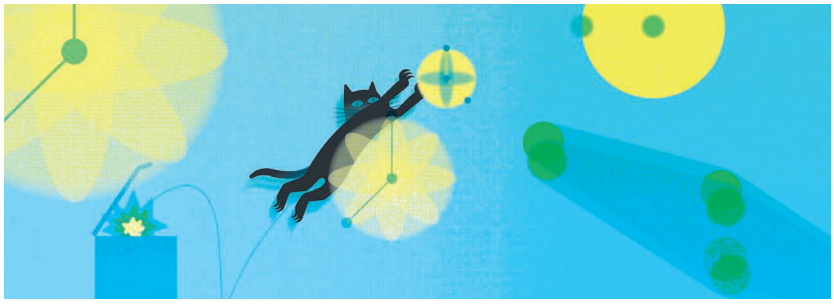
Quantum Optimization Applications With Quark and Quapps

Bridging the Gap Between Application and Hardware

Lukas Windgätter and Elisabeth Lobe, German Aerospace Center (DLR)

// *Solving discrete optimization problems is one of the most promising use cases on quantum computers. The libraries quark and quapps provide an easily usable, modular workflow to encode discrete optimization problems and interface them to both classical and quantum hardware.* //



Digital Object Identifier 10.1109/MS.2025.3564146
Date of publication 1 May 2025; date of current version 13 August 2025.

WITH THE ADVENT of more powerful and error-tolerant quantum computers, their use for commercial applications in industry and administration comes within reach. A promising use case for quantum computers is solving NP-hard discrete optimization problems. These are ubiquitous problems appearing in all industries, ranging from truck routing problems over energy grid distribution to scheduling or seat distribution problems in an airplane. However, finding a solution to these problems can be incredibly challenging for classical computers, and calculations can quickly become infeasible even for small problem instances. Here, quantum computers have the potential to massively boost these calculations. Currently, there are several promising algorithms and hardware platforms dedicated to tackle the problem of finding optimal solutions to optimization problems. Among the most promising algorithms for digital quantum computers are the quantum approximate optimization algorithms (QAOA),[1] Grover's algorithm and variational eigensolvers. Another very successful approach is to use special hardware that is tailored to specifically solve optimization problems using the adiabatic computation paradigm. These machines are called quantum annealers.

However, for a user trying to solve an optimization problem at hand, this jungle of different algorithms and machines with their different interfaces can easily become too much to handle without significant expertise in quantum information theory. Therefore, specific software is needed to bridge the gap between the actual problem and the different hardware and algorithmic platforms. This gap is exactly what our open source Python software library with the complementary packages **quark**[2] and **quapps**[3] aims to fill.

While **quark** provides a software tool to easily implement and reformulate discrete optimization problems and to interface to different hardware platforms and solvers, **quapps** provides a library which implements the most common optimization problems based on **quark**. These can either be loaded and used directly or modified to the users' needs. The basic idea of how to implement an optimization problem using **quark** has already been presented in the work by Lobe.[4] Here, we want to go into more detail on the modular structure, the extension with **quapps** and the interfaces to different solvers. Note that the packages are not intended to solve the optimization problems themselves, but to prepare the calculations by transforming the formulations into the common entry point of most quantum optimization approaches, the quadratic unconstrained binary optimization (QUBO) problem formulation, which we describe in more detail in the next section.

We additionally provide a set of instantiated optimization problems, which is derived from the **quapps** library with the work of Lobe and Windgätter.[5] Its purpose is to simplify the benchmarking of the possible quantum algorithm and quantum hardware combinations.

From the Problem to the Quantum Hardware

The package **quark** is a software framework which aims to provide a user-friendly, yet powerful tool to formulate discrete optimization use cases and interface them to several different classical and quantum solvers. An overview of the functionality is given in Figure 1 and will be explained in more detail in this and the following section.

To understand the basic structure of **quark**, one needs to understand the basic building blocks of a discrete optimization problem and its formulation as a QUBO problem. A discrete optimization problem is defined as

$$\begin{aligned} \min \quad & f(y) \\ \text{s.t.} \quad & \dots \\ & \ell_k \le c_k(y) \le u_k \\ & \dots \\ \text{with} \quad & y_i \in \mathbb{Z}. \end{aligned} \tag{1}$$

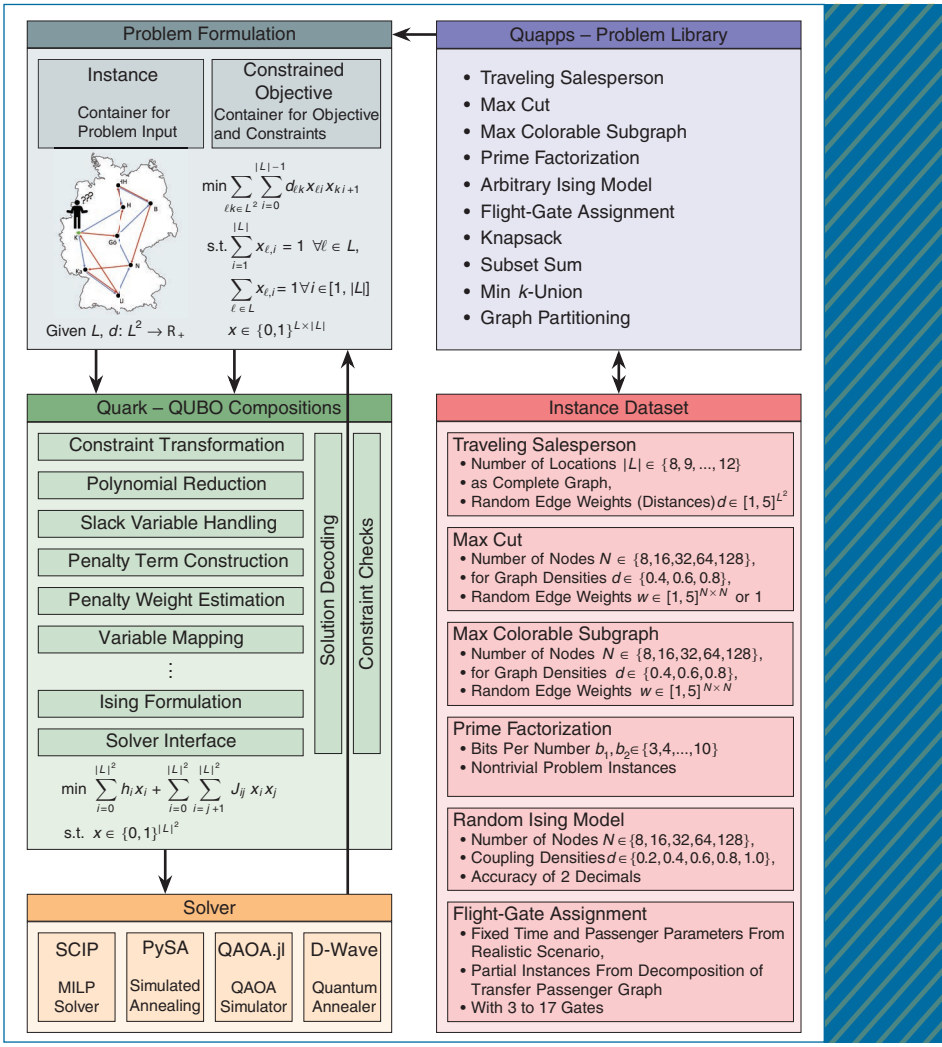


FIGURE 1. Overview of the packages **quark** and **quapps**. The discrete optimization problem is implemented by two input containers separating the parameter and the model description. The left boxes show the exemplary traveling salesperson problem. The QUBO transformation [cf. (1)–(4)] and reversely the solution decoding are performed by the **quark** package. It also provides an interface to multiple solvers. A set of standard problems can be directly loaded using the **quapps** library. The instance data set contains five random instances for each setup of corresponding parameters for each problem type, resulting in a total of 764 problem instances. SCIP: Solving Constraint Integer Programs; PySA: Python Simulated Annealing.

The objective function f as well as the functions in the constraints c_k, for an arbitrary number of constraints k, can have various different formats. However, they have in common to act on a discretely valued vector y of a given length with entries yi. While f describes the goal of the optimization, e.g., minimizing some cost, the constraints incorporate restrictions, e.g., allowing for a city to be visited only once by a salesperson on the route. To be usable for a quantum device, the generic optimization problem has to be reformulated as a QUBO. A QUBO problem has an objective function over binary variables, which is at most quadratic in any x_i and no further constraints are added, i.e.,

$$\begin{aligned} &\min \ \sum_i h_i x_i + \sum_{i,j} J_{i,j} x_i x_j \\ &\text{s.t.} \quad x_i \in \{0,1\}. \end{aligned} \tag{2}$$

This means a specific QUBO problem is only defined by the given values for the h_i and $J_{i,j}$ parameters, which can be arbitrary real numbers. Those are usually represented in a vector $h \in \mathbb{R}^n$ and a matrix $J \in \mathbb{R}^{n \times n}$ with n being the number of binary variables.

Transforming the optimization problem (1) to its QUBO form (2) can be highly nontrivial. The major mathematical steps are to transform all inequality constraints c_k in (1) to equality constraints, for instance in the standard form via slack variables $z_k \in \mathbb{Z}$ with

$$c'_k(y) = c_k(y) - z_k = 0, \tag{3}$$

to encode all integer variables using a binary representation and finally to reduce the degree of the objective function f and the constraints to quadratic and linear degree, respectively. The resulting quadratic objective function $f'(x)$ and linear functions of the equality constraints $c'_k(x)$ over binary variables x can then be used to compose the QUBO objective function by adding the constraint terms using a penalty weight λ_k

$$q(x) = f'(x) + \sum_k \lambda_k (c'_k(x))^2. \tag{4}$$

This can now be expanded into the QUBO form as in (2).

All the previously mentioned transformation steps between the generic and the QUBO optimization formulations are implemented in the **quark** package, through the application of several QUBO composition steps as shown in Figure 1. The final QUBO can then be the input for different solvers. Currently, we support four: the classical mixed-integer problem solver SCIP (Solving Constraint Integer Programs),[6] the classical Python Simulated Annealing (PySA) solver,[7] which implements the simulated annealing algorithm, the D-Wave quantum annealer and the QAOA.jl solver.[8] The latter implements the QAOA on a classical quantum computer simulator, which uses YAO[9] as simulation back end.

Modular Implementation Concept

In the following, we highlight our implementation concept of optimization problems. Throughout the remainder of the article, the traveling salesperson problem (TSP) serves as an example problem to illustrate the lifecycle of a generic optimization problem. The definition of TSP as optimization problem is shown in Figure 1, and the illustration of the most important programming steps of such a problem are shown in Figure 2.

In **quark**, the encoding of an optimization problem is separated into two logical components. One is the **Instance** class, which defines the data container that provides all necessary input for the actual problem formulation. In general, the input format is highly problem specific and thus needs to be defined individually by the user. In the case of the TSP, the instance container should be able to encode a weighted complete graph, e.g., with input in form of a dictionary of all edges and their weights. Conveniently, this structure can also be used to implement read and write functions. These can be implemented easily using the **quark** input-output (IO) utilities for HDF5 format files.

Based on this instance definition, the formulation of the problem's objective function and constraints [cf. (1)] is done using the **ConstrainedObjective** class. To implement the functions, **quark** provides the **Polynomial** classes, which allow to create and compute with symbolical polynomials using integer and binary variables. For the TSP problem, this means creating one polynomial for the quadratic objective function and one for each of the linear constraints. Having this basic input, **quark** can generate a QUBO, like shown in equation (2), from the **ConstrainedObjective** definition. This is illustrated in Figure 2, where the objective function and constraint is first transformed into a binary quadratic form stored in the **ObjectiveTerms** class [collecting individual terms such as (3)] and finally transformed into the QUBO format via the specification of the penalty weights. The final QUBO is stored in the **Objective** class.

All necessary steps in this workflow, such as the reduction of cubic or higher-order polynomials, the constraint handling and much more, can be easily applied through the features of **quark**, which are illustrated in Figure 1. Note that, while

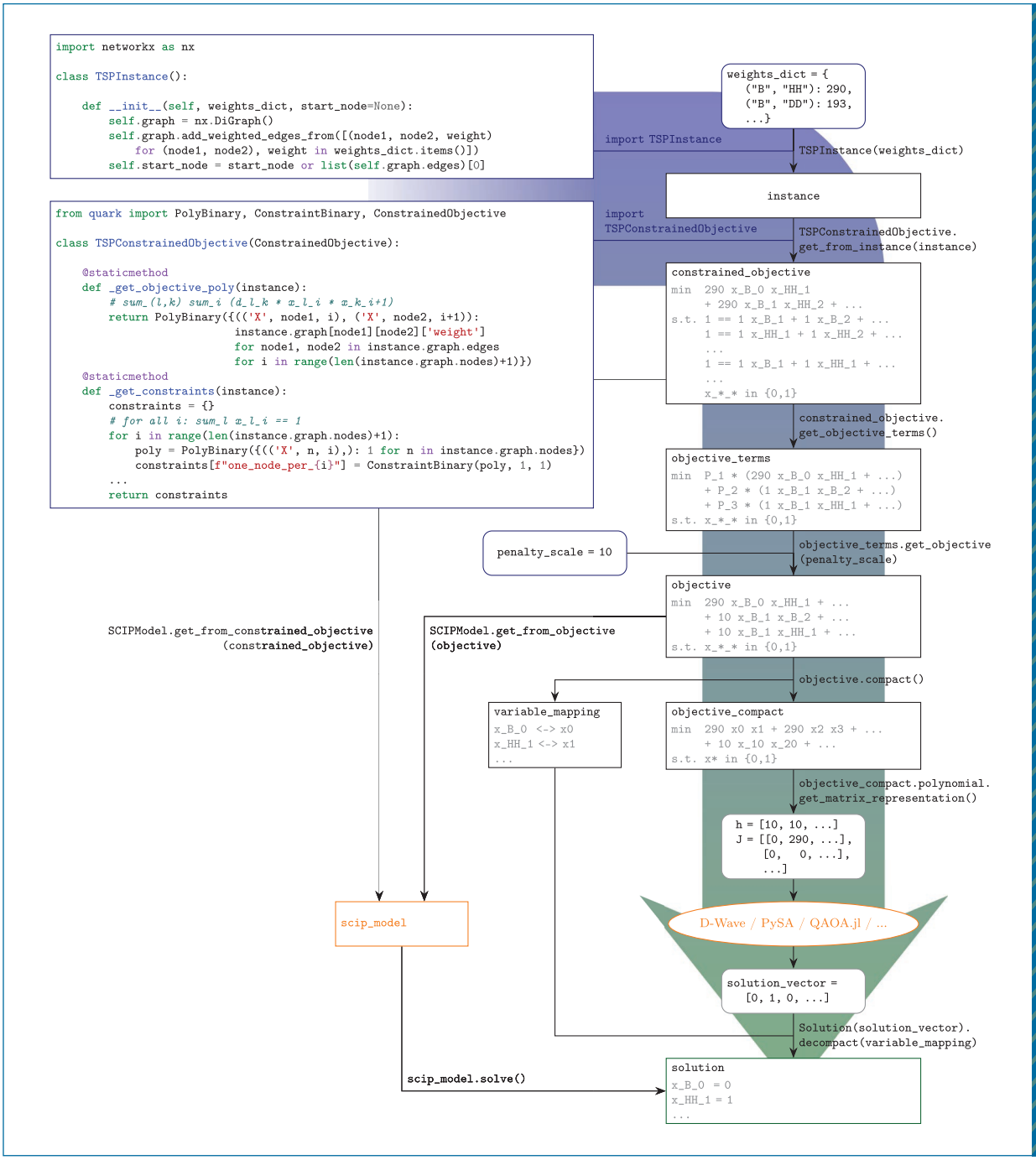


FIGURE 2. Simplified class workflow diagram in quark for the TSP problem. Users have to provide an **Instance** and a **ConstrainedObjective** class [cf. (1)] to define the use case, either by themselves or by loading them from **quapps**. A specific instance is defined via a dictionary containing the graph edges. Then all shown transformation steps to the QUBO format are automatically available. The blue boxes indicate the user input and the yellow boxes the supported solvers and algorithms.

those encompass some standard mathematical procedures, the optimal reduction of polynomials for instance is a nontrivial problem. For an exhaustive study of the performance of the different implemented reduction methods, we refer to the work by Schmidbauer et al.[10]

The final QUBO output is encoded in an appropriate format and can then be interfaced to multiple different computation back ends. For SCIP a corresponding **ScipModel** can be created and solved either directly from the optimization problem encoded in the **ConstrainedObjective** class or from the corresponding QUBO in the **Objective** class. The other solvers directly accept the QUBO coefficients as input. The computational result is returned as bit string or set of bit strings from the solvers and can then be decoded to the original problem formulation and checked against the problem constraints, as illustrated in Figure 1. In the example case of the TSP, the assignment of the binary variables describes the shortest path to traverse all nodes of the given input graph.

To show the possible outcome of such a simulation of the TSP, we have performed simulations of ten random TSP instances for increasing problem sizes, i.e., increasing number of stops, with edge weights between 0 and 10. We used the exact SCIP solver, the D-Wave Advantage2 prototype2.6 quantum annealer, computing 1,000 samples per instance, and its classical counterpart, the simulated annealing algorithm with 100 samples per instance. Note that we did not optimize the different algorithm and solver parameters, and thus our results do not serve as a benchmark but shall give the reader an idea of the different simulation back ends.

Figure 3(a) shows the success probabilities of the two heuristic methods to find the optimal solution, which has been calculated using the SCIP solver. One sees the exponential decrease in finding the exact ground state for both algorithms for increasing problem sizes, with the quantum algorithm being outperformed by its classical counterpart. In the cases where the success probability is zero, only suboptimal solutions were found. Figure 3(b) shows the computation time. These show the exponential increase in computation time of the classical algorithms against the constant scaling of the quantum annealing. Therefore, with improving quantum hardware, it is expected that these might be able to outperform the classical algorithms. The QUBO generation by **quark** performs decently, having computation times at least an order of magnitude faster than the classical solvers. Therefore, it is currently no computational bottleneck.

The **quapps** package is based on **quark** and provides a library of standard implementations of some

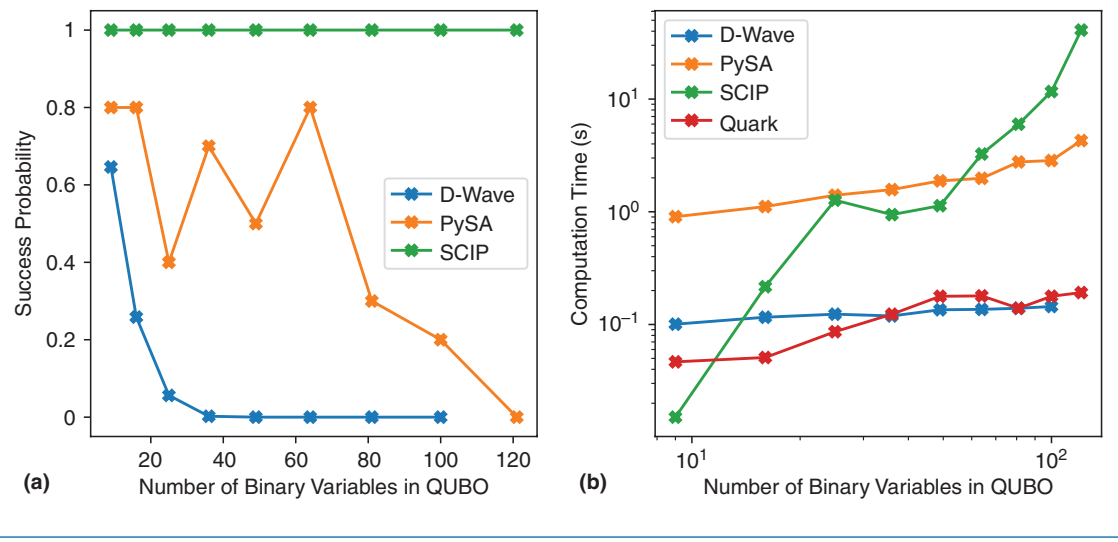


FIGURE 3. Exemplary results for the TSP problem for 10 random instances using different solver back ends. (a) The success probability with increasing problem size against the exact SCIP solution. (b) The computation time per instance for each solver. The red curve shows QUBO generation time by **quark**.

common discrete optimization problems, such as the TSP. It serves multiple purposes. First, it is a low-level entry point for users, who are new to both quantum computing and discrete optimization, to get familiar with exemplary optimization problems, their implementation in quark, and their formulation as QUBOs for quantum optimization. Currently, eight different problems can be easily loaded and instantiated using only basic input, like a graph. Second, these standard problems are often the base of more complex real-world problems, and users can easily extend them to their needs.

But the main strength of the quapps package is the easy generation of ready-to-use data for benchmarks. For this reason, the problem instances in quapps contain random instance generators, which allow for quick generation of such data. This is particularly useful for quantum algorithms in the noisy intermediate-scale quantum (NISQ) era, where every quantum hardware is still prone to error and every machine has a different set of requirements on both problem and algorithm to be a suitable solver. As such, the performance of different optimization problems and quantum algorithms can differ significantly depending on the hardware back end and have to be tested individually. Generally, it is a priori not known which of the algorithm and hardware choices are the best to solve a given optimization problem. This makes benchmarking the use case on different quantum computation platforms using different algorithms inevitable. Therefore, in conjunction to the quapps library, we also provide already instantiated optimization problems for these tests.

Instance Data Set for Benchmarking

In addition to the two open source libraries described in the previous chapters, we also provide an instance data set. It consists of already instantiated optimization problems for the problem types that are implemented in the quapps library. They are publicly available and will be permanently updated with new problem implementations added to the quapps library.[5]

We have chosen these random instances to test different aspects of possible quantum hardware. Therefore, for all problems implemented in quapps, we provide examples with increasing problem sizes, e.g., increasing graph sizes, which translates to a higher qubit count needed on the quantum hardware. Another typical property which we vary is the graph densities for all applications that are representable as a graph. This is useful to test the chosen hardware and algorithm setup for problems which require a high qubit connectivity.

Similarly, other input problem properties that may affect the performance of a quantum algorithm are covered by the instance data set. Specifically, the different problem types result for instance in different coefficient structures: while the unweighted Max Cut problem only yields coefficients of 1 and has no linear terms, the prime factorization formulation results in increasing powers of two for both the linear and quadratic terms and the random Ising models spread the coefficient values broadly.

As shown in Figure 1, the package quapps currently implements eight specific problem types for quantum optimization. A detailed description of each of these problem types can be found in the corresponding repository.[3] We will therefore not explain the problems in more detail here. All problem instances were saved in HDF5 files using our build in quark IO routines.

For six of the quapps problems, we have created benchmark instances using the random generators implemented within these problems. They provide problem instances with randomly chosen input values which depend on the structure of the problem, e.g., a fully connected graph with random edge weights. In view of the qubit numbers that will be available in the near future, we limited ourselves to instances with a maximum of 128 binary variables in the resulting reformulated Ising problem because each of these binary variables is mapped to at least one qubit but usually multiple qubits on the hardware to embed the problem into the hardware graph.

We have estimated that these problems are the maximum problem sizes runnable on current quantum hardware. These range between ≈ 56 qubits for ion trap architecture to a few hundreds of qubits for superconducting systems. The analog D-Wave annealer provides over 5,000 qubits. The latter system architectures, however, do not have fully connected qubits. Therefore, an optimization problem has to be embedded onto the hardware which significantly reduces the computable problem sizes. Nevertheless, we will update our instance data set constantly to match the increasing hardware resources in the coming years. All together this resulted in different combinations of the defining parameters, which are described for each problem in Figure 1. In total, we provide 764 already instantiated problems in our data set, which are ready to use for algorithmic and hardware benchmarking and testing.

Comparison With Other Tools

In this section, we embed our software into the growing landscape of quantum software. Specifically, we compare the quark package with some of the most used software packages for formulating QUBOs, which are D-Wave ocean SDK,[11] Qiskit,[12] Munich Quantum Toolkit - Quantum Auto Optimizer (MQT-QAO),[13] N-choose-K,[14] and PyQUBO.[15] The goal is to highlight the similarities and differences between the different packages and to embed our work into the zoo of existing frameworks. As criteria, we have chosen several advanced methods of the QUBO handling as well as a number of different interfaced standard solvers and algorithms.

Table 1. Comparison of different QUBO handling packages for some advanced QUBO handling techniques and their interfaced solvers/algorithms. Note, that we have not found the preprocessing features in the D-Wave ocean suite, but we also did not manage to verify their nonexistence due to the vast structure of this code.

	D-Wave ocean SDK[11]	Qiskit[12]	MQT-QAO[13]	N-choose-K[14]	Py-QUBO[15]	quark[2]
QUBO and Solution handling						
Multiple integer encodings	✗[d]	✗[d]	✓	✗	✓	✓
Can handle arbitrary polynomial constraints	✓	✗	✓	✗	✗	✓
Can handle boolean constraints	✗	✗	✓	✓	✓	✗
Advanced polynomial reduction techniques[a]	✓	✗	✓	✗	✗	✓
Automatic problem simplification in preprocessing[b]	?	✗	✗	✗	✗	✓
Validity and redundancy checks of constraints in preprocessing[c]	?	✗	✗	✗	✗	✓
Automated slack variable handling	✓	✗	✓	✗	✗	✓
Decoding of solution object	✓	✓	✓	✓	✓	✓
Automated constraint checks of solution	✓	✓	✓	✓	✓	✓
IO-routines for problem instances	✗	✗	✗	✗	✗	✓
Interfaced solver						
D-Wave annealer	✓	✓	✓	✓	✓	✓
Simulated annealing	✓	✓	✓	✗	✓	✓
QAOA simulator	✗	✓	✓	✓	✗	✓
VQE solver	✗	✓	✓	✓[e]	✗	✗
MI(NL)P solver	✗	✓	✗	✓[e]	✗	✓
Grover search	✗	✓	✓	✓[e]	✗	✗

[a] *Polynomial reduction techniques beyond the simple reduction of variable pairs in the naive ordering as they appear.*
[b] *Checks if the given objective function and constraints fix the value of some binary variables and substitute them accordingly.*
[c] *Includes some checks if the given constraints are commensurate with one another and if redundant ones exist. The latter will be omitted.*
[d] *Provides only binary encoding.*
[e] *Can be accessed via the Qiskit optimize package similar to the QAOA algorithm.*
MQT-QAO: Munich Quantum Toolkit-Quantum Auto Optimizer.

The results of this comparison are shown in Table 1. It shows that, while some basic features, such as solution decoding and constraint checks, are available in all listed packages, many of the more advanced QUBO handling techniques are absent in competing tools. This includes even essential functions, such as the handling of slack variables and polynomial reductions, which are highly nontrivial to implement as a nonexpert and vital for the QUBO formulation.

Analyzing the table, we identify the **quark** package, the MQT-QAO package and the D-Wave ocean as the most versatile tools. In terms of usability, the MQT-QAO package provides a highly optimized workflow for the handling of optimization problems, which makes it a strong competitor of our **quark** package. While **quark** has a few more preprocessing features which aim at simplifying the optimization problem and checking for redundant constraint in a preprocessing step as well as preimplemented IO-routines, it features no native connection to the Grover Search algorithm as the MQT-QAO package does. The biggest distinction between **quark** and MQT-QAO from a user perspective is the interfaced **quapps** package, which already provides a growing set of preimplemented optimization problems.

A similar set of preimplemented problems is only provided within the D-Wave Leap platform, which is a very powerful software suite but is in our experience, not intuitive in its handling. It provides many different problem classes, which can have very different input types and functionalities, that can be confusing for nonexpert users to handle. Therefore, we believe that our **quark** package is a useful addition to the existing quantum software, as it provides very easy problem encoding and handling and equips the users with all necessary tools to easily create benchmark instances of their own problems for different solvers and quantum algorithms.

In this work, we have outlined with **quark** and **quapps** two intertwined software packages that fill the need of easy-to-use software to code and ultimately run discrete optimization problems on both quantum and classical hardware. While **quark** provides powerful methods to encode and transform optimization problems into a format suitable for quantum computers and classical solvers, **quapps** provides a library of standard optimizations problems that can easily be used for both performance tests and as a base for more complex problems.

Through its interface to multiple different solver back ends, this allows also nonexpert users to test the best optimization platform and to identify the suitability of quantum algorithms for their problem. In this context benchmarking different quantum algorithms and hardware back ends plays a vital role within the NISQ era because of the variety of different quantum computer platforms. An optimization problem instance data set derived from the **quapps** package is provided to simplify this procedure. The data set will be updated continuously. We will soon add more problem types, such as the subset sum, knapsack, and graph partitioning problem, which have recently been added in **quapps**. The implementation of a load balancing and an antenna optimization problem is still ongoing. Bigger instance sizes will be added,

ABOUT THE AUTHORS

LUKAS WINDGÄTTER is a postdoctoral student at the Institute of Software Technology at the German Aerospace Center (DLR), Linder Höhe, 51147 Cologne, Germany. His research interests include quantum simulation and optimization of physical systems using quantum annealers and universal quantum computers. Windgätter received his Ph.D. in theoretical quantum physics from the University of Hamburg. Contact him at lukas.windgaetter@dlr.de.

ELISABETH LOBE is a research associate and group lead at the Institute of Software Technology at the German Aerospace Center (DLR), Linder Höhe, 51147 Cologne, Germany. Her research interests include quantum computing and optimization with a focus on the investigation of the capabilities of solving combinatorial optimization problems using quantum annealing technologies. Lobe received her Ph.D. in mathematics at Otto von Guericke University. Contact her at elisabeth.lobe@dlr.de.

whenever more powerful quantum hardware is available in the next years to run these.

In conjunction, the **quark** and **quapps** packages offer an important contribution to make the rapidly increasing field of quantum computing accessible to a broader audience, which has no prior knowledge in quantum information, and to simplify algorithm benchmarking for current quantum computers.

Acknowledgment

The presented research project is part of the project "Algorithms for quantum computer development in hardware-software codesign (ALQU)," which was made possible by the DLR Quantum Computing Initiative (QCI) and the German Federal Ministry for Economic Affairs and Climate Action (BMWK): https://qci.dlr.de/alqu. We also greatly acknowledge the effort of the other contributors put into the continuous development of the **quark** and **quapps** packages: https://gitlab.com/quantum-computing-software/quark/-/blob/development/CONTRIBUTORS and https://gitlab.com/quantum-computing-software/quapps/-/blob/development/CONTRIBUTORS.

References

1. E. Farhi, J. Goldstone, and S. Gutmann, "A quantum approximate optimization algorithm," 2014, *arXiv:1411.4028*.
2. E. Lobe and L. Windgätter. *quark - QUantum Application Reformulation Kernel*. (Version 1.2., Feb. 21, 2025). [Online]. Available: https://doi.org/10.5281/zenodo.13944213
3. E. Lobe and L. Windgätter. *quapps - QUantum APPlicationS*. (Version 1.0., Feb. 24 2025). [Online]. Available: https://doi.org/10.5281/zenodo.13944088
4. E. Lobe, "quark: QUantum application reformulation Kernel," in *INFORMATIK - Designing Futures: Zukünfte Gestalten*, Bonn, Germany: Gesellschaft für Informatik e.V., 2023, pp. 1115–1120, doi: 10.18420/inf2023_123.
5. E. Lobe and L. Windgätter. *quapps Instance Data Set*. (Version 1.0., Oct. 17, 2024). [Online]. Available: https://doi.org/10.5281/zenodo.13944133
6. S. Bolusani et al., "The SCIP optimization suite 9.0," Feb. 2024. [Online]. Available: https://optimization-online.org/2024/02/the-scip-optimization-suite-9-0/
7. S. Mandra, H. Munoz-Bauza, A. A. Asanjan, L. Brady, A. Lott, and D. B. Neira. *PySA: Fast Simulated Annealing in Native Python*. (Version 0.1.0., Mar. 2023). [Online]. Available: https://github.com/nasa/pysa
8. A. Misra-Spieldenner, T. Bode, P. K. Schuhmacher, T. Stollenwerk, D. Bagrets, and F. K. Wilhelm, "Mean-field approximate optimization algorithm," *PRX Quantum*, vol. 4, Sep. 2023, Art. no. 030335, doi: 10.1103/PRXQuantum.4.030335. [Online]. Available: https://github.com/FZJ-PGI-12/QAOA.jl/releases/tag/v1.3.3
9. X.-Z. Luo, J.-G. Liu, P. Zhang, and L. Wang, "Yao.jl: Extensible, efficient framework for quantum algorithm design," *Quantum*, vol. 4, p. 341, Oct. 2020, doi: 10.22331/q-2020-10-11-341. [Online]. Available: https://github.com/QuantumBFS/Yao.jl/releases/tag/v0.9.1
10. L. Schmidbauer, K. Wintersperger, E. Lobe, and W. Mauerer, "Polynomial reduction methods and their impact on QAOA circuits," in *Proc. IEEE Int. Conf. Quantum Softw. (QSW)*, 2024, pp. 35–45, doi: 10.1109/QSW62656.2024.00018.
11. "dwavesystems/dwave-ocean-sdk: Installer for D-wave's ocean tools," GitHub, version 8.2.0, Feb. 25, 2025. [Online]. Available: https://github.com/dwavesystems/dwave-ocean-sdk/
12. "qiskit-optimization 0.6.1," GitHub Pages, Feb. 28, 2024. [Online]. Available: https://github.com/qiskit-community/qiskit-optimization/
13. D. Volpe, N. Quetschlich, M. Graziano, G. Turvani, and R. Wille, "Towards an automatic framework for solving optimization problems with quantum computers," in *Proc. IEEE Int. Conf. Quantum Softw. (QSW)*, Shenzhen, China, Aug. 28, 2024, pp. 46–57, doi: 10.1109/QSW62656.2024.00019. [Online]. Available: https://github.com/cda-tum/mqt-qao/releases/tag/v0.3.0
14. E. Wilson, F. Mueller, and S. Pakin, "Combining hard and soft constraints in quantum constraint-satisfaction systems," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2022, pp. 1–14, doi: 10.1109/SC41404.2022.00018. [Online]. Available: https://github.com/lanl/NchooseK/tree/908e0a43db4af59ae2bb78b2f410c0330e1723f1
15. M. Zaman, K. Tanahashi, and S. Tanaka, "PyQUBO: Python library for mapping combinatorial optimization problems to QUBO form," *IEEE Trans. Comput.*, vol. 71, no. 4, pp. 838–850, Apr. 2022, doi: 10.1109/TC.2021.3063618. [Online]. Available: https://github.com/recruit-communications/pyqubo/releases/tag/1.5.0