TP5 Correction

February 24, 2023

1 Algorithme de Metropolis et du recuit simulé

Dans une première partie, on utilise l'algorithme de Metropolis pour construire une chaine de Markov de probabilité invariante μ donnée.

Dans une deuxième partie, on implémente l'algorithme du recuit simulé pour optimiser une fonction sur un espace discret grand. Le problème d'optimisation considéré est celui du "voyageur de commerce" qui est un problème classique en optimisation discrète.

[1]:
```
import numpy as np
from scipy import stats
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme()
from numpy.random import default_rng
rng = default_rng()
```

1.1 Algorithme de Metropolis

Soit E un ensemble dénombrable et μ une probabilité sur E. L'algorithme de Metropolis permet de construire une chaîne de Markov de probabilité invariante μ.

Soit P une matrice stochastique, dite de sélection (ou de proposition), qui permet de parcourir l'espace d'état E et ρ la fonction définie par

$$\forall x, y \in E, \quad \rho(x,y) = \frac{\mu(y)P(y,x)}{\mu(x)P(x,y)} \wedge 1.$$

On considère la matrice stochastique

$$Q(x,y) = \begin{cases} P(x,y)\rho(x,y) & \text{si } x \neq y \\ 1 - \sum_{z \neq x} Q(x,z) & \text{si } x = y. \end{cases}$$

Alors sous les hypothèses adhoc, la chaine de Markov $(Y_n)_{n\geq 1}$ de transition Q a pour unique probabilité invariante μ et $(Y_n)_{n\geq 1}$ converge en loi vers μ.

1.2 Petit exemple sur Z

On considère la probabilité μ sur $E = \mathbf{Z}$ définie par

$$\forall x \in \mathbf{Z}, \quad \mu(x) = C \exp(-\frac{|x|}{5}).$$

Soit P la matrice de transition de la marche aléatoire symmétrique sur $\mathbf{Z}$ définie par $P(x, x-1) = P(x, x+1) = \frac{1}{2}$.

1.2.1 Question: simulation de la chaine de transition Q

Ecrire la dynamique de la chaine $(Y_n)_{n\geq 0}$ de matrice de transition Q (et de loi invariante μ). Coder cette dynamique dans une fonction `phi` de sorte que l'appel suivant crée un échantillon de 1 000 trajectoires sur $\{0, \dots, 100\}$. “‘{python} n_steps = 101 n_paths = 1000

sample = np.zeros((n_steps, n_paths)) for n in range(1, n_steps): phi(sample[n-1], sample[n]) “‘

[2]:
```
def mu(x):
    return np.exp(- np.abs(x) / 5)

def phi(in_n, out_np1):
    proposals = in_n + (2*rng.integers(2, size=len(in_n))-1)
    ratio_rho = np.minimum(mu(proposals) / mu(in_n), 1)
    accepted = rng.uniform(size=len(in_n)) < ratio_rho
    out_np1[accepted] = proposals[accepted]
    out_np1[~accepted] = in_n[~accepted]
    return out_np1
```

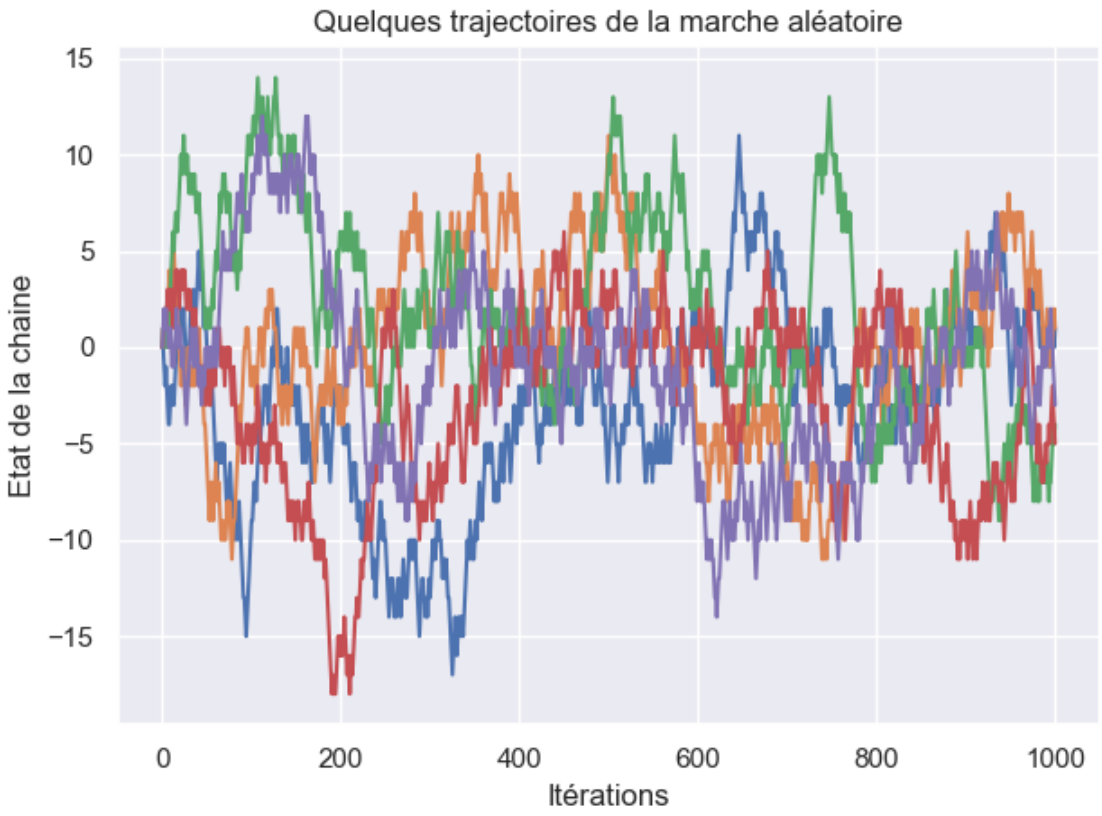
[3]:
```
n_steps = 1001
n_paths = 10000

sample = np.zeros((n_steps, n_paths))
for n in range(1, n_steps):
    phi(sample[n-1], sample[n])
```

1.2.2 Question: trajectoires et histogramme

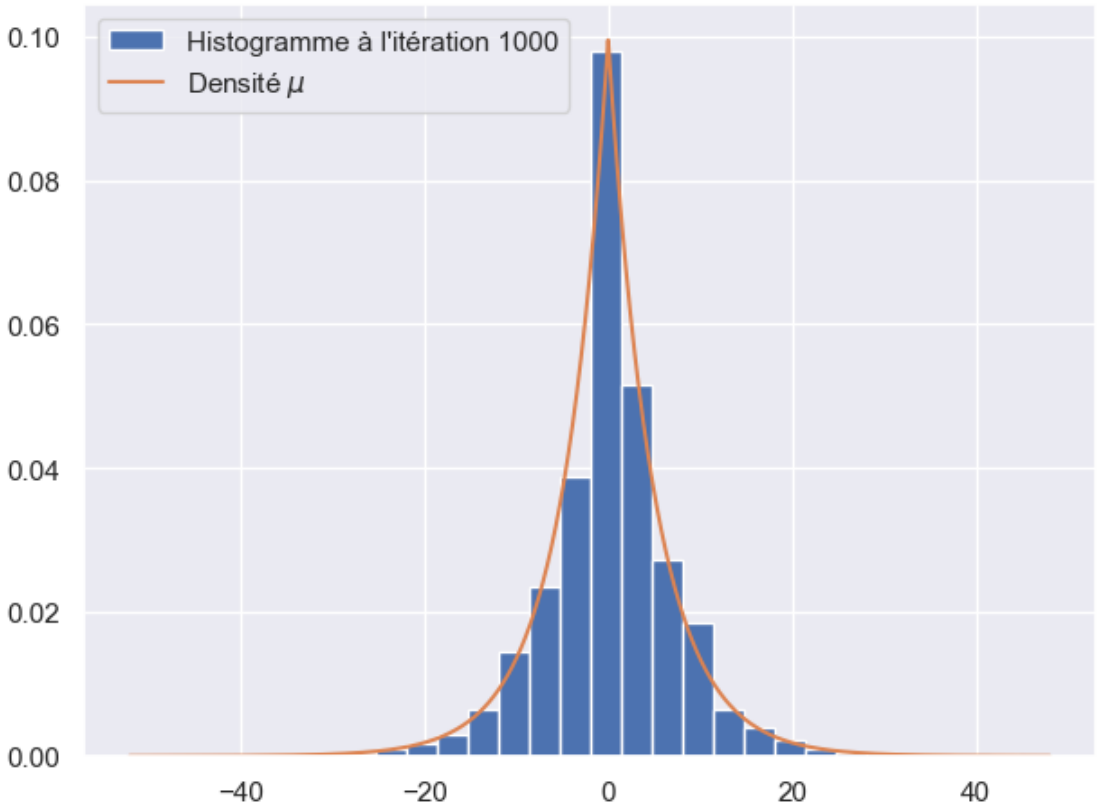
Visualiser quelques trajectoires de $(Y_n)_{n=0,\dots,1000}$ et comparer l'histogramme à l'itération 1000 avec la densité μ.

[4]:
```
fig, ax = plt.subplots(layout='tight')
ax.plot(np.arange(n_steps), sample[:,:5])
ax.set_ylabel('Etat de la chaine')
ax.set_xlabel('Itérations')
ax.set_title('Quelques trajectoires de la marche aléatoire')
plt.show()
```



[5]:
```
xx = np.linspace(min(sample[-1]), max(sample[-1]), 5000)
cnorm = 2/(1-np.exp(-0.2))-1
yy = mu(xx) / cnorm

fig, ax = plt.subplots(layout='tight')
ax.hist(sample[-1], bins=30, density=True, label=f"Histogramme à l'itération␣
↪{n_steps-1}")
ax.plot(xx, yy, label=fr"Densité $\mu$")
ax.legend()
plt.show()
```



1.3 Algorithme du recuit simulé

Soit E fini et $H : E \to \mathbf{R}$ une fonction dont on cherche un minimum. On considère une mesure de Gibbs paramétrée par $T > 0$ de la forme

$$\mu_T(x) = \frac{1}{Z_T} \exp(-\frac{H(x)}{T}),$$

où Z_T est la constante de normalisation souvent appelée fonction de partition. Lorsque T tend vers 0 la mesure μ_T se concentre sur l'ensemble des minimums de H, on considère donc une suite déterministe $(T_n)_{n\geq 1}$ qui tend vers 0 et l'algorithme de Metropolis (inhomogène) suivant:

$$Q_n(x,y) = \begin{cases} P(x,y) \exp\left(-\frac{(H(y)-H(x))_+}{T_n}\right) & \text{si } x \neq y \\ 1 - \sum_{z \neq x} Q_n(x,z) & \text{si } x = y. \end{cases}$$

où P est une matrice stochastique *irréductible symétrique*.

1.4 Problème du voyageur de commerce

L'énoncé du problème du voyageur de commerce est le suivant : étant donné N points (des « villes ») et les distances séparant chaque point, trouver un chemin de longueur totale minimale qui passe

exactement une fois par chaque point et revienne au point de départ. La distance considérée ici est la distance euclidienne.

Soit N villes fixées représentées par un vecteur $V = (V_1, \dots, V_N) \in ([0,1]^2)^N$. Une composante de ce vecteur V est la coordonnée d'une ville dans $[0,1]^2$ et la distance euclidienne d entre 2 villes V_i et V_j est notée dans la suite $d(V_i, V_j)$. Un trajet est identifié par une permutation de $\{1, \dots, N\}$.

Ainsi $E = \mathcal{S}_N$ l'ensemble des permutations de $\{1, \dots, N\}$ et H que l'on appelle ici la longueur d'un trajet $x \in E$ est définie par

$$\forall x \in E, \quad H(x) = \sum_{i=1}^{N-1} d(V_{x_i}, V_{x_{i+1}}) + d(V_{x_N}, V_{x_1})$$

Pour information il y a $\text{Card}\,(H(E)) = \frac{N!}{2N}$ longueurs de trajets possibles.

1.4.1 Question: chargement des données

En python on peut accéder en lecture à un fichier texte en utilisant la fonction `open` qui crée un objet que l'on peut interoger (parcourir) pour lire le contenu du fichier. La méthode `readlines` d'un tel objet crée une liste donc chaque élément est une chaine de caractère. Par exemple la cellule suivante renvoie les 4 premières lignes du fichier `'data/villes.csv'`.

[6]:
```
file = open('data/villes.csv', mode = 'r')
lines = file.readlines()
file.close()
lines[:4]
```

[6]:
```
[' x, y\n',
 '0.509881702484563, 0.323756906203926\n',
 '0.304272808600217, 0.0117385489866138\n',
 '0.564815347781405, 0.434346224879846\n']
```

En utilisant les méthodes `split` et `strip` des chaines de caractères créer un `np.array` `villes` contenants les coordonnées des 30 villes du fichier `data/villes.csv`. Le `np.array` doit donc avoir 30 lignes et 2 colonnes et être composé de réels (`float64`).

Alternative: utiliser la fonction `read_csv` du module `pandas`.

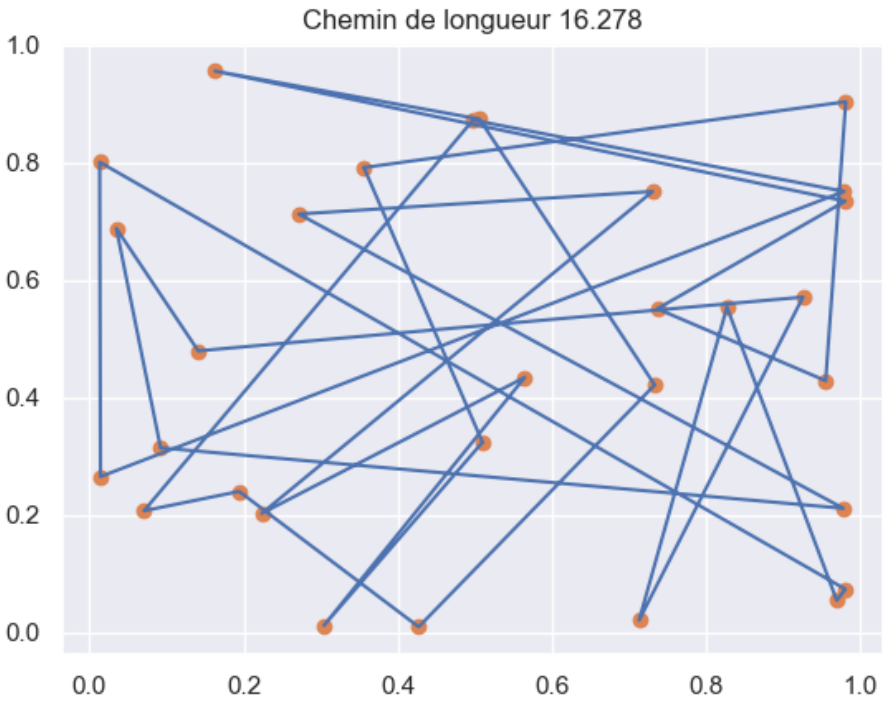
[7]:
```
villes_list = []
for line in lines[1:]:
    line = line.split(',')
    coords = [float(i.strip()) for i in line]
    villes_list.append(coords)
villes = np.array(villes_list)
```

[8]:
```
import pandas as pd
df = pd.read_csv("data/villes.csv")
villes = df.to_numpy()
```

[9]:
```
N, d = villes.shape
```

1.4.2 Question: préparatifs

- Définir une fonction `distance(v1, v2)` qui calcule la distance euclidienne entre 2 villes (2 lignes de la matrice `villes`).
- Définir une fonction `H(x)` qui à une configuration $x \in E$ (une permutation de $\{1, \dots, n\}$) renvoie la longueur du trajet (longueur totale en revenant au point de départ).
- Définir une fonction `plot_path(x, ax)` qui affiche le chemin de la configuration `x` dans l'objet `ax` de type `Axes`. Par exemple pour la configuration initiale `x = np.arange(N)` vous devez obtenir ce tracé:



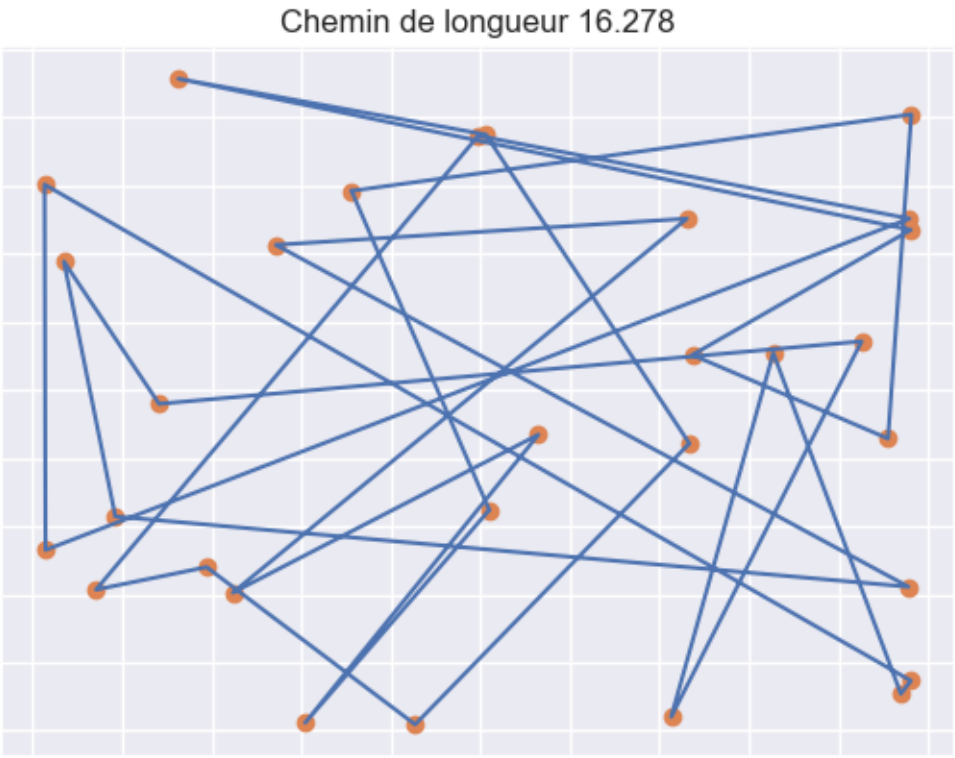
[10]:
```
def distance(v1, v2):
    return np.sqrt(np.sum((v1-v2)**2, axis=-1))
```

[11]:
```
def H(x):
    dist = np.sum(distance(villes[x[:-1]], villes[x[1:]]))
    dist += distance(villes[x[-1]], villes[x[0]])
    return dist
```

[12]:
```
def plot_path(x, ax):
    ax.plot(villes[x][:,0], villes[x][:,1])
    ax.plot(villes[x[[-1,0]]][:, 0], villes[x[[-1,0]]][:,1], color='C0')
    ax.scatter(villes[x][:,0], villes[x][:,1], color='C1')
    ax.set_title(f"Chemin de longueur {H(x):.3f}")
```

```
    ax.set_xticks(np.linspace(0, 1, 11), labels=[])
    ax.set_yticks(np.linspace(0, 1, 11), labels=[])
    return ax
```

[13]:
```
x0 = np.arange(N)
fig, ax = plt.subplots()
plot_path(x0, ax)
plt.show()
```



Pour mettre en oeuvre l'algorithme de recuit simulé il faut choisir une matrice de proposition P pour parcourir l'espace et une suite $(T_n)_{n\geq 1}$ qui tend vers 0. Dans la suite on définit la fonction de température T par

$$\forall n \geq 1, \quad T(n) = \frac{0.01}{\log(n+1)}$$

[14]:
```
def T(n):
    return 0.01 / np.log(n+1)
```

1.4.3 Question: première matrice de proposition P_1

La première matrice de transition que l'on considère pour explorer l'espace des configurations $E = \mathcal{S}_n$ est la plus intuitive. Etant donné un trajet $x \in E$ on tire 2 indices i et j de $\{1, \dots, N\}$ et

on échange ces indices c'est à dire qu'on propose le trajet voisin y défini si $i < j$ (par exemple) par

$$y = (x_1, \dots, x_{i-1}, x_j, x_{i+1}, \dots, x_{j-1}, x_i, x_{j+1}, \dots, x_N)$$

Définir une fonction `phi_P1` qui code la dynamique basée sur cette proposition. La fonction prend un argument `x` et renvoie un trajet voisin `y`.

Attention: l'argument `x` ne doit pas être modifié. Tester votre fonction.

[15]:
```
def phi_P1(x):
    i, j = rng.integers(N, size=2)
    y = np.copy(x)
    y[[i, j]] = x[[j, i]]
    return y
```

1.4.4 Question: algorithme du recuit simulé

Définir une fonction `phi_Q(phi_P, T, xn, n)` qui construit le prochain état de la chaine de Markov de transition Q_n sachant qu'on est dans l'état x_n à l'itération n.

Ecrire une fonction

`recuit_simule(phi_P, T=T, x0=np.arange(N), n_iters=3000)`

qui applique l'algorithme du recuit simulé sur `n_iters` itérations et renvoie le dernier trajet obtenu et la liste de toutes les longueurs obtenues de `0` à `n_iters`.

Tracer le chemin et le graphe des longueurs en fonction de n. Répliquer plusieurs fois pour obtenir différents résultats (assez variables).

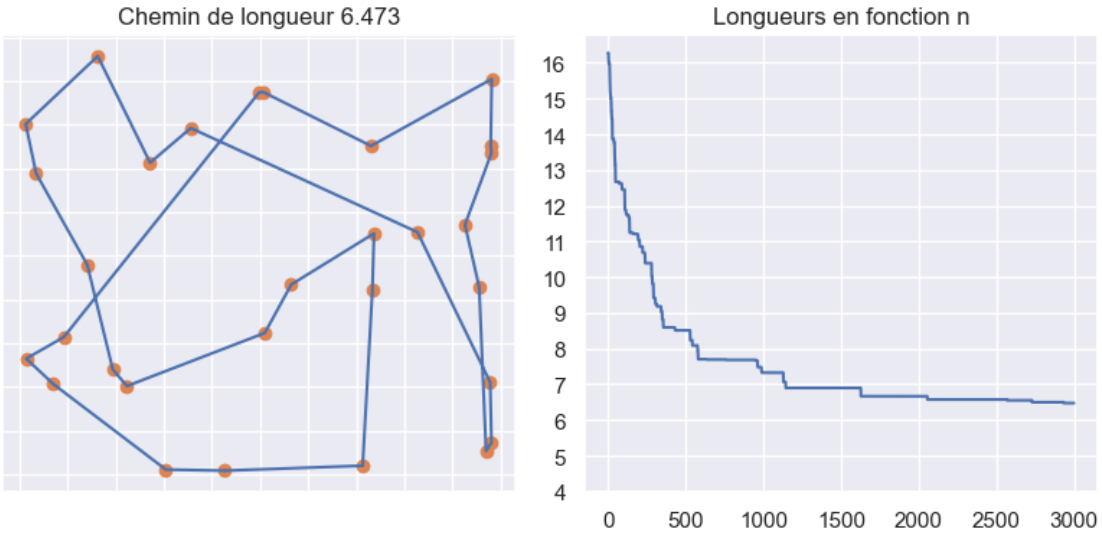
[16]:
```
def phi_Q(phi_P, T, xn, n):
    xnp1 = phi_P(xn)
    delta = H(xnp1) - H(xn)
    if delta < 0 or rng.random(1) < np.exp(-delta / T(n)):
        return xnp1
    else:
        return xn
```

[17]:
```
def recuit_simule(phi_P, T=T, x0=np.arange(N), n_iters=3000):
    longueurs = [ H(x0) ]
    x = x0
    for n in range(1, n_iters+1):
        x = phi_Q(phi_P, T, x, n)
        longueurs.append(H(x))
    return x, longueurs
```

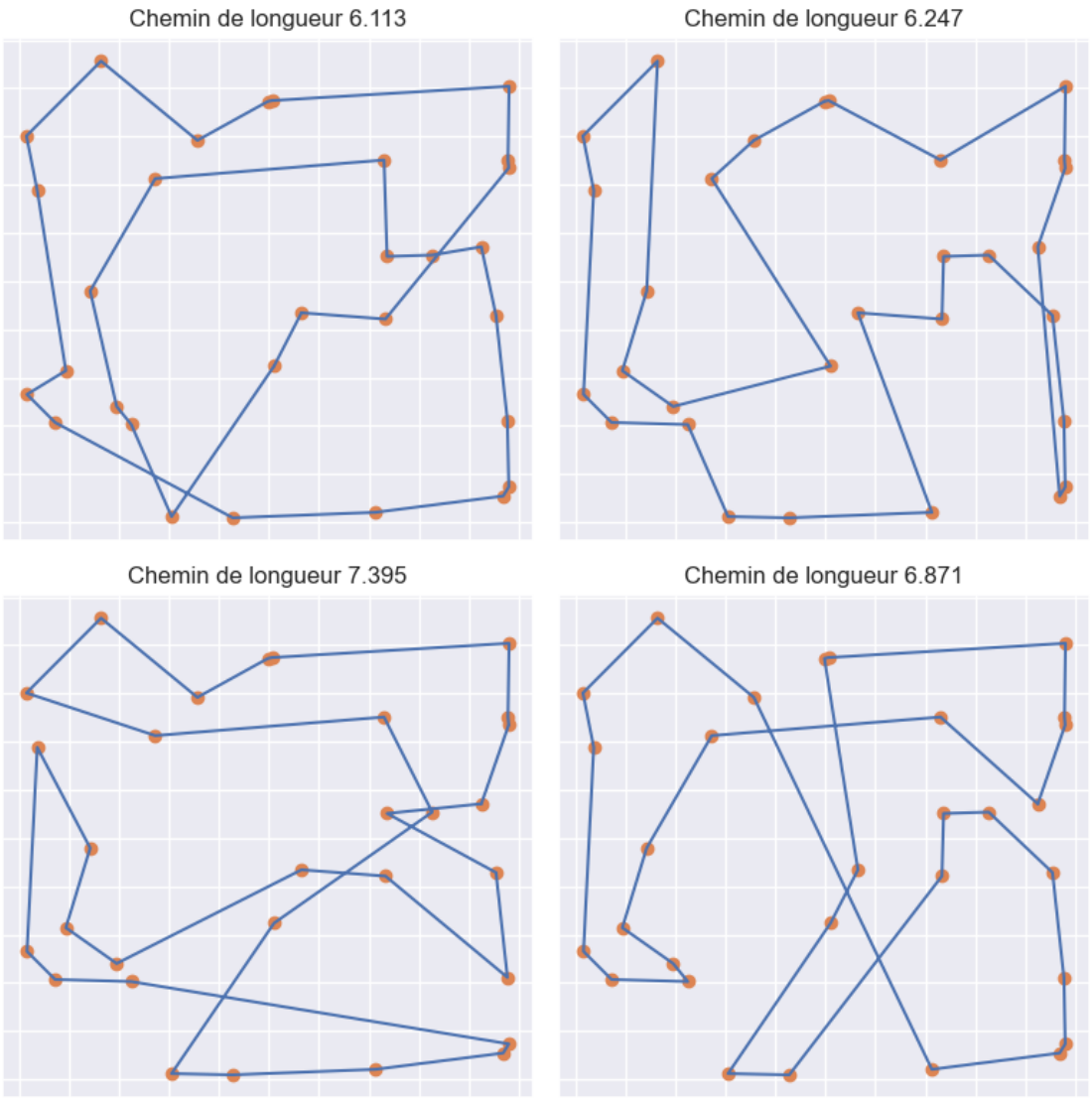
[18]:
```
xn, longueurs = recuit_simule(phi_P1)

fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(8,4), layout='tight')
plot_path(xn, ax1)
ax2.plot(np.arange(len(longueurs)), longueurs)
```

```
ax2.set_title("Longueurs en fonction n")
ax2.set_yticks(np.arange(4, 17))
plt.show()
```



[19]:
```
fig, axs = plt.subplots(nrows=2, ncols=2, figsize=(8,8), layout='tight')
for ax in axs.flat:
    xn, _ = recuit_simule(phi_P1)
    plot_path(xn, ax)
plt.show()
```



1.4.5 Question: variantes sur la matrice de sélection

- La première variante consiste à supprimer le cas où l'on propose en $n+1$ un trajet similaire à celui en n. C'est le cas avec la matrice P_1 car $P_1(x,x) = \frac{1}{N} > 0$. Pour cela il suffit de tirer 2 indices i et j de $\{1,\dots,N\}$ sans remise, et on aura donc $i \neq j$.
- La seconde variante consiste à partir de 2 indices i et j différents de couper le trajet en ces indices et de retourner le trajet entre ces 2 indices. Ainsi le trajet voisin proposé y à partir de x est défini par

$$y = (x_1, \dots, x_{i-1}, x_j, x_{j-11}, \dots, x_{i+1}, x_i, x_{j+1}, \dots, x_N)$$

Coder les fonctions `phi_P2` et `phi_P3` similaires à `phi_P1` qui correspondent aux variantes proposées ci-dessus.

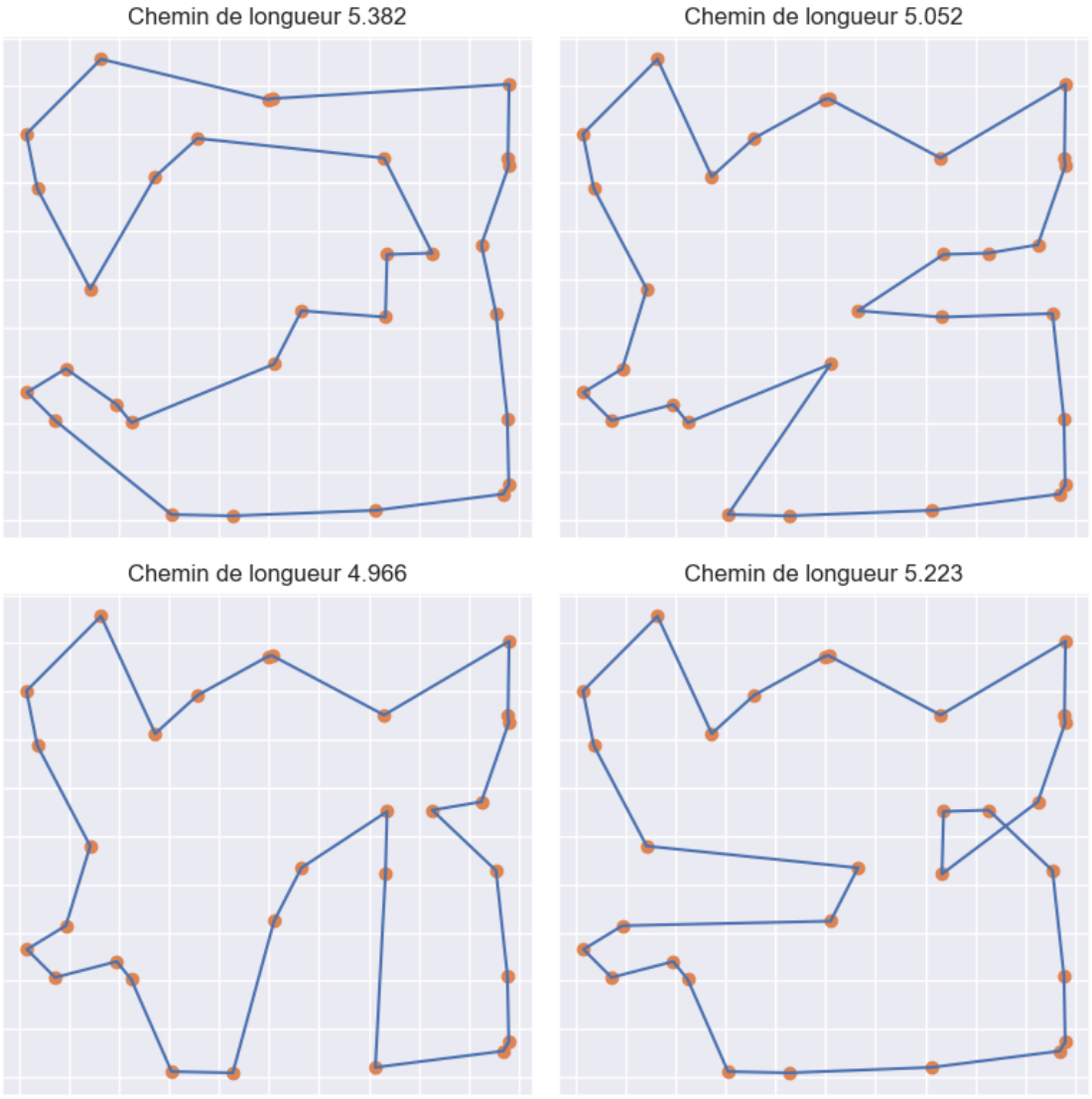
[20]:
```
def phi_P2(xn):
    i, j = rng.choice(N, size=2, replace=False)
    xnp1 = np.copy(xn)
    xnp1[[i, j]] = xn[[j, i]]
    return xnp1
```

[21]:
```
def phi_P3(xn):
    ij = rng.choice(N, size=2, replace=False)
    i, j = np.min(ij), np.max(ij)
    xnp1 = np.hstack([xn[:i], np.flip(xn[i:j]), xn[j:]])
    return xnp1
```

1.4.6 Question: résultats avec ces matrices de proposition

Appliquer l'algorithme de recuit simulé pour chaque stratégie d'exploration proposée. Répliquer plusieurs fois pour obtenir différents résultats.

[22]:
```
fig, axs = plt.subplots(nrows=2, ncols=2, figsize=(8,8), layout='tight')
for ax in axs.flat:
    xn, _ = recuit_simule(phi_P3)
    plot_path(xn, ax)
plt.show()
```



1.4.7 Question: extensions

Lire le fichier `data/best.csv` qui contient la configuration `x` optimale, c'est à dire de longueur minimale (sauf si vous trouvez mieux!). Tracer ce chemin.
Attention: les villes sont numérotées de 1 à N dans le fichier…

Quelque suggestions pour aller plus loin:

- Reprendre les questions précédentes en faisant varier la fonction de température.

- Modifier votre code pour construire simultanément M réalisations indépendantes de l'algorithme du recuit simulé.
- Donner la répartition empirique sur M trajectoires des longueurs trouvées pour chaque matrice de proposition P_1, P_2 et P_3.

[23]:
```
file = open('data/best.csv', mode = 'r')
x = np.empty(N, dtype=int)
for k, line in enumerate(file.readlines()):
    x[k] = int(line.strip())-1
file.close()
```

[24]:
```
fig, ax = plt.subplots()
plot_path(x, ax)
ax.set_title(fr"Meilleur chemin trouvé, longueur {H(x):.3f}")
plt.show()
```