Projet Processus Markovien en Temps Continu

Mohamed Jedny
mjedny@polytech-lille.net

Younoussa Sow
ysow@polytech-lille.net

Yue Sheng
ysheng@polytech-lille.net

6 avril 2021

Table des matières

1 Introduction

Le problème du voyageur de commerce est un problème en **NP-Complet**, c'est-à-dire que c'est un problème dont la solution exacte est difficile. Parmi les problèmes NP-complets notables, on peut citer :le problème du sac à dos, le problème du voyageur de commerce, les problèmes de coloration de graphe, etc. Ce que nous allons étudier cette fois, c'est **le problème du voyageur de commerce**.

En informatique, le problème du voyageur de commerce, ou problème du commis voyageur, est un problème d'optimisation qui, étant donné une liste de villes, et des distances entre toutes les paires de villes, détermine un plus court chemin pour visiter chaque ville une et une seule fois et qui se termine dans la ville de départ.

Pour résoudre ce problème, nous devons appliquer des algorithmes d'approximations qui vont nous permettre de trouver des solutions approchées de trajet optimale en un temps raisonnable pour un certain nombre de problèmes NP-complets.

Certains algorithmes connus sont les suivants :

— Les algorithmes d'optimisation stochastique

— Voisin le plus proche (nearest neighbor)

— Algorithme de recuit simulé (Recuit Algorithm)

— Algorithme de réseau neuronal

2 Recherche de la solution par une exploration exhaustive

Nous avons choisi d'implémenter la solution de cette question avec le `langage C`. Pour cela nous avons implémenter des modules qui vont nous servir d'avoir un code modulaire et facile à prendre en main. Nous avons un module `vext` qui est un tableau extensible, l'intérêt est de pouvoir ajouter autant d'éléments que nécessaire dans ce tableau sans se soucier de connaître une taille précise au départ. Nous avons également un module `disterre` qui nous permettra d'avoir une fonction de coût (pour notre cas, la fonction de coût est le calcul de la distance d'un itinéraire donné). Dans ce module nous avons aussi définie une structure de données `position_terrestre` qui va représenter la coordonnée d'une ville. Pour représenter une ville en mémoire, nous avons définit une stucture de données `ville` composé du nom de la ville et sa coordonnée terrestre. Un itinéraire donné est donc un vecteur de type `vext` contenant l'ensemble des villes qui seront utilisées dans l'algorithme.

Pour explorer tous les itinéraires, il est nécessaire de faire la permutation entre toutes les villes et de chercher l'itinéraire dont la distance est minimale. On sait (et les mesures que nous allons effectuer vont nous le confirmer) qu'une telle approche est de complexité $\mathcal{O}(n!)$.

Pour obtenir la solution exacte, nous devons faire une permutation entre l'ensemble des villes. Cette permutation est de l'ordre de $n!$. Pour illustrer ce phénomène, nous allons mesurer le temps d'exécutions et la mémoire nécessaire à l'ensemble des permutations. Ensuite nous allons comparer ces deux mesures avec la fonction factorielle pour voir si le comportement est le même. Pour cela nous avons fait une représentation graphique de l'ensemble de nos mesures effectuées (cf Figure 1).

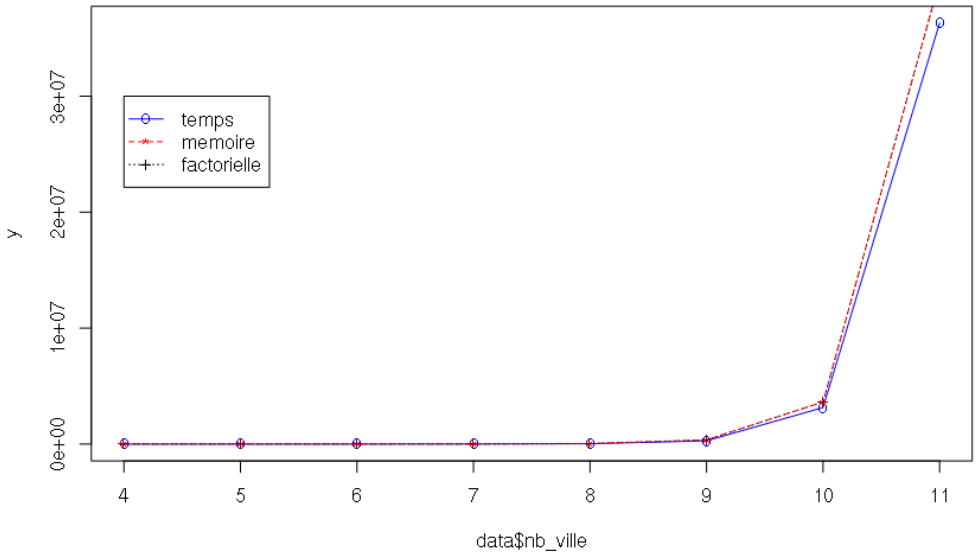

FIGURE 1 – Les courbes du temps, de la mémoire et de la fonction factorielle

La recherche exhaustive n'est plus pertinente à partir de 12 villes car le temps d'execution et la mémoire sont de l'ordre de la factorielle.

Pour éxecuter le programme *C* que nous avons écrit pour cette partie, il faut se rendre dans le dossier *src* du code source et de lancer la commande *make*.

3 Mesures de Gibbs

3.1 La chaîne de Markov

3.1.1 Structure du graphe

On peut utiliser la structure de graphe suivante : deux chemins sont voisins si on peut passer de l'un à l'autre en permutant deux villes. Nous noterons alors $\sigma \sim \sigma'$ lorsque $\{\sigma,\sigma'\}$ est une arête du graphe. Nous dirons alors que les configurations σ et σ' sont voisines.

3.1.2 La mesure stationnaire

L'idée (géniale !) des méthodes MCMC consiste à construire une chaîne de Markov $(X_n)_{n\in N}$ sur un espace d'états E admettant π comme mesure stationnaire et convergeant vers π (quelque soit le point de départ $X_0 = x_0$ de la chaîne). Supposons avoir à disposition une telle chaîne de Markov $(X_n)_{n\in N}$.

On souhaite pouvoir simuler la loi de probabilité P_T. On sait qu'en partant d'un élément quelconque $x_0 \in E$ et au bout d'un temps n très grand, la loi de la v.a. X_n est proche de π. Autrement dit, la valeur X_n qu'il est facile d'obtenir puisque les chaînes de Markov se simulent en général aisément, est une réalisation approchée de la loi π.

Ainsi, vu qu'on veut simuler (de valeur approchée) P_T, la mesure stationnaire de notre chaîne de Markov doit être exactement la loi de probabilité P_T.

3.1.3 La matrice de transition

La proposition 3.2.1 du cours nous définit pour une chaîne de Markov sur E admettant π comme mesure réversible la matrice de transition $P = (p_{i,j})_{i,j\in E}$ suivante : Si $i \neq j$,

$$p_{i,j} = \begin{cases} q_{i,j}\min\{\frac{\pi_j q_{j,i}}{\pi_i q_{i,j}}, 1\} & \text{si } q_{i,j} \neq 0 \\ 0 & \text{sinon.} \end{cases}$$

Dans notre cas, la mesure stationnaire est la loi de probabilité P_T donc :

$$\frac{\pi_\sigma}{\pi_{\sigma'}} = \frac{P_T(\sigma)}{P_T(\sigma')} = e^{-(H(\sigma)-H(\sigma'))/T}$$

On a aussi la probabilité de transition d'une configuration σ vers une autre configuration σ' sera donnée par la formule :

$$q_{\sigma,\sigma'} = \frac{1}{Card(\eta\in E,\eta\sim\sigma)}$$

Vu que chaque ville à un nombre égal de voisins que son voisin alors :

$$\frac{q_{j,i}}{q_{i,j}} = 1$$

Ainsi on obtient : Pour $\sigma \neq \sigma'$,

$$p_{i,j} = \begin{cases} \frac{1}{Card(\eta\in E,\eta\sim\sigma)}\min\{e^{-(H(\sigma)-H(\sigma'))/T}, 1\} & \text{si } \sigma \sim \sigma' \\ 0 & \text{sinon.} \end{cases}$$

3.2 Algorithme pour simuler P_T

3.2.1 Pseudo-Code

Pour pouvoir simuler P_T, on doit simuler la chaîne de Markov pour un n très grand/ Depuis le résultat de la section précédente, on déduit un algorithme de simulation de la chaîne de Markov de matrice de transition P (la fonction Random fournit une réalisation de la loi uniforme $[0, 1]$:

```
1  Initialiser X;
2  n ← 0;
3  tant que n < N faire
4  |  i ← X;
5  |  Choisir j au hasard parmi les voisins de i;
6  |  rho ← e^{-(H(σ)-H(σ'))/T};
7  |  si rho >= 1 alors
8  |  |  X ← j
9  |  sinon
10 |  |  U ← Random;
11 |  |  si U<=rho alors
12 |  |  |  X ← j;
13 |  |  fin
14 |  fin
15 |  n ← n + 1;
16 fin
17 Retourner X
```

3.2.2 Analyse

On peut voir cet algorithme comme une sorte de descente de gradient aléatoire, dont les pas qui vont vers les basses valeurs de la fonction sont automatiquement effectués. S'il n'y avait que ceux-là, l'algorithme resterait piégé dans les minima locaux. Pour sortir de ces pièges, on autorise l'algorithme à effectier éventuellement des pas dans la mauvaise direction : lorsque le voisin tiré au sort correspond à une énergie plus grande, il n'est choisi qu'avec une certaine probabilité et dans le cas contraire, le processus reste sur place.

Lorsque le paramètre de température T est grand, la chaîne explore bien l'espace mais ne reste donc pas sur les minima de H mais l'exploration est très lente : il faudra beaucoup de temps pour sortir d'un voisinage d'un minimum local. Il s'agit de tirer avantage des deux comportements en neutralisant les inconvénients de chacun. "Un point i est un minimum local si H(i) <= H(j) pour tout j voisin de i"

4 Récuit simulé

4.1 Pourquoi utiliser l'algorithme du récuit simulé

Raisonnons sur un ensemble E avec peu d'élements, par exemple 3 villes, numérotées 1, 2 et 3. Un moyen de résoudre le problème est d'identifier tous les chemins possibles et d'en calculer la distance. La solution est évidemment le chemin dont la distance est la plus faible.

Les chemins possibles, sachant qu'il faut revenir au point de départ, sont : (1,2,3,1), (1,3,2,1), (2,3,1,2), (2,1,3,2), (3,2,1,3) et (3,1,2,3). Il y a 6 combinaisons de chemins possibles. En fait, le nombre de possibilités s'élève à n ! (n factorielle). Pour n=3, cela fait 6 possibilités. Pour 10 villes, cela fait 3,6 millions de combinaisons possibles et pour 20 villes $2{,}4.10^{18}$ combinaisons possibles ! La méthode exhaustive n'est pas envisageable ! Le problème du voyageur de commerce (PVC) est de complexité non polynomiale, un problème que l'on appelle NP-complet.

C'est pour cela que nous allons utiliser un algorithme qui ne nous donnera peut-être pas le résultat exact mais une très bonne estimation, d'autant meilleure que nous choisirons de privilégier la qualité du résultat sur le temps de calcul...

4.2 Relation entre récuit simulé et PVC

La fonction à optimiser est toute trouvée : il s'agit de la distance totale à parcourir par le voyageur, qui est la somme des distances entre chacune des paires de villes voisines. L'état du système est caractérisé par le trajet entre les différentes villes et la distance totale associée qui est représentée par son énergie. Pour modifier l'état du système, il suffit de permuter l'ordre des villes.

L'algorithme de Metropolis sera adapté : l'énergie du système est la distance d à minimiser, la variation d'énergie étant la différence de distance entre deux villes. La température est un paramètre de contrôle sans signification fonctionnelle particulière. Mais le choix des températures initiale et finale contraindra l'efficacité de l'algorithme.

4.3 Recuit simulé vs recherche exhaustive

En comparant le temps d'éxecution de l'algorithme du `Recuit simulé` et celui de l'algorithme `exhaustif`, nous obtenons les graphiques suivantes. Nous observons que l'algorithme du `recuit simulé` est déjà plus intéressant que l'algorithme `exhaustif`. En calculant les temps pour un nombre de villes comprises entre [4; 11] on trouve les temps : 5.114654541015625, 5.119335412979126, 5.147562265396118, 5.19987678527832, 5.3103320598602295, 5.37992787361145, 5.423942804336548, 5.3737156391143.

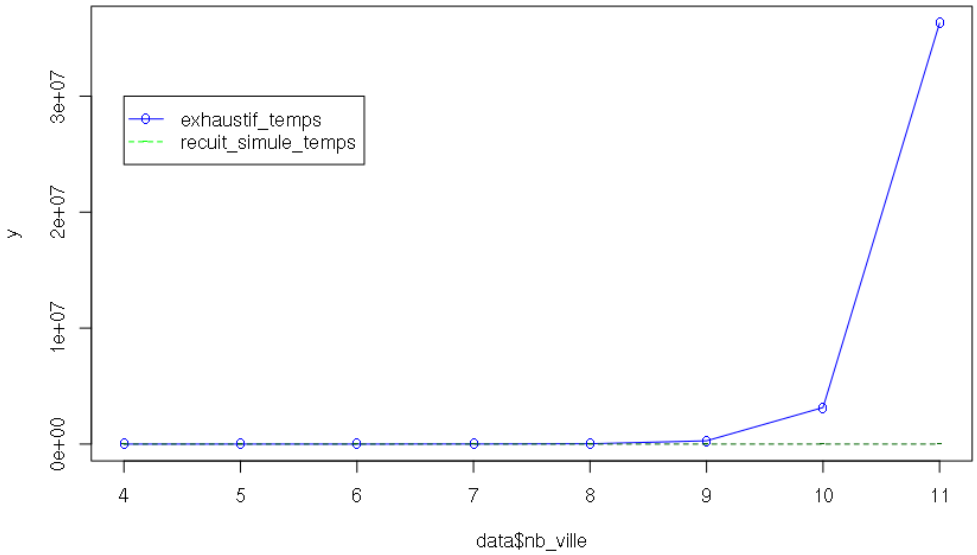

FIGURE 2 – Comparaison des temps d'éxecution des deux algos

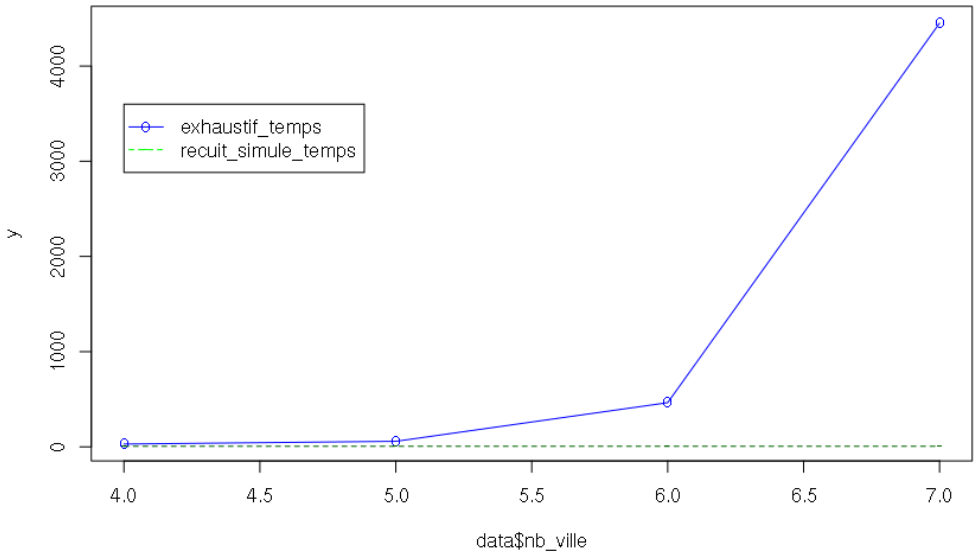

FIGURE 3 – Comparaison des temps d'éxecution des deux algos jusqu'à 7 villes

5 Script Python

5.1 Sélection des paramètres de l'algorithme

Les paramètres de température sont déclarés :

$$T0 = 10.0$$
$$Tmin = 1e\text{-}2$$
$$tau = 1e4$$

Le choix des valeurs de température intiale et de température minimum, celle qui fixe la fin du recuit, nécessite plusieurs essais. En règle générale, la valeur de T0 doit être du même ordre de grandeur que la variation d'énergie. La valeur de τ doit être choisie assez grande, pour que la diminution de la température soit suffisamment lente, mais pas trop pour ne pas avoir une masse trop grande de calculs.

Pour convertir nos coordonnées géographiques en cartesiennes on a besoin de définir le rayon de la terre et le centre de la france qui servira du point (0,0) au plan :

$$R = 6371.009 \text{ "}\textit{Rayon de la terre}\text{"}$$
$$\lambda 0 = 46.36 \text{ "}\textit{Latitude du centre}\text{"}$$
$$\varphi 0 = 2.29 \text{ "}\textit{Longitude du centre}\text{"}$$

5.2 Définition du problème

Dans un fichier *villes.txt* on a les données suivantes :

```
villes Latitude Longitude
MARSEILLE    43.30000  5.400000
PARIS    48.86667  2.333333
LILLE    50.63333  3.066667
BESANCON    47.25000  6.033333
BREST    48.40000 -4.483333
NANTES    47.21667 -1.550000
BORDEAUX    44.83333 -0.566667
PAU    43.30000 -0.366667
MONTPELLIER    43.60000  3.883333
```

FIGURE 4 – Une partie des données contenant la latitude et longitude des villes

On va récupérer les latitudes et longitudes et les transformer en coordonnées cartésiennes pour optimiser le calcul. Pour convertir on a utilisé la formule de la projection equirectangulaire :

λ est la longitude de la location qu'on veut projeter
φ est la latitude de la location qu'on veut projeter
$$x = R * (\lambda - \lambda 0) * \cos(\varphi 0)$$
$$y = R * (\varphi - \varphi 0)$$

Le trajet initial est simplement l'indexation du vecteur des villes selon leurs coordonnées. Il faut sauvegarder le trajet initial afin de pouvoir l'afficher à l'issue du calcul :

$$villes = lectureVilles("villes.txt")$$
$$N = len(villes)$$
$$trajet = arange(N)$$
$$trajet\text{-}init = trajet.copy()$$

5.3 La mesure de l'énergie du système

L'énergie totale du système est mesurée par la somme des distances entre les villes dans l'ordre de parcours du trajet. Dans le principe, il suffit donc de calculer la distance entre les villes consécutives sur le trajet et de sommer ces distances. Dans la pratique, il faut optimiser au mieux ce

calcul. Quelques petites précisions pour éclaircir ce code :
-L'opérateur c concatène deux matrices, ici les matrices 1D de chaque coordonnée des villes, afin de construire une matrice des coordonnées des villes dans l'ordre du trajet.
-La fonction roll, avec le paramètre -1, opère un shift gauche d'un élément (-1), ce qui code $(x_{i+1}x_i)2$ de manière assez rapide.

On obtient donc la fonction EnergieTotale (cf Figure 10) :

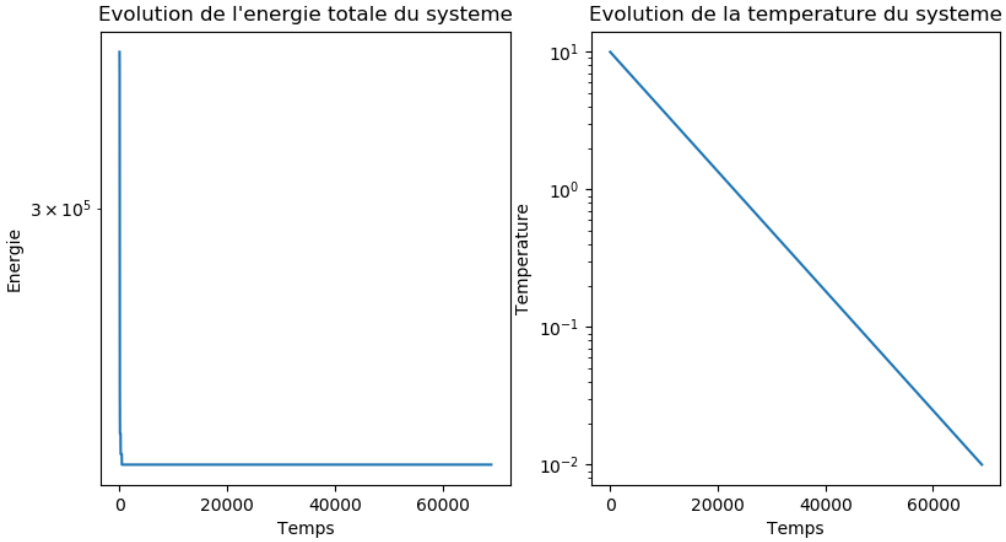

FIGURE 5 – Fonction de calcul de la distance totale

5.4 La création du fluctuation dans le système

La fonction Fluctuation (cf Figure 6) "swappe" un certain nombre de segments, fixés entre i et j, dans le trajet, ce qui simule une fluctuation d'énergie dans le système. Tel qu'elle est codée, cette fonction provoque le swap inverse si elle est appelée une seconde fois avec les mêmes valeurs (i,j).

```
# fonction de fluctuation autour de l'état "thermique" du système
def Fluctuation(i,j):
    global trajet
    Min = min(i,j)
    Max = max(i,j)
    trajet[Min:Max] = trajet[Min:Max].copy()[::-1]
    return
```

FIGURE 6 – Fonction de changement de l'état thermique du système

5.5 L'implémentation de l'algorithme de Metropolis

Retranscription d'un algorithme décrit en pseudo-code

```python
# fonction d'implémentation de l'algorithme de Metropolis
def Metropolis(E1,E2):
    global T
    if E1 <= E2:
        E2 = E1  # énergie du nouvel état = énergie système
    else:
        dE = E1-E2
        if np.random.uniform() > np.exp(-dE/T): # la fluctuation est retenue avec
            Fluctuation(i,j)              # la proba p. sinon retour trajet antérieur
        else:
            E2 = E1 # la fluctuation est retenue
    return E2
```

FIGURE 7 – Algorithme de Metropolis

5.6 La loi de refroidissement

Là aussi, rien de particulier à dire.On peut toutefois changer la loi de variation, à condition que cette variation soit lente, sinon l'algorithme de recuit simulé ne fonctionnera pas.

$$T = T0 * \exp(-t/tau)$$

5.7 La boucle principale de calcul

Dans la boucle principale, on reprend l'algorithme de recuit simulé. On provoque une fluctuation aléatoire d'énergie dans le système, en d'autres termes, on provoque une petite modification du trajet entre deux villes différentes. Après celà on calcule l'énergie du système après cette fluctuation, on applique l'algorithme de Metropolis et on applique la loi de refroidissement et on reboucle

```python
# boucle principale de l'algorithme de recuit simulé
t = 0
T = T0
while T > Tmin:
    # choix de deux villes diff?rentes au hasard
    i = np.random.randint(0,N)
    j = np.random.randint(0,N)
    if i == j: continue

    # création de la fluctuation et mesure de l'énergie
    Fluctuation(i,j)
    Ef = EnergieTotale()
    Ec = Metropolis(Ef,Ec)

    # application de la loi de refroidissement
    t += 1
    T = T0*np.exp(-t/tau)

    # historisation des données
    if t % 10 == 0:
        Henergie.append(Ec)
        Htemps.append(t)
        HT.append(T)

# fin de boucle - affichage des états finaux
```

FIGURE 8 – Boucle principale

6 Les résultats

On a lancé le script pour qu'il puisse lire les données des 22 villes dans le fichier "villes.txt". C'est très peu et le temps de calcul est court. Le temps de calcul restera assez court pour N inférieur à la centaine. Après, cela devient un peu pénible.

Voici la première figure affichée (cf Figure 9) : à gauche le trajet au lancement du script. Le trajet entre les différentes villes est aléatoire. A droite, s'affiche le trajet optimisé entre les différentes villes.

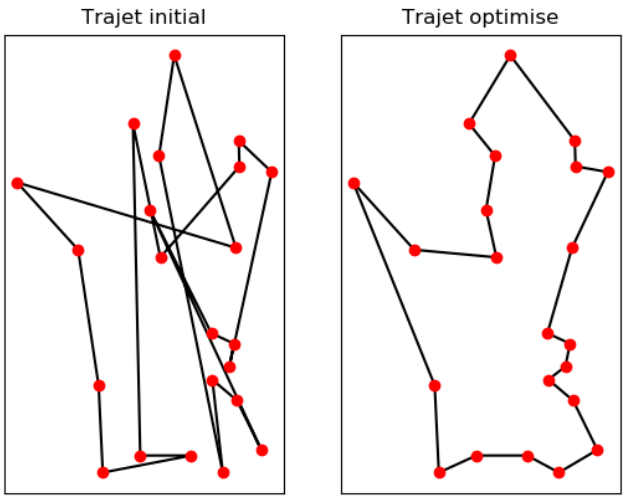

FIGURE 9 – Trajet au lancement et trajet optimisé

La figure suivante (cf Figure 10) montre l'évolution de l'énergie du système (à gauche) et de sa température (à droite) en fonction des itérations de l'algorithme de recuit simulé. Les axes des ordonnées sont gradués en semi-log, ce qui explique la droite d'évolution de la température.

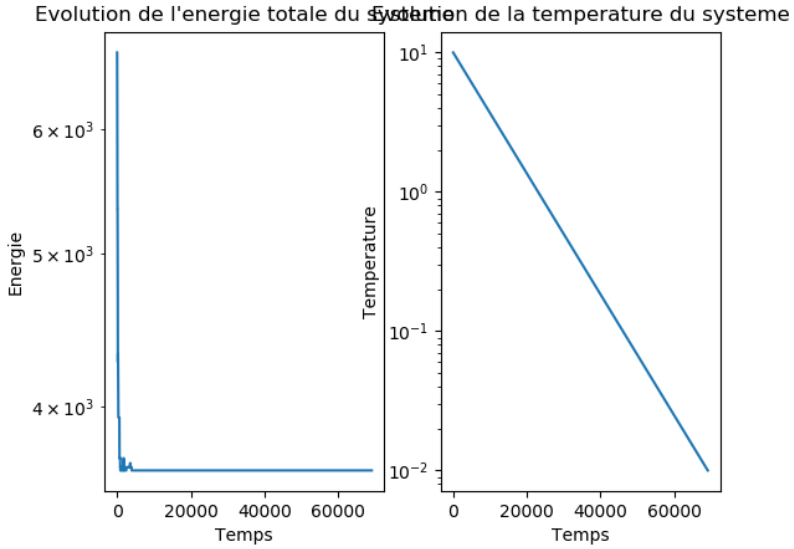

FIGURE 10 – Évolution de l'énergie et de la température du système

la distance du trajet minimal est 3641.5188688246476 qui correspond au trajet : `GRENOBLE, CHAMBERY, LYON, BESANCON, STRASBOURG, NANCY, METZ, LILLE, ROUEN, PARIS, ORLEANS, BOURGES, NANTES, BREST, BORDEAUX, PAU, TOULOUSE, MONTPELLIER, MARSEILLE, NICE, GAP, VALENCE`

7 Conclusion

Le problème du voyageur de commerce est toujours d'actualité dans la recherche en informatique, étant donné le nombre important de problèmes réels auxquels il correspond. Les problèmes dérivés et les extensions en sont très nombreux. Par exemple, des fenêtres de temps peuvent y être ajoutées. Ce concept consiste à imposer des contraintes de temps pour la traversée de chaque sommet. Autre exemple, il peut y avoir plusieurs voyageurs de commerce partant d'un même sommet, ou de sommets différents. Il suffit alors de considérer que les voyageurs de commerce sont des véhicules pour arriver à des problèmes de tournées de véhicules : étant donnée une flotte de véhicules, le problème consiste à déterminer les trajets de chacun pour livrer à moindre coût des clients en marchandise (chaque client est représenté par un sommet dans le graphe). Le nombre de véhicules peut être fixe ou non, les capacités des véhicules peuvent être les mêmes ou non, des fenêtres de temps peuvent être définies... Pour chacune de ces variantes, de nouvelles méthodes peuvent être explorées.