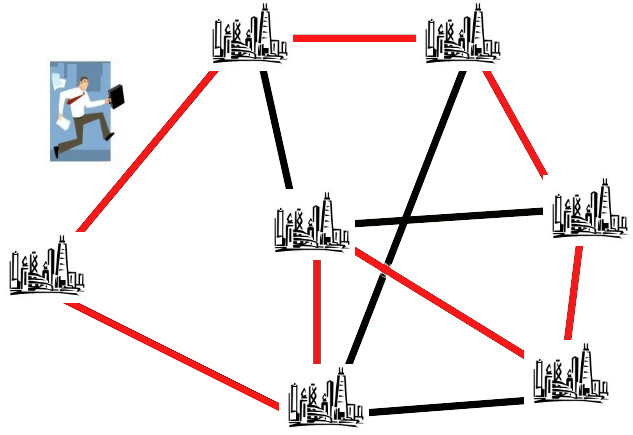
|  |
| --- |
| PROJET PS  PROBLÈME DU VOYAGEUR DE COMMERCE |



# **Introduction**

Un voyageur de commerce basé dans la ville v0 doit se rendre dans m villes v1, ..., vm. Il connaît les distances entre toutes les villes et voudrait déterminer l’ordre dans lequel il doit les visiter dans le but de minimiser la distance totale parcourue. Nous supposons dans un premier temps que les positions des m+1 villes sont tirées uniformément sur le carré[0,1]2, et de manière indépendante. La distance (euclidienne) entre les villes vi et v est notéeδ(vi,vj). Soitσ0=v0etσ= (σ1,...,σm)une permutation de l’ensemble {v1,...,vm} : σ représente un chemin ou une façon pour le voyageur de commerce de visiter une et une seule fois chacune des m villes dans un ordre donné.

Le problème du voyageur de commerce est de déterminer un chemin σ minimisant la distance totale parcourue :

(le voyageur de commerce doit retourner au point de départ !).

La recherche de la configuration optimale par énumération exhaustive de tous les chemins possibles est très vite impossible.

Ainsi, dans un premier temps nous allons illustrer ce propos en programmant cette recherche exhaustive, ensuite nous présenterons l’implémentation de notre algorithme d’optimisation stochastique sans le recuit simulé qu, enfin nous montrerons notre algorithme du recuit simulé permettant d’effectuer la recherche du plus court chemin de manière rapide et efficace, puis nous dévoilerons les résultats sur le jeu de données *20villes.txt* en dernière partie.

# 

# **Table des matières**

[**Introduction**](#_3h4tv4pnwa23) **1**

[**Table des matières**](#_fx1dwgwqclxd) **2**

[**1. Programmation de la recherche exhaustive**](#_h5qc1z69eg1o) **2**

[**2. Programmation de l’algorithme de Metropolis**](#_q4abstp9krky) **4**

[**3. Solution au du problème du voyageur de commerce via Metropolis**](#_6amls0jbzz24) **6**

[3.1. Implémentation de l’algorithme du recuit simulé pour le problème du voyageur de commerce](#_qguwrt5sgkwe) 6

[3.2. Proposition de critères d’arrêt](#_5iwf4m9t5upw) 8

[3.2.1. Retour du minimum](#_6m7j3wfm1uvq) 8

[3.2.2. Arrêt si il n’y pas de progression après n itération](#_h7wsdlysj7zd) 10

[3.2.3 fusion des algorithmes 3.2.1 et 3.2.2](#_iuhpahms6hk5) 12

[3.2.4 Algorithmes itératifs](#_mrxd3cvtr0s2) 14

[3.3. Test sur 8 villes](#_3bfio9d4kch4) 19

[3.4. Influence du schéma de température](#_p6sunqmzseih) 21

[**4. Application sur les 22 villes**](#_xuffvxkw3fgt) **22**

[**Conclusion**](#_jst3aoer47x8) **25**

# 1. Programmation de la recherche exhaustive

Pour un ensemble de villes, il existe au total chemins possibles. La ville de départ ne changeant pas la longueur du chemin, on peut choisir celui-ci de façon arbitraire, on a ainsi chemins différents. Enfin, chaque chemin pouvant être parcouru dans deux sens et les deux possibilités ayant la même longueur, on peut diviser ce nombre par deux. Par exemple, si on nomme les villes, , les chemins ,, ,,, et ont tous la même longueur, seul la ville de départ et le sens de parcours change. On a donc chemins candidats à considérer.

Ainsi pour 20 villes, le nombre de chemins candidats est .

Il en résulte que l’algorithme qui permet d’effectuer une recherche exhaustive possède une complexité en temps de soit en .

Par exemple, si le calcul d'un chemin prend une microseconde, alors le calcul de tous les chemins pour 10 points est de 181 440 microsecondes soit 0,18 seconde mais pour 15 points, cela représente déjà 43 589 145 600 microsecondes soit un peu plus de 12 heures et pour 20 points de 6 × 1016 microsecondes soit presque deux millénaires (1 927 années).

Nous pouvons donc dire que nous ne pouvons donc pas utiliser un tel algorithme pour trouver la solution. C’est la raison pour laquelle nous nous sommes penchés sur les algorithmes d’optimisation stochastiques.

# 

# **2.** Programmation de l’algorithme de Metropolis

Pour résoudre trouver une solution au problème de voyageur de commerce nous avons utilisées la chaîne de markov suivante :

* E l’espace d’état est l’ensemble des chemins possibles qui relie toute les villes. est élément de l’espace E. est un chemin de l’espace E voisin de . Deux chemins sont voisins si on peut passer de l’un à l’autre en permutant deux villes. Donc et communiquent.
* H() est la longueur total du chemin .

Voici la loi de probabilité Pt :

Pour définir Pt nous avons utilisé la démarche générale pour les mesures de Gibbs. Voici l’application de l’algorithme Métropolis qui permet de simuler (de manière approchée) Pt :

|  |
| --- |
| metropolis <- function(start\_value, matrice\_ville, iteration) {  # Nombre d'itération de metropolis  N = iteration    # vecteur des villes de départ  i = start\_value    for (n in 1:N) {  # tirage de j  j = i  ind\_ville\_permutation = sample(2:(length(i[1,])), 2)  tmp = i[ind\_ville\_permutation[1]]  j[ind\_ville\_permutation[1]] = j[ind\_ville\_permutation[2]]  j[ind\_ville\_permutation[2]] = tmp      # calcul de p  Hi = somme\_distance(data.distance, j)  Hj = somme\_distance(data.distance, i)  Pt = exp(-(1 / T)(Hi - Hj))  # changement de la valeur de i en fonction de p et du tirage de U.  if (p >= 1) {  i = j  }  else{  U = runif(1, 0, 1)  if (u <= p) {  i = j  }  }  }  return(i) } |

Ensuite, il s’agit d’implémenter le recuit simulé à notre algorithme afin d’améliorer son efficacité.

# 

# 3. Solution au du problème du voyageur de commerce via Metropolis

## 3.1. Implémentation de l’algorithme du recuit simulé pour le problème du voyageur de commerce

Voici notre implémentation de l’algorithme du recuit simulé :

|  |
| --- |
| # Métropolis version 2.0 avec l'implémentation du recuit simulé metropolis\_recuit <-  function(start\_value, matrice\_ville, iteration, h) {  # Nombre d'itération de metropolis  N = iteration    # vecteur des villes de départ  i = start\_value    for (n in 1:N) {  # tirage de j  j = i  ind\_ville\_permutation = sample(2:(length(i)), 2)  tmp = i[ind\_ville\_permutation[1]]  j[ind\_ville\_permutation[1]] = j[ind\_ville\_permutation[2]]  j[ind\_ville\_permutation[2]] = tmp      # calcul de p  T = h / log(n)  Hi = somme\_distance(data.distance, i)  Hj = somme\_distance(data.distance, j)  Pt = exp(-(1 / T) \* (Hj - Hi))    # changement de la valeur de i en fonction de Pt et du tirage de u.  if (Pt >= 1) {  i = j  }0  else{  u = runif(1, 0, 1)  if (u <= Pt) {  i = j  }  }  }  return(i)  } |

## 

## 3.2. Proposition de critères d’arrêt

Dans cette partie nous allons décrire les algorithmes que nous avons utilisés pour renforcer l’algorithme du recuit simulé.

### 3.2.1. Retour du minimum

Cette version va garder en mémoire le plus petit résultat obtenu lors des nombreuses récursions. Cette version ne contient pas de critère d’arrêt, mais sa fonctionnalité d’arrêt est utilisée dans d’autres algorithmes.

|  |
| --- |
| #Métropolis version 3.2.1 ( recuit simulé + stockage du minimun )  metropolis\_recuitMin <-  function(start\_value, matrice\_ville, iteration, h) {    # Nombre d'itération de metropolis  N = iteration    # vecteur des villes de départ  i = start\_value  Min = start\_value    for (n in 1:N) {  # tirage de j  j = i  ind\_ville\_permutation = sample(2:(length(i)), 2)  tmp = i[ind\_ville\_permutation[1]]  j[ind\_ville\_permutation[1]] = j[ind\_ville\_permutation[2]]  j[ind\_ville\_permutation[2]] = tmp      # recuit simulé  T = h / log(n)    # calcul de p  Hi = somme\_distance(matrice\_ville, i)  Hj = somme\_distance(matrice\_ville, j)  Pt = exp(-(1 / T) \* (Hj - Hi))    # Stockage du minimun  if (somme\_distance(matrice\_ville , Min) > Hj) {  Min = j  }    # changement de la valeur de i en fonction de p et du tirage de U.  if (Pt >= 1) {  i = j  }  else{  U = runif(1, 0, 1)  if (U <= Pt) {  i = j  }  }  }  return(Min)  } |

### 

### 3.2.2. Arrêt si il n’y pas de progression après n itération

Dans cette version l’algorithme va s’arrêter si, après n itérations, nous n’avons pas changé le parcours des villes. Nous avons fixé n arbitrairement via le calcul “ nombre de ville \* 2 “.

|  |
| --- |
| # Métropolis version 3.2.2 avec ( recuit simulé + stockage du minimun + arret ) # L'arrêt de metropolis a lieu en fonction de la variable compteur. Si ce dernier # est deux fois supérieur au nombre de ville à parcourir alors l'algorithme s'arrête. metropolis\_recuit\_arret <-  function(start\_value, matrice\_ville, iteration, h) {  # Nombre d'itération de metropolis  N = iteration  compteur = 0    # vecteur des villes de départ  i = start\_value  n = 1  for (n in 1:N) {  # tirage de j  j = i  ind\_ville\_permutation = sample(2:(length(i)), 2)  tmp = i[ind\_ville\_permutation[1]]  j[ind\_ville\_permutation[1]] = j[ind\_ville\_permutation[2]]  j[ind\_ville\_permutation[2]] = tmp      # calcul de p  T = h / log(n)  Hi = somme\_distance(matrice\_ville, i)  Hj = somme\_distance(matrice\_ville, j)  Pt = exp(-(1 / T) \* (Hj - Hi))    if (Hi > Hj) {  # remise à 0 du compteur  compteur = 0  } else{  compteur = compteur + 1    # si cela fait trop longtemps que nous n'avons pas trouvé un nouveau minimum, alors nous arretons metropolis  # si le compteur est supérieur au nombre de ville \* 2, alors nous arrêtons metropolis.  if (compteur > (length(start\_value) \* 2))  return(i)  }    # changement de la valeur de i en fonction de Pt et du tirage de U.  if (Pt >= 1) {  i = j    }  else{  U = runif(1, 0, 1)  if (U <= Pt) {  i = j    }    }  }    return(i)  } |

### 

### 3.2.3 fusion des algorithmes 3.2.1 et 3.2.2

Cette version reprend les fonctionnalités des algorithmes 3.2.1 et 3.2.2.

|  |
| --- |
| metropolis\_recuit\_arret\_min <-  function(start\_value, matrice\_ville, iteration, h) {  # Nombre d'itération de metropolis  N = iteration  compteur = 0    # vecteur des villes de départ  i = start\_value  Min = i  n = 1  for (n in 1:N) {  # tirage de j  j = i  ind\_ville\_permutation = sample(2:(length(i)), 2)  tmp = i[ind\_ville\_permutation[1]]  j[ind\_ville\_permutation[1]] = j[ind\_ville\_permutation[2]]  j[ind\_ville\_permutation[2]] = tmp      # calcul de p  T = h / log(n)  Hi = somme\_distance(matrice\_ville, i)  Hj = somme\_distance(matrice\_ville, j)  Pt = exp(-(1 / T) \* (Hj - Hi))    if (somme\_distance(matrice\_ville , Min) > Hj) {  Min = j    # remise à 0 du compteur  compteur = 0  } else{  compteur = compteur + 1    # si cela fait trop longtemps que nous n'avons pas trouvé un nouveau minimun, alors nous arretons metropolis  # si le compteur est supérieur au nombre de ville \* 2, alors nous arretons metropolis.  if (compteur > (length(start\_value) \* 2))  return(Min)  }    # changement de la valeur de i en fonction de Pt et du tirage de U.  if (Pt >= 1) {  i = j    }  else{  U = runif(1, 0, 1)  if (U <= Pt) {  i = j    }    }  }    return(Min)  } |

### 

### 3.2.4 Algorithmes itératifs

Dans cette sous-partie nous allons étudier les algorithmes itératifs. Ici nous itérons plusieurs fois sur les algorithmes vus avant avec des paramètres différents.

**Version 1 :** Nous réutilisons l'ordre des villes à parcourir renvoyé par metropolis\_recuit comme paramètre au prochain appel de metropolis\_recuit.

|  |
| --- |
| # v1 : nous réutilisons l'ordre des villes à parcourir renvoyé par metropolis\_recuit comme paramètre au prochain appel de metropolis\_recuit. it\_metropolis\_1 <-  function (init\_ville,  matrice\_ville,  nIt ,  nItMetro,  h) {  ville\_rec = init\_ville    for (i in 1:nIt) {  ville\_rec = metropolis\_recuit(ville\_rec, matrice\_ville, nItMetro , h)  }    return(ville\_rec)  } |

**Version 2 :** Nous réutilisons de l'ordre des villes qui minimise la distance à parcourir renvoyé par metropolis\_recuit.

|  |
| --- |
| it\_metropolis\_2 <-  function (init\_ville,  matrice\_ville,  nIt ,  nItMetro,  h) {  ville\_rec = init\_ville  vMin = init\_ville  distMin = somme\_distance(matrice\_ville, vMin)    for (i in 1:nIt) {  # réutilisons l'ordre des villes à parcourir renvoyé par metropolis\_recuit  ville\_rec = metropolis\_recuit(ville\_rec, matrice\_ville, nItMetro , h)    # Nous réutilisons seulement le résultat qui minimise la distance à parcourir  dist = somme\_distance(matrice\_ville, ville\_rec)  if (distMin > dist) {  vMin = ville\_rec  distMin = somme\_distance(matrice\_ville, vMin)    }  else {  ville\_rec = vMin  }    }    return(ville\_rec)  } |

**Version 3 :** Nousréutilisons de l'ordre des villes qui minimise la distance à parcourir renvoyé par metropolis\_recuit.

|  |
| --- |
| it\_metropolis\_3 <-  function (init\_ville,  matrice\_ville,  nIt ,  nItMetro,  h) {  nbVille = length(init\_ville)  ville\_rec = init\_ville    for (i in 1:nIt) {  ville\_rec = metropolis\_recuitMin(ville\_rec, matrice\_ville, nItMetro , h)    }    return(ville\_rec)  } |

**Version 4 :** Nous réutilisons de l'ordre des villes à parcourir renvoyé par metropolis\_recuit\_arret.

|  |
| --- |
| it\_metropolis\_4 <-  function (init\_ville,  matrice\_ville,  nIt ,  nItMetro,  h) {    ville\_rec = init\_ville  vMin = init\_ville  distMin = somme\_distance(matrice\_ville, vMin)    for (i in 1:nIt) {  ville\_rec = metropolis\_recuit\_arret(ville\_rec, matrice\_ville, nItMetro , h)    dist = somme\_distance(matrice\_ville, ville\_rec)    if (distMin > dist) {  vMin = ville\_rec  distMin = somme\_distance(matrice\_ville, vMin)    } else {  ville\_rec = vMin  }  }  return(vMin)  } |

**Version 5 :** Nous réutilisons de l'ordre des villes qui minimise la distance à parcourir renvoyé par metropolis\_recuit\_arret\_min.

|  |
| --- |
| it\_metropolis\_5 <-  function (init\_ville,  matrice\_ville,  nIt ,  nItMetro,  h) {  ville\_rec = init\_ville    for (i in 1:nIt) {  ville\_rec = metropolis\_recuit\_arret\_min(ville\_rec, matrice\_ville, nItMetro , h)    dist = somme\_distance(matrice\_ville, ville\_rec)    }    return(ville\_rec)  } |

## 

## 3.3. Test sur 8 villes

Nous avons testé nos algorithmes sur ces 8 villes tirés aléatoirement 10 fois. Voici un tableau qui résume l'efficacité de nos algorithme sur 8 villes :

|  |  |  |
| --- | --- | --- |
| **Algorithme** | **Temps moyen en secondes** | **Distances moyenne** |
| metropolis\_recuit(  itération = 600 ,  h = 60 ) | 0.14 | 2454663 |
| metropolis\_recuitMin(  itération = 600 ,  h = 60 ) | 0.21 | 2399869 |
| metropolis\_recuit\_arret(  itération = 600 ,  h = 60 ) | 0.01 | 2580741 |
| it\_metropolis\_1(  itérationMetro= 50 ,  itération = 600 ,  h = 60 ) | 7.12 | 2397649 |
| it\_metropolis\_2(  itérationMetro= 50 ,  itération = 600 ,  h = 60 ) | 7.09 | 2301036 |
| it\_metropolis\_3(  itérationMetro= 50 ,  itération = 600 ,  h = 60 ) | 10.55 | 2301036 |
| it\_metropolis\_4(  itérationMetro= 100 ,  itération = 600 ,  h = 60 ) | 0.71 | 2301036 |
| it\_metropolis\_5(  itérationMetro= 100 ,  itération = 600 ,  h = 60 ) | 0.62 | 2301036 |
| voyageur\_de\_commerce\_exhaustif | 23.49 | 2301036 |

Le nombre d'itérations a été choisi arbitrairement. Pour la variable h, la température, nous la choisissons en fonction du nombre de villes ( 10 \* nombres de villes).

Tout d’abord concernant l'efficacité des algorithmes, on remarque que la moyenne des résultats optimaux est de 2301036. En effet, c’est la valeur trouvée par notre algorithme voyageur\_de\_commerce\_exhaustif. Les algorithme itératif on quasi tous réussi à trouver les résultats optimaux. Les algorithmes simple : metropolis\_recuit\_arret, metropolis\_recuit et metropolis\_recuitMin n’ont pas toujours trouvé les résultats optimaux.

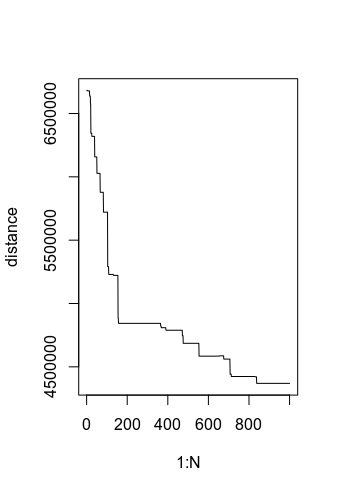
Ensuite pour le temps on remarque que les algorithmes itératifs avec arrêt sont très rapides. Ici it\_metropolis\_5 est 50 fois plus rapide que voyageur\_de\_commerce\_exhaustif et obtient un résultat optimal. Les autres algorithmes itératifs, sauf le premier, nous donne un résultat optimal mais avec un temps bien plus long. Concernant les autres algorithmes, ils sont les plus rapides.

Enfin, lorsque *m* est petit on peut se rendre compte que it\_metropolis\_5 est le meilleur. En effet, il donne le résultat optimal avec un temps record.

## 

## 3.4. Influence du schéma de température

Au départ avec une température haute, l’énergie diminue sans rester fixe, mais lorsque la température baisse on remarque que les sauts vers un minimum local sont moins nombreux et à la fin de l’algorithme il n’y a plus de saut. Il y a donc une convergence vers l’état d’équilibre thermique.



*Figure : Energie en fonctions des itérations*

# 4. Application sur les 22 villes

|  |  |  |
| --- | --- | --- |
| **Algorithme** | **Temps moyen en secondes** | **Distances moyenne** |
| metropolis\_recuit(  itération = 600 ,  h = 220 ) | 0.39 | 4980463 |
| metropolis\_recuitMin(  itération = 600 ,  h = 220 ) | 0.59 | 4721677 |
| metropolis\_recuit\_arret(  itération = 600 ,  h = 220 ) | 0.14 | 5535310 |
| it\_metropolis\_1(  itérationMetro= 50 ,  itération = 600 ,  h = 220 ) | 19.16 | 4510153 |
| it\_metropolis\_2(  itérationMetro= 50 ,  itération = 600 ,  h = 220 ) | 18.94 | 3887187 |
| it\_metropolis\_3(  itérationMetro= 50 ,  itération = 600 ,  h = 220 ) | 28.76 | 3904112 |
| it\_metropolis\_4(  itérationMetro= 100 ,  itération = 600 ,  h = 220 ) | 6.15 | 4137688 |
| it\_metropolis\_5(  itérationMetro= 100 ,  itération = 600 ,  h = 220 ) | 4.73 | 4409853 |

# 

Les algorithmes itératifs avec arrêt sont toujours aussi rapide mais bon que ceux sans arrêt. On remarque que it\_metropolis\_1 et les autres algorithmes nous donne des mauvais résultats. it\_metropolis\_2 semble être le meilleur dans cette situation.

Au cours de nos multiples tentatives de recherche de résultat nous avons abouti à une solution. En effet, lors de nos applications de notre algorithme nous ne sommes jamais passé en dessous des 3 640 152 mètres.

*Figure : Carte de la France avec le chemin le plus court trouvé pour parcourir les 22 villes proposées*

# Conclusion

En guise de conclusion, nous pouvons dire que nous avons pu observé qu’il est impossible de connaître la solution exacte du problème du voyageur de commerce car la complexité est telle qu’il n’est pas possible de trouver la solution avec un algorithme de recherche de chemin exhaustif. Par ailleurs nous avons travaillé sur des algorithmes d’optimisation stochastique, notamment sur l’algorithme qui permet d’utiliser la mesure du Gibbs et qui applique le recuit simulé.

De plus nous avons ajouté à cet algorithmes des propositions de critères d’arrêt ainsi que plusieurs méthodes permettant de faire des itérations de cet algorithme. C’est grâce à ces ajouts et à la recherche des bons paramètres que nous avons été en mesure d’obtenir notre meilleur résultat permettant de joindre les 22 villes proposés avec le chemin le plus court.