Delft University of Technology

Document Version
Final published version

Citation (APA)
Zielinski, S., Benkard, M., Nüßlein, J., Linnhoff-Popien, C., & Feld, S. (2024). SATQUBOLIB: A Python Framework for Creating and Benchmarking (Max-)3SAT QUBOs. In F. Phillipson, G. Eichler, C. Erfurth, & G. Fahrnberger (Eds.), *Innovations for Community Services - 24th International Conference, I4CS 2024, Proceedings* (pp. 48-66). (Communications in Computer and Information Science; Vol. 2109 CCIS). Springer. https://doi.org/10.1007/978-3-031-60433-1_4

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
In case the licence states “Dutch Copyright Act (Article 25fa)”, this publication was made available Green Open Access via the TU Delft Institutional Repository pursuant to Dutch Copyright Act (Article 25fa, the Taverne amendment). This provision does not affect copyright ownership.
Unless copyright is transferred by contract or statute, it remains with the copyright holder.

Sharing and reuse
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.

Green Open Access added to TU Delft Institutional Repository

'You share, we take care!' - Taverne project

https://www.openaccess.nl/en/you-share-we-take-care

Otherwise as indicated in the copyright section: the publisher is the copyright holder of this work and the author uses the Dutch legislation to make this work public.

SATQUBOLIB: A Python Framework for Creating and Benchmarking (Max-)3SAT QUBOs

Sebastian Zielinski[1(✉)], Magdalena Benkard[1], Jonas Nüßlein[1], Claudia Linnhoff-Popien[1], and Sebastian Feld[2]

[1] Institute for Informatics, LMU Munich, 80538 Munich, Germany
sebastian.zielinski@ifi.lmu.de

[2] Quantum and Computer Engineering, Delft University of Technology, 2628 CD Delft, The Netherlands

Abstract. In this paper, we present an open-source Python framework, called satqubolib. This framework aims to provide all necessary tools for solving (MAX)-3SAT problems on quantum hardware systems via Quadratic Unconstrained Binary Optimization (QUBO). Our framework solves two major issues when solving (MAX)-3SAT instances in the context of quantum computing. Firstly, a common way of solving satisfiability instances with quantum methods is, to transform these instances into instances of QUBO, as QUBO is the input format for quantum annealers and the Quantum Approximate Optimization Algorithm (QAOA) on quantum gate systems. Studies have shown, that the choice of this transformation can significantly impact the solution quality of quantum hardware systems. Thus, our framework provides thousands of usable QUBO transformations for satisfiability problems. Doing so also enables practitioners from any domain to immediately explore and use quantum techniques as a potential solver for their domain-specific problems, as long as they can be encoded as satisfiability problems. As a second contribution, we created a dataset of 6000 practically hard and satisfiable SAT instances that are also small enough to be solved with current quantum(-hybrid) methods. This dataset enables meaningful benchmarking of new quantum, quantum-hybrid, and classical methods for solving satisfiability problems.

Keywords: 3-satisfiability · Optimization · QUBO transformation · Software framework

1 Introduction

Satisfiability problems play a central role in computer science. They have been among the first problems for which NP-completeness has been shown [9] and are often used to prove a given problem's hardness. Besides their use in theoretical computer science, they lie at the heart of many real-world application

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
F. Phillipson et al. (Eds.): I4CS 2024, CCIS 2109, pp. 48–66, 2024.
https://doi.org/10.1007/978-3-031-60433-1_4

domains, like circuit design and verification [20], error diagnosis in software systems [16], planning [25], configuration planning in wireless sensor networks [10], scheduling [5], solving problems in social networks [23], dependency resolution (e.g. in package managers of many operating systems) [2] and many more. For some domains, leveraging the advanced capabilities of contemporary SAT solvers, which are programs designed to solve satisfiability problems, proves to be an efficient, cost-effective, and maintainable approach for solving domain-specific problems. This is achieved by reformulating these domain-specific problems into satisfiability problems and then applying a modern SAT solver for resolution [13,17,24].

Despite significant research efforts over several decades, the satisfiability problem remains intractable. Thus, finding better methods of solving this problem is still an ongoing field of research. New methods of solving satisfiability problems employ AI techniques, such as artificial neuronal networks, to assist established methods in searching for correct solutions or creating new SAT solvers. Our paper primarily concerns solving satisfiability problems through quantum computing, which has gained much attention in the last decade.

With quantum computers' increased availability and capabilities in recent years, quantum computing has evolved from a mere theoretical domain to a field of practical interest. The proof that specific algorithms for quantum computers, like Shor's algorithm [27], theoretically provide exponential speedup for a classically intractable problem fuels the interest of researchers of different domains to find quantum algorithms for their domain-specific problems. However, some significant obstacles must be overcome to yield the full potential that theoretical findings in quantum computing promise. While many of these obstacles concern the physical manufacturing of quantum computers and their components, we only focus on software and application development challenges for quantum computers.

The first challenge related to the application of quantum computing we want to address is the input format of quantum computers and algorithms. An established practical method of solving satisfiability problems on quantum hardware includes transforming a satisfiability problem into an instance of Quadratic Unconstrained Binary Optimization (QUBO). These transformations are often highly technical, while software implementations are often unavailable. Furthermore, studies have shown that the choice of a transformation from a satisfiability instance to a QUBO instance can impact the solution quality significantly [18,31]. To leverage the full capabilities of currently available quantum hardware, it is thus insufficient to implement an arbitrary transformation from satisfiability problems to instances of QUBO. Several different transformations should be implemented and compared to get the best results. Practitioners of a non-quantum field thus need to read, understand, and implement several QUBO transformation methods only to be able to employ quantum methods to solve their domain-specific problems. Another challenge concerns the benchmarking of quantum and quantum-hybrid algorithms. Quantum hardware has been too small in the past decade to meaningfully compare its outputs to that of classical

solvers. However, this slowly begins to change. The main benefit of quantum computers is the potential to solve problem instances that currently cannot be solved efficiently by available classical methods. Thus, to receive a meaningful estimation of the capabilities of quantum and quantum-hybrid methods, they should be benchmarked on a set of problem instances that are hard for classical solvers. However, due to the limited size of current quantum computers, often, there is no dataset consisting of problem instances that are practically hard to solve for currently available classical solvers while also being small enough to be able to be solved on currently available quantum hardware using quantum or quantum-hybrid methods. To address these challenges, we present an open-source Python framework called ***satqubolib***. The main contributions of satqubolib are:

1. Ready to use Python implementations for thousands of transformations from satisfiability instances to instances of QUBO.
2. A dataset consisting of 6000 practically hard-to-solve satisfiability instances, which are also small enough to be solved on currently available quantum hardware through quantum or quantum-hybrid methods.
3. Implementation of two methods to quickly create practically hard-to-solve satisfiability instances of arbitrary sizes.
4. Example implementations for closely related questions (like a direct comparison of quantum-based SAT solvers vs. state-of-the-artclassical SAT solvers).

The main goals of our framework are to facilitate the use of QUBO-based quantum methods for practitioners of quantum and non-quantum domains alike and to increase the reproducibility of studies concerned with solving satisfiability problems through quantum computing.

The remainder of this paper is structured as follows. Section 2 introduces the necessary foundations of satisfiability and QUBO problems. In Sect. 3, we provide an overview of related work. Section 4 describes our framework's core architecture and most important usage scenarios. In Sect. 5, we conclude the paper and state future work.

2 Foundations

In this chapter, we will introduce the necessary definitions that are used throughout the remainder of this paper.

2.1 Satisfiability Problems

Satisfiability problems are concerned with the satisfiability of Boolean formulae. Thus, we will first define a Boolean formula:

Definition 1 (Boolean formula [3])**.** Let $x_1, ..., x_n$ be Boolean variables. A *Boolean formula* consists of the variables $x_1, ..., x_n$ and the logical operators AND($\wedge$), OR($\vee$), NOT($\neg$). Let $z \in \{0, 1\}^n$ be a vector of Boolean values. We

identify the value 1 as TRUE and the value 0 as FALSE. The vector z is also called an *assignment*, as it assigns truth values to the Boolean variables $x_1, ..., x_n$ as follows: $x_i = z_i$, where z_i is the $i-th$ component of z. If ϕ is a Boolean formula, and $z \in \{0,1\}^n$ is an assignment, then $\phi(z)$ is the evaluation of ϕ when the variable x_i is assigned the Boolean value z_i. If there exists a $z \in \{0,1\}^n$, such that $\phi(z)$ is TRUE, we call ϕ satisfiable. Otherwise, we call ϕ unsatisfiable [3].

Satisfiability problems are often given in conjunctive normal form, which we will define next:

Definition 2 (Conjunctive Normal Form [3])**.** A Boolean formula over variables $x_1, ..., x_n$ is in *Conjunctive Normal Form (CNF)* if it is of the following structure:

$$\bigwedge_i \Big(\bigvee_j y_{i_j} \Big)$$

Each y_{i_j} is either a variable x_k or its negation $\neg x_k$. The y_{i_j} are called the *literals* of the formula. The terms $(\vee_j y_{i_j})$ are called the *clauses* of the formula. A k*CNF* is a CNF formula, in which all clauses contain at most k literals. [3]

Given a Boolean formula ϕ in k*CNF*, the satisfiability problem is the task of determining whether k*CNF* is satisfiable or not. This problem was one of the first problems for which NP-completeness has been shown. [9]. In this paper, we will especially consider 3*CNF* problems, which we will refer to as 3SAT problems.

Solving satisfiability problems on a quantum annealer or through special algorithms on quantum gate systems requires a transformation to an optimization problem (see Subsect. 2.2)). Thus in these cases, we are solving a generalization of the satisfiability problem, the *MAX-SAT* problem. In the MAX-SAT problem, we are given a Boolean formula ϕ consisting of m clauses. The task is to find an assignment of truth values to the variables of ϕ such that as many clauses as possible are satisfied. Finding an assignment in the MAX-SAT problem that satisfies m clauses is thus equivalent to solving the corresponding satisfiability problem (i.e., determining whether ϕ is satisfiable). MAX-SAT is thus also NP-hard. Throughout this paper, we will call instances of satisfiability problems (SAT and MAX-SAT alike) *satisfiability instances* or *SAT instances*

2.2 Quadratic Unconstrained Binary Optimization (QUBO)

To solve satisfiability problems on a quantum annealer or a quantum gate computer using the quantum approximate optimization algorithm, they must be transformed into an instance of QUBO first.

A QUBO instance is defined as follows: [12]:

$$\text{minimize} \quad H(x) = x^T \mathcal{Q} x = \sum_i^n \mathcal{Q}_{ii} x_i + \sum_{i<j} \mathcal{Q}_{ij} x_i x_j \tag{1}$$

The n-dimensional vector $x = (x_1, x_2, ..., x_n) \in \{0,1\}^n$ represents an assignment of Boolean values to the Boolean variables x_i (for $1 \leq i \leq n$). Furthermore,

$Q \in \mathbb{R}^n \times \mathbb{R}^n$ is a $n \times n$-dimensional upper triangular matrix of constants, which is often also called the *QUBO matrix* [12]. To solve this problem, a vector $x = (x_1, x_2, ..., x_n) \in \{0,1\}^n$ needs to be found, such that $H(x) = x^T \mathcal{Q} x$ is minimal. This problem is NP-hard [12].

3 Related Work

As the field of solving satisfiability problems is already several decades old, a lot of helpful tools for solving these problems and benchmarking new solvers have been created. In this section, we will briefly mention the most important ones.

The first of these tools is SATLIB [14], which provides a collection of satisfiability problems of different types (for example, randomly generated instances or satisfiability instances that stem from graph coloring instances) that can be used to benchmark solvers. However, SATLIB was published in 2000, and according to the project's homepage [14], it has not been updated for over 20 years. We have solved all benchmark problems within the library to assess the difficulty of the provided instances with today's SAT solvers and modern hardware. We found that apart from a few instances that reached a self-imposed timeout of 6 h (i.e., the instances have not been solved within 6 h), all other instances were solved mostly within milliseconds, with some exceptions taking several seconds to a few minutes.

Another resource for finding possibly hard satisfiability instances that can be used for benchmarking is the annual SAT Competition [4]. At this competition, SAT solvers are benchmarked with regard to their capability of solving a set of pre-selected potentially hard-to-solve satisfiability instances within a certain amount of time. Although some of these instances are really hard (i.e., it can take several days to weeks to solve with state-of-the-artSAT solvers on state-of-the-artCPUs), these instances can get very large (i.e., several hundred thousand to millions of clauses). As current quantum hardware only has a limited amount of qubits available, satisfiability instances that consist of more than a couple of hundred clauses cannot be solved directly on these devices. However, the goal of our framework is to provide satisfiability instances that are challenging to state-of-the-artSAT solvers but can also be solved using quantum or quantum-hybrid methods on currently (or nearly) available quantum hardware. Thus, as part of our framework, we provide practically hard instances of different sizes (i.e., number of clauses) that can already be solved on currently available quantum hardware directly (depending on the method) or by using quantum-hybrid methods.

PySAT [15] is an open-source Python toolkit that provides many helpful features, like utility functions for manipulating formulas or creating encodings for pseudo-Boolean constraints into conjunctive normal form (CNF). The easy access to some of the state-of-the-art SAT solvers it provides is especially helpful. Our framework, however, is completely different from PySAT. We do not implement similar features that PySAT has already implemented but rather offer new and additional features that are concerned with the transformation of satisfiability instances into instances of QUBO, which is not in the scope of PySAT.

However, we will use PySAT in our `examples` package to demonstrate how to use state-of-the-art SAT solvers to create benchmark comparisons.

Finally, there is the Python framework PyQUBO [30], whose goal is to aid in the construction of QUBO formulations from given objective functions. It strives to create concepts that make it easy to read and write Python code concerning formulating QUBO problems and efficiently solving combinatorial optimization problems. PyQUBO can be seen as a helpful tool for developing a solution (especially for quantum annealing) for several classes of combinatorial optimization problems. However, because of the breadth of possible applications it offers, it sacrifices depth (i.e., PyQUBO cannot provide every possible QUBO transformation for every possible combinatorial optimization problem). Regarding solving satisfiability problems, PyQUBO offers a (singular) method to transform satisfiability problems into instances of QUBO. However, as shown by recent studies [18,31], it does not suffice to use an arbitrary method of transforming a given satisfiability instance to an instance of QUBO. One should instead evaluate multiple different methods of performing this transformation, as the results can change up to orders of magnitude just by changing the QUBO transformation. Thus, more than PyQUBO's provided functionality concerning the transformation of satisfiability problems to instances of QUBO is required for researchers or practitioners who want to leverage the full capabilities of quantum technologies to solve instances of satisfiability problems specifically.

4 Satqubolib: Creating and Benchmarking (Max-)3SAT Instances

This section explains the core architecture and functionalities of `satqubolib`. An overview of our framework's modules, packages, and features can be seen in Fig. 1. The framework is available via github[1] or as a pip package[2]. As shown in Fig. 1, the `dataset` of practically hard SAT instances (used to develop and benchmark QUBO transformations and quantum and quantum-hybrid algorithms), as well as the `examples` Python package, are only available via GitHub and not part of the pip package. We decided to separate the dataset from the pip installation, as the dataset will take up much space as it grows. Additionally, we decided to include several convenience implementations (e.g., for using SAT solvers via PySAT [15], metaheuristics like simulated annealing and tabu search) that require additional dependencies that are not needed for the core functionalities of `satqubolib`. Thus, we split the framework's core functionality from the benchmarking data and the complementary examples. In the following chapters, we will now demonstrate core usages. As we cannot present or explain every usage scenario, the `examples` package contains additional code examples and comments explaining the relevant concepts.

[1] https://github.com/ZielinskiSebastian/satqubolib/.

[2] pip install satqubolib

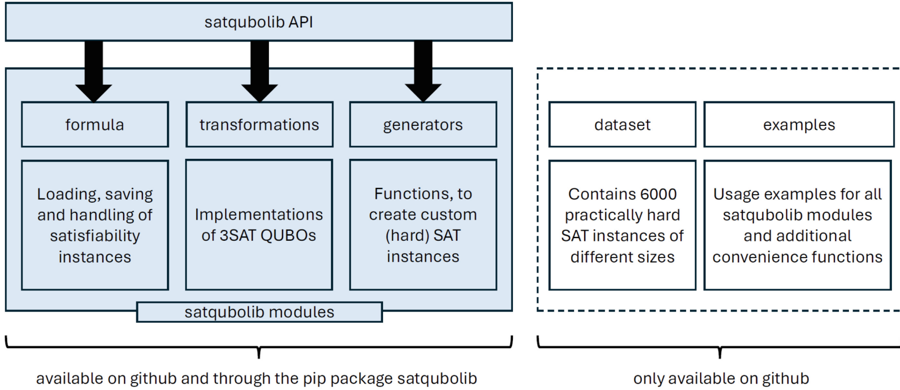

Fig. 1. Core architecture of the satquoblib's Python framework

4.1 Handling SAT Instances: The Satqubolib.formula Module

When using our framework to solve problems, at first, a 3SAT instance is needed. The python class `satquoblib.formula.CNF` within the `satquoblib.formula` module is the class representing a satisfiability instance in `satquoblib`. There are two possibilities for creating a `CNF` instance. The following listing shows these methods in line 2 and line 3.

```
1 from satqubolib.formula import CNF
2 my_formula = CNF([[1,2,3], [-2,-5,7]]
3 my_formula = CNF.from_file("/path/to/file/")
```

The first method, shown in line 2 of the above listing, is to represent the 3SAT instance $(x_1 \vee x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_5 \vee x_7)$, as a list of lists `[[1,2,3], [-2,-5,7]]`, where each of the inner lists represents a clause. The second method of creating a `CNF` instance, shown in line 3 of the above listing, is to load it directly from a file, in which the formula is specified in the .dimacs file format [1]. The .dimacs file format [1] is the standard format for representing instances of satisfiability problems in CNF form.

Besides the above-shown functionality, the `CNF` object contains several more convenience functions (e.g., to load metadata of the dataset we created for the framework). These usages and explanations can be found in the corresponding `examples` module (see Subsect. 4.5).

4.2 Creating SAT QUBOs: The Satqubolib.transformations Module

The `satqubolib.transformations` module contains all methods for transforming satisfiability instances to instances of QUBO. These transformations create QUBO instances Q that differ in several aspects, like the dimension of Q, the

density of Q (i.e., the number of non-zero elements in Q), and the energy spectrum. It has been shown in multiple studies ([18,31]) that the choice of a 3SAT-to-QUBO transformations can significantly impact the solution quality when solving these instances on quantum annealing hardware. Thus, our framework aims to provide implementations of state-of-the-art QUBO transformations for satisfiability problems.

One of the first 3SAT-to-QUBO transformations often used in scientific publications is the transformation by Choi [8]. Because this transformation was one of the first mappings from 3SAT to QUBO, it was also included in the seminal paper by Lucas [19]. Due to its widespread use in many publications of the past decade and to better understand this paper, we want to explain this transformation in more detail.

Choi's Transformation [8] is based on a well known reduction from 3SAT to the maximum independent set (MIS) problem. Let $G = (V, E)$ be an empty graph and $\phi = C_1 \wedge ... \wedge C_m$ be a 3SAT formula consisting of m clauses $C_1, ..., C_m$ over n Boolean variables $x_1, ..., x_n$. We now expand G as follows:

1. Let $y_{i,j}$ be the literal at position i of clause j of ϕ. For each literal we add a new vertex to V. We name this vertex according to the literals they represent, i.e., the vertex corresponding to literal $y_{i,j}$ is called vertex $v_{i,j}$.
2. Let $y_{i,j}$ and $y_{k,j}$ be two literals of the same clause, then we add an edge $(v_{i,j}, v_{k,j})$ to E.
3. Let $y_{i,j}$ and $y_{k,l}$ be two literals representing the same binary variable x_z but with different signs (i.e. $y_{i,j} = \neg y_{k,l}$) then we add an edge $(v_{i,j}, v_{k,l})$ to E.

Solving MIS on G will now also represent a solution to the 3SAT problem for formula ϕ. This transformation results in a QUBO matrix of dimension $3m \times 3m$.

Since the proposal of Choi's transformation, several new QUBO transformations for satisfiability problems have been proposed, e.g. [7,18,21,22,29]. We have explicitly implemented the transformations that result in QUBO matrices of dimension larger than $(n + m) \times (n + m)$. With regard to the transformations that lead to QUBO matrices of dimension $(n+m) \times (n+m)$, we only implemented transformations *explicitly*, if we have found studies using these transformations for practical benchmarking (or comparing). We want to highlight the use of the word *explicitly* here. An explicit implementation in our framework is provided by a dedicated class. There exist many thousand of $(n + m) \times (n + m)$-dimensional transformations. Creating an individual class for all of these transformations is infeasible. However, a method theoretically able to automatically create all $(n + m) \times (n + m)$-dimensional transformations from satisfiability problems to instances of QUBO has been published recently. This method is called the Pattern QUBO method [32]. By implementing this method, our framework provides ready-to-use access to all of these $(n+m) \times (n+m)$-dimensional transformations through a unified interface. Furthermore, `satqubolib` comes with a functionality that can create explicit implementations (i.e., dedicated standalone classes) for all of the transformations the Pattern QUBO method created. Because a

deeper understanding of this method is vital for efficiently using our framework, we explain the Pattern QUBO method in depth and demonstrate how to use it.

The **Pattern QUBO** approach was introduced in [32]. It is a meta method capable of automatically identifying thousands of different QUBO transformations for satisfiability problems. QUBO matrices created by this method are all of the dimension $(n + m) \times (n + m)$, where n is the number of variables and m is the number of clauses of a given 3SAT instance. This method can be seen as a generalization of existing QUBO transformations for the 3SAT problem. Manually created QUBO transformations (like the methods proposed by Chancellor [7], or Nüßlein [22]) follow some custom logic to finally arrive at a QUBO representation of a given 3SAT instance. However, all of these methods have in common that 1) all satisfying solutions for the 3SAT instance correspond to the minimum in the created QUBO optimization problem corresponding to the 3SAT instance, and 2) all non-satisfying solutions do not correspond to the minimum of the QUBO optimization problem. As it turns out, it is possible to exploit this property (solutions to a given 3SAT instance corresponds to optimal energy in the corresponding QUBO instance) to create new 3SAT-to-QUBO transformations automatically. Thus, this method provides an easy way to create and use previously published, but also to create and use thousands of previously unknown 3SAT-to-QUBO transformations, without the need to understand all the different logical deductions for these transformations, let alone the effort to implement all of them. By implementing the Pattern QUBO method, our framework provides:

1. Thousands of different ready-to-use $(n + m) \times (n + m)$-dimensional QUBO transformations.
2. Implicit implementations of all existing $(n+m) \times (n+m)$-dimensional QUBO transformations that result from superimposing clause QUBOs (like in the transformations by Nüßlein [22] and Chancellor [7].

Figure 2 shows an illustration of the general idea behind the Pattern QUBO method.

To transform an instance ϕ of 3SAT, consisting of m clauses, to an instance of QUBO, one first starts by sorting the variables of each clause, such that negated variables are always at the end of the clause. This does not change the formula or the difficulty of solving the formula, but it reduces the effort of finding transformations from a clause to QUBO, as only four possible clause types are remaining:

Type 0 - no negations: $(a \vee b \vee c)$
Type 1 - one negation: $(a \vee b \vee \neg c)$
Type 2 - two negations: $(a \vee \neg b \vee \neg c)$
Type 3 - three negations: $(\neg a \vee \neg b \vee \neg c)$

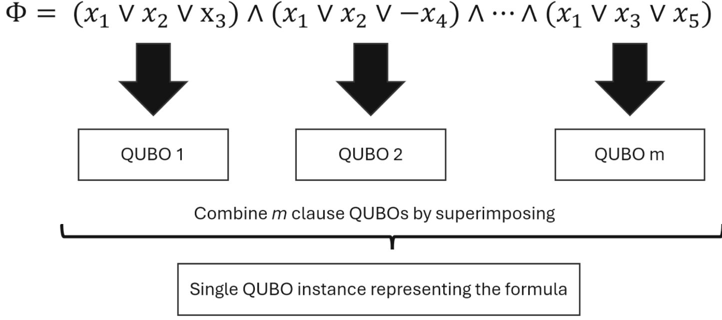

Fig. 2. Schematic representation of the Pattern QUBO method presented in [32]. Each clause of a given 3SAT formula ϕ gets transformed to a QUBO instance representing this clause. Each QUBO matrix one receives by transforming a clause of the 3SAT instance to an instance of QUBO is called a *clause QUBO*. All m of these clause QUBOs get combined into a single QUBO instance representing the formula ϕ.

As a next step, the Pattern QUBO method searches automatically for ways to transform each of these four prototype clauses (type 0 through type 3) to an instance of QUBO. For each of the 4 clauses, the search method starts with an empty (4×4)-dimensional QUBO matrix. The search method can insert a pre-determined set of values specified by the user into the QUBO matrix during its search process. As the search method is an exhaustive search, the running time of this method scales exponentially with the size of the set of values the algorithm is allowed to use. Using a small set, like {-1, 0, 1}, the search method only needs a few seconds to create 6 pattern QUBOs for clauses of type 0, 7 pattern QUBOs for clauses of type 1, 6 pattern QUBOs for clauses of type 2 and 8 pattern QUBOs for clauses of type 3.

We want to emphasize the following key takeaway of the Pattern QUBO method to fully make use of the benefits of its implementation in our framework: The Pattern QUBO method finds *multiple* ways to transform a clause of a given type (type 0 through type 3) to instances of QUBO (6 ways for clauses of *type 0*, 7 for clauses of *type 1*, 6 for clauses of *type 2* and 8 for clauses of *type 3*). However, we only need one of these pattern QUBOs per clause type. Thus, any 4-tuple (t_0, t_1, t_2, t_3), where t_i is a pattern QUBO for a clause of *type* i, can be seen as a valid transformation for a 3SAT instance to an instance of QUBO. Thus, in the above case, we have $2016 = 6 \cdot 7 \cdot 6 \cdot 8$ valid transformations from 3SAT instances to instances of QUBO. Choosing different combinations of transformations (t_0, t_1, t_2, t_3) from clauses to instances of QUBO leads to different (up to isomorphism) 3SAT-to-QUBO transformations with different properties. These properties have been empirically shown to influence the solution quality of current quantum annealing up to an order of magnitude [31]. Therefore, we encourage a user of this framework to try to combine different transformations

from clauses to instances of QUBO, to create different 3SAT-to-QUBO transformations in order to get the most benefit in practical and research applications (i.e., by empirically finding a transformation that is particularly beneficial for the given problems at hand).

We now demonstrate how to use the `satqubolib.transformations` module. The first use case is using an *explicitly* implemented QUBO transformation, like Choi's transformation, for a given 3SAT instance. This can be done as follows:

```
1  from satqubolib.formula import CNF
2  from satqubolib.transformations import ChoiSAT
3  my_formula = CNF([[1,2,3], [-2,-5,7]]
4  choi_model = ChoiSAT(my_formula)
5  choi_model.create_qubo()
6  qubo = choi_model.qubo
7  choi_model.print_qubo()
8  # suppose a QUBO solver gave the following solution:
9  solution = {0: 0, 1: 0, 2: 0, 3: 1, 4: 0, 5: 0}
10 output = choi_model.is_solution(solution)
11 # output in this case is a tuple: (False, 1)
```

We first created a `satquoblib.formula.CNF` object (line 3) in the listing above, as described in Subsect. 4.1. Then a `satqubolib.transformations.ChoiSAT` object, which implements Choi's transformation [8], is created in line 4. To create the QUBO matrix resulting from Choi's transformation applied to the formula we initialized the ChoiSAT object with, the `create_qubo()` function is called in line 5. The created QUBO is saved in the `qubo` variable of the `choi_model` object, which can be accessed as shown in line 6. For convenience, we provide every transformation within the `satqubolib.transformations` module with a visualization method called `print_qubo()` to inspect the created QUBO matrix. An example of the usage of this method is shown in line 7. As a next step, the created QUBO matrix is used as an input for a QUBO solver, like D-Wave's quantum annealing hardware, the QAOA algorithm on quantum gate computers, or classical heuristical methods like simulated annealing or tabu search. The output of these solvers are usually Python dictionaries of the form:

```
{variable_1: value, ..., variable_n: value}.
```

To check whether a solution given by a QUBO solver does indeed solve the given 3SAT instance, all QUBO transformations in our framework provide the method `is_solution(solution_dictionary)`. This method returns a tuple of the form (boolean, integer). The boolean indicates whether the solution is indeed a solution for the given 3SAT instance. The integer represents the number of satisfied clauses.

We now demonstrate how to use the Pattern QUBO method.

```
1  from satqubolib.transformations import PatternQUBOFinder
2  from transformations import PatternQUBONM
3  from satqubolib.formula import CNF
4  my_formula = CNF([[1,2,3], [-2,-5,7]]
5  pqf = PatternQUBOFinder(8)
```

```
6  clause_qubos = pqf.find({1,0,-1})
7  new_transformation = PatternQUBONM(my_formula)
8  new_transformation.add_clause_qubos(clause_qubos[0][0],
       clause_qubos[1][0], clause_qubos[2][0], clause_qubos
       [3][0])
9  new_transformation.create_qubo()
10 new_transformation.export("/path/to/file")
```

First, an object of `satqubolib.transformations.PatternQUBOFinder` must be created (line 5 of the above listing). The parameter value 8 in the constructor of the `PatternQUBOFinder` object in line 5 of the above listing denotes the number of parallel processes the Pattern QUBO search algorithm will use. As this search procedure is an exhaustive search, parallelizing the search is highly beneficial. To search for Pattern QUBOs for all clause types, the `find(allowed_values: set)` function of the `pqf` object is called (line 6 of the above listing). As explained previously, the `allowed_values` parameter of the `find(allowed_values: set)` function in line 6 of the above listing is the set of values the Pattern QUBO method is allowed to use (i.e., the Pattern QUBO method will examine all possible QUBO matrices that only consists of the values 1, 0 and -1). After the search for Pattern QUBO matrices is completed, a new 3SAT-to-QUBO transformation can be created. This is done in line 7 of the above listing. At this point, the transformation is not yet functional. To make this transformation a fully working 3SAT-to-QUBO transformation, one pattern QUBO for each clause type (clause type 0 through clause type 3) has to be added to the transformation. This is done in line 8 of the above listing. After this step is completed, the transformation can create a QUBO matrix representing the given satisfiability instance.

Finally, we would like to highlight a unique functionality of this implementation, shown in line 10 of the above listing. Each concrete instantiation of an object of type `satqubolib.transformations.PatternQUBONM`, where the four clause QUBOs have already been added (see line 8 of the above listing), can be exported to a standalone file. That is, if one finds a Pattern QUBO transformation that works particularly well for the given type of 3SAT instances, it can be exported to a file such that it does not require `satqubolib` as a dependency but instead works as a standalone implementation of a QUBO transformation.

4.3 Generating Hard SAT Instances: The Satqubolib.generators Module

In the `satqubolib.generators` module, we implemented methods that enable a user of `satqubolib` to create satisfiability instances of arbitrary parameterization (i.e., an arbitrary number of clauses and variables). Using different parameters for these algorithms will create instances that empirically vary in difficulty (i.e., choosing significantly more variables than clauses almost always results in trivially solvable SAT instances). We used the same methods to create the dataset of practically hard 3SAT instances that `satqubolib` provides. The algorithm implementations are provided through the classes BalancedSAT

and NoTriangleSAT in the `satqubolib.generators` module. The former is an implementation of the method provided by Spence [28] while the latter is an implementation of the method provided by Escamocher et al. [11]. The following listing shows how to use the Balanced SAT method to create new 3SAT instances.

```
1 from satquoblib.generators import BalancedSAT
2
3 balanced_generator = BalancedSAT(3, 10, 20)
4 # generate() returns an object of type satqubolib.formula.
     CNF
5 cnf = balanced_generator.generate()
```

The first parameter (the number 3) of the constructor of the BalancedSAT object in line 3 of the above listing denotes the number of variables in each clause. In this case, each clause consists of precisely three variables. The second parameter (the number 10) refers to how many variables the satisfiability instance possesses. The last parameter (the number 20) denotes the number of clauses of the satisfiability instance. It is known that to create hard satisfiability instances, the number of variables and the number of clauses of a satisfiability instance need to be in a certain ratio [26]. This ratio is different for each method of creating satisfiability instances. To create hard instances, the ratio of clauses to variables was experimentally determined to be around 3.6 (that is, 3.6 times more clauses than variables) for the Balanced SAT [28] method. For the No Triangle SAT method, this ratio seems to be around 4.1 (that is 4.1 more times more clauses than variables) [11]. However, not every instance with this clauses-to-variables ratio is also hard to solve. A modern SAT solver (like Kissat3.1.0 [6]) could be used to determine practical hardness. Suppose a modern SAT solver takes more than a few minutes to solve an instance. This instance is potentially interesting for a benchmark with a quantum, quantum-hybrid, or classical QUBO solver. For the creation of the dataset that is part of `satqubolib`, we found that using a clause-to-variable ratio of approximately 0.3 - 0.5 below the above-mentioned empirically determined hardness ratio yields the best results for creating hard satisfiable instances that take less than multiple hours to solve with Kissat3.1.0.

4.4 Benchmark Dataset of Practically Hard Satisfiability Instances

The goal of the dataset included in `satqubolib` is to enable the following pursuits meaningfully:

1. Benchmarking SAT solvers
2. Benchmarking quantum, quantum-hybrid, and classical QUBO solvers
3. Provide data input for creating non-trivial QUBOs

We will now first explain how we created our dataset before we explain how this dataset enables the endeavors mentioned above.

Our dataset consists of 6000 practically hard, satisfiable SAT instances simultaneously small enough to be solved with current quantum hardware using quantum or quantum-hybrid methods. The 2023 SAT Competition defines satisfiability instances as hard if the Minisat SAT solver cannot solve an instance within one minute on an AMD Ryzen 7 Pro 3700U CPU and 16GB RAM [4]. However, as the solving process is stopped after one minute, it is unclear how long it would take to solve this instance. The instance could be solved within the next minute or within a time frame of many hours. Furthermore, as the solving process is stopped, it is unclear whether the instance is satisfiable or not. For our dataset, we considered SAT instances as *practically hard* if they are satisfiable and if the Kissat3.1.0 SAT [6] solver needs more than 10 min to solve the instance on an AMD Ryzen Threadripper PRO 5965WX 4.5 GHz CPU. We decided to use the Kissat SAT solver instead of the Minisat SAT solver, as the Kissat SAT solver is currently among the best-performing SAT solvers.

Out of the 6000 instances our dataset comprises, 3000 were created randomly using the Balanced SAT method [28], while the other 3000 were created using the No Triangle SAT method [11]. As mentioned earlier in this section, we aim to create small and hard SAT instances. We found that utilizing the Balanced SAT and No Triangle SAT methods, generating satisfiability instances with fewer than 600 clauses proves exceedingly challenging. This difficulty arises from the advanced capabilities of modern SAT solvers and CPUs. Regardless of the algorithm parameterization, a majority of the created instances are solved within seconds. Creating practically hard instances between 600 and 800 clauses with these methods is generally possible but computationally very expensive, as formulas are often either solved within a few minutes or take many hours to be solved (i.e., by trying to create formulas in that range, we often hit a self-imposed solver timeout of 6 h, without a result of the SAT solver). Hence, we started our dataset with instances that consist of 800 clauses. We created 1000 instances for each of the SAT instance sizes *800 clauses*, *900 clauses*, and *1000 clauses* using the Balanced SAT method and repeated this procedure for the No Triangle SAT method. The problem-solving time distribution, for the 3000 instances created with the Balanced SAT method is shown in Fig. 3.

The y-axis of Fig. 3 shows the solution time in seconds. Note that the y-axis is log-scaled. The x-axis displays the instance sizes given by the number of clauses a SAT instance possesses. The least time the Kissat3.1.0 SAT solver needed to solve any of the 3000 SAT instances was slightly above 10 min, while the most time the solver needed to solve a SAT instance was 5 h and 30 min. This information is part of a metadata header that every SAT instance of our dataset possesses. The full header is shown in the following listing:

```
1  c Solution 1 -2 3
2  c Time 600
3  c Solver Kissat3.1.0
4  c CPU AMD_Ryzen_Threadripper_PRO_5965WX_4.5_GHZ
```

As is convention in the Dimacs file format, each line of the above listing starts with the letter *c*, which signals any potential solver that this line is a comment

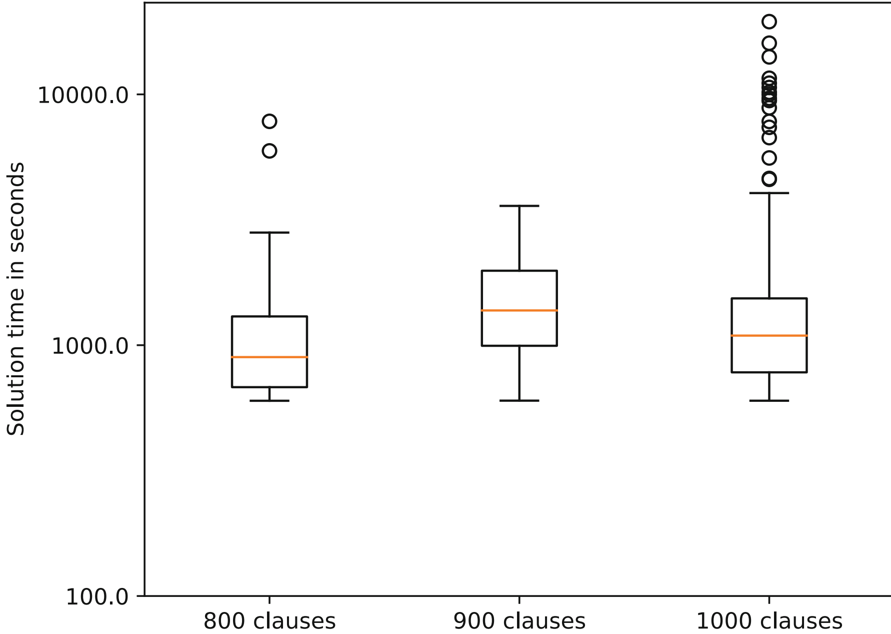

Fig. 3. Problem-solving time distribution for SAT instances created with the Balanced SAT method. Each boxplot consists of the solution times, measured in seconds, of 1000 SAT instances of a fixed instance size (800 clauses, 900 clauses, and 1000 clauses). Note that the y-axis is log-scaled.

and can be skipped. The first information each SAT instance contains is its solution. In line 1 of the above listing, the example solution `1 -2 3` is the short version of $x_1 = True$, $x_2 = False$, $x_3 = True$. Line 2 of the above listing states the exact time needed to solve this instance using the SAT solver shown in line 3 of the above listing with the CPU shown in line 4 of the above listing.

Because of the construction of our dataset, we enable multiple pursuits. First, we enable meaningful benchmarking of new quantum and quantum-hybrid algorithms for QUBO-based solving of SAT instances. QUBO-based SAT solvers, including quantum and quantum-hybrid methods, belong to a class of SAT solvers called *incomplete solvers*. These types of solvers, often including heuristic methods, ideally return a correct answer if the SAT instance is satisfiable. However, if these solvers do not return a correct answer, it does not immediately follow that the given SAT instances are indeed not satisfiable. In this case, the solver may not have found a correct solution yet. Thus, by providing a dataset of only satisfiable instances alongside a satisfying solution for this instance, QUBO-based SAT solvers can be benchmarked by their ability to find answers to solvable problems. Furthermore, our instances are not only challenging for state-of-the-artSAT solvers but also rather small, allowing these instances to be solved by purely quantum or quantum-hybrid methods on currently available hardware. Note that depending on the chosen hardware system and method

of modeling SAT instances as instances of QUBO, a pure quantum method might not be possible right now. However, because of the small size of the provided SAT instances, quantum hardware available in the near future should be able to solve these SAT instances by employing pure quantum methods.

As these instances are also challenging for state-of-the-artclassical SAT solvers, this dataset can also be used to benchmark new classical methods of solving satisfiability problems.

Finally, we will aid the development of new QUBO solvers. Hard QUBO instances are needed to benchmark the capabilities of QUBO solvers. Using our dataset of hard SAT instances and applying the 3SAT-to-QUBO transformations provided by our framework to these instances, potentially hard-to-solve (but in any case interesting) QUBO instances can be created. We assume these QUBO instances to be hard, as they represent SAT instances that cannot be efficiently solved using state-of-the-artSAT solvers. If a QUBO solver could solve these QUBO instances efficiently, this QUBO solver would be the preferred method of solving the corresponding SAT instances.

4.5 The Satqubolib.examples Package

We demonstrated the most important usage scenario for each of the modules `satqubolib` provides. However, these explanations do not span the whole range of implemented functionalities. Therefore, `satqubolib` provides the `examples` package: `satqubolib.examples`. This package contains one module for each of satqubolib's base modules (e.g., there is `satqubolib.examples.formula`, `satqubolib.examples.transformations` and `satqubolib.examples.generators`) providing documented examples for various usage scenarios of the respective module. Furthermore, we included convenience functions and usage scenarios, such as using a SAT solver provided by PySAT, to test whether the satisfiability instances created by our `generators` module are hard and satisfiable.

4.6 Framework Maintenance and Community Involvement

By implementing several known 3SAT-to-QUBO transformations and the Pattern QUBO method, our framework provides thousands of QUBO transformations. However, there probably exist interesting 3SAT-to-QUBO transformations that we still need to implement. To increase the reproducibility of scientific studies and the range of possible applications, we encourage contributing to `satqubolib` by providing implementations of QUBO models via pull requests in the framework's GitHub[2], or by hinting us of their existence. We also plan to keep our dataset up to date. As quantum computers' computational capabilities grow, we plan to add new, more challenging-to-solve instances to the benchmark dataset to reflect this change. Therefore, we also encourage submissions of new practically hard satisfiability instances that are at the same time small enough to

[2] (https://github.com/ZielinskiSebastian/satqubolib).

be solved by currently available quantum hardware (either directly or through quantum-hybrid methods). The contributed satisfiability instances do not need to be created via the Balanced SAT or No Triangle SAT methods.

5 Conclusion

Satisfiability problems are ubiquitous in practical computer science. For many domains (like planning, dependency resolution, and more), transforming a domain-specific problem into an instance of a satisfiability problem is an effective and maintainable method of solving these problems due to the capabilities of modern SAT solvers. In this paper, we presented `satqubolib`, an open-source Python framework that facilitates scientific publications' reproducibility and the use, development, and benchmarking of QUBO-based quantum and quantum-hybrid methods for solving satisfiability problems. Our framework thus also bridges the gap between quantum (optimization) technologies and researchers or practitioners of these related domains by providing easy-to-use mappings from resulting satisfiability instances to instances of QUBO, which is the input model for several different quantum methods (like quantum annealing or the QAOA algorithm on quantum gate systems).

By implementing several well-known and widespread transformations from satisfiability problems to instances of QUBO and the Pattern QUBO method, our framework provides thousands of ready-to-use 3SAT-to-QUBO transformations. As part of the framework, we also created a dataset containing 6000 practically hard and satisfiable 3SAT instances of different sizes using the Balanced SAT and the No Triangle SAT methods to enable meaningful benchmarking of newly developed quantum and quantum hybrid methods. Each formula within the dataset possesses a metadata header that contains the solution of this formula, the solution time, the processor, and the SAT solver that was used to solve the formula. Finally, we also enable the user to create custom satisfiability instances of arbitrary sizes (and hardness) by implementing and providing the Balanced SAT and No Triangle SAT methods. We hope that `satqubolib` will help to bridge the gap between application domains, in which SAT solvers are a vital part of the solution process, and quantum technologies.

In the future, we plan to include further structurally different satisfiability instances (of equal hardness and size) to enable broader benchmarking of quantum and quantum-hybrid methods and, thus, to identify possibly interesting domains for applying these methods. Furthermore, we want to include harder satisfiability instances of equal size that take even longer (i.e., multiple days), where we do not know whether these instances are satisfiable or not. As the capabilities of modern SAT solvers and hardware systems grow, we will continually update the instance sizes (i.e., the number of clauses of the satisfiability instances) within our dataset to ensure that the dataset remains a viable resource for benchmarking. Finally, we plan to keep track of future publications on transformations from satisfiability problems to instances of QUBO and will provide them in our framework.

Acknowledgments. The partial funding of this paper by the German Federal Ministry of Education and Research through the funding program "quantum technologies — from basic research to market" (contract number: 13N16196) is gratefully acknowledged.

References

1. 2011, S.C.: Sat competition 2011: Benchmark submission guidelines (2011). https://satcompetition.github.io/2023/benchmarks.html. Accessed 9 Feb 2024
2. Abate, P., Di Cosmo, R., Gousios, G., Zacchiroli, S.: Dependency solving is still hard, but we are getting better at it. In: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 547–551. IEEE (2020)
3. Arora, S., Barak, B.: Computational complexity: a modern approach. Cambrtidge University Press (2009)
4. Balyo, T., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.): Proceedings of SAT Competition 2023: Solver, Benchmark and Proof Checker Descriptions. Department of Computer Science Series of Publications B, Department of Computer Science, University of Helsinki, Finland (2023)
5. Barták, R., Salido, M.A., Rossi, F.: Constraint satisfaction techniques in planning and scheduling. J. Intell. Manuf. **21**, 5–15 (2010)
6. Biere, A., Fleury, M.: Kissat homepage (2020). https://fmv.jku.at/kissat/ Accessed Feb 11 2024
7. Chancellor, N., Zohren, S., Warburton, P.A., Benjamin, S.C., Roberts, S.: A direct mapping of max k-sat and high order parity checks to a chimera graph. Sci. Rep. **6**(1), 37107 (2016)
8. Choi, V.: Adiabatic quantum algorithms for the np-complete maximum-weight independent set, exact cover and 3sat problems. arXiv preprint arXiv:1004.2226 (2010)
9. Cook, S.A.: The complexity of theorem-proving procedures. In: Proceedings of the Third Annual ACM Symposium on Theory of Computing, pp. 151–158 (1971)
10. Duan, Q., Al-Haj, S., Al-Shaer, E.: Provable configuration planning for wireless sensor networks. In: 2012 8th International Conference on Network and Service Management (cnsm) and 2012 Workshop on Systems Virtualiztion Management (svm), pp. 316–321. IEEE (2012)
11. Escamocher, G., O'Sullivan, B., Prestwich, S.D.: Generating difficult sat instances by preventing triangles. arXiv preprint arXiv:1903.03592 (2019)
12. Glover, F., Kochenberger, G., Du, Y.: A tutorial on formulating and using qubo models. arXiv preprint arXiv:1811.11538 (2018)
13. Gutin, G., Karapetyan, D.: Constraint branching in workflow satisfiability problem. In: Proceedings of the 25th ACM Symposium on Access Control Models and Technologies, pp. 93–103 (2020)
14. Hoos, H.H., Stützle, T.: Satlib: an online resource for research on sat. Sat **2000**, 283–292 (2000)
15. Ignatiev, A., Morgado, A., Marques-Silva, J.: PySAT: a python toolkit for prototyping with SAT oracles. In: Beyersdorff, O., Wintersteiger, C.M. (eds.) Theory and Applications of Satisfiability Testing – SAT 2018: 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9–12, 2018, Proceedings, pp. 428–437. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-94144-8_26

16. Jose, M., Majumdar, R.: Cause clue clauses: error localization using maximum satisfiability. ACM SIGPLAN Notices **46**(6), 437–446 (2011)
17. Karapetyan, D., Parkes, A.J., Gutin, G., Gagarin, A.: Pattern-based approach to the workflow satisfiability problem with user-independent constraints. J. Artif. Intell. Res. **66**, 85–122 (2019)
18. Krüger, T., Mauerer, W.: Quantum annealing-based software components: An experimental case study with sat solving. In: Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, pp. 445–450 (2020)
19. Lucas, A.: Ising formulations of many np problems. Front. Phys. **2**, 5 (2014)
20. Marques-Silva, J.P., Sakallah, K.A.: Boolean satisfiability in electronic design automation. In: Proceedings of the 37th Annual Design Automation Conference, pp. 675–680 (2000)
21. Nüßlein, J., Gabor, T., Linnhoff-Popien, C., Feld, S.: Algorithmic qubo formulations for k-sat and hamiltonian cycles. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion, pp. 2240–2246 (2022)
22. Nüßlein, J., Zielinski, S., Gabor, T., Linnhoff-Popien, C., Feld, S.: Solving (max) 3-sat via quadratic unconstrained binary optimization. arXiv preprint arXiv:2302.03536 (2023)
23. Radicchi, F., Vilone, D., Yoon, S., Meyer-Ortmanns, H.: Social balance as a satisfiability problem of computer science. Phys. Rev. E **75**(2), 026106 (2007)
24. Rintanen, J.: Planning as satisfiability: Heuristics. Artif. Intell. **193**, 45–86 (2012)
25. Rintanen, J., Heljanko, K., Niemelä, I.: Planning as satisfiability: parallel plans and algorithms for plan search. Artif. Intell. **170**(12–13), 1031–1080 (2006)
26. Selman, B., Mitchell, D.G., Levesque, H.J.: Generating hard satisfiability problems. Artif. Intell. **81**(1–2), 17–29 (1996)
27. Shor, P.W.: Algorithms for quantum computation: discrete logarithms and factoring. In: Proceedings 35th Annual Symposium on Foundations of Computer Science, pp. 124–134. IEEE (1994)
28. Spence, I.: Balanced random sat benchmarks. SAT Competition **2017**, 53 (2017)
29. Verma, A., Lewis, M., Kochenberger, G.: Efficient qubo transformation for higher degree pseudo boolean functions. arXiv preprint arXiv:2107.11695 (2021)
30. Zaman, M., Tanahashi, K., Tanaka, S.: Pyqubo: Python library for mapping combinatorial optimization problems to qubo form. IEEE Trans. Comput. **71**(4), 838–850 (2021)
31. Zielinski, S., Nüßlein, J., Stein, J., Gabor, T., Linnhoff-Popien, C., Feld, S.: Influence of different 3sat-to-qubo transformations on the solution quality of quantum annealing: A benchmark study. In: Proceedings of the Companion Conference on Genetic and Evolutionary Computation, pp. 2263–2271. GECCO '23 Companion, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3583133.3596330
32. Zielinski, S., Nüßlein, J., Stein, J., Gabor, T., Linnhoff-Popien, C., Feld, S.: Pattern qubos: algorithmic construction of 3sat-to-qubo transformations. Electronics **12**(16) (2023). https://doi.org/10.3390/electronics12163492, https://www.mdpi.com/2079-9292/12/16/3492