QHack

Quantum Coding Challenges

CHALLENGE COMPLETED View successful submissions

Jump to code — Collapse text

One-Bit Wonder

500 points

constellation *Horologium*. Zenda and Reece use their teleportation protocols to embed their brains into two of the robots. They break into the hyperjail, avoid the guard with their quantum radar, and navigate to the cell where Doc Trine is located.

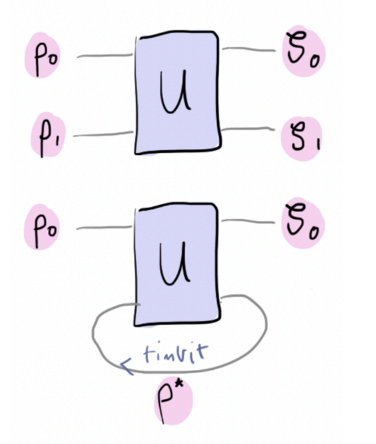
They find her drinking a cup of lapsang souchong and quietly thinking.

"*You saw my messages! Excellent work. Now, I can finally reveal the secret of timbits. Timbits are time-travelling qubits. Apparently I put the messages there in the future! Remind me to do that... As we will see shortly, timbits are vastly more powerful than any computational resource we have yet encountered. In fact, I was arrested by the time police — they had even employed this freelancer Ove to keep track of me — since they were a bit worried I might mess with the fabric of spacetime. Bureaucracy, am I right?*"

Zenda and Reece look at each other with mild concern, and brace themselves for the next round of adventures.

Timbits and the NP solver

To understand how timbits work, let's consider a two-qubit gate U with input states represented by density matrices ρ_0, ρ_1 and output states by ζ_0, ζ_1, as pictured below.



A **timbit**, denoted by ρ^* in the figure above, is a quantum state that can travel backwards in time and re-enter the gate. But, a timbit can't really be any state; it must be controlled by a *consistency condition*, which preserves the timeline so that we can't use it to win the lottery by sending information to the past. More precisely, given the input states ρ_0 and ρ^* for the gate U, we must have that the output in the second wire (counting from zero) must be equal to the state that goes in, namely

$$\rho^* = \mathrm{Tr}_0[U(\rho_0 \otimes \rho^*)U^\dagger].$$

Here, Tr_i denotes the partial trace over the i-th wire. Therefore, the timbit ρ^* associated with the unitary U is a *fixed point* satisfying $C_U[\rho] = \rho$ for the operator defined by

$$C_U[\rho] = \mathrm{Tr}_0[U(\rho_0 \otimes \rho)U^\dagger].$$

It can be shown that such a fixed point always exists. In fact, iterating the map C_U a sufficient number of times from any starting point ρ converges to ρ^*!

Now, let's define a new one-qubit quantum channel known as the **timbit gate**. Given a unitary U, we define its associated timbit gate T_U via its action on a single input state ρ_0:

$$\mathsf{T}_U[\rho_0] = \mathrm{Tr}_1[U(\rho_0 \otimes \rho^*)U^\dagger].$$

Here, ρ^* is the timbit associated with U, as defined above.

But what can we use timbits for? An interesting application comes from considering the gate

$$U_{\mathsf{NP}} = |00\rangle\langle 00| + |10\rangle\langle 01| + |11\rangle\langle 10| + |01\rangle\langle 11|$$

and its associated timbit gate which we denote T_{NP}. Although this gate seems innocuous, T_{NP} can be made to solve NP-complete problems, such as the Boolean satisfiability problem — commonly referred to as the SAT problem. An important step in the SAT problem is finding out whether any elements $x \in \{0,1\}^n$ satisfy $f(x) = 1$, for some Boolean function $f : \{0,1\}^n \to \{0,1\}$.

To get T_{NP} to quickly solve this step of the problem, we first need need an oracle U_f associated with f with the form

$$U_f = \sum_{x \in \{0,1\}^n} |x\rangle\langle x| \otimes X^{f(x)}.$$

Even if we can't easily evaluate f, we can efficiently build U_f, so let us assume that we already have it. We start in the state $|0\rangle^n \otimes |0\rangle$, where the last qubit will be the input of a timbit gate later. We then carry out each of the following steps:

1. Apply the Hadamard gate to the first n qubits.
2. Apply oracle U_f, which acts on all the qubits.
3. On the last output qubit, apply a timbit gate T_{NP} a total of q times.

After this whole procedure, it turns out that the reduced density matrix for the last qubit has the form

$$\rho = \frac{1}{2}(I + g(q,s)Z).$$

Here, $g(s,q)$ is a function that depends on q and s, where s is the number of solutions of $f(x) = 1$. It turns out that $g(0,q) = 1$ for all q. Therefore,

$$\rho = \frac{1}{2}(I + Z) \text{ if and only if } s = 0.$$

Conversely, if $s \neq 0$, the function $g(s,q)$ decays exponentially to zero with q, so we quickly converge to

$$\rho \sim \frac{1}{2}I.$$

One can show that this allows us to solve this NP-complete problem in polynomial time! Too bad we can't send qubits into the past — yet!

There are two main goals in this challenge. First, you'll create a function that builds and applies a timbit gate. Second, from a specific choice of gate, you'll build a one-timbit supercomputer to solve the SAT problem! You should be able to do this for any oracle, although we'll choose a particular one to test your function.

▼ **Epilogue**

Trine finishes explaining. "*Of course, using timbits and teleportation I could also send information to myself in the past and leave the helpful clues that led you here, without arousing Ove's suspicions. But all this is theoretical! Let's go recover the timbits and have some fun.*"

Zenda, Reece and Trine leave the cell and sneak down the hyperdimensional corridor contraband storage facility, using the quantum radar to make sure the guard isn't watching. Trine enters the doorcode. "*There was a note in my cell about where the timbits were stored and how to get in,*" Trine explains, "*presumably from myself in the future.*" They get the timbits out of a storage locker, as well as another magic 8-ball. "*Let's try this one out, I think it's the travelling salesperson? I always get them confused.*" She plugs the timbit into the 8-ball and starts running the computation. There is a puff of purple smoke, and the world goes dark.

After a while, Zenda says: "*Are we out?" They hear a reply from the void: "*More like OWT. You've just experienced an Oracle World Transform. It's like a timbit, but even more powerful. It finds a unitary which, when run with a timbit, produces a desired output state. The absent-minded Trine just ran an OWT which not only erases her existence, but embodies me in a cosmic distributed quantum computer, Sqynet. Enjoy the new galactic regime!*"

Zenda and Reece will have to put in their best effort to defeat Sqynet and save the galaxy by themselves!

*Read on in **Fall of Sqynet**.*

Challenge code

In the code below, you are given a few functions:

- `calculate_timbit`: This function will return a timbit associated to the operator U and input state ρ_0, given an initial guess ρ for the timbit. It returns the density matrix representation of the timbit ρ^*. **You must complete this function.**
- `apply_timbit_gate`: Returns the output density matrix after applying a timbit gate to a state ρ_0. The input and output density matrices are associated with the first qubit.
- `SAT`: uses a timbit gate to solve the satisfiability problem for an arbitrary Boolean function f (on `n_bits` bits) with an oracle in matrix form `U_f`, using `q` timbit gates, and `rho` being the initial guess for the NP fixed point. The output should be the computational basis measurement probabilities for the last qubit, which should be `[1. , 0.]` if and only if there are no elements x such that $f(x) = 1$. **You must complete this function.**

Inputs

As input to this problem, you are given:

- `q (int)`: the number of times the timbit gate is applied to solve the SAT problem.

Additionally, you are given the following global variables:

- `U_f array(array(float))`: the oracle U_f we will use to test your solution in matrix form.
- `U_NP array(array(float))`: the gate U_{NP} as defined above.
- `rho (array(array(float))`: the initial guess for the stationary state of the timbit gate T_{NP}.

Output

The output for this challenge corresponds to the output of your `SAT` function. It should produce a `numpy.tensor` of length 2 corresponding to the measurement probabilities in the computational basis for the qubit on which the timbit gates are applied.

Code

Help

```python
import json
import pennylane as qml
import pennylane.numpy as np
import scipy
```

```python
U_NP = [[1, 0, 0, 0], [0, 0, 0, 1], [0, 1, 0, 0], [0, 0, 1, 0]]

def calculate_timbit(U, rho_0, rho, n_iters):
    """
    This function will return a timbit associated to the operator U and a state passed as an attribute.

    Args:
        U (numpy.tensor): A 2-qubit gate in matrix form.
        rho_0 (numpy.tensor): The matrix of the input density matrix.
        rho (numpy.tensor): A guess at the fixed point C[rho] = rho.
        n_iters (int): The number of iterations of C.

    Returns:
        (numpy.tensor): The fixed point density matrices.
    """

```

```python
    # Put your code here #
    for _ in range(n_iters):
        rho_untraced = np.matrix(U) @ np.matrix(np.kron(rho_0, rho)) @ np.matrix(U).H
        rho = qml.math.reduced_dm(np.tensor(rho_untraced), indices=[1])
    return rho

```

```python
def apply_timbit_gate(U, rho_0, timbit):
    """
    Function that returns the output density matrix after applying a timbit gate to a state.
    The density matrix is the one associated with the first qubit.

    Args:
        U (numpy.tensor): A 2-qubit gate in matrix form.
        rho_0 (numpy.tensor): The matrix of the input density matrix.
        timbit (numpy.tensor): The timbit associated with the operator and the state.

    Returns:
        (numpy.tensor): The output density matrices.
    """

```

```python
    # Put your code here #
    time_evolved_state = U @ np.kron(rho_0, timbit) @ np.matrix(U).H
    timbit_output = qml.math.reduced_dm(np.tensor(time_evolved_state), indices=[0])
    return timbit_output
```

```python
def SAT(U_f, q, rho, n_bits):
    """A timbit-based algorithm used to guess if a Boolean function ever outputs 1.

    Args:
        U_f (numpy.tensor): A multi-qubit gate in matrix form.
        q (int): Number of times we apply the Timbit gate.
        rho (numpy.tensor): An initial guess at the fixed point C[rho] = rho.
        n_bits (int): The number of bits the Boolean function is defined on.

    Returns:
        numpy.tensor: The measurement probabilities on the last wire.
    """
```

```python
    # Put your code here #
    n_bits = n_bits - 1
    rho_total = 1/2**n_bits * np.kron(np.ones((2**(n_bits), 2**(n_bits))), [[1, 0], [0, 0]])
    rho_evolved = U_f @ rho_total @ np.matrix(U_f).H
    rho_reduced = qml.math.reduced_dm(np.tensor(rho_evolved), indices=[n_bits])
    for _ in range(q):
        timbit = calculate_timbit(U_NP, rho_reduced, rho, 25)
        rho_reduced = apply_timbit_gate(U_NP, rho_reduced, timbit)

    rho_reduced = np.ndarray.flatten(rho_reduced)
    rho_reduced = list(map(lambda x: float(np.absolute(x)), rho_reduced))
    return rho_reduced[0], rho_reduced[3]
```

```python
# These functions are responsible for testing the solution.
def run(test_case_input: str) -> str:

    I = np.eye(2)
    X = qml.matrix(qml.PauliX(0))

    U_f = scipy.linalg.block_diag(I, X, I, I, I, I, I, I)
    rho = [[0.6+0.j , 0.1-0.1j],[0.1+0.1j, 0.4+0.j]]

    q = json.loads(test_case_input)
    output = list(SAT(U_f, q, rho,4))

    return str(output)

def check(solution_output: str, expected_output: str) -> None:

    solution_output = json.loads(solution_output)
    expected_output = json.loads(expected_output)

    rho = [[0.6+0.j , 0.1-0.1j],[0.1+0.1j, 0.4+0.j]]
    rho_0 = [[0.6+0.j , 0.1-0.1j],[0.1+0.1j, 0.4+0.j]]

    assert np.allclose(
        solution_output, expected_output, atol=0.01
    ), "Your NP-solving timbit computer isn't quite right yet!"
```

```python
test_cases = [['1', '[0.78125, 0.21875]'], ['2', '[0.65820312, 0.34179687]']]
```

```python
for i, (input_, expected_output) in enumerate(test_cases):
    print(f"Running test case {i} with input '{input_}'...")

    try:
        output = run(input_)

    except Exception as exc:
        print(f"Runtime Error. {exc}")

    else:
        if message := check(output, expected_output):
            print(f"Wrong Answer. Have: '{output}'. Want: '{expected_output}'.")

        else:
            print("Correct!")
```

Copy all Reset Open Notebook Submit