

TP3

November 9, 2023

1 TP 3 TSP

Questions :

Après familiarisation avec le code proposé, répondre aux questions suivantes : 1. Visualisation : Le programme proposé génère des instances du problème sous la forme d'une matrice des distances. Afin d'améliorer la visualisation des solutions proposées, procédez de la manière suivante : - Chaque ville à visiter a des coordonnées cartésiennes (X,Y) générées aléatoirement - La matrice des distances entre les villes est calculée en utilisant la distance euclidienne - Afficher les points et la tournée proposée (droite liant chaque paire de points successifs) 2. Faites varier les paramètres du programme de telle sorte à traiter des problèmes de tailles différentes (4 à 20 villes). Faites aussi varier les paramètres du recuit simulé, et représenter graphiquement l'évolution du coût de la solution en fonction du nombre d'itérations. 3. Programmer l'algorithme génétique permettant de résoudre le TSP (Vous pouvez vous inspirer du TP précédent et des éléments présentés dans TSPexemple.py). - Choisir les paramètres du croisement et de la mutation en fonction des résultats observés 4. Programmer l'algorithme de colonies de fourmis adapté au TSP : - Choisir les paramètres de l'algorithme en fonction des résultats observés 5. Comparer les trois approches du point de vue de la qualité des solutions proposées et du temps de calcul. Pour ce faire, générer des instances représentatives (minimum 4) du problème TSP.

```
[1]: import random
import TSPexemple1 as tsp
import matplotlib.pyplot as plt
import numpy as np
import time
```

1. Visualisation

Chaque ville à visiter a des coordonnées cartésiennes (X,Y) générées aléatoirement

```
[2]: def generate_random_cities(nb_cities):
    cities = []
    for i in range(nb_cities):
        cities.append((random.random(), random.random()))
    return cities

print(generate_random_cities(10))
```

```
[(0.38993122280336623, 0.05624068899857504), (0.4261104286270786,
0.7282528690051742), (0.7231345485474123, 0.8993231769847634),
```

```
(0.5958050317861062, 0.4878473892720495), (0.5297115863358092,
0.8617179757480236), (0.6432625237824734, 0.11958088296761271),
(0.33724760835264067, 0.7993824217349794), (0.7745083600942312,
0.4848411878406519), (0.08529763013084035, 0.287942971972891),
(0.49078630784513777, 0.25755090609021136)]
```

La matrice des distances entre les villes est calculée en utilisant la distance euclidienne

```
[3]: def distance_entre_2_villes(coordonnees):
    ville_1 = coordonnees[0]
    ville_2 = coordonnees[1]
    distance = ((ville_1[0] - ville_2[0])**2 + (ville_1[1] - ville_2[1])**2)**0.5
    return distance

def matrice_distance_ville(coordonnees):
    distances = np.zeros((len(coordonnees), len(coordonnees)))
    for i in range(len(coordonnees)):
        for j in range(len(coordonnees)):
            if i != j:
                # print("Distance entre la ville", i, "et la ville", j, ":",
                distance_entre_2_ville((coordonnees[i], coordonnees[j])))
                distances[i][j] = distance_entre_2_villes((coordonnees[i],
                coordonnees[j]))
    return distances

print(matrice_distance_ville(generate_random_cities(10)))
```

```
[[0.          1.06806065 0.84754372 1.00969877 0.94105786 0.68307511
 0.57991361 0.74794594 0.96812551 0.33107279]
 [1.06806065 0.          0.22061006 0.44308205 0.61967587 0.53561438
 0.60142103 0.79123897 0.18795256 0.87985258]
 [0.84754372 0.22061006 0.          0.42916339 0.55397439 0.3747415
 0.41528944 0.6570734 0.19081282 0.66877059]
 [1.00969877 0.44308205 0.42916339 0.          0.19583649 0.32681123
 0.4301777 0.43828519 0.26884754 0.9609849 ]
 [0.94105786 0.61967587 0.55397439 0.19583649 0.          0.29362404
 0.3809809 0.2662384 0.43410363 0.95813811]
 [0.68307511 0.53561438 0.3747415 0.32681123 0.29362404 0.
 0.10338027 0.28722554 0.36577205 0.6654721 ]
 [0.57991361 0.60142103 0.41528944 0.4301777 0.3809809 0.10338027
 0.          0.30197732 0.44577762 0.58019432]
 [0.74794594 0.79123897 0.6570734 0.43828519 0.2662384 0.28722554
 0.30197732 0.          0.60637482 0.846095 ]
 [0.96812551 0.18795256 0.19081282 0.26884754 0.43410363 0.36577205
 0.44577762 0.60637482 0.          0.83334417]
 [0.33107279 0.87985258 0.66877059 0.9609849 0.95813811 0.6654721
 0.58019432 0.846095 0.83334417 0.          ]]
```

1.1 Recuit simulé

On va utiliser l'algorithme de recuit simulé pour résoudre le problème du TSP et plus particulièrement celui proposé en exemple :

```
[4]: NOMBRE_DE_VILLES = 10
    TEMPERATURE = 1000
    FACTEUR = 0.99
    TOUR = 1000

    coor_ville = generate_random_cities(NOMBRE_DE_VILLES)
    distances_ville = matrice_distance_ville(coor_ville)

    ordre_ville, distance_totale = tsp.recuit_simule(NOMBRE_DE_VILLES,
    ↪distances_ville, TEMPERATURE, FACTEUR, TOUR)
```

Afficher les points et la tournée proposée (droite liant chaque paire de points successifs)

```
[5]: print(ordre_ville, distance_totale)

def afficaghe(coor_ville_p, ordre_ville_p):
    # Extraction des coordonnées dans l'ordre de visite
    x = [coor_ville_p[i][0] for i in ordre_ville_p]
    y = [coor_ville_p[i][1] for i in ordre_ville_p]

    # Ajout du point de départ à la fin pour fermer la boucle
    x.append(x[0])
    y.append(y[0])

    # Affichage des points
    plt.scatter(x, y)

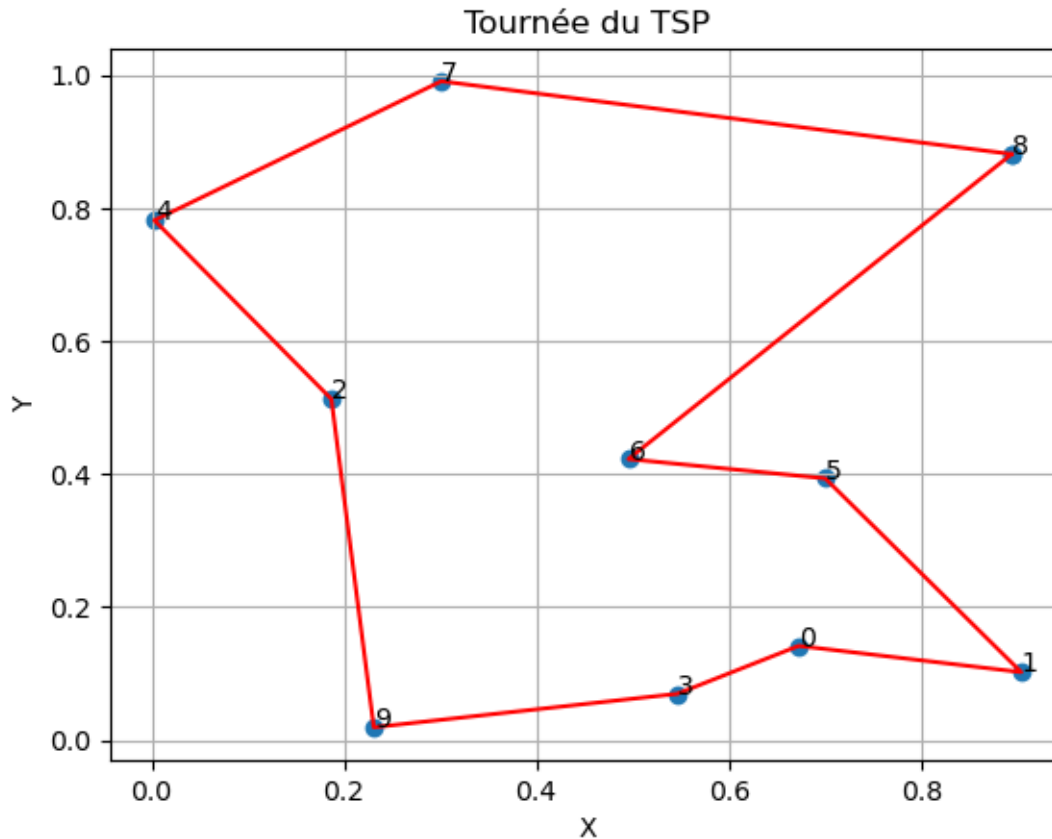
    # Affichage de la tournée
    plt.plot(x, y, 'r')

    # Ajout d'étiquettes pour chaque point
    for i in range(len(ordre_ville_p)):
        plt.annotate(ordre_ville_p[i], (x[i], y[i]))

    # Affichage du graphique
    plt.title("Tournée du TSP")
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.grid(True)
    plt.show()

    afficaghe(coor_ville, ordre_ville)
```

```
[8, 6, 5, 1, 0, 3, 9, 2, 4, 7] 3.6582862814709767
```



2. Faites varier les paramètres du programme

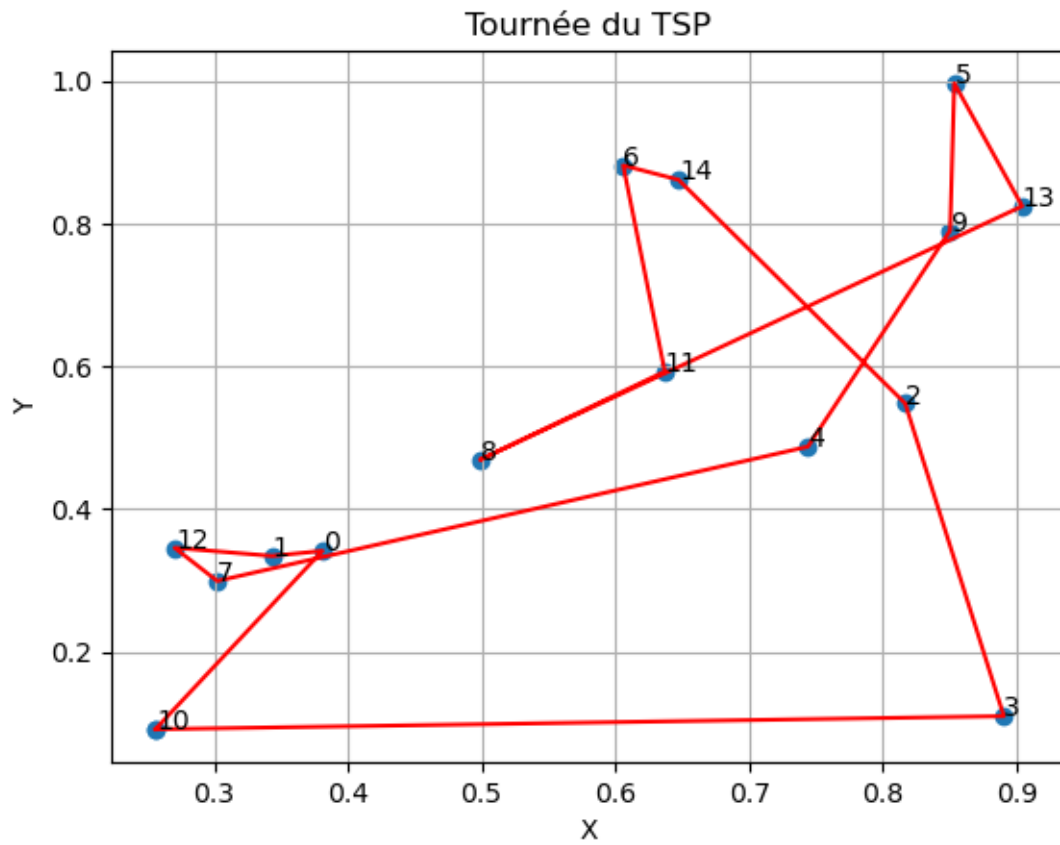
Faites varier les paramètres du programme de telle sorte à traiter des problèmes de tailles différentes (4 à 20 villes). Faites aussi varier les paramètres du recuit simulé, et représenter graphiquement l'évolution du coût de la solution en fonction du nombre d'itérations.

```
[6]: NOMBRE_DE_VILLES = 15 # 10
TEMPERATURE = 10000 # *10
FACTEUR = 0.999 # 0.99
TOUR = 50000 # *5

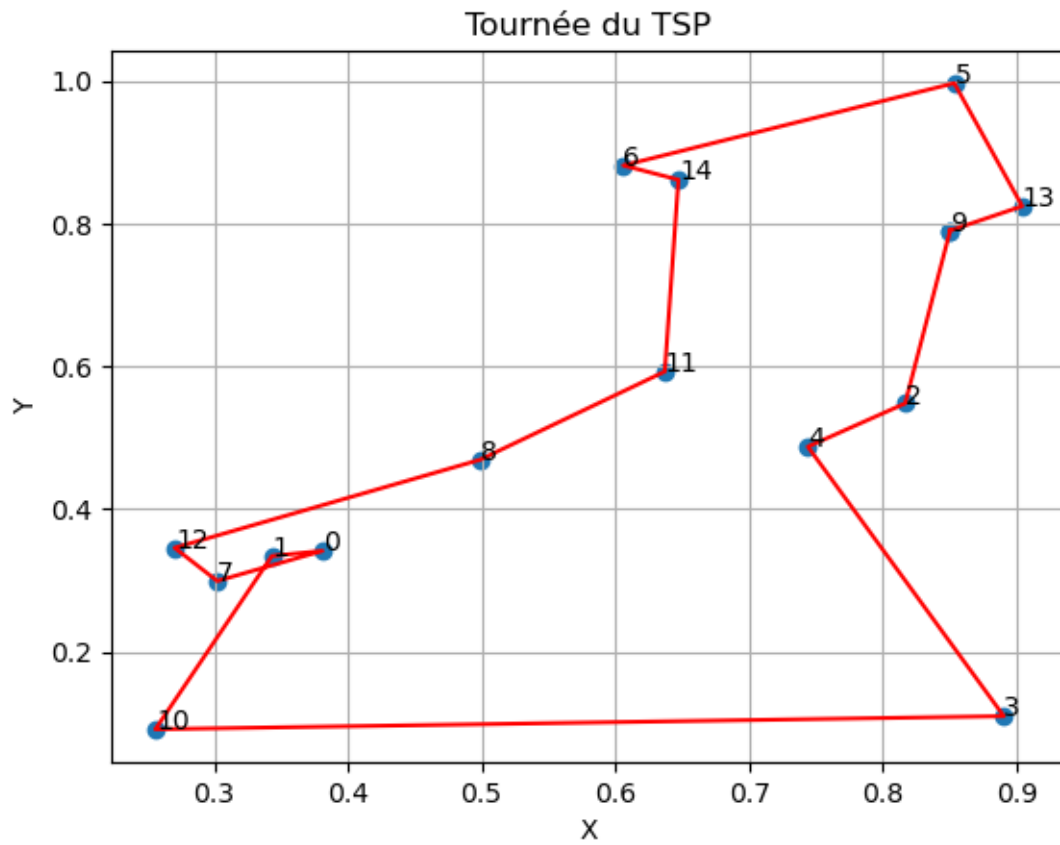
coor_ville = generate_random_cities(NOMBRE_DE_VILLES)
distances_ville = matrice_distance_ville(coor_ville)

for i in range(1000, TOUR, 10000):
    ordre_ville, distance_totale = tsp.recuit_simule(NOMBRE_DE_VILLES,
    ↪distances_ville, TEMPERATURE, FACTEUR, i)
    print(f"Le nombre d'itération est {i}, la distance {distance_totale}")
    afficaghe(coor_ville, ordre_ville)
```

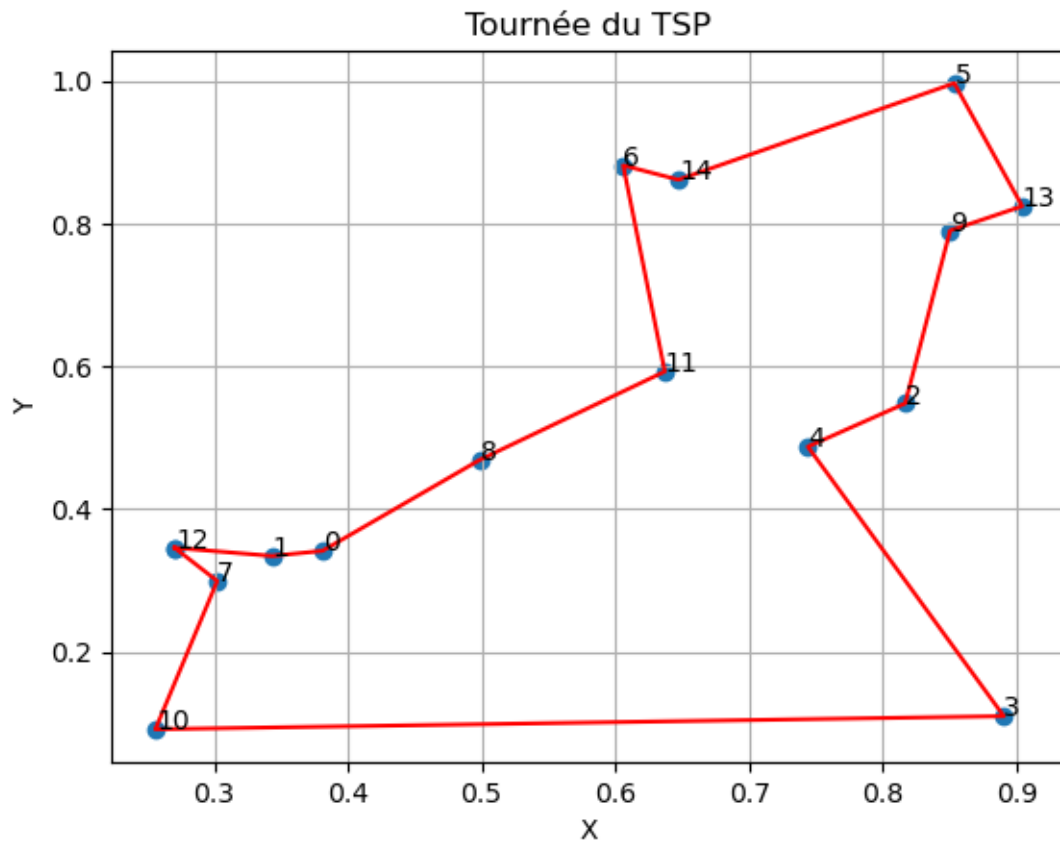
Le nombre d'itération est 1000, la distance 4.132348029266207



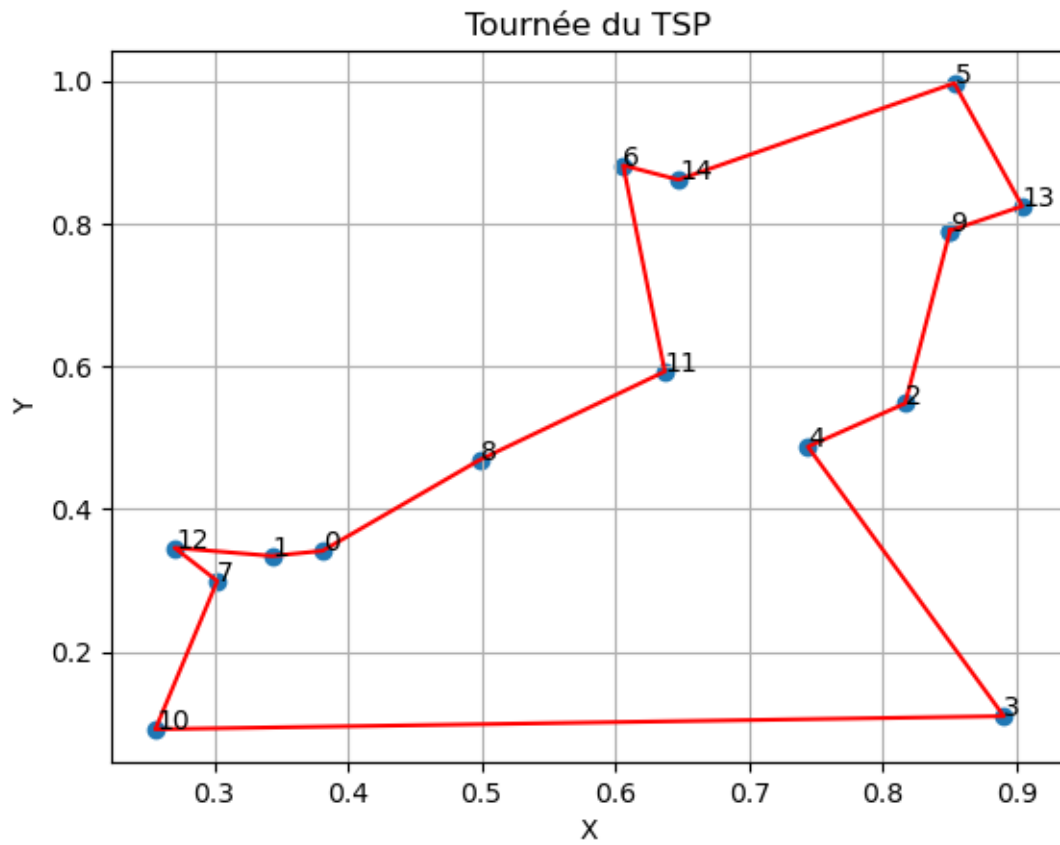
Le nombre d'itération est 11000, la distance 3.09801435600775



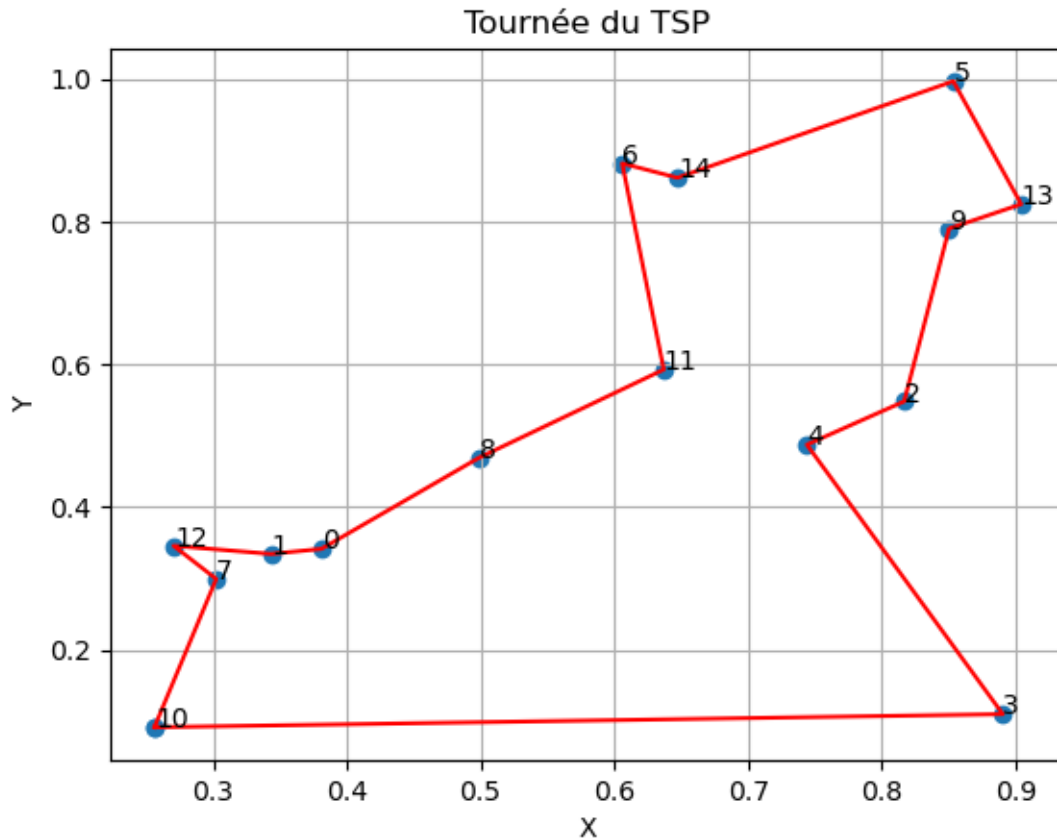
Le nombre d'itération est 21000, la distance 2.946547592057438



Le nombre d'itération est 31000, la distance 2.9465475920574384



Le nombre d'itération est 41000, la distance 2.946547592057438



1.2 Algorithmes génétiques

3. Programmer l'algorithme génétique

Programmer l'algorithme génétique permettant de résoudre le TSP (Vous pouvez vous inspirer du TP précédent et des éléments présentés dans TSPexemple.py). - Choisir les paramètres du croisement et de la mutation en fonction des résultats observés

```
[7]: def population_initiale(nombre_ville, taille_population):
    population = []
    for i in range(taille_population):
        population.append(np.random.permutation(nombre_ville))
    return population

print(population_initiale(10, 10))
```

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

```
[8]: def evaluation_totale(population, distances, nombre_ville):
    evaluation = []
    for i in range(len(population)):
        evaluation.append(tsp.cal_distance(population[i], distances,
↪nombre_ville))
    return evaluation

print(evaluation_totale(population_initiale(10, 10), distances_ville, 10))
```

```
[5.655229599618785, 3.817633546802632, 4.057064685805815, 5.03159443446587,
5.06192268470026, 4.464712937762468, 4.726906305569667, 4.658751559581468,
5.614607816229906, 4.821524343733737]
```

```
[9]: # Selection par la méthode de la roulette des deux parents

def selection_roulette(population, evaluation):
    # Calcul de la somme des évaluations des individus de la population.
    somme = sum(evaluation)

    # Création d'une liste de probabilités pour chaque individu.
    # Chaque probabilité est proportionnelle à l'évaluation de l'individu.
    proba = [i/somme for i in evaluation]

    # Calcul des probabilités cumulées pour faciliter la sélection.
    proba_cumul = np.cumsum(proba)

    # Initialisation de la liste des parents sélectionnés.
    parents = []
    index_parent = []

    # Sélection de 2 parents.
    for i in range(2):
        # Génération d'un nombre aléatoire entre 0 et 1.
        alea = random.random()

        # Sélection d'un parent en fonction de la probabilité cumulée.
        for j in range(len(proba_cumul)):
            if alea < proba_cumul[j]:
                # Ajout du parent sélectionné à la liste des parents.
                parents.append(population[j])
                # Enregistrement de l'indice du parent sélectionné.
                index_parent.append(j)
                break

    # Retour des parents sélectionnés et de leurs indices.
    return parents, index_parent
```

```
print(selection_roulette(population_initiale(10, 10),  
    ↪evaluation_totale(population_initiale(10, 10), distances_ville, 10)))
```

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

[10]: *# Mutation par inversion de deux villes*

```
def mutation_inversion(individu):  
    mutation = individu.copy()  
    alea = random.randint(0, len(individu)-1)  
    alea2 = random.randint(0, len(individu)-1)  
    mutation[alea], mutation[alea2] = mutation[alea2], mutation[alea]  
    return mutation  
  
print(mutation_inversion([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]))
```

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

[11]: *# Croisement en 1 point*

```
def croisement(enfants):  
    enfant_1 = enfants[0]  
    enfant_2 = enfants[1]  
    alea = random.randint(0, len(enfant_1)-1)  
    enfant_1[alea:], enfant_2[alea:] = enfant_2[alea:], enfant_1[alea:]  
    return enfant_1, enfant_2  
  
print(croisement([[0, 1, 2, 3, 4, 5, 6, 7, 8, 9], [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]  
    ↪0])))
```

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

[12]: *# Correction d'un individu*

```
def reparer_individu(individu):  
    # Calcul de la taille de l'individu (nombre de villes dans le chemin).  
    taille_individu = len(individu)  
  
    # Identification des villes qui ne sont pas présentes dans l'individu.  
    # Cela crée un ensemble de toutes les villes (supposé être de 0 à  
    ↪taille_individu - 1)  
    # et soustrait l'ensemble des villes présentes dans l'individu.  
    villes_manquantes = set(range(taille_individu)) - set(individu)  
  
    # Identification des villes qui apparaissent plus d'une fois dans  
    ↪l'individu.
```

```

villes_doubles = set([x for x in individu if individu.count(x) > 1])

# Créer une copie de l'individu pour éviter de modifier l'original.
individu_repare = individu[:]

# Boucle sur chaque ville qui apparaît plus d'une fois.
for ville in villes_doubles:
    # Tant qu'il y a des doublons de cette ville dans l'individu...
    while individu_repare.count(ville) > 1:
        # Trouver l'indice du premier doublon de la ville.
        index_ville_double = individu_repare.index(ville, individu_repare.
↪index(ville) + 1)
        # Prendre une ville manquante pour remplacer le doublon.
        ville_manquante = villes_manquantes.pop()
        # Remplacer le doublon par la ville manquante.
        individu_repare[index_ville_double] = ville_manquante

    # Retourner l'individu réparé (sans doublons et avec toutes les villes
↪présentes).
    return individu_repare

print(reparer_individu([9, 1, 2, 3, 5, 5, 6, 7, 8, 9]))

```

[9, 1, 2, 3, 5, 4, 6, 7, 8, 0]

[13]: # Algorithme génétique

```

def algorithme_genetique(distances, nombre_ville, nombre_iter, proba_mutation,
↪taille_population):
    population = population_initiale(nombre_ville, taille_population)
    for i in range(nombre_iter):
        evaluation = evaluation_totale(population, distances, nombre_ville)
        parents = selection_roulette(population, evaluation)[0]
        enfants = croisement(parents)
        # On répare les enfants
        # Conversion en liste de listes
        enfants = [list(arr) for arr in enfants]
        enfants[0] = reparer_individu(enfants[0])
        enfants[1] = reparer_individu(enfants[1])
        # On remplace les parents par les enfants s'ils sont meilleurs
        if tsp.cal_distance(enfants[0], distances, nombre_ville) < tsp.
↪cal_distance(parents[0], distances, nombre_ville):
            population[parents[1][0]] = enfants[0]
            if tsp.cal_distance(enfants[1], distances, nombre_ville) < tsp.
↪cal_distance(parents[1], distances, nombre_ville):
                population[parents[1][1]] = enfants[1]

```

```

        # On mute la population au hasard
        for j in range(len(population)):
            if random.random() < proba_mutation:
                population[j] =
↪reparer_individu(list(mutation_inversion(population[j])))
        # On cherche le meilleur individu
        evaluation = evaluation_totale(population, distances, nombre_ville)
        index_meilleur = evaluation.index(min(evaluation))
        best_individu = population[index_meilleur]
        return best_individu, min(evaluation)

```

1.2.1 Test de l'algorithme génétique

```

[14]: # Test de l'algorithme génétique

NOMBRE_DE_VILLES = 10
NOMBRE_ITER = 50000
PROBA_MUTATION = 0.1
TAILLE_POPULATION = 1000

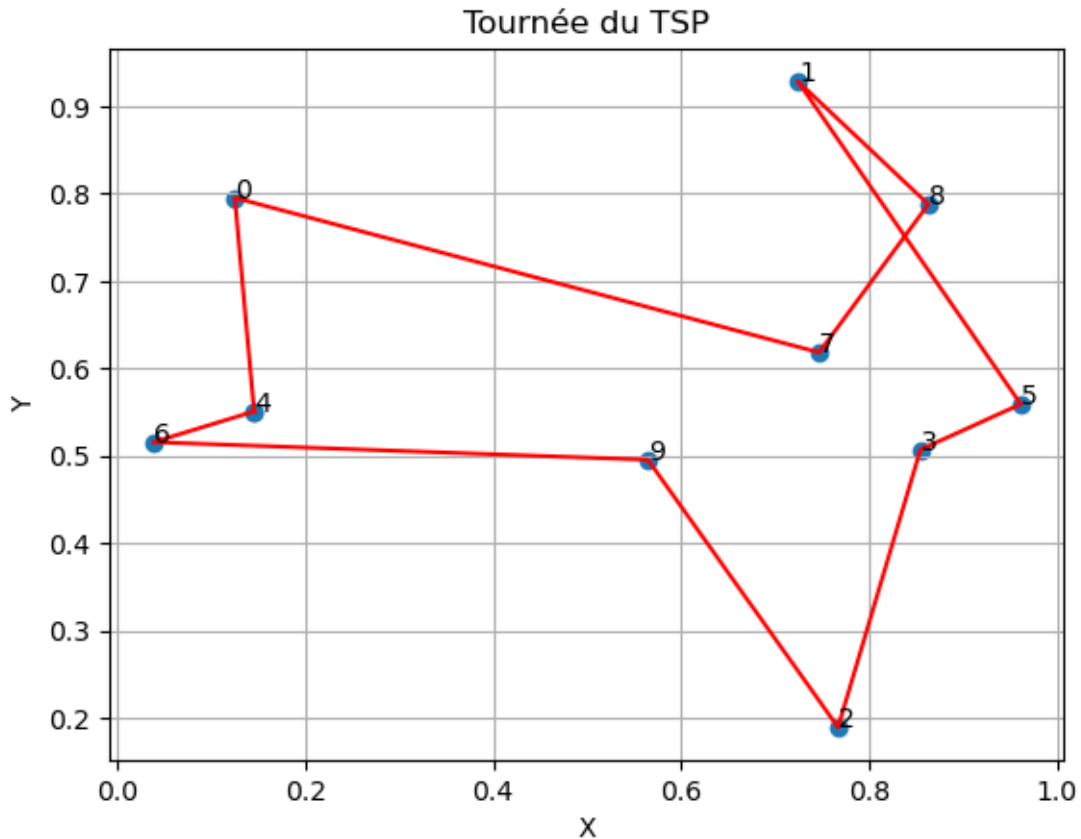
coor_ville = generate_random_cities(NOMBRE_DE_VILLES)
distances_ville = matrice_distance_ville(coor_ville)

meilleur_individu, distance_totale = algorithme_genetique(distances_ville,
↪NOMBRE_DE_VILLES, NOMBRE_ITER, PROBA_MUTATION, TAILLE_POPULATION)
print(f"Le meilleur individu est {meilleur_individu} avec une distance de
↪{distance_totale}")

afficaghe(coor_ville, meilleur_individu)

```

Le meilleur individu est [4, 6, 9, 2, 3, 5, 1, 8, 7, 0] avec une distance de 3.1893937983398004



1.3 Colonie de fourmis

4. Programmer l'algorithme de colonies de fourmis

Programmer l'algorithme de colonies de fourmis adapté au TSP : - Choisir les paramètres de l'algorithme en fonction des résultats observés

```
[15]: import numpy as np

def ant_colony_optimization(distances, n_fourmis, n_best, n_iterations, declin,
    ↪alpha, beta):
    n_villes = distances.shape[0]

    # Initialisation du dépôt de phéromones
    pheromone = np.ones((n_villes, n_villes)) / n_villes
    # Garder la trace de la meilleure solution
    plus_court_chemin = None
    meilleur_distance = float('inf')

    for _ in range(n_iterations):
```

```

tous_chemins = []
toutes_distances = []

for _ in range(n_fourmis):
    chemin = [np.random.randint(0, n_villes)] # Commencer à une ville
    ↪ aléatoire
    while len(chemin) < n_villes:
        ville_actuelle = chemin[-1]
        # Calculer la probabilité de transition pour la prochaine ville
        probabilités = np.zeros(n_villes)
        for ville_suivante in range(n_villes):
            if ville_suivante not in chemin:
                trace = pheromone[ville_actuelle, ville_suivante] **
    ↪ alpha
                visibilite = (1.0 / distances[ville_actuelle,
    ↪ ville_suivante]) ** beta
                probabilités[ville_suivante] = trace * visibilite
        # Normaliser pour obtenir des probabilités
        probabilités /= probabilités.sum()
        # Choisir la prochaine ville
        ville_suivante = np.random.choice(n_villes, p=probabilités)
        chemin.append(ville_suivante)

    tous_chemins.append(chemin)
    # Calculer la longueur du chemin
    distance = sum([distances[chemin[i], chemin[i+1]] for i in
    ↪ range(-1, n_villes-1)])
    toutes_distances.append(distance)

    # Vérifier si nous avons une nouvelle meilleure solution
    if distance < meilleur_distance:
        meilleur_distance = distance
        plus_court_chemin = chemin

    # Trier les fourmis - les meilleures en premier
    fourmies_triees = np.argsort(toutes_distances)
    # Mise à jour du dépôt de phéromones
    pheromone *= declin
    for i in range(n_best):
        for j in range(-1, n_villes-1):
            pheromone[tous_chemins[fourmies_triees[i]][j],
    ↪ tous_chemins[fourmies_triees[i]][j+1]] += 1.0 /
    ↪ toutes_distances[fourmies_triees[i]]

    return plus_court_chemin, meilleur_distance

# Données du problème TSP (matrice de distance)

```

```

NOMBRE_DE_VILLES = 10
NOMBRE_ITER = 10000
PROBA_MUTATION = 0.1
TAILLE_POPULATION = 1000

coor_ville = generate_random_cities(NOMBRE_DE_VILLES)
distances_ville = matrice_distance_ville(coor_ville)

DISTANCES = distances_ville

# Paramètres de l'algorithme
N_FOURMIS = 10
N_BEST = 5
N_ITERATIONS = 100
DECLIN = 0.95
ALPHA = 1
BETA = 2

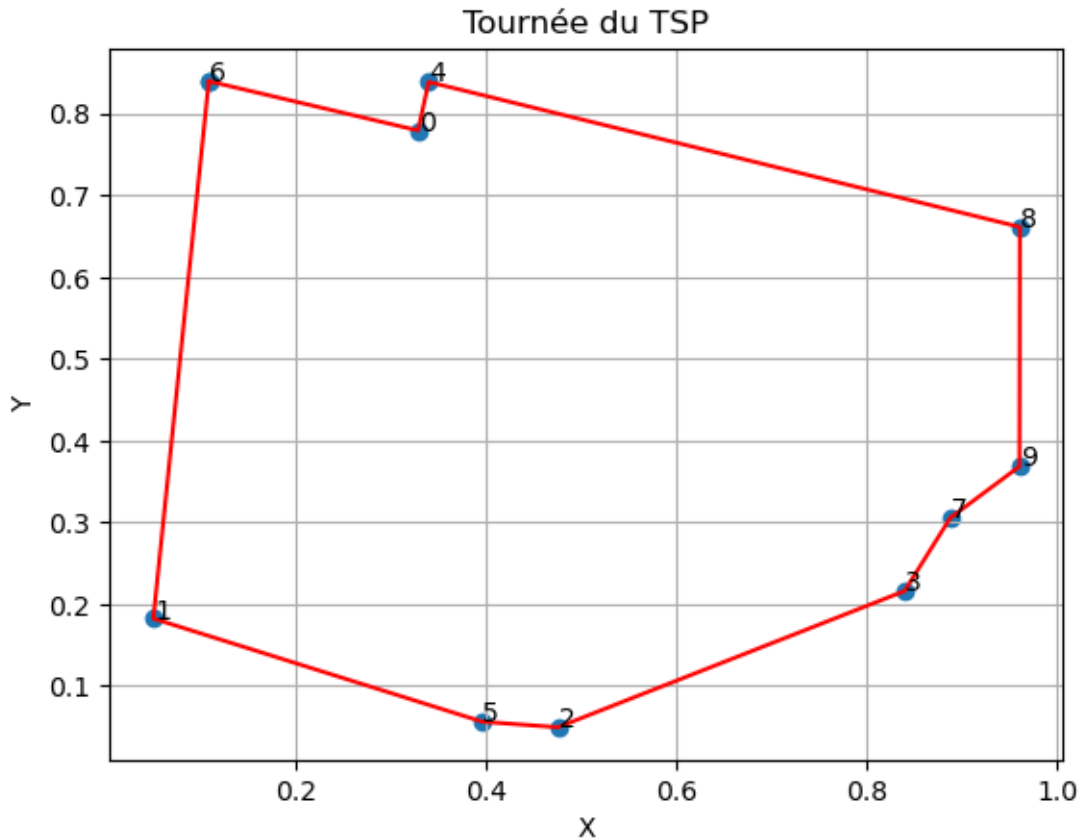
# Lancer l'algorithme
plus_court, meilleur_dist = ant_colony_optimization(DISTANCES, N_FOURMIS,
↳ N_BEST, N_ITERATIONS, DECLIN, ALPHA, BETA)

print("Meilleur chemin trouvé:", plus_court)
print("Distance du meilleur chemin:", meilleur_dist)

afficaghe(coor_ville, plus_court)

```

Meilleur chemin trouvé: [6, 0, 4, 8, 9, 7, 3, 2, 5, 1]
Distance du meilleur chemin: 2.933636938808536



1.4 Comparaison des heuristiques

5. Comparaison

Comparer les trois approches du point de vue de la qualité des solutions proposées et du temps de calcul. Pour ce faire, générer des instances représentatives (minimum 4) du problème TSP.

```
[16]: # Données du problème TSP (matrice de distance)
# Paramètres
NOMBRE_DE_VILLES = 15

# Recuit simulé
TEMPERATURE = 10000
FACTEUR = 0.99
TOUR = 10000

# Algorithme génétique
NOMBRE_ITER = 10000
PROBA_MUTATION = 0.1
TAILLE_POPULATION = 1000
```

```

# Colonie de fourmis
N_FOURMIS = 15
N_BEST = 5
N_ITERATIONS = 100
DECLIN = 0.95
ALPHA = 1
BETA = 2

coor_ville = generate_random_cities(NOMBRE_DE_VILLES)
distances_ville = matrice_distance_ville(coor_ville)

temps_debut_recuit = time.time()
ordre_ville_recuit, distance_totale_recuit = tsp.
    ↳recuit_simule(NOMBRE_DE_VILLES, distances_ville, TEMPERATURE, FACTEUR, TOUR)
temps_fin_recuit = time.time()

print(f"Le temps d'exécution du recuit simulé est de {temps_fin_recuit -
    ↳temps_debut_recuit} secondes")
print(f"La distance totale est de {distance_totale_recuit}")
afficaghe(coor_ville, ordre_ville_recuit)

temps_debut_genetique = time.time()
meilleur_individu_gene, distance_totale_gene =
    ↳algorithm_genetique(distances_ville, NOMBRE_DE_VILLES, NOMBRE_ITER,
    ↳PROBA_MUTATION, TAILLE_POPULATION)
temps_fin_genetique = time.time()

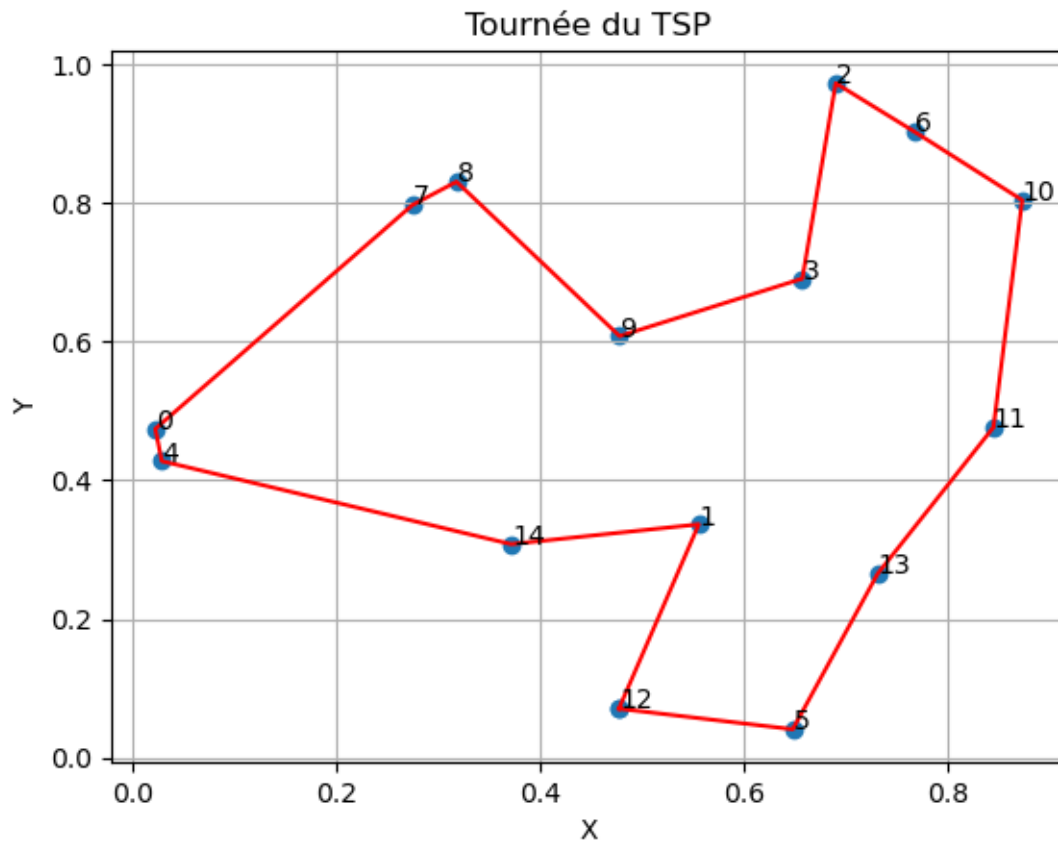
print(f"Le temps d'exécution de l'algorithme génétique est de
    ↳{temps_fin_genetique - temps_debut_genetique} secondes")
print(f"La distance totale est de {distance_totale_gene}")
afficaghe(coor_ville, meilleur_individu_gene)

temps_debut_fourmis = time.time()
plus_court_fourmis, meilleur_dist_fourmis =
    ↳ant_colony_optimization(distances_ville, N_FOURMIS, N_BEST, N_ITERATIONS,
    ↳DECLIN, ALPHA, BETA)
temps_fin_fourmis = time.time()

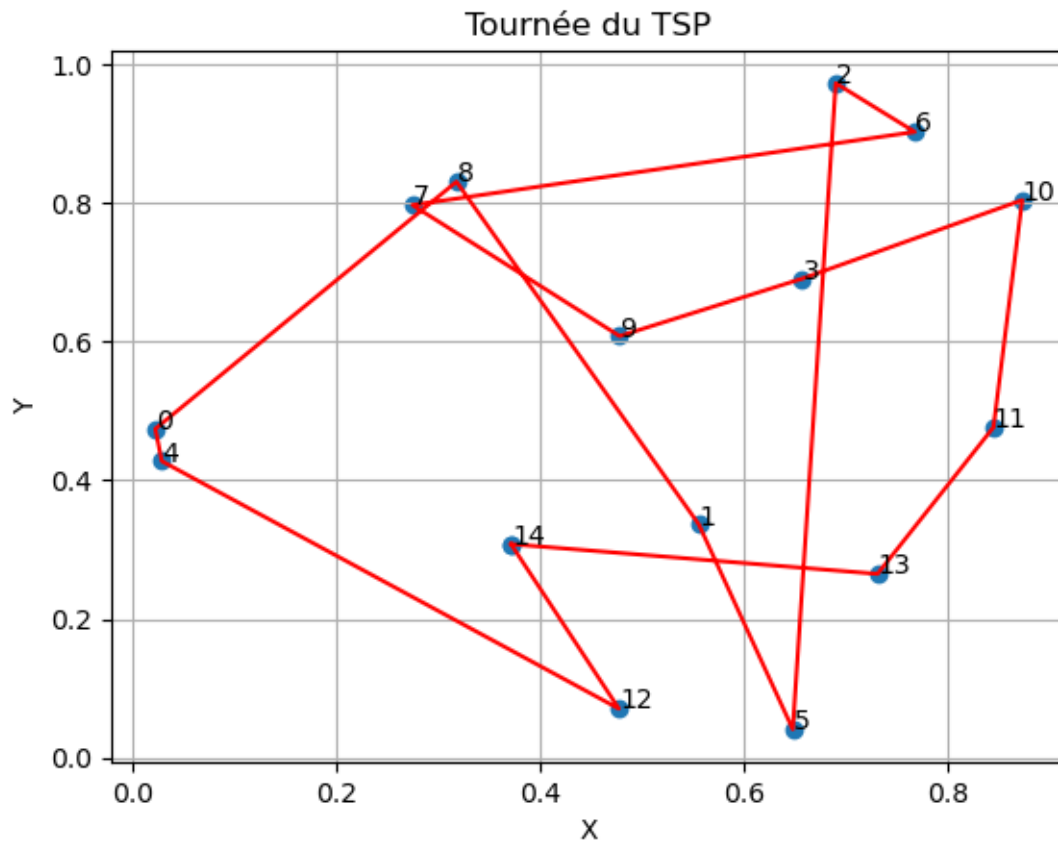
print(f"Le temps d'exécution de l'algorithme de colonies de fourmis est de
    ↳{temps_fin_fourmis - temps_debut_fourmis} secondes")
print(f"La distance totale est de {meilleur_dist_fourmis}")
afficaghe(coor_ville, plus_court_fourmis)

```

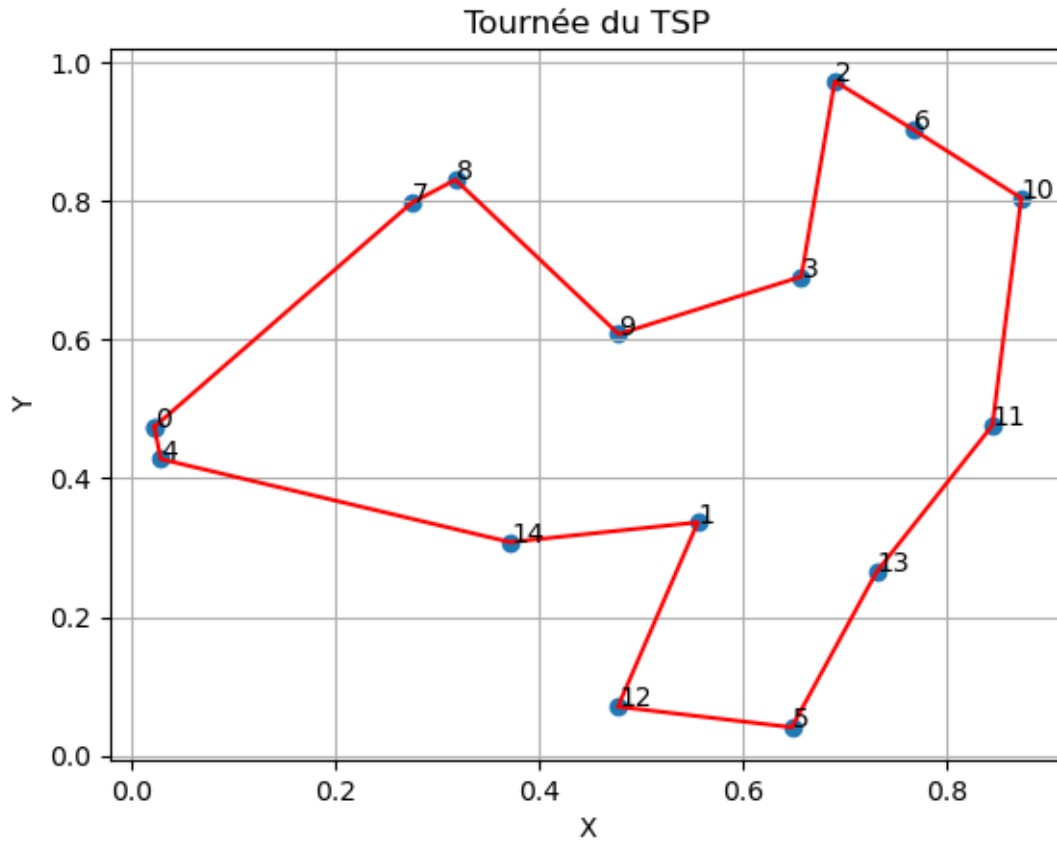
Le temps d'exécution du recuit simulé est de 2.6000499725341797 secondes
 La distance totale est de 3.3246008901232003



Le temps d'exécution de l'algorithme génétique est de 35.74132299423218 secondes
La distance totale est de 5.3942441552397975



Le temps d'exécution de l'algorithme de colonies de fourmis est de
0.25608015060424805 secondes
La distance totale est de 3.324600890123199



1.4.1 Conclusion sur les heuristiques

On remarque que l'algorithme de colonies de fourmis est le plus rapide et donne l'une des meilleures solutions. L'algorithme génétique est le plus lent et donne la moins bonne solution. Le recuit simulé est entre les deux. Les heuristiques sont donc dans l'ordre : colonies de fourmis, recuit simulé, algorithme génétique. De manière générale le choix des heuristiques dépend du problème à résoudre.