Qiskit Mermin Evaluation

Oct 01, 2020

CONTENTS:

CHAPTER ONE

EVALUATION

`mermin_on_qiskit.evaluation.`**`evaluate_monomial`**`(`*n*, *n_measure*, *circuit*, *a_a_p_coeffs*, *shots*, *is_simulation=True*, *monitor=False*, *local=True*`)`

Draws the circuit if there are some additions and runs it to get the measurement of a monomial

Parameters

- **`n`** (*`int`*) – The number of qubits.
- **`n_measure`** (*`int`*) – The measurement to be performed. Dictates whether a_i or a'_i is used on each wire.
- **`circuit`** (*`QuantumCircuit`*) – The original quantum circuit.
- **`a_a_p_coeffs`** (*`array[float]`*) – The coefficients of the matrices used to calculate Mermin operators.
- **`shots`** (*`int`*) – The number of repetitions of each circuit. Default: 1024.
- **`is_simulation`** (*`boolean`*) – This determines if we are in a case of a local test or a real IBM machine test.
- **`monitor`** (*`boolean`*) – If true a monitor is attached to the job.
- **`local`** (*`boolean`*) – If true, the job run on a local simulator.

Returns float – The result of the measurement probabilities on one monomial.

`mermin_on_qiskit.evaluation.`**`evaluate_polynomial`**`(`*n*, *circuit*, *a_a_p_coeffs*, *shots=1024*, *is_simulation=True*, *monitor=False*, *local=True*`)`

Makes all the implementation and calculation

Caution! [] The IBMQ account must be loaded before the execution of this function if the variable is_simulation is set to False.

Parameters

- **`n`** (*`int`*) – The number of qubits.
- **`circuit`** (*`QuantumCircuit`*) – The original quantum circuit.
- **`a_a_p_coeffs`** (*`list[list[any]]`*) – Lists of lists of elements as described above (packed coefficients).
- **`shots`** (*`int`*) – The number of times that the measurements are made. This is only in case of a local test.

- **is_simulation** (*boolean*) – To specify if the codes are to run locally or on the IBM machine.
- **monitor** (*boolean*) – If true a monitor is attached to the job.
- **local** (*boolean*) – If true, the job run on a local simulator.

Returns float – The result of all the calculations.

mermin_on_qiskit.evaluation.**measures_exploitation**(*measures_dictionary*, *shots*)

Calculates the measurements probabilities

For every possible cases (for example, with n = 2 : 00 01 10 11), the probability to get this combination when measuring is calculated.

In order to obtain this probabilities, we sum the values of the cases where the number of 1 in the measurement is even and when it's then odd.

For example : even_results = values of 00 and 11 measurements odd_results = values of 01 and 10 measurements

Parameters

- **measures_dictionary** (*dict*) – the dictionary containing the measurements and their values.
- **shots** (*int*) – the number of times that the measurements are made. This is only in case of a local test.

Returns float – The total probability of the dictionary measurement.

mermin_on_qiskit.evaluation.**mermin_IBM**(*n*)

Returns the Mermin polynomials under a vector form. This form helps to form the corrects monomials that involves in every mermin evaluation.

Example : In this case, the involving monomials are only the second one, the third one, the fith one and the last one because the others are equal to zero. >>> mermin_IBM(3) [0.0, 0.5, 0.5, 0.0, 0.5, 0.0, 0.0, -0.5]

Parameters **n** (*int*) – The number of qubits.

Returns list(float) – The list of numbers corresponding to the existence and the value each monomial.

CHAPTER

TWO

BASIS CHANGE

`mermin_on_qiskit.basis_change.`**`U3_gates_placement`**(*n*, *n_measure*, *a_a_p_coeffs*, *circuit*)

Places the U3 gates according to the mermin_IBM monomial

Parameters

- **`n`** (*`int`*) – the size of the register to be evaluated
- **`n_measure`** (*`int`*) – The measure to be performed. Dictates whether a_i or a'_i is used on each wire
- **`a_a_p_coeffs`** (*`list[list[real]]`*) – Contains the list of coefficients for a_i and a'_i in the packed shape
- **`circuit`** (*`QuantumCircuit`*) – The circuit on which the measures are appended

Returns None

`mermin_on_qiskit.basis_change.`**`convert_in_binary`**(*number_to_convert*, *number_of_bits=0*)

Converts an int into a string containing its bits

Example :

```
>>> convert_in_binary(5,3)
101
>>> convert_in_binary(5,5)
00101
```

Parameters

- **`number_to_convert`** (*`int`*) – The number that is going to be converted.
- **`number_of_bits`** (*`int`*) – The number of bits required.

Returns str – The converted number.

`mermin_on_qiskit.basis_change.`**`mermin_coeffs_to_U3_coeffs`**(*x*, *y*, *z*)

Generates the coefficients of the U3 gate from mermin coefficients: x*X + y*Y + z*Z = U3(theta, phi, -phi-pi)

Parameters

- **`x`** (*`float`*) – The coefficient alpha for the matrix X.
- **`y`** (*`float`*) – The coefficient beta for the matrix Y.
- **`z`** (*`float`*) – The coefficient gamma for the matrix Z.

Returns (float, float) – The two angles of U3 gate.

CHAPTER

THREE

RUN

`mermin_on_qiskit.run.`**`load_IBMQ_account`**`()`

Loads the IMBQ account. If it fails a first time, the IBMQ token will be prompted and the account loading will be attempted a second time. If it fails a second time. Exits by letting the $Error$ be raised.

Raises:

IBMQAccountCredentialsInvalidFormat: If the default provider stored on disk could not be parsed.

IBMQAccountCredentialsNotFound: If no IBM Quantum Experience credentials can be found.

IBMQAccountMultipleCredentialsFound: If multiple IBM Quantum Experience credentials are found.

IBMQAccountCredentialsInvalidUrl: If invalid IBM Quantum Experience credentials are found.

IBMQProviderError: If the default provider stored on disk could not be found.

`mermin_on_qiskit.run.`**`runCircuit`**`(`*qc*, *simulation=True*, *return_count=True*, *monitor=False*, *local=True*, *shots=1024*`)`

Runs the QuantumCircuit qc in IBM Quantum Experience.

Parameters

- **`qc`** (*QuantumCircuit*) – Quantum circuit to be executed
- **`simulation`** (*bool*) – If $True$, the experience runs on a simulator, which substantially faster than on a quantum processor (due to the demand on those). Otherwise, runs on one of the quantum processors.
- **`return_count`** (*bool*) – If the circuit contains measures, and return_count is set to $True$, then the count of the result will be returned, otherwise, the result will be directly returned.
- **`monitor`** (*bool*) – If $True$, a $job_monitor$ will be displayed after the job is submitted.

TODO : add local and shots docs

Returns dict[str:int] or Result – Depending on return_count, $runCircuit$ either returns the result (of type Result) of the run or the count of this result, which would be the equivalent of calling $result.get_counts()$.

CHAPTER

FOUR

COEFFICIENTS SHAPES

There are three format used for the algorithms:

1. A flat list of coefficients, organized as such: $[x1, y1, z1, x2, y2, ..., xn, yn, zn, x'1, y'1, z'1, x'2, y'2, ..., x'n, y'n, z'n]$. This format is called the *unpacked coefficients* and is used for the QFT optimization. 2. A list of coefficients grouped by families of operators: $[[[x1, y1, z1], [x2, y2, ..., [xn, yn, zn]], [[x'1, y'1, z'1], [x'2, y'2, ..., [x'n, y'n, z'n]]]$ in other words, you have the whole a family and the the whole a' family, and in a family, you have $a1$, $a2$, and so on… Each a is described by it's three coefficients: x, y and z. This format is called *packed coefficients* and is used to easily manipulate coefficients. 3. A list of coefficients grouped by operator: $[[x1, y1, z1], [x'1, y'1, z'1], [x2, y2, z2], [x'2, y'2, z'2], ...]$ in other words, the list is formed as such: $[a1, a'1, a2, a'2, ...]$ This format was previously used for evaluation in Qiskit, allowing for a simpler data flow. But it has the inconvenient of being less true to the maths behind all this so it has been dropped. This format is called *mixed*

With the functions of this module, one may switch between *1.* and *2.* and between *2.* and *3.*, allowing tho switch freely between the three formats.

`mermin_on_qiskit.coefficients_shapes.`**`coefficients_format_mixed_to_packed`**`(`*_a_a_prime_coeffs*`)`

Format the coefficients in the shape previously used for evaluation

Example: >>> coefficients_format_mixed_to_packed([[1, 2, 3], [7, 8, 9], [4, 5, 6], [10, 11, 12]]) ([[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]])

Parameters **`_a_a_prime_coeffs`** (*`list[list[any]]`*) – List of lists of elements as described above (mixed coefficients).

Returns tuple(list[list[any]]) – Tuple of list of list of elements (packed coefficients).

`mermin_on_qiskit.coefficients_shapes.`**`coefficients_format_packed_to_mixed`**`(`*_a_coeffs*, *_a_prime_coeffs*`)`

Format the coefficients in the shape now used for evaluation

Example: >>> coefficients_format_packed_to_mixed([[1,2,3],[4,5,6]], [[7,8,9],[10,11,12]]) [[1, 2, 3], [7, 8, 9], [4, 5, 6], [10, 11, 12]]

Parameters **`_a_coeffs, _a_prime_coeffs`** (*`list[list[any]]`*) – Lists of lists of elements as described above (packed coefficients).

Returns list[any] – List of list of elements (mixed coefficients).

`mermin_on_qiskit.coefficients_shapes.`**`coefficients_format_packed_to_unpacked`**`(`*_a_coeffs*, *_a_prime_coeffs*`)`

Unpacks two lists of lists of three elements to one list of elements

Example: >>> coefficients_format_packed_to_unpacked([[1,2,3],[4,5,6]],[[7,8,9],[10,11,12]]) [1,2,3,4,5,6,7,8,9,10,11,12]

Parameters **`_a_coeffs, _a_prime_coeffs`** (*`list[list[any]]`*) – Lists of lists of elements as described above (packed coefficients).

Returns list[any] – List of elements (unpacked coefficients).

`mermin_on_qiskit.coefficients_shapes.`**`coefficients_format_unpacked_to_packed`**`(`*_a_a_prime_coeffs*`)`

Packs a list of elements in two lists of lists of three elements

Example: >>> coefficients_format_unpacked_to_packed([1,2,3,4,5,6,7,8,9,10,11,12]) ([[1,2,3],[4,5,6]],[[7,8,9],[10,11,12]])

Parameters **`_a_a_prime_coeffs`** (*`list[any]`*) – List of elements (unpacked coefficients).

Returns tuple[list[list[any]]] – Lists of lists of elements as described above (packed coefficients).

CHAPTER

FIVE

QFT MAIN

This module builds the QFT in Qiskit and runs it

mermin_on_qiskit.QFT.**QFT_lenght**(*nWires*)

mermin_on_qiskit.QFT.**all_QFT_circuits**(*nWires*)

mermin_on_qiskit.QFT.**build_QFT_0_to_k**(*nWires*, *k*, *measure=False*)

Builds the QFT on nWires wires up to the k^{th} state generates. Note that the whole QFT can be generated in Qiskit using the QFT method

Parameters

- **nWires** (*int*) – number of wires
- **k** (*int*) – number of gates in the output

Returns QuantumCirctuit – k^{th} first gates of the QFT circuit

mermin_on_qiskit.QFT.**get_coef_from_optimization_file**(*filename*, *iteration*, *evaluation=False*)

The file fed in this function must be a csv file with one of columns being named "iteration", an other one being named "coefficients" and if the $evaluation$ parameter is set to $True$ a column named "intricationValue". The "iteration" column must contain integers, the "coefficients" column must contain tuples of real numbers (the mermin coefficients) and the "intricationValue" column must contain real numbers.

Example:

```
>>> get_coef_from_optimization_file("../Sage/grover_pac/examples/qft_
↪optimization/1-1-4.csv",2)
([(-0.197738971530022, -0.00983193840670331, 0.980205403028067),
    (0.892812904656093, -0.035586934469795, 0.449019695976237),
    (-0.892282320991669, -0.00788653439204609, -0.451408974457756),
    (0.982839628418978, 0.012341254672589, -0.184048793102134)],
  [(0.430894968126063, -0.0211632981654613, 0.902153890006799),
    (-0.984337747324624, 0.0105235779205133, 0.175978559772592),
    (0.984221659519729, -0.0118117927364157, -0.176545196719091),
    (0.883187500168151, 0.0083188846423974, 0.468946303647911)])
```

Parameters

- **filename** (*str*) – Name of the CSV file containing the information about the Mermin coefficients.
- **iteration** (*int*) – Designates the line from which the data must be retrieved.
- **intricationValue** (*bool*) – If $True$, the evaluation will be returned as well as the coefficients.

Returns tuple[list[tuple[real]]], real(optional) – The mermin coefficients previously optimized in packed shape, eventually with the optimum computed with these coefficients.

`mermin_on_qiskit.QFT.`**`periodic_state`**(*l*, *r*, *nWires*)

Returns the periodic state $|\varphi^{l,r} >$ of size 2^{nWires}. We have:

$|\varphi^{l,r} >= \sum_{i=0}^{A-1} |l + ir > /sqrt(A)$ with $A = floor((2^{nWires} - l)/r) + 1$

In this definition, `l` is the shift of the state, and `r` is the period of the state.

Example: Since $|\varphi^{1,5} >= (|1 > +|6 > +|11 >)/sqrt(3) = (|0001 > +|0110 > +|1011 >)/sqrt(3)$,

```
>>> periodic_state(1,5,4)
(0, 1/3*sqrt(3), 0, 0, 0, 0, 1/3*sqrt(3), 0, 0, 0, 0, 1/3*sqrt(3), 0, 0, 0, 0)
```

Parameters

- **`l`** (*`int`*) – The shift of the state.
- **`r`** (*`int`*) – The period of the state.
- **`nWires`** (*`int`*) – The size of the system (number of qubits).

Returns vector – The state defined by `l`, `r` and `nWires` according to the definition given above.

CHAPTER

SIX

HYPERGRAPHSTATES

mermin_on_qiskit.hypergraphstates.**circuit_creation**(*n*, *hyperedges*)

Creates an empty circuit with the number of qubits required.

Parameters

- **n** (*int*) – The number of qubits of which depends the number of wires to create.
- **hyperedges** (*list[list[int]]*) – A list containing the lists of the vertices which are linked by an hyperedge.

Returns QuantumCircuit – A circuit with the required number of quantum wires and classical wires.

mermin_on_qiskit.hypergraphstates.**circuit_initialisation**(*n*, *hyperedges*)

Creates an empty circuit with the number of qubits required and places the circuit in the initial state before adding gates for calculations. Places and Hadamard gate on every main (non additional) qubits wire. This is needed in order to place the qubits in a $|+>$ state which is $(|0>+|1>)/sqrt(2)$.

Parameters

- **n** (*int*) – The number of qubits of which depends the number of wires to create.
- **hyperedges** (*list[list[int]]*) – A list containing the lists of the vertices which are linked by an hyperedge.

Returns QuantumCircuit – The created and initialized circuit.

mermin_on_qiskit.hypergraphstates.**edges_layout**(*n*, *hyperedges*, *circuit*)

Disposes all the gates corresponding to the edges. In fact, for a two-vertices edge, there not much to do, as for a three-vertices edge, too. For more an edge of than three vertices, things are a little different. First, Toffoli gates are used to link qubits two by two. The target qubit here is an auxiliary qubit. Then, the last link is a simple CZ. But all the Toffoli gates that we put create an entanglement which is removed by replacing exactly the same gates again after the CZ.

Parameters

- **n** (*int*) – The number of qubits.
- **hyperedges** (*list[list[int]]*) – The list of the vertices which are linked by an hyperedge.
- **circuit** (*QuantumCircuit*) – The circuit that will be modified.

Returns None

CHAPTER SEVEN

HYPERGRAPHSTATES USED FOR OPTIMIZATION

mermin_on_qiskit.hypergraphstates_optimization.hypergraphstates.**convert_in_binary**(*number_to_c num- ber_of_bits=*

Converts an int into a string containing its bits.

Example:

```
>>> convert_in_binary(5,3)
 101
 and
>>> convert_in_binary(5,3)
00101
```

Parameters

- **number_to_convert** (*int*) – the number that is going to be converted.
- **number_of_bits** (*int*) – the number of bits required.

Returns String number : The converted number.

mermin_on_qiskit.hypergraphstates_optimization.hypergraphstates.**corresponding_state_determi**

Determines the states corresponding to an hyperedge. That is the states where the vertices linked by the hyperedge are equal to 1.

Parameters

- **n** (*int*) – The number of qubits.
- **state** (*list (int)*) – The state of the vector.
- **hyperedges** (*list (list (int))*) – a list containing the lists of the vertices which are linked by an hyperedge.

Param boolean – True if the state is in an hyperedge.

mermin_on_qiskit.hypergraphstates_optimization.hypergraphstates.**hyperedges_computation**(*n, state_ hy- per- edges*

Puts the phases in the right places. Wherever there is an edge or an hyperedge between some vertices, a minus sign is put where those vertices are all in state 1.

Example:

```
>>> hyperedges_computation(2, [0.5, 0.5, 0.5, 0.5], [[0,1]])
[0.5, 0.5, 0.5, -0.5]
```

Parameters

- **n** (*int*) – The number of qubits.
- **state_vector** (*list (int)*) – The initialized state vector.
- **hyperedges** (*list [list [int]*) – a list containing the lists of the vertices which are linked by an hyperedge.

Returns list(int) – The correct state vector.

mermin_on_qiskit.hypergraphstates_optimization.hypergraphstates.**putting_in_list**(*number*)

Puts every figure of the number in a list.

Example:

```
>>> putting_in_list(000)
 [0, 0, 0]
```

Parameters **number** (*int*) – The number to split.

Returns list(int) – The list with every digit of the number in a case.

mermin_on_qiskit.hypergraphstates_optimization.hypergraphstates.**state_vector_initialisatio**

Initializes the state vector; which is to create an array with the size of 2 to the power of n. Every number in the array is equal to 1 over the square root of 2 to the power of n.

Example:

```
>>> state_vector_initialisation(3)
[0.35355339 0.35355339 0.35355339 0.35355339 0.35355339 0.35355339
 0.35355339 0.35355339]
```

Parameters **n** (*int*) – The number of qubits.

Returns list(int) – The initialized state vector.

mermin_on_qiskit.hypergraphstates_optimization.hypergraphstates.**states_formation**(*n*)

Calculates every state for n qubits.

Example:

```
>>> states_formation(3)
 [[0, 0, 0], [0, 0, 1], [0, 1, 0], [0, 1, 1], [1, 0, 0], [1, 0, 1], [1, 1, 0],
↪ [1, 1, 1]]
```

Parameters **n** (*int*) – The number of qubits.

Returns list(list(int)) – A list of all the possible states of the qubits which are also contained in a list.

CHAPTER

EIGHT

MERMIN POLYNOMIALS USED FOR OPTIMIZATION

mermin_on_qiskit.hypergraphstates_optimization.mermin_polynomials.**a_matrix**(*n*, *t*)

Constitutes the matrix a of the Mermin polynomial.

Parameters

- **n** (*int*) – The number of qubits.
- **t** (*np.array(list(float))*) – The table of coefficients.

Returns np.array(complex) – The matrice a of mn.

mermin_on_qiskit.hypergraphstates_optimization.mermin_polynomials.**a_prime_matrix**(*n*, *t*)

Constitutes the matrix a' of the Mermin polynomial.

Parameters

- **n** (*int*) – The number of qubits.
- **t** (*np.array(list(float))*) – The table of coefficients.

Returns np.array(complex) – The matrice a' of mn.

mermin_on_qiskit.hypergraphstates_optimization.mermin_polynomials.**first_coefficients_genera**

Generates the very first coefficients for the calculation of MU.

Example :

```
>>> first_coefficients_generation(2)
[[ 0.24006446 -0.97020025 0.03287126]
 [0.72092088 -0.59054414  0.36267162]
 [-0.76022821 0.64723032 -0.056089]
 [-0.0278048  0.48298397 -0.87518763]]
```

Parameters **n** (*int*) – The number of qubits.

Returns np.array(list(float)) – The table of list of the coefficient taken randomly.

mermin_on_qiskit.hypergraphstates_optimization.mermin_polynomials.**mermin**(*n*, *t*)

Calculates the Mermin polynomial mn.

Parameters

- **n** (*int*) – The number of qubits.
- **t** (*np.array(list(float))*) – The table of coefficients.

Returns np.array(complex) – The Mermin polynomial mn.

`mermin_on_qiskit.hypergraphstates_optimization.mermin_polynomials.`**`mermin_prime`**(*n*, *t*)

Calculates the Mermin polynomial mn'.

Parameters

- **`n`** (*`int`*) – The number of qubits
- **`t`** (*`np.array(list(float))`*) – The table of coefficients

Returns np.array(complex) – The Mermin polynomial mn'

`mermin_on_qiskit.hypergraphstates_optimization.mermin_polynomials.`**`mu_calculation`**(*mn*, *mn_prime*, *vec-tor*, *type_of_mu*)

Calculates MU, the value of the calculation of the vector with the mermin polynomial.

Parameters

- **`mn`** (*`np.array(complex)`*) – The Mermin polynomial mn.
- **`mn_prime`** (*`np.array(complex)`*) – The Mermin polynomial mn'.
- **`vector`** (*`list(int)`*) – The state vector.
- **`bool`** (*`type_of_mu`*) – If False, the classical calculation will be made. If not, another method is used.

Returns float – The value of the calculation.

`mermin_on_qiskit.hypergraphstates_optimization.mermin_polynomials.`**`mu_file_saving`**(*n*, *file_path*, *first_mu*, *first_paramete* *max-i-mal_mu*, *max-imi-sa-tion_paramete* *type_of_mu*)

Creates a file to write the various calculation parameters

Parameters

- **`n`** (*`int`*) – The number of qubits.
- **`file_path`** (*`string`*) – The path where the file is to be saved
- **`first_mu`** (*`float`*) – The value of the calculation of the vector with the mermin polynomial
- **`first_parameters`** (*`np.array(list(float))`*) – The first coefficients of Mu calculation
- **`maximal_mu`** (*`float`*) – The value of the maximal Mu calculated

mermin_on_qiskit.hypergraphstates_optimization.mermin_polynomials.**new_coefficients_generat**

Random generation of new parameters for MU maximization

Parameters

- **n** (*int*) – The number of qubits
- **old_coefficients** (*np.array (list (float))*) – The table of the coefficients that didn't maximize Mu
- **alpha** (*int*) – The value of the descent step (used in the random walk method)

Returns np.array(list(float)) – Table of new coefficients

mermin_on_qiskit.hypergraphstates_optimization.mermin_polynomials.**xbest_calculation**(*n*, *type_of_n* *al-* *pha*, *al-* *pha_mini* *c_maximu* *vec-* *tor*, *file_path*, *sav-* *ing_file=*

Maximizes Mu. The algorithm used here is called the Random walk method. The principle is simple. We randomly generate first parameters which are used to calculate Mu. The first value of Mu is called Mu0. Then, we calculate new parameters based on the previous ones and a variable called the descent step. With the new parameters, we calculate a new value of Mu. If this value is better than the previous one, we keep it and continue the researches until a counter is at its maximum value. The goal here is to take a big circle of research scope and to reduce it (by decreasing the decent step) more and more until the maximum value of Mu is found.

Parameters

- **n** (*int*) – The number of qubits.
- **bool** (*type_of_mu*) – If False is specified, the classical calculation will be made. If not, another method is used.
- **alpha** (*int*) – The value of the descent step.
- **alpha_minimum** (*int*) – The minimum value of the descent step (which is the length of the radius).
- **c_maximum** (*int*) – The maximum value of the counter.
- **vector** (*list (int)*) – The vector for the calculation of Mu.
- **file_path** (*string*) – The path where the file is to be saved.
- **saving_file** (*boolean*) – If set to True, a file will be created / overloaded with the information about the calculation of Mu. If not, only the calculations are made.

Returns np.array(list(float)) – The array that contains the parameters that maximizes Mu.

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX