ML2R Coding Nuggets
Sorting as a QUBO

Christian Bauckhage*
Machine Learning Rhine-Ruhr
Fraunhofer IAIS
St. Augustin, Germany

Pascal Welke†
Machine Learning Rhine-Ruhr
University of Bonn
Bonn, Germany

ABSTRACT

Having previously considered sorting as a linear programming problem, we now cast it as a quadratic unconstrained binary optimization problem (QUBO). Deriving this formulation is a bit cumbersome but it allows for implementing neural networks or even quantum computing algorithms that sort. Here, however, we consider a simple greedy QUBO solver and implement it using *NumPy*.

1 INTRODUCTION

Previously [2], we saw how to express the problem of sorting the elements x_i of an n-dimensional, real-valued vector

$$\boldsymbol{x} = [x_1, x_2, \ldots, x_n]^\mathsf{T} \tag{1}$$

as a linear programming problem. Our approach was to understand sorting as the problem of estimating an $n \times n$ **permutation matrix** $\boldsymbol{P}$ such that the elements y_i of the permuted vector

$$\boldsymbol{y} = \boldsymbol{P}\boldsymbol{x} \tag{2}$$

obey $y_i \leq y_{i+1}$. To devise an objective function whose minimizer corresponds to the sought after permutation matrix, we made use of the **rearrangement inequality**. Given the following auxiliary n-dimensional vector

$$\boldsymbol{n} = [1, 2, \ldots, n]^\mathsf{T} \tag{3}$$

this inequality implies that the expression $-\boldsymbol{n}^\mathsf{T}\boldsymbol{y}$ is minimal, if $y_i \leq y_{i+1}$. We also recalled that the $n \times n$ permutation matrices form the vertices of the **Birkhoff polytope**

$$\mathcal{B}_n = \left\{ \boldsymbol{M} \in \mathbb{R}^{n \times n} \;\middle|\; \boldsymbol{M} \geq \boldsymbol{0} \wedge \boldsymbol{M}\boldsymbol{1} = \boldsymbol{1} \wedge \boldsymbol{M}^\mathsf{T}\boldsymbol{1} = \boldsymbol{1} \right\} \tag{4}$$

of doubly stochastic matrices and therefore found that sorting is to solve

$$\begin{aligned} \boldsymbol{P} = \underset{\boldsymbol{M} \in \mathbb{R}^{n \times n}}{\operatorname{argmin}} \quad & -\boldsymbol{n}^\mathsf{T}\boldsymbol{M}\boldsymbol{x} \\ & \boldsymbol{M} \geq \boldsymbol{0} \\ \text{s.t.} \quad & \boldsymbol{M}\boldsymbol{1} = \boldsymbol{1} \\ & \boldsymbol{M}^\mathsf{T}\boldsymbol{1} = \boldsymbol{1} \end{aligned} \tag{5}$$

In this note, we will assume an even more abstract point of view on sorting and cast it as a quadratic unconstrained binary optimization problem (QUBO).

Why would we do this? Would it have practical advantages? No, at least not on commodity hardware! QUBOs are notoriously difficult to solve and, w.r.t. computational efficiency, QUBO solvers cannot compete with conventional sorting algorithms.

However, the fact that we can cast the sorting problem as a QUBO means that we can solve it using Hopfield networks. This, in turn, establishes that sorting can be done on next generation computing devices such as neuromorphic computers or quantum computers. This note therefore demonstrates how we might "rethink" familiar problems to then be able to solve them on emerging platforms.

Based on the linear program in (5), we next derive a QUBO formulation of the sorting problem (section 2). Since sorting is not a difficult problem, our QUBO is rather well behaved and can be solved using a greedy optimization algorithm which we implement in *NumPy* (section 3). Readers who would like to experiment with our code should be familiar with *NumPy* and *SciPy* [10] and only need to

```
import numpy as np
import numpy.random as rnd
```

2 THEORY

Our discussion in this section is split into two major parts: we first derive a QUBO for sorting and then consider a simple (albeit inefficient) algorithms that solves our QUBO.

2.1 A Formulation of Sorting as a QUBO

Looking at (5), we quickly realize that we can rewrite this linear programming problem in terms of an integer programming problem over binary matrices $\boldsymbol{Z} \in \{0,1\}^{n \times n}$. After all, permutation matrices are but binary matrices with a single 1 in each of their rows and columns. In other words, the problem in (5) is equivalent to

$$\begin{aligned} \boldsymbol{P} = \underset{\boldsymbol{Z} \in \{0,1\}^{n \times n}}{\operatorname{argmin}} \quad & -\boldsymbol{n}^\mathsf{T}\boldsymbol{Z}\boldsymbol{x} \\ & \boldsymbol{Z}\boldsymbol{1} = \boldsymbol{1} \\ \text{s.t.} \quad & \boldsymbol{Z}^\mathsf{T}\boldsymbol{1} = \boldsymbol{1} \end{aligned} \tag{6}$$

Since (6) only searches over binary and thus non-negative matrices, we have dropped the non-negativity constraint. Regarding the two sum-to-one constraints, we note that they force the sought after binary matrix to have exactly one 1 per row and column. In other words, any feasible minimizer of (6) will be a permutation matrix.

Similar to what we did in [2], we will henceforth work with the transpose $-\boldsymbol{x}^\mathsf{T}\boldsymbol{Z}^\mathsf{T}\boldsymbol{n}$ of the objective function in (6). We also recall from [2] that we can write

$$\boldsymbol{Z}^\mathsf{T}\boldsymbol{n} = \boldsymbol{N}\boldsymbol{z} \tag{7}$$

* 0000-0001-6615-2128
† 0000-0002-2123-3781

where $z \in \{0,1\}^{n^2}$ and $N \in \mathbb{R}^{n \times n^2}$ are given by

$$z = \text{vec}(Z) \tag{8}$$

$$N = I \otimes n^\intercal \tag{9}$$

and I and $\otimes$ denote the $n \times n$ identity matrix and the Kronecker product. Similar arguments apply to the expressions in the two constraints in (6). That is, we further have

$$Z\mathbf{1} = C_r\, z \tag{10}$$

$$Z^\intercal \mathbf{1} = C_c\, z \tag{11}$$

where the two $n \times n^2$ matrices C_r and C_c on the right are given by

$$C_r = \mathbf{1}^\intercal \otimes I \tag{12}$$

$$C_c = I \otimes \mathbf{1}^\intercal \tag{13}$$

Consequently, we can rephrase the problem of estimating an optimal permutation matrix as the problem of finding an optimal binary vector, namely

$$z^* = \underset{z \in \{0,1\}^{n^2}}{\text{argmin}} \; -x^\intercal N z \quad \text{s.t.} \quad \begin{matrix} C_r z = \mathbf{1} \\ C_c z = \mathbf{1} \end{matrix} \tag{14}$$

Once this problem has been solved, the sought after permutation matrix can be computed as $P = \text{mat}(z^*)$ to then obtain the sorted version $y = Px$ of x.

Now, to turn the linear constrained binary problem in (14) into a quadratic unconstrained binary problem, we first of all note the following implications

$$C_r z = \mathbf{1} \;\Leftrightarrow\; \left\| C_r z - \mathbf{1} \right\|^2 = 0 \tag{15}$$

$$C_c z = \mathbf{1} \;\Leftrightarrow\; \left\| C_c z - \mathbf{1} \right\|^2 = 0 \tag{16}$$

Second of all, we expand the two Euclidean distances as

$$\left\| C_r z - \mathbf{1} \right\|^2 = z^\intercal C_r^\intercal C_r z - 2\, \mathbf{1}^\intercal C_r z + \mathbf{1}^\intercal \mathbf{1} \tag{17}$$

$$\left\| C_c z - \mathbf{1} \right\|^2 = z^\intercal C_c^\intercal C_c z - 2\, \mathbf{1}^\intercal C_c z + \mathbf{1}^\intercal \mathbf{1} \tag{18}$$

Since $\mathbf{1}^\intercal \mathbf{1} = n$ is a constant independent of z, we therefore have the following Lagrangian for the minimization problem in (14)

$$\begin{aligned} L(z, \lambda_r, \lambda_c) &= -x^\intercal N z \\ &\quad + \lambda_r \left(z^\intercal C_r^\intercal C_r z - 2\, \mathbf{1}^\intercal C_r z \right) \\ &\quad + \lambda_c \left(z^\intercal C_c^\intercal C_c z - 2\, \mathbf{1}^\intercal C_c z \right) \qquad (19) \\ &= z^\intercal \left(\lambda_r\, C_r^\intercal C_r + \lambda_c\, C_c^\intercal C_c \right) z \\ &\quad - \left(x^\intercal N + 2\, \mathbf{1}^\intercal \left(\lambda_r\, C_r + \lambda_c\, C_c \right) \right)^\intercal z \qquad (20) \\ &\equiv z^\intercal R z - r^\intercal z \qquad (21) \end{aligned}$$

Here, λ_r and λ_c are two Lagrange multipliers which we henceforth treat as parameters that have to be set manually.

But all of this is to say that the linear programming problem in (14) can just as well be cast as a quadratic unconstrained binary optimization problem, namely

$$z^* = \underset{z \in \{0,1\}^{n^2}}{\text{argmin}} \; z^\intercal R z - r^\intercal z \tag{22}$$

Again, if we could solve this QUBO for z^*, the actually sought after permutation matrix would be $P = \text{mat}(z^*)$ and would allow us to obtain the sorted version $y = Px$ of x.

2.2 A Greedy Solution Algorithm

Our sorting QUBO in (22) constitutes a **discrete optimization problem**. Its decision variable $z \in \{0,1\}^{n^2}$ is a binary vector whose entries do not vary continuously. Optimization techniques based on calculus do therefore not immediately apply[1]. In fact, since we are searching the set of all 2^{n^2} binary vectors for an optimal z^*, we have turned sorting into a **combinatorial optimization problem**. In general, these are difficult to solve.

However, due to the specific structure of its ingredients R and r, our QUBO in (22) is rather well behaved. Though we will not prove it here, this is to say that any *local minimum* of its objective function also is a *global minimum*. Next, we will exploit this to devise a simple greedy algorithm for solving (22).

To begin with, we note that it is common to call the objective in (22) an energy function. We henceforth follow this convention and refer to

$$E(z) = z^\intercal R z - r^\intercal z \tag{23}$$

as the *energy* of z.

Second of all, since z is an n^2-dimensional binary vector, we may think of its entries z_i as *bits*. Now, assume we were given any solution candidate z. We could compute its energy $E(z)$ and then ask which of its bits z_i should be flipped (i.e. set to $\neg z_i$) in order to maximally decrease the current energy and thus to maximally improve the current solution.

Note that this idea is computationally expensive as it requires n^2 individual evaluations of the energy function in (23). Nevertheless, we can use it to iteratively update an initial guess of the solution until no further decrease in energy is possible.

Third of all, in order to slightly improve on the overall runtime of this search procedure, we recall that, for z to be a valid solution of (22), only n of its bits can be active (i.e. in state 1). If we thus were to start our search for the solution with the vector of all 0s, we could iteratively *activate* optimally chosen bits until the number of active bits equals n.

In short, solving the sorting QUBO in (22) can be accomplished using the following greedy optimization algorithm:

Algorithm 1 greedy search for a solution to (22)

initialize $z = \mathbf{0}$
initialize $E = \mathbf{0}$

while $\sum_i z_i < n$ **do**

  for $i = 1, \ldots, n^2$ **do**
    $E_i = E(z_1, \ldots, \neg z_i, \ldots, z_{n^2})$

  $a = \text{argmin}_i\, E_i$
  $z_a = \neg z_a$

[1] In an upcoming *Coding Nugget*, we will study a clever way of making them applicable.

Listing 1: setting up the QUBO in (22)

```
1  def initializeSortQUBO(vecX, lr=None, lc=None):
2      n = len(vecX)
3
4      vecN = np.arange(n) + 1
5
6      matI = np.eye(n)
7      vec1 = np.ones(n)
8
9      matN  = np.kron(matI, vecN)
10     matCr = np.kron(vec1, matI)
11     matCc = np.kron(matI, vec1)
12
13     if lr is None or lc is None:
14         vecX = vecX / np.sum(vecX)
15         lr = lc = n
16
17     matR = lr * matCr.T @ matCr + lc * matCc.T @ matCc
18     vecR = vecX @ matN + 2 * vec1 @ (lr * matCr + lc * matCc)
19
20     return matR, vecR
```

Listing 2: greedily solving the QUBO in (22)

```
1  def energy(vecZ, matR, vecR):
2      return vecZ @ matR @ vecZ - vecR @ vecZ
3
4
5  def flipZi(vecZ, i):
6      vecZ[i] = 1 - vecZ[i]; return vecZ
7
8
9  def solveSortQUBO(matR, vecR):
10     n2 = len(vecR)
11     n  = np.sqrt(n2)
12
13     vecZ = np.zeros(n2)
14     enrg = np.zeros(n2)
15
16     while np.sum(vecZ) < n:
17         for i in range(n2):
18             enrg[i] = energy(flipZi(np.copy(vecZ), i), matR, vecR)
19
20         a = np.argmin(enrg)
21
22         vecZ = flipZi(vecZ, a)
23
24     return vecZ
```

3 PRACTICE

Next, we look at how to implement the greedy procedure in Alg. 1 in order to solve the QUBO in (22) and thus to sort the entries of a vector $\boldsymbol{x} \in \mathbb{R}^n$.

To work with a practical example, we first create a random vector $\boldsymbol{x}$ of, say, $n = 5$ entries $0 \leq x_i \leq 100$ which we represent as a one-dimensional *NumPy* array `vecX`. To keep things legible, we will force the x_i to be integers and proceed as follows

```
n    = 5
vecX = rnd.randint(100, size=n)
```

In order to inspect the entries of this random vector, we simply use

```
print (vecX)
```

and may obtain something like this

```
[46 52 12 10 51]
```

Given $\boldsymbol{x}$, we next initialize the parameters $\boldsymbol{R}$ and $\boldsymbol{r}$ of our QUBO. To this end, we apply

```
matR, vecR = initializeSortQUBO(vecX)
```

and thus use function `initializeSortQUBO` in Listing 1. Its three parameters are the array `vecX` we just created and two scalars `lr` and `lc` which represent the multipliers λ_r and λ_c in (19)–(21). The latter can be set by the user, however, we choose their default values to be `None` and shortly explain why.

Within `initializeSortQUBO`, lines 2–11 repeat ideas we already discussed in [2]: Arrays `matI` and `vec1` represent the $n \times n$ identity matrix and the n-dimensional vector of all ones, respectively. Arrays `matN`, `matCr`, and `matCc` implement the matrices $\boldsymbol{N}$, $\boldsymbol{C}_r$, and $\boldsymbol{C}_c$ which we defined in equations (9), (12), and (13). In order to compute these arrays, we apply the *NumPy* function `kron` which realizes the Kronecker product.

Lines 13–15 address the crucial open question of how to set the two Lagrange multipliers λ_r and λ_c which parameterize $\boldsymbol{R}$ and $\boldsymbol{r}$. To make a long story short, their optimal choice depends on (the size of the entries x_i of) the vector $\boldsymbol{x}$ we want to sort. As a workaround of this issue, the default behavior of our code is to normalize $\boldsymbol{x}$ to have an L_1 norm of 1 and then to set both λ_r and λ_c to n.

Finally, lines 17 and 18 compute two arrays `matR` and `vecR` which represent matrix $\boldsymbol{R}$ and vector $\boldsymbol{r}$; these computations are nothing but direct implementations of the respective terms in equation (20).

Having initialized `matR` and `vecR`, we next call

```
vecZ = solveSortQUBO(matR, vecR)
```

to determine the solution $\boldsymbol{z}^*$ of our sorting QUBO. That is, we use function `solveSortQUBO` in Listing 2.

This function is a straightforward *NumPy* implementation[2] of Alg. 1: Line 13 initializes an all zeros array `vecZ` which represents the binary vector $\boldsymbol{z}$ which we will refine iteratively. Line 14 initializes an array `enrg` in which we store the intermediate energy values E_i required by our algorithm.

The `while`-loop in line 16 simply realizes the *while*-loop in Alg. 1.

The `for`-loop in line 17 iterates over the n^2 entries of $\boldsymbol{z}$. In each iteration, we create a copy of the current instance of array `vecZ` (using `np.copy(vecZ)`); in this copy, we flip the i-th entry (using function `flipZi`), compute the corresponding energy (using function `energy`), and store the result in `enrg[i]`. Once this loop has terminated, line 20 determines the index of the smallest entry of `enrg` and line 22 activates the corresponding entry of `vecZ`.

Having thus obtained the solution `vecZ` to our QUBO, we next turn it into the required permutation matrix $\boldsymbol{P}$. This is as simple as

```
matP = vecZ.reshape(n,n).T
```

For our running example, the resulting array `matP` turns out to be

```
[[0. 0. 0. 1. 0.]
 [0. 0. 1. 0. 0.]
 [1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1.]
 [0. 1. 0. 0. 0.]]
```

And, to verify that this permutation matrix does indeed solve our exemplary problem, we use

[2] Note, however, that we sacrifice efficiency for readability. Readers are encouraged to try to tweak our implementation towards better performance (for instance, by cleverly avoiding costly copy operations).

```
print ('vecX = ', vecX.astype(float))
print ('vecY = ', matP @ vecX)
```

which yields

```
vecX =  [46. 52. 12. 10. 51.]
vecY =  [10. 12. 46. 51. 52.]
```

Success! We have solved a QUBO to obtain a sorted version $\boldsymbol{y} = \boldsymbol{P}\boldsymbol{x}$ of an unordered vector $\boldsymbol{x}$.

NOTE: While our example illustrates that sorting can be done by solving a QUBO, this is really not a good idea when working with conventional computers. Even for moderate problem sizes n, the above approach becomes unbearably slow. Readers can see this for themselves by trying to sort a vector $\boldsymbol{x}$ of, say, $n = 50$ elements. However, those with access to adiabatic quantum computers [5] or digital annealers [3, 7, 8] might find our formulation of the sorting problem much more appealing.

4 SUMMARY AND OUTLOOK

In this note, we saw how to cast the sorting problem as a quadratic unconstrained binary optimization problem (QUBO) and presented a greedy search procedure for its solution.

Yet, from the point of view of computational efficiency, we cannot recommend the above approach since it is much slower than conventional sorting algorithms. At the same time, the problem formulation and solution we presented here are not thet smartest ways of thinking about sorting as a QUBO. Indeed, (22) can be further rewritten and then be solved using Hopfield nets that work in an informed manner [1, 11].

Moreover, the additional rewrite will also allow for quantum sorting. This, too, is not really remarkable because sorting is not the kind of problem that requires heavy machinery. However, the modeling approach we sketched in this note points to new solutions for much more demanding permutation problems [4, 6, 9, 12] and we will substantiate this claim in later notes.

ACKNOWLEDGMENTS

This material was produced within the Competence Center for Machine Learning Rhine-Ruhr (**ML2R**) which is funded by the Federal Ministry of Education and Research of Germany (grant no. 01|S18038C). The authors gratefully acknowledge this support.

REFERENCES

[1] C. Bauckhage, R. Sanchez, and R. Sifa. 2020. Problem Solving with Hopfield Networks and Adiabatic Quantum Computing. In *Proc. IJCNN*. IEEE.
[2] C. Bauckhage and P. Welke. 2021. *ML2R Coding Nuggets: Sorting as Linear Programming*. Technical Report. MLAI, University of Bonn.
[3] J. Boyd. 2018. Silicon Chip Delivers Quantum Speeds. *IEEE Spectrum* 55, 7 (2018).
[4] G.D. Evangelidis and C. Bauckhage. 2013. Efficient Subframe Video Alignment Using Short Descriptors. *IEEE Trans. Pattern Analysis and Machine Intelligence* 35, 10 (2013).
[5] M. Johnson and et al. 2011. Quantum Annealing with Manufactured Spins. *Nature* 473, 7346 (2011).
[6] J. Kunegis, D. Fay, and C. Bauckhage. 2010. Network Growth and the Spectral Evolution Model. In *Proc. CIKM*. ACM.
[7] S. Mücke, N. Piatkowski, and K. Morik. 2019. Hardware Acceleration of Machine Learning Beyond Linear Algebra. In *Proc. ECML/PKDD*.
[8] S. Mücke, N. Piatkowski, and K. Morik. 2019. Learning Bit by Bit: Extracting the Essence of Machine Learning. In *Proc. LWDA*.
[9] A. Nowak, S. Villar, A.S. Bandeira, and J. Bruna. 2017. Revised Note on Learning Algorithms for Quadratic Assignment with Graph Neural Networks. *arXiv:1706.07450 [stat.ML]* (2017).
[10] T.E. Oliphant. 2007. Python for Scientific Computing. *Computing in Science & Engineering* 9, 3 (2007).
[11] L. von Rueden, S. Mayer, K. Beckh, B. Georgiev, S. Giesselbach, R. Heese, B. Kirsch, J. Pfrommer, A. Pick, R. Ramamurthy, M. Walczak, J. Garcke, C. Bauckhage, and J. Schuecker. 2019. Informed Machine Learning – A Taxonomy and Survey of Integrating Knowledge into Learning Systems. *arXiv:1903.12394 [stat.ML]* (2019).
[12] M.M. Zavlanos and G. J. Pappas. 2008. A Dynamical Systems Approach to Weighted Graph Matching. *Automatica* 44, 11 (2008).