# GraphWorkflow Documentation

The `GraphWorkflow` class is a pivotal part of the workflow management system, representing a directed graph where nodes signify tasks or agents and edges represent the flow or dependencies between these nodes. This class leverages the NetworkX library to manage and manipulate the directed graph, allowing users to create complex workflows with defined entry and end points.

### Attributes

Attribute	Type	Description	Default
`nodes`	`Dict[str, Node]`	A dictionary of nodes in the graph, where the key is the node ID and the value is the Node object.	`Field(default_factory=dict)`
`edges`	`List[Edge]`	A list of edges in the graph, where each edge is represented by an Edge object.	`Field(default_factory=list)`
`entry_points`	`List[str]`	A list of node IDs that serve as entry points to the graph.	`Field(default_factory=list)`
`end_points`	`List[str]`	A list of node IDs that serve as end points of the graph.	`Field(default_factory=list)`
`graph`	`nx.DiGraph`	A directed graph object from the NetworkX library representing the workflow graph.	`Field(default_factory=nx.DiGraph)`
`max_loops`	`int`	Maximum number of times the workflow can loop during execution.	`1`

### Methods

#### `add_node(node: Node)`

Adds a node to the workflow graph.

Parameter	Type	Description
`node`	`Node`	The node object to be added.

Raises:

- `ValueError`: If a node with the same ID already exists in the graph.

#### `add_edge(edge: Edge)`

Adds an edge to the workflow graph.

Parameter	Type	Description
`edge`	`Edge`	The edge object to be added.

Raises:

- `ValueError`: If either the source or target node of the edge does not exist in the graph.

#### `set_entry_points(entry_points: List[str])`

Sets the entry points of the workflow graph.

Parameter	Type	Description
`entry_points`	`List[str]`	A list of node IDs to be set as entry points.

Raises:

- `ValueError`: If any of the specified node IDs do not exist in the graph.

#### `set_end_points(end_points: List[str])`

Sets the end points of the workflow graph.

Parameter	Type	Description
`end_points`	`List[str]`	A list of node IDs to be set as end points.

Raises:

- `ValueError`: If any of the specified node IDs do not exist in the graph.

#### `visualize() -> str`

Generates a string representation of the workflow graph in the Mermaid syntax.

Returns:

- `str`: The Mermaid string representation of the workflow graph.

#### `run(task: str = None, *args, **kwargs) -> Dict[str, Any]`

Function to run the workflow graph.

Parameter	Type	Description
`task`	`str`	The task to be executed by the workflow.
`*args`		Variable length argument list.
`**kwargs`		Arbitrary keyword arguments.

Returns:
- `Dict[str, Any]`: A dictionary containing the results of the execution.

Raises:
- `ValueError`: If no entry points or end points are defined in the graph.

## Functionality and Usage

### Adding Nodes

The `add_node` method is used to add nodes to the graph. Each node must have a unique ID. If a node with the same ID already exists, a `ValueError` is raised.

```python
wf_graph = GraphWorkflow()
```

node1 = Node(id="node1", type=NodeType.TASK, callable=sample_task)
wf_graph.add_node(node1)

### Adding Edges

The `add_edge` method connects nodes with edges. Both the source and target nodes of the edge must already exist in the graph, otherwise a `ValueError` is raised.

```python
edge1 = Edge(source="node1", target="node2")
wf_graph.add_edge(edge1)
```

### Setting Entry and End Points

The `set_entry_points` and `set_end_points` methods define which nodes are the starting and ending points of the workflow, respectively. If any specified node IDs do not exist, a `ValueError` is raised.

```python
wf_graph.set_entry_points(["node1"])
wf_graph.set_end_points(["node2"])
```

### Visualizing the Graph

The `visualize` method generates a Mermaid string representation of the workflow graph. This can be useful for visualizing the workflow structure.

```python
print(wf_graph.visualize())
```

### Running the Workflow

The `run` method executes the workflow. It performs a topological sort of the graph to ensure nodes are executed in the correct order. The results of each node's execution are returned in a dictionary.

```python
results = wf_graph.run()
print("Execution results:", results)
```

## Example Usage

Below is a comprehensive example demonstrating the creation and execution of a workflow graph:

```python

import os
```

```
from dotenv import load_dotenv



from swarms import Agent, Edge, GraphWorkflow, Node, NodeType


from swarm_models import OpenAIChat


load_dotenv()


api_key = os.environ.get("OPENAI_API_KEY")


llm = OpenAIChat(
    temperature=0.5, openai_api_key=api_key, max_tokens=4000
)
agent1 = Agent(llm=llm, max_loops=1, autosave=True, dashboard=True)
agent2 = Agent(llm=llm, max_loops=1, autosave=True, dashboard=True)


def sample_task():
    print("Running sample task")
    return "Task completed"


wf_graph = GraphWorkflow()
wf_graph.add_node(Node(id="agent1", type=NodeType.AGENT, agent=agent1))
wf_graph.add_node(Node(id="agent2", type=NodeType.AGENT, agent=agent2))
wf_graph.add_node(
    Node(id="task1", type=NodeType.TASK, callable=sample_task)
)

wf_graph.add_edge(Edge(source="agent1", target="task1"))

wf_graph.add_edge(Edge(source="agent2", target="task1"))


wf_graph.set_entry_points(["agent1", "agent2"])

wf_graph.set_end_points(["task1"])


print(wf_graph.visualize())


# Run the workflow

results = wf_graph.run()

print("Execution results:", results)

```

In this example, we set up a workflow graph with two agents and one task. We define the entry and end points, visualize the graph, and then execute the workflow, capturing and printing the results.

## Additional Information and Tips

- **Error Handling**: The `GraphWorkflow` class includes error handling to ensure that invalid operations (such as adding duplicate nodes or edges with non-existent nodes) raise appropriate exceptions.
- **Max Loops**: The `max_loops` attribute allows the workflow to loop through the graph multiple times if needed. This can be useful for iterative tasks.
- **Topological Sort**: The workflow execution relies on a topological sort to ensure that nodes are

processed in the correct order. This is particularly important in complex workflows with dependencies.

## References and Resources

- [NetworkX Documentation](https://networkx.github.io/documentation/stable/)
- [Pydantic Documentation](https://pydantic-docs.helpmanual.io/)
- [Mermaid Documentation](https://mermaid-js.github.io/mermaid/#/)