x)$ queries efficiently. The *rank* operation can be carried out in constant time and fast in practice using sublinear space on top of the bit sequence [5]. Moreover, this data structure allows to query directly from the compressed form of the data structure. In fact, it allows to query the following: list successor and predecessor nodes and check a link in the graph.

2.4.3 Dynamic k^2-tree. The dynamic k^2-tree introduces the insertion and deletion operations without having to decompress the whole data [14]. In their work, they demonstrated that the gap between static and dynamic variants of the indexing problem can be almost closed. The main idea behind the dynamic k^2-tree is to keep the data distributed among several static k^2-trees structures also known as collections $C = \{E_1, ...E_r\}$. However, the E_0 is represented through a dynamic and uncompressed adjacency list in order to achieve the optimal amortized cost for each operation we must control the number r of edges is each set E_i. The uncompressed container E_0 can be implemented with an adjacency list and a hash table that maps an edge to a position in the adjacency list. This way we can access an edge in $\mathcal{O}(1)$. Moreover, the first set E_0 contains at most $m/\log^2 m$ edges according to [14]. In general, each E_i has $m_i/\log^{2-i\varepsilon} m_i$, for some constant $\varepsilon > 0$, where m_i is the number of edges in E_i. If we have that $i = r$ then we have that $m_r = m/\log^{2-r\varepsilon} m$ and that $m_r \leq m$ which implies that $r \leq 2/\varepsilon$, when m is at least 3. In [8] it was demonstrated we should use $\varepsilon = 1/4$ which gives us $r = 8$, so we will have 7 static k^2-trees to represent each E_i. Hence for each E_i maximum edges follows a geometric progression. Regarding the space required to represent the data structure we must consider E_0 and the collections C. For E_0 we have $\mathcal{O}(m_0 \log(m_0))$ to represent the adjacency list plus $\mathcal{O}(m_0 \log(m_0))$ bits for a coupled hash table to answer the existence of edges in constant time, where $m_0 \leq m/\log^2 m$ is the number of edges in E_0. Regarding the collections C, the required space for each set E_i, where $1 \leq i \leq r$ is represented by a static k^2-tree which requires $k^2 m_i(\log_{k^2}(n^2/m_i + \mathcal{O}(1)))$ bits [4], where $m_i \leq m/\log^{2-i\varepsilon} m$. Hence, overall the space required is $k^2 m(\log_k(n^2/m) + 2log\log(n)) + \mathcal{O}(k^2/\varepsilon) + \mathcal{O}(m) bits$ [8]. The operations supported by this data structure are insertions, deletions, listing neighbors of a node and checking the existence of an link. The insertion is carried out depending on the current size of E_0. If $|E_0| < m_0$, then we just add the new edge (u, v) in the adjacency list and we are done. Otherwise, we first build a new k^2-tree with all the edges in E_0 and then we need to find the container E_j $0 < j \leq r$ such that $\sum_{i=0}^{j} m_i \leq m_j$, and rebuild E_j with all the edges from $E_0, ..., E_j$ by performing successive unions of k^2-trees [4]. The amortized analysis of the insertion cost follows the argument presented by [14] for the general case with $0 < j \leq r = \lfloor 2/\varepsilon \rfloor$, gives us a time complexity of $\mathcal{O}(\log_k(n) \log^{\varepsilon} m(1/\varepsilon))$. Similarly to the insertion operation, the deletion operation also takes into consider both E_0 and the collection C. In the first case, if the edge exists in E_0, then we remove it from the hash table. Otherwise we need to find $0 < j \leq r$

[1]The authors consider the length of an integer interval is the number of integers it contains.

such that $(u, v) \in E_j$ and, if there is such j, set the corresponding bit to zero in E_j. During the deletion, we need to mark how many edges have been deleted in C until $m' > n/loglogm$ edges were marked. Once we reach this value we need rebuild C again. Deleting and edge in E_0 takes constant 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 $\mathcal{O}(\log_k n)$ [4], and we might have to look in each collection E_i, with $0 < i \leq r = \lfloor 2/\varepsilon \rceil$. Once an edge is found, marking it for deletion takes constant time. However, in need of rebuilding, after $m = loglogm$ deleted edges, the costs in this case is $\mathcal{O}(m \log_k n)$, since it has an amortized cost of $\mathcal{O}(\log_k nloglogm)$ per deleted edge. Overall deleting an edge has then an amortized cost of [8] $\mathcal{O}((\log_k n)/\varepsilon + \log_k n \log(\log(m)))$.

3 GRAPH APIS

This is a concept in software that essentially refers to how multiple applications can interact with and obtain data from one another. A Library is a well-defined interface by which the behavior is invoked. In addition, the behavior is provided for reuse by multiple independent programs. Libraries typically follow design patterns and have its code organized in such a way that there is no need to re-implement the same behaviour. We have analysed four different graph libraries [2] [13] [9] [1]. We verify that all of them present a similar interface for their different graph data structures and algorithms.

3.1 Extended functionality

The union operation was extended to the SDSL, since it is pivotal to be able to implement the insertion and deletion operations in the dynamic k^2-tree. Besides, the node, edge and neighbor iterators were added. For the insertion operation after the size of the E_0 is surpassed we need to create a new k^2-tree with all its edges and afterwards we need to perform successive unions until the collection that can accommodate all the edges from the previous containers plus the E_0. However, this operation was not implemented in SDSL so we had to extend this functionality in the library. All libraries support a wide range of useful algorithms for its data structures. Accordingly, we intended for our API to support some basic search algorithms, namely BFS and DFS and also more specific algorithms more related to Web Graphs: the counting triangles in a graph [11] and pageRank [16].

3.2 Improved performance on the addition of an edge

We also decided to add to our work another implementation for the add operation suggested by Munro in [14]. During the addition of a new edge, when the container E_0 is full, it is needed to integrate the new k^2-tree from the E_0 with the others k^2-tree containers performing consecutive unions. In this phase of the operation represents a bottleneck that can be mitigated. We present two different kinds of approaches to tackle this issue.

3.2.1 Munro and Delayed Union. As discussed in [14], it is possible to mitigate this bottleneck. This can be achieved by delaying the union operation while the E_0 container is not yet complete. So instead of waiting for the completion of all the necessary unions, this is mitigated by processing proportional iterations of the union operation while the E_0 is yet incomplete. Nonetheless, this approach is complex to implement, so we also implemented a much more simpler version of the union delay for comparison and testing purposes. In this second delay version we only delayed the union operation (as a whole) to the next addition. It is important to note, that a copy of the k^2-trees collections no longer is needed, since after the first union process, although is unfinished, the data structure remains coherent with all the edges (including the new edge) are present in the data structure. Overall, it is expected to this version take as much time and memory as the original add operation version while reducing some of the spikes.

3.2.2 Background Thread. Additionally, it was also implemented a parallel version where a background thread processes all time-consuming operations, that is the conversion from E_0 and the new edge to a k^2-tree and the unions operations. There is an edge case however. At the time of adding a new edge, the previous rebuild the data structure might not be over. Two different approaches were carried out. In the first approach, the E_0 is incremented and the edge is inserted in this container, delaying the new union processing in the background. Hence, the thread in the background is triggered to process the unions as soon as the prior union processing has finished. This method has the disadvantage of having extra memory usage. Due to the extra rebuilds of E_0 we also implemented a second version of the parallel version, where instead of inserting the edge in the E_0, the main thread waits for the previous rebuild to be finish in order to avoid the increase growth of the E_0.

4 EXPERIMENTAL ANALYSIS

The experiments were performed on a 8-core machine AMD Ryzen 7 2700X Eigh-Core Processor @2.04GHz machine with 32K L1d cache, 64K L1i cache, 512K L2 cache, 8192K L3 cache and system memory of 64GB RAM. All the operations except the add, were evaluated from a previously serialized dynamic k^2-tree, reading the whole graph from secondary storage. Our implementation was compiled with `g++ 7.5.0` and the SDK implementation was compiled with `gcc 7.5.0` both using the `-O3` optimization flag. We used both real and synthetic datasets. In 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. The `sdk2tree`[2] corresponds to the SDK implementations and the `sdslk2tree`[3] corresponds to our implementation with the k^2-tree from SDSL[4]. Real-world graphs were obtained from the Laboratory of Web Algorithmics [5] [1]. Besides, the synthetic datasets were generated from the partial duplication model [6]. 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. The generated random graphs [8] have a selection probability of $p = 0.5$, which is within the range of interesting selection probabilities [6]. The number of edges for those graphs is approximately 25 times

[2]https://github.com/aplf/sdk2tree
[3]https://github.com/joo95h/dynamic_k2tree
[4]https://github.com/joo95h/sdsl-lite
[5]http://law.di.unimi.it/datasets.php

Dataset	$\lvert V\rvert$ (M)	$\lvert E\rvert$ (M)	`sdk2tree` (bit/edge)	`sdslk2tree` (bit/edge)
`dm50K`	0.05	1.11	21.26	25.01
`dm100K`	0.10	2.59	22.76	27.24
`dm500K`	0.50	11.98	27.97	32.25
`dm1M`	1.0	27.42	29.49	34.31
`uk-2007-05`	0.10	3.05	3.16	3.51
`in-2004`	1.38	16.92	3.14	3.56
`uk-2014-host`	4.77	50.83	9.58	11.02
`indochina-2004`	7.42	194.11	2.59	2.93
`eu-2015-host`	11.26	386.92	5.71	6.60

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

the number of vertices. The elapsed time was measured using the `clock()` function [6] and we considered the peak of memory usage was obtained with `GNU time` [7] by the maximum resident size and the disk space to. For all evaluated operations, we measured the average time per individual operation where each time and memory resulted from the average of 5 individual executions. In 1, also shows the compression ratio in bit per edge for both implementations. We denote a big gap between both datasets; in the real Web Graphs datasets the compression ratio was much better than in the `dmgen` datasets. In the real datasets the edges are ordered since the websites usually point to links within the same website, promoting, in our case, for the diagonal to be filled of the matrix representation of the graph. Thus, in our k^2-trees we will have less filled paths, that is less 0s in T.

4.1 Union Operation

In 1 we have the average time and memory to perform the union operation between the same k^2-tree for the synthetic (`dmgen`) and real Web Graphs datasets. For the synthetic and the Web Graphs datasets, the average time for both implementations was very similar. However, there was some discrepancy regarding memory. The memory plot shows the T and L bit vectors of the k^2-tree in the SDSL bit vector implementation consumes more memory than in the SDK. In 1, it is difficult to understand the differences between the two implementations regarding the average time. However, from the collected data, we know that our implementation is faster for smaller k^2-trees than the SDK.

4.2 `DKTree` Operations

Now that we have analyzed the implemented operation in the k^2-tree now we move on to the dynamic data structure. In this section we will evaluate and compare the list neighbourhoods, check individual link, add and delete edge operations between our

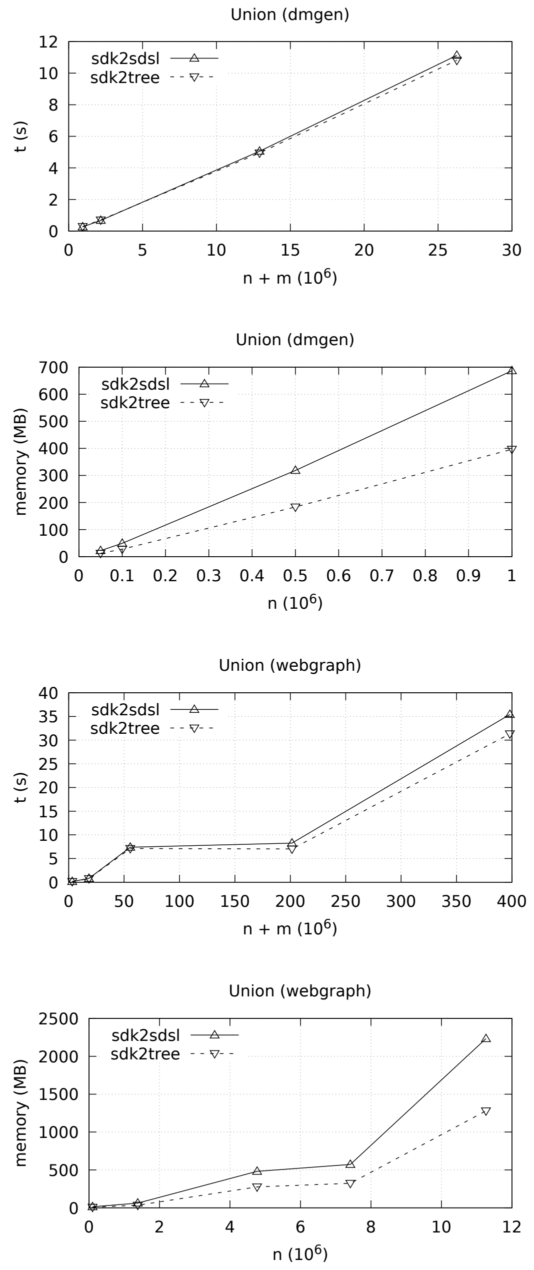
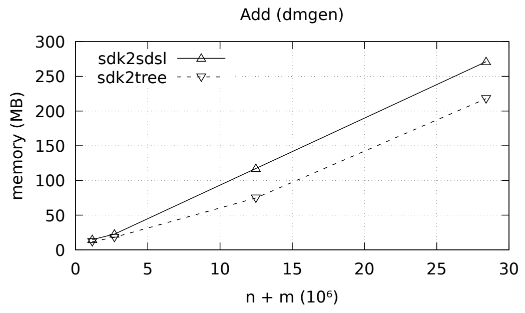


Figure 1: Average time taken to perform the union operation between the same k^2-tree for synthetic (`dmgen`) and real Web Graphs (`webgraph`) datasets.

implementation and the SDK's. Finally, we also will evaluate the edge and neighbour iterators and in the end of this chapter we will analyze the implemented algorithms for the library. Since for all implementations, the data structure was loaded from memory, all operations present the same memory plot in 2. As previously seen in 1, the SDSL implementation consumes more memory than the

[6]http://man7.org/linux/man-pages/man3/clock.3.html
[7]https://www.gnu.org/software/time/



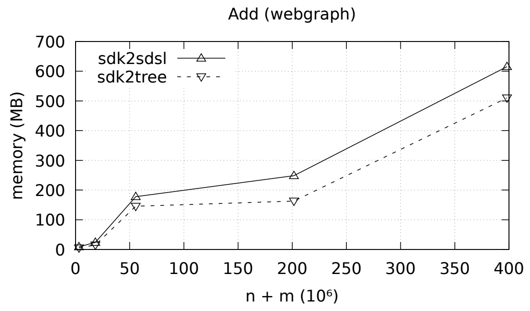
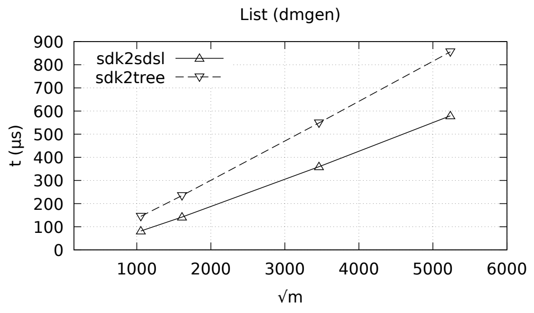


Figure 2: Resident memory peak for synthetic (dmgen) and Web Graph datasets (webgraph).



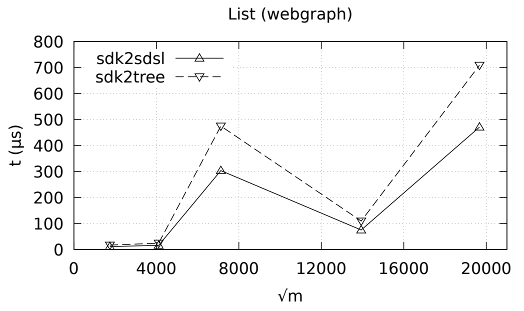


Figure 3: List neighbors operation average time for synthetic (dmgen) and real Web Graph datasets (webgraph).

SDK's, consequently our serialized data structure will also occupy more space.

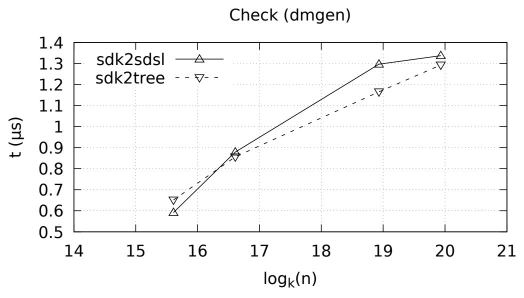
4.3 Operations

From 3 4 5 and 6 we can see that our version had better performance overall. Although, in 5 we can see that we had slightly different results for the synthetic and Web Graph datasets. For the synthetic dataset, our implementation had better performance regarding time and worst regarding resident memory peak. On another hand, in the Web Graphs datasets, the time difference between the two implementations was very close, although the SDK implementation was faster. The reason behind this is that in C we have smaller k^2-trees in each container, giving a small advantage to our implementation during the union time. However, for the Web Graphs datasets, the stored k^2-trees in C have more edges, turning the advantage to the SDK implementation.

4.4 Augmented Add Operation Versions

We also implemented four more versions for the add operation. In the following plots we compare time and resident peak memory usage. As previously mentioned, the Munro's version was divided into two versions during the development phase and we also have two alternative parallel versions. For easier understanding we have the following:

- `add edge` – first implemented version.
- `add edge delay` – during the rebuild, each necessary union (whole) operation is delayed for the next add operation.



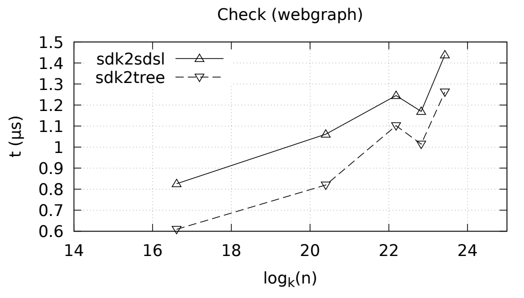
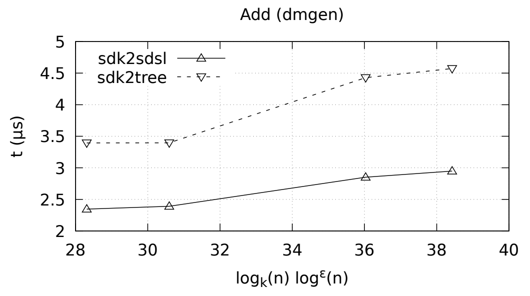


Figure 4: Check operation average time for synthetic (dmgen) and real Web Graph (webgraph) datasets.



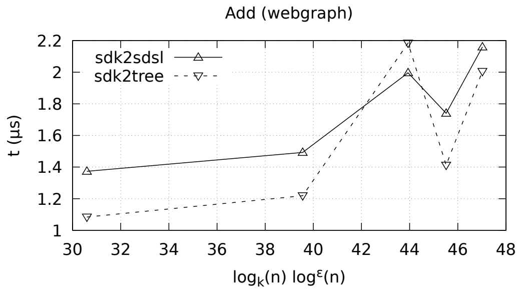
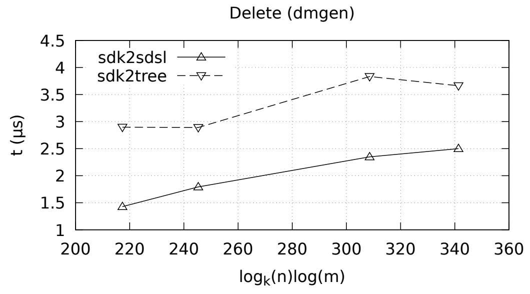


Figure 5: Add operation average time for synthetic (`dmgen`) and Web Graph (`webgraph`) datasets.



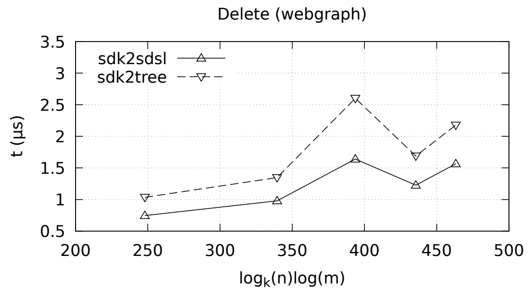


Figure 6: Delete operation average time for synthetic (`dmgen`) and Web Graph (`webgraph`) datasets.

- `add edge munro` – during the rebuild, the union operations are distributed in E_0 size iterations, according to Munro et. al. [14].
- `add edge parallel` – A background thread runs the union operations when rebuilding the data structure and the main thread doesn't wait for the background thread to finish.
- `add edge wait` – A background thread runs the union operations when rebuilding the data structure and the main thread waits until the background thread works.

Analyzing the plot in 7, for the `add edge` and `add edge parallel wait` are visible some big and medium spikes. For the `add edge delay` we have half of the size of the spikes from the two previous versions and, finally, for the `add edge parallel` we can see fewer and smaller spikes and finally for the `add edge munro` we there are no spikes at all.

The implementations' behaviour deviates in each dataset. Nonetheless, the maximum resident peak memory is the same for both. As expected, the `add edge delay` took the same time and memory comparing with the `add edge`, since the only difference between these two versions is the moment of conclusion of the rebuild process, where in the `add edge delay` is in the upcoming additions. Next, we will be looking at the `add edge munro` version. This was one of the slowest versions, due to the additional necessary copies to keep the data structure coherent at the time of the unions process. In fact, all extra implemented versions, except the `add edge delay` need to create a copy of the current state of C before starting the rebuild process, however the `add edge munro` has no background thread to help to speed the overall process. This is noticeable in the memory plots, where the `add edge munro` and `add edge wait` have very similar maximum resident peaks. Regarding the `add edge parallel` was the fastest version in the synthetic datasets and the slowest in the real Web Graphs datasets. The increase of the size of E_0 comes at a great cost for more dense graphs, just like the Web Graphs. Moreover, this version had the highest memory growth, ending consuming three times more memory than the other versions. Lastly, we are going to analyze the `add edge wait`. For the synthetic, this version took longer than the original implementation `add edge`, meaning the main thread waited for the background thread to finish quite often. Recall 5. Notice how the average time to add an edge in the synthetic datasets is higher than in the Web Graph datasets. The meaning of this difference is that the unions operations take longer in the synthetic datasets. An additional evidence that supports this is the compression ratios in 1. Hence, the performance of this version will be better in the Web Graphs datasets as the background thread won't make the main thread wait as much as in the synthetic datasets.

4.5 Iterators

For both edge and neighbor iterators we only plotted against the synthetic datasets. For the evaluation of the edge iterator, we loaded the graphs from memory and iterated over the whole datasets. In 8, we have the comparison between the time and resident peak memory consumption against the expected theoretical results. The collected data shows the implemented edge iterator meets the theoretical bounds for time and memory. During the edge iteration,

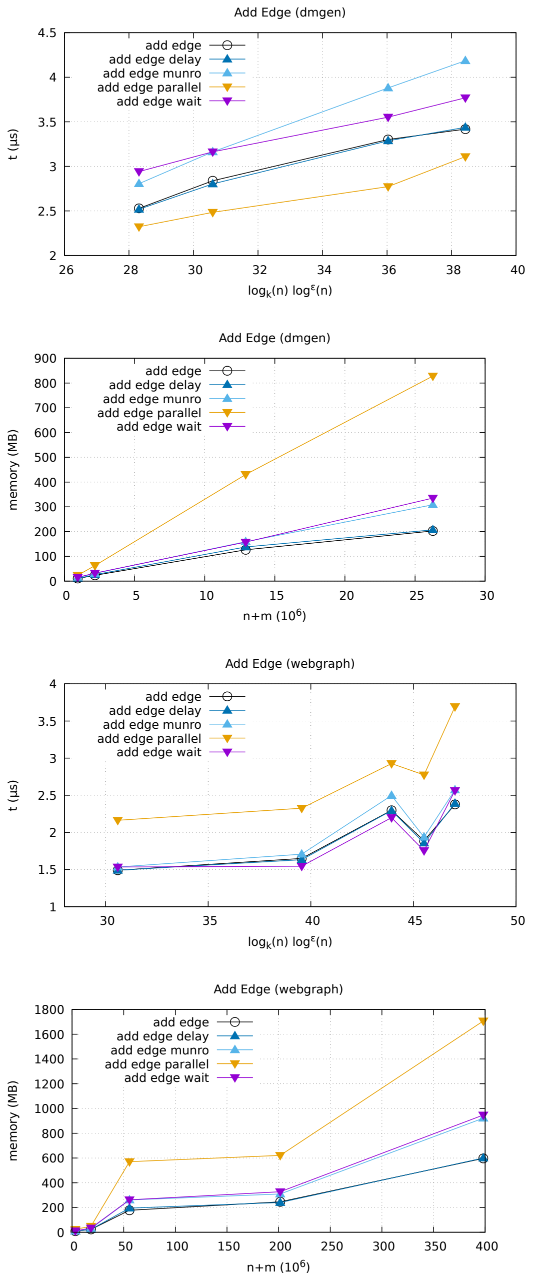


Figure 7: Average time and resident peak memory for the four different versions of the add operation for the synthetic (dmgen) and Web Graph (webgraph) datasets.

the whole k^2-tree is visited. Thus, for a smaller k^2-tree, the edge iterator will take less time.

The neighbor iterator was evaluated with the same dataset in 3 and we also plotted against the list neighborhood method for a richer comparison between the two implementations. From the experimental results in 9 show a straight line for the synthetic datasets. However, there is a cost for maintaining a stack with the current

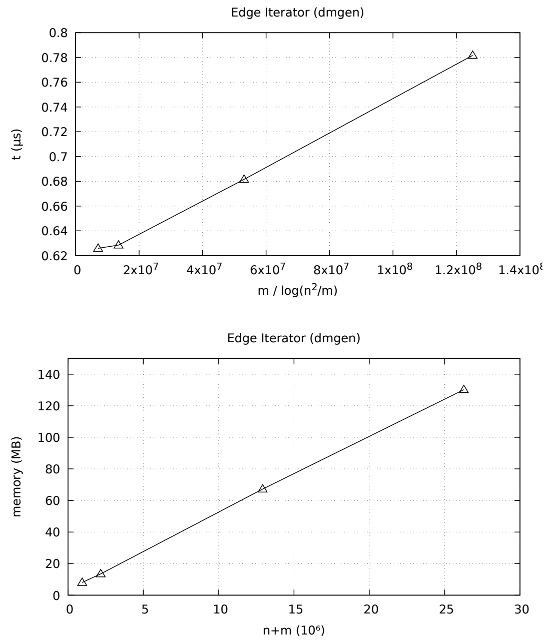


Figure 8: Edge Iterator operation average time (dmgen) and average memory resident peak (webgraph) for synthetic datasets.

state of the navigation in the k^2-tree. Nonetheless, the memory usage was identical for both iterator and method implementations.

5 ALGORITHMS

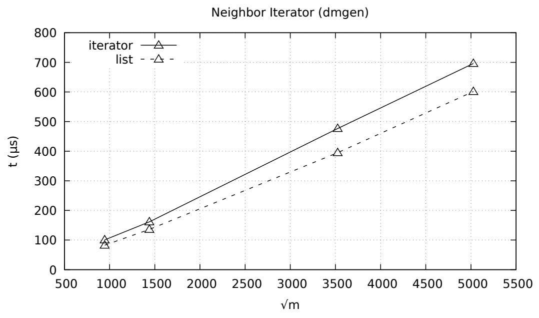
We implemented some well known graph algorithms, for which we compare consumed memory and execution time against expected theoretical results.

5.1 Breadth First Search and Depth First Search

In 10 we show the behavior of BFS. For the running time, as we increase the dataset size, the plotted curve is a straight line for the duplication model and an almost straight line for the Web graphs, which shows the implementation follows the expected theoretical time of BFS given by $\mathcal{O}(n\sqrt{m} + m)$. Note the $\sqrt{m}$ due to the cost of listing of neighbourhoods. As expected, the peak memory while running BFS is bounded by $\mathcal{O}(n+m)$. The results for DFS are shown in 11. It has a behavior similar to 10 for both the Web graphs and duplication model graphs, as expected.

5.2 Clustering Coefficient and Count Triangles with Hash table

For clustering coefficient and triangle counting algorithms, we present their running time and memory usage results only for the duplication model graphs. We omit results on the Web graph datasets as they are structurally different and thus performance measurements with these algorithms does not follow the same behavior as we increase dataset size. Note that we used a classic



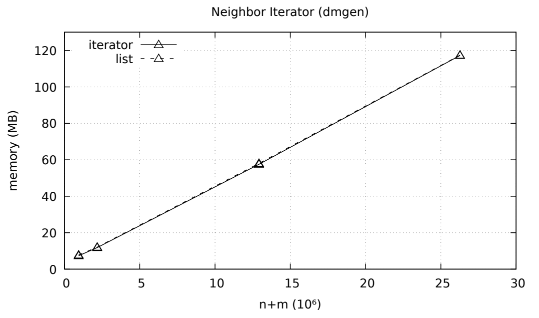
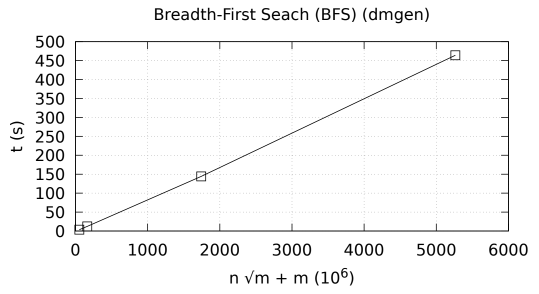


Figure 9: Neighborhood Iterator and List Neighborhood operations average time (dmgen) and average memory resident peak (webgraph) for synthetic datasets.



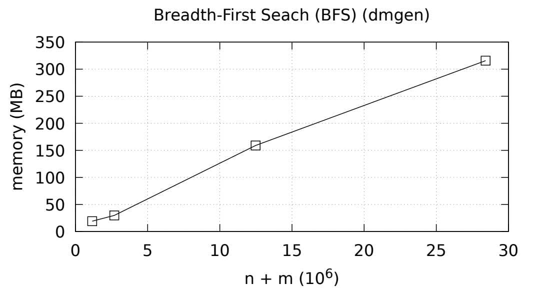
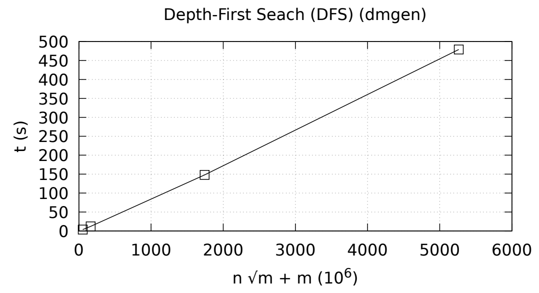


Figure 10: BFS operation average time and average memory resident peak (webgraph) for synthetic datasets.



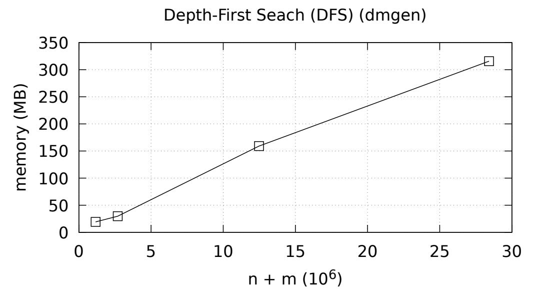


Figure 11: DFS operation average time and average memory resident peak (webgraph) for synthetic datasets.

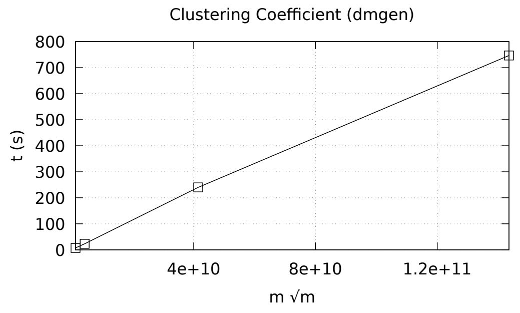
algorithm for computing both the clustering coefficient and counting triangles with an hash table. We present the evaluation for the computation of the (global) clustering coefficient in 12. On the left side we have the execution time while on the right side we have the peak resident memory. The theoretical and empirical complexities were in tune as we tested with bigger datasets. 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 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 $\mathcal{O}(m\sqrt{m})$.

5.3 Count Triangles Visiting Neighbors

Since we can answer queries on edge existence with proposed data structures in $\mathcal{O}(\log_2(n)\log(m))$ time, we implemented an algorithm for counting triangles using edge queries directly against the data structure, and without relying on a hash table. Results are provided in 13, being accordingly to the expected theoretical bounds. Note that the expected running time becomes now $\mathcal{O}(m\sqrt{m}\log_2(n)\log(m))$ since we no longer can 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 being essentially the space required to the compact graph data structure.

5.4 PageRank

For the pageRank algorithm we plotted with the duplication model graphs. We did not test with the Web Graph datasets since the



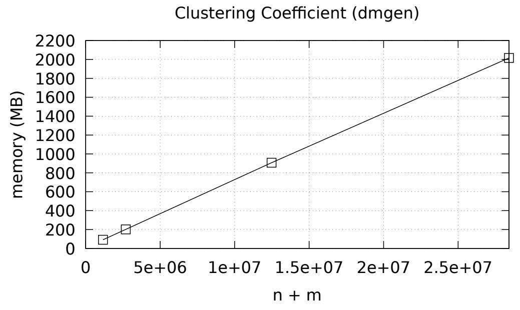
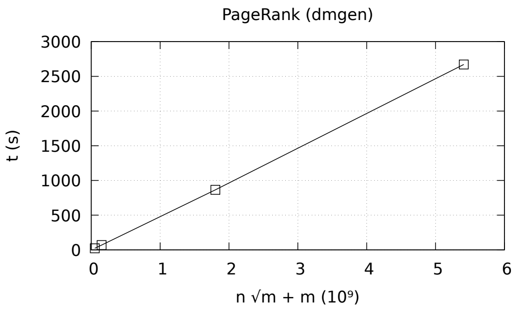


Figure 12: Clustering coefficient time (`dmgen`) and memory peak usage (`webgraph`) for the synthetic dataset.



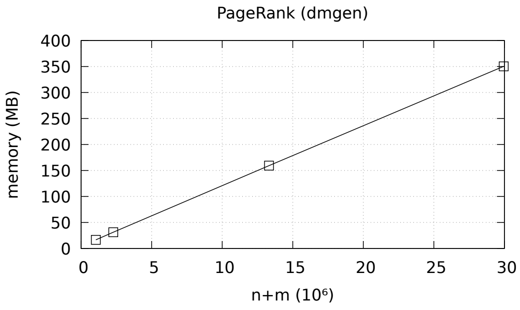
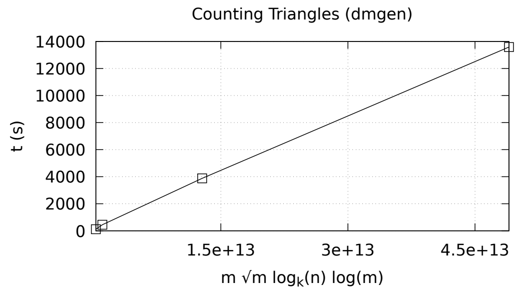


Figure 14: PageRank time (`dmgen`) and resident peak memory (`webgraph`) for synthetic datasets.



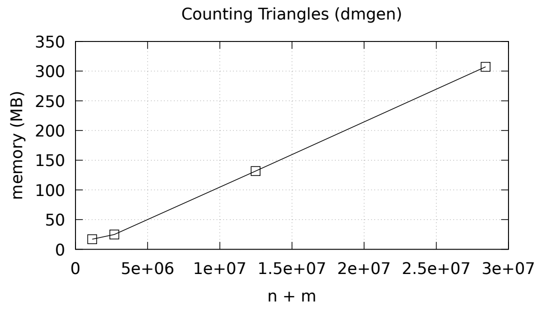


Figure 13: Counting triangles with neighborhood iterator time (`dmgen`) and memory peak usage (`webgraph`) for synthetic datasets.

number of iterations to converge the algorithm is different due to the different graph properties. In 14 we show the computation for the pageRank algorithm with our data structure. On the right side we have the execution time and on the left side we have the peak resident memory. The theoretical and empirical memory complexity is in tune as we tested with bigger datasets.

6 CONCLUSION

In this work we provide a tested and refactored C++ version of the SDK library while also extended its functionality by implementing edge and neighborhood iterators. To achieve this, we used the bit vector and k^2-tree from SDSL [12]. Nonetheless, the union operation, edge and neighborhood iterators were lacking in this library, so we had to implement them too. We started our experimental analysis with the union operation. In fact, we compare our implementation against the SDK's. We have seen that our implementation was faster for k^2-trees with less edges, although it presented a faster growth. So overall the SDK union operation was faster and also presented a smaller maximum resident memory peak for all datasets. Indeed, the higher memory occurred for all studied operations due to our representation having a higher memory consumption in the k^2-tree, as seen during the study of the union operation. Regarding the comparison between our implementation and the SDK's, first we analysed the list neighbors and check the presence of a link, where our implementation showed better time performance and a higher memory usage. Also, we performed the same study for the add and delete operations. For the add operation, our implementation showed better results for the synthetic datasets than the Web Graphs due to our union operation being faster for k^2-trees with less edges and slower otherwise. Last but not least, our implementation performed better for the delete operation for the same reason. In addition, we conducted a study on the addition operation followed by Munro's et al. [14] suggestion. From this work, two implementations emerged: one that delays the (full) union operations to the next addition operation and a more granulated version where all the unions cycle's are split and distributed among the next $|E_0|$ add operations. In fact, the Munro's version takes slightly more time and memory than the previous version since it is needed to copy the k^2-tree collections before the rebuild process, in order to keep the coherence of the data structure. Moreover, two parallel versions were implemented. The first one which at the time of rebuilding the k^2-tree collections when the E_0 is full and the previous rebuild is unfinished, instead it inserts the new edge in the E_0, increasing its size. This parallel version demonstrated to be fast for less dense graphs (`dmgen`) while consuming three times more memory than any other implementation. For the Web Graphs datasets, the need for rebuilding the E_0 demonstrated to be really costly for both in time and space, being out performed by all the other implementations in this environment. Thereafter, a second parallel implementation was also studied. This implementation waits when the E_0 is full while the previous rebuild is unfinished. The results for this version were highly linked with the size of the bit vectors of the k^2-trees, since for the synthetic datasets, the wait periods were higher as the unions in the background thread took longer to perform while in the real Web Graph datasets, since the bit vectors of the k^2-trees were smaller, consequently the union operations took less time to perform paving the way for this version to out perform all other versions in these datasets. As our intention was to develop a graph library, we also implemented some well-known algorithms for basic search, namely the BFS and the DFS and also some web graph's related algorithms: the Clustering Coefficient, two different counting triangles algorithms and finally the pageRank algorithm. The results show that our algorithm implementations were correct, since all plots' curves were straight lines. Moreover, we conducted a small study over the Counting Triangles algorithms. We compared two different implementations; one used an edge table to query the existence of an edge in $\mathcal{O}(1)$ and another implementation where we used the neighborhood iterator over the data structure. Although the neighborhood version is slower, the collected data shows that the maximum amount of memory used in the hash implementation was `2.2GB` while in the neighborhood version was `350MB`.

7 LIMITATIONS AND FUTURE WORK

An additional method that could be added is the processing of the reverse neighborhoods for a node: both the method and the iterator. This could be easily done since SDSL already offers this method for a k^2-tree. Another interesting operation that could be implemented is the common neighbors between two vertices since it is widely used in social graphs. There are common intuitions about how social graphs are generated, for example, it is common to talk informally about nearby nodes sharing a link. Our data structure is well suited for parallelism. Since we have r k^2-tree sub-collections the read operations such as listing neighbors and checking if the graph contains an edge. Since they are read-only operations these could be computed in parallel, likely improving the performance of these operations. The purpose of any library is to be used. In our case, more general graph and Web Graph algorithms could be introduced to the `Algorithm` class. Finally, the dynamic structure implemented in this work with k^2-trees, where its composition consists in an uncompressed container E_0 and a collection of the static data structures C, can be implemented with other static data structures such as the `WebGraph` [1]. Thus, we could add dynamism to `WebGraph` where C is composed several containers of `WebGraphs` instead of k^2-trees.

REFERENCES

[1] P. Boldi and S. Vigna. 2004. The Webgraph Framework I: Compression Techniques. In *Proceedings of the 13th International Conference on World Wide Web* (New York, NY, USA) *(WWW '04)*. Association for Computing Machinery, New York, NY, USA, 595–602. https://doi.org/10.1145/988672.988752

[2] Boost. 2015. Boost C++ Libraries. http://www.boost.org/. Last accessed 2020-09-12.

[3] Nieves R. Brisaboa, Susana Ladra, and Gonzalo Navarro. 2009. k2-Trees for Compact Web Graph Representation. In *String Processing and Information Retrieval*, Jussi Karlgren, Jorma Tarhio, and Heikki Hyyrö (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 18–30.

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

[5] Vijay Chandru. 1996. *Foundations of Software Technology and Theoretical Computer Science: 16th Conference, Hyderabad, India, December 18-20, 1996, Proceedings.* Vol. 16. Springer Science & Business Media.

[6] Fan Chung, Linyuan Lu, T Gregory Dewey, and David J Galas. 2003. Duplication models for biological networks. *Journal of computational biology* 10, 5 (2003), 677–687.

[7] Francisco Claude and Susana Ladra. 2011. Practical Representations for Web and Social Graphs. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management* (Glasgow, Scotland, UK) *(CIKM '11)*. Association for Computing Machinery, New York, NY, USA, 1185–1190. https://doi.org/10.1145/2063576.2063747

[8] M. E. Coimbra, A. P. Francisco, L. M. S. Russo, G. De Bernardo, S. Ladra, and G. Navarro. 2020. On Dynamic Succinct Graph Representations. In *2020 Data Compression Conference (DCC)*. 213–222. https://doi.org/10.1109/DCC47342.2020.00029

[9] Gabor Csardi and Tamas Nepusz. 2006. The igraph software package for complex network research. *InterJournal* Complex Systems (2006), 1695. http://igraph.org

[10] Eduardo F. D'Azevedo, Mark R. Fahey, and Richard T. Mills. 2005. Vectorized Sparse Matrix Multiply for Compressed Row Storage Format. In *Computational Science – ICCS 2005*, Vaidy S. Sunderam, Geert Dick van Albada, Peter M. A. Sloot, and Jack J. Dongarra (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 99–106.

[11] Santo Fortunato. 2010. Community detection in graphs. *Physics reports* 486, 3-5 (2010), 75–174.

[12] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. 2014. Succinct Data Structure Library - SDSL. https://github.com/simongog/sdsl-lite.

[13] Jure Leskovec and Rok Sosič. 2016. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)* 8, 1 (2016), 1.

[14] Ian Munro, Yakov Nekrich, and Jeffrey Scott Vitter. 2015. Dynamic data structures for document collections and graphs. In *Proceedings of the 34th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 277–289.

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

[16] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web.* Technical Report. Stanford InfoLab.

[17] Naila Rahman, Rajeev Raman, et al. 2006. Engineering the LOUDS succinct tree representation. In *International Workshop on Experimental and Efficient Algorithms*. Springer, 134–145.

[18] Keith H Randall, Raymie Stata, Rajiv G Wickremesinghe, and Janet L Wiener. 2002. The link database: Fast access to graphs of the web. In *Proceedings DCC 2002. Data Compression Conference*. IEEE, 122–131.

[19] Robert Sedgewick. 2002. Algorithms in C, part 5. *Reading-MA, USA: Addison-Wesley, Chapters* 17 (2002), 18.