Coding Nuggets
Faster QUBO Brute-Force Solving

Sascha Mücke*
Lamarr Institute
TU Dortmund University
Dortmund, Germany

ABSTRACT

This article describes an improved brute-force solving strategy for Quadratic Unconstrained Binary Optimization (QUBO) problems that is faster than naive approaches and easily parallelizable. The implementation in Python is discussed in detail, and an additional C implementation is provided.

1 INTRODUCTION

Quadratic Unconstrained Binary Optimization (QUBO) is the problem of finding a binary vector $\boldsymbol{x}^* \in \{0,1\}^n$ that minimizes the function

$$f_{\boldsymbol{Q}}(\boldsymbol{x}) \coloneqq \boldsymbol{x}^\top \boldsymbol{Q}\boldsymbol{x} = \sum_{\substack{i,j\in[n]\\ i\leq j}} Q_{ij}x_ix_j \;, \tag{1}$$

where $\boldsymbol{Q} \in \mathbb{R}^{n\times n}$ is an upper triangular matrix, and $[n]$ denotes the set $\{1,\dots,n\}$. It is an NP-hard optimization problem [16] with numerous applications, ranging from economics [9, 13] over satisfiability [11] and resource allocation [15, 17] to Machine Learning [1–3, 6, 14], among others. In recent years it has gained renewed attention because it is equivalent to the Ising model, which can be solved physically through quantum annealing [8, 10], for which specialized quantum computers have been developed [5].

Aside from quantum annealing, QUBO can be solved—exactly or approximately—with a wide range of optimization strategies. A comprehensive list of approaches can be found in [12].

The most straightforward way to solve QUBO is using brute force. For large n, this strategy quickly becomes infeasible, as the number of binary vectors grows exponentially in n. However, for n up to about 30, which is roughly the order of magnitude currently solvable (approximately) with the hybrid quantum algorithm QAOA [7], brute force is an easy-to-implement and reliable way to obtain the minimal bit vector.

In this article I describe a technique to reduce the computational cost of brute-force QUBO solving by a factor of roughly n compared to the naive approach. To this end I use the concept of Gray codes to traverse the space $\{0,1\}^n \coloneqq \{0,1\}^n$ in a way that allows for the value of $f_{\boldsymbol{Q}}$ to be updated incrementally, without evaluating all $n\cdot(n+1)/2$ entries of $\boldsymbol{Q}$. A Python implementation using NumPy is developed in Section 3, and ways to improve running time by using just-in-time compilation and parallelization are described.

2 THEORY

Generally, computing the value of $f_{\boldsymbol{Q}}(\boldsymbol{x})$ requires evaluating a sum over all entries of $\boldsymbol{Q}$. However, if the value of $\boldsymbol{x}$ is already known to be v, and we want to compute the value of $\tilde{\boldsymbol{x}}$ that differs from $\boldsymbol{x}$ in only a single bit, we can "update" v to obtain $f_{\boldsymbol{Q}}(\tilde{\boldsymbol{x}})$, which requires only n values of $\boldsymbol{Q}$ (see Section 2.1).

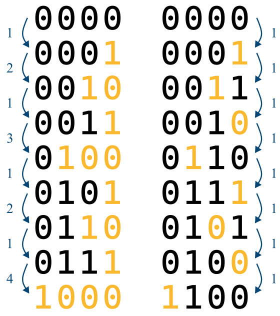

Figure 1: Gray Code

Now all that is missing is a way to traverse all vectors in $\{0,1\}^n$ by changing only one bit at a time. Luckily, we can take inspiration from Gray codes, which have exactly the right properties for this task and are easy to compute (see Section 2.2).

2.1 Updating QUBO values

Let $\boldsymbol{Q} \in \{0,1\}^{n\times n}$ be a fixed QUBO matrix, and $\boldsymbol{x} \in \{0,1\}^n$ a binary vector. Further, let $v = f_{\boldsymbol{Q}}(\boldsymbol{x})$. Now, assume we want to flip the ℓ-th bit of $\boldsymbol{x}$ to obtain $\tilde{\boldsymbol{x}}$ and calculate the new value $\tilde{v} = f_{\boldsymbol{Q}}(\tilde{\boldsymbol{x}})$. Instead of calculating $\tilde{\boldsymbol{x}}^\top\boldsymbol{Q}\tilde{\boldsymbol{x}}$ explicitly, we can look at the difference between v and $\tilde{v}$:

$$\begin{aligned} & \tilde{v} = v + \Delta_\ell \\ \Leftrightarrow\; & \Delta_\ell = \tilde{v} - v \\ & \quad = \sum_{i\leq j} Q_{ij}(\tilde{x}_i\tilde{x}_j - x_ix_j) \\ & \quad = s_\ell\left(\sum_{i=1}^{\ell-1} Q_{i\ell}x_i + Q_{\ell\ell} + \sum_{j=\ell+1}^{n} Q_{\ell j}x_j\right), \end{aligned} \tag{2}$$

where $s_\ell = \tilde{x}_\ell - x_\ell \in \{-1,+1\}$.

As we can see from the last line, we only need to read the elements in the ℓ-th row and column of $\boldsymbol{Q}$ to calculate Δ_ℓ. This reduces the computational cost from $O(n^2)$ to $O(n)$ per evaluation of $f_{\boldsymbol{Q}}$. As the vector $\mathbf{0} \coloneqq (0,\dots,0)^\top$ always has value 0 for any $\boldsymbol{Q}$, we can start there and never need to fully calculate Eq. (1), if we find a way to successively flip bits and traverse all $\boldsymbol{x} \in \{0,1\}^n$. This problem is addressed next.

[*] 0000-0001-8332-6169

Algorithm 1 Improved QUBO Brute-Force Solving

```
$\boldsymbol{x} \leftarrow \mathbf{0}$
$v \leftarrow 0$
$v^* \leftarrow \infty$
$i \leftarrow 1$
while $i < 2^n$ do
    $\ell \leftarrow \mathrm{CTZ}(i)$
    $x_\ell \leftarrow 1 - x_\ell$
    $\Delta_\ell \leftarrow \sum_{i=1}^{\ell-1} Q_{i\ell} x_i + Q_{\ell\ell} + \sum_{j=\ell+1}^{n} Q_{\ell j} x_j$
    $v \leftarrow v + (2x_\ell - 1)\Delta_\ell$
    if $v < v^*$ then
        $\boldsymbol{x}^* \leftarrow \boldsymbol{x}$ // memorize minimizing $\boldsymbol{x}$…
        $v^* \leftarrow v$ // …and its value
    $i \leftarrow i + 1$
return $\boldsymbol{x}^*, v^*$
```

2.2 Gray Code

When starting at 0 and counting up, the respective binary representations of successive numbers often differ in more than one bit (see Fig. 1, left). For example, to get from $2^k - 1$ to 2^k for any $k \geq 0$, exactly $k + 1$ bits must be flipped. These transitions where many bits flip at once are nown as *Hamming cliffs* [4]. Therefore, this way of traversing all bitvectors cannot be used with the aforementioned updating method.

Gray code is an ordering of the natural numbers $\pi : \mathbb{N}_0 \longrightarrow \mathbb{N}_0$ that removes Hamming cliffs, i.e., for any $k \in \mathbb{N}_0$ we find that $\pi(k)$ and $\pi(k+1)$ differ in one bit when represented in binary (see Fig. 1, right). The sequence $\pi(0), \dots, \pi(2^k - 1)$ for any $k > 1$ can be constructed recursively in binary from the sequence for $k - 1$ through

$$\pi(\ell) \coloneqq \begin{cases} \mathtt{0} \cdot \pi(\ell) & \text{if } \ell < 2^{k-1} \\ \mathtt{1} \cdot \pi(2^k - \ell - 1) & \text{else} \end{cases} \tag{3}$$

Here, $\cdot$ denotes string concatenation. For $k = 1$ the code is just $(\mathtt{0}, \mathtt{1})$.

As only one bit is flipped at a time, we can express π equivalently as the sequence of bit indices to flip, starting at $\mathbf{0}$ and using 0-indexing. This sequence turns out to be $0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, \dots$, which is known as the *binary carry sequence*[1]. Interestingly, it coincides with the number of trailing zeros when writing the natural numbers $1, 2, 3, \dots$ in binary, which gives us a very efficient way to calculate its terms on a CPU (see Section 3). The number of trailing zeros of a number k is denoted by $\mathrm{CTZ}(k)$.

Now that we have all necessary ingredients for a more efficient brute-force strategy, we can combine everything into Algorithm 1. In the following section, I show how to put it into practice by implementing it in Python, using *NumPy* and *Numba* to improve the running time, which is why we need to

```
import numpy as np
from numba import njit
```

3 PRACTICE

For comparison, two versions of brute-force QUBO solving methods will be implemented here: A naive approach and the improved version described in Algorithm 1.

Assume that the parameters of the QUBO problem instance are given as an upper-triangular *NumPy* matrix `Q` of shape `(n, n)`. For the naive approach, we will simply loop over all binary vectors in “ascending order”, i.e., we count up from 0 through $2^n - 1$ and convert each number to its binary representation. This can be done somewhat efficiently by using NumPy’s broadcasting functionality: Assume we want to represent an integer `k` as a binary vector of length `n`. We can write:

```
places = 2**np.arange(n)
x = (k & places) > 0
```

The operator `&` performs an element-wise logical AND on the binary representation of `k` and each element in `k`. If the i-th bit of `k` is set, `k & 2**i` equals `2**i`, otherwise it evaluates to `0`. Thus, to make the result binary, we can simply check if it is greater than zero. Doing this for every bit index i and collecting the result in an array yields the binary representation of `k`.

To evaluate the QUBO objective function Eq. (1), we can simply perform the twofold matrix-vector product as

```
    v = x @ Q @ x
```

All that is missing is the loop over all 2^n vectors, and a running minimality check. In summary, the naive brute-force algorithm could look as follows:

```
def naive_brute_force(Q):
    n = Q.shape[0]
    # initialize bit vector and value
    x = np.zeros(n)
    v = 0
    # initialize minimal bit vector and value
    x_min = np.zeros(n)
    v_min = 0

    places = 2 ** np.arange(n) # can be outside loop
    for k in range(1, 2**n):
        x[:] = (k & places) > 0 # get binary vector from k
        v = x @ Q @ x     # get QUBO objective value
        if v < v_min:     # check for minimality
            x_min[:] = x # memorize binary vector..
            v_min = v     # ..and value
    return x_min, v_min
```

Notice that we can start the loop at 1 and directly set `x_min` and `v_min` to 0, as $f_{\boldsymbol{Q}}(\mathbf{0}) = 0$ regardless of $\boldsymbol{Q}$.

Now let us implement the improved version described in Algorithm 1. There are only few changes compared with the naive version:

1. Update $\boldsymbol{x}$ incrementally in Gray code order
2. Update function value v incrementally.

To address the first point, we need to implement the CTZ function. In Python, we have a builtin method `int.bit_count()`, which counts the 1 bits in the binary representation of an integer. We can use this function to determine CTZ of a number `k` by

```
(k ^ (k-1)).bit_count()-1
```

Intuitively, if a number has ℓ trailing zeros, then subtracting 1 yields a number with ℓ trailing ones and a zero in $(\ell + 1)$-th place. The XOR (`^`) of `k` and `k-1` thus consists of $\ell + 1$ ones, from which we

[1] https://oeis.org/A007814

need to subtract 1 to get just ℓ. This is the index we need to flip $\boldsymbol{x}$ at:

```
x[l] = 1-x[l]
```

The second point can be implemented by first making the (triangular) QUBO matrix symmetric, which lets us simply read one row instead of both row and column. Further, we save the diagonal containing the linear terms in a separate array:

```
qua  = np.triu(Q, 1) # clear diagonal
qua += qua.T         # make symmetric
lin  = np.diag(Q)
```

This lets us write Eq. (2) as

```
delta = (2*x[l]-1) * (qua[l]@x + lin[l])
```

and the complete implementation reads as follows:

```
def improved_brute_force(Q):
    n = Q.shape[0]
    # initialize bit vector and value
    x = np.zeros(n)
    v = 0
    # initialize minimal bit vector and value
    x_min = np.zeros(n)
    v_min = 0
    # separate Q
    qua  = np.triu(Q, 1)
    qua += qua.T
    lin  = np.diag(Q)
    for k in range(1, 2**n):
        l = (k ^ (k-1)).bit_count()-1
        x[l] = 1-x[l]
        delta = (2*x[l]-1) * (qua[l]@x + lin[l])
        v += delta
        if v < v_min:
            x_min[:] = x
            v_min = v
    return x_min, v_min
```

While this code works perfectly fine, Python loops are comparatively slow. For this reason, we can use the package numba, which allows for just-in-time (JIT) compilition of certain, structurally simple functions to C. In many cases, this increases the performance considerably. Most NumPy functionality is covered by numba. We only need to change a single line of our code, as the Python function int.bit_count() is not supported. We can replace the respective line with

```
l = int(np.log2(k ^ (k-1)))
```

which is functionally equivalent. To enable JIT compilation, we simply need to add a decorator before both function definitions:

```
from numba import njit

@njit
def naive_brute_force(Q):
    ...

@njit
def improved_brute_force(Q):
    ...
```

The resulting running times of both methods are shown in Fig. 2. The experiment was conducted on an Intel Core i7-8700 CPU running Python 3.10.5 on an Arch-based Linux system. Running times are averaged over 10 random QUBO instances. We can clearly see that the improved version is, on average, faster by a factor of about 10 than the naive version on the tested value range.

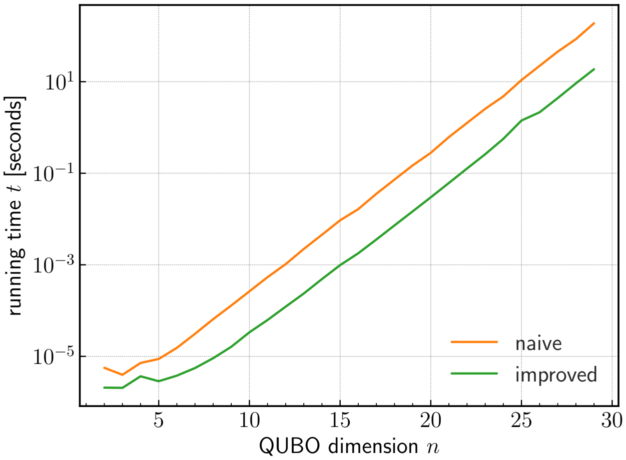

Figure 2: Running time comparison between naive brute-force solving and Algorithm 1; lower is better.

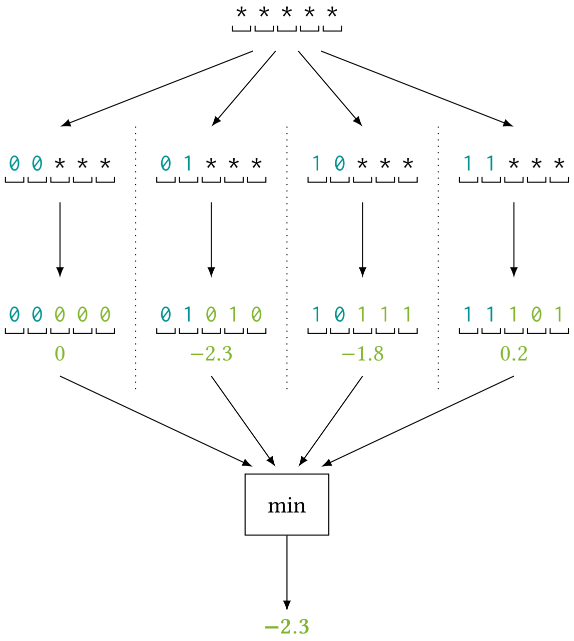

Figure 3: Parallel brute-force solving by fixing 2 bits of a 5 bit vector. The bits marked with * are to be optimized. The steps seperated by dotted lines can be performed on four different CPUs in parallel.

3.1 Parallelization

The code presented in the previous section can be easily modified to be executed in parallel. To this end, we can fix the last m bits of every $\boldsymbol{x}$ and let the brute-force loop only run over the first $n - m$ bits, and their results combined by taking the minimum.

This way, we obtain 2^m sub-problems, one for each assignment of m bits, which can be evaluated in parallel and the results combined by taking their minimum. This is exemplarily shown in Fig. 3 for $n = 5$ and $m = 2$. Using this technique, if there are P CPUs available, the number of additional bits that can be solved in the same amount of time is $\lfloor \log_2(P) \rfloor$.

A C implementation of Algorithm 1 using this parallelization method is given in Appendix A.

4 CONCLUSION

In this work we have seen that brute-force solving QUBO problems can be made more efficient by updating the function value incrementally. To this end, we can leverage Gray codes to obtain an order of traversing all n-bit vectors in a way that allows us to compute said function value update more efficiently, reducing the computational complexity within the loop from $O(n^2)$ to $O(n)$. Moreover, we have seen how to implement the resulting algorithm in Python. We have further seen how `numba` can improve performance through JIT compilation, which requires next to no changes to the code. Finally, a parallelization scheme was presented that is very easy to implement, as demonstrated with the C code given in Appendix A.

While brute-force solving is still infeasible for large n, having a fast implementation of such an algorithm is still a valuable tool for research purposes. Often, experiments are conducted on low-dimensional QUBO instances, e.g., to validate theoretical properties. Current quantum algorithms like QAOA can solve QUBO instances with a low two-digit number of qubits. On our machine we are able to brute-force QUBO instances with $n = 30$ in under 4 seconds, which is valuable for quickly obtaining ground-truth solutions for benchmark problems.

Lastly, the algorithm presented here uses some interesting techniques to improve efficiency, that may serve as inspiration for similar problems.

The multi-threaded C implementation of Algorithm 1 will be included in the upcoming version `0.3.11` of my Python package `qubolite`[2]:

```
pip install qubolite
```

ACKNOWLEDGMENTS

This research has been funded by the Federal Ministry of Education and Research of Germany and the state of North-Rhine Westphalia as part of the Lamarr-Institute for Machine Learning and Artificial Intelligence.

REFERENCES

[1] Christian Bauckhage, Fabrice Beaumont, and Sebastian Müller. 2021. *ML2R Coding Nuggets: Hopfield Nets for Hard Vector Quantization.*

[2] Christian Bauckhage, Cesar Ojeda, Rafet Sifa, and Stefan Wrobel. 2018. Adiabatic Quantum Computing for Kernel k= 2 Means Clustering.. In *LWDA*. 21–32.

[3] C. Bauckhage, R. Ramamurthy, and R. Sifa. 2020. Hopfield Networks for Vector Quantization. In *Artificial Neural Networks and Machine Learning – ICANN 2020 (Lecture Notes in Computer Science)*. Springer International Publishing, 192–203. https://doi.org/10.1007/978-3-030-61616-8_16

[4] Richard A. Caruana and J. David Schaffer. 1988. Representation and Hidden Bias: Gray vs. Binary Coding for Genetic Algorithms. In *Machine Learning Proceedings 1988*. Morgan Kaufmann, 153–161. https://doi.org/10.1016/B978-0-934613-64-4.50021-9

[5] D-Wave Systems. 2021. *Technical Description of the D-Wave Quantum Processing Unit.*

[6] Prasanna Date, Davis Arthur, and Lauren Pusey-Nazzaro. 2020. QUBO Formulations for Training Machine Learning Models. *arXiv:2008.02369 [physics, stat]* (2020). arXiv:physics, stat/2008.02369

[7] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. 2014. A Quantum Approximate Optimization Algorithm. *arXiv preprint arXiv:1411.4028* (2014). arXiv:1411.4028

[8] Edward Farhi, Jeffrey Goldstone, Sam Gutmann, and Michael Sipser. 2000. Quantum Computation by Adiabatic Evolution. *arXiv preprint quant-ph/0001106* (2000). arXiv:quant-ph/0001106

[9] Peter L Hammer and Eliezer Shlifer. 1971. Applications of Pseudo-Boolean Methods to Economic Problems. *Theory and decision* 1, 3 (1971), 296–308.

[10] Tadashi Kadowaki and Hidetoshi Nishimori. 1998. Quantum Annealing in the Transverse Ising Model. *Physical Review E* 58, 5 (1998), 5355.

[11] Gary Kochenberger, Fred Glover, Bahram Alidaee, and Karen Lewis. 2005. Using the Unconstrained Quadratic Program to Model and Solve Max 2-SAT Problems. *International Journal of Operational Research* 1, 1-2 (2005), 89–100.

[12] Gary Kochenberger, Jin-Kao Hao, Fred Glover, Mark Lewis, Zhipeng Lü, Haibo Wang, and Yang Wang. 2014. The Unconstrained Binary Quadratic Programming Problem: A Survey. *Journal of Combinatorial Optimization* 28, 1 (2014), 58–81.

[13] DJ Laughhunn. 1970. Quadratic Binary Programming with Application to Capital-Budgeting Problems. *Operations research* 18, 3 (1970), 454–461.

[14] Sascha Mücke, Nico Piatkowski, and Katharina Morik. 2019. Learning Bit by Bit: Extracting the Essence of Machine Learning. In *Proceedings of the Conference on "Lernen, Wissen, Daten, Analysen" (LWDA) (CEUR Workshop Proceedings)*, Vol. 2454. 144–155.

[15] Florian Neukart, Gabriele Compostella, Christian Seidel, David von Dollen, Sheir Yarkoni, and Bob Parney. 2017. Traffic Flow Optimization Using a Quantum Annealer. *Frontiers in ICT* 4 (2017).

[16] Panos M Pardalos and Somesh Jha. 1992. Complexity of Uniqueness and Local Search in Quadratic 0–1 Programming. *Operations research letters* 11, 2 (1992), 119–123.

[17] Tobias Stollenwerk, Elisabeth Lobe, and Martin Jung. 2019. Flight Gate Assignment with a Quantum Annealer. In *Quantum Technology and Optimization Problems (Lecture Notes in Computer Science)*. Springer International Publishing, 99–110. https://doi.org/10.1007/978-3-030-14082-3_9

A C IMPLEMENTATION

The following code is an implementation of Algorithm 1 in C, which additionally uses the parallelization scheme described in Section 3.1. For multi-threading, the *OpenMP* interface is used (be sure to use the `-fopenmp` flag when compiling with `gcc` on Linux). The method `brute_force_parallel` expects the QUBO matrix as a 2D array of double-precision floats.

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <omp.h>

typedef unsigned char bit;

struct _result_t {
    bit *x_min;
    double v_min;
} typedef result_t;

result_t brute_force_worker(
    double **qubo,
    const size_t n,
    const size_t m) {
    bit x[n];
    memset(x, 0, n);
    double v = 0;
    bit *x_min = (bit*) malloc(n);
    double v_min;
    // set sub-vector for each thread
    size_t p = omp_get_thread_num();
    for (size_t i=0; i<m; ++i)
        x[n-i-1] = (p & (1<<i))>0 ? 1 : 0;
    // evaluate QUBO once on initial vector
    for (size_t i=n-m; i<n; ++i) {
        if (x[i] <= 0)
            continue;
    v += qubo[i][i];
        for (size_t j=i+1; j<n; ++j)
            v += x[j]*qubo[i][j];
    }
    // set initial minimal values
```

[2] https://github.com/smuecke/qubolite

```
    memcpy(x_min, x, n);
    v_min = v;

    size_t l;
    double delta;
    for (int64_t i=1; i<(1<<(n-m)); ++i) {
        l = __builtin_ctzll(i);
        x[l] ^= 1; // flip bit
        // calculate function value update
        delta = 0;
        for (size_t j=0; j<l; ++j)
            delta += x[j]*qubo[j][l];
        delta += qubo[l][l];
        for (size_t j=l+1; j<n; ++j)
            delta += x[j]*qubo[l][j];
        v += x[l] ? delta : -delta;
        if (v < v_min) {
            memcpy(x_min, x, n);
            v_min = v;
        }
    }
    result_t result = {x_min, v_min};
    return result;
}

result_t brute_force_parallel(double **qubo, const size_t n) {
    // determine max. number of threads
    const int64_t p_max = omp_get_max_threads();
    // take floor of log2 of p_max to get
    // max number of bits that can be fixed
    const size_t m = 63 - __builtin_clzll(p_max);
    const size_t M = 1<<m; // number of threads
    // allocate space for sub-results
    result_t results[M];
    // make sure to use exactly 2**m threads
    omp_set_dynamic(0);
    #pragma omp parallel num_threads(M)
    {
        results[omp_get_thread_num()]
        = brute_force_worker(qubo, n, m);
    }
    // get global minimum
    bit *x_min = (bit*) malloc(n);
    double v_min = 0;
    double v;
    for (size_t i=0; i<M; ++i) {
    v = results[i].v_min;
        if (v<v_min) {
        memcpy(x_min, results[i].x_min, n);
            v_min = v;
    }
    free(results[i].x_min);
    }
    result_t result = {x_min, v_min};
    return result;
}

int main() {
    const size_t n = 8;
    double **qubo = (double**) malloc(n*sizeof(double*));
    for (size_t i=0; i<n; ++i)
    qubo[i] = (double*) malloc(n*sizeof(double));

    for (size_t i=0; i<n; ++i) {
        for (size_t j=0; j<n; ++j) {
            if (i>j)
                qubo[i][j] = 0;
        else
        qubo[i][j] = (double) (i-j+2);
        }
    }

    result_t result = brute_force_parallel((double**) qubo, n);
    printf("minimum vector: ");
    for (size_t i=0; i<n; ++i)
    printf("%d", result.x_min[i]);
    printf("\nminimum value:  %f\n", result.v_min);

    // free all allocated resources
    free(result.x_min);
    for (size_t i=0; i<n; ++i)
    free(qubo[i]);
    free(qubo);
}
```