Deutsch's algorithm

Deutsch's algorithm solves the parity problem for the special case that $n = 1$. In the context of quantum computing this problem is sometimes referred to as *Deutsch's problem*, and we'll follow that nomenclature in this lesson.

To be precise, the input is represented by a function $f : \Sigma \to \Sigma$ from one bit to one bit. There are four such functions:

a	$f_1(a)$
0	0
1	0

a	$f_2(a)$
0	0
1	1

a	$f_3(a)$
0	1
1	0

a	$f_4(a)$
0	1
1	1

The first and last of these functions are *constant* and the middle two are *balanced*, meaning that the two possible output values for the function occur the same number of times as we range over the inputs. Deutsch's problem is to determine which of these two categories the input function belongs to: constant or balanced.

> **Deutsch's problem**
>
> Input: a function $f : \{0, 1\} \to \{0, 1\}$
> Output: 0 if f is constant, 1 if f is balanced

If we view the input function f in Deutsch's problem as representing random access to a string, we're thinking about a two-bit string: $f(0)f(1)$.

function	string
f_1	00
f_2	01
f_3	10
f_4	11

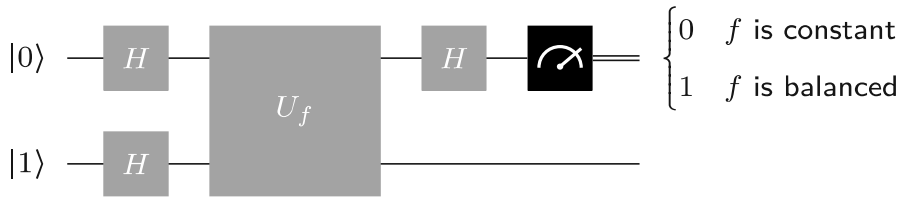
When viewed in this way, Deutsch's problem is to compute the parity (or, equivalently, the exclusive-OR) of the two bits.

Every classical query algorithm that correctly solves this problem must query both bits: $f(0)$ and $f(1)$. If we learn that $f(1) = 1$, for instance, the answer could still be 0 or 1, depending on whether $f(0) = 1$ or $f(0) = 0$, respectively. Every other case is similar; knowing just one of two bits doesn't provide any information at all about their parity. So, the Boolean circuit described in the previous section is the best we can do in terms of the number of queries required to solve this problem.

Quantum circuit description

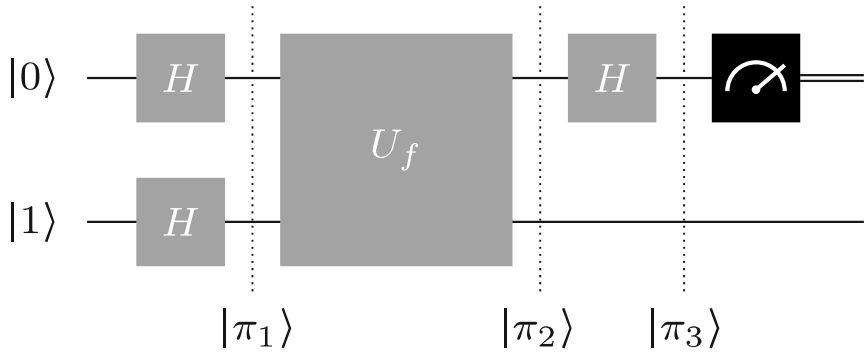
Deutsch's algorithm solves Deutsch's problem using a single query, therefore providing a quantifiable advantage of quantum over classical computations. This may be a modest advantage — one query as opposed to two — but we have to start somewhere. Scientific advances sometimes have seemingly humble origins.

Here is a quantum circuit that describes Deutsch's algorithm:



Analysis

To analyze Deutsch's algorithm, we will trace through the action of the circuit above and identify the states of the qubits at the times suggested by this figure:



The initial state is $|1\rangle|0\rangle$, and the two Hadamard operations on the left-hand side of the circuit transform this state to

$$|\pi_1\rangle = |-\rangle|+\rangle = \frac{1}{2}\big(|0\rangle - |1\rangle\big)|0\rangle + \frac{1}{2}\big(|0\rangle - |1\rangle\big)|1\rangle.$$

(As always, we're following Qiskit's qubit ordering convention, which puts the top qubit to the right and the bottom qubit to the left.)

Next, the U_f gate is performed. According to the definition of the U_f gate, the value of the function f for the classical state of the top/rightmost qubit is XORed onto the bottom/leftmost qubit, which transforms $|\pi_1\rangle$ into the state

$$|\pi_2\rangle = \frac{1}{2}\big(|0 \oplus f(0)\rangle - |1 \oplus f(0)\rangle\big)|0\rangle + \frac{1}{2}\big(|0 \oplus f(1)\rangle - |1 \oplus f(1)$$

We can simplify this expression by observing that the formula

$$|0 \oplus a\rangle - |1 \oplus a\rangle = (-1)^a\big(|0\rangle - |1\rangle\big)$$

works for both possible values $a \in \Sigma$. More explicitly, the two cases are as follows.

$$|0 \oplus 0\rangle - |1 \oplus 0\rangle = |0\rangle - |1\rangle = (-1)^0\big(|0\rangle - |1\rangle\big)$$
$$|0 \oplus 1\rangle - |1 \oplus 1\rangle = |1\rangle - |0\rangle = (-1)^1\big(|0\rangle - |1\rangle\big)$$

Thus, we can alternatively express $|\pi_2\rangle$ like this:

$$\begin{aligned} |\pi_2\rangle &= \frac{1}{2}(-1)^{f(0)}\big(|0\rangle - |1\rangle\big)|0\rangle + \frac{1}{2}(-1)^{f(1)}\big(|0\rangle - |1\rangle\big)|1\rangle \\ &= |-\rangle\left(\frac{(-1)^{f(0)}|0\rangle + (-1)^{f(1)}|1\rangle}{\sqrt{2}}\right). \end{aligned}$$

Something interesting just happened! Although the action of the U_f gate on standard basis states leaves the top/rightmost qubit alone and XORs the function value onto the bottom/leftmost qubit, here we see that the

state of the top/rightmost qubit has changed (in general) while the state of the bottom/leftmost qubit remains the same — specifically being in the $|-\rangle$ state before and after the U_f gate is performed. This phenomenon is known as the *phase kickback,* and we will have more to say about it shortly.

With one final simplification, which is to pull the factor of $(-1)^{f(0)}$ outside of the sum, we obtain this expression of the state $|\pi_2\rangle$:

$$\begin{aligned} |\pi_2\rangle &= (-1)^{f(0)}|-\rangle\left(\frac{|0\rangle + (-1)^{f(0)\oplus f(1)}|1\rangle}{\sqrt{2}}\right) \\ &= \begin{cases} (-1)^{f(0)}|-\rangle|+\rangle & \text{if } f(0)\oplus f(1) = 0 \\ (-1)^{f(0)}|-\rangle|-\rangle & \text{if } f(0)\oplus f(1) = 1. \end{cases} \end{aligned}$$

Notice that in this expression, we have $f(0)\oplus f(1)$ in the exponent of -1 as opposed to $f(1) - f(0)$, which is what we might expect from a purely algebraic viewpoint, but we obtain the same result either way. This is because the value $(-1)^k$ for any integer k depends only on whether k is even or odd.

Applying the final Hadamard gate to the top qubit leaves us with the state

$$|\pi_3\rangle = \begin{cases} (-1)^{f(0)}|-\rangle|0\rangle & \text{if } f(0)\oplus f(1) = 0 \\ (-1)^{f(0)}|-\rangle|1\rangle & \text{if } f(0)\oplus f(1) = 1, \end{cases}$$

which leads to the correct outcome with probability 1 when the right/topmost qubit is measured.

Further remarks on the phase kickback

Before moving on, let's look at the analysis above from a slightly different angle that may shed some light on the phase kickback phenomenon.

First, notice that the following formula works for all choices of bits $b, c \in \Sigma$.

$$|b\oplus c\rangle = X^c|b\rangle$$

This can be verified by checking it for the two possible values $c = 0$ and $c = 1$:

$$|b \oplus 0\rangle = |b\rangle = \mathbb{I}|b\rangle = X^0|b\rangle$$
$$|b \oplus 1\rangle = |\neg b\rangle = X|b\rangle = X^1|b\rangle.$$

Using this formula, we see that

$$U_f\big(|b\rangle|a\rangle\big) = |b \oplus f(a)\rangle|a\rangle = \big(X^{f(a)}|b\rangle\big)|a\rangle$$

for every choice of bits $a, b \in \Sigma$. Because this formula is true for $b = 0$ and $b = 1$, we see by linearity that

$$U_f\big(|\psi\rangle|a\rangle\big) = \big(X^{f(a)}|\psi\rangle\big)|a\rangle$$

for all qubit state vectors $|\psi\rangle$, and therefore

$$U_f\big(|-\rangle|a\rangle\big) = \big(X^{f(a)}|-\rangle\big)|a\rangle = (-1)^{f(a)}|-\rangle|a\rangle.$$

The key that makes this work is that $X|-\rangle = -|-\rangle$. In mathematical terms, the vector $|-\rangle$ is an *eigenvector* of the matrix X having *eigenvalue* -1.

We'll discuss eigenvectors and eigenvalues in greater detail in the upcoming lesson on *Phase estimation and factoring,* where the phase kickback phenomenon is generalized to other unitary operations.

Keeping in mind that scalars float freely through tensor products, we find an alternative way of reasoning how the operation U_f transforms $|\pi_1\rangle$ into $|\pi_2\rangle$ in the analysis above:

$$\begin{aligned}
|\pi_2\rangle &= U_f\big(|-\rangle|+\rangle\big) \\
&= \frac{1}{\sqrt{2}}U_f\big(|-\rangle|0\rangle\big) + \frac{1}{\sqrt{2}}U_f\big(|-\rangle|1\rangle\big) \\
&= |-\rangle\left(\frac{(-1)^{f(0)}|0\rangle + (-1)^{f(1)}|1\rangle}{\sqrt{2}}\right).
\end{aligned}$$

Implementation in Qiskit

Now let's see how we can implement Deutsch's algorithm in Qiskit. We'll start with a version check and then perform the imports required solely for this implementation. For the implementations of other algorithms that follows, we'll perform the required imports separately for the sake of greater modularity.

```
from qiskit import __version__

print(__version__)
```

Output:

```
2.1.1
```

```
from qiskit import QuantumCircuit
from qiskit_aer import AerSimulator
```

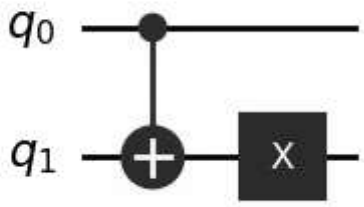
First we'll define a quantum circuit that implements a query gate for one of the four functions f_1, f_2, f_3, or f_4 from one bit to one bit described previously. As we already mentioned, the implementation of query gates is not really a part of Deutsch's algorithm itself; here we're essentially just showing one way to prepare the input, in the form of a circuit implementation of a query gate.

```
def deutsch_function(case: int):
    # This function generates a quantum circuit for or
    # from one bit to one bit

    if case not in [1, 2, 3, 4]:
        raise ValueError("`case` must be 1, 2, 3, or 4

    f = QuantumCircuit(2)
    if case in [2, 3]:
        f.cx(0, 1)
    if case in [3, 4]:
        f.x(1)
    return f
```

We can see what each circuit looks like using the `draw` method. Here's the circuit for the function f_3.

```
display(deutsch_function(3).draw(output="mpl"))
```

Output:



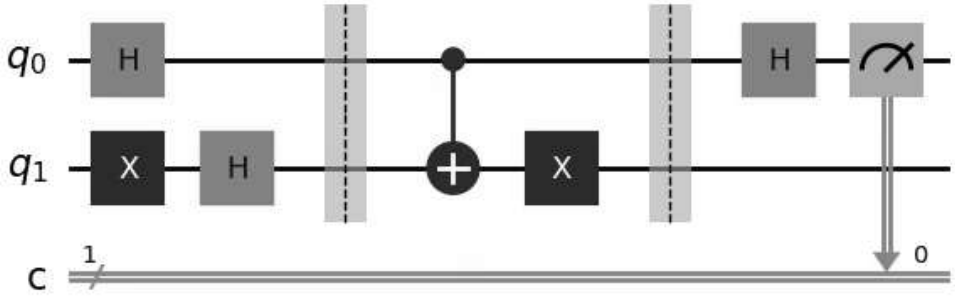
Next we will create the actual quantum circuit for Deutsch's algorithm, substituting the query gate with a quantum circuit implementation given as an argument. Shortly we'll plug in one of the four circuits defined by the function `deutsch_function` we defined earlier. Barriers are included to show the visual separation between the query gate implementation and the rest of the circuit.

```
def compile_circuit(function: QuantumCircuit):
    # Compiles a circuit for use in Deutsch's algorit

    n = function.num_qubits - 1
    qc = QuantumCircuit(n + 1, n)

    qc.x(n)
    qc.h(range(n + 1))

    qc.barrier()
    qc.compose(function, inplace=True)
    qc.barrier()

    qc.h(range(n))
    qc.measure(range(n), range(n))

    return qc
```

Again we can see what the circuit looks like using the `draw` method.

```
display(compile_circuit(deutsch_function(3)).draw(outp
```

Output:



Finally, we'll create a function that runs the circuit previously defined one time and outputs the appropriate result: "constant" or "balanced."

```
def deutsch_algorithm(function: QuantumCircuit):
    # Determine if a one-bit function is constant or r

    qc = compile_circuit(function)

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

We can now run Deutsch's algorithm on any one of the four functions defined above.

```
f = deutsch_function(3)
display(deutsch_algorithm(f))
```

Output:

```
'balanced'
```

Was this page helpful?

Yes No

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

Previous page Next page