OptGraphState Tutorials

Version 0.1.0

`https://github.com/seokhyung-lee/OptGraphState`

Seok-Hyung Lee `seokhyung.lee@sydney.edu.au`

April 16, 2023

Contents

OptGraphState is a python package that implements the graph-theoretical strategy to optimize the fusion-based generation of an arbitrary graph state that is proposed in Ref. [1]. The package has the following features:

- Finding a resource-efficient method of generating a given graph state through type-II fusions from multiple three-qubit linear graph states ($\left|G_*^{(3)}\right\rangle$).
- Computing the corresponding resource overhead, which is quantified by the average number of required $\left|G_*^{(3)}\right\rangle$ states.
- Visualizing the original graph (of the graph state you want to generate), unraveled graphs, and fusion networks. An unraveled graph is a simplified graph where the corresponding graph state is equivalent to the desired graph state up to fusions and single-qubit Clifford operations. A fusion network is a graph that instructs the fusions between $\left|G_*^{(3)}\right\rangle$ states required to generate the desired graph state.
- Various predefined sample graphs for input.

This tutorial is written in such a way that it is as understandable as possible without having a background in Ref. [1] except for 7. The API Reference is available in `https://seokhyung-lee.github.io/OptGraphState`.

1 Prerequisites

OptGraphState has the following prerequisites:

- `python == 3.9`
- `numpy == 1.24.2`
- `python-igraph == 0.10.4`
- `networkx == 3.1`
- `parmap == 1.6.0` (Optional, for multiprocessing)
- `matplotlib == 3.7.1` (Optional, for visualization)

The specified versions of these requirements are the ones with which we has tested OptGraphState; it does not mean that OptGraphState does not work with other versions.

OptGraphState heavily relies on the *python-igraph* library, which offers a convenient interface for creating, manipulating, analyzing, and visualizing graphs [2]. The majority of graph-theoretical algorithms used in the strategy are implemented using *python-igraph* (which is faster than *NetworkX* since it is written in C), with the exception of the maximum matching problem, which utilizes the *NetworkX* library. Nevertheless, users are not required to have in-depth knowledge of these libraries to use OptGraphState, though it may be helpful for more thorough investigations.

2 Installation

OptGraphState can be installed with `pip`:

```
$ pip install optgraphstate
```

After starting Python, import OptGraphState with

```
>>> import optgraphstate as ogs
```

3 Defining a `GraphState` object

Most features of OptGraphState are implemented via the `GraphState` class. A `GraphState` object is defined by providing the graph of the graph state that you want to generate. This can be done in the following three ways:

(1) Given by a list of edges. Each edge is given by a 2-tuple of vertex labels in $\{0, 1, 2, \cdots, n_V - 1\}$, where n_V is the number of vertices. For example, to define a `GraphState` object with a four-vertex linear graph:

```
>>> gs = ogs.GraphState(edges=[(0, 1), (1, 2), (2, 3)])
```

Note that the number of vertices is automatically set to be the largest vertex label plus one.

(2) Given directly as an `igraph.Graph` or a `networkx.Graph` object. If a `networkx.Graph` object is given, it is internally converted to an `igraph.Graph` object. For example, to define a `GraphState` object with a four-vertex linear graph:

```
>>> import igraph as ig
>>> g = ig.Graph(edges=[(0, 1), (1, 2), (2, 3)])
>>> gs = ogs.GraphState(graph=g)
```

or

```
>>> import networkx as nx
>>> g = nx.Graph()
>>> g.add_nodes_from([0, 1, 2, 3])
>>> g.add_edges_from([(0, 1), (1, 2), (2, 3)])
>>> gs = ogs.GraphState(graph=g)
```

(3) Chosen among predefined graphs. OptGraphState provides a range of predefined families of graphs. A `GraphState` object can be defined by giving the shape and parameters of the desired graph in the arguments `shape` (as a `str`) and `prms` (as a `tuple` or `int`), respectively. For example, to define a `GraphState` object with a six-vertex cycle graph:

```
>>> gs = ogs.GraphState(shape='cycle', prms=6)
```

To define a `GraphState` object with a 2D lattice graph that has dimensions of $(3, 3)$:

```
>>> gs = ogs.GraphState(shape='lattice', prms=(3, 3))
```

The full list of supported graphs are as follows (see Sec. 1.1, Fig. 2, and Sec. 3.3 of Ref. [1] for their descriptions and visualizations):

- `shape='random'`: Random graph for fixed numbers of vertices and edges, sampled by the Erdős–Rényi model [3].
  - `prms[0]` <`int`>: Number of vertices.
  - `prms[1]` <`int`>: Number of edges.
  - (Optional) `prms[2]` <`None` or `int`>: Random seed. If `None`, the current system time is used as the seed. If not given, the random number generator is not initialized.
- `shape='complete'`, `'star'`, `'linear'`, or `'cycle'`: Complete, star, linear, or cycle graph, respectively.

  - `prms` <`int`>: Number of vertices.

- `shape='lattice'`: Lattice graph.
  - `prms` <`tuple` of `int`>: Numbers of repeated vertices along the `len(prms)` axes.

- `shape='tree'`: Tree graph where all branches in each generation have an equal number of children.
  - `prms[0]` <`int`>: Degree of the root vertex.
  - `prms[i]` (`i` ≥ 1) <`int`>: Number of the children of each `i`th-generation branch.

- `shape='rhg'`: Raussendorf-Harrington-Goyal lattice with primal boundaries only.
  - `prms[0]`, `prms[1]`, `prms[2]` <`int`>: Sizes of the lattice along the three axes in the unit of a cell, respectively.

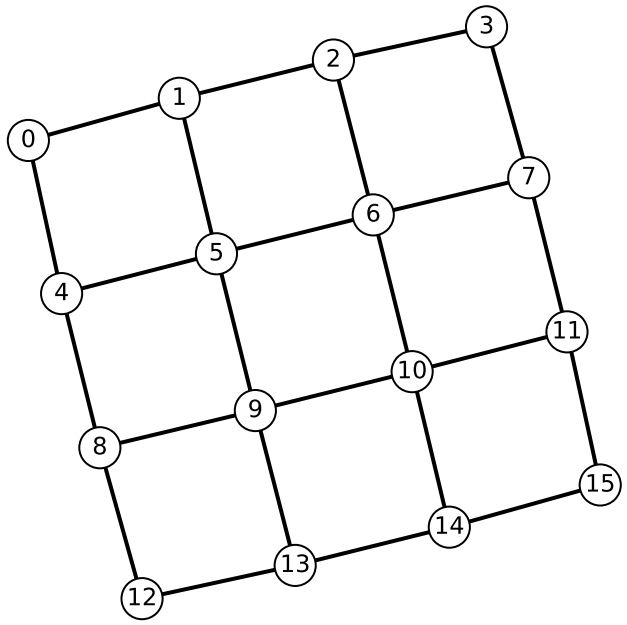
- `shape='repeater'`: Repeater graph with $4m$ vertices.
  - `prms` <`int`>: Parameter m.

- `shape='parity_encoding'`: (n, m) parity-encoded graph [4, 5].
  - `prms[0]` <`igraph.Graph`>: Logical-level graph. It can be generated with the *python-igraph* library directly or from the function `get_graph_from_edges()` or `get_sample_graph()`.
  - `prms[1]`, `prms[2]` <`int`>: Parameters n and m of the parity encoding.

- `shape='ptqc'`: Microcluster for parity-encoding-based topological quantum computing protocol [5].
  - `prms[0]`, `prms[1]` <`int`>: Parameter n and m of the parity encoding.
  - `prms[2]` <`bool`>: Whether the H-configuration is HIC (`True`) or HIS (`False`).
  - `prms[3]` <`bool`>: Whether the microcluster is a central (`True`) or side (`False`) one.

The graph of a `GraphState` object can be visualized by using `GraphState.plot_graph()`:

```
>>> gs = ogs.GraphState(shape='lattice', prms=(4, 4))
>>> gs.plot_graph()
(<Figure size 500x500 with 1 Axes>, <Axes: >)
```



4 Executing the strategy

The strategy is performed by iterating the three steps: unraveling the original graph, building a fusion network, and computing the resource overhead by determining the fusion order. To briefly explain these steps:

1. **Unraveling the graph**: Find an *unraveled graph* where the corresponding graph state is equivalent to the original graph state up to several fusions and single-qubit Clifford gates and is easier to generate than the original graph state. It is done by *unraveling* specific subgraph structures of the original graph, such as *bipartitely-complete graphs* and *cliques*.

2. **Building a fusion network**: A *fusion network* is graph where the nodes indicate individual three-qubit linear graph states ($\left|G_*^{(3)}\right\rangle$) and

the links indicate fusions between them that are required to generate the desired graph state. We use the term "node" and "link" instead of "vertex" and "edge" to distinguish them from the vertices and edges in the original and unraveled graphs.

3. **Determining the fusion order and computing the resource overhead**: Based on the built fusion network, the time order of fusions is determined using the *min-weight-maximum-matching-first* strategy. The resource overhead (namely, the average number of three-qubit linear graph states required to generate the desired graph state) is computed during this process.

See Ref. [1] for more details on the strategy. Note that these three steps are implemented in `GraphState.unravel_graph()`, `GraphState.build_fusion_network()`, and `GraphState.calculate_overhead()`, respectively.

Since the above process contains randomness, a different value is obtained each time the resource overhead is calculated. Therefore, we take an approach to find the optimal one by iterating the entire steps sufficiently.

You have two choices for the iteration: (i) Iterate for a fixed number of times or (ii) use the *adaptive iteration method*. Here, the adaptive iteration method with the given initial iteration number n_{init} and the multiplicative factor M means the following process: Denoting the process to iterate the three steps n times as $R(n)$, first perform $R(n_{\text{init}})$ and obtain $Q_{\text{opt}}^{(1)}$, which is the minimal resource overhead obtained so far. Then, perform $R(Mn_{\text{init}})$ and obtain $Q_{\text{opt}}^{(2)}$ similarly. If $Q_{\text{opt}}^{(1)} \leq Q_{\text{opt}}^{(2)}$, stop the iteration and return $Q_{\text{opt}}^{(1)}$. If otherwise, perform $R\big(M^2 n_{\text{init}}\big)$, obtain $Q_{\text{opt}}^{(3)}$, and stop the iteration if $Q_{\text{opt}}^{(2)} \leq Q_{\text{opt}}^{(3)}$, which returns $Q_{\text{opt}}^{(2)}$. If $Q_{\text{opt}}^{(2)} > Q_{\text{opt}}^{(3)}$, perform $R\big(M^3 n_{\text{init}}\big)$, obtain $Q_{\text{opt}}^{(4)}$, and so on.

(i) With a fixed iteration number: Use `GraphState.simulate()`. For example, to iterate 1000 times with the fusion success rate of 0.5 for a $(5, 5)$-lattice graph through multiprocessing:

```
>>> gs = ogs.GraphState(shape='lattice', prms=(5, 5))
>>> res = gs.simulate(
...     1000,         # iteration number
...     p_succ=0.5,   # fusion success rate
...     mp=True,      # whether to use multiprocessing or not
...     n_procs=None  # number of simultaneous processes
...                   # (number of CPU cores if None)
... )
Multiprocessing ON: n_procs = 8
Calculating for n_iter = 1000... Done. Best: 100352.00 (2.55 s)
>>> for k, v in res.items():
...     print(k, ':', v)
best_overhead : 112640.0
best_step : 12
best_seed : 2049002238
n_iter : 1000
unravel_bcs_first : False
```

(ii) With the adaptive iteration method: Use `GraphState.simulate_adaptive()`. For example, if the initial iteration number is 200 and the multiplicative factor is 2:

```
>>> gs = ogs.GraphState(shape='lattice', prms=(5, 5))
>>> res = gs.simulate_adaptive(
...     200,          # initial iteration number
...     mul=2,        # multiplicative factor
...     p_succ=0.5,   # fusion success rate
...     mp=True,      # whether to use multiprocessing or not
...     n_procs=None  # number of simultaneous processes
...                   # (number of CPU cores if None)
... )
Multiprocessing (n_procs = 8)
Calculating for n_iter = 200... Done. Best: 167936.00 (1.55 s)
Calculating for n_iter = 400... Done. Best: 100352.00 (1.74 s)
Calculating for n_iter = 800... Done. Best: 100352.00 (2.27 s)
>>> for k, v in res.items():
...     print(k, ':', v)
best_overhead : 100352.0
best_step : 12
best_seed : 1823346480
n_iter : 1400
unravel_bcs_first : False
```

For both methods, the outcome of a simulation is returned as a dictionary with the following keys and values:

- `'best_overhead'`: Smallest overhead value among the samples.
- `'best_step'`: Minimal number of steps (i.e., groups of fusions that can be done in parallel) for the best sample (that gives the smallest overhead).
- `'best_seed'`: Random seed of the best sample, which can be used to reproduce the result; e.g.,

```
>>> gs.simulate(1, p_succ=0.5, seed=2049002238).
```

- `'n_iter'`: Total number of iterations.
- `'unravel_bcs_first'`: Whether bipartitely-complete subgraphs (BCSs) are unraveled before cliques are unraveled, for the best sample.

Only the data of the best sample is obtained by default. If you want to get the data of all samples, set the keyword argument `get_all_data=True`. Similarly, set `get_all_graphs=True` to get all the unraveled graphs, and set `get_all_fusion_networks=True` to get all the fusion networks.

5 Obtaining instructions for fusion-based generation

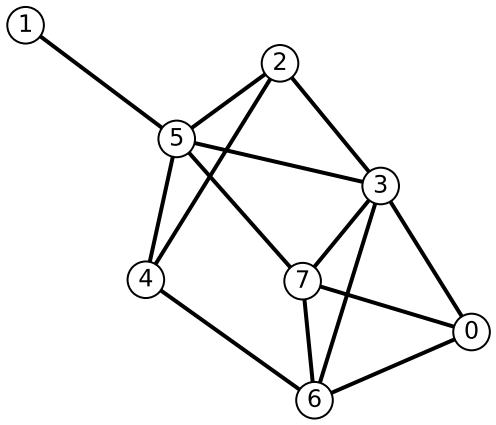
The returned outcome of `GraphState.simulate()` or `GraphState.simulate_adaptive()` does not provide detailed instructions on how to generate the desired graph state. However, after running the simulation, all information of the best sample is stored in the `GraphState` object: the original graph in `GraphState.graph`, the unraveled graph in `GraphState.unraveled_graph`, and the fusion network in `GraphState.fusion_network`. The easiest and most intuitive way to learn how to generate the desired graph state from initial $\left|G_*^{(3)}\right\rangle$ states is to visualize the obtained unraveled graph and fusion network.

Let us consider the following example:

```
>>> gs = ogs.GraphState(shape='random', prm=(8, 14, 4))
>>> gs.simulate(1000)
Multiprocessing OFF.
Calculating for n_iter = 1000...
Done. Best: 384.00 (1.45 s)
```

which has the following graph

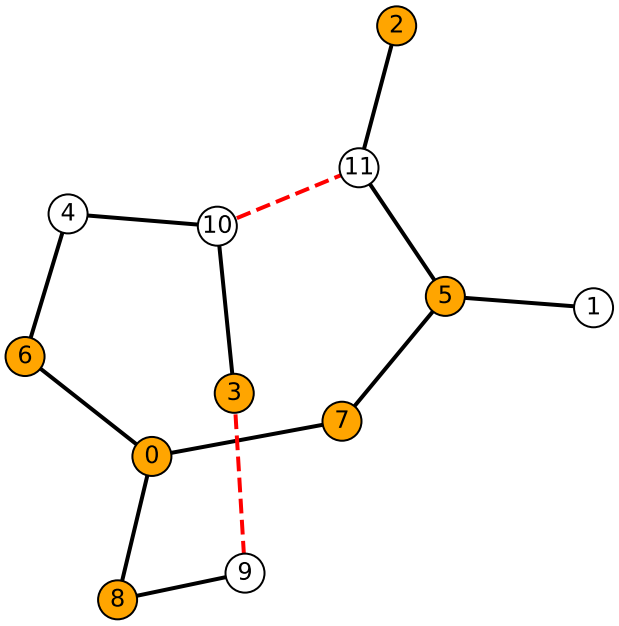
```
>>> gs.plot_graph(figsize=(4, 4))
(<Figure size 400x400 with 1 Axes>, <Axes: >)
```



The number inside each vertex is referred to as its *name*, which is unique.

First, use `GraphState.plot_graph()` to visualize the unraveled graph:

```
>>> gs.plot_graph(unraveled=True, figsize=(5, 5))
(<Figure size 400x400 with 1 Axes>, <Axes: >)
```



It can be interpreted as follows:

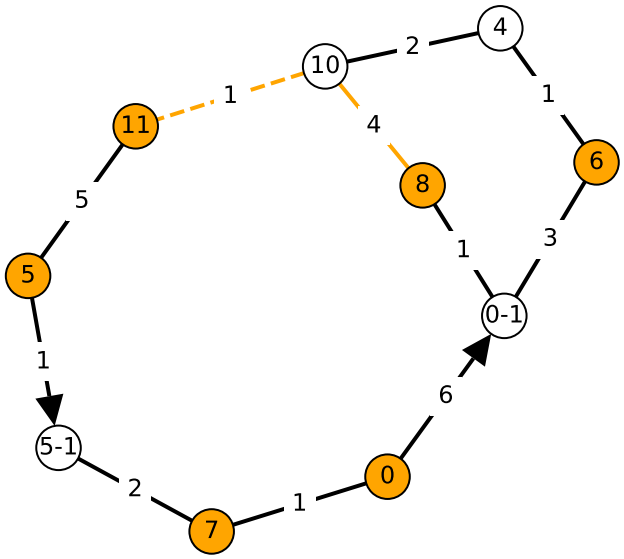
- Each vertex is a qubit, each black solid line is an edge of the unraveled graph, and each red dashed line indicates a fusion that should be done to recover the original graph state.
- The numbers inside the vertices are their unique names. The vertex names in the original graph are here as well, which mean that they are the same qubit. Other vertices in the unraveled graph (`'7'`–`'11'`) are for new qubits.
- The orange vertices mean that non-trivial Clifford gates should be applied to the qubits to recover the original graph state. Such gates can be identified by using `GraphState.get_clifford_in_graph()`:

```
>>> gs.get_clifford_in_graph('3')
'RX'
>>> gs.get_clifford_in_graph('7')
'RZ'
```

Namely, the qubits `'3'` and `'7'` should be subjected to $R_X(\pi/2) := \exp\left[i(\pi/4)\hat{X}\right]$ and $R_Z(\pi/2) := \exp\left[i(\pi/4)\hat{X}\right]$, respectively. Its returned value also can be something like `'RZ-RX'`, which means that the qubit should be subjected to $R_X(\pi/2)R_Z(\pi/2)$.

Next, use `GraphState.plot_fusion_network()` to visualize the fusion network:

```
>>> gs.plot_fusion_network(figsize=(5, 5))
(<Figure size 500x500 with 1 Axes>, <Axes: >)
```



It can be interpreted as follows:

- Each node is a $\left|G_*^{(3)}\right\rangle$ state and each link indicates a fusion required

between the $\left|G_*^{(3)}\right\rangle$ states. We refer to the central qubit of a $\left|G_*^{(3)}\right\rangle$ state as its *root* qubit and the other qubits as its *leaf* qubits. The numbers inside the nodes are their unique names.

- A number placed on each link means the *step index* of the fusion. The fusions should be performed in the order in which these numbers increase (from 1 to 6) and those with the same number can be done simultaneously.

- The line style (solid, dashed, or arrow) of a link indicates which qubits in the $\left|G_*^{(3)}\right\rangle$ states are involved in the fusion:

  - Solid line: Two leaf qubits are fused.
  - Dashed line: Two root qubits are fused.
  - Arrow: One leaf qubit and one root qubit are fused. The arrow points in the direction from the leaf qubit to the root qubit. (For example, the leaf qubit of the node `'5'` and the root qubit of the node `'5-1'` are fused.)

- An orange line indicates that non-trivial Clifford gates should be applied to one or both of the qubits involved in the fusion before the fusion is performed. To identify such Clifford gates, use `GraphState.get_clifford_in_fusion_network()`:

```
>>> gs.get_clifford_in_fusion_network(8, 10)
{'10': 'RX', '8': None}
```

  Namely, for the link connecting the nodes `'8'` and `'10'`, the (leaf) qubit of the node `'10'` involved in the fusion should be subjected to $R_X(\pi/2)$ beforehand.

- An orange node indicates that one or more qubits in the node should be subjected to non-trivial Clifford gates after all the fusions are performed. To identify such Clifford gates, use `GraphState.get_clifford_in_fusion_network()`:

```
>>> gs.get_clifford_in_fusion_network(0)
{'root': 'RX', 'leaves': None}
>>> gs.get_clifford_in_fusion_network(11)
{'root': None, 'leaves': ['RZ']}
```

Namely, the root qubit of the node `'0'` (which is not involved in any fusions) should be subjected to $R_X(\pi/2)$ and the final remaining leaf qubit of the node `'11'` should be subjected to $R_Z(\pi/2)$ Note that, if two leaf qubits remain for a node, the value for the key `'leaves'` may contain two elements.

- We need to know how the final remaining qubits after performing all the fusions correspond to the vertices in the original graph. For a vertex in the original graph, it is as follows:
  - If the vertex with the same name in the *unraveled* graph is connected to two or more vertices (with solid lines), it corresponds to the root qubit of the node with the same name. (For example, the vertex `'0'` corresponds to the root qubit of the node `'0'`.)
  - If the vertex with the same name in the *unraveled* graph is connected to only one vertex u, it corresponds to a final remaining leaf qubit of a node that has a name starting with the name of u. (For example, the vertices `'1'` and `'2'` correspond to the final remaining leaf qubits of the nodes `'5-1'` and `'11'`, respectively.)

6 List of features

6.1 Class variables of `GraphState`

- `GraphState.data` <`dict`>: Any data obtained during unraveling the graph, building the fusion network, and calculating the resource overhead.
- `GraphState.graph` <`igraph.Graph`>: Graph of the given desired graph state.
- `GraphState.unraveled_graph` <`None` or `igraph.Graph`>: Graph of the generated unraveled graph state.
- `GraphState.fusion_network` <`None` or `igraph.Graph`>: Generated fusion network.

See Sec. 7 for the list of vertex and edge attributes of these `igraph.Graph` class variables.

6.2 Execution of the strategy

- `GraphState.unravel_graph()`: Unravel the original graph.
- `GraphState.build_fusion_network()`: Build a fusion network from the original or unraveled graph.
- `GraphState.calculate_overhead()`: Determine the fusion order and calculate the resource overhead.
- `GraphState.initialize()`: Initialize the generated unraveled graph, fusion network, and calculated data.
- `GraphState.simulate()`: Iterate the strategy for a fixed number of times.
- `GraphState.simulate_adaptive()`: Iterate the strategy with the adaptive iteration method.

6.3 Extract information from graphs

- `GraphState.get_clifford_in_graph()`: Get a Clifford gate applied to a given vertex in the unraveled or original graph.
- `GraphState.get_clifford_in_fusion_network()`: Get the information about Clifford gates for a given node or link in the fusion network.

6.4 Visualization

- `GraphState.plot_graph()`: Plot the original or unraveled graph.
- `GraphState.plot_fusion_network()`: Plot the fusion network.
- `plot_graph()`: Plot a given graph. `GraphState.plot_graph()` is equal to `plot_graph(GraphState.graph)` and `GraphState.plot_graph(unraveled=True)` is equal to `plot_graph(GraphState.unraveled_graph)`.
- `plot_fusion_network()`: Plot a given fusion network. `GraphState.plot_fusion_network()` is equal to `plot_fusion_network(GraphState.fusion_network)`.

6.5 Graph tools

- `get_graph_from_edges()`: Generate a graph (`igraph.Graph`) from a given list of edges.
- `get_sample_graph()`: Generate a predefined graph (`igraph.Graph`) from a given shape and parameters.
- `find_nonoverlapping_bcss()`: Find a maximum set of bipartitely-complete subgraphs that do not share any vertices.
- `find_nonoverlapping_cliques()`: Find a maximum set of cliques that do not share any vertices.

6.6 Others

- `GraphState.copy()`: Return a shallow copy of the `GraphState` object.

7 More on the graphs and fusion network

For more advanced applications, you may need to directly deal with the `igraph.Graph` class variables in a `GraphState` object: `GraphState.graph`, `GraphState.unraveled_graph`, and `GraphState.fusion_network`. Here, we present the details on the vertex and edge attributes of these class variables. We assume that the reader is familiar with our strategy in Ref. [1] and the *python-igraph* library.

7.1 Original graph `GraphState.graph`

Vertex attributes

- `'name'` <`str`>: Unique label of the vertex.
- (Optional) `'clifford'` <`str`>: Clifford gates applied to the vertex. `'RX'` and `'RZ'` indicates the operators $R_X := \exp\left[i(\pi/4)\hat{X}\right]$ and $R_Z := \exp\left[i(\pi/4)\hat{Z}\right]$, respectively. If it is `'RX-RZ'`, it means that R_X is applied and then R_Z is applied.

7.2 Unraveled graph `GraphState.unraveled_graph`

Vertex attributes

- `'name'` <`str`>: Unique label of the vertex. If the vertex comes from the original graph (namely, if it is not a new vertex created while unraveling), it has the same label as the corresponding vertex in the original graph.
- `'ext_fusion'` <`str`>: Name of the other vertex that undergoes an external fusion with the vertex.
- `'clifford'` <`str`>: Clifford gates applied to the vertex.

7.3 Fusion network `GraphState.fusion_network`

Vertex (node) attributes

- `'name'` <`str`>: Unique label of the node. For a vertex `'i'` in the fusion network that is connected with two or more edges, its corresponding node group contains the nodes with the names `'i'`, `'i-1'`, `'i-2'`, and so on, where the node `'i'` is the seed node of the node group.
- `'seed'` <`bool`>: Whether the node is a seed node or not.
- `'clifford_root'` <`None` or `str`>: Clifford gates applied to the final remaining root qubit of the node. It is `None` if the root qubit undergoes a fusion or it is not subjected to any non-trivial Clifford gates.
- `'clifford_leaves'` <`None` or `list` of `str`>: Clifford gates applied to one or two final remaining leaf qubits of the node. It is `None` if there are no remaining leaf qubits subjected to non-trivial Clifford gates.

Edge (link) attributes

- `'name'` <`str`>: Unique label of the link.
- `'kind'` <`str`>: Type of the link. It is one of `'RR'`, `'LL'`, and `'RL'`, which respectively mean root-to-root, leaf-to-leaf, and root-to-leaf.
- `'root_node'` <`str`>: Name of the node (connected with the link) whose root qubit is involved in the fusion of the link, if the link has the type of root-to-leaf. It is `None` if the link has another type.
- `'cliffords'` <`dict`>: Clifford gates applied to the qubits involved in the fusion before the fusion is performed. Its keys are the names of the two nodes connected by the link and its values are the Clifford gates applied to the respective qubits.

- `'step'` <int>: Step index of the fusion that starts from 1. The fusions are performed in the order in which their step indices increase and those with the same step index can be done simultaneously. This attributes is not created before the fusion order is determined by `GraphState.calculate_overhead()`.

References

[1] S.-H. Lee and H. Jeong, "Graph-theoretical optimization of fusion-based graph state generation," [To be uploaded] (2023).

[2] G. Csardi and T. Nepusz, "The igraph software package for complex network research," InterJournal **Complex Systems**, 1695 (2006).

[3] P. Erdős and A. Rényi, "On random graphs I," Publicationes mathematicae **6**, 290–297 (1959).

[4] T. C. Ralph, A. J. F. Hayes, and A. Gilchrist, "Loss-tolerant optical qubits," Phys. Rev. Lett. **95**, 100501 (2005).

[5] S.-H. Lee, S. Omkar, Y. S. Teo, and H. Jeong, "Parity-encoding-based quantum computing with bayesian error tracking," arXiv:2207.06805 [quant-ph] (2022), 10.48550/arXiv.2207.06805.