The Journal of Open Source Software

rustworkx: A High-Performance Graph Library for Python

Matthew Treinish [1], **Ivan Carvalho** [2], **Georgios Tsilimigkounakis** [3], and **Nahum Sá** [4]

1 IBM Quantum, IBM T.J. Watson Research Center, Yorktown Heights, USA
2 University of British Columbia, Kelowna, Canada
3 National Technical University of Athens, Athens, Greece
4 Centro Brasileiro de Pesquisas Físicas, Rio de Janeiro, Brazil

DOI: 10.21105/joss.03968

Software

- Review
- Repository
- Archive

Editor: Vincent Knight

Reviewers:

- @szhorvat
- @inakleinbottle

Submitted: 28 October 2021
Published: 01 November 2022

In *rustworkx*, we provide a high-performance, flexible graph library for Python. *rustworkx* is inspired by *NetworkX* but addresses many performance concerns of the latter. *rustworkx* is written in Rust and is particularly suited for performance-sensitive applications that use graph representations.

Statement of need

rustworkx is a general-purpose graph theory library focused on performance. It wraps low-level Rust code (Matsakis & Klock, 2014) with a flexible Python API, providing fast implementations for graph data structures and popular graph algorithms.

rustworkx is inspired by the *NetworkX* library (Hagberg et al., 2008), but meets the needs of users that also need performance. Even though *NetworkX* is the de-facto standard graph and network analysis library for Python, it has performance concerns. *NetworkX* prefers pure Python implementations, which leads to bottlenecks in computationally intensive applications that use graph algorithms.

rustworkx addresses those performance concerns by switching to a Rust implementation. It has support for shortest paths, isomorphism, matching, multithreading via rayon (Stone & Matsakis, 2021), and much more.

Related work

The graph and network analysis ecosystem for Python is rich, with many libraries available. *igraph* (Csardi & Nepusz, 2006), *graph-tool* (Peixoto, 2014), *SNAP* (Leskovec & Sosič, 2016), and *Networkit* (Staudt et al., 2016) are Python libraries written in C or C++ that can replace *NetworkX* with better performance. We also highlight *SageMath*'s graph theory module (The Sage Developers, 2020), which has a stronger focus in mathematics than *NetworkX*.

However, the authors found that no library matched the flexibility that *NetworkX* provided for interacting with graphs. *igraph* is efficient for static large graphs, but does not handle graph updates as efficiently. *SNAP* and *Networkit* do not support associated edge data with arbitrary Python types. *graph-tool* supports associated edge data at the cost of maintaing the data in a separate data structure. Thus, the main contribution of *rustworkx* is keeping the ease of use of *NetworkX* without sacrificing performance.

We note that existing code using *NetworkX* needs to be modified to use *rustworkx*. *rustworkx* is not a drop-in replacement for *NetworkX*, which may be a possible limitation for some users.

The authors provide a *NetworkX* to *rustworkx* conversion guide in the documentation to aid in those situations.

Graph data structures

rustworkx provides two core data structures: `PyGraph` and `PyDiGraph`. They correspond to undirected and directed graphs, respectively. Graphs describe a set of nodes and the edges connecting pairs of those nodes. Internally, *rustworkx* leverages the *petgraph* library (bluss et al., 2021) to store the graphs using an adjacency list model and the *PyO3* library (Hewitt et al., 2021) for the Python bindings.

Nodes and edges of the graph may also be associated with data payloads. Payloads can contain arbitrary data, such as node labels or edge lengths. Common uses of data payloads include representing weighted graphs. Any Python object can be a data payload, which makes the library flexible because no assumptions are made about the payload types.

rustworkx operates on payloads with callbacks. Callbacks are functions that take payloads and return statically typed data. They resemble the named attributes in *NetworkX*. Callbacks are beneficial because they bridge the arbitrary stored data with the static types *rustworkx* expects.

A defining characteristic of *rustworkx* graphs is that each node maps to a non-negative integer node index, and similarly, each edge maps to an edge index. Those indices uniquely determine nodes and edges in the graph. Indices are stable, hence the index for a node v does not change even if another node u is removed. Moreover, indices separate the data representing the graph's structure, which is stored in Rust, from the payloads associated with nodes and edges, which are stored in Python.

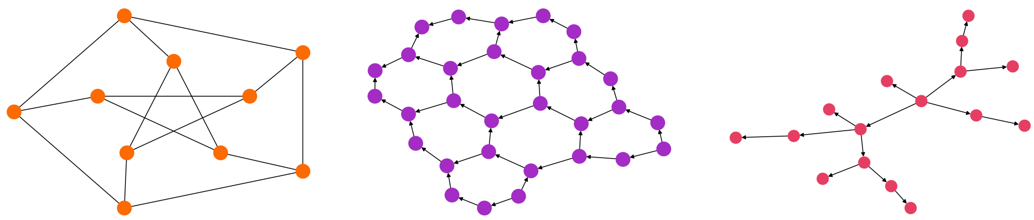

Figure 1: A Petersen graph, a hexagonal lattice graph, and a binomial tree graph created with **`rustworkx.generators`** and visualized with the **`rustworkx.visualization`** module.

Use Cases

rustworkx is suitable for modeling graphs ranging from a few nodes scaling up to 4 billion. The library is particularly suited for applications that have core routines executing graph algorithms. In those applications, the performance of *rustworkx* considerably reduces computation time. Examples of applications using *rustworkx* include the Qiskit compiler (Treinish et al., 2021), PennyLane (Bergholm et al., 2020), atompack (Ullberg, 2021), and qtcodes (Jha et al., 2021).

For common use cases, *rustworkx* can provide speedups ranging from 3x to 100x compared to the same code using *NetworkX* while staying competitive with other compiled libraries like *igraph* and *graph-tool*. The gains in performance are application-specific, but as a general rule, the more work that is offloaded to *rustworkx* and Rust, the larger are the gains.

We illustrate use cases with examples from the field of quantum computing that motivated the development of the library.

Graph Creation, Manipulation, and Traversal

The first use case is based on the manipulation of directed acyclic graphs (DAGs) by Qiskit using *rustworkx*. Qiskit represents quantum circuits as DAGs on which the compiler operates to perform analysis and transformations (Childs et al., 2019).

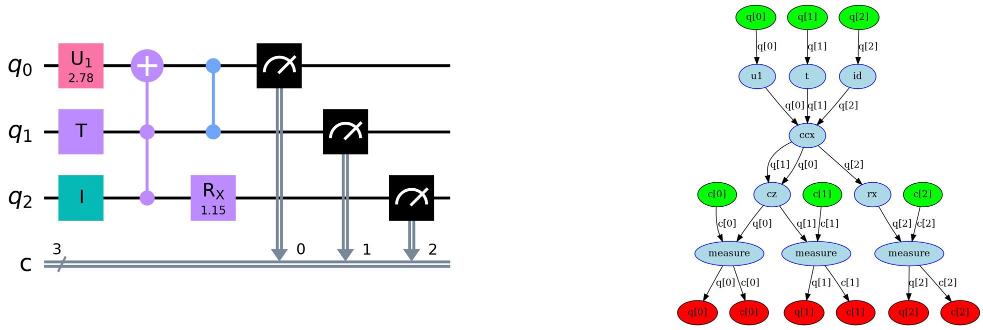

Figure 2: Quantum circuit and its equivalent representation as a DAG of instructions built by Qiskit.

Qiskit creates a DAG with nodes that represent either instructions or registers present in the quantum circuit (Cross et al., 2021) and with edges that represent the registers each instruction operates on. Qiskit also applies transformations to the instructions, which manipulate the graph by adding and removing nodes and edges. *rustworkx* brings the graph data structure underlying those operations.

In addition, Qiskit needs to traverse the graph. Some transformations, such as greedily merging instructions to reduce circuit depth, require graph traversal. *rustworkx* offers the methods for traversals such as breadth-first search, depth-first search, and topological sorting.

Subgraph Isomorphism

The second use case is based on the qubit mapping problem for Noisy Intermediate-Scale Quantum (NISQ) devices (Bharti et al., 2022; Preskill, 2018). NISQ devices do not have full connectivity among qubits, hence Qiskit needs to take into account an undirected graph representing the connectivity of the device when compiling quantum circuits. Qiskit transforms the quantum circuit such that the pairs of qubits executing two-qubit gates respect the device's architectural constraints. There are many proposed solutions to the qubit mapping problem, including algorithms based on subgraph isomorphism (Li et al., 2021).

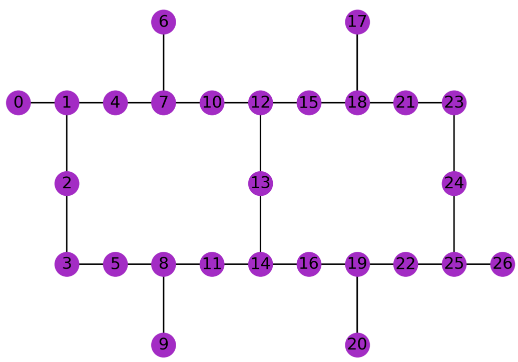

Figure 3: Graph representing the connectivity of the **`ibmq_montreal`** device. Qiskit can check if the required connectivity by a circuit is subgraph isomorphic to the device's connectivity when solving the qubit mapping problem.

rustworkx implements the VF2 algorithm (Cordella et al., 2004) and some of the VF2++ heuristics (Jüttner & Madarasi, 2018) to solve subgraph isomorphism. The implementations include both checking if a mapping exists and returning a mapping among the nodes. Qiskit leverages *rustworkx* to provide an experimental layout method based on VF2. Qiskit checks if the graph representing the connectvity required by the circuit and the connectivity provided by the device are subgraph isomorphic. If they are, Qiskit uses VF2 mapping to map the qubits without increasing the depth of the circuit.

Acknowledgements

We thank Kevin Krsulich for his help in getting *rustworkx* ready for use by Qiskit, Lauren Capelluto and Toshinari Itoko for their continued support and help with code review, and all of the *rustworkx* contributors who have helped the library improve over time.

References

Bergholm, V., Izaac, J., Schuld, M., Gogolin, C., Alam, M. S., Ahmed, S., Arrazola, J. M., Blank, C., Delgado, A., Jahangiri, S., McKiernan, K., Meyer, J. J., Niu, Z., Száva, A., & Killoran, N. (2020). *PennyLane: Automatic differentiation of hybrid quantum-classical computations*. https://doi.org/10.48550/ARXIV.1811.04968

Bharti, K., Cervera-Lierta, A., Kyaw, T. H., Haug, T., Alperin-Lea, S., Anand, A., Degroote, M., Heimonen, H., Kottmann, J. S., Menke, T., Mok, W.-K., Sim, S., Kwek, L.-C., & Aspuru-Guzik, A. (2022). Noisy intermediate-scale quantum algorithms. *Reviews of Modern Physics*, *94*(1). https://doi.org/10.1103/revmodphys.94.015004

bluss, Comets, J.-M., Borgna, A., Larralde, M., Mitchener, B., & Kochkov, A. (2021). Petgraph. In *GitHub repository*. GitHub. https://github.com/petgraph/petgraph

Childs, A. M., Schoute, E., & Unsal, C. M. (2019). Circuit Transformations for Quantum Architectures. In W. van Dam & L. Mancinska (Eds.), *14th Conference on the Theory of Quantum Computation, Communication and Cryptography (TQC 2019)* (Vol. 135, pp. 3:1–3:24). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. https://doi.org/10.4230/LIPIcs.TQC.2019.3

Cordella, L. P., Foggia, P., Sansone, C., & Vento, M. (2004). A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *26*(10), 1367–1372. https://doi.org/10.1109/TPAMI.2004.75

Cross, A. W., Javadi-Abhari, A., Alexander, T., Beaudrap, N. de, Bishop, L. S., Heidel, S., Ryan, C. A., Smolin, J., Gambetta, J. M., & Johnson, B. R. (2021). *OpenQASM 3: A broader and deeper quantum assembly language*. https://doi.org/10.48550/arXiv.2104.14722

Csardi, G., & Nepusz, T. (2006). The igraph software package for complex network research. *InterJournal*, *Complex Systems*, 1695. https://igraph.org

Hagberg, A. A., Schult, D. A., & Swart, P. J. (2008). Exploring network structure, dynamics, and function using NetworkX. In G. Varoquaux, T. Vaught, & J. Millman (Eds.), *Proceedings of the 7th Python in Science Conference* (pp. 11–15). http://conference.scipy.org/proceedings/SciPy2008/paper_2/

Hewitt, D., Kanagawa, Y., Kim, N., Grunwald, D., Niederbühl, A., messense, Kolenbrander, B., Brandl, G., & Ganssle, P. (2021). PyO3. In *GitHub repository*. GitHub. https://github.com/PyO3/pyo3

Jha, S., Chen, J., Householder, A., & Mi, A. (2021). Qtcodes. In *GitHub repository*. GitHub. https://github.com/yaleqc/qtcodes

Jüttner, A., & Madarasi, P. (2018). VF2++—an improved subgraph isomorphism algorithm. *Discrete Applied Mathematics*, *242*, 69–81. https://doi.org/10.1016/j.dam.2018.02.018

Leskovec, J., & Sosič, R. (2016). SNAP: A general-purpose network analysis and graph-mining library. *ACM Trans. Intell. Syst. Technol.*, *8*(1). https://doi.org/10.1145/2898361

Li, S., Zhou, X., & Feng, Y. (2021). Qubit mapping based on subgraph isomorphism and filtered depth-limited search. *IEEE Transactions on Computers*, *70*(11), 1777–1788. https://doi.org/10.1109/tc.2020.3023247

Matsakis, N. D., & Klock, F. S. (2014). The Rust language. *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, 103–104. https://doi.org/10.1145/2663171.2663188

Peixoto, T. P. (2014). The graph-tool Python library. *Figshare*. https://doi.org/10.6084/m9.figshare.1164194

Preskill, J. (2018). Quantum computing in the NISQ era and beyond. *Quantum*, *2*, 79. https://doi.org/10.22331/q-2018-08-06-79

Staudt, C. L., Sazonovs, A., & Meyerhenke, H. (2016). NetworKit: A tool suite for large-scale complex network analysis. *Network Science*, *4*(4), 508–530. https://doi.org/10.1017/nws.2016.20

Stone, J., & Matsakis, N. D. (2021). Rayon: A data parallelism library for Rust. In *GitHub repository*. GitHub. https://github.com/rayon-rs/rayon

The Sage Developers. (2020). *SageMath, the Sage Mathematics Software System (Version 9.1)*. https://doi.org/10.5281/zenodo.4066866

Treinish, M., Gambetta, J., Rodríguez, D. M., Marques, M., Bello, L., Wood, C. J., Gomez, J., Nation, P., Chen, R., Winston, E., Gacon, J., Cross, A., Krsulich, K., Sertage, I. F., Wood, S., Alexander, T., Capelluto, L., Puente González, S. de la, Rubio, J., … Woerner, S. (2021). *Qiskit/qiskit-terra: Qiskit terra 0.19.1* (Version 0.19.1) [Computer software]. Zenodo. https://doi.org/10.5281/zenodo.2583252

Ullberg, S. (2021). atompack: A flexible Python library for atomic structure generation. In *GitHub repository*. GitHub. https://github.com/seatonullberg/atompack