```
 1: from node import Node
 2: from graph import Graph
 3: from edge import Edge
 4:
 5:
 6: import unittest
 7:
 8: class TestGraph(unittest.TestCase):
 9:     """
10:     Classe de tests unitaires de la classe Graph. On ne teste pas
11:     le nombre de noeuds ni d'aretes pour des questions de variables
12:     locales et variables de classe.
13:     """
14:
15:     def setUp(self):
16:         self.__graph= Graph('Graphe')
17:
18:     def test_add_edge(self):
19:         "Verifie qu'on ne peut pas ajouter une arete sans ajouter les noeuds"
20:         node1 = Node(1)
21:         node2 = Node(2)
22:         with self.assertRaises(KeyError):
23:             self.__graph.add_edge(Edge(1,node1, node2))
24:
25:
26:     def test_get_nodes(self):
27:         "Verifie qu'on obtient bien la liste vide avec get_nodes sur un graphe vide"
28:         self.assertEqual(self.__graph.get_nodes(),[])
29:
30:
31:     def test_get_edges(self):
32:         "Verifie qu'on obtient bien la liste vide avec get_edges sur un graphe vide"
33:         self.assertEqual(self.__graph.get_edges(), [])
34:
35:     def test_nb_nodes(self):
36:         "Teste le nombre de noeuds"
37:         node = Node(1)
38:         self.__graph.add_node(node)
39:         self.assertEqual(self.__graph.get_nb_nodes(),1)
40:
41:     def test_nb_edges(self):
42:         "Teste le nombre d'aretes"
43:         node1 = Node(1)
44:         node2 = Node(2)
45:         self.__graph.add_node(node1)
46:         self.__graph.add_node(node2)
47:         self.__graph.add_edge(Edge(0,node1, node2))
48:         self.assertEqual(self.__graph.get_nb_edges(),1)
49:
50:     def test_weight(self):
51:         "Teste le calcul du poids du graphe"
52:         node1 = Node(1)
53:         node2 = Node(2)
54:         self.__graph.add_node(node1)
55:         self.__graph.add_node(node2)
56:         self.__graph.add_edge(Edge(0, node1, node2,1))
57:         self.assertEqual(self.__graph.tree_weight(),1)
58:
59:
60:
61:
62:
63: if __name__ == "__main__":
64:     unittest.main()
```

```
 1: import numpy as np
 2: from node import Node
 3: from disjoint_set import DisjointSet
 4:
 5: class Graph(object):
 6:     """
 7:     Une classe generique pour representer un graphe comme un ensemble de
 8:     noeuds.
 9:     """
10:     def __init__(self, name='Sans nom'):
11:         self.__name = name
12:         self.__adj = {} # Matrice d'adjacence
13:         self.__edges = 0
14:
15:     def add_node(self, node):
16:         "Ajoute un noeud au graphe."
17:         self.__adj.setdefault(node,{})
18:
19:     def retrieve_nodes_from_id(self, *ids):
20:         "Renvoie les noeuds a partir de leurs ids."
21:         return [node for node in self.__adj.keys()
22:                 if node.get_id() in ids]
23:
24:     def add_edge(self, edge):
25:         "Ajoute une arete au graphe."
26:         (n1, n2) = edge.get_nodes()
27:         try:
28:             # checking if both nodes are there
29:             nodes = [self.__adj[n1], self.__adj[n2]]
30:             # adding the new edge
31:             self.__adj[n2][n1] = self.__adj[n1][n2] = edge
32:             self.__edges = edge.get_id() + 1
33:         # if one (or both) node(s) missing
34:         except KeyError as ke:
35:             raise KeyError("At least node {0} missing. Add all nodes before\
36:                     adding edges.".format(ke))
37:
38:     def get_name(self):
39:         "Donne le nom du graphe."
40:         return self.__name
41:
42:     def get_nodes(self):
43:         "Donne la liste des noeuds du graphe."
44:         return self.__adj.keys()
45:
46:     def get_nb_nodes(self):
47:         "Donne le nombre de noeuds du graphe."
48:         return len(self.get_nodes())
49:
50:     def get_edges(self):
51:         "Donne la liste des aretes du graphe."
52:         edges = []
53:         edges.extend([v for n in self.get_nodes() for v in self.__adj[n].values()])
54:         return list(set(edges)) # removing doubles
55:
56:     def get_nb_edges(self):
57:         "Donne le nombre d'aretes du graphe."
58:         return self.__edges
59:
60:     def tree_weight(self):
61:         "Calcule le poids de l'arbre."
62:         return sum([e.get_weight() for e in self.get_edges()])
63:
64:     def kruskal(self):
65:         "Retourne un arbre de recouvrement minimal s'il existe"
66:         min_tree = Graph('Arbre Minimal')
67:
68:         disj_sets = {}
69:
70:         nodes = self.get_nodes()
71:         # Le nombre de noeuds du graphe
72:         nb_nodes = self.get_nb_nodes
73:         # On remplit le dictionnaire de disjoint_sets
74:         for node in nodes:
75:             disj_sets[node] = DisjointSet(node)
76:
77:         edges = self.get_edges()
```

```
78:         # La liste est triee selon la comparaison implementee dans edge
79:         edges.sort()
80:
81:         # Construction de l'arbre
82:         for edge in edges:
83:             (node1,node2) = edge.get_nodes()
84:
85:             # Si l'union des deux disjoint_sets est reussie
86:             if disj_sets[node1].union_sets(disj_sets[node2]):
87:                 # On complete l'arbre minimal
88:                 min_tree.add_node(node1)
89:                 min_tree.add_node(node2)
90:                 min_tree.add_edge(edge)
91:             # Si tous les noeuds sont dans min_tree, c'est que l'arbre est fini
92:             if min_tree.get_nb_nodes() == nb_nodes:
93:                 break
94:
95:         return min_tree
96:
97:     def plot_graph(self):
98:         "Representation graphique du graphe avec Matplotlib."
99:
100:         import matplotlib.pyplot as plt
101:         from matplotlib.collections import LineCollection
102:
103:         fig = plt.figure()
104:         ax = fig.add_subplot(111)
105:
106:         # Plot nodes
107:         nodes = self.get_nodes()
108:         try:
109:             x = [node.get_data()[0] for node in nodes]
110:             y = [node.get_data()[1] for node in nodes]
111:
112:             # Plot edges
113:             edges = self.get_edges()
114:             edge_pos = np.asarray([(e.get_nodes()[0].get_data(),
115:                 e.get_nodes()[1].get_data()) for e in edges])
116:             edge_collection = LineCollection(edge_pos, linewidth=1.5,
117:                     antialiased=True, colors=(.8, .8, .8), alpha=.75, zorder=0)
118:             ax.add_collection(edge_collection)
119:             ax.scatter(x, y, s=35, c='r', antialiased=True, alpha=.75, zorder=1)
120:             ax.set_xlim(min(x) - 10, max(x) + 10)
121:             ax.set_ylim(min(y) - 10, max(y) + 10)
122:
123:             plt.ion()
124:             plt.show()
125:             plt.pause(0.001)
126:         except TypeError:
127:             print "Cannot display graph without node coordinates."
128:         return
129:
130:     def __repr__(self):
131:         name = self.get_name()
132:         nb_nodes = self.get_nb_nodes()
133:         nb_edges = self.get_nb_edges()
134:         s = 'Graphe %s comprenant %d noeuds et %d aretes' % (name, nb_nodes, nb_edges)
135:         for node in self.get_nodes():
136:             s += '\n  ' + repr(node)
137:         for edge in self.get_edges():
138:             s += '\n  ' + repr(edge)
139:         s += '\n' + 'Poids total : ' + repr(self.tree_weight())
140:         return s
141:
142:
143: if __name__ == '__main__':
144:     from node import Node
145:     from edge import Edge
146:     G = Graph(name='Graphe test')
147:     count = 0
148:     for k in range(5):
149:         n1 = Node(iden=count, name='test %d' % count)
150:         G.add_node(n1)
151:         count += 1
152:         n2 = Node(iden=count, name='test %d' % count)
153:         G.add_node(n2)
154:         count += 1
```

```
155:             G.add_edge(Edge(k, n1, n2, weight=42))
156:         print G
```

```
1: from node import Node
2: from disjoint_set import DisjointSet
3:
4: import unittest
5:
6: class TestDisjointSet(unittest.TestCase):
7:     """
8:     Classe de tests unitaires de la classe DisjointSet
9:     """
10:
11:     def setUp(self):
12:         node0 = Node(iden = 0)
13:         node1 = Node(iden = 1)
14:         self.__disjoint_set = DisjointSet(node0)
15:         self.__other_disjoint_set = DisjointSet(node1)
16:
17:
18:     def test_set_node(self):
19:         "Verifie qu'on ne peut pas modifier l'attribut noeud"
20:         n = Node(iden = 1)
21:         with self.assertRaises(AttributeError):
22:             self.__disjoint_set.node = n
23:
24:
25:     def test_set_parent(self):
26:         "Verifie qu'on ne peut pas modifier un parent existant"
27:         n1 = Node(iden = 1)
28:         self.__disjoint_set.parent = n1
29:         n2 = Node(iden = 2)
30:         with self.assertRaises(AttributeError):
31:             self.__disjoint_set.parent = n2
32:
33:     def test_union_true(self):
34:         "Verifie qu'on peut joindre deux sets non connexes."
35:         self.assertEqual(self.__disjoint_set.union_sets(self.__other_disjoint_set), True)
36:
37:     def test_union_false(self):
38:         "Verifie qu'on ne peut pas joindre deux sets connexes."
39:         self.__disjoint_set.union_sets(self.__other_disjoint_set)
40:         self.assertEqual(self.__disjoint_set.union_sets(self.__other_disjoint_set), False)
41:
42:     def test_find_root(self):
43:         "Verifie qu'on obtient la racine s'il s'agit de l'element courant"
44:         self.assertEqual(self.__disjoint_set.find_root(), self.__disjoint_set)
45:
46:     def test_find_root_rec(self):
47:         "Verifie qu'on obtient la racine s'il ne s'agit pas de l'element courant"
48:         self.__disjoint_set.union_sets(self.__other_disjoint_set)
49:         self.assertEqual(self.__other_disjoint_set.find_root(), self.__disjoint_set)
50:
51:
52: if __name__ == "__main__":
53:     unittest.main()
54:
```

```
 1:
 2: from node import Node
 3: from edge import EdgeException
 4: from edge import Edge
 5:
 6:
 7: import unittest
 8:
 9: class TestEdge(unittest.TestCase):
10:     """
11:     Classe de tests unitaires de la classe Edge
12:     """
13:
14:     def setUp(self):
15:         node1 = Node(1)
16:         node2 = Node(2)
17:         self.__edge = Edge(1,node1,node2)
18:
19:     def test_init(self):
20:         node = Node(1)
21:         with self.assertRaises(EdgeException):
22:             edge = Edge(2,node,node,1)
23:
24:
25: if __name__ == "__main__":
26:     unittest.main()
```

```
 1: import numpy as np
 2:
 3:
 4: def read_header(fd):
 5:     "Parse a .tsp file and return a dictionary with header data."
 6:
 7:     converters = {'NAME': str, 'TYPE': str, 'COMMENT': str, 'DIMENSION': int,
 8:                   'EDGE_WEIGHT_TYPE': str, 'EDGE_WEIGHT_FORMAT': str,
 9:                   'EDGE_DATA_FORMAT': str, 'NODE_COORD_TYPE': str,
10:                   'DISPLAY_DATA_TYPE': str}
11:     sections = converters.keys()
12:     header = {}
13:
14:     # Initialize header.
15:     for section in sections:
16:         header[section] = None
17:
18:     fd.seek(0)
19:     for line in fd:
20:         data = line.split(':')
21:         firstword = data[0].strip()
22:         if firstword in sections:
23:             header[firstword] = converters[firstword](data[1].strip())
24:
25:     return header
26:
27:
28: def read_nodes(header, fd):
29:     """
30:     Parse a .tsp file and return a dictionary of nodes, of the form
31:     {id:(x,y)}. If node coordinates are not given, an empty dictionary is
32:     returned. The actual number of nodes is in header['DIMENSION'].
33:     """
34:
35:     nodes = {}
36:
37:     node_coord_type = header['NODE_COORD_TYPE']
38:     display_data_type = header['DISPLAY_DATA_TYPE']
39:     if node_coord_type not in ['TWOD_COORDS', 'THREED_COORDS'] and \
40:             display_data_type not in ['COORDS_DISPLAY', 'TWOD_DISPLAY']:
41:
42:                 # Node coordinates are not given.
43:                 return nodes
44:
45:     dim = header['DIMENSION']
46:     fd.seek(0)
47:     k = 0
48:     display_data_section = False
49:     node_coord_section = False
50:
51:     for line in fd:
52:         if line.strip() == "DISPLAY_DATA_SECTION":
53:             display_data_section = True
54:             continue
55:         elif line.strip() == "NODE_COORD_SECTION":
56:             node_coord_section = True
57:             continue
58:
59:         if display_data_section:
60:             data = line.strip().split()
61:             nodes[int(data[0]) - 1] = tuple(map(float, data[1:]))
62:             k += 1
63:             if k >= dim:
64:                 break
65:             continue
66:
67:         elif node_coord_section:
68:             data = line.strip().split()
69:             nodes[int(data[0]) - 1] = tuple(map(float, data[1:]))
70:             k += 1
71:             if k >= dim:
72:                 break
73:             continue
74:
75:     return nodes
76:
77:
```

```
  78: def read_edges(header, fd):
  79:     "Parse a .tsp file and return the collection of edges as a Python set."
  80:
  81:     edges = set()
  82:     edge_weight_format = header['EDGE_WEIGHT_FORMAT']
  83:     known_edge_weight_formats = ['FULL_MATRIX', 'UPPER_ROW', 'LOWER_ROW',
  84:                                  'UPPER_DIAG_ROW', 'LOWER_DIAG_ROW',
  85:                                  'UPPER_COL', 'LOWER_COL', 'UPPER_DIAG_COL',
  86:                                  'LOWER_DIAG_COL']
  87:     if edge_weight_format not in known_edge_weight_formats:
  88:         return edges
  89:
  90:     dim = header['DIMENSION']
  91:
  92:     def n_nodes_to_read(n):
  93:         format = edge_weight_format
  94:         if format == 'FULL_MATRIX':
  95:             return dim
  96:         if format == 'LOWER_DIAG_ROW' or format == 'UPPER_DIAG_COL':
  97:             return n+1
  98:         if format == 'LOWER_DIAG_COL' or format == 'UPPER_DIAG_ROW':
  99:             return dim-n
 100:         if format == 'LOWER_ROW' or format == 'UPPER_COL':
 101:             return n
 102:         if format == 'LOWER_COL' or format == 'UPPER_ROW':
 103:             return dim-n-1
 104:
 105:     fd.seek(0)
 106:     edge_weight_section = False
 107:     k = 0
 108:     n_edges = 0
 109:     i = 0
 110:     n_to_read = n_nodes_to_read(k)
 111:
 112:     for line in fd:
 113:         if line.strip() == "EDGE_WEIGHT_SECTION":
 114:             edge_weight_section = True
 115:             continue
 116:
 117:         if edge_weight_section:
 118:             data = line.strip().split()
 119:             n_data = len(data)
 120:
 121:             start = 0
 122:
 123:             while n_data > 0:
 124:
 125:                 # Number of items that we read on this line
 126:                 # for the current node.
 127:                 n_on_this_line = min(n_to_read, n_data)
 128:
 129:                 # Read edges.
 130:                 for j in xrange(start, start + n_on_this_line):
 131:                     n_edges += 1
 132:                     if edge_weight_format in ['UPPER_ROW', 'LOWER_COL']:
 133:                       edge = (k, i+k+1, int(data[j]))
 134:                     elif edge_weight_format in ['UPPER_DIAG_ROW', \
 135:                                                 'LOWER_DIAG_COL']:
 136:                         edge = (k, i+k, int(data[j]))
 137:                     elif edge_weight_format in ['UPPER_COL', 'LOWER_ROW']:
 138:                         edge = (i+k+1, k, int(data[j]))
 139:                     elif edge_weight_format in ['UPPER_DIAG_COL', \
 140:                                                 'LOWER_DIAG_ROW']:
 141:                         edge = (i, k, int(data[j]))
 142:                     elif edge_weight_format == 'FULL_MATRIX':
 143:                         edge = (k, i, int(data[j]))
 144:                     edges.add(edge)
 145:                     i += 1
 146:
 147:                 # Update number of items remaining to be read.
 148:                 n_to_read -= n_on_this_line
 149:                 n_data -= n_on_this_line
 150:
 151:                 if n_to_read <= 0:
 152:                     start += n_on_this_line
 153:                     k += 1
 154:                     i = 0
```

```
155:                     n_to_read = n_nodes_to_read(k)
156:
157:                 if k >= dim:
158:                     n_data = 0
159:
160:             if k >= dim:
161:                 break
162:
163:     return edges
164:
165:
166: def plot_graph(nodes, edges):
167:     """
168:     Plot the graph represented by `nodes` and `edges` using Matplotlib.
169:     Very basic for now.
170:     """
171:
172:     import matplotlib.pyplot as plt
173:     from matplotlib.collections import LineCollection
174:
175:     fig = plt.figure()
176:     ax = fig.add_subplot(111)
177:
178:     # Plot nodes.
179:     x = [node[0] for node in nodes.values()]
180:     y = [node[1] for node in nodes.values()]
181:
182:     # Plot edges.
183:     edge_pos = np.asarray([(nodes[e[0]], nodes[e[1]]) for e in edges])
184:     edge_collection = LineCollection(edge_pos, linewidth=1.5, antialiased=True,
185:                                      colors=(.8, .8, .8), alpha=.75, zorder=0)
186:     ax.add_collection(edge_collection)
187:     ax.scatter(x, y, s=35, c='r', antialiased=True, alpha=.75, zorder=1)
188:     ax.set_xlim(min(x) - 10, max(x) + 10)
189:     ax.set_ylim(min(y) - 10, max(y) + 10)
190:
191:     plt.show()
192:     return
193:
194:
195: if __name__ == "__main__":
196:
197:     import sys
198:
199:     finstance = sys.argv[1]
200:
201:     with open(finstance, "r") as fd:
202:
203:         header = read_header(fd)
204:         print 'Header: ', header
205:         dim = header['DIMENSION']
206:         edge_weight_format = header['EDGE_WEIGHT_FORMAT']
207:
208:         print "Reading nodes"
209:         nodes = read_nodes(header, fd)
210:         print nodes
211:
212:         print "Reading edges"
213:         edges = read_edges(header, fd)
214:         edge_list = []
215:         for k in range(dim):
216:             edge_list.append([])
217:         for edge in edges:
218:             if edge_weight_format in ['UPPER_ROW', 'LOWER_COL', \
219:                     'UPPER_DIAG_ROW', 'LOWER_DIAG_COL']:
220:                 edge_list[edge[0]].append({edge[1]:edge[2]})
221:             else:
222:                 edge_list[edge[1]].append({edge[0]:edge[2]})
223:         for k in range(dim):
224:             edge_list[k].sort()
225:             print k, edge_list[k]
226:
227:     if len(nodes) > 0:
228:         plot_graph(nodes, edges)
```

```
 1: class Node(object):
 2:     """
 3:     Une classe generique pour representer les noeuds d'un graphe.
 4:     """
 5:
 6:     def __init__(self, iden, name='Sans nom', data=None):
 7:         self.__name = name
 8:         self.__data = data
 9:         self.__id = iden
10:
11:     def get_name(self):
12:         "Donne le nom du noeud."
13:         return self.__name
14:
15:     def get_id(self):
16:         "Donne le numero d'identification du noeud."
17:         return self.__id
18:
19:     def get_data(self):
20:         "Donne les donnees contenues dans le noeud."
21:         return self.__data
22:
23:     def __repr__(self):
24:         id = self.get_id()
25:         name = self.get_name()
26:         data = self.get_data()
27:         s  = '%s (id %d)' % (name, id)
28:         s += ' (donnees: ' + repr(data) + ')'
29:         return s
30:
31:
32: if __name__ == '__main__':
33:
34:     nodes = []
35:     for k in range(5):
36:         nodes.append(Node(iden = k))
37:
38:     for node in nodes:
39:         print node
```

```
class Edge(object):
    """
    Une classe generique pour representer les aretes d'un graphe.
    """

    def __init__(self, iden, node1, node2, weight = 0):
      if node1.get_id() == node2.get_id() and weight != 0:
          raise EdgeException('Une arete ne peut pas pointer sur elle-meme.')
      else:
        self.__nodes = (node1, node2)
        self.__weight = weight
        self.__id = iden

    def get_nodes(self):
        "Donne les noeuds."
        return self.__nodes

    def get_weight(self):
        "Donne le poids."
        return self.__weight

    def get_id(self):
        "Donne l'identifiant."
        return self.__id

    def __le__(self, other_edge):
        "Inferieur ou egal : comparaison en fonction du poids"
        return self.__weight <= other_edge.get_weight()

    def __lt__(self, other_edge):
        "Inferieur ou egal : comparaison en fonction du poids"
        return self.__weight < other_edge.get_weight()

    def __repr__(self):
        id = self.get_id()
        weight = self.get_weight()
        nodes = self.get_nodes()
        s  = 'Arete {i} (poids : {p}) '.format(i = id, p = weight)
        s += '({0} <---> {1})'.format(nodes[0].get_id(), nodes[1].get_id())
        return s

class EdgeException(Exception):
    def __init__(self, reason):
        self.__reason = reason
    def __str__(self):
        return self.__reason


if __name__ == '__main__':
    from node import Node
    aretes = []
    n1 = Node(iden=0)
    n2 = Node(iden=1)
    for k in xrange(5):
        aretes.append(Edge(iden = k, node1=n1, node2=n2))
    for arete in aretes:
        print arete
```

```
 1: from node import Node
 2:
 3: class DisjointSet(object):
 4:     """
 5:     Classe pour representer un sous-arbre dans un ensemble disjoint
 6:     """
 7:
 8:     def __init__(self, node):
 9:         self.__node = node
10:         # Le noeud est une racine par defaut
11:         # Le parent est de type disjoint set
12:         self.__parent = None
13:
14:     @property
15:     def node(self):
16:         "Accesseur du noeud courant"
17:         return self.__node
18:
19:     @node.setter
20:     def node(self,val):
21:         "Pas de modification possible du noeud courant"
22:         raise AttributeError("Le noeud courant n'est pas modifiable directement")
23:
24:     @property
25:     def parent(self):
26:         "Accesseur du noeud parent"
27:         return self.__parent
28:
29:     @parent.setter
30:     def parent(self,val):
31:         "Modification du parent"
32:         if self.__parent == None:
33:             self.__parent = val
34:         else:
35:             raise AttributeError("Utiliser la methode union_sets pour relier deux ensembles disjoints.")
36:
37:
38:     def union_sets(self,dset):
39:         """Realise l'union de deux sous ensembles disjoints par leurs racines.
40:         Renvoie True si l'union est possible, False si les deux ensembles sont connexes"""
41:         root1 = self.find_root()
42:         root2 = dset.find_root()
43:
44:         if root1 != root2:
45:             root2.parent = root1
46:             return True
47:         else:
48:             return False
49:
50:     def find_root(self):
51:         "Renvoie la racine de l'ensemble"
52:         # Si l'element courant est la racine
53:         if self.__parent == None:
54:             return self
55:         # Sinon on applique la methode au parent
56:         return self.__parent.find_root()
57:
58:     def __repr__(self):
59:         "Affiche les identifiants des noeuds jusqu'a la racine"
60:         s = '%d' % self.__node.get_id()
61:         if self.__parent == None:
62:             return s + " = racine."
63:         s += ' --> ' + str(self.__parent)
64:         return s
65:
66:
67: if __name__ == '__main__':
68:     # Creation de trois noeuds et des trois ensembles disjoints associes
69:     node1 = Node(1)
70:     node2 = Node(2)
71:     node3 = Node(3)
72:     set1 = DisjointSet(node1)
73:     set2 = DisjointSet(node2)
74:     set3 = DisjointSet(node3)
75:     # set2 devient la racine de set1
76:     set1.parent = set2
77:     print set1
```

```
78:
79:     # On affiche la racine de set1
80:     print set1.find_root()
81:
82:     # On essaye d'unir deux membres d'un meme ensemble
83:     print set1.union_sets(set2)
84:
85:     # On unit deux ensembles disjoints et on verifie le booleen de sortie
86:     print set1.union_sets(set3)
87:     # On affiche les trois sous-ensembles initiaux
88:     print set1, set2, set3
```

```
 1: if __name__ == "__main__":
 2:
 3:     import sys
 4:
 5:     from node import Node
 6:     from edge import Edge
 7:     from graph import Graph
 8:     import read_stsp as rs
 9:
10:     finstance = sys.argv[1]
11:
12:     with open(finstance, 'r') as fd:
13:
14:         header = rs.read_header(fd)
15:         dim = header['DIMENSION']
16:         edge_weight_format = header['EDGE_WEIGHT_FORMAT']
17:
18:         nodes = rs.read_nodes(header, fd)
19:         edges = rs.read_edges(header, fd)
20:
21:         # create Graph
22:         G = Graph(name='Graphe')
23:
24:         # convert dim
25:         dim = int(dim)
26:
27:         # add nodes to graph
28:         if len(nodes) == 0:
29:             nodes = {k:None for k in xrange(dim)}
30:         for node in nodes.items():
31:             # node id
32:             n = node[0]
33:             # node data
34:             d = node[1]
35:             G.add_node(Node(iden = n, name='Noeud {}'.format(n), data=d))
36:
37:         # add edges to graph
38:         nb_edges = sum(xrange(dim))
39:         for (e, edge) in zip(xrange(nb_edges), edges):
40:             # nodes
41:             (nid1, nid2) = (edge[0], edge[1])
42:             node_list = G.retrieve_nodes_from_id(nid1, nid2)
43:             n1 = node_list[0]
44:             n2 = node_list[0] if len(node_list)==1 else node_list[1]
45:             # weight
46:             w = edge[2]
47:             # edge id
48:             G.add_edge(Edge(iden = e, node1=n1, node2=n2, weight=w))
49:
50:         G.plot_graph()
51:
52:         # Kruskal's algorithm
53:         min_tree = G.kruskal()
54:         print min_tree
55:         min_tree.plot_graph()
```