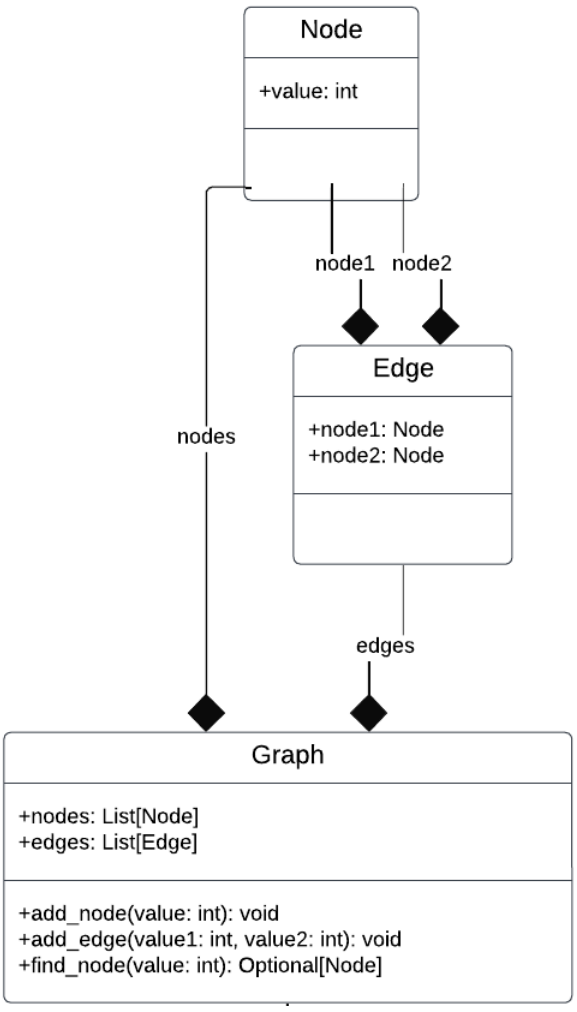
Graph Measures Report

World Design

```python
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
```



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.

2.5 Graph Centrality

A graph $\mathbf{G} = (V, E)$ consists of a set of vertices $V = \{v_1, v_2, ..., v_n\}, n = |V|$ and set of edges that connect pairs of vertices $E = \{e_1, e_2, ..., e_m\}, e = (u, v), u \in V, v \in V, m = |E|$. The graphs used in this thesis are *directed*, meaning that each edge is an ordered pair of vertices, connecting the two vertices in one direction only. In a *digraph* each pair of vertices may have an edge in one or both directions. Furthermore all graphs used herein are *connected*, meaning that there is at least one path of edges connecting all pairs of vertices.

2.5.1 Degree Centrality

The *degree* of a vertex is defined as the number of other vertices to which it is connected (by outgoing edges as the graphs used here are directed graphs). Degree *Centrality* is sometimes defined as normalised degree, but following the convention in Javadi et al. (2017), the degree centrality of a vertex is defined here as simply the degree.

$$C_D(v) = \deg(v) \tag{23}$$

2.5.2 Closeness Centrality

Closeness centrality of a vertex is the reciprocal of the total shortest path distances to all other vertices (Brandes and Erlebach, 1998). $l(u, v)$ between two vertices u and v is defined as the length of the shortest path (graph geodesic) from u to v.

$$C_C(v) = \frac{1}{\sum_{u \in V} l(u, v)} \tag{24}$$

2.5.3 Betweenness Centrality

Betweenness centrality of a vertex is the fraction of all the geodesics (shortest paths) between all pairs of vertices that pass through the vertex.

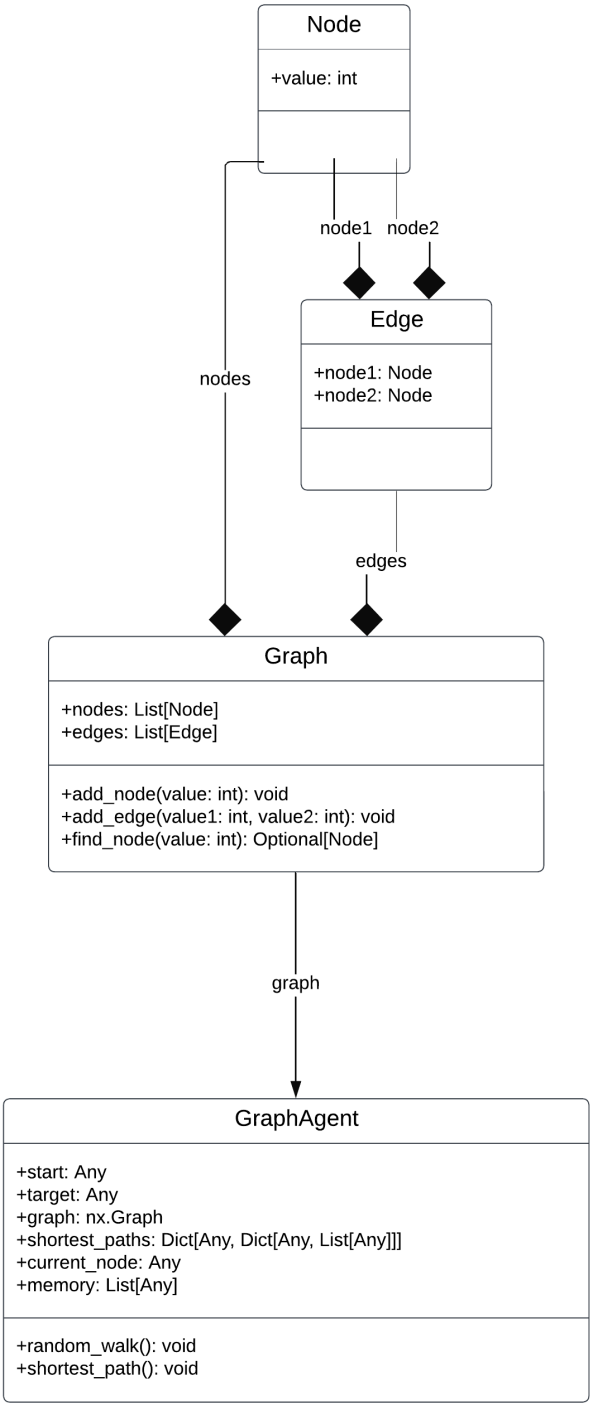
$$C_B(v) = \sum_{s \neq v \in V} \sum_{t \neq v \in V} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{25}$$

where σ_{st} is the number of geodesics between vertices s and t, and $\sigma_{st}(v)$ is the number of geodesics between s and t that pass through v (Brandes and Erlebach, 1998).

Agent Design

```python
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
```



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

Movement Mode	Description
Random Walk	The agent moves randomly from node to node without a specific path.
Shortest Path	The agent calculates the shortest path to the target and follows it.

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.

Metric	Simulation Result	Alignment
Degree Centrality	Varies by node	Reflects node connectivity
Closeness Centrality	Varies by node	Indicates efficiency of reaching other nodes
Betweenness Centrality	Varies by node	Shows control over flow

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.