PyQUBO: Python Library for Mapping Combinatorial Optimization Problems to QUBO Form

Mashiyat Zaman ⓘ, Kotaro Tanahashi ⓘ, and Shu Tanaka ⓘ

Abstract—We present PyQUBO, an open-source Python library for constructing quadratic unconstrained binary optimizations (QUBOs) from the objective functions and the constraints of optimization problems. PyQUBO enables users to prepare QUBOs or Ising models for various combinatorial optimization problems with ease thanks to the abstraction of expressions and the extensibility of the program. QUBOs and Ising models formulated using PyQUBO are solvable by Ising machines, including quantum annealing machines. We introduce the features of PyQUBO with applications in the number partitioning problem, knapsack problem, graph coloring problem, and integer factorization using a binary multiplier. Moreover, we demonstrate how PyQUBO can be applied to production-scale problems through integration with quantum annealing machines. Through its flexibility and ease of use, PyQUBO has the potential to make quantum annealing a more practical tool among researchers.

Index Terms—Quantum annealing, QUBO, Ising machine, combinatorial optimization, Python

✦

1 INTRODUCTION

COMBINATORIAL optimization is the calculation of the maxima or minima of a function within a discrete domain. Various combinatorial optimization problems, such as schedule and shift planning, delivery, and traffic flow, exist in daily life. Such problems often contain numerous possible solutions, making an exhaustive search intractable and hence creating an increasing demand for more efficient technology.

To overcome such computational limitations, a new type of computation technology known as the Ising machine was developed. In 2011, the first commercial quantum annealing machine was presented [1]. The hardware of existing quantum annealing machines has been developed based on the theories of quantum annealing [2] and adiabatic quantum computation [3], [4]. Ising machines are inspired not only by quantum annealing but also other principles that have been developed since the emergence of the first commercial quantum annealer [5], [6], [7], [8], [9], [10]. A number of studies utilizing Ising machines have been conducted in various fields: portfolio optimization[11], traffic optimization [12], rectangle packing optimization [13], item listing optimization for e-commerce websites[14], and materials design [15].

To use Ising machines in solving a problem, the energy function of Ising model or quadratic unconstrained binary optimization (QUBO) corresponding to the objective function and constraints of the problem must be prepared. Here, we refer to the energy function as Hamiltonian. However, programming Ising models and QUBOs for Ising machines may be challenging when the objective function and constraints are complicated. Thus, we developed PyQUBO as a Python library for programming QUBOs and Ising models. Using PyQUBO's high-level class objects, users can construct Hamiltonians intuitively. Not only does PyQUBO make it easier to read and write code, but it also makes a program more extensible, thereby allowing users to solve combinatorial optimization problems more efficiently. Through its accessibility and extensibility, PyQUBO has the potential to make quantum annealing more common among researchers across a wide range of fields.

In this paper, we demonstrate how PyQUBO can be used to express QUBO and the Hamiltonian of Ising model as easily readable Python code. The remainder of this paper is organized as follows. In Section 2, we introduce the Ising machine and explain how we use one to solve a combinatorial optimization problem. In Section 3, we formulate the combinatorial optimization problem, and in Section 4, we demonstrate how it can be expressed in terms of a QUBO or Ising model. In Section 5, we introduce PyQUBO and explore the motivation for its development. In Section 6, we demonstrate how combinatorial optimization problems can be solved by simply using the fundamentals of PyQUBO. In Section 7, we present advanced PyQUBO methods for writing and debugging complex problems. In Section 8, we explain PyQUBO for logical gates. In Section 9, we address the use of PyQUBO in conjunction with the D-Wave Ocean System software and D-Wave

• Mashiyat Zaman and Kotaro Tanahashi are with Recruit Co., Ltd., Chuo-ku, Tokyo 104-0032, Japan. E-mail: tanahashi@r.recruit.co.jp.
• Shu Tanaka is with Department of Applied Physics and Physico-Informatics, Keio University, Kanagawa 223-8522, Japan, with the Green Computing Systems Research Organization, Waseda University, Tokyo 162-0042, Japan, and also with the Precursory Research for Embryonic Science and Technology, Japan Science and Technology Agency, Kawaguchi-shi 332-0012, Japan. E-mail: shu.tanaka@appi.keio.ac.jp.

2000Q quantum annealing machine. In Section 10, we present internal implementations of PyQUBO and benchmark the performance with different implementations including other packages. Section 11 is devoted to the conclusion of the paper.

2 USE OF ISING MACHINES

In this section, we explain how Ising machines are used to solve combinatorial optimization problems. In general, five steps are involved in using Ising machines to solve optimization problems [16], as follows:

1) Discern a combinatorial optimization problem from the issue.
2) Represent the combinatorial optimization problem using an Ising model.
3) Embed the Ising model into the Ising machine according to the hardware specifications and determine the hyperparameters.
4) Search for the low-energy states of the Ising model.
5) Interpret the final state to obtain feasible solutions to the original combinatorial optimization problem.

First, we need to identify the combinatorial optimization problem from the issue in question. Next, we formulate the optimization problem as an Ising model or QUBO. In this process, constraint terms are introduced to the Hamiltonian to satisfy the constraints. If the original optimization problem contains non-binary discrete variables, such as integer variables, these must be encoded using binary variables. The details of this process are explained in Sections 3 and 4.

Third, we map our logical Ising model onto the physical Ising model, which consists of variables and interactions between variables that are implemented on the Ising machine. This mapping process is known as embedding. Because embedding is itself a combinatorial optimization problem, several efficient embedding algorithms have been proposed [17], [18], [19]. We also need to specify the hyperparameters, that is, the coefficients of the constraint terms. Fourth, we use the Ising machine to obtain the low-energy states of the physical Ising model. Finally, we interpret the variable states from the Ising machine and obtain the states corresponding to the logical Ising model. We determine whether the solution satisfies the problem constraints: if not, we repeat step 3 and update the hyperparameters.[1] Through these five steps, we can obtain feasible solutions to combinatorial optimization problems using the Ising machine.

3 COMBINATORIAL OPTIMIZATION

The mathematical formulation for a combinatorial optimization problem is expressed as follows:

$$z^* = \arg\min_z f(z), \quad z \in \mathcal{S},$$
$$\begin{cases} g_\ell(z) = 0 & (\ell = 1, \dots, L), \\ h_m(z) \leq 0 & (m = 1, \dots, M), \end{cases} \quad (1)$$

where z represents discrete integer decision variables of which number is n, $f(z)$ is the cost function, and $\mathcal{S}$ is the set
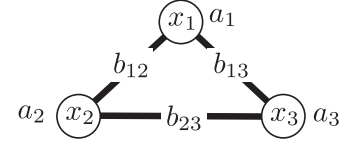

Fig. 1. Simple 3-binary variable system indicating weights a_i and interaction strengths b_{ij}.

of decision variables satisfying the given equality and inequality constraints $g_\ell(z)$ and $h_m(z)$.

Equation (1) can be rewritten as an optimization problem without any constraints using the penalty function method. Given the equality constraint $g(z) = 0$, we can consider the equation

$$z^* = \arg\min_z\{f(z) + \lambda[g(z)]^2\}, \quad z \in \mathcal{Z}^n. \quad (2)$$

Similarly, given the inequality constraint $h(z) \leq 0$, Eq. (1) can be rewritten as

$$z^* = \arg\min_z\{f(z) + \lambda\max[h(z), 0]\}, \quad z \in \mathcal{Z}^n. \quad (3)$$

In both equations, $z \in \mathcal{Z}^n$ indicates that the decision variables must be integers. For sufficiently large values of the coefficient λ, Eqs. (2) and (3) produce feasible solutions that satisfy the constraints with greater probability.

4 THE QUBO AND ISING MODEL

Ising machines use the QUBO or Hamiltonian of the Ising model to solve combinatorial optimization problems. The Ising model and QUBO are defined on an undirected graph $G = (V, E)$, where V and E are the sets of vertices and edges on G, respectively.

4.1 Ising Model
The Hamiltonian of the Ising model on G is expressed by

$$H_{\text{Ising}}(\boldsymbol{s}) = \sum_{i \in V} h_i s_i + \sum_{(ij) \in E} J_{ij} s_i s_j, \quad s_i \in \{-1, 1\}, \quad (4)$$

where s_i is the decision variable called spin at $i \in V$, h_i is the magnetic field at $i \in V$, and J_{ij} is the interaction at the edge i, j. Here h_i and J_{ij} are real numbers.

4.2 QUBO
The QUBO represents the cost function of a binary combinatorial optimization problem with linear and quadratic terms. Let x_i be the ith binary variable. Given the graph G, it is formulated as

$$H_{\text{QUBO}}(\boldsymbol{x}) = \sum_{i \in V} a_i x_i + \sum_{(ij) \in E} b_{ij} x_i x_j, \quad x_i \in \{0, 1\}, \quad (5)$$

where a_i and b_{ij} are real numbers. Here $b_{ij} = b_{ji}$ for arbitrary i and j. A QUBO defined on the undirected graph $G = (V, E)$ is illustrated in Fig. 1.

Let us confirm the equivalence between QUBO and Ising model. Using the relation $x_i = (s_i + 1)/2$, QUBO can be used to represent the combinatorial optimization problem in terms of the Ising model. The coefficients of Ising model are given by

---

1. Depending on the problem structure and how constraints are broken, it is possible to restore to a solution that satisfies constraints [20].

$$h_i = \frac{a_i}{2} + \sum_{j \in \partial_i} \frac{b_{ij}}{2}, \quad \forall i \in V, \tag{6}$$

$$J_{ij} = \frac{b_{ij}}{4}, \quad \forall (ij) \in E, \tag{7}$$

where ∂_i indicates the set of vertices connected to vertex i by the edges. By substituting Eqs. (6) and (7) into Eq. (4), we confirm the equivalence between QUBO and Ising model except for a constant value.

QUBO can be expressed in matrix form. Let Q_{ij} be a $|V| \times |V|$ matrix whose elements are given by

$$Q_{ij} = \begin{cases} a_i & [i = j, \forall i \in V] \\ b_{ij} & [\forall (ij) \in E \text{ and } i < j] \\ 0 & [\text{otherwise}] \end{cases}. \tag{8}$$

By using Q_{ij} and a column vector $\boldsymbol{x}$ generated by arranging binary variables, $H_{\text{QUBO}}(\boldsymbol{x})$ is rewritten by

$$H_{\text{QUBO}}(\boldsymbol{x}) = \sum_{i \leq j} Q_{ij} x_i x_j = \boldsymbol{x}^{\text{T}} Q \boldsymbol{x}, \tag{9}$$

where $\boldsymbol{x}^{\text{T}}$ represents the transpose of vector $\boldsymbol{x}$.

4.3 Combinatorial Optimization Problem Represented by Ising Model or QUBO

The equality and inequality constraints described by Eqs. (2) and (3) must be added as penalty terms that are also written as QUBOs. That is, the Hamiltonian of Ising model or QUBO can be generalized as

$$H = H_{\text{cost}} + \lambda H_{\text{const}}, \tag{10}$$

where λ determines the constraint term weight, H_{cost} is the cost function, and H_{const} is the penalty term, which is 0 when the constraint is satisfied and greater than 0 otherwise. Methods to construct H_{const} are explained in [21], [22], [23].

In summary, a combinatorial optimization problem can be made solvable by Ising machines by representing the cost function and constraints of the original problem using the linear and quadratic terms of the binary variables.

5 INTRODUCTION TO PyQUBO

Thus far, we have observed that the general combinatorial optimization problem needs to be formulated as QUBO (or an Ising model) to solve the problem using Ising machines. Practically, we take the following steps to obtain the QUBO corresponding to the optimization problem.

1) Formulate the problem as an integer programming (IP) problem.
2) Reformulate the optimization problem without constraints by introducing constraint terms to the objective function.
3) *Encode the integer variables with binary variables.*
4) *Expand the objective function.*
5) *Reduce the degree of higher-order terms.*
6) *Obtain the QUBO matrix from the coefficient of the polynomial.*

In the first step, we formulate the general combinatorial optimization problem presented in Eq. (1) as an IP problem. The formulations for various combinatorial optimization problems as IP problems have been studied in depth [24]. This process is also required when we solve the combinatorial optimization problem with general optimization solvers, such as Gurobi [25]. In the second step, we introduce the constraint terms into the objective function by using the penalty method, so that the optimization problem does not include any additional conditions (see Section 3). In the third step, the integer variables in the objective function are encoded with binary variables. As there are several means of encoding an integer variable and each type has different characteristics, we need to select the appropriate one carefully. In the fourth step, the objective function is expanded into a sum of products. If the products in the expanded polynomial have a degree greater than two, the order of the products needs to be reduced by introducing new variables and constraint terms into step 5. Finally, we combine the like terms of the expression to obtain the coefficient of QUBO.

It is necessary to use a program to conduct steps (3)–(6). However, to implement such a program, we need to calculate the expanded form of the objective function in advance, which is usually prepared by hand. Furthermore, when the objective function consists of several complicated terms, the program becomes complicated, which may result in software bugs. When we attempt to solve the problem with Ising machines, we generally try to create several formulations and encoding types for improved results. Thus, if we can transform the problem into QUBO rapidly, we can create different formulations and obtain superior results more efficiently.

To facilitate the creation of QUBOs, we developed a software tool known as PyQUBO. Using PyQUBO, one can define the Hamiltonian, that is, the objective function, with variable objects in a generic format. By calling the compile method of the expression, the QUBO matrix can be obtained instantly, without knowing the expanded form of the Hamiltonian (Section 6). If the order of the products in the polynomial exceeds two, the PyQUBO compiler automatically reduces the order and produces the corresponding QUBO matrix. PyQUBO provides several class modules, including multi-dimensional arrays (Supplementary Section A), which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TC.2021.3063618, integer classes with different encodings (Section 7.3), and logical gates (Section 8.1). These classes enable not only complicated expressions to be implemented quickly and easily but also the construction of modules on top of these (Section 8.2.1). As explained in step 5 of Section 2, the solutions need to be validated. PyQUBO also provides a feature to validate the solutions automatically (Section 7.1).

5.1 Quick Reference

A quick reference to each section, according to what the users wish to accomplish with PyQUBO, is provided below.

- *Automatic creation of QUBO*: PyQUBO automatically expands the terms of the Hamiltonian to produce the QUBO matrix, which is called compilation (Section 6.2). In addition, PyQUBO automatically

reduces the order of the polynomials during compilation (Section 6.2.3).

- *Validation of constraint satisfaction*: By using the `Model.decode_sample()` method, one can verify whether the given solutions are valid (Section 6.2.4).
- *Update of specific variables*: By defining a value with `Placeholder` when creating the Hamiltonian, the value we want to change can be specified even after compilation (Section 7.2).
- *Usage of integer decision variables instead of continuous decision variables*: Continuous variables can be approximately represented using the `Integer` class. For example, the continuous value $x \in [0, 1]$ is approximated by $\hat{x} \in \{0, 0.1, \ldots, 1.0\}$. In PyQUBO, $\hat{x}$ can be defined as `x=0.1*LogEncInteger("x", 0, 10)`. Other `Integer` class types are also available (Supplementary Section B), available online.
- *Usage of categorical variables*: In certain problems, such as the graph coloring problem [26], a discrete variable is used to represent the category $C \in \{C_1, C_2, \ldots, C_n\}$. This variable is known as a categorical variable. In PyQUBO, categorical variables can be defined using the `OneHotEncInteger` class. Refer to Supplementary Section B for further details, available online.
- *Usage of integer variables when inequality constraints exist*: PyQUBO provides various types of integer classes that can be used to create inequality constraints (Section 7.4).
- *Usage of logical variables*: PyQUBO provides logical gate classes and logical gate constraint classes, which are useful for formulating the satisfiability problem (SAT) or integer factoring as QUBOs (Section 8).
- *Connection to D-Wave machines and other Ising solvers*: As PyQUBO is included in the D-Wave Ocean package, PyQUBO can easily be integrated with the Ocean solver for D-Wave machines (Section 9). The format of QUBOs produced by PyQUBO is a key-value dictionary, which is compatible with other software tools for Ising solvers, such as OpenJij [27].
- *Validation of a QUBO created using PyQUBO*: The PyQUBO utility method `utils.asserts. assert_qubo_equal()` checks the equality of given QUBOs, which are symmetrized so that they can be compared.

6 PyQUBO Fundamentals

In this section, we will explain the essential classes of PyQUBO required to write and solve basic combinatorial optimization problems.

PyQUBO can be installed using `pip` as follows:

```
pip install pyqubo
```

Alternatively, GitHub users can install it from the source code:

```
git clone https://github.com/recruit-communi-
cations/pyqubo.git
cd pyqubo
python setup.py install
```

Supported Python versions are listed on the Github repository page.

6.1 Defining the Hamiltonian With the Express Class

The `Express` class is the abstract class of all operations used to write Hamiltonians in PyQUBO. Once defined, the Hamiltonian can easily be converted into a binary quadratic problem using the `compile()` class method.

6.1.1 Spin and Binary

The Hamiltonians of combinatorial optimization problems can be expressed in terms of the `Spin` $(-1, 1)$ and `Binary` $(0, 1)$ classes, corresponding to the Ising model and QUBO formulations, respectively. A `Binary` or `Spin` that is assigned to a variable must also be provided with a unique label. The labels of `Binary`/`Spin` variables can be used to interpret the coefficients of a QUBO or Ising problem more efficiently.

Codeblock 1. Creating spins and binaries in PyQUBO.

```
1 >>> from pyqubo import Binary, Spin
2 >>> a, b, c = Binary("a"), Binary("b"), Binary("c")
3 >>> p, q, r = Spin("p"), Spin("q"), Spin("r")
```

6.1.2 Add, Mul, and Num

PyQUBO interprets the built-in addition, multiplication, and power operators of Python, as well as the Python `int` and `float` values, as `Express` instances. `Spin` and `Binary` can also be added or multiplied using the `Add` and `Mul` classes, whereas numerical constants can be written using the `Num` class, given a float or integer as its parameter.

The following example is a simple Hamiltonian that is minimized when one of the binary variables a or b is 1

$$H = (a \times b - 1)^2, \quad a, b \in \{0, 1\}. \tag{11}$$

Codeblock 2 demonstrates how we can express the Hamiltonian using PyQUBO:

Codeblock 2. Arithmetic of `binary` or `spin` express variables is made possible using python operators.

```
1 >>> from pyqubo import Binary
2 >>> H = (Binary("a") * Binary("b") - 1)**2
```

6.2 Compilation of Express Instances

6.2.1 From Expression to Model

A `Model` instance is created from an `Express` class using the `compile()` method. `Model` contains information regarding the Hamiltonian of QUBO and Ising representations. Codeblock 3 shows how we compile Eq. (11) from the previous section.

Codeblock 3. Creating a model from a PyQUBO expression.

```
1 >>> from pyqubo import Binary
2 >>> H = (Binary("a") * Binary("b") - 1)**2
3 >>> model = H.compile()
```

6.2.2 From Model to QUBO or Ising

A model's QUBO or Ising formulations can be retrieved as Python dictionaries using the model class' `to_qubo()` and `to_ising()` methods. The `to_qubo()` method returns the QUBO and its energy offset. The QUBO takes the form of `dict[(label, label), value]`, where each label corresponds to a variable. The `to_ising()` method returns the Ising model as two dictionaries, corresponding to linear and quadratic terms, as well as their energy offset. The linear and quadratic outputs take the form `dict[label, value]` and `dict[(label, label), value]`, respectively. Codeblock 4 shows Eq. (11) represented as both a QUBO and an Ising model with energy offsets using the model constructed in Codeblock 3.

Codeblock 4. Converting a model into a QUBO or ising model returns an energy offset as well.

```
4  >>> qubo, qubo_offset = model.to_qubo()
5  >>> qubo
   {('a', 'b'): -1.0, ('b', 'b'): 0, ('a', 'a'): 0}
6  >>> qubo_offset
   1.0
7  >>> linear, quad, ising_offset = model.to_ising()
8  >>> linear, quad
   ({'b': -0.25, 'a': -0.25}, {('a', 'b'): -0.25})
9  >>> ising_offset
   0.75
```

6.2.3 Order Reduction Through Compilation

During compilation, if a Hamiltonian includes k-body interactions among `Binary` or `Spin` variables for $k > 2$, PyQUBO automatically reduces the expression to a quadratic by creating auxiliary variables representing the products of individual spins or binaries.

Reducing the order of the Hamiltonian by hand can be complicated, even for $k = 3$ or three-body interactions. For example, given the Hamiltonian $H = xyz$, where x, y, and z are binary variables, we introduce the auxiliary binary a, which represents the product of x and y. However, to maintain the relationship $a = xy$, we must also include the penalty term $AND(a, x, y) \equiv xy - 2a(x + y) + 3a$ to the Hamiltonian. The final Hamiltonian is $H = az + \alpha AND(a, x, y)$, where α is the penalty strength.

Codeblock 5. `Model.compile()` reduces the degree of an expression if it is greater than two.

```
1  >>> from pyqubo import Binary
2  >>> x, y, z = Binary("x"), Binary("y"), Binary("z")
3  >>> alpha = 2.0
4  >>> model = (x*y*z).compile(strength=alpha)
5  >>> qubo, offset = model.to_qubo()
6  >>> print(qubo)
   {('x', 'y'): 5.0, ('x', 'x*y'): -10.0,
    ('x*y', 'y'): -10.0, ('x*y', 'z'): 1.0,
    ('x', 'x'): 0.0, ('y', 'y'): 0.0,
    ('x*y', 'x*y'): 15.0, ('z', 'z'): 0}
```

Meanwhile, PyQUBO performs this reduction automatically, as demonstrated in Codeblock 5. The QUBO created in line 5 introduces a new variable labeled `'x*y'` representing the product of the individual binary variables.

6.2.4 Decode Solutions

The `Model.decode_sample()` method can interpret the solution from any PyQUBO or quantum annealing solver as an easy-to-read Python dictionary of variable labels and their corresponding values (0 or 1 for `vartype="BINARY"`; −1 or 1 for `vartype="SPIN"`). The function returns a `DecodedSample` object, which provides a dictionary of label-value pairs via the `sample` property, the energies of constraints via the `constraints()` method, and the energy of the solution. We will discuss the `constraints()` method in greater detail in Section 7.1.

Codeblock 6. Decoding a solution.

```
1  >>> decoded_sample = model.decode_sample(sol, vartype="BINARY")
2  >>> decoded_sample.sample
   {'a': 1, 'b': 1}
3  >>> decoded_sample.constraints()
   {'const1': (False, -3.0)}
4  >>> decoded_sample.energy
   -3.0
```

Codeblock 7. We use `spin` variables to distinguish the two sets of integers, which are represented by the coefficient values.

```
1  >>> import neal
2  >>> from pyqubo import Spin, solve_ising
3  >>> s1, s2, s3, s4 = Spin("s1"), Spin("s2"), Spin("s3"), Spin("s4")
4  >>> H = (4*s1 + 2*s2 + 7*s3 + s4)**2
5  >>> model = H.compile()
6  >>> qubo, offset = model.to_qubo()
7  >>> sampler = neal.SimulatedAnnealingSampler()
8  >>> sampleset = sampler.sample_qubo(qubo)
9  >>> decoded_samples = model.decode_sampleset(sampleset, vartype="SPIN")
10 >>> best = min(decoded_samples, key=lambda x: x.energy)
11 >>> best.sample
   {'s1': -1, 's2': -1, 's4': -1, 's3': 1}
```

6.3 The Number Partioning Problem

Using the classes described in Sections 6.1.1 and 6.2, we solve the *number partitioning problem*, which is described as follows. Given a set S of N positive integers, create two disjoint subsets of integers S_1 and S_2 such that their sums are equal. The Ising formulation of the number partitioning problem is

$$H = \left(\sum_{i=1}^{N} n_i s_i \right)^2, \tag{12}$$

where n_i describes the numbers in the set S and s_i is a spin variable. Given that s_i takes the value 1 or −1, the sum of two equally sized sets will be zero for optimal solutions.

Codeblock 7 shows how the Hamiltonian of the number partitioning problem with $S = \{4, 2, 7, 1\}$ can be prepared and solved using PyQUBO.

Here the detailed explanation of Codeblock 7 is given as follows. The sum of the multiplied terms and its exponent are represented using the corresponding Python addition (+), multiplication (∗), and exponent (∗∗) operators, as indicated in line 4. In lines 5 and 6, the Hamiltonian is compiled and converted into a QUBO, and in line 8, a possible solution is identified using the `sample_qubo()` method of `SimulatedAnnealingSampler`. In line 9, `decode_sampleset()` is used to retrieve the spin values corresponding to the labels used in line 3. The solution can easily be

validated by comparing the sums of the coefficients corresponding to the spins with the value 1 or −1. The solution demonstrates one manner in which the given set of integers can be separated into subsets such that their sums are equal. That is, other solutions where the difference is 0 may exist, whereas in other cases, depending on the set of integers provided, none may exist at all.

7 PyQUBO ADVANCED USE

Numerous combinatorial optimization problems can be written and solved using the PyQUBO classes explained in Section 6. Advanced users interested in representing larger or more complicated problems can take advantage of PyQUBO's many other utilities or even define their own Hamiltonian class using the `UserDefinedExpress` class. In this section, we explain the `Constraint`, `Placeholder`, and `Integer` classes of PyQUBO and demonstrate the manner in which these can be used to write more complicated combinatorial optimization problems.

7.1 Constraint

In Section 6.3, we observed that a solution to the number partitioning problem, given a small set S, is easily validated by hand. However, this process becomes difficult and time consuming for larger problems with many auxiliary variables, such as the knapsack problem. The `Constraint` class provides automatic validation regarding whether a solution satisfies the given constraints. The `Constraint` class specifies the parts of a Hamiltonian that must be satisfied by a valid solution to an optimization problem. Each `Constraint` instance takes the section of the Hamiltonian comprising a constraint and string label as its parameters. If a Hamiltonian defined with the `Constraint` class is compiled as a model, the `decode_sample()` function returns a `DecodedSample` object that also provides information about the constraints via the `constraints()` method. The information is represented as a dictionary of constraint labels and tuples containing a boolean value and number, which correspond to whether the constraint is satisfied and the energy of the penalty term, respectively. The `DecodedSample` object also contains the energy of a given solution via the `energy` property, as well as corresponding variable and value pairs in the `sample` property. Codeblock 8 shows a simple application of the `Constraint` class. In line 3, we create a sum of two binary variables with a penalty term that is minimized when one of the variables is 1 and the other is 0. Because the penalty term is wrapped with the `Constraint` class, we are able to use the `decode_sample()` method to check whether a given solution satisfies the constraint represented by the penalty term. Lines 5 to 8 show that when both binary values are 1, `constraints()` method returns a false value with a label `one_hot` indicating the constraint is not satisfied. When the feasible solution $(a = 1, b = 0)$ is given, the `constraints()` method returns a true value indicating the constraint is satisfied.

Codeblock 8. When used with the `constraint` class, `decode_sample()` will return information about the constraints.

```
1  >>> from pyqubo import Binary, Constraint
2  >>> a, b = Binary('a'), Binary('b')
3  >>> exp = a+b+Constraint((a+b-1)**2, label='one_hot')
4  >>> model = exp.compile()
5  >>> decoded_sample = model.decode_sample(
       {'a': 1, 'b': 1}, vartype='BINARY')
6  >>> print(decoded_sample.constraints())
   {'one_hot': (False, 1.0)}
7  >>> decoded_sample = model.decode_sample(
       {'a': 1, 'b': 0}, vartype='BINARY')
8  >>> print(decoded_sample.constraints())
   {'one_hot': (True, 0.0)}
```

7.2 Placeholder

Depending on the given Hamiltonian, the compilation of models can be computationally expensive. If any value in the Hamiltonian is changed, the model must be recompiled in order to calculate the new QUBO. The `Placeholder` class makes it possible to update the constants and coefficients within the Hamiltonian without recompiling the model, thereby saving a significant amount of time in the situation where we need to update some values in Hamiltonian, such as parameter tuning of the penalty strength. Placeholders substitute constants and are identified by their string labels. When creating a QUBO or Ising models, or when decoding a solution, `Placeholder` values must be specified using a Python dictionary of label-value pairs as an additional `feed_dict` parameter.

Codeblock 9 shows how the `Placeholder` can be used after compiling an expression into a `Model` instance. In lines 5 and 7, when converting the `Model` into a QUBO, we include a `feed_dict` specifying the value of `Placeholder` a.

7.3 Integer

The `Integer` class can easily create integer encodings of variables using the sums of `Binary` terms with various encoding types. PyQUBO supports four types of integer encoding classes: `OneHotEncInteger`, `UnaryEncInteger`, `LogEncInteger` and `OrderEncInteger` (Supplementary Section B), available online. All four classes require a label and a tuple of lower bound and upper bound as arguments, and represent values in the range $[lower, upper]$, where $lower$ and $upper$ are the lower and upper bound value, respectively.

Codeblock 9. `Placeholder` values can be updated after compiling a model.

```
1  >>> from pyqubo import Binary, Placeholder
2  >>> x, y, a = Binary('x'), Binary('y'), Placeholder('a'
       )
3  >>> exp = a*x*y + 2.0*x
4  >>> model = exp.compile()
5  >>> qubo_1 = model.to_qubo(feed_dict={'a': 3.0})
6  >>> qubo_1
   ({('x', 'x'): 2.0, ('x', 'y'): 3.0, ('y', 'y'): 0},
       0.0)
7  >>> qubo_2 = model.to_qubo(feed_dict={'a': 5.0})
8  >>> qubo_2
   ({('x', 'x'): 2.0, ('x', 'y'): 5.0, ('y', 'y'): 0},
       0.0)
```

Codeblock 10. While the knapsack problem can be expressed and solved using the classes introduced in section 6 alone, the `constraint`, `placeholder`, `array`, and `integer` classes are useful for both debugging and streamlining the code.

```python
from pyqubo import Binary, Constraint, Placeholder,
    Array, OneHotEncInteger

weights = [1, 3, 7, 9]
values = [10, 2, 3, 6]
max_weight = 10

# create the array of 0-1 binary variables
# representing the selection of the items
n=len(values)
items = Array.create('item', shape=n, vartype="BINARY")

# define the sum of weights and values using variables
knapsack_weight = sum(
    weights[i] * items[i] for i in range(n))
knapsack_value = sum(
    values[i] * items[i] for i in range(n))

# define the coefficients of the penalty terms,
# lmd1 and lmd2, using Placeholder class
# so that we can change their values after compilation
lmd1 = Placeholder("lmd1")
lmd2 = Placeholder("lmd2")

# create Hamiltonian and model
weight_one_hot = OneHotEncInteger("weight_one_hot",
        value_range=(1, max_weight), strength=lmd1)
Ha = Constraint((weight_one_hot - knapsack_weight)**2,
        "weight_constraint")
Hb = knapsack_value
H = lmd2*Ha - Hb
model = H.compile()

# use simulated annealing (SA) sampler of neal package
sampler = neal.SimulatedAnnealingSampler()

feasible_sols = []
# search the best parameters: lmd1 and lmd2
for lmd1_value in range(1, 10):
    for lmd2_value in range(1, 10):

        feed_dict = {'lmd1': lmd1_value, "lmd2":
        lmd2_value}
        qubo, offset = model.to_qubo(feed_dict=
        feed_dict)
        bqm = model.to_bqm(feed_dict=feed_dict)
        bqm.normalize()
        sampleset = sampler.sample(bqm, num_reads=10,
        sweeps=1000, beta_range=(1.0, 50.0))
        dec_samples = model.decode_sampleset(sampleset,
        feed_dict=feed_dict)
        best = min(dec_samples, key=lambda x: x.energy)

        # store the feasible solution
        if not best.constraints(only_broken=True):
            feasible_sols.append(best)

best_feasible = min(feasible_sols, key=lambda x: x.
    energy)
print(f"selection = {[best_feasible.sample[f'item[{i
    }]'] for i in range(n)]}")
print(f"sum of the values = {-best_feasible.energy}")

[output]
selection = [1, 0, 0, 1]
sum of the values = 16.0
```

7.4 The Knapsack Problem

We use the PyQUBO classes described above to create and solve the *knapsack problem*, which is described as follows. Given a set of N items with integer weights and values, determine which items to include in a collection such that the total weight is less than or equal to a weight limit W and the total value V is as large as possible. The total weight and total value of the knapsack is represented by

$$W = \sum_{\alpha=1}^{N} w_\alpha x_\alpha, \quad w_\alpha \in \mathcal{Z} \tag{13}$$

$$V = \sum_{\alpha=1}^{N} v_\alpha x_\alpha, \quad v_\alpha \in \mathcal{Z}, \tag{14}$$

where x_α takes the value 0 or 1 depending on whether the object α is in the knapsack, and w_α and v_α are the integer weight and value of each item α, respectively. The Hamiltonian of the knapsack problem takes the form $H = H_A - H_B$, where H_A represents the weight constraint and H_B is the total value of the items collected. Here, since Ising machines solve minimization problems, the sign of the second term is set to minus. The definitions of H_A and H_B are as follows:

$$H_A = \lambda_1 \left(1 - \sum_{n=1}^{W} y_n \right)^2 + \lambda_2 \left(\sum_{n=1}^{W} n y_n - \sum_{\alpha=1}^{N} w_\alpha x_\alpha \right)^2, \tag{15}$$

$$H_B = \sum_{\alpha=1}^{N} v_\alpha x_\alpha, \tag{16}$$

where y_n takes either the value 1 if the knapsack weight is n or 0 otherwise [21], [23].

Below, we demonstrate how the knapsack problem can be written and solved using PyQUBO.

The detail of Codeblock 10 is as follows. In lines 3-5, we define the weights and values of the items in our set, as well as our weight limit W. In line 10, we use the `Array` class to create the set of `Binary` variables x_α, with its size set equal to the number of items. Thereafter, we calculate the knapsack weight and value sums $\sum_{\alpha=1}^{N} w_\alpha x_\alpha$ and $\sum_{\alpha=1}^{N} v_\alpha x_\alpha$ in lines 13 and 14, respectively.

Next, we prepare the Hamiltonian. In lines 19-20, we define `Placeholder` instances representing the coefficient λ_1 and λ_2 in Eq. (15). In line 23, we create a `OneHotEncInteger`, with λ_1 as its `strength` argument, to represent the integer value $\sum_{n=1}^{W} n y_n$ for the range given by the weight limit (Eq. (15)). In line 24, we take the squared difference between the `Integer` instance and knapsack weight sum, and wrap it with the `Constraint` class. In line 25, H_B is written as the knapsack value sum.

To solve our Hamiltonian, we first compile it into a model instance in line 27. Then, we initialize the `Placeholder` value as a Python dictionary. In lines 34-47, we create a loop in which we search the values of λ_1 and λ_2 so that the feasible solutions can be obtained. In the loop, we create a QUBO as a `BQM` instance using `to_bqm`, normalize the QUBO matrix of `bqm` using `normalize` in order to make it easy to tune the solver parameters, sample solutions through simulated annealing using the `SimulatedAnnealingSampler` of the neal package, and then retrieve the list of `DecodedSample` instances using `decode_sampleset()`. The `BQM` class is defined in the D-Wave's `dimod` package and represents QUBOs or Ising models. The method decodes a `SampleSet` object and returns a `DecodedSample` like `decode_sample()` method. In lines 46-

47, we retrieve the broken constraints using `constraints` with the `only_broken` argument set to true. If the solution is feasible, we append the solution to the list `feasible_-sols`. In line 49, we obtain the feasible solution with the lowest energy. We obtained the final solution to select 1st and 4th items with the value sum equal to 16.

8 PYQUBO FOR LOGICAL GATES

A logical gate is an electronic device that implements a boolean function by taking binary inputs, performing a logic operation, and providing a single binary output.

The AND, OR, NOT, and XOR logic operations can be represented using the logical gate and logical constraint classes of PyQUBO. In the following, we present the corresponding circuit component and truth table of each logic class, as well as examples of its application in PyQUBO.

8.1 Logical Gates

The PyQUBO logical gate classes `Not`, `And`, `Or`, and `Xor` correspond to the four logic operations. Like their electronic analogs, the logical gate classes perform the specified logic operation on given binary inputs.

We summarize the analytical representations of the logic operations below

$$NOT(a) = 1 - a \qquad (17)$$

$$AND(a, b) = ab \qquad (18)$$

$$OR(a, b) = a + b - ab \qquad (19)$$

$$XOR(a, b) = a + b - 2ab, \qquad (20)$$

where a and b represent binary inputs. Fig. 2 shows each operation's circuit element and truth table.

8.2 Logical Constraints

The logical constraint classes `NotConst`, `AndConst`, `OrConst`, and `XorConst` are constraints based on the four operators discussed. However, unlike the logical gates in Section 8.1, each constraint requires three `Express` class inputs (or two in the case of `NotConst`) corresponding to the logical expression operands, as well as the expected output and a label. In other words, the logical constraint classes, instead of performing the specified operation, represents an entire logic expression as a constraint.

As discussed in Section 6.2.4, the `decode_sample()` function determines whether a given solution violates the constraints. Another approach to verifying the solution of an expression wrapped by a logical constraint is to determine the model energy, as indicated in Codeblock 11 using `OrConst`. When the provided variables satisfy the constraint, the energy is 0.0, but when they break the constraint, the energy is 1.0. Therefore, unlike the logical gate classes, logical constraints do not provide solutions themselves.

8.2.1 Preparing a Multi-Bit Binary Multiplier

A digital binary multiplier is an electronic circuit that is used to multiply two binary numbers, namely, a multiplier
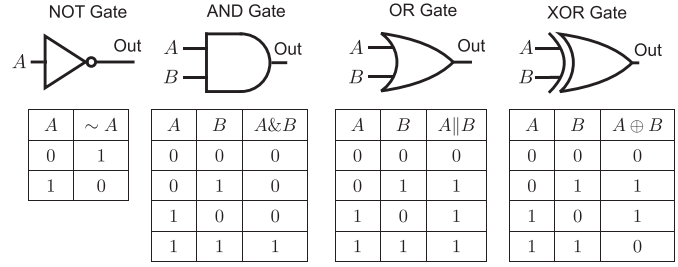

Fig. 2. A summary of logical gate circuit elements and their truth tables.

A	$\sim A$
0	1
1	0

A	B	$A\&B$
0	0	0
0	1	0
1	0	0
1	1	1

A	B	$A\|B$
0	0	0
0	1	1
1	0	1
1	1	1

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

and multiplicand. The bit size of the resulting product corresponds to the sum of the two input bit sizes. Digital multipliers consist of AND gates as well as half- and full-adders. For a j-bit multiplier and k-bit multiplicand, a multiplier circuit requires $j \times k$ AND gates and $(j-1) \times k$ adders, as illustrated in Fig. 3. The logical gates of PyQUBO can be used to represent the AND, OR, and XOR gates that comprise the multiplier as well as its half- and full-adders. By using logical constraints instead of logical gates, the binary multiplier can be adapted to solve integer factoring problems. For example, given a fixed product **p** as a constraint, the multiplier QUBO can be solved for all integer pairs **a** and **b** satisfying the product [28]. To prepare the digital multiplier, first we need to construct the half-adder and full-adder constraints. The half-adder adds two single binary digits and returns their sum and carry values as shown in the circuit diagram in Fig. 4.

Codeblock 11. The energy of a compiled logical constraint expression indicates whether or not the provided operands and output satisfy the operator logic. In this example, the energy is 0.0 when the expression is true and 1.0 when it is false.

```
1 >>> from pyqubo import OrConst, Binary
2 >>> a, b, c = Binary('a'), Binary('b'), Binary('c')
3 >>> exp = OrConst(a, b, c, 'or')
4 >>> model = exp.compile()
5 >>> model.energy({'a': 1, 'b': 0, 'c': 1}, vartype='
    BINARY')
0.0
6 >>> model.energy({'a': 0, 'b': 1, 'c': 0}, vartype='
    BINARY')
1.0
```

Codeblock 12 demonstrates how a half-adder is built using the PyQUBO logical constraint classes.

The detail of Codeblock 12 is as follows. We create a new class `HalfAdderConst` that creates the Hamiltonian for a half-adder given two inputs, as well as the sum and carry. In lines 7 and 8, we define the XOR and AND constraints, which provide the sum and carry, respectively. In line 9, we define the Hamiltonian as the sum of the two constraints. Line 16 uses the `dimod.ExactSolver()` method to calculate all possible solutions of the half-adder Hamiltonian. As seen in Codeblock 11, valid solutions have 0 energy. The full-adder takes three binary values – the two operands A and B and a carry value C_{in} from another adder – and outputs the sum S and C_{out}, as with the half-adder. As indicated in Fig. 5, the full-adder can be constructed from two half-adders and an OR gate. Here, A and B are the inputs of one half-adder, whereas their sum-output and C_{in} are the inputs of the second half-adder. The two half-adder carries
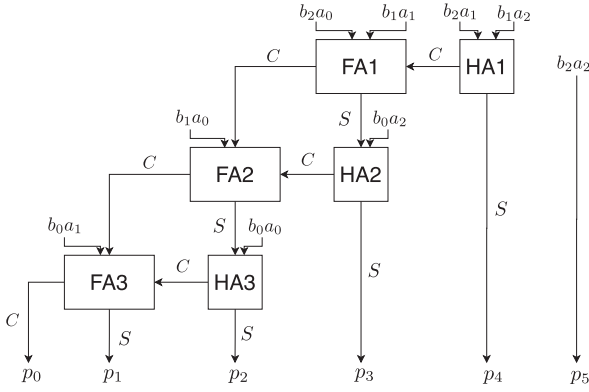
Fig. 3. Diagram for a 3-bit binary multiplier. The arrows labeled C and S refer to the sum and carry of the individual half- and full-adders (labeled HA and FA). The $b_m a_n$ labels represent the mth and nth digit binaries joined by an AND gate.
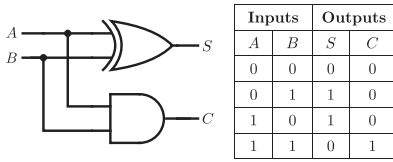

Fig. 4. Half-adder circuit and truth table.

are joined by the OR gate to output C_{out}, and the sum-output of the second half-adder provides S.

A simple 3-bit binary multiplier uses three half-adders, three full-adders, and nine AND constraints. It takes the multiplicand, multiplier, and product as lists or arrays of binary values (as well as a unique string label as a prefix) and creates a Hamiltonian. Users interested in designing a binary multiplier using PyQUBO should refer to the PyQUBO GitHub repository [29].

9 PYQUBO WITH D-WAVE OCEAN

In this section, we demonstrate how PyQUBO and D-Wave can be used in tandem to solve the knapsack problem once again (Eqs. (15) and (16)).

9.1 Introduction to D-Wave

To use a D-Wave machine to solve a given problem, the logical graph representing the corresponding QUBO or Ising model must be embedded into the physical graph of D-Wave hardware. D-Wave machines use the Chimera or Pegasus architecture, depending on the hardware generation [30], [31]. Here, qubit is a spin variable in quantum annealing machines. Embedding a model into the target architecture requires grouping multiple qubits into *chains* that represent single theoretical qubits [17], [18], [19]. In addition to the problem QUBO, chained qubits are assigned the interaction with a constant *chain strength* so that their values are the same across all low-energy solutions. While the magnitude of the chain strength must be tuned depending on the problem, the embedding process allows even complex structures to be mapped onto the D-Wave machine.

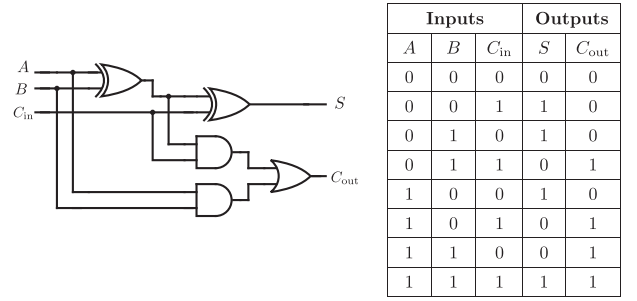D-Wave System solvers are configured to solve problems on corresponding working graphs, which are the qubits and couplers that are available for computation. Here, couplers adjust the value of the interaction.

Fig. 5. Full-adder circuit diagram and truth table, where A, B, and C_{in} are binary inputs.

Codeblock 12. Half-adder built using PyQUBO's logical constraint classes. Note that the outputs s and c must also be included as arguments.

```
1   from pyqubo import XorConst, AndConst, OrConst
2   from pyqubo import Binary, Spin, UserDefinedExpress
3   import dimod

5   class HalfAdderConst(UserDefinedExpress):
6       def __init__(self, a, b, s, c, label):
7           self.xor_const = XorConst(a,b,s,f'{label}_xor')
8           self.and_const = AndConst(a,b,c,f'{label}_and')
9           express = self.and_const + self.xor_const
10          super().__init__(express)

12  a, b, s, c = Binary("a"), Binary("b"), Binary("s"),
        Binary("c")
13  model = HalfAdderConst(a, b, s, c, 'ha').compile()
14  qubo, offset = model.to_qubo()

16  samplset = dimod.ExactSolver().sample_qubo(qubo)
17  for s in model.decode_sampleset(samplset):
18      print(s.sample['a'], s.sample['b'], s.sample['s'],
            s.sample['c'], s.energy)

    # Output
    # 0 0 0 0 0.0
    # 0 1 1 0 0.0
    # 1 0 1 0 0.0
    # 1 1 0 1 0.0
    # 0 1 1 1 1.0
    # 1 1 1 1 1.0
    # ...
```

9.2 Programming With PyQUBO and D-Wave Ocean

A model written in PyQUBO can be used as inputs for the D-Wave Ocean sampler. Samplers provide a sample set of solutions from the low-energy states of the objective function of an optimization problem. Creating a D-Wave sampler requires an endpoint, a D-Wave Application Programming Interface (API) token, which is available to all D-Wave Leap accounts, and a specific solver name. Based on Codeblock 13, a method for solving a knapsack problem using the D-Wave sampler is specifically described. In lines 13-24, we define the Hamiltonian of the knapsack problem. Here, we use `LogEncInteger` instead of `OneHotEncInteger` used in Codeblock 10 to show the integer class object can be easily replaced. In line 26, we create our sampler using the default API endpoint URL, our account API token, and the solver *Advantage_system1.1*. In line 38, we create our embedding, which we map onto the quantum annealing machine using the `FixedEmbeddingComposite` class in line 40. Defining our embedding before the QUBO saves time, however finding the optimal embedding could in turn lead to
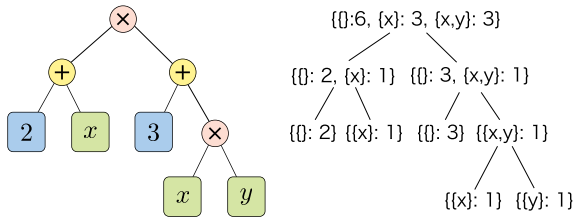
Fig. 6. (Left) The binary tree of the expression defined in the Codeblock 14. (Right) The generated polynomials at each node. The polynomials are shown as a hash map.

more efficient problem solving. In lines 42-49, we define the sampler's keyword arguments, which depend on the selected D-Wave solver. The following parameters are common across all hardware solvers: `num_reads` and `annealing_time`, which correspond to the number of anneals and time per anneal respectively; `num_spin_reversal_transforms`, which sets the number of gauge transformations to be performed on the problem [32]; and `auto_scale`, which indicates whether h_i and J_{ij} of the Ising model (Eq. (4)) are rescaled.

The two parameters `chain_strength` and `chain_break_fraction` in lines 47 and 48 belong to the `FixedEmbeddingComposite` class, which maps problems to the sampler using the given embedding. As discussed above, the `chain_strength` parameter must be tuned for the problem.

Next, we create an objective function, which provides the solution to a QUBO, its energy, as well as broken constraints, given a feed dictionary for `Placeholder` values and the above-mentioned keyword arguments. In line 52, we create a QUBO as `BQM` from a `Model` instance. In line 53, we normalize the QUBO such that the parameter tuning is not affected by the scale of the problem. In line 54, we use the sampler to retrieve a set of solutions to the normalized QUBO, and in line 55, we use the PyQUBO function `Model.decode_sampleset()` to interpret these solutions. The decode method's arguments are `sampleset`, the solution from a sampler; and `feed_dict`, which, as before, is a dictionary of placeholder key-value pairs. In line 56, we return the `DecodedSample` object with the lowest energy.

Finally, in lines 59-64, we execute the objective function using different `Placeholder` values and append each feasible solution to the list `feasible_sols`. In this code, we search only one parameter `lmd` since we are using `LogEncInteger`, which does not have an extra constraint like `OneHotEncInteger`. Finally, we show the sum of the values of the best feasible solution.

10 IMPLEMENTATION AND BENCHMARKING

In this section, we show how PyQUBO is implemented internally and benchmark the performance with different implementations including other packages.

10.1 Internal Representation of Expressions

In PyQUBO, the expression of a Hamiltonian is represented by a binary tree, which is called AST (Abstract Syntax Tree). For example, the expression created by Codeblock 14 is represented by the binary tree shown in Fig. 6 (left). The leaves of the tree are composed of "number" and "variable" nodes, shown as rectangles in Fig. 6 (left). The internal nodes are

composed of "sum" or "product" nodes, shown as circles in Fig. 6 (left).

Codeblock 13. Using the D-wave sampler to find a solution to the knapsack problem.

```
1   import dimod
2   from dwave.system.samplers import DWaveSampler
3   from dwave.system.composites import
        FixedEmbeddingComposite
4   from minorminer.busclique import find_clique_embedding
5   import dwave_networkx as dnx
6   from pyqubo import Binary, Constraint, Placeholder,
        Array, LogEncInteger

8   # weights, values and the maximum weight of the
        knapsack problem
9   weights = [1, 3, 7, 9]
10  values = [10, 2, 3, 6]
11  max_weight = 10

13  n=len(values)
14  items = Array.create('item', shape=n, vartype="BINARY")
15  knapsack_weight = sum(weights[i] * items[i] for i in
        range(n))
16  knapsack_value = sum(values[i] * items[i] for i in
        range(n))

18  # create Hamiltonian and model
19  weight_one_hot = LogEncInteger("weight_one_hot",
        value_range=(1, max_weight))
20  Ha = Constraint((weight_one_hot - knapsack_weight)**2,
        "weight_constraint")
21  Hb = knapsack_value
22  lmd = Placeholder("lmd")
23  H = lmd*Ha - Hb
24  model = H.compile()

26  dw_sampler = DWaveSampler(
27      endpoint="https://cloud.dwavesys.com/sapi",
28      token="your-token",
29      solver="Advantage_system1.1")

31  graph_size=16
32  sampler_size=len(model.variables)
33  p16_working_graph = dnx.pegasus_graph(
34                          graph_size,
35                          node_list=dw_sampler.nodelist,
36                          edge_list=dw_sampler.edgelist)

38  embedding = find_clique_embedding(sampler_size,
        p16_working_graph)

40  sampler = FixedEmbeddingComposite(dw_sampler, embedding
        )

42  sampler_kwargs = {
43      "num_reads": 100,
44      "annealing_time": 20,
45      "num_spin_reversal_transforms": 4,
46      "auto_scale": True,
47      "chain_strength": 2.0,
48      "chain_break_fraction": True
49  }

51  def objective(feed_dict):
52      bqm = model.to_bqm(index_label=True, feed_dict=
            feed_dict)
53      bqm.normalize()
54      sampleset = sampler.sample(bqm, **sampler_kwargs)
55      dec_samples = model.decode_sampleset(sampleset,
            feed_dict=feed_dict)
56      return min(dec_samples, key=lambda x: x.energy)

58  # search best parameters lmd within [1,2,...,5]
59  feasible_sols = []
60  for lmd_value in range(1, 5):
61      feed_dict = {'lmd': lmd_value}
62      s = objective(feed_dict)
63      if not s.constraints(only_broken=True):
64          feasible_sols.append(s)

66  best_feasible = min(feasible_sols, key=lambda x: x.
        energy)
67  print(f"selection = {[best_feasible.sample[f'item[{i
        }]'] for i in range(n)]}")
68  print(f"sum of value = {-best_feasible.energy}")

[output]
selection = [1, 0, 0, 1]
sum of value = 16.0
```

10.2 Compilation (QUBO Creation)

We define compilation in PyQUBO as the process to produce quadratic polynomials from the expression of the Hamiltonian. The compilation is composed of the following two steps.

- Step 1: Expanding an expression into a polynomial.
- Step 2: Reducing the order of the polynomial obtained in the step 1.

We can easily implement order reduction by just replacing the pair of variables with the new auxiliary variable, as explained in Section 6.2.3. In the following sections, we mainly explain the expansion of the expression (i.e., step 1).

In the compilation process, the quadratic polynomial with binary variables corresponding to the QUBO matrix is produced from the binary tree representing the expression. In PyQUBO, polynomials are represented by a "hash map" with products of binary variables as keys and coefficients as values. Products of variables can be represented by "set" since we only deal with 0-1 binary variables. For example, we can confirm xy^2z is represented by the set $\{x, y, z\}$ by using the relationship $xy^2z = xyz$, $x, y, z \in \{0, 1\}$.

When we represent the set with elements A, B as $\{A, B\}$ and the hash map with key k and value v as $\{k : v\}$, the polynomial $a + bx_0 + cx_0x_1$ can be written as $\{\{\} : a, \{x_0\} : b, \{x_0, x_1\} : c\}$, where a, b and c are coefficients, x_0 and x_1 are 0-1 binary variables, and $\{\}$ is an empty set. The Python-like pseudo code to expand the expression is shown in Codeblock 15. If we pass the root node object of the binary tree to the function, the expanded polynomial is returned as a hash map. The functions `poly_sum`, `poly_prod` calculate the sum and the product of the input polynomials, respectively. We assume that the input node has the property `type`, which indicates the type of the node (i.e., number, variable, sum or product). The "product" or "sum" node has the properties `left` and `right`, each of which contains the child node corresponding to the inputs of the product or sum operation. The "number" node has the property `value` which contains the number itself. The generated polynomials by `expand()` function at each node, are shown in Fig. 6 (right).

Codeblock 14. An example expression created by PyQUBO objects. It corresponds to the binary tree in Fig. 6 (left).

```
1  x, y = Binary('x'), Binary('y')
2  H = (x + 2)*(x*y + 3)
```

10.3 Data Structure of Product of Variables

The products of variables in polynomials are represented by a set. In calculating the product of polynomials (i.e., `poly_prod()` function in Codeblock 15), we need to calculate the product of two products of variables. This operation can be implemented by the union operation of the two input sets representing the product of variables. In the following example, we calculate the product of xy and yz.

$$xy \times yz = xyz, \ x, y, z \in \{0, 1\}. \tag{21}$$

We can confirm that the set $\{x, y, z\}$ is a union of the two sets: $\{x, y\}$ and $\{y, z\}$. Since a polynomial is implemented as

a hash map, the product of variables need to work as a key of the hash map. This means that the set must implement the equal function and the hash function. We summarize the required implementations for a set representing products of variables.

- Constructing a new set object from an object.
  Ex) `setA ← newSet(a)`.
- Union operation to calculate the product.
- Equality operation to work as a key of hash map.

Using a profiling tool, we observed that the equal function is called the most frequently among the operations above. Let us consider the appropriate implementation of the set. First, we considered using the set class provided by the C++ standard library, i.e., tree-based set (`std::set`) and the hash-based set (`std::unordered_set`). The time complexity of equality operation is $O(k)$ for the tree-based set [33] and $O(k^2)$ for the hash-based set [34] in the worst case, where k is the size of the set. Therefore, the tree-based set is suitable in the case where the equality operation needs to run fast.

Codeblock 15. The python-like pseudo code to expand the expression.

```
1   def expand(node):
2     if node.type=='sum':
3       return poly_sum(
4         expand(node.left), expand(node.right))
5     elif node.type=='product':
6       return poly_prod(
7         expand(node.left), expand(node.right))
8     elif node.type=='variable':
9       return {{var.label}: 1}
10    elif node.type=='number':
11      return {{}: node.value}
```

Second, we considered using a sorted array to represent a set. By comparing elements from the head of the sorted array, we found that the time complexity of the equality operation is $O(k)$ in the worst case. By merging elements from the head of the array, we discovered that the time complexity of the union operation is $O(k)$ in the worst case[35]. Since the time complexity of the union operation of the tree-based set is $O(k\log k)$, the union operation of the sorted array is expected to be faster than that of the tree-based set.

10.4 Benchmark of Memory Size and Running Time

We measured the memory size and the running time required to construct the expression and the QUBO matrix with a couple of types of combinatorial problems: graph partition problems (GP) and traveling salesman problems (TSP) [21]. Graphs used in GP are generated as a binomial graph with the edge density set to 0.3. While the QUBO matrix of GP is dense, i.e., all elements of the matrix are non-zero, the QUBO matrix of TSP contains about n_c^3 non-zero elements out of n_c^4, where n_c is the number of cities in TSP. We chose these two problems for the benchmark since each QUBO matrix has different characteristics in terms of the density. In the measurement of the memory size, we measured the maximum memory size for all processes (i.e., from constructing the expression through producing the QUBO matrix). We measured the running time to construct the expression and produce the QUBO matrix separately. We compared the following implementations in this benchmarking.
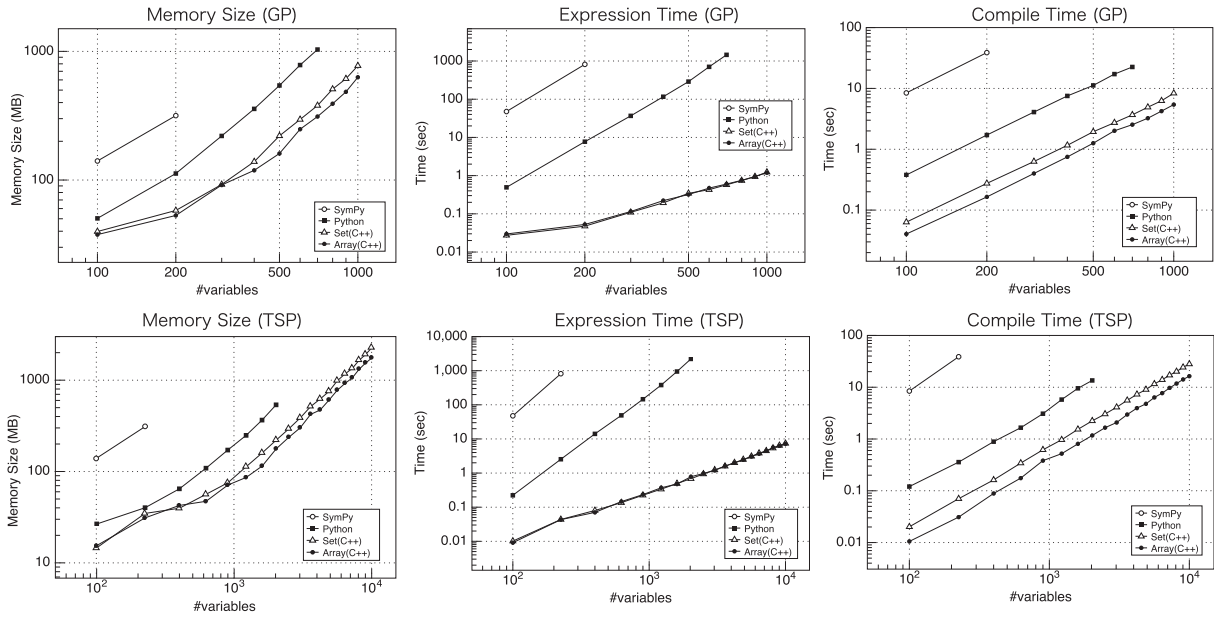
Fig. 7. Dependence of the memory size and the running time on the number of variables in QUBOs for graph partition problems (GP) and traveling salesman problems (TSP). The expression time is the running time to construct the expression, and the compile time is the running time to create the model object from the expression and produce a QUBO matrix by using the `to_qubo()` method.

- *SymPy:* Using SymPy [36] package to create QUBOs from the expression.
- *Python:* Older PyQUBO (version 0.4.0) implemented entirely in Python.
- *Set(C++):* Using tree-based set (`std::set`) for products of variables, implemented in C++.
- *Array(C++):* Using sorted array for products of variables, implemented in C++. Equivalent to PyQUBO (version 1.0.7).

SymPy [36] is a well-used general symbolic tool. We used the `expand()` method of symbol objects in SymPy to create a QUBO matrix. While PyQUBO (version 0.4.0) is implemented entirely in Python, PyQUBO (version 1.0.7) is internally implemented in C++11. We compared two implementations of the products of variables in C++, the tree-based set `std::set` and the sorted array. We ran our experiments on Mac OSX 10.15 and Intel Core i7 1.7 GHz with 16 GB memory. We used SymPy version 1.1.1 for our calculation. We set the time limit as 3500 seconds, and stopped calculations whose running time exceeded this duration.

10.5 Result of Benchmark

We show the dependence of memory size and running time on the number of variables in a QUBO (Fig. 7). We could not run SymPy and Python with larger problem sizes because of the time limit. We observed that the memory size and the running time of PyQUBO in C++ are better than that of PyQUBO in Python or SymPy. We also confirmed that the memory size and the compile time of Array(C++) are superior compared to Set(C++). In graph partition problems and traveling salesman problems, the number of terms n is m^2 and $m^{3/2}$, respectively, where m is the number of variables of QUBO. Based on the benchmark results, we can estimate the time complexity with respect to the number of terms n in the Hamiltonian as shown in Table 1. While the time complexity of constructing expressions in SymPy and Python is $O(n^2)$, it

TABLE 1
The Estimated Time Complexity With Respect to the Number of Terms n in the Hamiltonian, Based on the Benchmark Result (Fig. 7)

	Expression Time	Compile Time
SymPy	$O(n^2)$	$O(n)$
Python	$O(n^2)$	$O(n)$
Set(C++)	$O(n)$	$O(n)$
Array(C++)	$O(n)$	$O(n)$

is reduced to $O(n)$ in C++. Since the Python implementation requires that the sum of expressions is stored as a list, it creates a copy of the entire list when we add a new term to the expression, which leads to the complexity of $O(n^2)$. The time complexity of constructing an expression in C++ is $O(n)$ because the expression is represented by binary trees like Fig. 6 (left), i.e., an added term is just referenced from the original expression, which does not require objects to be copied when we add a new term.

11 CONCLUSION

The increasing availability of Ising machines including quantum annealing machines has been an advantage for research into practical solutions to large combinatorial optimization problems. However, Ising machines are limited to solving QUBOs and Ising model Hamiltonians, which are difficult to create and implement for many, if not most, optimization problems. The PyQUBO package offers a means of writing QUBOs and Ising model Hamiltonians in an intuitive and readable manner. Not only does it accommodate a wide variety of problems, but it also supports the automatic validation of constraints and parameter tuning, which are essential for debugging large problems. Furthermore, as an open-source library, it enables users to create and contribute new functions that are suited to their needs. We expect that

many researchers will use PyQUBO as they integrate Ising machines including quantum annealing into their fields.

ACKNOWLEDGMENTS

REFERENCES

[1] M. W. Johnson et al., "Quantum annealing with manufactured spins," Nature, vol. 473, no. 7346, pp. 194–198, 2011.
[2] T. Kadowaki and H. Nishimori, "Quantum annealing in the transverse Ising model," Phys. Rev. E, vol. 58, no. 5, 1998, Art. no. 5355.
[3] E. Farhi, J. Goldstone, S. Gutmann, and M. Sipser, "Quantum computation by adiabatic evolution," 2000, quant-ph/0001106.
[4] E. Farhi, J. Goldstone, S. Gutmann, J. Lapan, A. Lundgren, and D. Preda, "A quantum adiabatic evolution algorithm applied to random instances of an NP-complete problem," Science, vol. 292, no. 5516, pp. 472–475, 2001.
[5] M. Yamaoka, C. Yoshimura, M. Hayashi, T. Okuyama, H. Aoki, and H. Mizuno, "A 20k-spin Ising chip to solve combinatorial optimization problems with CMOS annealing," IEEE J. Solid-State Circuits, vol. 51, no. 1, pp. 303–309, Jan. 2016.
[6] T. Inagakietal, "A coherent Ising machine for 2000-node optimization problems," Science, vol. 354, no. 6312, pp. 603–606, 2016.
[7] P. L. McMahon et al., "A fully programmable 100-spin coherent Ising machine with all-to-all connections," Science, vol. 354, no. 6312, pp. 614–617, 2016.
[8] M. Aramon, G. Rosenberg, E. Valiante, T. Miyazawa, H. Tamura, and H. Katzgrabeer, "Physics-inspired optimization for quadratic unconstrained problems using a digital annealer," Front. Phys., vol. 7, 2019, Art. no. 48.
[9] M. Maezawa et al., "Toward practical-scale quantum annealing machine for prime factoring," J. Phys. Soc. Japan, vol. 88, no. 6, 2019, Art. no. 061012.
[10] H. Goto, K. Tatsumura, and A. R. Dixon, "Combinatorial optimization by simulating adiabatic bifurcations in nonlinear hamiltonian systems," Sci. Adv., vol. 5, no. 4, 2019, Art. no. eaav2372.
[11] G. Rosenberg, P. Haghnegahdar, P. Goddard, P. Carr, K. Wu, and M. L. De Prado, "Solving the optimal trading trajectory problem using a quantum annealer," IEEE J. Sel. Top. Signal Process., vol. 10, no. 6, pp. 1053–1060, 2016.
[12] F. Neukart, G. Compostella, C. Seidel, D. von Dollen, S. Yarkoni, and B. Parney, "Traffic flow optimization using a quantum annealer," Front. ICT, vol. 4, 2017, Art. no. 29.
[13] K. Terada et al., "An Ising model mapping to solve rectangle packing problem," in Proc. Int. Symp. VLSI Des. Autom. Test, 2018, pp. 1–4.
[14] N. Nishimura, K. Tanahashi, K. Suganuma, M. J. Miyama, and M. Ohzeki, "Item listing optimization for e-commerce websites based on diversity," Front. Comput. Sci., vol. 1, 2019, Art. no. 2.
[15] K. Kitai et al., "Designing metamaterials with quantum annealing and factorization machines," Phys. Rev. Res., vol. 2, no. 1, 2020, Art. no. 013319.
[16] S. Tanaka, Y. Matsuda, and N. Togawa, "Theory of Ising machines and a common software platform for Ising machines," in Proc. 25th Asia South Pacific Des. Autom. Conf., 2020, pp. 659–666.
[17] J. Cai, W. G. Macready, and A. Roy, "A practical heuristic for finding graph minors," 2014, arXiv:1406.2741.
[18] V. Choi, "Minor-embedding in adiabatic quantum computation: I. The parameter setting problem," Quantum Inf. Process., vol. 7, no. 5, pp. 193–209, 2008.
[19] T. Boothby, A. D. King, and A. Roy, "Fast clique minor generation in chimera qubit connectivity graphs," Quantum Inf. Process., vol. 15, no. 1, pp. 495–508, 2016.
[20] S. Kanamaru et al., "Mapping constrained slot-placement problems to Ising models and its evaluations by an Ising machine," in Proc. IEEE 9th Int. Conf. Consum. Electron., 2019, pp. 221–226.
[21] A. Lucas, "Ising formulations of many NP problems," Front. Phys., vol. 2, 2014, Art. no. 5.
[22] S. Tanaka, R. Tamura, and B. K. Chakrabarti, Quantum Spin Glasses, Annealing and Computation. Cambridge, U.K.: Cambridge Univ. Press, May 2017.
[23] K. Tanahashi, S. Takayanagi, T. Motohashi, and S. Tanaka, "Application of Ising machines and a software development for Ising machines," J. Phys. Soc. Japan, vol. 88, no. 6, 2019, Art. no. 061010.
[24] L. A. Wolsey and G. L. Nemhauser, Integer and Combinatorial Optimization, vol. 55. Hoboken, NJ, USA: Wiley, 1999.
[25] Gurobi Optimization, LLC, "Gurobi optimizer reference manual," 2020. [Online]. Available: http://www.gurobi.com/
[26] D. O'Malley and V. V. Vesselinov, "ToQ. JL: A high-level programming language for D-wave machines based on julia," in Proc. IEEE High Perform. Extreme Comput. Conf., 2016, pp. 1–7.
[27] Jij Inc., 2019. [Online]. Available: https://github.com/OpenJij/OpenJij
[28] M. Maezawa, K. Imafuku, M. Hidaka, H. Koike, and S. Kawabata, "Design of quantum annealing machine for prime factoring," in Proc. 16th Int. Superconductive Electron. Conf., 2017, pp. 1–3.
[29] Recruit Communications Co., Ltd., "Pyqubo," 2018. [Online]. Available: https://github.com/recruit-communications/pyqubo
[30] T. Albash, W. Vinci, A. Mishra, P. A. Warburton, and D. A. Lidar, "Consistency tests of classical and quantum models for a quantum annealer," Phys. Rev. A, vol. 91, no. 4, 2015, Art. no. 042314.
[31] D-Wave Systems Inc., "D-wave QPU architecture: Topologies," 2020. [Online]. Available: https://docs.dwavesys.com/docs/latest/c_gs_4.html
[32] S. Boixo et al., "Evidence for quantum annealing with more than one hundred qubits," Nat. Phys., vol. 10, no. 3, pp. 218–224, 2014.
[33] operator==,!=(std::set), 2020. [Online]. Available: https://en.cppreference.com/w/cpp/container/set/operator_cmp
[34] operator==,!=(std::unordered_set), 2020. [Online]. Available: https://en.cppreference.com/w/cpp/container/unordered_set/operator_cmp
[35] std::merge, 2020. [Online]. Available: https://en.cppreference.com/w/cpp/algorithm/merge
[36] A. Meurer et al., "SymPy: Symbolic computing in python," PeerJ Comput. Sci., vol. 3, 2017, Art. no. e103. [Online]. Available: https://doi.org/10.7717/peerj-cs.103

Mashiyat Zaman received the BA degree from Amherst College, Amherst, Massachusetts, in 2018. He is currently a data engineer at Recruit Co., Ltd., Japan.

Kotaro Tanahashi received the MEng degree from Kyoto University, Japan, in 2015. He currently works for Recruit Co., Ltd., Japan as a machine learning engineer. He is also a project manager of MITOU Target Program at Information-technology Promotion Agency (IPA).

Shu Tanaka received the Doctor of Science degree from The University of Tokyo, Japan, in 2008. He is presently an associate professor with the Department of Applied Physics and Physico-Informatics, Keio University, Japan, and a visiting associate professor in Green Computing Systems Research Organization, Waseda University, Japan. His research interests are quantum annealing, Ising machine, statistical mechanics, and materials science. He is a member of JPS.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.