HAL
archives-ouvertes.fr

Algorithmes génétiques : un croisement adapté aux fonctions partiellement séparables

Nicolas Durand, Jean-Marc Alliot, Joseph Noailles

▶ **To cite this version:**

Nicolas Durand, Jean-Marc Alliot, Joseph Noailles. Algorithmes génétiques : un croisement adapté aux fonctions partiellement séparables. AE 94, European Conference on Artificial Evolution, Sep 1994, Toulouse, France. pp xxxx. hal-00937680

HAL Id: hal-00937680

https://hal-enac.archives-ouvertes.fr/hal-00937680

Submitted on 25 Apr 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Algorithmes Génétiques : un croisement adapté aux fonctions partiellement séparables

Nicolas Durand Jean-Marc Alliot Joseph Noailles

CENA* ENAC† ENSEEIHT‡

Soumis aux Journées "Evolutions Artificielles 94"

19-23 septembre 94

Abstract

Dans cet article, nous proposons une nouvelle méthode de croisement pour résoudre des problèmes d'optimisation globale comportant un grand nombre de variables et dont la fonction d'évaluation peut se décomposer en une somme de fonctions ne faisant pas intervenir toutes les variables. Cette méthode de croisement nécessite l'introduction d'une "fitness locale" associée à chaque variable et d'un paramètre d'incertitude Δ qui permet de moduler le déterminisme de l'opérateur. Cet opérateur de croisement, utilisé avec une méthode de sharing et de recuit simulé rend les algorithmes génétiques très efficaces pour minimiser des problèmes comportant beaucoup de variables ou fortement combinatoires, tels que la minimisation d'un polynôme de grande taille ou la résolution du problème du voyageur de commerce.

1 Introduction

Dans bon nombre de problèmes d'optimisation, la fonction à optimiser dépend de nombreuses variables, et peut se décomposer comme somme de sous-fonctions prenant en compte une partie des variables seulement. Les algorithmes génétiques s'avèrent être de bons outils d'optimisation globale en raison de leur capacité à sortir des minima locaux. Néanmoins, leurs performances sont souvent pénalisées par le caractère très aléatoire des opérateurs de croisement et de mutation. Pour remédier à ce phénomène, bon nombre d'évolutionnistes utilisent des heuristiques qui permettent de favoriser les "bons" croisements ou les "bonnes" mutations et d'éviter les "mauvaises". Le but de cet article est de présenter un opérateur de croisement efficace, utilisable dans tous les problèmes où la fonction à optimiser peut se décomposer en somme de sous-fonctions ne dépendant pas de toutes les variables du problème.

Après une présentation formelle du type de problèmes auxquels s'adresse cette méthode, nous présenterons l'opérateur de croisement. Une illustration de l'efficacité de cette méthode sera ensuite proposée au travers d'un exemple simple comportant une vingtaine de variables. Enfin la dernière partie propose une application de cet outil au classique problème du Voyageur de Commerce, largement étudié dans la littérature, aussi bien pour son intérêt pratique qu'en raison de sa grande complexité [Bra90, GL85, GGRG85, HGL93, OSH89, WSF89]. En effet, pour tester la performance d'un algorithme d'optimisation globale, le problème du Voyageur de commerce, NP-Complet, permet d'offrir de nombreux exemples de très grande taille dont on connaît les solutions comportant beaucoup d'optima locaux.

2 Problèmes concernés

Il s'agit de tous les problèmes où la fonction F à optimiser dépend de plusieurs variables x_1, x_2, .., x_n, et peut se décomposer en somme de fonctions F_i ne dépendant pas de toutes les variables. On remarquera que bon nombre de fonctions multiplicatives peuvent se transformer en fonction additives grâce à des transformations simples. On s'intéresse donc aux fonctions pouvant s'écrire.

$$F(x_1, x_2, .., x_n) = \sum_{k=1}^{m} F_i(x_{j_1}, x_{j_2}, .., x_{j_{n_i}})$$

Soit :

$$E_k = \{i / x_k \in variables\, de\, F_i\}$$

Pour définir notre opérateur de croisement, nous allons introduire pour chaque variable x_k sa "fitness

*Centre d'Etudes de la Navigation Aérienne

†Ecole Nationale de l'Aviation Civile

‡Ecole Nationale Supérieure d'Electronique, d'Electrotechnique, d'Informatique et d'Hydraulique de Toulouse

locale" $G_k(x_k, x_1, x_2, .., x_n)$ définie comme suit :

$$G_k(x_k, x_1, x_2, .., x_n) = \sum_{i \in E_k} F_i(x_{j_1}, x_{j_2}, .., x_{j_{n_i}})$$

Le but de cet article est de montrer comment utiliser cette "fitness locale" pour définir un nouvel opérateur de croisement permettant de réduire la taille des populations et convergeant très rapidement vers la solution optimale. Pour cela, l'algorithme génétique a également été amélioré en le combinant avec des techniques de recuit simulé et de sharing.

3 L'opérateur de croisement

Le principe est très simple. L'opérateur de croisement consiste, à partir de deux chromosomes parents, à créer deux nouveaux chromosomes. Les premiers opérateurs de croisement opérant sur des chaînes de bits prenaient brutalement le début du père 1 et la fin du père 2 pour le fils 1 et l'inverse pour le fils 2. On a montré par la suite que cette technique avait le gros défaut de reproduire plus souvent les schémas courts. Certaines techniques de croisement à plusieurs points ont été proposées avec plus ou moins de succès. Les évolutionnistes travaillant sur des chromosomes composées de variables réelles ont proposé des croisements de type barycentrique, avec des techniques aléatoires de définition des poids affectés à chaque variable.

Le codage que nous adopterons pour appliquer notre croisement consiste à représenter le chromosome par la liste brute des variables du problème. S'il s'agit de bits, nous aurons une liste de bits, s'il s'agit de variables réelles, nous aurons une liste de réels. La technique que nous proposons ici consiste à retenir pour chaque variable, pour la conception des fils, celle des deux parents qui a la meilleur fitness locale, ceci à un facteur Δ près. Par exemple, dans le cas où l'on cherche un minimum pour F, pour la k^ime variable, si :

$$G_k(pere_1) < G_k(pere_2) - \Delta$$

alors le fils 1 se verra affecté la variable x_k du père 1. Si par contre :

$$G_k(pere_1) > G_k(pere_2) + \Delta$$

il se verra affecté la variable x_k du père 2. Si enfin :

$$\|G_k(pere_1) - G_k(pere_2)\| \leq 2 \quad \Delta$$

la variable x_k du fils 1 sera choisie aléatoirement. Si l'on applique la même stratégie pour le fils 2, les deux fils risquent de se ressembler, surtout si Δ est faible, ce qui n'est pas souhaitable. On peut donc pour le deuxième fils choisir une autre valeur de Δ ce qui permet d'être plus ou moins déterministe, ou alors, comme pour les techniques de croisement classique choisir le complémentaire du fils 1, à savoir utiliser la variable du père non choisi par le fils 1.

On peut remarquer qu'il est facile d'introduire un opérateur de mutation qui s'appuie sur le même principe et modifie avec une plus forte probabilité les variables ayant une mauvaise fitness locale.

4 Le recuit simulé et le sharing

Pour assurer la diversité de la population courante, on utilise des méthodes de recuit simulé et de sharing.

4.1 Le recuit simulé

Lors de l'opération de croisement, les deux enfants générés à partir de leurs parents ne sont pas forcément meilleurs que leurs parents. On peut alors se poser la question de savoir s'il vaut mieux garder les parents ou les enfants (on a choisi ici une méthode de croisement avec remplacement). De nombreux résultats [DASF94a, DASF94b] montrent qu'une technique de sélection utilisant le recuit simulé permet d'améliorer considérablement la vitesse de convergence de l'algorithme génétique. Le principe est le suivant : si le fils est meilleur que ses deux pères, il est conservé. Sinon, on évalue ΔE_1 et ΔE_2 la différence de fitness entre le fils et ses deux parents. La probabilité que l'on garde le fils plutôt que le père i est alors $\exp(-\frac{\Delta Ei}{T})$ où T est la température de recuit qui décroît en fonction du temps selon un schéma de recuit bien particulier dont on trouvera plus de détail dans [MG92, AK89]. Le recuit simulé permet de se sortir des minima locaux souvent très nombreux et de parcourir, au moins lors des premières générations une plus grande partie de l'espace des variables.

4.2 Le sharing

Utiliser une méthode de sharing s'avère indispensable dès que l'on rend plus déterministe l'algorithme. Si l'on veut s'assurer une diversité suffisante de la population et éviter les accumulations autour du meilleur élément, il faut pénaliser les chromosomes trop représentés. L'inconvénient des méthodes de sharing classiques réside dans le fait qu'elles sont très coûteuses en temps (elles croissent en n^2). Yin et Germay [YG] proposent une méthode de sharing en $n\log(n)$ faisant intervenir une notion de cluster et nécessitant l'introduction de la notion de barycentre d'éléments de population. Cette méthode est facilement utilisable dès lors que l'on travaille avec des variables réelles. L'adaptation est plus difficile lorsque l'on travaille sur des variables entières ou des chaînes de bits. Pour le problème du voyageur de commerce, par exemple, définir le barycentre de la ville i et de la ville j n'a de sens que si $i = j$. Aussi, cette méthode de sharing ne pourra être utilisée que si l'on se limite à définir des clusters ne contenant que des éléments identiques. Cette méthode revient à diviser la fitness de chaque individu par le nombre de fois qu'il apparaît dans la

population. Cette méthode simpliste est déjà très efficace et suffira à assurer une diversité de la population dans bon nombre de cas.

5 Application à un polynôme

Pour illustrer l'intérêt de notre opérateur de croisement, considérons le polynôme suivant :

$$F(x_1, x_2, .., x_n) \quad = \quad \sum_{i \neq j} (x_i - x_j)^2$$

On cherche le minimum de ce polynôme sur l'espace des entiers compris entre 0 et 20. Il y a trivialement 21 optima globaux à ce problème. La taille de l'espace de recherche est 21^{20} soit plus de 10^{27} . Le codage de notre problème se résume à une chaîne d'entiers. On définit les fitness locales conne suit :

$$G_k(x_1, x_2, .., x_n) \quad = \quad \sum_{i \neq k} (x_i - x_k)^2$$

Pour résoudre ce problème, nous avons effectué deux simulations. La première utilise une méthode de croisement classique, à savoir un croisement aléatoire en un point de la chaîne et ne fait pas appel à l'opérateur de sharing décrit précédemment. La deuxième utilise la méthode de croisement décrite ci-dessus, et la combine avec la méthode de sharing du paragraphe précédent.

Le recuit simulé décrit précédemment est utilisé dans les deux cas. Pour des raisons de normalisation, au lieu de minimiser F, nous avons maximisé $\frac{1}{1+F}$. Les paramètres des simulations sont les suivants :

Taille de la population : 600
Nombre de générations : 100
Probabilité de croisement : 0.6
Probabilité de mutation : 0.15

Les simulations ont été faites sur une HP 720, le temps de calcul moyen est de l'ordre de 300 cpu.

Les figures 1 et 2 nous donnent l'évolution du meilleur élément de population dans les deux simulations. On observe que la convergence vers un optimum est beaucoup plus rapide lorsque l'on utilise le croisement avec fitness locale décrit précédemment. L'optimum est alors atteint dès la $22^{\text{ème}}$ génération au lieu de la $63^{\text{ème}}$ lorsque l'on utilise le croisement classique. Cet amélioration n'est pas imputable au sharing qui a tendance a privilégier la diversité au détriment de la rapidité de convergence.

Les figures 3 et 4 nous donnent l'évolution de la fitness moyenne dans les deux cas. On observe que la moyenne n'atteint pas 0.4 à la dernière génération dans la première simulation, alors qu'elle dépasse cette valeur dès la $45^{\text{ème}}$ génération dans la deuxième simulation, ceci malgré l'utilisation du sharing qui pénalise les meilleurs éléments lors de la sélection et par conséquent la fitness moyenne.

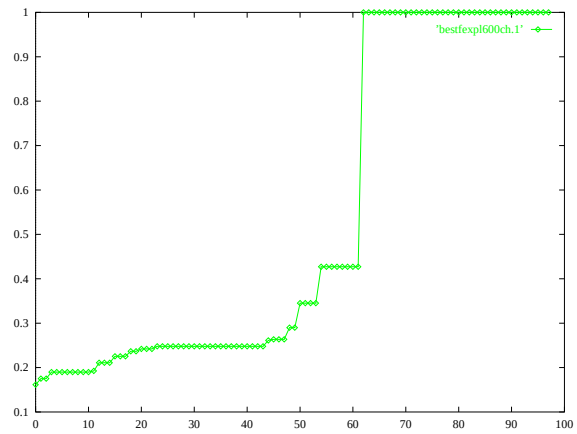

Figure 1: Valeur du meilleur élément, croisement classique sans sharing.

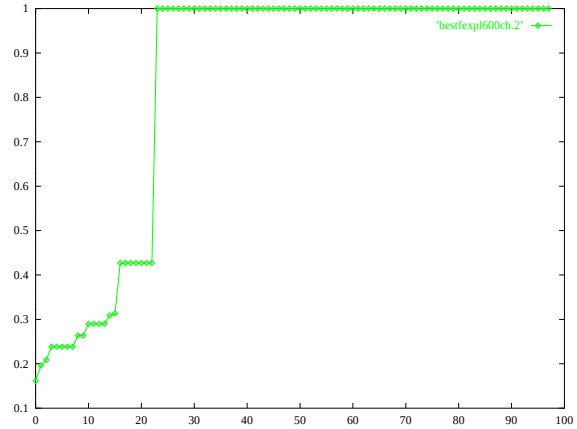

Figure 2: Valeur du meilleur élément, nouveau croisement avec sharing.

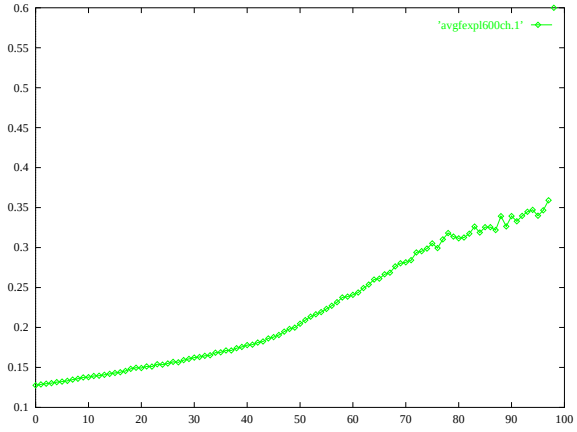

Figure 3: Moyenne des fitness, croisement classique sans sharing.

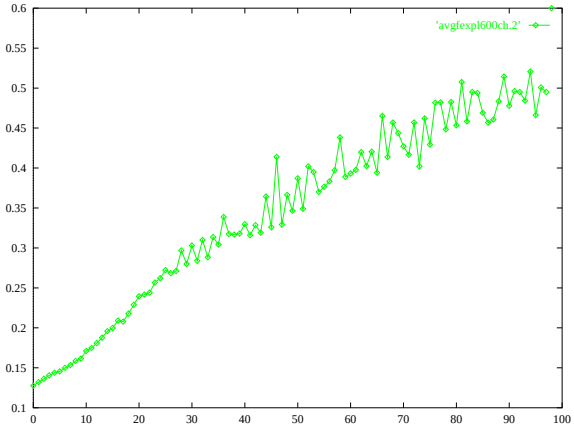

Figure 4: Moyenne des fitness, nouveau croisement avec sharing.

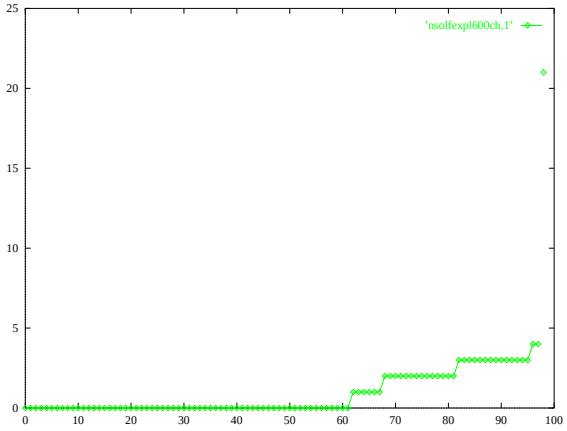

Figure 5: Nombre de solutions optimales, croisement classique sans sharing.

Le résultat le plus spectaculaire apparaît dans les figures 5 et 6 donnant le nombre de solutions optimales obtenues dans les deux simulations. Là il est tout à fait clair que le sharing a joué un rôle important dans la deuxième simulation puisque les 21 optima globaux sont obtenus, alors que sans sharing, ni notre croisement, seuls 4 optima sont obtenus. Une troisième simulation permet d'observer que la présence du sharing seul ne permet d'obtenir que 3 optima globaux (la figure associée à ce résultat n'est pas donnée pour ne pas charger l'exposé), ce qui confirme le grand intérêt de notre opérateur de croisement.

6 Application au problème du Voyageur de Commerce

Afin de montrer la généralité de ce nouvel opérateur de croisement, nous proposons dans cette section d'appliquer l'opérateur de croisement défini précédemment pour résoudre un problème bien connu de grande taille. Notre opérateur de croisement permet de limiter l'influence du codage sur la convergence de l'algorithme, l'ordre des villes n'ayant plus d'importance. Le preprocessing introduit par Bui et Moon [BM94] n'a alors plus aucune utilité. Les résultats sont très encourageant pour des problèmes de l'ordre de 200 villes. L'objectif n'est pas de battre un nouveau record de rapidité dans la résolution du problème du voyageur de commerce, mais bien de tester notre opérateur de croisement sur ce problème. 0n pourra noter que l'opérateur de mutation introduit est très simple et pourrait sans doute être fortement amélioré. Un opérateur d'optimisation locale composé de deux modules simples a été ajouter, il permet d'apporter des améliorations locales. Après une description du codage des données utilisé, nous décrirons les fonctions fitness locales G_i adoptées ainsi que les opérateurs de mutation et d'amélioration locale rajoutés. La quatrième partie exposera les résultats obtenus.

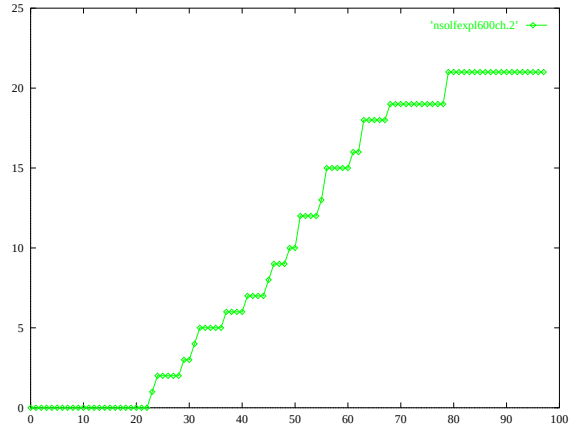

Figure 6: Nombre de solutions optimales, nouveau croisement avec sharing.

6.1 Codage des données

Le problème du voyageur de commerce peut se coder de diverses façons, tenant compte des positions respectives des villes les unes par rapport aux autres ou non. Avec les méthodes de croisement classiques à un ou plusieurs points l'ordre dans lequel les villes sont codées a une influence sur la convergence. Pour notre algorithme, cet ordre n'a aucune importance. Les données seront donc codées sous forme d'une liste d'entiers représentant l'indice de la ville suivante. Ainsi le code *bcdea* représente le chemin *abcdea*. Pour faciliter l'utilisation des divers opérateurs, nous aurons besoin de connaître le prédécesseur de la ville dans laquelle on se trouve. Pour éviter une recherche côuteuse en temps du prédécesseur, nous introduisons dans le codage une redondance : à chaque ville on associe l'indice de son successeur et celui de son prédécesseur. Ainsi le code (be, ca, db, ec, ad) représente le chemin *abcdea*.

Un tableau de distances est initialisé en début d'al-

gorithme afin de limiter les calculs. Un deuxième tableau est créé dans lequel pour chaque ville, on classe dans l'ordre les villes les plus proches. Ceci permettra lors des opérations de croisement et de mutation de sélectionner rapidement des villes pas trop éloignées.

6.2 L'opérateur de croisement

Au lieu de définir une seule fitness locale G_i, pour des raisons de symétrie, chaque variable ou ville sera dotée de deux fitness locales f_s et f_p. Soit n_{prof} un nombre compris entre 1 et la longueur totale du chromosome. Pour chaque ville à l'intérieur d'un chromosome, on détermine la longueur du chemin lorsque l'on parcours n_{prof} villes dans le sens direct (c'est à dire en prenant la ville suivante à chaque fois). La fitness locale f_s représente donc cette valeur. On peut faire de même en parcourant les villes dans le sens opposée (c'est à dire en considérant chaque fois la ville précédente). On affectera à f_p cette valeur. Ainsi, notre codage s'enrichit. Il comporte désormais quatre listes au lieu de deux. Les deux premières listes permettent de définir l'ordre des villes dans le cycle, les deux dernières sont composées des fitness locales de chaque ville.

Le principe de notre croisement est le suivant : on choisit une ville au hasard qui constitue le point de départ de notre croisement. Le processus suivant est répété n fois où n est le nombre total de villes : pour choisir la ville suivante (voir figure 7), on va comparer les valeurs f_p et f_s des deux parents, représentant quatre chemins possibles. Parmi ces quatre chemins, certains parcourent des villes déjà utilisées. On pénalise alors les f_p, f_s des chemins correspondant. Le choix de la ville suivante se fera en considérant le chemin le plus court, à une certaine valeur Δ près. Si le chemin le plus court est distant de moins de Δ d'un autre, la ville suivante est choisie au hasard entre ces deux villes. On retrouve bien là le principe de croisement détaillé précédemment. La valeur Δ a pour but de ne pas rendre l'algorithme trop déterministe. Dans le cas où les quatre villes suivantes possibles ont déjà été utilisées, on choisira de façon heuristique la ville la plus proche disponible. Pour construire le deuxième fils, on effectue la même opération en prenant un point de départ différent.

Pour nos simulations, nous avons choisi Δ constante et valant la distance moyenne entre les villes. On peut imaginer que Δ décroisse au fur et à mesure des générations. Par contre, on à intérêt à choisir n_{prof} petit en début de convergence et de le faire croître de façon à prospecter de plus en plus loin dans les chemins possibles. Nous prendrons par exemple n_{prof} proportionnel au nombre de générations effectuées et variant entre 1 et $\frac{n}{5}$.

6.3 L'opérateur de mutation

L'opérateur de mutation utilisé est très simple. Cependant, son rôle est utile pour parcourir le plus largement possible l'espace de recherche. Observons le tableau donnant pour chaque ville les villes les plus

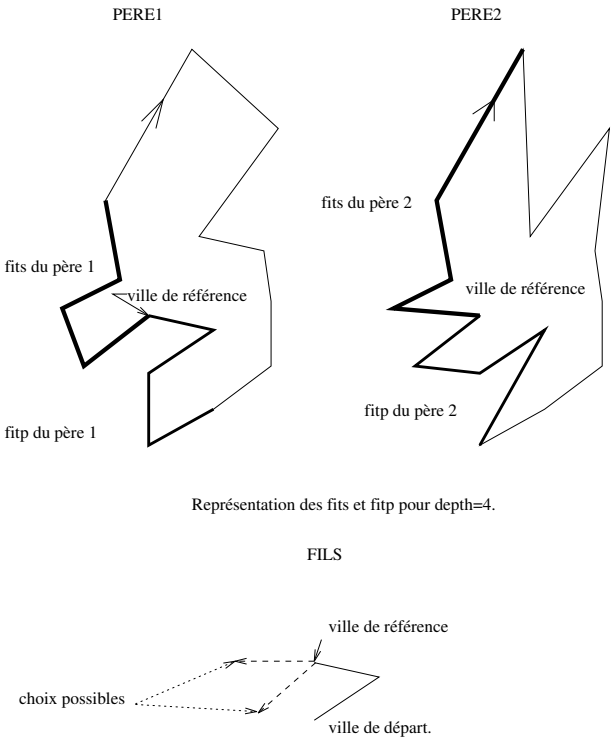

Figure 7: Stratégie de croisement.

proches dans l'ordre des distances croissantes. Il est clair que si toutes les villes sont reliées à la ville la plus proche, on a un chemin optimal. En général, un certain nombre de villes ne sont pas reliées aux villes les plus proches au profit d'autres. Nous appellerons ville de rang k une ville dont la ville suivante est la $k^{\text{ième}}$ dans l'ordre croissant des villes les plus proches. Il est évident que passer d'une solution courante à la solution optimale du problème améliore au moins le rang d'une des n villes considérées. L'opérateur de mutation découle de cette observation. Son principe est de relier quelques villes à leurs villes voisines les plus proches et de compléter ensuite le chemin en parcourant l'ancien chemin dans l'ordre d'apparition des villes. La figure 8 donne un exemple de mutation sur un problème à 19 villes. L'une d'entre elles est choisie au hasard. La mutation consiste à rechercher cinq fois la ville la plus proche non parcourue, dans l'exemple, on parcourt les villes 0, 3, 2, 1, 9, 11, et l'on complète ensuite le graphe avec les villes dans leur ancien ordre d'apparition.

6.4 Deux opérateurs locaux

Les opérateurs de croisement et de mutation précédemment décrits entraînent la formation de chemins imparfaits que l'on peut corriger trivialement. Nous introduisons ici deux opérateurs de correction que nous appellerons loc_1 et loc_2 et que nous appliquerons à tous les chromosomes avant chaque évaluation.

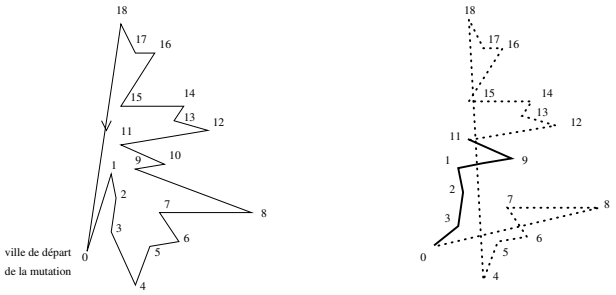

Figure 8: L'opérateur de mutation.

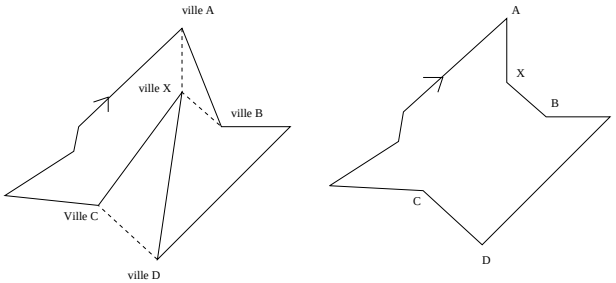

Figure 9: L'opérateur de correction Loc_1.

6.4.1 L'opérateur Loc_1

Cet opérateur consiste, pour chaque ville X (voir figure 9), à comparer la somme des coûts des arcs $XC + XD - CD$ avec les coûts $XA + XB - AB$ où A et B sont des villes successives proches de X. Si la deuxième somme est inférieure à la première, on reliera X aux villes A et B.

6.4.2 L'opérateur Loc_2

Cet opérateur a pour but de "déboucler les boucles". Pour chaque arc AB (voir figure 10, on cherche des villes consécutives proches C et D telles que la somme $AB + CD$ soit supérieure à $AC + BD$. On déboucle alors la boucle.

Il est à noter que ces deux opérateurs ne nécessitent pas que la distance entre les villes respecte l'inégalité triangulaire. Il n'est par ailleurs pas question de faire une recherche exhaustive de tous les arcs pouvant être concernés mais seulement de regarder les villes proches de chaque ville ou arc concerné. Le coût de ces deux opérateurs est donc simplement proportionnel au nombre de villes considérées dans le problème.

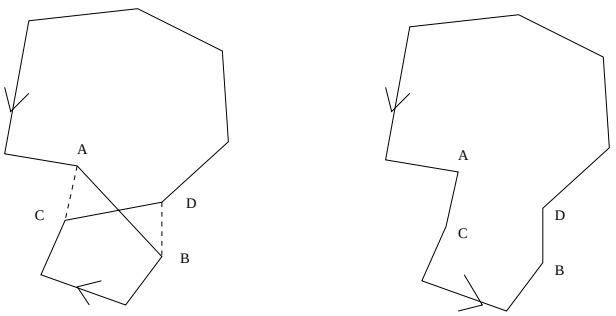

Figure 10: L'opérateur de correction Loc_2.

Pb considéré	lin105	kroa100	kroa200
Taille du problème	105	100	200
Taille de la pop	50	50	100
Valeur du minimum	14379	21282	29368
Nb min de gens	2	2	40
Nb moy de gens	14	6	110
Nb max de gens	30	43	268
Temps min (en sec)	1	2	130
Temps moy (en sec)	6	3	441
Temps max (en sec)	12	17	1418

Table 1: Résultats numériques sans sharing du problème du voyageur de commerce.

6.5 Résultats numériques

Pour toutes les résultats évoqués ci-dessous, les populations initiales sont construites de façon aléatoire. L'algorithme a été testé sur un certain nombre de problèmes classiques dont on connaît les solutions optimales. Pour chacun de ces problèmes, les paramètres des simulations sont les suivants :
Probabilité de croisement : 0.6
Probabilité de mutation : 0.15

Pour chacun de ces problèmes, nous avons fait tourner l'algorithme une cinquantaine de fois. Les simulations ont été réalisées sur HP720 sans méthode de sharing pour limiter le temps de calcul. L'algorithme a toujours trouvé la solution optimale. Les résultats sont présentés dans le tableau 1. La première ligne donne la référence du problème [Rei91]. La deuxième ligne donne sa taille. On donne ensuite la taille de la population utilisée pour résoudre le problème, la valeur numérique de l'optimum, le nombre minimum, moyen et maximum de générations pour résoudre le problème. Le temps minimum, moyen et maximum pour atteindre l'optimum.

Les résultats sont très bons pour des problèmes de l'ordre de 100 villes et tout à fait encourageant pour 200 villes. Ils montrent l'intérêt de cette approche locale utilisée pour le croisement.

7 Conclusion

L'opérateur de croisement proposé dans cet article peut s'adapter à un grand nombre de problèmes d'optimisation partiellement séparables. Il nécessite l'introduction de la notion de fitness locale associée à une variable. On peut influencer son comportement par l'intermédiaire du paramètre Δ qui permet de le

rendre plus ou moins déterministe. Associé à une méthode de sharing et de recuit simulé, il permet d'obtenir rapidement plusieurs optima locaux ou globaux tout en continuant à parcourir l'espace de recherche. Suffisamment général, cet opérateur simple devrait améliorer la résolution de nombreux problèmes combinatoires de grande taille.

Bibliographie

[AK89] Emile Aarts et Jan Korst. *Simulated annealing and Boltzmann machines.* Wiley and sons, 1989. ISBN: 0-471-92146-7.

[BM94] Thang N. Bui et Byung R. Moon. A new genetic approach for the traveling salesman problem. Dans *IEEE Conference on Evolutionnary Computation.* IEEE, June 1994.

[Bra90] H. Braun. On traveling salesman problems by genetic algorithms. Dans *1st Workshop on Parallel Problem Solving from Nature*, October 1990.

[DASF94a] Daniel Delahaye, Jean-Marc Alliot, Marc Schoenauer, et Jean-Loup Farges. Genetic algorithms for partitioning airspace. Dans *Proceedings of the Tenth Conference on Artificial Intelligence Application.* CAIA, 1994.

[DASF94b] Daniel Delahaye, Jean-Marc Alliot, Marc Schoenauer, et Jean-Loup Farges. Genetic algorithms for air traffic assignment. Dans *Proceedings of the EuropeanConference on Artificial Intelligence.* ECAI, 1994.

[GGRG85] J. Grefenstette, R. Gopal, B. Rosmaita, et D. Gucht. Genetic algorithms for the traveling salesman problem. Dans *1st International Conference on Genetic Algorithms and their Applications*, 1985.

[GL85] D. Goldberg et R. Lingle. Alleles, loci, and the travelling salesman problem. Dans *1st International Conference on Genetic Algorithms and their Applications*, 1985.

[HGL93] A. Homaifar, S. Guan, et G. Liepins. A new genetic approach on the traveling salesman problem. Dans *Fifth International Conference on Genetic Algorithms*, July 1993.

[MG92] Samir W. Mahfoud et David E. Goldberg. Parallel recombinative simulated annealing: a genetic algorithm. IlliGAL Report 92002, University of Illinois at Urbana-Champaign, 104 South Mathews Avenue Urbana IL 61801, April 1992.

[OSH89] I. Oliver, D. Smith, et J. Holland. Permutation crossover operators on the travelling salesman problem. Dans *Second International Conference on Genetic Algorithms*, July 1989.

[Rei91] G. Reinelt. Tsplib - a traveling salesman problem library. *Journal on Computing 3*, pages 376–384, 1991.

[WSF89] D. Whitley, T. Starkweather, et D. Fuquay. Scheduling problems and traveling salesman: The genetic edge recombination operator. Dans *Third International Conference on Genetic Algorithms*, 1989.

[YG] X. Yin et N. Germay. A fast genetic algorithm with sharing scheme using cluster analysis methods in multimodal function optimization. Rapport technique, Laboratoire d'Electronique et d'Instrumentation, Catholic University of Louvain.