

```
In [35]: import numpy as np
import matplotlib.pyplot as plt
from tqdm import *
import random
```

L'algorithme du recuit simulé

L'algorithme du recuit simulé permet de résoudre des problèmes d'optimisation non convexe, pour lesquels la méthode de descente du gradient n'est pas appropriée. On s'intéresse ici à une application au problème du voyageur de commerce (section 6.4 dans le poly).

1) Données numériques

On souhaite calculer le **plus court chemin cyclique** passant par toutes les capitales des états fédéraux américains (Hawaii et Alaska exclus). Ce problème provient de la collection d'exemples **TSPLIB**. Les données dont on dispose sont les coordonnées géographiques des villes, regroupées dans une matrice `uscapitals_list` :

```
In [36]: uscapitals_list = np.genfromtxt("uscapitals.csv", delimiter=",")
```

La i-ème ligne de la matrice `uscapitals_list` est le couple de coordonnées de la i-ème ville.
Par exemple :

```
In [37]: print(uscapitals_list[3])
[401. 841.]
```

Dans l'exemple que nous considérons ici, le parcours optimal est connu et stocké dans la variable `uscapitals_opt` :

```
In [38]: uscapitals_opt = np.genfromtxt("uscapitals_opt.csv", delimiter=",").astype(int)-1
print(uscapitals_opt)
[ 0  7 37 30 43 17  6 27  5 36 18 26 16 42 29 35 45 32 19 46 20 31 38 47
 4 41 23  9 44 34  3 25  1 28 33 40 15 21  2 22 13 24 12 10 11 14 39  8]
```

Pour calculer la distance (en milles) entre les villes à partir de leurs coordonnées, on utilise la distance euclidienne classique :

```
In [39]: def euclidian_dist(citya, cityb):
    return np.linalg.norm(cityb-citya)
```

```
In [40]: uscapitals_distances = np.zeros((len(uscapitals_list),
                                         len(uscapitals_list)))
for i in range(len(uscapitals_list)):
    for j in range(len(uscapitals_list)):
        uscapitals_distances[i,j] = euclidian_dist(uscapitals_list[i], uscapitals_list[j])
```

```
In [41]: cities = uscapitals_list
optimal = uscapitals_opt
distances = uscapitals_distances
```

2) L'algorithme

Objectif à minimiser Le coût V d'un parcours est la somme des distances entre villes consécutives. On identifie un parcours avec une permutation σ de l'ensemble $\{0, 2, \dots, 15\}$, que l'on représente par une liste. $\sigma(k)$ est donc la k -ème ville que l'on visite en suivant le parcours σ . On a donc :

$$V(\sigma) = \sum_{i=1}^{n-1} d(\sigma(i), \sigma(i+1)) + d(\sigma(n), \sigma(1))$$

TODO : La fonction `V` le tableau de l'ordre des villes `tour` et la matrice `distances` des distances entre villes, et retourne un nombre réel `c`.

```
In [42]: def V(tour, distances):
    c = 0
    for i in range(len(tour)-1):
        c = c + distances[tour[i]][tour[i+1]]
    c = c + distances[tour[-1]][tour[0]]
    return c
```

Algorithme du recuit simulé : À chaque itération, l'algorithme propose un nouveau parcours, obtenu en tirant au hasard deux nombres i et j et en invertissant les villes $\sigma(i)$ et $\sigma(j)$ dans l'état courant.

TODO : La fonction `proposition` prend un tableau d'ordre des villes et retourne un nouveau tableau permué. Par exemple, `proposition([0,1,2,3,4])` pourrait produire `[1,0,2,3,4]`.

```
In [43]: def proposition(tour):
    i, j = np.random.choice(len(tour), size=2, replace=False)
    tour[i], tour[j] = tour[j], tour[i]
    return tour
print("Exemple :", [0,1,2,3,4], "->", proposition([0,1,2,3,4]))
```

Exemple : `[0, 1, 2, 3, 4] -> [4, 1, 2, 3, 0]`

Le deuxième ingrédient est le schéma de température, qui régit la décroissance de la température au cours du temps. Plusieurs choix sont possibles. On utilise pour commencer une décroissance polynomiale :

```
In [44]: def T(i):
    return 300*((i+1)**(-0.03))
```

TODO : Dans la fonction suivante, on implémente l'algorithme. Les arguments sont

- la matrice des distances entre villes `distances`

- le nombre d'itérations `N_iter`
- l'état initial `initial_tour`
- une fonction de proposition de changement `proposition`
- une fonction `T` pour le schéma de température

La fonction retourne

- l'état final `tour` après `N_iter` étapes
- l'état `best` qui a la valeur minimale pour la fonction coût (`V`) parmi tous les états visités
- la liste `costseq` des valeurs de la fonction coût à chaque étape de l'algorithme
- la liste `bestseq` des meilleures valeurs dans la fonction coût trouvées par l'algorithme

```
In [45]: def recuit_simule(distances, N_iter, initial_tour, proposition, T):
    current_tour = initial_tour
    best_tour = initial_tour
    best_cost = V(initial_tour, distances)
    costseq = [best_cost]
    bestseq = [best_cost]

    for i in range(N_iter):
        new_tour = proposition(current_tour)
        new_cost = V(new_tour, distances)
        delta_cost = new_cost - costseq[-1]

        # Décider si Le nouveau parcours doit être accepté
        if delta_cost < 0 or np.random.rand() < np.exp(-delta_cost / T(i)):
            # Accepter Le nouveau parcours et mettre à jour le coût actuel
            current_tour = new_tour
            costseq.append(new_cost)

            # Vérifier si Le nouveau coût est le meilleur et le mettre à jour si nécessaire
            if new_cost < best_cost:
                best_tour = new_tour
                best_cost = new_cost
                bestseq.append(new_cost)
            else:
                # Si Le nouveau parcours n'est pas accepté, répéter le coût actuel
                costseq.append(costseq[-1])
                # Suivre le meilleur coût trouvé jusqu'à présent
                bestseq.append(best_cost)

    # Renvoyer Le parcours actuel, le meilleur parcours, La séquence des coûts et La séquence des meilleurs coûts
    return current_tour, best_tour, costseq, bestseq
```

Une collection d'états initiaux possibles est stockée dans la liste `problem_initial_tours` : le i -ème élément est obtenu en appliquant i transpositions aléatoires au parcours optimal.

```
In [46]: problem_initial_tours = []
for i in range(10):
    problem_initial_tours.append(optimal.copy())
    for k in range(i):
        problem_initial_tours[i] = proposition(problem_initial_tours[i])
```

On peut à présent tester l'algorithme et visualiser la décroissance dans la fonction objectif :

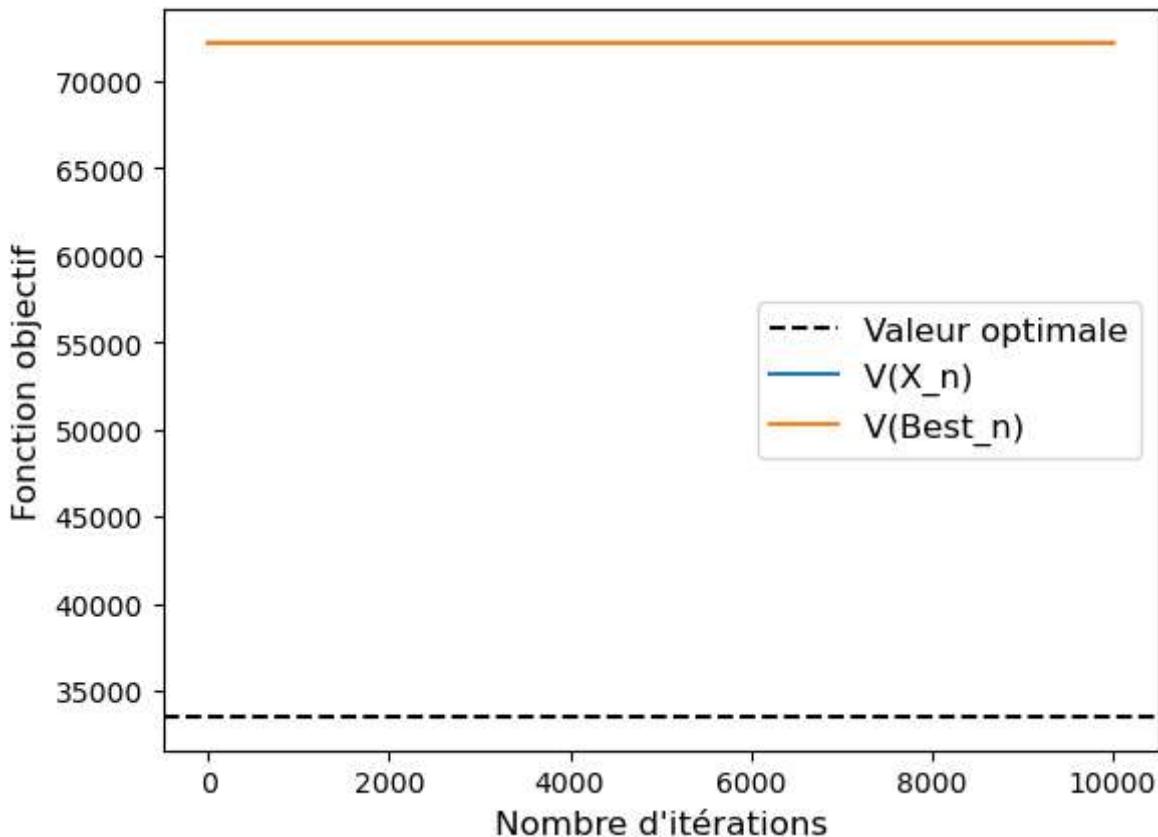
```
In [47]: N_iter = 10000
i = 8
initial_tour = problem_initial_tours[i]

tour, best, cseq, bseq = recuit_simule(distances, N_iter, initial_tour, proposition, 1)
```

```
In [48]: def plot_recuit(cseq, bseq, best, optimal, legend=True, label=""):
    p1 = plt.axhline(V(optimal, distances), label='Valeur optimale', c="black", ls="--")
    p2, = plt.plot(cseq, label='V(X_n) '+label)
    p3, = plt.plot(bseq, label='V(Best_n) '+label)
    if legend:
        plt.legend(fontsize=12)
        plt.ylabel("Fonction objectif", fontsize=12)
        plt.xlabel("Nombre d'itérations", fontsize=12)
```

```
In [49]: gap = V(best, distances) - V(optimal, distances)
print("La différence entre la meilleure valeur de la fonction coût trouvée par l'algorithme et la valeur optimale est", gap)
plot_recuit(cseq, bseq, best, optimal)
```

La différence entre la meilleure valeur de la fonction coût trouvée par l'algorithme et la valeur optimale est 98949.37538290798



```
In [50]: tt = optimal
total = len(cities)

plt.figure(figsize=(12,6))

plt.subplot(1, 2, 1)
plt.title('Solution optimale')
for i in range(total-1):
    plt.plot(np.array([cities[tt[i], 0], cities[tt[i+1], 0]]),
             np.array([cities[tt[i], 1], cities[tt[i+1], 1]]), '-ro')
```

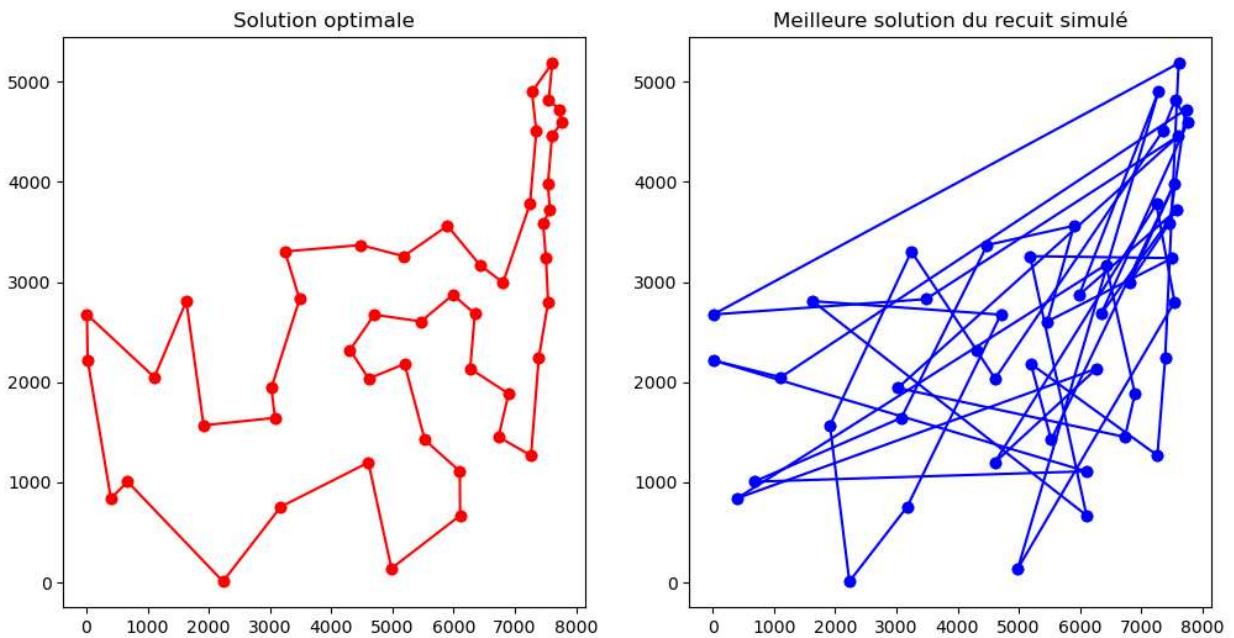
```

plt.plot(np.array([cities[tt[total-1], 0], cities[tt[0], 0]]), np.array(
    [cities[tt[total-1], 1], cities[tt[0], 1]]), '-ro')

plt.subplot(1, 2, 2)
plt.title('Meilleure solution du recuit simulé')
for i in range(total-1):
    plt.plot(np.array([cities[best[i], 0], cities[best[i+1], 0]]),
        np.array([cities[best[i], 1], cities[best[i+1], 1]]), '-bo')

plt.plot(np.array([cities[best[total-1], 0], cities[best[0], 0]]), np.array(
    [cities[best[total-1], 1], cities[best[0], 1]]), '-bo')
plt.show()

```



3) Rôle de la proposition de changement

Dans la partie précédente, nous avons considéré la proposition de changement consistant à intervertir deux villes choisies au hasard. D'autres propositions sont possibles. Par exemple, au lieu d'intervertir deux villes, on peut en intervertir $k > 1$.

Une autre possibilité consiste à choisir deux villes i, j au hasard et à les intervertir dans le parcours, ainsi que toutes les villes entre les deux.

TODO : implémenter la fonction `proposition_reverse`, qui prend un tableau d'ordre des villes et retourne un tableau permué. Par exemple, `proposition_reverse([0, 1, 2, 3, 4, 5, 6])` pourrait produire `[0, 1, 2, 5, 4, 3, 6]`.

```
In [51]: def proposition_reverse(tour):
    i, j = np.random.choice(len(tour), size=2, replace=False)
    if j > i :
        while j > i :
            a = tour[i]
            tour[i] = tour[j]
            tour[j] = a
            i = i + 1
```

```

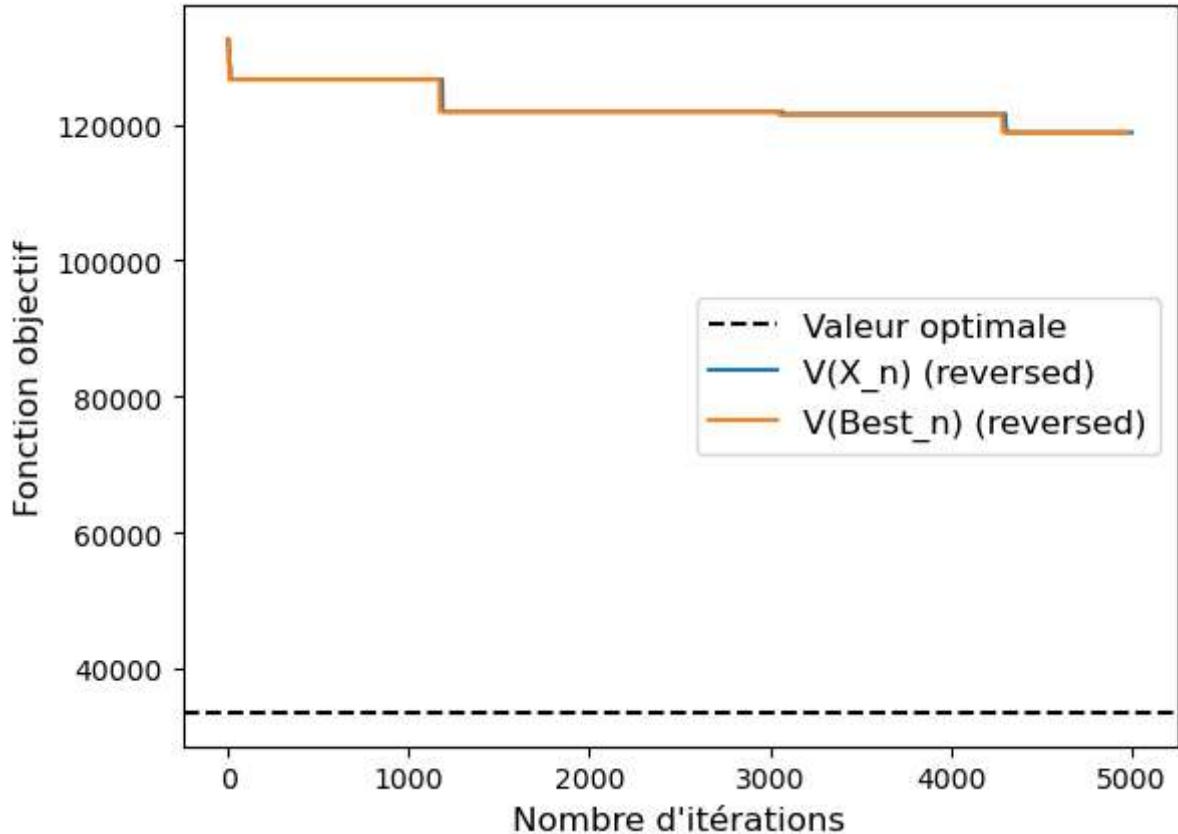
        j = j -1
    else:
        while j < i :
            a = tour[j]
            tour[j] = tour[i]
            tour[j] = a
            j = j + 1
            i = i -1
    return tour
print("Exemple : ", [0,1,2,3,4,5,6], "->", proposition_reverse([0,1,2,3,4,5,6]))

```

Exemple : [0, 1, 2, 3, 4, 5, 6] -> [0, 1, 2, 4, 3, 5, 6]

```
In [52]: tour, best, cseq, bseq = recuit_simule(distances, 5000, initial_tour, proposition_reversed)
gap = V(best, distances) - V(optimal, distances)
print("La différence entre la meilleure valeur de la fonction coût trouvée par l'algorithme et la valeur optimale est", gap)
plot_recuit(cseq, bseq, best, optimal, label="(reversed)")
```

La différence entre la meilleure valeur de la fonction coût trouvée par l'algorithme et la valeur optimale est 118762.83013566301



Enfin, on peut raffiner la règle `proposition_reverse` pour favoriser les interversions entre sommets proches dans l'itinéraire. Par exemple, étant donné un sommet aléatoire i , on choisit le sommet j avec probabilité $\frac{C}{|i-j|+1}$, avec C une constante de normalisation.

TODO : adapter la fonction `proposition_reverse` pour ajouter une pondération dans le choix des sommets (on pourra utiliser la fonction `np.choice`).

```
In [53]: def proposition_reverse_weighted(tour):
    n = len(tour)
    i = np.random.choice(n)
```

```

weights = [1 / (abs(i - j) + 1) for j in range(n)]
C = 1 / sum(weights)
weights = [C * weight for weight in weights]
j = np.random.choice(n, p=weights)
# En bas est autre maniere de reverse les element entre i,j
if i < j:
    tour[i:j+1] = tour[i:j+1][::-1]
else:
    tour[j:i+1] = tour[j:i+1][::-1]

return tour
print("Exemple :", [0,1,2,3,4,5,6], "->", proposition_reverse_weighted([0,1,2,3,4,5,6])

```

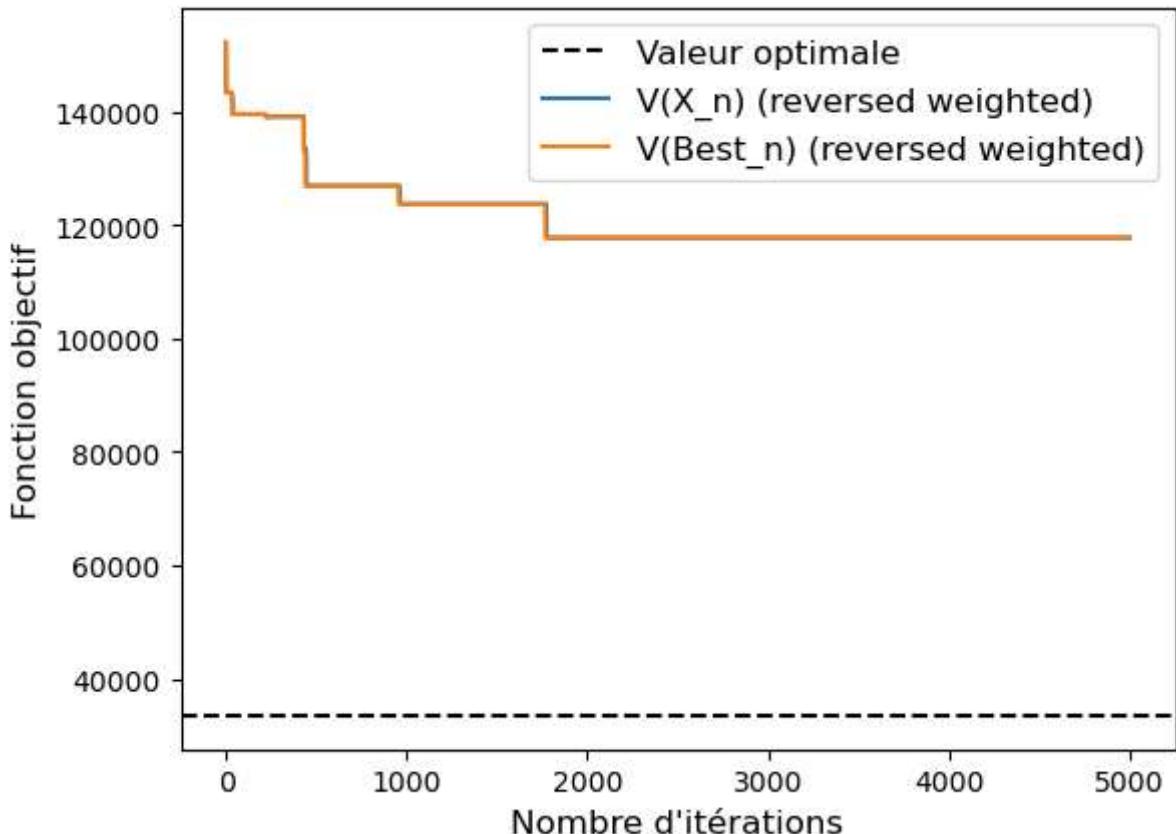
Exemple : [0, 1, 2, 3, 4, 5, 6] -> [0, 3, 2, 1, 4, 5, 6]

```

In [54]: tour, best, cseq, bseq = recuit_simule(distances, 5000, initial_tour, proposition_rev
gap = V(best, distances) - V(optimal, distances)
print("La différence entre la meilleure valeur de la fonction coût trouvée par l'algorithme et la valeur optimale est 111154.83145790186
plot_recuit(cseq, bseq, best, optimal, label="(reversed weighted)")

```

La différence entre la meilleure valeur de la fonction coût trouvée par l'algorithme et la valeur optimale est 111154.83145790186



Les méthodes `proposition_reverse` et `proposition_reverse_weighted` donnent des résultats proches à première vue. Pour les comparer, on effectue chaque simulation 50 fois et on compare les histogrammes des performances :

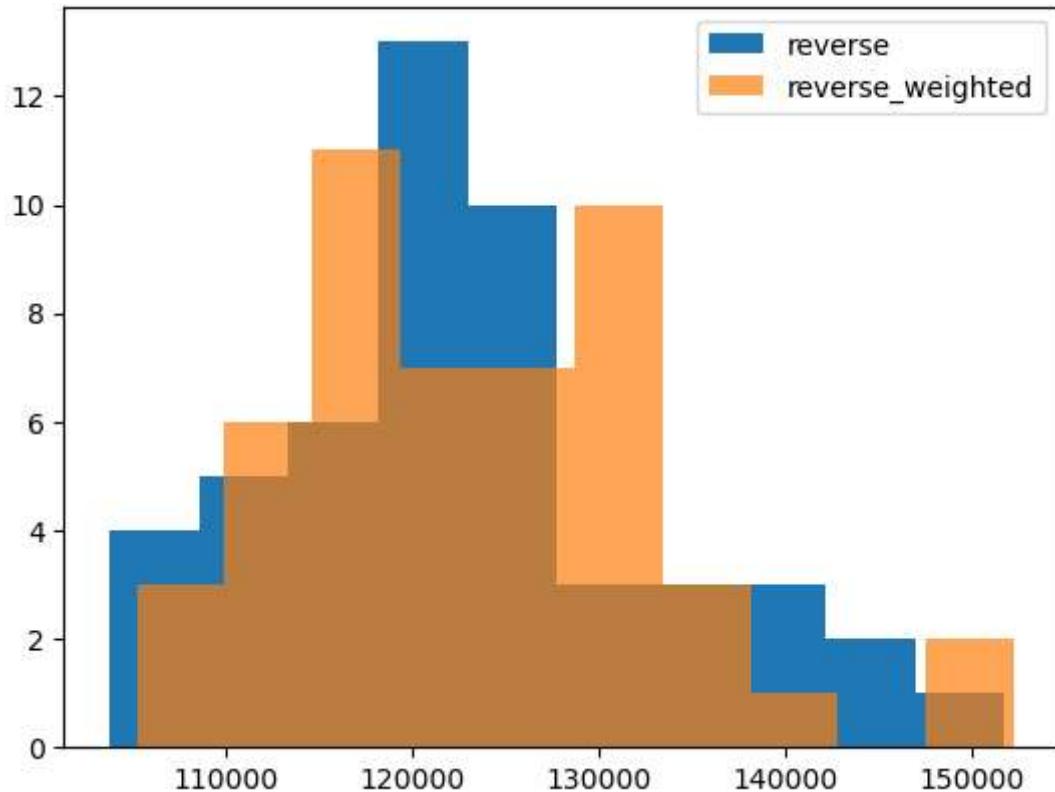
```

In [55]: gaps_reverse = []
for _ in range(50):
    tour, best, cseq, bseq = recuit_simule(distances, 5000, initial_tour, proposition_rev
    gap = V(best, distances) - V(optimal, distances)
    gaps_reverse.append(gap)

```

```
In [56]: gaps_reverse_weighted = []
for _ in range(50):
    tour, best, cseq, bseq = recuit_simule(distances, 5000, initial_tour, proposition_
    gap = V(best, distances) - V(optimal, distances)
    gaps_reverse_weighted.append(gap)
```

```
In [57]: plt.hist(gaps_reverse ,label="reverse")
plt.hist(gaps_reverse_weighted, label="reverse_weighted", alpha=0.7)
plt.legend()
plt.show()
```



4) Schéma de température

On a utilisé jusqu'ici le schéma de température $T(i) = 300(i + 1)^{-0.3}$ à décroissance polynomiale. D'autres choix sont bien sûr possibles. On peut par exemple varier les constantes du schéma $T_{\text{polynomial}}(i) = a(i + 1)^b$:

```
In [58]: def T_polynomial(a, b):
    def T(i):
        return a*((i+1)**b)
    return T
```

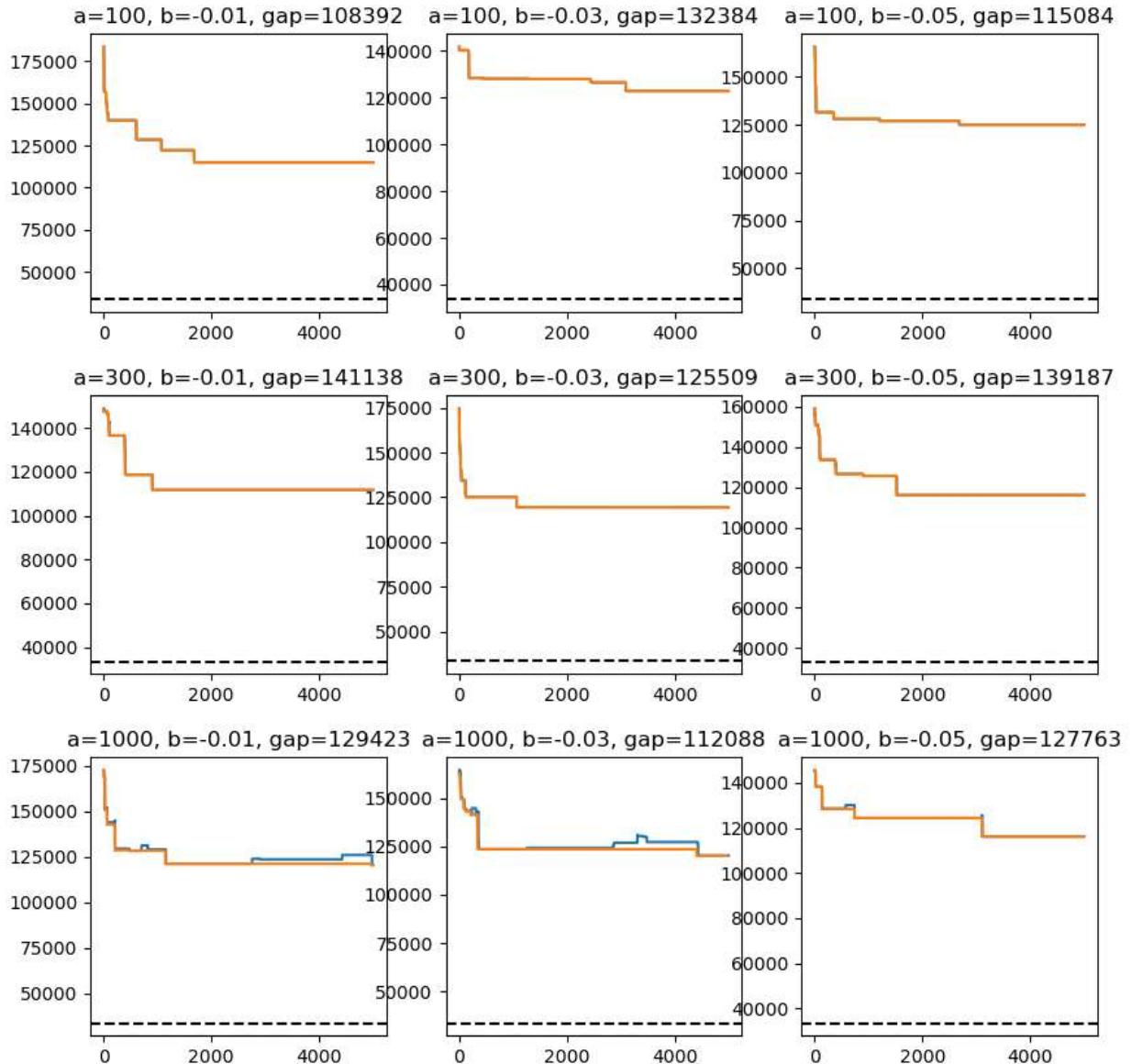
On peut maintenant tester la performance de l'algorithme du recuit simulé sur plusieurs valeurs de a et b :

```
In [59]: plt.figure(figsize=(10,10))
plt.subplots_adjust(hspace=0.3)
i = 1
for a in [100, 300, 1000]:
```

```

for b in [-0.01, -0.03, -0.05]:
    T = T_polynomial(a, b)
    tour, best, cseq, bseq = recuit_simule(distances, 5000, initial_tour, proposit
    gap = V(best, distances) - V(optimal, distances)
    plt.subplot(3,3,i)
    plt.title("a={}, b={}, gap={}".format(a,b,round(gap)))
    plot_recuit(cseq, bseq, best, optimal, legend=False)
    i += 1
plt.show()

```



A titre de comparaison, on peut également tester un schéma de température à décroissance exponentielle $T_{exponential}(i) = a * b^i$ (avec $0 < b < 1$).

In [60]:

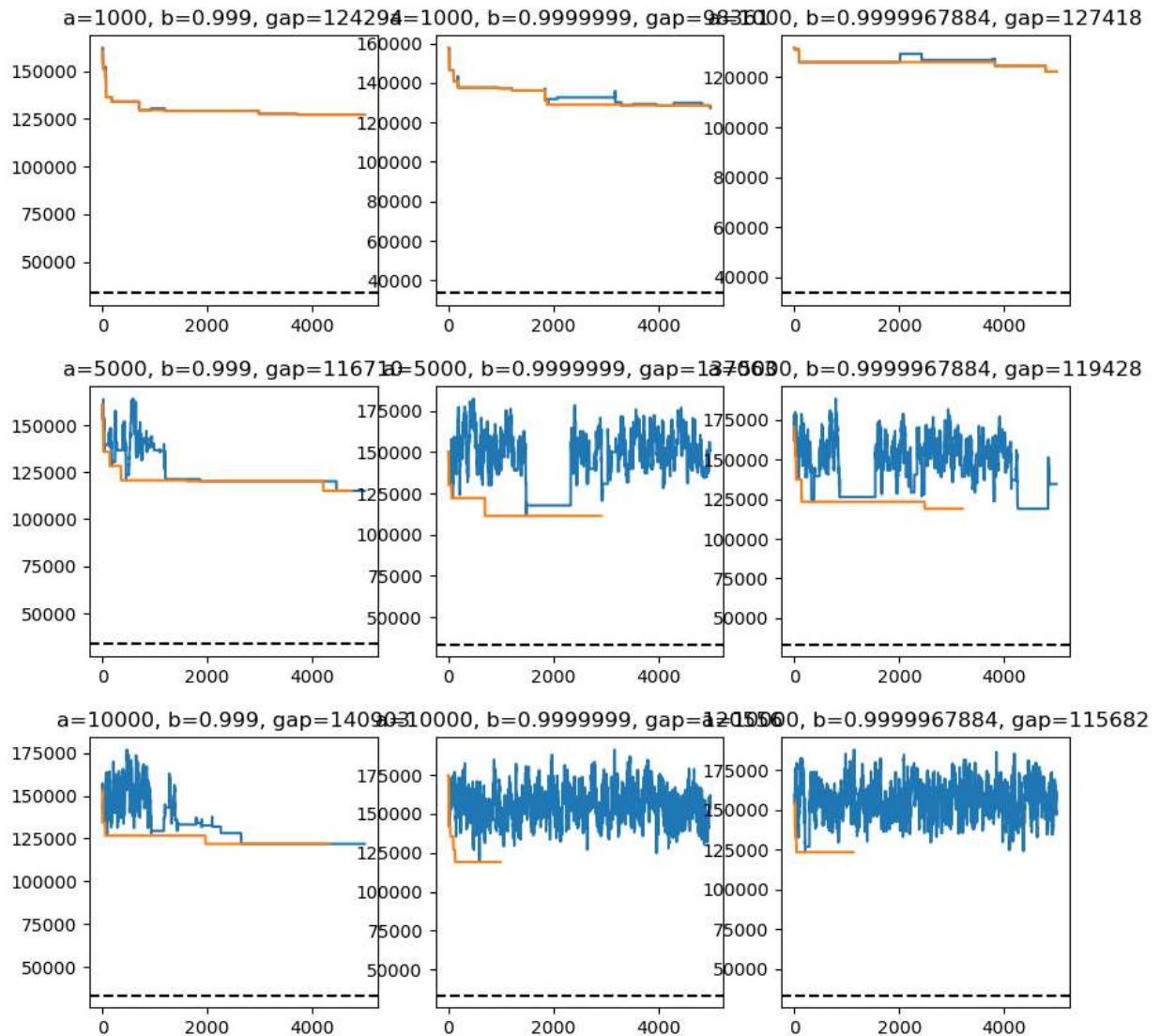
```

def T_exponential(a, b):
    def T(i):
        return a * (b ** i)
    return T

```

TODO : faire des tests similaires pour les températures exponentielles. a est de l'ordre de 10^3 - 10^4 , et b doit être proche de 1. Si on veut faire tourner le code pour $N_{iter} = 5000$, il vaut mieux choisir b entre 0.999 et 1 pour éviter des erreurs de compilation (overflow).

```
In [61]: plt.figure(figsize=(10,10))
plt.subplots_adjust(hspace=0.3)
i = 1
for a in [1000, 5000, 10000]:
    for b in [0.999, 0.9999999, 0.9999967884]:
        T = T_exponential(a, b)
        tour, best, cseq, bseq = recuit_simule(distances, 5000, initial_tour, proposition)
        gap = V(best, distances) - V(optimal, distances)
        plt.subplot(3,3,i)
        plt.title("a={}, b={}, gap={}".format(a,b,round(gap)))
        plot_recuit(cseq, bseq, best, optimal, legend=False)
        i += 1
plt.show()
```



On peut aussi utiliser des schémas non décroissants : l'idée est alors d'alterner les phases de réchauffement et de refroidissement. On essaye ici

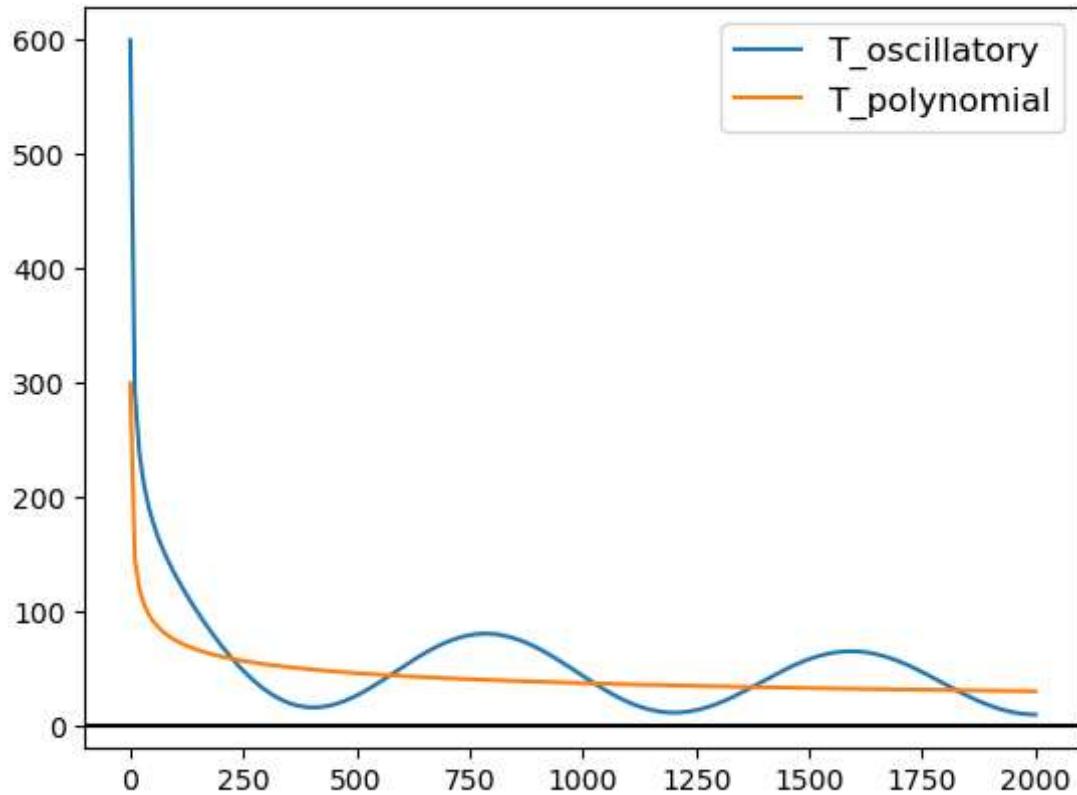
$$T_{oscillatory}(i) = a(i+1)^b(\cos((2\pi i)/c)^2 + d).$$

```
In [62]: def T_oscillatory(a, b, c, d):
    def T(i):
        return a*((i+1)**b)*(np.cos(2*np.pi*i/c)**2 + d)
    return T
```

```

x = np.linspace(0,2000,200)
plt.plot(x, T_oscillatory(500,-0.3,1600,0.2)(x), label="T_oscillatory")
plt.plot(x, T_polynomial(300,-0.3)(x), label="T_polynomial")
plt.axhline(0, c="black")
plt.legend(fontsize=12)
plt.show()

```

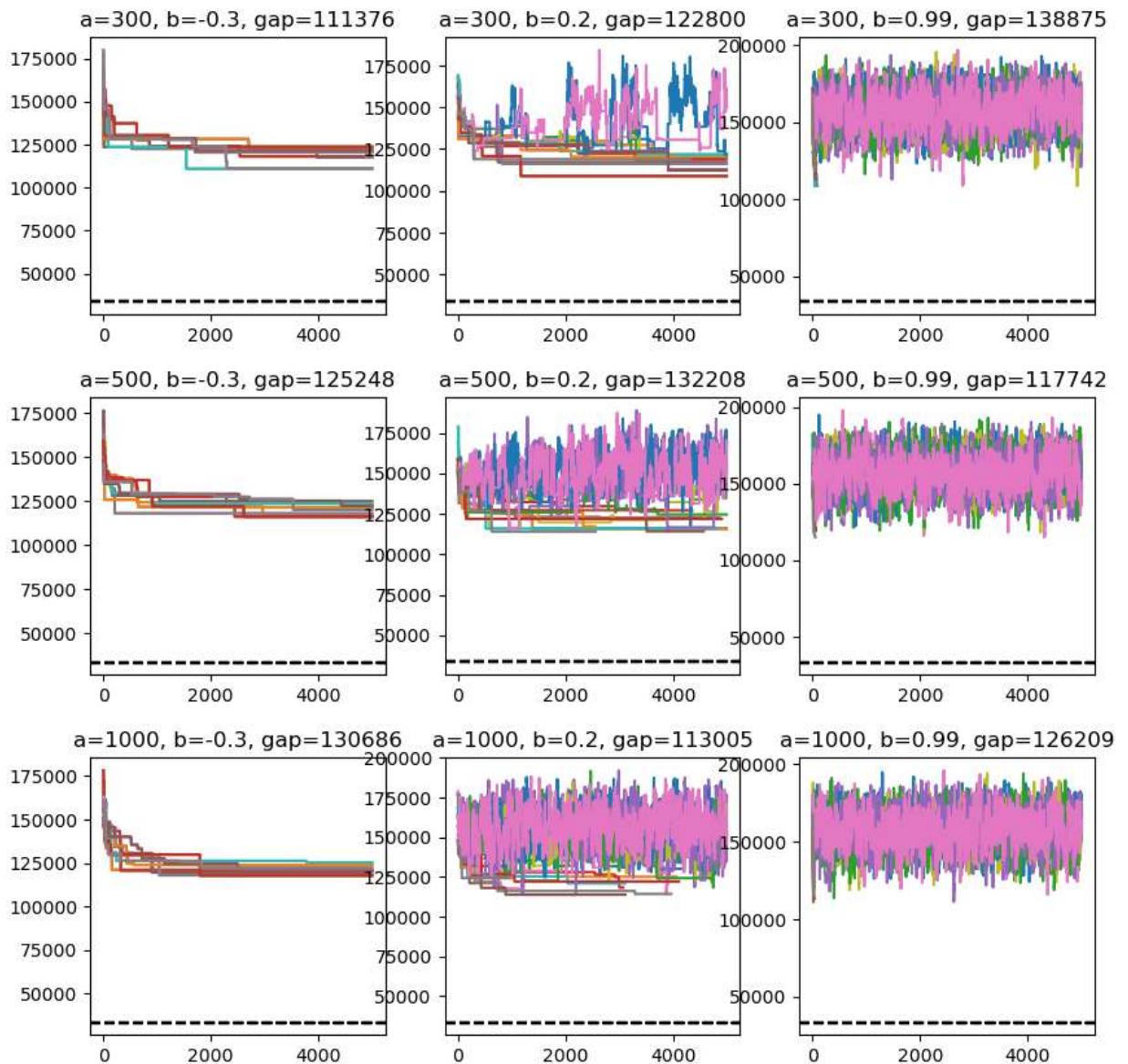


TODO : observer le comportement du schéma de température oscillant et l'influence des paramètres du schéma sur le résultat.

```

In [63]: plt.figure(figsize=(10,10))
plt.subplots_adjust(hspace=0.3)
i = 1
for a in [300, 500, 1000]:
    for b in [-0.3, 0.2, 0.99]:
        for c in [1600,2000,5000]:
            for d in [0.2,0.6,2]:
                T = T_oscillatory(a, b,c, d)
                tour, best, cseq, bseq = recuit_simule(distances, 5000, initial_tour,
                gap = V(best, distances) - V(optimal, distances)
                plt.subplot(3,3,i)
                plt.title("a={}, b={}, gap={}".format(a,b,round(gap)))
                plot_recuit(cseq, bseq, best, optimal, legend=False)
                i += 1
plt.show()

```



5) Pour aller plus loin...

Vous pouvez tester l'algorithme sur des instances plus difficiles. Les données sont organisées comme celles du problème `us_capitals`.

- Le problème `gr96` consiste à calculer le plus court cycle reliant 96 villes en Afrique
- Le problème `tsp225` relie 225 villes générées automatiquement

Dans le jeu de données `gr96`, les villes sont données en coordonnées sphériques (latitude, longitude). Pour calculer les distances entre villes, on utilise la fonction `earth_dist`, définie ci-dessous à l'aide du module `geopy` ([disponible via pip](#)):

```
In [64]: !pip install geopy
from geopy import distance

def earth_dist(citya, cityb):
    return distance.distance(citya, cityb).miles
```

```
Requirement already satisfied: geopy in c:\users\asus\anaconda3\lib\site-packages (2.4.1)
```

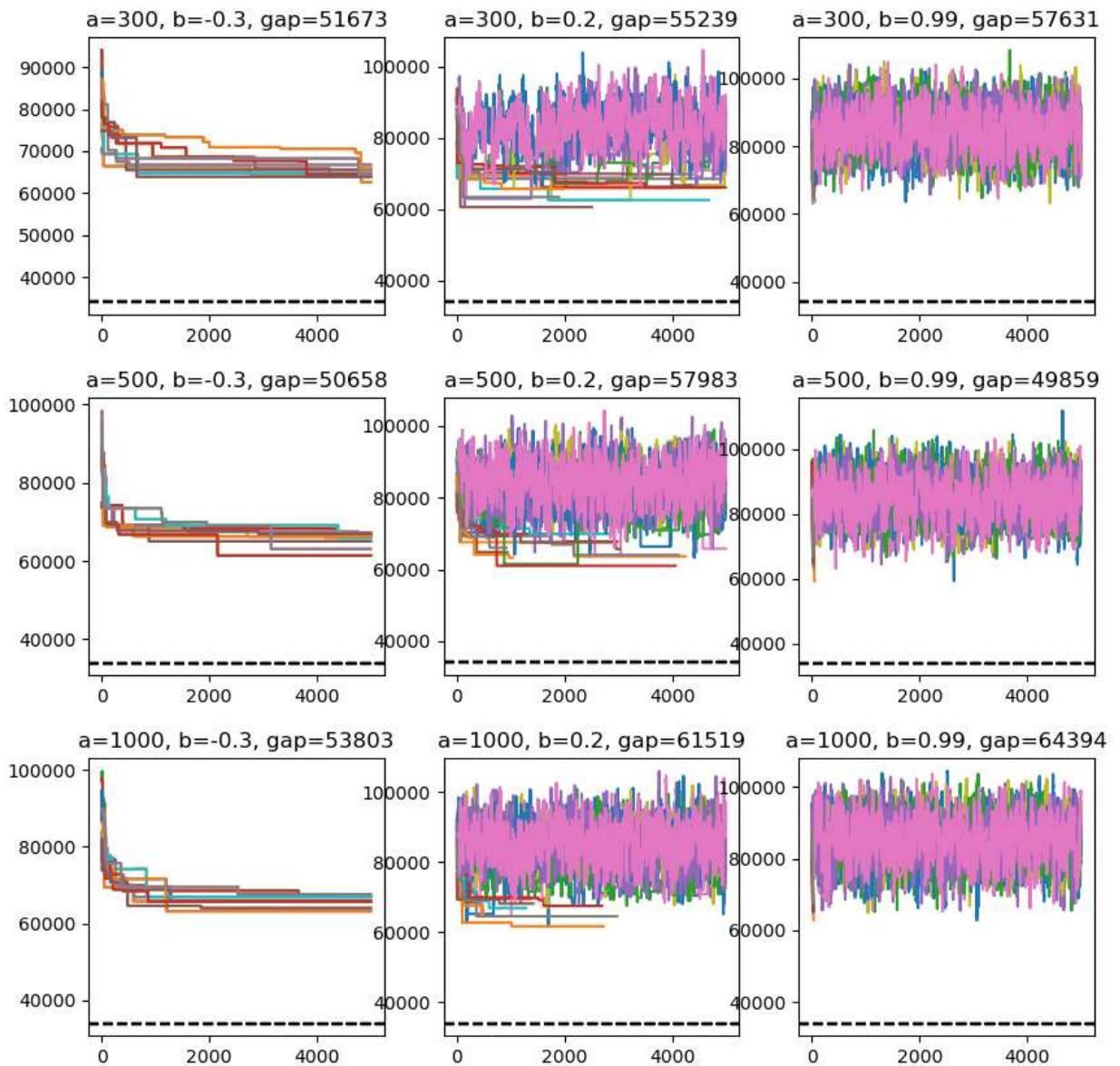
```
Requirement already satisfied: geographiclib<3,>=1.52 in c:\users\asus\anaconda3\lib\site-packages (from geopy) (2.0)
```

```
In [65]: gr96_list = np.genfromtxt("gr96.csv", delimiter=",")  
gr96_opt = np.genfromtxt("gr96_opt.csv", delimiter=",").astype(int)-1  
gr96_distances = np.array([[earth_dist(ca, cb) for ca in gr96_list] for cb in
```

```
In [66]: tsp225_list = np.genfromtxt("tsp225.csv", delimiter=",")  
tsp225_opt = np.genfromtxt("tsp225_opt.csv", delimiter=",").astype(int)-1  
tsp225_distances = np.array([[euclidian_dist(ca, cb) for ca in tsp225_list] for cb in
```

```
In [67]: cities = gr96_list  
optimal = gr96_opt  
distances = gr96_distances
```

```
In [68]: plt.figure(figsize=(10,10))  
plt.subplots_adjust(hspace=0.3)  
i = 1  
for a in [300, 500, 1000]:  
    for b in [-0.3, 0.2, 0.99]:  
        for c in [1600,2000,5000]:  
            for d in [0.2,0.6,2]:  
                T = T_oscillatory(a, b,c, d)  
                tour, best, cseq, bseq = recuit_simule(distances, 5000, initial_tour,  
                gap = V(best, distances) - V(optimal, distances)  
                plt.subplot(3,3,i)  
                plt.title("a={}, b={}, gap={}".format(a,b,round(gap)))  
                plot_recuit(cseq, bseq, best, optimal, legend=False)  
                i += 1  
plt.show()
```



```
In [76]: problem_initial_tours = []
for i in range(10):
    problem_initial_tours.append(optimal.copy())
    for k in range(i):
        problem_initial_tours[i] = proposition(problem_initial_tours[i])
```

```
In [77]: N_iter = 10000000
i = 8
initial_tour = problem_initial_tours[i]

tour, best, cseq, bseq = recruit_simule(distances, N_iter, initial_tour, proposition, 1)
```

```
In [78]: tt = optimal
total = len(cities)

plt.figure(figsize=(12,6))

plt.subplot(1, 2, 1)
plt.title('Solution optimale')
for i in range(total-1):
    plt.plot(np.array([cities[tt[i], 0], cities[tt[i+1], 0]]),
```

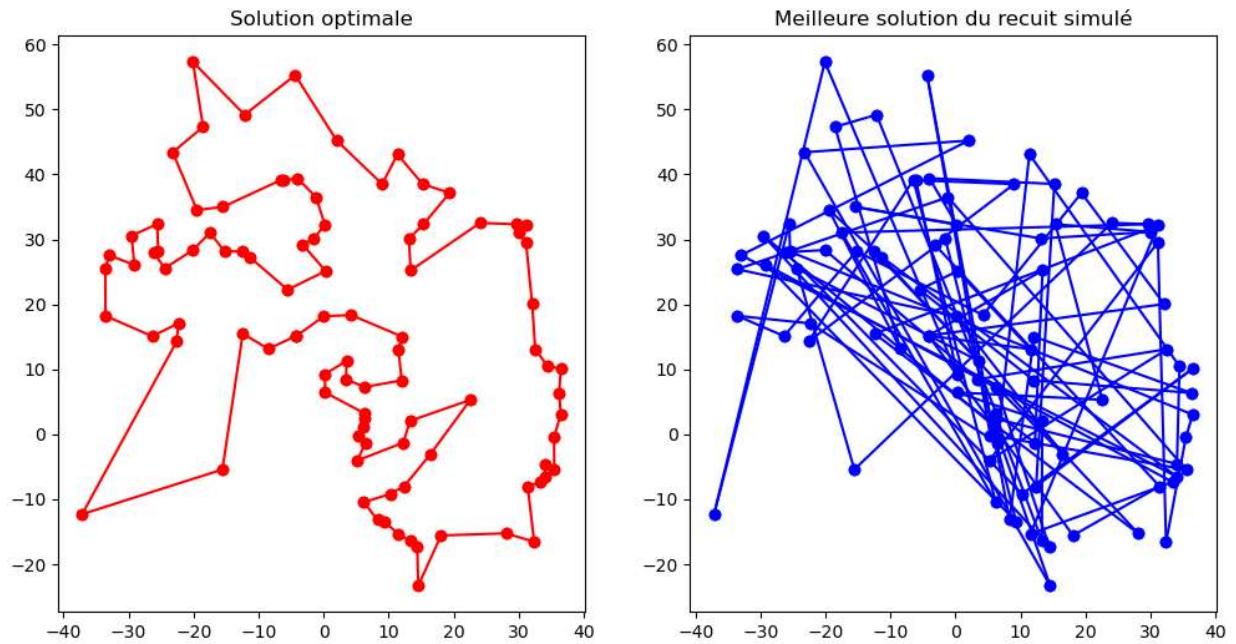
```

        np.array([cities[tt[i], 1], cities[tt[i+1], 1]]), '-ro')
plt.plot(np.array([cities[tt[total-1], 0], cities[tt[0], 0]]), np.array(
    [cities[tt[total-1], 1], cities[tt[0], 1]]), '-ro')

plt.subplot(1, 2, 2)
plt.title('Meilleure solution du recuit simulé')
for i in range(total-1):
    plt.plot(np.array([cities[best[i], 0], cities[best[i+1], 0]]),
            np.array([cities[best[i], 1], cities[best[i+1], 1]]), '-bo')

plt.plot(np.array([cities[best[total-1], 0], cities[best[0], 0]]), np.array(
    [cities[best[total-1], 1], cities[best[0], 1]]), '-bo')
plt.show()

```



6) Conclusions

1. Que fait l'algorithme du recuit simulé ? Pour quels types de problèmes est-il pertinent ?
1. Quel est le rôle de la proposition de changement ? Comment la choisir ?
1. Quel est le rôle du schéma de température ? Comment le choisir ?