TÉCNICO LISBOA

A graph algorithm library based on compact data structures

Joana Modesto Hrotkó

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Alexandre Paulo Lourenço Francisco

Examination Committee

Chairperson: Prof. José Luís Brinquete Borbinha
Supervisor: Prof. Alexandre Paulo Lourenço Francisco
Member of the Committee: Prof. Francisco João Duarte Cordeiro Correia dos Santos

January 2021

Acknowledgments

I would like to acknowledge my dissertation supervisor Prof. Alexandre Paulo Francisco and PhD colleague Miguel Coimbra for their insight, support and sharing of knowledge that has made this Thesis possible. It was a great privilege and honor to work and study under the guidance of Prof. Alexandre Paulo Francisco and I am extremely grateful for what he has offered me.

I am extending my heartfelt thanks to my long-term boyfriend Joaquim Espada for his love, patience and support during the discussions I had with him on research work and thesis preparation.

I wish to express my love and my gratitude to my parents, sister and grandmother for their constant support, encouragement, never-ending patience, without whose love, this work would never be possible.

Moreover, I would like to express my deep and sincere gratitude to my team and dear colleagues from Volkswagen Digital Solutions for helping and allowing me to conclude this important phase of my life. A heartfelt thanks to Andrej Petrovic, Laura Brehm, Rita Castro and Wilson Mália for helping me review this document.

Last but not least, to all my friends that helped me grow as a person and were always there for me during the good and bad times in my life. A special thanks to Nuno Sabino and Tomás Cunha for all the times we went through together and for giving friendship a new meaning.

To each and every one of you – Thank you.

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.

Keywords

Compact Representations; Dynamic Graphs; k^2-tree; Graph Library; Web Graphs

Resumo

Abordamos o problema de representação de grafos dinâmicos usando k^2-trees. A estrutura de dados k^2-tree é uma das estruturas de dados sucintas propostas para representar grafos estáticos e relações binárias em geral. Baseia-se em representações compactas de vetores de bits estáticos. Ao adicionar dinamismo às estruturas de dados compactas estáticas, também podemos representar grafos dinâmicos. No entanto, esta abordagem sofre de um problema bem conhecido da indexação dinâmica compactada, isto é o problema de manter uma coleção em mudança de modo a que possamos consultar a estrutura de dados de forma eficiente. Neste trabalho, apresentamos uma implementação baseada na k^2-tree que segue, em vez disso, as ideias de Munro para contornar esta questão. Refatorizámos e extendemos o trabalho de Coimbra construindo uma biblioteca em `C++`. Esta biblioteca inclui iteradores de arestas, nós e de vizinhança eficientes, bem como alguns algoritmos ilustrativos. Também incluímos um estudo sobre a operação de adição proposta inicialmente por Munro. Os nossos resultados experimentais mostram que nossa implementação é competitiva na prática.

Palavras Chave

Representações Compactas; Gráficos Dinâmicos; k^2-tree; Bibliotecas de Grafos; Web Graphs

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

API	Application Programming Interface
BGL	Boost Graph Library
BFS	Breadth-First Search
CSC	Compressed sparse column
CSR	Compressed sparse row
DFS	Depth-First Search
SDSL	Succinct Data Structure Library
SDK	Static Dynamic k^2-tree
STL	Standard Template Library
SNAP	Stanford Network Analysis Platform
URL	Uniform Resource Locator

1

Introduction

Contents

1.1 Motivation

Graphs are a natural way of modeling connections in the World Wide Web and social networks [4]. In this case, 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 operations over the graph include forward querying, finding the predecessors of a node and checking the presence of a link. Among them, forward querying is one of the most widely used graph operations. It can be used to determine the connection between two pages or two people, filter out all the pages linked by a specific page, determine a person's communication range, etc.

A graph is most commonly represented with an adjacency matrix or list. For small scale graph data, these two approaches can provide efficient querying. However, with the rapid development of the Internet and the extreme growth of the World Wide Web's scale, graphs are generating at an unprecedented pace and are accumulating a large amount of data. How to analyze and use these data has become a key opportunity and an extreme challenge for many fields. To satisfy the efficient operation of some basic algorithms and operations on large-scale graph data, in recent years, many scholars have designed many data structures with good performance for the compression storage of graphs and proposed algorithms to extend operations on these graphs. 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.

Web graphs, where nodes are Web pages and relations are hyperlinks, can be seen as a binary relation between two (usually equal) sets of Web pages A and B. In this context, basic binary relation operations are translated into queries to find the direct or reverse neighbors of a node. Consider a binary relation between two sets A and B, defined as a subset $R \subseteq A \times B$. Typical operations of interest in a binary relation are: determine whether a pair (a, b) is in R, find all the elements $b \in B$ such that $(a, b) \in R$, given $a \in A$, and vice versa. More sophisticated ones aim, for example, at retrieving all pairs $(a, b) \in R$ where $a \in [a1, a2]$ and $b \in [b1, b2]$.

1.2 The Problem

There are two natural ways to represent binary relations: a binary adjacency matrix or an adjacency list. On large binary relations, reducing space while retaining functionality is crucial in order to operate efficiently in main memory. Therefore, simple representations such as plain adjacency matrices are usually unfeasible in these datasets.

A topic that is strongly related to the problem of managing large volumes of data is compression, which seeks a way of representing data using less space. Most compression algorithms require decompressing all of the data from the beginning before we can access any element of the data structure. Therefore, compression generally serves as a space-saving archival method, although it is not useful for managing more data in main memory.

Compact data structures aim precisely at this challenge. According to Navarro et. al. [5] 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.

`WebGraph` [2] is a state-of-the-art framework that takes advantage of Web Graph properties in order to compress the data. Moreover, Brisaboa et. al. [6] introduced a compact data structure called k^2-tree. It was initially proposed for the compression of Web graphs, where it was shown to be very competitive. However, just like the other compressed representations of graphs and binary relations, including the `WebGraph`, k^2-trees are essentially static. This discourages their use in cases where the binary relation changes due to the insertion or deletion of edges.

1.3 Contributions

We propose an easy to use, tested and extendable Application Programming Interface (API) of the dynamic k^2-tree based on the implementation of Static Dynamic k^2-tree (SDK) [7]. Moreover, we have extended the Succinct Data Structure Library (SDSL) library [8] with the implementation of union operation and also edge, node and neighbour iterators for the k^2-tree data structure. Since we are proposing a graph library, we additionally propose the implementation of some well-known algorithms as an extension of the SDK library. This work also resulted in a paper submitted for publication in the journal of Information and Computation.

1.4 Organization of the Document

In this document we start by formalizing graph definition that will be used throughout the whole document in Chapter 2. Moreover, we will overview some simple graph representations such as the adjacency matrix and the adjacency list and also compressed representations which include the Compressed Sparse Row/Column. We continue this chapter by introducing the current state-of-art compact data structures, namely the Web graph, the main protagonist of this work – the k^2-tree and the dynamic k^2-tree. Then we

proceed to Chapter 3 where we will review the architecture of popular graph APIs. In Chapter 4 the solution for this work will be explained in detail. We are going to address the implementation specifics, the library overall structure and we also present the extended features. Next, in Chapter 5 we will evaluate our implementation against the SDK's by evaluating the overall performance regarding time and space. Not only will we compare but also assess the extended functionalities. We finish this work with Chapter 6, where we will give our final remarks and sum the whole work while also pointing out its limitations and future work.

2

Background

Contents

This chapter covers all the theoretical concepts applied in our work. We will start by giving a formal definition of a graph followed by its most common representations: the adjacency list and the adjacency matrix. Next, we will address compact structures starting with the Compressed Sparse Row and Column as an introduction to compressed graph representations and also cover the Web graph as an optimized compacted structure for binary relations in graphs. Thereafter, both the static and dynamic k^2-trees structure will be addressed, being the core data representation tackled in this work. By the end of this chapter, a general and final discussion covering all data structures is going to be presented.

2.1 Graph Definition

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 connects two vertices which are are not necessarily distinct, so we have $E \subseteq V \times V$. This data structure can be denoted as $G = (V, E)$ [9]. To avoid notational ambiguities, we shall always assume that $V \cap E = \emptyset$.

The number of vertices of a graph G is the graph's order, written $|G|$, while the number of edges is denoted by $||G||$. Graphs are finite or infinite according to their order. The empty graph ($\emptyset$, $\emptyset$) can be denoted as $\emptyset$. An edge is written as (v, w) where v and w are two vertices $v, w \in V$. A vertex v is incident with an edge e if $v \in e$; then e is an edge at v. The two vertices incident with an edge are its end vertices or ends. The set of all edges in E at a vertex v is denoted by $E(v)$. Two vertices v, w of G are adjacent or neighbours if (v, w) is an edge of G. Two edges that $e_1 \neq e_2$ are adjacent if they have an end in common. If all the vertices of G are pairwise adjacent, then G is complete. More formally, a set of vertices or of edges is independent (or stable) if no two of its elements are adjacent.

There are different types of graphs with different properties. A directed graph (or digraph) is a pair of (V, E) of disjoint sets of vertices and edges together with two maps: init: $E \to V$ and ter: $E \to V$. These assign to every edge e an initial vertex init(e) and a terminal vertex ter(e). The edge e is said to be $directed$ from init(e) to ter(e). Note that a directed graph may have several edges between the same two vertices v, w. Such edges are called multiple edges; if they have the same direction, they are parallel. If init(e)=ter(e), the edge e is called a loop. A directed graph D is an orientation of an (undirected) graph G which arises from an undirected graph simply by directing every edge from one of its ends to the other. In undirected graphs self-loop representations are forbidden, and so every edge consists of two distinct vertices. In undirected graphs the adjacency relation is symmetric while in directed graphs, the adjacency relation is not necessarily symmetric.

A graph can be defined as dense or sparse. Graphs with a number of edges roughly quadratic in their number of vertices are usually called dense and sparse otherwise. Depending on the type of the graph we can calculate how dense it is. For undirected graphs its density is given by $\frac{2|E|}{|V|(|V|-1)}$ while for

directed graphs is $\frac{|E|}{|V|(|V|-1)}$.

Finally we have the definition of weighted graphs. With each edge e of G let there be associated a real number $w(e)$, called its weight where w is a function defined as $w : E \to \mathbb{R}$. Then G, together with these weights on its edges, is called a weighted graph [10].

2.2 Graph Representations

In this section we will discuss the two most common ways to represent a graph $G = (V, E)$ as an adjacency matrices or as adjacency lists. Either way applies to both directed and undirected graphs.

We introduce a simple graph in Figure 2.1 which will be represented in different data structures throughout this chapter.

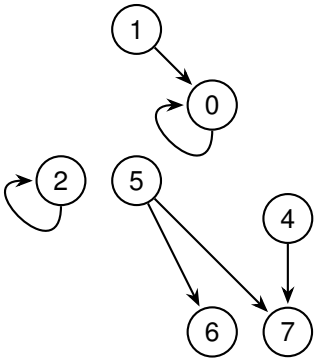

Figure 2.1: A simple directed graph.

2.2.1 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 whether there is an edge connecting two given vertices [11].

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 of 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 vertices v and w in the graph, and to be 0 otherwise [12]. The time to retrieve an edge in this representation is $\mathcal{O}(1)$ and memory usage is $\mathcal{O}(|V|^2)$.

This is the case for a directed graph or a digraph. However, for a undirected graph, each edge is actually represented by two entries: the edge (v, w) is represented by the value 1 in both $M[v][w]$ and $M[w][v]$. In this type of representation, generally we assume the number of vertices is known when the graph is initialized.

Additionally, for weighted graphs with edge-weight function f, we can simply store the weight $f(u, v)$ of the edge $(u, v) \in E$ as the entry in row u and column v of the adjacency matrix M. If an edge does not exist, we can store a NIL value as its corresponding matrix entry, though for many problems it is convenient to use a value such as 0 or ∞.

Taking the example graph from Figure 2.1, its adjacency matrix representation would be the following:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure 2.2: Adjacency Matrix for the graph in Figure 2.1

As we can see for this graph, we have a highly sparse matrix since most of the matrix's entries are 0.

2.2.2 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 [12].

A graph G consists of an array Adj of size $|V|$, that is one entry for each vertex in V. For each $u, v_i \in V$, the adjacency list $Adj[u]$ contains all the vertices v_i such that the edge $(u, v_i) \in E$. We are able to represent both directed and undirected graphs with an adjacency list. In a directed graph, the sum of the lengths of all the adjacency lists is $|E|$, since an edge of the form (u, v), is represented by having v appear in $Adj[u]$. If G is undirected, the sum of the lengths of all the adjacency lists is $2|E|$, since if (u, v) is an undirected edge, then u appears in v's adjacency list and vice versa. For both directed and undirected graphs, the adjacency list representation has the desirable property that the amount of memory it requires is $\Theta(V + E)$ [13].

We can readily adapt adjacency lists to represent weighted graphs. For example, let G be a weighted graph with weight function f. We simply store the weight $f(u, v)$ of the edge $(u, v) \in E$ with vertex v in u's adjacency list. The adjacency list representation is quite robust since we can modify it to support many other graph variants.

There are other ways to implement an 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 extra memory usage for the hash table, however it allows to search an edge in $\mathcal{O}(1)$ instead of $\mathcal{O}(|Adj[u]|)$.

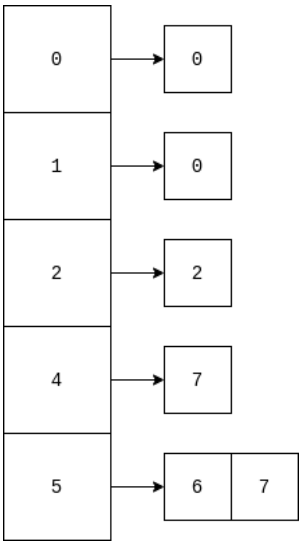

Figure 2.3: Adjacency List for the example graph in Figure 2.1.

The adjacency list for the graph in Figure 2.1 is represented in Figure 2.3. Comparing with the previous adjacency matrix representation in Figure 2.2 it is clear that this representation is more appropriate since the amount of memory used is much smaller than in the adjacency matrix.

2.3 Compressed and Compact Representations

In this section, we will address compressed and compact graph representations as an optimization for storage of the previous data structures. Taking from the previous representations, we will go further with more sophisticated data structures used to compress large graphs. First, we will present the compressed sparse row and column data structure which are the only compressed (and not compact) structures presented in this work. Thereafter, we proceed to the Web Graph data structure which uses more complex adjacency lists based on blocks of successors nodes and some graph properties. Finally, we end this section by presenting the k^2-tree data structure which is part of this work.

For simplicity, from this section until the end of this work we will refer to $|V|$ as n and $|E|$ as m.

2.3.1 Compressed Sparse Row and Column

Compressed sparse row (CSR) and Compressed sparse column (CSC) are widely known and the most used formats of sparse data structures. Mainly, they are used for write-once-read-many tasks. The Compressed Row and Column Storage formats are the most general: they make absolutely no assumptions about the sparsity structure of the matrix, and they don't store any unnecessary elements. On the other hand, they are not very efficient, needing an indirect addressing step for every single scalar operation in

a matrix-vector product.

2.3.1.A Compressed Sparse Row

In a sparse matrix, most of the entries are zeros. There are several ways to store a general sparse matrix. The CSR or Yale format [14] is commonly used to compress this kind of matrices.

The CSR represents a $a \times b$ matrix M by three linear arrays: `val`, `colInd` and `rowPtr`. The `val` array contains all the non-zero entries in M in row major order. The `colInd` array contains the column index in M of each element of `val`. Hence, its size will be the size of nnz, where nnz are the non-zero values. Finally we have the `rowPtr` array with size $a+1$ which stores the cumulative number of non-zero elements up to (not including) the i^{th} row. This array is defined by the following recursive relation:

$$\texttt{rowPtr[n]} = \begin{cases} 0 & \text{if } n \text{ is } 0 \\ \texttt{rowPtr}[n-1] + nnz \text{ in the } n+1th \text{ row} & \text{otherwise} \end{cases}$$

For instance, to find the number of non-zero elements in row i, we perform $\texttt{rowPtr}[i+1] - \texttt{rowPtr}[i]$.

However, this data structure is only memory efficient for matrices where the nnz are less than $(a \times (b-1)-1)/2$. The direct array representation requires b^2 memory, while the CSR requires $2\times\ nnz+a+1$.

From the matrix representation of the example we introduced in this chapter in Figure 2.2, we have calculated its CSR representation presented in Figure 2.4.

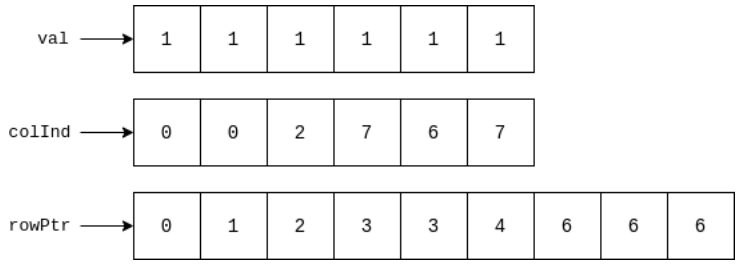

Figure 2.4: Compressed Sparse Row for the example graph from the matrix in Figure 2.2

Notice how the `val` array is redundant in this case as all entries are always $1s$ for binary representations. If our example graph was weighted, the `val` array would not be redundant.

2.3.1.B Compressed Sparse Column

Analogous to CSR, the CSC representation where the values are indexed first by column with a column-major order. The main difference to the previous representation is that the columns of M are stored instead of the rows. In other words, the CSC format is the CSR format for M^T.

The CSC format is specified by the arrays `val`, `colPtr`, `rowInd`, where `rowInd` stores the row indices of each nnz, and `colPtr` stores the index of the elements in `val` which start a column of M.

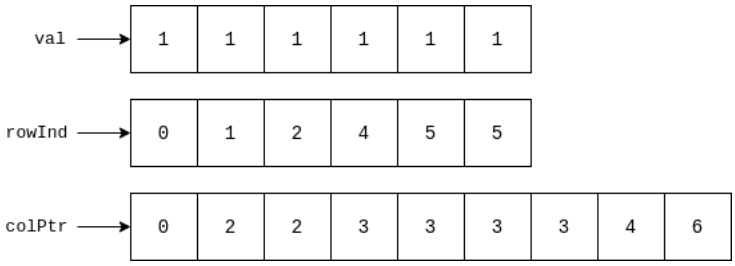

Figure 2.5: Compressed Sparse Column representation for the example graph from matrix in Figure 2.2.

Next, we will present the studied compact data structures.

2.3.2 Web Graph

The 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 [2]. Web graphs contain millions of nodes and although sparse, representing it with an adjacency matrix would be way too big to fit in main memory, even on computers with high resources. To overcome this difficulty, we can access the graph from external memory, which however, requires designing special offline algorithms even for the most basic problems (e.g. computing the shortest path). In order to solve this, the graph is never actually loaded into memory, but rather read in a streaming fashion from external storage using a small amount of memory.

In order to compress and efficiently store a Web Graph, the empirical observations in the structure of hyperlinks in a typical subset of the web are exploited. Two features of the Web Graph links are usually quoted as locality and similarity. These were originally exploited by the Connectivity Server [15] and LINK database [16]. 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, that is the index of the source and target are close to each other when ordering them lexicographicaly. Additionally, the **similarity** refers to the pages that occur close to each other (in lexicographic 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.

2.3.2.A LINK Database

The latter approach, which can be referred to as Web Graph compression, can be traced back to the LINK database [16]. This work presented techniques to compress the links in order to accommodate larger graphs, where some of them presented around 6 billion edges. They based their work on the Connectivity Server [15] which is a collection of three databases: the URL Database, the Host Database

and the Link Database. The URL Database contains the relation of URL ids with their host names. The Link Database maps a URL id to their sets of URL ids outlinks and inlinks. This database returns both outlinks and the inlinks of the graph in an adjacency list.

In the LINK database work the authors 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. In this work they took advantage of the similarity of the successors list by specifying the successor of the list of a node with a partial copy of a previous list and adding whatever remains. This is achieved using a list of bits, one for each successor in the referenced list, which marks whether the successor should be copied or not. This technique was introduced as the delta values where instead of storing the absolute values of the URL-ids in each adjacency list they stored the differences between the neighbors list, which are called the delta or gap values. Hence, they store much smaller values than the absolute values.

Together, all these techniques reduce space requirements to under 6 bits per link.

2.3.2.B WebGraph

More recently, the LINK database [16] has led to the development in `Java` of the `WebGraph` framework [2] which still provides some of the best practical compression-versus-speed trade-offs. Similar 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 above a certain threshold. Thus, each list of extra nodes will be compressed by using a list of integer intervals and a list of residuals which are only compressed using differences. This technique is named the differential compression, in which the differences between the neighbour lists are recorded by a sequence of copy blocks, that is the sequence of neighbour nodes is preceded by a block count indicating the number of blocks that will follow. This is possible because many of the neighbour lists have a considerable number of common nodes among them.

The differential compression allows the `WebGraph` to compress as little as 3.08 bits per link.

2.3.3 k^2tree

The k^2-tree [6] is a very efficient compressed data structure that competes directly with the `WebGraph` framework. The k^2-tree is a novel Web graph representation based on a compact tree structure that takes advantage of large empty areas of the adjacency matrix of the graph. It offers the least space

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

usage at 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 consists of two bit vectors[2]:

- T (tree) – 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 [6].

- L (leaves) – stores the last level of the tree. It represents the values of (some) original cells of the adjacency matrix.

The construction of this data structure starts from the adjacency matrix representation of a graph. It is applied a MX-Quadtree strategy [18], where the matrix is divided into k^2 submatrices of size n^2/k^2. During this phase the recursion process stops when we reach submatrices with size $k \times k$, representing the last level of the tree. In this level, each entry of the matrix corresponds to a leaf node ordered by a row major fashion and being stored in bit vector L. Regarding the middle levels, each cell represents the presence or absence of children nodes. For a given an internal node of the tree, if all k^2 child nodes are $0s$, then its value its value in T is 0. Otherwise, if it has at least one child node, its value is 1.

Therefore, each node contains 1 bit of data. Ideally, n is a power of k. If it is not, the matrix size is extended with enough 0's to the right and to the bottom, making the width $n' = k^{\lceil \log_k n \rceil}$ [19].

In order to build a k^2-tree from the example graph in Figure 2.1, we would build it from the adjacency matrix representation in Figure 2.2. Applying the MX-Quadtree strategy in Figure 2.6, we can build the a conceptual tree as presented in Figure 2.7 which will be saved as two bitmaps T (tree) and L (leaves).

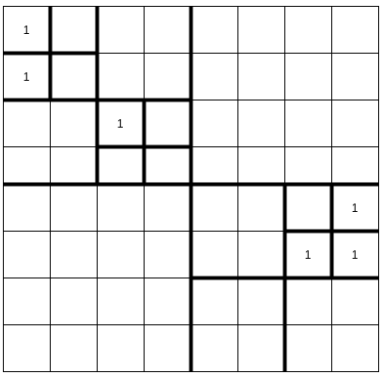

Figure 2.6: MX-Quadtree strategy applied in the matrix representation from Figure 2.2

[2]The Bit Vector or LOUDS representation [17] allows a representation of a tree based on an string of 1s and 0s. This data structure has two main operations: the $rank_1(x)$ which returns the number of 1 bits to the left of, and including, position x in the bit-string and $select_1(i)$ which given an index i, returns the position of the i^{th} 1 bit in the bit-string.

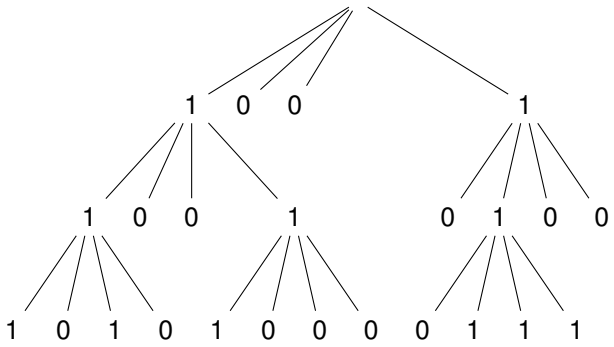

Figure 2.7: k^2-tree representation of the graph in Figure 2.1

The bit vector T stores all the bits from all the internal nodes except the leaves. The first bits of this array are the ones from the root following a breadth first order until the $h-1$ level. Finally, L stores the last level h of the tree. Therefore we have the following bit vector to represent our k^2-tree:

$$T = 1001\ 1001\ 0100,$$
$$L = 1010\ 1000\ 0111,$$
$$T||L = 1001\ 1001\ 0100\ 1010\ 1000\ 0111.$$

With the bit vectors representation, it is possible to perform queries directly from the compressed representation. 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 $rank_1(T, x) \times k^2 + i$ 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 supoprt $rank_1(T, x)$ 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 [20, 21].

2.3.3.A Construction Space and Time

The authors from [19] present an alternative build of a k^2-tree given different representations of a Web Graph. According to them, a k^2-tree can be built from an adjacency matrix with time $\mathcal{O}(n^2(1/k) + 1/w)$, where w is the length in bits of the computer word and takes $\mathcal{O}(s)$ space, where $s = |T| + |L|\ bits$. Moreover, they also present the possibility to build a k^2-tree from an adjacency list representation since it is more realistic in real life problems. This method has a time complexity of $\mathcal{O}(n^2/k^2 + m + s/w)$ and requires $\mathcal{O}(n)$ additional words for space.

From an adjacency matrix

Assume that our input is an $n \times n$ adjacency matrix. Construction of our tree is easily carried out bottom-

up in linear time and optimal space (that is, using the same space as the final tree). Their procedure builds the tree recursively. It consists of a depth-first traversal of the tree that outputs a bit array T_l for each level l of the tree. If we are at the last level $l = h$, we read the k^2 corresponding matrix cells. If all are 0, we return 0 and we do not output any bit string, as none of those 0s will be explicitly represented; otherwise we output the k^2 bits and return 1. If we are not at the last level, we make the k^2 recursive calls for the children. If all return 0, we return 0, otherwise we output the k^2 answers of the children and return 1. The overall algorithm is described in Algorithm 2.1.

Algorithm 2.1: `Build(n,l,p,q)`

```
for i ← 0...k − 1 do
  for j ← 0...k − 1 do
    if l = h then
      C ← C||a_{p+i,q+1}
    else
      C ← C||Build(n/k, l + 1, p + i × (n/k), q + j × (n/k))
    end if
  end for
end for
if C = 0^{k^2} then
  return 0
end if
T_L ← T_l||C
```

The output for each call is stored separately for each level, so that the k^2 bits that are output at each level are appended to the corresponding bit array T_l. As we fill the values of each level left-to-right, the final T is obtained by concatenating all levels but the last one, which is indeed L. Algorithm 2.1 shows the construction process. It is invoked as $Build(n' = k^h, 1, 0, 0)$, where the first parameter n is the (extended) submatrix width, the second, l, is the current level, the third, p, is the row offset of the current submatrix, and the fourth, q, is its column offset. After running it we have $T = T_1||T_2||...||T_{h-1}||$ and $L = T_h$.

The total time is clearly linear in the number of cells in the matrix, $\mathcal{O}(n^2)$. By accessing up to w consecutive (say, horizontal) bits of the matrix in one operation, the time can be reduced to $\mathcal{O}((n^2/k^2)k(1 + k/w))$ to scan the matrix plus $\mathcal{O}(n^2/k^2)$ for the recursive invocations, for a total complexity of

$$\mathcal{O}(n^2(1/k + 1/w)).$$

From an adjacency list

Representing the complete matrix for the construction process is not realistic on Web graphs, because the matrix is too sparse. We use the adjacency lists representation of the matrix instead, that is, for each Web page p we have the list of Web pages q such that p has a link pointing to q. By using the adjacency

list we can still achieve the same time by setting up n cursors, one per row, so that each time we have to access a a_{pq} we compare the current cursor of row p with value q. If they are equal, we know $a_{pq} = 1$ and move the cursor to the next node of the list of row p. Otherwise we know $a_{pq} = 0$. This works because all of our queries to each matrix row p are increasing in column value.

In this case we pay $\mathcal{O}(n^2/k^2)$ in recursive invocations to reach each leaf, plus $\mathcal{O}(t/w)$ to write down bitmap T, plus $\mathcal{O}(1 + k^2/w)$ to initialize each chunk of k^2 bits in L, plus $\mathcal{O}(1)$ time to go over each entry of the adjacency list and write the corresponding 1 in the chunk of L. This gives a total of

$$\mathcal{O}(n^2/k^2 + m + t/w + \ell(1/k^2 + 1/w)) = \mathcal{O}(n^2/k^2 + m + s/w).$$

2.3.3.B Check individual Link

For most compressed graph representations, to determine if page p is linked to page q we have no choice but to extract all the successors of p and check if q is in the set. However, in the k^2-tree representation we can answer such queries in $\mathcal{O}(\log_k n)$ time by descend to one child at each level of the tree, so that we can determine if the cell a_{pq} of the adjacency matrix is 1 or 0. We start at the root node and descending recursively to the child node that represents the submatrix containing the cell a_{pq} of the adjacency matrix.

Recall that $h = \lfloor \log_k n \rfloor$ is the height of the tree. Then the nodes at level l represent square submatrices of size k^{h-l}, and these are divided into k^2 submatrices of size k^{h-l-1}. For instance, the root at level $l = 0$ represents the whole square matrix of width $k^h = n'$. Let us call p_l the relative row position of Web page p at level l, and q_l the relative column position of Web page q at level l. Cell (p_l, q_l) at a matrix of level l belongs to the submatrix at row $\lfloor p_l/k^{h-l-1} \rfloor$ and column $\lfloor q_l/k^{h-l-1} \rfloor$. This corresponds to the child number $k \times \lfloor p/k^{h-l-1} \rfloor + \lfloor q/k^{h-l-1} \rfloor$ of the node that represents the matrix at level l. The relative row position for Web page p in this child node is $p_l + 1 = p_l \mod k^{h-l-1}$. The relative column position for Web page q is $q_l + 1 = q_l \mod k^{h-l-1}$. Hence, starting from the root node at level $l = 0$, where $p_0 = p$, $q_0 = q$, at each level l we descend to child $k \times \lfloor p/k^{h-l-1} \rfloor + \lfloor q/k^{h-l-1} \rfloor$, if it is not a zero, and compute the relative position of cell (p, q) in the submatrix. If we reach the last level and find a 1 at cell (p, q), then there is a link, else there is not. The pseudo code looks much simpler in Algorithm 2.2.

The worst-case navigation time to check if a Web page p points to another Web page q is

$$\mathcal{O}(\log_k n)$$

since a full traversal from the root node to a leaf node is required for any pair of connected Web pages. Besides, the time can be even lower for nonexistent links.

Algorithm 2.2: `Check_Link(n,p,q,z)`

if $z \geq |T|$ **then**
  return $L[z - |T|]$
else
  if $z = -1$ **or** $T[z] = 1$ **then**
    $y \leftarrow rank(T, z) \times k^2$
    $y \leftarrow y + \lfloor p/(n/k) \rfloor \times k + \lfloor q/(n/k) \rfloor$
    $Check\_Link(n/k, p \mod (n/k), q \mod (n/k), y)$
  else
    return 0
  end if
end if

2.3.3.C Successor and Predecessor

To find the successors or predecessors of a page p we need to locate which cells in row a_{p*} (column a_{*q}) of the adjacency matrix have a 1. Again, these are obtained by a top-down tree traversal, but instead of choosing just one child node, as for single-link queries, the algorithm must choose k out of the k^2 children of a node. As before, we describe the formula that maps global row numbers to the children numbers at each level. Being p_l the relative row position of interest at level l, row p_l of the submatrix of level l corresponds to children number $k \times \lfloor p_l / k^{h-l-1} \rfloor + j$, for $0 \leq j < k$. Similarly, column q in level l corresponds to children number $j \times k + \lfloor q_l / k^{h-l-1} \rfloor$, for $0 \leq j < k$. The pseudo code for both methods can be found in Algorithm 2.3 and Algorithm 2.4.

Algorithm 2.3: `Successors(n, p, q, z)`

if $z \geq |T|$ **then**
  if $L[z - |T|] = 1$ **then**
    return q
  end if
else
  if $z = -1$ **or** $T[z] = 1$ **then**
    $y \leftarrow rank(T, z) \times k^2 + k \times \lfloor p/(n/k) \rfloor$
    for $j \leftarrow 0...k - 1$ **do**
      $Successor(n/k, p \mod (n/k), q + (n/k) \times j, y + j)$
    end for
  end if
end if

The navigation time to retrieve a list of successors or predecessors has no worst-case guarantees better than $k^h = \mathcal{O}(n)$, as a row $p - 1$ full of $1s$ followed by p full of $0s$ could force a Successors query on p to go until the leaves across all the row, to return nothing. All in all, the time complexity for this method is on average [6]

$$\mathcal{O}(\sqrt{m}).$$

Algorithm 2.4: `Predecessors(n, p, q, z)`

```
if z ≥ |T| then
  if L[z − |T|] = 1 then
    return q
  end if
else
  if z = −1 or T[z] = 1 then
    y ← rank(T, z) × k² + ⌊q/(n/k)⌋
    for j ← 0...k − 1 do
      Successor(n/k, q mod (n/k), p + (n/k) × j, y + j × k)
    end for
  end if
end if
```

2.4 Dynamic k^2tree

The dynamic k^2-tree introduces the insertion and deletion operations without having to decompress the whole data [1]. 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, E_0 is represented through a dynamic and uncompressed adjacency list in order to achieve the optimal amortized cost for each operation. Thus, we must control the number of edges in 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 [1].

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 $m_r = m/\log^{2-r\varepsilon} m$ and $m_r \leq m$ which implies that $r \leq 2/\varepsilon$, when m is at least 3. In [7] 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 the maximum number of edges follow 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 [19], where $m_i \leq m/\log^{2-i\varepsilon} m$. Hence, overall the space required is [7]

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

The operations supported by this data structure are insertions, deletions, listing neighbors of a node and checking the existence of a link. The operations of listing the neighborhood and querying the existence of a link are computed by querying firstly in the E_0 and afterwards in the rest or all the E_i if necessary. Thus the querying cost increases by a factor of $\mathcal{O}(1/\varepsilon)$ from the static version of the k^2-tree.

The insertion and deletion operations are not as intuitive as these operations.

2.4.1 Insertions

As previously mentioned, we represent E_0 as an adjacency list and C as a collection of k^2-trees. The insertion is carried out depending on the current size of E_0. If $m_0 < |E_0|$, 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, which is the sub-collection that can accommodate all edges from E_0 until E_j. The rebuild of the data structure is accomplished by performing successive unions. The pseudo code for the insertion operation is shown in Algorithm 2.5.

Algorithm 2.5: `Insert(u,v)`

```
if |E_0| < m_0 then
    E_0.add_edge(u, v)
else
    sum_m ← 0
    for 0 < i < r do
        sum_m ← sum_m + m_i
        if sum_m ≥ m_j then
            break
        end if
    end for
    tree_new ← E_0.to_k2tree()
    for 0 < j ≤ i do
        tree_new ← Union(tree_new, E_j)
        E_j.clear()
    end for
    E_i ← tree_new
end if
```

So if $m_0 < |E_0|$, then the insertion takes constant time since we are relying on an adjacency list coupled with a hash table to maintain adjacencies. Otherwise, we need to build a k^2-tree from E_0 which takes $\mathcal{O}(m_0 \log_k n)$ time [19]. Afterwards, we need to find some E_j that has enough capacity to accommodate the edges from E_0 plus all the previous collections E_i, for $0 < i \leq j$. The pairwise union of at most j k^2-trees representing collections E_0, ..., 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 [1] for the general case with $0 < j \leq r = \lfloor 2/\varepsilon \rfloor$, giving a time complexity of:

$$\mathcal{O}((1/\varepsilon) \log_k n \log^{\varepsilon} n).$$

2.4.2 Deletions

Similarly to the Insertion operation, the deletion operation also takes into consideration 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 $0 < j \leq r$ 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 $m' > n/\log(\log n)$ edges were marked. Once we reach this value we need to rebuild C again. For easier understanding we show the algorithm in Algorithm 2.6.

Algorithm 2.6: `Delete(u,v)`

```
if E_0.contains_edge(u, v) then
    E_0.delete_edge(u, v)
else
    total_marked ← 0
    for 0 < i < r do
        if E_i.contains_edge(u, v) then
            E_i.delete_edge(u, v)
            total_marked ← total_marked + E_i.marked_edges()
        end if
    end for
    if m' > n/log(log n) then
        rebuild(C) {C = E_1, ..., E_r}
    end if
end if
```

Deleting an edge in E_0 takes constant time. Checking and deleting an edge in our collections takes $\mathcal{O}(\log_k n)$, since checking if an edge exists in a given k^2-tree takes $\mathcal{O}(\log_k n)$ [19], 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, marking it for deletion takes constant time. However, in need of rebuilding after deleting $m/\log\log n$ edges costs in this case is $\mathcal{O}(m \log_k n)$, since it has an amortized cost of $\mathcal{O}(\log_k n \log(\log n))$ per deleted edge. Overall deleting an edge has an amortized cost of [7]

$$\mathcal{O}((1/\varepsilon + \log\log n) \log_k n)$$

2.5 Discussion

Graph Representation

Previously we have described the two most common ways to represent graphs in Section 2.2.1 and Section 2.2.2. We have presented the adjacency matrix representation which provides constant access time, although the space required in worst case is $\mathcal{O}(n^2)$. Then we introduced the adjacency list which is preferable for sparse graphs (m is much less than n^2), however if m is close to n^2, we would choose the adjacency matrix, since in any case we should use $\Theta(n^2)$ memory. Nonetheless, representing an adjacency list with a coupled hash table can reduce the time for finding an edge to $\mathcal{O}(1)$ while not using extra memory. In Table 2.1, is taken into account the worst-case costs all within constant factors for large n and m [12].

	Adjacency Matrix	Adjacency List	Adjacency List (Hash)
Space	n^2	$n+m$	$n+m$
Build	n^2	n	n
Insert edge	1	1	1
Remove edge	1	n	1
Check edge	1	n	1
List neighbors	n	n	n

Table 2.1: Worst case time complexity for adjacency matrix and adjacency list.

In order to add a new vertex in the matrix representation the storage must be increased to $\mathcal{O}((n+1)^2)$. Besides, to achieve this we need to copy the whole matrix and therefore the complexity is $\mathcal{O}(n^2)$. In the list representation, if there are two pointers in the adjacency list, one that points to the head node and the other one that points to the rear node, the insertion of a vertex can be done directly in $\mathcal{O}(1)$ time. All in all, the adjacency matrix is a better representation for static and dense graphs while the adjacency list provides a more feasible dynamic representation.

However, in our case, we are working with very large graphs. A Web Graph consists of millions of nodes and trillions of edges requiring large computing machinery. Web Graphs pose many issues such as storage, scalability and processing. Thus, these representations do not meet the requirements to handle these kinds of graphs, especially memory wise. For instance, if we have a graph with 1 million nodes and 10 million edges, the space required in the adjacency matrix representation would be $(10^6)^2 \times 4\texttt{bytes} = 3.63\texttt{TB}$ and for the adjacency list would be $(10^6 + 10^7) \times 4\texttt{bytes} = 4.19\texttt{MB}$.

Compressed and Compact Representations

On large binary relations, reducing space while retaining functionality is crucial in order to operate efficiently in main memory. Therefore, simple representations such as plain adjacency matrices are usually unfeasible in these cases. In the case of the adjacency list, it can efficiently compress sparse binary relations, however it usually lacks the ability to efficiently retrieve information on ranges of elements.

Several different compressed and compact graph representations were presented in this chapter. We started by presenting the CSR and CSC as the simplest data structure able to compress a graph from an adjacency matrix. In our case we only intend to represent binary relations so the val component becomes redundant since all its values would be 1. In this data structure a link is represented by 32 `bits per link`. Moreover, this data structure can only be used to store a graph without allowing to query directly over the data structure. In this data structure, this is only possible if the graph is uncompressed from main memory. Consequently, the CSR and CSC are not considered compact data structures but compressed data structures.

The limitations of simple data structures has led to different proposals for compressing general binary relations, as well as specific ones such as Web graphs [2, 15, 16], k^2-trees [6, 22] and dynamic k^2-trees [1].

In the LINK Database [16] and `WebGraph` [2], each node is a URL, and a directed arc from node x to node y whenever there is a hyperlink in page x leading to page y. For this specific problem, these data structures exploit the inner redundancy of the web when represented as an adjacency list. Unlike the CSR and CSC, these compact data structures allow querying over the compressed data without having to uncompress the graph. Furthermore, the work of `WebGraph` [2] exploits the storing data in blocks which allows a better compression rate of 3.08 `bits per link`.

We also presented the k^2-tree [6, 22] which was originally designed to represent Web Graphs. In fact, the k^2-tree data structure provides a compact representation of a graph. To reduce the space requirements for sparse graphs, a hierarchical decomposition is used where a sub-division consisting only of zeros is represented by a single 0 `bit`. Just like the `WebGraph`, it takes advantage of particular characteristics of the Web Graphs such as the existence of large areas with a high density of ones or zeros. Therefore it achieves a very small space between 1.3 and 3 `bits per link`. Thereafter, this data structure also provides direct operations from the compressed form without having to uncompress the whole graph from main memory. A sublinear number of extra bits are needed to enable constant-time rank operations on the bitmaps [5, 20, 21], which allows the representation to test the existence of a single edge in $\mathcal{O}(\log_k n)$ time and retrieve the successors/predecessors of a node in $\mathcal{O}(\sqrt{m})$.

Finally, we presented the dynamic k^2-tree [1] where the main idea is to represent a graph dynamically while supporting edge insertions and deletions, as well as common operations over graphs. This is achieved by using a collection of static edge sets $C = \{E_0, ..., E_r\}$. 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. In order to achieve the optimal amortized cost for each operation, we must control the number of edges m_i in each set E_i and the number r of such sets. Not only must we control the number of edges in each E_i in C but also in E_0 which must contain at most $m/\log^2 n$ elements. In general, we require that m_i is at most $m/\log^{2-i\varepsilon} n$, for some constant $\varepsilon > 0$. Moreover, we must also satisfy

$m_r = m/\log^{2-r\varepsilon} n \leq m$. Notice how if ε is a constant, so is r. Hence, the maximum number of edges per static set follows a geometric progression. The dynamic data structure supports additions and removals with competitive performance. The amortized cost to insert an edge is $\mathcal{O}((1/\varepsilon)\log_k n \log^{\varepsilon} n)$ and for removing an edge is $\mathcal{O}((1/\varepsilon + \log\log n)\log_k n)$. Thanks to this structure, 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 $\mathcal{O}(1/\varepsilon)$.

The overall compression ratios of these four structures are summarized in Table 2.2. We do not yet present the compression ratio of the dynamic k^2-tree. Nonetheless, we will be discussing it in Chapter 5.

Data Structure	`bits per link`
Compressed Row/Column	32
LINK Database	6
WebGraph	3.08
k^2-tree	$1-3$

Table 2.2: Compression ratio in `bits per link` for the Compact Data Structures presented in this Chapter.

The k^2-tree and `WebGraph` data structures have the most efficient compression ratios of those presented in the table. In the current state of the art, these static data structures are the most efficient compact structures offering a great compression ratio as well as allowing direct queries over the compressed graph. The dynamic k^2-tree, however, is the only data structure that provides the operations of insertion and deletion in addition to the same queries that k^2-tree offers with an extra factor of $\mathcal{O}(1/\varepsilon)$.

In the next chapters we will evaluate the average time and memory usage of the dynamic k^2-tree and in Chapter 6 we will compare our data structure with the ones in Table 2.2.

3

Graph APIs

Contents

In this chapter we will cover different approaches used to design graph APIs. This is a concept in software that essentially refers to how multiple applications can interact with each other and obtain data from one another. APIs operate on an agreement of inputs and outputs. So the main objective of this chapter is to understand the most adequate methods that should be implemented for a graph API. To attain this goal, we must take into consideration the data structures' performance for different types of queries demanded by the API's implementation while maintaining the time and space requirements.

3.1 Libraries

Libraries is a well-defined interface by which the behavior is invoked. For instance, whomever wants to write a higher-level program can use a library to make system calls instead of implementing those system calls over and over again. 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.

Typically, libraries have their data concepts wrapped around **Containers** [23]. These are abstract data types whose instances are collections of other objects. An important aspect of containers is that they all have a unified interface for accessing and transversing the object structure. This common interface is typically used by other methods or to implement more advanced algorithms. Thus the implementation of new algorithms is simplified since each method is implemented in each different type of object, so the same algorithm can be substituted with a different kind of object container without having to re-implement the same code.

Another important concept relative to libraries are the iterators. The **Iterator** [24] is an object that enables to transverse an aggregated container, giving access to the data elements of a container. This pattern decouples algorithms from containers without exposing its underlying representation. In `C/C++` pointers themselves are iterators. Taking the example from the container `vector`, it declares the nested types `iterator` and `const_iterator`, that allows to iterate over its elements.

Algorithms provide a variety of functionalities over the Containers. Typically these are implemented with the Visitor pattern [25]. Hence, the algorithm is separated from an object structure on which it operates. A practical result of this separation is the ability to add new operations to existing object structures without modifying the structures. Algorithms are templates, and are parameterized by the type of iterator, so they are not restricted to a single type of container.

3.2 Boost Graph Library

The Boost project [26] is a set of C++ libraries that have special emphasis on developing industrial-strength, high performance software using modern programming languages and techniques, having into special account generic programming. For this work we will focus on the Boost Graph Library (BGL) since it is dedicated to generic graph programming providing many graph structures and algorithms. Considering that it is a generic library it follows heavily the concepts used in the Standard Template Library (STL), which is a C++ library of container classes, algorithms, and iterators.

3.2.1 Containers

The structure of BGL provides several graph interfaces which intend to represent essentially the adjacency list for sparse graphs and the adjacency matrix representation for dense graphs. These representations are highly parameterized representations so that it can be optimized for different situations, for instance if the graph is directed or undirected or to allow efficient access to just the out-edges or for fast vertex insertion and removal at the cost of extra space overhead. For very large graphs that need to be compacted, they represent them in a compressed sparse row. This class does not provide any mutability, that is one cannot add or remove any vertexes or edges. The overall class hierarchy can be seen in Figure 3.1.

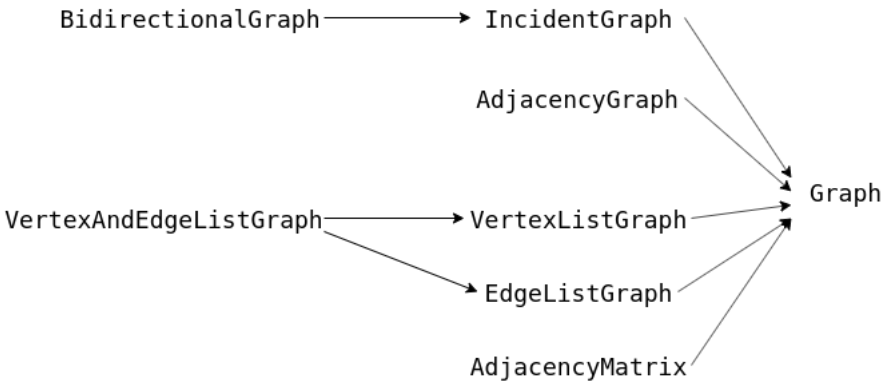

Figure 3.1: Graph concepts of Boost Graph Lobrary

The graph abstraction consists of several different kinds of collections: the vertices and edges for the graph and also the out-edges, in-edges and adjacent vertices for each vertex.

3.2.2 Iterators

In the BGL interface defines a function that returns a pair of iterator objects: the first iterator points to the first object in the sequence and the second iterator points past the end of the sequence as a typical C++

Iterators	
Vertex	transverses all vertices of the graph.
Edge	transverses all edges of the graph.
Out-Edge	accesses all of the out-edges for a given vertex u. Its value type is an edge descriptor. Each edge descriptor in this iterator range will have u as the source vertex and a vertex adjacent to u as the target vertex (regardless of whether the graph is directed or undirected).
In-Edge	is similar to the out-edge iterator, the difference relies in the in-edge access of a vertex u.
Adjacency	provides access to the vertices adjacent to a given vertex.

Table 3.1: Iterators decription from Boost Graph Library.

iterator implementation. In fact, there are five kinds of graph iterators, one for each kind of collection, presented in Table 3.1.

Iterators are used to access each of these collections. The reason for this is that the purpose of a concept is to summarize the requirements for particular algorithms.

3.2.3 Interface and Algorithms

Typically, algorithms do not need every kind of graph operation, but only a small subset. Due to the high-end use of the library, it is important to denote that during its implementation there were many graph data structures that could not provide efficient implementations of all the operations, but instead provide highly efficient implementations of the operations necessary for a particular algorithm.

The algorithms in BGL, several visitor concepts provide a mechanism for extending an algorithm for customizing what is done at each step of the algorithm. Visitors allow the user to insert their own operations at various steps within a graph algorithm. In BGL a wide range of types of algorithms are implemented such as search (Breadth-First Search (BFS) and Depth-First Search (DFS)), shortest path problems, minimum-spanning tree, connected components, minimum flow. Finally we show the most relevant methods of the interface provided by this library in the following Table 3.2.

3.3 SNAP

Stanford Network Analysis Platform (SNAP) [27] is a general-purpose graph library implemented in C++ and Python that provides high-level operations for analysis and manipulation of large networks.

3.3.1 Containers

In this library, the implementation is centered in both graph and network containers. In fact, it provides several types of graphs and networks, including directed and undirected graphs and multigraphs.

SNAP supports graphs and networks. Graphs describe topologies. That is nodes with unique integer ids and directed/undirected/multiple edges between the nodes of the graph. Networks are graphs with

VertexListGraph	
vertices	returns all vertices
vertex_iterator	returns a vertex iterator
EdgeListGraph	
edges	returns all edges in a graph
num_edges	returns the number of edges in a graph
edges_iterator	returns an edge iterator
source(edge)	returns a vertex_descriptor to u for the edge $= (u, v)$
target(edge)	returns a vertex_descriptor to v for the edge $= (u, v)$
IncidentGraph	
out_edges(vertex)	returns all the out edges for vertex.
out_edges_iterator(vertex)	returns an iterator for the out edges of vertex
out_degree(vertex)	returns the number of out edges for vertex
MutableGraph	
add_vertex	adds a new vertex to the graph
remove_vertex(vertex)	removes the vertex
add_edge(edge)	adds the edge to the graph
remove_edge(edge)	removes the edge
remove_edge(edge_iterator)	removes the edge that the iterator points to

Table 3.2: Boost Interface

data on nodes and/or edges of the network. Data types that reside on nodes and edges are simply passed as template parameters which provides a very fast and convenient way to implement various kinds of networks with rich data on nodes and edges. The graph containers are the following:

- **TUNGraph** - undirected graph
- **TNGraph** - directed graphs
- **TNEGraph** - directed multigraphs in which multiple edges can exist between a pair of nodes
- **TBPGraph** - bipartide graph

They also offer network containers:

- **TNodeNet** - directed graphs with node attributes
- **TNodeEDatNet** - directed graphs with node and edge attributes
- **TNodeEdgeNet** - directed multigraphs with node and edge attributes
- **TNEANet** - directed multigraphs with dynamic node and edge attributes

While designing SNAP data structures, it was required to be flexible and efficient during the manipulation of graphs, which means that adding or deleting nodes and edges had to be reasonably fast and not prohibitively expensive. This requirement is specially relevant for dynamic graphs, in which the graph structure is not known in advance, that is nodes and edges are added and deleted over time. Moreover,

high performance is mandatory in order to provide flexibility [27]. However, these two requirements are opposed to each other implementation wise.

The flexibility requirement is achieved by using hash table-based representations, while speed is achieved by using vector-based representations. As a result, SNAP graphs are represented by a hash table of nodes where each node has one or two vectors of adjacent nodes. For undirected graphs, adjacent nodes are represented using only one vector, while for directed graphs the nodes own two adjacent vectors, one for outcoming nodes and another for the incoming nodes. For simple graphs, edges are treated as a pair of nodes while in multigraphs edges have explicit ids so that two edges between the same pair of nodes can be distinguished. For the last case, an additional hash table is required to map edges ids to the source and destination nodes.

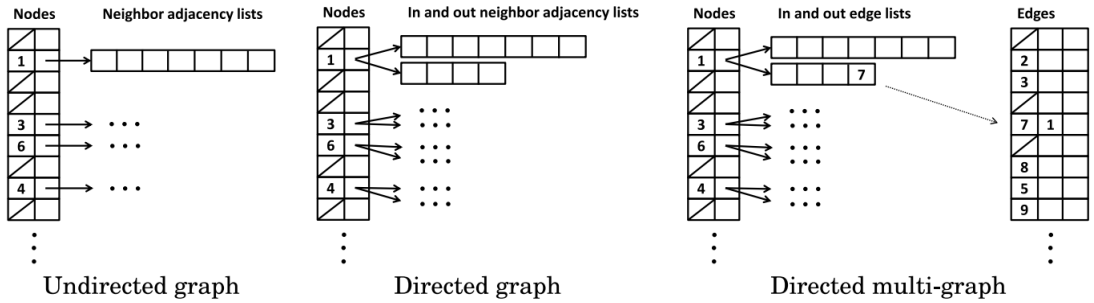

Figure 3.2: A diagram of graph data structures in SNAP. Node ids are stored in a hash table, and each node has one or two associated vectors of neighboring node or edge ids.

Taking into account that most real-world networks are sparse, with node degrees significantly smaller than the number of nodes. The benefits of maintaining sorted vectors significantly outweighs the overhead of sorting. Thus, maintaining the sorted values in the adjacency vectors was a design approach in order to allow fast access.

SNAP has proven to optimize memory usage for large graphs representation, even though it uses more memory for storing nodes than some alternative representations. Moreover, it requires less memory for storing edges. This design choice was chosen since for a vast majority of relevant networks have more edges than nodes.

3.3.2 Iterators

Many SNAP operations are based on node and edge iterators which allow for efficient implementation of algorithms that work on networks regardless of their type (directed, undirected, graphs, networks) and specific implementation. This library provides node and edge iterators.

They also provide a common interface among the iterators. The iteration starts with the method `Begin` which gives the first element of the iteration, then the `Next` method which increments the iterator

onto the next element and, finally, the `End` method which returns an integer or a string marking the element past the-end element.

3.3.3 Interface and Algorithms

SNAP has many specific types of graph and network implementations, providing a wide variety of classes. All graph classes have implemented the methods shown in Table 3.3.

Nodes	
`AddNode`	Adds a node
`DelNode`	Deletes a node
`IsNode`	Tests if a node exists
`GetNodes`	Returns the number of nodes
Edges	
`AddEdge`	Adds an edge
`DelEdge`	Deletes an edge
`IsEdge`	Tests if an edge exists
`GetEdges`	Returns the number of edges
Graph Methods	
`Clr`	Removes all nodes and edges
`Empty`	Tests if the graph is empty
`Dump`	Prints the graph in a human readable form
`Save`	Saves a graph in a binary format to disk
`Load`	Loads a graph in a binary format from disk
Node and Edge Iterators	
`BegNI`	Returns the start of a node iterator
`EndNI`	Returns the end of a node iterator
`GetNI`	Returns a node (iterator)
`NI++`	Moves the iterator to the next node
`BegEI`	Returns the start of an edge iterator
`EndEI`	Returns the end of an edge iterator
`GetEI`	Returns a edge (iterator)
`EI++`	Moves the iterator to the edge node

Table 3.3: SNAP graph methods

Moreover, SNAP also offers a wide range of algorithm implementations such as BFS, DFS, shortest paths, spanning trees, graph diameter, PageRank, core-periphery algorithms and much more.

3.4 igraph

We continue our research with `igraph` [28]. This library is heavily used in network analysis and other scientific domains such as biomedical research. The core implementation is written in `C` but most people use `igraph` through its high-level interfaces in `R`, `Python` and `Mathematica`.

3.4.1 Containers

This library has the representation of both directed and undirected graphs. The igraph graphs are multisets of ordered (if directed) or unordered (if undirected) labeled pairs of vertex ids, that is the edges. They present the type of igraph_t as their graph representation class. Moreover, they present their own implementations for vectors, stacks and queues.

3.4.2 Iterators

igraph shares the same base concepts as Boost's and STL iterators for nodes and edges as well as visitors applied to different algorithms.

Unlike the previously presented libraries, the igraph provides selectors. The selectors refer a sequence of vertices or edges independently from the graph. A vertex selector is a way to specify the class of vertices to be visited. Thereafter, it might specify that all vertices of a graph or all the neighbors of a vertex can be visited.

3.4.3 Interface and Algorithms

A condensed interface can be shown in Table 3.4.

Graph Methods	
igraph_empty	Creates an empty graph with some vertices and no edges.
igraph_vcount	The number of vertices in a graph.
igraph_ecount	The number of edges in a graph.
igraph_edge	Gives the head and tail vertices of an edge.
igraph_neighbors	Adjacent vertices to a vertex.
igraph_incident	Gives the incident edges of a vertex.
igraph_degree	This function calculates the in-, out- or total degree of vertices.
Adding and Deleting Vertices and Edges	
igraph_add_edge	Adds a single edge to a graph.
igraph_add_vertices	Adds vertices to a graph.
igraph_delete_edges	Removes edges from a graph.
igraph_delete_vertices	Removes vertices (with all their edges) from the graph.

Table 3.4: General igraph interface.

Regarding the algorithms from igraph library, BFS, DFS and random walks are some of the algorithms' visitors implemented in this library.

3.5 WebGraph

WebGraph [2] is a framework to study web graphs written in Java, C++ and Python. This library provides a simple way to manage very large graphs, exploiting modern compression techniques using a set of

simple codes, called ζ codes, which are particularly suitable for storing web graphs (or, in general, integers with a power-law distribution in a certain exponent range).

3.5.1 Containers and Interface

`WebGraph` provides several classes of graphs based on the `ImmutableGraph` abstract class. The authors highlight the fact that two `ImmutableGraphs` may not fit into main memory. This static representation of a graph demands that the subclasses of `ImmutableGraphs` implement certain methods presented in Table 3.5.

`ImmutableGraph`	
`numArcs`	Returns the number of edges of the graph
`numNodes`	Returns the number of nodes of the graph
`load`	Creates a new `ImmutableGraph` by loading a graph file from disk to memory
`loadMapped`	Creates a new `ImmutableGraph` by memory-mapping a graph file
`loadOffline`	Creates a new `ImmutableGraph` by loading offline a graph file
`store`	Stores the `ImmutableGraph`.
`nodeIterator`	Returns a node iterator for scanning the graph sequentially from a given node.
`outdegree`	Returns the number of successors of a given node
`sucessors`	Returns a lazy iterator over the successors of a given node.

Table 3.5: `ImmutableGraph` methods in WebGraph.

Moreover, there is the `Transform` class that allows to manipulate the `ImmutableGraph`. Most methods of this class receive an `ImmutableGraph` and return a new `ImmutableGraph` that represents the transformation applied. The partial interface can be seen in Table 3.6. This class allows to compute filters, permutations, maps and transposes of the `ImmutableGraphs`.

`Transform`	
`filterArcs`	Returns a graph with some arcs possibly stripped, according to the given filter.
`map`	Remaps the graph nodes through a function specified via an array.
`transpose`	Returns an immutable graph obtained by reversing all arcs in the graph.
`union`	Returns the union of two `ImmutableGraph`.

Table 3.6: `Transform` partial interface.

3.5.2 Iterators

In the `WebGraph` library, lazy iterators for nodes and successors are provided. Nonetheless, this API does not support any edge iterators.

Transversing the nodes of the graph can be done with `NodeIterator` which can start from the first node or from a specific node, given its id. Moreover, the `NodeIterator` can also give the successors and outdegree of the current node. Regarding the successors iterators of a node, this is done with a `LazyIntIterator`, which an instance of this class represent a (skippable) iterator over the node integers.

3.5.3 Algorithms

In WebGraph offers algorithms which take advantage of gap compression [16] and differential compression [2]. These algorithms are controlled by several parameters, which provide different tradeoffs between access speed and compression ratio. It is important to emphasize that these compression algorithms allow accessing a compressed graph without actually decompressing it by using lazy techniques that delay the decompression until it is actually necessary. In this library, they present the implementation of a parallel BFS and other categories of algorithms such as strongly connected components, hyper ball and geometric centralities.

3.6 Discussion

In this chapter four different graph libraries' APIs and architectures were presented. From this review, we verify that all presented libraries offer a similar API for their different graph data structures. This does not come as a surprise as in graph theory, graphs present an interface such as adding and removing an edge, retrieving the successors, the incident nodes and getting the number of edges and nodes. Besides, processing each of the items in a collection is a common operation among the presented libraries. Thus, we consider to be a requirement to implement iterators in our work.

We propose the following interface for our library in Table 3.7

Functions	
add_edge(u,v)	Adds the edge (u,v) to the graph
del_edge(u,v)	Removes the edge (u,v) to the graph
contains(u,v)	Checks if the edge (u,v) exists in the graph
list_neighborhood(u)	Returns a list of the neighbors of node u
load(filename)	Loads the graph from filename
serialize(filename)	Serializes the graph in filename
Getters	
get_number_nodes()	Returns the number of vertices in the graph
get_number_edges()	Returns the number of edges in the graph
Iterators	
edge_begin()	Returns an iterator pointing to the first edge
node_begin()	Returns an iterator pointing to the first node of the graph.
neighborhood_begin(u)	Returns an iterator pointing to the first neighbor of u

Table 3.7: Proposed interface for our work.

The structure of the libraries SNAP, Boost, igraph follows the structure of the standard template library since its design provides flexibility and extensibility. Through the template mechanism of C++, containers are suited for objects of the most varied classes. Nonetheless, the WebGraph library also contains a similar mechanism by implementing abstract classes such as ImmutableGraph and Transform. This indicates that our library should also offer some level of abstraction.

Moreover, all libraries present iterators to transverse over their graph data structures. The most common iterators among the presented libraries were the node, edge and successors.

All libraries offered different kinds of algorithms. The most common algorithms were the search algorithms BFS and DFS. Depending on the data structure or the kind of users of the libraries, different families of algorithms were implemented among the different libraries. For instance, in `WebGraph` we had more Web Graphs and compression related algorithms and in SNAP and `Boost` a more general purpose algorithms. Needless to say, the latter two have the advantage of being more mature libraries providing a wider choice of algorithms.

We would like to avoid our library providing the user a bad interface. If we look at some examples of these libraries, we can agree that it is complex and difficult to understand how to use some of them, specially in `C/C++`. As an example from `igraph`, let's take a look at the signature of the BFS.

```
int igraph_bfs(const igraph_t *graph,
               igraph_integer_t root, const igraph_vector_t *roots,
               igraph_neimode_t mode, igraph_bool_t unreachable,
               const igraph_vector_t *restricted,
               igraph_vector_t *order, igraph_vector_t *rank,
               igraph_vector_t *father,
               igraph_vector_t *pred, igraph_vector_t *succ,
               igraph_vector_t *dist, igraph_bfshandler_t *callback,
               void *extra);
```

Even though this allows for more customized BFS searches, the signature is too complex. In this case, perhaps it should be better to implement more specialized methods for different kinds of BFS searches, just as it was implemented in SNAP.

We would like to end this chapter by mentioning the importance of documentation. The documentation should allow the user to easily navigate through the library methods and concepts.

This chapter concludes the research necessary for our work. We will now proceed to to describe our solution.

4

Solution

Contents

Let us explain our solution and implementation details. The main goal of this work was to build a library based on the SDK library [7] while offering an API similar to previously studied graph libraries [2,27–29]. In this chapter we will explain the structure of our code and how we implemented the extended functionalities such as the iterators, algorithms and extended operations.

4.1 Implementation Process

We decided to implement our work in C++ since it supports object-oriented programming, a rich standard library, automatic memory management mechanisms and all presented libraries had support for this language, indicating it is widely used. However the standard C++ library does not offer a bit vector [17] and k^2-tree [6] implementations, so we used these data structures from the SDSL [8] [1], where the k^2-tree is implemented using the static bit vectors, just like in the SDK [7]. We decided to use this implementation because all its components were highly tested and documented facilitating the refactoring and development.

The Google Test framework [2] as used in SDSL so we decided to also include it in our work since it offers a modern framework for testing. The Boost library [29] was also included for the serialization and load methods.

4.1.1 The API

The API was built having in mind an easy and familiar interface as well as considering the previous presented libraries' interfaces [2, 27–29] in Chapter 3. As previously seen, most APIs present similar methods such as `add_edge`, `remove_edge`, `neighbors`, `number_nodes`, `number_edges`, `edge_iterator` and `neighbour_iterator`. In our work we intended to have an intuitive interface in order to make the library easy to use. Our API supports the operations presented in Table 4.1.

4.1.2 Code Structure

Our code can be found in `https://github.com/joo95h/dynamic_k2tree`. The code structure was intended to be as analogous as possible to the theoretical dynamic k^2-tree concepts while maintaining performance. The main classes of the project are:

- `DKTree` – This is the class that represents a graph and follows the dynamic k^2-tree data structure [1]. This class has a `Container0` which is an adjacency list coupled with a hash table. It has an array of k^2-tree pointers that represents the collection of containers $C = \{E_1, ...E_r\}$, where we used $r = 8$. This class implements the methods presented in Table 4.1.

[1] https://github.com/simongog/sdsl-lite, 2014
[2] Google test: https://github.com/google/googletest, 2019

Our API	
`get_number_edges()`	Returns the total number of edges in the graph
`get_number_nodes()`	Returns the number of nodes that the graph supports, that is n.
`add_edge(x,y)`	Adds the edge $e = (x, y)$ to the graph.
`contains(x, y)`	Checks if the edge $e = (x, y)$ exists in the graph
`del_edge(x, y)`	Deletes the edge $e = (x, y)$ from the graph.
`list_neighbour(x)`	Lists the nodes that are adjacent to x.
`serialize(ostream)`	Serializes the graph in an ostream. It will create some auxiliary files to store the k^2-tree in the graph which by default are saved in the current directory.
`load(istream)`	Loads the graph from an istream. By default it will load the k^2-trees from the current directory. This directory should be the same where you decided to serialize. It cleans these files by the end of the process. You can set the clean flag to false in order to not delete the k^2-tree serialized files.

Table 4.1: The methods offered by our work

- `Container0` – This class represents the E_0, that is, the adjacency list coupled with the hash table.

Regarding the `Container0`, it is composed of an `EdgeHashTable`, a vector of edges, `elements`, and a map, `adjacency_map`. The `EdgeHashTable` is a wrapper of an `unordered_map<Edge, uint>`, where the `Edge` class corresponds to an edge $e = (u, v)$ and it maps an edge to a position in the `elements` vector. The `elements` array also represents the adjacency list for node `u`. This is achieved by saving in `elements` the node id and a pointer to the next (and previous) edge position. However this would not be enough, since we need to know the beginning of the adjacency list of a node in constant time. The beginning of the adjacency list for node `u` is marked in the `adjacency_map` where for each node `u` is mapped the position of the initial edge `(u,j)` in `elements`, where `j` $\in \mathbb{R}$ and is the first successor of `u`. In Figure 4.1 we illustrate the overall structure of `Container0`.

We can query if an edge `e` exists in `Container0` by querying `EdgeHashTable[e]` in constant time. The `Container0` also supports listing the neighbors of a node `u` by first querying the `adjacency_map[u]=p` which will give us the first position `p` in `elements`, where the first neighbor of `u` exists, that is `elements[p]=(u,j)`. From the item in position `p` from `elements` we can navigate to the next position pointing to the next edge where the following neighbor is. This data structure composition allows to retrieve the list of neighbors from a node in linear time with the number of neighbors of a node. From the example in Figure 4.2 we can easily visualize this process.

We insert a new edge (u, v) in `Container0` by creating a new edge in the last position of `elements`, t. Besides, we create a new entry in the `EdgeHashTable` with the new edge and its corresponding position in `elements`, that is `EdgeHashTable[(u,v)] = t`. Finally, we insert the new position t in the head of

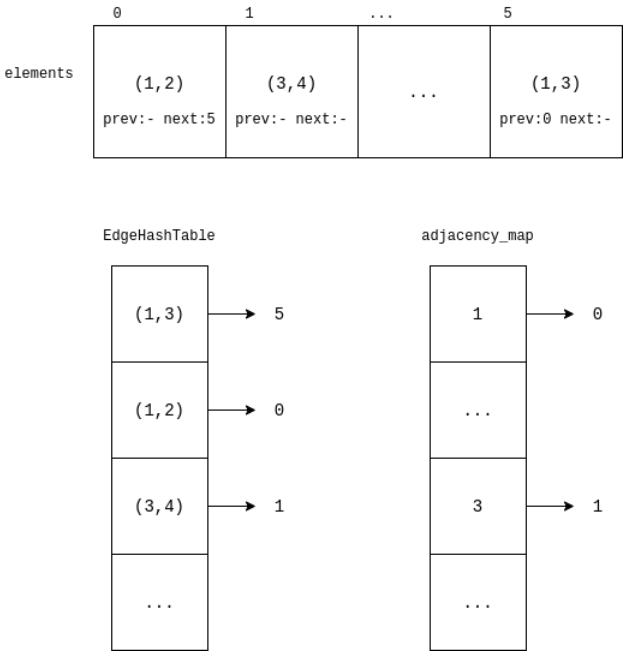

Figure 4.1: Data structure of `Container0`

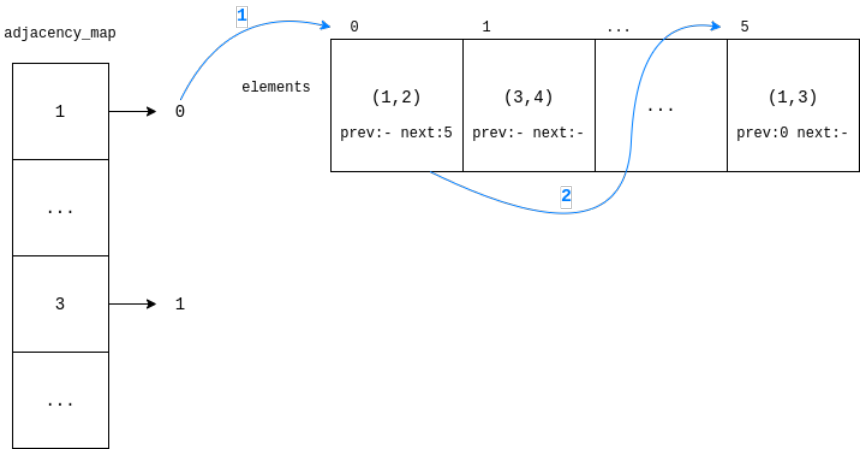

Figure 4.2: This example illustrates the operation to list the neighbors of node `u=1` in `Container0`. In order to get the first neighbor of node with id `1` first we will go to the `adjacency_map[1] = 0` where we get the position `p=0`. So now we know that the first neighbor is in position `0` in `elements`. In this position we have the `Edge (1, 2)`, so we got our first neighbor `2` (1). The `Edge` will have a pointer `next` that points to the next position which is `p=5` in `elements` where the next (and in this case last) neighbor of `1` is (2). So the final neighbor of `1` is `3` which gives us the final list of neighbors `[2,3]`.

the adjacency list of u, `adjacency_map[u] = t` and we set the next and prev pointers accordingly. The `Container0` also has `edge_free` vector that stores the valid positions in `elements`, since the deletion of edges is done by marking the deleted edges. A position is no longer valid when we delete an edge from `elements`, in this case we flag the position `p` of `elements` that have been deleted in the same position in `edge_free`. So when we want to delete the edge `e` we query `EdgeHashTable[e]=p` and then we can finally flag `edge_free[p]`.

4.1.3 Iterators

As previously shown in Chapter 3, all the graph interfaces offered a way to iterate over the graph either by its nodes, edges or neighbors of a particular node. In this work, the iterators were implemented following the `C++` iterator pattern, just like other `C++` libraries [27–29]. In this design pattern, the iterator class has a `const` pointer to the class they intend to iterate over, which in our case is a `DKTree`. The beginning of the iteration starts with the method `begin` in the class we wish to iterate over, which will return the first position of the iterator. Besides, we need to implement the method `end` that will mark the past-the-end position of the iterator. Finally, it is necessary to override the increment operator (`operator++`), in order to increment the iterator. By following this pattern, we can achieve the following iterator usage:

```
for(auto it = tree.edge_begin(); it != tree.edge_end(); ++it) {
    std::cout << *it << std::endl;
}
```

4.1.3.A k^2tree Iterators

To implement the iterator in `DKTree`, we first had to implement the k^2-tree iterators, so that we can iterate over the whole collection C. In order to do this, we also had to extend the k^2-tree from SDSL to support the node, edge and neighbor iterators.

Edge Iterator

Regarding the edge iterator was implemented using a depth-first search, keeping its current state in a stack. We used a small extra memory factor equal to the value of the height of the tree, given by $h = \lfloor \log_k n \rfloor$. It is important to stress that the edges returned by this iterator are unsorted.

One major difference between the k^2-tree implementation from SDSL and SDK is that this data structure in the SDSL is composed by two bit vectors T and L while in the SDK has one bit vector T $\|$ L.

The transversal of the tree is done by levels. At each level we will need to access a position at T until we reach the last level and we find the leaves' position in L. To do this we will need to know in which position x we are at the bit vectors T and L and in each level. In order to keep track of these positions,

we saved the initial position of each level of T and L in a vector called $pointer_{TL}$. This vector is defined by the following function:

$$pointer_{TL}(x) = \begin{cases} 0 & \text{if } x \text{ is } 0 \text{ or } h \\ k^2 & \text{if } x \text{ is } 1 \\ rank_1(pointer_{TL}(x-1)+1) \times k^2 & \text{otherwise} \end{cases}$$

where $0 < n \leq h$ and $rank_1$ is the rank function of the bit vector in T. As we proceed in the iteration, $pointer_{TL}$ will be used to keep track of the current level.

Each time we go one level deeper, we will save the current state in a stack that is composed by the current level l, the current position x in T and L and the current entry of the matrix of that level given by the row dp and column dq. We start to transverse the tree with the initial state of $l = -1, x = -1, dp = 0, dq = 0$. While transversing the tree we can meet one of the three following cases:

1. When $l < h - 1$, an internal node of the tree is being processed. If the current position of x in T is 1 we need to search in all its k^2 child nodes of the next level. In order to do so, we also need to update the current position of the level l in $pointer_{TL}$ by incrementing it by k^2. For each child node we will push its state into the stack. At the last level, if an edge is found we return true otherwise we remove the child's node state from the stack and return false, propagating it onto the upper level.

2. The second-to-last level $l = h-1$ is very similar to the previous one, however we update the current position dp and dq in a different manner since x will be at the last level which will be a position in the bit vector L. The details are shown in the pseudo-code bellow.

3. When $l = h$ we are at the last level of the tree. If $L[x] = 1$ then we return true and update the edge (dp, dq), else we return false.

When the edge iterator is incremented, we first remove the latest state from the stack until we find an edge. Finally, we end the iteration if the state stack is empty. The overall algorithm is presented in Algorithm 4.1.

Algorithm 4.1: `Edge_iterator_transverse(l, x, dp, dq, edge, stack)`

```
if x = −1 or l < h − 1 and T[x] = 1 then
    y ← pointer_TL[l + 1]
    pointer_TL[l + 1] ← pointer_TL[l + 1] + k^2
    for 0 < i < k do
        for 0 < j < k do
            stack.push(dp, dq, y, i, j, l)
            l ← l + 1
            x ← y + k × i + j
            dp ← dp + k^(h−l) × i
            dq ← dq + k^(h−l) × j
            if Edge_iterator_transverse(l, x, dp, dq, edge, stack) then
                return true
            end if
            stack.pop()
        end for
    end for
end if
if l = h − 1 and T[x] = 1 then
    y ← pointer_TL[l + 1]
    pointer_TL[l + 1] ← pointer_TL[l + 1] + k^2
    for 0 < i < k do
        for 0 < j < k do
            stack.push(dp, dq, y, i, j, l)
            l ← l + 1
            x ← y + k × i + j
            dp ← dp + i
            dq ← dq + j
            if Edge_iterator_transverse(l, x, dp, dq, edge, stack) then
                return true
            end if
            stack.pop()
        end for
    end for
end if
if l = h and L[x] = 1 then
    edge ← (dp, dq)
    return true
end if
return false
```

Since the whole tree needs to be visited, the overall time complexity to iterate all edges in the graph is $\mathcal{O}(|T| + |L|) = \mathcal{O}(m/\log(n^2/m))$.

Neighbor Iterator

The neighbor iterator for a node follows the same Algorithm 2.3 from the $Successors$ from the k^2-tree presented in Section 2.3.3.C, although we followed an iterative approach of this method.

We keep a stack of states (st) which are composed of $(n, row, column, level)$ representing the current position and level in the tree during the iteration. The pseudo code for the neighbor iterator can be seen in Algorithm 4.2 and Algorithm 4.3.

Algorithm 4.2: `Begin_neighbor(node)`

$N \leftarrow k^h/k$
$m \leftarrow k \times \lfloor node/N \rfloor$
$size \leftarrow k^h$
$stack \leftarrow \emptyset$
for $0 < j < k$ **do**
  $st \leftarrow (n/k, mod(i, n), n \times j, y + j)$
  $stack.push(st)$
end for
$Neighbor\_iterator(stack)$

Algorithm 4.3: `Neighbor_iterator(stack)`

while $!stack.empty()$ **do**
  $st \leftarrow stack.pop()$
  if $st.level \geq T.size$ **then**
    if $L[st.level - T.size]$ **then**
      $result \leftarrow st.column$
    end if
  end if
  if $T[st.level]$ **then**
    $y \leftarrow T\_rank_1(st.level + 1)k^2 + k\lfloor st.row/st.n \rfloor$
    for $0 < j < k$ **do**
      $st\_new \leftarrow (st.n/k, mod(st.row, st.n), st.column + s.n \times j, y + j)$
      $stack.push(st\_new)$
    end for
  end if
end while

When iterating over the neighborhood of a node, we keep a stack with the current iteration. Two possible cases ca happen while iterating: we can reach the last level of the tree, where we return the current value of the column if the bit in L is set, otherwise we continue to the next state in the stack. The other case, occurs when we are in middle of the tree. So if the bit is set in T, we add the next state to the stack and continue the iteration. The overall time complexity for this method is $\mathcal{O}(\sqrt{m})$ as the $Successor$ operation also follows this time complexity.

4.1.3.B `DKTree` Iterators

The `DKTree` iterators are a composition of `Container0` (E_0) iterators and k^2-tree iterators from `k_collections` (C). We start by iterating over `Container0` until we reach the last edge. Next, we iterate over each k^2-tree. If no edges were found or if we reach the last edge, we return `end()`, which is the past-end position of the iterator. There are three types of iterators:

- `DKTreeEdgeIterator` – The edge iterator. In `Container0` we simply get the vector iterator from the `elements` vector, where all the edges are saved.
- `DKTreeNeighborIterator` – The neighbor iterator. In `Container0` we will iterate over the `adjacency_map` in the same fashion as previously explained in Section 4.1.2.
- `DKTreeNodeIterator` – The node iterator. This iterator simply iterates over 0 until n.

4.1.4 Containers

Similarly to the libraries presented, we added a level of abstraction to our library. We implemented the interfaces `Graph` and `GraphIterator`. `Graph` is an interface for a graph which presents the same contract as presented in Table 4.1 plus the iterators' invoke methods namely `node_begin`, `node_end`, `edge_begin`, `edge_end`, `neighbor_begin` and `neighbor_end`.

4.2 Extended functionality

In this section we will cover the extended functionality added to SDSL and SDK. Aside from the k^2-tree iterators, we also extended the SDSL with the union operation, since it is pivotal to be able to implement the `add_edge` and `delete_edge` in the dynamic k^2-tree. In addition, we had to implement the delete method on the k^2-tree so that we could delete an edge in C. We extended the functionality of SDK regarding the algorithms implementation. As seen in Chapter 3, it is common for graph libraries to offer some algorithm implementations. Since this is the case, in this section we will also cover the implemented algorithms by our library.

Finally, we carried out a study on the add operation in order to improve its performance. At the end of this section, we will present the different add operation versions implemented.

4.2.1 Union Operation in k^2tree

For the insertion operation – in our API the `add_edge` method – after the E_0 reaches its maximum size, we create a new k^2-tree from it. In order to insert this new k^2-tree in C, we first need to calculate in which

container we should insert the incoming edges. Once we find the container E_j that can accommodate all the edges from E_0 until E_j, we perform the union operation pairwise among these the trees.

However, this operation was not implemented in SDSL so we added this functionality in the library.

Since k^2-trees store binary relations we can perform the union operation among the entries of the two k^2-trees [30]. To do so, the algorithm tranverses the bit vectors T || L in a breadth-first way. Moreover, a queue Q is maintained to store the state of the tranversal search by storing the tuple $\langle l, r_A, r_B \rangle$, where l is the current level of the k^2-tree and r_A and r_B indicate whether the processed internal nodes from A and B respectively have children or not. Additionally, we use two pointers p_A and p_B that will keep track of the current position of the respective bit vectors. Note that at each iteration p_A and p_B will be typically incremented asymmetrically as the algorithm proceeds. The union operation stops when there are no more tuples to process in the Q and the final result of this operation will be stored in a bitmap BT that will represent the final k^2-tree.

The algorithm begins by inserting the tuple $\langle 0, 1, 1 \rangle$ in Q, meaning that we start at the root where both trees A and B have children and $p_A = p_B = 0$. First, we remove the first state from Q and we will analyse all the k^2 children of the current level l. Depending on the values of r_A and r_B one out of four cases might happen:

1. $(r_A = 1,\ r_B = 1)$ – in both A and B internal node have children and the union operation is performed between the $A[p_A]$ and $B[p_B]$ and stored in the last position of BT. Furthermore, if $A[p_A] \cup B[p_B] = 1$ and $l < h$ then we insert the tuple $\langle l + 1, A[p_A], B[p_B] \rangle$ to Q. Also, p_A and p_B are incremented.

2. $(r_A = 0,\ r_B = 0)$ – none of the internal nodes have children and a 0 is added at the end of BT and neither p_A and p_B are incremented.

3. $(r_A = 1,\ r_B = 0) \vee (r_A = 0,\ r_B = 1)$ – only one of the k^2-trees has an internal node. Suppose that the node from A does not have children $(r_A = 0,\ r_B = 1)$, in this case, the algorithm copies the k^2 bits from B pointed by p_B adding the result to BT. Note that is possible since only p_B is incremented.

4.2.2 Delete Operation in k^2tree

It might be odd to say that we have implemented a delete operation on a static data structure, but this delete function in k^2-tree is needed when deleting an edge in a k^2-tree from C. Hence, it is natural for this operation not to exist in the SDSL library. When deleting an edge in a k^2-tree, we iterate over T and L, as if we were checking for the existence of a link until we get a position in the lead bit vector L. If the bit is set to 1 we flip the bit and increment the number of marked edges of the k^2-tree.

4.2.3 Algorithms

As previously analyzed in Chapter 3, all libraries support a wide range of useful algorithms for their data structures. Accordingly, we intended for our API to support some basic search algorithms, namely BFS and DFS and also algorithms specificly related to Web Graphs: counting triangles [31], clustering coefficient [32] and pageRank [33].

In our implementation, we created a visitor class named `Algorithm` which implements functions that manipulate over `Graph`.

4.2.3.A Breadth-First Search and Depth-First Search

The BFS and DFS are an important kernel used by many graph-processing applications. In many of these emerging applications of BFS, such as analyzing social networks, the input graphs are low-diameter and scale-free. Breadth-first search is an algorithm for searching graph data structures. It starts at the tree root, or some arbitrary node of a graph, and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level. The BFS uses the opposite strategy of depth-first search, which instead explores the node branch as far as possible before being forced to backtrack and expand other nodes [13].

Typically the time and memory complexity of both of these algorithms are $\mathcal{O}(n + m)$ and $\mathcal{O}(n)$. Nonetheless, for our data structure visiting the neighbors takes $\mathcal{O}(\sqrt{m})$, so it is expected in our final results for the final time complexity of these algorithms to be $\mathcal{O}(n\sqrt{m} + m)$. In the next chapter we will analyze these implementations.

4.2.3.B Counting triangles

There is much information to be gained by analyzing the large-scale data that is derived from social networks [32]. The best-known example of a social network is the “friends” relation found on sites like Facebook. Moreover, counting triangles inside a graph reveals the communities within a social-network. It has been demonstrated that the age of a community is related to the density of triangles [31]. That is, when a group has just formed, people pull in their like-minded friends, but the number of triangles is relatively small. If A brings in friends B and C, it may well be that B and C do not know each other. As the community matures, B and C may interact because of their membership in the community. Thus, there is a good chance that at sometime the triangle A, B, C will be complete.

These are important applications for a social network which can be seen as a Web Graph. Therefore, we decided to include this algorithm in our work.

We implemented two different algorithms to calculate the number of triangles in a graph.

- **Counting Triangles with a Hash Table** – This algorithm starts by computing the degree of each

node, taking a total of time of $\mathcal{O}(m)$. We create a hash table of edges H, with the pair of nodes as its key in order to be able to query if an edge exists in constant time. Next, we create another hash table with key equal to a single node. Given node v, we can retrieve the nodes adjacent to v in time proportional to the number of those nodes. After preparing our data, we count a triangle iff for each edge (x, v) there is a node u adjacent to v, such that $x \neq v \wedge u \neq x \wedge H[(u, x)]$. This algorithm takes $\mathcal{O}(m\sqrt{m})$ time and $\mathcal{O}(n + m)$ extra space.

- **Counting Triangles with the Neighbor Iterator** – This algorithm is very naive. In this case, we iterate over each edge (u, v) then for each neighbor of v, w if there is some neighbor of w, e such that there is an edge between (e, u). If there is, the number of triangles is incremented. This algorithm takes $\mathcal{O}(\sqrt{m}\log_k(n)\log(m))$ time and $\mathcal{O}(1)$ extra space.

The purpose of these two different implementations is to evaluate the trade-offs between time and memory usage regarding the methods of iterating over the edges. In the first method, the edges are saved in a hash table being expected to consume more memory while taking less time. In the second method where the edges are iterated using the neighbor iterator directly from the graph. In Chapter 5 we discuss the results among the algorithms.

4.2.3.C Clustering Coefficient

An important aspect of social networks is that they contain communities of entities that are connected by many edges. These typically correspond to groups of friends at school or groups of researchers interested in the same topic, for example. The clustering coefficient is considered as a way to identify communities [32].

Since we already had the counting triangles methods, we can easily calculate the clustering coefficient of a graph. We used a similar naive implementation as in the triangle counting with the hash table to measure this metric of a graph. Thus, it is expected for both time and space complexities to be the same as the counting triangles with a hash table, that is $\mathcal{O}(\sqrt{m})$.

4.2.3.D PageRank

Probably the most important analysis task on Web Graphs is related to measuring the importance of Web pages, as it lies at the heart of successful search engines. For example, one of the most important algorithms to find hubs and authorities on the Web, HITS [34], starts by selecting random pages and finding the induced subgraphs, which are the pages that point to or are pointed from the selected pages. Forward and backward navigation is also inherent to the definition of the well-known pageRank algorithm [33, 35], as well as variants such as Truncated pageRank [36] (especially if one wishes to estimate pageRank for isolated pages rather than for the whole graph [37]). Due to its relevance in the Web

Graphs' context, we also decided to add this algorithm to our library. Originally developed by Google and the first algorithm used for their search engine, pageRank [33] establishes the relative importance among web pages. This method computes a ranking for every web page based on the graph of the web. Every page has some number of forward links (out-edges) and back links (in-edges). Generally, highly linked pages are more important than pages with fewer links.

The overall time complexity for our data structure is $\mathcal{O}(n\sqrt{m})$ since for each node we iterate over the neighborhood while measuring the priority of the pages.

4.2.4 Improved performance on the addition of an edge

We decided to include in our work another implementation for the add operation suggested by Munro in [1]. 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 E_0 with the other k^2-tree containers by performing consecutive unions. This is the longest operation required in the add operation. Nonetheless, this bottleneck can be mitigated. In fact, it does not always occur when adding a new edge to the graph, however in some cases can take up much more than than the average time of the operation.

We present two different approaches to tackle this issue. In the first alternative the union process is delayed by dividing the union operation into smaller portions and the second where we use a background thread that will be responsible for performing the bottleneck process.

4.2.4.A Munro Delayed Union

As discussed in [1], 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. Instead of waiting for the completion of all the necessary unions, this is mitigated by processing proportional iterations of the union operation while E_0 is yet incomplete.

When adding a new edge T_n and E_0 is full, we start by converting it into a k^2-tree, C_j. Afterwards we will need to perform the unions until the sub-collection that can accommodate the previous sub-collections. First, a copy of C_j is created, L_j, which will be used for the union with C_{j+1}. L_j will allow for other queries to remain consistent since C_j and T_n will be queried while the union process is unfinished. This can be easier to visualize with Figure 4.3.

This version requires more space and takes more time than the add version originally implemented in SDK. However, the spikes should no longer occur at the time of a new rebuild of the k^2-tree collections unlike the SDK version.

Nonetheless, this approach is complex to implement, so we also implemented a 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

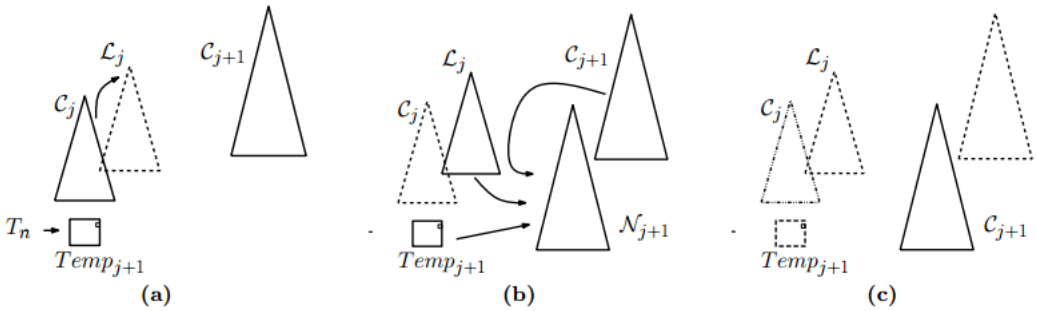

Figure 4.3: Suppose that C_{j+1} is the first sub-collection that can accommodate both C_j and a new edge T_n. If C_j must be rebuilt in the background, we "rename" C_j to L_j and initialize another (initially empty) C_j. New edge T_n is put into a separate collection $Temp_{j+1}$ (a). A background process creates a new collection N_{j+1} that contains all documents from L_j, C_{j+1} and $Temp_{j+1}$ (b). When N_{j+1} is finished, we discard C_{j+1}, L_j and $Temp_{j+1}$, and set $C_{j+1} = N_{j+1}$. This procedure guarantees that N_{j+1} is completed before the new sub-collection C_j must be re-built again [1].

collections is no longer needed, since after the first union process, although is unfinished, the data structure remains coherent with all the edges (including the new edge) present in the data structure. Overall, it is expected for this version to take as much time and memory as the original add operation version while reducing some of the time spikes.

4.2.4.B Background Thread

Additionally, a parallel version was implemented 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 of the data structure might not be over. What do we do? Well, 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 E_0, the main thread waits for the previous rebuild to finish in order to avoid the increase growth of E_0.

In the next chapter we will present a deeper analysis and discussion about these two different versions of the add operation.

5

Experimental Evaluation

Contents

In this chapter we will compare ou implementation of the add, delete, check link and list neighbor operations with SDK's. As we extended the implementation of SDSL of the union operation, we will also compare with the SDK's version. Besides, we will evaluate the performance of the implemented iterators, that is the edge and neighbor iterators.

Our implementation was compared with other data structures implementations [38, 39] in the submitted paper that we have contributed.

We presented the dynamic k^2-tree based on static bit vectors [17] with four different methods for adding an edge to the graph while reducing the time peaks, one of them described in [1] and the rest of them presented in the previous chapter. In the first method, in a single threaded process, the union operations are broken down into smaller parts of the union cycle and distributed among process the next $|E_0|$ number of add operations. The second threaded version of the delay is similar to this, however the unions operations are delayed to the next add operation. Lastly, a third and fourth method were presented where a parallel process is responsible for processing the union operations among the k^2-trees at the time of adding an edge. The difference between these two last methods is at the time of performing the unions operations when the prior unions process has not finished yet. During this time, the third method inserts the new edge in E_0 while the fourth method waits until the background thread finishes its work.

Finally, we will finish our evaluation with the graph algorithms implemented in this work, namely BFS, DFS [13], two variants of triangle counting [31], cluster coefficient [32] and the pageRank [33].

5.1 Methodology and Datasets

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, that is the graph was loaded from secondary storage during the tests. 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 Table 5.1 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`[1] corresponds to the SDK implementations and the `sdslk2tree`[2] corresponds to our implementation with the k^2-tree from SDSL[3].

Real-world graphs were obtained from the Laboratory of Web Algorithmics [4] [2, 3]. The synthetic

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

datasets were generated from the partial duplication model [40]. 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 [41]. The generated random graphs [7] have a selection probability of $p = 0.5$, which is within the range of interesting selection probabilities [40]. The number of edges for those graphs is approximately 25 times the number of vertices.

Dataset	$\|V\|$ (M)	$\|E\|$ (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 5.1: 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) [2,3] (`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 elapsed time was measured using the `clock()` function [5] and the peak of memory usage was obtained with `GNU time` [6] by the maximum resident size. 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.

For the union operation on SDSL we created and serialized a k^2-tree and then performed the operation on itself. On the dynamic k^2-tree evaluation, we considered four major operations: edge additions, removals, querying/checking and vertex neighbouring listing. For all tests we considered $k = 2$ for the static and dynamic k^2-trees.

In the addition operation we compared among the five different versions – the first add implementation plus the four versions presented in Chapter 4 – in order to analyze time and memory costs among the variations. Regarding removals, checking and vertex neighbouring listing, we have serialized each dataset as the first step. We considered a sample of 50% of the edges for the operations of removals and checking. Moreover, for the neighbouring listing we considered a sample of 50% of the vertices of each dataset.

In Table 5.1 the compression ratio in bit per edge for both implementations is also shown. We denote a big gap between both datasets; in the real Web Graphs datasets the compression ratio was much

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

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, the diagonal of the matrix of the graph to be filled. Thus, the k^2-trees we will have less filled paths with $0s$ in T and L.

Let us analyze the cost of each operation over the different datasets.

5.2 Union operation

We begin the analysis with the k^2-tree operation implemented in SDSL. In Figure 5.1 we show the average time to perform the union operation between the same k^2-tree for the synthetic (dmgen) and real Web Graphs datasets.

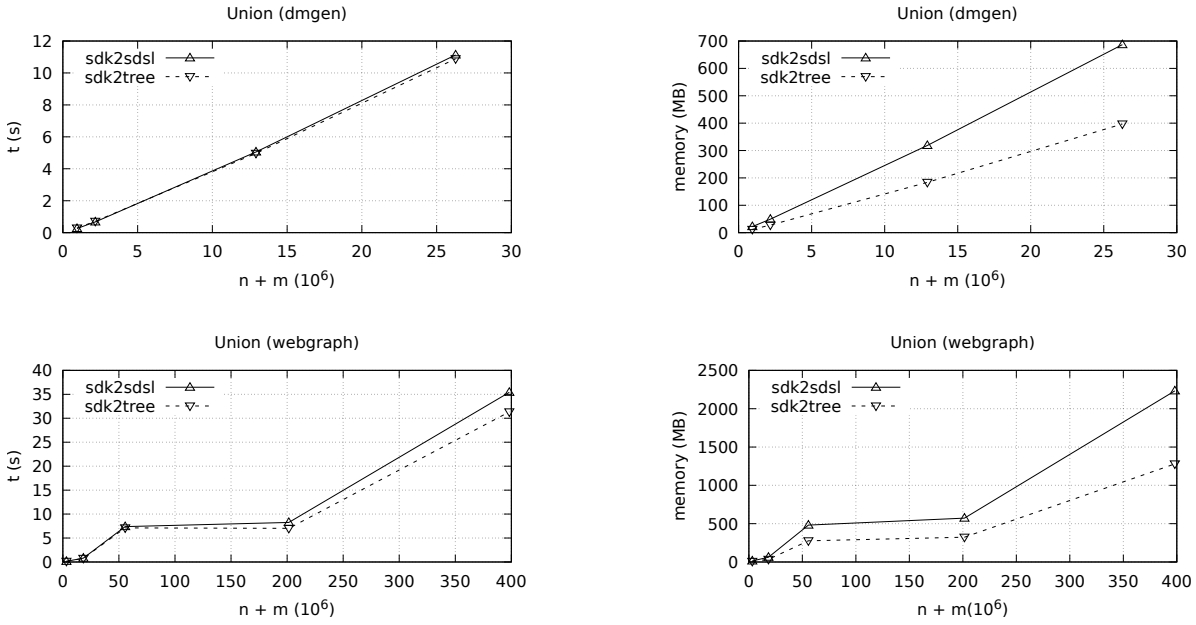

Figure 5.1: Average time taken to perform the union operation between the same k^2-tree for synthetic (dmgen) and real Web Graphs datasets.

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 that the implementation of the bit vectors in SDK is more space efficient.

In Figure 5.1, it is difficult to understand the differences between the two implementations regarding the average time. Next, we present the time table used on the previous plot. From the collected data in Table 5.2, we can see that our implementation is faster for smaller k^2-trees than the SDK.

Dataset	sdk2tree	sdslk2tree
dm50000	0.285130	0.260039
dm100000	0.703044	0.67462
dm500000	4.946079	5.05058
dm1000000	10.818093	11.1268
uk-2007-05	0.136336	0.144303
in-2004	0.755420	0.806749
uk-2014-host	7.117785	7.38885
indochina-2004	7.016849	8.24863
eu-2015-host	31.326306	35.4242

Table 5.2: Union evaluation time for Figure 5.1.

5.3 DKTree Operations

Now that we have analyzed the implemented operation in the k^2-tree, we move on to the dynamic version of the data structure. In this section we will evaluate and compare the list neighbourhoods, check individual link, add and delete edge operations. Finally, we will evaluate both neighbour and edge iterators. In the end of this chapter, the implemented algorithms for our library will be analyzed.

5.3.1 List Neighbourhoods Operation

In Figure 5.2 we are plotting against $\mathcal{O}(\sqrt{m})$, the average-case bound on the cost of listing vertex neighbourhoods for both implementations. In this evaluation, we measured the average time per listing the neighborhoods operation of 50% of the number of vertices of the dataset.

The plots shows the SDSL version was faster than the SDK however it spent more memory. The memory difference can be explained by the additional memory consumption of the k^2-trees in SDSL, as previously seen in Section 5.2. Regarding the time performance, the biggest difference between the two versions is the k^2-tree implementation, so we can conclude that the method to retrieve the neighbors in SDSL is faster than in the SDK. Thereafter, a straight line is visible for both time and memory for the synthetic datasets corroborating the correctness of the implementation. Nonetheless, this is not the case for the Web Graph datasets, since they do not present a proportional growth of the tree, unlike in synthetic datasets.

5.3.2 Check Individual Link Operation

In Figure 5.3 we are plotting against $\mathcal{O}(\log_k(n))$, the average-case bound on the cost of check individual link for both implementations. In this evaluation, we measured the average time per check operation of 50% of the number of edges of each dataset.

The plots show that both versions were very similar regarding time however once again the our implementation memory peak was higher than in the SDK's. For the indochina-2004 dataset we see

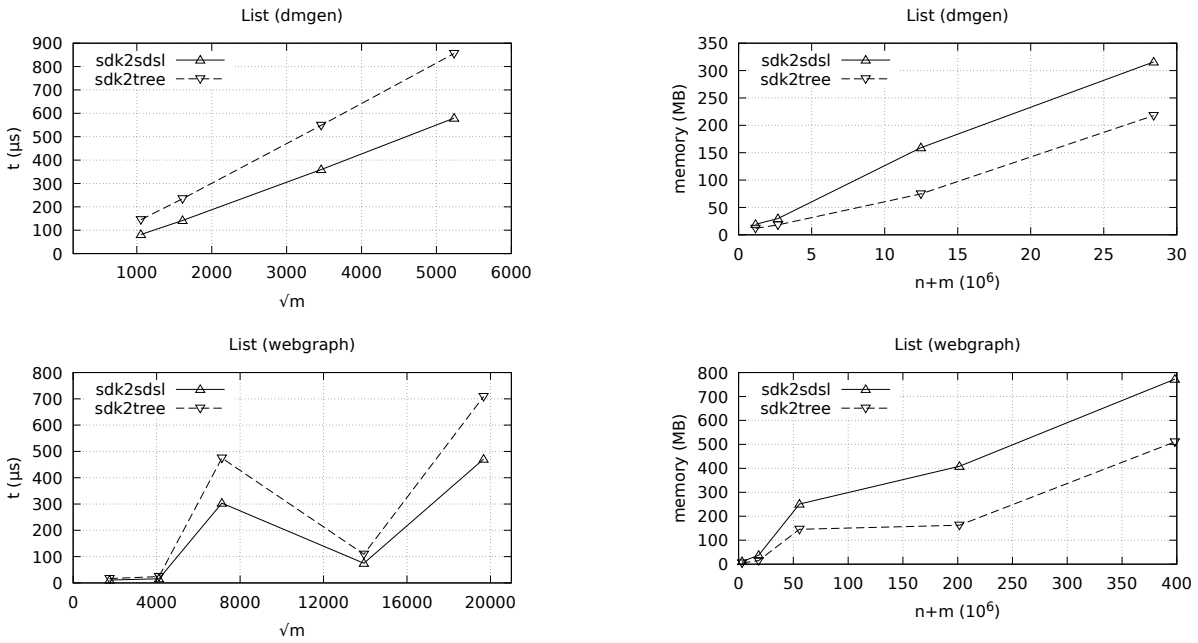

Figure 5.2: List neighbors operation average time (right) and resident memory peak (left) for synthetic (`dmgen`) and real Web Graph datasets.

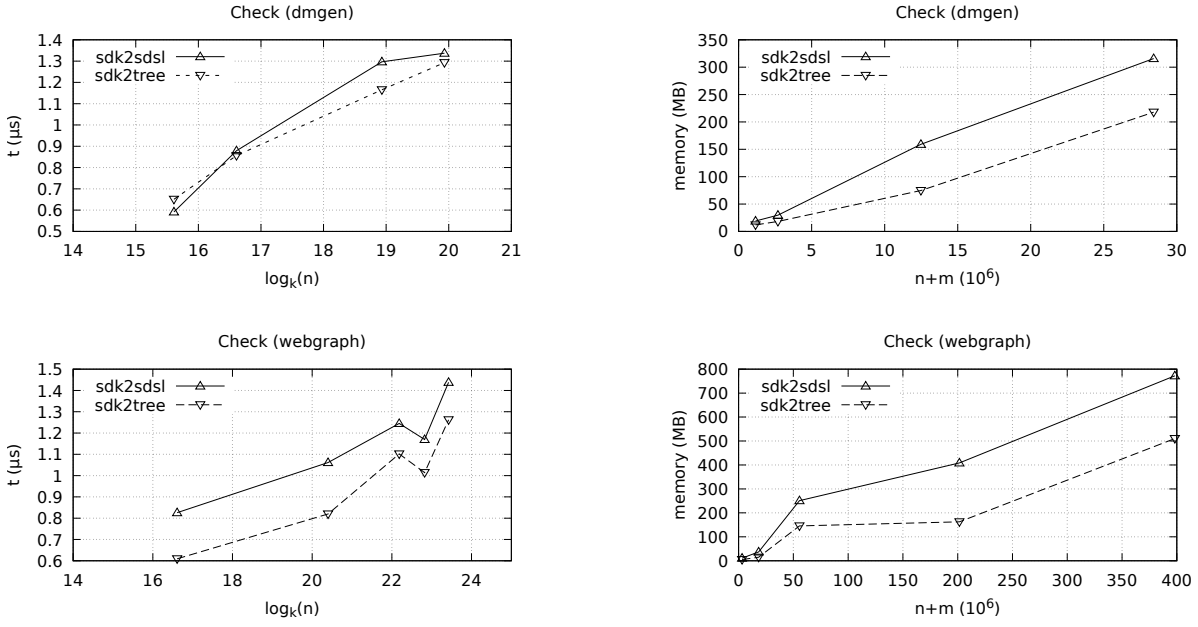

Figure 5.3: Check operation average time (right) and resident memory peak (left) for synthetic (`dmgen`) and real Web Graph datasets.

a reduction of time to perform the check operation. This can be explained due to the structure of the k^2-trees. In this case, the method takes less proportional time to reach the leaves level of the k^2-trees. This indicates that T has less paths with 0 internal nodes, reducing the number of backtracks during the recursion. This is also evident in the memory plot, where we see a much straighter growth from `uk-2014-host` to `indochina-2004` datasets.

5.3.3 Add operation

We plotted the experimental results against the theoretical time and memory complexity. In this case, the graphs were constructed directly from the datasets unlike the previous operations in which the graph was loaded from memory. Recall that we defined $\varepsilon = 0.25 \Rightarrow r = 8$, implying in our collection $C = \{E_1, ..., E_8\}$.

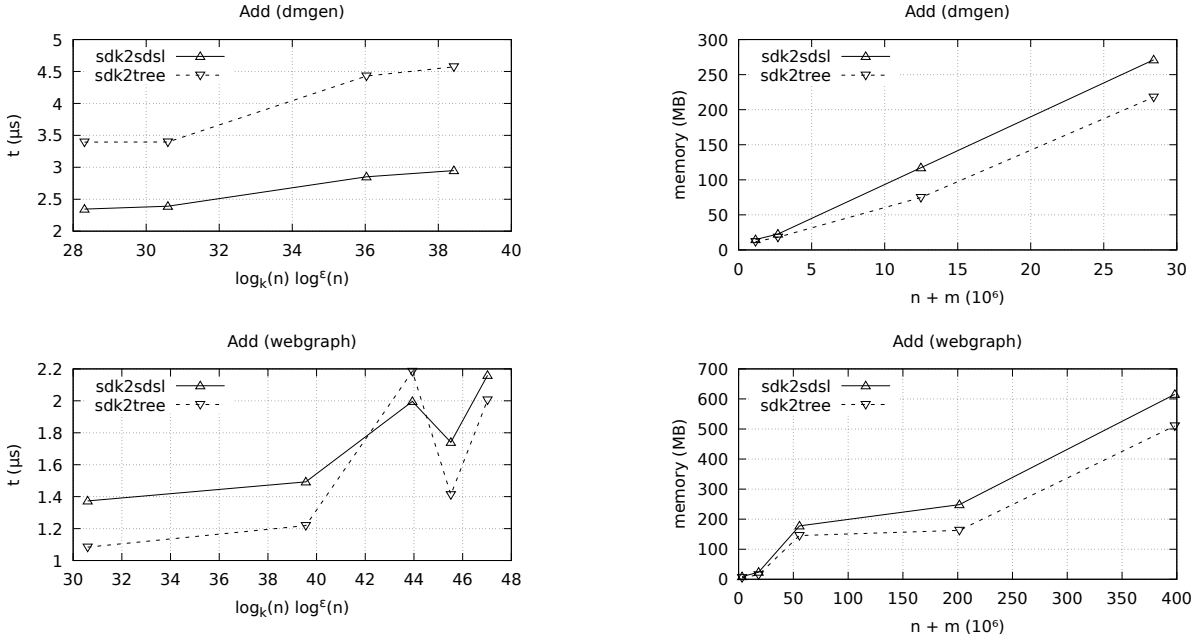

Figure 5.4: Add operation average time (right) and resident memory peak (left) for synthetic (`dmgen`) and Web Graph datasets.

In Figure 5.4 we show our experimental results regarding the add operation. 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 sub-collection, giving a small advantage to our implementation during the union time as previously seen in Table 5.2. However, for the Web Graphs datasets, the stored k^2-trees in C

have more edges turning the advantage to the SDK implementation.

Another interesting feature in the average time plots, is that the average time to add an edge in the Web Graph datasets is less than in the synthetic datasets. In Table 5.1, we have seen that the compression ratios between the two types of datasets relayed on the size of the bit vectors of the k^2-tree. Hence, the smaller the bit vectors used, the less time the union operation will take.

5.3.3.A Augmented Add Operation Versions

Additionally, four version were implemented for the add operation. In the following graphics 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 (whole) union operation is delayed for the next add operation.
- `add edge munro` – during the rebuild, the union operations are distributed in $|E_0|$ iterations, according to Munro et. al. [1].
- `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 finishes the previous union pahse if necessary.

In later computations, the `add edge` can take much longer to perform since it is necessary to rebuild some of the k^2-tree sub-collections.

In the following plot, we present the average time per adding a singular edge for the `uk-2007-05` dataset. In Figure 5.5, we pretend to compare the differences in the highest spikes when adding an edge to the graph among the different versions.

Analyzing the plot in Figure 5.5, for the `add edge` and `add edge parallel wait` some visible big and medium spikes are visible. For the `add edge delay`, we have half of the size of the spikes comparing with the two previous versions. For the `add edge parallel` we can see fewer and smaller spikes and for the `add edge munro` there are the fewest spikes.

However, in Figure 5.5 is hard to understand what is happening for smaller time spans. In Figure 5.6 we can take a closer look to what is happening at the bottom of the previous plot.

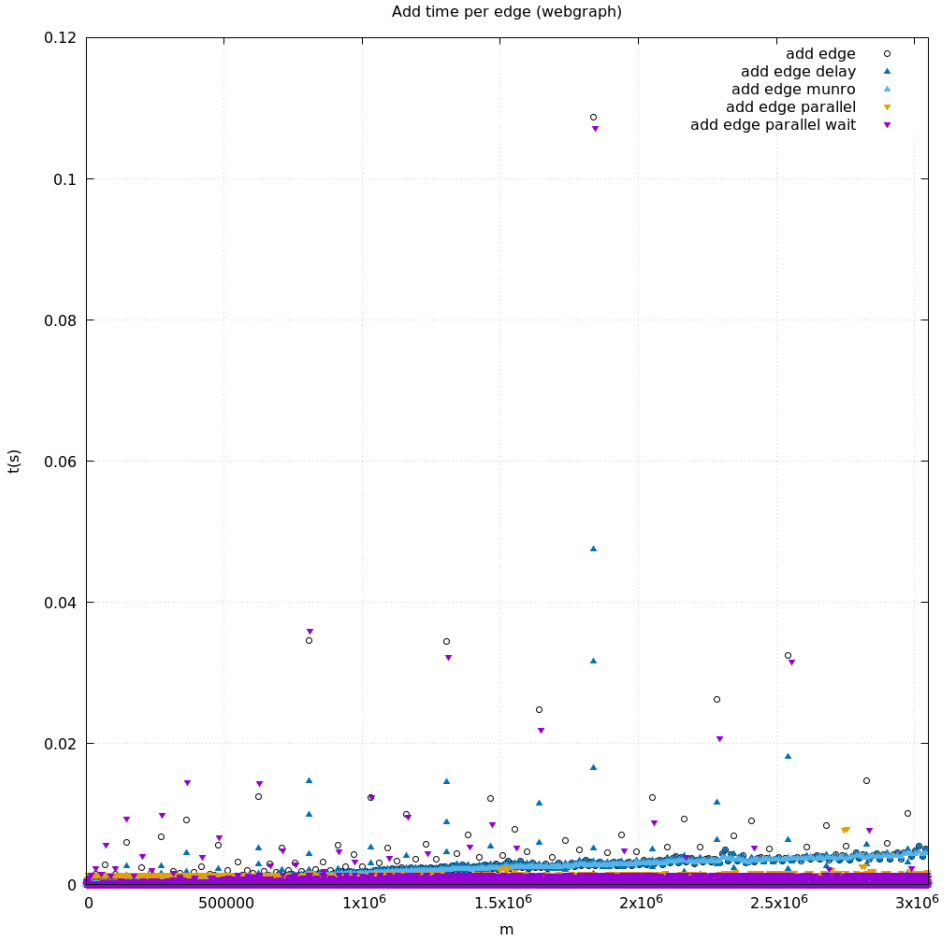

Figure 5.5: The time distribution when adding a new edge to `uk-2007-05` dataset with 1 million nodes.

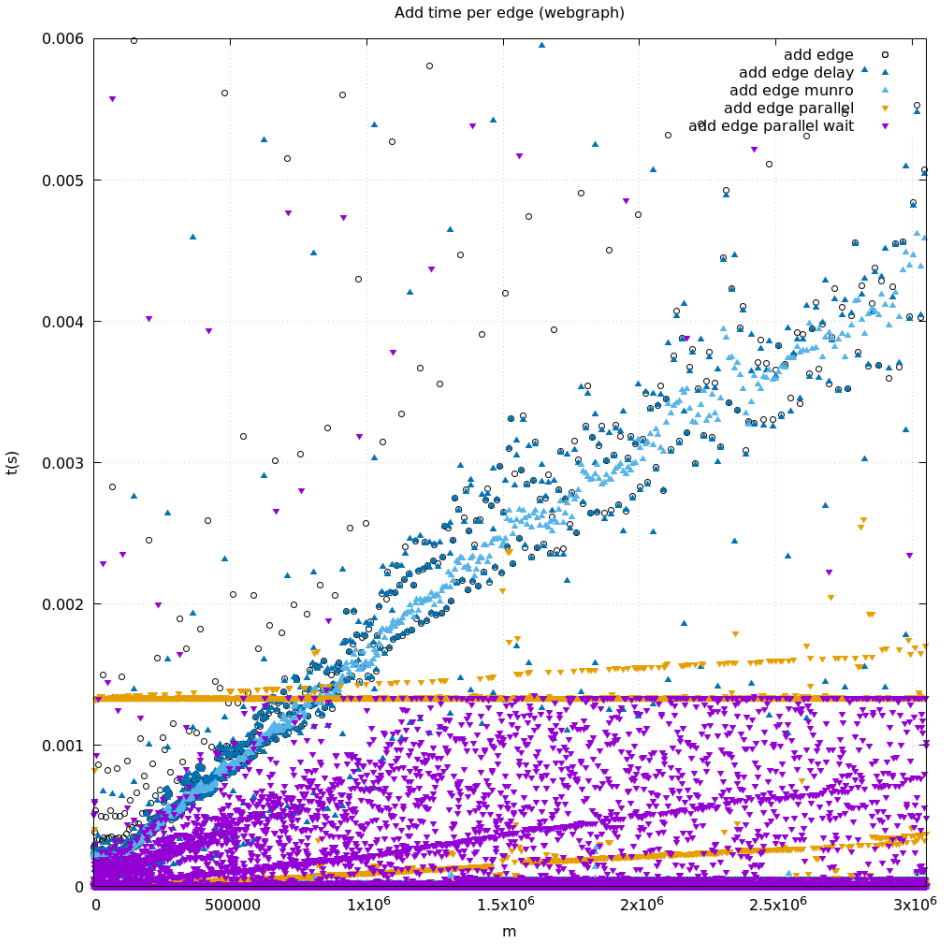

Figure 5.6: Zoomed scale at the time distribution when adding a new edge to `uk-2007-05` dataset with 1 million nodes.

In Figure 5.6 we can see more clearly what is happening for most edge additions. Both `add edge delay` and `add edge munro` follow a linear behaviour as expected. In these single threaded versions, as we add edges to the graph, the k^2-trees in collection C keep growing in size, taking longer to perform the union operations. However, `add edge delay` shows much more sparse points than the `add edge munro`, since the time of the unions of `add edge delay` is not as well distributed as in `add edge munro`.

Regarding the parallel implementations, these show two very different behaviours. On the one hand, the `add edge parallel` behaviour is mostly a horizontal line. This is expected, since the main thread only inserts edges in E_0 in constant time. However, due to not waiting for the background thread to finish the rebuild of the structure, the E_0 grows much more than in other versions. Thus, the time to build the k^2-tree from E_0 is greater. Consequently, bigger sub-collections are added to C, making the overall union time greater. Regarding the `add edge wait`, this version grows the slowest from all implementations, since it delegates the heavy work to the background thread and only waits when necessary, allowing this implementation to have a upper bound. The visible long spikes still arise in this version due to the longer rebuilds. However, for most iterations the addition of an edge takes less time.

All in all, `add edge munro` was the version with fewer spikes compared with all other versions and the `add edge wait` was the fastest version for most additions although it still occasionally presents big time spikes just as `add edge`. We have analyzed the time per adding an edge to the graph, however we still have to discuss how the overall average time and memory consumption varies among the different add operation versions.

We plotted the experimental results against the theoretical time and memory complexity. The following plots used the same data as in Figure 5.4, Figure 5.5 and Figure 5.6. In Figure 5.7 we have the average time of adding an edge in a graph among the four addition versions for both the synthetic and Web graphs datasets.

In Figure 5.7 shows that all implementations followed a line corroborating our theoretical expectations.

The implementations' behaviour deviates in each dataset. Nonetheless, the maximum resident peak memory performance is similar in both datasets.

First we will analyze the `add edge delay`. As expected, this version 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` it is mitigated to the upcoming additions instead of waiting for the overall rebuild to be concluded at once.

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, the `add edge wait` and 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

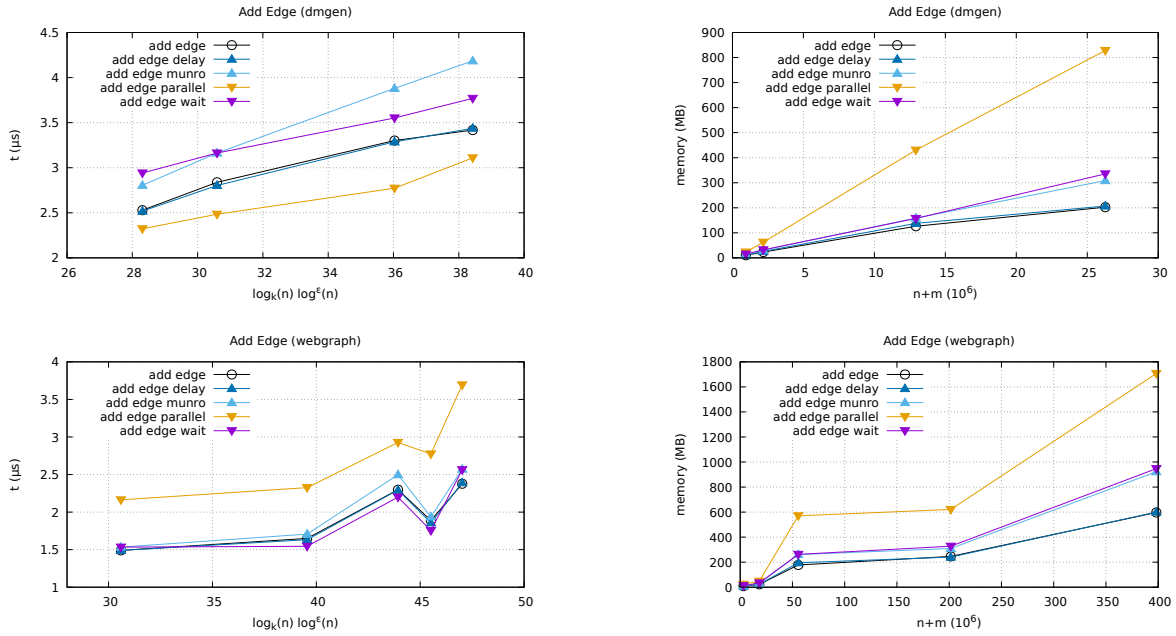

Figure 5.7: Average time (right) and resident peak memory (left) for the four different versions of the add operation for the synthetic (`dmgen`) and Web Graph datasets.

up 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` version, it 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 bigger graphs, in this case the Web Graphs. Moreover, this version had the highest memory growth, consuming at one point 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 Figure 5.4. Notice how the average time to add an edge in the synthetic datasets is higher than in the Web Graph datasets. Another evidence that supports this is the compression ratios in Table 5.1. Hence, the performance of this version will be better in the Web Graphs datasets as the background thread won't delay the main thread as much as in the synthetic datasets.

5.3.4 Deletion operation

For the delete operation evaluation we loaded the graph from memory just like in Section 5.3.1 and Section 5.3.2. For each dataset we deleted 50% of the edges in the graphs.

In Figure 5.8 we have the average time to delete an edge per dataset and the resident peak memory consumption.

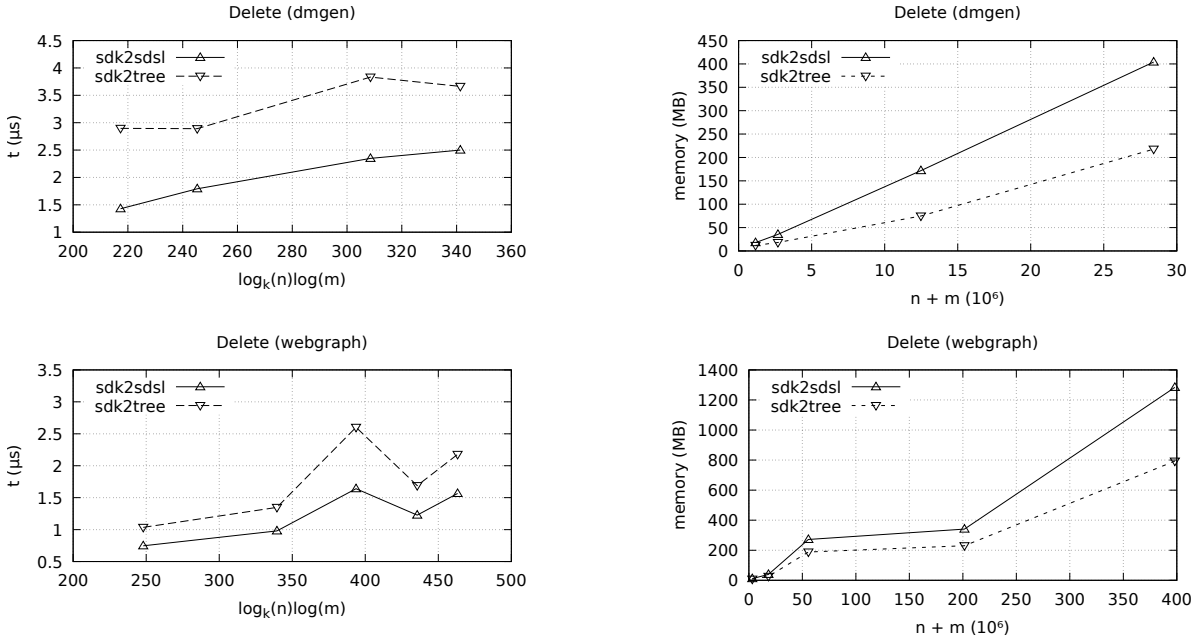

Figure 5.8: Delete operation average time (right) and resident memory peak (left) for synthetic (`dmgen`) and Web Graph datasets.

Regarding the results, they show that the experimental time and memory performance is bounded by the theoretical expectations. Moreover, our implementation took less time than the SDK implementation while spending more memory for both synthetic and real Web Graph datasets, since our implementation benefits from smaller k^2-trees. This is expected, since the delete and add operation share a similar overall algorithm in the sense of being necessary to rebuild the data structure at some point. As we have previously seen in Figure 5.4, our version took less time than the SDK's, so it would be expected for this operation to also take less time for our implementation since the k^2-trees become smaller.

5.4 Iterators

In this section we will analyse the time and memory results of the implemented iterators. The plots only show the results for our implementation there are no means for comparison.

5.4.1 Edge Iterator

For the evaluation of the edge iterator, we loaded the graphs from memory and iterated over the whole datasets.

In Figure 5.9, 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

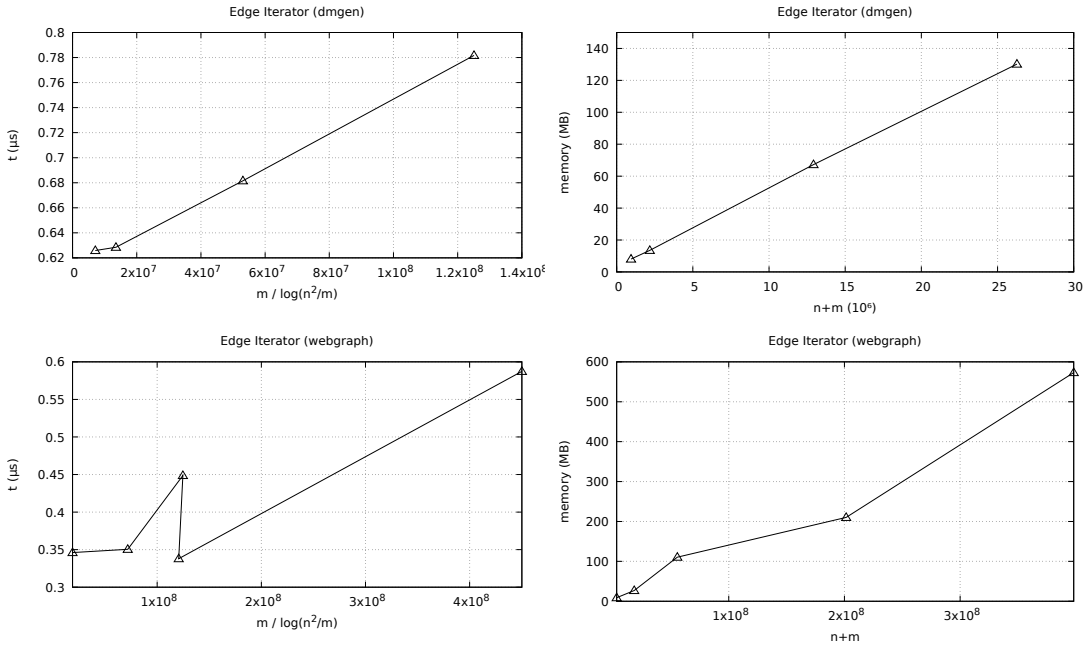

Figure 5.9: Edge Iterator average time (right) and resident memory peak (left) for synthetic (`dmgen`) and Web Graph datasets.

the theoretical bounds for time and memory. During the edge iteration, the whole k^2-tree is visited. Thus, for a smaller k^2-tree, the edge iterator will take less time. For the Web Graphs datasets, specifically the `indochina-2004` dataset showed an unexpected small time to iterate over all its edges. This could be explained by the smaller size of the tree, that is by the size of bit vectors T and L as we can corroborate by checking Table 5.1.

5.4.2 Neighbor Iterator

The neighbor iterator was evaluated with the same dataset in Section 5.3.1 and we also plotted against the our list neighborhood method for a richer comparison between the two implementations. The experimental results in Figure 5.10 show a straight line for the synthetic datasets. However, there is a cost for maintaining a stack with the current state of the navigation in the k^2-tree. Nonetheless, the memory usage was identical for both iterator and method implementations.

5.5 Algorithms

We implemented some well known graph algorithms, for which we compare consumed memory and execution time against expected theoretical results. For each algorithm, we present figures for the running time and for peak resident memory usage. Each figure shows results for the Web graph datasets

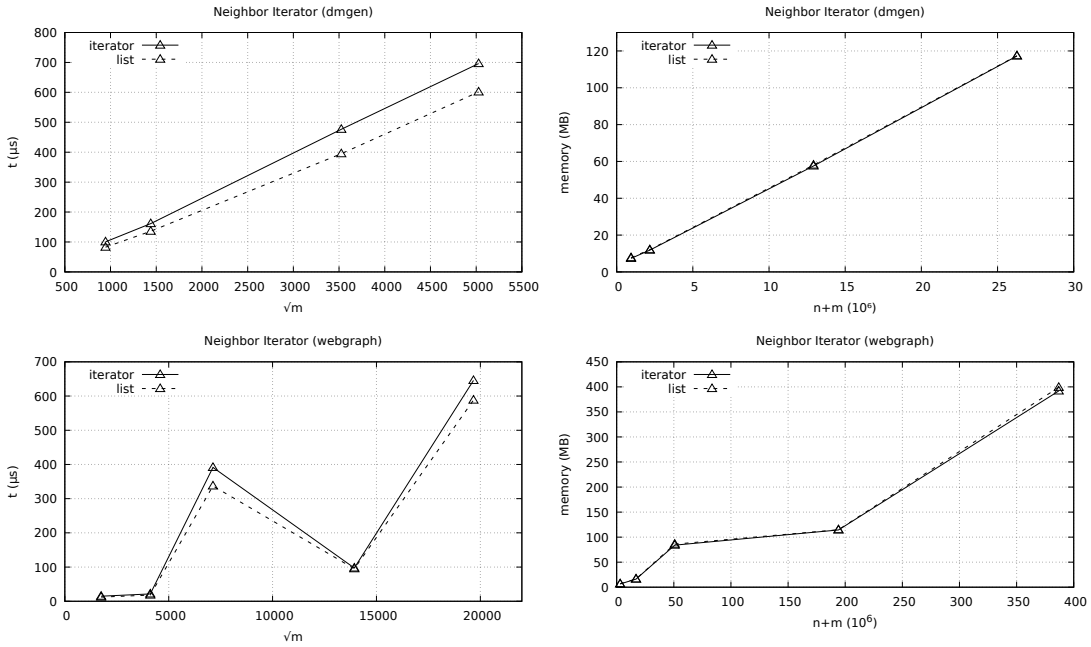

Figure 5.10: Neighborhood Iterator and List Neighborhood method average time (right) and resident memory peak (left) for synthetic (`dmgen`) and Web Graph datasets.

on the left and synthetic duplication model datasets on the right. We omit dataset `indochina-2004` from the graph algorithm tests because its topology does not allow for an adequate assessment of algorithms expected efficiency.

5.5.1 Breadth First Search

In Figure 5.11 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 $\mathcal{O}(n+m)$.

5.5.2 Depth First Search

The results for DFS are shown in Figure 5.12. It has a behavior similar to Section 5.5.1 for both the Web graphs and duplication model graphs, as expected.

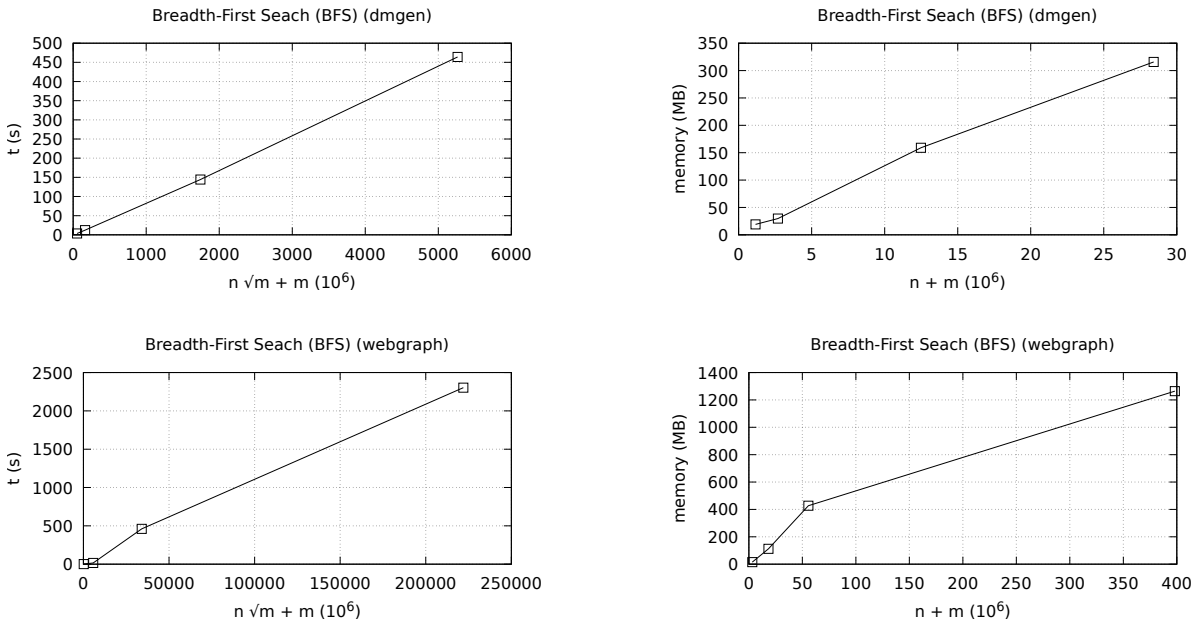

Figure 5.11: BFS operation average time (right) and resident memory peak (left) for synthetic (`dmgen`) and Web Graph datasets.

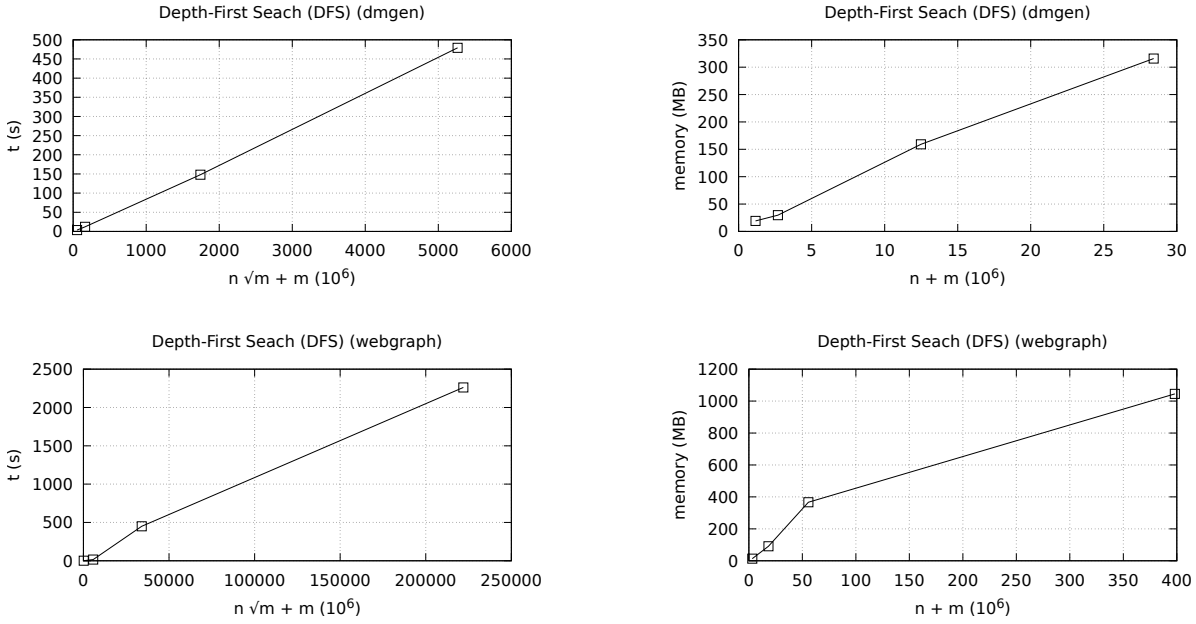

Figure 5.12: DFS operation average time (right) and resident memory peak (left) for synthetic (`dmgen`) and Web Graph datasets.

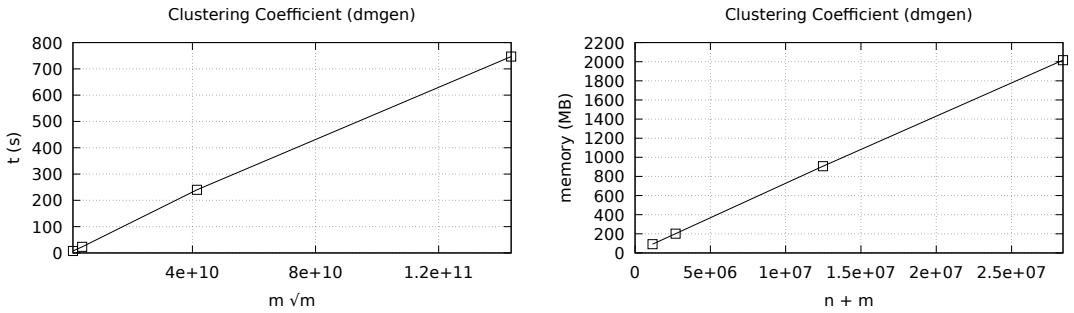

Figure 5.13: Clustering coefficient time (right) and memory peak usage (left) for the synthetic dataset.

5.5.3 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 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 Figure 5.13. 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 $\mathcal{O}(m\sqrt{m})$.

5.5.4 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 Figure 5.14 and are within 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.

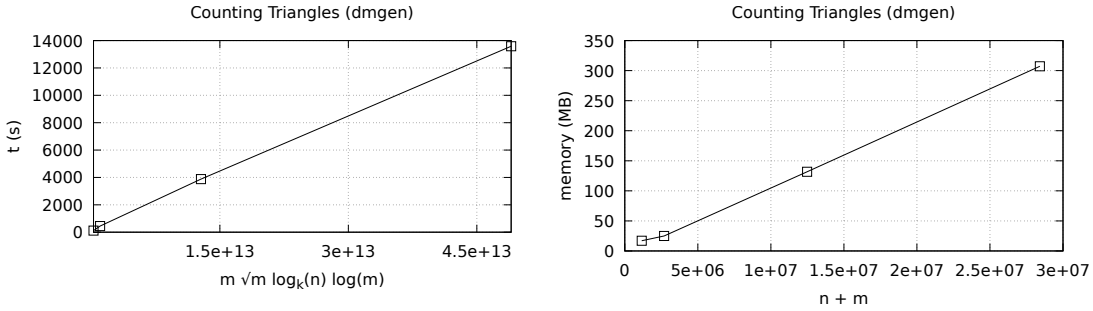

Figure 5.14: Counting triangles with neighborhood iterator time (right) and memory peak usage (left) for synthetic datasets.

5.5.5 PageRank

The pageRank algorithm was plotted only with the duplication model graphs. We did not test with the Web Graph datasets since the number of iterations to converge the algorithm is different due to the different graph properties. In Figure 5.15 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.

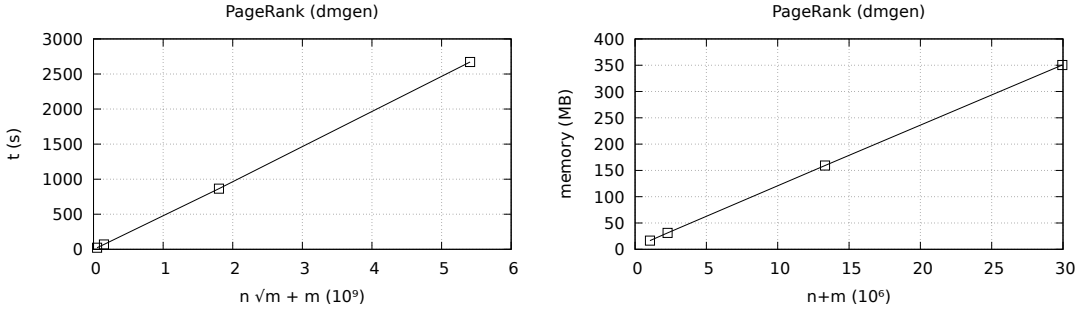

Figure 5.15: PageRank time (right) and resident peak memory (left) for synthetic datasets.

6

Conclusion

Contents

6.1 Conclusions

In this master thesis we started by introducing graph concepts followed by most common graph representations. Afterwards, we discussed some of the classical compressed representations namely the CSR and CSC [14]. Next, we introduced some state of the art compact graph representations, namely the LINK Database [16, 36], the `WebGraph` framework [2], the k^2-tree [6] and dynamic k^2-tree [1]. By the end of Chapter 2 we compared the compression ratio of all the data structures.

Afterwards, in Chapter 3, we investigated the current state of the art of graph libraries [2, 27–29] and proposed our graph API in Table 3.7.

This work had two main goals: refactor the SDK library [7] and extend its functionality. We believe both of this challenges were overcome with success. Here we provide a tested and refactored `C++` version of the SDK library and also extended its functionality by implementing edge and neighborhood iterators. To achieve this, we used the static bit vector and k^2-tree from SDSL [8]. Nonetheless, the union operation, edge and neighborhood iterators were lacking in this library, so implemented 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 neighbors list and check the presence of a link, where our implementation showed better time performance and a higher memory usage. We also conducted a 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 smaller height 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. [1] 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. The results reveal that the first delay version heavily decreases the amount of time spikes that occur at the time of adding an new edge. Regarding the second version proposed by Munro [1], the time spikes are mostly gone. In fact, the Munro's version takes slightly more time and memory than the previous version since it is needed to create a copy of the k^2-tree subcollections before the rebuild process, in order to keep the coherence of the data structure. It is important to understand that this version was much more cumbersome to implement and one might question if the end result is worth by comparing the final results, since the macro delay version decreased the time

spikes while not hindering time and space performance.

Moreover, two parallel versions were implemented.

In the first version, at the time of rebuilding the k^2-tree sub-collections when the E_0 is full and the previous rebuild is unfinished, inserts a new edge in E_0, increasing its size in time. This parallel version demonstrated to be very fast for the synthetic 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 very 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. In contrast, in the real Web Graph datasets the bit vectors of the k^2-trees were smaller and 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 search, namely the BFS and the DFS and also some Web Graph related algorithms: the Clustering Coefficient, two different counting triangles and finally the pageRank algorithm. The results show that our algorithm implementations were correct, since all plots' curves were linear. Moreover, we conducted a small study over the Counting Triangles algorithms. We compared two different implementations; one which 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`.

Finally, as we promised at the end of Chapter 2, we would like to compare our compression ratios in Table 5.1 with the presented state of the art compact data structures compression ratios in Table 2.2. Taking into account the Web Graphs datasets, we conclude that ours and the SDK's data structure are highly competitive with the previously presented data structures while being the only implementations to offer both dynamic behaviour and direct querying over the compressed graph.

6.2 Limitations and Future Work

As all things in life, there is room for improvement in our 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. However, we would need to implement this operation for E_0. To query the reverse neighbor with in `Container0` we would need to implement a similar data structure to `adj_map`. Thus, the reverse neighbor method implementation for our data structure would be very similar to the already implemented list neighbor method.

Another interesting operation that could be implemented is the common neighbors between two vertices since it is widely used in social graphs [42]. There are common intuitions about how social graphs are generated, for example, it is common to talk informally about nearby nodes sharing a link. There are also common heuristics for predicting whether two currently unlinked nodes in a graph should be linked. This could be applied to suggest friends in an online social network or to recommend movies or videos to users in a recommendation network.

In our work, we did not implement a very sophisticated node iterator. In our case, we simply iterate over $0, ..., |V|$. We could improve this iterator by iterating over the vertices that exist in fact in the graph. Moreover, most of the presented libraries [2,27–29] implemented a Node abstraction with several attributes and proprieties such as the outdegree (number of neighbors), indegree (number of incident nodes), id and the data held by the node.

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` [2]. Thus, we could add dynamism to `WebGraph` where C is composed several containers of `WebGraphs` instead of k^2-trees.

Bibliography

[1] I. Munro, Y. Nekrich, and J. S. Vitter, "Dynamic data structures for document collections and graphs," in *Proceedings of the 34th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, 2015, pp. 277–289.

[2] P. Boldi and S. Vigna, "The webgraph framework i: compression techniques," in *Proceedings of the 13th international conference on World Wide Web*. ACM, 2004, pp. 595–602.

[3] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," in *Proceedings of the 20th international conference on World wide web*, 2011, pp. 587–596.

[4] F. Claude and S. Ladra, "Practical representations for web and social graphs," in *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, ser. CIKM '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 1185–1190. [Online]. Available: https://doi.org/10.1145/2063576.2063747

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

[6] N. R. Brisaboa, S. Ladra, and G. Navarro, "k2-trees for compact web graph representation," in *String Processing and Information Retrieval*, J. Karlgren, J. Tarhio, and H. Hyyrö, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 18–30.

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

[8] S. Gog, T. Beller, A. Moffat, and M. Petri, "From theory to practice: Plug and play with succinct data structures," in *13th International Symposium on Experimental Algorithms, (SEA 2014)*, 2014, pp. 326–337.

[9] R. Diestel, *Graphentheory*. Springer, 2000.

[10] J. A. Bondy, U. S. R. Murty *et al.*, *Graph theory with applications*. Macmillan London, 1976, vol. 290.

[11] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*, 2nd ed. McGraw-Hill Higher Education, 2001.

[12] R. Sedgewick, "Algorithms in c, part 5," *Reading-MA, USA: Addison-Wesley, Chapters*, vol. 17, p. 18, 2002.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.

[14] E. F. D'Azevedo, M. R. Fahey, and R. T. Mills, "Vectorized sparse matrix multiply for compressed row storage format," in *Computational Science – ICCS 2005*, V. S. Sunderam, G. D. van Albada, P. M. A. Sloot, and J. J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 99–106.

[15] K. Bharat, A. Broder, M. Henzinger, P. Kumar, and S. Venkatasubramanian, "The connectivity server: Fast access to linkage information on the web," *Computer networks and ISDN Systems*, vol. 30, no. 1-7, pp. 469–477, 1998.

[16] K. H. Randall, R. Stata, R. G. Wickremesinghe, and J. L. Wiener, "The link database: Fast access to graphs of the web," in *Proceedings DCC 2002. Data Compression Conference*. IEEE, 2002, pp. 122–131.

[17] N. Rahman, R. Raman *et al.*, "Engineering the louds succinct tree representation," in *International Workshop on Experimental and Efficient Algorithms*. Springer, 2006, pp. 134–145.

[18] H. Samet, *Foundations of multidimensional and metric data structures*. Morgan Kaufmann, 2006.

[19] N. R. Brisaboa, S. Ladra, and G. Navarro, "Compact representation of web graphs with extended functionality," *Information Systems*, vol. 39, pp. 152–174, 2014.

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

[21] G. Jacobson, "Space-efficient static trees and graphs," in *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, ser. SFCS '89. Washington, DC, USA: IEEE Computer Society, 1989, pp. 549–554. [Online]. Available: https://doi.org/10.1109/SFCS.1989.63533

[22] "Compressed representation of dynamic binary relations with applications," 2017.

[23] P. M. Sant, “”rooted tree”,” in *Dictionary of Algorithms and Data Structures* [online], Paul E. Black, ed., 2004, available from: https://www.nist.gov/dads/HTML/rootedtree.html (accessed 13 March 2019).

[24] E. Gamma, R. Helm, R. Johnson, J. Vlissides, and D. Patterns, “Elements of reusable object-oriented software,” *Reading: Addison-Wesley*, 1995.

[25] J. Palsberg and C. B. Jay, “The essence of the visitor pattern,” in *Proceedings. The Twenty-Second Annual International Computer Software and Applications Conference (Compsac'98)(Cat. No. 98CB 36241)*. IEEE, 1998, pp. 9–15.

[26] J. Siek, A. Lumsdaine, and L.-Q. Lee, *The boost graph library: user guide and reference manual*. Addison-Wesley, 2002.

[27] J. Leskovec and R. Sosič, “Snap: A general-purpose network analysis and graph-mining library,” *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 8, no. 1, p. 1, 2016.

[28] G. Csardi and T. Nepusz, “The igraph software package for complex network research,” *InterJournal*, vol. Complex Systems, p. 1695, 2006. [Online]. Available: http://igraph.org

[29] B. Schling, *The Boost C++ Libraries*. XML Press, 2011.

[30] N. R. Brisaboa, G. de Bernardo, G. Gutiérrez, S. Ladra, M. R. Penabad, and B. A. Troncoso, “Efficient set operations over k2-trees,” in *2015 Data Compression Conference*. IEEE, 2015, pp. 373–382.

[31] S. Fortunato, “Community detection in graphs,” *Physics reports*, vol. 486, no. 3-5, pp. 75–174, 2010.

[32] J. Leskovec, A. Rajaraman, and J. D. Ullman, *Mining Social-Network Graphs*, 2nd ed. Cambridge University Press, 2014, p. 325–383.

[33] L. Page, S. Brin, R. Motwani, and T. Winograd, “The pagerank citation ranking: Bringing order to the web.” Stanford InfoLab, Tech. Rep., 1999.

[34] J. M. Kleinberg, R. Kumar, P. Raghavan, S. Rajagopalan, and A. S. Tomkins, “The web as a graph: measurements, models, and methods,” in *International Computing and Combinatorics Conference*. Springer, 1999, pp. 1–17.

[35] S. Brin and L. Page, “The anatomy of a large-scale hypertextual web search engine,” *Computer Networks*, vol. 30, pp. 107–117, 1998. [Online]. Available: http://www-db.stanford.edu/~backrub/google.html

[36] L. Becchetti, C. Castillo, D. Donato, R. Baeza-Yates, and S. Leonardi, “Link analysis for web spam detection,” *ACM Transactions on the Web (TWEB)*, vol. 2, no. 1, pp. 1–42, 2008.

[37] M. R. Henzinger, A. Heydon, M. Mitzenmacher, and M. Najork, “On near-uniform url sampling,” *Computer Networks*, vol. 33, no. 1-6, pp. 295–308, 2000.

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

[39] N. R. Brisaboa, A. Cerdeira-Pena, G. de Bernardo, and G. Navarro, “Compressed representation of dynamic binary relations with applications,” *Information Systems*, vol. 69, pp. 106–123, 2017.

[40] F. Chung, L. Lu, T. G. Dewey, and D. J. Galas, “Duplication models for biological networks,” *Journal of computational biology*, vol. 10, no. 5, pp. 677–687, 2003.

[41] A. Bhan, D. J. Galas, and T. G. Dewey, “A duplication growth model of gene expression networks,” *Bioinformatics*, vol. 18, no. 11, pp. 1486–1493, 2002.

[42] P. Sarkar, D. Chakrabarti, and A. W. Moore, “Theoretical justification of popular link prediction heuristics.” in *IJCAI proceedings-international joint conference on artificial intelligence*, vol. 22, no. 3. Citeseer, 2011, p. 2722.