Sorbonne Université

Projet STL

Compression de très grands graphes de terrain

Rapport

Auteurs:
Hichem Rami AIT EL HARA
Adel EL AMRAOUI
Anas YAHYAOUI

Superviseur:
Dr. Maximilien DANISCH

Contents

Abstract

Search engines, drug design and traffic management are a few examples of the applications of graph analysis. These graphs get larger and larger, sometimes reaching many trillions of links.

The goals of our project are:

- conceptualise a graph data structure that is compact in memory, and that allows us to realise some atomic operations on graphs (for example iterating over the neighbours of a given node or iterating over all the links of a graph).
- Use the data structure to write some simple algorithms like BFS (Breadth-first Search) or PageRank.
- Testing the implementations on graphs that contrain many billions of links.
- Submitting the implemetations, commented and with documentation.

Résumé

Les moteurs de recherche, la conception de médicament et la gestion du trafic ne sont que quelques exemples d'applications qui reposent sur l'analyse de graphe. Des graphes concernés sont de plus en plus massifs, atteignant parfois plusieurs trillions de liens.

Les objectifs de notre projet sont:

- Concevoir une structure de donnée de graphe compacte en mémoire permettant de réaliser certaines opérations atomiques de graphe (par exemple itérer sur les voisins d'un nœud donné ou itérer sur tous les liens du graphe).
- Utiliser la structure de données pour coder certains algorithmes simples tels que BFS (parcours en largeur) ou PageRank.
- Tester les implémentations sur des graphes contenant plusieurs milliards de liens.
- Livrer les implémentions commentées et avec documentation.

1 Introduction

A graph consists of a set of nodes "V" (also called vertices or points) and set of links "E" (also called edges or lines), a link represents a connection between two nodes, these connections are symmetrical in undirected graphs, and asymmetrical in directed graphs. We note $n = |V|$ the number of nodes of a graph and $m = |E|$ it's number of links. A node y is called a successor of a node x if there exists a link $e = \{x, y\}$ going from x to y. we note $S(x)$ the set $y_0, y_1...y_k$ of nodes where each node y_i is a successor of x.

In our case we work on real-world graphs, which are often directed, therefore we will be working with directed graphs, and either way, we can still represent undirected graphs with a directed graph by adding for each link $\{x, y\}$, a link $\{y, x\}$ if there isn't one. An example of a real-world graph is the web graph, in which every node represents a web page, and every link from a node x to another node y represents a hyperlink from the page x to the page y. Another example is graphs from social media sites, in which users can subscribe to one another, in this case the nodes represent the users, and a link from one node to another, means that the first user is subcribed to the other.

In the graphs we work on, each node is represented with a unique positive integer, and the links are represented by a pair of positive numbers, the first one being the source node and the second one being the destination node of the link. There are different ways in which the numbering of the nodes can be done, in the case of the web graph for example, an interesting way to do so is by sorting the nodes in a lexicographical order, and setting each node's number to it's position in that order. This method is particularly efficient in web graphs, because usually, web pages that have common prefixes in their URLs (like the domain name), tend to have many successors in common, this property is called **similarity**, they are also usually close to their successors and have successors that are close to one another, this property is called **locality**. These two properties are the basis of a major part of our graph compression technique.

The classical graph data structure implementations are the edge list, the incidence matrix and the adjacency list. The edge list, as it's name says is just a list of links, containing all of the links of the graph, it's size is $(2 \times m)$ times the size of a positive number data type. The incidence matrix, is a $(n \times n)$ matrix, in which for each pair of nodes i and j the cell $[i][j]$ of the matrix is equal to 1 if the link $\{i, j\}$ exists, and it's equal to 0 otherwise, which makes it's size $(n \times n)$ times the size of the smallest data type that can store 1s and 0s. The adjacency list data structure associates each node with a list of it's successors, it can be implemented by using an array of positive numbers of size m containing the concatenation of the lists of successors of the nodes, ordered by the number of the source nodebin increasing order, and an array of size n in which the ith cell contains the sum of the degrees (number of successors) of all the u_k nodes for $k \in \{0, 1...i\}$, which makes it's size $(n + m)$ times the size of a positive number data type. Knowing that real-world graphs are usually sparse, the adjacency list data structure is obviously the least costly of the three memory wise, and it is the data structure we based our work on.

Our programs are written in the C programming language, they can be found in https://github.com/hichaeh/PSTL-GraphCompression, for the positive numbers (the numbers of the nodes) we use the uint64_t data type which represents positive integers (and zero) that can be stored in 64 bits, so it's range of values is $[0, 2^{64} - 1]$, we chose this type to avoid all risks of overflow, for values that we know to be small like binary values, when

they are needed we use uint8_t, which is the smallest data type in C (along with char and other 8-bit data types), these two data types are available in the "stdint.h" header file.

The graph compression is done in two parts. In the first one we use variable-length compression codes to represent positive numbers using less bits that what their normal representation using the C predefined types would require. To do so efficiently, we wrote a set of functions (in PSTL-GraphCompression/bitArray.c) that enable us to do bitwise operations on uint8_t arrays, by doing so we make sure that each number takes exactly the number of bits it requires to be stored, on the other hand when using arrays of predefined types, all the values, no matter how big or small they are take the same size, which is the size of the data type, this part is detailed in Section 2. The second part consists of taking advantage of the properties of real-world graphs, especially the ones we cited previously (locality and similarity) to modify the adjacency lists in these graphs in ways that allow us to respresent the numbers of the successors of the nodes with smaller values, and still be able to recover them in a efficient manner, the main techniques used are referenciation and intervalisation, these two are detailed in Section 3.

2 Compressing an array of integers

2.1 Bit arrays

The first part of our project consists of writing functions that allow us to encode and decode integer values to and from an array of bits. In the C programming language, there is no bit sized data type, to emulate the way a bit array would work, we use uint8_t arrays, uint8_t being the type that holds an unsigned 8-bit integer, so every cell in those arrays has 8 bits, and we wrote a set of functions (in PSTL-GraphCompression/bitArray.c) that allow us to read and write a specific bit in the array, allowing us to work with those arrays as if they were bit arrays.

We apply this to our graphs by replacing the array of adjacency lists by an array of bits, and storing the successors by encoding them into the bit array. The array of accumulated degrees is replaced by an offset array in which the ith cell indicates the bit from which the encoding of the of the successors of the node i starts, and the $(i+1)$th indicates where that encoding ends, and where the encoding of the successors of the node $(i+1)$ starts.

By doing this alone there is a non negligeable gain in the memory usage of the adjacency list, but it also affects the speed in which we can access a random node from the list of successors, since not all the nodes hold the same number of bits in memory, the only way to read the kth node from the successors list is to decode the $k-1$ other nodes and the decode it to read it, so instead of a $O(1)$ access time complexity, we end up with $O(d * dt)$ as a worst case access time complexity, with d being the highest number of successors any of the nodes have, and dt being the worst case decoding time. But of course this was expected, trading speed for memory is unavoidable when you're compressing data, but since the main difficulty we face when working with real-word graphs is memory limitation, then the trade-off is worth making.

2.2 Compression codes

As we said previously, we used variable-length codes to encode and decode arrays of integers, this allows us to store integer values in memory using less bits than their usual representation in C (32 or 64 bits). These codes are Elias' γ and δ codes, the variable-length nibble code, and Boldi and Vigna's ζ_k codes [3].
For any positive integer x, let b be it's binary represenation and l the length of b.

Unary: write x $-$ 1 zeros and a one.

γ-coding: $l-1$ in unary followed by the last l $-$ 1 digits of b.

δ-coding: Write l in γ coding followed by the last l $-$ 1 digits of b.

nibble coding: Add zeros on the left of b so that l is a multiple of 3. Break b in blocks of 3 bits and prefix each block with a bit: 0 for all blocks except for the last one.

ζ_k-coding: h in unary such that $x \in [[2^{hk}, 2^{(h+1)k} - 1]]$ followed by a minimal binary coding of $x - 2^{hk}$ in the interval $x \in [[0, 2^{(h+1)k} - 2^{hk} - 1]]$.

In our case, to be able to code 0 in addition to positive integers, we modified the unary code, by adding a 0 to the left, so for a positive integer x, it's represented with x zeros followed by a one, instead of x-1 zeros. And the ζ_k codes were modified, so when we code a value x, it's coded as x+1, and when we decode a value x the returned value is x-1.

As we said previously, when we replace the adjacency list with a bit array containing the compressed lists of successors, to read a certain value from the reference list, we need to decode all the values that precede it. One way to store the adjacency list using less values, is by using the gaps technique, which needs the lists of successors to be sorted in increasing order, and for a node x with a list of successors $S(x) = \{s_0, s_1, ...s_k\}$, it's list of successors is modified and stored as $S'(x) = \{v(s_0 - x), s_1 - s_0 - 1, s_2 - s_1 - 1...s_k - s_{k-1} - 1\}$, with v being the map $v : Z \to N$:

$$v(x) = \begin{cases} 2\text{x} & \text{if } x \leq 0 \\ 2|\text{x}|\text{-}1 & \text{if } x > 0 \end{cases}$$

Since the difference between two nodes is at least 1, and by using the map v, the gaps that we store will still be unsigned integers, but with smaller values, which makes us use less bits, for the same successor lists. In the case of the web graph, these gaps in successor lists, follow a power-law distribution.

To be able choose the right compression code (with the right value for the parameter k if it's the ζ_k code) for a given distribution with a given parameter, we use pseudo-random generation algorithms that generates integers following a power law, the Poisson and the Binomial discrete probability distributions and we plotted the expected lengths using the compression codes we talked about earlier by the paramaters of these probability distributions.

The integer generation algorithms we use requires the use of the values from a uniformly distributed random varible, for this we use an implementation of the Mersenne twister[1] that is available online [5].

The Binominal distribution[2] with parameters n and p is the discrete probability distribution of the number of successes in a sequence of n independent experiments, with each experiment having a probability $p \in [0, 1]$ of succeeding and a probability $q = 1 - p$ of

[1]Mersenne Twister: https://en.wikipedia.org/wiki/Mersenne_Twister
[2]Binomial distribution: https://en.wikipedia.org/wiki/Binomial_distribution

failing. In our case it consits of picking n values using the Mersenne Twister in $[0, 1]$ and counting the number of times in which the picked value is $\leq p$.

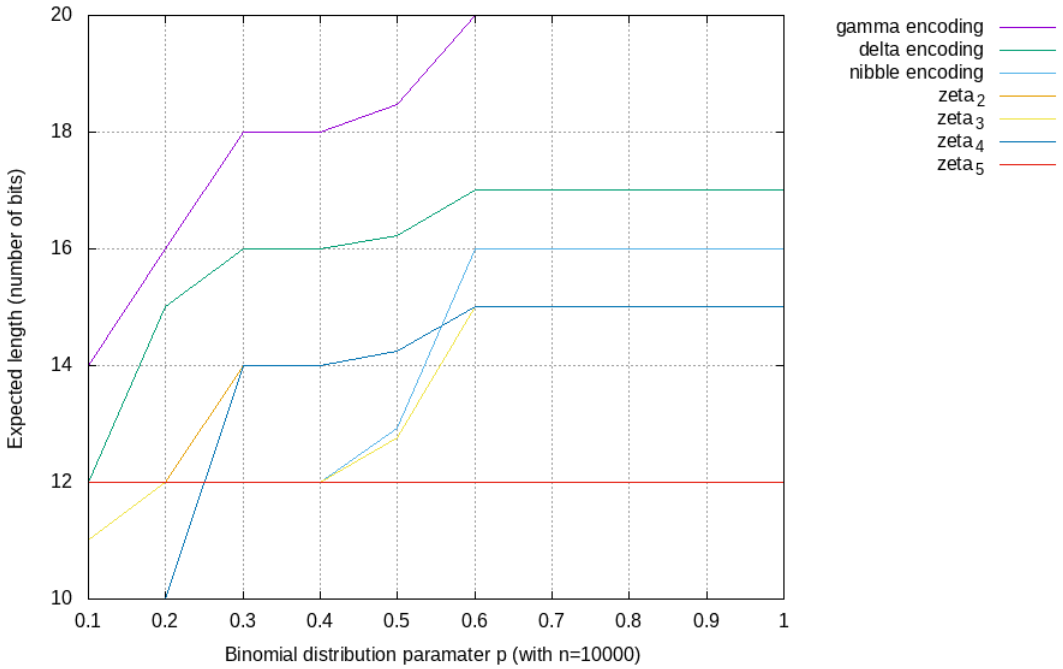

Figure 1: Expected length of compressed values generated with a binomial distribution by the distribution parameter p.

The Poisson distribution[1] is a discrete probability distribution that expresses the probability of a given number of events occurring in a fixed interval of time or space, when we know the expected number of occurences of those events λ. The distribution's formula is:

$$Po(x) = \frac{e^{-\lambda} \times \lambda^x}{x}$$

[1]Poisson distribution: https://en.wikipedia.org/wiki/Poisson_distribution

With x being the actual number of occurences of the event. To generate this probability distribution, we implemented in the C programming language Knuth's algorithm for the Poisson distribution. [4]

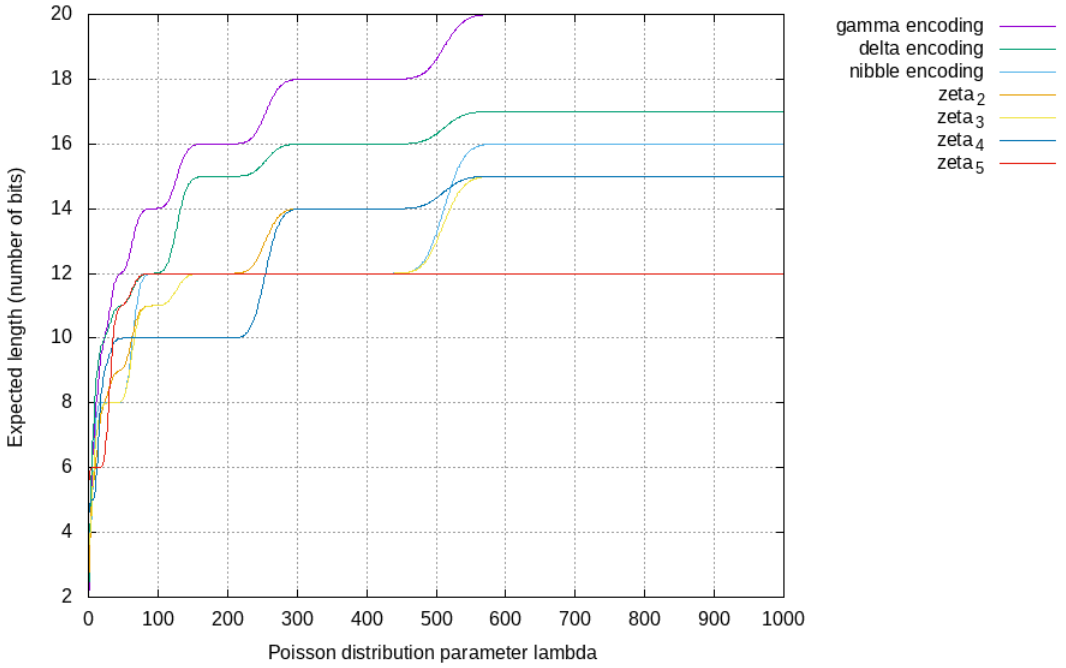

Figure 2: This figure represents the expected length of compressed values generated with a Poisson distribution by the distribution parameter λ.
We can notice that for λ between 0 and 200, the expected lengths of the codes cross each other more than they for higher values of λ. We notice a certain stability starting from $\lambda = 300$ up, γ and δ codes have the highest, expected legth, ζ_k codes have a lower expected length the lower the k, and nibble and ζ_3 codes are equal.

A power law[1] is a functional relationship between two quantities, where one of them is a power of the other. We work with Zipf's law[2] which is a discrete version of the power law, and it's formula is:

$$Pl(x) = \frac{1}{x^{\alpha} \times \sum_{i=1}^{n} (\frac{1}{i})^{\alpha}}$$

From the figures 1, 2 and 3, we notice that there isn't a good for all compression code, and that depending on how the values in the successor lists in the graph are distributed, choosing the right compression code could affect in significant way the compression rate.

3 Graph compression

This second part of our project, is based on the properties we talked about previously in real world graphs, locality and similarity, we take advantage of those to change the way values

[1]Power law: https://en.wikipedia.org/wiki/Power_law
[2]Zipf's law: https://en.wikipedia.org/wiki/Zipf's_law

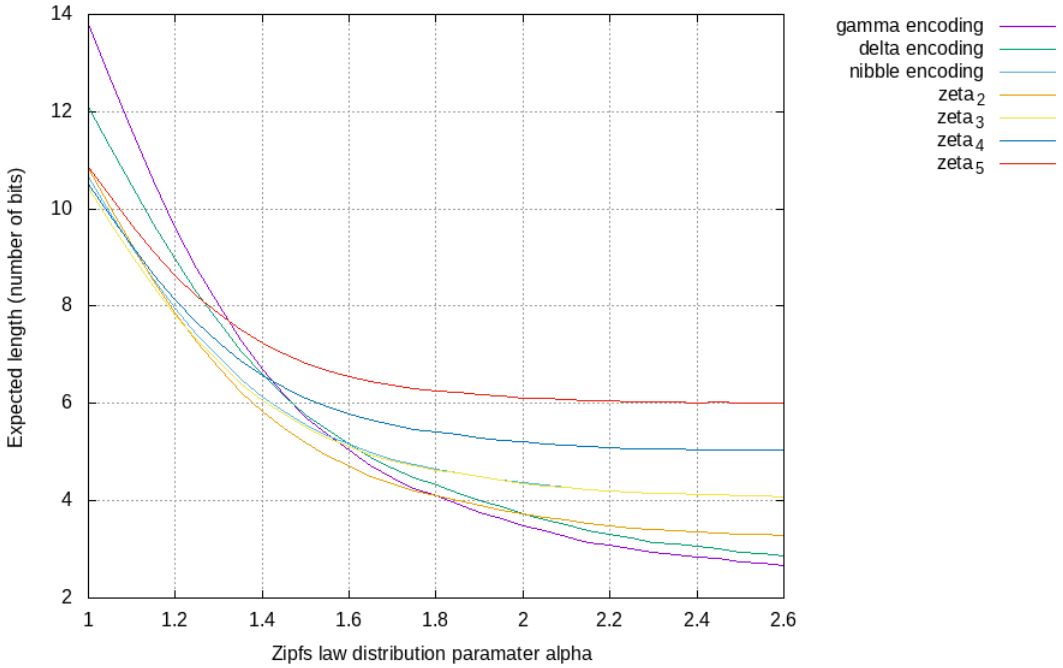

Figure 3: A figure representing the expected length of compressed values generated with Zipf's distribution by the distribution parameter α.

The first thing we notice is how for $\alpha < 1.3$ γ and δ codes have the highest expected lengths, and for $\alpha > 2.05$ they have the the lowest, another thing we notice is that contrary to the other graphs, in this one, the expected lengths of ζ codes increase as the parameter k increases, in this one as well nibble code and ζ_3 have the same expected lengths.

are stored in the successor lists, it's done with two principal techniques, referenciation and intervalisation that we will detail next.

3.1 Referenciation

This one uses the similarity property. Nodes that are close to one another tend to have similar successors in their successor lists. This techniques works in the following way: when we are about to store a successor list of a certain node, we compare it with a certain number of the successor lists of the preceeding nodes, the number of successor lists with which we compare the current list is called the **window size**, and the set of those succesor lists is called the **window**. We select the successor list of the preceeding node that has the most successors in common with the current one, we call that node the **referenced node**, and we make a copy list of it's successor list. A copy list is a list of bits that have the same size as the successor list of the referenced node, and in which the ith cell contains 1 if the ith successor of the referenced node is also a successor of the current node, and it contains a 0 otherwise.

The successor lists of our graphs are stored in an increasing order of their source nodes, therefore the current node has to be biggest than the referenced node. When we are about to store the successor list of the current node, we first store the **reference** which is the value $(currentnode - referencednode) \geq 0$ followed by the copy list (if the reference is

superior to 0, if the reference is equal to 0 it means that none of the successor lists of the window have nodes in common with the current successor list) and then followed by the left over nodes, which are successors to the current node, but aren't successors of the previous one, these left over values will be stored using the gaps method we explained in section 1.

Example:

Let's say we have a node x which has a the following successor list: $[1, 2, 5, 7, 9]$
If there is a node $x - v$ that has the following successor list: $[3, 4, 5, 8, 9, 10, 11]$
And if $x - v < x - windowsize$, that means that $x - v$ is part of the window, and if we suppose that none of the other successor lists of the nodes in the window have more values in common with $S(x)$ then the successor list of x will be stored this way:
$[ref = v, copylist = [0010100], leftovers = [1, 2, 5]]$
This is especially effective on long successor lists, in which the stored values are big.

A way to make this better, is by replacing the copy lists with a list of copy blocks. A block represents a sequence of 1s or 0s, we replace the copy list by a list containing the lengths of the blocks, considering the 1st to always be a 1-block, and by ignoring the last block because it's type can be deducted by the type of the previous one (it's the opposite of it) and it's length is the length of the referenced list minus the sum of the lengths of the encountered blocks.
So the list of successors of x from our last example will be coded this way:
$[ref = v, copyblocks = [02111], leftovers = [1, 2, 5]]$
In this case since the copylist starts with two 0s, the first block which is a 1-block has a size of 0, the last one, which is the 6th, is a 0 block and it's size is the size of the referenced successor list so 7 minus the sum of the sizes of the previous block 5 is equal to 2, and the copy list ends with two 0s so it's equivalent while in the case of very long sequences of 0s and 1s, they are stored using less bits. When it comes to the paramaterers of the referenciation, havin a bigger window size of course allows us to choose better reference noes, but comparing the successor lists one by one is costly, and doing too much of it will slow down loading the graphs. Another importat paramater in the maximum reference count, knowing that we can have very long references chains (S(x) that referes to S(y) that referefs to S(z) ...). Setting a maximum reference count as a limit of the size of that reference chain will be very usefull especially when it comes to the decoding time of the graph.

3.2 Intervalisation

This method uses the locality property, when the nodes are numbered correctly, the successors of a node tend to be close to one another. Intervalisation comes into play whe we have a sequence of successors in which every successor is equal to the successor that preceeds it plus 1, instead of storing this sequence, we only store the left extreme, which is the first value of the sequence, and it's length. Obviously for this to work the successor lists need to be sorted in increasing order. On the entire successor list, if we have many intervals, we store the number of intervals after the reference and the list of copy blocks (if there is one), by writing the number of intervals there are, their left extreme values, and then their lengths, but we only consider intervals starting from a certain size, that's called the **threshold**, it's minimum value is 2, because there can't be an interval with less

than two values. All the intervals that have a length that is at least equal to the threshold will be coded with this technique. For the left over values that were neither coded by referenciation, nor by intervalisation, we will code them using the gap technique at the end of the reference list.

Example:

Let's say we have the following list of left over values after doing a compression with copy blocks on a list of successors:

$[ref, copyblocks, leftoves = [8, 9, 10, 11, 14, 15, 16, 19]]$

if the threshold is 3, then after compression that list becomes:

$[ref, copyblocks, leftoves = [2, [8, 14], [4, 3], 19]]$

if the threshold is 4, then after compression that list becomes:

$[ref, copyblocks, leftoves = [1, [8], [4], 14, 15, 16, 19]]$

A way to only store the needed values is, for every length of an interval, since it's always greater or equal to the threshold, for every length we decrease it's value by the threshold vefore story it. And for the left extremes we decrease them by the value of the previous left extreme, since the successor lists are sorted, the last reached left extreme is always greater than the previous one. In the case of the first left extreme, we use the map v we defined in section 2.2.

3.3 The offset array

This is the last compression technique we use, which consists of modifying the offset array, given a value J, called the jump, instead of stoting the offset of every node's successor list in the adjacency list, we only store the offsets of $[0, J, 2J...]$, depending on a positiver number J. knowing that each offset is stored as a 64-bit unsigned integer, decreasing their number will affect greatly the total size of the compressed graph. But to be able to do so we need to add another value to the lists of successors, before storing the reference, we will store the number of bits that we will use to store all the other values of the compressed successor list. This is useful when we want to access the successor list of a node that isn't a multiple of J, in which case we have to iterate over all the successor lists of the preceeding nodes that belong to the same chunk of successor lists. To do so, instead of reading the entire lists, we just read the first value, if we're on the right list, we decode it, otherwise we jump to the next. This was also done to fasten the decompression speed.

3.4 The final compressed graph

Our final graph data structure (in: PSTL-GraphCompression/compAdjList.c) is built by composing all the previous compression techniques. Our program expects an edge list as input, stored in a file as a pair on ints on every line, separated by a space. The file needs to be sorted first by the first column (the source nodes of the links) then by the second column (the destination nodes), this garanties that when we read the file, the source nodes are received in an increasing order, the successor lists are sorted as well.

For each successor list we read $S(x)$, we choose a successor list $S(y)$ from the window that has the most successors in common with $S(x)$, if none of them have nodes in common with $S(x)$, then we don't do a compression by reference, otherwise $S(y)$ will serve as a reference list for $S(x)$, all the successors of $S(x)$ that are also successors of $S(y)$ are stored

with the copy blocks method. Then we iterate over the left over values to find which ones form an interval, those that do are stored with the intervals method. For the residual values, they are stored with the gaps technique. And of course all of these values are written using a compression code.

The following tables represent the results of this compression on some real world graphs, we used the ζ_4 compression code for storing the values.
Keys to read the tables:

- window size: it's the number of successor lists in the window, the window contains the ws previous successor lists, that were stored before the current one.
- max ref count: it's the maximum length of the reference chain.
- threshold: the minimum length of an interval, for it's values to be stored with the intervals technique, must be superior to 2.
- jump: the length of the chuncks of successor lists that we refer two in the offset array.
- n: the number of nodes, e: the number of links

orkut (n=3,072,441 e=117,185,083 bits/link=120.819)					
window size	max ref count	threshold	jump	bits/link	compression time
5	5	2	5	15.512	0h1m40s
5	5	5	5	15.335	0h1m41s
25	5	2	5	15.468	0h1m51s
5	25	2	5	15.498	**0h1m36s**
5	5	2	25	15.243	**0h1m36s**
25	25	2	25	15.135	0h1m57s
25	25	5	25	14.975	0h2m9s
50	50	2	50	15.069	0h2m27s
50	50	5	50	**14.913**	0h2m20s
50	50	25	50	14.923	0h2m21s

Figure 4: This table represents the compression rate and the execution time of the compression on the Orkut graph , with various configurations.

twitter2010 (n=41,652,229 e=1,468,365,182 bits/link=142.428)					
window size	max ref count	threshold	jump	bits/link	compression time
5	5	2	5	15.137	0h15m32s
5	5	5	5	14.944	0h15m31s
5	5	25	5	14.963	**0h15m23s**
50	5	2	5	15.202	0h18m54s
5	50	2	5	15.059	0h15m48s
5	5	2	50	14.811	0h15m33s
50	50	2	50	14.690	0h22m53s
50	50	5	50	**14.514**	0h22m53s
50	50	25	50	14.522	0h22m53s

Figure 5: This table represents the compression rate and the execution time of the compression on the twitter2010 graph with various configurations.

uk-2007-05 (n=105,896,434 e=3,738,733,648 bits/link=143.195)					
window size	max ref count	threshold	jump	bits/link	compression time
5	5	2	5	5.877	0h28m54s
5	5	5	5	5.860	0h28m42s
5	5	25	5	6.378	**0h28m30s**
50	5	2	5	6.647	0h31m24s
5	50	2	5	5.026	0h28m48s
5	5	2	50	5.047	0h28m41s
50	50	2	50	4.399	0h34m36s
50	50	5	50	**4.375**	0h34m38s
50	50	25	50	4.546	0h34m25s

Figure 6: This table represents the compression rate and the execution time of the compression on the uk-2007-05 graph with various configurations.

What we can take from the Figures 4, 5 and 6, is that one thing that determines the effectiveness of the compression, is the shape of the graph, and to which extent we can take advantage of the locality and similarity properties of that graph. When we first look at the best compression rate for the Orkut [1] and the twitter2010 [2] graphs, we notice that they are close to one another and they reach the best compression rate at the same configuration, which is interesting since both are social networking websites. On the other hand the compression rate of the graph uk-2007-05[3] [2, 1] is a lot higher probably due to it being from a section of the web graph, which is where the two properties come from. What we can also notice is that contrary to what we could assume simply increasing the window size does not always give a better compression rate, if we look at lines 1 and 4 of the figure 5,

[1] http://snap.stanford.edu/daa/com-Orkut.html
[2] https://snap.stanford.edu/data/twitter-2010.html
[3] http://law.di.unimi.it/webdata/uk-2007-05/

or 1 and 4 of the figure 6. From lines 8, 9, 10 from figure 4, and 7, 8, 9 from figure 5 and 6, we notice that having a threshold that is too small or too big might affect negatively our compression, so the choice of the right threshold is important. When it comes to the jump we notice that making it bigger is better, but that would slow down the access time of algorithms like BFS (Breadth First Search) that need random access to nodes (it can be tested with BFS and PageRank in PSTL-GraphCompression/compAdjList.c), and not only sequential access to successor lists. The same this with the max ref count, increasing is better, but having a max ref count that is too high will affect negatively the random access time to the successor lists.

4 Conclusion and perspectives

As we said previously, graph analysis is an important field in computer science and it has many real world applications (transportation, social media ... ect) but as we saw, in many cases when we work on real-world graphs we are faced with an obstacle that is their size and how big they get, and the fact that most modern machines have a random access memory that is too limited to hold such graphs, other solutions are essential if we want to work on these graphs, and we have shown in this project, that the graph compression we implemented is definetely one way to overcome that problem effectively. But as our results show, choosing the right compression code and the right configuration for the compression is important, and it affects in a non negligeable manner the efficiency of the compression.

References

[1] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In S. Srinivasan, K. Ramamritham, A. Kumar, M. P. Ravindra, E. Bertino, and R. Kumar, editors, *Proceedings of the 20th international conference on World Wide Web*, pages 587–596. ACM Press, 2011.

[2] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[3] P. Boldi and S. Vigna. The webgraph framework ii: Codes for the world-wide web. In *Data Compression Conference, 2004. Proceedings. DCC 2004*, pages 407–429, Mar. 2004.

[4] D. E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[5] E. Sultanik. The mtwister c library. https://github.com/ESultanik/mtwister, 2014.