**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})