WebGraph: The Next Generation (Is in Rust)

Tommaso Fontana
tommaso.fontana.96@gmail.com
Inria
DGDI
Paris, France

Sebastiano Vigna
sebastiano.vigna@unimi.it
Università degli Studi di Milano
Dipartimento di Informatica
Milan, Italy

Stefano Zacchiroli
stefano.zacchiroli@telecom-paris.fr
LTCI, Télécom Paris
Institut Polytechnique de Paris
Palaiseau, France

ABSTRACT

We report the results of a yearlong effort to port the WebGraph framework [4] from Java to Rust. For two decades WebGraph has been instrumental in the analysis and distribution of large graphs for the research community of TheWebConf, but the intrinsic limitations of the Java Virtual Machine had become a bottleneck for very large use cases, such as the Software Heritage Merkle graph [2] with its half a trillion arcs. As part of this clean-slate implementation of WebGraph in Rust, we developed a few ancillary projects bringing to the Rust ecosystem some missing features of independent interest, such as easy, consistent and zero-cost memory mapping of data structures. WebGraph in Rust offers impressive performance improvements over the previous implementation, enabling open-source graph analytics on very large datasets on top of a modern system programming language.

CCS CONCEPTS

• **Theory of computation → Data compression**; • **Information systems → Network data models**.

KEYWORDS

graphs, big data, compression, web graphs, social networks, Java, Rust

ACM Reference Format:
Tommaso Fontana, Sebastiano Vigna, and Stefano Zacchiroli. 2024. WebGraph: The Next Generation (Is in Rust). In *Companion Proceedings of the ACM Web Conference 2024 (WWW '24 Companion), May 13–17, 2024, Singapore, Singapore.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3589335.3651581

1 INTRODUCTION

Very large graphs, such as Web snapshots, large social networks, or software dependency graphs, can be analyzed using two approaches: distributing the computation on multiple computational units or compressing the graph to fit in the memory of a single one. A popular instance of the latter approach is the WebGraph framework [4], a compression framework with an open-source Java implementation that has been instrumental in the last two decades in the distribution and analysis of large graphs for research purposes. Hundreds of papers, many of which at TheWebConf, have been written using data and tools provided by WebGraph. Moreover, open data projects such as Common Crawl and Software Heritage (SWH) [5] have used WebGraph to compress and distribute their data.

WebGraph has been designed initially with web graphs in mind, but its approach has been shown to be effective in general: when Facebook's 3.74 degrees of separation where computed for the first time [1], the computation was done using WebGraph, which made it possible to represent the whole Facebook graph in 211 GB. Currently, SWH is able to represent their Merkle graph of source code artifacts [11], constituted of 34 billion nodes and 517 billion arcs, in main memory, allowing to satisfy software engineering use cases previously unattainable at this scale such as code clone and fork detection, software provenance tracking, and detection of unfixed security vulnerabilities. The largest, trillion-scale protein-protein similarity graphs ever released [9] were also created, indexed and distributed using WebGraph, showing, once again, the wide applicability of the framework to different fields.

In spite of its popularity, the Java language was getting in the way of practical usability of WebGraph on big graphs: the limitations on the array size (2^{31} elements), the unpredictable impact of garbage collection, and the difficulty of exploiting modern operating-system features like memory mapping were becoming bottlenecks.

Thus, inspired by the success of WebGraph in the SWH project, we decided to rethink and reimplement WebGraph in Rust, a modern, safe, high-performance language that is gaining traction in the systems programming community, and which provides the balance between safety and performance that is needed to move compression-based graph analytics to the present big graph era. The result is more than 100 000 committed lines of open-source code (50 000 released) distributed among a few Rust projects that aim at filling a few gaps in the Rust ecosystem.

Data availability. All the software discussed in this paper is open-source, released as multiple Rust *crates* (`webgraph`, `sux`, `epserde`, `dsi-bitstream`, `mem_dbg`) via crates.io and as source code on GitHub starting from https://github.com/vigna/webgraph-rs. The reproducibility package used for experimental evaluation is archived on Zenodo with DOI 10.5281/zenodo.10793566 and on SWH with SWHID swh:1:dir:4c44e237561e228a4c73907e34e866a1fe2acace.

2 JAVA IMPLEMENTATION LIMITATIONS

In this section, we discuss the Java limitations encountered with the previous implementation of WebGraph and, more generally, with large-graph processing.

Unfriendly memory model. The Java Virtual Machine (JVM) has a memory model based on relatively small, fast-cycling garbage-collected objects. This model is at odds with the approach of using

WWW '24 Companion, May 13–17, 2024, Singapore, Singapore

ACM ISBN 979-8-4007-0172-6/24/05
https://doi.org/10.1145/3589335.3651581

a large part of the memory for an immutable representation of data, which is a common need for in-memory graph analytics. While some improvements could be obtained via recent (still in preview at the time of writing) out-of-heap allocation features, such features introduce additional complexity in accessing the data.

Difficult representation of node-related data. Many algorithms require to store some form of complex data per node. For large graphs this could amount to submit to the garbage collector billions of objects. Future Java extensions might allow for value objects, which would mitigate the problem, but presently the only solution is to use cumbersome parallel arrays of primitive types instead.

32-bit collections. Arrays in Java are indexed by 32-bit integers, limiting their size to 2^{31} elements. The limit is carried on by data structure implementations belonging to the `java.util` package. It is possible to design data structures for larger collections by multiplexing storage on multiple arrays, as the external `fastutil` library does, but that comes with performance drawbacks.

Convoluted access to memory-mapping facilities. Use cases that need large static data structures, such as search-engine indices or friendship graph representations, rely on memory mapping to reduce loading times. Memory mapping assigns to the content of a file a virtual address; the file is then loaded on-demand and transparently by the operating system as part of memory access. Java memory-mapping facilities are very limited both in the allowed maximum size of a mapping and in the access mechanism. It is also impossible to refer to memory-mapped memory transparently, resulting in complex conditional code if one wants to support regular and memory transparently. Memory mapping is a requirement for very large graphs, because often interesting analyses can be done by only accessing a small fraction of the graph (e.g., software provenance tracking, or the computation of similar proteins). Memory mapping makes startup costs negligible and reduces I/O costs to the strict minimum of graph regions actually accessed.

Difficult access to native code. The mechanism to access native code from the Java Virtual Machine is the Java Native Interface (JNI), which is a complex and error-prone mechanism. Again, future expansions aim at improving this situation, but they are not available for production code yet.

3 BACKGROUND: RUST

Rust [10] is a modern, safe, high-performance language that is gaining traction in both industry and the open-source community. While Rust builds upon many important ideas from advanced functional languages such as Haskell and OCaml, its greatest innovation is its ownership model, which makes it possible to avoid at the same time explicit memory allocation/deallocation *and* garbage collection. This is achieved by making all assignment *moves* (as in C++'s move semantics) and by having a *borrow checker* proving that all references are valid. The borrow checker runs at compile time and proves, essentially, that if each variable would be wrapped in a read/write lock, the program would run smoothly anyway. The compiler then knows exactly when a value will not be used any longer, that is, when the only variable holding its value goes out of scope, and deallocates it automatically.

Another important characteristic is the compilation model. The Rust build system and package manager (`cargo`) builds binaries starting from a complete code base that contains the source code of all dependencies. There are no libraries as in traditional languages: binaries are built and linked starting from source code. This makes it possible to perform inlining and optimization well beyond the possibilities of system languages like C and C++, in which separate compilations partitions precompiled binary code (e.g., from 3rd party libraries) and source code under compilation (this separation is somehow lessened by templates and by recent link-time optimization features of C/C++ compilers, but not at the level of Rust). Moreover, Rust binaries can always be fully optimized for the current architecture, as even the code from dependencies is compiled locally.

Recently, Rust has become the first high-level language after C to be allowed into the Linux kernel. Moreover, several high-profile organizations have started to rewrite their core infrastructure in Rust: in fact, Rust was originally developed to rewrite the Firefox browser.

Rust has a natural, efficient approach to all the problems we mentioned in the previous section and it is thus a natural choice for a clean-slate reimplementation of WebGraph.

4 THE NEW RUST IMPLEMENTATION

Porting the WebGraph framework to Rust required porting a number of ancillary projects for which no Rust equivalent implementation was available, and to design an entirely novel approach to memory mapping of large immutable data structures that keep the memory safety of Rust without any performance degradation. Taken together the developed projects correspond to more than 100 000 committed lines of open-source Rust code, of which more than 50 000 are currently released.

One of the main goals of the rewrite was to use the Rust type system to avoid dynamic dispatch, which is a major source of overhead in the Java implementation, and which prevents inlining and aggressive optimization. Quite to our surprise, we managed to rewrite the entire code base without a single use of dynamic dispatch. The Rust sophisticated type system made it possible to write code that, once compiled, is entirely monomorphic, and thus can be fully inlined and optimized by the compiler.

We briefly describe below the Rust software components (and crates) that form the new WebGraph framework.

Bit streams (crate `dsi-bitstream`). The Java implementation of WebGraph uses custom classes to read and write bit streams. We have reimplemented these classes using state-of-the-art techniques and in particular accessing the underlying data at the granularity of memory words, rather than bytes, exploiting the endianness of the architecture (the previous Java implementation is big-endian only). The user has the possibility of fine-tuning the library to any architecture by reading or writing words of different sizes, and by choosing whether to use or not *decoding tables* to speed up the decoding of instantaneous codes representing small values.

Succinct data structures (`sux`). To store pointers to the compressed bitstream, WebGraph uses a Java implementation of the Elias-Fano representation of monotone sequences [7, 8] from the

Sux4J project. We used our experience with the C++ Sux project to reimplement the Elias-Fano representation in Rust, and to provide a number of variants that entirely composable. In particular, succinct indices (for rank, selection, etc.) can be selected via *functor-oriented programming*, where a functor transforms a data structure in a variant with additional properties (e.g., an index for ranking). This approach allows to freely combine different techniques for ranking and selection, with Rust monomorphization guaranteeing a zero-cost abstraction.

Binary serialization and memory mapping (`epserde`). The Rust ecosystem provides a few well-known serialization frameworks, but none of them supports memory mapping as a zero-cost abstraction: there is always a price to pay. We devised *ε-copy* as a new approach to the problem and implemented it in the novel ε-serde (de)serialization framework for Rust. Typically, *zero-copy* serialization frameworks make it possible to load serialized data structures in memory and use them directly, without further copying. This makes it possible, as an alternative, to map data structures into memory, without loading them explicitly. However, all current zero-copy serialization frameworks for Rust incur access penalties to maintain Rust strong safety guarantees, without modifying serialized data.

ε-copy is our solution for this problem: it is a strategy in which a small, negligible part of the data structure is actually copied, while the rest is linked to it by reference, making it possible to retain the same performance of the original data structure even when memory mapping it. The Rust type system makes ε-copy possible by letting users write methods that use the same code both for vectors (mutable sequences), which support structures at creation time, and immutable arrays (provided that no mutation methods are invoked), which support structures that are ε-copy deserialized.

Fast structure measurement and inspection (`mem_dbg`). Rust has facilities to compute the *stack size* of a value, that is, the space occupied by the main allocation of a value. However, if the value contains references to other pieces of data, the space occupied by the referenced data is not accounted for. We have developed a new crate, `mem_dbg`, that was instrumental in the development of succinct data structures and serialization. While there are a few crates that provide solutions for these problems by recursing into complex data types, they all ultimately have to rely on iteration on vectors and arrays. In our case (structures with dozens of billions of elements), measuring the size of a structure took several seconds. We leveraged the idea of zero-copy structures from ε-serde to avoid iteration, resulting in a library that can measure exactly the size of a structure in a few hundred nanoseconds.

WebGraph (`webgraph`). The first and foremost difference between the Java and Rust implementation is that, by using our new bitstream implementation, we can read data a word at a time (rather than a byte at a time, as in the Java implementation) and exploit the endianness of the architecture. We provide an implementation using function pointers for decoding, which can be used with any graph, and a monomorphic implementation with static dispatch that can be tailored to a specific choice of parameters, which provides further possibilities for the compiler to perform optimizations.

The second important step we took was decoupling the specific method of storage for the compression of the BV format [4] from the algorithm implementing the format. Traditionally the components (outdegrees, successor gaps, etc.) of the BV format are stored using instantaneous codes. We have generalized the format to use any kind of encoder, and we briefly report some preliminary results in Section 6.

The largest redesign w.r.t. the previous implementation is that of the labeling system. Labels attached to edges were an afterthought in the Java version, and despite their usage in several large-scale datasets [3, 9, 11], they were cumbersome to use and slow. The Rust implementation attacks the problem using a new, abstract compositional approach: labelings are simply structures associated with a graph that can enumerate labels associated with the successors of a node. In fact, a graph is simply an *integer* labeling, where integers are node identifiers. Labelings can be "zipped" together, with labels of the resulting labeling being pairs. Finally, a labeled graph is a labeling whose labels are pairs whose first coordinate is an integer. Thus, one can easily attach labels to a graph at any time by zipping labels to the arcs of a graph, or even combine labels transparently. Left and right projection operators make it possible to extract the graph underlying a labeling, or the labeling itself. All graph operations such as transposition are now implemented directly on labeled graphs; any unlabeled graph can be labeled in a trivial way using Rust unit type `()`, which has zero space occupancy. The resulting code is indistinguishable from code written on purpose for unlabeled graphs, as the compiler can deduce that the labels are irrelevant.

5 EXPERIMENTAL EVALUATION

In this section, we report the results of the first experimental benchmarks comparing the execution speed of the previous Java (GraalVM 21) implementation of WebGraph and the new Rust (1.75) one. Three very different graphs—whose characteristics are shown in Table 1—were used for this experimental evaluation:

`enwiki-2023` a 2023 snapshot of the English Wikipedia graph;
`eu-2015` a 2015 snapshot of the web graph of the `.eu` ccTLD;
`swh-2023` a 2023 snapshot of the Merkle graph of the Software Heritage archive.

The first two graphs are available from the Laboratory for Web Algorithmics[1], the third one from Software Heritage [11][2].

We consider two basic measures of access speed: random-access enumeration of successors and breadth-first visits. In the first case, we extract 10 million random nodes and we enumerate their successors; in the second case, we perform a breadth-first visit of the whole graph. Testing enumeration of successors is a direct measure of how much the new implementation relates to the old one, but it is also important to test more realistic scenarios, because principle the gains in speed might be leveled by overheads in the implementation of more complex algorithms. Results are shown in Table 2. We performed three runs of each experiment, with an additional one for the Java implementation to warm up the JVM and let the JIT perform runtime compilation and inlining. We report the best result out of three; the timing variance between runs was very low.

[1] http://law.di.unimi.it/datasets.php
[2] https://docs.softwareheritage.org/devel/swh-dataset/graph/dataset.html

Graph	Nodes	Arcs	Degree (avg.)	Bits/arc	Size (comp.)
enwiki-2023	4.2 M	101 M	24.93	13.55	267 MB
eu-2015	1 B	92 B	85.74	1.19	13 GB
swh-2023	34 B	491 B	14.38	3.07	176 GB

Table 1: Graphs used as experimental datasets for benchmarking purposes (see results in Table 2) and their characteristics: nodes, arcs, average degree, compressed bits/arc ratio, and compressed size.

Graph	WebGraph impl. Java	WebGraph impl. Rust	Java→Rust speedup
Random access (ns/arc)			
enwiki-2023	61	31	×1.97
eu-2015	24	17	×1.41
swh-2023	104	47	×2.21
BFS visit (ns/node)			
enwiki-2023	1450	734	×1.98
eu-2015	1580	971	×1.63
swh-2023	1140	359	×3.18

Table 2: Comparison of the speed of the Java and Rust implementation of WebGraph. The first two graphs were tested on an Intel® Core™ i7-12700KF CPU @3.60 GHz, the third on an Intel® Xeon® Gold 6342 CPU @2.80 GHz.

Random-access speed has increased with the Rust implementation from 50% to more than 200% depending on the graph. The cost of accessing highly compressible graphs, such as eu-2015 (1.2 bits/arc), depends mostly on the compression algorithm of the BV format; this part, mainly relying on algorithmic steps involving arrays, works reasonably in Java too. If, however, the graph is less compressible (e.g., enwiki-2023 with 13.5 bits/arc and swh-2023 with 3.07) reading quickly from the compressed bitstream becomes more important and Rust optimizations become more effective. Maybe surprisingly, the same performances hold even when considering a long-running application such as a breadth-first visit, which is 300% faster on swh-2023. This is partially due to the Rust compilation model, as aggressive inlining and optimization make the cost of calls to basic access functions negligible, and make it possible to integrate it directly with the application code.

6 FUTURE WORK

We already started to use the Rust implementation to test the feasibility of replacing instantaneous codes using a recently proposed entropy coder, *asymmetric numeral systems* [6], which is gaining traction in the compression community for its speed and compression performance. We can report that ANS appears to provide a further 10–15% percent of additional compression with respect to instantaneous codes, providing a new record in the compressibility of web and social graphs: for example, the high-compression version of the eu-2015 dataset, which uses 0.9 bits per arc using instantaneous codes, needs just 0.8 bits per arc using ANS. Decoding speed is about twice as slower than decoding instantaneous codes, but we are currently working on optimizing the decoding algorithm.

7 CONCLUSIONS

We have presented a new implementation of the WebGraph framework in Rust. The new implementation is much faster than the previous Java implementation, and provides several new features, such as transparent memory mapping, and a much-necessary redesign of the labeling system. While there is ongoing development to bring the Rust version to feature parity with the previous one, the new version is already usable for many applications, and it is being deployed in production at Software Heritage on a half-a-trillion arc graph. Analysis of web and social network graphs, the original goal of WebGraph in 2004, is now possible at much greater speed and in a more modern programming language.

While developing the new implementation, we have also developed a number of ancillary projects that are of independent interest, including a generic framework for memory-mapped (de)serialization which is almost zero-copy.

ACKNOWLEDGMENTS

This work is supported by project SERICS (PE00000014) under the NRRP MUR program funded by the EU - NGEU, and by project COREGRAPHIE (ANR-20-CE23-0002) of the French Agence Nationale de la Recherche (ANR). The authors would like to thank Valentin Lorentz, for his invaluable feedback and testing during the inception of WebGraph in Rust, and Lorenzo Cimini, for his contributions to the design and implementation of ANS-based compression.

REFERENCES

[1] Lars Backstrom, Paolo Boldi, Marco Rosa, Johan Ugander, and Sebastiano Vigna. 2012. Four Degrees of Separation. In *ACM Web Science 2012: Conference Proceedings*. ACM Press, 45–54. Best paper award.
[2] Paolo Boldi, Antoine Pietri, Sebastiano Vigna, and Stefano Zacchiroli. 2020. Ultra-large-scale Repository Analysis via Graph Compression. In *SANER 2020*. IEEE.
[3] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. 2008. A Large Time-Aware Graph. *SIGIR Forum* 42, 2 (2008), 33–38.
[4] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. ACM Press, Manhattan, USA, 595–601.
[5] Roberto Di Cosmo and Stefano Zacchiroli. 2017. Software Heritage: Why and How to Preserve Software Source Code. In *Proceedings of the 14th International Conference on Digital Preservation, iPRES 2017*. https://hal.archives-ouvertes.fr/hal-01590958/
[6] Jarek Duda. 2009. Asymmetric numeral systems. *CoRR* abs/0902.0271 (2009). arXiv:0902.0271 http://arxiv.org/abs/0902.0271
[7] Peter Elias. 1974. Efficient Storage and Retrieval by Content and Address of Static Files. *J. Assoc. Comput. Mach.* 21, 2 (1974), 246–260.
[8] Robert M. Fano. 1971. On the number of bits required to implement an associative memory. (1971). Memorandum 61, Computer Structures Group, Project MAC, MIT, Cambridge, Mass., n.d..
[9] Mohsen Koohi Esfahani, Paolo Boldi, Hans Vandierendonck, Peter Kilpatrick, and Sebastiano Vigna. 2023. On Overcoming HPC Challenges of Trillion-Scale Real-World Graph Datasets. In *IEEE Big Data 2023*. IEEE Comput. Soc. Press, 215–220. https://doi.org/10.1109/BigData59044.2023.10386309
[10] Nicholas D. Matsakis and Felix S. Klock. 2014. The Rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology (HILT '14)*. ACM, 103–104. https://doi.org/10.1145/2663171.2663188
[11] Antoine Pietri, Diomidis Spinellis, and Stefano Zacchiroli. 2019. The Software Heritage graph dataset: public software development under one roof. In *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019*. IEEE / ACM, 138–142. https://doi.org/10.1109/MSR.2019.00030