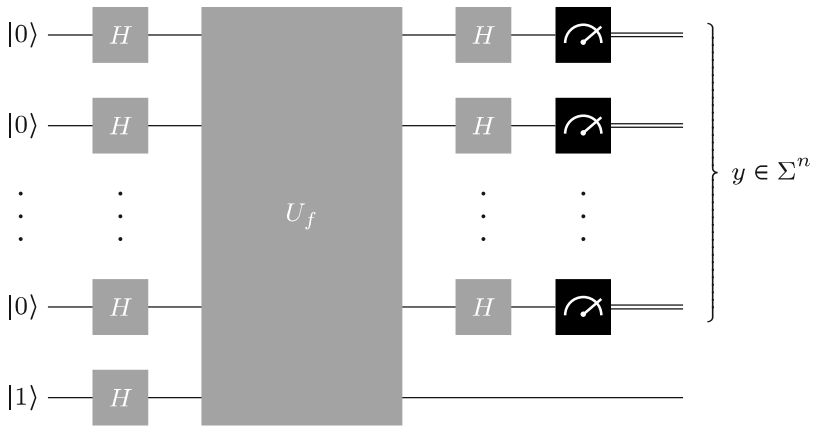
The Deutsch-Jozsa algorithm

Deutsch's algorithm outperforms all classical algorithms for a query problem, but the advantage is quite modest: one query versus two. The Deutsch-Jozsa algorithm extends this advantage — and, in fact, it can be used to solve a couple of different query problems.

Here's a quantum circuit description of the Deutsch-Jozsa algorithm. An additional classical post-processing step, not shown in the figure, may also be required depending on the specific problem being solved.



Of course, we haven't actually discussed what problems this algorithm solves; this is done in the two sections that follow.

The Deutsch-Jozsa problem

We'll begin with the query problem the Deutsch-Jozsa algorithm was originally intended to solve, which is known as the *Deutsch-Jozsa problem*.

The input function for this problem takes the form $f : \Sigma^n \to \Sigma$ for an arbitrary positive integer n. Like Deutsch's problem, the task is to output 0 if f is constant and 1 if f is balanced, which again means that the

number of input strings on which the function takes the value 0 is equal to the number of input strings on which the function takes the value 1.

Notice that, when n is larger than 1, there are functions of the form $f : \Sigma^n \to \Sigma$ that are neither constant nor balanced. For example, the function $f : \Sigma^2 \to \Sigma$ defined as

$$\begin{aligned} f(00) &= 0 \\ f(01) &= 0 \\ f(10) &= 0 \\ f(11) &= 1 \end{aligned}$$

falls into neither of these two categories. For the Deutsch-Jozsa problem, we simply don't worry about functions like this — they're considered to be "don't care" inputs. That is, for this problem we have a *promise* that f is either constant or balanced.

Deutsch-Jozsa problem
Input: a function $f : \{0,1\}^n \to \{0,1\}$
Promise: f is either constant or balanced
Output: 0 if f is constant, 1 if f is balanced

The Deutsch-Jozsa algorithm, with its single query, solves this problem in the following sense: if every one of the n measurement outcomes is 0, then the function f is constant; and otherwise, if at least one of the measurement outcomes is 1, then the function f is balanced. Another way to say this is that the circuit described above is followed by a classical post-processing step in which the OR of the measurement outcomes is computed to produce the output.

Algorithm analysis

To analyze the performance of the Deutsch-Jozsa algorithm for the Deutsch-Jozsa problem, it's helpful to begin by thinking about the action of a single layer of Hadamard gates. A Hadamard operation can be expressed as a matrix in the usual way,

$$H = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix},$$

but we can also express this operation in terms of its action on standard basis states:

$$H|0\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$$
$$H|1\rangle = \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle.$$

These two equations can be combined into a single formula,

$$H|a\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}(-1)^a|1\rangle = \frac{1}{\sqrt{2}}\sum_{b\in\{0,1\}}(-1)^{ab}|b\rangle,$$

which is true for both choices of $a \in \Sigma$.

Now suppose that instead of just a single qubit we have n qubits, and a Hadamard operation is performed on each. The combined operation on the n qubits is described by the tensor product $H \otimes \cdots \otimes H$ (n times), which we write as $H^{\otimes n}$ for conciseness and clarity. Using the formula from above, followed by expanding and then simplifying, we can express the action of this combined operation on the standard basis states of n qubits like this:

$$\begin{aligned}
&H^{\otimes n}|x_{n-1}\cdots x_1 x_0\rangle \\
&\quad = \big(H|x_{n-1}\rangle\big)\otimes\cdots\otimes\big(H|x_0\rangle\big) \\
&\quad = \left(\frac{1}{\sqrt{2}}\sum_{y_{n-1}\in\Sigma}(-1)^{x_{n-1}y_{n-1}}|y_{n-1}\rangle\right)\otimes\cdots\otimes\left(\frac{1}{\sqrt{2}}\sum_{y_0\in\Sigma}(-1)^{x_0 y}\right. \\
&\quad = \frac{1}{\sqrt{2^n}}\sum_{y_{n-1}\cdots y_0\in\Sigma^n}(-1)^{x_{n-1}y_{n-1}+\cdots+x_0y_0}|y_{n-1}\cdots y_0\rangle.
\end{aligned}$$

Here, by the way, we're writing binary strings of length n as $x_{n-1}\cdots x_0$ and $y_{n-1}\cdots y_0$, following Qiskit's indexing convention.

This formula provides us with a useful tool for analyzing the quantum circuit above. After the first layer of Hadamard gates is performed, the state of the $n+1$ qubits (including the leftmost/bottom qubit, which is treated separately from the rest) is

$$\big(H|1\rangle\big)\big(H^{\otimes n}|0\cdots 0\rangle\big) = |-\rangle\otimes\frac{1}{\sqrt{2^n}}\sum_{x_{n-1}\cdots x_0\in\Sigma^n}|x_{n-1}\cdots x_0\rangle.$$

When the U_f operation is performed, this state is transformed into

$$|-\rangle\otimes\frac{1}{\sqrt{2^n}}\sum_{x_{n-1}\cdots x_0\in\Sigma^n}(-1)^{f(x_{n-1}\cdots x_0)}|x_{n-1}\cdots x_0\rangle$$

through exactly the same phase kick-back phenomenon that we saw in the analysis of Deutsch's algorithm.

Then the second layer of Hadamard gates is performed, which (by the formula above) transforms this state into

$$|-\rangle \otimes \frac{1}{2^n} \sum_{x_{n-1}\cdots x_0 \in \Sigma^n} \sum_{y_{n-1}\cdots y_0 \in \Sigma^n} (-1)^{f(x_{n-1}\cdots x_0)+x_{n-1}y_{n-1}+\cdots+x_0y_0} |y_{n-1}$$

This expression looks somewhat complicated, and not too much can be concluded about the probabilities to obtain different measurement outcomes without knowing more about the function f.

Fortunately, all we need to know is the probability that every one of the measurement outcomes is 0 — because that's the probability that the algorithm determines that f is constant. This probability has a simple formula.

$$\left| \frac{1}{2^n} \sum_{x_{n-1}\cdots x_0 \in \Sigma^n} (-1)^{f(x_{n-1}\cdots x_0)} \right|^2 = \begin{cases} 1 & \text{if } f \text{ is constant} \\ 0 & \text{if } f \text{ is balanced} \end{cases}$$

In greater detail, if f is constant, then either $f(x_{n-1}\cdots x_0) = 0$ for every string $x_{n-1}\cdots x_0$, in which case the value of the sum is 2^n, or $f(x_{n-1}\cdots x_0) = 1$ for every string $x_{n-1}\cdots x_0$, in which case the value of the sum is -2^n. Dividing by 2^n and taking the square of the absolute value yields 1.

If, on the other hand, f is balanced, then f takes the value 0 on half of the strings $x_{n-1}\cdots x_0$ and the value 1 on the other half, so the $+1$ terms and -1 terms in the sum cancel and we're left with the value 0.

We conclude that the algorithm operates correctly provided that the promise is fulfilled.

Classical difficulty

The Deutsch-Jozsa algorithm works every time, always giving us the correct answer when the promise is met, and requires a single query. How does this compare with classical query algorithms for the Deutsch-Jozsa problem?

First, any *deterministic* classical algorithm that correctly solves the Deutsch-Jozsa problem must make exponentially many queries: $2^{n-1} + 1$ queries are required in the worst case. The reasoning is that, if a

deterministic algorithm queries f on 2^{n-1} or fewer different strings, and obtains the same function value every time, then both answers are still possible. The function might be constant, or it might be balanced but through bad luck the queries all happen to return the same function value.

The second possibility might seem unlikely — but for deterministic algorithms there's no randomness or uncertainty, so they will fail systematically on certain functions. We therefore have a significant advantage of quantum over classical algorithms in this regard.

There is a catch, however, which is that *probabilistic* classical algorithms can solve the Deutsch-Jozsa problem with very high probability using just a few queries. In particular, if we simply choose a few different strings of length n randomly, and query f on those strings, it's unlikely that we'll get the same function value for all of them when f is balanced.

To be specific, if we choose k input strings $x^1, \ldots, x^k \in \Sigma^n$ uniformly at random, evaluate $f(x^1), \ldots, f(x^k)$, and answer 0 if the function values are all the same, and 1 if not, then we'll always be correct when f is constant, and wrong in the case that f is balanced with probability just 2^{-k+1}. If we take $k = 11$, for instance, this algorithm will answer correctly with probability greater than 99.9%.

For this reason, we do still have a rather modest advantage of quantum over classical algorithms — but it is nevertheless a quantifiable advantage representing an improvement over Deutsch's algorithm.

Deutsch-Jozsa with Qiskit

```
from qiskit import QuantumCircuit
from qiskit_aer import AerSimulator
import numpy as np
```

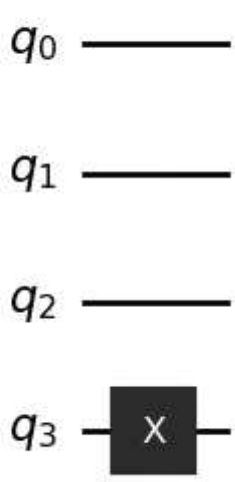
To implement the Deutsch-Jozsa algorithm in Qiskit, we'll start by defining a function `dj_query` that generates a quantum circuit implementing a query gate, for a randomly selected function satisfying the promise for the Deutsch-Jozsa problem. With a 50% chance, the function is constant, and with 50% change the function is balanced. For each of those two possibilities, the function is selected uniformly from the functions of that type. The argument is the number of input bits of the function.

```python
def dj_query(num_qubits):
    # Create a circuit implementing for a query gate i
    # satisfying the promise for the Deutsch-Jozsa pro

    qc = QuantumCircuit(num_qubits + 1)

    if np.random.randint(0, 2):
        # Flip output qubit with 50% chance
        qc.x(num_qubits)
    if np.random.randint(0, 2):
        # return constant circuit with 50% chance
        return qc

    # Choose half the possible input strings
    on_states = np.random.choice(
        range(2**num_qubits),  # numbers to sample fro
        2**num_qubits // 2,  # number of samples
        replace=False,  # makes sure states are only s
    )

    def add_cx(qc, bit_string):
        for qubit, bit in enumerate(reversed(bit_strin
            if bit == "1":
                qc.x(qubit)
        return qc

    for state in on_states:
        qc.barrier()  # Barriers are added to help vis
        qc = add_cx(qc, f"{state:0b}")
        qc.mcx(list(range(num_qubits)), num_qubits)
        qc = add_cx(qc, f"{state:0b}")

    qc.barrier()

    return qc
```

We can show the quantum circuit implementation of the query gate using the `draw` method as usual.

```python
display(dj_query(3).draw(output="mpl"))
```

Output:



Next we define a function that creates the Deutsch-Jozsa circuit, taking a quantum circuit implementation of a query gate as an argument.

```
def compile_circuit(function: QuantumCircuit):
    # Compiles a circuit for use in the Deutsch-Jozsa

    n = function.num_qubits - 1
    qc = QuantumCircuit(n + 1, n)
    qc.x(n)
    qc.h(range(n + 1))
    qc.compose(function, inplace=True)
    qc.h(range(n))
    qc.measure(range(n), range(n))
    return qc
```

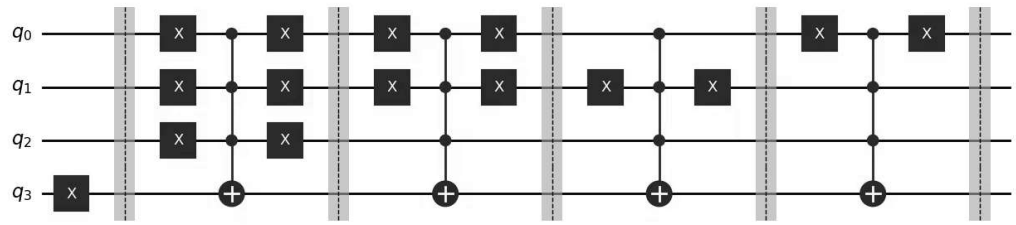
Finally, a function that runs the Deutsch-Jozsa circuit once is defined.

```
def dj_algorithm(function: QuantumCircuit):
    # Determine if a function is constant or balanced

    qc = compile_circuit(function)

    result = AerSimulator().run(qc, shots=1, memory=T
    measurements = result.get_memory()
    if "1" in measurements[0]:
        return "balanced"
    return "constant"
```

We can test our implementation by choosing a function randomly, displaying the quantum circuit implementation of a query gate for this function, and then running the Deutsch-Jozsa algorithm on that function.

```
f = dj_query(3)
display(f.draw("mpl"))
display(dj_algorithm(f))
```

Output:




```
'balanced'
```

The Bernstein-Vazirani problem

Next, we'll discuss a problem known as the *Bernstein-Vazirani problem*. It's also called the *Fourier sampling problem*, although there are more general formulations of this problem that also go by that name.

First, let's introduce some notation. For any two binary strings $x = x_{n-1} \cdots x_0$ and $y = y_{n-1} \cdots y_0$ of length n, we define

$$x \cdot y = x_{n-1}y_{n-1} \oplus \cdots \oplus x_0 y_0.$$

We'll refer to this operation as the *binary dot product*. An alternative way to define it is like so.

$$x \cdot y = \begin{cases} 1 & x_{n-1}y_{n-1} + \cdots + x_0 y_0 \text{ is odd} \\ 0 & x_{n-1}y_{n-1} + \cdots + x_0 y_0 \text{ is even} \end{cases}$$

Notice that this is a symmetric operation, meaning that the result doesn't change if we swap x and y, so we're free to do that whenever it's convenient. Sometimes it's useful to think about the binary dot product $x \cdot y$ as being the parity of the bits of x in positions where the string y has a 1, or equivalently, the parity of the bits of y in positions where the string x has a 1.

With this notation in hand we can now define the Bernstein-Vazirani problem.

Bernstein-Vazirani problem

Input: a function $f : \{0,1\}^n \to \{0,1\}$
Promise: there exists a binary string $s = s_{n-1} \cdots s_0$ for which $f(x) = s \cdot x$ for all $x \in \Sigma^n$
Output: the string s

We don't actually need a new quantum algorithm for this problem; the Deutsch-Jozsa algorithm solves it. In the interest of clarity, let's refer to the quantum circuit from above, which doesn't include the classical post-processing step of computing the OR, as the *Deutsch-Jozsa circuit*.

Algorithm analysis

To analyze how the Deutsch-Jozsa circuit works for a function satisfying the promise for the Bernstein-Vazirani problem, we'll begin with a quick observation. Using the binary dot product, we can alternatively describe the action of n Hadamard gates on the standard basis states of n qubits as follows.

$$H^{\otimes n}|x\rangle = \frac{1}{\sqrt{2^n}} \sum_{y \in \Sigma^n} (-1)^{x \cdot y} |y\rangle$$

Similar to what we saw when analyzing Deutsch's algorithm, this is because the value $(-1)^k$ for any integer k depends only on whether k is even or odd.

Turning to the Deutsch–Jozsa circuit, after the first layer of Hadamard gates is performed, the state of the $n + 1$ qubits is

$$|-\rangle \otimes \frac{1}{\sqrt{2^n}} \sum_{x \in \Sigma^n} |x\rangle.$$

The query gate is then performed, which (through the phase kickback phenomenon) transforms the state into

$$|-\rangle \otimes \frac{1}{\sqrt{2^n}} \sum_{x \in \Sigma^n} (-1)^{f(x)} |x\rangle.$$

Using our formula for the action of a layer of Hadamard gates, we see that the second layer of Hadamard gates then transforms this state into

$$|-\rangle \otimes \frac{1}{2^n} \sum_{x \in \Sigma^n} \sum_{y \in \Sigma^n} (-1)^{f(x) + x \cdot y} |y\rangle.$$

Now we can make some simplifications, in the exponent of -1 inside the sum. We're promised that $f(x) = s \cdot x$ for some string $s = s_{n-1} \cdots s_0$, so we can express the state as

$$|-\rangle \otimes \frac{1}{2^n} \sum_{x \in \Sigma^n} \sum_{y \in \Sigma^n} (-1)^{s \cdot x + x \cdot y} |y\rangle.$$

Because $s \cdot x$ and $x \cdot y$ are binary values, we can replace the addition with the exclusive-OR — again because the only thing that matters for an integer in the exponent of -1 is whether it is even or odd. Making use of the symmetry of the binary dot product, we obtain this expression for the state:

$$|-\rangle \otimes \frac{1}{2^n} \sum_{x \in \Sigma^n} \sum_{y \in \Sigma^n} (-1)^{(s \cdot x) \oplus (y \cdot x)} |y\rangle.$$

(Parentheses have been added for clarity, though they aren't really necessary because it's conventional to treat the binary dot product as having higher precedence than the exclusive-OR.)

At this point we will make use of the following formula.

$$(s \cdot x) \oplus (y \cdot x) = (s \oplus y) \cdot x$$

We can obtain the formula through a similar formula for bits,

$$(ac) \oplus (bc) = (a \oplus b)c,$$

together with an expansion of the binary dot product and bitwise exclusive-OR:

$$\begin{aligned} (s \cdot x) \oplus (y \cdot x) &= (s_{n-1} x_{n-1}) \oplus \cdots \oplus (s_0 x_0) \oplus (y_{n-1} x_{n-1}) \oplus \cdots \oplus \\ &= (s_{n-1} \oplus y_{n-1}) x_{n-1} \oplus \cdots \oplus (s_0 \oplus y_0) x_0 \\ &= (s \oplus y) \cdot x \end{aligned}$$

This allows us to express the state of the circuit immediately prior to the measurements like this:

$$|-\rangle \otimes \frac{1}{2^n} \sum_{x \in \Sigma^n} \sum_{y \in \Sigma^n} (-1)^{(s \oplus y) \cdot x} |y\rangle.$$

The final step is to make use of yet another formula, which works for every binary string $z = z_{n-1} \cdots z_0$.

$$\frac{1}{2^n}\sum_{x\in\Sigma^n}(-1)^{z\cdot x} = \begin{cases} 1 & \text{if } z = 0^n \\ 0 & \text{if } z \neq 0^n \end{cases}$$

Here we're using a simple notation for strings that we'll use several more times in the lesson: 0^n is the all-zero string of length n.

A simple way to argue that this formula works is to consider the two cases separately. If $z = 0^n$, then $z \cdot x = 0$ for every string $x \in \Sigma^n$, so the value of each term in the sum is 1, and we obtain 1 by summing and dividing by 2^n. On the other hand, if any one of the bits of z is equal to 1, then the binary dot product $z \cdot x$ is equal to 0 for exactly half of the possible choices for $x \in \Sigma^n$ and 1 for the other half — because the value of the binary dot product $z \cdot x$ flips (from 0 to 1 or from 1 to 0) if we flip any bit of x in a position where z has a 1.

If we now apply this formula to simplify the state of the circuit prior to the measurements, we obtain

$$|-\rangle \otimes \frac{1}{2^n}\sum_{x\in\Sigma^n}\sum_{y\in\Sigma^n}(-1)^{(s\oplus y)\cdot x}|y\rangle = |-\rangle \otimes |s\rangle,$$

owing to the fact that $s \oplus y = 0^n$ if and only if $y = s$. Thus, the measurements reveal precisely the string s we're looking for.

Classical difficulty

While the Deutsch-Jozsa circuit solves the Bernstein-Vazirani problem with a single query, any classical query algorithm must make at least n queries to solve this problem.

This can be reasoned through a so-called *information theoretic* argument, which is very simple in this case. Each classical query reveals a single bit of information about the solution, and there are n bits of information that need to be uncovered — so at least n queries are needed.

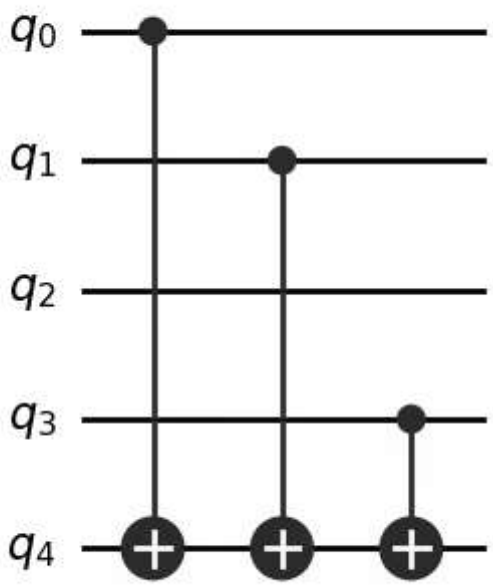
It is, in fact, possible to solve the Bernstein-Vazirani problem classically by querying the function on each of the n strings having a single 1, in each possible position, and 0 for all other bits, which reveals the bits of s one at a time. Therefore, the advantage of quantum over classical algorithms for this problem is 1 query versus n queries.

Bernstein-Vazirani with Qiskit

We've already implemented the Deutsch-Jozsa circuit above, and here we will make use of it to solve the Bernstein-Vazirani problem. First we'll define a function that implements a query gate for the Bernstein-Vazirani problem given any binary string s.

```
def bv_query(s):
    # Create a quantum circuit implementing a query ga
    # Bernstein-Vazirani problem.

    qc = QuantumCircuit(len(s) + 1)
    for index, bit in enumerate(reversed(s)):
        if bit == "1":
            qc.cx(index, len(s))
    return qc


display(bv_query("1011").draw(output="mpl"))
```

Output:



Now we can create a function that runs the Deutsch-Jozsa circuit on the function, using the `compile_circuit` function that was defined previously.

```
def bv_algorithm(function: QuantumCircuit):
    qc = compile_circuit(function)
    result = AerSimulator().run(qc, shots=1, memory=Tru
    return result.get_memory()[0]


display(bv_algorithm(bv_query("1011")))
```

Output:

```
'1011'
```

Remark on nomenclature

In the context of the Bernstein-Vazirani problem, it is common that the Deutsch-Jozsa algorithm is referred to as the "Bernstein-Vazirani algorithm." This is slightly misleading, because the algorithm *is* the Deutsch-Jozsa algorithm, as Bernstein and Vazirani were very clear about in their work.

What Bernstein and Vazirani did after showing that the Deutsch-Jozsa algorithm solves the Bernstein-Vazirani problem (as it is stated above) was to define a much more complicated problem, known as the *recursive Fourier sampling problem*. This is a highly contrived problem where solutions to different instances of the problem effectively unlock new levels of the problem arranged in a tree-like structure. The Bernstein-Vazirani problem is essentially just the base case of this more complicated problem.

The recursive Fourier sampling problem was the first known example of a query problem where quantum algorithms have a so-called *super-polynomial* advantage over probabilistic algorithms, thereby surpassing the advantage of quantum over classical offered by the Deutsch-Jozsa algorithm. Intuitively speaking, the recursive version of the problem amplifies the 1 versus n advantage of quantum algorithms to something much larger.

The most challenging aspect of the mathematical analysis establishing this advantage is showing that classical query algorithms can't solve the problem without making lots of queries. This is quite typical; for many problems it can be very difficult to rule out creative classical approaches that solve them efficiently.

Simon's problem, and the algorithm for it described in the next section, does provide a much simpler example of a super-polynomial (and, in fact, exponential) advantage of quantum over classical algorithms, and for this reason the recursive Fourier sampling problem is less often discussed. It is, nevertheless, an interesting computational problem in its own right.

Was this page helpful?

Yes No

Report a bug, typo, or request content on GitHub ↗.

Previous page Next page