**Graph Measures Report**

**World Design**

from typing import List, Optional

class Node:

    def \_\_init\_\_(self, value: int) -> None:

        self.value = value

class Edge:

    def \_\_init\_\_(self, node1: Node, node2: Node) -> None:

        self.node1 = node1

        self.node2 = node2

class Graph:

    """

    A class representing a graph.

    Attributes:

        nodes (List[Node]): A list of nodes in the graph.

        edges (List[Edge]): A list of edges in the graph.

    """

    def \_\_init\_\_(self) -> None:

        self.nodes: List[Node] = []

        self.edges: List[Edge] = []

    def add\_node(self, value: int) -> None:

        """

        Adds a new node to the graph.

        Args:

            value (int): The value of the new node.

        """

        node = Node(value)

        self.nodes.append(node)

    def add\_edge(self, value1: int, value2: int) -> None:

        """

        Adds a new edge to the graph.

        Args:

            value1 (int): The value of the first node.

            value2 (int): The value of the second node.

        """

        node1 = self.find\_node(value1)

        node2 = self.find\_node(value2)

        edge = Edge(node1, node2)

        self.edges.append(edge)

    def find\_node(self, value: int) -> Optional[Node]:

        """

        Finds a node in the graph by its value.

        Args:

            value (int): The value of the node to find.

        Returns:

            Optional[Node]: The found node, or None if not found.

        """

        for node in self.nodes:

            if node.value == value:

                return node

        return None

A diagram of a graph

Description automatically generated

When designing these classes, I chose a straightforward and intuitive object-oriented approach. The **"Node"** and **"Edge"** classes represent the fundamental components of a graph, whereas the **"Graph"** class encapsulates these components and provides methods for altering them. This architecture is straightforward to expand and modify, and it adheres to excellent software design principles.

**World metrics**

import networkx as nx

from typing import Dict

class GraphMetrics:

    """

    A class that calculates various centrality metrics for a given graph.

    Parameters:

    graph (networkx.Graph): The graph for which centrality metrics will be calculated.

    Methods:

    degree\_centrality(): Calculates the degree centrality for each node in the graph.

    closeness\_centrality(): Calculates the closeness centrality for each node in the graph.

    betweenness\_centrality(): Calculates the betweenness centrality for each node in the graph.

    """

    def \_\_init\_\_(self, graph: nx.Graph):

        self.graph = graph

    def degree\_centrality(self) -> Dict:

        """

        Calculates the degree centrality for each node in the graph.

        Returns:

        dict: A dictionary where the keys are the nodes and the values are their degree centrality scores.

        """

        return nx.degree\_centrality(self.graph)

    def closeness\_centrality(self) -> Dict:

        """

        Calculates the closeness centrality for each node in the graph.

        Returns:

        dict: A dictionary where the keys are the nodes and the values are their closeness centrality scores.

        """

        return nx.closeness\_centrality(self.graph)

    def betweenness\_centrality(self) -> Dict:

        """

        Calculates the betweenness centrality for each node in the graph.

        Returns:

        dict: A dictionary where the keys are the nodes and the values are their betweenness centrality scores.

        """

        return nx.betweenness\_centrality(self.graph)

The architecture makes use of Python's **NetworkX** library, which includes functions for calculating centrality measurements. The **"nx.betweenness\_centrality"** function is used for Betweenness Centrality, which computes the shortest pathways (geodesics) between all node pairs using Dijkstra's or Bellman-Ford's algorithms.

A paper with text and images

Description automatically generated with medium confidence

**Agent Design**

import random

import networkx as nx

from typing import Any, Dict, List

class GraphAgent:

    """

    A class representing an agent that performs actions on a graph.

    Attributes:

        start (Any): The starting node of the agent.

        target (Any): The target node that the agent wants to reach.

        graph (nx.Graph): The graph on which the agent performs actions.

        shortest\_paths (Dict[Any, Dict[Any, List[Any]]]): A dictionary containing the shortest paths between nodes.

    Methods:

        random\_walk: Performs a random walk on the graph until the target node is reached.

        shortest\_path: Moves along the shortest path from the start node to the target node.

        sense\_and\_store: Stores the current node in the agent's memory.

        get\_episode\_memory: Returns the memory of the agent, which contains the visited nodes.

        calculate\_shortest\_paths: Calculates the shortest paths between all pairs of nodes in the graph.

    """

    def \_\_init\_\_(

        self,

        start: Any,

        target: Any,

        graph: nx.Graph,

        shortest\_paths: Dict[Any, Dict[Any, List[Any]]],

    ):

        self.start = start

        self.target = target

        self.graph = graph

        self.shortest\_paths = shortest\_paths

        self.current\_node = start

        self.memory = []

    def random\_walk(self):

        while self.current\_node != self.target:

            neighbors = list(self.graph.neighbors(self.current\_node))

            self.current\_node = random.choice(neighbors)

            self.sense\_and\_store()

    def shortest\_path(self):

        path = self.shortest\_paths[self.start][self.target]

        for node in path:

            self.current\_node = node

            self.sense\_and\_store()

    def sense\_and\_store(self):

        self.memory.append(self.current\_node)

    def get\_episode\_memory(self) -> List[Any]:

        return self.memory

    def calculate\_shortest\_paths(self):

        shortest\_paths = dict(nx.all\_pairs\_shortest\_path(self.graph))

        return shortest\_paths

A diagram of a graph

Description automatically generated

**Simulation**

from GraphAgent import GraphAgent

from world import Graph

from GraphMetrics import GraphMetrics

import random

import networkx as nx

# Constants

NUM\_NODES = 5

NUM\_SIMULATIONS = 1000

# Create a Graph object

graph = Graph()

# Add nodes

for i in range(1, NUM\_NODES + 1):

    graph.add\_node(i)

# Add edges

for i in range(1, NUM\_NODES):

    graph.add\_edge(i, i + 1)

# Convert graph to a networkx.Graph object

nx\_graph = nx.Graph()

for node in graph.nodes:

    nx\_graph.add\_node(node.value)

for edge in graph.edges:

    nx\_graph.add\_edge(edge.node1.value, edge.node2.value)

# Calculate shortest paths

shortest\_paths = dict(nx.all\_pairs\_shortest\_path(nx\_graph))

def run\_simulation(start, target, graph, paths, is\_random\_walk):

    """

    Runs a simulation with the given parameters.

    Args:

        start: The starting node of the simulation.

        target: The target node of the simulation.

        graph: The graph representing the environment.

        paths: The available paths in the graph.

        is\_random\_walk: A boolean indicating whether to perform a random walk or find the shortest path.

    Returns:

        The episode memory of the agent after running the simulation.

    """

    agent = GraphAgent(start, target, graph, paths)

    if is\_random\_walk:

        agent.random\_walk()

    else:

        agent.shortest\_path()

    return agent.get\_episode\_memory()

# Run simulations

random\_walk\_results = [

    run\_simulation(

        \*random.sample(range(1, NUM\_NODES + 1), 2), nx\_graph, shortest\_paths, True

    )

    for \_ in range(NUM\_SIMULATIONS)

]

shortest\_path\_results = [

    run\_simulation(

        \*random.sample(range(1, NUM\_NODES + 1), 2), nx\_graph, shortest\_paths, False

    )

    for \_ in range(NUM\_SIMULATIONS)

]

# Print results

print("Random walk results:", random\_walk\_results)

print("Shortest path results:", shortest\_path\_results)

# Calculate metrics

metrics = GraphMetrics(nx\_graph)

degree = metrics.degree\_centrality()

closeness = metrics.closeness\_centrality()

betweenness = metrics.betweenness\_centrality()

print("Degree centrality:", degree)

print("Closeness centrality:", closeness)

print("Betweenness centrality:", betweenness)

**Evaluation**

A close-up of a sign

Description automatically generated

**Analysis of Movement Modes:** The Random Walk option most likely resulted in longer paths and more steps than the Shortest Path mode, which is optimized for efficiency. The results are expected because random walks do not use knowledge of the graph's structure, whereas shortest paths do. These findings are not surprising, as they are consistent with the fundamental concepts of graph theory.

A table with text on it

Description automatically generated

**Metric Analysis and Results** The simulation results should be consistent with graph metrics, as nodes with greater centrality values are more influential in the network. The degree of centrality correlates with the number of connections, proximity with the average distance to other nodes, and betweenness with the level of control over network flow. The results should come as no surprise if the simulations accurately reflect the graph's structure and the estimated metrics.

**Results for Graph Simulation Metrics:**

**Degree centrality:** {1: 0.25, 2: 0.5, 3: 0.5, 4: 0.5, 5: 0.25}

**Closeness centrality:** {1: 0.4, 2: 0.5714285714285714, 3: 0.6666666666666666, 4: 0.5714285714285714, 5: 0.4}

**Betweenness centrality:** {1: 0.0, 2: 0.5, 3: 0.6666666666666666, 4: 0.5, 5: 0.0}

The degree centrality results show that nodes 2, 3, and 4 are the most linked in the network, implying that they are important in the graph's connection. Closeness centrality reveals that node 3 is the most efficient at contacting other nodes, demonstrating its central location in the network. Betweenness centrality demonstrates node 3's substantial control over the flow of information. These measurements collectively indicate that node 3 is the most central and influential node in this network structure, and simulation results should reflect this by displaying node 3 as a frequent pass-through or target in the movement modes. Aligning the simulation results with these measures would corroborate the predicted behaviour given the graph's topology.