Implementation of the Bernstein-Vazirani algorithm in qSOA®

Contents

1 The Bernstein-Vazirani algorithm

1.1 The problem

Suppose we have a function $f_s : \{0,1\}^n \to \{0,1\}$, such that $f_s(\vec{x}) = \vec{s} \cdot \vec{x} \text{ mod} 2$, where $mod2$ accounts for the modulo operation (that is, it returns the remainder of dividing the result by 2). So we have a function that takes as an input a string of bits, and returns either 0 or 1. This function can be thought of as a black box, as we know the outputs for each input, but not the function form since $\vec{s}$ is unknown. So our goal will be to find the secret vector $\vec{s}$.

1.2 Classical solution

One simple approach to find $\vec{s}$ is to evaluate the quantities:

$$\begin{aligned} f_s(100\ldots0) &= s_1 \\ f_s(010\ldots0) &= s_2 \\ f_s(001\ldots0) &= s_3 \\ &\vdots \\ f_s(000\ldots1) &= s_n \end{aligned} \tag{1}$$

So that within n queries we would be able to find $\vec{s}$. In contrast, only one query is needed to find $\vec{s}$ using a quantum algorithm.

1.3 Quantum solution

We can see how the quantum algorithm works for a vector $\vec{s}$ of length 4 in figure 1.

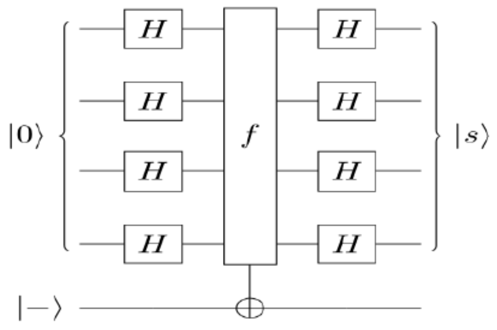


Figure 1: The Berstein-Vazirani algorithm

As can be seen, we have four input qubits (the same as the length of the string) and one auxiliary qubit. So the steps followed by the algorithm are:

1. Initialize the input qubits in the $|0\rangle$ state and the auxiliary qubit in the $|-\rangle$ state
2. Apply Hadamard gates to the input register

3. Query the oracle
4. Apply Hadamard gates to the input register
5. Measure

It can be shown mathematically, ref.[2], that by following this steps we arrive to the $|s\rangle$ state or in other words we get to known the hidden vector $\vec{s}$. In figure 2, we can see an example of the implementation of the algorithm for a string 1001.

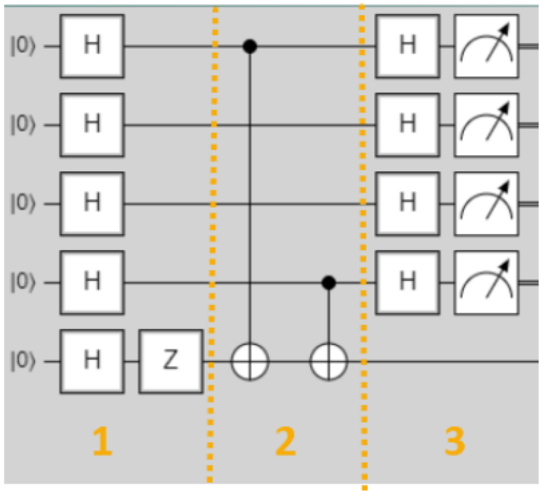


Figure 2: Berstein-Vazirani algorithm in QPath® for $x = 1001$

We can see how the $|-\rangle$ state is prepared, applying a Hadamard gate and a Z gate to the $|0\rangle$ state. The oracle is marked inside the yellow lines and we see that it works by applying a CNOT gate to the auxiliary qubit in the qubits corresponding to the positions where there is a 1 in the string. In this case it applies to the first and the last qubit.

1.4 Usefulness of the algorithm

Given that the secret string needs to be known in order to build the circuit, it is clear that this algorithm does not have any real-world version. However, the fact that it can solve a problem for any vector size in a single query shows us how big the potential of quantum computing is, and the speed-up it could bring to all tasks.

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 = 5  # Number of qubits
circuitG.h(list(range(n)))
circuitG.z(n-1)
circuitG.cx(0,n-1)
circuitG.cx(n-2,n-1)
circuitG.z(n-1)
circuitG.h()
circuitG.measure()

print(circuitG.getCircuitBody())
```

```
[['H', 'H', 'H', 'H', 'H'], [1, 1, 1, 1, 'Z'], ['CTRL', 1, 1, 1, 'X'], [1, 1, 1,
'CTRL', 'X'], ['H', 'H', 'H', 'H', 'Z'], [1, 1, 1, 1, 'H'], ['Measure'], [1,
'Measure'], [1, 1, 'Measure'], [1, 1, 1, 'Measure'], [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_BV'
assetNamespace = 'Manual.Gates.BV'
assetDescription = 'Creating the BV 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.BV.QC_qSOA_BV')
     # 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.BV.QC_qSOA_BV', '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_BV'
     assetNamespace = 'Manual.Gates.BV'
     assetDescription = 'Creating the BV 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

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 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*.

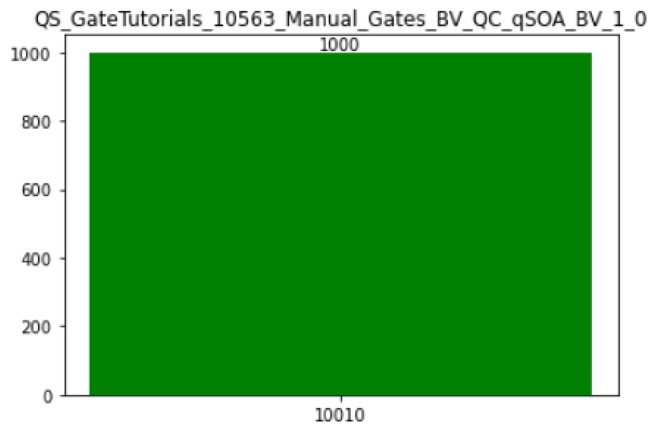
The function *runQuantumApplicationSync* gives an application object as output. For us to manage the results we need the *getQuantumExecutionResponse* function that returns the results as an execution object.

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

```
{'QS_GateTutorials_12345_Manual_Gates_BV_QC_qSOA_BV_1_0': {'10010': 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)
```

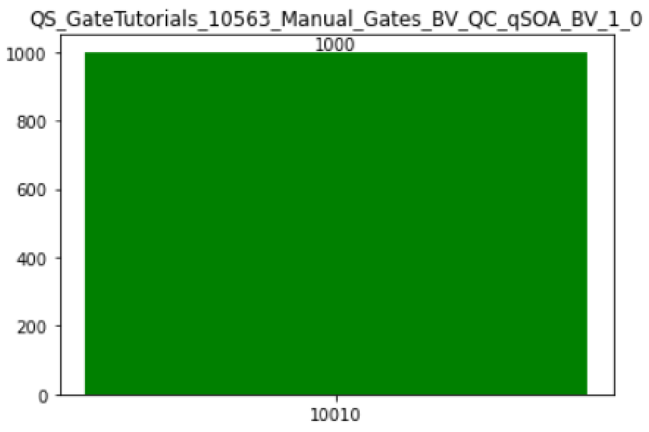
```
Device List: {'14': 'AMAZON BRAKET 25qbits Local Simulator', '2': 'QISKIT Local
Simulator', '1': 'Q# Local Simulator Framework'}
```

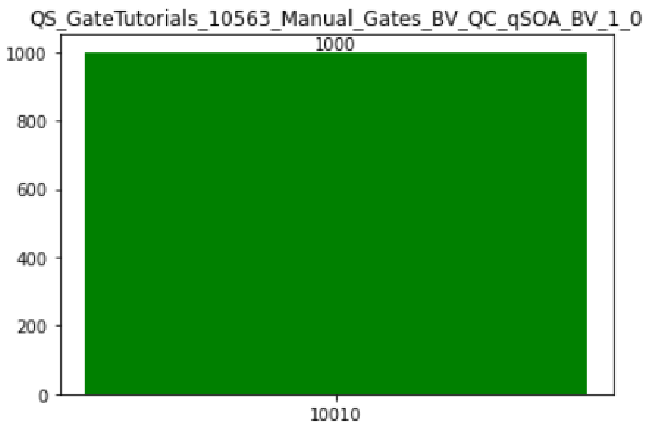
```
[17]: ## Run Quantum Gates Application
      exe_ApplicationNames = ['Task_Amazonsim','Task_Qiskitsim']
      exe_IdDevices = [14,2]

      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_BV_QC_qSOA_BV_1_0': {'10010': 1000}},
{'QS_GateTutorials_12345_Manual_Gates_BV_QC_qSOA_BV_1_0': {'01001': 1000}}]
```

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





We now proceed to analyze the results obtained in the simulation with just one device. It can be done analogously with the rest and the results should be equivalent.

As expected the measured result is the state $|10010\rangle$, where the four initial qubits represent the string we were looking for 1001, and the fifth one represents the state $|0\rangle$, as the inverted gates have been applied.

References

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

[2] IBM, *Qiskit Textbook.* Available online at: `https://qiskit.org/textbook/ch-algorithms/bersntein-vazirani.html`