On the application of WebGraph compression to social network graphs

Social Network Analysis for Computer Scientists — Course paper

Chika Opdam
c.opdam@umail.leidenuniv.nl
LIACS, Leiden University

Manuel Pérez Belizón
m.d.perez.belizon@umail.leidenuniv.nl
LIACS, Leiden University

ABSTRACT

Efficient compression of large scale graphs is crucial for storing and analyzing networks such as social graphs and web graphs. The WebGraph framework, introduced by Boldi and Vigna, leverages structural properties of web graphs to achieve high compression ratios. But applying these methods to social graphs presents challenges due to their distinct structural characteristics. Such as higher reciprocity and weaker hierarchical organization.

In this paper, we investigate wether node reordering techniques can improve compression of social graphs using the WebGraphs framework. We approach this problem by partitioning the graph into communities using the Louvain method, sorting the comminutes and nodes based on various metrics, and traversing the graph using BFS or DFS to relabel nodes.

Our results show that while traversal significantly improves compression ratios, the specific order of communities or nodes has no concrete impact on compression ratios. Additionally we found that compression ratios for social graphs where not as low at those for web graphs, which is likely due to the reciprocity of the edges in social graphs.

These findings show the limitations of simply applying web graph-oriented compression methods to social graphs without changing the compression method.

KEYWORDS

graph compression, web graph, social network analysis, network science

ACM Reference Format:
Chika Opdam and Manuel Pérez Belizón. 2024. On the application of WebGraph compression to social network graphs: Social Network Analysis for Computer Scientists — Course paper. In *Proceedings of Social Network Analysis for Computer Scientists Course 2024 (SNACS '24).* ACM, New York, NY, USA, 7 pages.

1 INTRODUCTION

In the era of big data, it has become increasingly important to find ways to efficiently store and process large datasets. Social networks, which model relationships and interactions between social actors, are a subset of such datasets. Due to their size and sparsity, efficient compression of social network graphs is vital for optimizing storage and enabling analysis.

In this paper, we investigate the possible application of the WebGraph compression method, proposed by Boldi and Vigna in [5], to social networks. The WebGraph framework achieves compression of web graphs by leveraging the lexicographical ordering of URL's, which minimizes the size of gap distributions in adjacency lists. Social networks are structured differently than web graph, which raises the question: Is it possible to use the WebGraph compression method to compress social networks if their nodes are reordered?

In order to find an answer to this question, we experiment with various methods for relabeling the nodes of social network graphs to create a gap distribution that approximates the properties of a lexicographically ordered web graph. We approach this problem in the following way: We first read the social network graph. Next, we apply a community detection algorithm to split the graph into communities. These communities are then ordered based on metrics such as size, edge count, modularity contribution, and edge density. Then we go through each community in order and order their nodes based on their out- and in-degrees. Then we traverse through each node in each community using breath-first-search (bfs) or depth-first-search (dfs). Lastly, we will relabel each node based on traversal order.

By doing this, we aim to create a relabeling strategy that optimizes the compressibility of social network graphs using the WebGraph framework.

The paper is structured as follows. In Section 2, we discuss other papers related to this paper, in Section 3 we introduce the necessary context and definition to understand this paper, in Section 4 we detail the methodology used for node relabeling and compression, in Section 5 we describe the data used for our experiments, in Section 6 we discuss the experiments we ran for the paper and list the results of these experiments, and finally, in Section 7 we conclude our paper and discuss future work.

2 RELATED WORK

With the increasing size and complexity of graph datasets such as web graphs and social networks, graph compression has been a very active area of research.

This paper is based on the WebGraph framework proposed by Boldi and Vigna [5], which is a foundational method for web graph compression. This framework uses various structural properties of web graphs, such as locality and similarity to achieve high compression rates while still allowing efficient graph traversal and querying times. Since the first introduction to the webgraph framework it has been expanded upon in subsequent research. For example, by

SNACS '24, Master CS, Fall 2024, Leiden, the Netherlands

porting the framework to rust, Fontana et. al. [9] achieved a significant speedup to the compression method compared to the original framework. And Versari et. al. introduced the Zuckerli compression system [16] which improves on the WebGraph in multiple ways using heuristic graphs algorithms.

Aside from the WebGraph framework, various compression methods have been proposed, like the K2-Trees [7] which makes use of large empty areas of adjacency matrices of web graphs to achieve high compression ratios.

A survey of various lossless graph compression methods for both web graph compression and social network compression has been done in [2].

3 PRELIMINARIES

In this section we will discuss various definitions and notations needed to understand the paper.

3.1 Web Graphs

A web graph $G = (V, E)$ is a directed graph consisting of a set of vertices representing web pages V, and a set of edges E representing URLs. If there is an edge $(u, v) \in E$ where $u, v \in V$ this means that webpage u has an URL which directs to v. These web graphs have some key structural features.

(1) **Locality**: Hyperlinks often link to pages hosted on the same domain or pages with similar topics. This results in clusters of closely related nodes.
(2) **Similarity**: Successor lists of nearby nodes often have a large overlap.

These properties are exploited for a high compression ratio in the WebGraph compression method [5].

3.2 WebGraph Compression Format

The WebGraph framework uses the previous mentioned structural features of webgraphs to achieve high compression rates using the following strategies:

- **Lexicographical node ordering**: Nodes are ordered lexicographically (by URL's). This, due to the local and lexicographically consecutive nature of web graphs, minimizes the gaps between nodes in adjacency lists.
- **Gap Encoding**: By storing gaps between the successor nodes in the adjacency list instead of the successor nodes themselves, networks with smaller gaps in their adjacency lists, such as web graphs, will result in fewer required bits for encoding.
- **Reference encoding**: A successor list which is similar to a previously stored successor list will be stored as a reference to the previous list, with only the differences between the lists encoded.
- **Interval encoding**: Intervals occurring in adjacency lists will be stored by their left extreme node and the length of the interval instead.

3.3 WebGraph Codes

In the field of Network Analysis power-law distributions are very common, for example, power-law distribution can be seen in parameters like in degrees, this also happens on web graphs compression when ordering lexicographically the URLs of a web graph one obtains ordered sequences of successors with small gaps between adjacent successors. The distribution of these gaps is usually a power-law distribution with exponent less than 2.

The WebGraph framework proposes a set of codes ζ, that are particularly good for storing web graphs or, in general, integers with power-law distribution in with exponent less than 2, just like gaps in successors lists in web graphs.

We will call these set of codes zeta-k codes, as they use a parameter k. So let x be a positive integer to be coded, b its binary representation, and l its length. Then zeta-k codes write $l - k$ in unary followed by the last k bits of b.

Different values for this parameter k will imply power-law distribution with different exponents

Some other codes are presented:

- Unary code. Write $x - 1$ zeroes followed by a one.
- Elias' γ code, we will call it δ code or delta code. Write l in unary, followed by the last (least significant) $l - 1$ digits of b.
- Elias' δ code, we will call it δ code or delta code. Write l in γ code, followed by the last (least significant) $l - 1$ digits of b.
- Variable-length nibble code, we will call it nibble code. First, pad the binary representation of $x - 1$ on the left with zeroes to obtain a string whose length is a multiple of three. Then, break the string into blocks of three bits, and prefix each block with a bit, which is zero for all blocks except for the last one.

The examples of how numbers from 1 to 16 are encoded using the different codes is shown in Figure 1. These codes will be later use to represent the different parts of the presented compression format on the next subsection and more detailed explanations and analysis can be found in the paper "Codes for the World Wide Web" by Boldi and Vigna. [6].

Integer	$\gamma = \zeta_1$	ζ_2	ζ_3	ζ_4	δ	nibble
1	1	10	100	1000	1	1000
2	010	110	1010	10010	0100	1001
3	011	111	1011	10011	0101	1010
4	00100	01000	1100	10100	01100	1011
5	00101	01001	1101	10101	01101	1100
6	00110	01010	1110	10110	01110	1101
7	00111	01011	1111	10111	01111	1110
8	0001000	011000	0100000	11000	00100000	1111
9	0001001	011001	0100001	11001	00100001	00011000
10	0001010	011010	0100010	11010	00100010	00011001
11	0001011	011011	0100011	11011	00100011	00011010
12	0001100	011100	0100100	11100	00100100	00011011
13	0001101	011101	0100101	11101	00100101	00011100
14	0001110	011110	0100110	11110	00100110	00011101
15	0001111	011111	0100111	11111	00100111	00011110
16	000010000	00100000	01010000	010000111	001011001	10000111

Figure 1: Examples of the different codes

3.4 WebGraph Compression Format

The final WebGraph compression format described in [5] uses different compression techniques like intervals, differential, reference and

gaps compression. This final compression format can be controlled by three parameters, R maximum reference count, W window size and L_{min} minimum length of the interval. These parameters offer flexibility in terms of speed and compression. The experimental results obtained in [5] used a fixed $L_{min} = 3$ and a fixed code zeta-3.

4 APPROACH

We want to explore different node orders to load into the WebGraph framework to see what node ordering best lends itself to the WebGraph compression method. This exploration of different node orders is driven by the observation that node order impacts the gap distribution of the adjacency lists, which impacts the compression efficiency.

This was done in two steps:

(1) Graph pre-processing. This is done in Python using the NetworkX package [11], which involves preparing the social graph by reordering its nodes.
(2) Loading the adjacency list of the processed graph into the WebGraph framework [3] to assess the compression performance.

4.1 Graph Pre-Processing

The pre-processing of the social graphs is done in multiple parts, each for testing hypotheses about how node and community ordering affects compression.

Community Detection. First the social graph will be divided into communities. This is done using the Louvain community detection method, a widely used algorithm for community detection which optimizes modularity. This is done because nodes within a community tend to have more connections with eachother than with nodes outside of the community, which we may equate to the locality we mentioned is a feature in web graphs. And which is a key factor used in WebGraph Compression.

Community Sorting. After we divided the graph into communities, we will order the communities based on some metric. This will determine the order of the communities we will traverse when relabeling. This will inform us on how structural properties of a graph influences compression rates.

The different metrics for ordering the communities are as follows:

- *Nothing*: Simply return a list communities without a specific order, this will serve as a base case to compare against to test the effectivity of community sorting.
- *Size*: Larger communities may contain more links and thus have a greater impact on locality
- *Modularity contribution*: Communities contributing more to the modularity of the graph may have a stronger internal cohesion, which could impact locality.
- *Edge count*: A large number of edges may mean that these will increase locality and thus intervals in the adjacency list.
- *Edge density*: As with the edge count, denser communities could mean that we can make good use of the interval compression method.

The implementation of these orderings follow the structure below, where they receive the number of communities and the subgraph of each community:

```
1 def
      community_sort_by_edge_count(communities,
      subgraphs):
2     sorted_communities =
      sorted(subgraphs, key=lambda x:
      x.number_of_edges(),
3
      reverse=True)
4     print(f"Found {communities}
      communities with highest edge count"
5           f"
      {nx.density(sorted_communities[1])}
      and smallest edge count"
6           f"
      {nx.density(sorted_communities[-1])}")
7     return sorted_communities
```

using other Networkx functions for the different orderings like density or modularity.

Node Sorting. The nodes in each community will then also be ordered. This will serve as a priority ranking when traversing the nodes within a community. This will show us how node attributes impact compression:

- *Nothing*: Simply return a list nodes without a specific order, this will serve as a base case to compare against to test the effectivity of node sorting.
- *Out degree*: Return a list of nodes ordered by out degree.
- *In degree*: Return a list of nodes ordered by in degree.

The implementation of these orderings follow the structure below, where they receive the subgraph of a community:

```
1 def node_sort_by_outdegree(C):
2     sorted_nodes = sorted(C.nodes,
      key=lambda x: C.out_degree(x),
      reverse=True)
3     return sorted_nodes
```

Graph traversal. After we have divided the graph into communities and sorted the communities as well as the nodes within each community, we will perform graph traversal within each community using a priority ranking.

We have implemented multiple graph traversal orders.

- *Nothing*: Leave the nodes as they are and do not traverse the graphs. This will serve as a baseline for graph traversal.
- *Breath first search*: We traverse the communities breath first search, starting with the highest priority node. When choosing a successor to go to next, we will choose the successor with highest priority which has not been visited yet.
- *Depth first search*: Same principle as with breath first search. We traverse the community depth first search while keeping in mind the priority of each node.

```
1 def bfs(C, sorted_nodes):
2     visited = set()
3     bfs_order = []
4
```

```
5     queue = deque()
6     node_index = {node: i for i, node in
      enumerate(sorted_nodes)}
7
8     for node in sorted_nodes:
9         if node not in visited:
10            queue.append(node)
11        while queue:
12          current = queue.popleft()
13          if current not in visited:
14              visited.add(current)
15              bfs_order.append(current)
16              nb =
      sorted(C.successors(current),
      key=lambda x: node_index[x])
17              queue.extend(nb)
18
19
20    return bfs_order
```

```
1 def dfs(C, sorted_nodes):
2     visited = set()
3     dfs_order = []
4     stack = []
5
6     node_index = {node: i for i, node in
      enumerate(sorted_nodes)}
7
8     for node in sorted_nodes:
9         if node not in visited:
10            stack.append(node)
11            while stack:
12                current = stack.pop()
13                if current not in visited:
14                    visited.add(current)
15
      dfs_order.append(current)
16                    nb =
      sorted(C.successors
      (current),key=lambda  x:
      node_index[x])
17                    stack.extend(nb)
18    return dfs_order
```

Two key optimizations were used to reduce considerably the computation time:

- Precomputing the node indices, line 6 in both algorithms, instead of using the built in *index* function (which is $O(n)$) and calculating the indexes on every call.
- Using a deque in the BFS algorithm is more efficient for popping elements from the front compared to a list, as `list.pop(0)` has $O(n)$ complexity while `deque.popleft()` has $O(1)$ complexity.

In the Python implementations for bfs and dfs with priorities shown above we traverse all V vertices once and for every vertex we sort its neighbors with time complexity of $O(k \log(k))$ in the worst case where k is the maximum outdegree, we also traverse all E edges exactly once, therefore both of these algorithms have a time complexity of $O(E + V * k \log(k))$ where E is the number of edges, V is the number of vertices and k is the number of neighbors.

Relabeling. The nodes of the social graph then are relabeled based on the order that the nodes were visited during graph traversal. This relabeling determines the gap distribution in the adjacency list, which is a primary factor determining WebGraph compression efficiency.

Exporting adjacency list. Lastly, a `.graph-txt` is generated, the file format is as follows: the first line contains the number of nodes, n. Then, n lines follow, the i-th line containing the successors of node i in increasing order (nodes are numbered from 0 to $n-1$). Successors are separated by a single space.

4.2 Compressing Social Network

Once we have the adjacency list as a `.graph-txt` file, we use the WebGraph framework [3] for Java. We use the class `ASCIIGraph`[4] to load the graph using the `loadOnce` method passing the adjacency list file as a `inputFileStream` as parameter. Then we use the `store` method of the `BVGraph`[4] class that compresses the graph using the compression format described in Section 3.2. We can pass as argument different parameters to this method like the window size, the maximum reference count, the minimum interval length or the codes. Once the graph is compressed, a `.property` file is generated, this file contains self-explaining entries that are necessary to correctly analyze the graph like the bits per link, bits per node and compression ratio.

This way we can generate property files for each ordering and compare some of their entries to evaluate which of them gets the best compression.

5 DATA

The datasets used to run our code are:

- The Stanford slashdot dataset [14], which depicts friend/foe links between the users of Slashdot in 2009
- The prosper user loans dataset [12], depicting loans between users on Prosper.com which is a peer-to-peer lending marketplace
- A flickr follower dataset [8, 13, 15, 17], containing a social graph depicting users following other users on flickr

A table with the characteristics of the datasets are listed in Table 1.

Dataset	Nodes	Edges	Mean Degree	gcc
SLASHDOT	82, 168	948, 464	11.54	0.06
PROSPER	89, 269	3, 394, 979	38.03	0.00
FLICKR	214, 626	9, 114, 557	42.47	0.08

Table 1: The datasets used in this project with their number of nodes, number of edges, mean degree and global clustering coefficient (gcc) listed.

All the used datasets represent social networks with directed edges, are big enough networks and have different structure with different proportion of nodes and edges.

For every dataset we download the compressed dataset file from their respective sources, we then unpack those compressed files extracting the network edge list into a `Networkx Digraph` so it is ready to the pre-processing described in 4.

6 EXPERIMENTS

In this section we will discuss the experiments we performed on all of our datasets.

6.1 Experimental setup

For our experiments we used two different environment setups:

(1) Python environment for pre-processing graphs using Networkx:
   - The Python environment was built in Google Colab[10].
   - Using Python notebooks allowed us to keep the code organized.
   - Google Colab CPU computing unit with 12 Gb of RAM was enough for running all our Python experiments.

(2) Java environment for compressing graphs using the WebGraph framework:
   - We used the Eclipse Integrated Development Environment (IDE) for development and execution.
   - Maven[1] allowed us to define the required dependencies in a centralized *pom.xml* file, automatically downloading and integrating the necessary libraries and dependencies into the project.

6.2 Results

The final results of each ordering and each dataset lie in the property file generated when compressing each graph.

With the help of a simple Python script we can extract and group in a file the entries we want to compare from each property file.

6.2.1 Community Ordering. Here we will evaluate the impact of the different community orderings, presenting the compression ratios achieved using the default WebGraph configuration to compress graphs($R = 3, W = 7, L_{min} = 4$ and zeta-3 code), node outdegree ordering and breath-first search traversal.

Ordering	Slash	Prosper	Flicker
None	0.881	0.736	0.722
Size	0.883	0.738	0.722
Modularity	0.88	0.736	0.72
Edge density	0.87	0.735	0.717
Edge count	0.883	0.737	0.721

Table 2: Compression ratios for the different community orderings

As shown in Table 2 compression ratios we can see that the impact of the different community orderings is very low or none.

6.2.2 Node Ordering. Here we will evaluate the impact of the different node orderings, presenting the compression ratios achieved using the default WebGraph configuration to compress graphs, edge density community ordering and breath-first search traversal.

Ordering	Slash	Prosper	Flicker
None	0.881	0.73	0.726
Outdegree	0.88	0.736	0.717
Indegree	0.879	0.72	0.717

Table 3: Compression ratios for the different node orderings

As shown in Table 3 compression ratios we can see that again the impact of the different node orderings is very low or none.

6.2.3 Graph Traversal. Here we will evaluate the impact of the different graph traversal orderings, presenting the compression ratios achieved using the default WebGraph configuration to compress graphs, edge density community ordering and outdegree node ordering.

Traversal	Slash	Prosper	Flicker
None	0.93	0.803	0.868
BFS	0.881	0.736	0.717
DFS	0.885	0.737	0.717

Table 4: Compression ratios for the different traversal methods

As shown in Table 4 compression ratios we can clearly see that our graphs compress better after a BFS or a DFS graph traversal ordering rather than no traversal at all.

We made a more extensive search for the better compression ratio, getting the compression ratios for every possible combination from that exhaustive search we got the following interesting results:

- The worst compression ratio for each dataset is 0.938 for Slash, 0.857 for Prosper and 0.886 for Flickr, all these results were obtained by compressing the graph with no reordering at all,no community, no node and no traversal order ordering combination.
- The best compression ratio for each dataset 0.871 for Slash, 0.72 for Prosper and 0.717 for Flickr, these results were obtained by a modularity contribution commmunity order, indegree node ordering and no traversal for Slash and edge density community ordering, indegree node ordering and BFS traversal ordering.

In our final experiment we are able to see the compression ratios using different zeta-k codes, for this experiment we used the best and worst configuration for each dataset, results are shown in Table 5 and Table 6.

As explained in [6] the performance of the different zeta-k codes depend on the exponent α of the power-law distribution of the gaps, the suggested zeta-k codes for the different ranges of α are summarized in the Table 7.

The results seen in Table 5 and Table 7 suggest that the best configurations, generate neighbors lists with a power-law gap distribution with exponent α between 1.16 and 1.27 for Prosper and

Zeta-k code	Slash	Prosper	Flicker
1	1.106	0.874	0.889
2	0.913	0.738	0.744
3	0.871	0.72	0.717
4	0.868	0.733	0.724
5	0.877	0.755	0.741
6	0.9	0.794	0.766

Table 5: Compression ratios for the different zeta-k codes, best configuration

Zeta-k code	Slash	Prosper	Flicker
1	1.206	1.08	1.145
2	0.989	0.893	0.937
3	0.938	0.857	0.883
4	0.929	0.86	0.886
5	0.939	0.876	0.895
6	0.956	0.901	0.907

Table 6: Compression ratios for the different zeta-k codes, worst configuration

Zeta-k code	α
Zeta-1	[1.57, 2]
Zeta-2	[1.27, 1.57]
Zeta-3	[1.16, 1.27]
Zeta-4	[1.11, 1.16]
Zeta-5	[1.08, 1.11]
Zeta-6	[1.06, 1.08]

Table 7: Suggested ranges for codes

Flicker datasets and between 1.11 and 1.16 for Slash dataset, said gaps distributions are shown in Figure 2.

7 CONCLUSION

In this paper, we investigated if the WebGraph compression method could be efficiently applied to social graphs if we apply some node reordering strategies. We have done this by comparing different ways to reorder communities within a graph, different ways to order nodes within each communities, and graph traversal methods.

We found that the specific order of communities or nodes has no significant impact on the compression ratios. But, the act of traversal itself does have a significant impact on compression, whether this is done by breath-first-search or depth-first-search. This is because traversal inherently introduces a degree of locality in the adjacency list, which minimizes gaps in the adjacency lists which is crucial for WebGraph's compression method.

7.1 Observations

What we observe from Table 2 is that community ordering has no real effect on compression. As not ordering the communities or ordering the communities using various metrics do not significantly impact the compression in a positive or negative way. The same can be said for node ordering as can be seen in Table 3. Though traversal, as seen in Table 4 does have a noticeable impact. This is because traversal ensures that nodes are processed in a way that often gives neighbors similar value, increasing locality. But additional ordering introduced by sorting communities and nodes based on metrics such as size, edge density or in- and out degrees seem to be negligible.

Similarly, we observed from Table 5 that using the optimal zeta-k code suggested in 7 also does not have a meaningful impact.

In general, compression ratios were not as low as we had hoped. A possible explanation for this lies in the uniformity of connectivity patterns within social graphs. Since unlike web graphs, where the graph structure often exhibits a strong hierarchical structure (e.g. a webpage A links to B but B does not link to A) social graphs usually have a lot of bidirectional edges due to reciprocal relationships (if A is friends with B, B is also friends with A) which gives us less inherent asymmetry to exploit. This abundance of reciprocal edges often result in scattered adjacency lists that are less suited for gap encoding and reference compression which is the point of the WebGraph compression method.

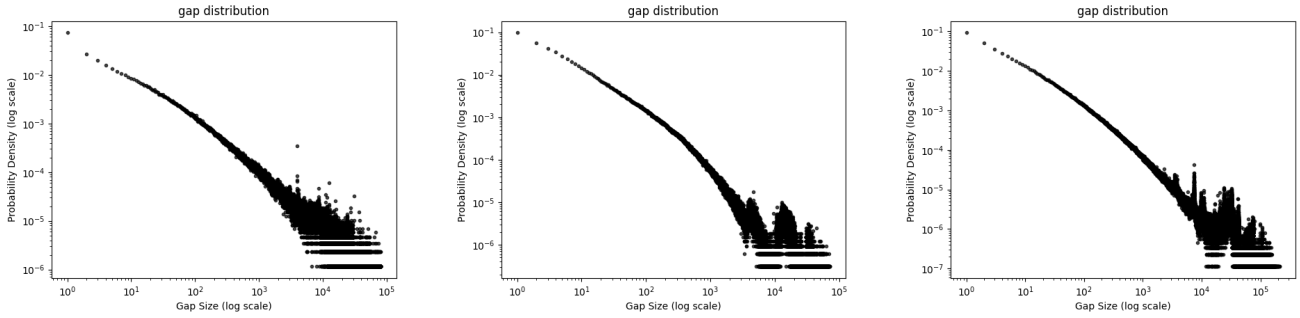
7.2 Future Work

Based on what we found, there are several interesting questions which can be explored in future work:

(1) **Revisiting graph partitioning:** While community detection did nod yield improvements in compression, it may be interesting to explore different ways of partitioning graphs tailored to the compression method. For example, minimizing edge cuts across partitions.
(2) **Adjusting the WebGraph framework:** The current WebGraph framework is optimized for asymmetric relationships. Developing new encoding schemes which are designed for reciprocal edges could lead to significant improvements in compressing social graphs.

REFERENCES

[1] Maven Apache. [n. d.]. Maven. https://maven.apache.org/.
[2] Maciej Besta and Torsten Hoefler. 2019. Survey and Taxonomy of Lossless Graph Compression and Space-Efficient Graph Representations. arXiv:1806.01799 [cs.DS] https://arxiv.org/abs/1806.01799
[3] P. Boldi and S. Vigna. 2004. WebGraph Framework. https://webgraph.di.unimi.it. Accessed: 24-11-2024.
[4] P. Boldi and S. Vigna. 2004. WebGraph Framework. https://webgraph.di.unimi.it/docs/it/unimi/dsi/webgraph/ASCIIGraph.html.
[5] P. Boldi and S. Vigna. 2004. The webgraph framework I: compression techniques. In *Proceedings of the 13th International Conference on World Wide Web* (New York, NY, USA) *(WWW '04)*. Association for Computing Machinery, New York, NY, USA, 595–602. https://doi.org/10.1145/988672.988752
[6] P. Boldi and S. Vigna. 2004. The Webgraph framework II: codes for the World-Wide Web. In *Proceedings of the Conference on Data Compression (DCC '04)*. IEEE Computer Society, USA, 528.
[7] Nieves R. Brisaboa, Susana Ladra, and Gonzalo Navarro. 2009. k2-Trees for Compact Web Graph Representation. In *String Processing and Information Retrieval*, Jussi Karlgren, Jorma Tarhio, and Heikki Hyyrö (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 18–30.
[8] T. Chen, M. A. Kaafar, A. Friedman, and R. Boreli. 2012. Is more always merrier?: a deep dive into online social footprints. In *2012 ACM workshop on Workshop on online social networks*. 67–72. https://doi.org/10.1145/2342549.2342565 Accessed via Netzschleuder.
[9] Tommaso Fontana, Sebastiano Vigna, and Stefano Zacchiroli. 2024. WebGraph: The Next Generation (Is in Rust). In *Companion Proceedings of the ACM Web Conference 2024* (Singapore, Singapore) *(WWW '24)*. Association for Computing Machinery, New York, NY, USA, 686–689. https://doi.org/10.1145/3589335.3651581
[10] Google. [n. d.]. Google Colab. https://colab.research.google.com/.



(a) Slash **(b) Prosper** **(c) Flickr**

Figure 2: Gaps distributions of the best configurations

[11] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. 2008. Exploring Network Structure, Dynamics, and Function using NetworkX. In *Proceedings of the 7th Python in Science Conference*, Gaël Varoquaux, Travis Vaught, and Jarrod Millman (Eds.). Pasadena, CA USA, 11 – 15.

[12] J. Kunegis. 2016. Prosper loans. https://doi.org/10.1145/2487788.2488173 Accessed via Netzschleuder.

[13] H. Kwak, C. Lee, H. Park, and S. Moon. 2010. What is Twitter, a social network or a news media?. In *In WWW '10: Proceedings of the 19th international conference on World wide web*. ACM, 591–600. https://doi.org/10.1145/1772690.1772751 Accessed via Netzschleuder.

[14] J. Leskovec and A. Krevl. 2009. SNAP Datasets: Stanford Slashdot Social network. https://snap.stanford.edu/data/soc-Slashdot0902.html.

[15] D. Perito, C. Castelluccia, M. A. Kaafar, and P. Manils. 2011. How unique and traceable are usernames?. In *In Privacy Enhancing Technologies, pages 1–17. Springer.* https://doi.org/10.1007/978-3-642-22263-4_1 Accessed via Netzschleuder.

[16] Luca Versari, Iulia M. Comsa, Alessio Conte, and Roberto Grossi. 2020. Zuckerli: A New Compressed Representation for Graphs. arXiv:2009.01353 [cs.DS] https://arxiv.org/abs/2009.01353

[17] Y. Zhang and J. Tang. 2011. Social Network Integration: Towards Constructing the Social Graph. *arXiv* 1311.2670 (2011). https://arxiv.org/abs/1311.2670 Accessed via Netzschleuder.