Implementation of the Map coloring problem in qSOA®

Contents

1 The map coloring problem

1.1 The problem

For a map divided into N regions and a list of K colors, the map coloring problem seeks to assign a color to each of these regions so that no adjacent regions have the same color. Mathematically, the map can be represented as an adjacency matrix, A, such that:

$$f(x) = \begin{cases} A_{ij} = 1 & \text{if regions i, j are adjacent} \\ A_{ij} = 0 & \text{otherwise} \end{cases}$$

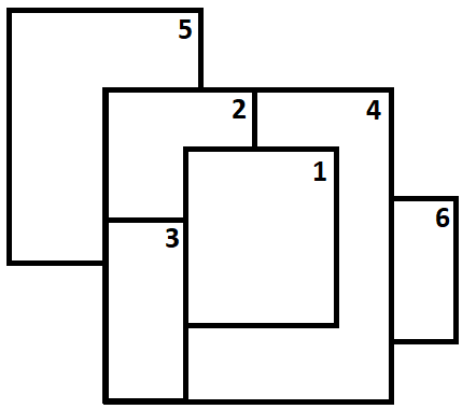
Therefore, our variables will be: $x_{ik} = 1$ if the region i is assigned the color k, and $x_{i,k}$ otherwise.

This will lead us to the following Hamiltonian:

$$H = \lambda_1 \sum_{i=1}^{N} \left(1 - \sum_{k=1}^{K} x_{ik}\right)^2 + \lambda_2 \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} \sum_{k=1}^{K} A_{ij} x_{ik} x_{jk} \tag{1}$$

Where the first constraint says that each region can be colored with one, and only one, color, and the second introduces a penalty each time two adjacent regions are assigned the same color. It must be noticed that the first constraint must always be verified (it makes no sense that a region is assigned more than one color), so we must set $\lambda_1 > \lambda_2$, so that we prioritise it.

The example we are going to work with has six different regions, as shown below, and 4 possible colors: red, yellow, green and blue.



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_AnnealingTutorials'}
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 an annealing circuit, we can either enter the circuit in *visual language*, VL, or *intermediate language*, IL. Either way the circuit has to be entered as a *String*, as shown below.

To establish our circuit we can define the following assets:

1. Parameters: we will define every scalar in our problem as a parameter
2. Auxiliary data: the rest of our parameters that aren't scalar will be defined here
3. Classes: elements that form the variables
4. Variables
5. Rules

In our case, the scalar parameters, which are going to be introduced as *Parameters*, are the number of regions, N, and the number of colors available, K. On the other hand, the adjacency matrix, A, is an *Auxiliary data*, and is introduced as a list.

Once we have all the parameters implemented, we define the classes and variables. In this problem our variables are going to be $x_{i,k}$, that depend on the different regions and colors, wo we need to add those two as classes.

When this is done, we implement the rules that we have talked about before. The rules are introduced as follows:

RULE(Name of the rule | "Description of the rule" | "Value of lambda" | {Mathematical expression});

and every mathematical expression in a rule can be defined with the following commands:

1. *SUMMATORY(from where to where | What wants to be summed)*
2. *SQUARED(What wants to be squared)*
3. *LINEAR(Variable | "Product")*
4. *OFFSET("Value")*
5. *QUADRATIC(Variable1 | Variable2 | "Product")*

[4]:
```
# Defining the body w/ annealing
AnnBody = '''
PARAM(N|6);
PARAM(K|4);
AUXDATA(A|"[[0, 1, 1, 1, 0, 0], [1, 0, 1, 1, 1, 0],
[1, 1, 0, 1, 1, 0], [1, 1, 1, 0, 0, 1],
[0, 1, 1, 0, 0, 0], [0, 0, 0, 1, 0, 0]]
");
CLASS(Regions|N|"We have 6 regions");
CLASS(Colors|K|"Available colors");
VARIABLE(x|{Regions,Colors}|"Variables of our problem");
RULE(Rule1|"Each region can only have one color assigned"|"1"|{
        SUMMATORY(from 1 to N iterate i|{
                SQUARED({
                        OFFSET("1")
                        ,SUMMATORY(from 1 to K iterate k|{
                                LINEAR(x[i,k]| "(-1)")
                        })
                })
        })
});
RULE(Rule2|"Two adjacent regions can't have the same color"|"0.5"|{
        SUMMATORY(from 1 to N-1 iterate i|{
                SUMMATORY(from i+1 to N iterate j|{
                        SUMMATORY(from 1 to K iterate k|{
                                QUADRATIC(x[i,k]|x[j,k]|"A[i,j]")
                        })
                })
        })
});
'''
```

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]:
```python
## Circuit creation
assetName = 'QC_qSOA_MapCol'
assetNamespace = 'Manual.Annealing.MapCol'
assetDescription = 'Creating the annealing circuit from qSOA'

assetBody = AnnBody
assetType = 'ANNEAL'
assetLevel = 'IL'

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

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.Annealing.MapCol.QC_qSOA_MapCol')
     # 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.Annealing.MapCol.
      ↪QC_qSOA_MapCol', '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_MapCol'
     assetNamespace = 'Manual.Annealing.MapCol'
     assetDescription = 'Creating the teleportation 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: {'13': 'AMAZON BRAKET Local ExactSolver', '7': 'DWAVE OCEAN Local
Simulator'}
Select a device to run the flow in: 13
```

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 the 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_AnnealingTutorials_12345_Manual_Annealing_MapCol_QC_qSOA_MapCol_1_0':
{'number_of_samples': '16777216', 'number_of_variables': '24', 'sample_energy':
'0.0', 'sample_occurence': '1', 'fullsample': {'x[1,1]': '1', 'x[1,2]': '0',
'x[1,3]': '0', 'x[1,4]': '0', 'x[2,1]': '0', 'x[2,2]': '0', 'x[2,3]': '0',
'x[2,4]': '1', 'x[3,1]': '0', 'x[3,2]': '0', 'x[3,3]': '1', 'x[3,4]': '0',
'x[4,1]': '0', 'x[4,2]': '1', 'x[4,3]': '0', 'x[4,4]': '0', 'x[5,1]': '0',
'x[5,2]': '1', 'x[5,3]': '0', 'x[5,4]': '0', 'x[6,1]': '1', 'x[6,2]': '0',
'x[6,3]': '0', 'x[6,4]': '0'}}}
```

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)
print(gates_representation)
```

```
QS_AnnealingTutorials_12345_Manual_Annealing_MapCol_QC_qSOA_MapCol_1_0
+-------------------+--------------------+---------------+------------------+
| number_of_samples | number_of_variables | sample_energy | sample_occurence |
+-------------------+--------------------+---------------+------------------+
|      16777216     |         24         |      0.0      |        1         |
+-------------------+--------------------+---------------+------------------+
+--------+-------+
|  Name  | Value |
+--------+-------+
| x[1,1] |   1   |
| x[1,2] |   0   |
| x[1,3] |   0   |
| x[1,4] |   0   |
| x[2,1] |   0   |
| x[2,2] |   0   |
| x[2,3] |   0   |
| x[2,4] |   1   |
| x[3,1] |   0   |
| x[3,2] |   0   |
| x[3,3] |   1   |
| x[3,4] |   0   |
| x[4,1] |   0   |
| x[4,2] |   1   |
| x[4,3] |   0   |
| x[4,4] |   0   |
| x[5,1] |   0   |
| x[5,2] |   1   |
| x[5,3] |   0   |
| x[5,4] |   0   |
| x[6,1] |   1   |
| x[6,2] |   0   |
| x[6,3] |   0   |
| x[6,4] |   0   |
+--------+-------+
```

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: {'13': 'AMAZON BRAKET Local ExactSolver', '7': 'DWAVE OCEAN Local
Simulator'}
```

[17]:
```
## Run Quantum Gates Application
exe_ApplicationNames = ['Task_Amazon','Task_DWave']
exe_IdDevices = [13,7]

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)
```

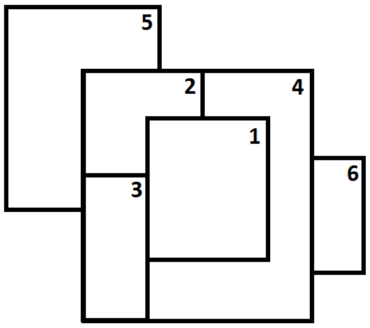
```
[{'QS_AnnealingTutorials_12345_Manual_Annealing_MapCol_QC_qSOA_MapCol_1_0':
{'number_of_samples': '16777216', 'number_of_variables': '24', 'sample_energy':
'0.0', 'sample_occurence': '1', 'fullsample': {'x[1,1]': '1', 'x[1,2]': '0',
'x[1,3]': '0', 'x[1,4]': '0', 'x[2,1]': '0', 'x[2,2]': '0', 'x[2,3]': '0',
'x[2,4]': '1', 'x[3,1]': '0', 'x[3,2]': '0', 'x[3,3]': '1', 'x[3,4]': '0',
'x[4,1]': '0', 'x[4,2]': '1', 'x[4,3]': '0', 'x[4,4]': '0', 'x[5,1]': '0',
'x[5,2]': '1', 'x[5,3]': '0', 'x[5,4]': '0', 'x[6,1]': '1', 'x[6,2]': '0',
'x[6,3]': '0', 'x[6,4]': '0'}}},
{'QS_AnnealingTutorials_12345_Manual_Annealing_MapCol_QC_qSOA_MapCol_1_0':
{'number_of_samples': '16777216', 'number_of_variables': '24', 'sample_energy':
'0.0', 'sample_occurence': '1', 'fullsample': {'x[1,1]': '1', 'x[1,2]': '0',
'x[1,3]': '0', 'x[1,4]': '0', 'x[2,1]': '0', 'x[2,2]': '0', 'x[2,3]': '0',
'x[2,4]': '1', 'x[3,1]': '0', 'x[3,2]': '0', 'x[3,3]': '1', 'x[3,4]': '0',
'x[4,1]': '0', 'x[4,2]': '1', 'x[4,3]': '0', 'x[4,4]': '0', 'x[5,1]': '0',
'x[5,2]': '1', 'x[5,3]': '0', 'x[5,4]': '0', 'x[6,1]': '1', 'x[6,2]': '0',
'x[6,3]': '0', 'x[6,4]': '0'}}}]
```

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

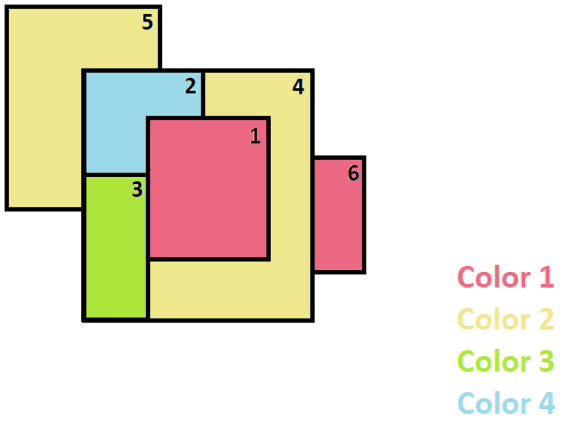
```
QS_AnnealingTutorials_12345_Manual_Annealing_MapCol_QC_qSOA_MapCol_1_0
+-------------------+---------------------+---------------+-------------------+
| number_of_samples | number_of_variables | sample_energy | sample_occurence |
+-------------------+---------------------+---------------+-------------------+
|      16777216     |          24         |      0.0      |         1         |
+-------------------+---------------------+---------------+-------------------+
+--------+-------+
|  Name  | Value |
+--------+-------+
| x[1,1] |   1   |
| x[1,2] |   0   |
| x[1,3] |   0   |
| x[1,4] |   0   |
| x[2,1] |   0   |
| x[2,2] |   0   |
| x[2,3] |   0   |
| x[2,4] |   1   |
| x[3,1] |   0   |
| x[3,2] |   0   |
| x[3,3] |   1   |
| x[3,4] |   0   |
| x[4,1] |   0   |
| x[4,2] |   1   |
| x[4,3] |   0   |
| x[4,4] |   0   |
| x[5,1] |   0   |
| x[5,2] |   1   |
| x[5,3] |   0   |
| x[5,4] |   0   |
| x[6,1] |   1   |
| x[6,2] |   0   |
| x[6,3] |   0   |
| x[6,4] |   0   |
+--------+-------+


QS_AnnealingTutorials_12345_Manual_Annealing_MapCol_QC_qSOA_MapCol_1_0
+-------------------+---------------------+---------------+-------------------+
| number_of_samples | number_of_variables | sample_energy | sample_occurence |
+-------------------+---------------------+---------------+-------------------+
|      16777216     |          24         |      0.0      |         1         |
+-------------------+---------------------+---------------+-------------------+
+--------+-------+
|  Name  | Value |
+--------+-------+
| x[1,1] |   1   |
| x[1,2] |   0   |
| x[1,3] |   0   |
| x[1,4] |   0   |
| x[2,1] |   0   |
| x[2,2] |   0   |
| x[2,3] |   0   |
| x[2,4] |   1   |
| x[3,1] |   0   |
| x[3,2] |   0   |
| x[3,3] |   1   |
| x[3,4] |   0   |
| x[4,1] |   0   |
| x[4,2] |   1   |
| x[4,3] |   0   |
| x[4,4] |   0   |
| x[5,1] |   0   |
| x[5,2] |   1   |
| x[5,3] |   0   |
| x[5,4] |   0   |
| x[6,1] |   1   |
| x[6,2] |   0   |
| x[6,3] |   0   |
| x[6,4] |   0   |
+--------+-------+
```

Given the results, it is easy to see that the solution would be coloring region 1 with color 1, region 2 with color 4, region 3 with color 3, region 4 with color 2, region 5 with color 2, and region 6 with color 1.

That is:



(a) Problem



(b) Solution

References

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