open_digraph.py

Exercise 1)

```
def random_int_list(n: int, bound: int, unique=False) -> List[int]:
    """
    Returns a list of n random integers between 0 and n
    :param n: int; numbers of integers wanted
    :param bound: int; maximum value of the integers
    :param unique: bool; set True if you want only unique integers
    """
    if unique and n > bound + 1:
        raise ValueError("Bound too small compared to n")
    res = []
    for _ in range(n):
        tmp = randint(0, bound)
        if unique:
            while tmp in res:  # All numbers must be different to be IDs
                tmp = randint(0, bound)
        res.append(tmp)
    return res
```

Exercise 2 to 6)

```
def random_int_matrix(n: int, bound: int, unique=False, null_diag=True,
symmetric=False,
                      oriented=False, dag=False) -> List[List[int]]:
    """
    Returns a matrix of nxn random integers between 0 and n
    :param n: int; numbers of rows and columns wanted
    :param bound: int; maximum value of the integers
    :param unique: bool; set True if you want only unique integers
    :param null_diag: bool; set True if you want the diagonal to be zeros
    :param symmetric: bool; set True if you want the matrix to be symmetric
    :param oriented: bool; set True if you want the matrix to define an
oriented graph
    :param dag: bool; set True if you want the matrix to define an acyclic
graph
    """
    if symmetric and (oriented or dag):
        raise ValueError("Matrix cannot be symmetric and oriented/acyclic")

    res = []
    for _ in range(n):
        res.append(random_int_list(n, bound, unique))

    if null_diag:
        for i in range(n):
            res[i][i] = 0

    if symmetric:
        for i in range(n):
            for j in range(i+1, n):
                res[j][i] = res[i][j]

    if oriented:
        for i in range(n):
            for j in range(i+1, n):
                if res[i][j] > 0:
```

```
                    res[j][i] = 0

    if dag:
        for i in range(n):
            for j in range(i+1, n):
                res[j][i] = 0

    return res
```

Exercise 7)

```
def graph_from_adjacency_matrix(matrix: List[List[int]]) -> OpenDigraph:
    """
    Returns an OpenDigraph from an adjency matrix
    :param matrix: List[List[int]]; the ajdency matrix
    """
    n = len(matrix)
    nodes = []
    for identity in range(n):
        children = {}
        parents = {}
        for i in range(n):
            if len(matrix[i]) != n or len(matrix[identity]) != n:
                raise ValueError("The matrix is not a squared matrix")

            if matrix[identity][i]:
                children[i] = matrix[identity][i]

            if matrix[i][identity]:
                parents[i] = matrix[i][identity]

        node = Node(identity, str(identity), parents, children)
        nodes.append(node)

    return OpenDigraph([], [], nodes)
```

Exercise 8)

```
@classmethod
def random(cls, n, bound, inputs=0, outputs=0, form="free") ->
'OpenDigraph':
    """
    Generates a random graph according to the constraints given by the
user.
    :param n: int; number of nodes in the graph
    :param bound: int; maximum value for edge weights
    :param inputs: int; number of input nodes (if 0, randomly chosen)
    :param outputs: int; number of output nodes (if 0, randomly chosen)
    :param form: str; form of the graph
    :return: OpenDigraph; randomly generated graph
    """
    if inputs < 0 or outputs < 0 or n < inputs + outputs:
        raise ValueError("Invalid input/output values")

    # Generate adjacency matrix according to the specified form
    if form == "free":
        matrix = random_int_matrix(n, bound)
    elif form == "DAG":
        matrix = random_int_matrix(n, bound, dag=True)
    elif form == "oriented":
```

```
        matrix = random_int_matrix(n, bound, oriented=True)
    elif form == "loop-free":
        matrix = random_int_matrix(n, bound, null_diag=True)
    elif form == "undirected":
        matrix = random_int_matrix(n, bound, symmetric=True)
    elif form == "loop-free_undirected":
        matrix = random_int_matrix(n, bound, symmetric=True,
null_diag=True)
    else:
        raise ValueError("Invalid graph form")

    # Create OpenDigraph instance from adjacency matrix
    graph = graph_from_adjacency_matrix(matrix)

    # Select inputs and outputs randomly (by checking for every node if
it's a possible input/output node)
    node_ids = graph.get_node_ids()

    inputs_list = [i for i in node_ids if
len(graph.get_node_by_id(i).get_parents()) == 0 and
                   len(graph.get_node_by_id(i).get_children()) == 1]
    if len(inputs_list) < inputs:
        raise ValueError("This graph has too few possibilities for inputs
nodes")
    inputs_list = sample(node_ids, inputs)

    outputs_list = [i for i in node_ids if
len(graph.get_node_by_id(i).get_children()) == 0 and
                    len(graph.get_node_by_id(i).get_parents()) == 1 and i
not in inputs_list]
    if len(outputs_list) < outputs:
        raise ValueError("This graph has too few possibilities for outputs
nodes")
    outputs_list = sample(node_ids, outputs)

    for node_id in inputs_list:
        graph.add_input_id(node_id)
    for node_id in outputs_list:
        graph.add_output_id(node_id)

    return graph
```

Exercise 9)

```
def node_id_to_index_map(self) -> Dict[int, int]:
    """
    Returns a dictionary mapping each node ID to a unique integer index.
    The indices are in the range 0 ≤ i < n, where n is the number of nodes
in the graph.
    :return: Dict[int, int];
    """
    node_ids = sorted(self.get_node_ids())  # Sort the node IDs
    node_index_map = {node_id: index for index, node_id in
enumerate(node_ids)}  # Map each node ID to its index
    return node_index_map
```

Exercise 10)

```
def adjacency_matrix(self) -> List[List[int]]:
    """
    Generates an adjacency matrix for the graph, ignoring inputs and
outputs.
    Considers all nodes in the graph.
    :return: List[List[int]]; The adjacency matrix representing the
connections between nodes.
    """
    # Get all nodes and their IDs
    nodes = self.get_nodes()
    node_ids = self.get_node_ids()

    # Initialize the adjacency matrix
    n = len(node_ids)
    adj_matrix = [[0 for _ in range(n)] for _ in range(n)]

    # Populate the adjacency matrix based on connections between nodes
    for node in nodes:
        node_id = node.get_id()
        children = node.get_children()
        for child_id, child_value in children.items():
            adj_matrix[node_id][child_id] = child_value  # Set the
corresponding cell to 1

    return adj_matrix
```

open_digraph_test.py

Exercise 1 to 6)

```
def test_random_int_matrix(self):
    with self.assertRaises(ValueError):
        random_int_matrix(5, 4)
        random_int_matrix(5, 6, symmetric=True, oriented=True)

    m = random_int_matrix(5, 10)
    for i in range(5):
        self.assertEqual(m[i][i], 0)

    m = random_int_matrix(5, 10, symmetric=True)
    for i in range(5):
        for j in range(i+1, 5):
            self.assertEqual(m[i][j], m[j][i])

    m = random_int_matrix(5, 10, oriented=True)
    for i in range(5):
        for j in range(i+1, 5):
            if m[i][j] > 0:
                self.assertEqual(0, m[j][i])

    m = random_int_matrix(5, 10, dag=True)
    for i in range(5):
        for j in range(i+1, 5):
            self.assertEqual(0, m[j][i])
```

Exercise 7)

```
def test_graph_from_adjacency_matrix(self):
    m = [[0, 1, 1, 0, 0],
         [0, 0, 0, 1, 2],
         [0, 0, 0, 2, 0],
         [1, 0, 0, 0, 1],
         [0, 0, 0, 0, 0]]
    n1 = Node(0, '0', {3: 1}, {1: 1, 2: 1})
    n2 = Node(1, '1', {0: 1}, {3: 1, 4: 2})
    n3 = Node(2, '2', {0: 1}, {3: 2})
    n4 = Node(3, '3', {1: 1, 2: 2}, {0: 1, 4: 1})
    n5 = Node(4, '4', {1: 2, 3: 1}, {})
    self.assertEqual(OpenDigraph([], [], [n1, n2, n3, n4, n5]),
graph_from_adjacency_matrix(m))
    with self.assertRaises(ValueError):
        graph_from_adjacency_matrix([[1, 1, 1], [1, 1, 1]])
```

Exercise 8)

```
# For these tests, we need to test if the code either is a
well_formed_graph or raises an error
# See next class how to do it correctly with no try/except
def test_random_OpenDigraph(self):
    # Free Form
    g = OpenDigraph.random(n=10, bound=9, form='free')
    self.assertTrue(g.is_well_formed())

    # DAG Form
```

```
    g = OpenDigraph.random(n=10, bound=9, form='DAG')
    self.assertTrue(g.is_well_formed())

    # Oriented Form
    g = OpenDigraph.random(n=10, bound=9, form='oriented')
    self.assertTrue(g.is_well_formed())

    # Loop-Free Form
    g = OpenDigraph.random(n=10, bound=9, form='loop-free')
    self.assertTrue(g.is_well_formed())

    # Undirected Form
    g = OpenDigraph.random(n=10, bound=9, form='undirected')
    self.assertTrue(g.is_well_formed())

    # Loop-Free Undirected Form
    g = OpenDigraph.random(n=10, bound=9, form='loop-free_undirected')
    self.assertTrue(g.is_well_formed())

    # Inputs/Outputs Consistency
    with self.assertRaises(ValueError):
        OpenDigraph.random(n=10, bound=9, inputs=5, outputs=5)
        self.assertEqual(len(graph.get_input_ids()), 5)
        self.assertEqual(len(graph.get_output_ids()), 5)

    # Invalid Form
    with self.assertRaises(ValueError):
        OpenDigraph.random(n=10, bound=9, form='invalid_form')

    # Invalid Inputs/Outputs Values
    with self.assertRaises(ValueError):
        OpenDigraph.random(n=10, bound=9, inputs=5, outputs=6,
form='oriented')
        OpenDigraph.random(n=10, bound=9, inputs=11, outputs=0,
form='oriented')
        OpenDigraph.random(n=10, bound=9, inputs=0, outputs=11,
form='oriented')
```

Exercise 10)

```
def test_adjency_matrix_OpenDigraph(self):
    m = [[0, 1, 1, 0, 0],
         [0, 0, 0, 1, 2],
         [0, 0, 0, 2, 0],
         [1, 0, 0, 0, 1],
         [0, 0, 0, 0, 0]]
    n1 = Node(0, '0', {3: 1}, {1: 1, 2: 1})
    n2 = Node(1, '1', {0: 1}, {3: 1, 4: 2})
    n3 = Node(2, '2', {0: 1}, {3: 2})
    n4 = Node(3, '3', {1: 1, 2: 2}, {0: 1, 4: 1})
    n5 = Node(4, '4', {1: 2, 3: 1}, {})
    g = OpenDigraph([], [], [n1, n2, n3, n4, n5])
    self.assertEqual(m, g.adjacency_matrix())
```