Yuehan Zhang

Contact me:
johnzhang514145@gmail.com

Life Update

By - Yuehan Zhang – Posted on 2025-11-17 – Posted in Quantum

Decoded Quantum Interferometry for the max-XORSAT Problem

Introduction

This tutorial compares the performance of classical methods and Decoded Quantum Interferometry (DQI) implemented by quairkit for the max-XORSAT problem.

MAX-XORSAT

Finding the best solution(s) from a large, finite set of possible candidates is known as **combinatorial optimization**. A well-known problem from this field is the **traveling salesman problem**. Mathematically, this can be phrased as trying to maximize an objective function whose domain is large and discrete.

The **max-XORSAT** problem is a simple example of this, where we are given an $m \times n$ matrix B (with $m > n$) and a vector $\mathbf{v}$ of length m, and are required to find the n-bit string $\mathbf{x}$ that satisfies the maximum number of constraints imposed by the m linear mod-2 equations,

$$B\mathbf{x} = \mathbf{v}$$

Since this system of equations is over $\mathbb{F}_2$, the matrix and vectors only contain zeros and ones.

The objective function we are aiming to maximize is:

$$f(\mathbf{x}) = \sum_{i=1}^{m} (-1)^{v_i + \mathbf{b}_i \cdot \mathbf{x}} = \sum_{i=1}^{m} f_i(\mathbf{x}),$$

where $\mathbf{b}_i$ is the i-th row of matrix B.

You can verify that this function represents the **number of satisfied equations minus the number of unsatisfied ones**, by considering that when the equation is satisfied, the exponent $v_i + \mathbf{b}_i \cdot \mathbf{x}$ is always even, and that it is odd in the opposite case.

Classical Solutions & Their Complexity

Type	Algorithm	Idea	Ti
Exact	Brute force	Try all 2^n assignments; evaluate each in $O(m)$	$O($
	Rank-based search	Do Gaussian elimination over $\mathbb{F}_2$ to reduce to rank (r); branch only on (r) “free” variables (or use nullspace basis)	$O($
	Branch-and-bound + elimination	Repeated elimination to simplify + upper bounds to prune	Ex (in de
	0–1 ILP / Pseudo-Boolean (PB)	Encode parity clauses; modern PB/SAT solvers with cutting/branching	Wo ex
Feasibility (XORSAT) only	Gaussian elimination	Solve (Bx=v) over $\mathbb{F}_2$	$O($
Approximation	Random assignment	Set each bit uniformly at random	$O($ eva

	Method of conditional expectations	Greedily fix bits to keep expected score ≥ baseline	$O($
	Max-2-Lin(2) via SDP (GW)	Reduce to signed Max-Cut; solve SDP, round (Goemans–Williamson)	Po $\tilde{O}($
	LP/SDP for k≥3	Relax to LP/SDP then round	Po
	Spectral (k=2)	Use top eigenvector of signed Laplacian; threshold	$O($
	Belief / Max-Sum Propagation	Message passing on factor graph	$O($
Heuristics	Local search (bit-flip / Kernighan–Lin style)	Maintain clause gains; flip improving bits or blocks	$O($ (ar
	Simulated annealing /	Escape local minima with randomness /	De sc

Tabu memory

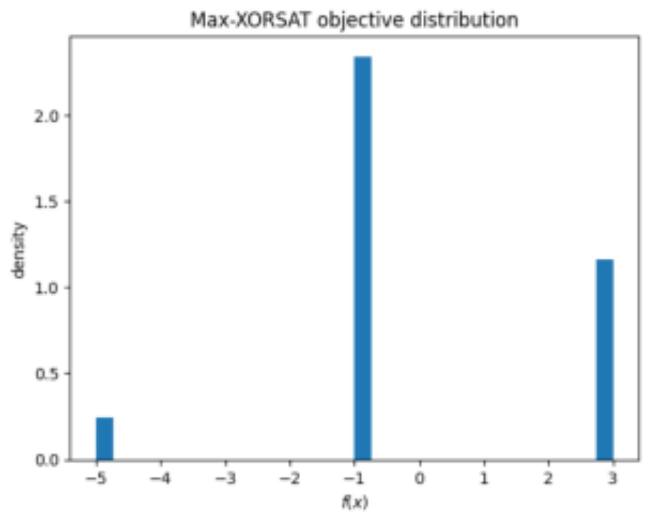
Quairkit Implementation

```
import torch
import quairkit as qkit
from quairkit import Circuit
from quairkit.database import *
from quairkit.qinfo import *
import matplotlib.pyplot as plt

qkit.set_dtype('complex128')

from quairkit import *
import numpy as np
from itertools import combination
import warnings
warnings.filterwarnings("ignore")
```

Let's define the conditions for our specific max-XORSAT problem and visualize the objective function in a histogram when randomly sampling bit strings $\mathbf{x}$ from a uniform distribution. Later, we'll use samples generated by the DQI algorithm and compare their quality, in terms of the objective function, to this initial plot to see how well DQI performs.

```
torch.manual_seed(0)
```

```
B = torch.tensor([[1, 0, 0, 0], [

v = torch.tensor([1, 0, 1, 0, 1],

m, n = B.shape
n_samples = 10000

def objective_function(x: torch.T
    Bx = (B @ x) % 2
    parity = (v + Bx) % 2
    f = (1 - 2 * parity).sum()
    return f.item()

samples = torch.randint(0, 2, siz
f_x_array_random = [objective_fun

plt.hist(f_x_array_random, bins=3
plt.xlabel(r"$f(x)$")
plt.ylabel("density")
plt.title("Max-XORSAT objective d
plt.show()
```



Amplitude

Encoding: Weight Coefficients

We are going to prepare the superposition $\sum_{k=0}^{\ell} w_k |k\rangle$.

```
p = 2
r = 1
d = (p - 2 * r) / torch.sqrt(torc
l = 2

num_weight_qubits = int(torch.cei
weight_register = range(num_weigh
m_register = range(num_weight_qub
n_register = range(m + num_weight

def w_k_optimal(m: int, l: int, d
    dtype = torch.float64
    diag_main_vals = torch.arange
    A = torch.diag(diag_main_vals

    i = torch.arange(1, l + 1, dt
    sup_vals = torch.sqrt(i * (m
    A[:-1, 1:] += torch.diag(sup_
    A[1:, :-1] += torch.diag(sup_

    eigvals, eigvecs = torch.lina
    w = eigvecs[:, -1]

    return w

w_k = w_k_optimal(m, l, d)
print("The optimal values for w a
```

The optimal values for w are [0.4385290096535146, 0.7071067811865476, 0.5547001962252291]

Prepare Dicke states with k excitations

For this step, we need a conditional operation that prepares Dicke states $|D^m_k\rangle$ — where the superscript is the number of qubits and the subscript is the number of excitations — for each index k in the weight register. Dicke states are another way of referring to a uniform superposition of all bit strings of a determined Hamming weight. For our particular example, we will prepare Dicke states with one and two excitations. Before implementing the conditional operation, let's briefly review a method for preparing such states as presented in [2].

```
def ccry(cir, theta, controls, ta
    """Implements CCRY using deco
    c1, c2 = controls
    cir.ry(target, theta / 4)
    cir.cnot([c2, target])
    cir.ry(target, -theta / 4)
    cir.cnot([c1, target])
    cir.ry(target, theta / 4)
```

```
        cir.cnot([c2, target])
        cir.ry(target, -theta / 4)
        cir.cnot([c1, target])

    def SCS(m_val, k, cir: Circuit):
        theta_1 = 2 * torch.arccos(
            torch.sqrt(torch.tensor(1
        )

        cir.cx([m_val - 2, m_val - 1]
        cir.cry(qubits_idx=[m_val - 1
        cir.cx([m_val - 2, m_val - 1]

        for ell in range(2, k + 1):
            theta_ell = 2 * torch.arc
                torch.sqrt(torch.tens
            )

            cir.cx([m_val - ell - 1,

            ccry(
                cir,
                theta_ell,
                controls=[m_val - 1,
                target=m_val - ell -
            )

            cir.cx([m_val - ell - 1,

    def prepare_dicke_cir(m, k):
        """Prepares a Dicke state wit
        cir = Circuit(m)

        for wire_idx in range(m - k,
            cir.x(wire_idx)
        for i in reversed(range(k + 1
            SCS(i, k, cir)
        for i in reversed(range(2, k
            SCS(i, i - 1, cir)

        return cir

    def embed_weights(cir, w_k, weigh
        """Prepare 2-qubit weight reg
           w0|00> + w1|01> + w2|10> +

           weight_register: [q0, q1]
        """
        vec = torch.tensor(w_k, dtype
        w0, w1, w2 = vec

        # --- Step 1: Apply an RY on
        # |ψ1> = sqrt(w0^2 + w1^2) |0
        #
        # That is: sin(α/2) = w2, cos
        s01 = torch.sqrt(w0**2 + w1**

        alpha = 2 * torch.arcsin(w2)

        q0, q1 = weight_register[0],
        cir.ry(q0, alpha)

        # The full state is now:
        # sqrt(w0^2 + w1^2) |0>_q0 |0

        # --- Step 2: In the subspace
        # |0>_q0 |0>_q1  ->  (w0/s01)
        #
        # i.e., apply a controlled-RY
        # cos(β/2) = w0/s01,  sin(β/2
        if s01 > 1e-12:
            w0p = w0 / s01
            w1p = w1 / s01
            beta = 2 * torch.arctan2(

            # Implement controlled RY
            cir.x(q0)
            cir.cry(qubits_idx=[q0, q
            cir.x(q0)

        # Final state (theoretically)
        # w0 |00> + w1 |01> + w2 |10>

    def weight_error_prep(m, w_k):
        total_systems = num_weight_qu
        cir = Circuit(total_systems)

        weight_register = [0, 1]
        embed_weights(cir, w_k, weigh

        error_idx = list(range(2, 2 +

        dicke1 = prepare_dicke_cir(m,
        U1 = dicke1.unitary_matrix()

        cir.x(weight_register[0])
        cir.control_oracle(
            oracle=U1,
            system_idx=[weight_regist
        )
        cir.x(weight_register[0])

        dicke2 = prepare_dicke_cir(m,
        U2 = dicke2.unitary_matrix()

        cir.x(weight_register[1])
        cir.control_oracle(
            oracle=U2,
            system_idx=[weight_regist
        )
        cir.x(weight_register[1])

        return cir

    state = zero_state(m+n+num_weight
    cir1 = weight_error_prep(m, w_k)
    state = cir1(state)
```

Uncompute the weight register

After preparing the Dicke states, we uncompute and discard the state of the weight register. In general, this is a straightforward process, as the Hamming weights encoded are known. We accomplished this by generating bit strings of length m with Hamming weight of 2 using the generate_bit_strings function. We then applied a controlled bit flip to the weight register for these specific cases. We did not need to perform any action for bit strings with a Hamming weight of 1, as the qubit state was already $|0\rangle$. From now on, we can choose to disregard the weight register.

```
    def mcx_matrix(num_controls):
        """Builds the unitary matrix
        size = 2 ** (num_controls + 1
        mat = np.eye(size, dtype=comp
        mat[-1, -1] = 0
        mat[-1, -2] = 1
        mat[-2, -1] = 1
        mat[-2, -2] = 0
        return torch.tensor(mat, dtyp

    def phase_Z(cir, v, error_start=1
        """Imparts a phase (-1)^{v.y}
        for i in range(len(v)):
            if v[i] == 1:
                cir.z(error_start + i

    def generate_bit_strings(length:
        """Generates all bit strings 
        results = []
        for positions in combinations
            bit_string = [0] * length
            for p in positions:
                bit_string[p] = 1
            results.append(bit_string
        return results

    def uncompute_weight_to_zero(cir,
        num_controls = len(m_register
        mcx_mat = mcx_matrix(num_cont

        for pattern in generate_bit_s
            flips = [m_register[j] fo

            for q in flips: cir.x(q)

            cir.oracle(
                mcx_mat,
                system_idx = m_regist
            )

            for q in flips: cir.x(q)

        for pattern in generate_bit_s
            flips = [m_register[j] fo

            for q in flips: cir.x(q)

            cir.oracle(
                mcx_mat,
                system_idx = m_regist
            )

            for q in flips: cir.x(q)
```

Encode constraints vector

To impart a phase $(-1)^{\mathbf{v}\cdot\mathbf{y}}$, we perform a Pauli-$Z$ on each qubit for which $v_i = 1$. This is simply a conditional operation within a for loop in the phase_Z function. Let's now implement this step, together with the weight uncomputation, in a function encode_v, and output the resulting quantum state.

```
def encode_v(m):
    cir = Circuit(2 + m +n)
    weight_register = [0,1]
    m_register = list(range(2, 2+
    uncompute_weight_to_zero(cir,
    phase_Z(cir, v, error_start=2

    return cir

cir2 = encode_v(m)
state = cir2(state)
print(state)
```

```
Backend: default-pure System dimension: [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2] System sequence: [6, 4, 2, 3, 5, 0, 1, 7, 8, 9, 10] [0.44+0.j 0. +0.j 0. +0.j ... 0. +0.j 0. +0.j 0. +0.j]
```

Encode matrix B in the syndrome register

We are almost finished, hang in there! Now, we need to compute $B^T\mathbf{y}$ in the syndrome register. While it may not be immediately obvious how to implement this as a unitary operation, the binary nature of the matrix and vector allows for a smooth translation. This operation can be realized using CNOT gates, with controls on the error register and targets on the syndrome register[1]. Specifically, a CNOT is applied for every entry $B^T_{ij} = 1$, controlled on the j-th qubit of the error register and with the i-th qubit of the syndrome register as the target. Let's implement this in the quantum function ndrome_prep.

```
def B_T_multiplication(cir, B_T, 
    m_local = B_T.shape[1]
    n_local = B_T.shape[0]
    for row_idx in range(n_local)
        for col_idx in range(m_lo
            if B_T[row_idx, col_i
                cir.cnot([error_s

def syndrome_prep_circuit():
    cir = Circuit(n+m+2)
    B_T = B.T
    B_T_multiplication(cir, B_T, 
    return cir

cir3 = syndrome_prep_circuit()
state = cir3(state)
```

Decoding

This step is the main challenge of the algorithm: uncomputing the error register $\mathbf{y}$ using the information from the syndrome register $\mathbf{s} = B^T\mathbf{y}$ in an efficient way. This would be an easy task if B were always a square matrix; however, since it is not, we need to solve an underdetermined linear system of equations $\mathbf{s} = B^T\mathbf{y}$ subject to the constraint $|\mathbf{y}| \leq \ell$ given by the known Hamming weights of $\mathbf{y}$. The problem we have just described is precisely the syndrome decoding problem, where B^T is the parity-check matrix, $\mathbf{s}$ is the syndrome, and $\mathbf{y}$ is the error.

The kernel of B^T defines an error-correcting code. The distance d of this code determines the number of errors it can correct, given by $\lfloor(d-1)/2\rfloor$. This condition ensures that the decoding problem has a unique solution. For this demo, we choose $\ell = 2$ such that it is less than half the distance of the code $d = 5$ and this condition is met. For a detailed discussion of the restrictions on ℓ, please refer to the original paper[1].

To keep things simple, we will use a straightforward approach for decoding by building a Lookup Table (LUT) where we compute the syndrome for each possible error using the classical function syndrome_LUT. While this function might look daunting, it is simply calculating a product between a matrix and a vector, and storing the results in a usable format. (Feel free to explore other decoders such as belief propagation.)

Then, for each syndrome in the syndrome register, the corresponding error is uncomputed in the error register using controlled bit-flip operations. We will now integrate this into our decoding quantum function and see how the syndrome register is uncomputed.

```
def syndrome_LUT(parity_check_mat
    """Generates the Lookup table
    num_data_qubits = parity_chec
    syndrome_dict = {}

    for i in range(2**num_data_qu
        error_bitstring = format(
        error_vector = torch.tens

        syndrome_vector = torch.m
        syndrome_bitstring = "".j

        if syndrome_bitstring not
            syndrome_dict[syndrom
        else:
            existing_error = synd
            existing_weight = sum
            current_weight = sum(
            if current_weight < e
                syndrome_dict[syn

    # Convert to list-of-lists
    lookup_matrix = []
    for syndrome_str, error_str i
        syndrome_list = [int(bit)
        error_list = [int(bit) fo
```

```
            lookup_matrix.append([syn

        # Sort by syndrome
        lookup_matrix.sort(key=lambda

        return lookup_matrix

    # Generate the lookup table
    decoding_table = syndrome_LUT(B.T

    def decoding(m, n, previous_state
        """Quantum circuit decoding a
        total_systems = m + n + num_w
        cir = Circuit(total_systems)

        # Uncompute syndrome register
        error_idx    = list(range(2,
        syndrome_idx = list(range(2 +

        for syndrome, error in decodi
            for i in range(len(error)
                if error[i] == 1:
                    # Flips for contr
                    flips = [syndrome

                    # Pre-flip: Apply
                    for wire in flips
                        cir.x(wire)

                    # Build and apply
                    num_controls = le
                    mcx_mat = mcx_mat
                    qubits_idx = synd
                    cir.oracle(mcx_ma

                    # Post-flip: Rest
                for wire in flips
                    cir.x(wire)

        # Execute on previous_state
        state = cir(previous_state)

        return state

    # Usage (assuming raw_state_vecto
    state = decoding(m, n, state)
    print(state)
```

```
Backend: default-pure System dimension: [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2] System sequence: [7, 8, 9, 10, 5, 3, 4, 2, 6, 0, 1] [0.44+0.j 0. +0.j 0. +0.j ... 0. +0.j 0. +0.j 0. +0.j]
```

Hadamard transform and sample

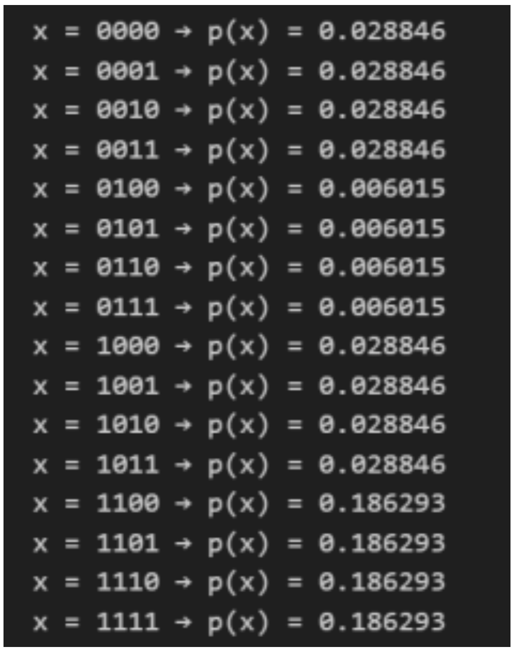
After the previous step, we obtained the Hadamard transform of the state we are looking for. The final step is to apply the Hadamard transform to this state to obtain $|P(f)\rangle = \sum_{\mathbf{x}} P(f(\mathbf{x}))\,|\mathbf{x}\rangle$. Let us write a DQI quantum function containing all the steps of the algorithm previously described.

```
    def Hadamard(previous_state):
```

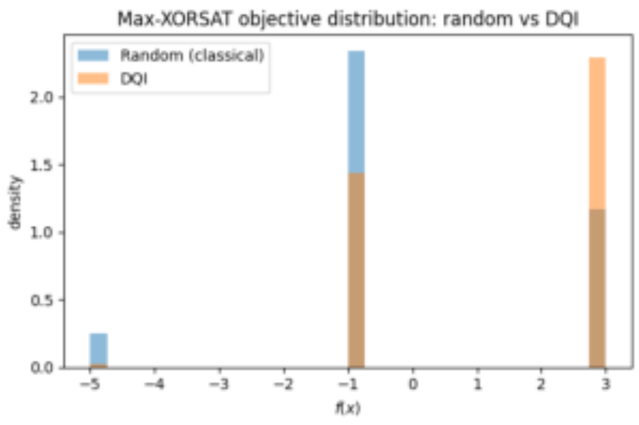
```
    total_systems = 2 + m + n
    cir = Circuit(total_systems)

    for q in range(2 + m, 2 + m +
        cir.h(q)

    state = cir(previous_state)

    traced_state = partial_trace(


    return traced_state
```

We will collect samples, calculate their objective values, and build a histogram to compare with the random sampling done at the beginning of the demo.

```
state = Hadamard(state)
prob = state.measure()

for i, p in enumerate(prob):
    bitstring = format(i, f"0{n}b
    print(f"x = {bitstring} → p(x

samples_dict = prob_sample(prob,

f_x_array_dqi = []

for bitstring, counts in samples_
    num = int(counts.item())
    x_bits = torch.tensor([int(b)
    fx = objective_function(x_bit
    f_x_array_dqi.extend([fx] * n

plt.figure(figsize=(6, 4))
plt.hist(f_x_array_random, bins=3
plt.hist(f_x_array_dqi,     bins=

plt.xlabel(r"$f(x)$")
plt.ylabel("density")
plt.title("Max-XORSAT objective d
plt.legend()
plt.tight_layout()
plt.show()
```



```
x = 0000 → p(x) = 0.028846
x = 0001 → p(x) = 0.028846
x = 0010 → p(x) = 0.028846
x = 0011 → p(x) = 0.028846
x = 0100 → p(x) = 0.006015
x = 0101 → p(x) = 0.006015
x = 0110 → p(x) = 0.006015
x = 0111 → p(x) = 0.006015
x = 1000 → p(x) = 0.028846
x = 1001 → p(x) = 0.028846
x = 1010 → p(x) = 0.028846
x = 1011 → p(x) = 0.028846
x = 1100 → p(x) = 0.186293
x = 1101 → p(x) = 0.186293
x = 1110 → p(x) = 0.186293
x = 1111 → p(x) = 0.186293
```



References

[1] Jordan, Stephen P., et al. “Optimization by decoded quantum interferometry.” Nature 646.8086 (2025): 831-836.

[2] Bärtschi, Andreas, and Stephan Eidenbenz. “Deterministic preparation of Dicke states.” International Symposium on Fundamentals of Computation Theory. Cham: Springer International Publishing, 2019.

Previous Article

Private Deployment of Team Note-Taking Software: Tutorial for Deploying Outline on Ubuntu

Next Article

Industrial integer linear programs with Decoded Quantum Interferometry

Leave a Reply

Your email address will not be published. Required fields are marked *.

Your N *

Your E *

Website

Comment

Save my name, email, and website in this browser for the next time I comment.

POST COMMENT