QuNeX : Quantum Network eXplorer

1 Introduction

`qunex` is a Python package developed for research in quantum networks. The package provides modular tools for constructing network topologies, identifying multiple edge disjoint paths, simulating quantum operations such as entanglement swapping and purification, and computing viable regions for a given quantum task. The package is intended for theoretical and numerical studies of large-scale quantum networks. Our implementation builds on the `NetworkX` Python package for graph construction and manipulation [1].

The structure of the package is as follows :

```
qunex/
|-- pyproject.toml
|-- README.md
|-- LICENSE
|-- qunex/
    |-- __init__.py
    |
    |-- network/
    | |-- __init__.py
    | |-- topology.py
    | | # Classes:
    | | # - RegularTopology
    | | # - RandomTopology
    | | # - BarabasiAlbertTopology
    | |
    | |-- assign_edge_parameters.py
    | # Classes:
    | # - EdgeParameters
    |
    |-- paths/
    | |-- find_paths.py
    | | # Functions:
    | | # - edge_disjoint_paths
    | |
    | |-- path_parameters.py
    | # Classes:
    | # - FindPathParameters
    |
    |-- network_operations/
    | |-- path_swapping.py
    | | # Functions:
    | | # - swapped_concurrence_werner
```

```
    | | # - swapped_fidelity_werner
    | | # - swapped_probability
    | |
    | |-- multipath_purification.py
    | # Functions:
    | # - multipath_purification_deutsch
    |-- entanglement_measure\
    | |-- compute_concurrence.py
    | # Functions:
    | # - concurrence
    |-- quantum_operations/
    | |-- entanglement_swapping.py
    | | # Functions:
    | | # - swapping
    | |-- noise_models.py
    | | # Functions:
    | | # - apply_kraus
    | | # - depolarizing_channel
    | | # - bit_flip_channel
    | | # - phase_flip_channel
    | | # - amplitude_damping_channel
    | | # - phase_damping_channel
    | | # - two_qubit_depolarizing_channel
    | | # - memory_decoherence
    | | # - measurement_error
    |-- viability/
        |-- viable_region.py
            # Classes:
            # - FindViableRegion
                # Functions:
                # - viable_radius
                # - viable_node_fraction
```

2 Different modules of the package

2.1 Network Module

This module offers creation of five network topologies (Random network, Barabasi-Albert network, Square lattice, Triangular lattice and Hexagonal lattice) and allows assignment of edge-parameters for each edge of the networks.

2.1.1 `network.topology`

This module contains three classes: `RegularTopology`, `RandomTopology`, and `BarabasiAlbertTopology`.

The `RegularTopology` class takes two inputs, `lattice_type` and `length`, and generates a two-dimensional regular lattice network of the specified type and size. Square, triangular, and hexagonal lattice topologies can be created by setting `lattice_type` to `"square"`, `"triangular"`, and `"hexagonal"`, respectively. For a given value of `length` L, the network consists of L rows and L columns of lattice cells (squares, triangles, or hexagons), depending on the chosen lattice type.

The `RandomTopology` class takes `num_nodes` and `num_edges` as inputs and generates a random graph with the specified number of nodes and edges.

The `BarabasiAlbertTopology` class takes `num_nodes` and `attached_edges` as inputs and constructs a scale-free network using the Barabási–Albert preferential attachment model, where each newly added node is connected to `attached_edges` existing nodes during the network construction process.

```
from qunex.network.topology import RegularTopology
from qunex.network.topology import BarabasiAlbertTopology
import networkx as nx

square_topo = RegularTopology(lattice_type="square", length=10)
G = square_topo.create()
nx.draw(G)

barabasi_topo = BarabasiAlbertTopology(num_nodes=100, attached_edges=5)
G = barabasi_topo.create()
nx.draw(G)
```

2.1.2 `network.assign_edge_parameters`

This module provides a convenient interface for assigning edge-parameters to the edges of a network. Edge parameters are sampled independently from user-specified probability distributions and attached to each edge as attributes.

The module accepts a dictionary whose keys correspond to edge-parameter names and whose values are callable objects (e.g., functions or lambda expressions) that return random samples from the desired distributions. Each edge in the network is assigned a value for every specified parameter.

```
from qunex.network.assign_edge_parameters import EdgeParameters
import numpy as np

edge_params = EdgeParameters({
    "concurrence": lambda: np.random.uniform(0.6, 1.0),
    "probability": lambda: np.random.uniform(0.2, 1.0)
})

G = edge_params.assign(G)
```

2.2 Paths Module

This module provides tools for finding multiple edge disjoint paths between any two nodes in the network and computing the path-parameters of any specified path via swapping at the intermediate nodes.

2.2.1 `paths.find_paths`

This module provides the function `edge_disjoint_paths`, which computes a specified number of edge-disjoint paths between two given nodes in a network. The function takes four inputs: `graph` (the network), `source`, `target`, and `num_paths`, where `num_paths` specifies the number of edge-disjoint paths to be returned.

If num_paths is set to "all", the function returns all available edge-disjoint paths between the source and target nodes.

```
from qunex.paths.find_paths import edge_disjoint_paths

paths = edge_disjoint_paths(G, source=0, target=10, num_paths=5)
```

2.2.2 paths.path_parameters

This module provides the FindPathParameters class, which is used to compute path-parameters from edge-parameters along a given path in a network. The class takes the graph (the network) and a specific path as inputs.

The method edge_parameters_of_path extracts and returns the values of a specified edge parameter along the given path. It takes parameter_name as input, which denotes the edge parameter to be considered.

The method compute evaluates the corresponding path parameter by successively applying a user-defined swapping rule at the intermediate nodes along the path. It takes parameter_name and parameter_swapping_rule as inputs, where parameter_swapping_rule specifies the function that governs how the chosen parameter is updated under swapping.

The swapping rule swapped_concurrence_werner, used in the example below, is implemented in the subsequent module and provides the concurrence update rule for entanglement swapping of Werner states.

```
import qunex as qx
from qunex.paths.find_paths import FindPathParameters

finder = FindPathParameters(G, path)
path_concurrence = finder.compute(
    parameter_name="concurrence",
    parameter_swapping_rule=qx.swapped_concurrence_werner
)
```

2.3 Network Operations Module

This module provides tools for performing entanglement swapping and multipath entanglement purification (MPEP) in quantum networks. It allows the computation of path-parameters after entanglement swapping and includes multipath purification protocol by Deutsch using alternate edge-disjoint paths.

2.3.1 network_operations.path_swapping

This module includes three functions: swapped_concurrence_werner, swapped_fidelity_werner, and swapped_probability. The first two functions compute the output concurrence and fidelity, respectively, after entanglement swapping, given a list of input concurrences or fidelities corresponding to Werner states. The function swapped_probability computes the output probability after swapping, which is given by the product of the input probabilities.

```
import qunex as qx

final_concurrence = qx.swapped_concurrence_werner(input_concurrences_list
    )
```

2.3.2 network_operations.multipath_purification

This module implements multipath entanglement purification (MPEP) [2] between two nodes using the function `multipath_purification_deutsch`. In the general case, the function takes five inputs: `graph` (the network), `mad_paths` (the list of available edge-disjoint paths), `parameter_name`, `swap_rule`, and `purification_rule`.

When `parameter_name` is set to `"concurrence"`, `"fidelity"`, or `"probability"`, the corresponding `swap_rule` and `purification_rule` are provided by default and should not be supplied explicitly. The parameter names `"concurrence"` and `"fidelity"` are used under the assumption that the entangled states distributed along the network edges are Werner states. For other types of entangled states, users should supply alternative parameter names and the appropriate swapping and purification rules.

```
import qunex as qx

mpep_concurrence = qx.multipath_purification_deutsch(
    G,
    mad_paths=paths,
    parameter_name="concurrence")
```

2.4 Entanglement Measure Module

This module provides the tools for quantifying the amount of entanglement in a bipartite quantum state.

2.4.1 entanglement_measure.compute_concurrence

This module includes the function `concurrence` that takes a bipartite density matrix as input and return the concurrence of that state.

```
import qunex as qx

state = [[0.19047705, 0.1758254, 0.17646639 ,0.00056875],
        [0.1758254, 0.37176256, 0.28325486, 0.01057298],
        [0.17646639, 0.28325486, 0.39528931, 0.0176381 ],
        [0.00056875, 0.01057298, 0.0176381, 0.04247107]]
conc = qx.concurrence(state)
```

2.5 Quantum Operations Module

This module provides the tool for performing entanglement swapping of two quantum states. It also includes tools for modeling noise and decoherence in quantum states using standard quantum noise channels. The noise models are implemented using the Kraus operator formalism and act directly on density matrices, returning the corresponding noisy quantum states.

2.5.1 quantum_operations.entanglement_swapping

This module includes the function `swapping` which takes two density matrices as inputs and returns the average (over the four Bell measurements) output concurrence after swapping the two density matrices.

```
import qunex as qx
```

```
state1 = [[0.19047705, 0.1758254, 0.17646639 ,0.00056875],
         [0.1758254, 0.37176256, 0.28325486, 0.01057298],
         [0.17646639, 0.28325486, 0.39528931, 0.0176381 ],
         [0.00056875, 0.01057298, 0.0176381, 0.04247107]]
state2 = [[0.34653073, 0., 0.04723906, 0.30549839],
         [0., 0.16163419, 0.00960421, 0.01667312],
         [0.04723906, 0.00960421, 0.01697134, 0.03058887],
         [0.30549839, 0.01667312, 0.03058887, 0.47486374]]
output_avg_conc = qx.swapping(state1, state2)
```

2.5.2 quantum_operations.noise_models

This module includes the following functions:

- `apply_kraus`: A utility function that applies a quantum channel specified by a list of Kraus operators to a given density matrix and returns the resulting state.
- `depolarizing_channel`: Implements the single-qubit depolarizing channel, where the input state undergoes a random Pauli error with a specified probability.
- `bit_flip_channel`: Models bit-flip noise by applying Pauli-X errors to the input state with a given probability.
- `phase_flip_channel`: Models phase-flip noise by applying Pauli-Z errors to the input state with a given probability.
- `amplitude_damping_channel`: Models energy relaxation processes through amplitude damping, capturing irreversible decay effects in single-qubit systems.
- `phase_damping_channel`: Models pure dephasing processes by suppressing off-diagonal elements of the density matrix without changing population terms.
- `two_qubit_depolarizing_channel`: Implements a two-qubit depolarizing channel, modeling correlated noise acting jointly on bipartite quantum states, such as those arising from imperfect entangling operations.
- `memory_decoherence`: Models decoherence due to finite quantum memory lifetimes using an effective phase damping process parametrized by the storage time and coherence time.
- `measurement_error`: Implements a classical measurement error model, returning an incorrect measurement outcome with a specified probability.

All quantum noise channels take a density matrix as input and return the corresponding noisy density matrix, ensuring compatibility with subsequent quantum operations and entanglement measures.

```
import qunex as qx
import numpy as np

state = np.array([[0.5, 0.5],
                  [0.5, 0.5]])

noisy_state = qx.depolarizing_channel(state, p=0.05)
```

The noise models provided in this module can be incorporated into simulations of entanglement distribution, swapping, and purification to study the impact of noise on large-scale quantum network performance.

2.6 Viability Module

This module provides tools for characterizing the viability of quantum tasks by computing the viable radius and the average fraction of viable nodes in a network with respect to specified task.

2.6.1 `viability.viable_region`

This module includes the `FindViableRegion` class, which is used to identify regions of a network that satisfy the requirements of a given quantum network task. The class takes the following inputs: `graph`, `parameter_name`, `parameter_swapping_rule` (not required when `parameter_name` is `"concurrence"`, `"fidelity"`, or `"probability"`, in which case the appropriate rule is applied by default), `parameter_threshold` (the minimum threshold required for the specified quantum task), `min_req_prob` (the minimum required success probability for task viability), and `num_samples` (the number of source nodes sampled for statistical averaging).

The radius of the viable region is computed using the method `viable_radius`, while the average fraction of viable nodes around a source node is evaluated using the method `viable_node_fraction`. Please refer to [3] for the detailed understanding of Viability of quantum tasks in a quantum network.

```
from qunex.viability.viable_region import FindViableRegion

region = FindViableRegion(
    G,
    parameter_name="concurrence",
    parameter_threshold=0.8,
    min_req_prob=0.5,
    num_samples=100,
)

radius = region.viable_radius()
node_frac = region.viable_node_fraction()
```

3 Typical Workflow

A typical simulation workflow using the `qunex` package proceeds through the following steps:

1. **Construct a network topology:** First, a network topology is generated using one of the available topology modules. Depending on the physical scenario of interest, the user may choose a regular lattice, a random graph, or a scale-free network. This step defines the underlying structure of the quantum network.

2. **Assign physical parameters to network edges:** Physical parameters associated with entangled links—such as concurrence, fidelity, or link success probability—are assigned to the edges of the network. These parameters are typically sampled from user-defined probability distributions to model realistic imperfections in quantum channels.

3. **Identify paths between network nodes:** Edge-disjoint network paths between pairs of nodes are identified. These paths are used for entanglement distribution via swapping, and purification operations.

4. **Perform entanglement swapping and purification:** Using the available paths, entanglement swapping is performed to generate long-distance entanglement. When multiple alternative paths are available, multipath entanglement purification protocols can be applied to enhance the quality of the distributed entangled states.

5. **Analyse network viability and performance:** The performance of the quantum network is analyzed by evaluating task-specific viability measures, such as the viable radius and the average fraction of viable nodes.

References

[1] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.

[2] Rajni Bala, Md Sohel Mondal, and Siddhartha Santra. Statistical analysis of multipath entanglement purification in quantum networks. *Phys. Rev. A*, 112:032601, Sep 2025.

[3] Md Sohel Mondal, Dov Fields, Vladimir S. Malinovsky, and Siddhartha Santra. Entanglement topography of large-scale quantum networks, 2024.