MARKOV CHAIN MONTE CARLO ET METROPOLIS-HASTINGS

Projet C++

ENSAE PARISTECH

24/01/2016

SJOHOLM Mathieu XU KAI

Introduction

Notre étude porte sur la Méthode de Monte-Carlo par chaînes de Markov (MCMC), et plus particulièrement sur l'algorithme de Metropolis-Hastings. L'objectif est de résoudre le problème du voyageur de commerce. Ce problème est le suivant: un voyageur doit visiter N villes pour vendre ses marchandises. Etant donné les coordonnées des différentes villes (et donc les distances entre chaque paire de villes), il veut trouver la plus petite distance totale à parcourir afin de minimiser ses frais. De plus, le voyageur visite chaque ville une seule fois, excepté la ville de départ dans laquelle il termine son séjour. Contrairement aux apparences, ce problème est plutôt complexe. En effet, à ce jour, on ne connait pas d'algorithme permettant de trouver une solution au problème dans un temps raisonnable. Une idée intuitive serait de répertorier tous les chemins possibles du voyageur, calculer les distances totales associées et trouver ainsi le minimum. Cependant, pour N villes différentes, il y a $N!$ chemins possibles, le point de départ restant inchangé. Dans ce cas, cela nous donne une complexité en temps de $O\left((N-1)\,!\right)$.

Plusieurs méthodes ont été élaborées pour trouver des solutions au problème du voyageur avec une complexité minimale. Une des méthodes est l'algorithme de Metropolis-Hastings, implémenté selon la Méthode de Monte-Carlo par chaînes de Markov.

I- Autour des méthodes de Monte-Carlo par chaînes de Markov

Le principe des méthodes de Monte-Carlo par chaînes de Markov est de simuler une chaîne de Markov dont la loi invariante est la distribution de probabilité à étudier. Les méthodes se basent sur le théorème de convergence d'une chaîne de Markov suivant: si la matrice de transition P est réductible, invariante, apériodique et Harris-récurrente, alors les puissances de P convergent vers la loi invariante de la chaîne de Markov.

1. L'algorithme de Metropolis-Hastings

L'algorithme de Metropolis-Hasting est une méthode de Monte-Carlo par chaînes de Markov qui utilise des marches aléatoires sur les chaînes de Markov. Une des particularités de cet algorithme est qu'il n'est pas nécessaire de calculer la distribution stationnaire π, mais qu'il suffit de la connaitre à une constante multiplicative près. On souhaite obtenir des tirages aléatoires d'un vecteur x avec une densité de probabilité π. On choisit alors une densité conditionnelle $P(x,y) = P(x/y)$. Chaque itération de l'algorithme se présente de la manière suivante :

- On tire $y_{t+1} \sim P(x_t, .)$
- On calcule $\alpha = \min(\frac{\pi(y_{t+1})P(y_{t+1},x_t)}{\pi(x_t)P(x_t,y_{t+1})}, 1)$
- $x_{t+1} = \begin{cases} y_{t+1} \ avec \ une \ probabilité \ \alpha \\ x_t \ avec \ une \ probabilité \ 1-\alpha \end{cases}$

On retrouve donc une chaîne de Markov, car chaque état dépend uniquement de l'état précédent. Pour un grand nombre d'itérations, les x_k suivent la distribution π.

2. Recuit simulé

Le recuit simulé est une technique s'inspirant de l'algorithme de Metropolis-Hastings qui imite le comportement d'un système thermodynamique. Elle permet de trouver une solution minimale à plusieurs types de problèmes. Le principe du recuit simulé est le suivant : on part d'une solution quelconque comprenant un état d'énergie initial E_0et une température de départ T_0 élevée. Le facteur de température représente le degré de liberté de la solution courante. Ensuite, on apporte à la solution une série de modifications. Les améliorations de la solution (ΔE<0) sont toujours acceptées, tandis que les détériorations sont acceptées avec une probabilité dépendante de la température, appelée probabilité de Boltzmann. La diminution progressive de la température va ainsi permettre à l'algorithme de converger vers un optimum qui se rapproche fortement de la meilleure solution possible en un temps raisonnable de calcul.

L'avantage de cette méthode est qu'elle permet d'éviter de nombreux minima locaux en n'omettant pas la possibilité d'accepter des détériorations du système en cours d'algorithme (surtout au début). Ainsi, elle permet d'explorer une partie plus large de l'espace des solutions. Cependant, cela reste une méthode arbitraire dans le choix des nombreux paramètres tels que la température initiale ou les critères d'arrêts. De plus, la sélection de ces paramètres est très importante.

3. Application au problème du voyageur de commerce

Le voyageur doit traverser N villes numérotées de 1 à N. On note d(a,b) la distance entre les villes a et b, et σ une permutation circulaire de {1,..,N}. Le voyageur cherche à minimiser la distance totale parcourue E en fonction de l'ordre spécifié par σ.

$$E(\sigma) = \sum_{k=0}^{N-1} d(\sigma^{k}(1), \sigma^{k+1}(1))$$

Le problème consiste donc à trouver une permutation σ qui minimise $E(\sigma)$.

Pour deux états E_1 et E_2 , la probabilité de Boltzmann est $e^{-\Delta E/T}$ avec $\Delta E = E_2 - E_1$.

L'algorithme adapté à la technique du recuit simulé se présente comme suit. A chaque étape, on tire une permutation σ' en permutant aléatoirement deux villes de l'itinéraire. On compare E(σ') et E(σ). Si E(σ')< E(σ), on accepte alors le nouveau parcours σ'. Sinon, ce dernier est accepté avec une probabilité $e^{-\Delta E/T}$. Autrement dit, la chance pour qu'un « plus mauvais » parcours soit retenu est d'autant plus grande que la différence de distance ΔE est faible et que la température est élevée. C'est pour cela qu'à la fin de chaque étape de l'algorithme, on diminue la température d'un taux de 5% (choisit arbitrairement).

II- Autour de la programmation orientée objet

1. Description de l'architecture du programme

a) Matrice de distance et coordonnée.

Nom
coorm

Méthodes
`coorm()`
`~coorm(void)`
`double dist(double,double,double, double)`
`void distm(int, double[maxn][2])`

Attributs (public) `double D[maxn][maxn]`

Nom
data

Méthodes
`data()`
`~data(void)`
`void CountLines(char *)`
`void getc(void)`

Attributs (public)
`int cityn`
`double C[maxn][2]`
`int LINES`

La classe data gère l'introduction des données (matrice de coordonnées) et calcule le nombre de villes, tandis que la classe coorm calcule la matrice de distance entre les différentes villes.

b) L'évolution de l'itinéraire

La classe path sert à initialiser et à stocker l'itinéraire que le programme parcourt. De plus, elle génère le nouvel itinéraire et calcule la distance totale parcourue.

Nom
path

Méthodes
`path();`
`~path(void);`
`void getvalue();`
`void totaldis(double[maxn][maxn], int);`
`path getnext(path p, double D[maxn][maxn], int n);`

Attributs (public)
`int city[maxn]` `double tdis`

c) Le recuit simulé

Nom sa
Méthodes `static path sima(path p, double D[maxn][maxn], int n)`

La classe sa introduit le processus du recuit simulé.

Pour les itérations du recuit simulé, on utilise une fonction statique.

2. Problèmes rencontrés et les solutions choisies

- *Comment modéliser le problème de voyageur de commerce ?*

On simplifie le problème en supposant qu'il y a seulement deux dimensions. Ainsi, on a seulement besoin d'importer les coordonnées des villes. Le programme travaille alors sur l'optimisation de cette question simplifiée.

- *Comment introduire les données et créer les objets utilisés dans le recuit simulé ?*

Pour importer les données à l'ordinateur, on choisit d'utiliser le dossier de format txt avec deux colonnes, qui est le standard output ou input des logiciels comme R ou SAS.

- *Comment décider la fin des itérations ?*

On suppose que le résultat optimal est atteint lorsque soit la température finale est atteinte soit le nombre de rejection de la nouvelle trajectoire dépasse un certain

nombre limite. Les choix de la température finale et du nombre limite sont déterminés arbitrairement.

3. Guide d'utilisation du programme

1 : On suppose que les données (les coordonnées des villes) sont stockées dans un dossier de format .txt . Après avoir compilé le programme, saisissez le nom du dossier, comme 'E:\data\example.txt', si le data est stocké dans le disque E, classeur data et dossier example .txt .

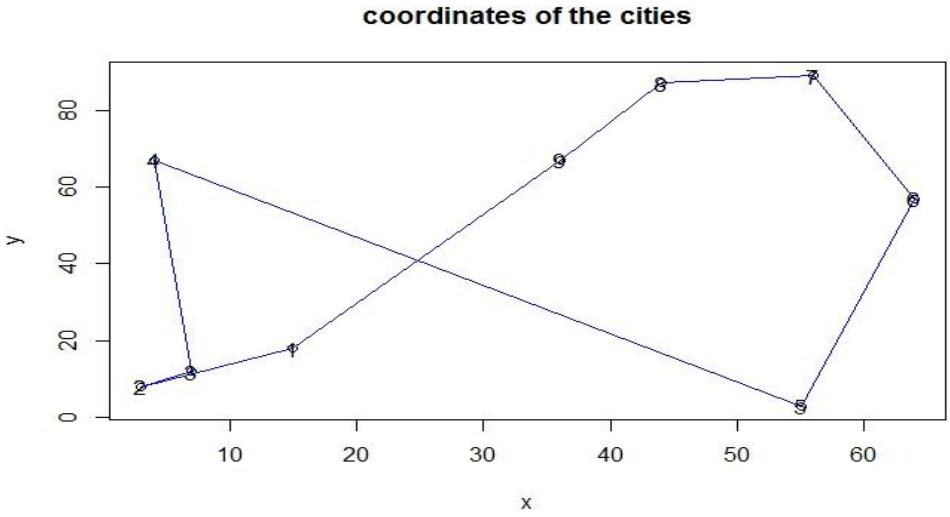
2 : Ensuite, le programme imprime toutes les routes obtenues par la circulation .Quand il s'arrête, il imprime le résultat optimal.

4. Résultats du programme.

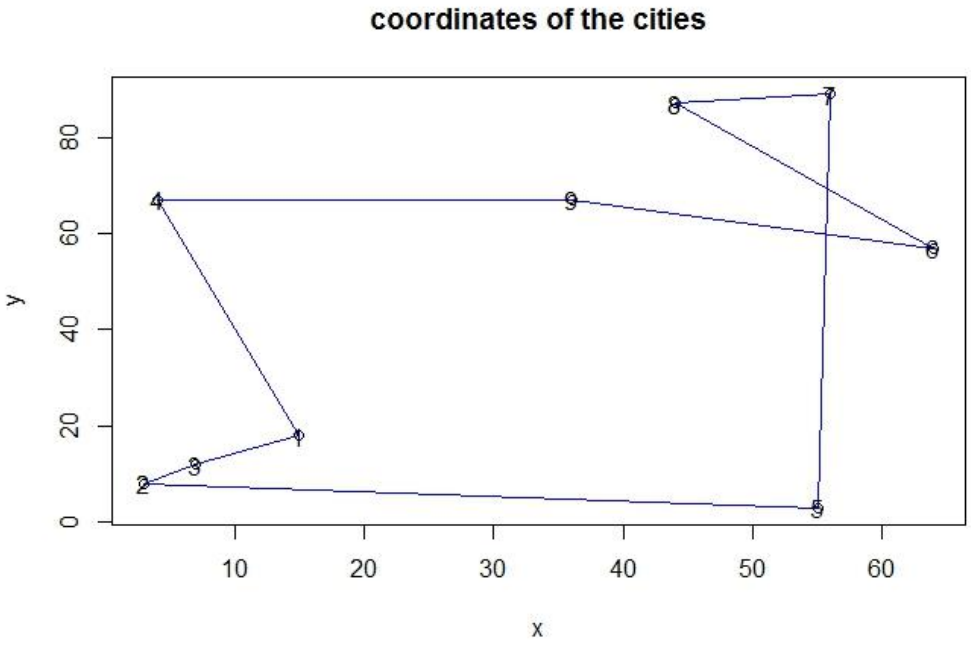
Pour la matrice de coordonnée ci-dessous,

1 :(15 ,18)	2 :(3,8)	3 :(7,12)	4 :(4,67)
5 :(55,3)	6 :(56,89)	7 :(44,87)	8 :(36,67)

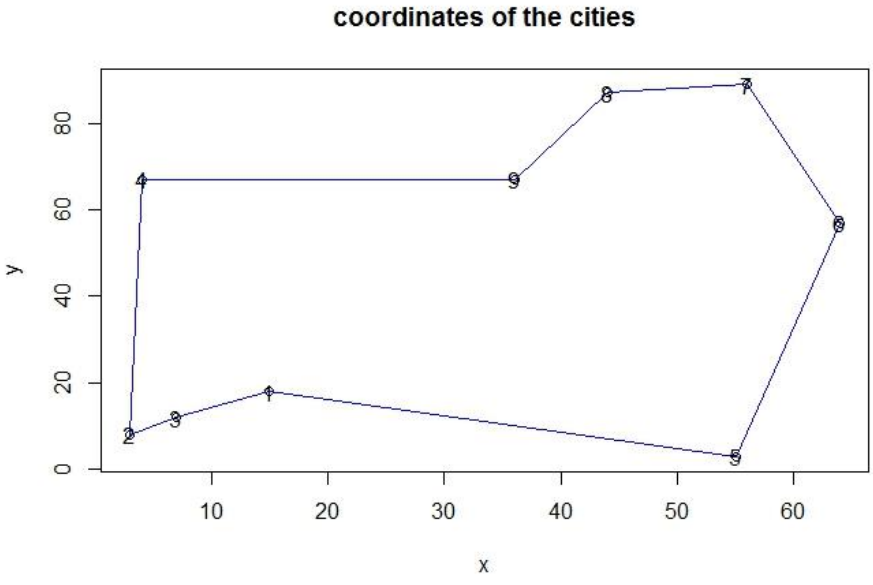
État original :



État intermédiaire:



État optimal:



Conclusion

Finalement, ce projet nous a permis d'approfondir des méthodes de Monte Carlo abordées cette année dans le cours de statistique, et plus particulièrement l'algorithme de Metropolis-Hastings. Cet algorithme nous a permis de résoudre un problème d'optimisation. Nous avons pu approfondir la programmation orientée objet, avec ses particularités telles que les classes ou les pointeurs. Nous nous sommes servis de la méthode statique qui nous a permis d'utiliser la fonction **sima** sans instancier un objet. Enfin, à l'aide de la fonction de la bibliothèque standard de fstream, nous avons introduit les données dans la classe data sans coût de saisie.