TIPE: Le problème de routage de véhicules VRP

Zakaria BHEDDAR

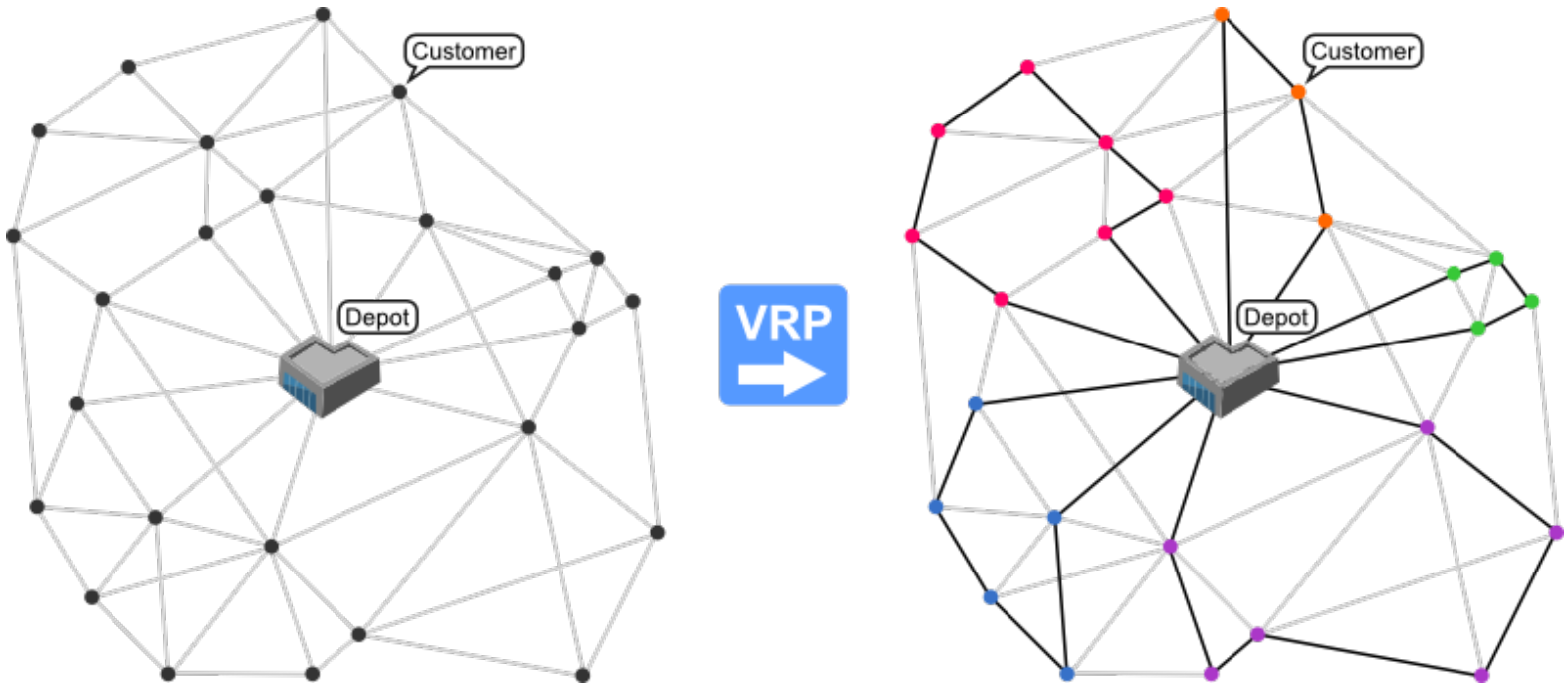
Mathématiques appliquées et Informatique pratique

Session 2023

Plan de l'exposé

Définition

Introduction



Exemple des chemins trouvés par la VRP

Définition

Problématique

- Le VRP est NP-Complet
- Le nombre de chemins possibles pour n villes ou destinations est $(n-1)!$

Définition

Problématique

- Pour 31 villes, on a $30!$ possibilités
- Si un ordinateur peut faire 4 millards d'opérations par seconde, le temps nécessaire pour calculer les posibilités est :

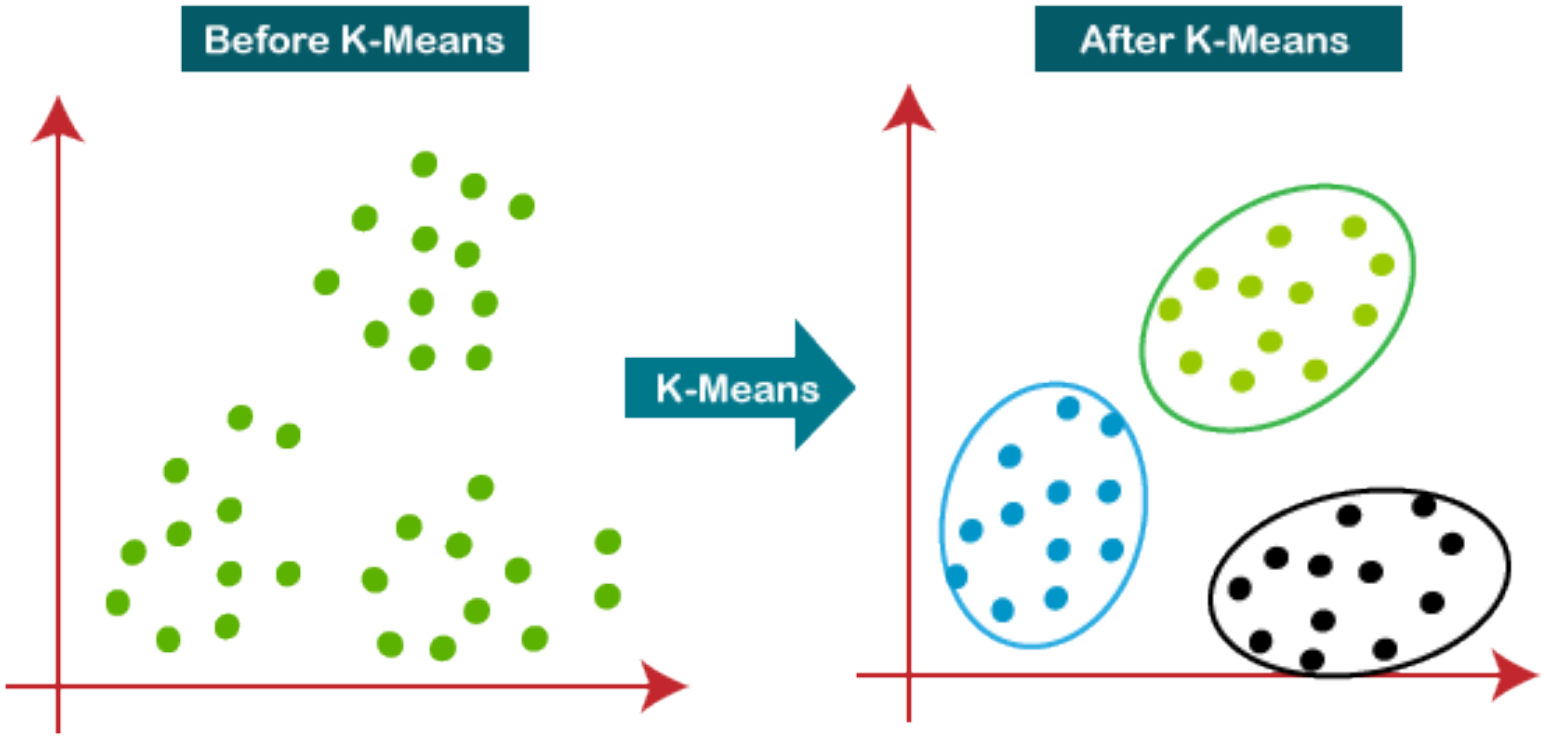
$$\frac{30!}{4 \cdot 10^9} \text{ secondes} \approx 10^{22} \text{ années}$$

Méthode de résolution

Algorithm Résolution

1: Début
2: Lire : nombre de véhicules, matrice de distance
3: kmeans pour k=nombre de véhicules
4: **for** $groupe$ in $Kmeans$ **do**
5: Application de l'algorithme de colonie de fourmis
6: **end for**
7: Résultat finale
8: Fin

K-moyennes (k-means)



Regroupement ou partionnement des points en des groupes ou clusters par k-means

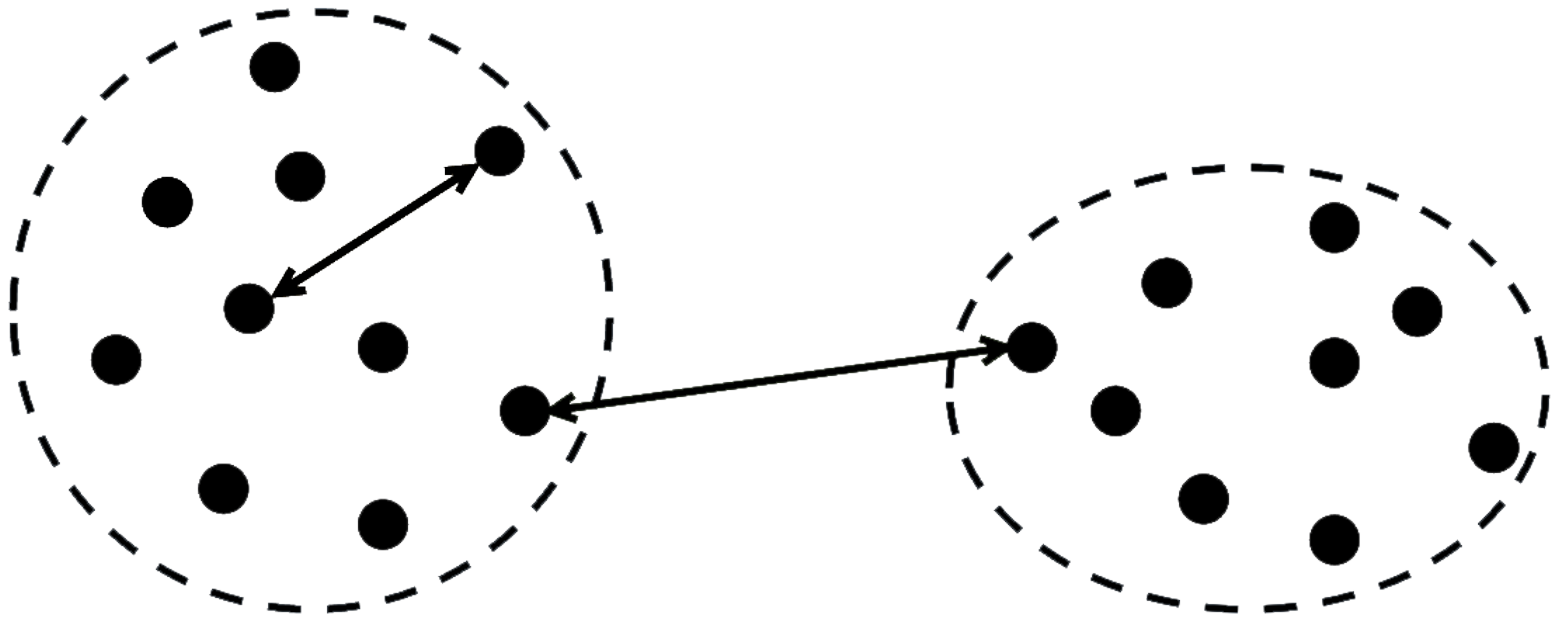
K-moyennes (k-means)

Description de l'algorithme

- Choisir k points qui représentent la position moyenne des partitions $m_1^{(1)}, ..., m_k^{(1)}$ initiales (au hasard par exemple)
- Répéter jusqu'à ce qu'il y ait convergence :
  - affecter chaque observation à la partition la plus proche $S_i^{(t)} = \{x_j : \|x_j - m_i^{(t)}\| \leq \|x_j - m_{i^*}^{(t)}\| \ \forall i^* = 1, .., k\}$
  - mettre à jour la moyenne de chaque cluster : $m_i^{(t+1)} = \frac{1}{|S_i^{(t)}|} \sum_{x_j \in S_i^{(t)}} x_j$

K-moyennes (k-means)

Optimisation du choix de K par la méthode du coude



minimiser la distance entre les points d'un cluster
$\iff$ maximiser la distance entre les centroides

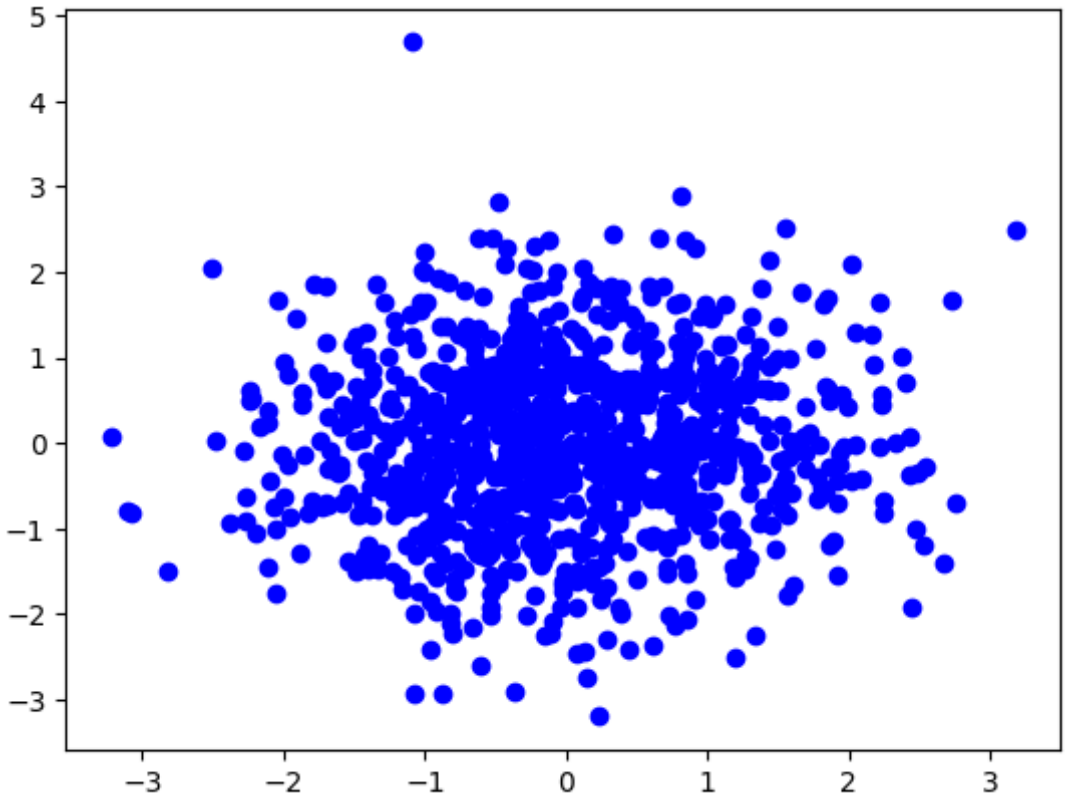
K-moyennes (k-means)
WCSS

Within Cluster Sum of Squares WCSS :

$$\sum_{P_i \in Cluster_1} distance(P_i, C1)^2 + \sum_{P_i \in Cluster_2} distance(P_i, C2)^2 + \cdots$$

K-moyennes (k-means)

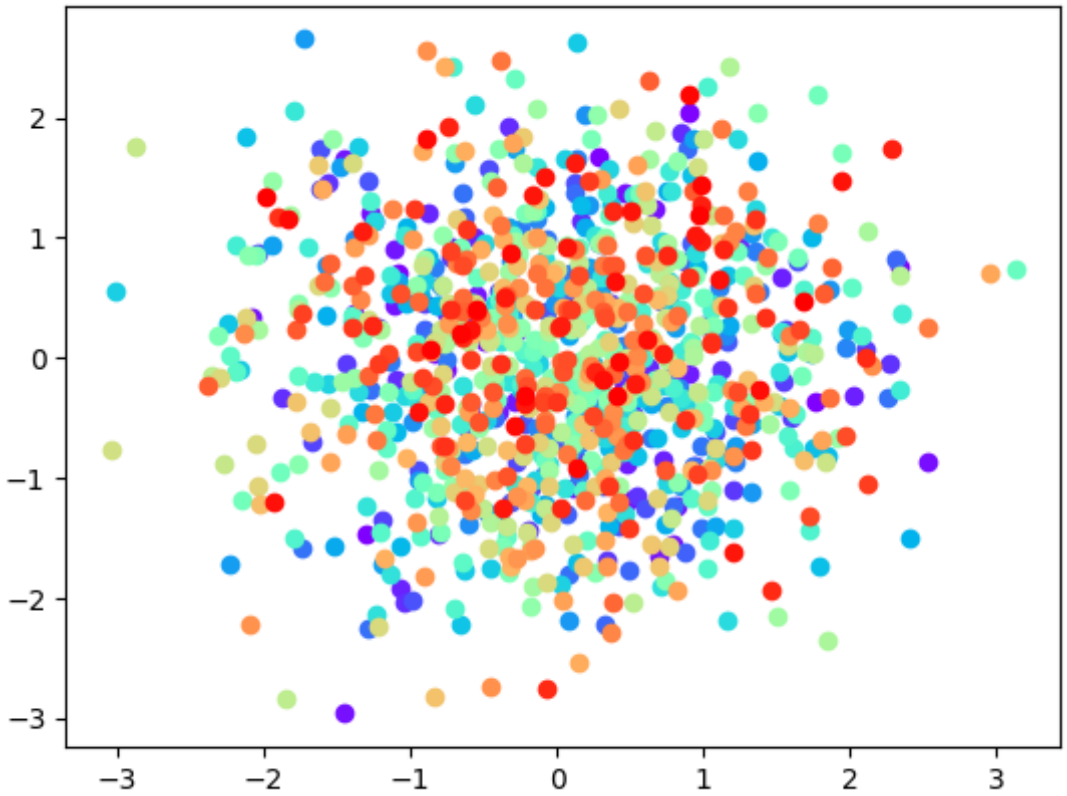
Utilité de cette optimisation 1/2



Points=1000 Clusters=1 WCSS=max

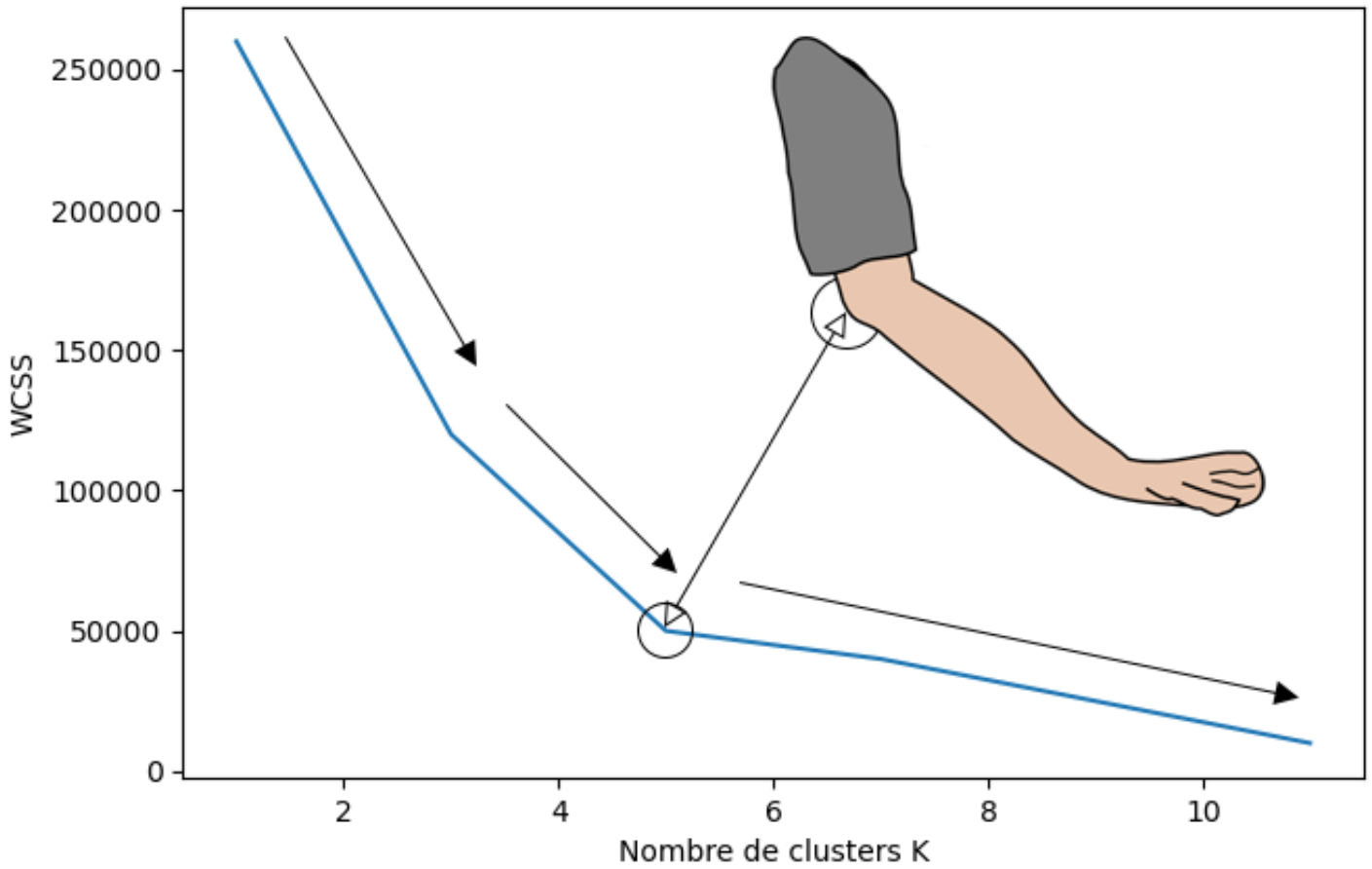
K-moyennes (k-means)

Utilité de cette optimisation 2/2

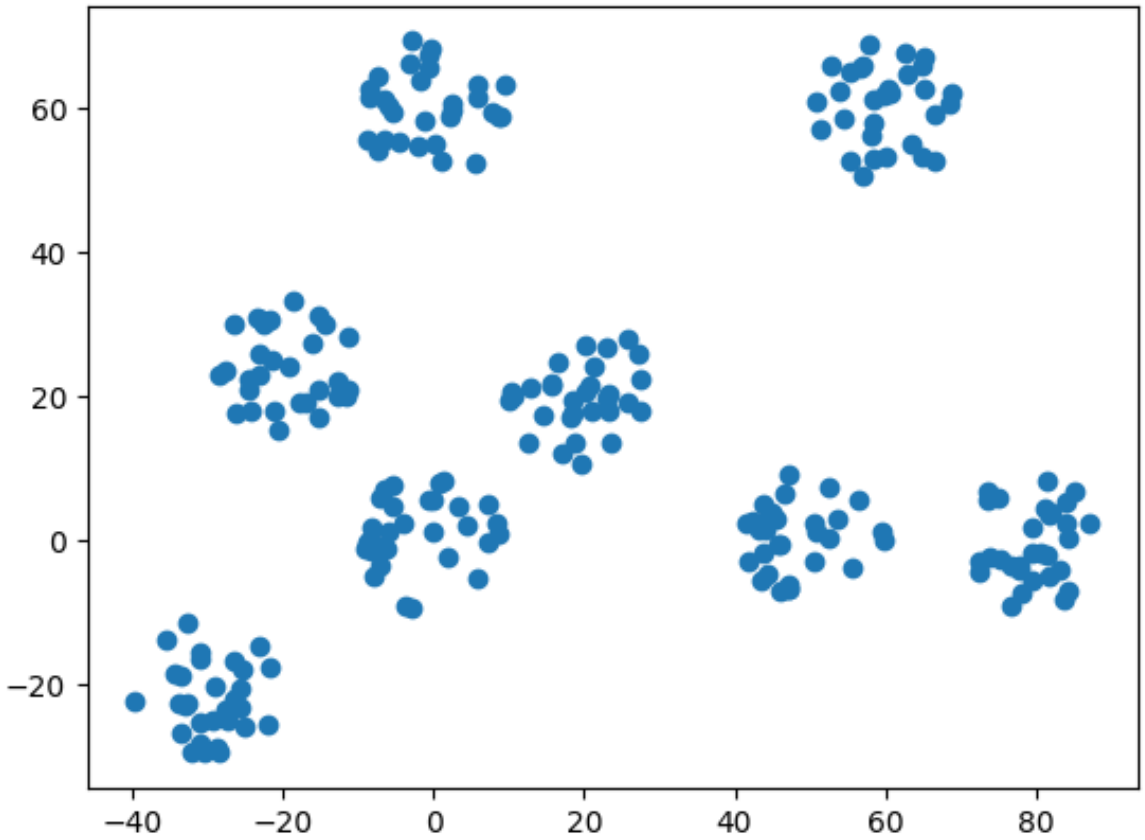


Points=1000 Clusters≈1000 WCSS≈0

méthode du coude

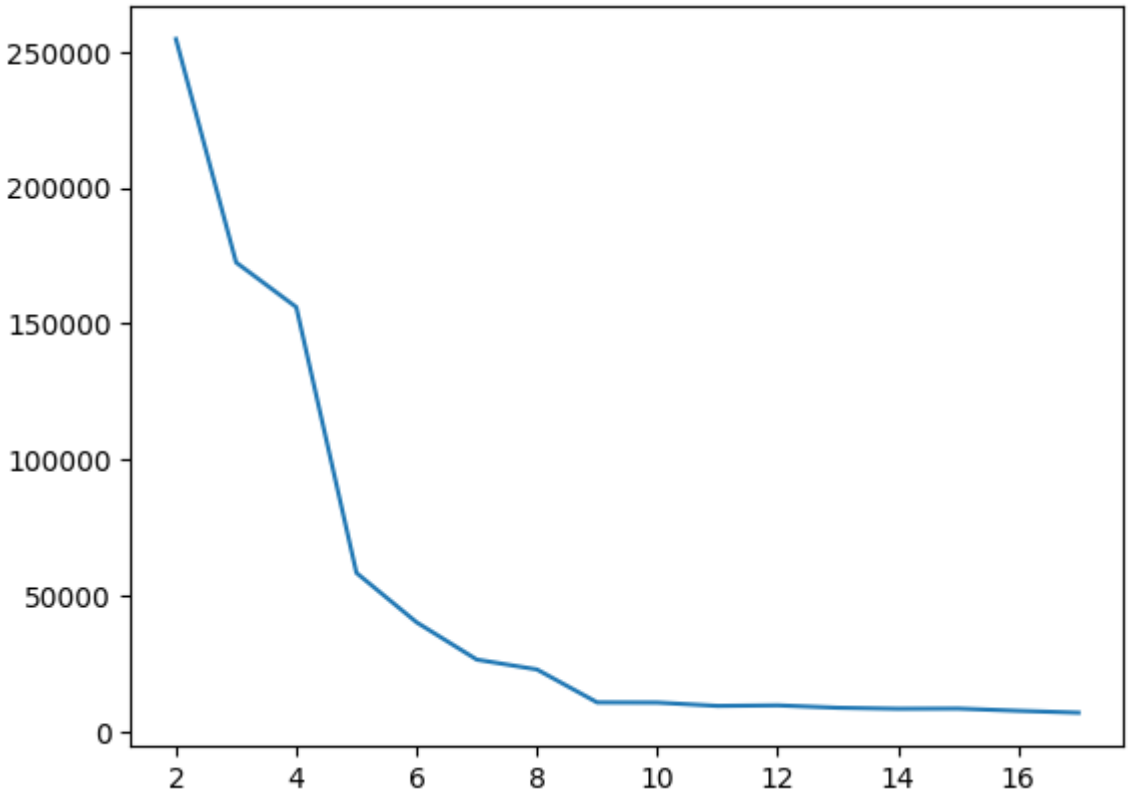


Petit exemple 1/2



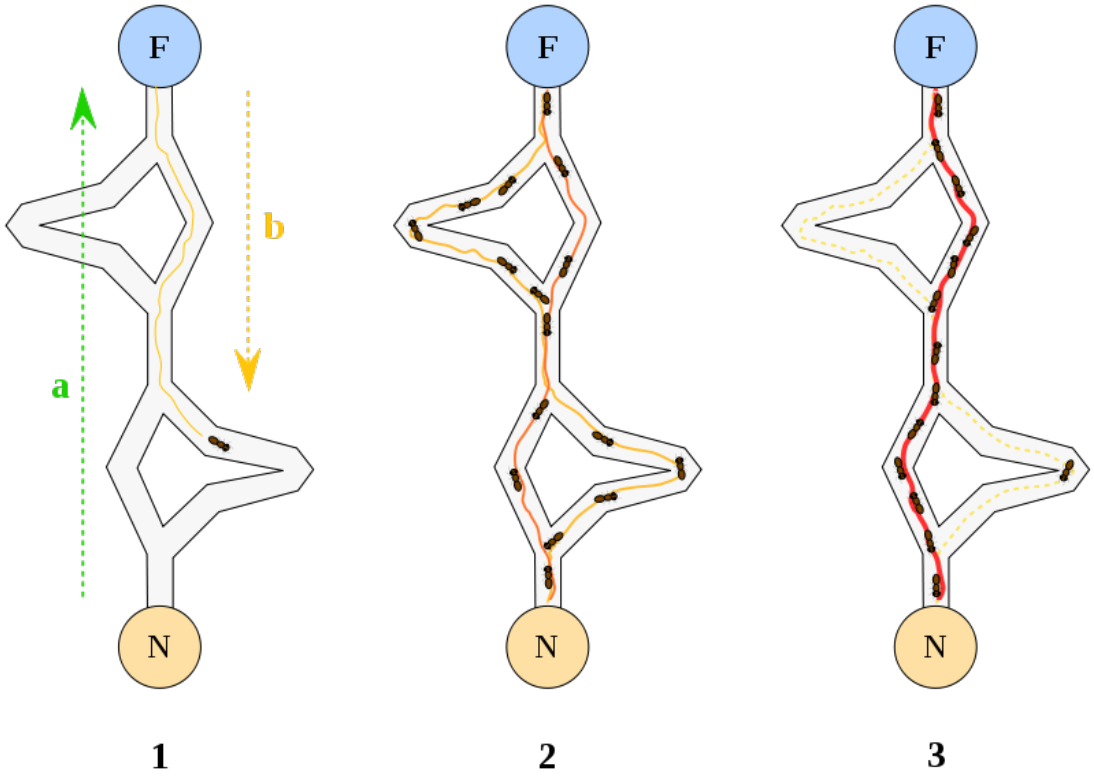
8 disques contenant 30 points aléatoires

Petit exemple 2/2



$$K_{coude} \approx 8$$

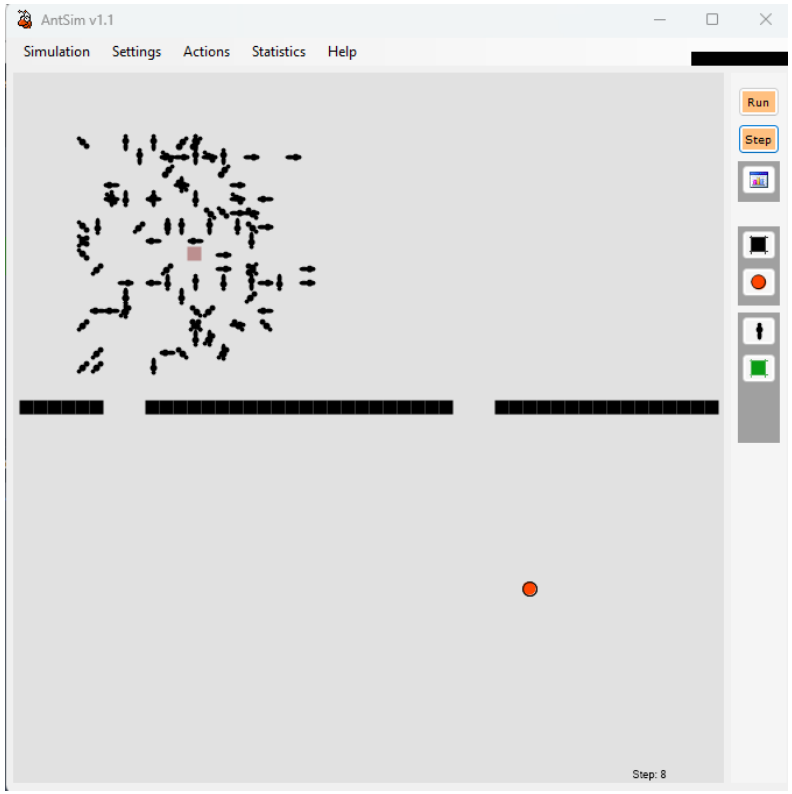
les colonies de fourmis



Expérience sur des fourmis

les colonies de fourmis

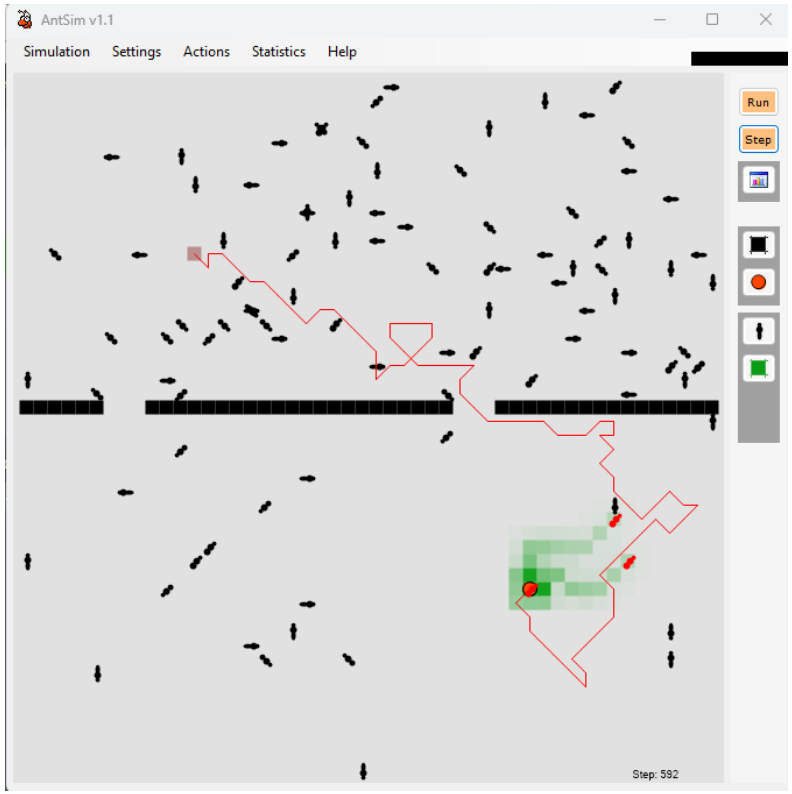
simulation AntSim1-1



départ des fourmis 1/3

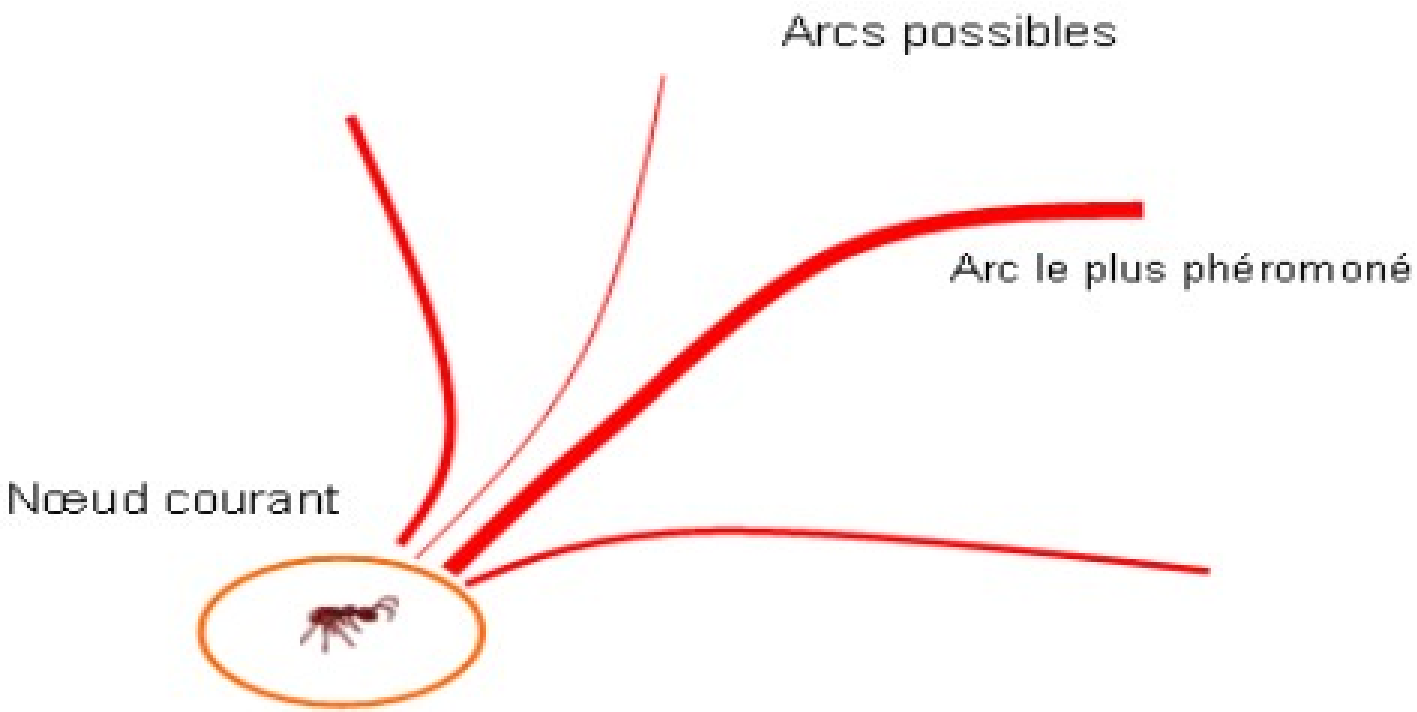
les colonies de fourmis

simulation AntSim1-1



détection de la 1ère route et dépôt du phéromone 2/3

les colonies de fourmis



une fourmis qui suit une suite de phéromone

les colonies de fourmis

simulation AntSim1-1

chemin optimisé 3/3

Données et notation

- La longueur du circuit μ représentant la distance totale parcourue :

$$L(\mu) = d_{1,p} + \sum_{i=1}^{q-1} d_{i,i+1}$$

- $\tau_{ij}(t)$ la valeur de τ_{ij} à l'instant t (la densité de phéromone)
- $\eta_{ij} = \frac{1}{d_{ij}}$ la constante indiquant la visibilité de la ville j à partir de la ville i

Algorithme de colonies de fourmis

Règle aléatoire de transition proportionnelle

$$p_{ij}^{k}(t) = \begin{cases} \dfrac{\tau_{ij}(t)^{\alpha} \cdot \eta_{ij}^{\beta}}{\sum\limits_{l \in N_i^k(t)} \tau_{il}(t)^{\alpha} \cdot \eta_{il}^{\beta}} & \text{si } j \in N_i^k \\ 0 & \text{si } j \notin N_i^k \end{cases}$$

- p_{ij}^{k} la probabilité que donne la fourmi k à un arc (i,j)
- N_i^k l'ensemble des voisins non visités du sommet i par la fourmi k dans le cycle courant
- α et β et γ sont des paramètres qui contrôlent l'importance que donne la fourmi a un arc

Algorithme de colonies de fourmis

Mise à jour des phéromones

Une fois la tournée des villes effectuée, une fourmi k dépose une quantité $\Delta\tau_{ij}^{k}(t)$ de phéromone sur chaque arête de son parcours :

$$\Delta\tau_{ij}^{k}(t) = \begin{cases} \frac{Q}{L^{k}(t)} & \text{si } (i,j) \in T^{k}(t) \\ 0 & \text{si } (i,j) \notin T^{k}(t) \end{cases}$$

où $T^{k}(t)$ est la tournée faite par la fourmi k à l'itération t, $L^{k}(t)$ la longueur du trajet et Q un paramètre de réglage

Algorithme de colonies de fourmis

Mise à jour des phéromones

À la fin de chaque itération de l'algorithme, les phéromones déposées aux itérations précédentes par les fourmis s'évaporent de :

$$\rho\tau_{ij}(t)$$

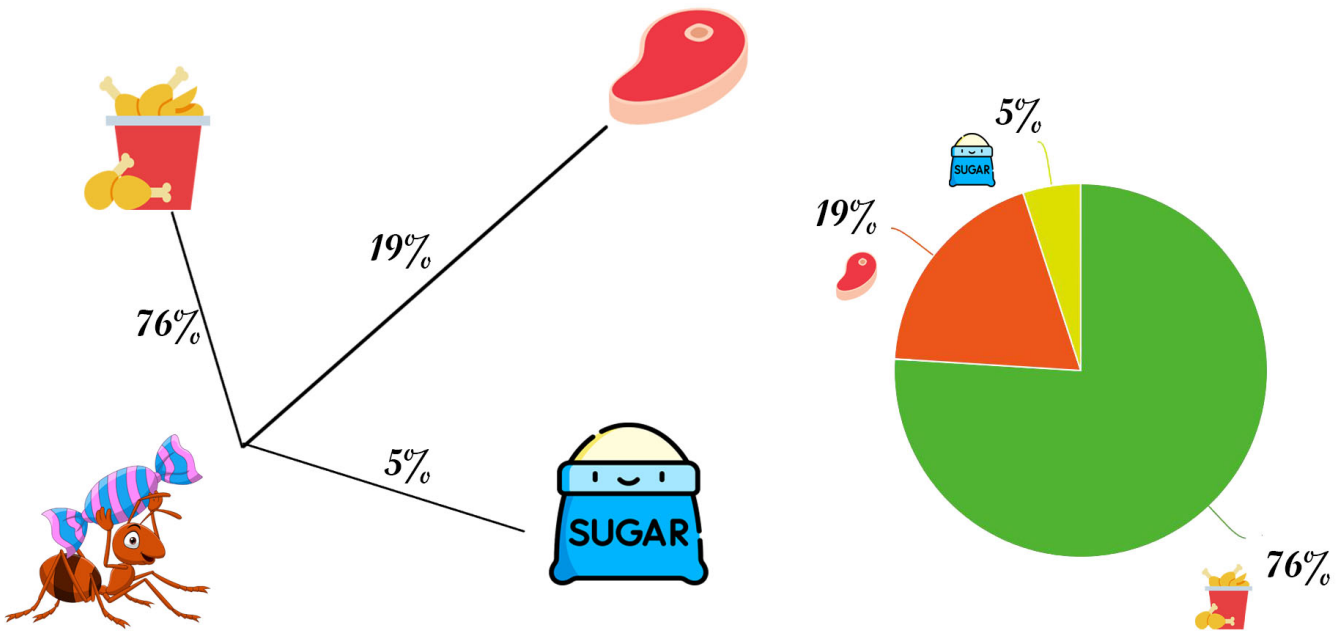
Et à la fin de l'itération, on a la somme des phéromones qui ne se sont pas évaporées et de celles qui viennent d'être déposées :

$$\tau_{ij}(t+1) = (1-\rho)\tau_{ij}(t) + \sum_{k=1}^{m} \Delta\tau_{ij}^{k}(t)$$

où m est le nombre de fourmis utilisées pour l'itération t et ρ un paramètre de réglage

Amélioration de l'ACO

La sélection par roulette



La sélection par roulette

Amélioration de l'ACO

La sélection par roulette

la probabilité	0,76	0,19	0,05
somme cumulative	1	0,24	0,05

Amélioration de l'ACO

La sélection par roulette

r est un nombre aléatoire entre $[0, 1]$ $\begin{cases} 0,24 < r \leq 1 \\ 0,05 < r \leq 0{,}24 \\ 0 \leq \text{r} \leq 0{,}05 \end{cases}$

On tire une variable aléatoire

r = 0,12

0,05 < r ≤ 0,24

donc on choisit la ville dans le sommet cumulé est égale à 0,24

Algorithm la sélection par roulette

1: Début
2: r est un nombre aléatoire entre [0, 1]
3: calculer la somme cumulative de la probabilité
4: trouver la première valeur cumulée de la probabilité supérieure à r
5: Fin

Algorithm les colonie de fourmis (ACO)

Début
Lire : nv, nf, dem, α, β, γ, Q, ρ , NImax
Calcul de la matrice des distances d_{ij}
Initialisation des matrices μ_{ij} et τ_{ij}
for $t = 1$ to $NImax$ **do**
    Placer les fourmis sur les villes
    for $k = 1$ to nf **do**
        for $v = 1$ to nv **do**
            N^k : l'ensemble des villes non visitées par la fourmi k
            Calculer la probabilité Pij selon la formule

Sélection par roulette
end for
Calculer le coût de la tournée L^k
Calculer $\Delta\tau_{ij}$
Mettre à jour les traces de phéromone
end for
L $\leftarrow$ min(L^k) la longeur minimum de chaque tour de chaque fourmis
end for
Fin

Complexité

Complexité Globale

$$\text{Complexité} = \mathcal{O}(NI_{max} \cdot n^2 \cdot m)$$

Simulation

Modélisation des villes

On choisit 144 villes aléatoirement modélisées en des points (x, y) et un dépôt ayant des voitures

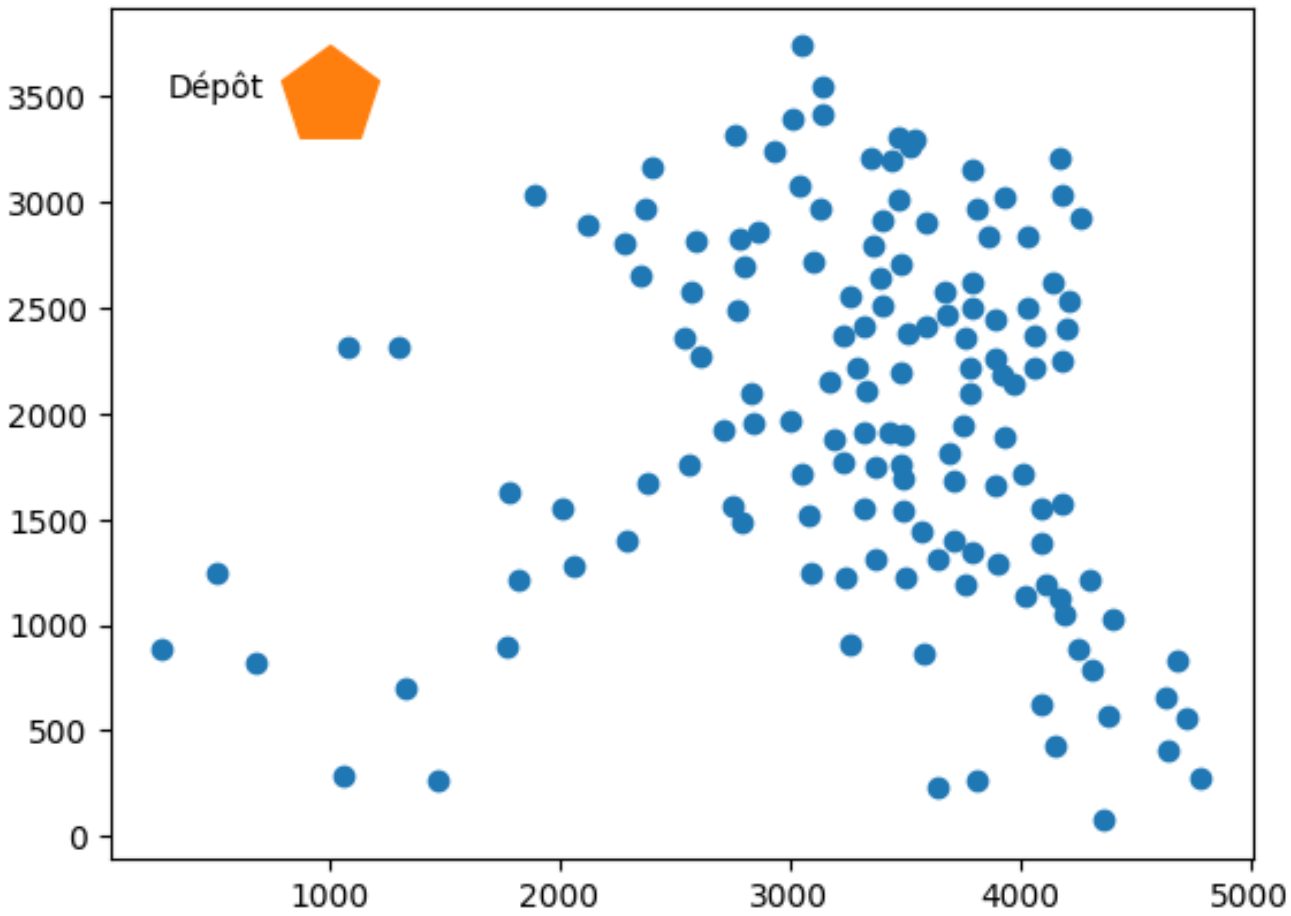
F4 | depot

ville	coords x	coords y
1	3639	1315
2	3569	1438
3	3904	1289
4	3506	1221
5	3237	1764
6	3089	1251
7	3238	1229
8	4172	1125
9	4020	1142
10	4095	626
11	4403	1022
12	4361	73

	coords x	coords y
depot	1000	3500

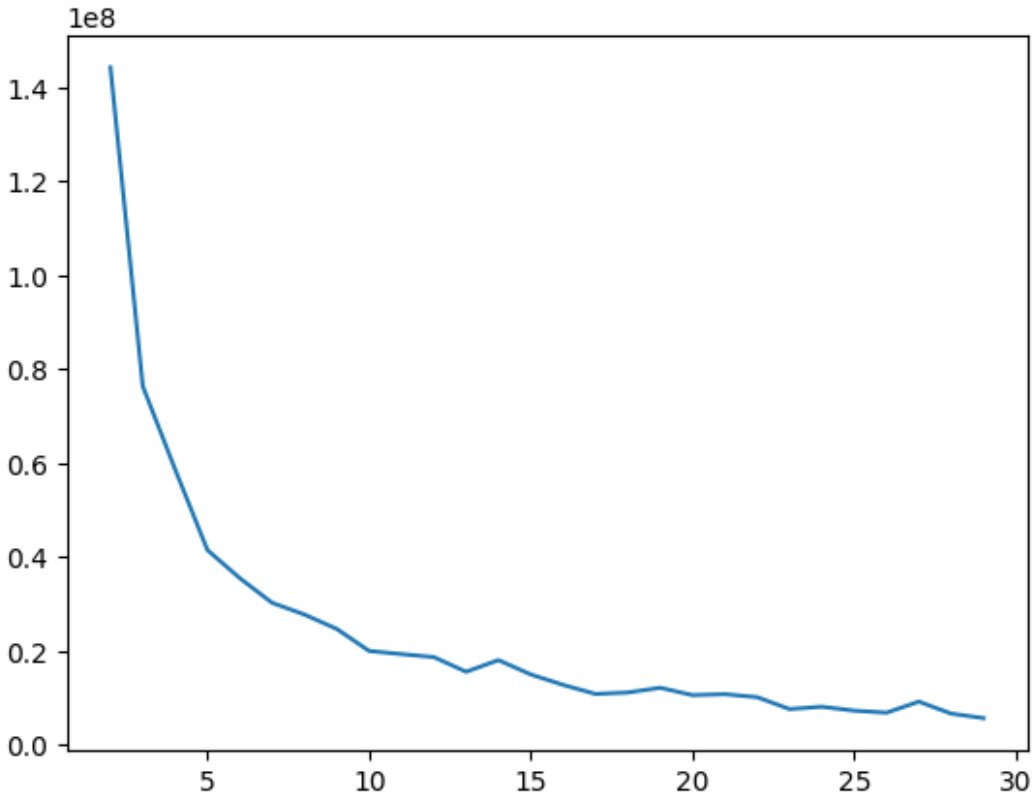
Simulation

Représentation



Simulation

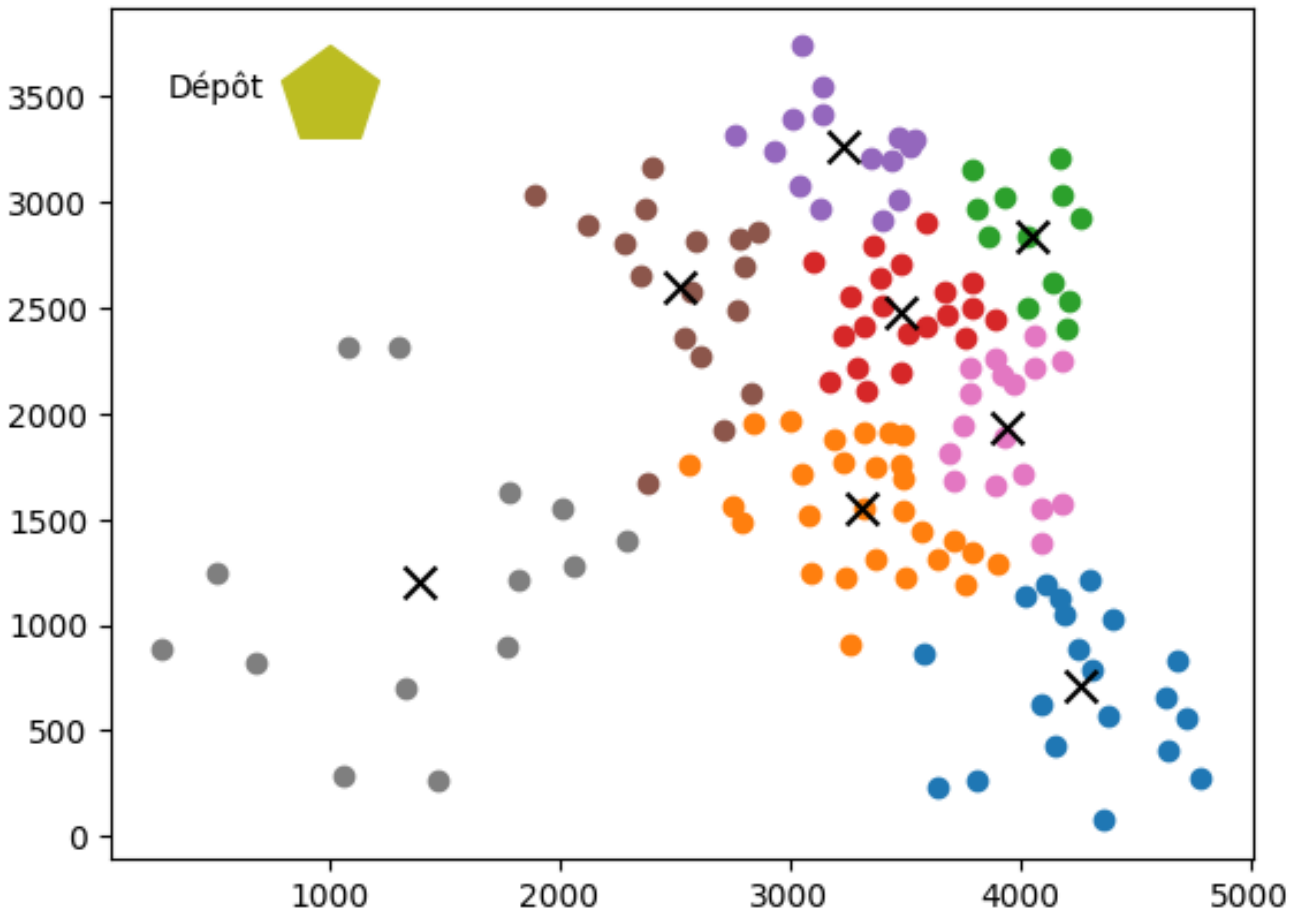
Choix du nombre de voitures optimal



$$K_{coude} \approx 8$$

Simulation

Partionnement



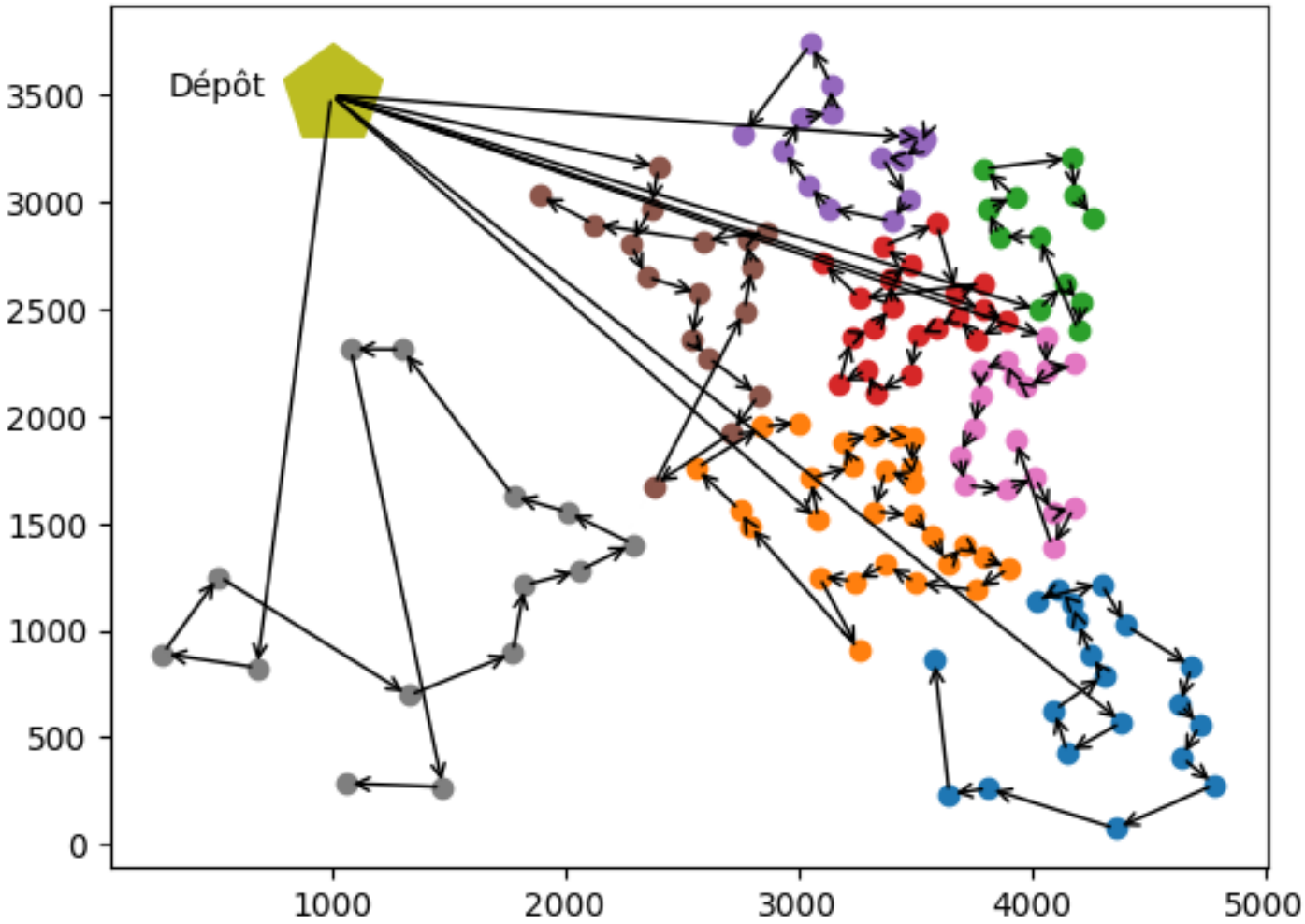
Simulation

Choix des paramètres

- le nombre de fourmis : 10
- générations : 100
- α : 1,0
- β : 10
- ρ : 0,5
- Q : 10

Temps d'exécution

Résolution


Dépôt
3500
3000
2500
2000
1500
1000
500
0
1000
2000
3000
4000
5000

Conclusion

Conclusion

Code utilisé ci-dessous

▸ Lien du code collab ici

annexes

May 31, 2023

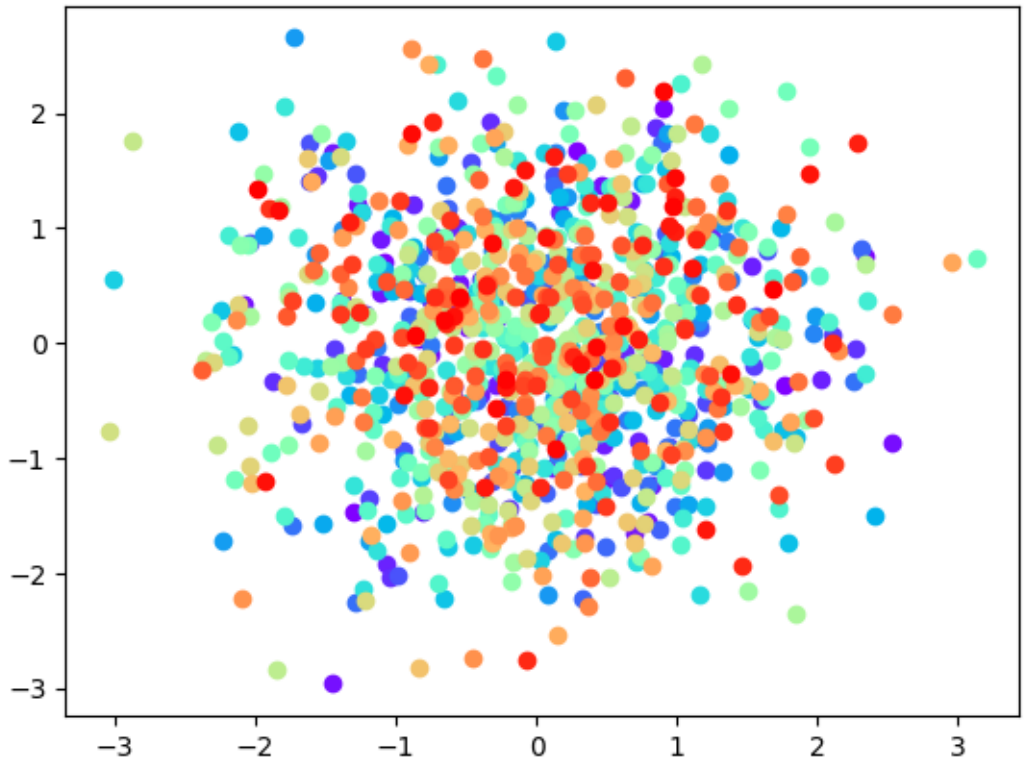
lien de tout le code: https://colab.research.google.com/drive/1AUrQZI6jfBJKkI-OPJlJ4t1v7-ON5984?usp=sharing

1 Modules nécessaires

[6]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial.distance import cdist
import matplotlib.cm as cm
import math
import random
```

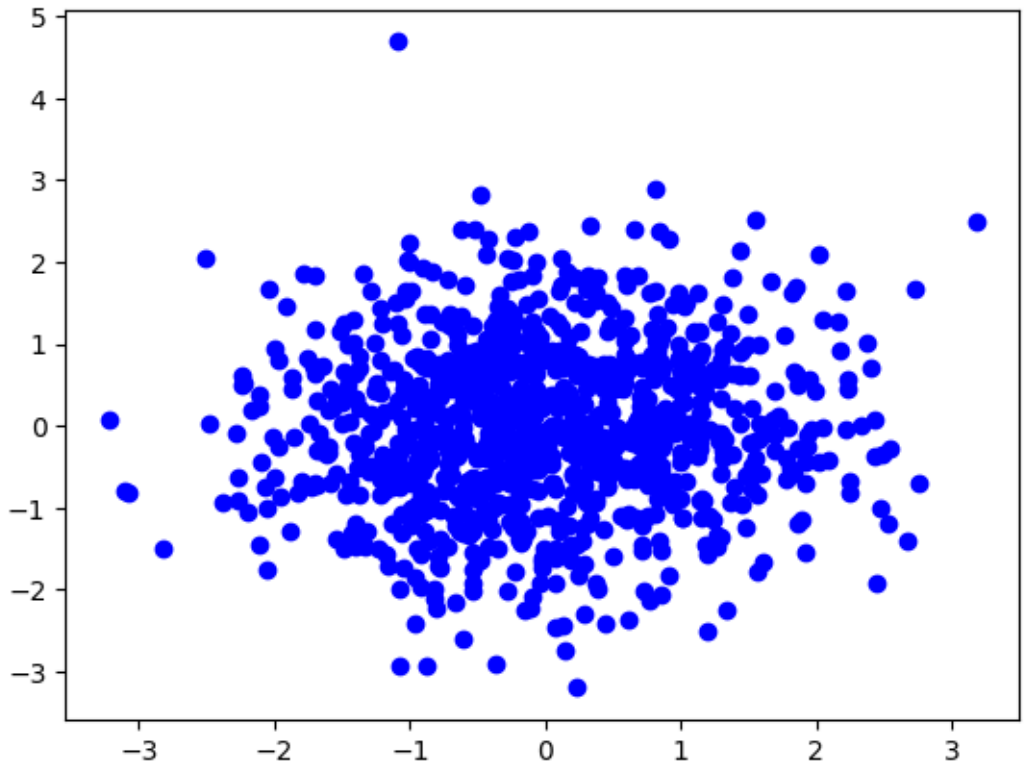
2 WCSS 1

[7]:
```
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(1000)
ys = [i+x+(i*x)**2 for i in range(1000)]
colors = cm.rainbow(np.linspace(0, 1, len(ys)))
for i in range(1000):
    a = np.random.randn(2)
    plt.scatter(a[0],a[1],color=colors[i])
plt.show()
```



3 WCSS 2

[8]:
```
import matplotlib.pyplot as plt
import numpy as np
for i in range(1000):
    a = np.random.randn(2)
    plt.scatter(a[0],a[1],color="b")
plt.show()
```



4 Distance euclidienne

[9]:
```
def distance(point1, point2):
    return np.sqrt(np.sum((point1 - point2)**2))
```

5 Algorithme de colonie de fourmis

[10]:
```
def ant_colony_optimization(points, n_ants, n_iterations, alpha, beta,␣
 ↪evaporation_rate, Q):
    n_points = len(points)
    pheromone = np.ones((n_points, n_points))
    best_path = None
    best_path_length = np.inf

    for iteration in range(n_iterations):
        paths = []
        path_lengths = []

        for ant in range(n_ants):
            visited = [False]*n_points
```

```
            current_point = np.random.randint(n_points)
            visited[current_point] = True
            path = [current_point]
            path_length = 0

            while False in visited:
                unvisited = np.where(np.logical_not(visited))[0]
                probabilities = np.zeros(len(unvisited))

                for i, unvisited_point in enumerate(unvisited):
                    probabilities[i] = pheromone[current_point,␣
↪unvisited_point]**alpha / distance(points[current_point],␣
↪points[unvisited_point])**beta

                probabilities /= np.sum(probabilities)

                next_point = np.random.choice(unvisited, p=probabilities)
                path.append(next_point)
                path_length += distance(points[current_point],␣
↪points[next_point])
                visited[next_point] = True
                current_point = next_point

            paths.append(path)
            path_lengths.append(path_length)

            if path_length < best_path_length:
                best_path = path
                best_path_length = path_length

        pheromone *= evaporation_rate

        for path, path_length in zip(paths, path_lengths):
            for i in range(n_points-1):
                pheromone[path[i], path[i+1]] += Q/path_length
            pheromone[path[-1], path[0]] += Q/path_length

    return path
```

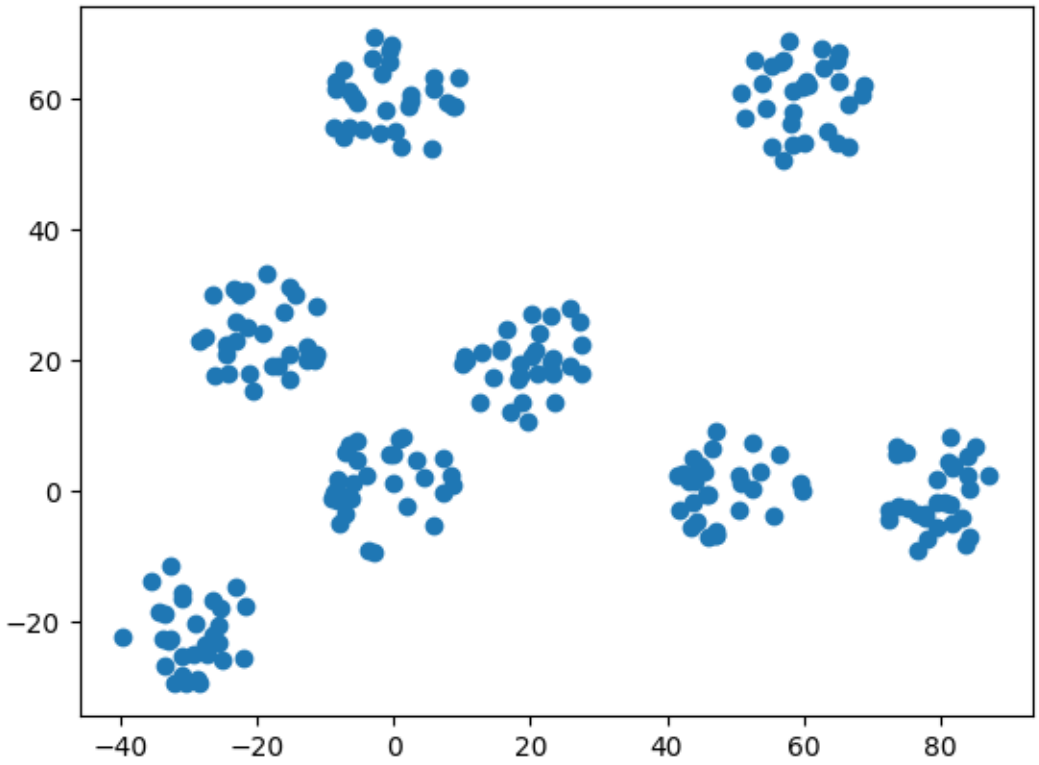
6 k-means

[11]:
```
def kmeans(x,k, no_of_iterations):
    idx = np.random.choice(len(x), k, replace=False)
    #Choix aléatoire de Centroides
    centroids = x[idx, :] #Etape 1
```

```
    #trouver la distance entre les centroïdes et tous les points de données
    distances = cdist(x, centroids ,'euclidean') #Etape 2

    #Centroïde avec la distance minimale
    points = np.array([np.argmin(i) for i in distances]) #Etape 3

    #Répéter les étapes ci-dessus pour un nombre défini d'itérations
    #Etape 4
    for _ in range(no_of_iterations):
        centroids = []
        for idx in range(k):
            #Mettre à jour les Centroides en prenant au moyen de Cluster il␣
 ↪appartient à
            temp_cent = x[points==idx].mean(axis=0)
            centroids.append(temp_cent)

        centroids = np.vstack(centroids) #Centroïdes mis à jour

        distances = cdist(x, centroids ,'euclidean')
        points = np.array([np.argmin(i) for i in distances])

    return points,centroids
```

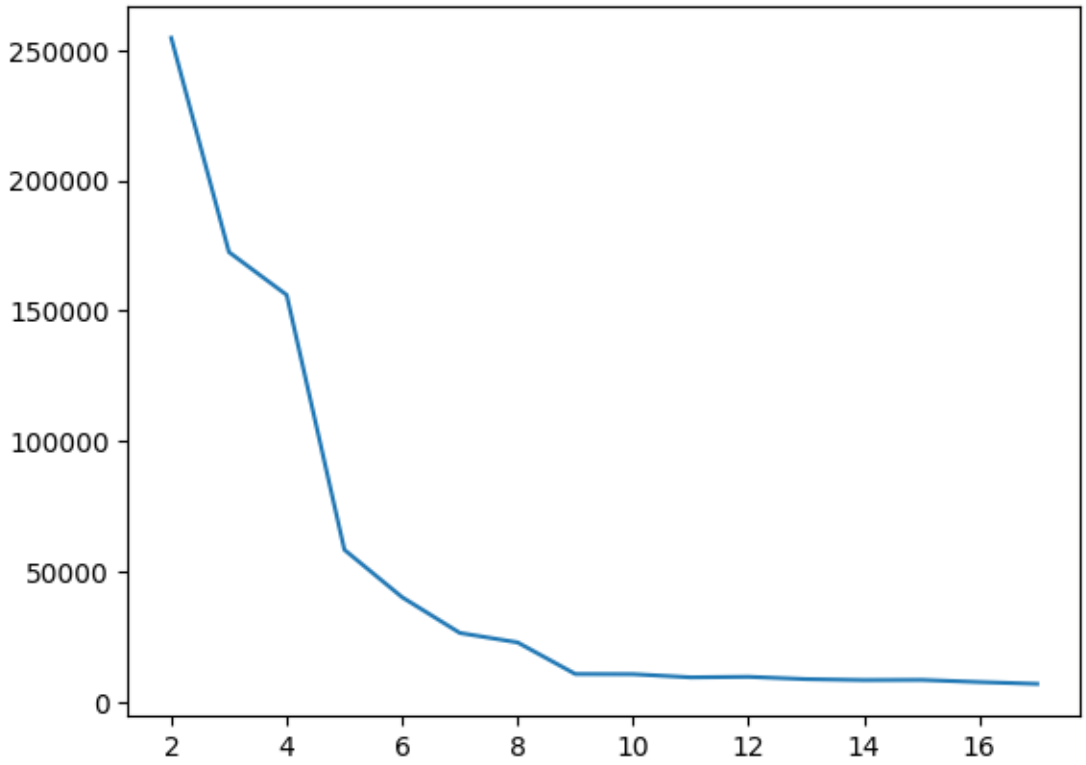
7 Petit exemple 1/2 creation de point

[16]:
```
def generer_point_aleatoire_cercle(m1,m2,n=20,R = 10):
  x = []
  y = []
  for i in range(n):
          theta = random.uniform(0,2*math.pi)
          r = random.uniform(0,R)
          x.append(math.sqrt(r*R)*math.cos(theta))
          y.append(math.sqrt(r*R)*math.sin(theta))
  return np.array(x)+m1,np.array(y)+m2
a = [0,20,0,-30,50,60,-20,80]
b = [0,20,60,-20,0,60,25,0]
aa = len(a)
axe1,axe2 = [],[]
for i in range(aa):
  s = generer_point_aleatoire_cercle(a[i],b[i],n=30,R = 10)
  axe1 = axe1+list(s[0])
  axe2 = axe2+list(s[1])
points = np.array([[axe1[i],axe2[i]] for i in range(len(axe1))])
plt.scatter(axe1,axe2)
plt.show()
```



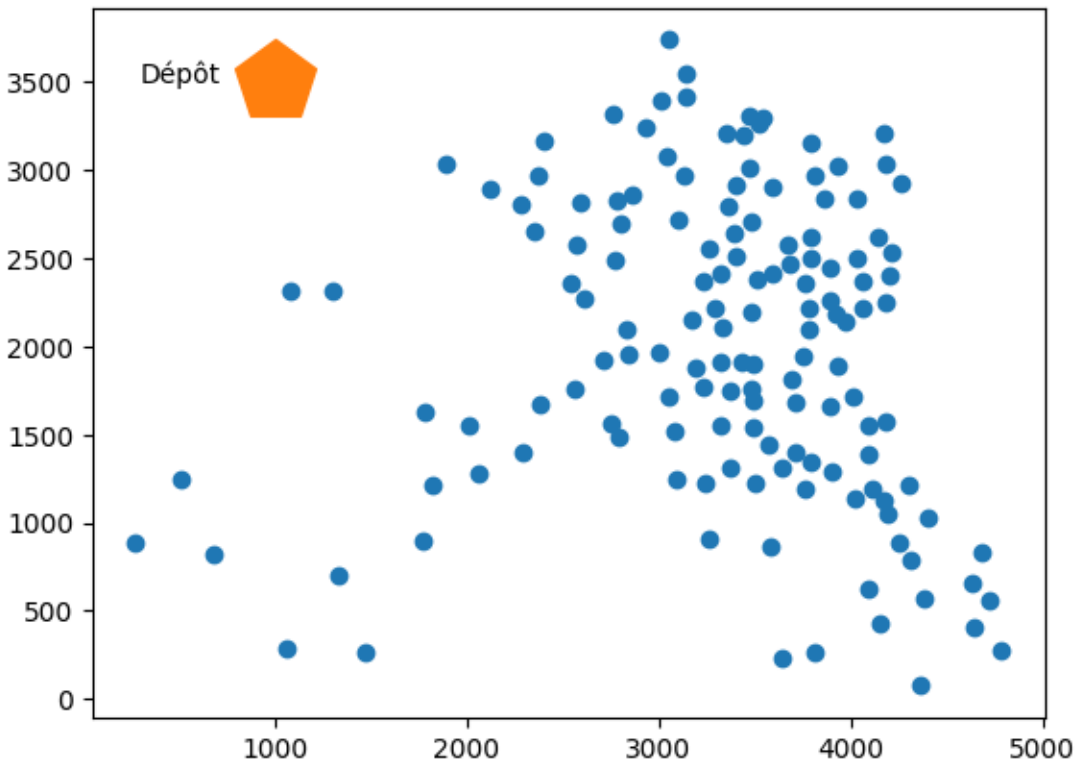
8 Petit exemple 2/2

[17]:
```
K = [2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17]
WCSS = []
for k in K:
  t = kmeans(points,k,1000)
  label = t[0]
  centroids = t[1]
  u_labels = np.unique(label)
  s = 0
  for i in u_labels:
    grp = points[label == i]
    for point in grp:
      s+=distance(point, centroids[i])**2
  WCSS.append(s)
plt.plot(K,WCSS)
plt.show()
```



9 Villes avant k-means

[18]:
```
df =␣
 ↪[(3639,1315),(3569,1438),(3904,1289),(3506,1221),(3237,1764),(3089,1251),(3238,1229),(4172,
villes = np.array(df)
depot = (1000,3500)
plt.scatter(villes[:,0],villes[:,1])
plt.scatter(depot[0],depot[1],marker = 'p',s = 1000)
plt.annotate("Dépôt",(depot[0]-700,depot[1]))
plt.show()
```

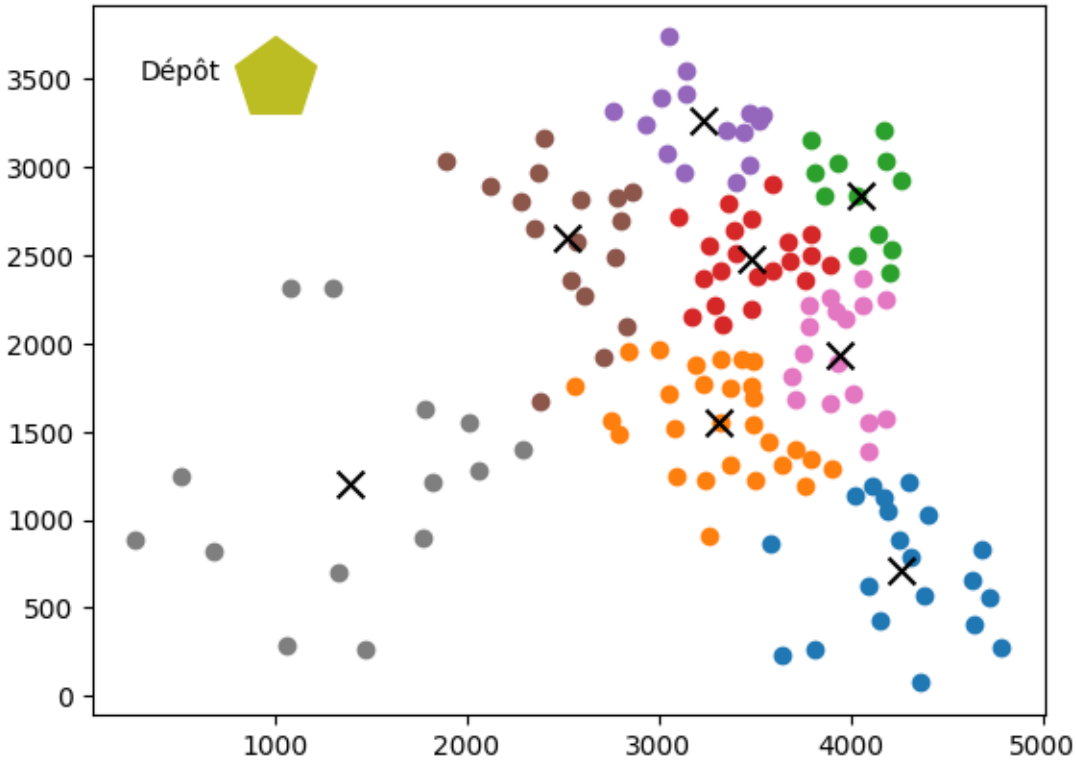


10 Choix du nombre de voitures

[ ]:
```
K = np.arange(2,30)
WCSS = []
for k in K:
  t = kmeans(villes,k,1000)
  label = t[0]
  centroids = t[1]
  u_labels = np.unique(label)
  s = 0
  for i in u_labels:
    grp = villes[label == i]
    for point in grp:
      s+=distance(point, centroids[i])**2
  WCSS.append(s)
plt.plot(K,WCSS)
plt.show()
```

11 Villes après k-means

[20]:
```
nbr_voitures = 8
t = kmeans(villes,nbr_voitures,1000)
label = t[0]
centroids = t[1]
u_labels = np.unique(label)
for i in u_labels:
    plt.scatter(villes[label == i , 0] , villes[label == i , 1] , label = i)
    plt.scatter(centroids[i][0],centroids[i][1],marker = 'x', color="black",s =␣
 ↪100)
plt.scatter(depot[0],depot[1],marker = 'p',s = 1000)
plt.annotate("Dépôt",(depot[0]-700,depot[1]))
plt.show()
```



12 Choix des paramètres

[22]:
```
n_ants = 100
n_iterations = 100
alpha = 1
beta = 10
```

```
evaporation_rate = 0.5
Q = 1
```

13 Regroupement des villes

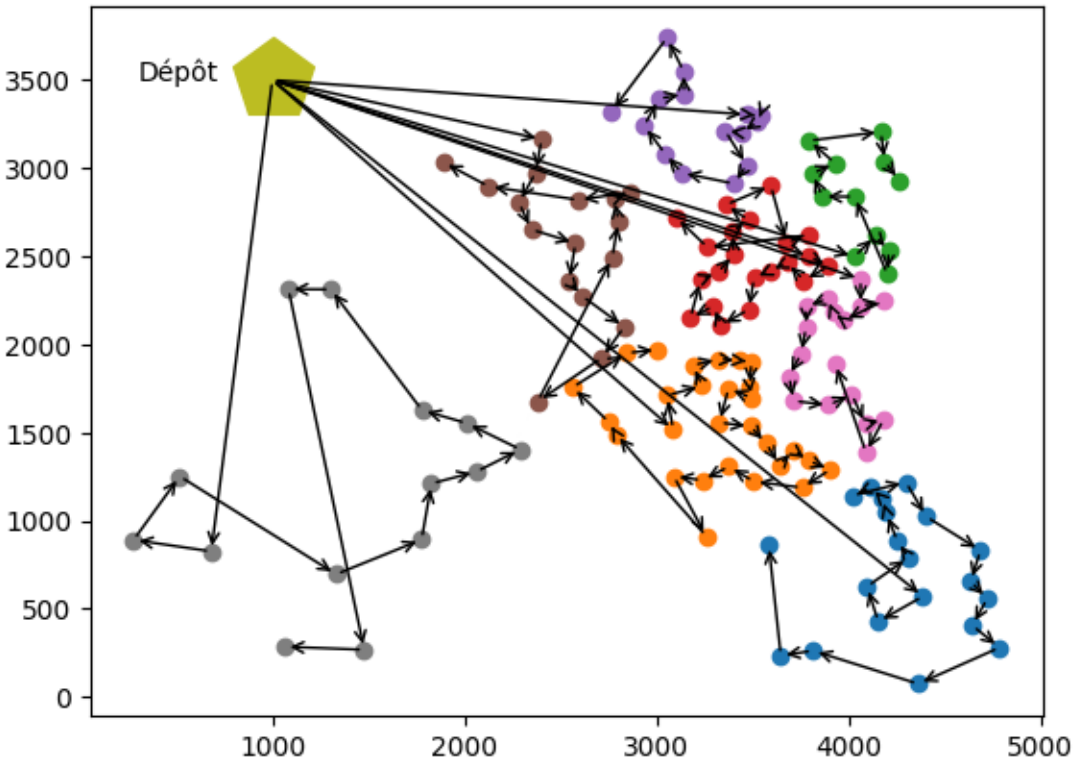
[23]:
```
groupes = []
for i in u_labels:
  groupes.append([])
for i in range(len(villes)):
  groupes[label[i]].append(villes[i])
for groupe in groupes:
  groupe = np.array(groupes,dtype=object)
```

14 Route selon groupe

[24]:
```
routes = []
for groupe in groupes:
  routes.append(ant_colony_optimization(groupe, n_ants, n_iterations, alpha,␣
  ↪beta, evaporation_rate, Q))
```

15 Traçage

[25]:
```
for i in range(len(routes)):
  s = np.array([[depot[0],depot[1]]]+[list(groupes[i][k]) for k in routes[i]])
  plt.scatter(s[:,0],s[:,1])
  for i in range(len(s)-1):
      plt.annotate('', xy=s[i], xytext=s[i+1], arrowprops=dict(arrowstyle='<-'))
plt.scatter(depot[0],depot[1],marker = 'p',s = 1000)
plt.annotate("Dépôt",(depot[0]-700,depot[1]))
plt.show()
```


Dépôt
3500
3000
2500
2000
1500
1000
500
0
1000
2000
3000
4000
5000