IFT6580 — Devoir 3

Eric Buist (buisteri@iro.umontreal.ca)

26 avril 2006

Résumé

Le problème du voyageur de commerce a été traité par un très grand nombre d'auteurs et avec différents algorithmes et heuristiques. Dans ce travail, nous allons évaluer l'efficacité du réseau élastique et d'un algorithme génétique dans la résolution approximative de ce problème. Nous allons comparer ces deux heuristiques et évaluer leur capacité à produire une solution proche de l'optimum.

1 Introduction

Le voyageur de commerce [7] est le problème de base des problèmes de tournées de véhicules plus complexes. Il consiste à déterminer le cycle hamiltonien de longueur minimale dans un graphe complet non orienté dont chaque sommet représente une ville ou un client et chaque arête, une route. Chaque arête a un coût qui représente habituellement la distance à parcourir pour aller d'un sommet vers un autre.

Nous allons nous concentrer sur des problèmes de type géométrique, c'est-à-dire qu'à chacune des villes $i = 0, \ldots, n-1$ du problème est associée une position $\mathbf{X}_i = (x_i, y_i)$ dans le plan cartésien. La distance $d_{ij} = d(\mathbf{X}_i, \mathbf{X}_j)$ entre les villes est simplement la distance euclidienne arrondie à l'entier le plus près. Ainsi, les distances satisfont l'inégalité du triangle et sont symétriques.

Il existe plusieurs techniques permettant de résoudre ce problème, mais aucun algorithme exact actuel n'est efficace puisque le problème est *NP*-complet. Il existe par contre une multitude d'heuristiques permettant de se rapprocher de la solution optimale.

Dans ce travail, nous allons tester deux de ces heuristiques : un réseau de neurones élastique et un algorithme génétique. Le réseau élastique consiste en un anneau qui s'étire sous l'effet de forces pour former un tour complet. L'algorithme génétique, quant à lui, construit une population de solutions initiales et effectue des croisements entre les meilleures solutions pour créer des générations successives de la population.

Nous allons évaluer la performance de ces deux métaheuristiques adaptées au problème du voyageur de commerce en examinant la solution produite et en la comparant avec la solution optimale. Nous allons aussi examiner la progression de la solution au cours de l'algorithme et, dans

le cas de l'algorithme génétique, la variation de la solution produite en fonction des nombres aléatoires sous-jacents.

La section suivante de ce travail décrit plus en détails les deux métaheuristiques testées. La section 3 traite des problèmes rencontrés pendant l'implantation de ces métaheuristiques. La section 4 présente les résultats expérimentaux obtenus et leur analyse. La section 5 conclut ce travail. En annexe, nous exposons la structure générale de notre programme implantant les deux métaheuristiques ainsi qu'un guide de l'utilisateur.

2 Métaheuristiques testées

2.1 Réseau élastique

Un réseau élastique [4] est un anneau initialement circulaire formé de points et sur lequel des forces sont exercées. La figure 1 donne le pseudocode de l'algorithme que nous avons implanté pour le réseau élastique. Cet algorithme diffère légèrement de celui présenté dans [4], prévoyant de nouvelles conditions d'arrêt pour traiter le cas de la divergence par imprécision numérique et l'absence de mouvement de l'anneau en raison de variations trop petites. L'algorithme consiste à effectuer un certain nombre r d'itérations avec une même valeur du paramètre K, chaque itération déplaçant l'anneau et l'étirant vers les villes. Après r itérations ou si l'anneau cesse de bouger, l'algorithme diminue la valeur de K. Lorsque l'algorithme ne diverge pas, il est possible de suivre l'anneau et de prendre les villes dans l'ordre pour former un tour complet. Nous allons maintenant décrire plus en détails les éléments fondamentaux de l'algorithme.

Nous avons choisi de placer le centre de l'anneau au centre des villes et de définir son rayon de façon relative à la ville la plus éloignée du centre. Pour ce faire, nous calculons le rectangle centré en (x_c, y_c) et de dimensions (w,h) minimales contenant toutes les villes. L'anneau est placé en (x_c, y_c) et son rayon est $\rho \min(w,h)/2$, où $\rho \in [0,1]$ est le *rayon relatif* de l'anneau. Le nombre de points sur l'anneau est $m = \theta n$ où $\theta \geq 1$ est le *nombre de points relatif* sur l'anneau. Les points sur l'anneau sont représentés par $\mathbf{Y}_j \in \Re^2$, où $j = 0, \ldots, m-1$. La détermination du centre et du rayon de l'anneau sont dans $O(n)$ puisqu'il faut tester toutes les villes tandis que la construction de l'anneau se fait dans $O(m)$. Puisque m dépend de n, l'initialisation de l'anneau se fait dans $O(n)$.

Lors de la mise à jour de l'anneau, pour chaque point $j = 0, \ldots, m-1$, $\mathbf{Y}_j = \mathbf{Y}_j + \Delta\mathbf{Y}_j$, où

$$\Delta\mathbf{Y}_j = \alpha \sum_{i=0}^{n-1} w_{i,j}(\mathbf{X}_i - \mathbf{Y}_j) + \beta K(\mathbf{Y}_{j+1} + \mathbf{Y}_{j-1} - 2\mathbf{Y}_j)$$

avec

$$w_{i,j} = \frac{\phi(d(\mathbf{X}_i, \mathbf{Y}_j), K)}{\sum_{\ell=0}^{m-1} \phi(d(\mathbf{X}_i, \mathbf{Y}_\ell), K)}.$$

La fonction $\phi(d,K)$ est donnée par

$$\phi(d,K) = e^{-d^2/(2K^2)}.$$

1. Construire l'anneau initial.
2. Initialiser $K = K_0$.
3. Répéter r fois
   (a) Mettre l'anneau à jour.
   (b) Si la distance minimale à l'anneau est inférieure à ε, arrêter puisque nous avons trouvé une solution.
   (c) Si la distance minimale à l'anneau est supérieure à Θ, interrompre l'algorithme et rapporter un échec de recherche en raison d'une divergence.
   (d) Si l'anneau n'a pas changé suite à sa mise à jour, interrompre la boucle pour sauter à 4.
4. $K = aK$, pour $a \in [0,1]$.
5. Arrêter l'algorithme si $K < K_{\text{MIN}}$.

FIG. 1 – Algorithme du réseau élastique

Le paramètre α détermine l'importance de la force attirant les points de l'anneau vers les villes tandis que le paramètre β règle l'importance de la force assurant la cohésion de l'anneau. Plus K est petit, moins la force de cohésion est importante et plus les points sont attirés vers les villes. Le poids $w_{i,j}$ représente quant à lui l'influence de la ville i sur le point j de l'anneau.

Pour calculer les poids, pour chaque valeur de i, il faut d'abord déterminer les valeurs de $\phi(d(\mathbf{X}_i, \mathbf{Y}_j), K)$. En supposant chaque calcul de $\phi(d,K)$ dans $O(1)$, le calcul des poids pour un i fixé est dans $O(m)$. Le calcul de tous les poids se fait donc dans $O(mn) = O(n^2)$.

Pour évaluer la distance minimale à l'anneau, l'algorithme itère sur chaque ville i et tente de trouver le point $j^*(i)$ de l'anneau le plus près. La distance minimale à l'anneau est donnée par

$$d_{\min} = \min_{i=0,\ldots,n-1} d(\mathbf{X}_i, \mathbf{Y}_{j^*(i)}) = \min_{\substack{i=0,\ldots,n-1 \\ j=0,\ldots,m-1}} d(\mathbf{X}_i, \mathbf{Y}_j).$$

Il est facile de voir, d'après la formule précédente, que le calcul de la distance minimale se fait dans $O(mn) = O(n^2)$.

La construction de la solution à partir de l'anneau est effectuée par un algorithme *bucket sort* [9] fonctionnant de la façon suivante. D'abord, un tableau A de m éléments est construit et rempli de -1. Ensuite, pour chaque ville i, l'algorithme calcule $j^*(i)$ comme pour calculer la distance minimale et stocke i à l'indice $j^*(i)$ du tableau A. Si cet indice est déjà occupé par un autre élément, c'est-à-dire $A(j^*(i)) \neq -1$, l'algorithme gère la *collision* en plaçant l'élément dans la case la plus proche contenant -1. S'il atteint la fin du tableau pendant la recherche d'une case libre, l'algorithme recommence à l'indice 0. Comme $m \geq n$, l'algorithme trouvera toujours une position dans A où mettre i. Il suffit ensuite d'énumérer chaque élément de A qui n'est pas égal à -1 pour obtenir le tour complet. L'énumération se fait dans $O(m)$, mais elle est dominée par le coût de construction de A qui est $O(mn) = O(n^2)$ en supposant qu'il n'y aucune collision.

Ainsi, chaque étape de l'algorithme du réseau élastique a un coût dans $O(n^2)$.

2.2 Algorithme génétique

L'algorithme génétique [5] pour le voyageur de commerce fait évoluer une population de solutions pour tenter de l'améliorer. D'abord, une population initiale de N solutions est construite de façon aléatoire. Deux heuristiques ont été employées pour la construction aléatoire : une permutation totalement aléatoire de sommets et une insertion guidée par une permutation de sommets.

Avec une certaine probabilité p, une solution purement aléatoire est obtenue. Elle consiste simplement en une permutation des nombres $0,\ldots,n-1$. Nous nous attendons à ce que les solutions produites de cette façon soient de très mauvaise qualité, mais elles peuvent engendrer de la diversité dans les croisements et le coût de construction est seulement $O(n)$.

Avec probabilité $1-p$, un individu est construit par *insertion stochastique* : tout d'abord, une permutation aléatoire des nombres $0,\ldots,n-1$ est générée, mais cette fois, les clients de la permutation sont insérés dans le tour à l'endroit minimisant le détour imposé. Cela fournit de meilleures solutions que la permutation aléatoire et accélère la convergence de l'algorithme. Par contre, nous pouvons nous attendre à une moins bonne solution que celle obtenue avec un algorithme glouton tel que *farthest insertion* [7] qui choisit toujours le sommet le plus loin du tour courant à chaque itération. Le coût de construction de la solution passe en $O(n^2)$, car pour chaque ville insérée, il faut tester au plus n points d'insertion.

Les solutions construites sont triées par ordre croissant de distance totale. Peu importe la structure de données utilisée pour stocker la population de taille N, le coût de ce tri est en moyenne dans $O(N\lg N)$.

À chaque nouvelle génération, N individus de la population courante sont choisis pour la *reproduction*. Ensuite, une nouvelle population est créée de la façon suivante. D'abord, un parent est choisi parmi les N candidats à la reproduction. Avec une certaine probabilité q, le parent est copié tel quel dans la population suivante. Avec probabilité $1-q$, un autre parent est choisi et un *croisement* a lieu, générant un ou plusieurs enfants qui sont ajoutés à la population suivante, jusqu'à ce que sa taille soit N. Le processus est repris jusqu'à ce que la population suivante, qui remplace la population courante, compte N individus.

Avec une certaine probabilité, chaque nouvel individu peut subir des *mutations* afin de créer de la diversité par des perturbations. Ces nouveaux individus sont alors triés en ordre croissant de distance totale.

L'algorithme construit ainsi un certain nombre de générations jusqu'à ce que tous les individus de la population soient identiques. L'algorithme s'arrête aussi si, après un certain nombre d'itérations, la meilleure solution n'est pas mise à jour.

Afin de tirer parti des processeurs à double cœur, le programme utilise plusieurs fils d'exécution pour l'algorithme génétique. Chaque fil, représentant une île, implante son propre algorithme avec sa propre population de N individus. Chaque fil garde sa propre meilleure solution, mais une meilleure solution globale est conservée et retournée en fin d'exécution. L'algorithme se termine lorsque toutes les îles ont cessé d'évoluer. Soit M le nombre d'îles et N_j la taille de la sous-population sur l'île j, où $j = 0,\ldots,M-1$.

Cette division en sous-populations équivaut simplement à faire plusieurs essais séquentiellement et prendre l'essai qui a donné le meilleur résultat. Pour tenter de faire davantage que des

essais indépendants, notre algorithme permet la *migration* d'individus d'une île vers une autre. Pour ce faire, à chaque génération, la migration a lieu avec une certaine probabilité. Pendant la migration initiée par une île j, une île source j' et un nombre n' sont choisis au hasard, uniformément dans $\{0,\ldots,M-1\}\backslash\{j\}$ et $\{0,\ldots,\min\{N_j,N_{j'}\}-1\}$, respectivement. Alors, les n' meilleurs individus de l'île source j' remplacent les n' pires individus de l'île de destination j.

Nous allons maintenant expliquer plus en détails comment se passent la sélection, les croisements et les mutations. Comme nous le verrons, le coût de calcul est dominé par le croisement et la mutation qui sont, dans notre cas, dans $O(Nn^2)$.

2.3 Sélection

Nous avons implanté la sélection avec la méthode des rangs et l'échantillonnage stochastique universel [6]. À chaque individu $i = 0,\ldots,N-1$. un rang

$$f_i = \text{MAX} - (\text{MAX} - \text{MIN})i/(N-1)$$

est associé. Avec cette fonction f_i, l'individu 0 reçoit le rang MAX tandis que pour l'individu $N-1$, le rang est MIN. De plus, si $\text{MIN} + \text{MAX} = 2$,

$$\sum_{j=0}^{N} f_i = N.$$

Avec cette fonction de rang, le nombre espéré de fois qu'un individu est sélectionné est

$$E_i = Np_i = \frac{Nf_i}{\sum_{i=0}^{N-1} f_i} = f_i.$$

MIN et MAX ajustent la *pression de sélection* : plus les deux extrêmes sont rapprochées, plus la loi de probabilité pour la sélection tend vers une loi uniforme. Plus MIN et MAX sont éloignés, plus la politique de sélection est élitiste, c'est-à-dire qu'elle favorise les individus avec un rang élevé, donc une distance totale petite.

L'échantillonnage stochastique universel génère N parents à partir d'un seul nombre aléatoire de façon à ce que le nombre de fois qu'un individu i est sélectionné soit entre $\lfloor E_i \rfloor$ et $\lceil E_i \rceil$. Pour générer les N nombres aléatoires, l'algorithme de sélection génère un nombre u sur $[0,1]$ et ajoute $1\,N-1$ fois à ce nombre, ce qui produit N nombres $u, u+1,\ldots,u+N-1$ entre 0 et N. Le parent i, pour $i = 0,\ldots,N-1$, est donné par le plus grand i' pour lequel $f_{i'} < u+i$. Cette sélection ne nécessite qu'un seul nombre aléatoire et est dans $O(N)$.

2.4 Croisement

Nous avons implanté la recombinaison des arêtes comme méthode de croisement. Cette technique dresse d'abord une matrice d'arêtes (*edge map*) à partir des parents : l'élément $A(i,j)$ de cette matrice de booléens de taille $n \times n$ est vrai si et seulement si une arête (i,j) existe dans au moins

un tour parent. La construction de cette matrice, en excluant l'allocation de l'espace en mémoire, se fait dans $O(n)$.

L'algorithme de croisement débute en créant une solution initiale ne contenant que la ville 0. Ensuite, tant que la solution n'est pas complète, il répète le processus suivant. D'abord, les éléments $A(i, j')$ de la matrice des arêtes sont mis à faux (dans $O(n)$), avec j' l'indice du dernier sommet ajouté. Cela permet d'empêcher l'algorithme de reprendre le sommet j' à une itération subséquente. Ensuite, l'algorithme sélectionne le sommet i' tel que $A(j', i')$ est vrai et minimisant la somme des arêtes actives $\sum_{\ell=0}^{n-1} I[A(i', \ell) = \text{Vrai}]$, où $I[C]$ est la fonction indicatrice qui vaut 1 si la condition C est vérifiée et 0 sinon. Une arête (i, j) est dite *active* si j ne fait pas encore partie de la solution en cours de construction, c'est-à-dire que $A(i, j)$ est vraie.

Initialement, l'algorithme exclut les sommets qui n'ont aucune arête active puisqu'il ne sera pas possible de sortir du sommet ajouté sans prendre une arête étrangère. Par contre, si tous les sommets accessibles depuis le dernier sommet ajouté ont 0 arête active, l'algorithme en prend un au hasard. Si plusieurs sommets comportent le même nombre d'arêtes actives, l'algorithme en choisit un également au hasard.

Si aucun sommet ne permet de sortir de j', c'est-à-dire $A(j', i)$ est faux pour tous i, une arête étrangère est alors ajoutée au tour enfant. Le nouveau sommet est alors choisi au hasard parmi tous les sommets non encore ajoutés dans le tour.

Si le nombre d'arêtes actives est précalculé pendant l'initialisation et mis à jour pendant le croisement, chaque ajout de sommet se fait dans $O(n)$. Ainsi, la construction du tour enfant se fait dans $O(n^2)$. En supposant qu'il n'y a aucune copie (pire cas), il faudra effectuer N croisements pour créer la prochaine génération si bien que chaque génération nécessite un coût de calcul dans $O(Nn^2)$.

2.5 Mutation

Chaque individu de la population subit une mutation qui consiste soit à effectuer des échanges aléatoires, soit en une recherche locale destinée à améliorer la solution.

Dans le cas des échanges aléatoires, l'algorithme tire un nombre aléatoire uniformément sur $[0, 1]$ et effectue un échange si ce nombre est plus petit que la probabilité de mutation. Pour effectuer un échange, l'algorithme tire deux nombres au hasard entre 0 et $n-1$ et échange les clients correspondants dans la solution si les deux nombres sont différents. L'algorithme effectue des échanges tant et aussi longtemps que le test de mutation passe ou après avoir atteint un nombre maximal d'échanges. Un échange n'est pas compté si les deux indices tirés sont identiques. Cet algorithme s'exécute en temps constant par rapport à n et à N.

La recherche locale est quant à elle effectuée avec un voisinage de type 2-opt, c'est-à-dire on retire deux arêtes et on en ajoute deux nouvelles pour refermer le tour. La recherche se produit avec une certaine probabilité déterminée de façon indépendante pour chaque individu. Elle peut prendre la solution voisine diminuant le plus la distance parcourue (*best improvement*) ou la première qui diminue la distance (*first improvement*). Le choix de l'échange 2-opt est dans $O(n^2)$ dans le pire cas. La recherche se termine lorsque plus aucun élément du voisinage ne permet d'améliorer la

solution. Il est également possible de limiter le nombre maximal d'itérations de la recherche locale pour améliorer la performance.

En supposant que le nombre d'itérations est limité, la mutation est alors dans $O(n^2)$. Si elle est appliquée sur tous les individus, son coût total est alors dans $O(Nn^2)$, comme le coût des croisements.

3 Implantation

Nous avons implanté les deux heuristiques de la section précédente dans le langage de programmation Java. Pour ce faire, nous avons construit une classe représentant un client (`Customer`) ainsi qu'une liste spécialisée de clients (`CustomerList`). Cette liste encapsule un tableau d'indices et un second tableau de booléens permettant de tester en temps constant, sans parcourir la liste, si un client est dans la liste. Les opérations sur la liste s'assurent également qu'un client n'est pas inséré plusieurs fois et interdisent l'ajout d'éléments `null`. Une méthode permet également de calculer la longueur totale d'un tour. Un problème est décrit par la classe `Problem` comme un ensemble de clients. Un objet problème précalcule également toutes les distances entre les paires de sommets et les ajoute dans un cache sauvant des calculs répétitifs de racines carrées pour une plus grande efficacité.

3.1 Réseau élastique

L'implantation du réseau élastique semble facile, mais des problèmes de stabilité numérique sont apparus, faisant exploser l'anneau. La méthode `Math.exp` de Java retourne 0 si le paramètre donné est trop élevé, ce qui peut produire un terme de normalisation nul et des poids $w_{i,j}$ infinis. Notre première tentative de solution consistait à changer la fonction de poids comme suit.

$$\begin{aligned} w_{i,j} &= \frac{\phi(d(\mathbf{X}_i,\mathbf{Y}_j),K)}{\sum_{\ell=0}^{m-1}\phi(d(\mathbf{X}_i,\mathbf{Y}_\ell),K)} \\ &= \frac{e^{-d(\mathbf{X}_i,\mathbf{Y}_j)^2/(2K^2)}}{\sum_{\ell=0}^{m-1}e^{-d(\mathbf{X}_i,\mathbf{Y}_\ell)^2/(2K^2)}} \\ &= \left[\sum_{\ell=0}^{m-1} e^{(d(\mathbf{X}_i,\mathbf{Y}_j)^2-d(\mathbf{X}_i,\mathbf{Y}_\ell)^2)/(2K^2)}\right]^{-1} \end{aligned}$$

Cette solution semblait fonctionner, mais elle augmentait beaucoup le temps de calcul puisqu'au lieu de calculer mn exponentielles, il fallait en calculer mn^2. Cela rendait impossible la résolution du tsp442 dans un temps raisonnable.

Nous sommes donc revenus à la formule originale et avons employé une normalisation de distance pour résoudre le problème d'imprécision de la fonction `Math.exp`. Pour ce faire, tous les points de l'anneau et les clients sont normalisés de façon à ce que $\mathbf{X}' = (\mathbf{X} - (\mu,\mu))/\sigma$. Grâce

à l'encapsulation, il a été possible d'ajouter la normalisation sans affecter les autres parties du programme, comme l'interface graphique par exemple.

Soit $(x_\mathrm{m}, y_\mathrm{m})$ la position maximale d'une ville avec $x_\mathrm{m} \leq x_i$ et $y_\mathrm{m} \leq y_i \ \forall i = 0, \ldots, n-1$. Le facteur de décalage est alors $\mu = \min(x_\mathrm{m}, y_\mathrm{m})$ tandis que le facteur d'échelle est $\sigma = \min(w, h)$. De cette façon, les points se voient transposés sur $[0, 1]$ sur un des axes. Il est important d'employer le même facteur de mise à l'échelle pour chaque axe afin de ne pas tordre l'espace des solutions et modifier le problème. Cette astuce a résolu le problème et a même accru la qualité des solutions produites.

Pour la vérification de la condition d'arrêt, il ne faut pas vérifier directement que $\Delta Y_j = 0$, car cela se produit très rarement. Il faut plutôt vérifier que $|\Delta Y_j| < \varepsilon$, où ε est près de 0. Nous avons pour cela employé le même ε que celui utilisé pour la condition d'arrêt avec distance minimale.

Notre algorithme de réseau élastique travaille la plupart du temps avec les distances au carré afin d'éviter de calculer des racines carrées. Il ne tient alors pas compte de l'arrondissement des distances à l'entier le plus près, mais le gain en efficacité compense largement cette perte de précision.

3.2 Algorithme génétique

Pour l'algorithme génétique, nous avons utilisé un tableau pour stocker la population. À chaque itération, un tri rapide dans $O(N \lg N)$ en moyenne est alors nécessaire pour traiter les individus.

Pour les nombres aléatoires, nous avons utilisé le générateur MRG32k3a de SSJ [2]. Ce générateur a une plus grande période que celui de Java et cette période peut être divisée en séquences indépendantes (*random streams*). Ainsi, nous pouvons associer une telle séquence à chaque île afin de maximiser la reproductibilité des expériences. Sans cette précaution, les résultats dépendraient de l'ordre d'exécution des tranches de temps allouées par le système d'exploitation.

Après plusieurs bogues avec la recherche locale, nous avons décidé de recopier la solution à traiter dans un tableau temporaire, effectuer la recherche locale puis recopier le tableau dans la liste chaînée. Nous avons constaté que le coût de cette copie, dans $O(n)$, est largement dominé par le coût de la recherche locale dont chaque itération est dans $O(n^2)$. Dans notre implantation, un échange 2-opt est effectué en permutant deux sommets du tour étant donné que la représentation utilisée est une liste de sommets et non une liste d'arêtes.

4 Résultats expérimentaux

Nous avons évalué les deux algorithmes implantés sur cinq problèmes de voyageur de commerce. Trois problèmes, tsp22, tsp51 et tsp76, sont considérés petits tandis que les deux autres, tsp100 et tsp442, sont moyens. Nous évaluerons la qualité des solutions produites par rapport aux solutions optimales du tableau 1 ainsi que le temps de calcul nécessaire. Les temps donnés sont des temps système et non pas des temps de processeur, car Java ne permet pas le calcul du temps de processeur. Java 5 fournit une méthode pour calculer ce temps, mais cette méthode bogue sous Linux avec des programmes comportant plusieurs fils d'exécution. Les expériences ont été exécutées

avec un Pentium D 2,8GHz d'Intel, en utilisant JRE 1.5.0_06 de Sun sous Linux Fedora Core 5 x86_64.

4.1 Réseau élastique

Nous avons testé le réseau élastique avec les paramètres donnés dans le tableau 2 et inspirés de [4]. Toutefois, au lieu de mettre $r = 200$, nous avons choisi $r = 90$, car l'algorithme ne convergeait plus au-delà de cette valeur. Les paramètres ε et Θ s'appliquent aux distances normalisées. Comme nous l'avons constaté, sans cette précaution, il faudrait adapter la valeur de ces paramètres à chaque problème. En particulier, un mauvais ajustement de ε provoque un arrêt trop rapide de l'algorithme, ce qui mène à une très mauvaise solution, ou un trop long temps d'exécution.

Nous avons appliqué le réseau élastique sur chaque problème pour produire les solutions du tableau 3. Nous avons ensuite appliqué la recherche locale avec *best improvement* et voisinage 2-opt sur chaque solution afin de l'améliorer ; les solutions produites sont aussi dans le tableau 3.

Nous avons employé cette technique au moment où le paramètre ε était mal ajusté et où les solutions produites par le réseau étaient très mauvaises. Une inspection visuelle de l'anneau produit pour le tsp51 nous a permis de rectifier le problème, mais nous avons malgré tout conservé la recherche locale dans les résultats.

Le tableau 4 présente quant à lui la liste des villes pour les meilleures solutions trouvées avec le réseau élastique combiné à la recherche locale.

Le tableau de résultats montre que la recherche locale améliore très peu la solution. Le réseau élastique produit de bonnes solutions en un temps très raisonnable. Toutefois, nous observons déjà une grande augmentation du temps d'exécution et une solution de moins bonne qualité pour le tsp442 par rapport au tsp100 et aux trois petits problèmes. Cela montre que l'algorithme risque de ne pas être efficace avec des problèmes de grande taille comportant des milliers de sommets.

Examinons maintenant plus en détails le comportement du réseau élastique. La figure 2 illustre la solution construite par le réseau pour le tsp22. L'anneau est représenté en lignes pointillées tandis que les lignes pleines représentent le tour. Les points numérotés représentent des villes tandis que les points non numérotés font partie de l'anneau. Les lignes rouges relient chaque ville avec le point le plus près sur l'anneau. Intuitivement, nous pouvons voir que les deux solutions de la figure pourraient être obtenus en étirant l'anneau de la bonne façon. L'algorithme pourrait donc être sensible aux variations dans les paramètres et surtout dans la structure du problème. En

TAB. 1 – Solution optimale des problèmes traités

Problème	Solution optimale
tsp22	278
tsp51	426
tsp76	538
tsp100	21 294
tsp442	50 778

TAB. 2 – Paramètres utilisés pour le réseau élastique

Taille relative de l'anneau	2,5	Rayon relatif	0,8
α	0,2	β	2
K_0	0,2	K_{MIN}	$0,02$
a	0,98	r	90
ε	$0,1$	Θ	10^{30}

TAB. 3 – Solutions trouvées par le réseau élastique

Problème	Réseau élastique			Réseau élastique + recherche locale		
	Distance	Temps	% optimal	Distance	Échanges	% optimal
tsp22	285	0,19s	2,5%	279	2	0,3%
tsp51	454	0,27s	6,6%	453	1	6,3%
tsp76	585	0,54s	8,7%	583	2	8,4%
tsp100	22 556	1,7s	5,9%	22 418	2	5,3%
tsp442	61 160	18s	20,4%	58 248	31	14,7%

particulier, il est à prévoir qu'il fonctionnera très mal pour un problème formé de quelques villes périphériques formant une enveloppe convexe et d'un grand nombre de villes disposées au centre de cette enveloppe. Par exemple, la densité des villes du tsp442, formant presqu'une grille, pourrait poser ce genre de difficultés.

Examinons maintenant la progression des itérations de l'algorithme. La figure 3 présente le graphique de la progression de la distance totale en fonction des itérations. Pour dresser ce graphique, à chaque itération, nous avons appliqué l'algorithme exposé à la section 2.1 pour construire une solution à partir de l'anneau et avons évalué la distance totale de cette solution. Sur le graphique, l'axe horizontal représente les itérations, quelle que soit la valeur de K. Un point foncé sur la courbe représente un changement de la valeur de K. De façon générale, la distance diminue avec le nombre d'itérations. Nous observons que la distance atteint toujours un pallier après un certain nombre d'itérations. Cela montre que nous pourrions ajuster ε plus finement pour faire arrêter l'algorithme plus tôt dans bien des cas. Par contre, une valeur de ε trop élevée provoquerait des arrêts prématurés menant à une très mauvaise solution. Nous pourrions aussi augmenter la valeur de K_{MIN}, car l'augmentation de la concentration de points foncés au fil des itérations montre que l'anneau change peu au moment où K est petit.

Nous pouvons aussi examiner la progression de la distance d_{min} au cours des itérations dont le graphique est présenté sur la figure 4. Cette distance oscille beaucoup en raison des mouvements de l'anneau. Par contre, l'amplitude des oscillations diminue au fur et à mesure que l'algorithme converge.

TAB. 4 – Meilleures solutions trouvées avec le réseau élastique combiné à la recherche locale

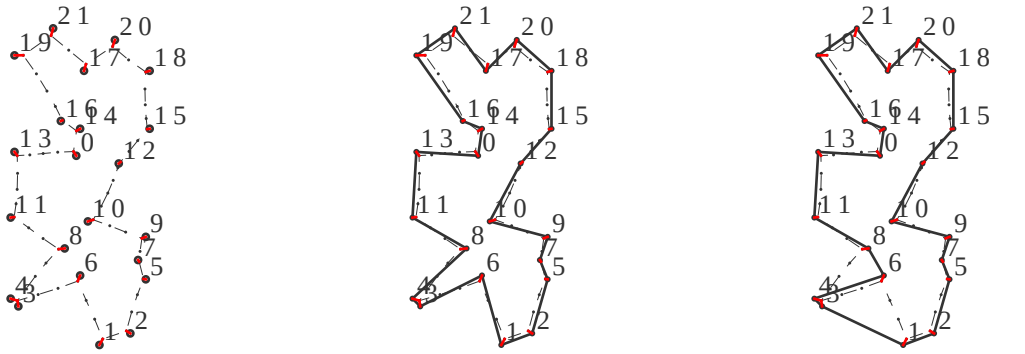
tsp22 10, 9, 7, 5, 2, 1, 3, 4, 6, 8, 11, 13, 0, 14, 16, 19, 21, 17, 20, 18, 15, 12

tsp51 15, 10, 1, 20, 28, 19, 34, 35, 2, 27, 30, 21, 0, 31, 26, 47, 7, 25, 6, 22, 42, 23, 24, 13, 5, 50, 45, 11, 46, 3, 17, 12, 40, 39, 18, 41, 16, 36, 43, 14, 44, 32, 38, 9, 29, 48, 4, 37, 8, 33, 49

tsp76 67, 74, 75, 25, 66, 33, 45, 51, 7, 53, 18, 34, 6, 52, 13, 58, 10, 65, 64, 37, 30, 9, 57, 71, 11, 39, 38, 8, 24, 54, 17, 49, 31, 43, 23, 48, 2, 16, 50, 5, 32, 15, 62, 22, 55, 40, 42, 41, 63, 21, 0, 72, 61, 1, 29, 73, 27, 60, 20, 47, 46, 35, 68, 70, 59, 69, 19, 36, 4, 14, 56, 12, 26, 28, 44, 3

tsp100 37, 65, 51, 45, 7, 22, 41, 21, 57, 28, 35, 66, 74, 6, 73, 60, 75, 79, 71, 56, 30, 77, 98, 44, 32, 67, 13, 64, 85, 95, 84, 58, 17, 81, 24, 19, 46, 11, 27, 20, 55, 94, 9, 25, 86, 54, 8, 5, 39, 82, 2, 4, 91, 47, 1, 99, 29, 50, 18, 36, 34, 61, 52, 59, 43, 92, 3, 70, 42, 76, 63, 15, 62, 53, 90, 38, 72, 14, 68, 83, 40, 23, 26, 87, 78, 12, 0, 49, 80, 33, 89, 16, 93, 31, 96, 97, 48, 69, 88, 10

tsp442 277, 295, 294, 293, 319, 320, 321, 356, 357, 434, 358, 359, 360, 343, 429, 322, 296, 297, 323, 428, 324, 298, 299, 300, 325, 361, 362, 363, 364, 430, 326, 301, 302, 327, 328, 303, 304, 329, 305, 330, 344, 365, 366, 367, 368, 369, 431, 370, 371, 372, 373, 374, 337, 336, 426, 335, 334, 333, 332, 331, 306, 436, 274, 422, 437, 271, 419, 267, 415, 264, 236, 413, 225, 410, 409, 263, 235, 262, 261, 421, 418, 259, 260, 412, 411, 258, 257, 256, 232, 407, 403, 408, 224, 215, 203, 191, 192, 204, 216, 205, 206, 194, 193, 180, 167, 156, 168, 181, 195, 196, 207, 217, 182, 197, 208, 218, 219, 209, 198, 183, 170, 159, 147, 146, 158, 169, 157, 145, 132, 122, 110, 111, 123, 133, 134, 124, 112, 382, 383, 97, 379, 378, 95, 96, 64, 32, 376, 375, 31, 30, 63, 94, 62, 29, 28, 61, 93, 435, 100, 92, 91, 60, 27, 59, 26, 25, 58, 90, 89, 57, 24, 23, 56, 22, 55, 88, 87, 377, 86, 54, 21, 20, 53, 85, 84, 52, 19, 51, 18, 17, 50, 82, 83, 381, 380, 438, 107, 118, 387, 119, 108, 384, 120, 109, 121, 131, 144, 143, 390, 130, 142, 389, 129, 141, 154, 393, 155, 166, 179, 178, 394, 165, 177, 397, 190, 189, 202, 214, 223, 222, 213, 401, 201, 200, 188, 396, 176, 164, 163, 187, 175, 162, 392, 152, 138, 139, 153, 140, 128, 127, 116, 104, 105, 117, 106, 99, 79, 80, 81, 49, 16, 15, 48, 14, 47, 78, 46, 13, 12, 45, 77, 76, 98, 75, 44, 11, 43, 10, 9, 42, 74, 73, 41, 8, 7, 40, 72, 71, 70, 39, 6, 5, 38, 4, 37, 69, 68, 36, 3, 35, 2, 1, 0, 441, 33, 34, 67, 66, 65, 101, 102, 113, 125, 385, 440, 103, 114, 126, 386, 388, 136, 150, 149, 135, 148, 160, 171, 184, 185, 172, 161, 173, 395, 398, 151, 115, 137, 391, 174, 186, 402, 210, 220, 211, 199, 212, 221, 229, 246, 245, 228, 244, 243, 242, 241, 240, 239, 234, 406, 227, 405, 400, 399, 404, 226, 233, 237, 265, 238, 266, 270, 273, 269, 268, 272, 275, 278, 280, 276, 425, 279, 439, 281, 427, 340, 341, 307, 308, 282, 283, 284, 310, 309, 338, 345, 346, 347, 432, 348, 349, 350, 351, 339, 311, 285, 286, 312, 313, 287, 288, 314, 315, 342, 352, 353, 354, 355, 433, 317, 318, 292, 291, 316, 290, 289, 423, 420, 424, 247, 248, 414, 249, 250, 230, 251, 252, 231, 255, 254, 253, 417, 416



(a) Seulement l'anneau (b) Sans recherche locale (c) Avec recherche locale

FIG. 2 – Solution construite par le réseau élastique pour le tsp22

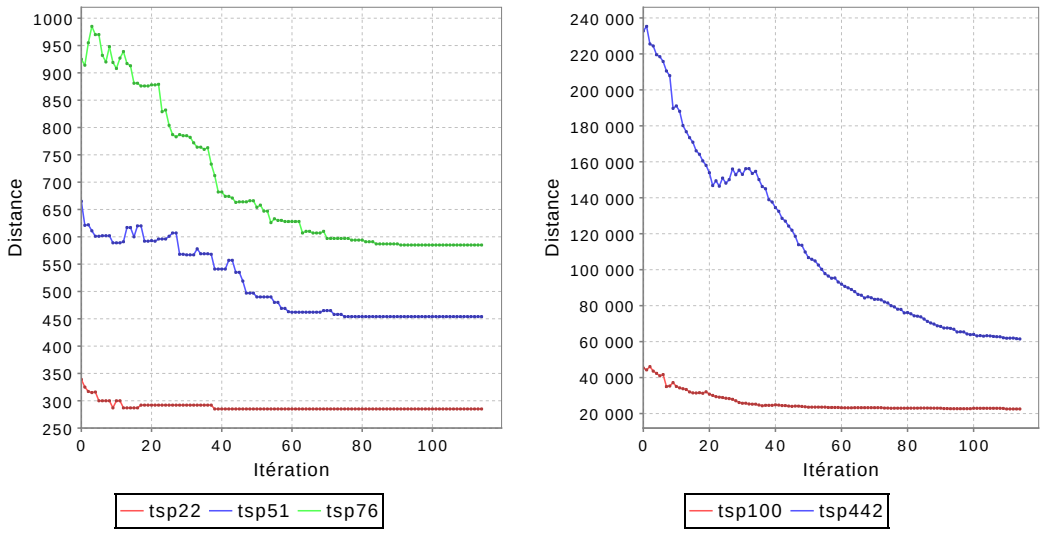


FIG. 3 – Progression de la solution au cours des itérations du réseau élastique

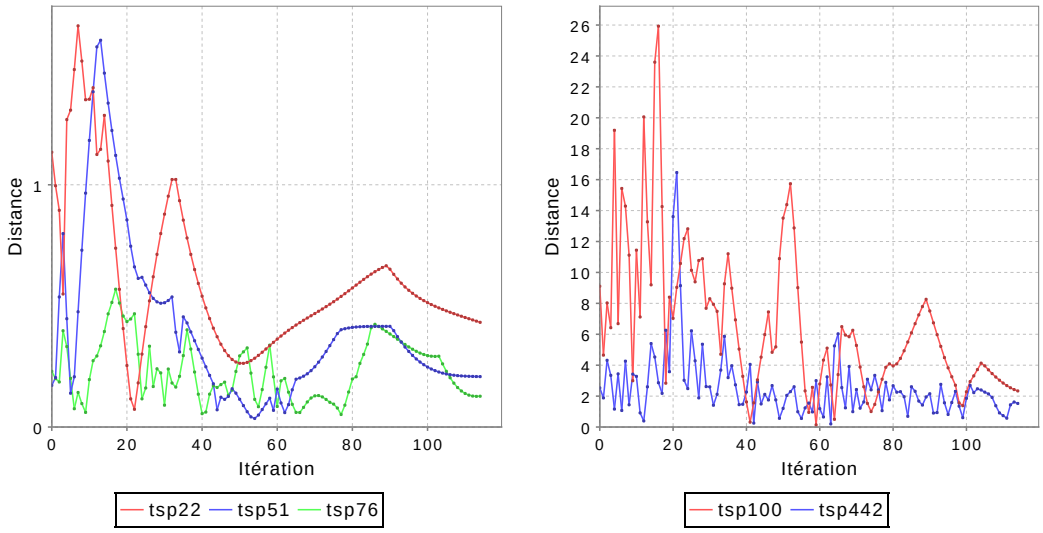


FIG. 4 – Progression de la distance minimale au cours des itérations du réseau élastique

4.2 Solution aléatoire

Avant d'évaluer les performances de l'algorithme génétique, examinons le comportement des algorithmes de construction aléatoire utilisés pour les solutions initiales. Pour évaluer ces heuristiques, nous avons choisi le germe par défaut du MRG32k3a qui est le germe 0 du tableau 7. Nous avons exécuté les deux heuristiques 10 000 fois sur chaque problème et relevé la meilleure solution, la pire solution et la distance totale moyenne. Dans les tableaux de résultats, le pourcentage d'écart à la solution optimale est donné, mais le temps de traitement est omis, car il est trop petit. Comme le montre le tableau 5, la permutation aléatoire donne de très mauvais résultats, mais l'heuristique d'insertion (tableau 6) est assez performante. Elle peut même produire de meilleures solutions que le réseau élastique de la section précédente. Par contre, en cas de malchance, la solution obtenue est assez mauvaise, quoique meilleure que celle obtenue par une permutation aléatoire.

TAB. 5 – Résultats obtenus avec la permutation aléatoire

Problème	Meilleur	Pire	Moyen
tsp22	501 (80,2%)	1 043 (275%)	801 (188%)
tsp51	1 268 (198%)	1 951 (358%)	1 652 (288%)
tsp76	2 103 (291%)	2 901 (439%)	2 522 (369%)
tsp100	136 361 (540%)	191 062 (797%)	163 099 (666%)
tsp442	716 192 (1 310%)	827 875 (1 530%)	772 674 (1 422%)

TAB. 6 – Résultats obtenus avec l'insertion stochastique

Problème	Meilleur	Pire	Moyen
tsp22	278 (0%)	333 (19,8%)	290 (4,6%)
tsp51	429 (0,7%)	508 (19,2%)	458 (7,6%)
tsp76	550 (2,2%)	638 (18,6%)	587 (9,2%)
tsp100	21 501 (0,97%)	25 430 (19,4%)	23 145 (8,7%)
tsp442	54 951 (8,2%)	61 014 (20,2%)	58 142 (14,5%)

4.3 Algorithme génétique

Nous avons appliqué l'algorithme génétique avec les cinq germes présentés au tableau 7 sur les cinq problèmes. Pour chaque problème, nous avons trouvé la meilleure solution, la pire solution et la moyenne des cinq solutions. Nous avons utilisé les paramètres du tableau 8 pour chacun de ces tests. Pour les problèmes tsp22, tsp51 et tsp76, nous avons utilisé une recherche locale de type *best improvement* avec un nombre illimité d'échanges 2-opt en guise de mutation. Par contre, pour les problèmes tsp100 et tsp442, nous avons utilisé la recherche locale de type *first improvement* et limité le nombre d'échanges 2-opt à 50 par recherche afin que l'algorithme s'exécute dans un temps raisonnable. Comme nous le verrons plus loin, un long temps d'exécution n'est pas susceptible de produire une meilleure solution.

Le tableau 9 présente les solutions obtenues avec l'algorithme génétique. Chaque entrée rapporte la distance totale de la solution (ou la moyenne des distances totales), le pourcentage d'écart de cette distance par rapport à la solution optimale et le temps (moyen) nécessaire pour obtenir la solution. L'algorithme génétique donne de très bonnes solutions, parfois même la solution optimale. La plupart du temps, la solution produite est meilleure que celle obtenue par l'heuristique d'insertion. Par contre, si elle est appliquée un grand nombre de fois, l'heuristique d'insertion arrive parfois à produire une meilleure solution. Dans tous les cas, l'algorithme a pour effet de rapprocher la pire solution de la meilleure. Ainsi, même si on est malchanceux, on obtient une solution assez bonne.

Nous pouvons nous interroger sur la pertinence de la recherche locale comme technique de mutation. Pour cela, prenons le tsp100 et les paramètres du tableau 8 avec une recherche locale de type *first improvement* limitée à 50 échanges. Avec ces paramètres et le germe 0, la solution obtenue a une distance totale de 21 715. Si nous appliquons la recherche locale sur tous les individus plutôt que la moitié, la distance totale reste alors la même. Par contre, avec aucune recherche locale, la

TAB. 7 – Germes utilisés pour le MRG32k3a

ID	Germe					
0	12 345	12 345	12 345	12 345	12 345	12 345
1	558 612 258	-374 730 548	-3 634 556 747	494 424 984	2 101 494 111	1 562 156 336
2	2 414 100 246	-2 192 353 346	32 016 405	-1 574 759 209	-421 346 314	1 722 074 874
3	-1 383 507 795	2 676 590 627	3 406 863 953	-4 166 754 944	-4 290 596 819	991 734 359
4	3 230 304 736	-2 034 818 463	1 304 720 569	-1 822 479 171	1 442 761 634	2 559 380 660

TAB. 8 – Paramètres de l'algorithme génétique

Nombre d'îles	2	Taille de la population sur chaque île	300
Probabilité de permutation aléatoire	0,4	Probabilité de copie	0,1
Probabilité de mutation	0,5	Probabilité de migration	0,2
MIN	0,5	MAX	1,5
Nombre max. d'itérations	150	Nombre d'itérations sans amélioration	100

TAB. 9 – Solutions obtenues avec l'algorithme génétique

Problème	Meilleur	Pire	Moyen
tsp22	278 (0%, 1s)	278 (0%, 1s)	278 (0%, 1s)
tsp51	427 (0,2%, 19s)	432 (1,4%, 19s)	429 (0,7%, 19s)
tsp76	552 (2,6%, 1min8s)	559 (3,9%, 1min10s)	555 (3,2%, 1min9s)
tsp100	21 540 (1,2%, 11s)	21 715 (2,0%, 12s)	21 628 (1,6%, 11s)
tsp442	55 030 (8,4%, 1min29s)	55 863 (10,0%, 1min30s)	55 592 (9,5%, 1min28s)

solution se dégrade : la distance totale est alors 21 866. Il est ainsi important que la recherche locale soit appliquée sur au moins une partie des individus.

Maintenant, est-il nécessaire d'utiliser deux heuristiques différentes pour construire les solutions initiales ? Prenons les paramètres originaux, avec probabilité de mutation fixée à $0,5$, et utilisons une probabilité de permutation aléatoire de 0. La distance totale devient alors 21 576. Si des permutations aléatoires sont générées avec probabilité 1, la distance devient 55 406. Cela montre que la diversité créée par les permutations aléatoires nuit à l'algorithme pour ce problème particulier. Cela montre aussi que les croisements peuvent améliorer la solution, car la solution produite est meilleure que celle obtenue par 10 000 permutations aléatoires.

Ces deux expériences montrent que l'heuristique d'insertion et la recherche locale sont beaucoup plus puissantes que l'algorithme génétique sur les problèmes étudiés. Cela explique pourquoi de nombreux paramètres n'ont aucun effet lorsque ces heuristiques sont en œuvre. Nous pouvons donc nous interroger sur les performances de l'algorithme génétique simple, avec des permutations aléatoires comme solutions initiales et sans la recherche locale. Pour le tsp100 avec le germe 0, si des permutations aléatoires sont employées comme solutions initiales et qu'il n'y a aucune mutation, la distance totale de la meilleure solution devient 113 649. Utiliser la recherche locale de type *first improvement* limitée à 50 échanges 2-opt réduit la distance totale à 51 970.

Essayons d'augmenter la pression de sélection de la façon suivante. Avec $\text{MIN} = 0,5$ et $\text{MAX} = 1,5$, la distance totale pour le tsp100 avec le germe 0 est 113 649. Si $\text{MIN} = 0,2$ et $\text{MAX} = 0,8$, la distance totale devient 107 744. Ainsi, augmenter la pression de sélection donne une meilleure solution puisque l'algorithme a tendance à sélectionner les individus avec une distance totale petite.

Nous pouvons aussi évaluer l'effet des migrations sur la qualité de la solution. Si la probabilité de migration est 0, la distance totale pour le tsp100 est 119 791. En réglant la probabilité de migration à 1, nous obtenons une solution avec une distance totale de 90 080. Par contre, plus la probabilité de migration est élevée, plus les résultats peuvent dépendre du planificateur de tâches

TAB. 10 – Meilleures solutions obtenues avec l'algorithme génétique

tsp22 16, 19, 21, 17, 20, 18, 15, 12, 10, 8, 6, 9, 7, 5, 2, 1, 3, 4, 11, 13, 0, 14

tsp51 28, 20, 33, 49, 15, 1, 21, 0, 31, 10, 37, 4, 48, 8, 29, 9, 38, 32, 44, 14, 36, 16, 43, 41, 39, 18, 40, 12, 24, 13, 17, 3, 46, 11, 45, 50, 26, 5, 47, 22, 23, 42, 6, 25, 7, 30, 27, 2, 35, 34, 19

tsp76 2, 43, 31, 49, 17, 54, 24, 8, 38, 71, 57, 30, 9, 37, 64, 65, 10, 58, 13, 52, 6, 34, 18, 7, 45, 33, 3, 74, 75, 66, 25, 11, 39, 16, 50, 5, 67, 1, 73, 29, 47, 28, 44, 26, 51, 53, 12, 56, 14, 4, 36, 19, 69, 59, 70, 68, 35, 46, 20, 60, 27, 61, 72, 32, 0, 21, 63, 41, 42, 40, 55, 22, 62, 15, 48, 23

tsp100 7, 22, 10, 89, 16, 93, 31, 96, 97, 48, 69, 88, 41, 21, 57, 28, 35, 66, 74, 6, 73, 60, 75, 79, 71, 56, 30, 77, 98, 44, 32, 67, 13, 64, 85, 95, 15, 62, 53, 90, 76, 42, 63, 84, 58, 2, 82, 39, 5, 17, 81, 24, 19, 46, 11, 27, 20, 55, 94, 9, 8, 54, 25, 86, 91, 47, 1, 18, 36, 50, 99, 29, 34, 61, 4, 52, 59, 43, 92, 3, 70, 38, 72, 14, 68, 83, 40, 23, 26, 87, 78, 12, 0, 49, 33, 80, 37, 65, 51, 45

tsp442 65, 101, 102, 113, 440, 103, 114, 385, 126, 386, 388, 150, 149, 136, 125, 135, 148, 160, 171, 184, 399, 404, 226, 233, 237, 265, 268, 272, 275, 278, 269, 238, 234, 406, 227, 405, 400, 185, 172, 161, 173, 395, 398, 186, 174, 151, 391, 115, 137, 392, 152, 162, 175, 187, 199, 212, 221, 211, 210, 402, 220, 228, 245, 244, 243, 242, 241, 240, 239, 266, 270, 273, 276, 425, 439, 279, 282, 283, 284, 311, 310, 338, 309, 308, 307, 281, 280, 427, 340, 341, 345, 346, 347, 432, 348, 349, 339, 350, 351, 342, 352, 353, 316, 315, 289, 288, 314, 313, 287, 312, 285, 286, 423, 420, 424, 290, 291, 317, 318, 292, 293, 294, 320, 319, 433, 354, 355, 356, 357, 434, 358, 359, 360, 429, 322, 321, 295, 277, 296, 297, 298, 323, 428, 343, 324, 325, 299, 300, 301, 302, 327, 326, 430, 361, 362, 363, 364, 365, 366, 367, 344, 328, 303, 304, 329, 305, 330, 331, 332, 431, 368, 369, 370, 371, 372, 373, 374, 337, 336, 426, 335, 334, 333, 306, 436, 274, 422, 437, 271, 419, 267, 415, 263, 262, 235, 261, 260, 421, 418, 259, 258, 257, 256, 255, 254, 253, 252, 417, 416, 251, 250, 249, 414, 248, 247, 246, 229, 222, 230, 231, 223, 214, 203, 191, 192, 215, 224, 232, 407, 408, 411, 412, 403, 216, 204, 205, 206, 217, 207, 196, 195, 194, 193, 180, 167, 156, 181, 168, 157, 145, 132, 122, 110, 111, 123, 133, 146, 158, 169, 182, 197, 208, 218, 410, 409, 413, 264, 236, 225, 219, 209, 198, 183, 170, 159, 147, 134, 124, 112, 382, 383, 97, 96, 379, 95, 63, 64, 32, 376, 375, 31, 30, 29, 62, 61, 28, 27, 26, 25, 24, 57, 58, 59, 60, 93, 94, 378, 435, 100, 92, 91, 90, 89, 88, 56, 23, 22, 55, 87, 377, 86, 85, 53, 54, 21, 20, 19, 18, 51, 52, 84, 83, 82, 81, 438, 107, 118, 380, 381, 108, 390, 143, 144, 131, 121, 109, 120, 384, 119, 130, 387, 129, 141, 389, 142, 155, 166, 179, 178, 394, 393, 154, 165, 177, 397, 190, 202, 189, 201, 401, 213, 200, 188, 176, 396, 163, 164, 153, 140, 128, 139, 138, 127, 116, 104, 105, 117, 106, 98, 75, 44, 76, 77, 78, 79, 99, 80, 48, 49, 50, 17, 16, 15, 47, 14, 13, 46, 45, 12, 11, 10, 43, 9, 8, 41, 42, 74, 73, 72, 40, 7, 6, 39, 71, 70, 38, 5, 4, 3, 37, 69, 68, 36, 35, 67, 66, 34, 2, 1, 0, 441, 33

du système d'exploitation et varier d'une exécution à l'autre.

Nous pouvons également tenter d'accroître le nombre d'îles à 6 et réduire la population sur chaque île à 100 individus. Sans migration, nous obtenons une distance totale de 120 238 pour le tsp100. Avec la migration à chaque génération, la distance totale baisse à 88 865. Ainsi, la migration peut améliorer la qualité de la solution, mais il n'est pas nécessaire de trop augmenter le nombre d'îles. Nous avons choisi d'utiliser deux îles plutôt qu'une seule afin de tirer profit des deux cœurs du Pentium D utilisé pour les tests.

Si, à partir de la configuration originale du tableau 8, nous utilisons des permutations aléatoires comme solution initiales mais les échanges aléatoires comme mutation, nous obtenons 119 499 avec une probabilité de mutation de $0,1$ et 106 748 avec une probabilité de mutation de 0. Pour ce problème et ces paramètres, il semble que les échanges aléatoires n'apportent aucun effet positif. Par contre, il pourrait arriver qu'il en soit autrement pour d'autres configurations.

Nous avons appliqué l'algorithme de nouveau sur tous les problèmes, cette fois avec les paramètres du tableau 11. Nous avons dû augmenter la taille de la population et le nombre maximal de générations afin d'essayer de compenser la mauvaise qualité des solutions initiales. Comme mutation, nous avons utilisé un maximum de 50 échanges aléatoires de sommets.

Comme nous pouvons le voir dans le tableau 12, les solutions obtenues sont beaucoup moins bonnes qu'avec la recherche locale et l'insertion stochastique et cela, pour un temps de calcul beaucoup plus long. Même avec de petits problèmes, la solution est mauvaise et, dans le cas du tsp442, la solution est totalement inacceptable. L'algorithme est très sensible aux paramètres et la migration rend les résultats variables d'une exécution à l'autre.

Examinons maintenant la progression de la solution en fonction du nombre de générations. La figure 5 présente la progression pour l'algorithme génétique avec des permutations et des échanges aléatoires. Nous observons que la solution s'améliore rapidement pendant les premières générations pour atteindre un pallier. Rendu au pallier, il devient très coûteux d'améliorer davantage la solution. Avec les paramètres originaux, le graphique se présentait comme une droite, car la meilleure solution était trouvée dans les toutes premières générations.

La figure 6 présente le graphique de la progression de la meilleure solution dans la population à chaque génération et sur chacune des deux îles pour le tsp22 et le tsp442, avec l'algorithme génétique avec insertion stochastique et recherche locale. Sur ces graphiques, la solution oscille pour atteindre un pallier après un certain nombre de générations. Ce pallier représente l'homogénéisation de la population après laquelle il n'est plus possible d'améliorer la solution. Cela montre que les croisements, bien que capables d'améliorer une mauvaise solution, détériorent les bonnes

TAB. 11 – Paramètres de l'algorithme génétique sans recherche locale et avec seulement des permutations aléatoires

Nombre d'îles	2	Taille de la population sur chaque île	1 500
Probabilité de permutation aléatoire	1	Probabilité de copie	0,1
Probabilité de mutation	0,1	Probabilité de migration	0,2
MIN	0,2	MAX	1,8
Nombre max. d'itérations	1 500	Nombre d'itérations sans amélioration	1 000

TAB. 12 – Résultats avec l'algorithme génétique avec des permutations aléatoires comme solutions initiales et des échanges aléatoires comme mutation

Problème	Meilleur	Pire	Moyen
tsp22	279 (0,4%, 37s)	296 (6,5%, 40s)	286 (3,0%, 38s)
tsp51	722 (69,5%, 2min21s)	754 (77,0%, 2min29s)	733 (72,2%, 2min30s)
tsp76	1 349 (150,7%, 4min1s)	1 426 (165,1%, 3min39s)	1 388 (158.0%, 3min58s)
tsp100	95 152 (347%, 5min15s)	97 888 (360%, 6min2s)	96 207 (352%, 5min48s)
tsp442	579 787 (1 042%, 1h27min)	589 317 (1 061%, 1h29min)	584 057 (1 050%, 1h27min)

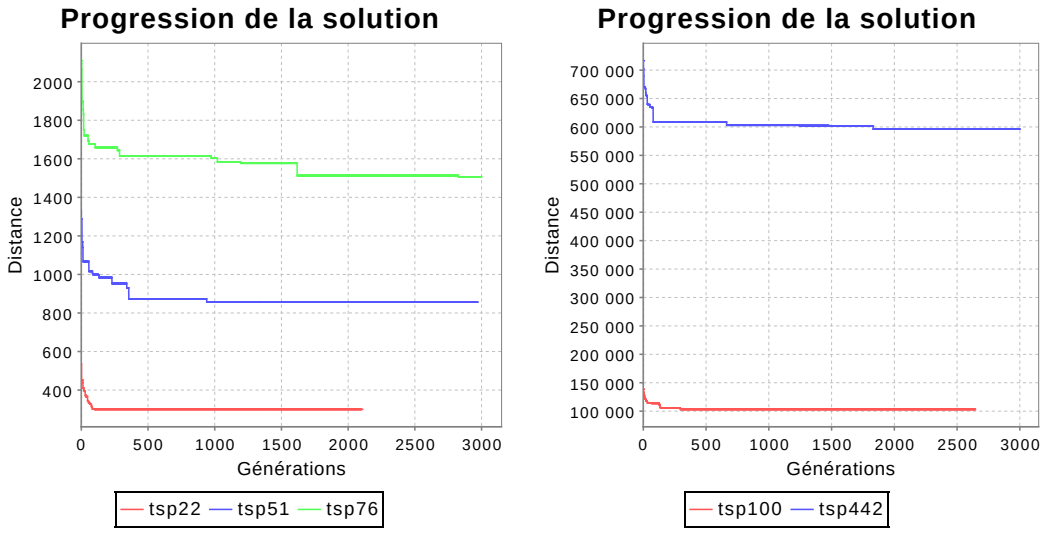


FIG. 5 – Progression de la meilleure solution en fonction des générations

solutions produites par insertion stochastique. Des graphiques très similaires peuvent être obtenus avec l'algorithme utilisant des permutations et des échanges aléatoires.

4.4 Comparaison des algorithmes

Le réseau élastique s'exécute dans $O(n^2)$ tandis que l'algorithme génétique s'exécute dans $O(Nn^2)$. Par contre, N est constant si bien que le coût asymptotique est identique à celui du réseau élastique. Le temps d'exécution dépend donc uniquement des paramètres de l'algorithme qui conditionnent le nombre d'itérations et le coût de chaque itération.

Le réseau élastique n'est pas un choix très avantageux, car plus le problème est grand, plus il est lent à calculer en raison des poids à déterminer lors de chaque itération. Il faut également bien ajuster les paramètres pour obtenir une bonne solution. Toutefois, même un réseau relativement bien ajusté ne bat pas l'insertion stochastique.

L'insertion stochastique est amplement suffisante pour de petits problèmes, mais elle n'est pas fiable : avec un peu de malchance, il est possible d'obtenir une solution qui s'éloigne beaucoup de l'optimum. L'algorithme génétique permet d'améliorer cette fiabilité au prix d'un temps d'exécution plus élevé.

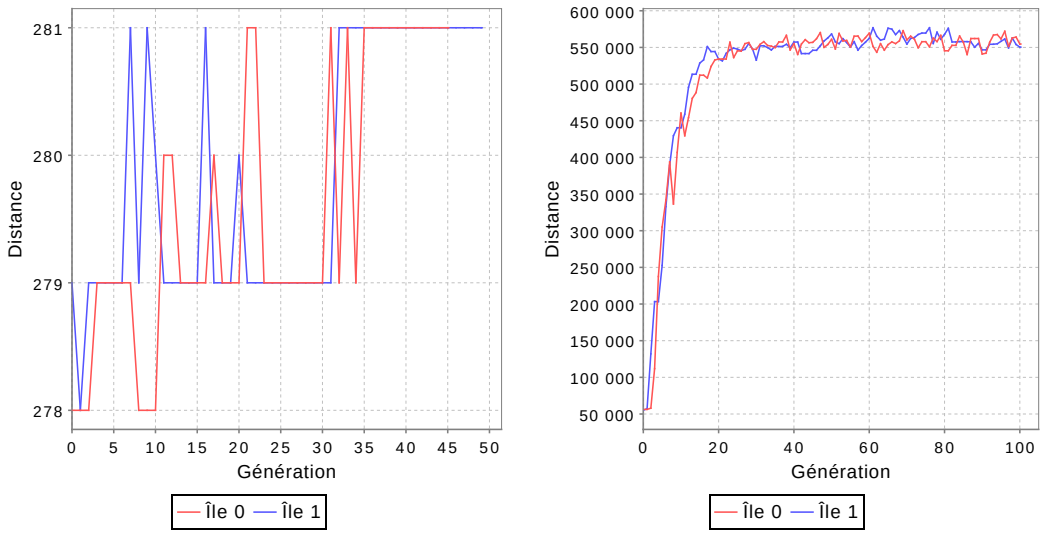
Par contre, l'utilisation de la recherche locale sur les individus de la population peut devenir très coûteuse en temps de calcul pour de gros problèmes. Il peut alors arriver que l'algorithme prenne un temps démesuré. Il faut donc bien ajuster les paramètres, par exemple en limitant le nombre d'échanges 2-opt par mutation, et aucune méthode n'existe pour guider cet ajustement.

Si les solutions initiales sont mauvaises et la mutation est aléatoire (ou s'il n'y a pas de mutation), l'algorithme devient très inefficace et il est difficile de trouver de bons paramètres. Il se peut alors que la seule façon de faire soit d'examiner le déroulement de l'algorithme par l'intermédiaire d'une interface graphique et d'ajuster les paramètres pendant son déroulement.

5 Conclusion

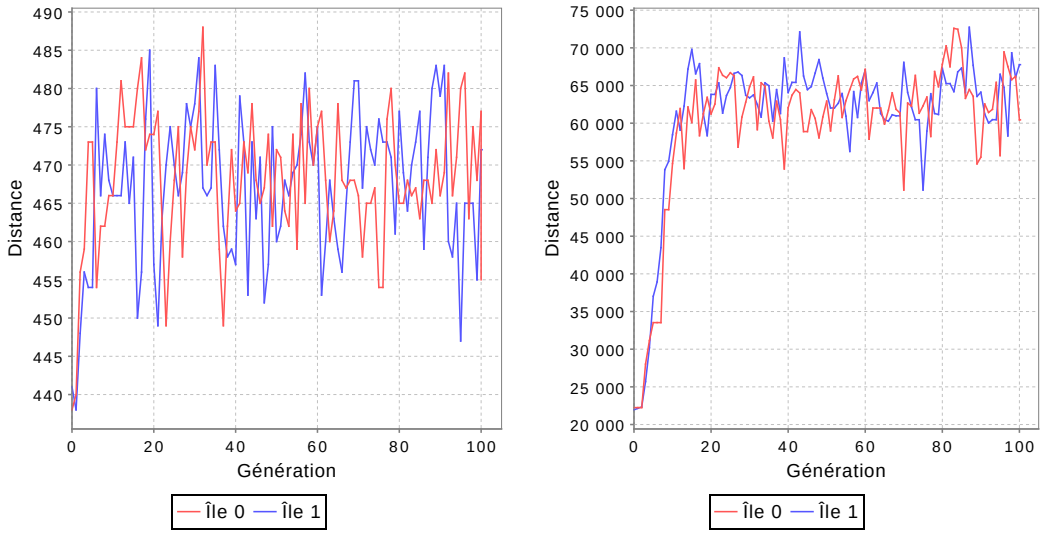
Nous avons étudié deux métaheuristiques pour résoudre le problème de voyageur de commerce et constaté que l'algorithme génétique battait le réseau élastique, autant au niveau temps d'exécution pour des problèmes de taille moyenne qu'au niveau de la qualité des solutions produites. Nous avons par contre remarqué que l'algorithme génétique lui-même contribue très peu à la construction de la solution. La convergence de l'algorithme pourrait par conséquent être améliorée puisque la meilleure solution est trouvée dans les premières générations. La construction d'un meilleur algorithme de croisement que la recombinaison des arêtes serait sans doute le meilleur moyen d'améliorer l'efficacité de l'algorithme génétique sur ce problème.

On pourrait également imaginer des politiques de modification dynamique des paramètres de l'algorithme. Par exemple, plusieurs algorithmes génétiques s'exécutant en parallèle pourraient être considérés comme des « méta-individus » sur lesquels une sélection et des croisements seraient appliqués. Les algorithmes génétiques des générations subséquentes seraient alors autorisés à poursuivre le travail sur le problème. Malheureusement, l'évaluation d'un algorithme génétique,



(a) tsp22 (b) tsp442

FIG. 6 – Progression de la meilleure solution à chaque génération pour le tsp22 et le tsp442



(a) tsp51 (b) tsp100

FIG. 7 – Progression de la meilleure solution à chaque génération pour le tsp51 et le tsp100

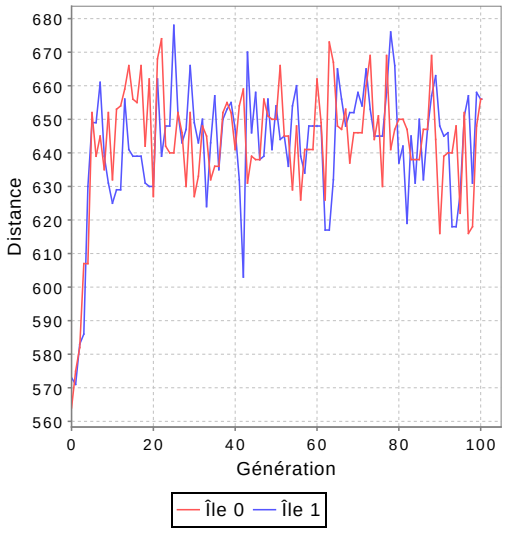


FIG. 8 – Progression de la meilleure solution à chaque génération pour le tsp76

les croisements et les mutations risquent d'être difficiles à concevoir et l'algorithme résultant pourrait être très lent.

Pour résoudre spécifiquement le problème de voyageur de commerce, il serait aussi possible de remplacer l'heuristique d'insertion par une heuristique plus puissante, comme GENI [1]. La recherche locale pourrait également être remplacée par des échanges *k*-opt ou Lin-Kernighan [3]. On pourra alors penser à remplacer l'algorithme génétique par une recherche à points de départ multiples ou un GRASP [8].

Références

[1] M. Gendreau, A. Hertz et G. Laporte. New insertion and postoptimization procedures for the traveling salesman problem. *Operations Research*, 40:1086–1094, 1992.

[2] Pierre L'Ecuyer. *SSJ : A Java Library for Stochastic Simulation*, 2004. URL `http://www.iro.umontreal.ca/~lecuyer`. Guide de l'utilisateur.

[3] S. Lin et B. W. Kernighan. An effective heuristic algorithm for the traveling salesman problem. *Operations Research*, 21(2):498–516, 1973.

[4] Jean-Yves Potvin. The traveling salesman problem : A neural network perspective. *ORSA Journal on Computing*, 5:328–348, 1993.

[5] Jean-Yves Potvin. Genetic algorithms for the traveling salesman problem. *Annals of Operations Research*, 63:339–370, 1996.

[6] Jean-Yves Potvin. Algorithmes génétiques — introduction. URL `http://www.iro.umontreal.ca/~potvin/ift6580`. Notes de cours IFT6580, 2006.

[7] Jean-Yves Potvin. Présentation sur les problèmes dynamiques de tournées de véhicules. URL `http://www.iro.umontreal.ca/~potvin/ift6580`. Notes de cours IFT6580, 2006.

[8] Mauricio G. C. Resende et Celso C. Ribeiro. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133, 1995.

[9] Clifford A. Shaffer. *A Practical Introduction to Data Structures and Algorithm Analysis*. Prentice Hall, deuxième édition, 2001.

A Structure du programme

Le programme permettant l'exécution du réseau élastique et de l'algorithme génétique a été implanté en Java et testé sous Fedora Core 5 x86_64 avec JRE 1.5.0_06 de Sun. Les classes sont divisées en plusieurs paquetages.

`data` Définit un problème de voyageur de commerce ainsi que des mécanismes pour lire les problèmes depuis des fichiers. Ce paquetage définit une classe représentant un client (ou une ville) ainsi qu'un problème qui regroupe un ensemble de clients. Il définit également une liste spécialisée ne pouvant accueillir que des clients et vérifiant qu'il n'y a aucun doublon.

`solbuilding` Contient des heuristiques de construction de solution stochastiques. Ce paquetage contient des classes implantant la génération de permutations aléatoires ainsi que l'insertion stochastique.

`solving` Contient l'implantation du réseau élastique et de l'algorithme génétique.

`tester` Contient différentes classes destinées à tester les algorithmes et produire des tableaux de résultats.

`gui` Contient une interface graphique de base permettant à l'utilisateur d'exécuter les algorithmes.

Pour plus d'informations à propos de la structure du programme, voir la documentation Javadoc en HTML.

B Utilisation du programme

L'exécution du programme nécessite Java 5. Les fichiers compilés du programme sont dans une archive JAR qui peut être exécutée directement avec la commande

```
java -jar lib/gentsp.jar
```

L'archive `ssj.jar` doit se trouve dans le même répertoire que `gentsp.jar`, mais il n'est pas nécessaire de mettre quelqu'archive que ce soit dans le `CLASSPATH` pour une simple exécution. La ligne de commande ci-haut fait apparaître une fenêtre de contrôle avec plusieurs onglets.

Avant de pouvoir utiliser les algorithmes, il faut ouvrir un problème en utilisant la commande **Open problem** du menu **File**. Les problèmes se présentent sous forme de fichiers textuels, avec une ville par ligne. Une ligne optionnelle peut être utilisée pour spécifier la distance minimale pour la solution optimale. Cette fonctionnalité n'est pas utilisée dans l'interface graphique, mais elle a été mise en œuvre dans le paquetage de tests. La figure 9 donne un exemple de problème pour le tsp22.

Lorsqu'un problème est ouvert, l'interface affiche la liste des clients dans l'onglet **Problem** de la fenêtre principale, comme sur la figure 10.

```
totalDistOpt 278
1  145 215
2  151 264
3  159 261
4  130 254
5  128 252
6  163 247
7  146 246
8  161 242
9  142 239
10 163 236
11 148 232
12 128 231
13 156 217
14 129 214
15 146 208
16 164 208
17 141 206
18 147 193
19 164 193
20 129 189
21 155 185
22 139 182
```

FIG. 9 – Contenu du fichier `tsp22.txt`

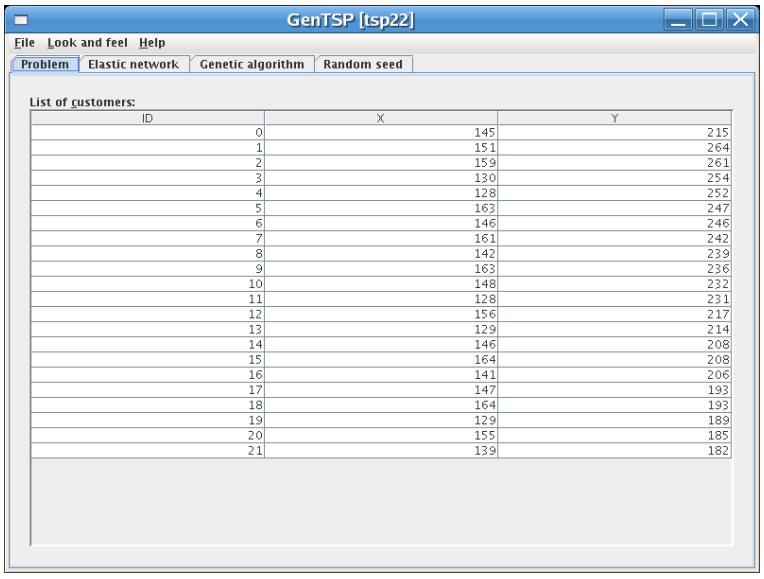


FIG. 10 – Fenêtre principale du programme

B.1 Utilisation du réseau élastique

Les paramètres du réseau élastique sont disponibles dans l'onglet **Elastic network** de la fenêtre principale. L'utilisateur spécifie tout d'abord la taille relative de l'anneau et son rayon relatif avant de cliquer sur le bouton **Init ring**. Cela active les autres contrôles et affiche l'anneau dans la petite fenêtre inférieure droite, comme sur la figure 11. L'utilisateur peut ajuster les divers paramètres du réseau élastique en modifiant les champs de texte. Les valeurs par défaut sont celles utilisées pour les tests dans ce rapport.

Pour démarrer l'algorithme, il suffit de cliquer sur le bouton **Start**. L'utilisateur peut alors suivre la progression de l'anneau en temps réel et arrêter l'algorithme en tout temps, avec le bouton **Stop**. L'algorithme s'arrête automatiquement lorsqu'une condition d'arrêt est vérifiée. La figure 12 présente le réseau élastique étiré formant une solution au tsp22.

B.2 Utilisation de l'algorithme génétique

L'onglet **Genetic algorithm** de la fenêtre principale permet d'utiliser l'algorithme génétique pour résoudre un TSP. L'utilisateur ajuste simplement les paramètres avant d'appuyer sur le bouton **Start**. Les paramètres par défaut sont ceux utilisés pour le tsp22, le tsp51 et le tsp76 dans nos tests.

Après avoir appuyé sur le bouton de démarrage, l'utilisateur peut regarder l'exécution de l'algorithme. Comme le montre la figure 13, l'utilisateur peut parcourir la population sur chacune des îles et consulter un journal des étapes d'exécution de l'algorithme. La meilleure solution est aussi disponible au bas de l'écran en tout temps.

L'utilisateur peut stopper l'algorithme à tout moment, mais un temps d'attente est parfois nécessaire puisque l'algorithme doit terminer la génération en cours. Ceci peut poser des problèmes avec le tsp442 si les paramètres de recherche locale ne sont pas ajustés.

B.3 Fonctionnalités diverses

L'onglet **Random seed** permet d'ajuster le germe du MRG32k3a utilisé pour la génération des nombres aléatoires de l'algorithme génétique. Par défaut, le germe de SSJ est employé.

Il est également possible de modifier l'apparence de l'interface par l'intermédiaire du menu **Look and Feel**. Cela permet par exemple de rendre l'interface semblable à celle de Windows ou d'utiliser (par défaut) l'interface multi-plateforme de Java.

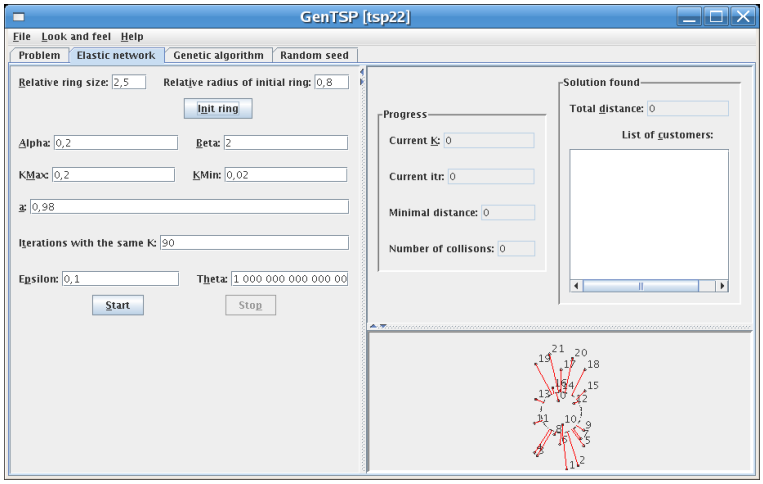


FIG. 11 – Réseau élastique initialisé

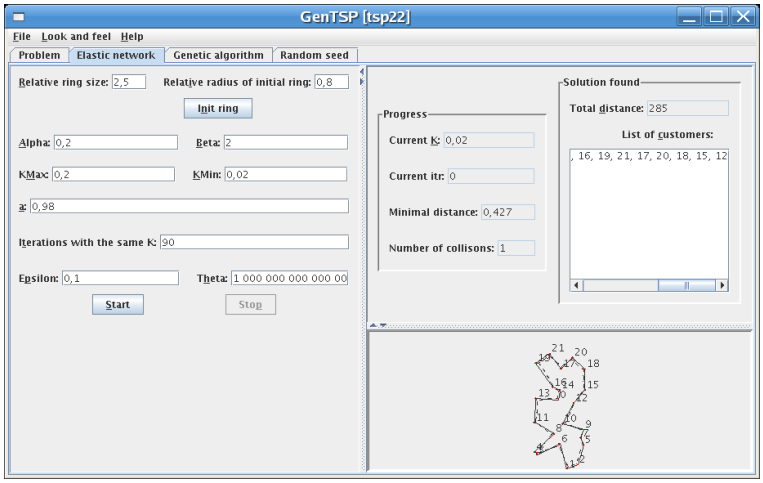


FIG. 12 – Réseau élastique étiré pour tsp22

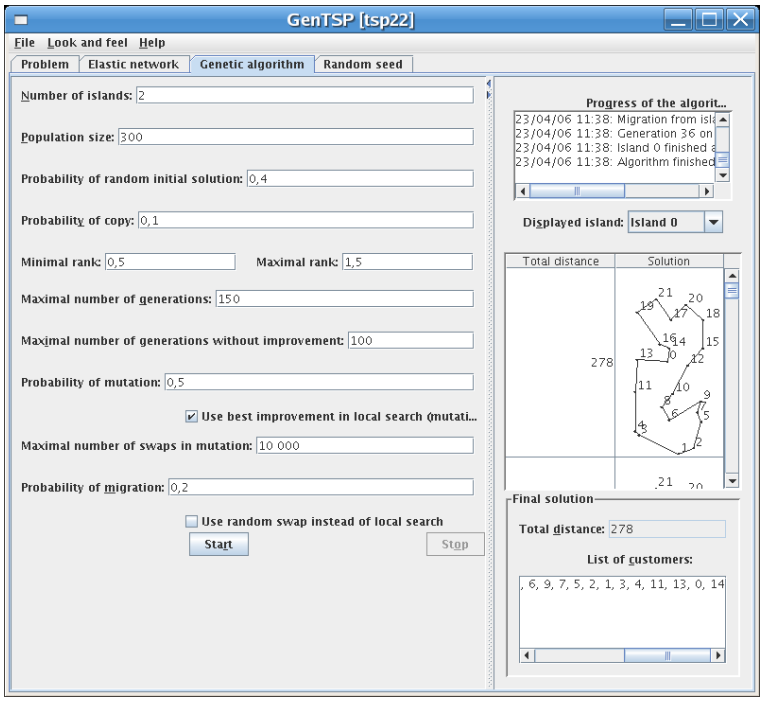


FIG. 13 – Algorithme génétique appliqué au tsp22