open_digraph.py

Exercise 1)

```
def iparallel(self, g) -> None:
    """
    Appends the graph g to self in parallel without modifying g.
    :param g: OpenDigraph; the graph to be appended in parallel
    """
    # Find the maximum index in self and the minimum index in g
    max_self_index = self.max_id()
    min_g_index = g.min_id()

    # Translate the indices of g by max_self_index - min_g_index + 1
    m = max_self_index - min_g_index + 1
    g_copy = g.copy()
    if m > 0:
        g_copy.shift_indices(m)

    # Add the nodes and connections of g to self
    for node in g_copy.get_input_ids():
        self.inputs.append(node)
    for node in g_copy.get_output_ids():
        self.outputs.append(node)
    for node in g_copy.get_node_ids():
        self.nodes[node] = g_copy.nodes[node]

def parallel(self, g1, g2) -> None:
    """
    Returns a new graph which is the parallel composition of g1 and g2
without modifying them.
    :param g1: OpenDigraph; the first graph
    :param g2: OpenDigraph; the second graph
    """
    # Find the maximum index in self and the minimum index in g
    max_g1_index = g1.max_id()
    min_g2_index = g2.min_id()

    # Translate the indices of g by max_self_index - min_g_index + 1
    m = max_g1_index - min_g2_index + 1
    g2_copy = g2.copy()
    if m > 0:
        g2_copy.shift_indices(m)

    # Add the nodes and connections of g1 to the new graph
    for node in g1.get_input_ids():
        self.inputs.append(node)
    for node in g1.get_output_ids():
        self.outputs.append(node)
    for node in g1.get_node_ids():
        self.nodes[node] = g1.nodes[node]

    # Add the nodes and connections of g2 to the new graph
    for node in g2_copy.get_input_ids():
        self.inputs.append(node)
    for node in g2_copy.get_output_ids():
        self.outputs.append(node)
    for node in g2_copy.get_node_ids():
        self.nodes[node] = g2_copy.nodes[node]
```

Exercise 2)

```
def icompose(self, f) -> None:
    """
    Performs the sequential composition of self and f.
    The inputs of self should be connected to the outputs of f.
    :param f: OpenDigraph; the graph to be composed sequentially with self
    """
    # Check that the number of outputs of f = the number of inputs of self
    if len(self.get_input_ids()) != len(f.get_output_ids()):
        raise ValueError("Number of outputs from f doesn't match the number
of inputs of self.")

    # Find the maximum index in self and the minimum index in g
    max_self_index = self.max_id()
    min_f_index = f.min_id()

    # Translate the indices of g by max_self_index - min_g_index + 1
    m = max_self_index - min_f_index + 1
    f_copy = f.copy()
    if m > 0:
        f_copy.shift_indices(m)

    # Add the nodes of f to self
    for node in f_copy.get_node_ids():
        self.nodes[node] = f_copy.nodes[node]

    # Connect outputs of f to inputs of self
    inputs = self.get_input_ids()
    outputs = f_copy.get_output_ids()
    for i in range(len(f.outputs)):
        self.get_node_by_id(inputs[i]).add_parent_id(outputs[i])
        self.get_node_by_id(outputs[i]).add_child_id(inputs[i])

    # New inputs are inputs of f
    self.inputs = f_copy.get_input_ids()

def compose(self, f1, f2) -> None:
    """
    Returns a third graph, which is the composition of f1 and f2, without
modifying them.
    :param f1: OpenDigraph; the first graph
    :param f2: OpenDigraph; the second graph
    """
    # Check that the number of outputs of f = the number of inputs of self
    if len(f1.get_input_ids()) != len(f2.get_output_ids()):
        raise ValueError("Number of outputs from f1 doesn't match the
number of inputs of f2.")

    # Find the maximum index in self and the minimum index in g
    max_f1_index = f1.max_id()
    min_f2_index = f2.min_id()

    # Translate the indices of g by max_self_index - min_g_index + 1
    m = max_f1_index - min_f2_index + 1
    f2_copy = f2.copy()
    if m > 0:
        f2_copy.shift_indices(m)

    # Add the nodes of f1 to self
    for node in f1.get_node_ids():
        self.nodes[node] = f1.nodes[node]
```

```
    # Add the nodes of f2 to self
    for node in f2_copy.get_node_ids():
        self.nodes[node] = f2_copy.nodes[node]

    # Connect outputs of f2 to inputs of f1
    inputs = f1.get_input_ids()
    outputs = f2_copy.get_output_ids()
    for i in range(len(f2.get_output_ids())):
        self.get_node_by_id(inputs[i]).add_parent_id(outputs[i])
        self.get_node_by_id(outputs[i]).add_child_id(inputs[i])

    # New inputs are inputs of f2
    self.inputs = f2_copy.get_input_ids()
    # New outputs are outputs of f1
    self.outputs = f1.get_output_ids()
```

Exercise 3)

```
@classmethod
def identity(cls, n: int) -> 'OpenDigraph':
    """
    Creates an open_digraph representing the identity over n children.
    :param n: int; number of children
    :return: OpenDigraph; the idendity over n children graph
    """
    t = [i for i in range(n)]
    nodes = [Node(identity=i, label='&', parents={}, children={}) for i in
range(n)]

    # Connect each node to itself
    for i in range(n):
        nodes[i].add_child_id(i)
        nodes[i].add_parent_id(i)

    return cls(inputs=t, outputs=t, nodes=nodes)
```

Exercises 4 and 5)

```
def connected_components(self) -> Tuple[int, Dict[int, int],
List['OpenDigraph']]:
    """
    Returns the number of connected components of the graph and a
dictionary
    associating each node id with the number of the connected component it
belongs to,
    plus a list of all components
    :return: Tuple[int, Dict[int, int], List[OpenDigraph]]; number of
connected components, a
             dictionary mapping node IDs to their connected component
number,
             and a list of OpenDigraph, each corresponding to a component
    """
    visited = set()
    dic = {}
    cpt = 0
    nodes = self.get_nodes()

    # Helper function for DFS traversal
    def dfs_util(node_id):
        if node_id in visited:
```

```
            return  # Node already visited
        visited.add(node_id)
        dic[node_id] = cpt

        # Explore all children of the current node
        for child_id in nodes[node_id].get_children():
            dfs_util(child_id)

        # Explore all parents of the current node
        for parent_id in nodes[node_id].get_parents():
            dfs_util(parent_id)

    # Start DFS for each node that is still unmarked
    for node in nodes:
        if node.get_id() not in visited:
            dfs_util(node.get_id())
            cpt += 1

    # Recreate all components
    res = []
    components = [i for i in range(cpt)]
    self_nodes = self.get_nodes()

    for component in components:
        # Get the nodes of the current component
        nodes = {i: self_nodes[i] for i in self.get_node_ids() if dic[i] ==
component}

        # Create new node IDs
        new_ids = {old_id: new_id for new_id, old_id in
enumerate(sorted(nodes.keys()))}

        # Create new inputs and outputs
        new_inputs = [new_ids[i] for i in self.get_input_ids() if i in
nodes]
        new_outputs = [new_ids[i] for i in self.get_output_ids() if i in
nodes]

        # Create new nodes with news IDs and their respective connections
        new_nodes = []
        for old_id in nodes:
            new_id = new_ids[old_id]
            parents = nodes[old_id].get_parents()
            children = nodes[old_id].get_children()
            new_parents = {new_ids[i]: parents[i] for i in parents if i in
new_ids}
            new_children = {new_ids[i]: children[i] for i in children if i
in new_ids}
            new_nodes.append(Node(new_id, nodes[old_id].get_label(),
new_parents, new_children))

        # Add the new subgraph
        res.append(OpenDigraph(new_inputs, new_outputs, new_nodes))

    return cpt, dic, res
```

open_digraph_test.py

Exercise 1)

```
def test_iparallel_OpenDigraph(self):
    n0 = Node(0, '&', {}, {})
    n1 = Node(1, '&', {}, {})
    n2 = Node(2, '|', {}, {})
    n3 = Node(3, '|', {}, {})
    g = OpenDigraph([0], [1], [n0, n1])
    g1 = OpenDigraph([2], [3], [n2, n3])
    g1_bis = OpenDigraph([2], [3], [n2, n3])
    g2 = OpenDigraph([0, 2], [1, 3], [n0, n1, n2, n3])

    g.iparallel(g1)
    self.assertEqual(g1, g1_bis)
    self.assertEqual(g, g2)

def test_parallel_OpenDigraph(self):
    n0 = Node(0, '&', {}, {})
    n1 = Node(1, '&', {}, {})
    n2 = Node(2, '|', {}, {})
    n3 = Node(3, '|', {}, {})
    g = OpenDigraph([0], [1], [n0, n1])
    g_bis = OpenDigraph([0], [1], [n0, n1])
    g1 = OpenDigraph([2], [3], [n2, n3])
    g1_bis = OpenDigraph([2], [3], [n2, n3])
    g2 = OpenDigraph([0, 2], [1, 3], [n0, n1, n2, n3])

    g3 = OpenDigraph()
    g3.parallel(g, g1)
    self.assertEqual(g, g_bis)
    self.assertEqual(g1, g1_bis)
    self.assertEqual(g2, g3)
```

Exercise 2)

```
def test_icompose_OpenDigraph(self):
    n0 = Node(0, '&', {}, {})
    n1 = Node(1, '&', {}, {})
    n2 = Node(2, '|', {}, {})
    n3 = Node(3, '|', {}, {})
    n0_bis = Node(0, '&', {3: 1}, {})
    n3_bis = Node(3, '|', {}, {0: 1})
    f = OpenDigraph([0], [1], [n0, n1])
    f1 = OpenDigraph([2], [3], [n2, n3])
    f1_bis = OpenDigraph([2], [3], [n2, n3])
    f2 = OpenDigraph([2], [1], [n0_bis, n1, n2, n3_bis])
    f3 = OpenDigraph([0], [], [n0, n2, n3])

    f.icompose(f1)
    self.assertEqual(f1, f1_bis)
    self.assertEqual(f, f2)
    with self.assertRaises(ValueError):
        f1.icompose(f3)

def test_compose_OpenDigraph(self):
```

```
    n0 = Node(0, '&', {}, {})
    n1 = Node(1, '&', {}, {})
    n2 = Node(2, '|', {}, {})
    n3 = Node(3, '|', {}, {})
    n0_bis = Node(0, '&', {3: 1}, {})
    n3_bis = Node(3, '|', {}, {0: 1})
    f = OpenDigraph([0], [1], [n0, n1])
    f_bis = OpenDigraph([0], [1], [n0, n1])
    f1 = OpenDigraph([2], [3], [n2, n3])
    f1_bis = OpenDigraph([2], [3], [n2, n3])
    f2 = OpenDigraph([2], [1], [n0_bis, n1, n2, n3_bis])
    f5 = OpenDigraph([0], [], [n0, n2, n3])

    f3 = OpenDigraph()
    f4 = OpenDigraph()
    f3.compose(f, f1)
    self.assertEqual(f, f_bis)
    self.assertEqual(f1, f1_bis)
    self.assertEqual(f2, f3)
    with self.assertRaises(ValueError):
        f4.compose(f1, f5)
```

Exercise 3)

```
def test_identity_OpenDigraph(self):
    g = OpenDigraph.identity(3)

    self.assertEqual(g.get_input_ids(), [0, 1, 2])
    self.assertEqual(g.get_output_ids(), [0, 1, 2])

    # Vérification des nœuds
    self.assertEqual(len(g.get_nodes()), 3)

    # Vérification des connexions
    for node_id in range(3):
        node = g.get_node_by_id(node_id)
        self.assertEqual(node.get_parents(), {node_id: 1})
        self.assertEqual(node.get_children(), {node_id: 1})
```