Le Problème du voyageur de commerce

Charles Bouillaguet

26 juin 2004

Définition du problème Le problème du voyageur de commerce est un problème classique d'optimisation : il s'agit de trouver la plus courte tournée permettant de visiter n villes et de revenir au point de départ en ne visitant chaque ville qu'une seule fois.

Ce problème se trouve généralement formulé dans le langage des graphes : on considère un graphe (S, A) où S, les sommets du graphe, représentent les villes, et A, les arêtes, représentent les routes. Un cycle à k sommets est alors un ensemble de k sommets $(s_0, s_1, \ldots, s_k)$ tels que $s_k = s_0$ et $(s_i, s_{i-1}) \in A$ pour $i = 1, 2, \ldots, k$. La longueur d'un tel cycle est la somme des longueurs des arêtes qui le composent. Un cycle qui relie tous les sommets fois et une seule est appelé cycle Hamiltonien. Dans la suite, on notera $n = |S|$ le nombre de sommet du graphe.

Le but du problème est de déterminer le cycle Hamiltonien de plus courte longueur. Par abus de langage, nous écririons souvent cycle, pour dire cycle hamiltonien.

Nous travaillerons avec certaines hypothèses :

- nous supposerons que la distance utilisée pour fournir la longueur des arêtes est euclidienne et qu'elle vérifie donc l'inégalité triangulaire) :

$$d(a, c) \leq d(a, b) + d(a, c) \quad (a, b, c) \in S^3$$

- Nous supposerons de plus que le graphe n'est pas orienté : les arêtes (a, b) et (b, a) ne sont qu'une seule et même chose, qu'on notera donc $\{a, b\}$.
- Enfin, nous supposerons que le graphe est complet, c'est-à-dire que chaque sommets est relié à tous les autres.

Justifions brievement que le plus court cycle que nous recherchons existe : l'hypothèse de la complétude du graphe permet d'affirmer que des cycles hamiltoniens existent. En effet $(s_0, \ldots, s_n, s_0)$ en est bien un. L'ensemble des cycles hamiltonien est donc non vide, et il est fini. On peut considérer l'élement de longueur minimale.

Explosion combinatoire Une approche très naïve du problème consisterait à déterminer la longueur de tous les cycles Hamiltoniens, et à prendre le plus court. On se heurte cependant à une difficulté de taille : leur nombre. Dénombrons-les : se donner un tel cycle, c'est se donner un ordre de parcours des n villes. Il y en a $\frac{(n-1)!}{2}$, car si on considère un parcours donné, il revient au même de commencer à partir du premier sommet, où à partir du 2ème, ..., ou à partir du n-ième. De plus, il revient au même de parcourir la boucle dans un sens ou dans l'autre (d'où la division par deux).

Dans le cadre de ce TIPE, nous avons travaillé sur un problème à 250 villes, issu d'un défi trouvé sur internet [1]. Nous avons donc pu comparer nos résultats avec ceux obtenus par d'autres participants. Dans toute la suite, nous poserons donc $n = 250$.

Il y a donc $\frac{249!}{2}$ cycles à tester. Sachant que $\frac{249!}{2} \simeq 10^{490}$, il y a bien plus de cycles à tester que d'atomes dans l'univers... Le soleil aura largement le temps d'exploser avant la fin des tests.

On peut voir l'ensemble des permutations de $[\![1, n]\!]$, dont la racine est le premier sommets, et dont la profondeur est de n. Les $(n-1)$ fils de la racine sont les sommets qui seront visités en deuxième position. Chacun d'entre eux a $(n-2)$ fils qui seront visités en troisième position, et ainsi de suite. Cet arbre à 250! feuilles, qui représentent des cycles hamiltoniens.

Nous avons mis au point un programme qui résound exactement le PVC (pour un nombre de ville modeste) en explorant systématiquement l'arbre des possibilités (en en faisant un parcours en profondeur). Sa complexité est donc à priori factorielle. Cependant, une astuce a été introduite, qui diminue notablement le temps d'éxecution : certaines arêtes très longues ne figureront jamais dans des cycles courts. L'exploration de branches entières de l'arbre pourrait donc être evitée s'il s'avère qu'elle conduirait à des cycles plus long que le plus court cycle déjà connu.

Le programme garde donc en mémoire la longueur du plus court cycle qu'il connait. Lors de l'exploration de l'arbre, si la somme des longueurs des arêtes conduisant du sommet actuel à la racine est plus longue que le plus court trajet connu, l'exploration de cette branche s'arrête : elle conduirait nécessairement à un trajet qui n'est pas le plus court.

En regardant le tableau du temps d'exécution de ce programme en fonction du nombre de ville (figure 2), on constate que le temps de calcul est environ multiplié par 3 lorsqu'on passe de 16 villes à 17, puis environ par 2 lorsqu'on passe de 17 à 18 et de 18 à 19, et enfin par 10 lorsqu'on passe de 19 à 20. Ces données ne sont pas suffisantes pour donner une formule même empirique de la complexité en fonction de n, mais il semble en tout cas clair qu'elle est inférieure à $O(n!)$ (sans la dernière valeur, nous aurions peut-être osé suggérer $O(2^n)$...). Il n'est toutefois envisageable de l'utiliser sur notre problème à 250 villes.

Nécessité de méthodes d'approximation En fait, à ce jour aucun n'algorithme ne fournit la solution du PVC en un temps qui serait une fonction polynomiale de nombre de ville n. Nous nous sommes donc tournés vers des méthodes d'approximation, fournissant un court cycle hamiltonien en un temps raisonnable.

La première que nous ayons mise en oeuvre est la méthode des plus proches voisin. Partant d'un sommet, on se rend au plus proche qu'on a pas encore visité, et on réitère le processus. Une fois qu'on a visité tous les sommets du graphe, on retourne au premier pour fermer le cycle. Cette méthode fournit un résultat en un temps très bref (notre implémentation a une complexité en $O(n^2)$).

Qualitativement, on voit (figure 3) que le résultat est améliorable. Il comporte des croisement (qu'il serait avantageux de décroiser), et lorsque l'on s'approche de la fin, l'algorithme doit relier quelques sommets épars qui ont été négligés auparavent, et qui sont très éloignés. La pénalité pour ne pas s'en être occupé auparavent est relativement importante, comme on le voit sur la figure.

```
type resultat = Cut | Chemin of float * int list;;
(*
 g : le graphe (une matrice des distances)
 depart : le sommet de d\'epart (pour pouvoir y revenir)
 actuel : le sommet o\`u nous sommes actuellement
 deja_vu : un tableau precisant les sommets d\'ej\`a visit\'es
 borne : le plus court chemin connu pour les sommets restants \`a voir
*)

let rec coupure_search g depart actuel deja_vu borne =
  let cout_min =  ref borne and parcours_min = ref [] and trouve = ref false
  and  teste = ref false in
  deja_vu.(actuel) <- 1;

  for i = 0 to (Array.length g) -1 do
    if deja_vu.(i) = 0 then
    begin
teste := true;
if !cout_min >= g.(actuel).(i) then
  let res = coupure_search g depart i deja_vu (!cout_min -. g.(actuel).(i)) in
  match res with
      Chemin (cout,parcours) when cout +. g.(actuel).(i) <= !cout_min ->
          cout_min := g.(actuel).(i) +. cout;
  parcours_min := i::parcours;
  trouve := true;
    | Cut -> ()
    end;
  done;
  deja_vu.(actuel) <- 0;
  match (!trouve, !teste) with
      true, _ -> (Chemin (!cout_min,!parcours_min))
    |false, true -> Cut
    |false, false -> Chemin (g.(actuel).(depart), [depart]);;
```

FIG. 1 – La fonction récursive explorant l'arbre des possibilités

Nombre de villes	**Temps de calcul**
13	0,5s
15	35s
16	4min 03s
17	13 min
18	35 min
19	65 min
20	10 h

FIG. 2 – Performance du programme de résolution exacte

Une méthode plus théorique Après cette première approche très empirique, et malgré tout peu satisfaisante, voyons maintenant une idée un peu plus raffinée. Nous allons construire un arbre couvrant minimal (A.C.M.) sur notre graphe (figure 4) : c'est un sous-graphe de (A, S) connexe, acyclique, qui a tous les sommets de A (il est couvrant), et dont la longueur est minimale (i.e. si on prend un autre arbre couvrant, il sera plus long).

On peut construire cet arbre, grâce à l'algorithme de Prim en un temps polynomial. Notre implémentation de cet algorithme (qui utilise des tas binaires) a une complexité en $O(n^2 * \log n)$. Une fois l'arbre construit, effecuons-en un parcours, comme indiqué sur la figure 5. C'est un parcours en profondeur où lorsqu'on repasse par un sommet en remontant, on le refait apparaître dans le parcours.

Ainsi, si un sommet à k fils, il apparaîtra $k+1$ fois dans le parcours. Chaque arête est ainsi parcourue deux fois : une fois dans chaque sens. Ainsi, la longueur du parcours obtenue est-elle exactement deux fois celle de l'arbre couvrant. Maintenant, supprimons les sommets apparaissant plus d'une fois, en laissant passer leur première occurence dans le parcours, puis en supprimant les suivantes. Le parcours donné en exemple devient : abcdefgh. Rajoutons une occurence du premier sommet à la fin : on obtient un cycle hamilonien. Ce faisant, on ne fait que diminuer la longueur du parcours, car la distance est euclidienne. L'inégalité triangulaire nous garantit que la longueur du parcours n'augmente pas lorsqu'on retire les sommets récurrents. On a ainsi obtenu un cycle dont la longueur est inférieure à deux fois la longueur de l'ACM. Notons-le H

Considérons maintenant le plus court cycle, et notons le M. Coupons une de ses arêtes. On obtient alors un arbre couvrant. Il est donc plus long que l'ACM, et on a alors :

$$L(H) \leq 2L(ACM) \leq 2L(M)$$

On a ainsi que le cyle H (figure 6) est au pire deux fois plus long que le plus court cycle.

Ce résultat est intéressant pour deux raisons : tout d'abord car c'est l'un des ares qui fournissent une majoration de l'erreur commise, et ensuite car il donne également une minoration de la longueur du plus court cycle (par la longeur de l'ACM).

De manière surprenante, le résultat de cette méthode (figure 6) est moins bon que celui fournit par la méthode des plus proches voisins, pour laquelle aucune borne théorique n'existe.

Optimisation locale Les solutions fournies par les deux méthodes précédentes sont peu satisfaisantes (on à l'impression qu'à la main on ferait mieux), car elles comportent soit des croisements soit de grandes arêtes qui ont l'air ridicule. On va se pencher ici sur une méthode systématique et rapide pour améliorer ces solutions.

Tout d'abord, on va munir l'ensemble des cycles hamiltoniens d'une distance définie par : la distance entre deux cycles a et b est

$$d(a, b) = n - (\text{le nombre d'arêtes que } a \text{ et } b \text{ ont en commun})$$

d est bien une distance, et le seul point non-immédiat à vérifier est l'inégalité triangulaire. $d(a, b)$ indique en fait le nombre d'arêtes que a et b ne partagent pas.

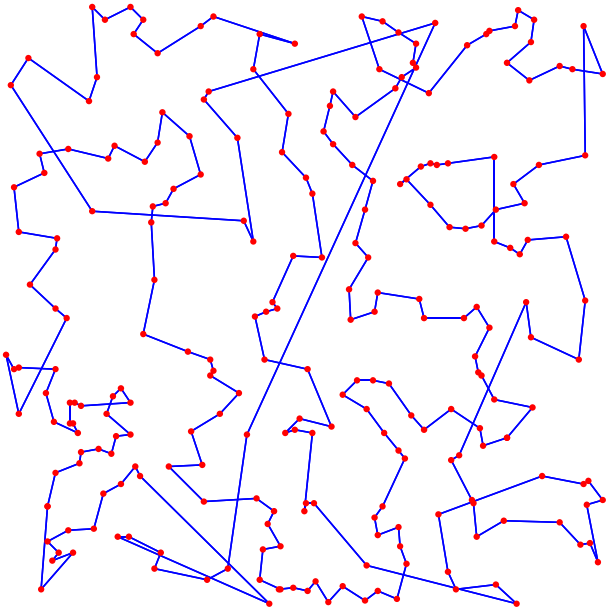

FIG. 3 – Méthode des plus proches voisins. longueur : 14,7321

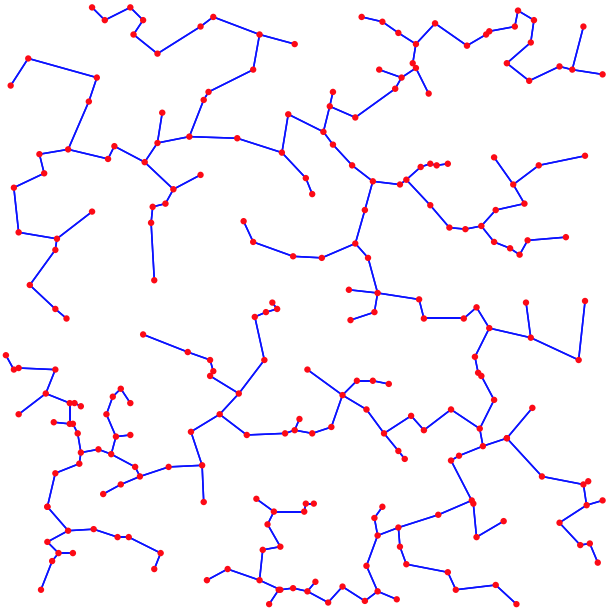

FIG. 4 – Un arbre couvrant minimal. longueur : 10,40

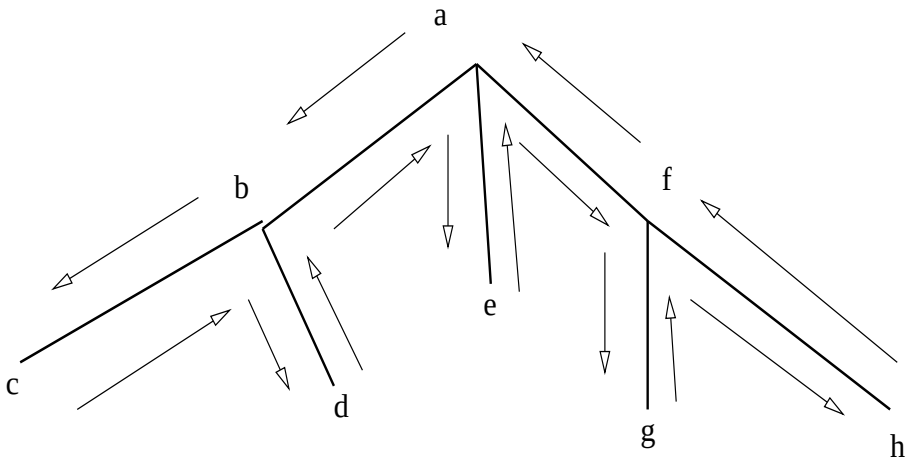

FIG. 5 – Le parcours généré est abcbdbaeafgfhfa

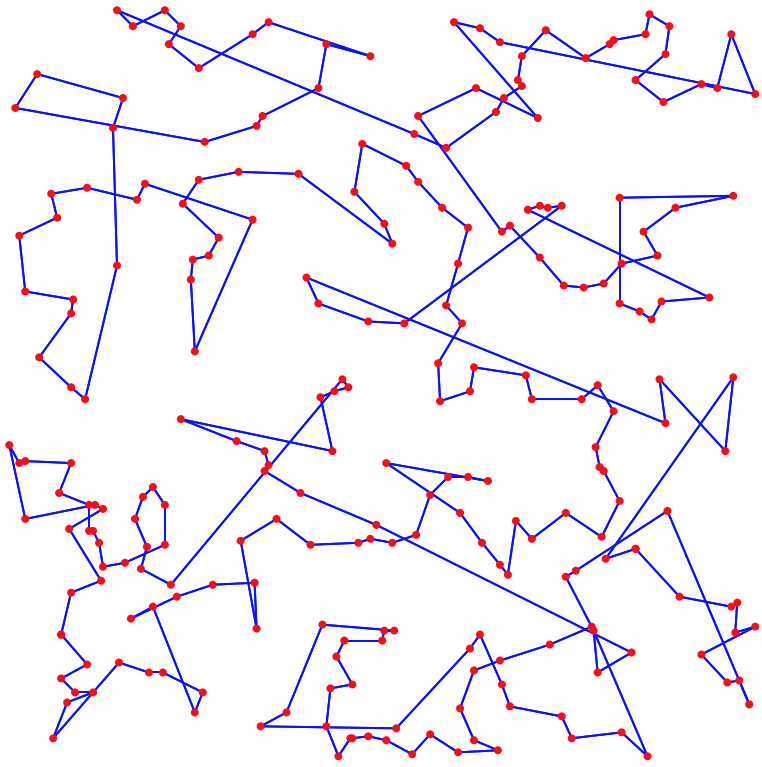

FIG. 6 – Méthode de l'arbre couvrant minimal. longueur : 16,348

L'idée est de partir d'une solution connue (par exemple, fournie par une des deux méthodes précedentes), et d'explorer systématiquement une spère de rayon 2 autour de cette solution à la recherche d'un cycle qui serait plus court encore. Cette méthode s'appelle donc l'optimisation 2-Opt. Notons $\mathcal{S}(a_0, 2)$ la sphère de rayon 2 centrée en $a_0 \quad (a_0 \in S)$.

Partant d'un cycle a_0 , les cycles qui sont dans $\mathcal{S}(a_0, 2)$ sont exactement les cycles obtenus en "croisant" (figure 7) deux arêtes de a_0 : en effet, si l'on part de a_0 et que l'on modifie deux arêtes, pour que le résultat soit encore un cycle, la seule solution consiste à croiser les deux arêtes. En particulier, si a_0 contient des croisements, les parcours obtenus en les décroisant seront dans $\mathcal{S}(a_0, 2)$, et ils seront plus courts que a_0 (en effet, en décroisant, on y gagne toujours, par conséquence de l'inégalité triangulaire).

Il ressort de cela qu'il y a $n(n-1)$ élements une telle sphère. Déterminer la longueur du plus court cycle dans $\mathcal{S}(a_0, 2)$ peut donc être fait en $O(n^2)$.

Une fois que l'on a déterminé le plus court cycle a_1 dans $\mathcal{S}(a_0, 2)$, on peut recommencer le processus en considérant la spère de rayon 2 centrée sur a_1 ... Et ainsi de suite.

Cet méthode améliore sensiblement les solutions précédentes (figure 9), et fournit un résultat très rapidement.

En pratique, cependant, l'algorithme converge au bout d'une cinquantaine d'itérations : il ne trouve pas de cycle plus court dans la spère. Cette rapidité de convergence est en fait problématique : il existe peut-être d'autres cycles plus court juste un peu plus loin, mais l'algorithme est tombé dans un "puit" duquel il ne peut plus sortir.

Nous allons donc maintenant envisager une manière de limiter cet effet qui limite la qualité des solutions trouvées.

Introduisons de l'erreur Pour contourner ce phénomène, nous modifions l'algorithme 2-Opt de la manière suivante :

Lors de l'exploration de $\mathcal{S}(a_0, 2)$, dès qu'un cycle a_1 plus court que a_0 est rencontré, l'exploration de $\mathcal{S}(a_0, 2)$ s'interrompt, et on commence l'exploration de la sphère centrée en a_1 . Toutefois, on va également de temps en temps choisir d'interrompre l'exploration de $\mathcal{S}(a_0, 2)$ pour commencer celle de a_1 , avec $a_1 \in \mathcal{S}(a_0, 2)$, même si a_1 est plus long que a_0 , de manière pseudo-aléatoire. En effet, pour tout élément b de $\mathcal{S}(a_0, 2)$, on calcule un nombre pseudo-aléatoire $x \in [0; 1]$. Lors de la k-ième itération de l'algorithme, b sera préféré à a si

$$x \leq f\big(L(a) - L(b), k\big)$$

$L(a)$ étant la longueur du cycle a. f est une fonction de seuil. Elle détermine la probabilité qu'un choix non optimal ait lieu. Nous avons utilisé une fonction du type :

$$f(x, k) = e^{x*(\rho + k\,\mu)}$$

SuggÃ©rÃ©e dans [2]. Voir figure 10, une représentation de f.

Empiriquement, nous avons déterminé des valeurs pour les constantes ρ et μ qui semblent fournir de bons résultats :

$$\rho = 45$$

$$\mu = 4,5$$

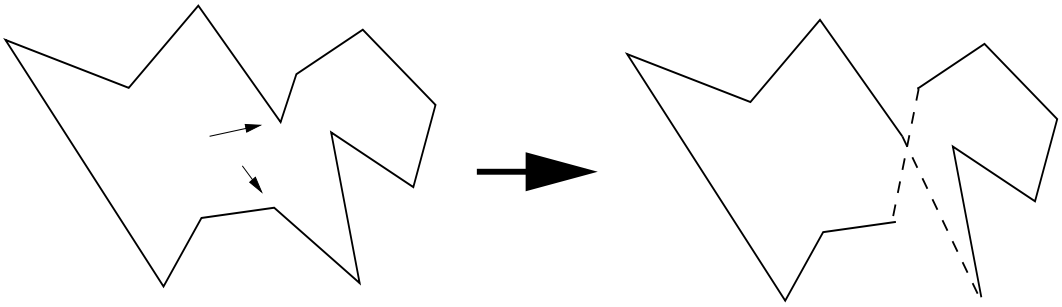

FIG. 7 – Un croisement transformant a_0 en un élement de $\mathcal{S}(a_0, 2)$

```
(*
calcule la diff\'erence de longueur du parcours p si on relie le premier point au j-i\‘eme
*)
let gain g p j n =
     g.(p.(n-1)).(p.(j)) +. g.(p.(0)).(p.(j+1))
-. ( g.(p.(n-1)).(p.(0)) +. g.(p.(j)).(p.(j+1)) );;



let rec two_opt g p =
  let n = Array.length g and actuelle = ref (List.tl p)
  and meilleure_trans = ref (0,0) and meilleur_gain = ref 0.0 in

   for i = 0 to n-2 do
     let tmp = Array.of_list !actuelle in
     for j = 1 to n-2 do
       let gain_1 = gain g tmp j n in
       if gain_1 < !meilleur_gain then
         (
           meilleur_gain := gain_1;
           meilleure_trans := tmp.(0), tmp.(j);
         )
      done;
      actuelle := (List.tl !actuelle) @ [List.hd !actuelle];
     done;

 match !meilleure_trans with
   0,0 -> p
 | i,j ->
    let bidon = (Trajet.connect_with_first j (Trajet.rotate_until i (List.tl p)))
    in
      two_opt g (bidon @ [List.hd bidon]);;
```

FIG. 8 – la fonction récursive réalisant l'optimisation 2-Opt

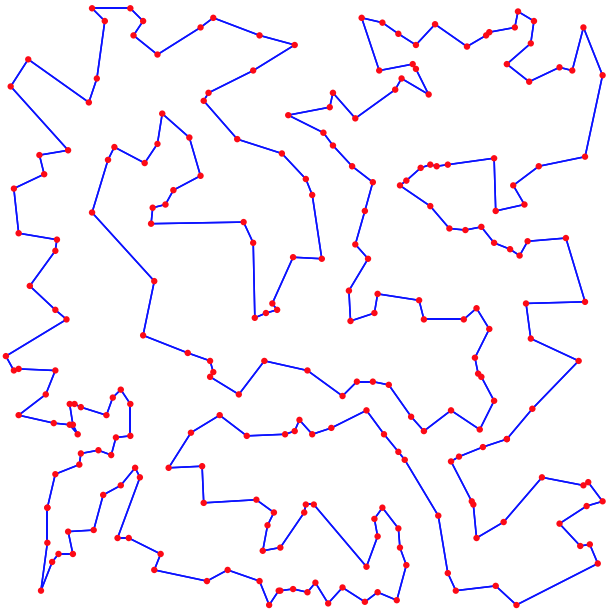

FIG. 9 – Optimisation locale 2-Opt appliquée au résultat de la méthode des plus proches voisins. Tous les croisements ont disparus. Longueur : 12,2641

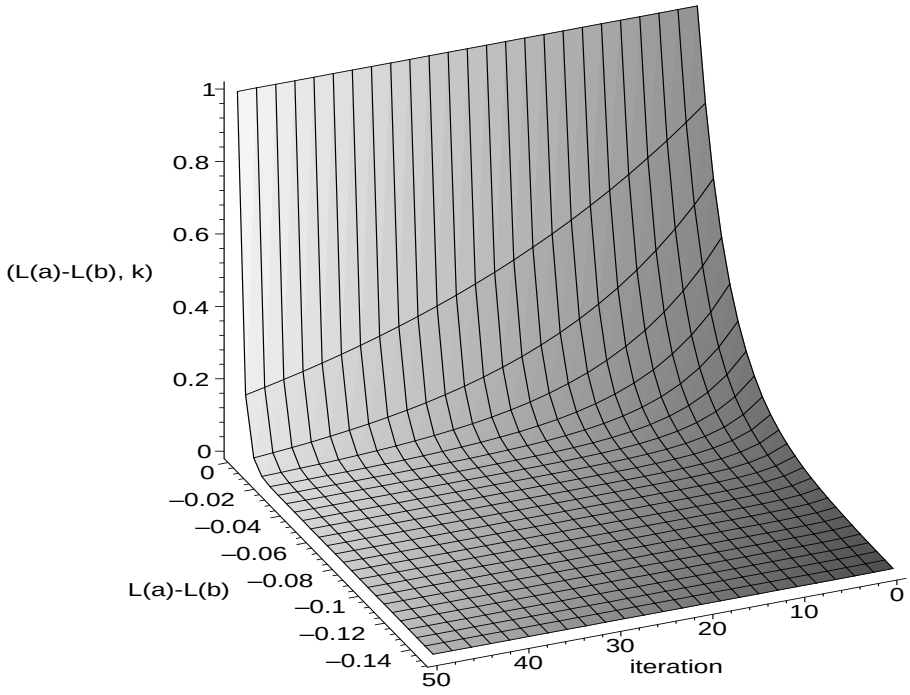

FIG. 10 – Une représentation de la fonction de seuil utilisée

Des valeurs trop grandes (peu d'aléa introduit) ne permettent pas de s'éloigner suffisament d'un minimum local et ne permettent donc pas d'améliorer réelement la solution. Des valeurs trop petites ralentissent la convergence, et si trop d'aléa est introduit, on finit par trop s'éloigner des bonnes solutions.

Sur la figure 11 on peut voir comment évolue la longueur des parcours choisis dans la sphère explorée, et commet l'ajout de l'aléa permet d'obtenir une meilleure solution.

Au bout d'un petit quart d'heure de calcul, nous sommes parvenus au résultat de la figure 12.

Conclusion et améliorations envisageables Une dizaine de candidats au défi des 250 villes ont trouvé une solution dont la longueur est de $11,809$. Il n'est bien sûr pas prouvé que ce soit l'optimum, mais cela semble probable. Nous n'avons donc pas gagné le défi :-). Toutefois, nous avons obtenu des solutions qui ne sont que $0,7\%$ plus longues que la plus courte connue, ce qui n'est pas si mal.

En guise de remarques finales, nous avancerons deux idées :

- Pour éviter les problèmes des minimums locaux dans 2-Opt, on pourrait, une fois qu'on en a etteint un, explorer une sphère plus grosse (de rayon 3, puis 4, etc.). Nous n'avons pas implémenté cette idée, mais le temps de calcul nécessaire est fortement croissant avec la taille des sphères.
- Les bonnes solutions que nous avons obtenues ont visiblement des troncs communs. On pourrait tirer parti de cela en mettant au point une méthode de fusion de deux solutions : on garde les tronçons communs, puis on détermine pour chaque portion intérmediaire laquelle des deux solutions est la meilleure.

Références

[1] Aupetit, A. Défi des 250 villes (http://labo.algo.free.fr/defi250/defi_des_250_villes.html). 2000.

[2] Charolois, R. Heuristiques pour le problème du voyageur de commerce (http://eleves.ec-lille.fr/~charoloi/backtojava/projects/x/salesman.fr.html). 2001.

[3] Cormen, Th., Leiserson, Ch., and Rivest, R. *Introduction à l'algorithmique*. Dunod, 1997, ch. 5 et 24, pp. 84–89 et 489–499.

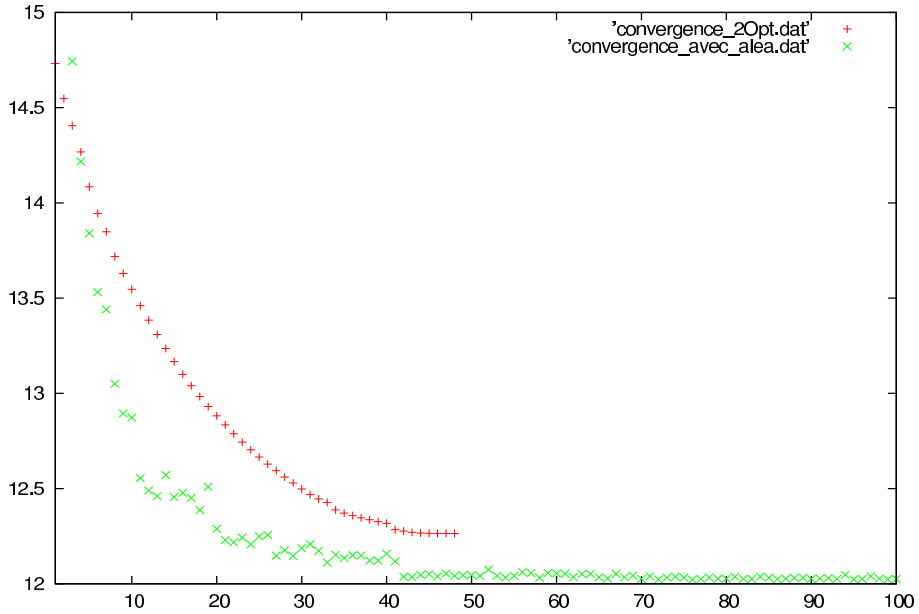

FIG. 11 – Convergence des algorithmes 2-Opt et 2-Opt avec aléa sur 150 itérations

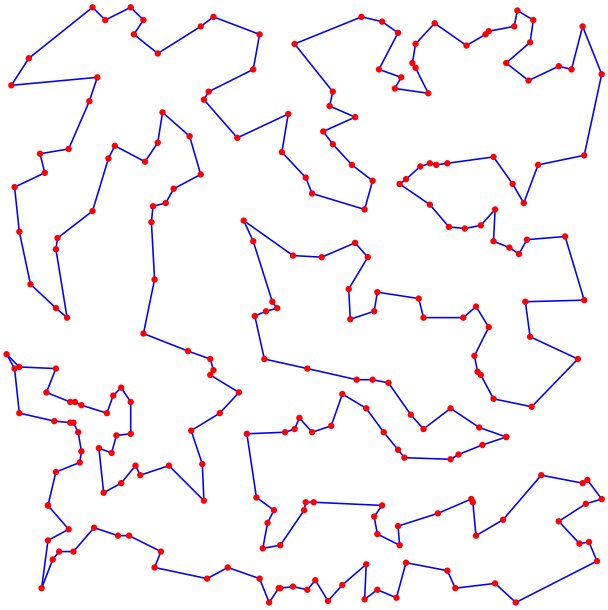

FIG. 12 – Longueur : 11.909