Codes Python TIPE

Coërchon Colin

2021/2022

Table des matières

1.1 - Triangulation.py

```
001| ## Triangulation ##
002|
003| def ind_voisin(s,degre,n):
004|     # Retourne l'indice du (s+degré) ième sommet.
005|     return (s+degre)%n
006|
007|
008| def produit_vec(S1,S2,M):
009|     # Retourne Vec(S1S2) ∧ Vec(S1M)
010|     V1=(S2[0]-S1[0],S2[1]-S1[1])
011|     V2=(M[0]-S1[0],M[1]-S1[1])
012|     return(V1[0]*V2[1] - V1[1]*V2[0])
013|
014|
015| def triangle(polygone,indices):
016|     # Retourne le triangle respectivement associé aux 3 indices parmi les sommets
du polygone
017|     return([polygone[indices[0]],polygone[indices[1]],polygone[indices[2]]])
018|
019|
020| def point_dans_triangle(polygone,indices,M):
021|     # Retourne True si le point M est dans le triangle associé aux indices
022|     T=triangle(polygone,indices)
023|     S0=T[0]
024|     S1=T[1]
025|     S2=T[2]
026|     p1=produit_vec(S0,S1,M)
027|     p2=produit_vec(S1,S2,M)
028|     p3=produit_vec(S2,S0,M)
029|     return((p1>0 and p2>0 and p3>0) or (p1<0 and p2<0 and p3<0))
030|
031|
032| def sommet_distance_max(polygone,indices):
033|     # Retourne, s'il existe, l'indice du sommet dans le triangle ABC qui se situe
le plus loin de la droite AC. Sinon, retourne None
034|     n=len(polygone)
035|     maxi=-1
036|     ind_max=-1
037|     T=triangle(polygone,indices)
038|     for i in range(n):
039|         M=polygone[i]
040|         if not(i in indices) and point_dans_triangle(polygone,indices,M):
041|             x = abs(produit_vec(T[0],T[2],M))
042|             if x>maxi:
043|                 ind_max=i
044|                 maxi=x
045|     if ind_max==-1: return(None)
046|     else: return(ind_max)
047|
048|
049| def sommet_gauche(polygone):
050|     # Retourne l'indice du sommet situé le plus à gauche dans le polygone
051|     n=len(polygone)
052|     if n<3: return('problème sur le nbre de sommets')
053|     xg = polygone[0][0]
054|     ig = 0
055|     for i in range(1,n):
056|         if polygone[i][0] < xg:
057|             xg = polygone[i][0]
058|             ig = i
059|     return ig
060|
061| # La fonction sommet_gauche est essentielle car elle permet de traiter uniquement
des triangles contenus dans le polygone.
062|
063|
```

```
064| def nouveau_polygone(polygone,i_d,i_f):
065|     # Retourne le sous-polygone allant du sommet d'indice i_d au sommet d'indice
i_f
066|     n=len(polygone)
067|     if n==3:
068|         return(polygone)
069|     else:
070|         new_p=[]
071|         i=i_d
072|         while i != i_f :
073|             new_p.append(polygone[i])
074|             i=ind_voisin(i,1,n)
075|         new_p.append(polygone[i_f])
076|         return(new_p)
077|
078|
079| def trianguler(polygone):
080|     n0 = len(polygone)
081|     if n0 < 3:
082|         return("Trop peu de sommets")
083|     else:
084|         def trianguler_rec(polygone,liste_triangles):
085|             n=len(polygone)
086|             i1=sommet_gauche(polygone)
087|             i0=ind_voisin(i1,-1,n)
088|             i2=ind_voisin(i1,1,n)
089|             T=triangle(polygone,[i0,i1,i2])             #On traite CE triangle
090|             i=sommet_distance_max(polygone,[i0,i1,i2])
091|             if i==None:    # Si i = None : aucun sommet du polygone est contenu
dans le traingle
092|                 liste_triangles.append(T)
093|                 poly1=nouveau_polygone(polygone,i2,i0)
094|                 if len(poly1)==3:
095|                     liste_triangles.append(poly1)
096|                 else:      # On procède récursivement sur les sommets qui restent
097|                     trianguler_rec(poly1,liste_triangles)
098|             else:          # Sinon : il y a (au moins) un sommet contenu dans le
triangle
099|                 poly1=nouveau_polygone(polygone,i1,i)
100|                 poly2=nouveau_polygone(polygone,i,i1)
101|                 if len(poly1)==3:
102|                     liste_triangles.append(poly1)
103|                 else:
104|                     trianguler_rec(poly1,liste_triangles)
105|                 if len(poly2)==3:
106|                     liste_triangles.append(poly2)
107|                 else:
108|                     trianguler_rec(poly2,liste_triangles)
109|                 # On procède récursivement sur les deux polygones ainsi créés
110|             return(liste_triangles)
111|
112|         return(trianguler_rec(polygone,[]))
113|
114| # La fonction prend en argument un polygone et retourne ainsi une triangulation
de ce dernier : une liste des triangles qui partitionne le polygone.
115|
```

1.2 - Parcours en largeur sur les triangles voisins.py

```python
## Détermination du chemin issu de la triangulation ##

def voisin(T1,T2):
    # Retourne True si les triangles T1 et T2 sont voisins
    nb_communs=0
    for x in T1:
        if x in T2:
            nb_communs+=1
    return(nb_communs==2)


def liste_adj(l):
    # Retourne la liste d'adjacence associée à la liste des triangles du polygone*
    n=len(l)
    adj=[[] for _ in range(n)]
    for i in range(n):
        for j in range(i+1,n):
            if voisin(l[i],l[j]):
                adj[i].append(j)
                adj[j].append(i)
    return(adj)


def parcours_en_largeur(l_adj,depart):
    n=len(l_adj)
    file=[depart]
    c=["blanc" for _ in range(n)]
    p=[-1 for _ in range(n)]
    def traite_fils(u,l_voisins):
        for i in l_voisins:
            if c[i]=="blanc":
                p[i]=u
                file.append(i)
                c[i]="gris"
    def traite(u):
        traite_fils(u,l_adj[u])
        c[u]="noir"

    while file != []:
        traite(file.pop(0))
    return(p)


def dans_triangle(T,M):
    # Retourne True si M est dans le triangle T (comprend les bords)
    S0=T[0]
    S1=T[1]
    S2=T[2]
    p1=produit_vec(S0,S1,M)
    p2=produit_vec(S1,S2,M)
    p3=produit_vec(S2,S0,M)
    return((p1>=0 and p2>=0 and p3>=0) or (p1<=0 and p2<=0 and p3<=0))


def trouve_depart_arrivee(l_T,d,a):
    # Retourne l'indice du triangle de départ, et celui d'arrivée
    n=len(l_T)
    i_d,i_a=0,0
    for i in range(n):
        if dans_triangle(l_T[i],d):
            i_d=i
        if dans_triangle(l_T[i],a):
            i_a=i
    return(i_d,i_a)


```

```
67| def centre_triangle(T):
68|     # Retourne le centre du triangle T : la moyenne respective des abscisses et
des ordonnées des sommets du triangle
69|     t_moy=np.round(np.mean(T,axis=0),2)
70|     return([t_moy[0],t_moy[1]])
71|
72|
73| def PCC_Triangulation(P,depart,arrivee):
74|     # Retourne le chemin du départ jusqu'à l'arrivée, résultant du parcours en
largeur
75|     l_T=trianguler(P)
76|     D,A=trouve_depart_arrivee(l_T,depart,arrivee)
77|     l_adj=liste_adj(l_T)
78|     p=parcours_en_largeur(l_adj,D)
79|     i=A
80|     chemin=[centre_triangle(l_T[i])]
81|     while i != D:   # On parcourt la liste des Pères pour construire le chemin
résultant
82|         i=p[i]
83|         T=l_T[i]
84|         x=centre_triangle(T)
85|         chemin = [x] + chemin
86|     chemin=[depart]+chemin+[arrivee]
87|     return(chemin)
88|
```

```
# 1.3 - Résolution du problème des sorties de frontière.py

## Résolution du problème sur PCC_Triangulation ##

def sommets_communs(T1,T2):
    # Retourne la liste des sommets en commun entre T1 et T2
    res=[]
    for x in T1:
        if x in T2:
            res.append(x)
    return(res)


def orientation(X,Y,M):
    p=produit_vec(X,Y,M)
    if p>0: return(1)
    elif p==0: return(0)
    else: return(-1)


def croisement(X,Y,P1,P2):
    # Retourne True si il y a croisement entre les segments [XY] et [P1P2]
    if X==P1 or X==P2 or X==P1 or X==P1 or orientation(P1,P2,X)==0 or
orientation(P1,P2,Y)==0:
        return False
    elif orientation(P1,X,P2)==orientation(P1,X,Y) and
orientation(P1,Y,X)==orientation(P1,Y,P2) and orientation(X,Y,P1) !=
orientation(X,Y,P2):
        return True
    else:
        return False


def croisement_polygone(X,Y,poly):
    # Retourne True si [XY] sort de la frontière du polygone (donc si il y a un
problème)
    for k in range(len(poly)):
        if croisement(X,Y,poly[k-1],poly[k]):
            return True
    return False


def distance_2points(P1,P2):
    # Retourne la distance à vol d'oiseau entre P1 et P2
    return(np.round(sqrt((P2[0]-P1[0])**2 + (P2[1]-P1[1])**2),2))


def PCC_Triangulation2(P,depart,arrivee):
    l_T=trianguler(P)
    D,A=trouve_depart_arrivee(l_T,depart,arrivee)
    l_adj=liste_adj(l_T)
    p=parcours_en_largeur(l_adj,D)
    i=A
    j=A
    chemin=[centre_triangle(l_T[i])]

    while i != D: # On parcourt la liste des Pères pour construire le chemin
résultant
        j=i
        i=p[i]
        T=l_T[i]
        x=centre_triangle(T)
        if len(chemin)>0 and croisement_polygone(x,chemin[0],P):
            # Cette partie sert uniquement à gérer les tracés qui passent hors
limites du polygone
            S_com = sommets_communs(l_T[j],l_T[i])
            d1=distance_2points(S_com[0],x)
            d2=distance_2points(S_com[1],x)
```

```
61|             if d1<=d2:
62|                 chemin = [S_com[0]] + chemin
63|             else:
64|                 chemin = [S_com[1]] + chemin
65|         else:
66|             chemin = [x] + chemin
67|
68|     chemin=[depart]+chemin+[arrivee]
69|     return(chemin)
70|
```

1.4 - Dijkstra.py

```
001| ## Dijkstra ##
002|
003| def dans_polygone(i,j,poly):
004|     # Retourne True si le segment [ij] reste dans l'intérieur du polygone
005|     n=len(poly)
006|     if abs(i-j)==1 or (i>=n or j>=n):
007|         return True
008|     else:
009|         X=poly[i]
010|         Y=poly[j]
011|         P0=poly[i-1]
012|         P1=poly[(i+1)%n]
013|         if orientation(P0,X,P1)== 1:
014|             return (orientation(P0,X,P1) == orientation(P0,X,Y) and
orientation(P1,X,P0) == orientation(P1,X,Y))
015|         else:
016|             return (orientation(P0,X,P1) != orientation(P0,X,Y) or
orientation(P1,X,P0) != orientation(P1,X,Y))
017|
018|
019| def graphe(poly,depart,arrivee):
020|     # Retourne la matrice d'adjacence du graphe des déplacements entre les
sommets du polygone
021|     S=poly+[depart]+[arrivee]
022|     n=len(S)
023|     g=(-1)*np.ones((n,n),float)
024|     for i in range(n):
025|         for j in range(i+1,n):
026|             if not(croisement_polygone(S[i],S[j],poly)) and
dans_polygone(i,j,poly):
027|                 g[i,j]=distance_2points(S[i],S[j])
028|                 g[j,i]=g[i,j]
029|     return(g)
030|
031|
032| def successeurs(s,G):
033|     # Retourne l'ensemble des sommets accessibles depuis le sommet s
034|     l=[]
035|     for i in range(np.shape(G)[0]):
036|         if G[s,i]!= -1 :
037|             l.append(i)
038|     return(l)
039|
040|
041| def ajouter_file(file,sommet,distance):
042|     # Retourne la file de priorité dans laquelle on a ajouté le sommet
043|     b=len(file)-1
044|     a=0
045|     while a<=b:
046|         m=(a+b)//2
047|         if sommet==file[m][0]:
048|             if distance<file[m][1]:
049|                 file.pop(m)
050|                 b=m-2
051|             else:
052|                 return(file)
053|         elif distance<file[m][1]:
054|             b=m-1
055|         else:
056|             a=m+1
057|     file=file[:a]+[(sommet,distance)]+file[a:]
058|     return(file)
059|
060|
061|
062|
```

```python
def Dijkstra(G):
    n=np.shape(G)[0]

    # Position du départ : n-2  / Position de l'arrivee : n-1
    i_d,i_a=n-2,n-1

    file=[(i_d,-1)] # La valeur -1 est arbitraire, elle n'intervient jamais
    C=["blanc" for _ in range(n)]
    D=[np.infty for _ in range(n)]
    Peres=[None for _ in range(n)]

    D[i_d]=0
    Peres[i_d]= -1
    while file != []:
        pivot=file.pop(0)[0]
        C[pivot]="noir"
        if pivot==i_a:
            break
        for s in successeurs(pivot,G):
            newD=np.round(D[pivot]+G[pivot,s],2)
            if C[s]=="blanc" and (newD < D[s]):
                D[s]=newD
                Peres[s]=pivot
                file=ajouter_file(file,s,newD)
    return(Peres)


def PCC_Dijkstra(poly,depart,arrivee):
    n=len(poly)
    G=graphe(poly,depart,arrivee)

    Peres=Dijkstra(G)
    chemin=[arrivee]
    i=Peres[n+1]
    while i != n:
        chemin=[poly[i]]+chemin
        i=Peres[i]
    chemin=[depart]+chemin
    return(chemin)

```

```
# 1.5 - Parcours en largeur glouton.py

## Greedy Best-First Search ##


def list_adj(poly,depart,arrivee):
    S=poly+[depart]+[arrivee]
    n=len(S)
    l=[[] for _ in range(n)]
    for i in range(n):
        for j in range(i+1,n):
            if not(croisement_polygone(S[i],S[j],poly)) and
dans_polygone(i,j,poly):
                l[i].append(j)
                l[j].append(i)
    return(l)


def GBFS(poly,depart,arrivee):
    l=list_adj(poly,depart,arrivee)
    n=len(l)
    # Position du départ : n-2  / Position de l'arrivee : n-1
    i_d,i_a=n-2,n-1

    P=poly+[depart]+[arrivee]
    file=[(i_d,-1)] # la valeur -1 est arbitraire, elle n'intervient jamais
    Peres=[None for _ in range(n)]
    C=["blanc" for _ in range(n)]

    Peres[i_d]= -1
    while file != []:
        pivot=file.pop(0)[0]
        C[pivot]="noir"
        if pivot==i_a:
            break
        for s in l[pivot]:
            if C[s]=="blanc":
                Peres[s]=pivot
                d=distance_2points(P[s],arrivee)
                file=ajouter_file(file,s,d)
                C[s]="gris"
    return(Peres)


def PCC_GBFS(poly,depart,arrivee):
    n=len(poly)
    # Position du départ : n / Position de l'arrivee : n+1

    Peres=GBFS(poly,depart,arrivee)
    chemin=[arrivee]
    i=Peres[n+1]
    while i != n:
        chemin=[poly[i]]+chemin
        i=Peres[i]
    chemin=[depart]+chemin
    return(chemin)

```

```python
# 1.6 - Astar.py

## A* ##

def Astar(poly,depart,arrivee):
    G=graphe(poly,depart,arrivee)
    n=np.shape(G)[0]

    # Position du départ : n-2  / Position de l'arrivee : n-1
    i_d,i_a=n-2,n-1

    P=poly+[depart]+[arrivee]
    file=[(i_d,-1)] # la valeur -1 est arbitraire, elle n'intervient jamais
    C=["blanc" for _ in range(n)]
    D=[np.infty for _ in range(n)]
    Peres=[None for _ in range(n)]

    D[i_d]=0
    Peres[i_d]= -1

    while file != []:
        pivot=file.pop(0)[0]
        C[pivot]="noir"

        if pivot==i_a:
            break
        for s in successeurs(pivot,G):
            newD=np.round(D[pivot]+G[pivot,s],2)
            if C[s]=="blanc" and (newD < D[s]):
                D[s]=newD
                Peres[s]=pivot
                d = np.round(newD + distance_2points(P[s],arrivee),2)
                file=ajouter_file(file,s,d)
    return(Peres)


def PCC_Astar(poly,depart,arrivee):
    n=len(poly)
    # Position du départ : n  / Position de l'arrivee : n+1

    Peres=Astar(poly,depart,arrivee)
    chemin=[arrivee]
    i=Peres[n+1]
    while i != n:
        chemin=[poly[i]]+chemin
        i=Peres[i]
    chemin=[depart]+chemin
    return(chemin)

```

```
# 1.7 - Rendu visuel.py

## Tracé de tous les chemins trouvés ##

from math import *
import time
import matplotlib.pyplot as plt
from matplotlib.patches import Polygon
from matplotlib.collections import PatchCollection
import numpy as np

def agrandir1():
    mngr = plt.get_current_fig_manager()
    mngr.window.setGeometry(400,50,1100,1000)
    #f.set_tight_layout(True)
    plt.tight_layout(pad=0.5)

def agrandir2():
    mngr = plt.get_current_fig_manager()
    mngr.window.setGeometry(50,50,1800,1000)
    #f.set_tight_layout(True)
    plt.tight_layout(pad=0.1)


## Les polygones ##

poly = [[0,0],[0.5,-1],[1.5,-0.2],[2,-0.5],[2,0],[1.5,1],[0.3,0],[0.5,1]]
depart=[0.3,0.5]
arrivee=[2,-0.3]


poly = [[-1.5,1.94],[-2.46,-4.3],[3.18,-1.8],[5.7,-6.84],[10,-4],[4.9,-1.56],
[12,-1],[5.2,0.42],[11.86,2.94],[5,2],[4.62,4.04],[3.92,2.28],[1.42,2.48],
[-0.76,3.72],[0,1.56],[3.28,0.92],[1.82,-1]]
depart=[-0.5,3.41]
arrivee=[-1.58,-0.44]


poly = [[-3.37,15.03],[-9.44,11.54],[-4.31,11.75],[-8.45,6.59],[-9.61,2.98],
[-16.96,-1.94],[-12.48,-0.72],[-6.99,-4.09],[0.86,0.86],[-2.17,6.39],[3.57,3.73],
[3.07,-2.09],[0.78,-5.25],[5.94,-2.26],[7.13,4.71],[12.76,-2.59],[17.08,3.52],
[11.37,2.18],[26.93,13.4],[6.48,6.39],[1.79,5.23],[0.55,8.51],[8.6,15.7],[14.79,12.7],
[16.79,14.95],[12.46,16.65],[1.2,16.4],[5.44,15.32],[2.03,15.28],[-6.20,1.19],
[-6.02,4.49],[-0.71,15.24],[-2.29,16.86]]
depart = [-6.70,12.70]
arrivee = [15,1.64]


## Les tracés ##

chemin_triangulation=PCC_Triangulation2(poly,depart,arrivee)
chemin_GBFS=PCC_GBFS(poly,depart,arrivee)
chemin_Dijkstra=PCC_Dijkstra(poly,depart,arrivee)
chemin_Astar=PCC_Astar(poly,depart,arrivee)


plt.plot(np.array(poly+[poly[0]])[:,0],np.array(poly+[poly[0]])[:,
1],color='black')

plt.plot(np.array(chemin_triangulation)[:,0],np.array(chemin_triangulation)[:,
1],color='red',linewidth=3,label="Triangulation")
plt.plot(np.array(chemin_GBFS)[:,0],np.array(chemin_GBFS)[:,
1],color='lime',linewidth=3,label='GBFS')
plt.plot(np.array(chemin_Dijkstra)[:,0],np.array(chemin_Dijkstra)[:,
1],color='magenta',linewidth=3,label='Dijkstra')
plt.plot(np.array(chemin_Astar)[:,0],np.array(chemin_Astar)[:,
1],color='blue',ls='--',linewidth=3,label='A*')

```

```
plt.plot(depart[0],depart[1],marker='o',c='g',markersize=10)
plt.plot(arrivee[0],arrivee[1],marker='o',c='orange',markersize=10)
plt.legend(loc='lower right',fontsize=23)


agrandir1()
plt.show()

def distance_chemin(l):
    d=0
    n=len(l)
    for i in range(n-1):
        d+=distance_2points(l[i],l[i+1])
    return(np.round(d,2))

d1 = distance_chemin(chemin_triangulation)
d2 = distance_chemin(chemin_GBFS)
d3 = distance_chemin(chemin_Dijkstra)
d4 = distance_chemin(chemin_Astar)

print("Distance du parcours avec Triangulation :",d1)
print("Distance du parcours avec Greedy Best-First Search :",d2)
print("Distance du parcours avec Dijkstra :",d3)
print("Distance du parcours avec A* :",d4)

```

2.1 - Implémentation naïve.py

```
01|
02| ## Delaunay - Implémentation naïve ##
03|
04| def Delaunay(ListePoints):
05|     ListeTriangle = list(combinations(ListePoints,3))
06|     Triangulation=[]
07|     for T in ListeTriangle:
08|         bool=True
09|         ListD=[]
10|
11|         A1=2*(T[1][0]-T[0][0])
12|         B1=2*(T[1][1]-T[0][1])
13|         C1=T[0][0]**2 + T[0][1]**2 - T[1][0]**2 - T[1][1]**2
14|
15|         A2=2*(T[2][0]-T[1][0])
16|         B2=2*(T[2][1]-T[1][1])
17|         C2=T[1][0]**2 + T[1][1]**2 - T[2][0]**2 - T[2][1]**2
18|
19|         X=(C1*B2-C2*B1) / (A2*B1-A1*B2)
20|         Y=(C1*A2-A1*C2) / (A1*B2-B1*A2)
21|
22|         R=(X-T[1][0])**2 + (Y-T[1][1])**2
23|
24|         for P in ListePoints:
25|             if not(P in T):
26|                 distance=(P[0]-X)**2+(P[1]-Y)**2
27|                 ListD.append(distance)
28|                 for D in ListD:
29|                     if D<=R:
30|                         bool=False
31|                         break
32|         if bool :
33|             Triangulation.append(T)
34|     return(Triangulation)
35|
36|
37| ## Affichage ##
38|
39| from math import *
40| import time
41| import random as r
42| import copy as c
43| import matplotlib.pyplot as plt
44| import matplotlib.image as mpimg
45| import numpy as np
46| from itertools import combinations
47|
48|
49| points = [[r.randint(-500,500)/100,r.randint(-500,500)/100] for _ in range(50)]
50|
51| #poly = [[-1.5,1.94],[-2.46,-4.3],[3.18,-1.8],[5.7,-6.84],[10,-4],[4.9,-1.56],
[12,-1],[5.2,0.42],[11.86,2.94],[5,2],[4.62,4.04],[3.92,2.28],[1.42,2.48],
[-0.76,3.72],[0,1.56],[3.28,0.92],[1.82,-1]]
52|
53| poly = [[-3.37,15.03],[-9.44,11.54],[-4.31,11.75],[-8.45,6.59],[-9.61,2.98],
[-16.96,-1.94],[-12.48,-0.72],[-6.99,-4.09],[0.86,0.86],[-2.17,6.39],[3.57,3.73],
[3.07,-2.09],[0.78,-5.25],[5.94,-2.26],[7.13,4.71],[12.76,-2.59],[17.08,3.52],
[11.37,2.18],[26.93,13.4],[6.48,6.39],[2.78,5.68],[-0.55,8.51],[8.6,15.7],
[14.79,12.7],[16.79,14.95],[12.46,16.65],[1.2,16.4],[5.44,15.32],[2.03,15.28],
[-6.84,1.19],[-6.02,4.49],[-0.71,15.24],[-2.29,16.86]]
54|
55|
56| f = plt.figure()
57|
58| f.add_subplot(1,2,1)
59|
```

```
60| l_triangles = Delaunay(points)
61| for T in l_triangles:
62|     plt.plot(np.array(list(T)+[T[0]])[:,0],np.array(list(T)+[T[0]])[:,
1],color='black')
63|
64|
65| f.add_subplot(1,2,2)
66|
67| l_triangles=Delaunay(poly)
68| for T in l_triangles:
69|     plt.plot(np.array(list(T)+[T[0]])[:,0],np.array(list(T)+[T[0]])[:,
1],color='black')
70|
71| agrandir2()
72| plt.show()
73|
```

2.2 - Tracé des cercles circonscrits.py

```
01|
02| ## Tracé des cercles circonscrits ##
03|
04| import matplotlib.patches as patches
05|
06| poly = [[-1.5,1.94],[-2.46,-4.3],[3.18,-1.8],[5.7,-6.84],[10,-4],[4.9,-1.56],
[12,-1],[5.2,0.42],[11.86,2.94],[5,2],[4.62,4.04],[3.92,2.28],[1.42,2.48],
[-0.76,3.72],[0,1.56],[3.28,0.92],[1.82,-1]]
07|
08|
09| def Delaunay2(ListePoints):
10|     ListeTriangle = list(combinations(ListePoints,3))
11|     Triangulation=[]
12|     Liste_C_R=[]
13|     for T in ListeTriangle:
14|         bool=True
15|         ListD=[]
16|
17|         A1=2*(T[1][0]-T[0][0])
18|         B1=2*(T[1][1]-T[0][1])
19|         C1=T[0][0]**2 + T[0][1]**2 - T[1][0]**2 - T[1][1]**2
20|
21|         A2=2*(T[2][0]-T[1][0])
22|         B2=2*(T[2][1]-T[1][1])
23|         C2=T[1][0]**2 + T[1][1]**2 - T[2][0]**2 - T[2][1]**2
24|
25|         X=(C1*B2-C2*B1) / (A2*B1-A1*B2)
26|         Y=(C1*A2-A1*C2) / (A1*B2-B1*A2)
27|
28|         R=(X-T[1][0])**2 + (Y-T[1][1])**2
29|
30|         for P in ListePoints:
31|             if not(P in T):
32|                 distance=(P[0]-X)**2+(P[1]-Y)**2
33|                 ListD.append(distance)
34|                 for D in ListD:
35|                     if D<=R:
36|                         bool=False
37|                         break
38|         if bool :
39|             C=(np.round(X,2),np.round(Y,2))
40|             RR=np.round(np.sqrt(R),2)
41|             Triangulation.append(T)
42|             Liste_C_R.append([C,RR])
43|     return(Triangulation,Liste_C_R)
44|
45|
46| f,ax= plt.subplots()
47|
48| T,L=Delaunay2(poly)
49| n=len(L)
50| for i in range(n):
51|     plt.plot(np.array(list(T[i])+[T[i][0]])[:,0],np.array(list(T[i])+[T[i][0]])[:,
1],color='black')
52|     ax.add_patch(plt.Circle(L[i][0],radius=L[i][1],fc='None',ec='blue'))
53|
54| agrandir1()
55| plt.show()
56|
```

3.1 - Parcours en largeur (BFS).py

```
01|
02| ## Variables globales ##
03|
04| cmax = 100
05| deplacement_diag=True       # vaut True si on autorise les déplacements diagonaux,
False sinon.
06|
07|
08| ## Parcours en largeur (ou Breadth First Search) ##
09|
10| def voisins(M,case,bool):
11|     # Retourne la liste vf telle que vf=[(v,d),...] où v est une case voisine et d
est un booléen qui vaut True lorsque le voisin est en diagonal
12|     # Le booléen permet d'ajouter ou non les cases diagonales dans les voisins
13|     i,j=case
14|     v=[(i-1,j),(i,j+1),(i+1,j),(i,j-1)]                # haut, droite, bas,
gauche
15|     l_cout=[cout(M,v[k]) for k in range(4)]
16|     vf=[]
17|     for k in range(4):
18|         if l_cout[k]!=cmax:
19|             vf.append((v[k],False))
20|     if bool:
21|         vd=[(i-1,j+1),(i+1,j+1),(i+1,j-1),(i-1,j-1)]    # On commence en haut à
droite...
22|         for k in range(4):
23|             if l_cout[k]!=cmax and l_cout[(k+1)%4]!=cmax and cout(M,vd[k])!=cmax:
24|                 vf.append((vd[k],True))
25|     return(vf)
26|
27|
28| def parcours_en_largeur(M,depart,arrivee):
29|     # depart et arrivee sous la forme d'un tuple
30|     p,q=np.shape(M)[:2]
31|     file=[depart]
32|     C=np.array([["blanc" for _ in range(q)] for _ in range(p)])
33|     P=np.array([[None for _ in range(q)] for _ in range(p)])
34|
35|     P[depart]=(-1,-1)
36|     while file!=[]:
37|         pivot=file.pop(0)
38|         C[pivot]="noir"
39|         if pivot==arrivee:
40|             break
41|         for (v,_) in voisins(M,pivot,deplacement_diag):
42|             if C[v]== "blanc":
43|                 file.append(v)
44|                 P[v]=pivot
45|                 C[v]="gris"
46|     return(P)
47|
48|
49| def PCC_BFS(M,depart,arrivee):
50|     P=parcours_en_largeur(M,depart,arrivee)
51|     x=arrivee
52|     chemin=[x]
53|     while x != depart:
54|         x=P[x]
55|         chemin = [x] + chemin
56|     return(chemin)
57|
```

3.2 - Parcours en largeur glouton (GBFS).py

```python

## Fonctions annexes ##

def distance_2points(case1,case2):
    #return(abs(case2[0]-case1[0])+abs(case2[1]-case1[1]))
    return(np.round(sqrt((case2[0]-case1[0])**2 + (case2[1]-case1[1])**2),3))


def ajouter_file(file,case,distance):
    b=len(file)-1
    a=0
    while a<=b:
        m=(a+b)//2
        if case==file[m][0]:
            if distance<file[m][1]:
                file.pop(m)
                b=m-2
            else:
                return(file)
        elif distance<file[m][1]:
            b=m-1
        else:
            a=m+1
    file=file[:a]+[(case,distance)]+file[a:]
    return(file)


## Parcours en largeur glouton (GBFS) ##

def GBFS(M,depart,arrivee):
    p,q=np.shape(M)[:2]
    file=[(depart,-1)]
    C=np.array([["blanc" for _ in range(q)] for _ in range(p)])
    P=np.array([[None for _ in range(q)] for _ in range(p)])

    P[depart]=(-1,-1)
    while file!=[]:
        pivot=file.pop(0)[0]
        C[pivot]="noir"
        if pivot==arrivee:
            break
        for (v,_) in voisins(M,pivot,deplacement_diag):
            if C[v]== "blanc":
                P[v]=pivot
                d=distance_2points(v,arrivee)
                file=ajouter_file(file,v,d)
                C[v]="gris"
    return(P)


def PCC_GBFS(M,depart,arrivee):
    P=GBFS(M,depart,arrivee)
    x=arrivee
    chemin=[x]
    while x != depart:
        x=P[x]
        chemin = [x] + chemin
    return(chemin)

```

3.3 - Dijkstra.py

```
01|
02| ## Première fonction de coût ##
03|
04| def cout(M,case):
05|     # Fonction à préciser pour chaque exemple, en fonction de M.
06|     moyenne=np.mean(M[case])
07|     if moyenne==0:
08|         return(cmax)
09|     elif moyenne==127:
10|         return(4)
11|     else:
12|         return(1)
13|
14|
15| ## Dijkstra ##
16|
17| def Dijkstra(M,depart,arrivee):
18|     p,q=np.shape(M)[:2]
19|     file=[(depart,0)]
20|     C=np.array([["blanc" for _ in range(q)] for _ in range(p)])
21|     P=np.array([[None for _ in range(q)] for _ in range(p)])
22|     D=np.array([[np.infty for _ in range(q)] for _ in range(p)])
23|
24|     P[depart]=(-1,-1)
25|     D[depart]=0
26|     while file!=[]:
27|         pivot=file.pop(0)[0]
28|         C[pivot]="noir"
29|         if pivot==arrivee:
30|             break
31|         for (v,d) in voisins(M,pivot,deplacement_diag):
32|             # La distance de v par rapport au pivot est multiplié par sqrt(2) si v
est en diagonale
33|             if d:
34|                 newD=np.round(D[pivot]+cout(M,v)*sqrt(2),2)
35|             else:
36|                 newD=np.round(D[pivot]+cout(M,v),2)
37|             if C[v]=="blanc" and newD < D[v]:
38|                 D[v]=newD
39|                 P[v]=pivot
40|                 file=ajouter_file(file,v,newD)
41|     return(P)
42|
43|
44| def PCC_Dijkstra(M,depart,arrivee):
45|     P=Dijkstra(M,depart,arrivee)
46|     x=arrivee
47|     chemin=[x]
48|     while x != depart:
49|         x=P[x]
50|         chemin = [x] + chemin
51|     return(chemin)
52|
```

```
# 3.4 - Astar.py

01|
02| ## A* ##
03|
04| def Astar(M,depart,arrivee):
05|     p,q=np.shape(M)[:2]
06|     file=[(depart,0)]
07|
08|     C=np.array([["blanc" for _ in range(q)] for _ in range(p)])
09|     P=np.array([[None for _ in range(q)] for _ in range(p)])
10|     D=np.array([[np.infty for _ in range(q)] for _ in range(p)])
11|
12|     P[depart]=(-1,-1)
13|     D[depart]=0
14|     while file!=[]:
15|         pivot=file.pop(0)[0]
16|         C[pivot]="noir"
17|         if pivot==arrivee:
18|             break
19|         for (v,d) in voisins(M,pivot,deplacement_diag):
20|             if d:
21|                 newD=np.round(D[pivot]+cout(M,v)*sqrt(2),2)
22|             else:
23|                 newD=np.round(D[pivot]+cout(M,v),2)
24|             if C[v]=="blanc" and newD < D[v]:
25|                 D[v]=newD
26|                 P[v]=pivot
27|                 d=np.round(newD + distance_2points(v,arrivee),2)
28|                 file=ajouter_file(file,v,d)
29|     return(P)
30|
31|
32| def PCC_Astar(M,depart,arrivee):
33|     P=Astar(M,depart,arrivee)
34|     x=arrivee
35|     chemin=[x]
36|     while x != depart:
37|         x=P[x]
38|         chemin = [x] + chemin
39|     return(chemin)
40|
41|
42| ## Affichage d'un exemple (pour Dijkstra et A*) ##
43|
44| a = np.array([[0,0,1,0.5,0.5,1,1,1,0,0],
45|               [0,0,1,0.5,0.5,0.5,0.5,1,1,1],
46|               [0,1,1,0.5,0,0.5,0.5,0.5,1,1],
47|               [1,1,1,0.5,0.5,0.5,0.5,0.5,0.5,1],
48|               [1,1,1,0.5,0.5,0.5,0.5,0,1,1],
49|               [1,1,1,0.5,0.5,0,0.5,0.5,1,1],
50|               [1,1,1,0.5,0.5,0.5,0.5,0.5,1,1],
51|               [0,0,1,1,0.5,1,1,1,1,0],
52|               [0,1,1,1,1,1,1,1,0,0],
53|               [0,0,0,1,1,0,1,1,1,0]])
54| depart=(4,1)
55| arrivee=(2,10)
56|
57|
58| def mur_ext(M):
59|     # Ajoute un mur extérieur et supprime ainsi les problèmes de sorti de la
matrice de l'algorithme
60|     p,q=np.shape(M)[:2]
61|     new_M=np.array([[[0,0,0] for _ in range(q+2)] for _ in range(p+2)])
62|     new_M[1:p+1,1:q+1]=M[:,:]
63|     return(new_M)
64|
65| M=mur_ext(convertir(a))
```

```
66| M[depart]=[40, 180, 99]
67| M[arrivee]=[235, 0, 0]
68|
69|
70| f = plt.figure()
71| f.add_subplot(1,2,1)
72|
73| chemin_Dijkstra=PCC_Dijkstra(M,depart,arrivee)
74|
75| plt.plot(np.array(chemin_Dijkstra)[:,1],np.array(chemin_Dijkstra)[:,
0],color='blue',linewidth=5)
76| plt.title("Dijkstra",fontsize=25)
77|
78| plt.imshow(M)
79|
80|
81| f.add_subplot(1,2,2)
82|
83| chemin_Astar=PCC_Astar(M,depart,arrivee)
84|
85| plt.plot(np.array(chemin_Astar)[:,1],np.array(chemin_Astar)[:,
0],color='blue',linewidth=5)
86| plt.title("A*",fontsize=25)
87|
88| agrandir2()
89| plt.imshow(M)
90| plt.show()
91|
```

```python
# 4.1 - Chasse au trésor.py

## Chasse au trésor ##

p,q=100,100    # Dimensions de la carte
deplacement_diag=True

img=mur_ext(255*np.ones((p,q,3),dtype=int))
depart=(33,23)
arrivee=(p,q-18)

c1=[224,205,169]        # Sable
c2=[53, 194, 53]        # Herbe
c3=[37, 77, 213]        # Eau
blanc=[255,255,255]     # Piège
noir=[0,0,0]            # Mur


## Les éléments décoratifs ##

link = 'C:/Colin/Cours/3ème année/TIPE/Images/'
image = 'Maison'

img0 = mpimg.imread (link + image + '.png')
maison = np.array(255*img0[:,:,:3],dtype=int)

n1,n2=np.shape(maison)[:2]              # Maison pixelisée
for i in range(n1):
    for j in range(n2):
        if np.mean(maison[i,j])>222:
            maison[i,j]=c2

link = 'C:/Colin/Cours/3ème année/TIPE/Images/'
image = 'coffre'

img0 = mpimg.imread (link + image + '.png')
coffre = np.array(255*img0[:,:,:3],dtype=int)

n1,n2=np.shape(coffre)[:2]              # Coffre au trésor pixelisé
for i in range(n1):
    for j in range(n2):
        if np.mean(coffre[i,j])>200:
            coffre[i,j]=c1


## Initialisation de la carte aléatoire ##

def mur_ext_False(M):
    # Retourne une matrice de booléen où les coefficients extérieurs sont False, et les coefficients intérieurs sont ceux de M
    new_M=np.array([[False for _ in range(q+2)] for _ in range(p+2)])
    new_M[1:p+1,1:q+1]=M[:,:]
    return(new_M)


def voisins_directs(case,bool):
    i,j=case
    l=[(i-1,j),(i,j+1),(i+1,j),(i,j-1)]
    if bool: return(l+[(i-1,j+1),(i+1,j-1),(i+1,j+1),(i-1,j-1)])
    else: return(l)


def applique_carre(M,B,case,couleur,k):
    # Fonction récursive qui applique sur k épaisseurs autour de la case la couleur souhaitée
    if k==0:
        return(M)
```

```
    else:
        M[case]=couleur
        l=voisins_directs(case,deplacement_diag)
        for (ii,jj) in l:
            if B[ii,jj]:
                M=applique_carre(M,B,(ii,jj),couleur,k-1)
        return(M)


def cout(M,case):
    moyenne=np.mean(M[case])
    if np.all(M[case]==c1):          # Sable
        return(2)
    elif np.all(M[case]==c2):        # Herbe
        return(1)
    elif np.all(M[case]==c3):        # Eau
        return(5)
    elif np.all(M[case]==blanc):     # Piège
        return(10)
    else:
        return(cmax)


def initialise(M,P,Pherbe,Peau):
    # Peau + Herbe + Ppiege = 1 et P = la proba que la case soit un obstacle
    M[1:p+1,1:q+1]=c1
    M[1:37,1:35]=c2
    M[p-16:p+1,q-18:q+1]=c1
    M[1:33,1:32]=maison
    M[p-15:p+1,q-17:q+1]=coffre
    B=mur_ext_False(np.array([[True for _ in range(q)] for _ in range(p)]))
    B[:36,:34]=False
    B[p-16:,q-18:]=False

    for i in range(1,p+1):
        for j in range(1,q+1):
            if B[i,j]:
                X=r.random()
                if X<=P:
                    P1=Pherbe
                    P2=P1+Peau
                    Y=r.random()
                    if Y<=P1:                                    # Herbe
                        M=applique_carre(M,B,(i,j),c2,4)
                    elif (P1<Y) and (Y<=P2):                     # Eau
                        M=applique_carre(M,B,(i,j),c3,5)
                    else:                                        # Piege
                        M=applique_carre(M,B,(i,j),blanc,1)

    M[1:round((2/3)*p),round((1/3)*q)+1]=noir
    M[round((1/3)*p):p+1,round((2/3)*q)+2]=noir
    pas=7
    M[33,69-pas:69+pas+1]=noir
    M[66,34-pas:34+pas+1]=noir
    return(M)

A=c.deepcopy(img)
A=initialise(A,0.2,0.07,0.05)

plt.imshow(A)
agrandir1()
plt.show()


## Tracé des chemins obtenus ##

plt.imshow(A)

```

```
133| chemin_BFS=PCC_BFS(A,depart,arrivee)
134| chemin_GBFS=PCC_GBFS(A,depart,arrivee)
135| chemin_Dijkstra=PCC_Dijkstra(A,depart,arrivee)
136| chemin_Astar=PCC_Astar(A,depart,arrivee)
137|
138| plt.plot(np.array(chemin_BFS)[:,1],np.array(chemin_BFS)[:,
0],color='magenta',linewidth=4)
139| plt.plot(np.array(chemin_GBFS)[:,1],np.array(chemin_GBFS)[:,
0],color='orange',linewidth=4)
140| plt.plot(np.array(chemin_Dijkstra)[:,1],np.array(chemin_Dijkstra)[:,
0],color='yellow',linewidth=4)
141| plt.plot(np.array(chemin_Astar)[:,1],np.array(chemin_Astar)[:,
0],color='red',linewidth=4)
142|
143| plt.plot(depart[1],depart[0],color="g",marker='o',markersize=4)
144| plt.plot(arrivee[1],arrivee[0],color='r',marker='o',markersize=4)
145|
146| agrandir1()
147| plt.show()
148|
```

```
# 4.2 - Ville de Paris.py

001|
002| ## Ville de Paris (avec quelques embouteillages) ##
003|
004| def voisins_directs(case,bool):
005|     i,j=case
006|     l=[(i-1,j),(i,j+1),(i+1,j),(i,j-1)]
007|     if bool: return(l+[(i-1,j+1),(i+1,j-1),(i+1,j+1),(i-1,j-1)])
008|     else: return(l)
009|
010|
011| def inter_cercle(M,centre,rayon,couleur):
012|     # Trace un disque sur M en choisissant un centre, un rayon, et une couleur
souhaités.
013|     cx,cy=centre
014|     tab_case=[[(i,j) for i in range(round(cx-rayon),round(cx+rayon)+1)] for j in
range(round(cy-rayon),round(cy+rayon)+1)]
015|     for l_case in tab_case:
016|         for case in l_case:
017|             if distance_2points(case,centre)<=rayon and np.mean(M[case])>=220:
018|                 M[case]=couleur
019|
020|
021| def inter_rectangle(M,coin,theta,largeur,longueur,couleur):
022|     # Trace un rectangle plein sur M en choisissant un coin inférieur droit, une
inclinaison thêta, une longueur, une largeur et une couleur souhaités.
023|     cx,cy=coin
024|     tab_case=[]
025|     for i in range(cx-longueur,cx):
026|         for j in range(cy,cy+largeur):
027|             vec=vecteur((i,j),coin)
028|             norme=np.linalg.norm(vec)
029|             v1 = vec / norme
030|             v2 = [-1,0]
031|             produit = np.dot(v1,v2)
032|             angle = abs(np.arccos(produit))
033|             (i2,j2)=(round(coin[0]+norme*cos(theta+angle)),round(coin[1]
+norme*sin(theta+angle)))
034|             tab_case.append((i2,j2))
035|             tab_case += voisins_directs((i2,j2),False)
036|
037|     for case in tab_case:
038|         if np.mean(M[case])>=220:
039|             M[case]=couleur
040|
041|
042| c1=np.array([255,0,0])          # Rouge
043| c3=np.array([252,161,11])       # Orange
044| c4=np.array([241,235,54])       # Jaune
045|
046|
047| def cout(M,case):
048|     moyenne=np.mean(M[case])
049|     if np.all(M[case]==c1):     # Rouge
050|         return(8)
051|     elif np.all(M[case]==c3):   # Orange
052|         return(6)
053|     elif np.all(M[case]==c4):   # Jaune
054|         return(2)
055|     else:
056|         if moyenne<220:
057|             return(cmax)
058|         else:
059|             return(1)
060|
061|
062| ## Image de travail ##
```

```
063|
064| link = 'C:/Colin/Cours/3ème année/TIPE/Images/'
065| image = 'Paris'
066|
067| img0 = mpimg.imread (link + image + '.jpg')
068| img = mur_ext(np.array(img0,dtype=int))
069|
070| plt.imshow(img)
071|
072| depart=(219,699)
073| arrivee=(445,208)
074|
075| inter_cercle(img,(345,465),40,c1)
076| inter_cercle(img,(225,254),40,c1)
077| inter_cercle(img,(378,651),40,c1)
078|
079| inter_rectangle(img,(381,280),pi/4,52,106,c3)
080| inter_rectangle(img,(317,427),1.12,20,23,c1)
081| inter_rectangle(img,(317,401),1.12,33,65,c3)
082| inter_rectangle(img,(278,348),1.12,16,80,c3)
083|
084| for _ in range(20):     # Les embouteillages aléatoires en jaune
085|    a,b=r.randint(100,550),r.randint(100,700)
086|    inter_cercle(img,(a,b),50,c4)
087|
088| A=c.deepcopy(img)
089|
090| agrandir1()
091| plt.show()
092|
093|
094| ## Tracés des chemins obtenus ##
095|
096| plt.imshow(A)
097|
098| chemin_BFS=PCC_BFS(A,depart,arrivee)
099| chemin_GBFS=PCC_GBFS(A,depart,arrivee)
100| chemin_Dijkstra=PCC_Dijkstra(A,depart,arrivee)
101| chemin_Astar=PCC_Astar(A,depart,arrivee)
102|
103| plt.plot(np.array(chemin_BFS)[:,1],np.array(chemin_BFS)[:,
0],color='blue',linewidth=3)
104| plt.plot(np.array(chemin_GBFS)[:,1],np.array(chemin_GBFS)[:,
0],color='cyan',linewidth=3)
105| plt.plot(np.array(chemin_Dijkstra)[:,1],np.array(chemin_Dijkstra)[:,
0],color='magenta',linewidth=3)
106| plt.plot(np.array(chemin_Astar)[:,1],np.array(chemin_Astar)[:,
0],color='green',linewidth=3)
107|
108| plt.plot(depart[1],depart[0],color="lime",marker='o',markersize=10)
109| plt.plot(arrivee[1],arrivee[0],color='r',marker='o',markersize=10)
110|
111| agrandir1()
112| plt.show()
113|
```

```
# 4.3 - Polygone aléatoire.py

01|
02| ## Polygone aléatoire ##
03|
04| def cout(M,case):
05|     moyenne=np.mean(M[case])
06|     if moyenne<20:
07|         return(1)
08|     else:
09|         return(cmax)
10|
11|
12| ## Image de travail ##
13|
14| link = 'C:/Colin/Cours/3ème année/TIPE/Images/'
15| image = 'Polygone_cercle'
16|
17| plt.imshow(img)
18|
19| img0 = mpimg.imread (link + image + '.png')
20| img = np.array(img0[35:,40:960,:3]*255,dtype=int)
21| img[202,627]=[0,0,0]
22| img[82,642]=[0,0,0]
23| img[85,643]=[0,0,0]
24|
25| depart=(105,237)
26| arrivee=(465,458)
27|
28| plt.plot(depart[1],depart[0],color="lime",marker='o',markersize=10)
29| plt.plot(arrivee[1],arrivee[0],color='r',marker='o',markersize=10)
30|
31| A=c.deepcopy(img)
32|
33| agrandir1()
34| plt.show()
35|
36|
37| ## Tracés des chemins obtenus ##
38|
39| plt.imshow(A)
40|
41| chemin_BFS=PCC_BFS(A,depart,arrivee)
42| chemin_GBFS=PCC_GBFS(A,depart,arrivee)
43| chemin_Dijkstra=PCC_Dijkstra(A,depart,arrivee)
44| chemin_Astar=PCC_Astar(A,depart,arrivee)
45|
46| plt.plot(np.array(chemin_BFS)[:,1],np.array(chemin_BFS)[:,
0],color='red',linewidth=3,label="BFS")
47| plt.plot(np.array(chemin_GBFS)[:,1],np.array(chemin_GBFS)[:,
0],color='cyan',linewidth=3,label="GBFS")
48| plt.plot(np.array(chemin_Dijkstra)[:,1],np.array(chemin_Dijkstra)[:,
0],color='blue',ls='--',linewidth=3,label="Dijkstra")
49| plt.plot(np.array(chemin_Astar)[:,1],np.array(chemin_Astar)[:,
0],color='magenta',linewidth=3,label="A*")
50|
51| plt.plot(depart[1],depart[0],color="lime",marker='o',markersize=10)
52| plt.plot(arrivee[1],arrivee[0],color='r',marker='o',markersize=10)
53|
54| plt.legend(loc='lower right',fontsize=18)
55|
56| agrandir1()
57| plt.show()
58|
```

5.1 - Implémentation de WAstar.py

```
01|
02| ## WA* avec coloraiage des cases ##
03|
04| def WAstar(M,w,depart,arrivee):
05|     # ici, depart et arrivée sont sous la forme d'un tuple
06|     p,q=np.shape(M)[:2]
07|     file=[(depart,0)]
08|
09|     C=np.array([["blanc" for _ in range(q)] for _ in range(p)])
10|     P=np.array([[None for _ in range(q)] for _ in range(p)])
11|     D=np.array([[np.infty for _ in range(q)] for _ in range(p)])
12|
13|     P[depart]=(-1,-1)
14|     D[depart]=0
15|
16|     while file!=[]:
17|         pivot=file.pop(0)[0]
18|         C[pivot]="noir"
19|         if pivot==arrivee:
20|             break
21|         for (v,d) in voisins(M,pivot,deplacement_diag):
22|             if d:
23|                 newD=np.round(D[pivot]+cout(M,v)*sqrt(2),2)
24|             else:
25|                 newD=np.round(D[pivot]+cout(M,v),2)
26|             if C[v]=="blanc" and newD < D[v]:
27|                 D[v]=newD
28|                 P[v]=pivot
29|                 d=np.round(newD + w*distance_2points(v,arrivee),2)
30|                 file=ajouter_file(file,v,d)
31|     return(P)
32|
33|
34| def PCC_WAstar(M,w,depart,arrivee):
35|     P=WAstar(M,w,depart,arrivee)
36|     x=arrivee
37|     chemin=[x]
38|     while x != depart:
39|         x=P[x]
40|         chemin = [x] + chemin
41|     return(chemin)
42|
43|
44| ## Fonction très utile qui calcule le coût du chemin calculé ##
45|
46| def cout_chemin(M,chemin):
47|     n=len(chemin)
48|     res=0
49|     for k in range(n-1):
50|         (i,j)=chemin[k]
51|         (i2,j2)=chemin[k+1]
52|         if (i2,j2) in [(i-1,j+1),(i+1,j+1),(i+1,j-1),(i-1,j-1)]:
53|             res += cout(M,(i2,j2))*sqrt(2)
54|         else:
55|             res += cout(M,(i2,j2))
56|     return(np.round(res,2))
57|
```

5.2 - Comparaison aux autres algorithmes.py

```
001|
002| ## WA* avec coloraiage des cases ##
003|
004| def WAstar(M,w,depart,arrivee):
005|     # ici, depart et arrivée sont sous la forme d'un tuple
006|     p,q=np.shape(M)[:2]
007|     file=[(depart,0)]
008|
009|     C=np.array([["blanc" for _ in range(q)] for _ in range(p)])
010|     P=np.array([[None for _ in range(q)] for _ in range(p)])
011|     D=np.array([[np.infty for _ in range(q)] for _ in range(p)])
012|
013|     P[depart]=(-1,-1)
014|     D[depart]=0
015|
016|     while file!=[]:
017|         pivot=file.pop(0)[0]
018|         C[pivot]="noir"
019|         M[pivot]=[249, 173, 41]     # On colorie en jaune une case visitée
020|         if pivot==arrivee:
021|             break
022|         for (v,d) in voisins(M,pivot,deplacement_diag):
023|             if d:
024|                 newD=np.round(D[pivot]+cout(M,v)*sqrt(2),2)
025|             else:
026|                 newD=np.round(D[pivot]+cout(M,v),2)
027|             if C[v]=="blanc" and newD < D[v]:
028|                 D[v]=newD
029|                 P[v]=pivot
030|                 M[v]=[246,253,27]   # On colorie en orange une case en cours de
visite
031|                 d=np.round(newD + w*distance_2points(v,arrivee),2)
032|                 file=ajouter_file(file,v,d)
033|     return(P)
034|
035|
036| ## Matrice de travail ##
037|
038| cmax = 100
039| deplacement_diag=False
040|
041| a = np.array([[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1],
042|               [0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1],
043|               [1,1,1,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1],
044|               [1,1,1,1,1,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1],
045|               [1,1,1,1,1,1,0,0,1,1,1,1,1,1,1,0,0,0,1,1,1,1,1,1,1],
046|               [1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,0,0,0,1,1,1,1,1],
047|               [1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1],
048|               [1,1,1,1,1,1,1,0,0,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1],
049|               [1,1,1,1,1,1,1,1,0,0,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1],
050|               [1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1],
051|               [1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1],
052|               [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,1,1,1],
053|               [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,1,1],
054|               [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,1,1],
055|               [1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1],
056|               [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]])
057|
058| M=mur_ext(convertir(a))
059|
060| depart=(7,2)
061| arrivee=(8,20)
062|
063| def agrandir2():
064|     mngr = plt.get_current_fig_manager()
065|     mngr.window.setGeometry(200,50,1500,1000)
```

```
066|     f.set_tight_layout(True)
067|     plt.tight_layout(pad=0.5)
068|
069|
070| ## WA* pour w=0 /// Dijkstra ##
071|
072| M=mur_ext(convertir(a))
073|
074| f = plt.figure()
075| f.add_subplot(1,2,1)
076|
077| M1=c.deepcopy(M)
078|
079| chemin=PCC_Dijkstra2(M1,depart,arrivee)
080| print('Coût du chemin :',cout_chemin(M1,chemin))
081|
082| plt.plot(np.array(chemin)[:,1],np.array(chemin)[:,0],color='blue',linewidth=5)
083| plt.title("Dijkstra",fontsize=30)
084|
085| M1[depart]=[40, 180, 99]
086| M1[arrivee]=[235, 0, 0]
087| plt.imshow(M1)
088|
089|
090| f.add_subplot(1,2,2)
091|
092| M2=c.deepcopy(M)
093|
094| chemin=PCC_WAstar(M2,0,depart,arrivee)
095| print('Coût du chemin :',cout_chemin(M2,chemin))
096|
097| plt.plot(np.array(chemin)[:,1],np.array(chemin)[:,0],color='blue',linewidth=5)
098| plt.title("WA* pour w=0",fontsize=30)
099|
100| M2[depart]=[40, 180, 99]
101| M2[arrivee]=[235, 0, 0]
102| plt.imshow(M2)
103|
104|
105| agrandir2()
106| plt.show()
107|
108|
109| ## WA* pour w=1 /// A* ##
110|
111| M=mur_ext(convertir(a))
112|
113| f = plt.figure()
114| f.add_subplot(1,2,1)
115|
116| M1=c.deepcopy(M)
117|
118| chemin=PCC_Astar2(M1,depart,arrivee)
119| print('Coût du chemin :',cout_chemin(M1,chemin))
120|
121| plt.plot(np.array(chemin)[:,1],np.array(chemin)[:,0],color='blue',linewidth=5)
122| plt.title("A*",fontsize=30)
123|
124| M1[depart]=[40, 180, 99]
125| M1[arrivee]=[235, 0, 0]
126| plt.imshow(M1)
127|
128|
129| f.add_subplot(1,2,2)
130|
131| M2=c.deepcopy(M)
132|
133| chemin=PCC_WAstar(M2,1,depart,arrivee)
```

```
print('Coût du chemin :',cout_chemin(M2,chemin))

plt.plot(np.array(chemin)[:,1],np.array(chemin)[:,0],color='blue',linewidth=5)
plt.title("WA* pour w=1",fontsize=30)

M2[depart]=[40, 180, 99]
M2[arrivee]=[235, 0, 0]
plt.imshow(M2)


agrandir2()
plt.show()




## WA* pour w>>1 et GBFS

M=mur_ext(convertir(a))

f = plt.figure()
f.add_subplot(2,3,2)

M1=c.deepcopy(M)

chemin=PCC_GBFS2(M1,depart,arrivee)
print('Coût du chemin :',cout_chemin(M1,chemin))

plt.plot(np.array(chemin)[:,1],np.array(chemin)[:,0],color='blue',linewidth=5)
plt.title("GBFS",fontsize=30)

M1[depart]=[40, 180, 99]
M1[arrivee]=[235, 0, 0]
plt.imshow(M1)


f.add_subplot(2,3,4)

M2=c.deepcopy(M)

chemin=PCC_WAstar(M2,5,depart,arrivee)
print('Coût du chemin :',cout_chemin(M2,chemin))

plt.plot(np.array(chemin)[:,1],np.array(chemin)[:,0],color='blue',linewidth=5)
plt.title("WA* pour w=5 ",fontsize=30)

M2[depart]=[40, 180, 99]
M2[arrivee]=[235, 0, 0]
plt.imshow(M2)


f.add_subplot(2,3,5)

M3=c.deepcopy(M)

chemin=PCC_WAstar(M3,10,depart,arrivee)
print('Coût du chemin :',cout_chemin(M3,chemin))

plt.plot(np.array(chemin)[:,1],np.array(chemin)[:,0],color='blue',linewidth=5)
plt.title("WA* pour w=10 ",fontsize=30)

M3[depart]=[40, 180, 99]
M3[arrivee]=[235, 0, 0]
plt.imshow(M3)


f.add_subplot(2,3,6)
```

```
202|
203| M4=c.deepcopy(M)
204|
205| chemin=PCC_WAstar(M4,50,depart,arrivee)
206| print('Coût du chemin :',cout_chemin(M4,chemin))
207|
208| plt.plot(np.array(chemin)[:,1],np.array(chemin)[:,0],color='blue',linewidth=5)
209| plt.title("WA* pour w=50 ",fontsize=30)
210|
211| M4[depart]=[40, 180, 99]
212| M4[arrivee]=[235, 0, 0]
213| plt.imshow(M4)
214|
215|
216| agrandir2()
217| plt.show()
218|
```

```
# 5.3 - Exemple sur la ville de Paris.py

01|
02| ## WA* appliqué à la ville de Paris ##
03|
04| ## Image de travail ##
05|
06| link = 'C:/Colin/Cours/3ème année/TIPE/Images/'
07| image = 'Paris'
08|
09| img0 = mpimg.imread (link + image + '.jpg')
10| img = mur_ext(np.array(img0,dtype=int))
11|
12| depart=(116,465)
13| arrivee=(366,252)
14|
15| inter_cercle(img,(345,465),40,c1)
16| inter_cercle(img,(225,254),40,c1)
17| inter_cercle(img,(378,651),40,c1)
18|
19| inter_rectangle(img,(381,280),pi/4,52,106,c3)
20| inter_rectangle(img,(317,427),1.12,20,23,c1)
21| inter_rectangle(img,(317,401),1.12,33,65,c3)
22| inter_rectangle(img,(278,348),1.12,16,80,c3)
23|
24| for _ in range(20):
25|     a,b=r.randint(100,550),r.randint(100,700)
26|     inter_cercle(img,(a,b),50,c4)
27|
28| A=c.deepcopy(img)
29|
30| plt.imshow(img)
31|
32| agrandir1()
33| plt.show()
34|
35|
36| ## Tracé des différents chemins en fonction du choix de w ##
37|
38| chemin1=PCC_WAstar(A,0,depart,arrivee)
39| chemin2=PCC_WAstar(A,1,depart,arrivee)
40| chemin3=PCC_WAstar(A,2,depart,arrivee)
41| chemin4=PCC_WAstar(A,5,depart,arrivee)
42| chemin5=PCC_WAstar(A,10,depart,arrivee)
43| chemin6=PCC_WAstar(A,50,depart,arrivee)
44|
45| plt.plot(np.array(chemin6)[:,1],np.array(chemin6)[:,
0],color=('#2DF497'),linewidth=3,label="w=50")
46| plt.plot(np.array(chemin5)[:,1],np.array(chemin5)[:,
0],color=('#00FFF7'),linewidth=3,label="w=10")
47| plt.plot(np.array(chemin4)[:,1],np.array(chemin4)[:,
0],color=('#00AEFF'),linewidth=3,label="w=5")
48| plt.plot(np.array(chemin3)[:,1],np.array(chemin3)[:,
0],color=('#0065FF'),linewidth=3,label="w=2")
49| plt.plot(np.array(chemin2)[:,1],np.array(chemin2)[:,
0],color=('#3100CD'),linewidth=3,label="w=1")
50| plt.plot(np.array(chemin1)[:,1],np.array(chemin1)[:,
0],color=('#1A245B'),linewidth=3,label="w=0")
51|
52| plt.plot(depart[1],depart[0],color="lime",marker='o',markersize=10)
53| plt.plot(arrivee[1],arrivee[0],color='r',marker='o',markersize=10)
54| plt.legend(loc='lower right',fontsize=25,framealpha=1)
55|
56| plt.imshow(A)
57|
58| agrandir1()
59| plt.show()
60|
```