QuantumPath Tutorials

Implementation of the Deutsch-Jozsa algorithm in qSOA®

Contents

1 The Deutsch-Jozsa algorithm

1.1 The problem

The Deutsch - Jozsa algorithm was the first example of a quantum algorithm that performed better than any classical algorithm, showing the world the clear advantages of using a quantum machine as a computational tool for a specific problem.

Let an unknown Boolean function be such that $f : \{0,1\}^n \rightarrow \{0,1\}$, with the assurance to be constant ($f(x) = a, \forall x \in \{0,1\}^n$, and for some $a \in (0,1)$), or balanced (the function returns 0 for half of the values and returns 1 for the other half). Then, the task of the algorithm is to determine whether the function is balanced or constant.

1.2 Classical solution

Classically, in the best case, two queries to the oracle can determine if the hidden boolean function, $f(x)$, is balanced: e.g., if we get both $f(0,0,0,\dots) \rightarrow 0$ and $f(1,0,0,\dots) \rightarrow 1$, then we know the function is balanced, as we have obtained the two different outputs.

In the worst case, if we continue to see the same output for each input that we try, we will have to check exactly half of all the possible inputs plus one to be sure of $f(x)$ being constant. As the total number of possible inputs is $2n$, this implies that we need $2^{n-1} + 1$ trial inputs to be certain that $f(x)$ is constant in the worst case.

For example, for a $4-$bit string, if we check 8 out of the 16 possible combinations, getting all 0's, it is still possible that the $9th$ input returns a 1 and $f(x)$ is balanced. Probabilistically, this is a very unlikely event. In fact, if we get the same as a continuum, we can express the probability of the function being constant as a function of k inputs as:

$$P_{constant}(k) = 1 - \frac{1}{2^{k-1}} \quad for\ 1 < k \leq 2^{n-1}$$

Realistically, we could opt to truncate our classical algorithm early, for example, if we were over x% confident. But if we want to be 100% confident, we will need to check 2^{n-1} inputs.

1.3 Quantum solution

To apply Deutsch-Jozsa's algorithm to a function $f : \{0,1\}^n \rightarrow \{0,1\}$, we need to have implemented it as an oracle, i.e., as a *black box*, U_f, with U_f such that

$$|x\rangle\,|y\rangle \rightarrow |x\rangle\,|f(x) \oplus y\rangle$$

We take as input of the circuit a first register of n qubits initialized in $|0\rangle^{\otimes n}$ and a second auxiliary register of a single qubit initialized in $|1\rangle$. We apply Hadamard gates on both registers, and make a call to the oracle. Then, we apply Hadamard gates on the first register and measure it, as we can see in Figure 1.

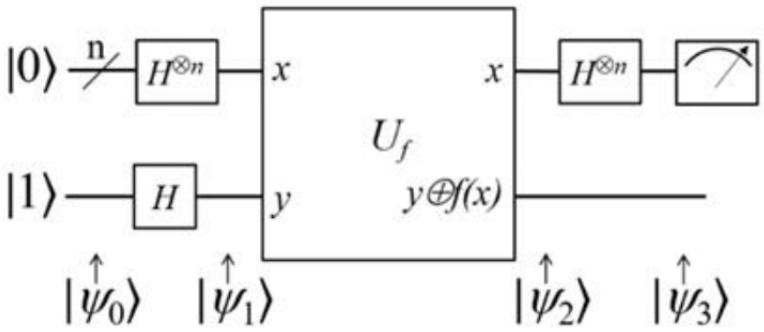


Figure 1: The Deutsch-Jozsa algorithm

So, before the measurement, ignoring the second register, the state of our circuit is:

$$|\Psi\rangle = \frac{1}{2^n}\sum_{y=0}^{2^n-1}\left[\sum_{x=0}^{2^n-1}(-1)^{f(x)}(-1)^{x}\right]|y\rangle \tag{1}$$

where the probability of measuring $|0\rangle^{\otimes n}$ is:

$$\left|\frac{1}{2^n}\sum_{x=0}^{2^n-1}(-1)^{f(x)}\right|^2$$

so that it is 0 if f is constant, and 1 if f is balanced. That is, our function is constant if and only if the returned measurement is $|0\rangle^{\otimes n}$.

In this tutorial we will execute the following function, and figure out whether it is balanced or constant.

$$f : (x_1, x_2, ...x_8) \rightarrow x_1 \oplus x_2 \oplus x_1x_2 \oplus x_3 \oplus x_1x_3 \oplus x_1x_2x_3 \oplus \bar{x_1}\bar{x_2}\bar{x_3} = 1$$

for every $(x_1, x_2, ...x_8) \in \{0,1\}^8$ where, given $x \in \{0,1\}$, $\bar{x}$ is the complementary, i.e. $\bar{x} = 1 \oplus x$, $\forall x \in \{0,1\}$.

It is easy to see that we are working with a constant function. Therefore, given what has been explained before, the expected result for this tutorial is the zero state, $|0\rangle^{\otimes n}$.

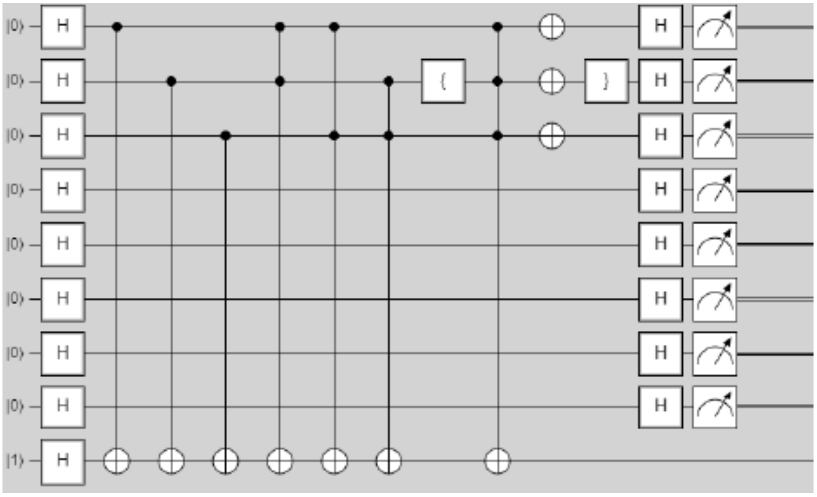


Figure 2: Deutsch-Jozsa algorithm in QPath®

1.4 Usefulness of the algorithm

This algorithm was the first in which a large difference in complexity was observed with respect to its classical version. To solve the problem classically, it is necessary to evaluate, in the worst case, $N/2+1$ times the function, with $N = 2^n$ being the size of the input domain, so its time complexity is $\mathcal{O}$(N/2+1). Quantumly, however, one can solve the problem with 100% confidence with a single call to the oracle (our unknown function), so its time complexity is reduced to $\mathcal{O}(1)$.

2 Implementation of the algorithm in qSOA®

Once we are aware of how the algorithm works, we are ready to implement it in qSOA®. This will allow us to create the circuit and execute it in different quantum computing providers, among many other things.

The process of implementing an algorithm in qSOA® is comprised of four steps:

1. Setting up qSOA® and selecting the quantum solution
2. Creating a circuit with the algorithm and assigning it to the solution
3. Introducing a circuit flow to control the number of launches of the algorithm
4. Executing the flow on different quantum devices.

As can be seen in qSOA's® manual, ref.[1], there are multiple ways to secure the connection depending on the context. Following the best practices of qSOA®, in this tutorial we are integrating the security in the code. Similarly, one can work with asynchronous or synchronous programming. Keeping in mind the purpose of this tutorial we will use the synchronous version.

2.1 Setting up qSOA®

Firstly, we import the SDK that has been previously installed, see ref.[1], and create the qSOA® workspace to work with.

[1]:
```
from QuantumPathQSOAPySDK import QSOAPlatform  # Import SDK
```

[2]:
```
# Create qSOA workspace, login manually
qsoa = QSOAPlatform()

username = 'username'
password = 'password' # password encrypted in SHA-256

authenticated = qsoa.authenticateEx(username, password)

print('Authentication completed:', authenticated)
```

```
Authentication completed: True
```

Then we review the existing solutions and select the one we are interested in.

[3]:
```
# Get catalogs
solutionList = qsoa.getQuantumSolutionList()
print("   ",solutionList)
idSolution = int(input("Select idSolution: "))
```

```
    {'12345': 'QS_GateTutorials'}
Select idSolution: 12345
```

2.1.1 Securing the connection

As has been said, qSOA® allows multiple business development contexts. Therefore, the user can secure the connection through a configuration file, *.qpath*, useful at a personal level, or in parameterized way, as it is done here.

2.2 Assigning a circuit to the solution

Once the solution has been selected, we must link it to the circuit that we are interested in implementing. Therefore, we first need to define the circuit and create it.

2.2.1 Defining the circuit

As we are working with a gates circuit, we can either enter the circuit in *visual language*, VL, or *intermediate language*, IL. In this case we are going to work with VL and, therefore, we can either enter the circuit as a *String* or as a *CircuitGates* object, which is what we are going to go for.

In order to define the circuit shown in figure 2, we just need to write the gates we want to use in the correspondent qubits.

[4]:
```
## Defining circuit w/ Circuit Gates
circuitG = qsoa.CircuitGates()
n = 9 # Number of qubits
circuitG.x(n-1)
circuitG.h(list(range(n)))
for i in range(3):
    circuitG.cx(i,n-1)
circuitG.mcg([0, 1], circuitG.x(n-1, False))
circuitG.mcg([0, 2], circuitG.x(n-1, False))
circuitG.mcg([2, 1], circuitG.x(n-1, False))
for i in range(2):
    circuitG.mcg([0, 1, 2], circuitG.x(n-1, False))
    circuitG.x([0, 1, 2])
circuitG.h()
circuitG.measure()

print(circuitG.getCircuitBody())
```

```
[['H', 'H', 'H', 'H', 'H', 'H', 'H', 'H', 'X'], ['1', '1', '1', '1', '1', '1', '1',
'1', 'H'], ['CTRL', 1, 1, 1, 1, 1, 1, 1,'X'], [1, 'CTRL', 1, 1, 1, 1, 1, 1, 'X'],
[1, 1, 'CTRL', 1, 1, 1, 1, 1, 'X'], ['CTRL', 'CTRL', 1, 1, 1, 1, 1, 1, 'X'],
['CTRL', 1, 'CTRL', 1, 1, 1, 1, 1, 'X'], [1, 'CTRL', 'CTRL', 1, 1, 1, 1, 1, 'X'],
['CTRL', 'CTRL', 'CTRL', 1, 1, 1, 1, 1, 'X'], ['X', 'X', 'X'], ['CTRL', 'CTRL',
'CTRL', 1, 1, 1, 1, 1, 'X'], ['X', 'X', 'X', 'H', 'H', 'H', 'H', 'H', 'H'], ['H',
'H', 'H'], ['Measure'], [1, 'Measure'], [1, 1, 'Measure'], [1, 1, 1, 'Measure'],
[1, 1, 1, 1, 'Measure'], [1, 1, 1, 1, 1, 'Measure'], [1, 1, 1, 1, 1, 1,
'Measure'], [1, 1, 1, 1, 1, 1, 1, 'Measure'], [1, 1, 1, 1, 1, 1, 1, 1, 'Measure']]
```

Note: Every gate can be applied to a single qubit, a list of them, introduced as a list, or to every qubit in the circuit, using ().

2.2.2 Creating the circuit

In order to create the circuit we are going to use the *createAssetSync* function. This function receives the following fields as inputs:

- idSolution: to associate the circuit with the solution we have selected before.
- assetName: to set the name of the circuit.
- assetNamespace: to associate the circuit with a class of circuits that share something in common. In this case, we associate the circuit with a set of basic circuits.
- assetDescription: to write a brief description of the algorithm.
- assetBody: to select the circuit we have previously defined.
- assetType: to select if we are working with a gates circuit or an annealing one.

- assetLevel: to select either visual language or intermediate language, according to the definition of the circuit.

[5]:
```
## Circuit creation
assetName = 'QC_qSOA_DJ'
assetNamespace = 'Manual.Gates.DJ'
assetDescription = 'Creating the DJ circuit from qSOA'

assetBody = circuitG
assetType = 'GATES'
assetLevel = 'VL'

CircuitManagementResult = qsoa.createAssetSync(idSolution, assetName,
assetNamespace, assetDescription, assetBody, assetType, assetLevel)
```

Note: Note that the synchronous version of this function is being used. This is because the *createAsset* function creates, compiles and transpiles the asset, and the synchronous function waits for all of it to be done before moving on. For a better understanding of how this functions works, we recommend using the asynchronous version, *createAsset*, and the *getAssetManagementResult* function.

2.3 Assigning a circuit flow to the circuit

The quantum flow is a box diagram that will allow us to control the number of runs of our algorithm. This is very useful, as the results obtained in quantum computing have a certain probability associated with them. That means that the more times we run the algorithm, the more robust the results will be.

2.3.1 Defining the flow

As happens with the circuit, the flow can be written in VL or IL. If we choose to do it with VL then it can be entered as a *String* or a *CircuitFlow* object. Otherwise, it can only be entered as a *String*. For this example we are choosing VL and *CircuitFlow*.

In order to define a flow we need:

1. Starting node
2. Initializing node: usually set to 0
3. Circuit node: where we write the circuit we want lo launch
4. Repeat node: where the number of repetitions can be establish
5. End node
6. Links between each node we have created

[8]:
```
# Defining flow w/ CircuitFlow
flow = qsoa.CircuitFlow()
startNode = flow.startNode()
initNode = flow.initNode(0)
circuitNode = flow.circuitNode('Manual.Gates.DJ.QC_qSOA_DJ')
# Namespace + CircuitName
repeatNode = flow.repeatNode(1000)
endNode = flow.endNode()

flow.linkNodes(startNode, initNode)
flow.linkNodes(initNode, circuitNode)
flow.linkNodes(circuitNode, repeatNode)
flow.linkNodes(repeatNode, endNode)
print(flow.getFlowBody())
```

[8]:
```
{'class': 'go.GraphLinksModel', 'nodeDataArray': [{'category': 'Start', 'text':
'Start', 'key': -1, 'loc': ''}, {'category': 'Init', 'text': '0', 'key': -2,
'loc': ''},{'category': 'Circuit', 'text': 'Manual.Gates.DJ.QC_qSOA_DJ', 'key':␣
 ↪-3, 'loc': ''}, {'category': 'Repeat', 'text':
'1000', 'key': -4, 'loc': ''}, {'category': 'End', 'text': 'End', 'key': -5,
'loc': ''}], 'linkDataArray': [{'from': -1, 'to': -2, 'points': []}, {'from':
-2, 'to': -3, 'points': []}, {'from': -3, 'to': -4, 'points': []}, {'from': -4,
'to': -5, 'points': []}]}
```

2.3.2 Creating the flow

In order to create the flow we are using *createAssetFlowSync*, although the *createAssetSync* function would also work. The inputs that this function requires are:

- idSolution: to associate the flow with the solution we have selected before.
- assetName: to set the name of the flow.
- assetNamespace: to associate the flow with a class of flows that share something in common. In this case, we associate the flow with a set of basic flows.
- assetDescription: to write a brief description of the algorithm.
- assetBody: to select the flow we have previously defined.
- assetLevel: to select either VL or IL, according to the definition of the flow.
- publish: to select if we want to publish the flow on qSOA or not.

[9]:
```
## Flow creation
assetName = 'QF_qSOA_DJ'
assetNamespace = 'Manual.Gates.DJ'
assetDescription = 'Creating the DJ flow from qSOA'
assetPublication = True
```

```
assetBody = flow
assetType = 'FLOW'
assetLevel = 'VL'

FlowManagementResult = qsoa.createAssetFlowSync(idSolution, assetName,
assetNamespace, assetDescription, assetBody, assetLevel,assetPublication)
```

Note: Note that the synchronous version of this function is being used. This is because the *createAssetFlow* function creates, compiles and transpiles the flow, and the synchronous function waits for all of it to be done before moving on. For a better understanding of how this functions works, we recommend using the asynchronous version, *createAssetFlow*, and the *getAssetManagementResult* function.

2.4 Implementation of the algorithm

Note: If you would like to execute any other previously created flow, you can do so by implementing this part of the tutorial. You will need to specify the ID Solution and the ID of the flow.

2.4.1 Executing the algorithm

We are now ready to execute the algorithm, so lets see on what platforms we can do so, and select the ones we are interested in.

[12]:
```
deviceList = qsoa.getQuantumDeviceList(idSolution)
print('Device List:', deviceList)
DeviceID = input("Select a device to run the flow in: ")
```

```
Device List: {'14': 'AMAZON BRAKET 25qbits Local Simulator', '2': 'QISKIT Local
Simulator', '1': 'Q# Local Simulator Framework'}
Select a device to run the flow in: 14
```

Now, we proceed to run the quantum algorithm with the *runQuantumApplicationSync* function.

[13]:
```
exe_application = qsoa.runQuantumApplicationSync('NameTheTask', idSolution,␣
↪FlowID, DeviceID)
```

Note: Note that the synchronous version of this function is being used. This is because *runQuantumApplication* launches an execution, and the synchronous function waits for the execution to be done before moving on. If we do not know how long the execution is going to take (due to execution itself or to the queue), or if it takes too long, the asynchronous version presents a clear advantage. However, for a better understanding of how this functions work, we recommend using the asynchronous version, *runQuantumApplication*.

The function *runQuantumApplicationSync* gives an application object as output. For us to manage the results we need the *getQuantumExecutionResponse* function that returns the results as an

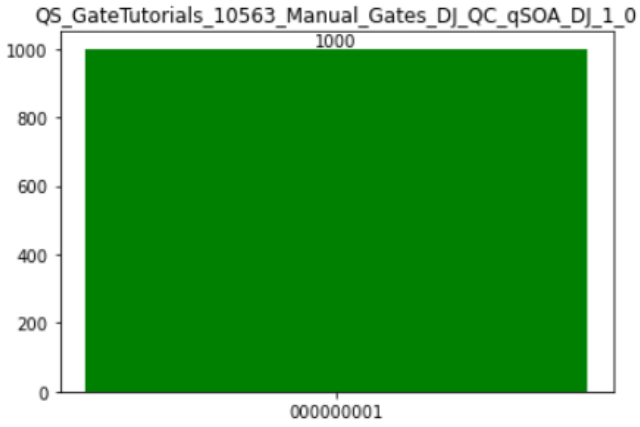
execution object.

[14]:
```
restok_execution = qsoa.getQuantumExecutionResponse(exe_application.
 ↪getExecutionToken(), idSolution, FlowID)
restok_histogram = restok_execution.getHistogram()
print(restok_histogram)
```

```
{'QS_GateTutorials_12345_Manual_Gates_DJ_QC_qSOA_DJ_1_0': {'000000001': 1000}}
```

Now that we have the results we asked for, we can proceed to represent them with the function *representResults*.

[15]:
```
# Circuit gate representation
gates_representation = qsoa.representResults(restok_execution)
```



2.4.2 Multiple devices

If we want to run the algorithm in multiple devices at the same time, we can do so by creating an array with the information needed and proceeding the same way as before.

[16]:
```
deviceList = qsoa.getQuantumDeviceList(idSolution)
print('Device List:', deviceList)
```

```
Device List: {'14': 'AMAZON BRAKET 25qbits Local Simulator', '2': 'QISKIT Local
Simulator', '1': 'Q# Local Simulator Framework'}
```

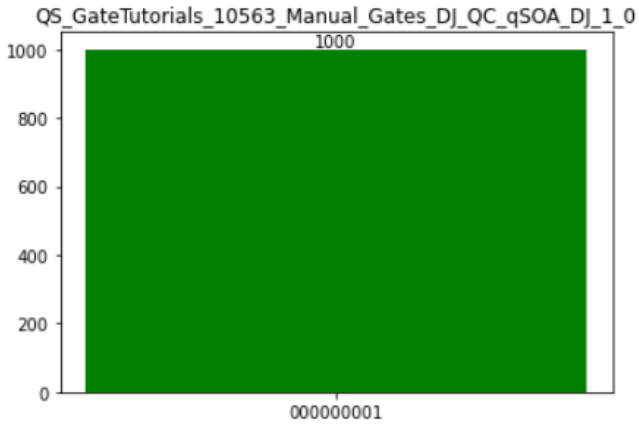
[17]:
```
## Run Quantum Gates Application
exe_ApplicationNames = ['Task_Amazonsim','Task_Qiskitsim']
exe_IdDevices = [14,2]

exe_Applications = [0] * len(exe_ApplicationNames)

for i in range(len(exe_ApplicationNames)):
```

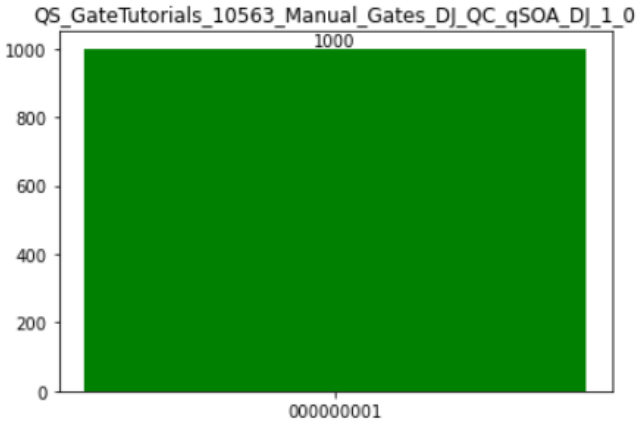
```
    exe_Applications[i] = qsoa.runQuantumApplicationSync(exe_ApplicationNames[i],
      idSolution, FlowID, exe_IdDevices[i])
```

[18]:
```
# Get quantum execution response with execution token
restok_Executions = [0] * len(exe_ApplicationNames)
restok_Histograms = [0] * len(exe_ApplicationNames)

for i in range(len(exe_ApplicationNames)):
    restok_Executions[i] = qsoa.getQuantumExecutionResponse(exe_Applications[i].
 ↪getExecutionToken(), idSolution, FlowID)
    restok_Histograms[i] = restok_Executions[i].getHistogram()

print(restok_Histograms)
```

```
[{'QS_GateTutorials_12345_Manual_Gates_DJ_QC_qSOA_DJ_1_0': {'000000001': 1000}},
{'QS_GateTutorials_12345_Manual_Gates_DJ_QC_qSOA_DJ_1_0': {'000000001': 1000}}]
```

[19]:
```
# Circuit gate representation
for i in range(len(exe_ApplicationNames)):
    Gates_Representations = qsoa.representResults(restok_Executions[i])
```





As it has been shown, the measurement obtained is $|0\rangle^{\oplus n}$, so the function f associated with the oracle U_f in this example is a constant function, just as we expected.

References

[1] aQuantum, *QPath® Python SDK User Guide.* Available on QPath®.

