Métaheuristiques pour l'optimisation difficile
Problème de placement de composantes électronique

Andric Leo - Jassim El Akrouch - Hajar Sillahi

Introduction

Dans ce TP on abordera le problème de placement de composants électronique, qui est l'un des problèmes typique posé dans l'optimisation difficile. Ce problème est lié à la difficulté de placer les composants avec une distance minimale reliant deux composant.
Afin de pouvoir atteindre cette objectif, nous avons étudié l'algorithme du recuit simulé inspiré du phénomène du recuit en métallurgie.

Nous avons pour ce TP réaliser un algorithme dans le langage de programmation python pour sa simplicité et élégance en terme de lisibilité.

Configuration initiale

Afin de pouvoir configurer le système, nous avons dû créer 25 composants ayant chacun un numéro unique, une position (x,y) qui nous permettra par la suite de pouvoir connaître les voisins des composant et les longueurs générer par ces même composants.

Les composants sont ensuite stockés dans un tableau matricielles afin de pouvoir être manipulé et répondre à des opérations telles que la permutation.

La permutation des composants

Cette étape est crucial, car nous pouvons seulement permuter deux composants à la fois. Alors, nous avons programmé une fonction permettant de réaliser cette opération de manière à ce que nous stockons l'information de la position des composants.

```
def permutation(self,compo,mat):  # Méthodes permettant de permuter deux composants de la matrice
    tempx=self.x                  # On a créé une variable temporaire en x qui contiendra les position ordonné en x ([x][y])
    tempy=self.y                  # On a créé une variable temporaire en y qui contiendra les position ordonné en y ([x][y])
    self.x=compo.x                # On deplace maintenant les valeurs qu'on a stocker dans tempx et tempy dans compo.x
    self.y=compo.y                # On deplace maintenant les valeurs qu'on a stocker dans tempy et tempy dans compo.y
    compo.x=tempx                 # On deplace maintenant les valeur de tempx dans la compo.x
    compo.y=tempy                 # On deplace maintenant les valeur de tempy dans la compo.y
    mat[self.x][self.y]=self      # permet l'affiche des composant dans la matrice apres permutattions
    mat[compo.x][compo.y]=compo   # permet l'affiche des composant dans la matrice apres permutattions
    print
```

La distance manhattan

Cette distance est une règle à suivre afin de pouvoir obtenir la distance entre deux composant présent à deux endroit différent de la matrice.
Pour pouvoir trouver cette distance nous avons suivi cette formule

$$d(A,B) = |X_B - X_A| - |Y_B - Y_A|$$

```python
def distance(self, composant):    # Méthodes permettant de calculer la distance entre deux composants
    val = 0                       # Initialisation de la valeur = 0 (la distance finale)
    step = 5                      # distance entre deux voisins
    dxy = ((np.abs(self.x-composant.x))+(np.abs(self.y-composant.y))) # valeur absolue de la distance entre deux voisins
    val = step*dxy                # Valeur final de la distance en comprenant le distance entre voisins (le pas)
    #print("la distance entre les deux composants: ", val)
    return val
```

Configuration aléatoire des composants

Initialement, nous voulons que tous les composants soit dans le désordre afin de pouvoir appliquer l'algorithme du recuit simulé pour retrouver la configuration optimale ayant une longueur total de 200 avec tous les composants ordonné dans l'ordre.

La figure suivante nous montre les composants afficher dans le désordre.

```
longueur total du graph:
645.0

645.0
```

```python
# Affiche les composants dans le désordre
afficher(mat)
```

```
1       19      15      13      9
17      10      21      5       14
22      6       23      24      16
12      25      8       4       3
20      7       11      18      2
```

Longueur des connexions

On crée une méthode permettant de retourner la valeur de la longueur entre un composant et un autre composant quelconque. Ensuite, on créé une autre méthode permettant de pouvoir calculer la longueur total en parcourant la liste des connexion des composants.
La longueur totale aléatoire définit donc l'énergie du système.

Temperature Initiale

Cette procédure expérimentale se fait avant l'exécution du programme du recuit simulé. Elle nous permet de faire 100 transformation élémentaire faisant évoluer la variation d'énergie "dE" pour calculer la moyenne des valeurs absolue de dE => | dE |.
Ainsi, nous pouvons déduire la température initial tau0 (tau0 étant le taux initiale d'acceptation des perturbations dégradante choisi manuellement) de cette façon:

$$\text{Exp}(-<|\ \Delta E\ |>/\text{tau0}) = t0$$

Nous avons alors, choisi une valeur de 0.88 pour tau0.
Cette valeur influence le temps de calculs de l'algorithme.

Sachant que la configuration initiale aléatoire a une énergie élevée, on choisit donc une valeur de probabilité initiale d'acceptation généralement au delà de 0.8 qui permet d'augmenter l'efficacité de l'algorithme pour obtenir un résultat optimal.

Transformation élémentaire

Ici, nous avons permuter les composants aléatoirement dans la matrice et calculer la variation d'énergie pour tester sous différentes conditions ainsi arriver à trouver un équilibre thermodynamique.
Ainsi nous étudions deux cas, selon la règle d'acceptation de Metropolis:

- L'acception de la modification quand dE < 0
- L'acception de la modification avec une probabilité Exp(- ΔE/T)

```
# Initialisation de dE
dE = 0
tau0 = 0.88

#fixation de la temperature initial

for i in range(0,100):
    ei = longueurtotal(mat)
    #permutation des composant desordonné avec r1 et r2
    random1 = random.choice(tableau)
    random2 = random.choice(tableau)
    random1.permutation(random2, mat)
    ej = longueurtotal(mat) # nouvelle
    dE += ej - ei

dE = np.abs((dE)/100)
t0 = (-dE/np.log(tau0))
print("dE: ", dE, " tau0: ", t0)
```

```
longueur total du graph:
630.0
dE:  0.15  tau0:  1.173402517889478
```

Equilibre thermodynamique

L'équilibre thermodynamique est atteinte quand les compteur Na = 12*N et Ne = 100*N, où N=25 (nombre de variable d'optimisation possible).
Comme indiqué pendant le cour, au début de l'optimisation à température élevée, il y a beaucoup d'acceptation, alors on atteint 300 perturbations acceptées avant d'avoir effectuer 2500 perturbations.
A température basse, il y a peu d'acception, car on atteint 2500 perturbation effectués avant d'en avoir au 300.

Système figé

Une fois avoir trouvé l'équilibre thermodynamique, on test à 3 palliers de température successif avec 0 acceptions pour arrêter l'algorithme et enfin visualiser notre matrice et voir l'ordre de nos composants.

Si la configuration optimale n'est pas atteinte on reboucle en diminuant la valeur de la température initiale de cette façon:

tau0 = 0.9*tau0

Dans notre cas, on a voulu tester à 4 palliers différents, ce qui a rendu l'algorithme un peu plus long en temps de calcul.

Solution optimale

Une fois que l'algorithme s'arrête il nous affiche automatiquement le résultat des longueurs total dans le graph et aussi le tableau ordonné.
On en conclut que la configuration optimal du problème est atteinte.

```
25      20      15      10      5
24      19      14      9       4
23      18      13      8       3
22      17      12      7       2
21      16      11      6       1
longueur total du graph:
200.0
longueur totals des connections: 200.0
```

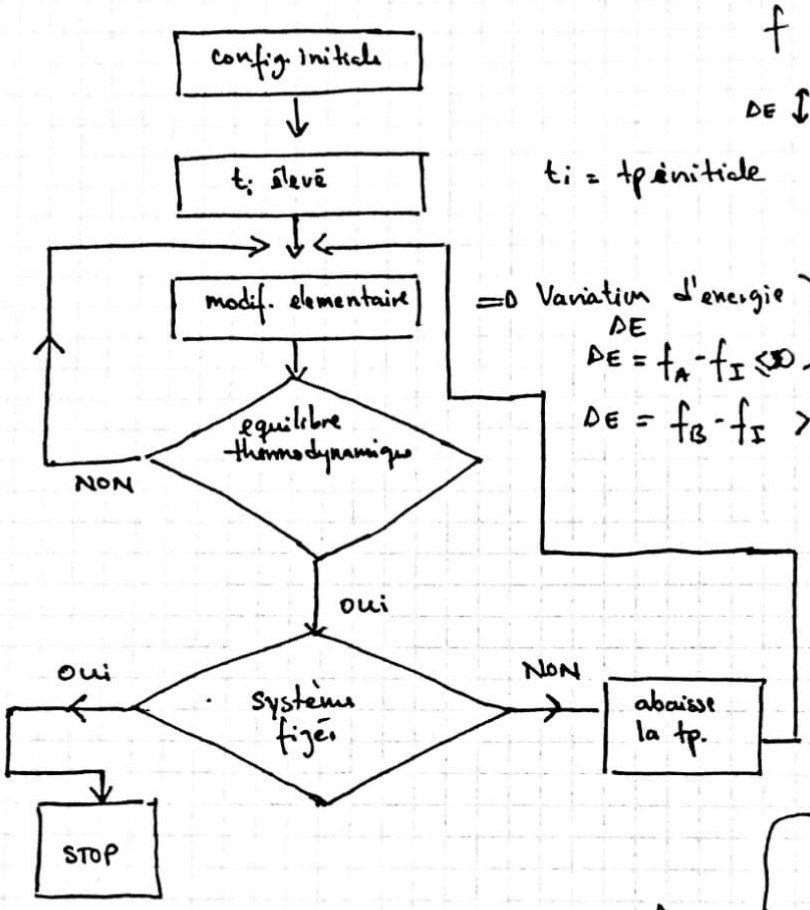
Conclusion

La méthode du recuit simulé est une méthode fiable, simple à mettre en oeuvre. Elle est aussi bien adapté au problème de ce TP, car on observe de bon résultat malgrés le changement de tau0. Cependant, le temps de calcul peut être très long, comme dans notre cas il nous a fallu être patient avant d'obtenir le résultat, mais cela peut être optimisé en trouvant des valeurs plus précise pour tau0.

Cependant, ce TP a notamment été bénéfique car nous avons pu mieu comprendre ce sujet de métaheuristique d'optimisation difficile dans le fond et la forme. Pouvoir l'appliquer sur un différent langage de programmation nous a permis d'avoir de nouvelles compétences en matière de programmation et algorithmique et aussi d'écrire ce programme de façon claire et concis.

Le langage de programmation utiliser à permit de pouvoir programmer cette algorithme facilement, et ainsi optimiser le probleme a chaque exécution du programme.

Algorithme du recuit simulé.



t_i = tp initiale

$\Rightarrow$ Variation d'énergie ΔE

$\Delta E = f_A - f_I \leq 0$ ① $\Delta E \leq 0$: modif. accepté

$\Delta E = f_B - f_I > 0$ ② $\Delta E > 0$ ↳ modif. accepté avec la proba. $e^{-\frac{\Delta E}{T}}$

ΔE donné

- $e^{-\frac{\Delta E}{T}} \simeq 1$ T élevé
- $e^{-\frac{\Delta E}{T}} \simeq 0$ T faible.
  - ↳ Comportement de la descente itérative

Tp donné.

- $e^{-\frac{\Delta E}{T}} \simeq 0$ ΔE élevé
- $e^{-\frac{\Delta E}{T}} \simeq 1$ ΔE Petit