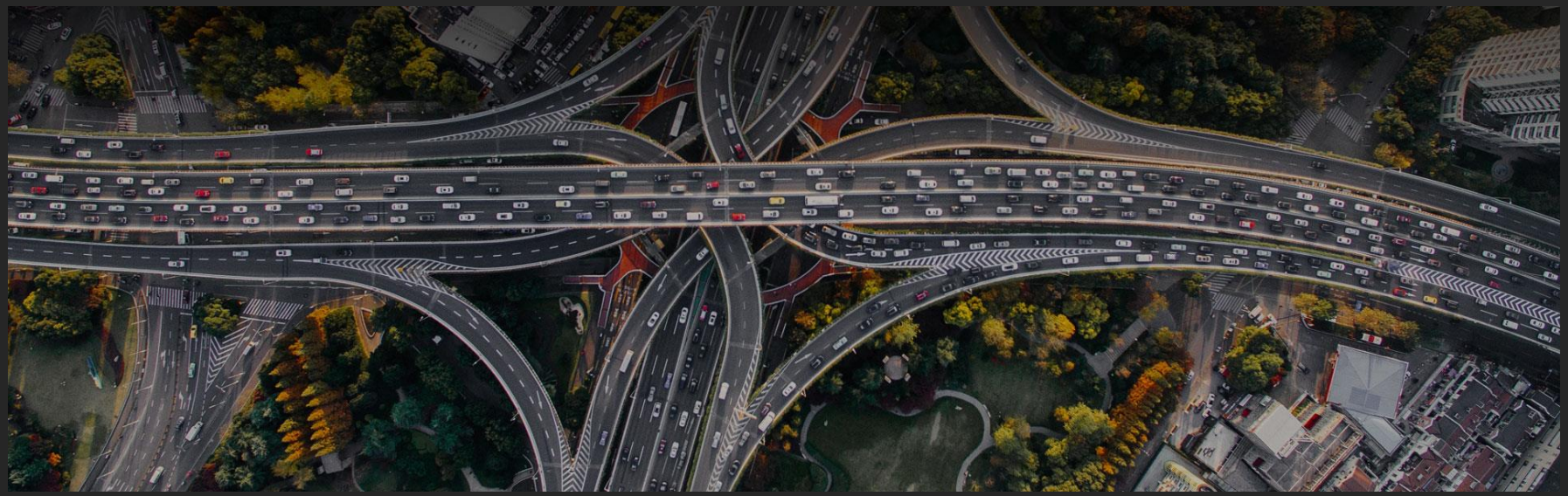
# Optimisation de trajectoires dans un environnement simple

COËRCHON Colin

Candidat n° 11759



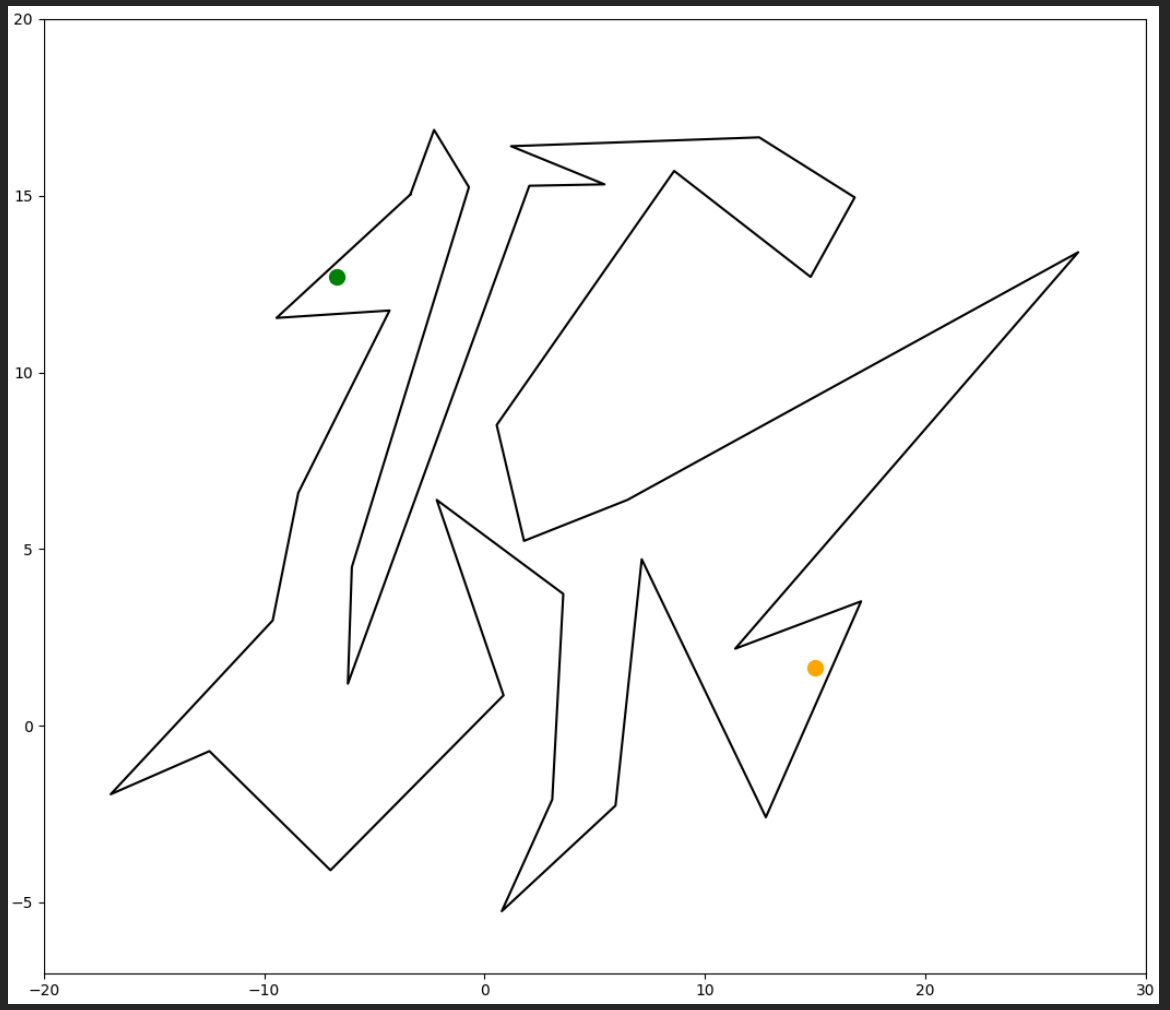
Session 2021/2022

# Sommaire

- Espace polygonal simple
  - Triangulation
  - Parcours en largeur
  - Dijkstra et A*
  - Analyse des résultats
- Quadrillage de l'espace
  - Algorithmes de parcours en largeur
  - Dijkstra et A*
  - Divers exemples
- Ouverture sur d'autres idées intéressantes

# I. Espace polygonal simple

- On se donne un polygone, un point de départ et un point d'arrivée
- Ces deux points sont intérieurs au polygone



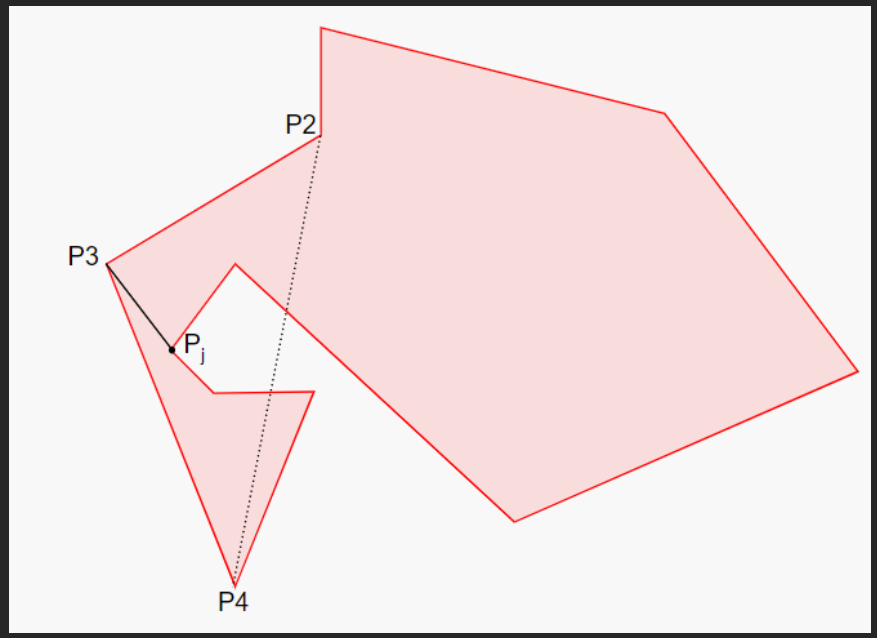
# Triangulation du polygone

- Tout polygone admet une triangulation Annexe 1
- On détermine ainsi une triangulation par un algorithme récursif sur l'ensemble des sommets du polygone.

On se donne ainsi un polygone $P$ tel que :

$$\mathcal{P} = (P_1, P_2, P_3, P_4, \cdots, P_j, \cdots, P_n)$$

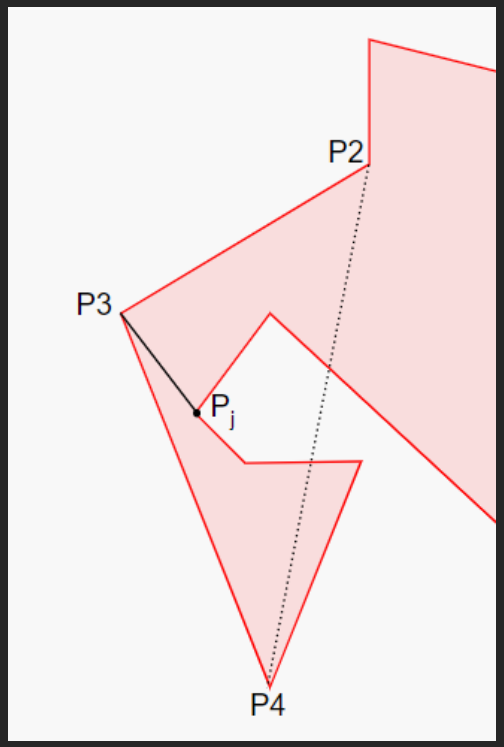
$$\mathcal{P}_1 = (P_3, P_4, \cdots, P_j) \quad \mathcal{P}_2 = (P_3, P_j, \cdots, P_2)$$



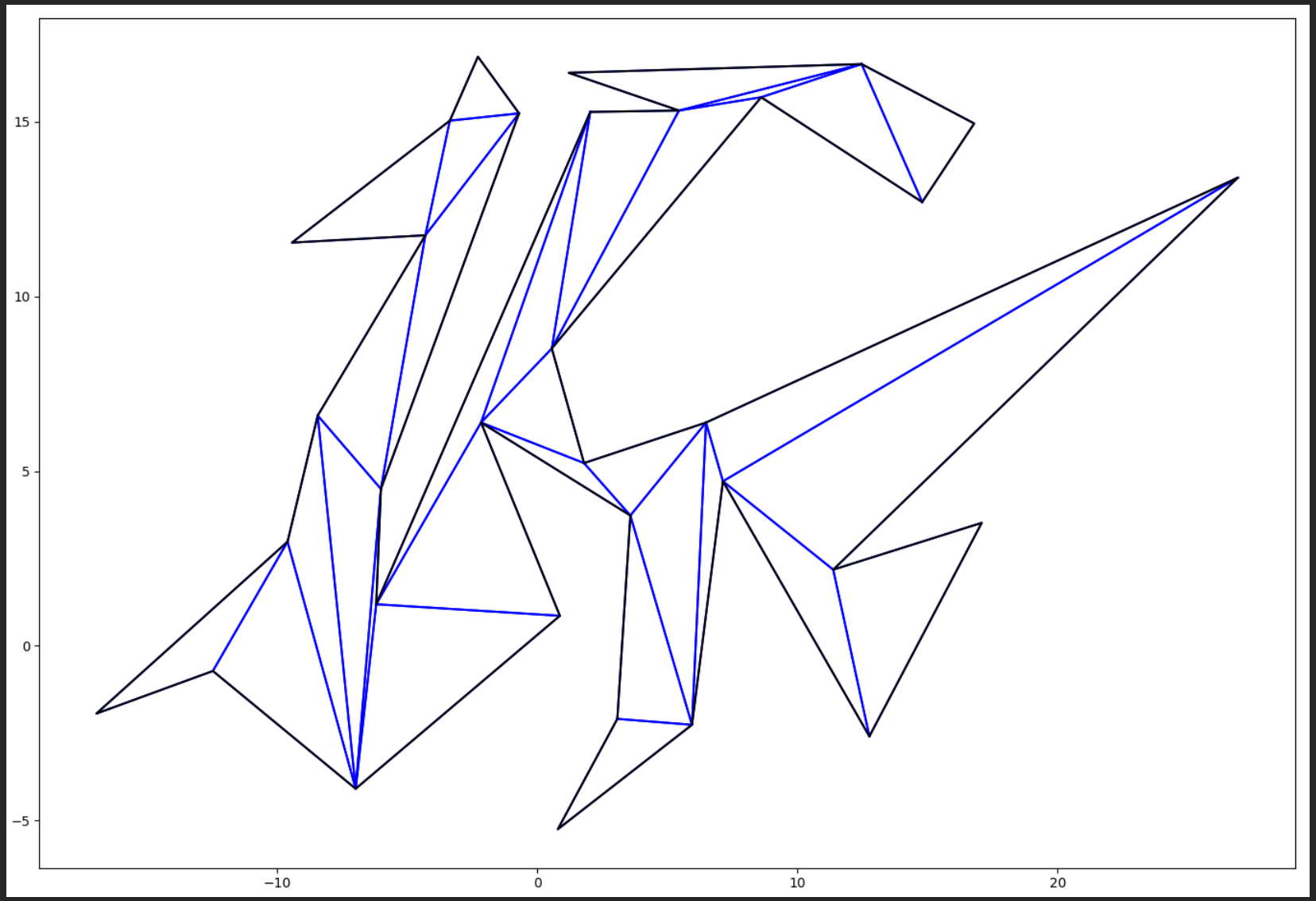
**Fonction** `Trianguler_rec` (Polygone, liste_triangles)

- `i` ≔ indice du sommet situé le plus à gauche
- T ≔ triangle formé par $(P_{i-1}, P_i, P_{i+1})$
- `j` ≔ indice du sommet dans T le plus loin de $(P_{i-1}P_{i+1})$
- **Si** `j` n'existe pas :
  - **Rajouter** T à liste_triangles
- **Sinon** :
  - Créer $\mathcal{P}_1$ et $\mathcal{P}_2$
  - **Si** taille ($\mathcal{P}_1$) = 3 :
    - **Ajouter** $\mathcal{P}_1$ à liste_triangles
  - **Sinon** :
    - `Trianguler_rec` ($\mathcal{P}_1$, liste_triangles)
  - **Si** taille ($\mathcal{P}_2$) = 3 :
    - **Ajouter** $\mathcal{P}_2$ à liste_triangles
  - **Sinon** :
    - `Trianguler_rec` ($\mathcal{P}_2$, liste_triangles)
- **Retourner** liste_triangles

**Fin**

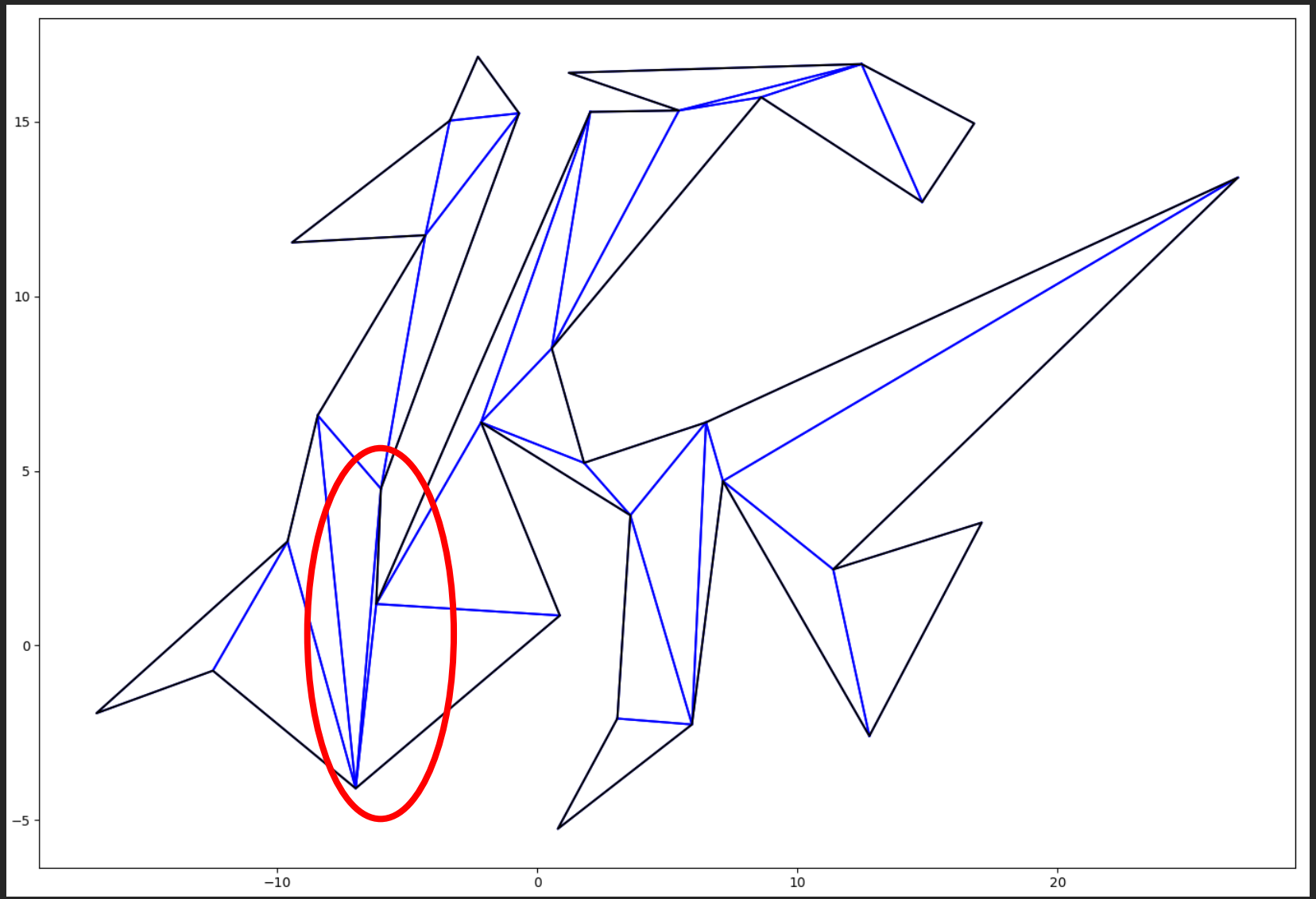


# Triangulation du polygone

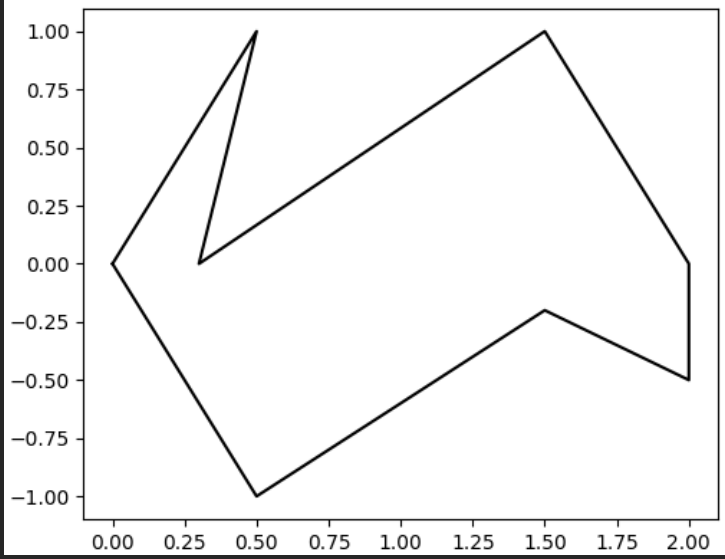


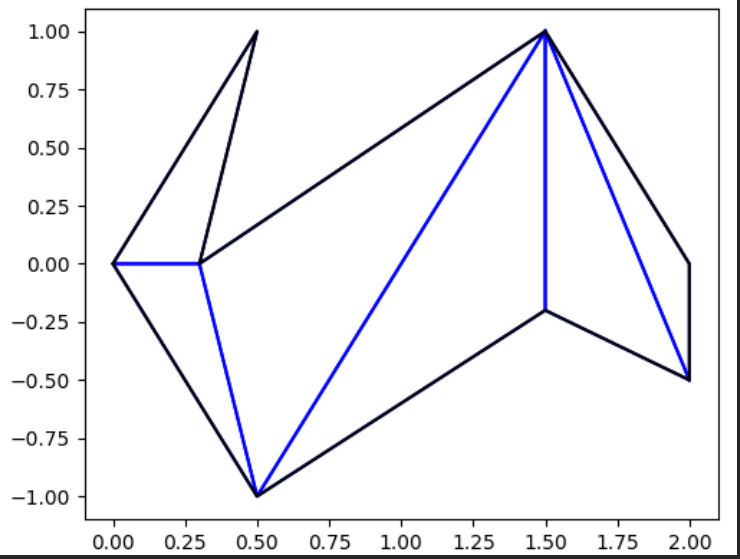
# Triangulation de Delaunay…

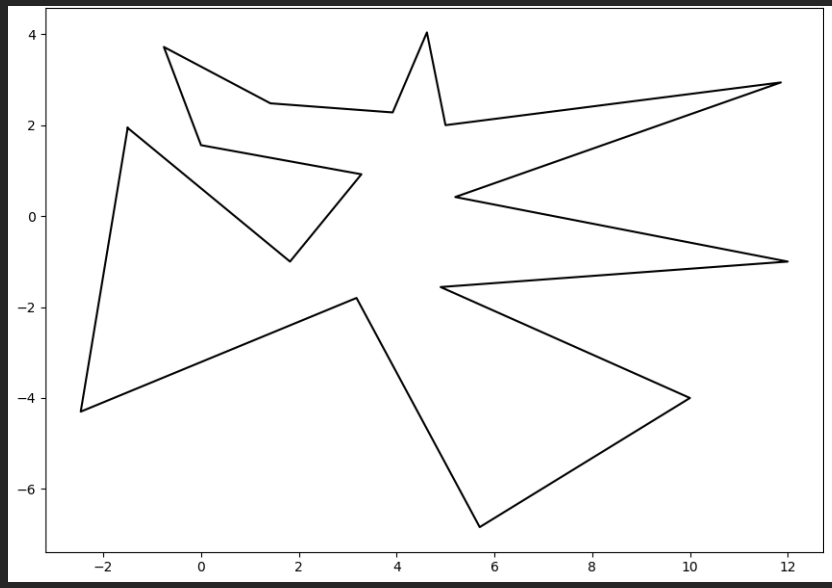
Annexe 2

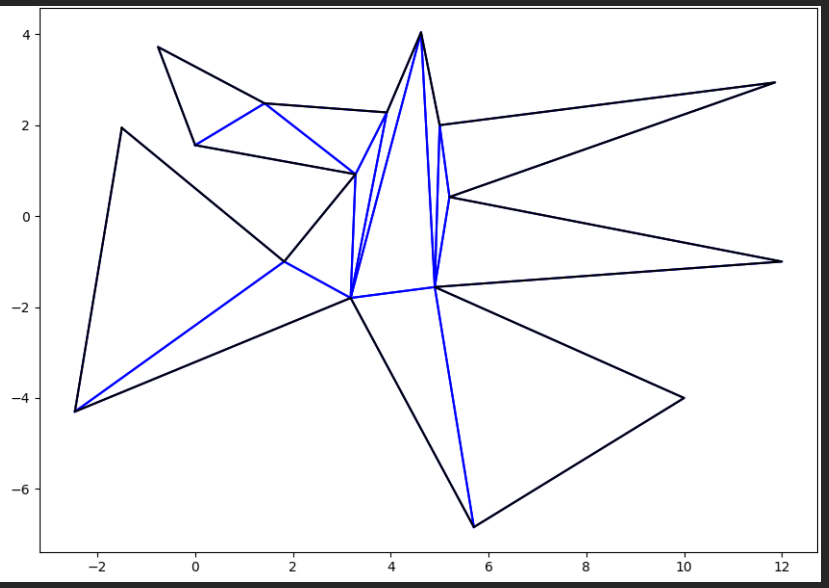


# D’autres exemples



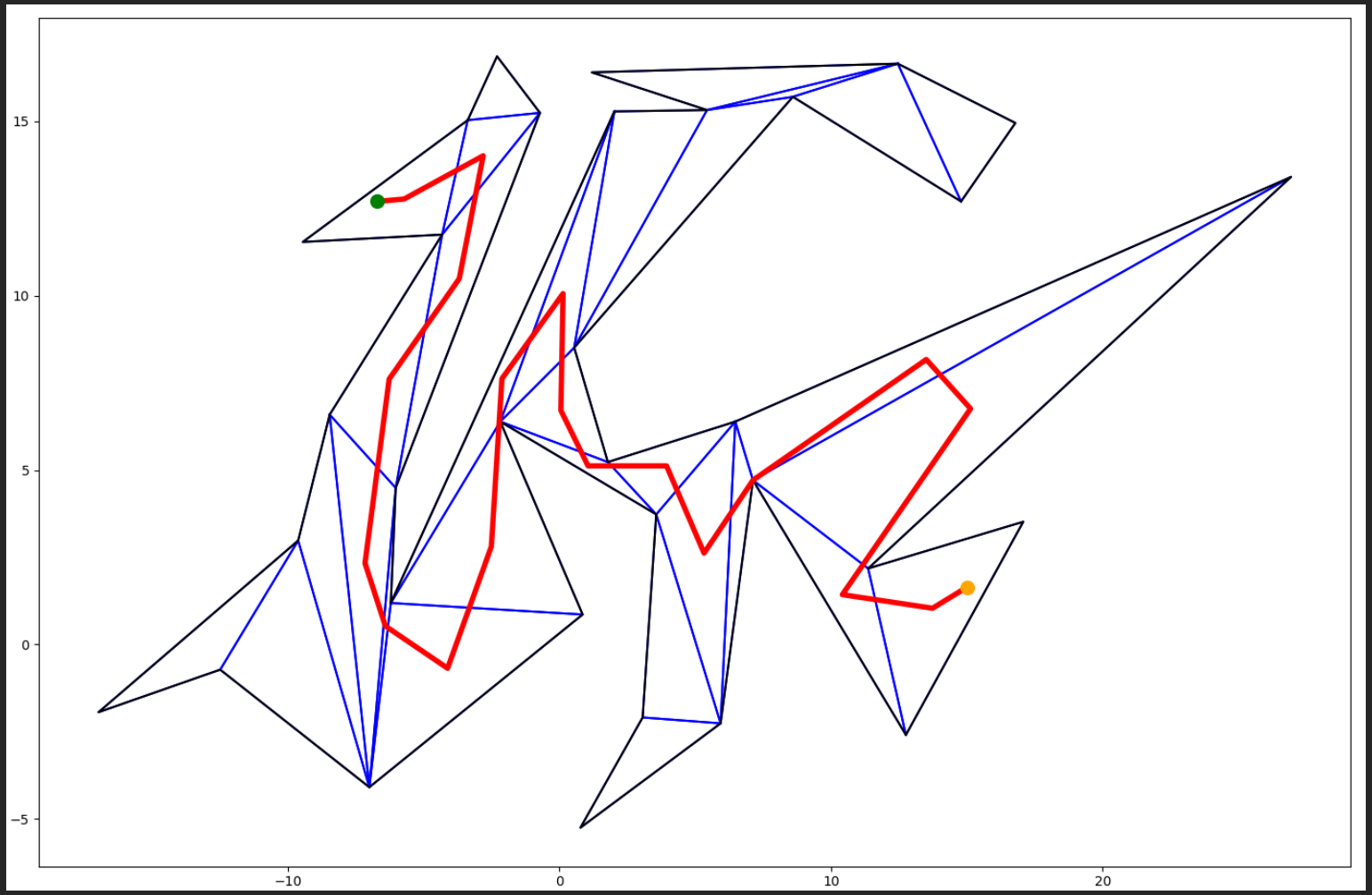



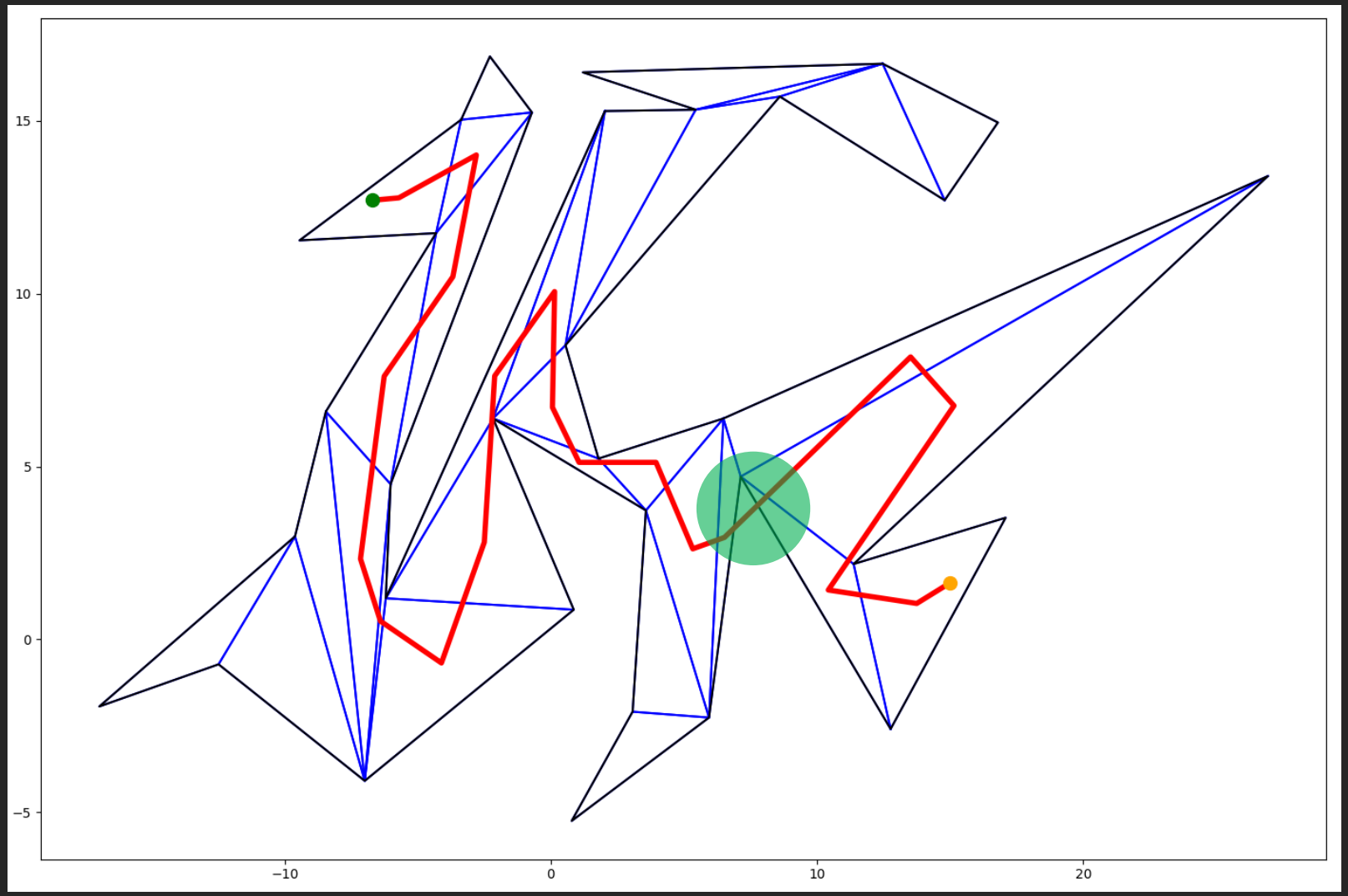
# Parcours en largeur

La fonction Triangulation nous donne une liste de triangles.

On effectue alors un parcours en largeur sur les triangles voisins.



# Petit problème…

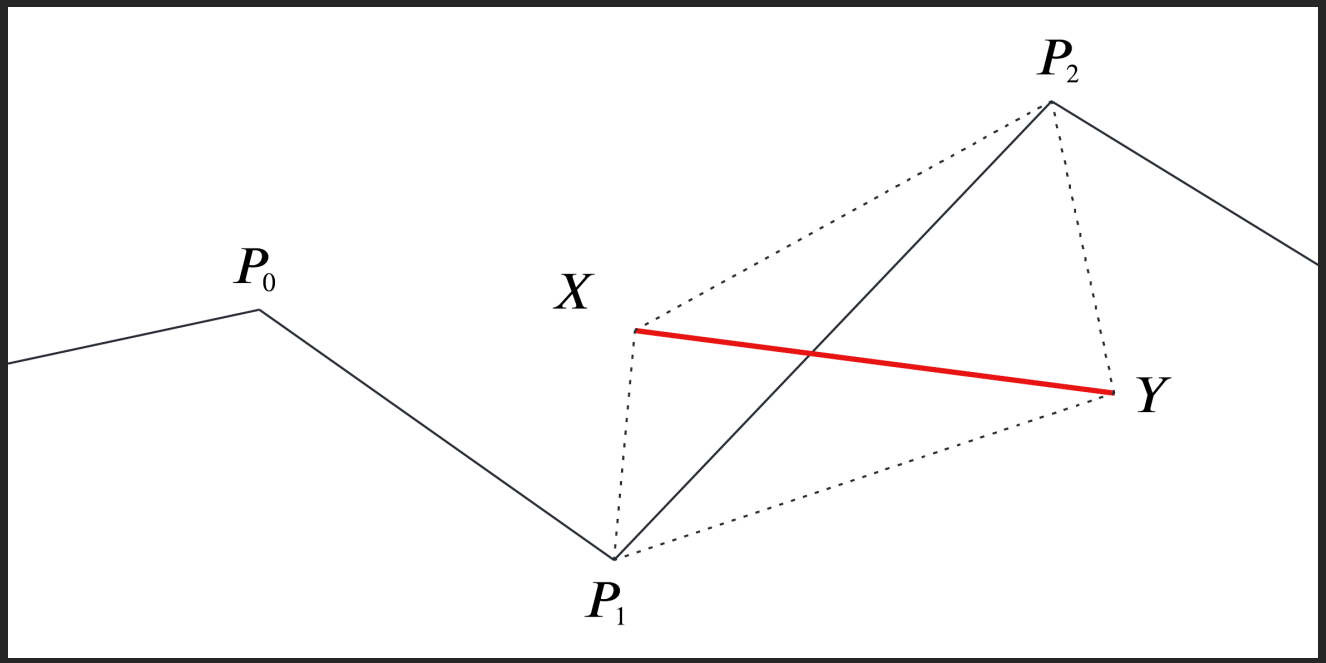


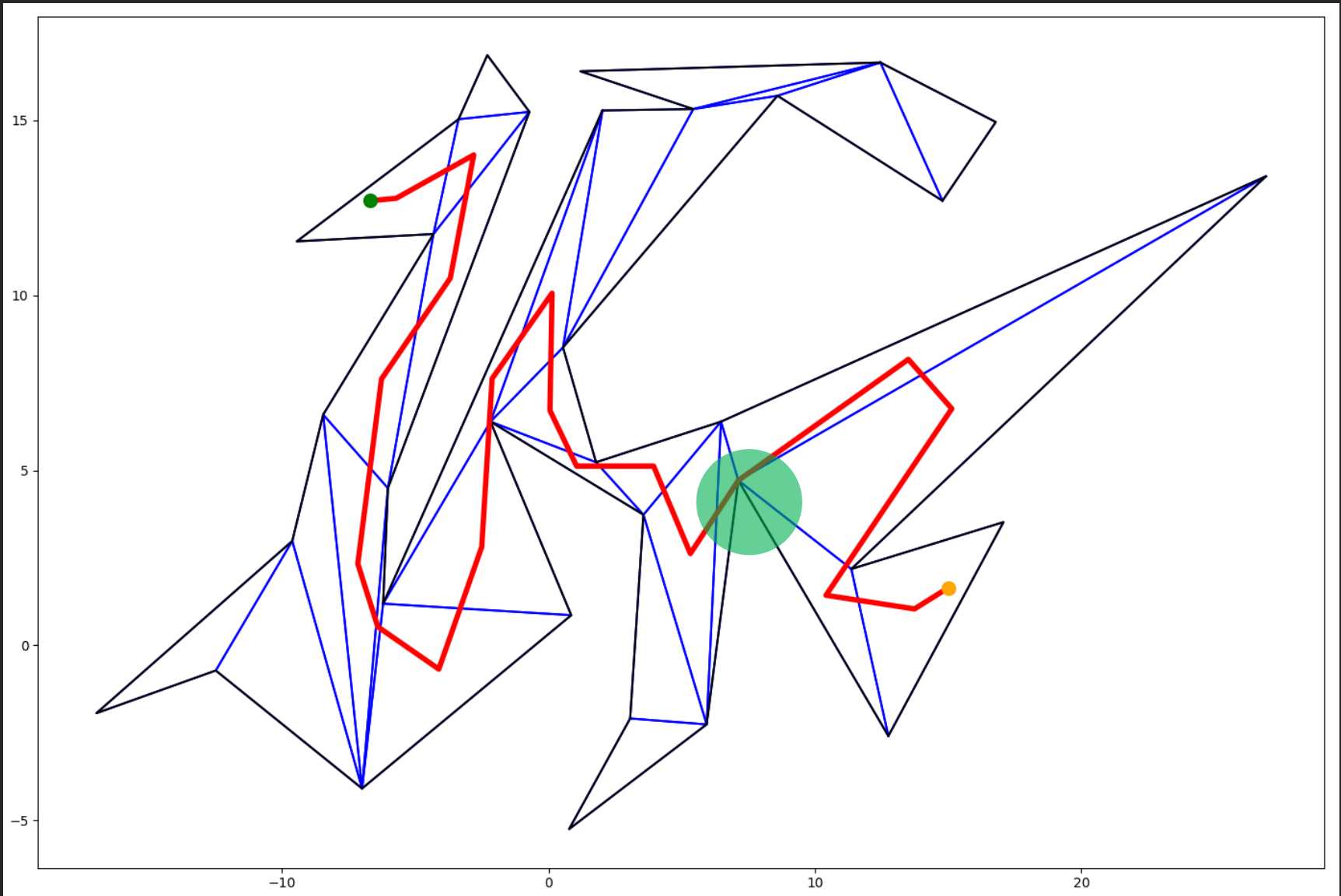

Résultat non corrigé

# Comment résoudre ce problème ?

Il est nécessaire pour cela de trouver une condition de croisement entre une section $[XY]$ du chemin, et la frontière du polygone $\mathcal{P}$.

$$[XY]\ coupe\ la\ frontière\ [P_1P_2] \overset{(*)}{\Leftrightarrow} \begin{cases} sign(\overrightarrow{P_1X} \wedge \overrightarrow{P_1P_2}) = sign(\overrightarrow{P_1X} \wedge \overrightarrow{P_1Y}) \\ sign(\overrightarrow{P_1Y} \wedge \overrightarrow{P_1P_2}) = sign(\overrightarrow{P_1Y} \wedge \overrightarrow{P_1X}) \\ sign(\overrightarrow{XY} \wedge \overrightarrow{XP_2}) \neq sign(\overrightarrow{XY} \wedge \overrightarrow{XP_2}) \end{cases}$$





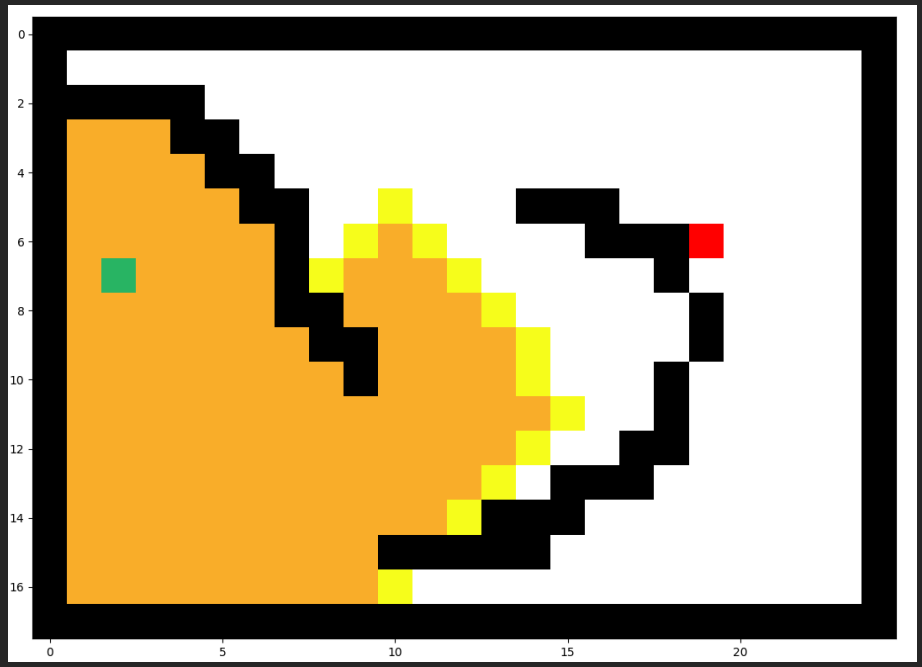

Résultat corrigé

# Algorithme de Dijkstra

- On s'intéresse maintenant aux sommets du polygone
- On utilise une matrice d'adjacence $G$ pour représenter ce graphe.

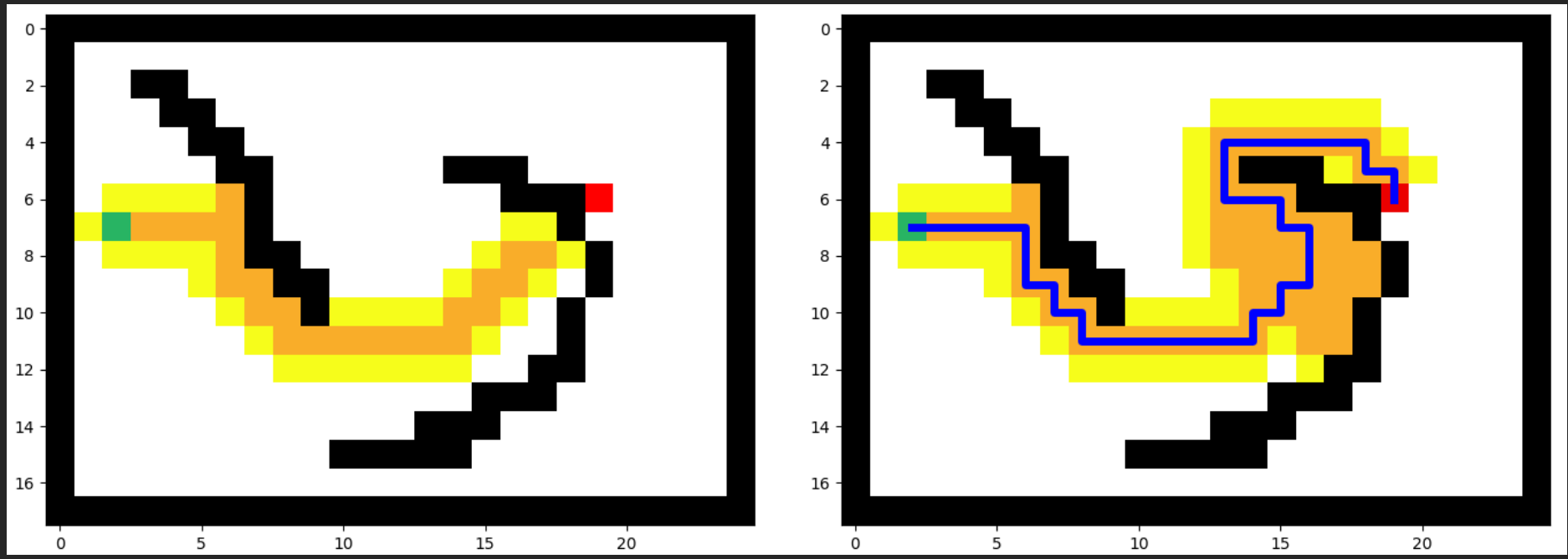
$$\forall\,(i,j)\in[\![0\,,n]\!],\qquad G_{ij}=\begin{cases}S_iS_j & \text{si } (*) \text{ n'est pas vérifiée}\\ -1 & \text{sinon}\end{cases}$$

On applique alors Dijkstra sur le graphe pondéré ainsi créé à l'aide d'une file de priorité.



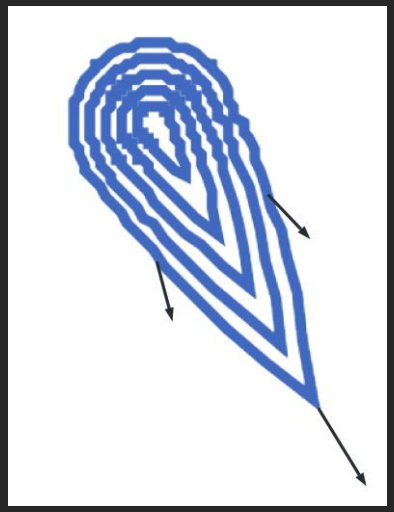
# Parcours en largeur glouton
(GBFS)

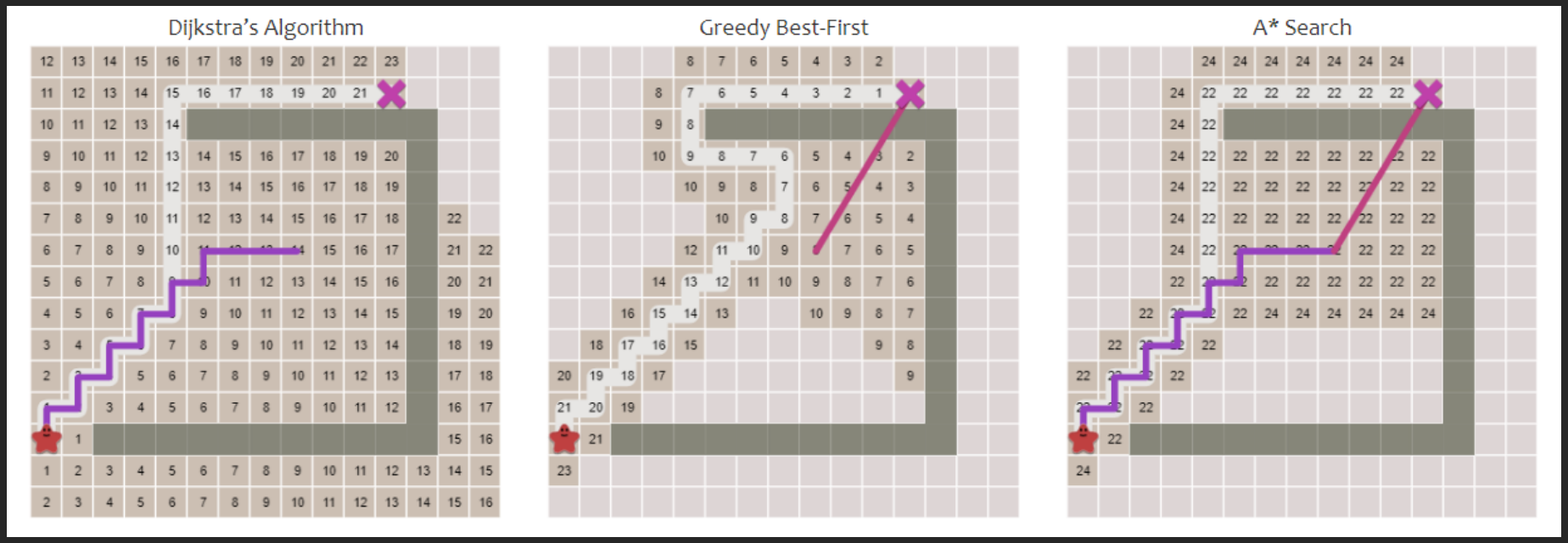
- GBFS = Parcours en largeur + connaissance de la position d'arrivée
- « Glouton » car l'emplacement le plus proche de l'objectif sera exploré en premier



# A*

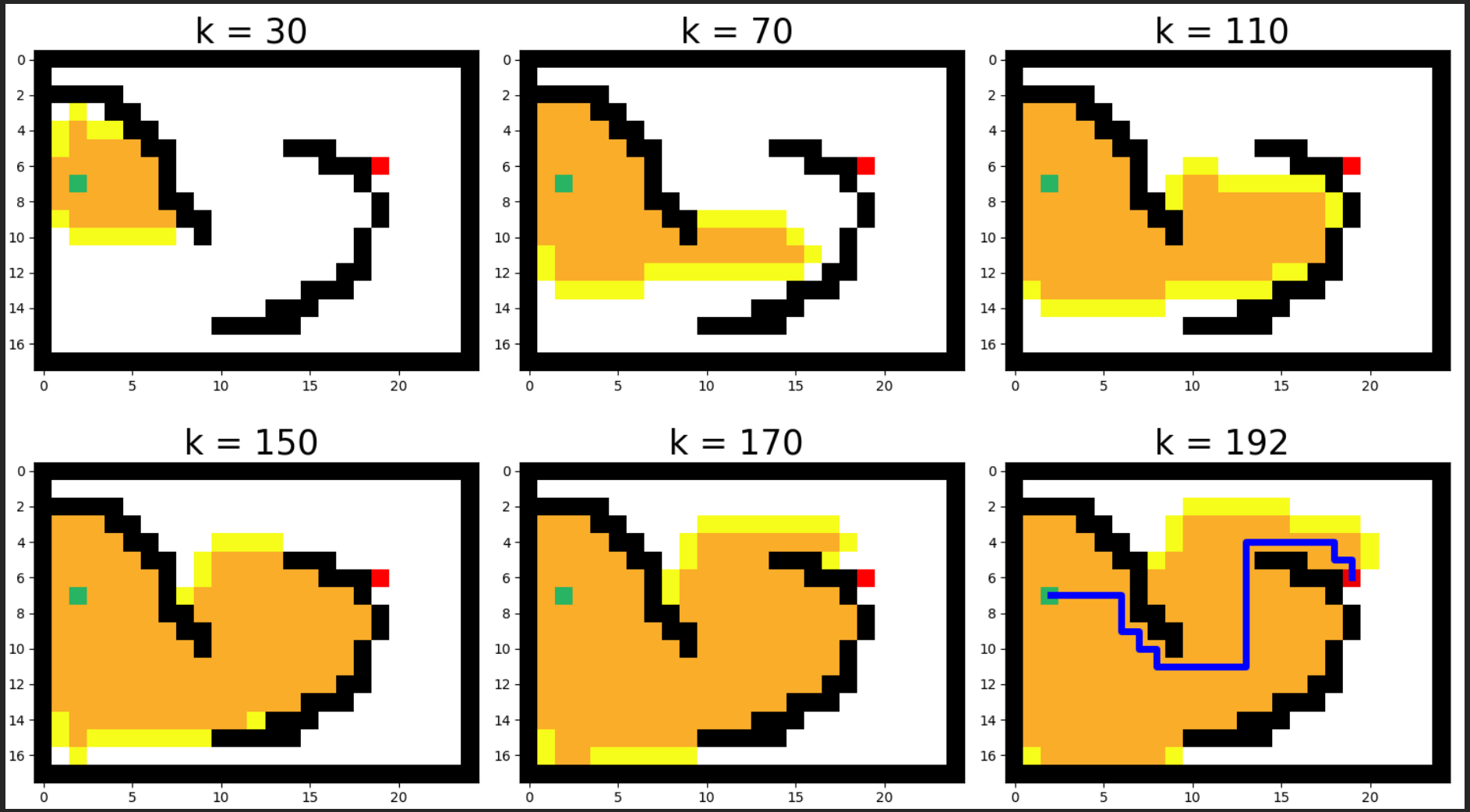
- L'algorithme de Dijkstra calcule la distance à partir du point de départ. GBFS estime la distance par rapport à l'arrivée.
- A* utilise la somme de ces deux distances.






Source : Redblobgames

# Fonctionnement de A*



# Optimalité de A* ?

- A* utilise une fonction heuristique $f = g + h$ dans la file de priorité.
- Pour retourner une solution optimale, l'algorithme A* doit utiliser une **fonction heuristique admissible** :

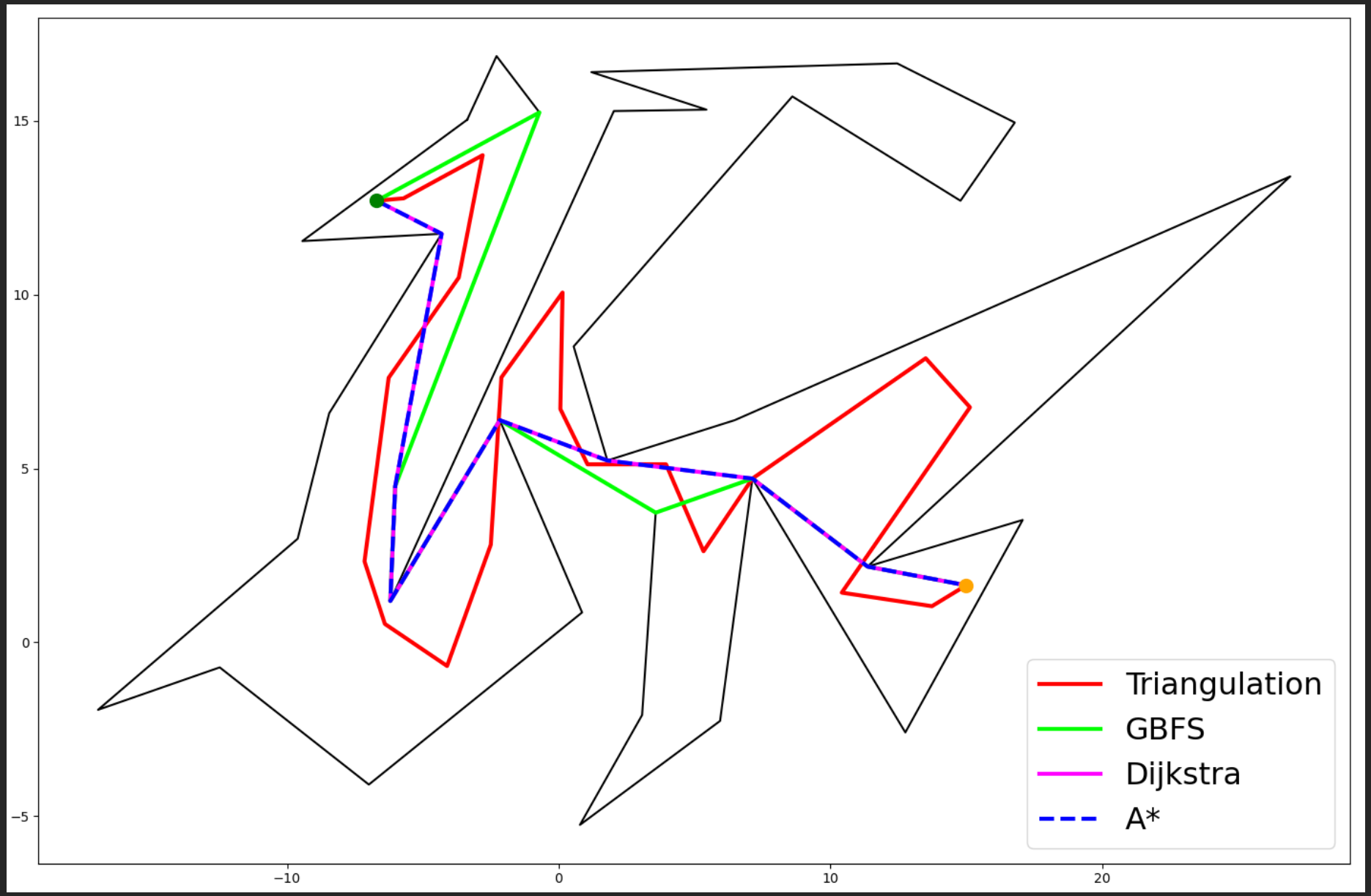
La fonction $h$ ne doit jamais surestimer la distance du but.

- Dans notre cas, on prendra $h$ comme la « distance à vol d'oiseau ».

$$\forall\, i \in [\![1\,, n]\!], \qquad h(i) = S_i A$$

avec $A$ l'arrivée souhaitée et $S_i$ le sommet actuellement visité.

# Résultats numériques


15
10
5
0
−5
−10
0
10
20
Triangulation
GBFS
Dijkstra
A*

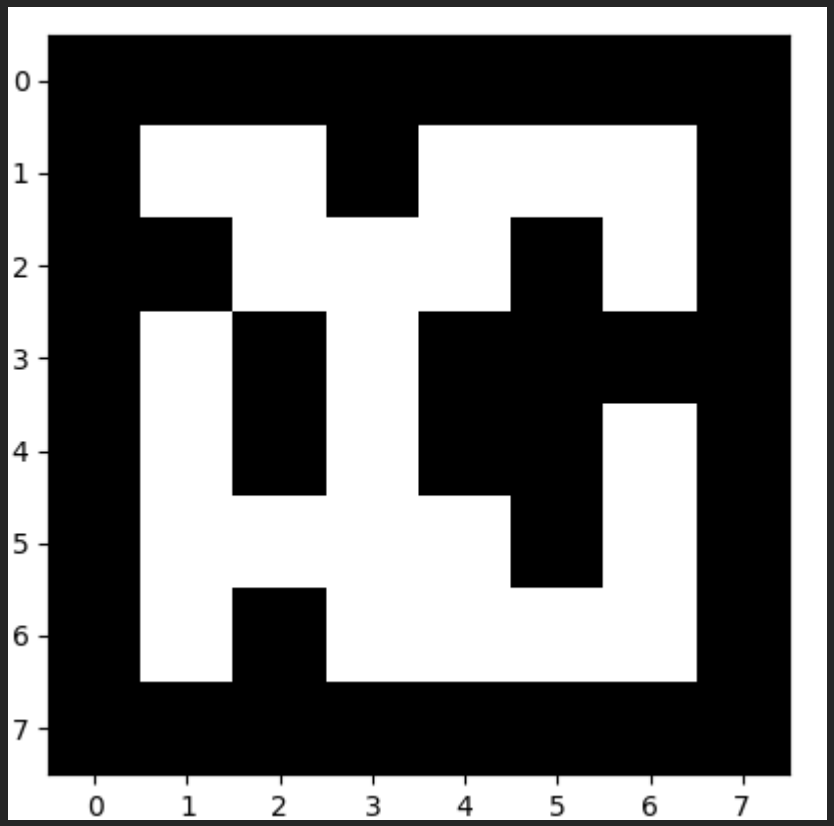
# Temps de calculs

| Algorithme | Triangulation | GBFS | Dijkstra | A* |
|---|---|---|---|---|
| Complexité (pire des cas) | $\mathcal{O}(n^2)$ | $\mathcal{O}(n^2 \log(n))$ | $\mathcal{O}(n^2 \log(n))$ | $\mathcal{O}(n^2 \log(n))$ |
| Temps calcul moyen (en secondes) | 0.007 | 0.042 | 0.046 | 0.045 |
| Optimalité du chemin | Non | Non | Oui | Oui |

Preuve des complexités en annexe

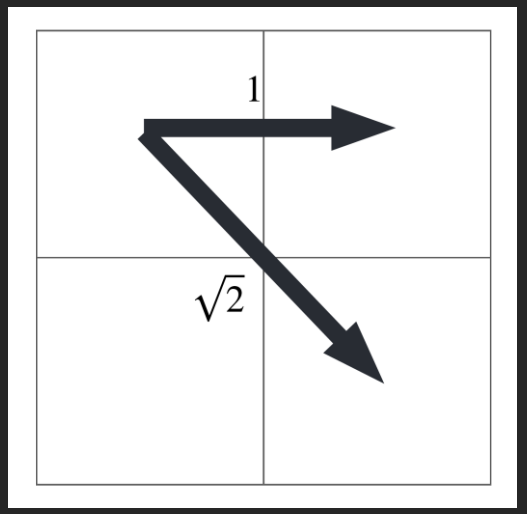
Annexe 3

# II. Quadrillage de l'espace

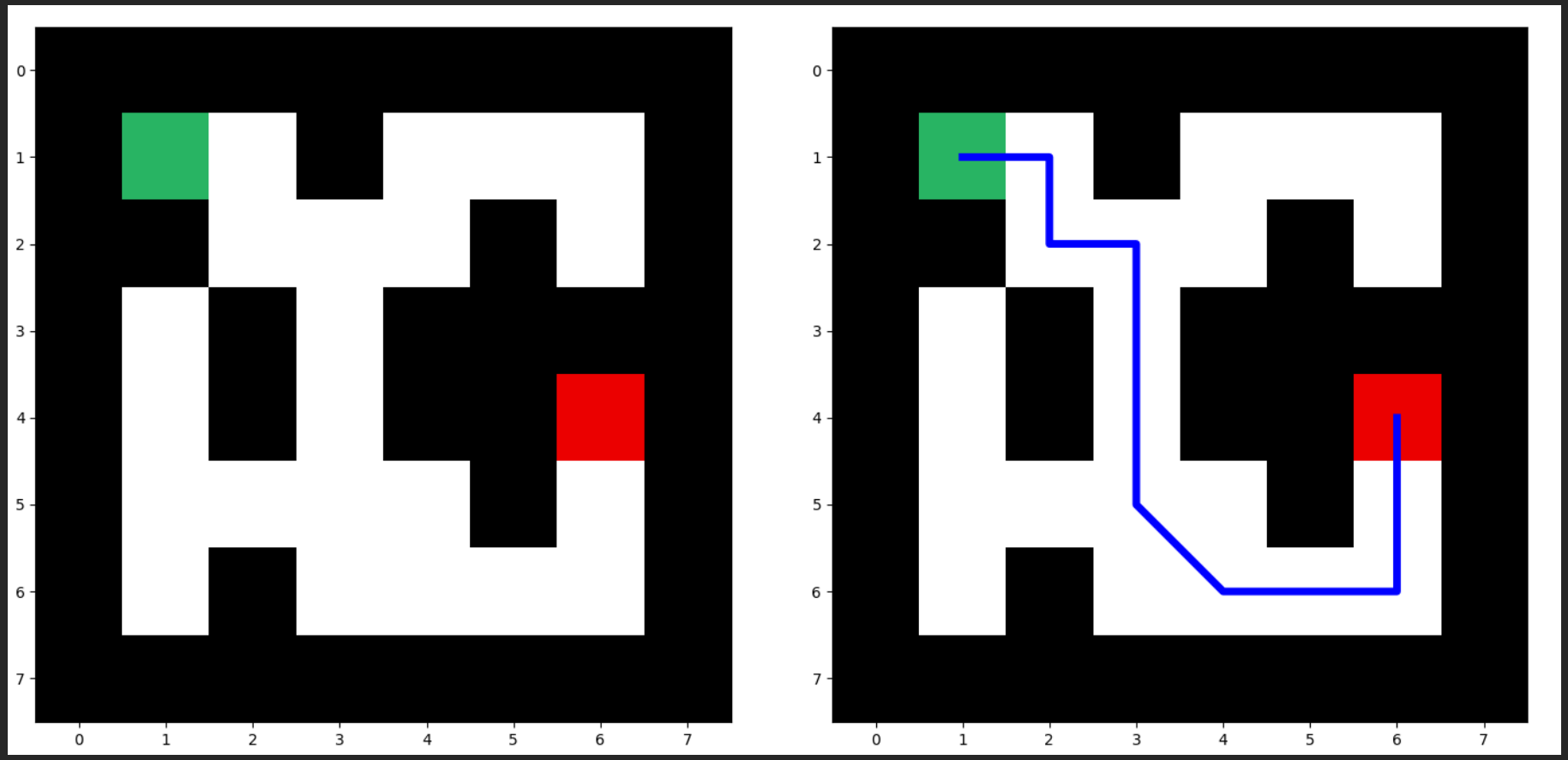



Importation d'image avec le module

**matplotlib.image**

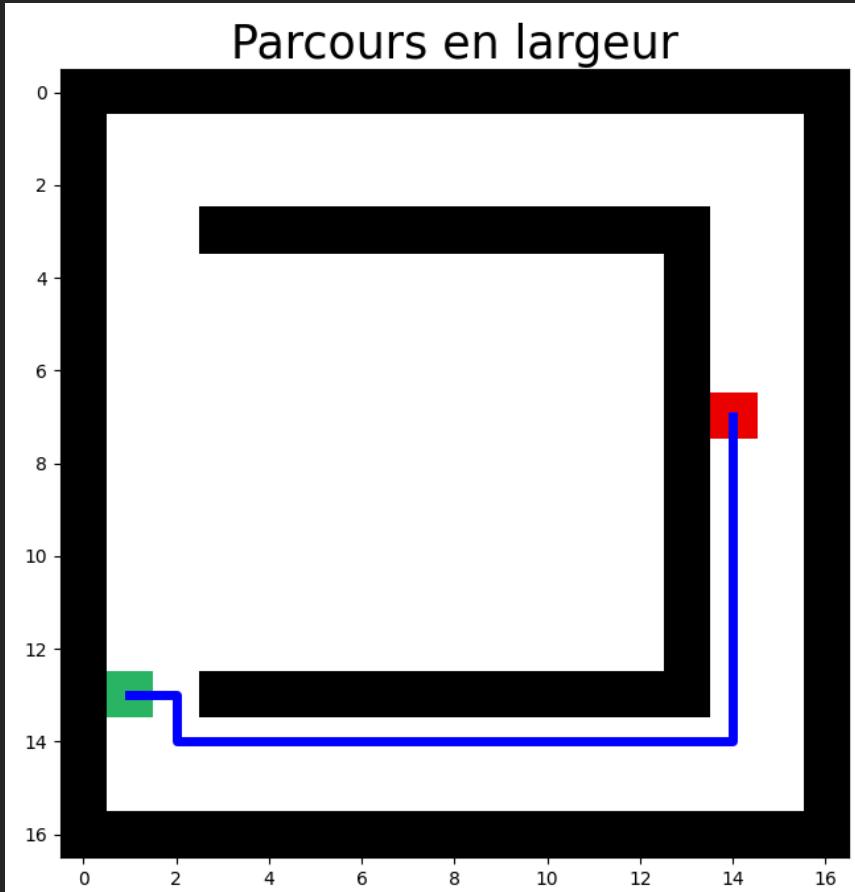


# Parcours en largeur

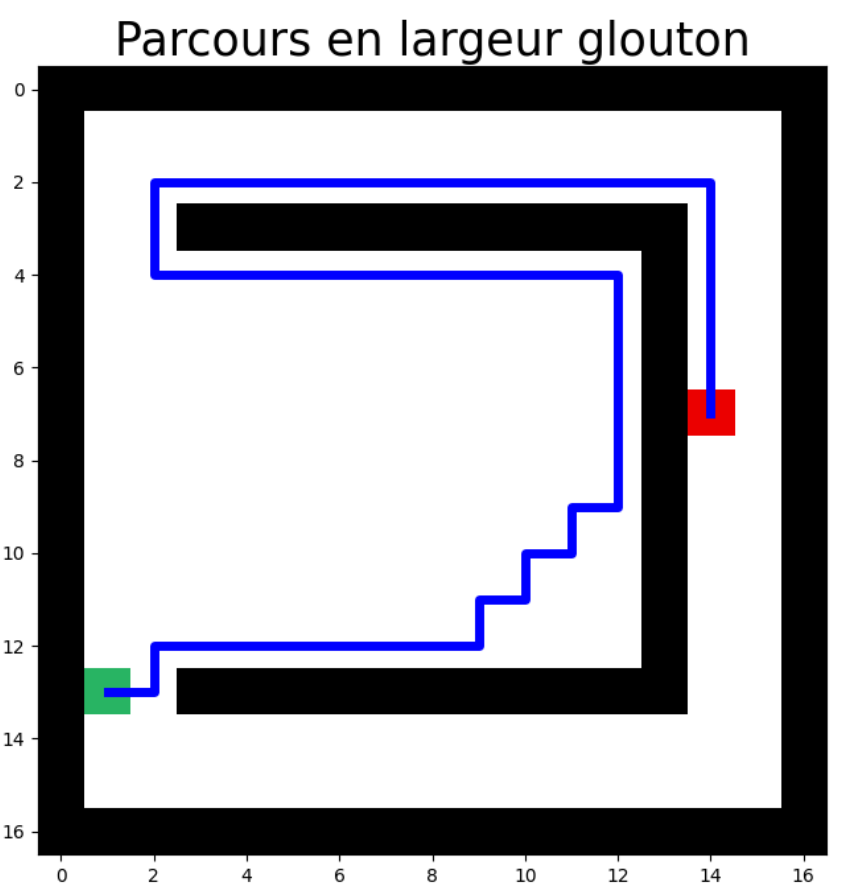



Coût du chemin : 11.41

# Parcours en largeur glouton

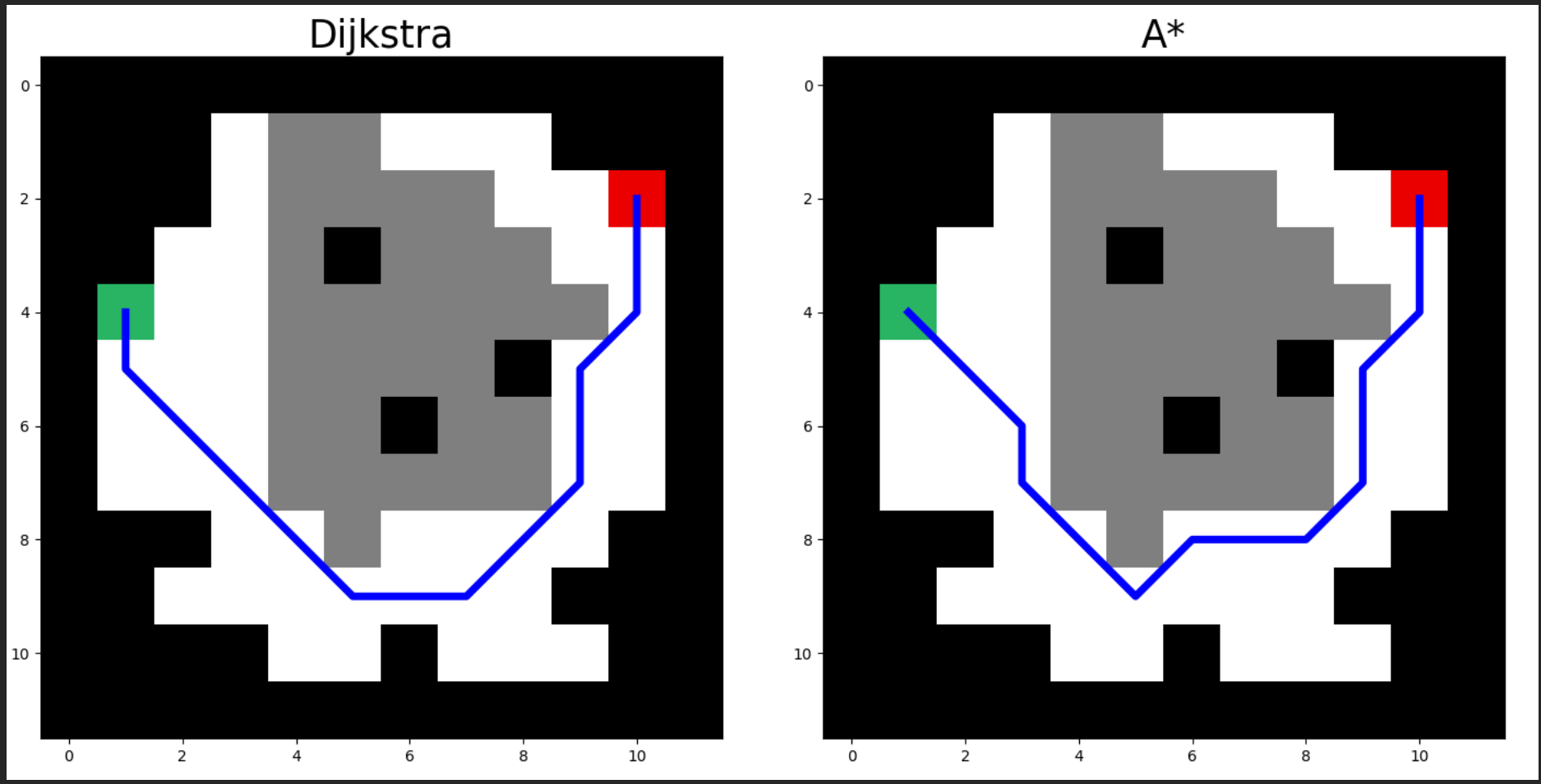



Coût du chemin : 21




Coût du chemin : 49

# Dijkstra et A*




Coût des <u>deux</u> chemins : **16,9**

# Divers exemples plus complexes

- Petite chasse au trésor
- Ville de Paris

- Un Polygone complexe Annexe 4

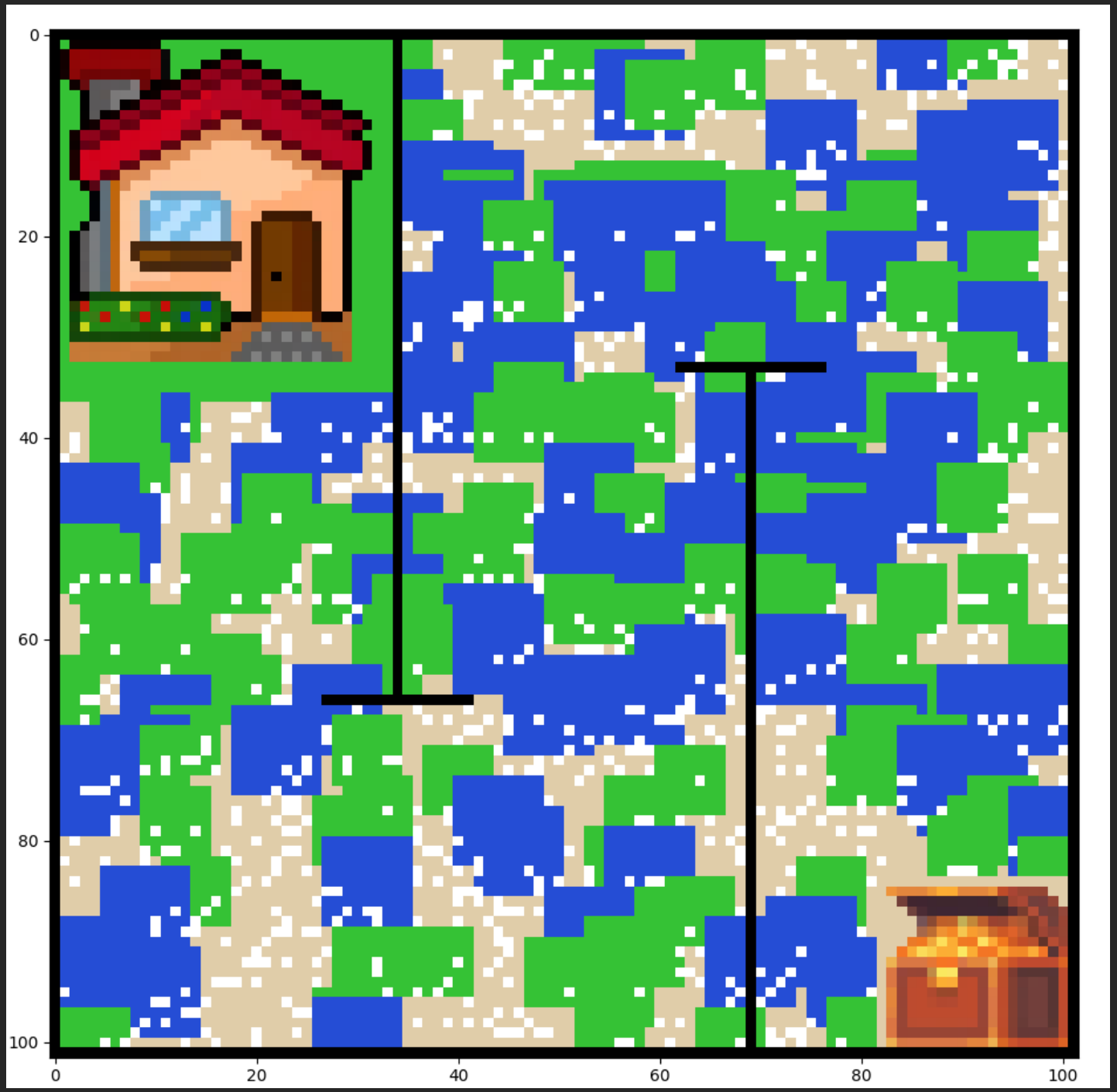
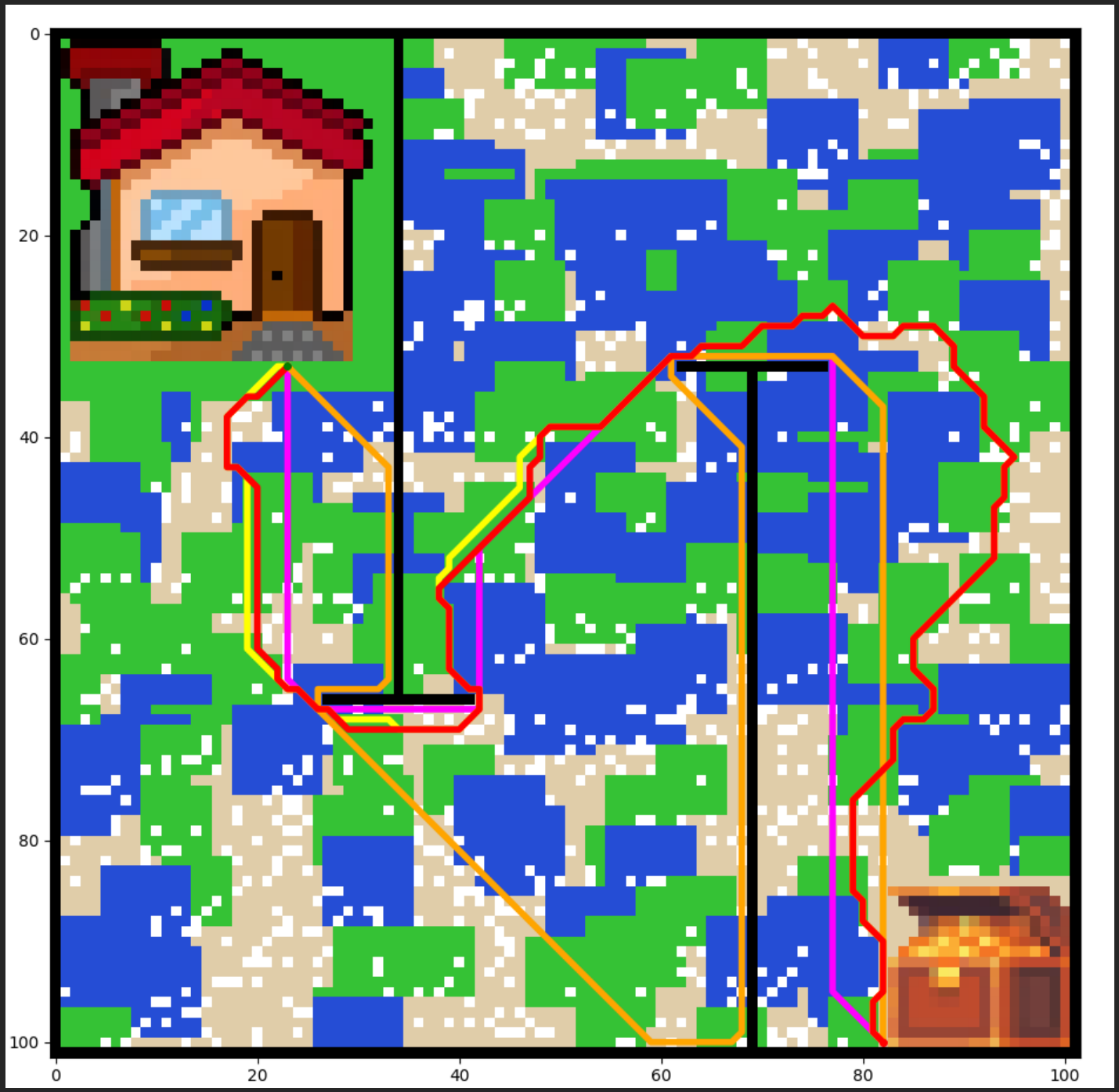



Table des coûts :

Herbe : 1
Sable : 2
Eau : 5
Pièges : 10
Mur : $C_{max}$

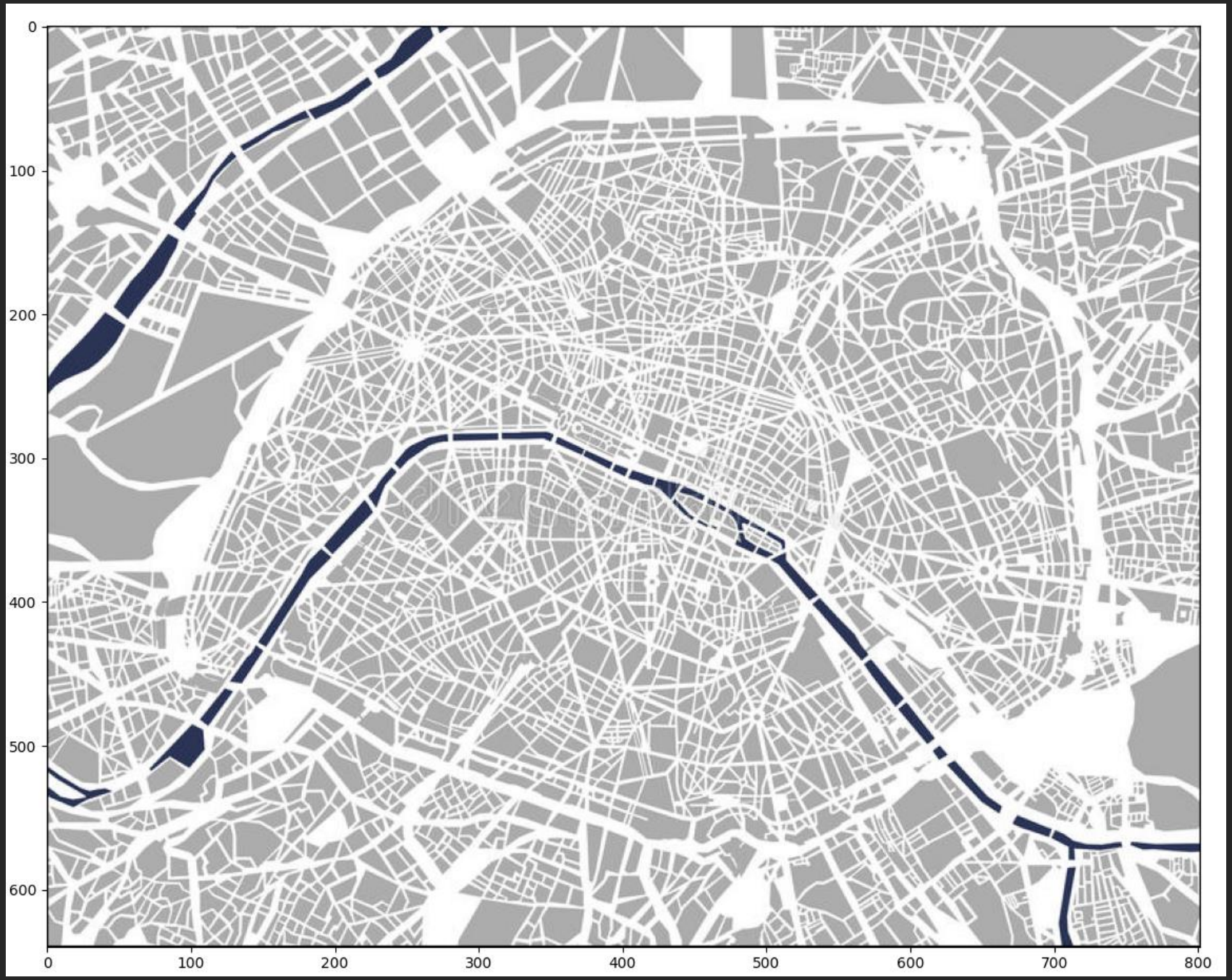

0
20
40
60
80
100
0
20
40
60
80
100

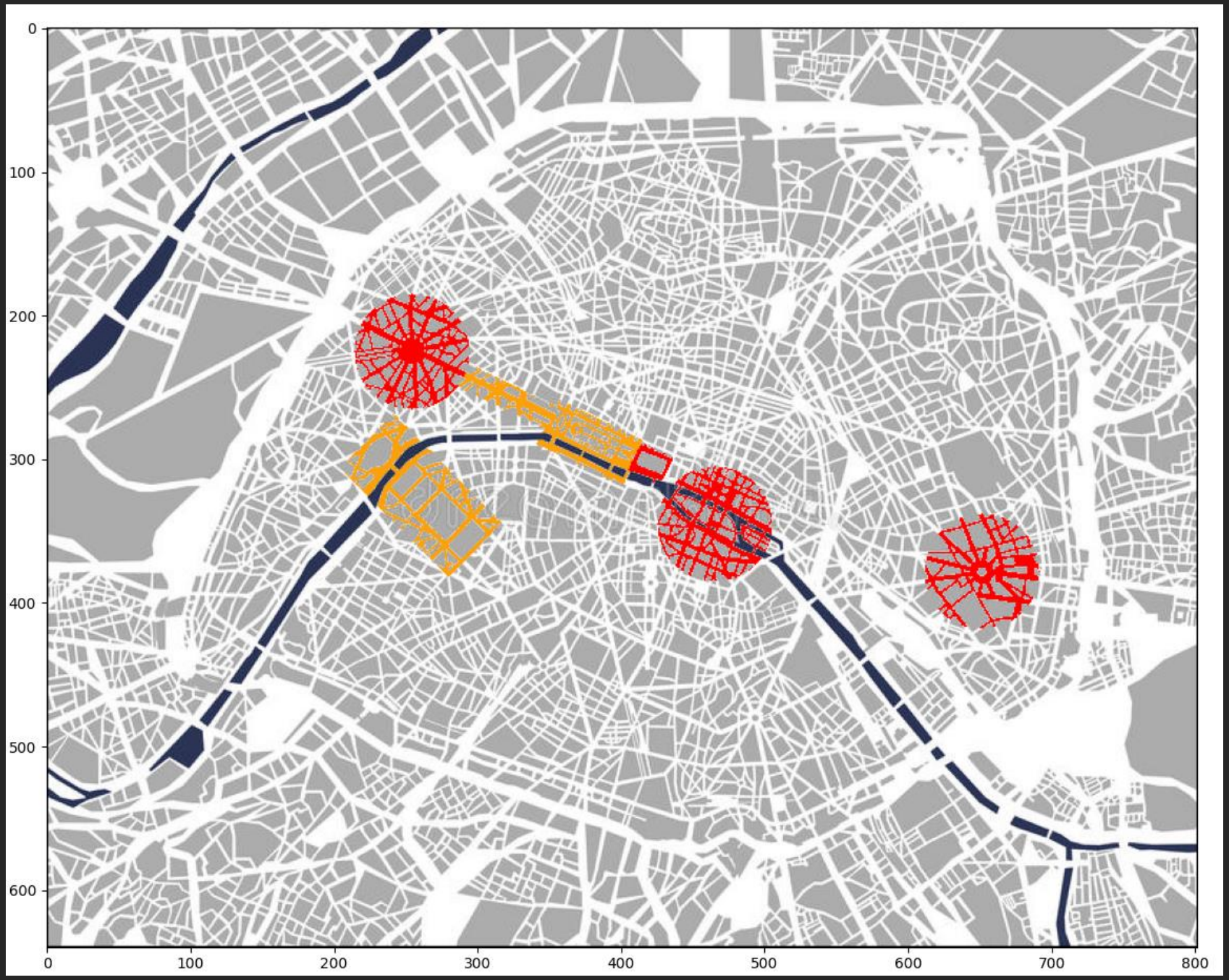
# Résultats numériques

| Algorithme | Parcours en largeur | GBFS | Dijkstra | A* |
|---|---|---|---|---|
| Complexité (pire des cas) | $\mathcal{O}(n^2)$ | $\mathcal{O}(n^2 \log(n))$ | $\mathcal{O}(n^2 \log(n))$ | $\mathcal{O}(n^2 \log(n))$ |
| Temps calcul moyen (en secondes) | 2.88 | 1.23 | 9.51 | 8.14 |
| Coût du chemin | 670.66 | 983.34 | 356.31 | 356.31 |
| Optimalité du chemin | Non | Non | Oui | Oui |

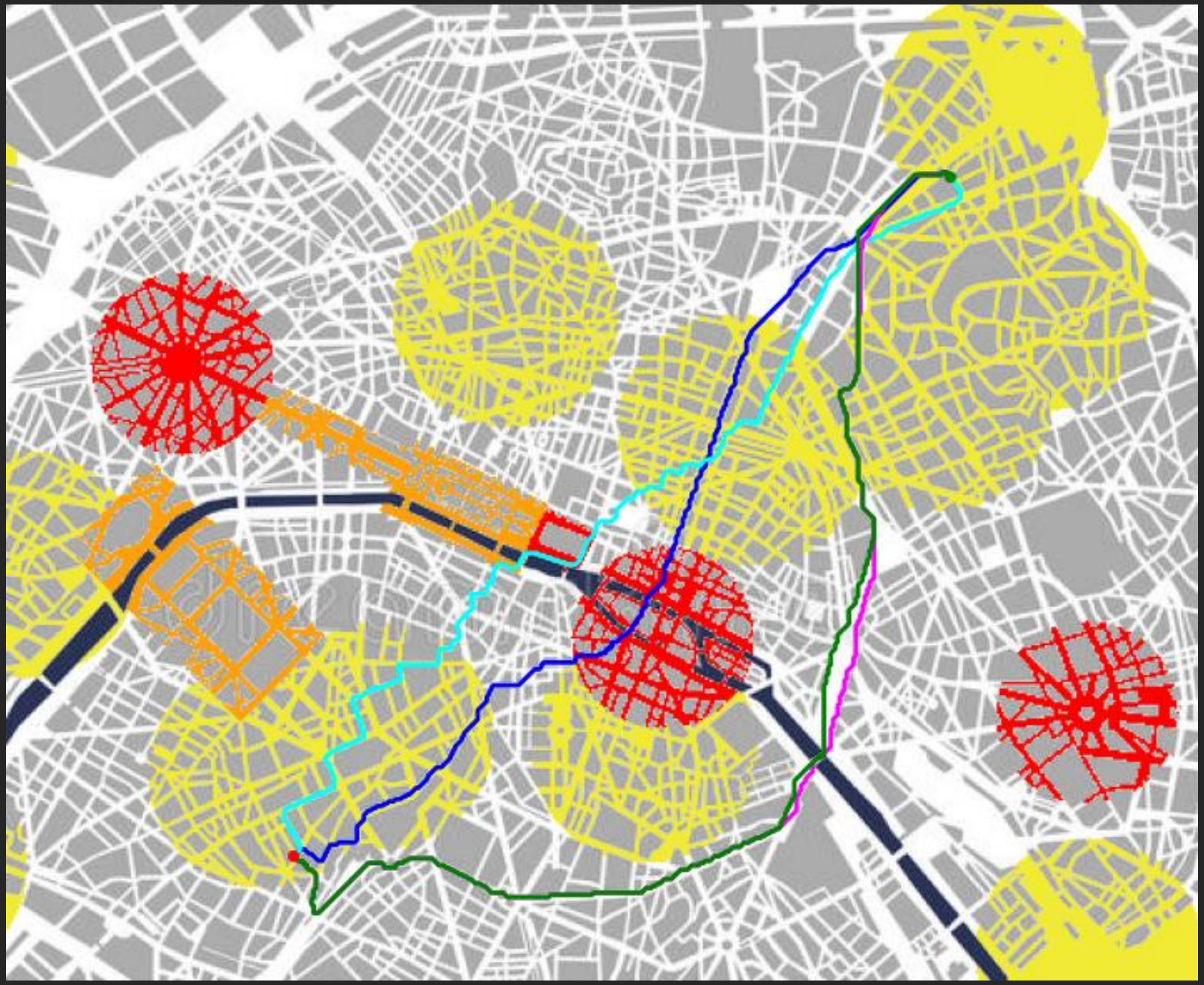
Preuve des complexités en annexe

Annexe 3


0
100
200
300
400
500
600
0
100
200
300
400
500
600
700
800


0
100
200
300
400
500
600
0
100
200
300
400
500
600
700
800


0
100
200
300
400
500
600
0
100
200
300
400
500
600
700
800



# Résultats numériques

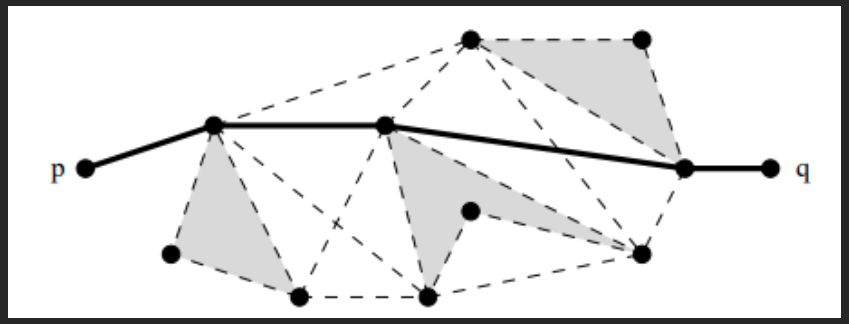
| Algorithme | Parcours en largeur | Greedy best first Search | Dijkstra | A* |
|---|---|---|---|---|
| Complexité (pire des cas) | $\mathcal{O}(n^2)$ | $\mathcal{O}(n^2 \log(n))$ | $\mathcal{O}(n^2 \log(n))$ | $\mathcal{O}(n^2 \log(n))$ |
| Temps calcul moyen (en secondes) | 32.8 | 0.62 | 118.4 | 37.9 |
| Coût du chemin | 1025.5 | 931.9 | 557.2 | 557.2 |
| Optimalité du chemin | Non | Non | Oui | Oui |

Preuve des complexités en annexe

Annexe 3

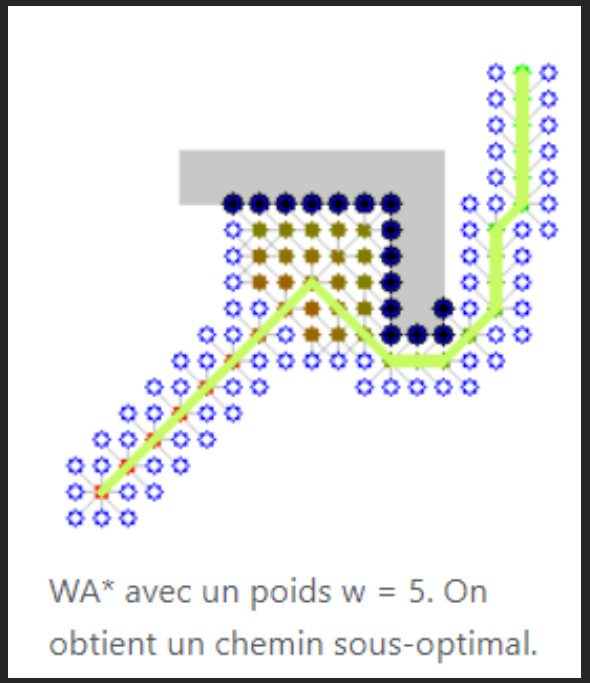
# D’autres idées intéressantes…

- Calculs de visibilité dans un environnement polygonal 2D




Source : Thèse de Stéphane Rivière

- Différents variant de A*. Exemple notable : WA*



WA* avec un poids w = 5. On obtient un chemin sous-optimal.

Source : Wikipédia

Annexe 5

# Annexes

- Annexe 1 : Tout polygone admet une triangulation
- Annexe 2 : Triangulation de Delaunay
- Annexe 3 : Complexité des algorithmes énoncés
- Annexe 4 : Exemple d'un polygone complexe
- Annexe 5 : Algorithme WA* pour différents W

- Annexe 6 : Codes Python

# Annexe 1

**Théorème** : Tout polygone admet une triangulation

**Preuve** :

Soit $\mathcal{P}$ un polygone à $n$ ($\geq 3$) sommets. On résonne par récurrence forte sur $n$.

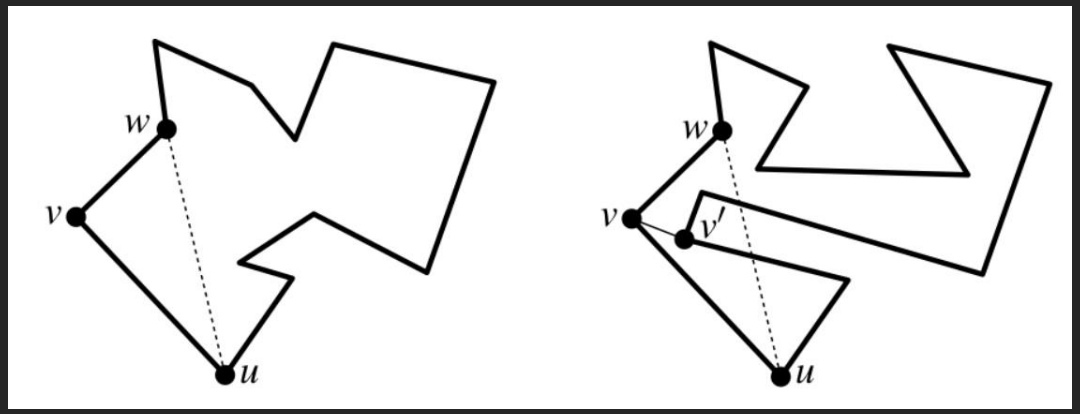
- Initialisation : $n = 3$. La propriété est triviale : tout polygone à 3 sommets est un triangle. L'initialisation est établie.

- Hérédité : Supposons la propriété vraie pour tout rang $k$ avec $k \in [\![3\,, n-1]\!]$. Montrons qu'elle reste vraie au rang $n$.

On doit prouver l'existence d'une diagonale dans $\mathcal{P}$. Soit $v$ le sommet le plus à gauche de $\mathcal{P}$ (et le plus bas en cas de non unicité).

Soit $u$ et $w$ les deux sommets voisins de $v$ sur la frontière de $\mathcal{P}$.

- Si le segment $[uw]$ est à l'intérieur de $\mathcal{P}$, on a trouvé une diagonale.
- Sinon il existe un ou plusieurs sommets dans le triangle $(u, v, w)$ ou sur la diagonale $[uw]$. Soit $v'$ le sommet le plus éloigné de $[uw]$, le segment $[vv']$ n'intersecte pas le contour de $\mathcal{P}$ (car sinon, ce segment aurait un sommet plus éloigné de $[uw]$, ce qui est contradictoire).




Puisqu'une diagonale existe, elle partage le polygone $\mathcal{P}$ en deux sous-polygones $\mathcal{P}_1$ et $\mathcal{P}_2$, comportant respectivement $m_1$ et $m_2$ sommets, tous deux inférieurs à $n$. Donc, par hypothèse de récurrence, $\mathcal{P}_1$ et $\mathcal{P}_2$ peuvent être triangulés.
Donc $\mathcal{P}$ admet une triangulation.

L'hérédité est établie. □

# Annexe 2

## Triangulation de Delaunay

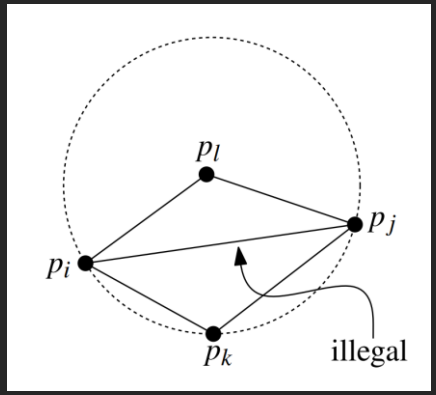
**Définition** :
La triangulation de Delaunay d'un ensemble $\mathcal{P}$ de points du plan est une triangulation $DT(\mathcal{P})$ telle qu'aucun point de $\mathcal{P}$ n'est à l'intérieur du cercle circonscrit d'un des triangles de $DT(\mathcal{P})$.
Les triangulations de Delaunay maximisent le plus petit angle de l'ensemble des angles des triangles, évitant ainsi les triangles « allongés ».

En effet, comme il n'y a qu'un nombre fini de différentes triangulations d'un ensemble de points $\mathcal{P}$ donné, cela implique qu'il y a une triangulation optimale, celle qui maximise l'angle minimum.

**Théorème** : Soit $\mathcal{P}$ un ensemble de points du plan. Une triangulation $T$ de $\mathcal{P}$ est légal si et seulement si $T$ est une triangulation de Delaunay de $\mathcal{P}$.

**Preuve** : *Computational Geometry : Algorithms and Applications* (Chapitre 9)

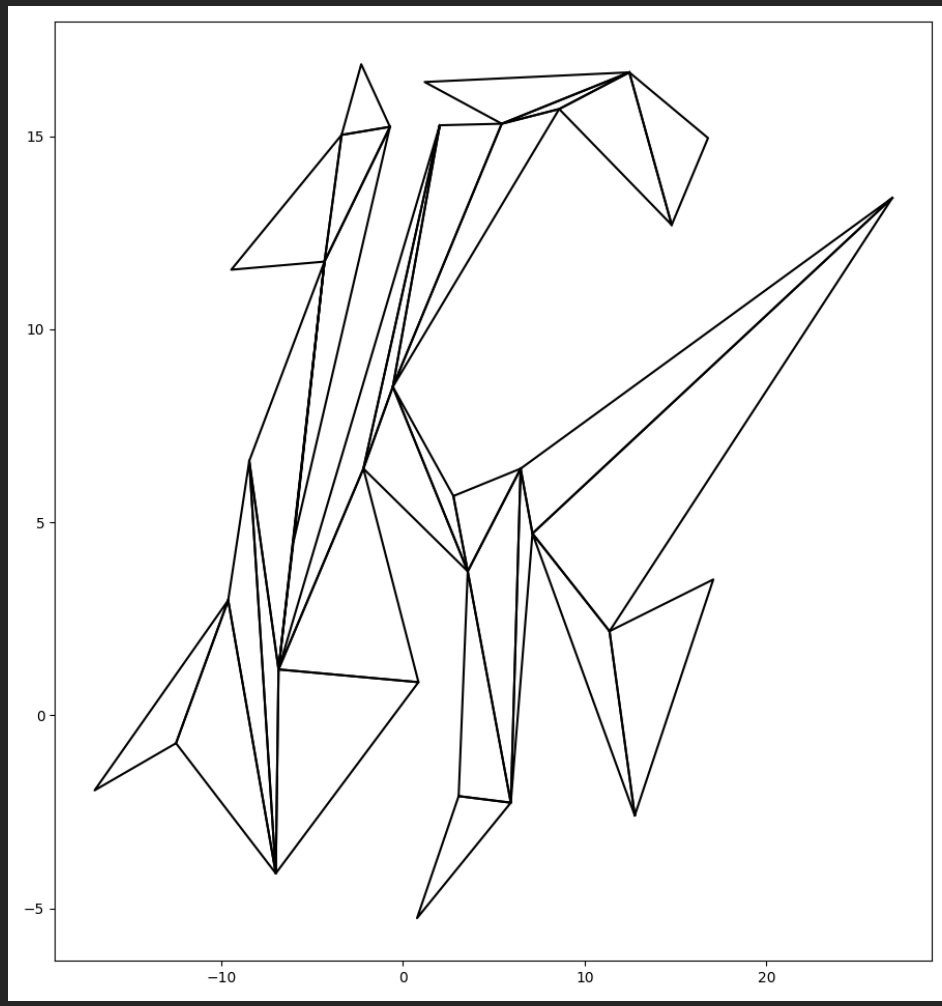
Une triangulation $T$ est dite « légal » lorsqu'elle ne contient aucune arête illégale



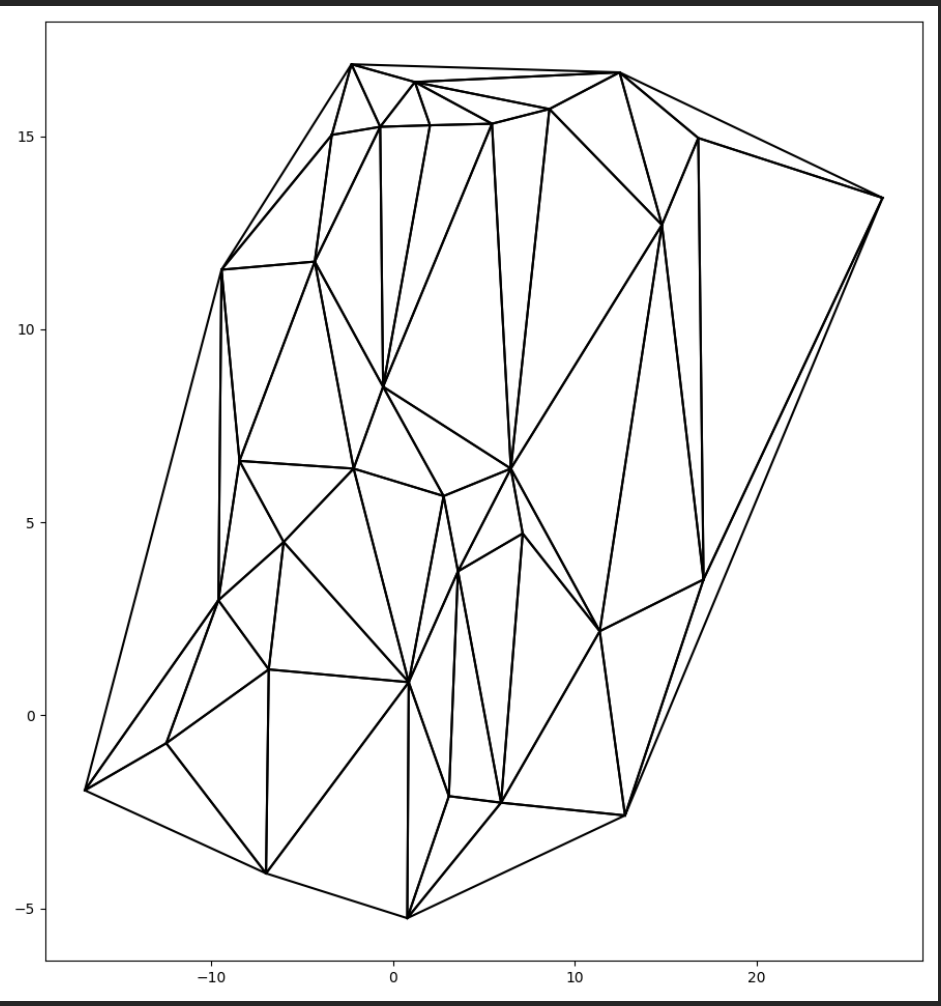

**Implémentation** :

- ITERATIVE : Ajouter les sommets un par un en recalculant la triangulation des parties du graphe affectées par cet ajout. Et cela à l'aide d'une fonction de basculement.
  **Complexité** : $\mathcal{O}(n^2)$ , et $\mathcal{O}(n \ln(n))$ en moyenne

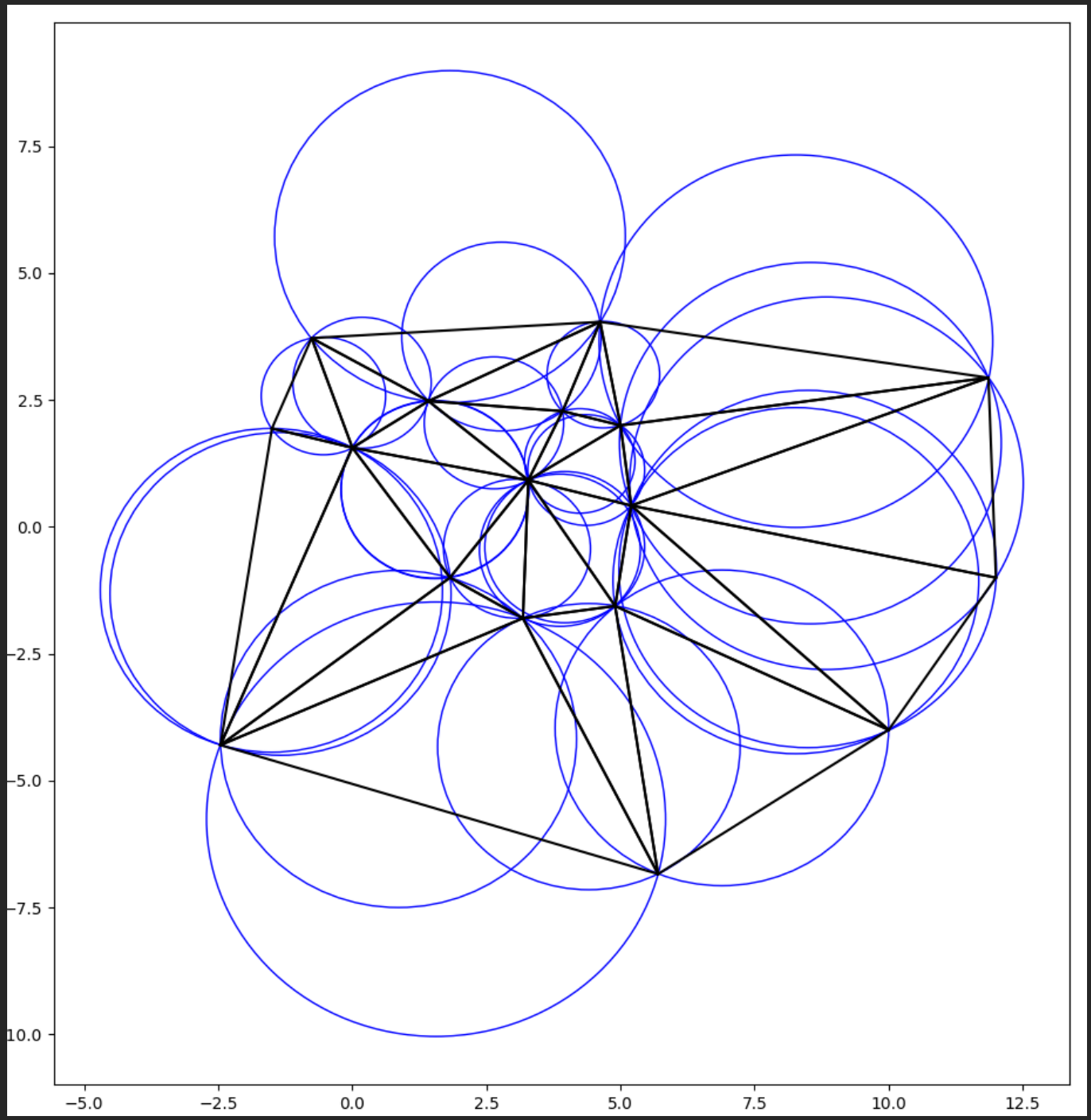
- RECURSIVE : Méthode de diviser pour régner → algorithme de Guibas et Stolfi.
  **Complexité** : $\mathcal{O}(n \ln(n))$

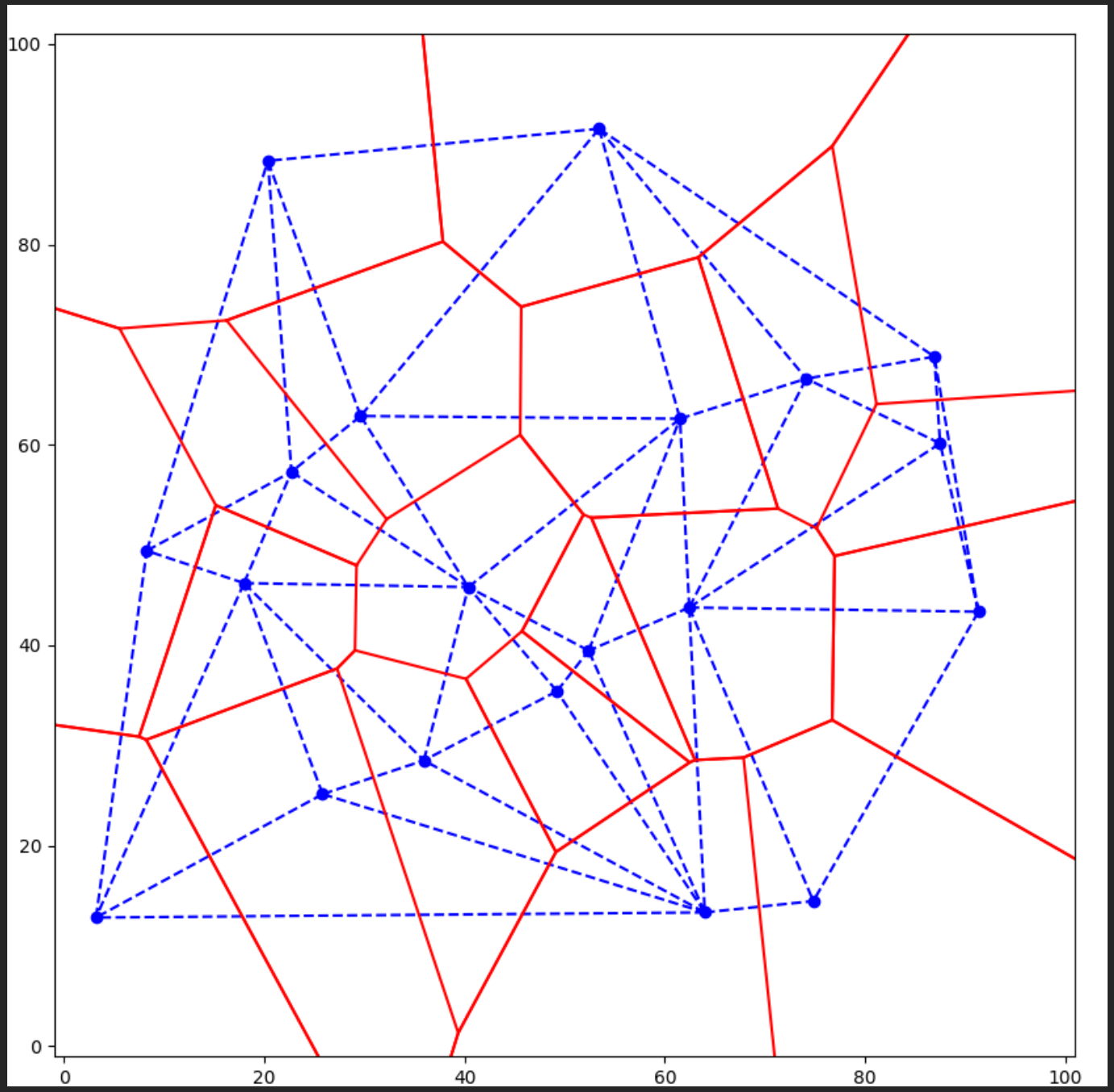



Triangulation précédente




Triangulation de Delaunay


7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
10.0
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
12.5




Source : github.com/jmespadero/pyDelaunay2D

# Annexe 3

## Preuve des complexités

**Algorithme de triangulation** :

Notons $T$ le coût de la complexité temporelle dans le pire des cas.
L'algorithme de triangulation est une fonction récursive. Elle découpe le polygone en deux polygones distincts sur lesquels elle s'applique alors récursivement.
La relation de récurrence sur le coût $T$ est alors :

$$T(n) = T(n_1) + T(n - n_1 + 2) + f(n)$$

Où :

- $n$ est le nombre de sommets du polygone.
- $n_1$ est le nombre de sommets du premier sous-polygone.
- $n - n_1 + 2$ correspond au nombre de sommets du deuxième sous-polygone.
- $f(n)$ est une fonction de coût intervenant à chaque appel de la fonction.

Les fonctions `ind_voisin` et `triangle` sont des fonctions de coût constants, et les fonctions `sommet_gauche`, `sommet_distance_max` et `nouveau_polygone` sont des fonctions en $\mathcal{O}(n)$ avec $n$ le nombre de sommets du polygone.
Donc :

$$f(n) = 4 \times \mathcal{O}(n) + \mathcal{O}(1)$$

$$\Rightarrow \quad f(n) = \mathcal{O}(n)$$

Donc :

$$f(n - n_1 + 2) = \mathcal{O}(n - n_1 + 2) = \mathcal{O}(n - n_1)$$

Et finalement :

$$f(n_1) + f(n - n_1 + 2) = \mathcal{O}(n)$$

L'algorithme parcourt l'ensemble des sommets du polygones, en ajoutant au total, à chaque itération, 2 sommets. Le cas d'arrêt étant $n = 3$. Ainsi :

$$T(n) \leq \sum_{k=3}^{n+2n} f(n) = \sum_{k=3}^{3n} \mathcal{O}(n) = (3n - 2) \times \mathcal{O}(n) = \mathcal{O}(n^2)$$

Par conséquent:

$$\boxed{T(n) = \mathcal{O}(n^2)}$$

**Parcours en largeur :**

Notons $C$ le coût de la complexité temporelle dans le pire des cas.
L'algorithme de parcours en largeur applique la fonction `traite` à chaque élément de la file, tant que celle-ci n'est pas vide. On a donc au maximum une application par sommet. Celle-ci effectue un appel à `traite_fils`, qui est en $\mathcal{O}(p)$ avec $p$ la longueur de la liste d'adjacence du sommet traité. Le coût total de la boucle `while` est donc dominé par la somme totale des longueurs des listes d'adjacences.
L'extraction du sommet dans la file est en $\mathcal{O}(1)$.

Donc :

$C = \mathcal{O}(n + m)$ avec $n$ le nombre de sommets et $m$ le nombre d'arêtes.

Finalement, comme $m \leq n^2$, on conclut :

$$C = \mathcal{O}(n + n^2) \Rightarrow \boxed{C = \mathcal{O}(n^2)}$$

**GBFS, Dijkstra et A* :**

Notons $C$ le coût de la complexité temporelle dans le pire des cas.
Dans ces 3 algorithmes, les sommets du graphes sont parcourus selon l'ordre dicté par une file de priorité. La structure de ces 3 algorithmes est la même. Ce qui les différencie est alors le choix de la file de priorité.

Par recherche dichotomique, l'insertion d'un élément dans la file de priorité se fait en $\mathcal{O}(\log(n))$. Par contre, l'extraction se fait en $\mathcal{O}(1)$.

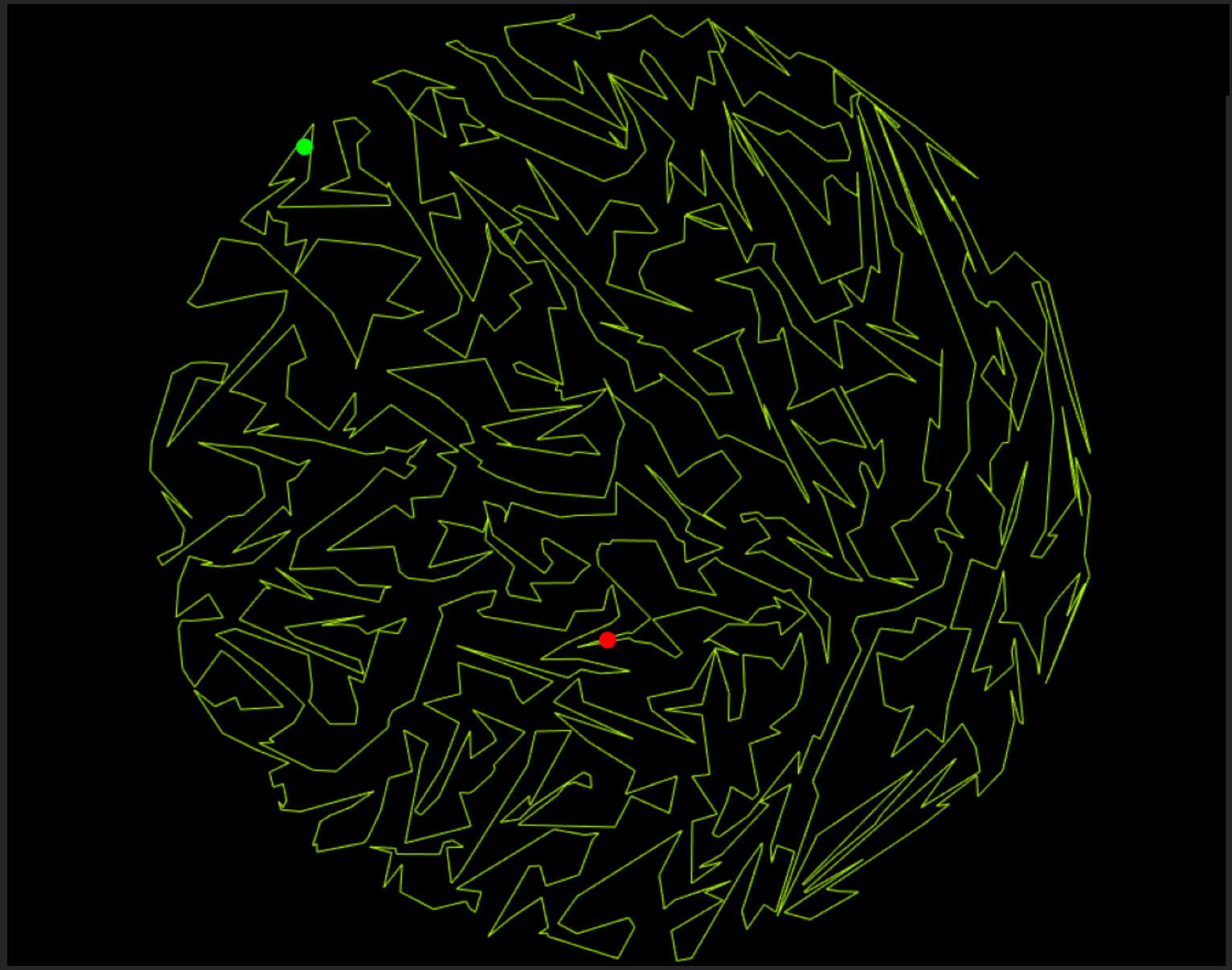
À l'image de la preuve de la complexité de l'algorithme du parcours en largeur, le coût total de la boucle `while` est donc dominé par la somme totale des longueurs des listes d'adjacences, multiplié par le coût de l'insertion du sommet dans la file de priorité.
Donc :

$$C = \mathcal{O}((n+m)\log(n))$$

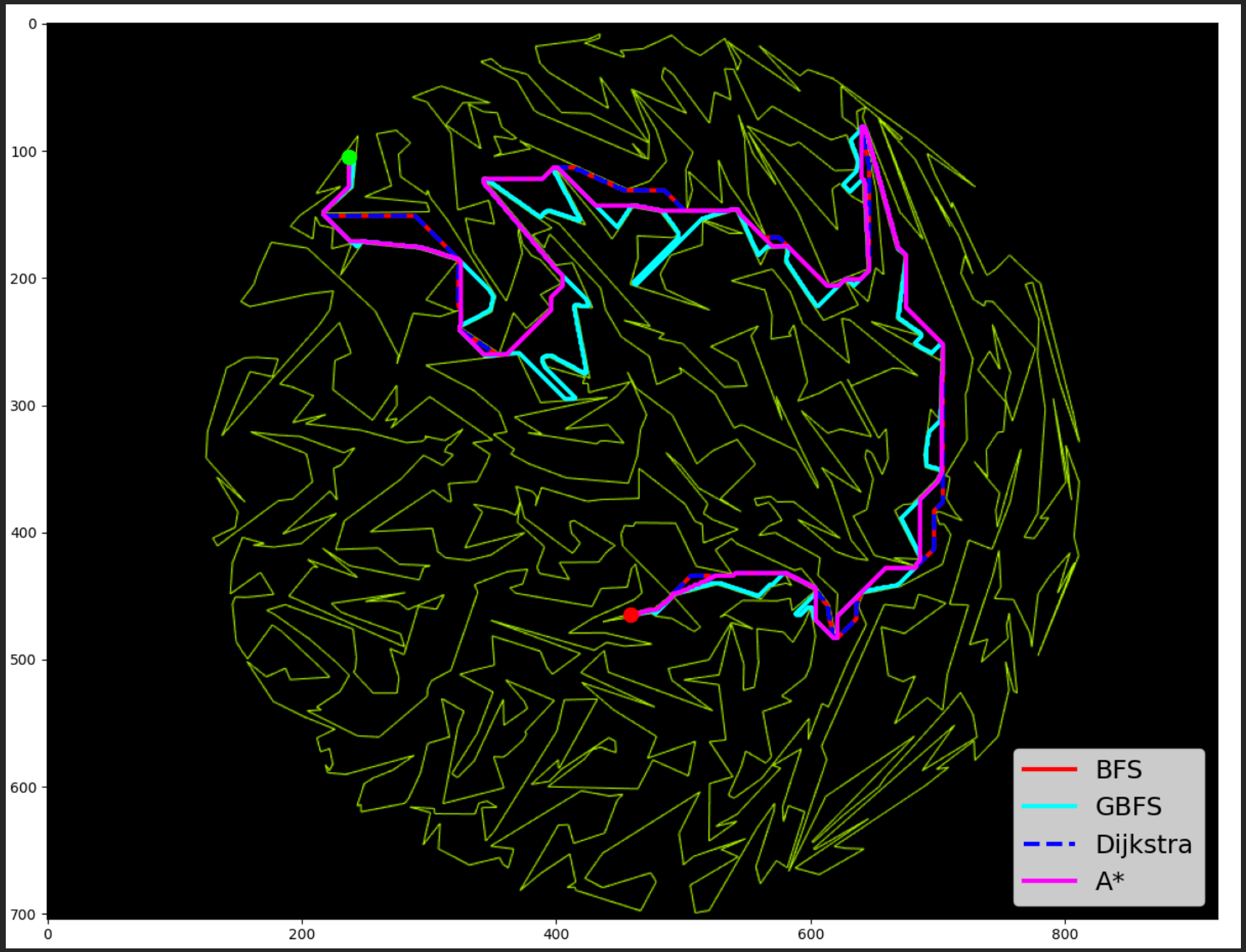
Or, $m \leq n^2$. On conclut :

$$\boxed{C = \mathcal{O}(n^2 \log(n))}$$

# Annexe 4



Source : github.com/aewallin/randompolygon


0
100
200
300
400
500
600
700
0
200
400
600
800
BFS
GBFS
Dijkstra
A*

# Annexe 5

## Algorithme WA*

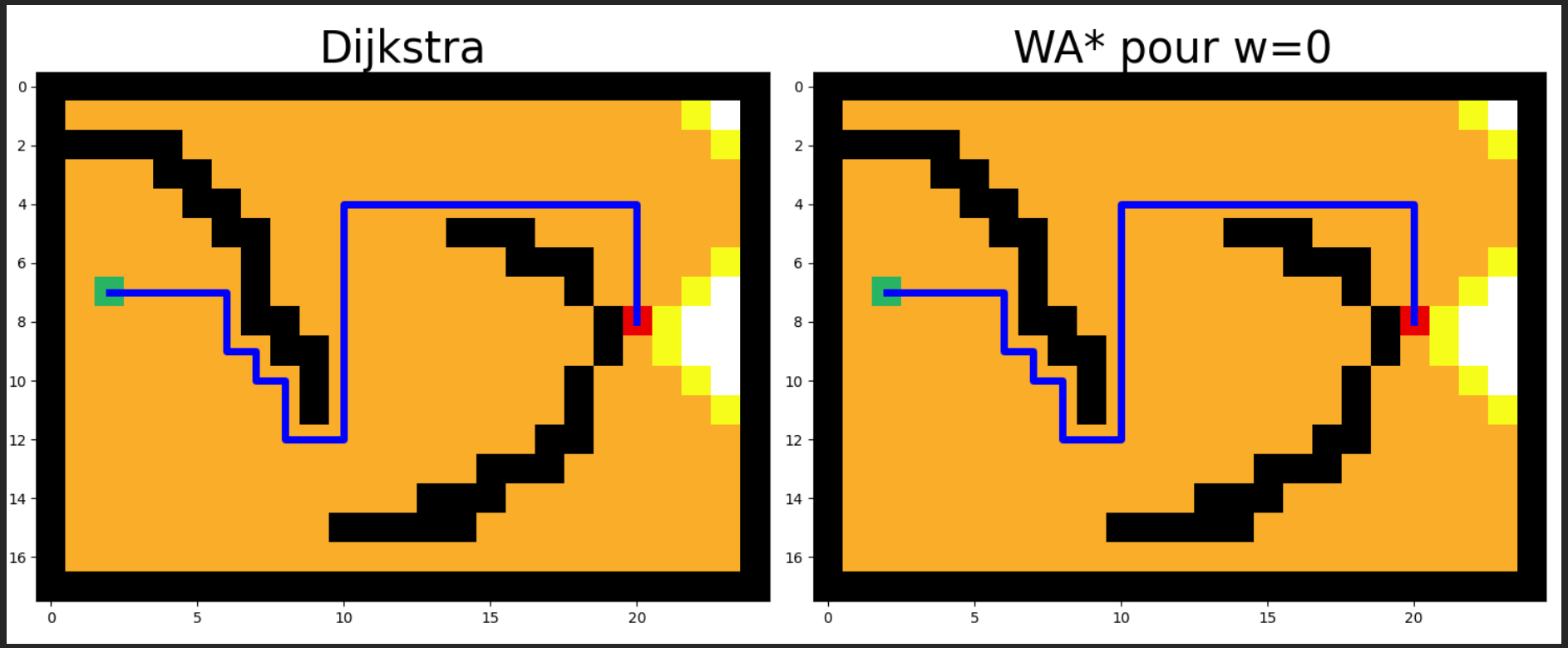
A* utilise une fonction heuristique $f = g + h$ dans la file de priorité.

L'algorithme WA* est simplement un algorithme A* tel que :

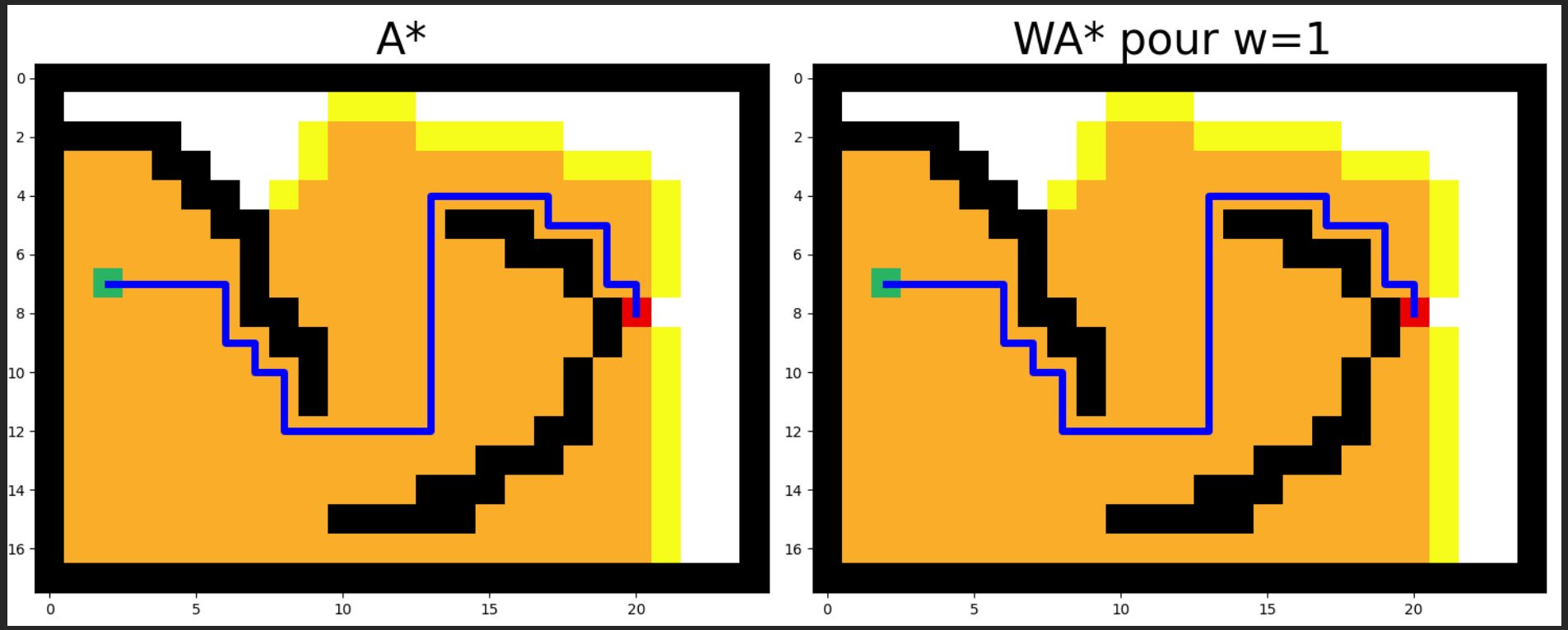
$$f = g + w \times h \qquad \text{avec } w \in \mathbb{R}_+$$

- Si $w = 0$, on a : $f = g$. Donc WA* est exactement Dijkstra.
- Si $w = 1$, on a : $f = g + h$. Donc WA* est exactement A*.
- Si $w \gg 1$, on a : $f \approx w \times h$. Donc WA* se rapproche de l'algorithme de parcours en largeur glouton (GBFS).

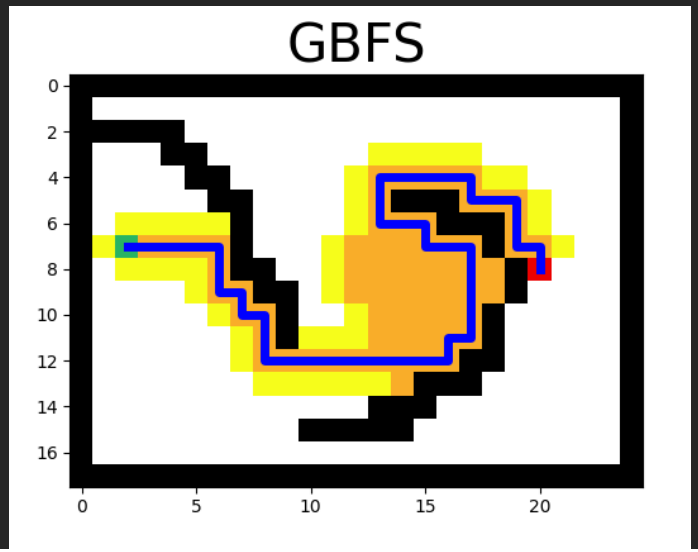
# WA* pour $w = 0$



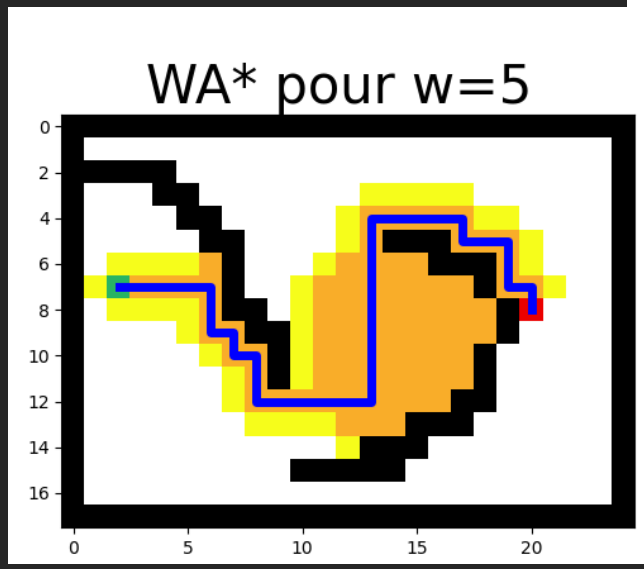
# WA* pour $w = 1$



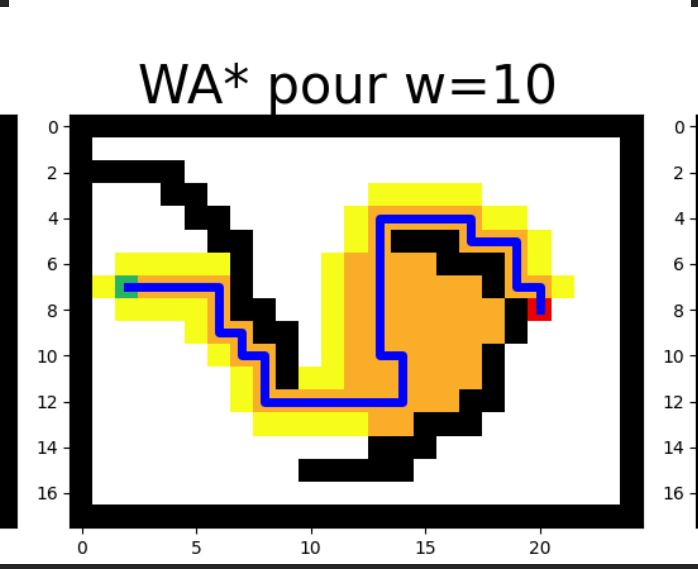
# WA* pour $w \gg 1$



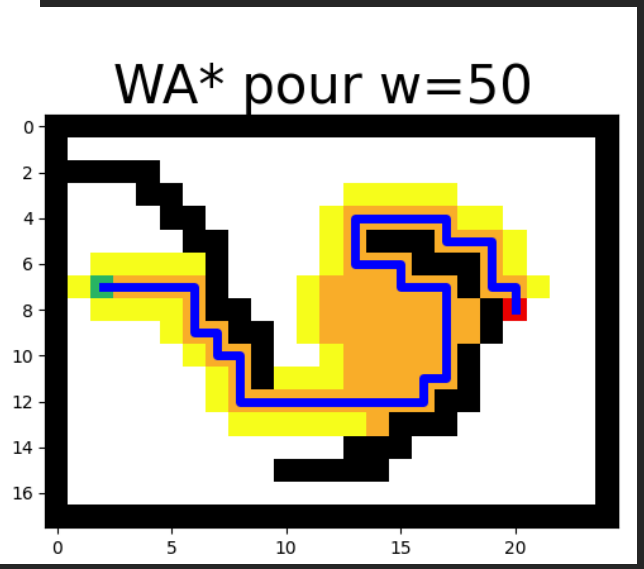

Coût : 43




Coût : 35

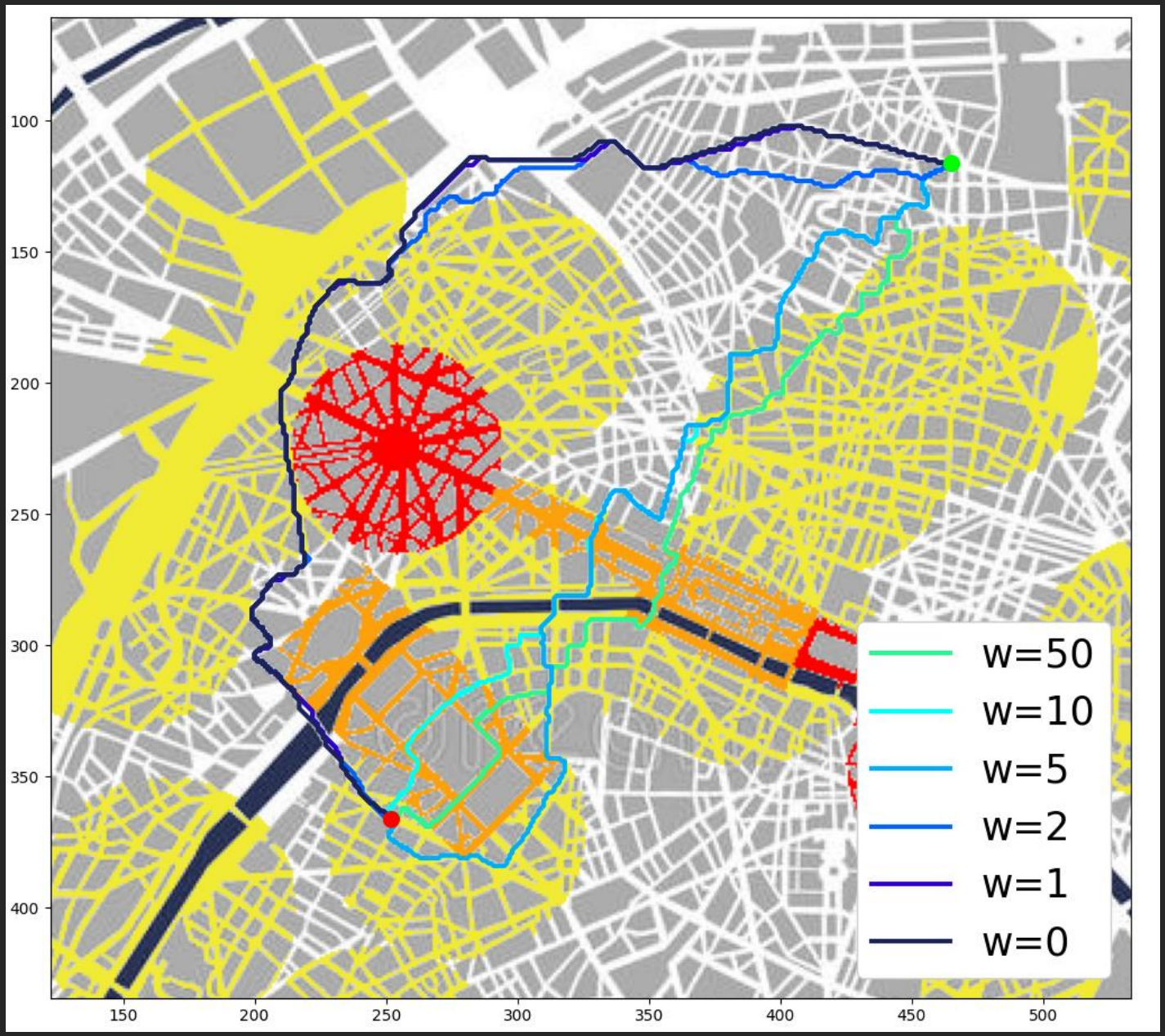



Coût : 37




Coût : 43

# Pour Paris…


w=50
w=10
w=5
w=2
w=1
w=0
100
150
200
250
300
350
400
150
200
250
300
350
400
450
500

# Petit comparatif

| Pondération $w$ | $w = 0$ | $w = 1$ | $w = 2$ | $w = 5$ | $w = 10$ | $w = 50$ |
|---|---|---|---|---|---|---|
| Temps calcul moyen (en secondes) | 115.8 | 64.6 | 27.7 | 2.1 | 0.74 | 0.64 |
| Coût du chemin | 739.88 | 739.88 | 756.23 | 922.16 | 1053.44 | 1325.18 |
| Optimalité du chemin | Oui | Oui | Non | Non | Non | Non |

# Annexe 6

## Codes Python

# Codes sur un espace polygonal

```
##### Triangulation #####

def ind_voisin(s,degre,n):
    # Retourne l'indice du (s+degré) ième sommet.
    return (s+degre)%n


def produit_vec(S1,S2,M):
    # Retourne Vec(S1S2) ∧ Vec(S1M)
    V1=(S2[0]-S1[0],S2[1]-S1[1])
    V2=(M[0]-S1[0],M[1]-S1[1])
    return(V1[0]*V2[1] - V1[1]*V2[0])


def triangle(polygone,indices):
    # Retourne le triangle respectivement associé aux 3 indices parmi les sommets du polygone
    return([polygone[indices[0]],polygone[indices[1]],polygone[indices[2]]])


def point_dans_triangle(polygone,indices,M):
    # Retourne True si le point M est dans le triangle associé aux indices
    T=triangle(polygone,indices)
    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))
```

```python
def sommet_distance_max(polygone,indices):
    # 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
    n=len(polygone)
    maxi=-1
    ind_max=-1
    T=triangle(polygone,indices)
    for i in range(n):
        M=polygone[i]
        if not(i in indices) and point_dans_triangle(polygone,indices,M):
            x = abs(produit_vec(T[0],T[2],M))
            if x>maxi:
                ind_max=i
                maxi=x
    if ind_max==-1: return(None)
    else: return(ind_max)


def sommet_gauche(polygone):
    # Retourne l'indice du sommet situé le plus à gauche dans le polygone
    n=len(polygone)
    if n<3: return('problème sur le nbre de sommets')
    xg = polygone[0][0]
    ig = 0
    for i in range(1,n):
        if polygone[i][0] < xg:
            xg = polygone[i][0]
            ig = i
    return ig

# La fonction sommet_gauche est essentielle car elle permet de traiter uniquement des triangles contenus dans le polygone.
```

```python
def nouveau_polygone(polygone,i_d,i_f):
    # Retourne le sous-polygone allant du sommet d'indice i_d au sommet d'indice i_f
    n=len(polygone)
    if n==3:
        return(polygone)
    else:
        new_p=[]
        i=i_d
        while i != i_f :
            new_p.append(polygone[i])
            i=ind_voisin(i,1,n)
        new_p.append(polygone[i_f])
        return(new_p)
```

```python
def trianguler(polygone):
    n0 = len(polygone)
    if n0 < 3:
        return("Trop peu de sommets")
    else:
        def trianguler_rec(polygone,liste_triangles):
            n=len(polygone)
            i1=sommet_gauche(polygone)
            i0=ind_voisin(i1,-1,n)
            i2=ind_voisin(i1,1,n)
            T=triangle(polygone,[i0,i1,i2])              #On traite CE triangle
            i=sommet_distance_max(polygone,[i0,i1,i2])
            if i==None:     # Si i = None : aucun sommet du polygone est contenu dans le traingle
                liste_triangles.append(T)
                poly1=nouveau_polygone(polygone,i2,i0)
                if len(poly1)==3:
                    liste_triangles.append(poly1)
                else:       # On procède récursivement sur les sommets qui restent
                    trianguler_rec(poly1,liste_triangles)
            else:           # Sinon : il y a (au moins) un sommet contenu dans le triangle
                poly1=nouveau_polygone(polygone,i1,i)
                poly2=nouveau_polygone(polygone,i,i1)
                if len(poly1)==3:
                    liste_triangles.append(poly1)
                else:
                    trianguler_rec(poly1,liste_triangles)
                if len(poly2)==3:
                    liste_triangles.append(poly2)
                else:
                    trianguler_rec(poly2,liste_triangles)
                # On procède récursivement sur les deux polygones ainsi créés
            return(liste_triangles)

        return(trianguler_rec(polygone,[]))

# La fonction prend en argument un polygone et retourne ainsi une triangulation de ce dernier :
une liste des triangles qui partitionne le polygone.
```

```
##### 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)
```

```python
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)


def centre_triangle(T):
    # Retourne le centre du triangle T : la moyenne respective des abscisses et des ordonnées des
sommets du triangle
    t_moy=np.round(np.mean(T,axis=0),2)
    return([t_moy[0],t_moy[1]])
```

```python
def PCC_Triangulation(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
    chemin=[centre_triangle(l_T[i])]
    while i != D:   # On parcourt la liste des Pères pour construire le chemin résultant
        i=p[i]
        T=l_T[i]
        x=centre_triangle(T)
        chemin = [x] + chemin
    chemin=[depart]+chemin+[arrivee]
    return(chemin)
```

Il y a malheureusement des problèmes qui peuvent survenir si la fonction reste telle quelle…

## 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))
```

```python
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)
            if d1<=d2:
                chemin = [S_com[0]] + chemin
            else:
                chemin = [S_com[1]] + chemin
        else:
            chemin = [x] + chemin

    chemin=[depart]+chemin+[arrivee]
    return(chemin)
```

```
##### Dijkstra #####

def dans_polygone(i,j,poly):
    # Retourne True si le segment [ij] reste dans l'intérieur du polygone
    n=len(poly)
    if abs(i-j)==1 or (i>=n or j>=n):
        return True
    else:
        X=poly[i]
        Y=poly[j]
        P0=poly[i-1]
        P1=poly[(i+1)%n]
        if orientation(P0,X,P1)== 1:
            return (orientation(P0,X,P1) == orientation(P0,X,Y) and orientation(P1,X,P0) ==
orientation(P1,X,Y))
        else:
            return (orientation(P0,X,P1) != orientation(P0,X,Y) or orientation(P1,X,P0) !=
orientation(P1,X,Y))


def graphe(poly,depart,arrivee):
    # Retourne la matrice d'adjacence du graphe des déplacements entre les sommets du polygone
    S=poly+[depart]+[arrivee]
    n=len(S)
    g=(-1)*np.ones((n,n),float)
    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):
                g[i,j]=distance_2points(S[i],S[j])
                g[j,i]=g[i,j]
    return(g)
```

```python
def successeurs(s,G):
    # Retourne l'ensemble des sommets accessibles depuis le sommet s
    l=[]
    for i in range(np.shape(G)[0]):
        if G[s,i]!= -1 :
            l.append(i)
    return(l)


def ajouter_file(file,sommet,distance):
    # Retourne la file de priorité dans laquelle on a ajouté le sommet
    b=len(file)-1
    a=0
    while a<=b:
        m=(a+b)//2
        if sommet==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]+[(sommet,distance)]+file[a:]
    return(file)
```

```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)
```

```
##### 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)
```

```python
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)
```

```
#####    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)
```

```python
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)
```

```
##### Tracé de tous les chemins trouvés #####

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)
```

# Codes sur un espace quadrillé

```python
cmax = 100
deplacement_diag=True       # vaut True si on autorise les déplacements diagonaux, False sinon.


##### Parcours en largeur (ou Breadth First Search) #####

def voisins(M,case,bool):
    # 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
    # Le booléen permet d'ajouter ou non les cases diagonales dans les voisins
    i,j=case
    v=[(i-1,j),(i,j+1),(i+1,j),(i,j-1)]                # haut, droite, bas, gauche
    l_cout=[cout(M,v[k]) for k in range(4)]
    vf=[]
    for k in range(4):
        if l_cout[k]!=cmax:
            vf.append((v[k],False))
    if bool:
        vd=[(i-1,j+1),(i+1,j+1),(i+1,j-1),(i-1,j-1)]    # On commence en haut à droite...
        for k in range(4):
            if l_cout[k]!=cmax and l_cout[(k+1)%4]!=cmax and cout(M,vd[k])!=cmax:
                vf.append((vd[k],True))
    return(vf)
```

```python
def parcours_en_largeur(M,depart,arrivee):
    # depart et arrivee sous la forme d'un tuple
    p,q=np.shape(M)[:2]
    file=[depart]
    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)

    k=1
    while file!=[]:
        if k%10000==0 : print(k)
        pivot=file.pop(0)
        C[pivot]="noir"
        if pivot==arrivee:
            break
        for (v,_) in voisins(M,pivot,deplacement_diag):
            if C[v]== "blanc":
                file.append(v)
                P[v]=pivot
                C[v]="gris"
        k+=1
    return(P)

def PCC_BFS(M,depart,arrivee):
    P=parcours_en_largeur(M,depart,arrivee)
    x=arrivee
    chemin=[x]
    while x != depart:
        x=P[x]
        chemin = [x] + chemin
    return(chemin)
```

```
##### 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)
    k=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"
        k+=1
    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)
```

```
##### Dijkstra #####

def Dijkstra(M,depart,arrivee):
    p,q=np.shape(M)[:2]
    file=[(depart,0)]

    C=np.array([["blanc" for _ in range(q)] for _ in range(p)])
    P=np.array([[None for _ in range(q)] for _ in range(p)])
    D=np.array([[np.infty for _ in range(q)] for _ in range(p)])

    P[depart]=(-1,-1)
    D[depart]=0
    k=1
    while file!=[]:
        pivot=file.pop(0)[0]
        C[pivot]="noir"
        if pivot==arrivee:
            break
        for (v,d) in voisins(M,pivot,deplacement_diag):
            if d:
                newD=np.round(D[pivot]+cout(M,v)*sqrt(2),2)
            else:
                newD=np.round(D[pivot]+cout(M,v),2)
            if C[v]=="blanc" and newD < D[v]:
                D[v]=newD
                P[v]=pivot
                file=ajouter_file(file,v,newD)
        k+=1
    return(P)

def PCC_Dijkstra(M,depart,arrivee):
    P=Dijkstra(M,depart,arrivee)
    x=arrivee
    chemin=[x]
    while x != depart:
        x=P[x]
        chemin = [x] + chemin
    return(chemin)
```

```
#####  A*  #####

def Astar(M,depart,arrivee):
    p,q=np.shape(M)[:2]
    file=[(depart,0)]

    C=np.array([["blanc" for _ in range(q)] for _ in range(p)])
    P=np.array([[None for _ in range(q)] for _ in range(p)])
    D=np.array([[np.infty for _ in range(q)] for _ in range(p)])

    P[depart]=(-1,-1)
    D[depart]=0
    k=1
    while file!=[]:
        pivot=file.pop(0)[0]
        C[pivot]="noir"
        if pivot==arrivee:
            break
        for (v,d) in voisins(M,pivot,deplacement_diag):
            if d:
                newD=np.round(D[pivot]+cout(M,v)*sqrt(2),2)
            else:
                newD=np.round(D[pivot]+cout(M,v),2)
            if C[v]=="blanc" and newD < D[v]:
                D[v]=newD
                P[v]=pivot
                d=np.round(newD + distance_2points(v,arrivee),2)
                file=ajouter_file(file,v,d)
        k+=1
    return(P)

def PCC_Astar(M,depart,arrivee):
    P=Astar(M,depart,arrivee)
    x=arrivee
    chemin=[x]
    while x != depart:
        x=P[x]
        chemin = [x] + chemin
    return(chemin)
```

```
##### Carte aléatoire #####

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]
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]
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)

chemin_BFS=PCC_BFS(A,depart,arrivee)
chemin_GBFS=PCC_GBFS(A,depart,arrivee)
chemin_Dijkstra=PCC_Dijkstra(A,depart,arrivee)
chemin_Astar=PCC_Astar(A,depart,arrivee)

plt.plot(np.array(chemin_BFS)[:,1],np.array(chemin_BFS)[:,0],color='magenta',linewidth=4)
plt.plot(np.array(chemin_GBFS)[:,1],np.array(chemin_GBFS)[:,0],color='orange',linewidth=4)
plt.plot(np.array(chemin_Dijkstra)[:,1],np.array(chemin_Dijkstra)[:,0],color='yellow',linewidth=4)
plt.plot(np.array(chemin_Astar)[:,1],np.array(chemin_Astar)[:,0],color='red',linewidth=4)

plt.plot(depart[1],depart[0],color="g",marker='o',markersize=4)
plt.plot(arrivee[1],arrivee[0],color='r',marker='o',markersize=4)

agrandir1()
plt.show()
```

```
##### Paris avec quelques embouteillages #####

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 inter_cercle(M,centre,rayon,couleur):
    # Trace un disque sur M en choisissant un centre, un rayon, et une couleur souhaités.
    cx,cy=centre
    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)]
    for l_case in tab_case:
        for case in l_case:
            if distance_2points(case,centre)<=rayon and np.mean(M[case])>=220:
                M[case]=couleur

def inter_rectangle(M,coin,theta,largeur,longueur,couleur):
    # Trace un rectangle plein sur M en choisissant un coin inférieur droit, une inclinaison thêta,
unelongueur, une largeur et une couleur souhaités.
    cx,cy=coin
    tab_case=[]
    for i in range(cx-longueur,cx):
        for j in range(cy,cy+largeur):
            vec=vecteur((i,j),coin)
            norme=np.linalg.norm(vec)
            v1 = vec / norme
            v2 = [-1,0]
            produit = np.dot(v1,v2)
            angle = abs(np.arccos(produit))
            (i2,j2)=(round(coin[0]+norme*cos(theta+angle)),round(coin[1]+norme*sin(theta+angle)))
            tab_case.append((i2,j2))
            tab_case += voisins_directs((i2,j2),False)

    for case in tab_case:
        if np.mean(M[case])>=220:
            M[case]=couleur
```

```
c1=np.array([255,0,0])          # Rouge
c2=np.array([124,50,0])         # Marron
c3=np.array([252,161,11])       # Orange
c4=np.array([241,235,54])       # Jaune

def cout(M,case):
    moyenne=np.mean(M[case])
    if np.all(M[case]==c1):     # Rouge
        return(8)
    elif np.all(M[case]==c3):   # Orange
        return(6)
    elif np.all(M[case]==c4):   # Jaune
        return(2)
    else:
        if moyenne<220:
            return(cmax)
        else:
            return(1)
```

```
## Image de travail ##

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

img0 = mpimg.imread (link + image + '.jpg')
img = mur_ext(np.array(img0,dtype=int))

plt.imshow(img)

depart=(219,699)
arrivee=(445,208)

inter_cercle(img,(345,465),40,c1)
inter_cercle(img,(225,254),40,c1)
inter_cercle(img,(378,651),40,c1)

inter_rectangle(img,(381,280),pi/4,52,106,c3)
inter_rectangle(img,(317,427),1.12,20,23,c1)
inter_rectangle(img,(317,401),1.12,33,65,c3)
inter_rectangle(img,(278,348),1.12,16,80,c3)

for _ in range(20):     # Les embouteillages jaune aléatoires
   a,b=r.randint(100,550),r.randint(100,700)
   inter_cercle(img,(a,b),50,c4)

A=c.deepcopy(img)

agrandir1()
plt.show()
```

```
## Tracés des chemins obtenus ##

chemin_BFS=PCC_BFS(A,depart,arrivee)
chemin_GBFS=PCC_GBFS(A,depart,arrivee)
chemin_Dijkstra=PCC_Dijkstra(A,depart,arrivee)
chemin_Astar=PCC_Astar(A,depart,arrivee)

plt.imshow(A)

plt.plot(np.array(chemin_BFS)[:,1],np.array(chemin_BFS)[:,0],color='blue',linewidth=3)
plt.plot(np.array(chemin_GBFS)[:,1],np.array(chemin_GBFS)[:,0],color='cyan',linewidth=3)
plt.plot(np.array(chemin_Dijkstra)[:,1],np.array(chemin_Dijkstra)[:,0],color='magenta',linewidth=3)
plt.plot(np.array(chemin_Astar)[:,1],np.array(chemin_Astar)[:,0],color='green',linewidth=3)

plt.plot(depart[1],depart[0],color="lime",marker='o',markersize=10)
plt.plot(arrivee[1],arrivee[0],color='r',marker='o',markersize=10)

agrandir1()
plt.show()
```