Projet Optimisation
le Récuit Simulé

AARAB Abdelkbir

Table de Matière :

1. L'algorithme de recuit simulé : historique :

Le recuit physique :

– Ce processus est utilisé en métallurgie pour améliorer la qualité d'un solide. En cherchant à atteindre un état d'énergie minimale qui correspond à une structure stable du métal.

– En partant d'une haute température à laquelle la matière est devenue liquide, la phase de refroidissement conduit la matière à retrouver sa forme solide par une diminution progressive de la température.

Le recuit simulé (Simulated Annealing) :

– Expériences réalisées par Metropolis et al. dans les années 50 pour simuler l'évolution de ce processus de recuit physique (Metropolis53).

– L'utilisation pour la résolution des problèmes d'optimisation combinatoire est beaucoup plus récente et date des années 80 (Kirkpatrick83, Cerny85).

– Le recuit simulé est la première métaheuristique qui a été proposée.

Principes généraux :

• L'idée est d'effectuer un mouvement selon une distribution de probabilité qui dépend de la qualité des différents voisins :

- Les meilleurs voisins ont une probabilité plus élevée ;
- Les moins bons ont une probabilité plus faible.

• On utilise un paramètre, appelé la température (notée T) :

1. T élevée : tous les voisins ont à peu près la même probabilité d'être acceptés.
2. T faible : un mouvement qui dégrade la fonction de coût a une faible probabilité d'être choisi.
3. T=0 : aucune dégradation de la fonction de coût n'est acceptée.

• La température varie au cours de la recherche : T est élevée au début, puis diminue et finit par tendre vers 0.

Schéma d'algorithme de recuit simulé

- Engendrer une configuration initiale S0 de S ; S := S0
- Initialiser T en fonction du schéma de refroidissement
- Répéter
  - Engendrer un voisin aléatoire S'de S
  - Calculer
  - $\Delta = f(S') - f(S)$
  - Si CritMetropolis (Δ, T), alors S : = S'
  - Mettre T à jour en fonction du schéma de refroidissement
- Jusqu'à <condition fin>
- Retourner la meilleure configuration trouvée

Le critère de Metropolis

- fonction CritMetropolis (Δ, T)
  - Si $\Delta \leq 0$ renvoyer VRAI
  - Sinon
    - avec une probabilité de exp (- Δ / T) renvoyer VRAI
    - Sinon renvoyer FAUX

- Un voisin qui améliore (Δ <0) ou à coût égal (Δ =0) est toujours accepté.

- Une dégradation faible est acceptée avec une probabilité plus grande qu'une dégradation plus importante.

- La fonction CritMetropolis (Δ, T) est une fonction stochastique : appelée deux fois avec les mêmes arguments, elle peut renvoyer tantôt « vrai » et tantôt « faux ».

Paramètre de température

• Dans la fonction CritMetropolis (Δ, T), le paramètre T (température) est un réel positif.

• La température permet de contrôler l'acceptation des dégradations :

– Si T est grand, les dégradations sont acceptées avec une probabilité plus grande.

– A la limite, quand T tend vers l'infini, tout voisin est systématiquement accepté.

– Inversement, pour T=0, une dégradation n'est jamais acceptée.

• La fonction qui spécifie l'évolution de la température est appelé le schéma de refroidissement.

Schéma de refroidissement

• La fonction qui spécifie l'évolution de la température est appelé le schéma de refroidissement (cooling Schedule).

• Dans le recuit simulé standard la température décroît par paliers.

• Par exemple, on pourrait avoir trois paramètres : la température initiale, la longueur d'un palier (nombre d'itérations avant de changer la température) et le coefficient de décroissance (si décroissance géométrique).

• On peut aussi utiliser d'autres schémas de refroidissement :

– On peut faire décroître la température à chaque itération.

– On utilise parfois une température constante (algorithme de Metropolis).

– On peut utiliser des schémas plus complexes, dans lesquels la température peut parfois remonter.

Algorithme de recuit simulé (avec paliers et refroidissement géométrique) :

- Engendrer une configuration initiale S0 ; S : = S0
- T : = T0
- Répéter
  - nb_moves := 0
  - Pour i : = 1 à iter_palier
    - Engendrer un voisin S' de S
    - Calculer Δ = f(S') – f(S)
    - Si CritMetropolis (Δ, T), alors
      - S : = S'; nb_moves := nb_moves + 1
  - acceptance_rate : = i / nb_moves
  - T : = T * coeff
- Jusqu'à <condition fin>
- Retourner la meilleure configuration trouvée

Réglage des paramètres du recuit simulé :

- Dans le cas du recuit simulé classique (avec refroidissement), le réglage des paramètres n'est pas évident.

- Pour régler les paramètres, il peut être utile d'observer le pourcentage d'acceptation (acceptance rate) au cours de la recherche = nombre de mouvements réellement exécutés / nombre d'itérations.

- **Température initiale :**
Si la température initiale est trop élevée, le début de la recherche ne sert à rien.

- **Critère d'arrêt :**
Il est inutile de poursuivre si :
  - le pourcentage d'acceptation devient très faible,
  - la fonction d'évaluation cesse d'évoluer

Réglage des paramètres du recuit simulé : implantation de Johnson et al.

• Température initiale
– Un paramètre fixe le pourcentage d'acceptation initial.

• Longueur d'un palier :

– Un paramètre borne le nombre d'essais (itérations) par palier
– Un autre paramètre borne le nombre changements (mouvements) pour le début de la recherche

• Critère d'arrêt :

– Un compteur sert à déterminer si l'algorithme stagne. Le compteur est fixé à zéro au début La recherche s'arrête quand le compteur atteint un certain seuil.

– A la fin d'un palier, **le compteur** est :

- incrémenté si le pourcentage d'acceptation est inférieur à un seuil.
- remis à zéro si la qualité de la meilleure solution a évolué au cours du palier.

Implémentation du recuit simulé :

• Quand on utilise le recuit simulé avec des paliers et que la fonction de coût prend des valeurs entières, on peut utiliser un tableau qui mémorise la probabilité d'acceptation associée à chaque valeur de Δ.

• L'implémentation naturelle du recuit simulé consiste à engendrer un mouvement, puis à appliquer le critère de Metropolis. Une implémentation équivalente (en termes de configurations engendrées) est le recuit sans rejet (Rejectless Annealing).

Il consiste à calculer la probabilité de chaque mouvement, puis à appliquer la roulette biaisée.

Origine de la Méthode :

C'est une amélioration de l 'algorithme hill-climbing pour **minimiser le risque d'être piégé dans des maxima/minima locaux**

♦ au lieu de regarder le meilleur voisin immédiat du nœud courant, avec une Certaine probabilité on va regarder un moins bon voisin immédiat » on espère ainsi s'échapper des optima locaux.

♦ au début de la recherche, la probabilité de prendre un moins bon voisin est plus élevée et diminue graduellement

- Le nombre d'itérations et la diminution des probabilités sont définis a l'aide d'un schéma (Schedule) de « températures », en ordre décroissant

♦ ex. : schéma de 100 itérations [2-°, 2-1, 2-2, ... , 2-99]
♦ la meilleure définition du schéma va varier d'un problème à l' autre.

Algorithme simulated annealing

Algorithme SIMULATED-ANNEALING(*noeudInitial, schema*) // *cette variante maximise*

1. déclarer deux nœuds : *n, n'*
2. déclarer : *t, T, ΔE,*
3. *n = noeudInitial*
4. pour *t* = 1 ... taille(*schema*)
5. *T = schema*[*t*]
6. *n'* = successeur de *n* choisi au hasard
7. $\Delta E = F(n') - F(n)$ // *si on minimisait, ΔE = F(n) – F(n')*
8. si $\Delta E > 0$ alors assigner *n = n'* // *amélioration p/r à n*
9. sinon assigner *n = n'* seulement avec probabilité de $e^{\Delta E / T}$

6. retourner *n*

plus *T* est petit,
plus $e^{\Delta E / T}$ est petite

D'autres Améliorations : *tabu search*

• L'algorithme Simulated annealing minimise le risque d'être piégé dans des optima locaux
♦ par contre, il n'élimine pas la possibilité d'osciller indéfiniment en revenant à un nœud antérieurement visite

- **Idée** : on pourrait enregistrer **les nœuds visités**

♦ on revient à A* et approches similaires !
♦ mais c'est impraticable si L'espace d'états est trop grand

• L'algorithme tabu search (recherche taboue) enregistre seulement les k derniers nœuds visités

♦ L'ensemble tabou est ('ensemble contenant les k nœuds
♦ Le paramètre k est choisi empiriquement
♦ Cela n'élimine pas les oscillations, mais les réduit
♦ il existe en fait plusieurs autres façons de construire ('ensemble taboue...

D'autres Améliorations : *beam* search

• **L'idée** : plutôt que maintenir un seul nœud solution n, en pourrait maintenir un ensemble de k nœuds différents.

1. on commence avec un ensemble de k nœuds choisis aléatoirement
2. à chaque itération, tous les successeurs des k nœuds sont générés
3. on choisit les k meilleurs parmi ces nœuds et on recommence

- Cet algorithme est appelé **local beam search** (exploration locale par faisceau)
  ♦ à ne pas confondre avec tabu search

- Variante stochastique beam search

♦ plutôt que prendre les k meilleurs, on assigne une probabilité de choisir chaque nœud, même s'il n'est pas parmi les k meilleurs (comme dans simulated annealing)

➢ Implémentation sous Python

```
import numpy as np

import matplotlib.pyplot as plt

def h(x):

if x < -1 or x > 1:

y = 0
```

- **Exemple explicatif :**

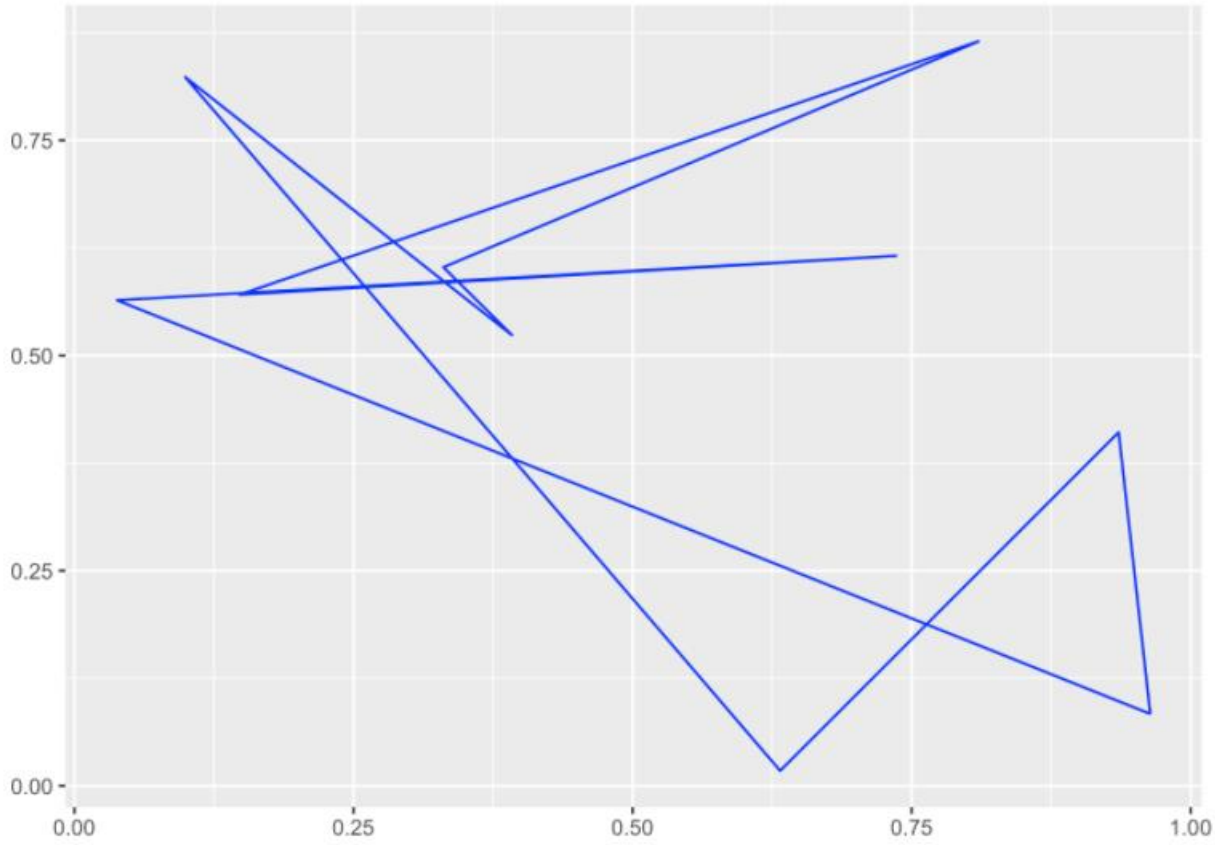
Le Voyageur de Commerce

Un commercial doit passer par N villes et revenir à son point de départ, il se déplace en avion et il y a des vols directs entre toutes les villes. Le but est de trouver le trajet optimal, c'est-à-dire la distance minimale à parcourir.

Il y a (N−1)! façons possibles d'effectuer le trajet. Il est donc clairement impossible d'effectuer une recherche exhaustive si N n'est pas petit.

Pour 10 villes, voici un exemple de représentation :

```
N <- 10
M <- matrix(runif(2*N),nrow=N,ncol=2)
trajini <- rbind(M,M[1,])
library(ggplot2)
ggplot(data.frame(trajini))+aes(x=trajini[,1],y=trajini[,2])+geom_path(col="blue"
)+xlab("")+ylab("")
```



Ce parcours n'est clairement pas optimal.

3. Construction de la matrice dist des distances entre villes :

```
distx <- (M[,1]%*%matrix(1,nrow=1,ncol=N)-matrix(1,nrow=N,ncol=1)%*%t(M[,1]))^2
disty <- (M[,2]%*%matrix(1,nrow=1,ncol=N)-matrix(1,nrow=N,ncol=1)%*%t(M[,2]))^2
dist <- sqrt(distx+disty)
```

4. On cherche donc

$$\operatorname{argmin}_{\sigma\in\mathcal{S}_N} D_\sigma = \operatorname{argmin}_{\sigma\in\mathcal{S}_N} \sum_{\ell=1}^{N} d(M_{\sigma(\ell)}, M_{\sigma(\ell+1)}).$$

Partant de σ, il suffit de tirer ℓ et ℓ' uniformément entre 1 et N, puis d'inverser $\sigma(\ell)$ et $\sigma(\ell')$ dans le vecteur définissant σ. Ceci définit sans ambiguïté la permutation $\sigma^{\ell,\ell'}$ (égale à σ si $\ell=\ell'$). Notons que cette transition est symétrique : la probabilité de passer de σ à $\sigma^{\ell,\ell'}$ est égale à celle de passer de $\sigma^{\ell,\ell'}$ à σ, à savoir $2/N^2$. Notons aussi qu'il y a bien d'autres façons d'explorer l'espace S_N, la littérature foisonne sur ce sujet.

Pour l'implémentation de l'algorithme, on commence par construire une fonction associant à une permutation σ la distance du parcours correspondant :

```
D <- function(perm,dist){
  d <- dist[perm[N],perm[1]]
  for (i in 1:(N-1)){d <- d+dist[perm[i],perm[i+1]]}
  return(d)}
```