Optimisation des trajets de livraison

BERTINCHAMP Lucas - 29753

Session 2021

Réseaux routiers

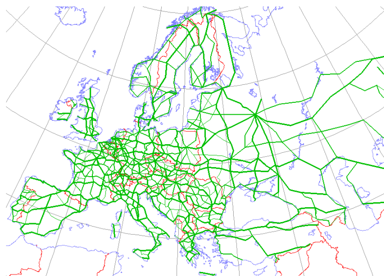


Figure – Europe

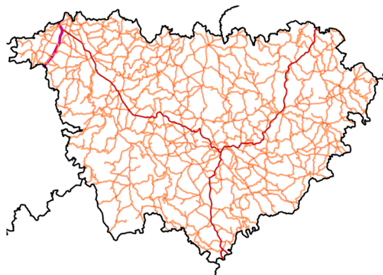


Figure – Haute-Loire

Problème du voyageur de commerce

Caractéristiques du problème :

- Déterminer le plus court chemin
- Passer au maximum une fois par point
- Problème NP-complet

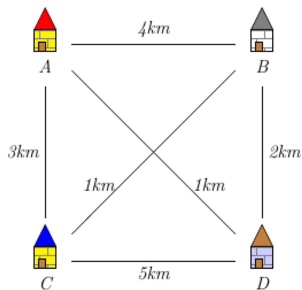


Figure – Exemple de problème

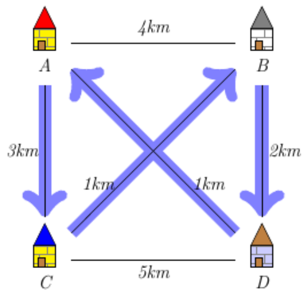


Figure – Solution du problème

Sommaire

1. Modélisation
   - Exemple et définitions
   - Modélisation informatique
2. Recherche naïve du plus court chemin
   - Développement de l'algorithme
   - Résultats
3. Algorithmes d'approche du plus court chemin
   - Difficultés et simplification
   - Arbre couvrant minimal
   - Plus proche voisins amélioré
4. Comparaisons des méthodes

Exemple de représentation

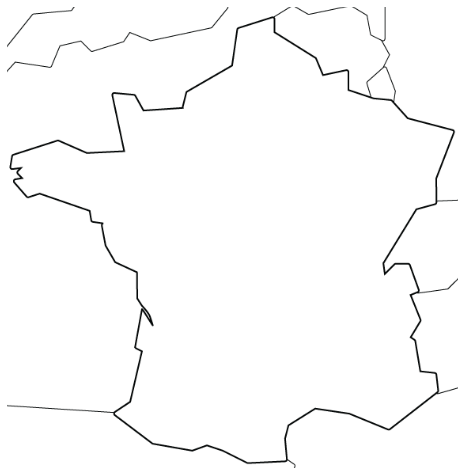


Figure – Sur la France

Exemple de représentation

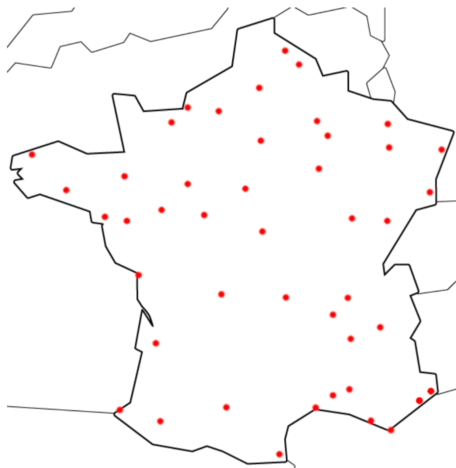


Figure – Sur la France

Exemple de représentation

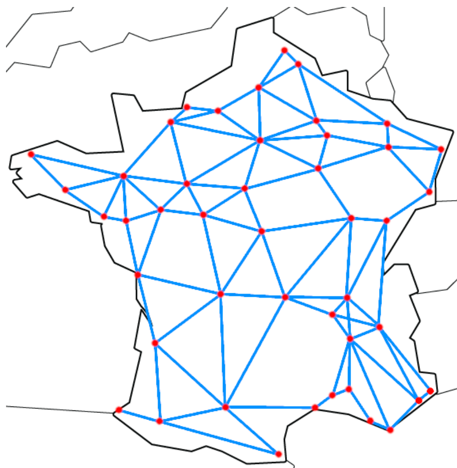


Figure – Sur la France

Exemple de représentation

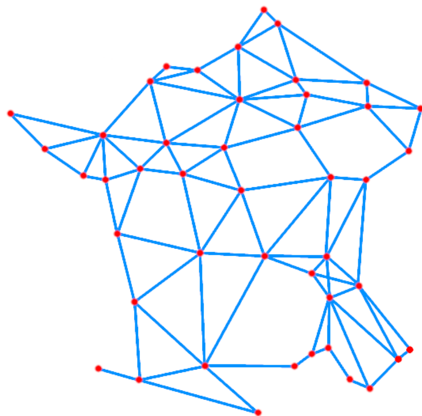


Figure – Graphe obtenu

Quelques définitions

- Voisin : Deux sommets sont voisins s'ils sont reliés par une arête
- Chemin : Suite d'arêtes passant par tous les points d'un graphe

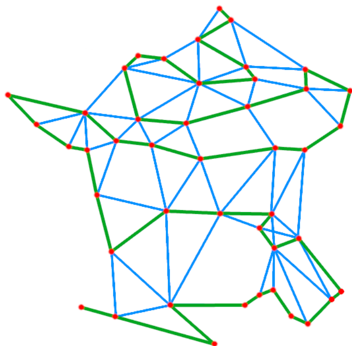


Figure – Graphe exemple

En informatique

Différents objets :

- 2 variables : nb_villes / nb_voisins
- type graphe (fonction generation_graphe())
- chemin

```
([0, 1, 6, 2, 3, 4, 9, 8, 7, 5], 2243.170474884116)
```

Figure – Un chemin

```
[[  0.          83.86894539 340.58772732 647.74223268   0.
  503.01689832 225.8583627  521.64739049 618.86428238   0.        ]
 [ 83.86894539   0.         258.60007734   0.           0.
  574.42144807 209.11719202 544.69257384   0.           0.        ]
 [340.58772732 258.60007734   0.         430.6855001  475.01578921
  782.85758603 289.36482164   0.         446.4000448    0.        ]
 [647.74223268   0.         430.6855001    0.          94.9210198
    0.         441.78048848 538.25087088  84.50443775 137.80058055]
 [  0.           0.         475.01578921  94.9210198    0.
    0.           0.           0.         174.25555945 173.41568557]
 [503.01689832 574.42144807 782.85758603   0.           0.
    0.         507.29281485 387.34222594   0.           0.        ]
 [225.8583627  209.11719202 289.36482164 441.78048848   0.
  507.29281485   0.         356.70716281 401.40378673   0.        ]
 [521.64739049 544.69257384   0.         538.25087088   0.
  387.34222594 356.70716281   0.         457.28000175 515.3882032 ]
 [618.86428238   0.         446.4000448   84.50443775 174.25555945
    0.         401.40378673 457.28000175   0.         117.17508268]
 [  0.           0.           0.         137.80058055 173.41568557
    0.           0.         515.3882032  117.17508268   0.        ]]
```

Figure – Une matrice ...

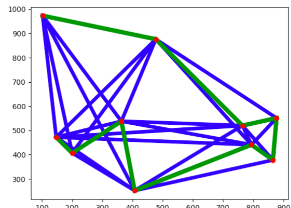


Figure – ... et son graphe associé

Principe de l'algorithme

On note $v_i = \{voisins\ du\ sommet\ i\ non\ visit\acute{e}\}$ et
$C_i = \{chemins\ comman\c{c}ant\ par\ i\}$

$$C_i = \bigcup_{n=1}^{Card(v_i)} C_{(i, v_i(n))} \text{ ; fonction chemins()}$$

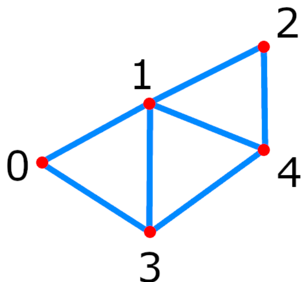


Figure – Graphe

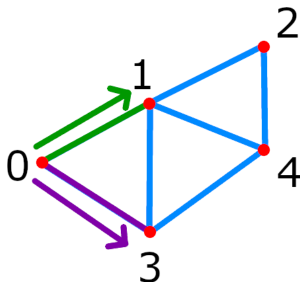


Figure – Découpage

Principe de l'algorithme

On note $v_i = \{voisins\ du\ sommet\ i\ non\ visit\acute{e}\}$ et
$C_i = \{chemins\ comman\c{c}ant\ par\ i\}$

$$C_i = \bigcup_{n=1}^{Card(v_i)} C_{(i, v_i(n))} \text{ ; fonction chemins()}$$

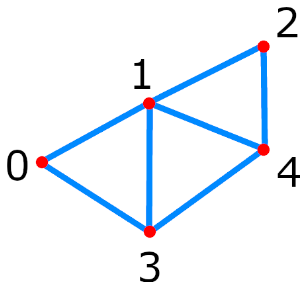


Figure – Graphe

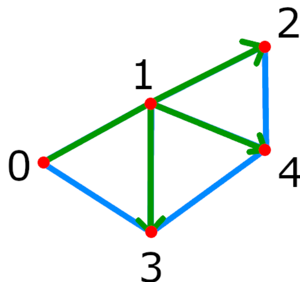


Figure – Découpage

Problème récursivité

- "RecursionError : maximum recursion depth exceeded"
- Code : liste, Code[i] = voisin de i dans le chemin

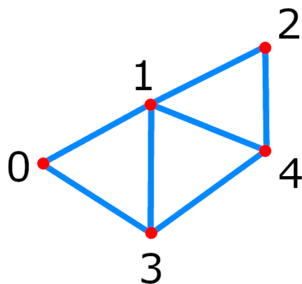


Figure – Graphe

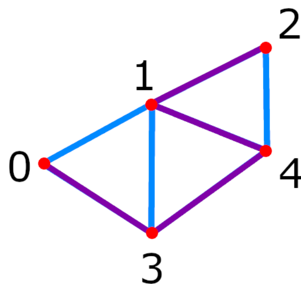
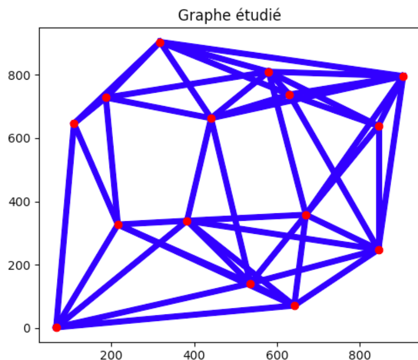


Figure – Chemin codé

code = [1, 3, 0, 2, 0]

Problème récursivité



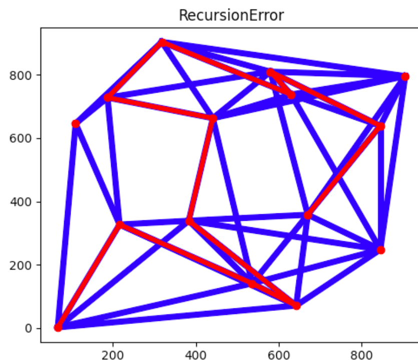


Figure – Exemple de RecursionError

code = [1, 2, 3, 5, 0, 0, 6, 3, 1, 2, 4, 5, 3, 5, 0]

Sur un exemple

nb_villes = 13 / nb_voisins = 6 -> 12630 chemins
Chemin minimal : 3325 / Chemin maximal : 6095
Chemin moyen : 4822

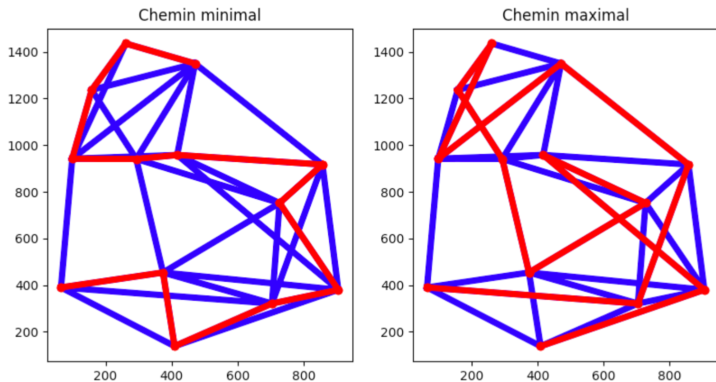


Figure – Graphe et chemins

Exploitation des données

- 16000 graphes aléatoires
- Base SQL

	10 villes / 6 voisins	11 villes / 7 voisins
Nombre de graphes	500	500
Nombre de chemins moyens	1730	11000
Nombre maximal de chemin	9610	85000
Temps d'execution (500)	8 minutes	3 heures 30
Temps d'exe. moyen (1)	1 seconde	25 secondes

Compléxité : O(nb_villes !)

Algorithmes classiques non performants

- Plus proche voisin
- Algorithme génétique
- Permutation locale de sommets

On autorise désormais les chemins passant plusieurs fois par le même sommet

Détermination de l'arbre couvrant minimal

Définitions :

- Arbre couvrant : Graphe n'admettant pas de cycle et passant par tous les sommets
- Arbre couvrant minimal : Arbre dont la somme des longueurs de ses aretes est minimal

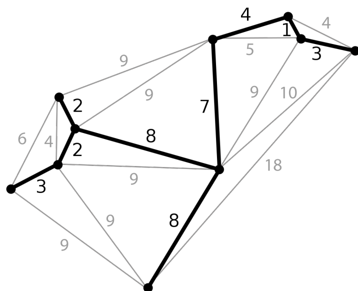


Figure – Arbre couvrant minimal

File de priorité

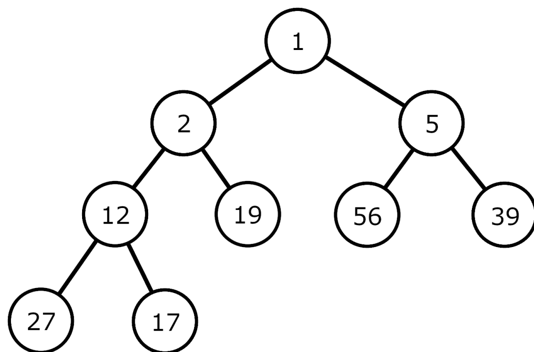


Figure – Représentation par tas

File de priorité

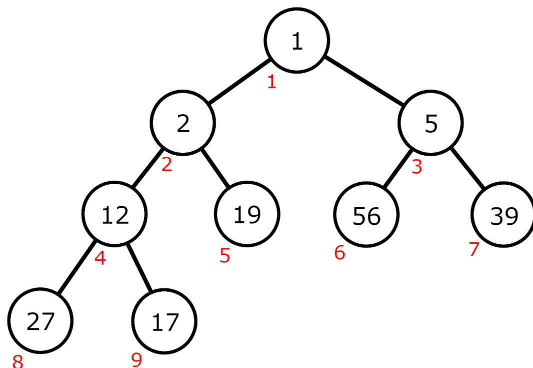


Figure – Représentation par tas

Liste obtenue : [1, 2, 5, 12, 19, 56, 39, 27, 17]

File de priorité

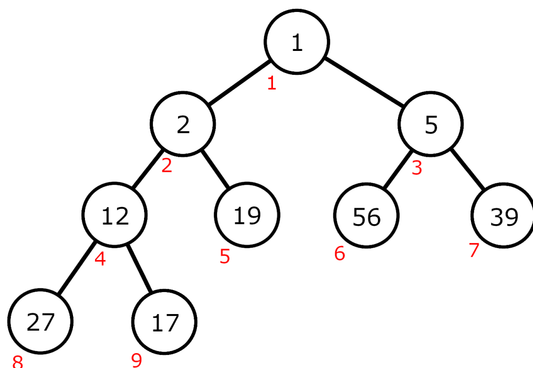


Figure – Représentation par tas

Fonctions implémentées : cree_file() / est_file() / ajoute_file() / premier_file() / reste_file() / change_priorite()

Algorithme de Prim

Objectif : construire un arbre de proche en proche : prim()

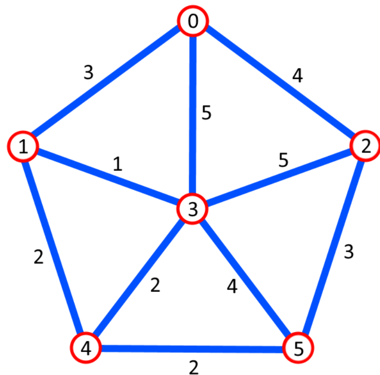


Figure – Graphe

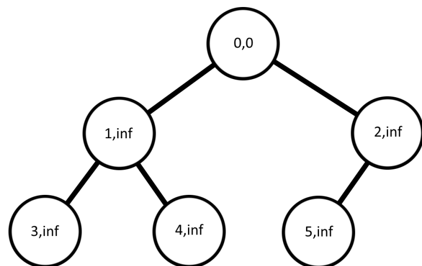


Figure – File de priorité

Algorithme de Prim

Objectif : construire un arbre de proche en proche : prim()

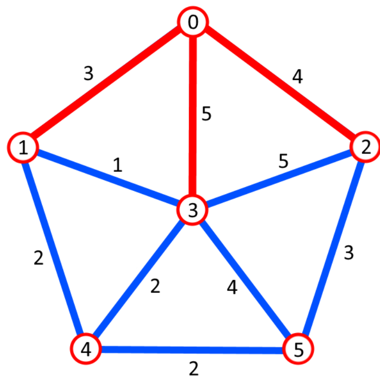


Figure – Graphe

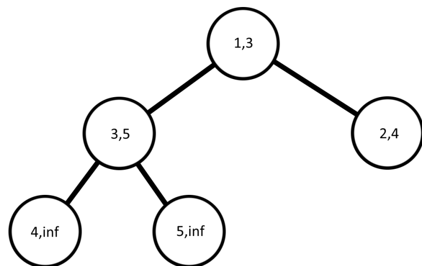


Figure – File de priorité

Algorithme de Prim

Objectif : construire un arbre de proche en proche : prim()

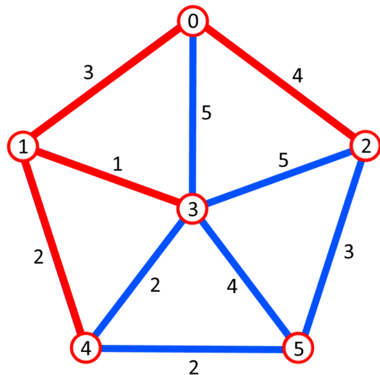


Figure – Graphe

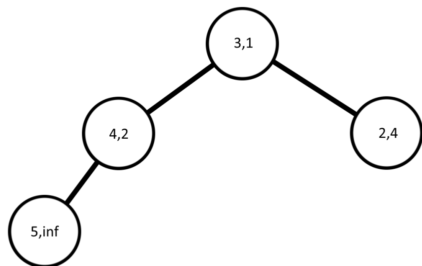


Figure – File de priorité

Algorithme de Prim

Objectif : construire un arbre de proche en proche : prim()

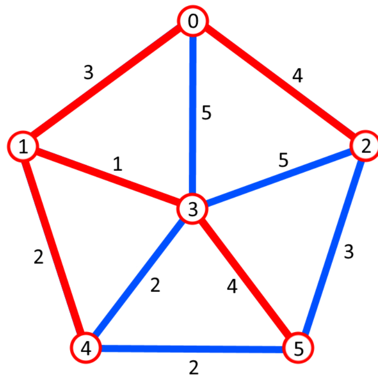


Figure – Graphe

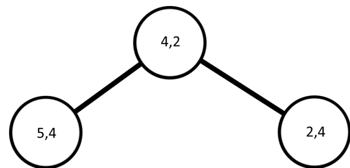


Figure – File de priorité

Algorithme de Prim

Objectif : construire un arbre de proche en proche : prim()

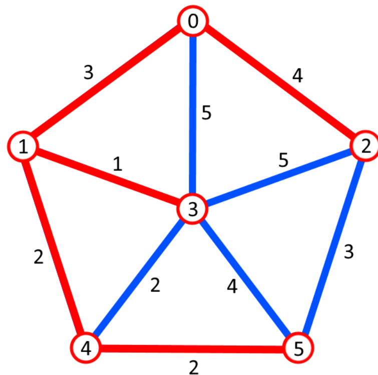


Figure – Graphe

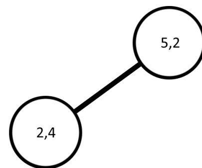


Figure – File de priorité

Algorithme de Prim

Objectif : construire un arbre de proche en proche : prim()

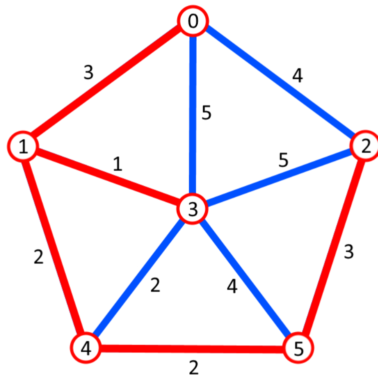


Figure – Graphe

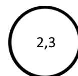


Figure – File de priorité

Application de l'algorithme

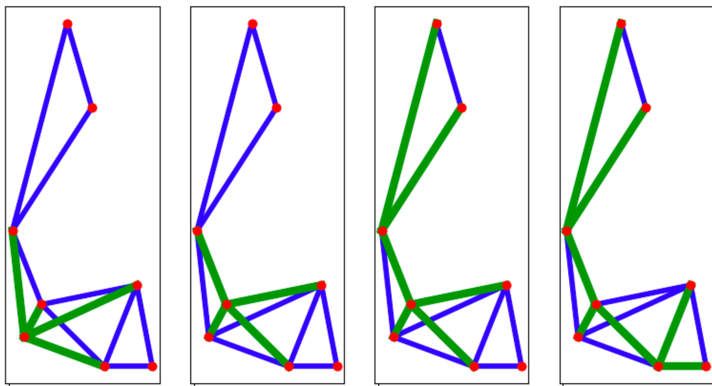


Figure – Application : itérations 1 à 4

Application de l'algorithme

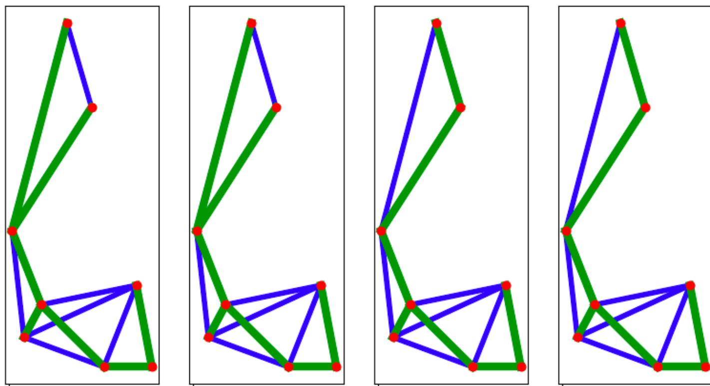


Figure – Application : itérations 5 à 8

Détermination du chemin associé

Quelques règles :

- Départ du point 0
- Si 1 voisin possible : on le prend
- Si 2 voisins possibles ou plus : aller-retours sur chemin courts

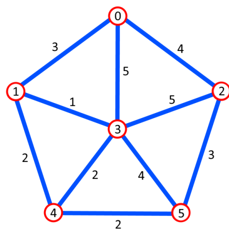


Figure – Graphe

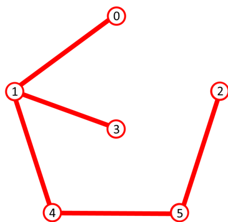


Figure – Arbre couvrant associé

Chemin obtenu : [0,1,3,1,4,5,2]

Résultats

Pourcentage d'erreur : $e = \frac{|l - l_{min}|}{l_{min}} * 100$

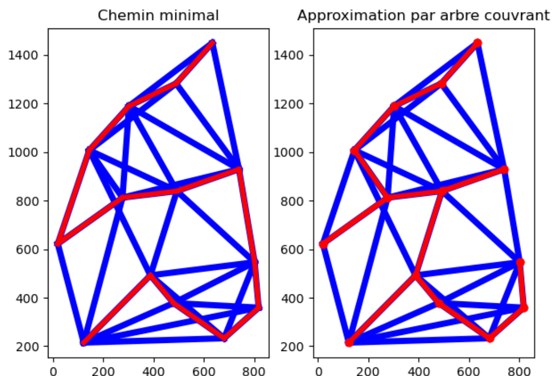


Figure – Exemple : 34% d'erreur

Pour nb_voisins = 6 :

nb_villes	Erreur moyenne
7	20
8	20.2
9	21.1
10	22.9
11	25.6
12	27.2
13	28.9

Plus proches voisins amélioré

Objectif : former un ensemble de longs chemins à relier : plus_proche2()

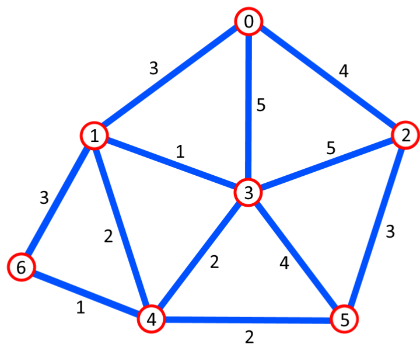


Figure – Graphe

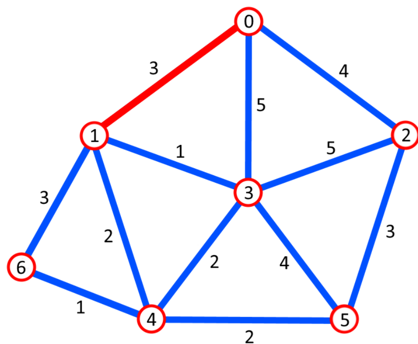


Figure – 1ère itération

Plus proches voisins amélioré

Objectif : former un ensemble de longs chemins à relier : plus_proche2()

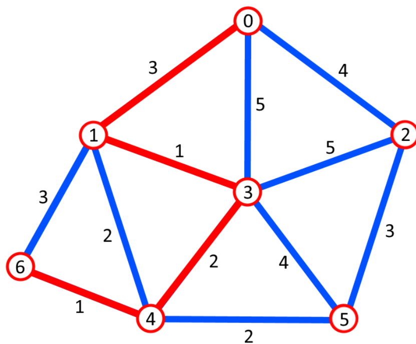


Figure – 1er chemin

Plus proches voisins amélioré

Objectif : former un ensemble de longs chemins à relier : plus_proche2()

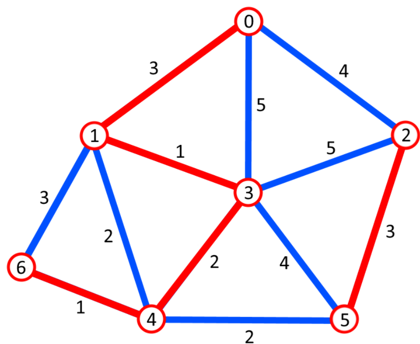


Figure – 2ème chemin

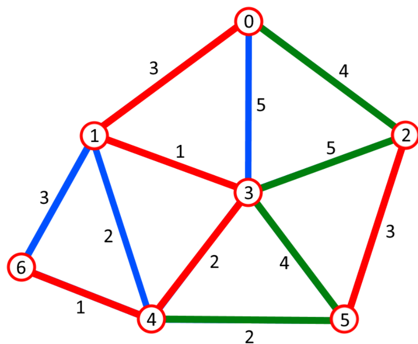


Figure – Liaisons possibles

Plus proches voisins amélioré

Objectif : former un ensemble de longs chemins à relier : plus_proche2()

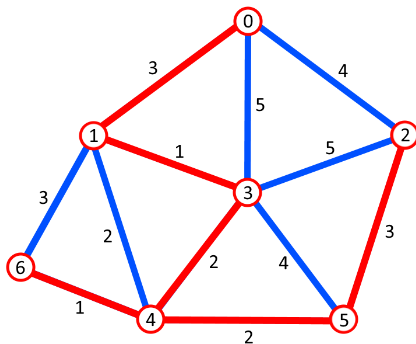


Figure – Chemin final

Comparaisons sur un exemple

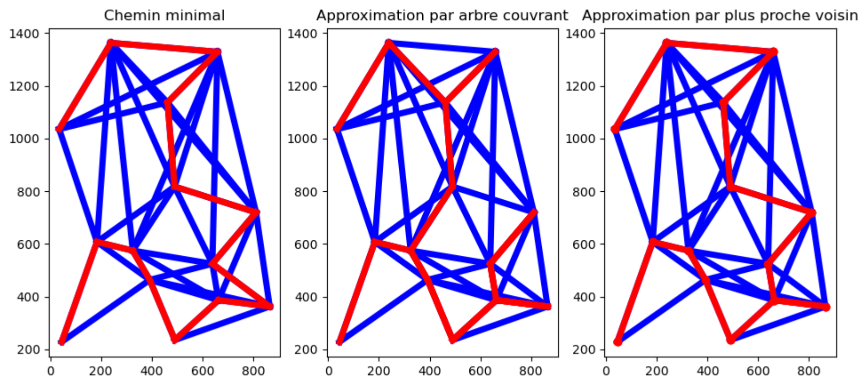


Figure – Par différentes méthodes

Arbre Couvrant : e = 44% / Plus proche voisins : e = 2%

Comparaisons générales

En notant N : nb_villes / Nv : nb_voisins / Na : nb_aretes
Compléxités :

- Naif : O(N !)
- Arbre Couvrant : O(Na x log(N))
- Plus proche voisins : O((N)^2 x Nv)

Pour nb_voisins = 6 :

nb_villes	7	8	9	10	11	12	13
Erreur couvrant	20	20.2	21.1	22.9	25.6	27.2	28.9
Erreur proche	8	8.6	8.7	9	10	11	12

Temps d'exécution « 1s

Conclusion

-> Répond au problème initial

Points positifs :

- Temps d'exécution très faible
- Approximations correctes

Limites :

- Perd en précision pour nb_villes élevé

Ouverture

Le blob :

Figure – Photo d'un blob

Ouverture

Le blob :

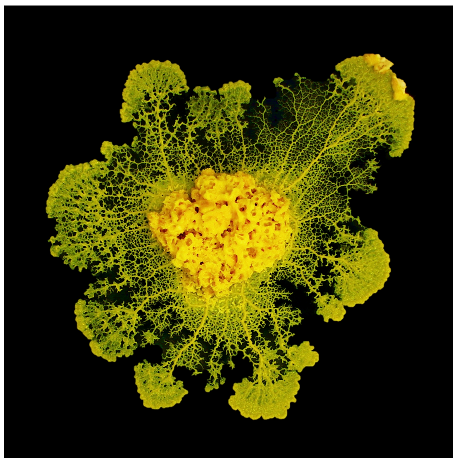


Figure – Blob et réseaux

Annexe 1 : Fonctions générales

```
1 ##Importations
2 from math import *
3 from random import randint
4 from Graphe import Graphe
5 import matplotlib.pyplot as plt
6 import sys
7 from copy import deepcopy
8 sys.setrecursionlimit(2000)
9
10
11 ##Fonctions générales
12
13
14 def longueur_chemin(chemin, g):
15     """Renvoie la longueur d'un chemin entre plusieurs villes dans un graphe"""
16     total = 0
17     for i in range(len(chemin)-1):
18         total += distance_sommet(chemin[i], chemin[i+1], g)
19     return total
20
21
22 def chemin_minimal(liste_chemin, g):
23     """Renvoie le chemin de taille minimale dans une liste de chemin"""
24     mini = (liste_chemin[0], longueur_chemin(liste_chemin[0], g))
25     for i in range(1, len(liste_chemin)):
26         l = longueur_chemin(liste_chemin[i], g)
27         if l < mini[1]:
28             mini = (liste_chemin[i], l)
29     return mini
30
31
32 def chemin_maximal(liste_chemin, g):
33     """Renvoie le chemin de taille maximale dans une liste de chemin"""
34     maxi = (liste_chemin[0], longueur_chemin(liste_chemin[0], g))
35     for i in range(1, len(liste_chemin)):
36         l = longueur_chemin(liste_chemin[i], g)
37         if l > maxi[1]:
38             maxi = (liste_chemin[i], l)
39     return maxi
40
41
42 def code_voisin_vers_chemin(g, code):
43     """Transforme un code en chemin"""
44     chemin = [0]
45
46     for _ in range(len(code)-1):
47         chemin.append(voisin_sommet(chemin[-1], g)[code[chemin[-1]]])
48
49     return chemin
50
```

Annexe 2 : Fonctions graphes (1/2)

```
51 ##Fonctions graphes
52
53
54 def generation_graphe(nb_villes, nb_voisins):
55     """Génère un graphe à partir du nombre de ville et du nombre de voisins"""
56     g = Graphe(nb_villes)
57
58     for i in range(nb_villes):
59         a = i % 5
60         b = i // 5
61         c = int(taille_ecran[0]/5)
62         d = int(taille_ecran[1] / (nb_villes // 5))
63
64
65         g.sommet.append([randint(a*c, a*c+c), randint(b*d, b*d + d)])
66
67
68     for s in range(nb_villes*nb_voisins):
69         r = randint(0, nb_villes - 1)
70         j = sommet_le_plus_proche(r, g)
71         if len(voisin_sommet(r, g)) != nb_voisins and len(voisin_sommet(j, g)) != nb_voisins:
72             g.ajouter_arete(r, j, distance_sommet(r, j, g), False)
73     return g
74
75
76 def afficher_graphe(g, ax, couleur="blue", taille=5):
77     """Affiche un graphe dans une fenetre Matplotlib"""
78     for i in range(len(g.sommet)):
79         for j in range(i):
80             if g.arete[i][j] != 0:
81                 ax.plot([g.sommet[i][0], g.sommet[j][0]], [g.sommet[i][1], g.sommet[j][1]], color=couleur,
82                         linewidth=taille)
83     for x in g.sommet:
84         plt.plot(x[0], x[1], 'ro')
85
86
87 def voisin_sommet(s, g):
88     """Donne la liste des voisins d'un sommet d'un graphe"""
89     ligne_matrice = g.arete[s]
90     voisins = []
91     for i in range(len(ligne_matrice)):
92         if ligne_matrice[i] != 0:
93             voisins.append(i)
94     return voisins
95
96
97 def distance_sommet(s1, s2, g):
98     """Donne la distance entre deux sommets d'un graphe"""
99     return sqrt((g.sommet[s2][0]-g.sommet[s1][0])**2+(g.sommet[s2][1]-g.sommet[s1][1])**2)
100
```

Annexe 2 : Fonctions graphes (2/2)

```
102 def verifier_chemin(c, g):
103     """Vérifie si une liste de sommets correspond à un chemin dans un graphe"""
104     for i in range(len(c)):
105         if g.arete[c[i]][c[i+1]] == 0:
106             return False
107     return True
108 
109 
110 def sommet_le_plus_proche(s, g):
111     """Renvoie le plus proche voisin d'un sommet d'un graphe"""
112     mini = 10000000000
113     vp = s
114     for i in range(len(g.sommet)):
115         if s == i or g.arete[s][i] != 0:
116             continue
117         else:
118             if distance_sommet(s, i, g) < mini:
119                 vp = i
120                 mini = distance_sommet(s, i, g)
121     return vp
122 
123 
124 def voisin_le_plus_proche(s, g, interdit):
125     """Renvoie le plus proche voisin d'un sommet d'un graphe n'appartenant pas à ceux interdit"""
126     mini = 10000000000
127     vp = s
128     voisin = voisin_sommet(s,g)
129     for i in voisin:
130         if g.arete[s][i] < mini and not(i in interdit):
131             vp = i
132             mini = g.arete[s][i]
133     return vp
134 
135 
136 def tracer_chemin(parcours, ax, g, couleur, taille):
137     """Affiche un chemin dans une fenetre Matplotlib"""
138     for i in range(len(parcours)-1):
139         ax.plot([g.sommet[parcours[i]][0], g.sommet[parcours[i+1]][0]],
140                 [g.sommet[parcours[i]][1], g.sommet[parcours[i+1]][1]],
141                 color=couleur, linewidth=taille)
142         plt.pause(0.01)
```

Annexe 3 : Méthode naïve (1/2)

```
147 def chemin_na(g, point_actuel, point_visite, code_voisins, parcours, nouveau=True):
148     """Renvoie un chemin généré de manière récursif"""
149     try:  # Vérifie si on atteint la limite de récursivité de python
150 
151         if len(parcours) == len(g.sommet) and nouveau:  # Si le parcours est de taille n -> Terminé
152             return parcours, code_voisins, 0
153 
154         if point_visite not in parcours:  # Si le point à visiter n'est pas dans le parcours on l'ajoute
155             parcours.append(point_visite)
156             code_voisins[point_actuel] += 1
157             return chemin_na(g, point_visite, (voisin_sommet(point_visite, g))[code_voisins[point_visite]],
158                              code_voisins, parcours)
159 
160         else:  # Si le point à visiter est dans le parcours
161 
162             code_voisins[point_actuel] += 1
163 
164             if code_voisins[point_actuel] >= len(
165                     voisin_sommet(point_actuel, g)):  # Si supérieur au nombre de voisin total de point actuel
166                 while code_voisins[point_actuel] >= len(voisin_sommet(point_actuel, g)):
167                     code_voisins[point_actuel] = 0
168                     try:
169                         point_actuel = parcours[-2]
170                     except:
171                         return [], [], 0
172                     else:
173                         parcours = parcours[0:len(parcours)-1]
174 
175             return chemin_na(g, point_actuel, (voisin_sommet(point_actuel, g))[code_voisins[point_actuel]],
176                              code_voisins, parcours)
177 
178     except RecursionError:
179 
180         while code_voisins[point_actuel] >= len(voisin_sommet(point_actuel, g)):
181             code_voisins[point_actuel] = 0
182             try:
183                 point_actuel = parcours[-2]
184 
185             except:
186                 return [], [], 0
187             else:
188                 parcours = parcours[0:len(parcours)-1]
189 
190         return parcours, code_voisins, 1
```

Annexe 3 : Méthode naïve (2/2)

```
193 def chemins(g):
194     """Renvoie la liste de tous les chemins d'un graphe"""
195     liste_chemin = []
196     c = 0
197 
198     chemin, code_voisin, rec = chemin_na(g, 0, voisin_sommet(0,g)[0], [0]*len(g.sommet), [0])
199 
200     while rec == 1:
201         chemin, code_voisin, rec = chemin_na(g, chemin[-1],  voisin_sommet(chemin[-1], g)[code_voisin[chemin[-1]]],
202                                              code_voisin, chemin)
203 
204     liste_chemin.append(chemin)
205 
206     while chemin != []:
207         chemin, code_voisin, rec = chemin_na(g, chemin[-1],  voisin_sommet(chemin[-1], g)[code_voisin[chemin[-1]]],
208                                              code_voisin, chemin, False)
209 
210         while rec == 1:
211             chemin, code_voisin, rec = chemin_na(g, chemin[-1],  voisin_sommet(chemin[-1], g)[code_voisin[chemin[-1]]],
212                                                  code_voisin, chemin)
213 
214         if chemin not in liste_chemin and chemin != []:
215             liste_chemin.append(chemin)
216 
217     return liste_chemin
```

Annexe 4 : Algorithmes inefficaces

```
222 def recuit(g):
223     """Génère un chemin du graphe et l'améliore par permutation de sommets"""
224     c, code, rec = chemin_na(g, 0, voisin_sommet(0, g)[0], [0]*len(g.sommet), [0])
225     while rec == 1:
226         c, code, rec = chemin_na(g, c[-1], c[-1], code, c)
227     tracer_chemin(c, "red", 5)
228     d1 = longueur_chemin(c)
229     print("Chemin aléatoire : ", d1)
230     for _ in range(10000):
231         a = randint(0, len(c)-1)
232         d = randint(0, len(c)-a-1)
233         if g.arete[a][a+d] != 0:
234             cold = deepcopy(c)
235             b = cold[a+d]
236             cold[a+d] = c[a]
237             cold[a] = b
238             if longueur_chemin(cold) < longueur_chemin(c):
239                 c = cold
240     d2 = longueur_chemin(c)
241     print("Chemin aléatoire amélioré : ", d2)
242     print("Amélioration : ", d1-d2)
243     return longueur_chemin(c)
244 
245 
246 def generation_population(g, nb_individu):
247     """Génére un ensemble de chemin à fusionner"""
248     population = []
249 
250     chemin, code_voisin, rec = chemin_na(g, 0, voisin_sommet(0, g)[0], [0] * len(g.sommet), [0])
251     while rec == 1:
252         chemin, code_voisin, rec = chemin_na(g, chemin[-1], voisin_sommet(chemin[-1], g)[code_voisin[chemin[-1]]],
253                                              code_voisin, chemin)
254 
255     population.append(chemin)
256 
257     for i in range(1, nb_individu):
258 
259         chemin = population[-1]
260         chemin, code_voisin, rec = chemin_na(g, chemin[-1], voisin_sommet(chemin[-1], g)[code_voisin[chemin[-1]]],
261                                              code_voisin, chemin, False)
262         while rec == 1 or chemin in population:
263             chemin, code_voisin, rec = chemin_na(g, chemin[-1], voisin_sommet(chemin[-1], g)[code_voisin[chemin[-1]]],
264                                                  code_voisin, chemin, False)
265         population.append(chemin)
266 
267     return population
```

Annexe 5 : Tas et files de priorité (1/3)

```
274 def echange(t, i, j):
275     """Permute deux noeux du tas"""
276     t[i], t[j] = t[j], t[i]
277     return t
278
279
280 def test_tas(l, cle):
281     """Vérifie si une liste donnée correspond à un tas"""
282     try:
283         for i in range(len(l)):
284             if cle[l[2*i+1]] < cle[l[i]]:
285                 return False
286             elif cle[l[2*i+2]] < cle[l[i]]:
287                 return False
288     except IndexError:
289         return True
290
291
292 def insertion(t, x, cle_x, cle):
293     """Insère un noeud dans le tas"""
294     t.append(x)
295     cle[x] = cle_x
296     indice_x = len(t) - 1
297     while cle[t[int((indice_x-1)/2)]] > cle[t[indice_x]]:
298         echange(t, int((indice_x - 1) / 2), indice_x)
299         indice_x = int((indice_x - 1) / 2)
300     return t
301
302
303 def enfant_indice(t, i, cle):
304     """Donne les indices des enfants du tas"""
305     dernier_indice = len(t) - 1
306     if 2*i + 1 > dernier_indice:
307         return -1
308     elif 2*i + 1 == dernier_indice:
309         return dernier_indice
310     else:
311         if cle[t[2*i+1]] < cle[t[2*i+2]]:
312             return 2*i+1
313         else:
314             return 2*i+2
```

Annexe 5 : Tas et files de priorité (2/3)

```
317 def descendre(t, i, cle):
318     """Descend un élément dans un tas"""
319     p = enfant_indice(t, i, cle)
320     if p > 0:
321         if cle[t[p]] < cle[t[i]]:
322             echange(t, i, p)
323             descendre(t, p, cle)
324     return t
325
326
327 def remonter(t, i, cle):
328     """Remonte un élément dans un tas"""
329     indice_x = i
330     while cle[t[int((indice_x - 1) / 2)]] > cle[t[indice_x]]:
331         echange(t, int((indice_x - 1) / 2), indice_x)
332         indice_x = int((indice_x - 1) / 2)
333     return t
334
335
336 def changer_priorite(t, sommet, valeur, cle):
337     """Change la priorité d'un élément et génère le nouveau tas associé"""
338     ancien = cle[sommet]
339     cle[sommet] = valeur
340
341     if valeur < ancien:
342         t = remonter(t, t.index(sommet), cle)
343
344     elif valeur > ancien:
345         t = descendre(t, t.index(sommet), cle)
346     return t
347
348
349 def min_tas(t, cle):
350     """Renvoie l'élément minimal du tas et le tas privé de cet élément"""
351     minimum = t[0]
352     if len(t) == 1:
353         return minimum, []
354     else:
355         a = t.pop()
356         t[0] = a
357         descendre(t, 0, cle)
358     return minimum, t
```

Annexe 5 : Tas et files de priorité (3/3)

```
364 def cree_file():
365     return []
366
367
368 def est_vide(f):
369     return f == []
370
371
372 def ajoute_file(f, x, cle_x, cle):
373     return insertion(f, x, cle_x, cle)
374
375
376 def premier_file(f):
377     return f[0]
378
379
380 def reste_file(f, cle):
381     return min_tas(f, cle)
```

Annexe 6 : Arbre couvrant (1/2)

```
384 def prim(g, s0):
385     """Détermine l'arbre couvrant minimal d'un graphe"""
386 
387     file = cree_file()
388 
389     cle_sommet = [float("inf")]*len(g.sommet)
390     pred = [None]*len(g.sommet)
391     cle_sommet[s0] = 0
392 
393     for s in range(0, len(g.sommet)):
394         ajoute_file(file, s, cle_sommet[s], cle_sommet)
395 
396     i=0
397 
398     while not est_vide(file):
399         i+=1
400         res = reste_file(file, cle_sommet)
401         s1, file = res[0], res[1]
402         for s2 in range(len(g.sommet)):
403             if g.arete[s1][s2] != 0 and s2 in file:
404                 if cle_sommet[s2] >= g.arete[s1][s2]:
405                     pred[s2] = s1
406                     changer_priorite(file, s2, g.arete[s1][s2], cle_sommet)
407 
408     return pred
409 
410 
411 def arbre_to_graphe(g_ref, a):
412     """Génère le graphe associé à une liste représentant un arbre"""
413     g = Graphe(len(g_ref.sommet))
414     g.sommet = g_ref.sommet
415     for i in range(len(a)):
416         if a[i] != None:
417             g.ajouter_arete(a[i], i, distance_sommet(a[i], i, g_ref), False)
418     return g
```

Annexe 6 : Arbre couvrant (2/2)

```
421 def contient(elements, l):
422     """Vérifie si une liste contient un ensemble d'éléments"""
423     for i in elements:
424         if not(i in l):
425             return False
426     return True
427
428
429 def chemin_arbre(a, s0):
430     """Trouve le chemin parcourant un arbre"""
431     def aux(a, sommet, chemin):
432
433         chemin.append(sommet)
434
435         if len(voisin_sommet(sommet, a)) == 1:
436             if voisin_sommet(sommet, a)[0] in chemin:
437                 return chemin
438             else:
439                 return aux(a, voisin_sommet(sommet, a)[0], [sommet])
440
441         elif len(voisin_sommet(sommet, a)) >= 2:
442
443             l = []
444
445             for i in voisin_sommet(sommet, a):
446                 if not (i in chemin):
447                     l.append(aux(a, i, [sommet]))
448
449             while len(l) != 1:
450                 mini = (float("inf"), -1)
451                 for i in range(len(l)):
452                     if longueur_chemin(l[i], a) <= mini[0]:
453                         mini = (longueur_chemin(l[i], a), i)
454
455                 chemin = chemin + l[mini[1]][1:]
456
457                 for j in range(len(l[mini[1]])-1):
458                     chemin.append(l[mini[1]][-2-j])
459
460                 l.pop(mini[1])
461
462             chemin = chemin + l[0][1:]
463
464             return chemin
465
466     c = aux(a, s0, [])
467     return (c, longueur_chemin(c, a))
```

Annexe 7 : Plus proches voisins (1/3)

```
471 def plus_proche(g, s0):
472     """Détermine une suite de sommet par la méthode des plus proches voisins"""
473     chemin = [s0]
474 
475     while voisin_le_plus_proche(chemin[-1], g, chemin) != chemin[-1]:
476         r = voisin_le_plus_proche(chemin[-1], g, chemin)
477         chemin.append(r)
478 
479     return (chemin, longueur_chemin(chemin, g))
480 
481 
482 def plus_proche_restreint(g, s0, interdit):
483     """Détermine une suite de sommet par la méthode des plus proches voisins n'appartenant par à interdit"""
484     chemin = [s0]
485 
486     while voisin_le_plus_proche(chemin[-1], g, chemin + interdit) != chemin[-1]:
487         r = voisin_le_plus_proche(chemin[-1], g, chemin + interdit)
488         chemin.append(r)
489 
490     return (chemin, longueur_chemin(chemin, g))
```

Annexe 7 : Plus proches voisins (2/3)

```
493 def plus_proche2(g, s0):
494     """Détermine un chemin par fusion de petits chemins créés par la méthode des plus proches voisins"""
495     chemin = plus_proche(g, s0)[0]
496 
497     if len(chemin) == len(g.sommet):
498         return (chemin, longueur_chemin(chemin, g))
499 
500     else:
501         while len(chemin) < len(g.sommet) or not contient(range(len(g.sommet)), chemin):
502 
503             liste_petit_chemin = []
504 
505             for i in range(len(g.sommet)):
506                 if not (i in chemin):
507                     liste_petit_chemin.append(plus_proche_restreint(g, i, chemin)[0])
508 
509             maxi = (0, len(liste_petit_chemin[0]))
510             for i in range(len(liste_petit_chemin)):
511                 if len(liste_petit_chemin[i]) > maxi[1]:
512                     maxi = (i, len(liste_petit_chemin[i]))
513 
514             petit_chemin = liste_petit_chemin[maxi[0]]
515 
516             fusion_possible = []
517 
518             for i in chemin:
519                 for j in petit_chemin:
520                     if g.arete[i][j] != 0:
521                         fusion_possible.append((i, j))
522 
523             nouveaux_chemins = []
```

Annexe 7 : Plus proches voisins (3/3)

```
525             for c in fusion_possible:
526 
527                 graphe_chemin = Graphe(len(g.sommet))
528                 graphe_chemin.sommet = g.sommet
529 
530                 for i in range(0, len(chemin) - 1):
531                     graphe_chemin.ajouter_arete(chemin[i], chemin[i + 1], g.arete[chemin[i]][chemin[i + 1]], False)
532 
533                 for i in range(0, len(petit_chemin) - 1):
534                     graphe_chemin.ajouter_arete(petit_chemin[i], petit_chemin[i + 1],
535                                                 g.arete[petit_chemin[i]][petit_chemin[i + 1]], False)
536 
537                 graphe_chemin.ajouter_arete(c[0], c[1], g.arete[c[0]][c[1]], False)
538 
539                 nouveaux_chemins.append(chemin_arbre(graphe_chemin, 0)[0])
540 
541             longueurs_chemins = [longueur_chemin(ch, g) for ch in nouveaux_chemins]
542             indice = longueurs_chemins.index(min(longueurs_chemins))
543 
544             chemin = nouveaux_chemins[indice]
545 
546         return (chemin, longueur_chemin(chemin, g))
547 
548 taille_ecran = (1000, 1000)
```

Annexe 8 : Statistiques (1/3)

```
1 from main import *
2 import time
3
4 def comparaison_1_graphe(nb_villes, nb_voisins, afficher_stats=True, afficher_graph=True):
5
6     graphe = generation_graphe(nb_villes, nb_voisins)
7
8     # Chemin naif
9
10    temps_debut_naif = time.time()
11    liste_chemin = chemins(graphe)
12    temps_execution_naif = time.time() - temps_debut_naif
13
14    chemin_mini_naif = chemin_minimal(liste_chemin, graphe)
15
16    # Chemin arbre couvrant
17
18    temps_debut_couvrant = time.time()
19    arbre_couvrant_liste = prim(graphe, 0)
20    graphe_arbre_couvrant = arbre_to_graphe(graphe, arbre_couvrant_liste)
21    chemin_mini_couvrant = chemin_arbre(graphe_arbre_couvrant, 0)
22    temps_execution_couvrant = time.time() - temps_debut_couvrant
23    pourcentage_erreur_couvrant = round((abs(chemin_mini_couvrant[1] - chemin_mini_naif[1]) /
24                                        chemin_mini_naif[1]) * 100)
25
26    # Chemin plus proche voisins
27
28    temps_debut_proche = time.time()
29    chemin_mini_proche = plus_proche2(graphe, 0)
30    temps_execution_proche = time.time() - temps_debut_proche
31    pourcentage_erreur_proche = round((abs(chemin_mini_proche[1] - chemin_mini_naif[1]) /
32                                      chemin_mini_naif[1]) * 100)
33
34    total = 0
35    for chemin in liste_chemin:
36        total += longueur_chemin(chemin, graphe)
```

Annexe 8 : Statistiques (2/3)

```
38      if afficher_stats:
39
40          print("Nombre de chemins total : ", len(liste_chemin))
41          print("Chemin minimal : ", chemin_mini_naif)
42          print("Moyenne chemin : ", total / len(liste_chemin))
43          print("Temps d'execution tout chemin : ", temps_execution_naif)
44          print("\n")
45
46          print("Chemin par arbre :" + str(chemin_mini_couvrant))
47          print("Temps d'execution arbre couvrant : ", temps_execution_couvrant)
48          print("Pourcentage erreur : " + str(pourcentage_erreur_couvrant) + " %")
49          print("\n")
50
51          print("Chemin par plus proche :" + str(chemin_mini_proche))
52          print("Temps d'execution plus proches voisins : ", temps_execution_proche)
53          print("Pourcentage erreur : " + str(pourcentage_erreur_proche) + " %")
54
55      if afficher_graph:
56
57          ax1 = plt.subplot(131)
58          plt.title("Chemin minimal")
59          ax2 = plt.subplot(132)
60          plt.title("Approximation par arbre couvrant")
61          ax3 = plt.subplot(133)
62          plt.title("Approximation par plus proche voisin")
63
64          afficher_graphe(graphe, ax1, "blue")
65          afficher_graphe(graphe, ax2, "blue")
66          afficher_graphe(graphe, ax3, "blue")
67
68          tracer_chemin(chemin_mini_naif[0], ax1, graphe, "red", 3)
69
70          tracer_chemin(chemin_mini_couvrant[0], ax2, graphe, "red", 3)
71
72          tracer_chemin(chemin_mini_proche[0], ax3, graphe, "red", 3)
73
74          plt.show()
75
76      return len(liste_chemin), total, chemin_mini_naif, chemin_mini_couvrant, chemin_mini_proche, temps_execution_naif, \
77          temps_execution_couvrant, temps_execution_proche, pourcentage_erreur_couvrant, pourcentage_erreur_proche
```

Annexe 8 : Statistiques (3/3)

```
80 def statistique_boucle(nb_villes, nb_voisins, nb_iterations):
81
82     fichier = open("stats.txt", "a")
83     i = 0
84
85     for _ in range(nb_iterations):
86         try:
87             res = comparaison_1_graphe(nb_villes, nb_voisins, False, False)
88             fichier.write(str(nb_villes) + " " + str(nb_voisins) + " " + str(res[0]) + " " + str(
89                 (res[1] / res[0])) + " " + str(res[2][1]) + " " + str(res[3][1]) + " " + str(res[4][1]) + " " + str(
90                 res[5]) + " " + str(res[6]) + " " + str(res[7]) + " " + str(res[8]) + " " + str(res[9]) + "\n")
91             i+=1
92             if i%50 == 0:
93                 print(i)
94
95         except ZeroDivisionError:
96             continue
97
98         except IndexError:
99             continue
100
101     fichier.close()
102
103
104 def statistiques(n_initial, n_final, nb_iterations):
105     for n in range(n_initial, n_final):
106         for v in range(3, 7):
107             if v >= n:
108                 continue
109             statistique_boucle(n, v, nb_iterations)
```

Annexe 9 : Objet Graphe

```
1  from numpy import *
2
3  class Graphe:
4
5      def __init__(self, n):
6          self.sommet = [] #liste de couple de coordonnées
7          self.arete = zeros((n, n))
8
9      def ajouter_arete(self, i, j, l, oriente=True):
10         self.arete[i][j] = l
11         if not oriente:
12             self.arete[j][i] = l
13
14     def supprimer_arete(self, i, j, oriente=True):
15         self.arete[i][j] = 0
16         if not oriente:
17             self.arete[j][i] = 0
18
```