Le drone livreur, ou le nouveau voyageur de commerce - Codes sources T.I.P.E.

Godard Evann, candidat n°39875
Ringoot Axel, candidat n°28686

June 7, 2023

Sommaire :

Listing 1: **Implémentation de la classe modélisant les drones**

```
class Drone():

  def __init__(self, graphe, l_client=[], autonomie=1800):  # Autonomie en s
    self.graphe = graphe
    self.autonomie = autonomie
    self.liste_client = l_client
    self.nombre_client = len(l_client)
    self.speed = 3  # speed en m/s
    self.noeud_courant = graphe.noms_som["Base"]


## Getter/Setter :

  def get_autonomie(self):
    return self.autonomie

  def get_liste_client(self):
    return self.liste_client
```

```
19
20      def get_speed(self):
21          return self.speed
22
23      def get_noeud_courant(self):
24          return self.noeud_courant
25
26      def _set_speed(self, new_speed):
27          self.speed = new_speed
28
29      def _set_autonomie(self, new_auto):
30          self.autonomie = new_auto
31
32  ## Affichage :
33
34      def print_etat(self):
35          """
36          In : None
37          Out : None
38          Goal : informe par effet de bord de l'etat global du drone (nb de clients et
39                  autonomie restante ainsi que sa position actuelle)
40          """
```

```
41      print("{} client(s) restant a livrer.\nPosition du drone : {}.\nAutonomie restante: {} minutes\n'
42        .format(len(self.get_liste_client()),
43                self.get_noeud_courant().get_nom(),
44                round(self.get_autonomie() / 60, 1)))
45
46  ## Autres méthodes :
47
48    def deplacement(self, n):
49      """
50      In : un sommet n [Noeud]
51      Out : None
52      Goal : determine si le drone courant peut se déplacer du sommet ou il est a celui
53      pris en parametre en fonction de son autonomie restante; si oui effectue le deplacement,
54      sinon previent l'utilisateur
55      """
56      d = self.graphe.get_arcs()[self.noeud_courant.get_nom()][n.get_nom()]
57
58      assert self.autonomie >= d / self.speed, (
59        "Batterie restante dans le drone insuffisante")
60
61      self.autonomie -= d / self.speed
62      self.noeud_courant = n
```

```
63      print("Destination atteinte")
64
65    def trajet(self):
66      """
67      In : None
68      Out : None
69      Goal : lance la mission du drone : il va livrer les clients dans l'ordre donne par la liste
70      """
71      for i in range(len(self.liste_client)):
72        self.print_etat()
73        self.deplacement(self.liste_client.pop(0))
74
```

Listing 2: **Implémentation de la classe modélisant les graphes**

```
from noeud import *
from unionfind import *
from tasmin import *


class Graphe():

  def __init__(self):
    self.sommets = []
    self.noms_som = {}
    self.taille = 0
    self.arcs = {}
## Rq : les arcs sont sous forme de dico dont les clefs sont les NOMS des sommets [str] et
## les clefs sont des dicos stockant les voisins (clefs) et leur distance (valeurs associees).

## Getter/Setter :

  def get_sommets(self):
```

```
19      return self.sommets
20
21    def get_taille(self):
22      return self.taille
23
24    def get_arcs(self):
25      return self.arcs
26
27
28  ## Autres méthodes :
29
30    def ajout_sommet(self, som):
31      """
32      In : un sommet som [Noeud]
33      Out : None
34      Goal : rajoute par effet de bord le sommet en parametre dans le graphe courant
35      (i.e. creer des arcs pondérés par la distance avec tous les autres sommets du graphe)
36      """
37      self.arcs[som.get_nom()] = {}
38      for i in range(self.taille):
39        d = som.distance(self.sommets[i])
40        self.arcs[som.get_nom()][self.sommets[i].get_nom()] = d
```

```
41        self.arcs[self.sommets[i].get_nom()][som.get_nom()] = d
42
43      self.sommets.append(som)
44      self.noms_som[som.nom] = som
45      self.taille += 1
46
47    def graphe_induit(self, l):
48      """
49        In : Une liste de noeuds l [list x string]
50        Out : Un graphe res [Graphe]
51        Goal : Construire le graphe induit dans self par les sommets de l
52      """
53      res = Graphe()
54      res.taille = len(l)
55
56      for s in l:
57        res.arcs[s] = {}
58        res.sommets.append(self.noms_som[s])
59        res.noms_som[s] = self.noms_som[s]
60
61      for s in l:
62        for vois in self.get_arcs()[s].keys():
```

```
63
64          if vois in l:
65            d = self.get_arcs()[s][vois]
66            res.arcs[s][vois] = d
67            res.arcs[vois][s] = d
68
69      return res
70
71
72  def creer_graphe(l):
73    """
74    In : Une liste l de triplets contenant le nom et les coordonnées de sommets
75                  [list x string*float*float]
76    Out : Un graphe contenant autant de sommets que de triplets dans la liste
77          ayant les caractéristiques contenues dans chaque triplet [Graphe]
78    Goal : Creer un graphe contenant les "points strategiques"
79    """
80    g = Graphe()
81    for i in l:
82      n = Noeud(i[0], i[1], i[2])
83      g.ajout_sommet(n)
```

```
84
85    return g
86
87
88  def recense_aretes(g):
89    """
90    In : Un graphe g [Graphe]
91    Out : Une liste d'aretes et leur poids [list x (string*string)*float]
92    Goal : Creer une liste contenant toutes les aretes du graphe avec leur poids
93              (distance euclidienne entre les 2 extremites)
94    """
95    res = []
96    for k in g.get_arcs().keys():
97      for i in g.get_arcs()[k].keys():
98        if k < i:
99          res.append(((k, i), g.get_arcs()[k][i]))
100
101   return res
102
103
104 def kruskal(g):
105   """
```

```
    In : Un graphe g [Graphe]
    Out : Une liste d'aretes [list x string*string]
    Goal : Applique l'algorithme de Kruskal pour determiner un
             arbre couvrant de cout minimal du graphe entre
    """
    n = g.get_taille()
    l = {}
    r = {}

    for i in g.get_arcs().keys():
      l[i] = i
      r[i] = 0

    uf = UnionFind(l, r)
    tasmin = TasMin()
    aretes = recense_aretes(g)

    for e in aretes:
      tasmin.push(e[0], e[1])

    acm = []
```

```
127     taille = 0
128
129     while taille < n - 1:
130       (x, y) = tasmin.pop()
131       rep_x = uf.trouver(x)
132       rep_y = uf.trouver(y)
133
134       if rep_x != rep_y:
135         uf.union(x, y)
136         acm.append((x, y))
137         taille += 1
138
139     return acm
140
141
142 def poids_chemin(g, sommets):
143     """
144     In : Un graphe g [Graphe]
145          Une liste de sommets [list x string]
146     Out : Un flottant [float]
147     Goal : Calcule le poids du chemin décrit par la liste
148     """
```

```
149   c = 0
150   for i in range(len(sommets) - 1):
151     c += g.get_arcs()[sommets[i]][sommets[i + 1]]
152   return c
153
154
```

Listing 3: **Implémentation de la classe modélisant les noeuds/sommets**

```
1  from math import sqrt
2
3  class Noeud():
4
5    def __init__(self, nom, x, y):
6      self.nom = nom
7      self.x = x
8      self.y = y
9      self.pos = (x, y)
10
11 ## Getter/Setter :
12
13   def get_nom(self):
14     return self.nom
15
16   def get_pos(self):
17     return self.pos
18
```

```
19    def get_x(self):
20      return self.x
21
22    def get_y(self):
23      return self.y
24
25    def _set_nom(self, new_name):
26      self.nom = new_name
27
28    def _set_x(self, new_x):
29      self.x = new_x
30      self.pos = (self.x, self.y)
31
32    def _set_y(self, new_y):
33      self.y = new_y
34      self.pos = (self.x, self.y)
35
36    def _set_pos(self, new_pos):
37      new_x, new_y = new_pos
38      self.x = new_x
39      self.y = new_y
40      self.pos = new_pos
```

```
41
42
43  ## Autres méthodes :
44
45    def distance(self, som):
46      """
47      In : un sommet som [Noeud]
48      Out : la distance euclidienne entre 2 sommets [float]
49      Goal : etablir la distance euclidienne entre les 2 sommets par leurs coordonnées
50      """
51      return sqrt((self.x - som.x)**2 + (self.y - som.y)**2)
52
53  def to_str (l):
54    """
55    In : une liste de sommets l [list x Noeud]
56    Out : une liste de string [list x str]
57    Goal : donne la liste contenant le noms, dans l'ordre, des sommets contenus dans l
58    """
59    return [s.get_nom() for s in l]
60
```

Listing 4: **Implémentation de la classe modélisant les files de priorité minimale via des tas min**

```
class TasMin():

  def __init__(self):
    self.val = []
    self.size = 0

## Getter / Setter

  def get_val(self):
    return self.val

  def get_size(self):
    return self.size

  def _set_val(self, new_val):
    self.val = new_val

```

```
18    def _set_size(self, new_size):
19      self.size = new_size
20
21  ## Affichage
22
23    def __str__(self):
24      return str(self.val)
25
26    def __repr__(self):
27      return str(self.val)
28
29
30  ## Methodes
31    def est_vide(self):
32      return (self.get_size() == 0)
33
34    def push(self, x, p):
35      """
36      In : Un element x [alpha] et son poids p [float]
37      Out : None
38      Goal : Insere un element dans la file de priorite avec son poids
39      """
```

```
40      i = 0
41      while i < self.size and self.val[i][1] < p:
42        i += 1
43      self.val = self.val[:i] + [(x, p)] + self.val[i:]
44      self.size += 1
45
46    def pop(self):
47      """
48      In : None
49      Out : Un element de la liste [alpha]
50      Goal : Sort l'element de priorite minimale de la file de priorite
51      """
52      assert self.size != 0, "Tas vide"
53      v = self.val.pop(0)[0]
54      self.size -= 1
55      return v
56
57
```

Listing 5: **Implémentation de la classe modélisant les union-find de manière optimisée**

```
class UnionFind():

  def __init__(self, lien={}, rang={}):
    self.liens = lien  # lien[i] sera le representant de i
    self.rangs = rang  # rang[i] est la hauteur de i dans son arbre

## Getter / Setter

  def _set_liens(self, nv_liens):
    self.liens = nv_liens

  def _set_rangs(self, nv_rangs):
    self.rangs = nv_rangs

  def get_liens(self):
    return self.liens

  def get_rangs(self):
```

```
19      return self.rangs
20
21  ## Affichage
22
23    def __str__(self):
24      return "{}\n{}\n".format(self.liens, self.rangs)
25
26    def __repr__(self):
27      return "{}\n{}\n".format(self.liens, self.rangs)
28
29
30  ## Methodes
31
32    def partitionner(self, n):
33      """
34      In : Un entier n [int]
35      Out : None
36      Goal : Creer un nombre voulu de groupes
37      """
38      self._set_liens([i for i in range(n)])
39      self._set_rangs([0 for i in range(n)])
40
```

```
41    def trouver(self, i):
42      """
43      In : Un element i contenu dans la structure [beta]
44      Out : Le representant de cet element [beta]
45      Goal : Sert a donner le representant de l'element donne (pour savoir son groupe)
46                              de maniere optimisee
47      """
48      if self.liens[i] == i:
49        return i
50
51      else:
52        p = self.trouver(self.liens[i])
53        self.liens[i] = p
54        return p
55
56    def union(self, i, j):
57      """
58      In  : Deux elements i et j contenu dans la structure [beta]
59      Out : None
60      Goal : Effectue l'union des 2 groupes contenant les deux elements de maniere optimisee
61      """
62      rep_i = self.trouver(i)
```

```
    -
    rep_j = self.trouver(j)

    if rep_i != rep_j:  ## Le representant de plus haut rang englobe celui de rang le plus bas
      if self.rangs[rep_i] > self.rangs[rep_j]: self.liens[rep_j] = rep_i
      elif self.rangs[rep_i] < self.rangs[rep_j]: self.liens[rep_i] = rep_j
      else:
        self.liens[rep_j] = rep_i
        self.rangs[rep_i] += 1


```

Listing 6: **Implémentation du découpage des zones via l'algorithme des k-moyennes**

```
from random import *
from noeud import *


def plus_proche(s, l):
  """
  In : Un sommet s [Noeud]
       Une liste l de noeuds [list x Noeud]
  Out : Le nom d'un sommet [string]
  Goal : Determine le sommet le plus proche du sommet entre en parametre parmi ceux
                        contenu dans la liste
  """
  min = float('inf')
  res = None
  for p in l:
    d = p.distance(s)

    if d < min:
```

```
        min = d
        res = p

    return res.get_nom()


def pos_moyenne(l):
    """
    In : Une liste de sommets [Noeud]
    Out : Un tuple de flottant [float*float]
    Goal : Calcul la position moyenne des sommets contenus dans la liste
    """
    x = [s.get_x() for s in l]
    y = [s.get_y() for s in l]

    moy_x = sum(x) / len(x)
    moy_y = sum(y) / len(y)

    return (moy_x, moy_y)


def k_moyennes(g, k, iter=5):
```

```
    """
    In : Un graphe g [Graphe]
         Un entier k (correspond au nombre de drones) [integer]
         Un entier iter (nombre de fois a recalculer la position du barycentre), par defaut a 5 [Int]
    Out : Une liste de listes de noeuds [list x list x Noeud]
    Goal : Determine k zones distinctes (sous forme de liste de listes) du graphe
                      a la maniere de l'algorithme des k-moyennes
    """
    x = [s.get_x() for s in g.get_sommets()]
    y = [s.get_y() for s in g.get_sommets()]
    min_x = min(x)
    max_x = max(x)
    min_y = min(y)
    max_y = max(y)

    barycentres = []
    couleur = {}

    ## Determine la position aleatoire des k barycentres :
    for i in range(k):
      x_rd = randint(int(min_x), int(max_x))
      y_rd = randint(int(min_y), int(max_y))
```

```
63
64      barycentres.append(Noeud(i, x_rd, y_rd))
65      couleur[i] = []
66
67  ## Affecte a chaque sommet une le barycentre le plus proche
68    for s in g.get_sommets():
69      spp = plus_proche(s, barycentres)
70      couleur[spp].append(s)
71
72  ## Reapplique les etapes precedentes pour ameliorer la precision des barycentres
73    for i in range(iter):
74      for j in range(k):
75        if couleur[j] != []:
76          barycentres[j]._set_pos(pos_moyenne(couleur[j]))
77          couleur[j] = []
78
79      for s in g.get_sommets():
80        couleur[plus_proche(s, barycentres)].append(s)
81
82
83  ## Affecte a chaque barycentre les sommets les plus proches d'eux
```

```
84    res = [barycentres]
85    for j in range(k):
86      res.append([s for s in couleur[j]])
87
88    return res
89
90
```

Listing 7: **Implémentation de l'affichage de l'algorithme des k-moyennes**

```
import matplotlib.pyplot as plt
from noeud import *
from graphe import *

colors = ["red", "green", "blue", "purple", "pink", "orange"]
img = plt.imread("coupé.png")


def aff_k_moy(g, k_moy, carte=False):
  """
  In : Un graphe g [Graphe]
       Une liste de listes de sommets [list x list x Noeud]
       Un booleen carte, par defaut a False [bool]
  Out : None
  Goal : Affiche le decoupage de la zone a couvrir selon l'algorithme des k-moyennes
  """
  fig, ax = plt.subplots()

  ## Affiche la carte de France contenant les villes choisies en fond de tout
```

```
19    ## Affiche la carte de France contenant les villes choisies en jeu de test
20    if carte:
21      ax.imshow(img, extent=[560, 8850, 690, 9450])
22
23  ## Affichage des barycentres
24    for j in range(len(k_moy[0])):
25      s_j = k_moy[0][j]
26      x_j = s_j.get_x()
27      y_j = s_j.get_y()
28      ax.plot(x_j, y_j, marker="x", color=colors[j])
29
30
31  ## Affichage des villes, coloriees selon le barycentre qui leur est associe
32    for i in range(1, len(k_moy)):
33      for j in range(len(k_moy[i])):
34        s_j = k_moy[i][j]
35        x_j = s_j.get_x()
36        y_j = s_j.get_y()
37        ax.plot(x_j, y_j, marker="o", color=colors[i - 1])
38
39    plt.show()
40
```

Listing 8: **Implémentation de l'algorithme glouton du plus proche voisin**

```
1  from graphe import *
2  from noeud import *
3  from drone import *
4  from math import floor, sqrt
5  from copy import deepcopy
6  from data import *
7
8
9  def glouton(g, som_dep, num_drone):
10   """
11     In : Un graphe g operationnel [Graphe]
12          Un sommet de depart som_dep [Noeud]
13     Out : Une liste de sommets [list x Noeud]
14     Goal : Etablie la liste (par methode gloutonne : plus-proche-voisin) des sommets
15            les plus proches pour effectuer le parcours le plus petit possible
16     """
17   n = g.get_taille()
18
```

```
19    a_parcourir = list(g.get_sommets())
20
21    if som_dep in a_parcourir:
22      a_parcourir.remove(som_dep)
23
24    liste_parcours = [som_dep]
25
26    for i in range(n - 1):
27      sommet_courant = liste_parcours[i]
28      m = n - 1 - i
29      nmin = 0
30
31  # distance du sommet courant à un sommet arbitrairement choisi
32      dmin = g.get_arcs()[a_parcourir[nmin].get_nom()][sommet_courant.get_nom()]
33
34      for j in range(1, m):
35
36      # distance de tous les autres sommets au sommet courant
37        d = g.get_arcs()[a_parcourir[j].get_nom()][sommet_courant.get_nom()]
38
39        if d < dmin:  # mise a jour sommet plus proche du courant
40          dmin = d
```

```
41          nmin = j
42
43      liste_parcours.append(a_parcourir.pop(nmin))
44
45    liste_parcours.append(som_dep)
46
47    return to_str(liste_parcours)
48
49
50  def multi_glouton(g, nb_drones):
51    """
52      In : Un graphe g operationnel [Graphe]
53           Un nombre de drones nb_drones [integer]
54      Out : Une liste de listes de noeuds [list x list x Noeud]
55      Goal : Ajoute, a tour de role le sommet le plus proche de chaque drone a sa liste de parcour
56      """
57    n = g.get_taille()
58    liste_parcours = [[g.get_sommets()[0]] for i in range(nb_drones)]
59    a_parcourir = g.get_sommets()[1:]
60
61    for i in range(floor((n - 1) / nb_drones)):
62      for x in range(nb_drones):
```

```
63        sommet_courant = liste_parcours[x][-1]
64        nmin = 0
65        dmin = g.get_arcs()[a_parcourir[nmin].get_nom()][sommet_courant.get_nom(
66        )]  # distance du sommet courant à un sommet arbitrairement choisi
67        m = len(a_parcourir)
68
69      for j in range(1, m):
70        d = g.get_arcs()[a_parcourir[j].get_nom()][sommet_courant.get_nom(
71        )]  # distance de tous les autres sommets au sommet courant
72
73        if d < dmin:  # mise a jour sommet plus proche du courant
74          dmin = d
75          nmin = j
76
77    liste_parcours[x].append(a_parcourir.pop(nmin))
78
79    for i in range(floor((n - 1) / nb_drones) * nb_drones, n - 1):
80      x = i - floor((n - 1) / nb_drones) * nb_drones
81      sommet_courant = liste_parcours[x][-1]
82      nmin = 0
83      dmin = g.get_arcs()[a_parcourir[nmin].get_nom()][sommet_courant.get_nom(
```

```
84        )]  # distance du sommet courant à un sommet arbitrairement choisi
85        m = len(a_parcourir)
86
87      for j in range(1, m):
88        d = g.get_arcs()[a_parcourir[j].get_nom()][sommet_courant.get_nom(
89        )]  # distance de tous les autres sommets au sommet courant
90
91        if d < dmin:  # mise a jour sommet plus proche du courant
92          dmin = d
93          nmin = j
94
95      liste_parcours[x].append(a_parcourir.pop(nmin))
96
97      return liste_parcours
98
99
100   def print_parcours(l):
101     """
102       In : Une liste l de noeuds [list x Noeud]
103       Out : None
104       Goal : Affiche le parcours cree par l'algorithme glouton
105       """
```

```
106   for i in range(len(l)):
107     print("{}\n".format(l[i].get_nom()))
108
109
110 def print_multi_parcours(l):
111   """
112     In : Une liste l de noeuds [list x Noeud]
113     Out : None
114     Goal : Affiche a la console le parcours effectue par chaque drone
115     """
116   for i in range(len(l)):
117     print("Parcours du drone n°", i)
118     for j in range(len(l[i])):
119       print("  {} \n".format(l[i][j].get_nom()))
120
121
122 def creer_chemin(l, n):
123   """
124     In : Une liste l de noeuds [list x Noeud]
125          Un entier n [integer]
126     Out : Une liste de listes de triplets [list X list x string*float*float]
```

```
    Goal : Etablit les sommets les plus proches repartis en n sous-zones distinctes
    """
  liste_x = [s.get_x() for s in l]
  liste_y = [s.get_y() for s in l]
  bg = (min(liste_x), min(liste_y))  #coin bas gauche de la zone à quadriller
  hd = (max(liste_x), max(liste_y))  #coin haut droite de la zone à quadriller

  step_x = (hd[0] - bg[0]) / sqrt(n)
  step_y = (hd[1] - bg[1]) / sqrt(n)

  grid = [[] for i in range(n)]

  for i in range(len(l)):
    x = 0
    y = 0

    while (bg[0] + (x + 1) * step_x < l[i].get_x()):
      x += 1

    while (bg[1] + (y + 1) * step_y < l[i].get_y()):
      y += 1

```

```
149       grid[x * int(sqrt(n)) + y].append(
150         (l[i].get_nom(), l[i].get_x(), l[i].get_y()))
151
152     return grid
153
154
155   def multi_chemin(g, n):
156     """
157       In : Un graphe g [Graphe]
158            Un entier n [integer]
159       Out : Une liste de listes de noeuds [list X list x Noeud]
160       Goal : Applique l'algorithme glouton aux differents sommets de chaque sous-zones,
161                       etablissant le parcours que chaque drone prendra
162       """
163     grid = creer_chemin(g.get_sommets(), n)
164
165     listes_parcours = [glouton(creer_graphe(l)) for l in grid]
166
167     return listes_parcours
168
169   ---------------------------------------------------------------------------------------
```

Listing 9: **Implémentation de l'algorithme du Branch and Bound**

```
1  from random import *
2  from math import floor, sqrt
3  from copy import deepcopy
4
5  from graphe import *
6  from noeud import *
7  from drone import *
8  from decoupage_zones import *
9  from data import *
10 from tasmin import *
11 from unionfind import *
12 from affichage import *
13 from main_glouton import *
14 from analyse_resultat import *
15
16
17 def const_tasmin_aretes_sommet(g, som):
18     """
```

```
    In : Un graphe g [Graphe]
         Un sommet som du graphe [string]
    Out : Une file de priorite min [TasMin]
    Goal : Construit la file de priorite min contenant toutes les aretes incidentes au sommet
                                    entre en parametre
    """
    t = TasMin()
    for s in g.get_arcs()[som].keys():
      t.push((som, s), g.get_arcs()[som][s])

    return t


def cmp_aretes(a1, a2):
    """
    In : 2 aretes [Noeud*Noeud]
    Out : True si les aretes sont les memes, False sinon [Bool]
    Goal : Regarde si les 2 aretes donnees en parametres sont les memes (dans un graphe non oriente)
    """
    x1, y1 = a1
    x2, y2 = a2

```

```
41    return ((x1 == x2 and y2 == y1) or (x1 == y2 and y1 == x2))
42
43
44  def approx_cout_aux(g, n, sommets_requis=[]):
45    """
46    In : Un graphe g [Graphe]
47         Un entier n correspondant au nombre de sommets à parcourir au départ [int]
48         Une liste de noms de sommets, par defaut vide [list x string]
49    Out : Un flottant [float]
50    Goal : Determine le cout d'acces maximal de chaque sommet, en differenciant ceux requis
51    et les autres :
52    -> somme des deux aretes de plus bas cout de chaque sommet non requis + cout des aretes reliant
53                          les sommets requis le tout divise par 2
54    """
55    c = 2 * poids_chemin(g, sommets_requis)
56    nb_som = len(sommets_requis)
57
58    ## Si le drone ne doit parcourir qu'un sommet, le résultat est immediat
59    if n == 1:
60      return c
61
62    l_utilises = []
```

```
63    for i in range(nb_som - 1):
64      l_utilises.append((sommets_requis[i], sommets_requis[i + 1]))
65
66    ## Si tous les sommets du graphe sont deja "requis", il ne reste plus qu'a ajouter le cout
67    ##       de l'arete permettant de fermer le tour au poids total du chemin :
68    if nb_som == n:
69      return (c + g.get_arcs()[sommets_requis[0]][sommets_requis[-1]]) / 2
70
71    premier_tasmin = const_tasmin_aretes_sommet(g, sommets_requis[0])
72    a = premier_tasmin.pop()  # Arete de poids minimal du premier sommet requis
73
74    if nb_som == 1:
75      b = premier_tasmin.pop(
76      )  # Arete de poids minimal du dernier sommet requis (le même que le premier)
77
78    else:
79      dernier_tasmin = const_tasmin_aretes_sommet(g, sommets_requis[-1])
80      b = dernier_tasmin.pop()  # Arete de poids minimal du dernier sommet requis
81
82      ## Verifie que l'arete de poids minimal du premier sommet requis n'est pas deja empruntee
83      ##  pour aller au second sommet requis, sinon tire la seconde arete de plus petit cout :
```

```
84      if cmp_aretes(a, (sommets_requis[0], sommets_requis[1])):
85        a = premier_tasmin.pop()
86
87      ## Verifie que l'arete de poids minimal du dernier sommet requis n'est pas deja emprunte
88      ##      pour fermer le tour, sinon tire la seconde arete de plus petit cout :
89      if cmp_aretes(b, (sommets_requis[-1], sommets_requis[-2])):
90        b = dernier_tasmin.pop()
91
92      c += g.get_arcs()[b[0]][b[1]]
93
94    l_utilises.append(a)
95    l_utilises.append(b)
96
97    c += g.get_arcs()[a[0]][a[1]]
98    som_restants = list(
99      g.get_arcs().keys())  # Liste des sommets du graphe non requis
100
101   for s in sommets_requis:
102     som_restants.remove(s)
103
104
105 ## Derniere etape : calcul le cout d'acces aux sommets non requis
```

```
##             (tire leurs 2 aretes de plus bas cout) :
  for s in som_restants:
    t = const_tasmin_aretes_sommet(g, s)
    x1, y1 = t.pop()
    x2, y2 = t.pop()
    l_utilises.append((x1, y1))
    l_utilises.append((x2, y2))
    c += g.get_arcs()[x1][y1] + g.get_arcs()[x2][y2]

  return c / 2


def approx_cout(g, som_dep, restants, n, l=[]):
  """
  In : Un graphe g [Graphe]
       Le nom du sommet de depart du tour som_dep (doit être dans restant) [string]
       Une liste restants des noms des sommets non encore requis [list x string]
       Un entier n correspondant au nombre de sommets à parcourir au départ [int]
       Une liste l des noms des sommets requis, par defaut vide, sert d'accumulateur [list x string]
  Out : Une liste de noms de sommets [list x string]
  Goal : Calcule le poids maximal de maniere optimale pour un tour dans le graphe donne
```

```
127             (sert de borne superieure a l'algorithme de Branch and Bound)
128     """
129     ## Condition de depart :
130     if l == []:
131       assert som_dep in restants, "Le sommet de départ doit faire partie de la liste restants"
132       l = [som_dep]
133       restants.remove(som_dep)
134
135
136   ## Condition de fin :
137     if len(l) == n:
138       return l + [l[0]]
139
140     m = float('inf')  # Va servir de borne sup
141     res = ""  # Sert a stocker le meilleur sommet a prendre pour minimiser
142     #                       le cout total
143
144     ## Calcul le cout du chemin requis a tous les sommets encore disponibles
145     ##               puis ajoute le sommet minimisant ce cout :
146     for s in restants:
147       c = approx_cout_aux(g, n, l + [s])
148
```

```
149       if c < m:
150         m = c
151         res = s
152 
153     restants.remove(res)
154     l.append(res)
155 
156     return approx_cout(g, som_dep, restants, n, l)
157 
158 
159 def calcul_borne_inf(g, som_dep, som_req=[], som_lib=[]):
160     """
161     In : Un graphe g [Graphe]
162          Le nom du sommet de depart du tour som_dep [string]
163          Une liste de noms de sommets som_req [list x string]
164          Une liste de noms de sommets som_lib [list x string]
165     Out : Un flottant [float]
166     Goal : Determine le cout d'accès minimal de chaque sommet, en differenciant ceux requis
167                                 et les autres
168     """
169     if som_req == []:
```

```
170       som_req = [som_dep]
171
172     if som_lib == []:
173       som_lib = to_str(g.get_sommets())
174
175       if som_dep in som_lib:
176         som_lib.remove(som_dep)
177
178     cout = 0
179     ## Applique l'algorithme de Kruskal au graphe induit par les sommets libres
180     g_induit_lib = g.graphe_induit(som_lib)
181     acm_lib = kruskal(g_induit_lib)
182
183     ## Somme les poids des aretes dans l'ACM
184     for a in acm_lib:
185       x, y = a[0], a[1]
186       cout += g.get_arcs()[x][y]
187
188
189   ## Rajoute les poids des aretes deja requises
190     cout += poids_chemin(g, som_req)
191
```

```
192   return cout
193
194
195 def branch_and_bound_aux(g, som_dep, som_lib, num_drone, som_req=[]):
196   """
197   In : Un graphe g [Graphe]
198        Un sommet som_dep [Noeud]
199        Une liste de noms de sommet som_lib [list x string]
200        Un entier num_drone [int]
201   Out : None
202   Goal : Applique le principe de Branch and Bound au graphe donne pour y trouver le
203             tour de cout minimal (met a jour les accumulateurs et var. globales par effet de bord)
204   """
205   global bornes_sup
206   global parcours_bb
207
208   n = g.get_taille()
209
210 ## Cas de fin :
211   if len(som_req) == g.get_taille():
212     poids = poids_chemin(g, som_req + [som_dep.get_nom()])
```

```
213
214     if poids < bornes_sup[num_drone]:
215       bornes_sup[num_drone] = poids
216       parcours_bb[num_drone] = som_req + [som_dep.get_nom()]
217
218 ## Corps principal :
219   else:
220
221     file_prio = TasMin()
222
223     for s in som_lib:
224       cout = approx_cout_aux(g, n, som_req + [s])
225
226   ## Ne considere que les parcours pouvant faire diminuer la borne superieure
227       if cout < bornes_sup[num_drone]:
228         file_prio.push(s, cout)
229
230     parcours_possibles = TasMin()
231
232   ## Considere tous les noeuds libres par ordre croissant de leur score
233     while not file_prio.est_vide():
234       s = file_prio.pop()
```

```
                -     - -
235
236         new_som_lib = list(som_lib)
237         new_som_lib.remove(s)
238
239         branch_and_bound_aux(g,som_dep,new_som_lib,num_drone,som_req+[s])
240
241 def branch_and_bound(g, som_dep, num_drone):
242     """
243     In : Un graphe g [Graphe]
244          Un sommet som_dep [Noeud]
245          Un entier num_drone [int]
246     Out : Une liste de sommet [list x string]
247     Goal : Calcule le parcours optimal pour le drone numéro num_drone
248     """
249     global bornes_sup
250     global parcours_bb
251
252     bornes_sup[num_drone]=float('inf')
253
254     a_parcourir = list(g.get_sommets())
255     a_parcourir.remove(som_dep)
```

```
256     a_parcourir = to_str(a_parcourir)
257
258     branch_and_bound_aux(g,som_dep,a_parcourir,num_drone,[som_dep.get_nom()])
259
260     return parcours_bb[num_drone]
261
262
263 g_16 = creer_graphe(alentours_Janson)
264 g_fr = creer_graphe(villes_france)
265
266 bornes_sup = [float('inf')]*100
267 parcours_bb = [[]]*100
268
269 ## TEST TOTAL
270 scores = calcul_score_moyen(g_fr, branch_and_bound, g_fr.noms_som["Paris"], 10)
271
272 aff_efficacité(scores)
273
```

Listing 10: **Implementation de l'analyse des resultats**

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3
4  from noeud import *
5  from graphe import *
6  from decoupage˙zones import *
7
8
9  def const_jeu_test(g, n=10):
10   """
11     In : Un graphe g [Graphe]
12          Un entier n [int]
13     Out : Une liste de listes de listes de listes de noms de sommet
14               [list x list x list x list x string]
15     Goal : Pour k allant de 1 au nombre de sommets du graphe, crée n découpages
16               de la zone a couvrir par la methode des k-moyennes du graphe
17
18       Construction du jeu de test (les imbrications) :
```

```
19        -> jeu[i] : jeu de test pour i+1 drones ;
20        -> jeu[i][j] : test n°j pour i+1 drones ;
21        -> jeu[i][j][k] : sommets a parcourir par le k-ieme drone du j-ieme test pour i+1 drones
22    """
23    taille = g.get_taille()
24    jeu = [[] for i in range(taille)]
25
26    for k in range(1, taille + 1):
27      for i in range(n):
28        jeu[k - 1].append(k_moyennes(
29          g, k)[1:])  # k_moyennes()[0] donne les barycentres inutiles ici
30
31    return jeu
32
33
34  def aff_jeu_test(jeu):
35    """
36      In : Une liste de listes de listes de listes de noms de sommet
37                [list x list x list x list x Noeud]
38      Out : None
39      Goal : Affiche le jeu de test créé par la fonction const_jeu_test
40    """
```

```
41    for i in range(len(jeu)):
42      print("\nTest pour {} drones :\n".format(i + 1))
43
44      for j in range(len(jeu[i])):
45        print("\tTest n° {} :\n".format(j + 1))
46
47        for k in range(i + 1):
48          print("\tParcours du drone n° {} : {}\n".format(
49            k + 1, to_str(jeu[i][j][k])))
50
51
52  def calcul_score_moyen(g, fonction_res, som_dep, n=10):
53    """
54    In : Un graphe g [Graphe]
55         Une fonction fonction_res [fun]
56         Un noeud som_dep [Noeud]
57         Un entier n [int]
58    Out : Une liste de flottants [list x float]
59    Goal : Determine le score moyen (sur n simulations) de tours determines par la methode
60    implentee avec la fonction_res pour tous nombre de drones possible ([|1 ; nb_sommets-1|])
61    """
62
```

```
63    a_parcourir = g.get_sommets()
64    a_parcourir.remove(som_dep)
65    g_a_parcourir = g.graphe_induit(to_str(a_parcourir))
66
67    taille = g_a_parcourir.get_taille()
68
69    jeu_test = const_jeu_test(g_a_parcourir, n)
70
71    scores_moyens = []
72
73    ## Le but est de faire varier le nombre de drone de 1 au nombre de sommets
74    for i in range(taille):
75
76      scores_i_drones = []
77
78      for test_j in range(len(jeu_test[i])):
79
80        poids_test_j = []
81
82        ## Permet d'obtenir le trajet du k-ieme drone
83        for drone_k in range(len(jeu_test[i][test_j])):
```

```
84
85          ## Ne s'interesse qu'aux drones ayant au moins 1 sommet attribue
86          if len(jeu_test[i][test_j][drone_k]) > 0:
87            g_induit = g.graphe_induit(
88              to_str(jeu_test[i][test_j][drone_k]) + [som_dep.get_nom()])
89
90            acm_induit = kruskal(g_induit)
91            poids_acm = 0
92
93            ## Determine le cout total de l'ACM, servant de cout ideal pour le tour
94            for a in acm_induit:
95              x, y = a[0], a[1]
96              poids_acm += g.get_arcs()[x][y]
97          ## Determine le cout du tour trouve pour le k-ieme drone
98            parcours_drone_k = fonction_res(g_induit, som_dep, drone_k)
99            poids_test_j.append((poids_chemin(g, parcours_drone_k), poids_acm))
100             # le 2e element servira a effectuer des calculs plus tard
101
102       max_poids = 0
103
104       ## Determine le drone ayant le parcours le plus long
105       ## Etant le dernier a terminer son tour, sert de representant pour les autres drones du meme te
```

```
106       pire_parcours = None
107       for a in poids_test_j:
108         if a[0] > max_poids:
109           max_poids = a[0]
110           pire_parcours = a
111
112     ## Le score d'un test est determiner tel que :
113     ## s(test) = (valeur_atteinte - valeur_ideale) * (1 + cout(test)) > 0
114     ## Ici le cout est le nombre de drones utilise.
115     ## Le but est de reduire son impact pour ameliorer le visuel des graphiques
116       scores_i_drones.append(
117         (pire_parcours[0] - pire_parcours[1]) * (1 + (i + 1)**0.5))
118
119   ## Stocke toutes les moyennes des tests faits pour chaque nombre de drones
120   ## (n = simulations faites pour chaque nombre de drones)
121     scores_moyens.append(sum(scores_i_drones) / n)
122
123   return scores_moyens
124
125
126 def aff_efficacité(scores):
```

```
127     """
128     In : Une liste de flottants [list x float]
129     Out : None
130     Goal : Affiche le score moyen de la methode de resolution du probleme
131                       selon le nombre de drones utilises
132     """
133     fig, ax = plt.subplots()
134
135     nb_drones = range(1, len(scores) + 1)
136
137     ax.bar(nb_drones, scores)
138
139     ax.set_ylabel('Coût')
140     ax.set_xlabel('Nombre de drones')
141
142     plt.show()
143
```

Listing 11: **Jeux de coordonnées de test**

```
ville_france = [
  ("Paris", 7482.1, 5121), ("Amiens", 8594.9, 4928.7), ("Auxerre", 6194.5, 6090.6),
  ("Angouleme", 4432.2, 3453.9), #("Ajaccio", 543, 9486), ("Bastia", 1467, 9778),
  ("Brest", 7013.6, 781.5), ("Bourges", 5445.3, 5094.7), ("Bordeaux", 3274, 3045.1),
  ("Bayonne", 1310, 2373.4), ("Cherbourg", 8292.9, 2803.1),
  ("Clermnond-Ferrand", 3924.7, 5724), ("Dijon", 5462.6, 7044.8),
  ("Grenoble", 3151.7, 7576), ("Le Havre", 8149.5, 3844.4), ("Lyon", 3846.9, 6988.5),
  ("Lille", 9396.9, 5646), ("Limoges", 4272.2, 4466), ("Le Mans", 6400, 3715),
  ("Metz", 7831.4, 7627.8), ("Mulhouse", 6089.9, 8524.8), ("Marseille", 1166.4, 7432.3),
  ("Montpelier", 1379.1, 6039.2), ("Nancy", 7394.1, 7584.5), ("Nice", 1510.7, 8531.5),
  ("Nantes", 5449.5, 2602), ("Orleans", 6357.9, 4682.8), ("Pau", 865.7, 3151.2),
  ("Perigueux", 3114, 4057), ("Poitiers", 4900.3, 3834.4), ("Reims", 7876.9, 6220.1),
  ("Rennes", 6265, 2439.1), ("Rouen", 8111.7, 4328.3), ("Strasbourg", 7535.7, 8757.3),
  ("Saint-Etienne", 3463.9, 6490.5), ("Troyes", 6780.9, 6427), ("Toulouse", 1157.1, 4499.8),
  ("Tours", 5691.1, 3891.9)
]
```

```
19 alentours_Janson = [
20   ("Franprix Troca", 600, 0), ("Le Petit L'Or", 75, 300),
21   ("Carrefour Lamartine", 0, 550), ("Lycée", 225, 500),
22   ("McDo", 375, 800), ("Frog XVI", 900, 425),
23   ("Monop Troca", 900, 375), ("Franprix Victor Hugo", 700, 950),
24   ("Monop Belles feuilles", 425, 675)]
25
```