Résolution du Problème du Voyageur de Commerce
– Métaheuristique –

André Biancheri Tchilinguirian

Table des matières

I Introduction **1**

II Résolution par colonies de fourmis **3**
1 Les fourmis . . . 3
2 Principe . . . 5
3 Algorithme . . . 6
4 Application sur un exemple . . . 6

III Autres approches métaheuristiques **7**
1 Amélioration 2-opt . . . 7
1.1 Principe . . . 7
1.2 Application sur un exemple . . . 7
1.3 Algorithme . . . 8
2 Algorithme Génétique . . . 8
2.1 Langage spécifique à la génétique . . . 8
2.2 Principe . . . 8
2.3 Algorithme . . . 9
2.4 Application sur un exemple . . . 9
3 Recuit simulé . . . 10
3.1 Un peu de thermo . . . 10
3.2 Principe . . . 10
3.3 Algorithme . . . 11
3.4 Application sur un exemple . . . 11

IV Défi des 250 villes **12**
1 Présentation . . . 12
2 Résultat de nos algorithmes . . . 12
2.1 Algorithme des fourmis . . . 12
2.2 Algorithme 2-opt . . . 12
2.3 Algorithme génétique . . . 12
2.4 Algorithme de recuit simulé . . . 12
3 Combinaison de nos algorithmes . . . 13

Références **14**

Annexe
Performance de la Machine et Validation du choix du matériel . . . i
Implémentation CaML - Algorithme des fourmis . . . iii
Implémentation CaML - Algorithme 2-opt . . . viii
Implémentation CaML - Algorithme Génétique . . . xi
Implémentation CaML - Algorithme du recuit simulé . . . xvii
Compléxité des algorithmes . . . xxi
Implémentation du calcul de compléxité . . . xxiv

I Introduction

Certains problèmes de rangement ou de prévision de trajet sont trop complexes pour être résolus de manière exhaustive. On peut se contenter alors de solutions quasi-optimales. Pour rechercher de telles solutions, on fait appel à des algorithmes méta-heuristiques inspirés de la Nature. En effet il arrive que la somme d'"intelligences" individuelles sous forme d'une "intelligence" collective soit plus performante que le plus puissant des super-ordinateurs.

Problème du Voyageur de commerce Un exemple connu de problème difficile est le problème du voyageur de commerce. Son nom vient de la situation fictive d'un représentant de commerce qui devrait faire une tournée en passant par un certain nombre de villes. Cependant à l'échelle d'un pays comme les États-Unis, le coût en argent et en temps d'un tel voyage peut devenir prohibitif si l'on ne prévoit pas un trajet optimal à l'avance.

En langage plus mathématique, le problème consiste à trouver un chemin ou un cycle hamiltonien de poids minimum étant donné un graphe où les sommets sont les villes et les arêtes les routes (terrestres ou aériennes) les joignant. Ce problème fait partie d'une famille de problèmes trop complexes pour qu'une solution soit recherchée de façon systématique parmi toutes les solutions possibles. Le nombre de chemins ou de cycles possibles dans notre problème est en N !, N le nombre de villes. Dans ces conditions, la puissance de calcul d'un superordinateur est dépassée dès 30 villes.

Nombre de ville	Nombre de possibilité	Temps d'exécution sur "K" [1]
5	120	$0.12 \cdot 10^{-12}$ sec
20	$\simeq 10^{18}$	40 min
30	$\simeq 10^{32}$	10^{10} ans

TABLE 1 – Illustration de la complexité du problème du voyageur de commerce

Mais le problème n'est pas seulement complexe à résoudre par une machine, contrairement à certain problème tel que le *jeu de go* où des enfants expérimentés arrivent à battre les ordinateurs actuels, ce problème est également complexe pour un humain, pour mettre cette difficulté en évidence on a demandé à plusieurs personnes de résoudre un problème du voyageurs de commerce à 20 villes et on remarque tout d'abords que les résultats donnés sont multiples, mais en plus que les résultats rendus par les algorithmes présentés dans la suite de ce dossier sont bien meilleurs

FIGURE 1 – Résolution d'un PVC à 20 villes par des pairs

1. Le super ordinateur "K" est l'un des plus puissant au monde ayant une capacité de calcul lui permettant d'exécuter des millions de milliard d'opération par seconde (ce qu'on appelle pétaflops)

Métaheuristique Pour résoudre ce problème, nous allons nous intéresser à une famille d'algorithmes particulière : les algorithmes méta-heuristiques. Ceux-ci se basent sur la réunion d'un grand nombre de phénomènes plus ou moins aléatoires pour trouver un minimum global à un problème d'optimisation. L'utilisation de procédés stochastiques sert à se sortir de minimum locaux vers lesquels pourraient converger des algorithmes plus classiques. Ces procédés viennent souvent de l'observation de phénomènes naturels comme le brassage génétique, l'organisation des atomes en fonction de la température ou même les colonies de fourmis.

Tous ces exemples sont particulièrement intéressants :

- les fourmis, dont l'intelligence est limitée individuellement, optimisent néanmoins leurs trajets grâce à leur système de communication ;
- la génétique, bien que croisant des gènes aléatoirement, conduit à une amélioration globale du génome ;
- la technique du recuit simulé, en alternant phase de réchauffement puis de refroidissement d'un métal, permet d'obtenir édifice atomique très solide.

L'application du premier exemple au problème du voyageur de commerce est immédiate mais les deux autres exemples conduisent eux aussi à sa résolution.

II Résolution par colonies de fourmis

1 Les fourmis

Les fourmis sont des insectes qui communiquent "chimiquement" à l'aide de phéromones. Ce moyen de communication leur permet, entre autre, de choisir un plus court chemin.

Expérience : Nous avons reproduit une expérience pour mettre en évidence cette particularité.

Protocole :

- ⇝ Isoler des fourmis
- ⇝ Attendre un certain temps qu'elles se calment et reprennent un comportement naturel
- ⇝ Construire des chemins reliant la sortie de la fourmilière à une source de nourriture.

  La construction de ces chemins a posé quelques soucis : En effet, à l'origine nous avions opté pour des chemins en carton mais ce matériaux n'est pas assez neutre et a donc posé des soucis (pour le dépôt de phéromone). Nous avons finalement choisi d'utiliser du verre ou du plexiglas car ce sont des terrains neutres.

  Ce problème met en évidence que les phéromones sont l'une des bases du choix du plus court chemin
- ⇝ Libérer les fourmis

FIGURE 2 – Expérience

Observation Les fourmis se sont d'abord dispersées aléatoirement et, petit à petit, seuls les plus courts chemins ont été sélectionnés jusqu'à ce qu'il n'en reste qu'un : parmis les plus optimaux.

FIGURE 3 – schéma de l'expérience

Propriété Une fourmi, lorsqu'elle se déplace, dépose des phéromones sur le chemin et la fourmi suivante prend le chemin sur lequel il y a le plus de phéromone.

C'est en suivant ce principe, illustré par l'expérience, que les fourmis choisissent le plus court chemin. En effet au départ, une fourmi choisit son chemin de façon aléatoire (du moins de façon seulement visuelle). Donc plus le chemin est court, plus le nombre de fourmis qui y passent, en un temps donné, est grand et par conséquent plus le chemin est court, plus la concentration en phéromones sur ce chemin sera importante et les chemins les plus longs sont peu à peu délaissés jusqu'à ce qu'au final le plus court chemin soit sélectionné.

2 Principe

Problème : On a un ensemble de villes, le but est de trouver un des plus courts chemins passant une seule fois par toutes ces villes et revenant à sont point de départ

Principe :
- Une fourmi part d'une ville v_0
- Elle choisit la suivante au hasard de la manière suivante :
  - → Plus la ville est loin et moins elle a des chances d'être choisie ;
  - → Plus le chemin menant à cette ville est couvert de phéromones de fourmis plus cette ville a des chances d'être choisie.
- Une fois la ville v_1 choisie on applique à nouveau le principe à partir de la ville v_1 vers l'ensemble des villes moins la ville v_0
- On réitère le processus jusquà ce qu'il ne reste aucune ville
- On retourne au point de départ
- Plus le chemin parcouru est long moins on y place de phéromones
- Une autre fourmi part de la ville v_0 jusqu'à ce que toutes les fourmis soient partis
- Les phéromones s'évaporent au cours du temps
- Tant que l'utilisateur le désire, on recommence l'algorithme

FIGURE 4 – Amélioration par phéromones

Choix de la ville de départ : On prend une ville au hasard dans la liste donné par l'utilisateur, ou bien celle qui se situe en tête de liste.

Initialisation des phéromones : On initialise les phéromones en créant une matrice de taille $N \times N$ rempli de 0. Dans la suite cette matrice sera symétrique tel que le coefficient (i, j) de la matrice représente le taux de phéromones qu'il y a sur le trajet reliant la ville représenté par l'entier i et celle représentée par l'entier j

Dépôt des phéromones : Quand une fourmi à effectué son trajet totale elle dépose un nombre ζ_i de phéromones. sur chacun des trajets t_i. Plus le trajet est long moins ζ est élevé. On pourra prendre $\zeta = \frac{1}{d}$ avec d la distance parcourue

Évaporation des phéromones : À chaque itération on effectuera une opération qui traduit l'évaporation des phéromones. Par exemple si le trajet à un nombre ζ_i de phéromones, elle en aura $f(\zeta_i)$ après évaporation avec f une fonction décroissante. On pourra prendre $f(x) = kx \;\; k \in]0; 1[$

Choix de la ville suivante : Ce choix se fait en fonction de deux facteurs : D et ζ avec D qui est la distance du trajet et ζ le taux de phéromone situé sur ce trajet. Plus D est grand moins la fourmi aura de chance de choisir ce trajet et plus ζ est grand plus la fourmi aura de chance de choisir ce trajet

On donnera plus d'importance aux phéromones qu'à la distance du trajet.

Si on a n trajet possible, nous représentons ces trajets par un entier i, le choix se fait de la façon suivante : on sépare l'ensemble $[0;1[$ en n sous-ensemble $P_i = [\alpha_i;\beta_i[$ tel que : $\begin{cases} \cap_{i=1}^{n} P_i & = & \emptyset \\ \cup_{i=1}^{n} P_i & = & [0;1[\end{cases}$.

La longueur $l_i$de chaque P_i sera évaluer en fonction de D_i, de ζ_i, de $D_{tot} = \sum_{i=1}^{n} D_i$ et de $\zeta_{tot} = \sum_{i=1}^{n} \zeta_i$. On a $l_i = m_d \frac{1}{D_i\left(\sum_{k=1}^{n} \frac{1}{D_k}\right)} + m_\zeta \frac{\zeta_i}{\zeta_{tot}}$ qui convient avec $m_d + m_\zeta = 1$ tel que $m_\zeta > m_d$ (ce qui garantie qu'on accorde plus d'importance aux phéromones), on remarquera que $\sum_{i=1}^{n} l_i = 1$.

Le choix de la ville s'effectue alors de la manière suivante : on tire un nombre aléatoire $t \in [0;1[$. Si $t \in P_j$ alors la fourmi choisi le j-ieme trajet

Dernière fourmi : A la fin de toute les itérations voulu par l'utilisateur, on lance une dernière fourmi qui parcourt les trajets en allant a chaque fois là ou il y a le plus de phéromones.

3 Algorithme

Données : n, L, j % n est le nombre de fourmis parcourant les villes ; L contient les villes à parcourir avec des informations sur les distances entre les villes ; j le nombre d'itération du programme
Résultat : L' : liste % contient la liste du parcours de villes à effectuer

```
début
    A ← Choisir-ville-départ L
    P ← initialisation-des-phéromones
    pour m allant de 1 à j faire
        liste ← L
        pour i allant de 1 à n faire
            liste ← Enlever-ville(A, liste);
            B ← A
            tant que L non vide faire
                B ← Choisir-ville-suivante(liste)
                liste ← Enlever-ville(B, liste);
            fin
            P ← Déposer-phéromones P Evaporation-phéromones
        fin
    fin
    Rendre L' = dernière-fourmi P
fin
```

Algorithme 1: Algorithme des fourmis

4 Application sur un exemple

Tout nos algorithme seront tester sur une même liste de ville (dans chacun des cas le résultat le plus performant sera retenu)

`Algo_fourmis50`fourmis `50`fois

Temps : 5.4 sec

Parcours : 1121

III Autres approches métaheuristiques

Par recherche sur internet, il en est resorti que, d'une part un algorithme s'inspirant du parcours des fourmis avait déjà été élaboré avant nous, ce qui n'est pas surprenant car nous avons fait ce choix du fait, justement, que les fourmis avait un mode de déplacement qui leur faisait choisir un chemin optimale

Parmis les algorithmes s'inspirant du voyageur de commerce, en voici une liste non exhaustive :
- Recherche local : 2-Opt
- Algorithme génétique
- Recuit simulé

1 Amélioration 2-opt

1.1 Principe

Le principe de l'algorithme « amélioration 2-opt » est assez simple et repose sur l'idée suivante :
- On génère un premier trajet
- On l'améliore jusqu'à ce qu'un critère d'arrêt ait été vérifié.

Génération du premier trajet : On peut générer le trajet de telle sorte qu'à la i-ème étape, le choix de la $(i+1)$-ème ville est la plus proche voisine de la i-ième ville parmi les villes à parcourir.

Amélioration du trajet : Une amélioration possible est de supprimer tout croisement.

En s'appuyant sur *l'inégalité du parallélogramme*, prenons le cas d'un trajet ou un chemin direct relie A à B, un chemin relie B à C et ensuite un chemin relie C à D (exemple illustré dans la figure 5). On voit que si $(\lg AB + \lg CD > \lg AC + \lg BD)$ [2] alors il y a présence de croisement et il apparaît plus efficace alors de passer de A à C, de relier C à B puis de passer de B à D.

FIGURE 5 – un exemple de croisement

Critère d'arrêt : Un critère d'arrêt possible est que la fonction d'amélioration définie précédemment ne modifie plus le trajet. Autrement dit dès qu'il n'y a plus de croisement dans le trajet.

1.2 Application sur un exemple

Temps : 0.02 sec

Parcours : 1082

2. où $\lg AB$ représente la distance de A à B

1.3 Algorithme

```
Données : L % contient les villes à parcourir avec des informations sur les distances entre les villes
Résultat : L' : liste % contient la liste du parcours de villes à effectuer
début
    L' ← Générer-trajet-initial S
    B1 ← true
    tant que B1 faire
        L'' ← amélioration L'
        si Trajet L'' = Trajet L' alors
            B1 ← false
        fin
        si Trajet L'' < Trajet L' alors
            L' ← L''
        fin
    fin
fin
```

Algorithme 2: Algorithme 2-Opt

Complexité en $O(l^2)$

2 Algorithme Génétique

2.1 Langage spécifique à la génétique

l'algorithme s'inspire grandement de la théorie de l'évolution et des principes génétiques donc nous utiliserons un langage spécifique à la génétique lors de cette étude

individu : un trajet possible

population : un ensemble de trajets

mutation : modification aléatoire dans le trajet d'un individu

adaptation : plus le trajet est court, plus il est adapté au problème

sélection naturelle : élimination des individus les moins adaptés

2.2 Principe

1. On génére aléatoirement une première population d'individu
2. On fait la sélectionne la moitié des individus (sélection naturelle)
3. On arrange les individus restant en couple et on crée pour chaque couple un couple d'enfant qui représente des trajets ayant des caratéristique du trajet de leurs deux parents
4. les enfants peuvent avoir une chance sur q d'avoir une mutation (paramètre q définie par l'utilisateur)
5. On retourne à l'étape 2.

Première génération : On génére aléatoirement n trajet possible. Le nombre n est fixé par l'utilisateur. Plus il sera grand plus l'algorithme sera efficace et long.

Sélection : On ne souhait garder que la moitié des individus, il existe deux méthodes pour faire cette sélection.

méthode élitiste : On ne conserve que la meilleure moitié

méthode par tournois : on regoupe les individus par couple et on les laisse se battre. Le moins adapté et supprimé et l'autre est conservé

On appliquera chacune des deux méthode une fois sur deux.

Passage à la génération suivante on regroupe les individu par couple et chaque couple passe par l'opérateur croisement

l'opérateur de croisement (c.f. figure 6) prend en entrée deux individu et donne en sortie ces deux mêmes individus ainsi que deux "enfants" qui reprennnent des caractéristiques de leurs deux "parents". L'endroit où il y a croisement est définie aléatoirement.

Durant leurs naissances (ou juste après, nous sommes en informatique, nous avons le droit) on applique a un individu sur q l'opérateur mutation qui modifie de façon aléatoire le trajet. Par exemple interchanger la position de deux points dans le trajet.

FIGURE 6 – un exemple de croisement

Critère d'arrêt : Cette algorithme ne termine *à priori* pas. Il faut donc demander à l'utilisateur de fixer un nombre m fini d'itération à faire

2.3 Algorithme

Données : n, i, q, L % n est le nombre d'individu par population, i le nombre d'itération voulu, q l'inverse du nombre de chance qu'un individu soit un mutant à la naissance, et L contient les villes à parcourir avec des informations sur les distances entre les villes
Résultat : L' : liste % contient la liste du parcours de villes à effectuer
début
  $P' \leftarrow$ **Générer-population-initial** S
  pour *z = 1 à i* **faire**
    si $z = 0 \mod 2$ **alors**
      $P \leftarrow$ **Sélection-élitiste** P
    fin
    si $z = 1 \mod 2$ **alors**
      $P \leftarrow$ **Sélection-par-tournois** P
    fin
    $P \leftarrow$ **Passage-génération-suivante** P
  fin
  Élément-le-mieu-adapté P
fin

Algorithme 3: Algorithme Génétique

Complexité en $i * O(l^2)$

2.4 Application sur un exemple

Choix des paramètres Un des points faibles de cet algorithme, sinon son caractère aléatoire, c'est le « grand » nombre de variable que doit fournir l'utilisateur.

Nous avons obtenus des résultats satisfaisants, en un temps raisonnable , avec les paramètres suivant :

$$\begin{cases} n=2\,000 \\ q=1 \\ i=1\,00 \end{cases}$$

Temps : 47 sec

Parcours : 1039.9

3 Recuit simulé

3.1 Un peu de thermo

Le principe du recuit simulé repose sur une analogie avec la thermodynamique : une dégradation provisoire peut entraîner une amélioration plus tard. On peut donc autoriser une transformation locale provoquant une augmentation de la fonction d'erreur avec une certaine probabilité.

Propriété . *L'état d'un matériau dépend de la température à laquelle il est porté, or la distribution de Gibbs-Boltzmann mesure la probabilité* $P(X)$ *de visiter l'état* X *en fonction de son énergie* $E(X)$ *et de la température* T *:*

$$P(x) = \frac{e^{-\frac{\Delta E}{kT}}}{N(T)}$$

(k *est la constante de Boltzmann et* $N(t)$ *la fonction de partition)*

Si $\frac{T}{T_0} \ll \frac{E}{E_0}$, la probabilité d'observer un état donné sera à peu près la même quel que soit l'état.

Si T est faible alors les état de faible énergie deviendront plus probables

Si T est presque nulle, alors seuls les états d'energie nulles auront une probabilité non négligeable d'être observé.

En métallurgie, l'utilisation de cette propriété sert à produire des cristaux ayant une configuration d'énergie minimale.

Si on veut obtenir un état d'énergie minimum (un cristal) il faut T décroisse lentement, afin que l'équilibre thermique s'instaure entre deux décroissances de température.

FIGURE 7 – Recuit simulé.

3.2 Principe

Dans toute cette partie, on pourra assimiler "Distance du parcours" à "Énergie" .

1. On génère un premier trajet ;
2. On effectue un test de croisement
   - Si le test est positif alors on enleve le croisement ;
   - Sinon on crée le croisement avec une probabilité $e^{\frac{E}{T}}$;
3. On diminue la température ;
4. Si celle-ci est négligeable, alors on va à l'étape suivante. Sinon on retourne à l'étape 2 ;
5. On retourne à l'étape 2 autant de fois que l'utilisateur l'a décidé .

Premier trajet : On peut générer le trajet aléatoirement, ou alors on peut utiliser un algorithme glouton.

Test de croisement : On test si la permutation entre deux villes améliore le trajet, alors on effectue la permutation. Sinon on effectue la permutation avec une probabilité de $e^{\frac{E}{T}}$

Si cette variation est négative (c'est-à-dire qu'elle fait baisser l'énergie du système), elle est appliquée à la solution courante.

Sinon, elle est acceptée avec une probabilité $e^{-\frac{\Delta_E}{T}}$.

Diminution de la température

Diminution linéaire : $T \longleftarrow \lambda T$ avec $\lambda \in [0; 1[$

Diminution par palier : On garde T constant jusqu'à ce qu'on atteint un équilibre. et on diminue la température ensuite.

On choisiera la première méthode.

Critère d'arrêt : Dès que $T < T_0$ on s'arrête. La température critique T_0 est définie par l'utilisateur.

Et on réapplique le principe autant de fois que l'utilisateur le veut

3.3 Algorithme

Données : L , T, T_0, λ, i % T est la température initial du système, T_0 est la température critique, λ est la raison de la suite de la température, L contient les villes à parcourir avce des informations sur les distances entre les villes

Résultat : L' : liste % contient la liste du parcours de villes à effectuer

```
début
    L' ← Générer-trajet-initial S
    T ← L'
    pour k = 1 à i faire
        tant que T > T0 faire
            L'' ← transformation L'
            si Trajet L'' < Trajet L' alors
                L' ← L''
            fin
            sinon
                Tirage d'un nombre p ∈ [0; 1]
                si p > e^(-(Trajet L'' − Trajet L')/T) alors
                    L' ← L''
                fin
            fin
        fin
        si L' < T alors
            T ← L'
        fin
    fin
    Retourner T
fin
```

Algorithme 4: Algorithme de Recuit Simulé

3.4 Application sur un exemple

Temps : 32 sec

Parcours : 1048

IV Défi des 250 villes

1 Présentation

Le défi des 250 villes est un défi consultable sur Internet [7]

Nous nous sommes intéressé à ce problème pour tester si nos algorithmes étaient vraiment performant car c'est un problème ou de bon résultat ont déjà été trouvé : le meilleur jusqu'à ce jour (et sans doute le meilleur) est : 11.809

2 Résultat de nos algorithmes

2.1 Algorithme des fourmis

`Ant_algo L n m` signifie qu'on a lancé `n` fourmis `m` fois

```
#Ant_algo L 50 50
temps : 7158.09 - distance : 19.824

#Ant_algo L 500 10;;
temps : 13349.29 - distance : 21.452

#Ant_algo L 500 100;;
temps : 108123.34 - distance : 21.452

#Ant_algo L 10 2000;;
temps : 50176.23 - distance : 15.683 ::
```

2.2 Algorithme 2-opt

C'est le plus rapide et a de bon résultat si on l'initialise avec un algorithme glouton, mais il ne peut pas être améliorer tout seul

```
#Algo2opt L =
temps : 3.265 sec - distance : 15,102
```

2.3 Algorithme génétique

Il est très long et l'amélioration se fait très lentement après une heure (c.f. les tracés de stats en annexe).

`Algogenetique L n k m` signifie qu'on a lancé `n` generation `m` fois avec un facteur mutagène de `k`.

```
#Algogenetique L 20 2 500000;;
temps : 51790.711 sec - distance : 25.486

#Algogenetique L 20 2 1000000;;
temps : 109206.092 sec - distance : 24.479

#Algogenetique L 20000 10 1000;;
temps :  1082.185 sec - distance : 28.518

#Algogenetique L 20000 10 1000;;
temps :  106546.239 sec - distance : 24.742
```

2.4 Algorithme de recuit simulé

Le recuit simulé est surtout utile pour finalisé une amélioration, Ici on utilise le recuit simulé pour essayer d'améliorer l'algorithme par colonies de fourmis (celui qui a donné 15.68).

`Algorecuit L Ti To k m` signifie qu'on a une temperature initiale `Ti`, une température critique `To` et qu'on lance l'algorithme `m` fois avec un facteur décroissance de température `k`.

```
# Algorecuit L 50. 0.1 0.999999 1000;;
temps : 2304.89 sec - distance : 14.334
(*Ici on améliore l'algorithme 2-opt*)
```

```
# Algorecuit L 50. 1. 0.999 100;;
temps : 545.95sec - distance : 12.266
(*Amelioration de l'algorithme génétique le
plus performant obtenu ci-contre*)

# Algorecuit L 50. 0.1 0.999999 1000;;
temps : 2304.89 sec - distance : 11,933
(*Amelioration de l'algorithme par colonie de
fourmis le plus performant obtenu ci-contre*)
```

On peut synthétiser ce qu'on peut estimer être les meilleurs algorithmes par le tableau suivant :

Algo	Parcours	Temps
Fourmis	15.7	13 h
Algo génétique	24.48	30 h
Recuit simulé	$11,9$	38min

3 Combinaison de nos algorithmes

Il apparaît que les algorithmes que nous avons sont plus ou moins efficaces, mais sur une très longue durée pour certain.

Une idée, pour les rendre plus performant, serait de faire une combinaison de nos algorithmes

Une combinaison qui a donné de bon résultat est la suivante : Fourmi/2-opt–Génétique–Recuit

1. On lance 50 fois l'algorithme des fourmis, les fourmis étant elles-mêmes aidés par l'algorithme 2-opt
2. Ces 50 trajets sont injectés dans un algorithme génétique
3. La sortie est ensuite amélioré par l'algorithme de recuit simulé

Avec cet algorithme, après environ 1h30 (mais après 2j pour faire varier efficacement les paramètres), nous avons obtenu un parcours d'une longueur de 11.809, nous retombons donc sur le trajet le plus performant obtenu sur ce défi

Références

[1] Thomas H. CORMEN Charles E. LEISERSON Ronald L. RIVEST Clifford STEIN *Introduction à l'algorithmique* 2e édition, Dunod, 2005.

[2] http://fr.wikipedia.org/ *Wikipedia* , Wikimedia Foundation, 2001.

[3] http://fr.wikiversity.org/ *Wikiversité* , Wikimedia Foundation, 2006.

[4] Donald E. KNUTH Enumeration and Backtracking *The Art of Computer Programming, vol 4A* , Addison Wesley, 2006.

[5] FRANK MITTELBACH Michel GOOSSENS *Latex companion*

[6] http://themarvinproject.free.fr/ *???* , ???,???.

[7] http://labo.algo.free.fr/defi250/defi_des_250_villes.html *Défi des 250 villes* , ???,???.

Annexe

Calculs mené avec les configuration suivantes :

###Processeur
Intel Core 2 Duo T6500
Vitesse d'horloge théorique de la RAM : 2.1 GHz
Mémoire de la RAM : 3.00 GB

###OS :
Distribution GNU/Linux : Debian Squeeze
Noyau : Linux 2.6
Architecture : i686

###Logiciel utilisé
LinCaml v.3.1
CamlLight v.0.75

```
-----------------------------------------------------------------------------------------
Sous la meme machine, même algorithme (données statistiques) :
100 tris bulles de vecteurs de taille 500 en comptant leur création

WinCaml sous Windows :  14.55 s
LinCaml sous Windows : 13.64 s
JavaCaml sous Windows : 13.76 s
Caml-light sous Windows (**) :

(**) Le compilateur caml-light directement en ligne de commande
-----------------------------------------------------------------------------------------
Sous la meme machine, même algorithme (données statistiques) :
100 tris bulles de vecteurs de taille 500 en comptant leur création

* 64bits
-- LinCaml sous Debian : 34.03 s
-- JavaCaml sous Debian : 34.0 s
-- Caml-light sous Debian (**) : 12.13 s
* 32 bits
-- LinCaml sous Debian : 13.98 s
-- JavaCaml sous Debian : 12.81 s
-- Caml-light sous Debian (**) : 12.47 s


(**) Le compilateur caml-light directement en ligne de commande
-----------------------------------------------------------------------------------------
Sous la meme machine, même algorithme (données statistiques) :

JavaCaml sous Windows 32bits : 11.03 s
JavaCaml sous Debian 32bits : 8.47 s

-----------------------------------------------------------------------------------------
Sous la meme machine, même algorithme (données statistiques) :

* en 32bits
-- Calcul en C : 43.2 s
-- Calcul en Caml : 45.68 s
* en 64bits
-- Calcul en C : 13.8 s
-- Calcul en Caml : 67.85 s
-----------------------------------------------------------------------------------------
```

Implémentation CAML
– Algorithme des fourmis –

```
#open"random";;

(**Definition des types **)

type ville == float * float;;
type pheromone == float;;
type distance == float;;


(***Fonction de base ***)

let Calcul_distance (A:ville) (B:ville) =
let ((xa,ya),(xb,yb))=(A,B) in
 (sqrt((xa -. xb)**2. +. (ya -. yb)**2.) : distance);;

let Int_of_ville (villes:ville vect) (B:ville) =
let res = ref 0 in
let l = vect_length villes -1 in
for i = 0 to l do
if villes.(i)=B then res := i ;
done;
!res;;

let rec Enlever a liste = match liste with
  |[] -> []
  |h::t when h=a -> t
  |h::t -> h::(Enlever a t);;

let copy_matrix M =
let l = vect_length M in
let P = make_matrix l l M.(0).(0) in
for i = 0 to l-1 do
for j = 0 to l-1 do
P.(i).(j)<-M.(i).(j);
done; done;
P;;


(**Choix de la ville de depart**)

let Choix_ville_depart (L: ville list) = hd(L);;
(*On prend le premier element d'une liste de ville donné*)


(**Fonction sur les phéromones**)

let Creer_pheromones_depart (n:int) = (*n est le nombre de ville*)
make_matrix n n ((0.:pheromone));;
(*On crée une matrice de taille n*n, rempli de 0 car a l'origine aucun trajet n'ont de pheromones*)
(*Au cours de l'algorithme, des qu'une fourmis ira de la ville i a la ville j elle deposera des
pheromones dans la case i,j (et i,j car cette matrice est symétrique) *)

let Depot_pheromones (P:pheromone vect vect) (V:ville vect) i j=
(*la fourmis depose les pheromones apres avoir effectue le trajet i à j, elle en depose une quantite 1/
(distance du trajet)*)
if i=j then () else P.(i).(j) <- 1./.(Calcul_distance V.(i) V.(j)) +. P.(i).(j);
(*on depose aussi sur j,i pour conserver notre matrice symétrique*)
if i=j then () else P.(j).(i) <- P.(i).(j);;
(*La fourmi dépose les pheromone dans la matrice des pheromones, dans l'algorithme final cette
fonction sera applique a une matrice auxiliare pour evite la gene de certain pheromones dans le
parcours de nous fourmis*)

let Evaporation_pheromones (P:pheromone vect vect) k =
(*pheromone <- k x pheromone *)
let n = vect_length P in
for i = 0 to (n-1) do
for j = 0 to (n-1) do
P.(i).(j) <- k *. P.(i).(j);
done;
done;P;;
```

```
(**Choix de la ville suivante**)

(*la fonction délicate denotre algorithme*)
let Choix_ville_suivante (L:ville vect) (V:ville vect) (P:pheromone vect vect) (B:ville) =
let n = vect_length V -1 in
let ib = Int_of_ville L B in
let (dinv,zetatot) =
    let (res1,res2)=(ref 0.,ref 1.) in
    for k = 0 to (n-1) do
      res1 := !res1 +. (1./.(Calcul_distance V.(k) B));
      res2 := !res2 +. P.(ib).(Int_of_ville L (V.(k)));
    done; (!res1,!res2,!res3) in
let md =(1./.5.) in
let ms = 1. -. md in
let longueur = make_vect (n+1) 0. in
for i = 0 to n do
  let A = V.(i) in
  let ia = Int_of_ville L A in
  let zetaab = P.(ia).(ib) in
  let dab = Calcul_distance A B in
  longueur.(i) <- md*. ProdD /. (dinv *. dab) +. ms*.zetaab/.(zetatot);
done;
let p = random__float 1. in
let l_aux = ref 0. in
let i = ref (-1) in
while p> (!l_aux)
  do incr i; l_aux := !l_aux +. longueur.(!i);
  done;
V.(!i);;


(**choix final de la derniere fourmis**)

let Choix_ville_suivante_lastant (L:ville vect) (V:ville vect) (P:pheromone vect vect) (B:ville) =
let n = vect_length V -1 in
let i = ref 0 in
let ib = Int_of_ville L B in
let zetamax = ref 0. in
for k=0 to n do
  let A = V.(k) in
  let ia = Int_of_ville L A in
  let zetaab = P.(ia).(ib) in
  zetamax := max (!zetamax) zetaab;
  if (!zetamax = zetaab) then i:=k;
done;
V.(!i);;
```

```
(**Algo des fourmis**)

let Algo_fourmis (L:ville list) (n:int) (j:int) =

let M = list_length L in
let A = Choix_ville_depart L in
let P = ref(Creer_pheromones_depart M) in
let villes = vect_of_list L in

for m = 1 to j do
let P_aux = copy_matrix (!P) in
(*On met les pheromones dans une matrice auxiliaire pour que des modifications n'influent pas sur le
trajet des fourmis suivantes*)
for i = 1 to n (*trajet de chaque fourmis*)
  do
    let liste = ref L in
    liste := Enlever A (!liste);
    let B = ref A in
    while (!liste <> [])
      (*tant qu'il reste des chemins à parcourir, la fourmi va a la prochaine ville*)
      (*On applique le principe de l'algorithme*)
      do
        let vecte = vect_of_list (!liste) in
        let B_aux = Choix_ville_suivante villes vecte P_aux !B in
        Depot_pheromones P_aux villes (Int_of_ville villes !B) (Int_of_ville villes B_aux);
        B := B_aux;
        liste := Enlever !B (!liste);
      done;
    Depot_pheromones P_aux villes (Int_of_ville villes !B) (Int_of_ville villes A);
   done;
   P:=(P_aux);
   P:= Evaporation_pheromones (!P) 0.9;(*P <- 0.9 P*)
done;

(*Ensuite une fourmi parcours le passage en passant la ou il y a le plus de pheromone, c'est son
trajet qui nous interesse et qui correspond à celui ou il y a le plus de fourmis qui sont passe*)
let resultat = ref [A] in
let B = ref A in
  let liste = ref (Enlever A L) in
while (!liste<>[])
  do
  B := Choix_ville_suivante_lastant villes (vect_of_list (!liste)) !P !B;
  resultat := (!B)::(!resultat);
  liste := Enlever (!B) (!liste);
  done;
!resultat;;
```

Courbe de d'évolution des résultats

Implémentation CAML
– Algorithme 2-opt –

```
(**Definition des types**)

type point == float * float;;

(**Fonction usuelle**)

let distance p1 p2 = (*donne la distance de p1 a p2*)
  let ((x1,y1),(x2,y2)) = (p1,p2) in
  sqrt((x1-.x2)**2. +. (y1-.y2)**2.);;

let plus_proche_voisin p l = (*renvoie le ppv de p et la liste sans ce ppv*)
let rec aux p liste = match liste with
  | [] -> failwith""
  | [a] -> (a,distance p a,[]);
  | h::t -> let (ppv,dppv,l2) = aux p t in
(*la fnct auxiliare donne le plus proche voisin et la liste sans ce point, la distance est utile pour
ne parcourir la liste qu'une seule fois*)
    let dph = distance p h in
    if (min dph dppv) = dph
      then (h,dph,t)
      else (ppv,dppv,h::l2) in
let (a,b,c)=aux p l in (a,c);;

(**Generation du trajet initial**)

let generer_trajet_initial V =  (*on genere une suite de ppv*)
  let pi = ref( hd(V)) in
  let (L,aparcourir) = (ref [],ref V) in
  while  (!aparcourir <> [])
    do
    let (p1,l2) = (plus_proche_voisin !pi !aparcourir) in
    pi := p1;
    L := !pi::(!L);
    aparcourir := l2;
    done;
  !L;;

(**Amelioration du trajet, on enleve les croisements**)

let permuter i j V =
(*On permute i et j*)
(*on prend ce chemin à l'envers, c'est à dire en commencant par la fin*)
(*  ->                <-
    ---              -----
    \ /              \   /
     X     devient    | |
    / \               / \    *)
  let L = ref[] in
  for k = i to j
    do L:=V.(k)::(!L) done;
  for k = i to j
    do V.(k) <- hd(!L); L:=tl(!L); done;;

let amelioration2 trajet =
  let V = vect_of_list trajet in
  let n = ((vect_length V) -1) in
  for i = 0 to (n-1) do
  for j = 0 to (n-1) do
    if ( distance V.(i) V.(i+1) +. distance V.(j) V.(j+1) >
         distance V.(i) V.(j)   +. distance V.(i+1) V.(j+1) )
       then permuter (i+1) j V
       else ();
    (*on enleve le cas ou le croisement est entre le debut et la fin aussi*)
    if ( distance V.(i) V.(i+1) +. distance V.(n) V.(0) >
         distance V.(i) V.(n)   +. distance V.(i+1) V.(0) )
       then permuter (i+1) n V
       else ()
   done;done;
   (list_of_vect V);;
```

```
(**On applique le principe**)

let deuxopt L =
  let V = generer_trajet_initial L in
  let trajet1 = ref V in
  let trajet2 = ref [] in
  let L = ref V in
  while (!trajet1 <> !trajet2)
(*tant qu'il y a des croisements, on les enleve*)
  do
  trajet2 := !trajet1;
  trajet1 := !trajet2;
  done;
  !trajet1;;
```

Implémentation CAML
– Algorithme génétique –

```
#open"random";;
#open"sys";;

(**Definition des types**)

type point == float*float;;
type individu == point vect;;
type population == individu list;;

(**Fonction usuelle**)

let Copy parent =
  let l = vect_length parent in
  let fils = make_vect l parent.(0) in
  for k = 1 to (l-1)
    do fils.(k)<-parent.(k) done;
  fils;;

let rec Est_dedans e liste = match liste with
  |[] -> false
  |t::q when t=e -> true
  |t::q -> Est_dedans e q;;

let rec Enlever e liste = match liste with
  |[] -> []
  |t::q when t=e -> q
  |t::q -> t::( Enlever e q);;

let rec Extraire (i:int) (liste: point list) =  match (i,liste) with (*extrait le ieme élèment d'une
liste*)
  |(_,[a]) ->  (a,[])
  |(0,t::q) -> (t,q)
  |(m,t::q) -> let (a,l) = Extraire (m-1) q in (a,t::l);;

let Distance p1 p2 =
  let ((x1,y1),(x2,y2)) = (p1,p2) in
  sqrt((x1 -. x2)**2. +. (y1 -. y2)**2.);;

let Adaptation (i1:individu) = (*plus le trajet d'un individu est long, moins il est adapté*)
  let l = vect_length i1 in
  let S = ref 0. in
  for i = 0 to (l-2)
    do S:=Distance i1.(i) i1.(i+1) +. !S; done;
  S := !S +. Distance i1.(l-1) i1.(0);
!S;;

(**Generer la population initial**)

let Creer_individu L =
  let l = ref(list_length L) in
  let (liste,res,i) = (ref L,make_vect (!l) (hd(L)),ref 0 ) in
 while !liste <> []
   do let (u,v) = (Extraire (random__int (!l-1)) (!liste)) in
                        liste := v;
      res.(!i) <- u;
                        incr i;
   done;
res;;

let Generer_population_initial n L =
  let l = ref [] in
  for k = 1 to n do l:=(Creer_individu L)::(!l) done;
!l;;
```

```
(**Selection naturelle**)

(*Selection elitiste*)
let Tri_rapide liste =
  let rec partition l l1 l2 e = match l with
    |[] -> (l1, l2)
    |a::liste -> if Adaptation a > Adaptation e
                    then partition liste (a::l1) l2 e
                    else partition liste l1 (a::l2) e in
  let rec tri_rapide_aux liste = match liste with
    |[] -> []
    |e::l-> let (l1, l2) = partition l [] [] e in
       (tri_rapide_aux l1)@(e::(tri_rapide_aux l2)) in
tri_rapide_aux liste;;

let Selection_elitiste p =  (*On trie les points et on conserve les n/2 meilleurs*)
  let l = list_length p in
  let L = ref(Tri_rapide p) in
  let res = ref [] in
  for k = 1 to (l/2)
    do res := hd(!L)::(!res); L := tl(!L); done;
  !res;;

(*selection par tournois*)
let rec Selection_par_tournois p = match p with
  |[] -> []
  |[a] -> [a]
  |i1::i2::t -> if (Adaptation i1) > (Adaptation i2)
                  then i2::(selection_par_tournois t)
                  else i1::(selection_par_tournois t);;
```

```
(**Passage a la generation suivante**)

(*Croisement (fct principale) *)
let Croisement_aux parent1 parent2 i j =
  let (fils1,fils2) = (Copy parent1,Copy parent2) in
  let l = vect_length parent1 in
  let (L1,L2)=(ref [],ref []) in
  for k = i to j (*On crée le bout de croisement à injecter aux enfants*)
     do L1:=fils1.(k)::(!L1); L2:=fils2.(k)::(!L2); done;
  let (L1_aux,L2_aux) = (ref !L1,ref !L2) in (*passage délicat, on évite les villes en double fils1*)
  for k = (l-1) downto 0
    do
      if (k<i || k>j)
        then begin
             if Est_dedans fils1.(k) (!L2_aux)
               then begin
                    let b = ref true in
                    while !b
                      do
                      if (!L1_aux=[]) then b:=false (*on évite ainsi des boucles infinies*) else
                      if (Est_dedans (hd(!L1_aux)) (!L2_aux))
                         then begin
                              L2_aux:= Enlever (hd(!L1_aux)) (!L2_aux);
                              L1_aux:= tl(!L1_aux);
                              end
                         else begin
                              fils1.(k) <- hd(!L1_aux);
                              L1_aux := tl(!L1_aux);
                              b:=false;
                              end
                      done;
                    end
               else ()
             end
        else ()
    done;
  let (L1_aux,L2_aux) = (ref !L1,ref !L2) in (*même combat*)
  for k = (l-1) downto 0
    do
      if (k<i || k>j)
        then begin
             if Est_dedans fils2.(k) (!L1_aux)
               then begin
                    let b = ref true in
                    while !b
                      do
                      if (!L2_aux=[]) then b:=false (*on évite ainsi des boucles infinies*) else
                      if (Est_dedans (hd(!L2_aux)) (!L1_aux))
                         then begin
                              L1_aux:= Enlever (hd(!L2_aux)) (!L1_aux);
                              L2_aux:= tl(!L2_aux);
                              end
                         else begin
                              fils2.(k) <- hd(!L2_aux);
                              L2_aux := tl(!L2_aux);
                              b:=false;
                              end
                      done;
                    end
               else ()
             end
        else ()
    done;
  for k = j downto i (*On applique le crossing over sur les enfants*)
    do
      fils1.(k) <- hd(!L2);
      fils2.(k) <- hd(!L1);
      L1:= tl(!L1);
      L2:= tl(!L2);
    done;
(parent1,parent2,fils1,fils2);;
```

```
(*on effectue une mutation, on interchange deux villes au hasard*)
let Mutation fils =
let l = vect_length fils in
let (i,j) = (random__int (l-1),random__int (l-1)) in
let u = fils.(i) in
   fils.(i) <- fils.(j);
   fils.(j) <- u;;

let Croisement parent1 parent2 facteur_mutagene =
  let l = vect_length parent1 in
  let (u,v)=(random__int(l-1),random__int(l-1)) in
  let (i,j) = (min u v,max u v) in  (*on a i et j qui définissent le croisement*)
  let (p1,p2,f1,f2) = Croisement_aux parent1 parent2 i j in
  if (random__int facteur_mutagene) = 0 then Mutation f1 else (*on effectue la mutation*)
  if (random__int facteur_mutagene) = 0 then Mutation f2 else
  ();
  (p1,p2,f1,f2);;


let rec Generation_suivante P facteur_mutagene = match P with
  |[] -> []
  |[a] -> [a]
  |i1::i2::t -> let (p1,p2,f1,f2)= Croisement i1 i2 facteur_mutagene in
                                                                    p1::p2::f1::f2::
(generation_suivante t facteur_mutagene);;

(*un fois l'algo fini, on cherchera l'individu le mieux adapté au problème parmis toute la population*)
let rec Element_mieux_adapte P = match P with
  |[a] -> a
  |t::q -> let a = Element_mieux_adapte q in
           if Adaptation t < Adaptation a then t else a;;

(**Algorithme genetique**)

let algogenetique L n q iteration = (*L liste des points à parcourir, n le nombre d'individus choisis,
q facteur mutagène*)
let t = ref (time()) in
let P = ref (Generer_population_initial n L) in
for m = 0 to iteration
  do
    if m = 0 mod 2
      then P := Selection_elitiste !P
      else P := Selection_par_tournois !P;
      P := Generation_suivante !P q;
  done;
print_float(time() -. !t);
Element_mieux_adapte !P;;
```

Courbe de d'évolution des résultats

Implémentation CAML
– Algorithme du recuit simulé –

```
(**Def du type de point**)

type point == float * float;;

(**Fonction usuelle**)

let Distance p1 p2 = (*donne la distance de p1 a p2*)
  let ((x1,y1),(x2,y2)) = (p1,p2) in
  sqrt((x1-.x2)**2. +. (y1-.y2)**2.);;

let Plus_proche_voisin p l = (*renvoie le ppv de p et la liste sans ce ppv*)
let rec aux p liste = match liste with
  | [] -> failwith""
  | [a] -> (a,distance p a,[]);
  | h::t -> let (ppv,dppv,l2) = aux p t in
(*la fnct auxiliaire donne le plus proche voisin et la liste sans ce point, la distance est utile pour
ne parcourir la liste qu'une seule fois*)
          let dph = distance p h in
          if (min dph dppv) = dph
            then (h,dph,t)
            else (ppv,dppv,h::l2) in
let (a,b,c)=aux p l in (a,c);;

(**Generation du trajet initial**)

let Generer_trajet_initial V =  (*on génére une suite de ppv*)
  let pi = ref( hd(V)) in
  let (L,aparcourir) = (ref [],ref V) in
  while  (!aparcourir <> [])
    do
    let (p1,l2) = (Plus_proche_voisin !pi !aparcourir) in
    pi := p1;
    L := !pi::(!L);
    aparcourir := l2;
    done;
  !L;;
```

```
(**On applique le test de croisements**)

let Permuter i j V =
(*On permute i et j*)
(*on prend ce chemin à l'envers, c'est à dire en commençant par la fin*)
(*    ->                <-
    -----              -----
    \   /              \   /
      X      devient    | |
    /   \              /   \    *)
   let L = ref[] in
   for k = i to j
     do L:=V.(k)::(!L) done;
   for k = i to j
     do V.(k) <- hd(!L); L:=tl(!L); done;;

let Amelioration trajet T =
  let V = vect_of_list trajet in
  let n = ((vect_length V) -1) in
  for i = 0 to (n-1) do
  for j = 0 to (n-1) do
    if ( Distance V.(i) V.(i+1) +. Distance V.(j) V.(j+1) >
         Distance V.(i) V.(j)   +. Distance V.(i+1) V.(j+1) )
       then Permuter (i+1) j V (*si ca ameliore alors on change*)
       else begin
              let p = random__float 1. in(*sinon on effectue le chgt avec un proba en exp(-Delta E/T)*)
                if (p < exp(-.(Distance V.(i) V.(j)   +. Distance V.(i+1) V.(j+1))/.(T)))
                                                                then Permuter (i+1) j V
                  else ();
            end;

    (*on enleve le cas ou le croisement est entre le debut et la fin aussi*)
    if ( Distance V.(i) V.(i+1) +. Distance V.(n) V.(0) >
         Distance V.(i) V.(n)   +. Distance V.(i+1) V.(0) )
       then Permuter (i+1) n V (*meme combat*)
       else begin
              let p = random__float 1. in
                if (p < exp(-.(Distance V.(i) V.(j)   +. Distance V.(i+1) V.(j+1))/.(T)))
                                                                then Permuter (i+1) j V
                  else ();
            end;
   done;done;
   (list_of_vect V);;


(**On applique le principe**)

let Recuit L T T0 lambda =
  let temps = ref(time()) in
  let trajet1 = ref L in
  let t = ref T in
    while (!t > T0)
 (*tant que la temperature est élévée*)
    do
    trajet1 := Amelioration (!trajet1) !t;
    t := !t *. lambda;
    done;
  (!trajet1);;


let Algo_Recuit L T T0 lambda=
  let temps = ref(time()) in
  let trajet1 = ref(recuit (Generer_trajet_initial L) T T0 lambda) in
  let i = ref 2. in
  while( T /. (10.*. !i))>T0
    do
      trajet1 := Recuit (!trajet1) (T /. !i) T0 lambda ;
      i := !i +. 1.;
    done;
  !trajet1;;
```

Courbe de d'évolution des résultats

Complexité théorique

Complexité asymptotique

On prendra comme opérations unitaires les appels aux fonctions auxiliaires, les complexités seront donc données pour un nombre de ville fixé

Fourmis :

Le programme principal est constitué de deux boucles `for`.

La complexité théorique est donc en $\mathcal{O}(n \cdot j)$ avec n le nombre de fourmis et j le nombre d'itérations (de trajets effectués par chacune des fourmis)

Génétique

Le programme principale est constitué de deux boucles `for`.

La complexité théorique est donc en $\mathcal{O}(n \cdot i)$ avec n le nombre d'individus et i le nombre d'itérations (nombre de passages aux nouvelles générations)

Recuit

Un calcul de recuit est constitué d'une boucle `while`. T_{k_i} est la référence (valeur initiale à T_{k_0}), $T_{k_{i+1}} \leftarrow \lambda T_{k_i}$ avec comme condition d'arrêt que $T_{k_i} < T_c$ et le calcul termine car $\lambda < 1$.

La complexité théorique d'un tel calcul est donc en $\mathcal{O}\left(\frac{\ln(\frac{T_c}{T_{k_0}})}{\ln \lambda}\right)$. En effet à la i^e itération $T_{k_i} = T_{k_0}\lambda^i$, dernière itération $cT_c \simeq T_0\lambda^c$

Le programme de recuit est lui-même constitué d'une boucle `while` qui effectue un calcul de recuit simulé avec comme température initiale $T_{k_0} = \frac{T_0}{k}$ où k est un entier naturel strict. positif. k est la référence (valeur initial à 1), $k \leftarrow k+1$ avec comme condition d'arrêt $\frac{T_0}{10*k} \leqslant T_c$. Le programme se termine bien car on a $T_c > 0$ et $\frac{T_0}{10*k} \xrightarrow[k\to\infty]{} 0$

On effectue ainsi $\left\lfloor \frac{T_0}{10T_c} \right\rfloor$ calcul de recuit.

La complexité du programme est donc en $\sum_{k=1}^{\left\lfloor \frac{T_0}{10T_c} \right\rfloor} \mathcal{O}\left(\frac{\ln(\frac{T_c}{T_{k_0}})}{\ln \lambda}\right) = \sum_{k=1}^{\left\lfloor \frac{T_0}{10T_c} \right\rfloor} \mathcal{O}\left(\frac{\ln(\frac{kT_c}{T_0})}{\ln \lambda}\right) = \mathcal{O}\left(\frac{\ln\left(\left(\left\lfloor \frac{T_0}{10*T_c} \right\rfloor\right)! * \frac{T_c}{T_0}\right)}{\ln \lambda}\right)$

Limite de la complexité théorique

Problème lié à la complexité théorique : Le problème de ce type de complexité et qu'elle ne nous donne qu'un "comportement". Que ce soit un programme récursif ou itératif on considère comme 'unité' l'appel à la fonction dans le cas d'un programme récursif et une itération dans le cas d'un programme itératif. Seulement la complexité rendue est peu précise, on ne rend pas une complexité temporelle (compter le nombre d'opérations élémentaires) mais une majoration de la complexité asymptotique (comportement).

Opération		$\text{Complexité}_{\text{thé}}$	$\text{Complexité}_{\text{expé}}$ (entier)	$\text{Complexité}_{\text{expé}}$ (flottant)
Addition	$x + y$	$\mathcal{O}(\log x + \log y)$	1 add	1 mult
Soustraction	$x - y$	$\mathcal{O}(\log x + \log y)$	1 add	1 mult
Multiplication	$x \cdot y$	$\mathcal{O}(\log x \cdot \log y)$	1 mult $\simeq$ 10 add	1 à 2 mult
Division	$x = q \cdot y + r$	$\mathcal{O}(\log q \cdot \log y)$	5 à 10 mult	2 à 10 mult

FIGURE 1 – Exemple de valeur pour la complexité théorique et expérimentale

Problème lié à l'énumération exhaustive : Compter les opérations élémentaires est une tâche fastidieuse et pas forcément payant, dans le sens où la complexité ne sera pas forcément précise, lorsqu'on travaille sur des données importantes, comme par exemple avec des matrices 250×250, car la complexité d'un opération peut être amené à dépendre des paramètres, voir figure 2

Opération	Complexité (cas favorable)	Complexité (cas défavorable)
Un appel à une cellule	$\simeq$ 40 add	$\simeq$ 1400 add

FIGURE 2 – Exemple de cas de complexité variable

Avantage de la complexité expérimentale : Voilà pourquoi il semble plus utile de calculer une complexité expérimentale. On trace g le temps d'exécution en fonction de la taille du paramètre et on conclut au f tel que $g \sim f$ (en faisant une régression).

Cette méthode a pour avantage de nous donner une très bonne idée de la complexité réelle de la fonction.

Complexité expérimentale

Fourmis

Génétique

Recuit

Récapitulatif

Algorithme	Variable & Complexité	
Fourmis	Villes	$x^{2.84}$
	nombre de fourmis	x
	nombre d'itération	x
Génétique	Villes	x^2
	nombre d'individus	x
	nombre d'itération	x
	facteur mutagène	cst
Recuit	Villes	x^2
	température	$\ln x$
	facteur décroissance	$\frac{1}{\ln x}$

FIGURE 3 – Tableau récapitulatif

```
#open"sys";;
let miroir liste =
  let rec aux l1 l2 =
  match l1 with
   |[] -> l2
   |h::t -> aux t (h::l2) in
  aux liste [];;

(*****Fonction qui calcule la complexité*****)

let Calcul_Complexite fonction n m i =
(*fonction ne doit avoir qu'un paramètre à faire varier*)
(*on fait varier le parametre de n à m avec un pas de i*)
(*Ce sera des flottants pour permettre un tracé plus facile*)
(*On peut aisément adapter le code pour y mettre des entiers*)
let t = ref (time())
and L = ref [] in

let k = ref n in
while !k < m
  do
  t:= time();
  fonction !k;
  L := (k,time()-. !t):: !L ;
  print_float !k; print_string " "; print_float (time()-. !t); print_newline();
  k:=!k +. i;
  done;
miroir(!L);;

(*****Tracé de fonction*****)

(**Fonction graphique élémentaire **)
#open "graphics";;
let pixel_x (x,xmin,xmax)=int_of_float((x-.xmin)/.(xmax-.xmin)*.float_of_int(size_x()-1));;
let pixel_y (y,ymin,ymax)=int_of_float((y-.ymin)/.(ymax-.ymin)*.float_of_int(size_y()-1));;
let coord_x (x,xmin,xmax)=float_of_int(x)*.(xmax-.xmin)/.float_of_int(size_x()-1)+.xmin;;
let coord_y (y,ymin,ymax)=float_of_int(y)*.(ymax-.ymin)/.float_of_int(size_y()-1)+.ymin;;

(**Fonction principale qui trace une liste de point (f(x),g(y)) dans un cadre prédéfini**)
let tracer_liste (xmin,xmax,ymin,ymax) liste f g (n,m) (cercle,axe,droite) texte =
(*Donnée : coordonnée des extremités de la fenetre, la liste des points à tracer, les fonctions f et
g, n et m le 'pas' de la légende, cercle, axe et droite trois booléens qui indique si on affiche les
points tracés avec des cercle, si on donne la légende et si on trace une droite reliant le premier
point avec le dernier. Enfin texte est le texte à afficher dans la fenêtre graphique*)
  open_graph"";
  clear_graph();
  let (xn,yn) = dernier_element liste in
  let (x0,y0) = hd(liste) in
  wait 4.; (*on laisse le temps à l'utilisateur de mettre la taille de la fenêtre graphique comme il
la souhaite*)
    let tracer_axe n m = (*fonction auxilaire qui trace un axe*)
     let i = ref xmin in
     while !i <= xmax
       do
       moveto (pixel_x(!i,xmin,xmax)) (pixel_y(ymin,ymin,ymax));
       draw_string(string_of_int( int_of_float(!i)));
       i := !i +. n;
       done ;
     let j = ref ymin in
       while !j <= ymax
       do
       moveto (pixel_x(xmin,xmin,xmax)) (pixel_y(!j,ymin,ymax));
       draw_string(string_of_int( int_of_float(!j)));
       j := !j +. m;
       done
  in
    let tracer_droite() = (*fonction auxilaire qui trace la droite reliant les points extemes*)
      set_color green;
      moveto (pixel_x(f x0,xmin,xmax)) (pixel_y(g y0,ymin,ymax));
      lineto (pixel_x(f xn,xmin,xmax)) (pixel_y(g yn,ymin,ymax));
      set_color black
   in
```

```
if droite then tracer_droite();
if axe then tracer_axe n m;

   let afficher_texte texte =(*fonction auxiliaire qui affiche le texte*)
     if abs2(xmax) > abs2(xmin) then
     if abs2(ymax) > abs2(ymin) then
      begin moveto (pixel_x((xmax-.xmin)/.2. ,xmin,xmax)) (pixel_y((7.*.ymax-.ymin)/.8.,ymin,ymax));
      draw_string texte
      end
      else begin moveto (pixel_x((xmax-.xmin)/.2. ,xmin,xmax)) (pixel_y
((7.*.ymin-.ymax)/.8.,ymin,ymax));
      draw_string texte
      end
      else if abs2(ymax) > abs2(ymin) then
      begin moveto (pixel_x((xmin-.xmax)/.2. ,xmin,xmax)) (pixel_y((7.*.ymax-.ymin)/.8.,ymin,ymax));
      draw_string texte
      end
      else begin moveto (pixel_x((xmin-.xmax)/.2. ,xmin,xmax)) (pixel_y
((7.*.ymin-.ymax)/.8.,ymin,ymax));
      draw_string texte
      end;
 in
afficher_texte texte;

  (*fonction qui trace les point (f x,g y) et les relie*)
  moveto (pixel_x(f x0,xmin,xmax)) (pixel_y(g y0,ymin,ymax));
  let rec aux = function
    |[] -> ();
    |(x,y)::t -> lineto (pixel_x(f x,xmin,xmax)) (pixel_y(g y,ymin,ymax));
                 if cercle then begin
                           set_color red;
                           draw_circle (pixel_x(f x,xmin,xmax)) (pixel_y(g y,ymin,ymax)) 2;
                           set_color black;
                                end;
                 aux t in
  aux liste;;


(**Fonction qui calcul automatiquement le cadre**)
let boite liste f g =
let (a,b) = hd liste in
let x = ref (f a) and X = ref (f a)
and y = ref (g b) and Y = ref (g b) in
let rec aux = function
  |[] -> ()
  |(a,b)::t -> if f a < !x then x:= f a else
               if f a > !X then X:= f a;
               if g b < !y then y:= g b else
               if g b > !y then Y:= g b;
               aux t in
aux liste;
(!x,!X,!y,!Y);;

(**Fonction qui trace la liste de points (f(x),g(y)) dans un cadre adapté*)
let tracer_stats l f g (cercle,axe,droite) (n,m) texte=
tracer_liste (boite l f g) l f g (n,m) (cercle,axe,droite) texte ;;


(***** Fonction qui permet d'enregistrer une liste de point dans un .txt*****)

exception Fichier_existe;;
let ouvrir text =  (*close_in est dangereux car supprime le fichier si déjà existant, la fonction
ouvrir ouve un fichier s'il n'existe pas et renvoie une erreur sinon*)
  try
  let c = open_in text in close_in c;
  raise Fichier_existe;
  with |Fichier_existe-> failwith("Fichier "^text^" existe");
       |_ -> open_out text;;

let inserer_texte l dossier nom=
cd dossier;
let c = ouvrir nom in
```

```
  let rec aux = function
    |[] -> ()
    |(i,j)::t -> let text = ("("^(string_of_float i)^","^(string_of_float j)^")\n") in
                 output c text 0 (string_length text);
                 aux t in
   aux l;
close_out c;;

(*****Fonction qui permet d'entrer une liste de point d'un .txt dans une liste*****)

let couple_of_string texte =
if texte.[0]<>`(` then failwith"couple_of_string";
let k = ref 1 in
while texte.[!k]<>`,` do incr k done;
let a = sub_string texte 1 (!k-1) in
let u = !k+1 in
let j = ref 1 in
while texte.[!k]<>`)` do incr k; incr j; done;
let b = sub_string texte u (!j-2) in
a,b;;

let texte_of_liste dossier fichier =
  cd dossier;
  let c = open_in fichier in
  let L = ref [] in
  let k = ref true in
  while !k = true
   do
     try
     let ligne = input_line c in
     let (i,j)=couple_of_string ligne in
     let n = float_of_string i
     and m = float_of_string j in
     L:=(n,m)::(!L);
     with End_of_file -> k:= false;
  done;
close_in c;
miroir(!L);;


(***Exemple***)

include"Métaheuristique/Fourmis";;

let Dimension = 100.;; (*dimension de l'espace*)
let rec creer_Liste_villes = function
  | 0. -> []
  | n -> (random__float Dimension, random__float Dimension)::creer_Liste_villes(n-.1.);;

let liste = Calcul_Complexite f 1. 100. 1.
    where f = fun x -> Algo_Fourmis (creer_Liste_villes x) 100 100;;
```